# GPIO和LED

裸机实验值usb启动配合dnw工具下载

背景知识介绍：

回顾S5PV210启动方式，必须将OM5打到VCC，才能从USB启动。

S5PV210 的启动过程：开机时先执行内部的iROM中的BL0，然后BL0做了一系列初始化后在读取外部的OMpin的设置来确定用户选择了从哪里启动。当检测到我们设置的是USB启动时，S5PV210 就会从USB OTG 接口试图连接主机进行下载。

dnw工具介绍

dnw是一个软件，是三星公司编写的，这个软件的功能是通过USB线连接开发板和电脑主机，然后从主机下载文件镜像到开发板中去烧录系统。

dnw使用注意1：dnw是需要装驱动的。

dnw使用注意2：dnw使用时通过usb线下载，所以一定要插USB线。

dnw使用注意3：dnw下载时需要设置dnw下载内存地址，在dnw软件的菜单“Configuration”中设置 Download Address为0xD00200010（BL1的从oxD00200000开始，并且前四个字节为校验信息，usb下载时候不使用。），确认即可

win7 X64版本驱动安装非常麻烦，因为微软启用了USB设备驱动签名政策

dnw驱动安装

x210开发版使用了软开关，但是我们这里还没到操作系统没去处理开关，所以在整个裸机实验中必须手工按下POWER键才能保持开机，只要一抬起来就关机了。。。

dnw驱动装好的标志是：开发板开机从usb启动后，设备管理器中显示已经安装的设备，并且关键是dnw工具中USB：OK

usb启动实验总结：

usb启动方式主要使用来调试程序的，其实分析S5PV210即可知道，我们这里是把裸机程序当做BL1来使用了。

裸机实验之SD卡下载

背景知识

一般情况下，用USB下载来调试裸机程序比较方便：但有时候电脑使用dnw会蓝屏，这时候只有使用SD卡下载调试程序是不错的选择。

把OM5打开到GND，以从SD通道启动。

从SD启动时会先从iNand（SD0）启动执行，当iNand启动做检验和失败才会转为启动SD2，而我们做裸机实验时是通过SD2来启动裸机程序镜像的，因此需要先破坏内部的iNand的uboot才可以强迫开发板从SD2启动去执行我们的裸机程序。

擦除开发板iNand中uboot的方法

之前说过linux和android系统下的擦除方法

这里当系统还没有安装时，在uboot下擦除的方法：

help movi //查看movi命令的帮助

movi write u-boot 0x30000000 //任选的一个地址中的数写到uboot中，导致校验和失败。

windows下制作SD启动卡

用SD卡烧写工具。方法等同于第三部分讲过的SD卡刷机的操作。

linux下制作启动SD卡

SD卡启动和USB启动的优劣势对比：如果电脑本身支持usb启动下载而不蓝屏，建议就用usb下载调试。

手动安装交叉编译工具链

windows中装软件的特点

windows中装软件使用安装包，安装包解压后有两种情况：一种是一个安装文件（.exe .msi）,双击进行安装，下一步直到安装完毕。安装完毕后在桌面生成快捷方式，我们使用快捷方式启动这些程序；另一种是所谓的绿色软件、免安装软件。这种不用安装，直接解压打开里面的.exe就可以运行

linux中装软件的特点

linux中装软件比较复杂，一般有以下几种方法

第一种：在线安装。譬如ubuntu中使用apt-get install vim 来安装vim软件

第二种：自己下载安装包来安装，这种方式的缺陷就是你不知道你下载的安装包和你的系统是否匹配。

第三种：最装逼的一种方式，就是源代码安装。

总结：我们安装交叉编译工具链（arm-linux-gcc）实际采用第二种安装方式。

交叉编译工具链的选择：

我们选择交叉编译工具链的原则：和我们所使用的目标平台（给哪款SoC编程）尽量去匹配，譬如我们开发S5PV210的程序就是用arm-2009q3这个版本，因为三星官方在开发S4PV210时就是用这个版本的交叉编译工具链，这样可以最大限度的避免稀奇古怪的问题出现。

