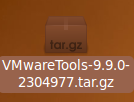
## 安装VMware tools：

安装VMware tools可以方便在windows跟Linux虚拟机里面互相拷贝文件。

1.点击虚拟机，然后点击安装VMware tools



2.将文件拷贝到linux桌面，右击解压。

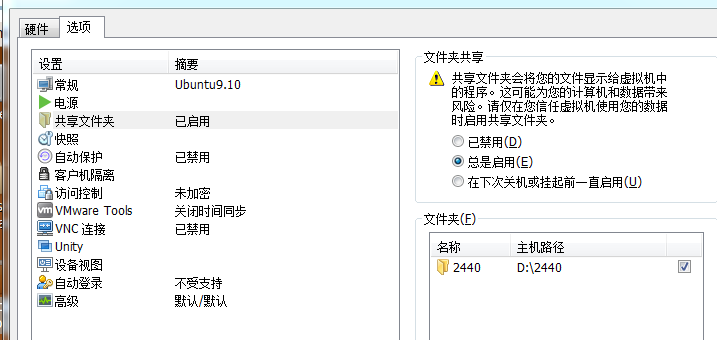


3.进入解压生成的文件夹，./vmware­-install.pl进行安装。

4.安装过程中出现提示一直敲回车即可，安装完毕后重启linux，便可以在window跟linux之间复制文件了。

### 共享文件夹

建立一个文件夹，linux跟windows可以共享，同时编辑这个文件夹里面的内容。



Linux下mnt里面可以查看共享的文件夹。



## Makefile 的规则

目标：依赖 1 依赖2

命令

例子：

生成的文件hello，依赖hello.c a.c

hello: hello.c ,a.c

gcc -o hello hello.c a.c

使用命令make，就可以生成可执行文件hello。

命令执行的条件有两点：

1. 没有hello文件
2. hello.c或a.c修改了

通配符方式：

hello: hello.o a.o

gcc –o $@ -c $^ //两个依赖hello.o 跟a.o用$^表示

%.o : %.c

gcc –o $@ -c $< //目标可以用$@来表示，第一个依赖可以用$<表示

### 变量的定义与引用

object = main.o

$(object)

默认的情况下，make命令会在当前目录下按顺序找寻文件名为“GNUmakefile”、“makefile”、“Makefile”的文件，找到了解释这个文件。

make命令开始时，会把找寻include所指出的其它Makefile，并把其内容安置在当前的位置。就好像C/C++的#include指令一样。如果文件都没有指定绝对路径或是相对路径的话，make会在当前目录下首先寻找，如果当前目录下没有找到，那么，make还会在下面的几个目录下找：  
  
1、如果make执行时，有“-I”或“--include-dir”参数，那么make就会在这个参数所指定的目录下去寻找。  
2、如果目录<prefix>/include（一般是：/usr/local/bin或/usr/include）存在的话，make也会去找。  
  
如果有文件没有找到的话，make会生成一条警告信息，但不会马上出现致命错误。它会继续载入其它的文件，一旦完成makefile的读取，make会再重试这些没有找到，或是不能读取的文件，如果还是不行，make才会出现一条致命信息。如果你想让make不理那些无法读取的文件，而继续执行，你可以在include前加一个减号“-”。如：  
  
-include <filename>  
其表示，无论include过程中出现什么错误，都不要报错继续执行。和其它版本make兼容的相关命令是sinclude，其作用和这一个是一样的。

1、读入所有的Makefile。  
2、读入被include的其它Makefile。  
3、初始化文件中的变量。  
4、推导隐晦规则，并分析所有规则。  
5、为所有的目标文件创建依赖关系链。  
6、根据依赖关系，决定哪些目标要重新生成。  
7、执行生成命令。

1-5步为第一个阶段，6-7为第二个阶段。第一个阶段中，如果定义的变量被使用了，那么，make会把其展开在使用的位置。但make并不会完全马上展开，make使用的是拖延战术，如果变量出现在依赖关系的规则中，那么仅当这条依赖被决定要使用了，变量才会在其内部展开。

### Export

如果你要传递变量到下级Makefile中，那么你可以使用这样的声明：

    export <variable ...>

### 文件的引用

在Makefile使用include关键字可以把别的Makefile包含进来，这很像C语言的#include，被包含的文件会原模原样的放在当前文件的包含位置。include的语法是：  
  
include <filename>

### subset

函数调用，很像变量的使用，也是以“$”来标识的，其语法为：$( )或${ }。  
参数间以逗号分隔，函数名和参数间以空格分隔。函数调用以“$”开头，以圆括号或花括号把函数名和参数括起。

$(subst FROM, TO, TEXT)，即将字符串TEXT中的子串FROM变为TO。

### patsubst

建立一个简单的Makefile  
src=$(wildcard \*.c ./sub/\*.c)  
dir=$(notdir $(src))  
obj=$(patsubst %.c,%.o,$(dir) )  
  
all:  
@echo $(src)  
@echo $(dir)  
@echo $(obj)  
@echo "end"  
  
执行结果分析：  
第一行输出：  
a.c b.c ./sub/sa.c ./sub/sb.c  
  
wildcard把指定目录 ./ 和 ./sub/ 下的所有后缀是c的文件全部展开。  
  
第二行输出：  
a.c b.c sa.c sb.c  
notdir把展开的文件去除掉路径信息  
  
第三行输出：  
a.o b.o sa.o sb.o  
  
在$(patsubst %.c,%.o,$(dir) )中，patsubst把$(dir)中的变量符合后缀是.c的全部替换成.o，  
任何输出。  
或者可以使用  
obj=$(dir:%.c=%.o)  
效果也是一样的。  
  
这里用到makefile里的替换引用规则，即用您指定的变量替换另一个变量。  
它的标准格式是  
$(var:a=b) 或 ${var:a=b}  
它的含义是把变量var中的每一个值结尾用b替换掉a

### Origin

origin 函数的作用是告诉你变量是哪里来的，其出生状况如何，他并不改变变量。其语法是：

$(origin <variable>)

上面，<variable> 为变量的名字，而不是引用，所以一般没有 $ 字符在前。origin 函数通过返回值来告诉你 <variable> 的出生情况。下面用实例说明：

1. 当从来未定义过该变量时，origin 函数返回 "undefined" 。如下面的 Makefile 代码：

all:

@echo $(origin V)

运行输出：

$ make

undefined

2. 如果该变量为环境变量，那么返回 "enviroment" 。如下面的 Makefile 代码：

all:

@echo $(origin USER)

运行输出：

$ make

environment

其中 USER 这个变量为系统定义的当前用户，使用 env 命令可以看到。

3. 如果变量是个默认定义，那么返回 "default"。如下面的 Makefile 代码：

all:

@echo $(origin CC)

运行输出：

$ make

default

4. 如果一个变量被定义在 Makefile 文件中，那么返回 "file" 。如下面的 Makefile 代码：

V := 1

all:

@echo $(origin V)

运行输出：

$ make

file

5. 如果变量来自命令行，那么返回 "command line" 。如下面的 Makefile 代码：

all:

@echo $(origin MyVar)

运行方法：

$ make MyVar="Are you ok?"

command line

6. 如果变量被 override 被重新定义过，那么返回 "override"。如下面的 Makefile 代码：

verride SHELL = /bin/sh

all:

@echo $(origin SHELL)

运行输出：

$ make

override

上面，SHELL 原本是个环境变量，但在 Makefile 里被 override 指示符重定义过。

7. 如果变量是自动化变量(如 $@, $< 等)，那么返回 "automatic" 。如下面的 Makefile 代码：

all:

@echo $(origin @)

运行输出：

$ make

Automatic

### @

@echo $(CURDIR)

@不将echo命令显示出来，CURDIR是makefile的内嵌变量，显示当前路径

### Wildcard

通配符

它的用法是：$(wildcard PATTERN...)

一般我们可以使用“$(wildcard \*.c)”来获取工作目录下的所有的.c文件列表。复杂一些用法；可以使用“$(patsubst %.c,%.o,$(wildcard \*.c))”，首先使用“wildcard”函数获取工作目录下的.c文件列表；之后将列表中所有文件名的后缀.c替换为.o。这样我们就可以得到在当前目录可生成的.o文件列表。因此在一个目录下可以使用如下内容的Makefile来将工作目录下的所有的.c文件进行编译并最后连接成为一个可执行文件：

### 编译其他目录下面的makefile

all:

make -C ./test version

编译当前目录下test目录里面的makefile，目标为version

假如在目录M下执行makefile  
target1:  
     make -C A target\_n    
     make -C B target\_n  
     ……  
  
make -C A，进入A目录，执行A目录下的makefile，这时你的工作环境就在A目录，你可以在目录A下makefile中的target\_n里面echo `pwd`看下

### Make –f

Make –f make.linux

指定编译make.linux文件

Make命令只会去找名称为makefile的文件。Make –f可以指定执行哪个名字的makefile

### arm-linux-ld

arm-linux-ld -Ttext 0x30000000 head.o leds.o -o sdram\_elf

输出sdram.elf文件

0x30000000处是代码运行时，应该处于的位置

第一个放置的是head.o接着放leds.o

Attention：

两个概念：

1.代码存放的位置

2.代码运行时应该位于哪里

代码存放的位置根据下载器而定，可以下载到0x0地址或者其他地址。

但是要运行代码的话，此时代码必须要在0x30000000处

OUTPUT\_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm");指定输出可执行文件elf格式，32位ARM指令，小端模式  
/\*OUTPUT\_FORMAT("elf32-arm", "elf32-arm", "elf32-arm")\*/  
OUTPUT\_ARCH(arm) 指定输入平台为ARM  
ENTRY(\_start) 指定输出可执行文件的起始代码为\_start  
SECTIONS  
{  
 . = 0x00000000;   定位当前地址为0地址

 . = ALIGN(4);        代码以四字节对齐  
 .text      :               指定代码段  
 {  
   cpu/arm920t/start.o (.text)  代码的第一个代码段  
   \*(.text)         其他代码段  
 }

 . = ALIGN(4);  
 .rodata : { \*(.rodata) }  指定只读数据段

 . = ALIGN(4);  
 .data : { \*(.data) } 指定读写数据段

 . = ALIGN(4);  
 .got : { \*(.got) } 指定got段，got段式是uboot自定义的一个段，非标准段

 \_\_u\_boot\_cmd\_start = .;其赋值为当前位置，即起始位置  
 .u\_boot\_cmd : { \*(.u\_boot\_cmd) } u\_boot\_cmd段，Uboot把所有的uboot命令放在该段。  
 \_\_u\_boot\_cmd\_end = .;把其赋值为当前位置，即结束位置