交叉编译工具链的安装：

步骤1：现将安装包从windows中共享到linux中去。可以用共享文件夹，也可以用samba，也可以cuteftp。

步骤2：解压 tar -jxvf arm-2009q3.tar.bz2

到此相当于程序已经安装完毕，真正的应用程序安装在/usr/local/arm/arm-2009q3/bin目录下

注释：linux中的目录管理方法。技术角度来说：linux中所有目录的性质是一模一样的，所以把软件安装到哪里都行，但是如果胡乱放置，将来程序可能不好找。一般用户自己安装的软件都放在/user目录下。我们安装arm-linux-gcc 就在/user/local/底下创建一个arm文件夹，然后装到里面。

安装后的测试

到真正的应用程序安装目录下，去执行arm-linux-gcc -v

执行方法是：./arm-none-linux-gnueabi-gcc -v

执行后可以得到一长串输出，其中有“gcc version 4.4.1”字样，即表示安装成功。wo

将工具链导出到环境变量

export PATH=/usr/local/arm/arm-2009q3/bin:$PATH 终端关闭后失效

vi ~/.bashrc

export ...

source .bashrc

为工具链创建arm-liunx-xxx符号链接

cd /usr/local/arm/arm-2009q3/bin/

ln arm-none-linux-gnueabi-gcc -s arm-linux-gcc

Makefile 基本工作原理

目标：目标顶格写，后面是冒号（冒号后面是依赖）

依赖：依赖是产生目标的原材料

命令：命令前面一定是Tab，不能使顶格，也不能是多个空格，命令是要生成那个目标需要的动作。

其一，当我们执行make xx的时候，Makefile会自动执行xx这个目标下面的命令语句

其二，当我们make xx 的时候，是否执行命令取决于依赖的。依赖如果成立就执行命令，否 则不执行。

其三，我们直接执行make和make第一个目标效果是一样的（第一个目标其实就是默认目 标）

Makefile进阶：看《跟我一起学Makefile》作者 陈皓

mkv210\_image.c的使用演示

该程序的Makefile：

vi Makefile （使用第一个字母大写或者全部小写都可以）

led.bin:start.o

arm-none-linux-gnueabi-ld -Ttext 0x0 -o led.elf $^ #链接

arm-none-linux-gnueabi-objcopy -o binary led.elf led.bin #从.o文件生成可以烧写的镜像

arm-none-linux-gnueabi-objdump -D led.elf >led\_elf.dis #反汇编

gcc mkv210\_image.c -o mkmini210 # 上位机编译usb启动到SD卡启动程序

./mkmini210 led.bin 210.bin #按规则mkv210\_image.c从led.bin 生成210.bin 镜像

%.o: %.S

arm-none-linux-gnueabi-gcc -o $@ $< -c #编译.S文件生成.o

%.o: %.c

arm-none-linux-gnueabi-gcc -o $@ $< -c #编译.c文件生成.o

clean:

rm \*.o \*.elf \*.bin \*.dis mkmini210 -f

裸机程序中的Makefile是把程序的编译和链接过程分开的。（平时使用gcc a.c -o exe 这种方式来编译时，实际上是把编译和链接过程一步完成了，在内部实际上编译和链接永远是分开独立进行的，编译要用编译器gcc，链接要使用链接器ld）

链接器得到的led.elf其实就是我们的可执行程序，（如果是在操作系统下，这个led.elf就可以执行了）但是在嵌入式裸机中我们需要的是可以烧写的文件（可烧写的文件就叫镜像image），因此我们需要用led.elf为原材料来制作镜像，制作工具是交叉编译工具链中的arm-linux-objcopy