 . = ALIGN(4);  
 \_\_bss\_start = .;  把\_\_bss\_start赋值为当前位置，即bss段的开始位置  
 .bss : { \*(.bss) }  ；指定bss段  
 \_end = .;；把\_end赋值为当前位置，即bss段的结束位置  
}

### -nostdinc –I

指定编译器从-I后面跟着的文件夹中查找头文件

### arm-linux-ar

生成lib文件

arm-linux-ar -r -o libc.a div64.o lib1funcs.o ctype.o muldi3.o printf.o string.o vsprintf.o

### 反汇编

arm-linux-objdump –d u-boot > dump\_u-boot.txt

arm-none-eabi-objdump.exe

生成u-boot对应的汇编

### -O2

GNU编译器提供-O选项供程序优化使用:  
-O 提供基础级别的优化  
-O2 提供更加高级的代码优化,会占用更长的编译时间  
-O3 提供最高级的代码优化

### Findstring

$(findstring <find>,<in> )  
  
名称：查找字符串函数——findstring。  
功能：在字串<in>中查找<find>字串。  
返回：如果找到，那么返回<find>，否则返回空字符串。  
示例：

### = ：= ？=

在Makefile中我们经常看到 = := ?= +=这几个赋值运算符，那么他们有什么区别呢？我们来做个简单的实验

新建一个Makefile，内容为：  
ifdef DEFINE\_VRE  
    VRE = “Hello World!”  
else  
endif

ifeq ($(OPT),define)  
    VRE ?= “Hello World! First!”  
endif

ifeq ($(OPT),add)  
    VRE += “Kelly!”  
endif

ifeq ($(OPT),recover)  
    VRE := “Hello World! Again!”  
endif

all:  
    @echo $(VRE)

敲入以下make命令：  
make DEFINE\_VRE=true OPT=define 输出：Hello World!  
make DEFINE\_VRE=true OPT=add 输出：Hello World! Kelly!  
make DEFINE\_VRE=true OPT=recover  输出：Hello World! Again!  
make DEFINE\_VRE= OPT=define 输出：Hello World! First!  
make DEFINE\_VRE= OPT=add 输出：Kelly!  
make DEFINE\_VRE= OPT=recover 输出：Hello World! Again!

从上面的结果中我们可以清楚的看到他们的区别了  
= 是最基本的赋值  
:= 是覆盖之前的值  
?= 是如果没有被赋值过就赋予等号后面的值  
+= 是添加等号后面的值

## Shell

<http://c.biancheng.net/cpp/view/7005.html>

<http://www.runoob.com/linux/linux-shell-basic-operators.html>

### #！bin/bash

#!/bin/bash是指此脚本使用/bin/bash来解释执行。

其中，#!是一个特殊的表示符，其后，跟着解释此脚本的shell路径。

bash只是shell的一种，还有很多其它shell，如：sh,csh,ksh,tcsh,...

### $#

$0 这个程式的执行名字  
$n 这个程式的第n个参数值，n=1..9  
$\* 这个程式的所有参数,此选项参数可超过9个。  
$# 这个程式的参数个数  
$$ 这个程式的PID(脚本运行的当前[进程ID](https://www.baidu.com/s?wd=%E8%BF%9B%E7%A8%8BID&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YkuWDdPvw9PHfvP1DdPHRL0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnWmzrHRkP1mv)号)  
$! 执行上一个背景指令的PID(后台运行的最后一个进程的[进程ID](https://www.baidu.com/s?wd=%E8%BF%9B%E7%A8%8BID&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YkuWDdPvw9PHfvP1DdPHRL0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnWmzrHRkP1mv)号)  
$? 执行上一个指令的返回值 (显示最后命令的退出状态。0表示没有错误，其他任何值表明有错误)  
$- 显示shell使用的当前选项，与set命令功能相同  
$@ 跟$\*类似，但是可以当作数组用

### 赋值

File=book

=两边不能有空格，makefile可以允许有空格

### 执行其他文件夹中的脚本&&

脚本中执行其他目录的脚本或者命令，可以用&&连接两条命令，当前面命令执行正常时，才执行后面命令  
比如：  
cd /root/replace/scripts && ./run.out  
cd /root/replace/scripts && ./b.sh

### dos2unix

dos2unix命令用来将DOS格式的文本文件转换成UNIX格式的（DOS/MAC to UNIX text file format converter）。

DOS下的文本文件是以\r\n作为断行标志的，表示成十六进制就是0D 0A。而Unix下的文本文件是以\n作为断行标志的，表示成十六进制就是 0A

### Find

find命令原理：从指定的起始目录开始，递归地搜索其各个子目录，查找满足寻找条件的文件，并可以对其进行相关的操作。

格式：find [查找目录] [参数] [匹配模型]

例如：

1、find . -name "\*.sh"

    查找在当前目录（及子目录）下找以sh结尾的文件。

2、find . -perm 755

    查找在当前目录（及子目录）下找属性为755的文件。

3、find -user root

    查找在当前目录（及子目录）下找属主为root的文件。

4、find /var -mtime -5

     查找在/var下找更改时间在5天以内的文件。

5、find /var -mtime +3

     查找在/var下找更改时间在3天以前的文件。

6、find /etc -type l

     查找在/etc下查找文件类型为|的链接文件。

7、find . -size +1000000c

      查找在当前目录（及子目录）下查找文件大小大于1M的文件，1M是1000000个字节。

8、find . -perm 700 |xargs chmod 777

      查找出当前目录（及子目录）下所有权限为700的文件，并把其权限重设为777。

9、find . -type f |xargs ls -l

      查找出文件并查看其详细信息。

### 文件符比较

-e filename  如果 filename存在，则为真  [ -e /var/log/syslog ]

-d filename  如果 filename为目录，则为真  [ -d /tmp/mydir ]

-f filename  如果 filename为常规文件，则为真  [ -f /usr/bin/grep ]

-L filename  如果 filename为符号链接，则为真  [ -L /usr/bin/grep ]

-r filename  如果 filename可读，则为真  [ -r /var/log/syslog ]

-w filename  如果 filename可写，则为真  [ -w /var/mytmp.txt ]

-x filename  如果 filename可执行，则为真  [ -L /usr/bin/grep ]

filename1-nt filename2  如果 filename1比 filename2新，则为真  [ /tmp/install/etc/services -nt /etc/services ]

filename1-ot filename2  如果 filename1比 filename2旧，则为真  [ /boot/bzImage -ot arch/i386/boot/bzImage ]

### 数值，字符串比较

在shell中字符串与数值的比较方法是不同的，要注意区分  
  
整数比较：  
    -eq       等于,如:if [ "$a" -eq "$b" ]  
    -ne       不等于,如:if [ "$a" -ne "$b" ]  
    -gt       大于,如:if [ "$a" -gt "$b" ]  
    -ge       大于等于,如:if [ "$a" -ge "$b" ]  
    -lt       小于,如:if [ "$a" -lt "$b" ]  
    -le       小于等于,如:if [ "$a" -le "$b" ]  
    <       小于(需要双括号),如:(("$a" < "$b"))  
    <=       小于等于(需要双括号),如:(("$a" <= "$b"))  
    >       大于(需要双括号),如:(("$a" > "$b"))  
    >=       大于等于(需要双括号),如:(("$a" >= "$b"))  
  
字符串比较：  
    =       等于,如:if [ "$a" = "$b" ]  
    ==     等于,如:if [ "$a" == "$b" ], 与=等价  
               注意:==的功能在[[]]和[]中的行为是不同的,如下:  
               1 [[ $a == z\* ]]    # 如果$a以"z"开头(模式匹配)那么将为true  
               2 [[ $a == "z\*" ]] # 如果$a等于z\*(字符匹配),那么结果为true  
               3  
               4 [ $a == z\* ]      # File globbing 和word splitting将会发生  
               5 [ "$a" == "z\*" ] # 如果$a等于z\*(字符匹配),那么结果为true  
  
    !=      不等于,如:if [ "$a" != "$b" ]， 这个操作符将在[[]]结构中使用模式匹配.  
    <       小于,在ASCII字母顺序下.如:  
               if [[ "$a" < "$b" ]]  
               if [ "$a" \< "$b" ]     在[]结构中"<"需要被转义.  
    >       大于,在ASCII字母顺序下.如:  
           if [[ "$a" > "$b" ]]  
           if [ "$a" \> "$b" ]  在[]结构中">"需要被转义.  
    -z       字符串为"null".就是长度为0.  
    -n       字符串不为"null"

### 脚本传入参数

|  |  |
| --- | --- |
| **参数处理** | **说明** |
| $# | 传递到脚本的参数个数 |
| $\* | 以一个单字符串显示所有向脚本传递的参数 |
| $$ | 脚本运行的当前进程ID号 |
| $! | 后台运行的最后一个进程的ID号 |
| $@ | 与$\*相同，但是使用时加引号，并在引号中返回每个参数。 |
| $- | 显示Shell使用的当前选项，与set命令功能相同。 |
| $? | 显示最后命令的退出状态。0表示没有错误，其他任何值表明有错误 |

"name=$1"中$1为系统提供的位置参数，$0代表程序的名称（./test.sh），[$1 $2...]从1开始为传递的参数。linux系统除了提供位置参数还提供内置参数，所有内置参数如下：

$0 ----当前脚本文件名

$n ----传递给脚本或函数的参数，n是一个数字，表示第几个参数，$1表示第一个参数，$2表示第二个参数 ...