我们使用arm-linux-objdump工具进行反编译（反汇编），反汇编其实就把编译后的elf格式的可执行程序反过来得到的对应的汇编程序，得到他的汇编源代码。我们使用反汇编主要是用来学习。见本部分最后一节。

mkv210\_image.c这个程序其实最终不是在开发板上执行的，而是在主机linux（就是用来执行make对整个项目进行编译的那个机器）中执行的，因此编译这个程序用gcc而不是arm-linux-gcc.这个.c文件编译后得到一个可执行程序mkmini210，目的是通过执行这个mkmini210程序而由led.bin得到210.bin。（210.bin是通过SD卡启动时的裸机镜像，这个镜像这个镜像需要由led.bin来加工得到，加工的具体原理要看mkv210\_image.c）

mkv210\_image.c的主要作用就是由usb启动时使用的led.bin制作得到由sd卡启动的镜像210.bin

背景知识：S5PV210的启动过程回顾

分析启动过程可知：210启动后先执行内部IROM中的BL0，BL0执行完成后会根据OMpin的配置选择一个外部设备来启动，（有很多，我们实际使用的有两个，usb启动和SD卡启动）。

在usb启动时内部BL0读取到BL1后不做校验，直接从BL1的实质内部0xd0020010开始执行，因此usb启动的景象leb.bin不需要头信息，因此我们从usb启动是直接将镜像下载到0xd0020010去执行即可，不管头信息了：从SD启动时，BL0会首先读sd卡得到完整的镜像，（完整指的是led.bin和16字节的头），然后BL0会自己根据你的实际镜像（指led.bin）来计算一个校验和checksum，然后和你完整镜像的头部中的checksum来对比，如果对应则执行BL1，如果不对应则启动失败（会转入执行2st启动，即SD2启动，如果这里已经是2st启动了，这里校验和不通过就死定了）。

mkv210\_image.c的作用：为BL1添加校验头

我们编译连接时只得到了led.bin，这个210.bin得到的和交叉编译工具链是完全无关的，由led.bin得到210.bin的过程是三星的S5PV210所特有的，因此我们需要自己去完成，为此我们写了mkv210\_image.c来完成。

mkv210\_image.c

#include<stdio.h>

#include<string.h>

#include<stdlib.h>

#define BUFSIZE (16\*1024)

#define IMG\_SIZE (16\*1024)

#define SPL\_HEADER\_SIZE 16

#define SPL\_HEADER "S5PC110 HEADER " //这里是为了站位16字节，任意内容都可

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

FILE \*fp

char \*Buf,\*a;

int BufLen;

int nbytes,fileLen;

unsigned int checksum,count;

int i;

//检验3个参数

if(argc!=3){

printf("Usage:mkx210 <source file> <destination file> \n");

return -1;

}

//分配16KB的buffer

BufLen=BUFSIZE;

Buf=(char\*)malloc(BufLen);

if(!Buf){

printf("Alloc buffer failed!\n");

return -1;

}

memset(Buf,0x00,BufLen);

//读源bin到buffer

//打开源bin

fp=fopen(argv[1],"rb");

if(fp==NULL){

printf("source file open error\n");

free(Buf);

return -1;

}

//获取源bin长度

fseek(fp,0L,SEEK\_END); //定位到文件尾部

fileLen=ftell(fp); //得到文件长度

fseek(fp,0L,SEEK\_SET); //再次定位到文件头

//源bin长度不得超过16KB-16byte

count=(fileLen<(IMG\_SIZE-SPL\_HEADER\_SIZE))?fileLen:(IMGZ\_SIZE-SPL\_HEADER\_SIZE);

//buffer[0-15]存放"S5PC110 HEADER "

memcopy(&Buffer[0],SPL\_HEADER,SPL\_HEADER\_SIZE);

//读源bin到buffer[16]

nbytes=fread(Buf+SPL\_HEADER,1,count,fp);

if(nbyites!=count){

printf("source file read error\n");

free(Buf);

fclose(fp);

return -1;

}

fclose(fp);

//计算校验和

//从弟16个byte开始统计buffer中共有几个1

//从弟16byte开始计算，把buffer中的所有的字节数据加合起来的到的结果

a=Buf+SPL\_HEADER\_SIZE;

for(i=0,checksum=0;i<IMG\_SIZE-SPL\_HEADER\_SIZE;i++)

checksum+=(0x000000FF)&\*a++;

//将校验和保存在buffer[8-15]

a=Buf+8;

\*((unsigned int \*)a)=checksum;

//拷贝buffer中的内容到目的bin

//打开目的bin

fp=fopen(argv[2],"wb");

if(fp==NULL){

printf("destination file open error\n");

free(Buf);

return -1;

}

//将16KB的buffer拷贝到目的bin中

a=Buf;

nbytes=fwrite(a,1,BufLen,fp);

if(nbytes!=BufLen){

printf("destination file write error \n");

free(Buf);

fclose(fp);

return -1;

}

free(Buf);

fclose(fp);

return 0;

}

整个程序的工作流分析

整个程序中首先申请一个16KB大小的buffer，然后把所有内容按照各自的位置填充进去，最终把填充好的buffer写入到一个文件（名叫210.bin）就形成了我们想要的镜像。

代码详解：

1. 检验用户传参是不是三个
2. 分配16KB buffer并且填充为0
3. .........

glibc读写文件接口

linux中要读取一个文件，可以使用fopen打开文件，fread读取文件读完之后关闭文件。

要写文件用fwrite来写，这些函数是glibc的库函数，在linux中用man 3 可以查找

校验和的计算方法

算法：校验和其实就是需要校验的内存区域中，所有内存中的内容按照字节为单位来进行相加，最终相加的和即为校验和，实现时大家要注意指针的类型为char\*

查阅原理图了解板载Led硬件接法

4颗Led正极接3.3v，负极接了SoC上的一个GPIO，具体详细的接法是：

D22：GPJO\_3

D23：GPJO\_4

D24：GPJO\_5

D25：PWMTOUT1(GPD0\_1)

GPIO的概念的引入

GPIO：general purpuse input output通用的输入输出

GPIO就是芯片上的引脚，芯片上的引脚有些不是GPIO，只有一部分是，作为GPIO的这类引脚，它的功能和特点是可以被编程控制他的工作模式，也可以编程控制他的电压高低等

如果当时设计电路时候，把LED接在非GPIO上就无法控制亮灭了。

我们要操作的硬件是LED，但是LED实际是通过GPIO间接控制的，所以当前我们实际要操作的设备其实是SoC的GPIO。要操作这些GPIO，必须通过设置他们的寄存器。

查阅手册可知，GPJ0相关的寄存器有以下

GPJ0CON,（GPJ0 control）GPJ0控制寄存器，用来配置各引脚的工作模式

GPJ0DAT, （GPJ0 data）当引脚配置为input/output模式时，寄存器的相应位和引脚的电平高低相对应

GPJ0PUD,（pull up down）控制引脚内部的弱上拉、下拉。

GPJ0DRV,（driver）配置GPIO的驱动能力

GPJ0NPDN,

GPJ0PUDPDN，

在驱动LED点亮时，应将GPIO配置为output模式

实际上正真操控LED的硬件，主要的有：GPJ0CON,GPJ0DAT 这么2个。

如何点亮LED，编程步骤是：

1、操控GPJ0CON寄存器中，选中output模式

2、操控GPJ0DAT寄存器，相应位设置为0；

GPxCON、GPxDAT寄存器分析

GPJ0端口一共有8个引脚，分别记为：GPJ0\_0~GPJ0\_7，相关重要的寄存器就是GPJ0CON和GPJ0DAT；GPJ0CON寄存器中设置8个引脚的工作模式（32/8，每个引脚可以分到4位，譬如GPJ0\_0对应的bit位bit0~bit3。工作方法是：给相应的寄存器位写入相应的值，该引脚硬件就会按照相应的模式去工作。譬如给bit12~bit15写入0x0001,GPJ0\_3引脚就变成输出模式了）