$# ----传递给脚本或函数的参数个数

**$\* ----传递给脚本或函数的所有参数，当它被双引号（" "）包含时，"$\*" 会将所有的参数作为一个整体，以"$1 $2 ... $n"的形式输出所有参数**

**$@ ----传递给脚本或函数的所有参数，当它被双引号（" "）包含时，与$\*稍有不同，"$@" 会将各个参数分开，以"$1" "$2" … "$n" 的形式输出所有参数**

**$\* 和$@不被双引号(" ")包含时，都以"$1" "$2" … "$n" 的形式输出所有参数**

### 函数传入参数

脚本名称为test\_shell.sh ,内容如下：

function test()

{

echo $1

echo $2

}

echo $1

echo $2

test 111 222

执行命令./test\_shell.sh hah eee

输出为

root@fanny-OptiPlex-790:/home/richard/8QXP/shell\_test# ./test\_shell.sh hah eee

hah

eee

111

222

### TEE

在执行Linux命令时，我们可以把输出重定向到文件中，比如 ls >a.txt，这时我们就不能看到输出了，如果我们既想把输出保存到文件中，又想在屏幕上看到输出内容，就可以使用tee命令了

格式：tee

只输出到标准输出，因为没有指定文件嘛。

格式：tee file

输出到标准输出的同时，保存到文件file中。如果文件不存在，则创建；如果已经存在，则覆盖之。（If a file being written to does not already exist, it is created. If a file being written to already exists, the data it previously  
contained is overwritten unless the `-a' option is used.）

### 判断文件是否存在

function deploy()

{

cp $1 $2

md5sum $1

md5sum $2/$1

}

function find\_bin\_and\_copy()

{

binfile="./build/output/imx8qx\_SCU/rtlstub/\*.map"

# binfile="./hah/\*.bin"

if [ -f $binfile ];then

cp -r ./build/output ../../../../ROM\_unit\_test

return 0

else

return 1

fi

}

#copy makefile and test code to the destination folder

deploy Makefile ../git/imx-rom/mcurom\_src/code/boot\_rom/imx8\_b0/SCU/boot\_test

deploy seco\_test.c ../git/imx-rom/mcurom\_src/code/boot\_rom/imx8\_b0/SCU/boot\_test

#clean the ROM project and re-make

cd ../git/imx-rom/mcurom\_src/code &&make clean && make PRODUCT=imx8qx\_SCU RTL\_STUB=1 ZEBU\_DEBUG=1 TEST=test\_seco\_set\_stage all

#detect if the map has been generated, copy the build folder to ROM\_unit\_test folder once it generated

find\_bin\_and\_copy

ret=$?

while [ $ret -ne 0 ]

do

echo "find bin file"

find\_bin\_and\_copy

ret=$?

if [ $ret -eq 0 ];then

break

fi

done

### source

source命令用法：  
source FileName  
作用:在当前bash环境下读取并执行FileName中的命令。  
注：该命令通常用命令“.”来替代。  
如：source .bash\_rc 与 . .bash\_rc 是等效的。  
注意：source命令与shell scripts的区别是，  
source在当前bash环境下执行命令，而scripts是启动一个子shell来执行命令。这样如果把设置环境变量（或alias等等）的命令写进scripts中，就只会影响子shell,无法改变当前的BASH,所以通过文件（命令列）设置环境变量时，要用source 命令。

## BAT

### 暂停

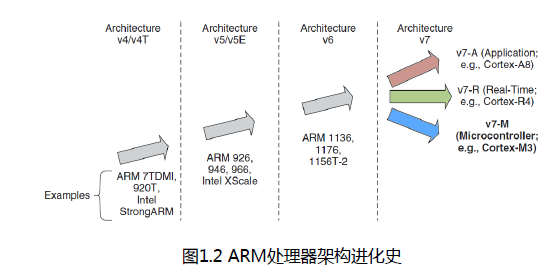
echo .&pause

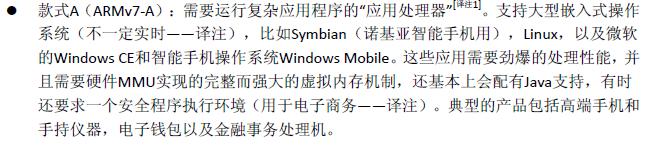
### 延时

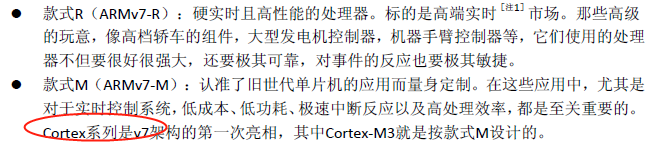
choice /t 1 /d y /n >nul

## ARM 架构V7，CM3

### 处理器架构史



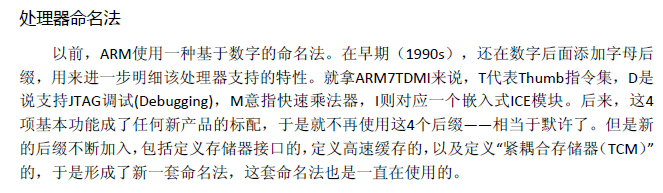




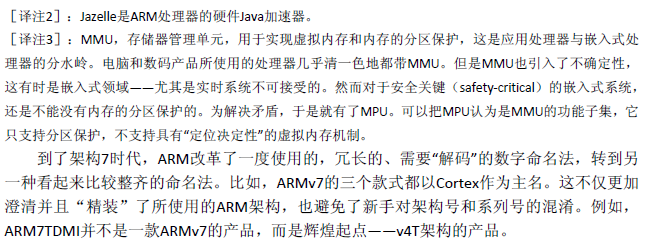
ARMv7架构的Cortex-A5,A7,A8,A9,A12,A15，

[ARMv8架构](https://www.baidu.com/s?wd=ARMv8%E6%9E%B6%E6%9E%84&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1Y3rHuWPhD4P1-hPH0zrHIB0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPHckrHcvrHDdrj03n101nj6d)的Cortex-A53,A57，A72

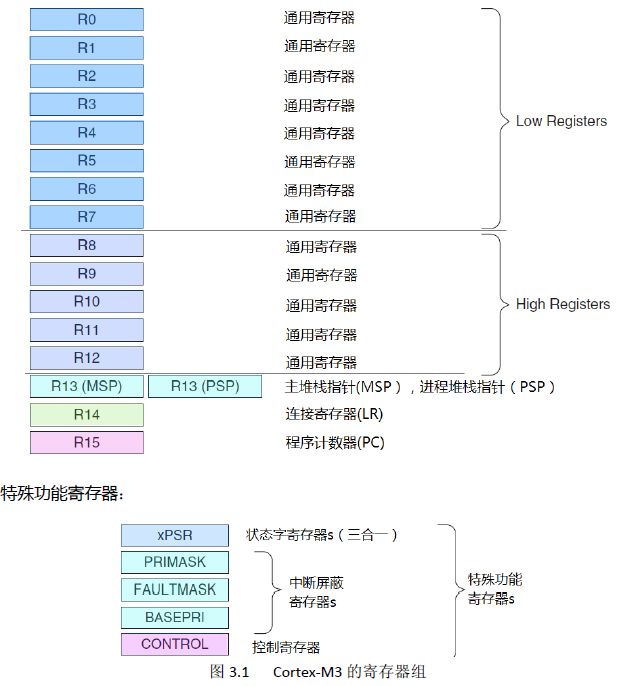
### 处理器命名



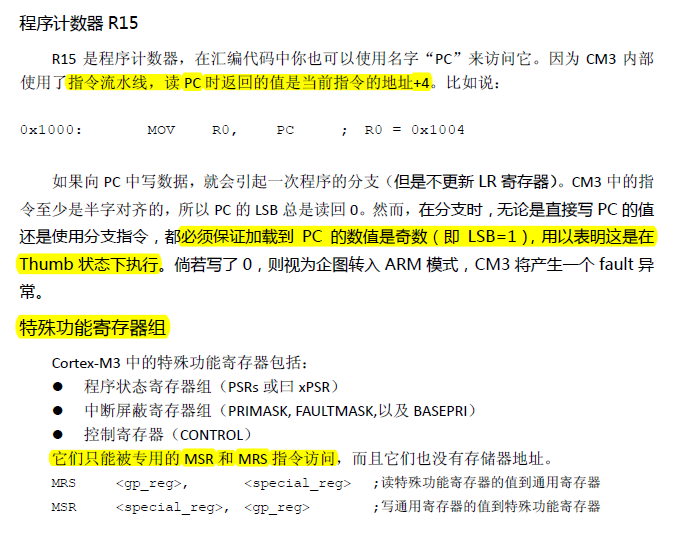




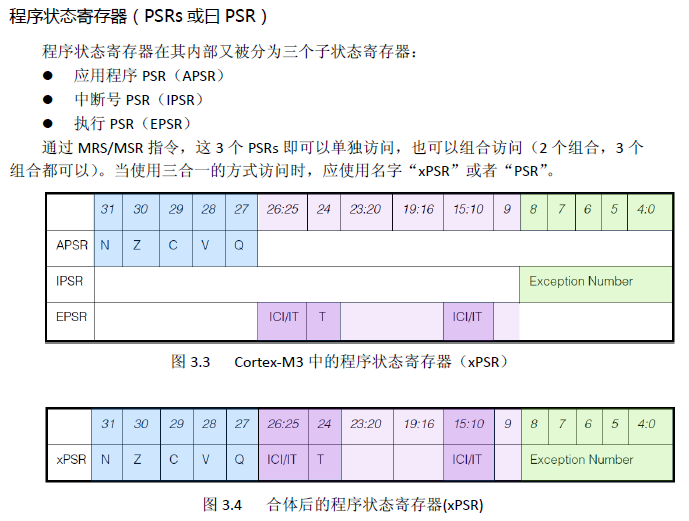
### 寄存器简介



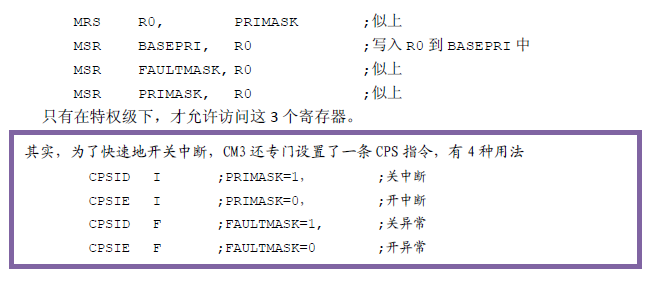
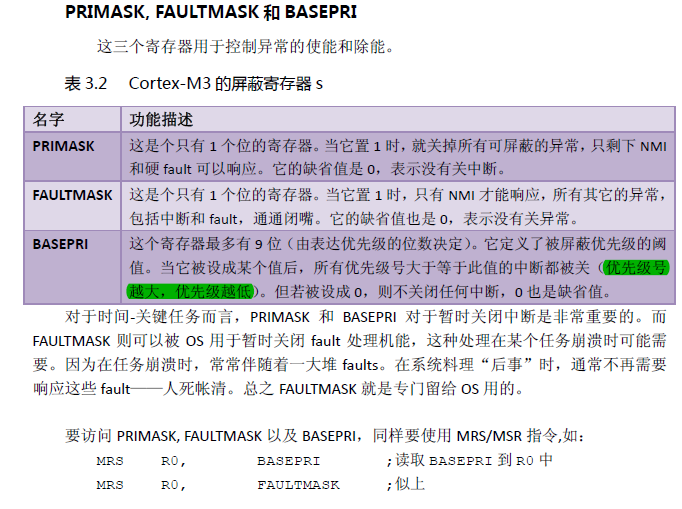
#### 程序计数寄存器PC，为何PC值是奇数