编程先决条件：

1. 硬件接法和引脚：GPJ0\_3 GPJ0\_4 GPJ0\_5 低电平亮/高电平灭
2. GPJ0CON（0xE02000240）寄存器和GPJ0DAT(0xE0200244)寄存器
3. 工程管理：Makefile等

led.S

/\*

\*文件名：led.s

\*作者：查伟伟

\*描述：这是我们一步步点亮LED的第一个裸机程序

\*/

#define GPJ0CON 0xE0200240

#define GPJ0DAT 0xE0200244

\_start:

//第一步：把0x11111111写入0x02000240(GPJ0CON)的位置

ldr r0, =0x11111111 //从后面的=可以看出用的是ldr伪指令，因为需要编译器来判断这个数是合法立即数还是非法立即数。一般写代码都用ldr伪指令

ldr r1, =GPJ0CON

str r0, [r1] //寄存器间接寻址，功能是把r0中的数写入到r1中的数为地址的寄存器中去

//第二步：把0x0写入0xE0200244(GPJ0DAT)位置

ldr r0, =0x0

ldr r1, =GPJ0DAT

str r0, [r1] //把0写入到GPJ0DAT寄存器中，即引脚输出为低电平，LED点亮

flag:

b flag //这两行写了一个死循环。因为裸机程序是直接在CPU上运行的，CPU

会逐行运行裸机程序知道CPU断电关机。如果所有的程序代码都执行完了，CPU就会跑飞，（跑飞以后就是未定义了，所以千万不能让CPU跑飞）不让程序跑飞的方法就是在整个程序执行完后添加死循环。

b是跳转指令，跳转到自己继续跳转，所以死循环。

编译、下载、运行看结果

编译时用我们的工程管理，直接make编译得到led.bin和210.bin

下载运行可以用usb启动dnw下载，也可以用SD卡烧录下载，根据自己的情况使用，

一般是使用usb下载，如果电脑主板插上dnw会死机，那只有用SD卡下载了。

总结和回顾：

软件控制硬件的接口就是寄存器。

代码优化：

1. 用宏定义定义寄存器地址

2、用b . 实现死循环

3、用.global将\_start链接属性改为外部，消除链接时的警告。

#define GPJ0CON 0xE0200240

#define GPJ0DAT 0xE0200244

.global \_start //把\_start链接属性改为外部，这样其他文件就能看见\_start了。

\_start:

//第一步：把0x11111111写入0x02000240(GPJ0CON)的位置

ldr r0, =0x11111111 //从后面的=可以看出用的是ldr伪指令，因为需要编译器来判断这个数是合法立即数还是非法立即数。一般写代码都用ldr伪指令

ldr r1, =GPJ0CON

str r0, [r1] //寄存器间接寻址，功能是把r0中的数写入到r1中的数为地址的寄存器中去

//第二步：把0x0写入0xE0200244(GPJ0DAT)位置

ldr r0, =0x0

ldr r1, =GPJ0DAT

str r0, [r1] //把0写入到GPJ0DAT寄存器中，即引脚输出为低电平，LED点亮

b . //.代表当前地址

在写代码时用位运算让编译器帮我们计算这个特定值。

常用的位运算：

位与（&） 位或（|） 位非（~） 移位（左移<< 右移>>）

只亮中间两颗led灯3和5，上述代码可改为：

ldr r0, =((1<<3)|(1<<5)) //等效于0b00101000

编写一个延时函数实现LED闪烁效果。

闪烁效果：=亮+延时+灭+延时+亮+延时+........