#### 状态寄存器

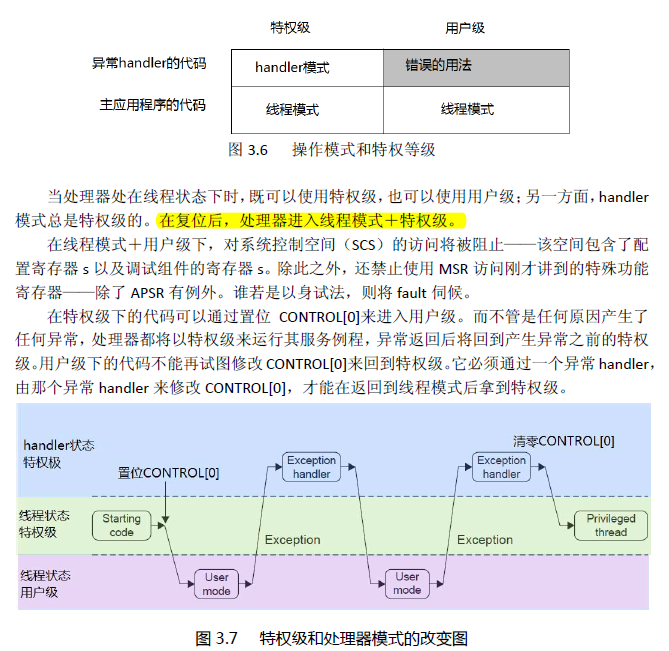


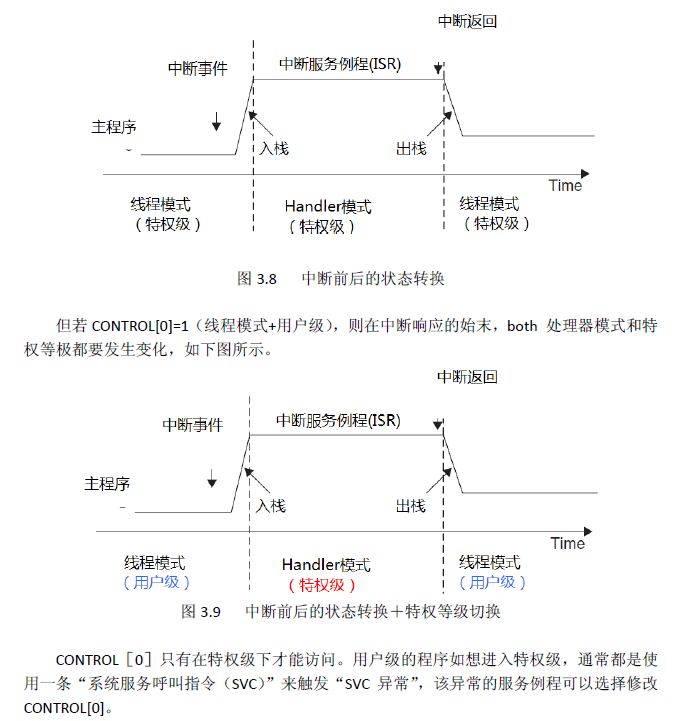
#### 异常使能除能寄存器



#### 控制寄存器，CPU操作模式

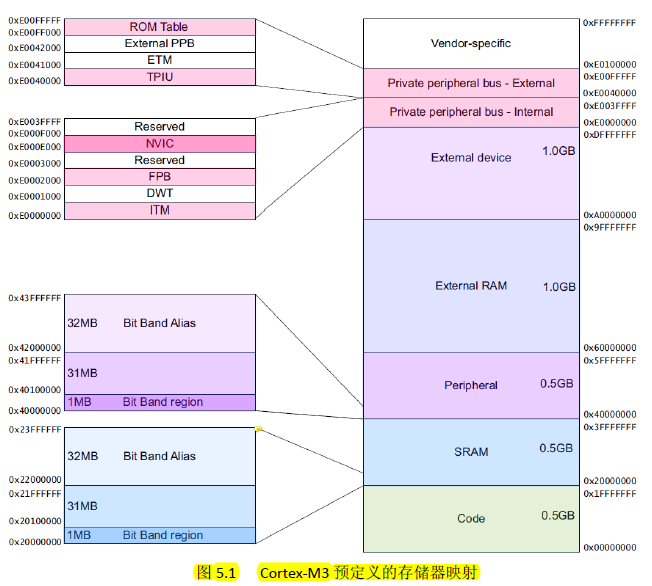




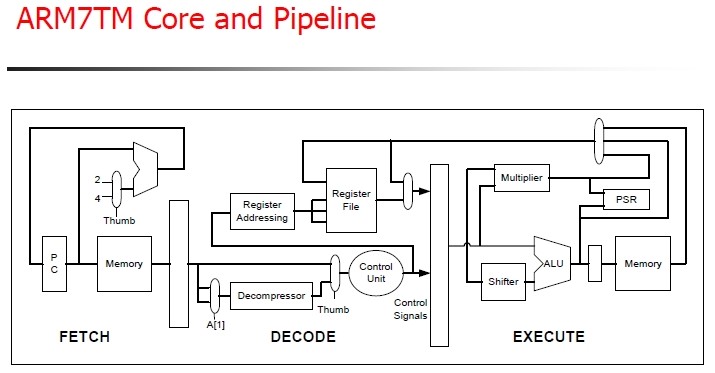


处理中断时，CPU处于特权模式

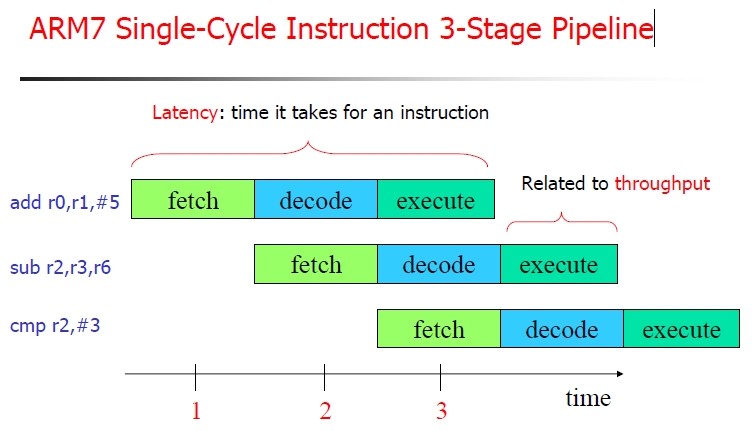
### ARM M3 预定义Memory map



### 三级流水线



首先，对于ARM7对应的流水线的执行情况，如下面这个图所示：

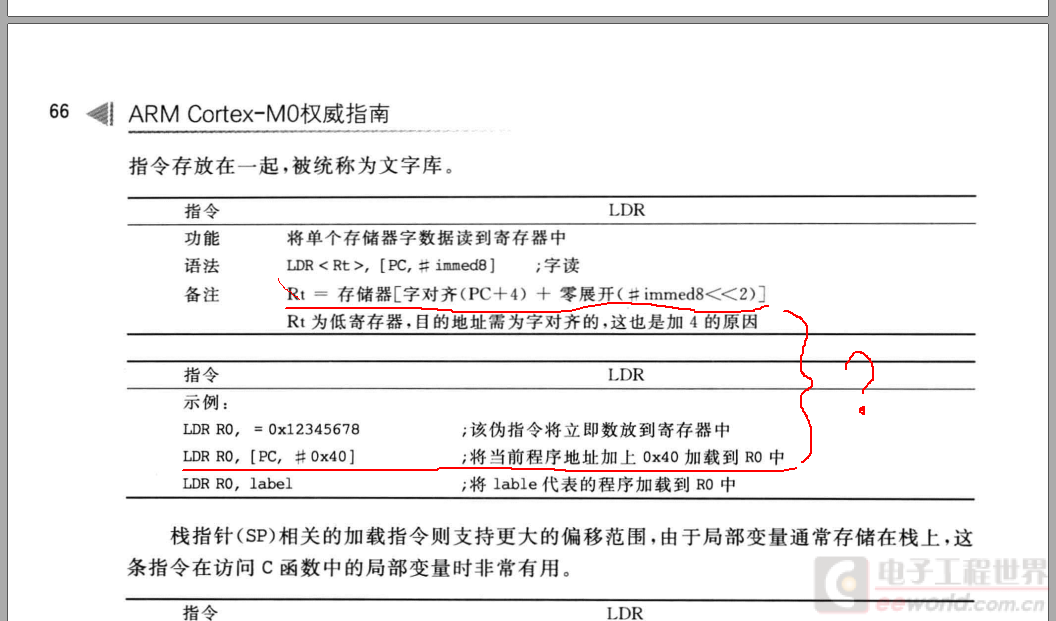


执行第一条指令 add r0,r1, #5的时候，PC已经指向cmp r2,#3的地址了，

**PC总是指向当前正在"取值"所对应的地址,而不是当前正在执行代码所对应的位置。**

1、LDR命令的理解

(1)LDR命令



关于“Rt = 存储器[字对齐(PC + 4)] + 零展开(#immed8<<2)” 的理解：

(A)PC值是当前正在执行命令的地址+4(三级流水线)，即PC总是指向当前正在取指的地址。

PC是32位，需要按字对齐，即要求其最低两bit为0。

(B)#immed8<<2中，immed8指的是机器码的低字节。



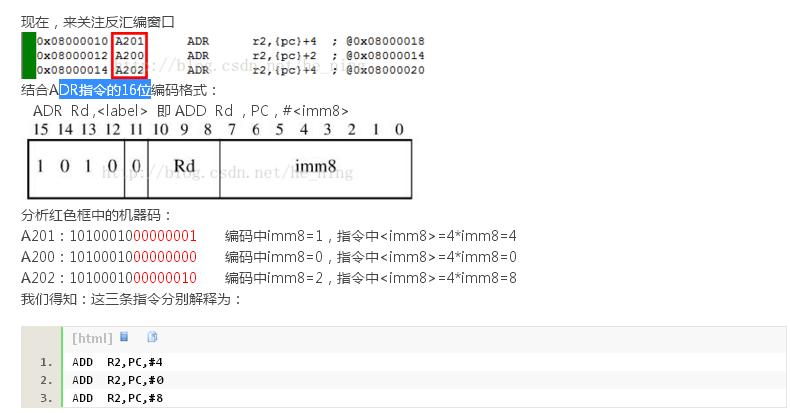
以上述图片中其中一行反汇编代码为例，进行举例说明：

代码地址 机器码 汇编代码 (pc+40)所对应的地址

0x000000D4 480A LDR r0， [pc，#40]； @0x00000100

其中机器码“480A”中的“A<<2”对应immed8，即immed8 = (0x0A<<2)=40。

下面是一个类似的示例：



(2)PC值

谈谈PC值的计算方法，PC值=当前正在执行指令所在地址+4，加4的原因是Cortex-M0内核使用的是3级流水线结构，且采用的是16位的THUMB指令，

按16位数据进行取指（即PC每次加2），由于是3级流水线结构，在指令中使用到的PC值的时候实际上PC已经指向第三条指令地址了，所以要加4。

至于在各个内核的处理器中汇编代码是否能够通用的问题，也能解决，原因是我们实际在写汇编代码的时候几乎是不会用到PC作为基址的寻址方式（除非

是自找难受或者是对所写程序了如指掌的大神），编译器会在编译的时候自动处理，所以不会存在由于流水线级数不同而导致的代码不通用的问题。

(3)关于0x000000D4位置"LDR r0，[pc，#40]；@0x00000100"的理解：

LDR  R0 , =0x50000100，使用的是32位立即数寻址方式；但是经过Keil编译之后的汇编代码却是：LDR  r0 , [pc , #40]，使用的是基于PC寄存器的基址寻址方

式，也就是说编译器在编译时会自动去选择合适的方式来实现相同的功能，而且从我了解到的资料中，大部分的立即数寻址方式在最终生成的代码中其实都是基

于“文字池”，然后使用PC作为基址的寻址方式。

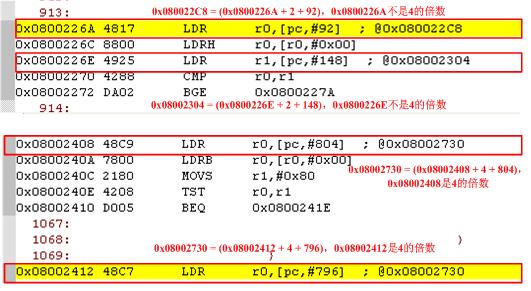
2、PC值与当前正在执行命令所在地址的关系(以Cortex-M0为例进行说明)

在实际的调试过程中，PC的取值并不是每次都是当前正在执行命令的地址+4，偶尔是当前正在执行命令的地址+2，让我困惑了很久。

后来突然想到PC是32位，要求以字对齐，联想是不是和当前正在执行命令的地址有关。查看几处反汇编代码，和我理解的一致，但不知道是不是最根本的原因。

如果我的理解有误，也希望达人能够指正。

下面的图片中标记了PC取值不同的地方：



最后一个例子错误，应该+2

当前执行指令的PC+4后，需要向下字对齐。

## Jlink

### 读取数据到文件中

savebin  F:\3518\uboot\_tools\3518a.bin 0x82000000 0x1000000（16M）

## Linux指令

### 提升文件夹读写权限

sudo chmod –R 777 user/local

将user/local文件夹变成读写权限

777代表读写执行权限

### 登录linux

ssh [maxx@10.192.241.185](mailto:maxx@10.192.241.185)

### 从linux拷贝文件到本地C盘

scp [maxx@10.192.241.185:/home/xxx.file](mailto:maxx@10.192.241.185:/home/xxx.file) /c

### 从linux拷贝文件夹到本地C盘

scp –r maxx@10.192.241.185:/home /c

### 本地文件上传到linux

Chmod 777 /home/Richard //linux机器上提升文件夹的权限

scp file.txt maxx@10.192.241.185:/home/richard

### 删除文件\文件夹

rm –f file

rm –rf folder

rm –rf /home/test/\* //删除test 文件夹下面所欲的内容

r : recursive递归的，删除文件夹下面的所有内容

f : force强制删除

### 挂载U盘

首先使用命令查看U盘是否挂载，sdb

df -h

若没有，先在mnt下创建一个USB的目录，然后使用命令查看设备名称

fdisk –l

cat /proc/partitions

如果fdisk –l没有反应，加上sudo重试

挂载USB设备

Mount /dev/sdb1 /mnt/usb/

卸载USB

Umount /dev/sdb1

### SD卡格式化、建立分区

先umount SD card

umount /dev/sdb1

umount /dev/sdb2

df -h

删除分区

fdisk /dev/sdb

d

1

d

2

查看分区信息

P

建立扇区

x

h //header

s //sector

c //cylines

r //return

建立分区

n p 1

n e 2

第一个分区配置成FAT32格式，uboot从这里启动，windows可识别

t 1 c

激活第一分区的bootable

a 1

写入分区信息，并退出格式化菜单

w

格式化两个分区

mkfs.msdos -F32 /dev/sdb1 -n LABEL1

mkfs.ext3 -L LABEL2 /dev/sdb2

### dd

dd if=uboot.imx of=/dev/sdb bs=1024 seek=1

向/dev/sdb中写入ubbot.imx，一次写入1024字节，跳过/dev/sdb的前1\*1024后写入uboot.imx

### 查看文件大小

Ls –lht

### 查找变量Grep

在当前文件夹中的所有文件中查找关键字file

grep –nir ‘file’ ./\*

### 查找文件Find

Find / -name hah

查找当前文件夹下面hah的文件或者文件夹

### 添加环境变量

export PATH=$PATH:/usr/sbin/

添加/usr/sbin/到环境变量中

### Ln

ln -s /bin/less /usr/local/bin/less

如果/usr/local/bin/less 不存在，自动创建less文件夹

这是linux中一个非常重要命令，请大家一定要熟悉。它的功能是为某一个文件在另外一个位置建立一个同不的链接，这个命令最常用的参数是-s,具体用法是：ln -s 源文件 目标文件。   
当 我们需要在不同的目录，用到相同的文件时，我们不需要在每一个需要的目录下都放一个必须相同的文件，我们只要在某个固定的目录，放上该文件，然后在其它的 目录下用ln命令链接（link）它就可以，不必重复的占用磁盘空间

-s 是代号（symbolic）的意思。   
这 里有两点要注意：第一，ln命令会保持每一处链接文件的同步性，也就是说，不论你改动了哪一处，其它的文件都会发生相同的变化；第二，ln的链接又软链接 和硬链接两种，软链接就是ln -s \*\* \*\*,它只会在你选定的位置上生成一个文件的镜像，不会占用磁盘空间，硬链接ln \*\* \*\*,没有参数-s, 它会在你选定的位置上生成一个和源文件大小相同的文件，无论是软链接还是硬链接，文件都保持同步变化。   
如果你用ls察看一个目录时，发现有的文件后面有一个@的符号，那就是一个用ln命令生成的文件，用ls -l命令去察看，就可以看到显示的link的路径了。   
  
ln是linux中又一个非常重要命令，它的功能是为某一个文件在另外一个位置建立一个同步的链接.当我们需要在不同的目录，用到相同的文件时，我们不需要在每一个需要的目录下都放一个必须相同的文件，我们只要在某个固定的目录，放上该文件，然后在 其它的目录下用ln命令链接（link）它就可以，不必重复的占用磁盘空间。

。

### >

> 直接把内容生成到指定文件，会覆盖源文件中的内容，还有一种用途是直接生成一个空白文件，相当于touch命令

>>尾部追加，不会覆盖掉文件中原有的内容

### Vi

按ese进入命令模式

I 进行编辑后再按esc，然后输入wq，保存

q！强制推迟

177gg进入指定行

查找字符串

Esc进入命令模式，输入/后，输入需要查找的字符串，然后回车

### Samba

/etc/samba/smb.conf

### Tee

Ls | tee 1.txt

Linux桌面显示ls的内容的同时将显示的内容输出到1.txt中

### PS

grep 是搜索  
例如： ps -ef | grep java  
表示查看所有进程里 CMD 是 java 的进程信息  
ps -aux | grep java  
-aux 显示所有状态  
ps  
3. kill 命令用于终止进程  
例如： kill -9 [PID]  
-9 表示强迫进程立即停止  
通常用 ps 查看进程 PID ，用 kill 命令终止进程  
网上关于这两块的内容

## 汇编

### EXPORT

用import，该标识符表明要调用的函数为本模块外部定义的  
export标识符表示本模块中定时的符号可以为外部模块使用

### .global

.text .global 是arm-gcc编译器的关键词。  
.text 指定了后续编译出来的内容放在代码段【可执行】;  
.global 告诉编译器后续跟的是一个全局可见的名字【可能是变量，也可以是函数名】；  
在本例中，\_start是一个函数的起始地址，也是编译、链接后程序的起始地址。由于程序是通过加载器来加载的，必须要找到 \_start名字的函数，因此\_start必须定义成全局的，以便存在于编译后的全局符合表中，供其它程序【如加载器】寻找到。

### .word

word expression就是在当前位置放一个word型的值，这个值就是expression    
举例来说，    
\_rWTCON:    
.word 0x15300000    
就是在当前地址，即\_rWTCON处放一个值0x15300000

### C语言跟汇编混合编程

C语言与汇编语言混合编程应遵守的规则

ARM编程中使用的C语言是标准C语言，ARM的开发环境实际上就是嵌入了一个C语言的集成开发环境，只不过这个开发环境与ARM的硬件紧密相关。

在使用C语言时，要用到和汇编语言的混合编程。若汇编代码较为简洁，则可使用直接内嵌汇编的方法；否则要将汇编程序以文件的形式加入到项目中，按照ATPCS(ARM/Thumb过程调用标准，ARM/Thumb Procedure Call Standard)的规定与C程序相互调用与访问。