延时函数原理

在汇编中实现延时的方法：用一些没有目的代码来执行消耗时间，达到延时效果

#define GPJ0CON 0xE0200240

#define GPJ0DAT 0xE0200244

.global \_start //把\_start链接属性改为外部，这样其他文件就能看见\_start了。

\_start:

//第一步：把0x11111111写入0x02000240(GPJ0CON)的位置

ldr r0, =0x11111111 //从后面的=可以看出用的是ldr伪指令，因为需要编译器来判断这个数是合法立即数还是非法立即数。一般写代码都用ldr伪指令

ldr r1, =GPJ0CON

str r0, [r1] //寄存器间接寻址，功能是把r0中的数写入到r1中的数为地址的寄存器中去

//第二步：把0x0写入0xE0200244(GPJ0DAT)位置

ldr r0, =((0<<3)|(0<<4)|(0<<5)) //等效于0b00000000,全亮。

ldr r1, =GPJ0DAT

str r0, [r1] //把0写入到GPJ0DAT寄存器中，即引脚输出为低电平，LED点亮

bl delay //使用bl进行函数调用，把当前地址放入lr，以便返回。

ldr r0, =((1<<3)|(1<<4)|(1<<5)) //等效于0b00111000,全灭。

ldr r1, =GPJ0DAT

str r0, [r1] //把0写入到GPJ0DAT寄存器中，即引脚输出为低电平，LED点亮

bl delay //使用bl进行函数调用，把当前地址放入lr，以便返回。

ldr r0, =((0<<3)|(0<<4)|(0<<5)) //等效于0b00000000,全亮。

ldr r1, =GPJ0DAT

str r0, [r1] //把0写入到GPJ0DAT寄存器中，即引脚输出为低电平，LED点亮

bl delay //使用bl进行函数调用，把当前地址放入lr，以便返回。

b . //.代表当前地址

delay:

ldr r2, =100000

ldr r3, =0x0

delay\_loop:

sub r2, r2, #1 //r2=r2-1

cmp r2,r3 //cmp会影响CPSR中的z标志位，下一句中eq就会成立

bne delay\_loop

mov pc, lr //函数调用返回

加入循环：

#define GPJ0CON 0xE0200240

#define GPJ0DAT 0xE0200244

.global \_start //把\_start链接属性改为外部，这样其他文件就能看见\_start了。

\_start:

//第一步：把0x11111111写入0x02000240(GPJ0CON)的位置

ldr r0, =0x11111111 //从后面的=可以看出用的是ldr伪指令，因为需要编译器来判断这个数是合法立即数还是非法立即数。一般写代码都用ldr伪指令

ldr r1, =GPJ0CON

str r0, [r1] //寄存器间接寻址，功能是把r0中的数写入到r1中的数为地址的寄存器中去

//第二步：把0x0写入0xE0200244(GPJ0DAT)位置

flash:

ldr r0, =((0<<3)|(0<<4)|(0<<5)) //等效于0b00000000,全亮。

ldr r1, =GPJ0DAT

str r0, [r1] //把0写入到GPJ0DAT寄存器中，即引脚输出为低电平，LED点亮

bl delay //使用bl进行函数调用，把当前地址放入lr，以便返回。

ldr r0, =((1<<3)|(1<<4)|(1<<5)) //等效于0b00111000,全灭。

ldr r1, =GPJ0DAT

str r0, [r1] //把0写入到GPJ0DAT寄存器中，即引脚输出为低电平，LED点亮

bl delay //使用bl进行函数调用，把当前地址放入lr，以便返回。

b flash

delay:

ldr r2, =100000

ldr r3, =0x0

delay\_loop:

sub r2, r2, #1 //r2=r2-1

cmp r2,r3 //cmp会影响CPSR中的z标志位，下一句中eq就会成立

bne delay\_loop

mov pc, lr //函数调用返回

汇编编写及调用函数的方式

汇编中整个汇编的主程序是一个死循环，这个死循环是我们的汇编程序的主体，类似于C中的main函数。其它函数必须写在主死循环程序的外面，不然会出错。

汇编编写delay延时函数时，要注意函数的初始化和函数体的位置，不能把初始化写在了循环体内。

汇编中调用函数用bl指令，子函数中最后用mov pc, lr 来返回

实现流水灯效果

流水灯又叫跑马灯，实现的效果就是：挨着的LED灯依次点亮熄灭（同时只有一个灯是亮着的）

LED1亮延时+LED2亮延时+ LED3亮延时+循环

#define GPJ0CON 0xE0200240

#define GPJ0DAT 0xE0200244

.global \_start //把\_start链接属性改为外部，这样其他文件就能看见\_start了。

\_start:

//第一步：把0x11111111写入0x02000240(GPJ0CON)的位置

ldr r0, =0x11111111 //从后面的=可以看出用的是ldr伪指令，因为需要编译器来判断这个数是合法立即数还是非法立即数。一般写代码都用ldr伪指令

ldr r1, =GPJ0CON

str r0, [r1] //寄存器间接寻址，功能是把r0中的数写入到r1中的数为地址的寄存器中去

//要实现流水灯效果只要在主循环中实现一圈的流水显示效果即可

flash:

ldr r0, =((0<<3)|(1<<4)|(1<<5)) //可以改成 ldr r0, = ~(1<<3) 更简单

ldr r1, =GPJ0DAT

str r0, [r1] //把0写入到GPJ0DAT寄存器中，即引脚输出为低电平，LED点亮

bl delay //使用bl进行函数调用，把当前地址放入lr，以便返回。

ldr r0, =((1<<3)|(0<<4)|(1<<5)) //等效于0b00111000,全灭。

ldr r1, =GPJ0DAT

str r0, [r1] //把0写入到GPJ0DAT寄存器中，即引脚输出为低电平，LED点亮

bl delay //使用bl进行函数调用，把当前地址放入lr，以便返回。

ldr r0, =((1<<3)|(1<<4)|(0<<5)) //等效于0b00111000,全灭。

ldr r1, =GPJ0DAT

str r0, [r1] //把0写入到GPJ0DAT寄存器中，即引脚输出为低电平，LED点亮

bl delay //使用bl进行函数调用，把当前地址放入lr，以便返回。

b flash

delay:

ldr r2, =100000

ldr r3, =0x0

delay\_loop:

sub r2, r2, #1 //r2=r2-1

cmp r2,r3 //cmp会影响CPSR中的z标志位，下一句中eq就会成立

bne delay\_loop

mov pc, lr //函数调用返回

编程控制一个硬件的步骤：

1. 分析一个硬件的工作原理：
2. 分析原理图
3. 分析数据手册
4. 找到相关的SFR
5. 写代码设置寄存器得到想要的效果

反汇编工具objdump的使用简介

反汇编原理&为什么要反汇编

arm-linux-objdump -D led.elf > led\_elf.dis

objdump是gcc工具链中的反汇编工具，作用是由编译连接好的elf格式的可执行程序反过来得到的汇编源代码 -D表示反汇编 >左边的是elf的可执行程序（反汇编时的原材料），>右边的是反汇编生成的反汇编程序

反汇编的原因有以下：

1. 逆向破解。
2. 调试程序时反汇编代码可以帮助理解程序（我们学习时使用objdump主要目的是这个），尤其是在理解链接脚本、链接地址等概念时。
3. 把C语言源代码编译连接生成的可执行程序反汇编后得到的汇编代码，可以帮助我们理解C语言和汇编语言之间的对应关系。非常有助于深入理解C语言。

反汇编文件的格式和看法

（汇编 assembly 反汇编 dissembly）

标号地址、标号名字、指令地址、指令机器码、指令机器码反汇编得到的指令

扩展：ARM汇编中用地支池的方式来实现非法立即数

初识指令地址

下载烧录执行的bin文件，内部其实是一条一条的指令机器码，这些指令每一条都有一个指令地址，这个地址是链接的时候ld给指定的（ld根据我们写的链接脚本来指定）

展望：反汇编工具帮助我们分析链接脚本

反汇编的时候得到的指令地址是链接器考虑了链接脚本之后得到的地址，而我们写代码时通过指定连接脚本来让链接器给我们链接合适的地址

但是有时候我们写的链接脚本有误（或者我们不知道这个链接脚本会怎么样），这时候可以通过看返回汇编文件来分析这个链接脚本的效果，看是不是我们想要的，如果不是可以改了再看。