在C程序和ARM汇编程序之间相互调用时必须遵守ATPCS规则。ATPCS规定了一些子程序间调用的基本规则，哪寄存器的使用规则，堆栈的使用规则和参数的传递规则等。

1)寄存器的使用规则

子程序之间通过寄存器r0~r3来传递参数，当参数个数多于4个时，使用堆栈来传递参数。此时r0~r3可记作A1~A4。

在子程序中，使用寄存器r4~r11保存局部变量。因此当进行子程序调用时要注意对这些寄存器的保存和恢复。此时r4~r11可记作V1~V8。

寄存器r12用于保存堆栈指针SP，当子程序返回时使用该寄存器出栈，记作IP。

寄存器r13用作堆栈指针，记作SP。

寄存器r14称为链接寄存器，记作LR。该寄存器用于保存子程序的返回地址。

寄存器r15称为程序计数器，记作PC。

2)堆栈的使用规则

ATPCS规定堆栈采用满递减类型(FD,Full Descending)，即堆栈通过减小存储器地址而向下增长，堆栈指针指向内含有效数据项的最低地址。

3)参数的传递规则

整数参数的前4个使用r0~r3传递，其他参数使用堆栈传递；浮点参数使用编号最小且能够满足需要的一组连续的FP寄存器传递参数。

子程序的返回结果为一个32位整数时，通过r0返回；返回结果为一个64位整数时，通过r0和r1返回；依此类推。结果为浮点数时，通过浮点运算部件的寄存器F0、D0或者S0返回。

2、汇编程序调用C程序的方法

汇编程序的书写要遵循ATPCS规则，以保证程序调用时参数正确传递。在汇编程序中调用C程序的方法为：首先在汇编程序中使用IMPORT伪指令事先声明将要调用的C语言函数；然后通过BL指令来调用C函数。

例如在一个C源文件中定义了如下求和函数：

int add(int x,int y){

return(x+y);

}

调用add()函数的汇编程序结构如下：

IMPORT add ;声明要调用的C函数

……

MOV r0,1

MOV r1,2

BL add ;调用C函数add

……

当进行函数调用时，使用r0和r1实现参数传递，返回结果由r0带回。函数调用结束后，r0的值变成3。

3、C程序调用汇编程序的方法

C程序调用汇编程序时，汇编程序的书写也要遵循ATPCS规则，以保证程序调用时参数正确传递。在C程序中调用汇编子程序的方法为：首先在汇编程序中使用EXPORT伪指令声明被调用的子程序，表示该子程序将在其他文件中被调用；然后在C程序中使用extern关键字声明要调用的汇编子程序为外部函数。

例如在一个汇编源文件中定义了如下求和函数：

EXPORT add ;声明add子程序将被外部函数调用

……

add ;求和子程序add

ADD r0,r0,r1

MOV pc,lr

……

在一个C程序的main()函数中对add汇编子程序进行了调用：

extern int add (int x,int y); //声明add为外部函数

void main(){

int a=1,b=2,c;

c=add(a,b); //调用add子程序

……

}

当main()函数调用add汇编子程序时，变量a、b的值会给了r0和r1，返回结果由r0带回，并赋值给变量c。函数调用结束后，变量c的值变成3。

4、C程序中内嵌汇编语句

在C语言中内嵌汇编语句可以实现一些高级语言不能实现或者不容易实现的功能。对于时间紧迫的功能也可以通过在C语言中内嵌汇编语句来实现。内嵌的汇编器支持大部分ARM指令和Thumb指令，但是不支持诸如直接修改PC实现跳转的底层功能，也不能直接引用C语言中的变量。

嵌入式汇编语句在形式上独立定义的函数体，其语法格式为：

\_\_asm

{

指令[;指令]

……

[指令]

}

其中“\_\_asm”为内嵌汇编语句的关键字，需要特别注意的是前面有两个下划线。指令之间用分号分隔，如果一条指令占据多行，除最后一行外都要使用连字符“\”。

5、基于ARM的C语言与汇编语言混合编程举例

下面给出了一个向串口不断发送0x55的例子：

该工程的启动代码使用汇编语言编写，向串口发送数据使用C语言实现，下面是启动代码的整体框架：

……

IMPORT Main

AREA Init,CODE,READONLY;

ENTRY

……

BL Main ;跳转到Main()函数处的C/C++程序

……

END ;标识汇编程序结束

下面是使用C语言编写的主函数：

#include "..\inc\config.h" //将有关硬件定义的头文件包含进来

unsigned char data; //定义全局变量

void main(void){

Target\_Init(); //对目标板的硬件初始化

Delay(10); //延时

data=0x55; //给全局变量赋值

while(1) {

Uart\_Printf("%x",data); //向串口送数

Delay(10);

}

}

Bne:不相等跳转指令

### ldr伪指令

LDR伪指令的形式是“LDR Rn,=expr”

COUNT [EQU](https://www.baidu.com/s?wd=EQU&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1Yknjn3nyRdmHcdPjc1rjRL0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3En1b1nHDdPWnk) 0x40003100  
LDR R1,=COUNT

就是将地址0x40003100存放到R1寄存器中

### ldr加载指令

此指令是arm指令，非伪指令

LDR R0，[R1] ；将存储器地址为R1的字数据读入寄存器R0。

LDR R0，[R1，R2] ；将存储器地址为R1+R2的字数据读入寄存器R0。

LDR R0，[R1，#8] ；将存储器地址为R1+8的字数据读入寄存器R0。

LDR R0，[R1],R2 ；将存储器地址为R1的字数据读入寄存器R0，并将R1+R2的值存入R1。

LDR R0，[R1],#8 ；将存储器地址为R1的字数据读入寄存器R0，并将R1+8的值存入R1。

### STR

STR指令的格式为：  
STR{条件}  源寄存器，<存储器地址>  
STR指令用亍从源寄存器中将一个32位的字数据传送到存储器中。该指令在程序设计中比较常用，寻址方式灵活多样，使用方式可参考指令LDR。  
指令示例：  
STR R0，[R1]，＃8             ；将R0中的字数据写入以R1为地址的存储器中，并将新地址R1＋8写入R1。

STR R0，[R1，＃8]     ；将R0中的字数据写入以R1＋8为地址的存储器中。

str     r1, [r0]          ；将r1寄存器的值，传送到地址值为r0的（存储器）内存中

### bne

1:

ldr r4, [r2], #4 @ 读取设置值，并让r2加4

str r4, [r1], #4 @ 将此值写入寄存器，并让r1加4

cmp r1, r3 @ 判断是否设置完所有13个寄存器

bne 1b @ 若没有写成，继续

mov pc, lr @ 返回

bne 1b什么意思呢？一开始老是在找1b的标号处，找了些许时间也没有发现哪里有这个标号。

后来查找相关资料发现，原来0~9的数字为局部标签。局部标签可以重复使用，语法为：

xf:往前跳的意思，就是还未执行的程序，x代表0~9的某个标签，f代表forward的意思。

xb:往后跳的意思，回到原来已经执行过的语句，x达标0~9的某个标签，b代表backward的意思。

上面bne 1b 的意思是，如果r1！=r3 即r1的地址的不等于r3的地址，也就是还没设置完这13个寄存器，就要重新跳到标号1处

### MRS

状态寄存器传送到通用寄存器

MRS R0， CPSR ; 将CPSR寄存器的内容传送到目的寄存器R0

MRS R0， SPSR ；将SPSR寄存器的内容传送到目的寄存器R0

### MSR

通用寄存器传送到状态寄存器

MSR CPSR\_c, R0 ；R0寄存器的内容传送到状态寄存器CPSR\_c

### CPSR\_x

CPSR有4个8位区域

CPSR\_XX

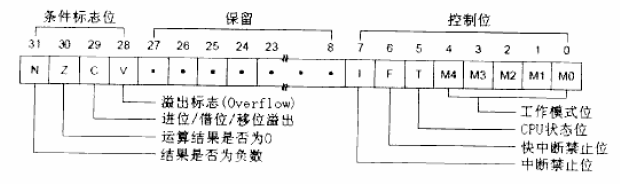
XX可以为：

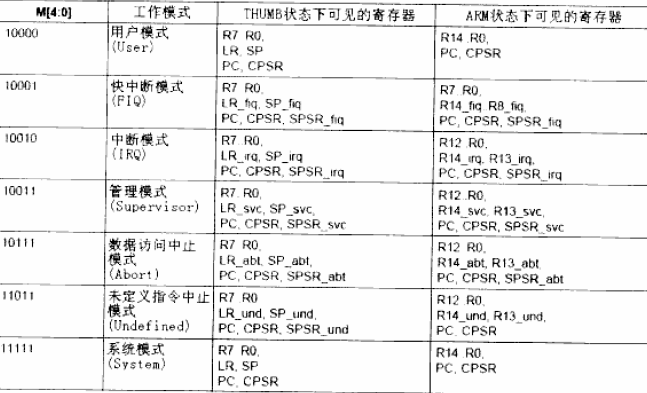
C 控制欲，PSR[7:0]

X 拓展域，PSR[15:8]

S 状态域，PSR[23：16]

F 标志域，PSR[31：24]





0xd3 = 11010011b

MSR CPSR\_c，＃0xd3;进入管理模式

### adrl

adrl r2, mem\_cfg\_val

将mem\_cfg\_val的相对于PC的地址赋值给r2

因为编译的时候指定了链接地址，查看反汇编可以发现mem\_cfg\_val的地址是

30000070 <mem\_cfg\_val>:

30000070: 22011110 andcs r1, r1, #4 ; 0x4

30000074: 00000700 andeq r0, r0, r0, lsl #14

如果从0x30000070这个地址取值肯定是没有值可取，因为此时SDRAM还没有初始化。

但是我们的程序是存放在0x0地址处的，通过adrl可以计算出mem\_cfg\_val存放的地址。

反汇编出来是如下两个指令：

3000004c <memsetup>:

3000004c: e3a01312 mov r1, #1207959552 ; 0x48000000

30000050: e28f2018 add r2, pc, #24 ; 0x18

30000054: e1a00000 nop (mov r0,r0)

30000058: e2813034 add r3, r1, #52 ; 0x34

### bic

BIC―――――位清除指令

指令格式：

BIC{cond}{S} Rd,Rn,operand2

BIC指令将Rn 的值与操作数operand2 的反码按位逻辑”与”，结果存放到目的寄存器Rd 中。指令示例：BIC R0,R0,#0x0F ；将R0最低4位清零，其余位不变。

### Orr

ORR―――――逻辑”或”操作指令

指令格式：ORR{cond}{S} Rd,Rn,operand2 ORR指令将操作数operand2 与Rn 的值按位逻辑”或”，结果存放到目的寄存器Rd 中。指令示例：

ORRS R1,R1,R2 ；R1=R1|R2，并根据运算的结果更新标志位

ORR R0,R0,#0x0F ；R0=R0|0x0F，将R0最低4位置1，其余位不变。

## ARM寄存器

子程序之间通过寄存器r0~r3来传递参数，当参数个数多于4个时，使用堆栈来传递参数。此时r0~r3可记作A1~A4。

在子程序中，使用寄存器r4~r11保存局部变量。因此当进行子程序调用时要注意对这些寄存器的保存和恢复。此时r4~r11可记作V1~V8。

寄存器r12用于保存堆栈指针SP，当子程序返回时使用该寄存器出栈，记作IP。

寄存器r13用作堆栈指针，记作SP。寄存器r14称为链接寄存器，记作LR。该寄存器用于保存子程序的返回地址。

寄存器r15称为程序计数器，记作PC。

## 编译过程

预处理🡺编译🡺汇编🡺链接

预处理识别一些宏等

编译.c🡺.S

汇编.s🡺.o

链接将.o文件组合起来，生成可执行文件

## C

### 宏中的#

我们使用#把宏参数变为一个字符串,用##把两个宏参数贴合在一起.

<http://www.cnblogs.com/hnrainll/archive/2012/08/15/2640558.html>

#include <stdio.h>  
　　#define paster( n ) printf( "token " #n" = %d\n ", token##n )  
　　int main()  
　　{  
　　int token9=10;  
　　paster(9);  
　　return 0;  
　　}  
　　输出为  
　　[leshy@leshy src]$ ./a.out  
　　token 9 = 10

### Attribute

GNU C 的一大特色就是\_\_attribute\_\_ 机制。\_\_attribute\_\_ 可以设置函数属性（Function Attribute ）、变量属性（Variable Attribute ）和类型属性（Type Attribute ）。

\_\_attribute\_\_ 书写特征是：\_\_attribute\_\_ 前后都有两个下划线，并切后面会紧跟一对原括弧，括弧里面是相应的\_\_attribute\_\_ 参数。

\_\_attribute\_\_ 语法格式为：\_\_attribute\_\_ ((attribute-list))

其位置约束为：放于声明的尾部“ ；” 之前。

关键字\_\_attribute\_\_ 也可以对结构体（struct ）或共用体（union ）进行属性设置。大致有六个参数值可以被设定，即：aligned, packed, transparent\_union, unused, deprecated 和 may\_alias 。

在使用\_\_attribute\_\_ 参数时，你也可以在参数的前后都加上“\_\_” （两个下划线），例如，使用\_\_aligned\_\_而不是aligned ，这样，你就可以在相应的头文件里使用它而不用关心头文件里是否有重名的宏定义。

**aligned (alignment)**

该属性设定一个指定大小的对齐格式（以字节 为单位），例如：

struct S {

short b[3];

} \_\_attribute\_\_ ((aligned (8)));

typedef int int32\_t \_\_attribute\_\_ ((aligned (8)));

该声明将强制编译器确保（尽它所能）变量类 型为struct S 或者int32\_t 的变量在分配空间时采用8 字节对齐方式。

### C函数指针

引申：

C语言如何跳转到一个地址处执行函数？

1. void (\*Func)(); //声明一个函数指针，类型为void，变量为空
2. Func = (void(\*)())(0x401); //把0x401强制转换为函数指针，类型为void，参数为空
3. (\*Func)();

或者 Func(); //调用函数

汇编方法：

\_\_ASM

(

"ldr r0,0x401\n"

"blx r0\n"

);

貌似这样子，但是链接的时候遇到了问题。待研究….

在汇编中调用C的函数，参数的传递也是通过ATPCS来实现的

**在汇编中调用C的函数，**

需要在汇编中IMPORT 对应的C函数名，然后将C的代码放在一个独立的C文件中进行编译，剩下的工作由连接器来处理

blx 执行完函数后跳转到当前位置，并切换处理器的为thumb状态。lr寄存器中存储PC值，执行完函数后通过PC返回

## 内存组织形式



### 代码段

代码段用来存放可执行文件的操作指令，可以说是可执行文件在内存中的镜像。代码段需要防止在运行时被修改，只允许读取操作，不允许写入操作。（不可写）

### 数据段

存放已经初始化的全局变量跟静态变量

### BSS段

存放未初始化的全局变量，BSS需要清0

### 堆

动态分配的内存段，大小不固定，可以动态扩张或者缩减。Malloc分配内存，free释放内存

向上增长（从堆申请的地址会越来越大）

### 栈

存储局部变量。先进后出，向下增长。（从栈申请的地址会越来越小）

## 函数返回局部变量

这里面有两个概念：

1.返回指向局部变量的指针

2.返回局部变量

返回局部变量的数据是没问题的，因为此时返回的是局部变量的拷贝。

返回指向局部变量的指针，就要看这个指针指向的内存空间是否被释放。

局部变量空间开辟在栈上，函数退出后，这个空间就被释放。

静态局部变量存储在数据段，这个空间不会被释放

堆的空间不会被释放。

Get memory

http://blog.csdn.net/zhuxiaoyang2000/article/details/8084629

形参不能改变实参的值

程序的局部变量存在于（栈）中  
程序的全局变量存在于（静态存储区）中  
程序动态申请的数据存在于（堆）中

char \* getmemory(char \*p)

{

p = (char \*)malloc(sizeof(char)\*100);

strcpy(p , "hello world");

return p;

}

int main()

{

char \* str = NULL;

getmemory(str);

printf("%s",str);

return 0;

}

1. 形参p不能改变实参str的内容. Str是null，对此处赋值程序会崩溃

void getmemory(char \*\*p)

{

\*p = (char \*)malloc(sizeof(char)\*100);

}

int main()

{

char \* str = NULL;

getmemory(&str);

strcpy(str, "hello world");

printf("%s",str);

return 0;

}

2. 使用二级指针可以改变指针的内容

char \*getmemory()

{

char p[] = "hello world";

return p;

}

int main()

{

char \* str = NULL;

str = getmemory();

printf("%s",str);

return 0;

}

3. p[]是局部变量，里面存储了hello world， 建立在栈上。当程序退出后，栈被摧毁。P指向的内存中可能是一堆无法预测的内容，输出可能是乱码

char \*getmemory()

{

static char p[] = "hello world";

return p;

}

int main()

{

char \* str = NULL;

str = getmemory();

printf("%s",str);

return 0;

}

3.1修改为静态变量，不会被释放

char \*getmemory()

{

char \*p = "hello world";

return p;

}

int main()

{

char \* str = NULL;

str = getmemory();

printf("%s",str);

return 0;

}

4. hello world是常量字符串，全局变量，只读。。

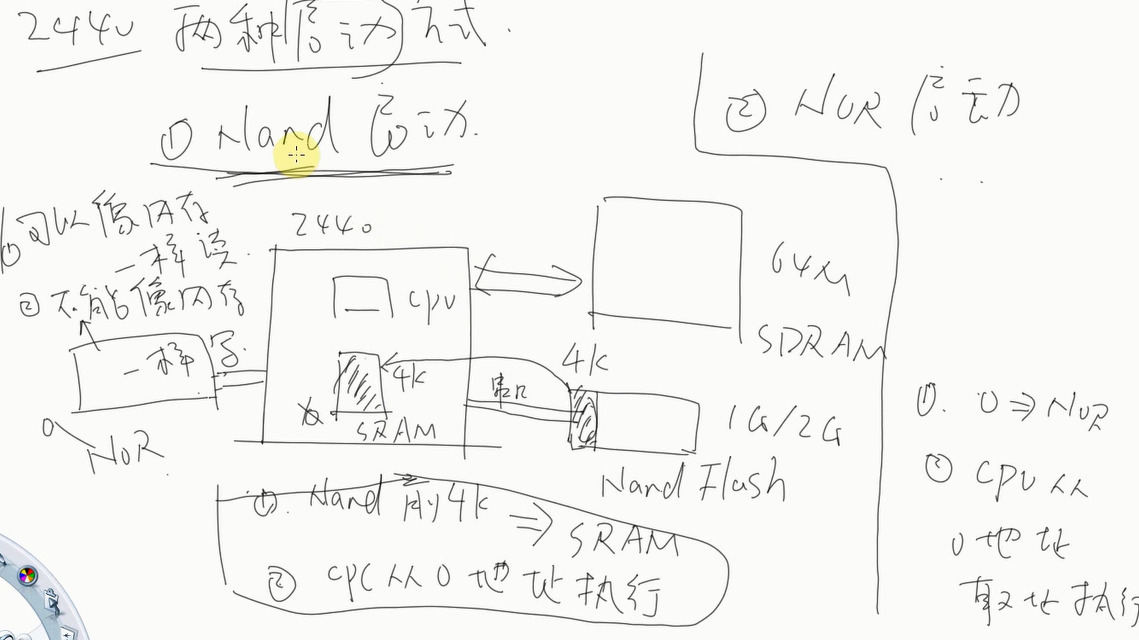
P指向全局变量，所以返回的永远是hello world。

若getmemory后添加 strcpy(str, “abdc”),则运行会报错。

## GCC命令介绍

-c 编译不链接

## 2440两种启动方式



### Nand启动

硬件自动完成下面两个动作

1.nand前4k自动复制到SRAM中

2.cpu从0地址执行

### NOR启动

Nor的特性，可以像内存一样读数据，不能像内存一样写数据。需要先擦除然后写数据

1. 0地址指向nor

2. cpu从0地址取址执行

## C函数调用前后需要做的事情

两个问题：

Main函数被谁调用，执行完返回哪里？

函数执行前需要一个启动文件，启动文件需要做的工作

### 软件方面的初始化

1.设置栈，函数的运行需要栈

2.设置返回地址

3.调用main

4.清理工作



### 硬件方面的初始化

1. 关看门狗
2. 初始化2440时钟
3. 初始化SDRAM

.text

.global \_start

\_start:

ldr r0, =0x53000000 @ WATCHDOG寄存器地址

mov r1, #0x0

str r1, [r0] @ 写入0，禁止WATCHDOG，否则CPU会不断重启。

@如上是硬件初始化

ldr sp, =1024\*4 @ 设置堆栈，注意：不能大于4k, 因为现在可用的内存只有4K ，nand flash中的代码在复位后会移到内部ram中，此ram只有4K

bl main @ 调用C程序中的main函数。

@bl指令会将main的下一条地址也就是halt\_loop保存到lr寄存器中

\_halt\_loop:

b \_halt\_loop

@这里的清理工作就是跳到halt\_loop循环中

## GPIO点灯

### ++i跟i++的区别

而 a = ++i，相当于 i=i+1; a = i;   
而 a = i++，相当于 a = i; i=i+1;

都是i加1，区别是

i++先执行完后面的语句再加1

++i先加1再执行后面的语句

#define GPFCON (\*(volatile unsigned long \*)0x56000050)

#define GPFDAT (\*(volatile unsigned long \*)0x56000054)

#define GPF4\_out (1<<(4\*2))

#define GPF5\_out (1<<(5\*2))

#define GPF6\_out (1<<(6\*2))

int main(void)

{

GPFCON = GPF4\_out|GPF5\_out|GPF6\_out; // 将LED1,2,4对应的GPF4/5/6三个引脚设为输出

GPFDAT = ~(1<<4);

wait(100000);

GPFDAT = ~(1<<5);

wait(100000);

GPFDAT = ~(1<<6);

wait(100000);

return 0

}

以LED1为例：

LED1对应GPF4，查看原理图可知，配置GPF4为输出，当输出为0时，led点亮。

2440RM里面查询可知，GPFDAT配置bit4为0时GPF4即输出电压为0

GPFCON 寄存器，对应地址为0x56000050

GPFDAT寄存器0x56000054

### 反汇编

00000000 <\_start>:

0: e3a00453 mov r0, #1392508928 ; 0x53000000

4: e3a01000 mov r1, #0 ; 0x0

8: e5801000 str r1, [r0]

c: e3a0da01 mov sp, #4096 ; 0x1000

10: eb00000c bl 48 <main>

00000048 <main>:

48: e92d4030 stmdb sp!, {r4, r5, lr}

4c: e3a04b61 mov r4, #99328 ; 0x18400

50: e3a05456 mov r5, #1442840576 ; 0x56000000

54: e3e02010 mvn r2, #16 ; 0x10

58: e2844e2a add r4, r4, #672 ; 0x2a0

5c: e3a03c15 mov r3, #5376 ; 0x1500

60: e1a00004 mov r0, r4

64: e5853050 str r3, [r5, #80]

68: e5852054 str r2, [r5, #84]

6c: ebffffe8 bl 14 <wait>

70: e3e03020 mvn r3, #32 ; 0x20

74: e1a00004 mov r0, r4

78: e5853054 str r3, [r5, #84]

7c: ebffffe4 bl 14 <wait>

80: e3e03040 mvn r3, #64 ; 0x40

84: e1a00004 mov r0, r4

88: e5853054 str r3, [r5, #84]

8c: ebffffe0 bl 14 <wait>

90: e3a00000 mov r0, #0 ; 0x0

94: e8bd8030 ldmia sp!, {r4, r5, pc}

Disassembly of section .comment:

#### 为什么要定义死循环

此处生成的反汇编在汇编文件里面没有定义死循环。

从反汇编文件里面可以看出，main的调用先压栈然后出栈。

Attention

如果main 里面没有死循环，汇编里面也没有死循环。那么当main函数执行完毕，PC不知道下一个动作往哪里走。便会重新指回0，所以一直在重复执行代码。跟死循环的效果看起来一样，都是在不停点灯。

但是有的单片机会进入无序状态，就是跑飞

有的单片机会停止

所以，main函数里面一定要加上while(1),防止程序跑飞。

汇编标号必须以\_打头，不然gcc识别不到。这句待推敲，有时候能识别。

## 存储控制器

2440有8个bank，存储控制器访问外设通过bank实现

2440有ADDR0-ADDR26共计27根地址线，对应的寻址空间是2^27=128M，8个bank实现1G的寻址

SDRAM位宽是4byte，也就是说CPU一次访问可以读取4个byte，字节对齐就跟这个位宽有关。所以对应地址线的接法从addr2开始接到SDRM上面的addr0，详解参考[SDRAM与CPU地址线的接法](#_SDRAM与CPU地址线的接法)

SDRAM一般来说有4个L-BANK，这里的bank是指SDRAM

SDRAM的寻址采用行列寻址

发出bank信号的同时发出行信号，接下来发出列信号，找到相应的存储单元，就可以进行数据的读写了

2.根据什么原理判定与BA0、BA1连接的是LADDR24、LADDR25？

答：在搞清楚问题1的情况下，再来分析问题2。与BA0、BA1相连的两根地址线是寻址地址中的最高两位，故可以得出，连接LADDR24和LADDR25后，CPU总寻址2^26=64Mbyte，我的SDRAM是64Mbyte，刚刚好。

所以说，BA0、BA1接哪根地址线是根据SDRAM的容量大小来定的。

还要补充一点，首先我们将两片SDRAM看做一个整体，那么64Mbyte的SDRAM每个L-Bank大小应该为16Mbyte，怎样推算出来？每个L-Bank上有13根行地址线，9根列地址线，可寻址2^22=4M个，而由问题1我们得知，每单元大小为4Byte，所以每个L-Bank的大小为4M \* 4byte = 16Mbyte。

最后，总结下CPU对SDRAM访问的几个步骤：

1.CPU发出的片选信号nSCS0有效，选中SDRAM芯片。

2.SDRAM中有四个L-Bank，需要两根地址信号来选中其中一个，本文中使用LADDR24和LADDR25作为选择信号。

3.对被选中的芯片进行统一的行/列寻址。

4.找到了存储单元后，被选中的芯片进行统一的数据传输。

<http://blog.csdn.net/lee244868149/article/details/50450232>

## SDRAM

RAM: random access memory，RAM

SRAM: Static RAM.静态RAM，不用刷新，速度可以非常快，像CPU内部的cache，都是静态RAM，缺点是一个内存单元需要的晶体管数量多，因而价格昂贵，容量不大。

DRAM: Dynamic RAM.动态RAM，需要刷新，容量大。 SDRAM ：同步动态RAM，需要刷新，速度较快，容量大。

SDRAM:同步动态RAM，需要刷新，速度较快，容量大

DDR SDRAM:双通道同步动态RAM，需要刷新，速度快，容量大

DRAM是用PN结电容存储0/1的，由于漏电的存在，时间长了两级板电位差会消失也就是1会变成0，所以过一段时间需要根据里面的内容补充电荷，这叫刷新。刷新过程中不能读写。  
SDRAM是DRAM的一种，增加了同步时钟，提高了读写速率。从原理来说现在的DDR3也是SDRAM的一种。不过约定俗成的说法是SDRAM只包含最早的单倍速读写SDRAM  
SARM存储数据的单元是D触发器，这种[双稳态电路](https://www.baidu.com/s?wd=%E5%8F%8C%E7%A8%B3%E6%80%81%E7%94%B5%E8%B7%AF&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YLPyu-nHm3ujN-PhF9mymk0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnHbYP1czrj03nHmzn161njfvr0)的0/1状态都是稳定的所以不需要刷新，只要有电就能一直存下去。  
从存储密度来说，DRAM/SDRAM每个bit需要一个晶体管，而SRAM最少需要4个，高速SRAM需要6个以上，而且由于晶体管之间的互联SRAM复杂得多，占了很大的空间，所以同制程的SRAM容量要小的多。  
从速度来说，DRAM/SDRAM需要刷新和回写，极速比不上SRAM。所以CPU的缓存是SRAM，主内存用SDRAM  
从省电的角度来说，DRAM/SDRAM要刷新比较费电，而省电设计的SRAM只要一个[纽扣电池](https://www.baidu.com/s?wd=%E7%BA%BD%E6%89%A3%E7%94%B5%E6%B1%A0&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YLPyu-nHm3ujN-PhF9mymk0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnHbYP1czrj03nHmzn161njfvr0)就可以保存数年之久（比如以前的游戏卡）

ROM是只读内存（Read-Only Memory）的简称，是一种只能读出事先所存数据的固态[半导体存储器](https://www.baidu.com/s?wd=%E5%8D%8A%E5%AF%BC%E4%BD%93%E5%AD%98%E5%82%A8%E5%99%A8&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YLuHfvmHfzm1Nhmvuhn1uW0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPjnknjnkrjc3)。其特性是一旦储存资料就无法再将之改变或删除。通常用在不需经常变更资料的电子或[电脑系统](https://www.baidu.com/s?wd=%E7%94%B5%E8%84%91%E7%B3%BB%E7%BB%9F&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YLuHfvmHfzm1Nhmvuhn1uW0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPjnknjnkrjc3)中，资料并且不会因为电源关闭而消失  
DRAM（Dynamic Random Access Memory），即动态随机存取存储器最为常见的系统内存。DRAM 只能将数据保持很短的时间。为了保持数据，DRAM使用电容存储，所以 必须隔一段时间刷新（refresh）一次，如果存储单元没有被刷新，存储的信息就会丢失。 （关机就会丢失数据）  
SRAM是英文Static RAM的缩写，它是一种具有静止存取功能的内存，不需要刷新电路即能保存它内部存储的数据

而DRAM（Dynamic Random Access Memory)每隔一段时间，要刷新充电一次，否则内部的数据即会消失，因此SRAM具有较高的性能，但是SRAM也有它的缺点，即它的集成度较低，相同容量的DRAM内存可以设计为较小的体积，但是SRAM却需要很大的体积，且功耗较大。所以在主板上SRAM存储器要占用一部分面积  
SDRAM:Synchronous Dynamic Random Access Memory，同步动态随机存储器，同步是指 Memory工作需要同步时钟，内部的命令的发送与数据的传输都以它为基准；动态是指存储阵列需要不断的刷新来保证数据不丢失；随机是指数据不是线性依次存储，而是自由指定地址进行数据读写

### 位宽

一次传输的位数，即数据位。

字节对齐跟此有关，如果位宽为4byte，CPU一次可以读取4字节

### 链接地址跟加载地址

像bl、b、adr（adr属于伪指令，一般被编译器解释成sub指令）指令属于位置无关指令

不管程序装载在哪个位置上，bl、b、adr指令都能正确的运行，其原因是bl、b、adr指令的地址域是基于PC的相对偏移寻址，相当于[pc+offset]。当ARM启动时，ARM自动取0x00000000位置上的指令，此时PC=0x00000000。

 ldr伪指令中目的寄存器如果是pc,则ldr是与位置相关的指令，u-boot.map文件可以看出，\_start\_armboot=0x33f80044, 即pc=0x33f80004。这样uboot就跳到SDRAM上去运行了，且这条指令刚好处在其运行地址处，所以程序就能正确的运行。

### SDRAM寻址的原理

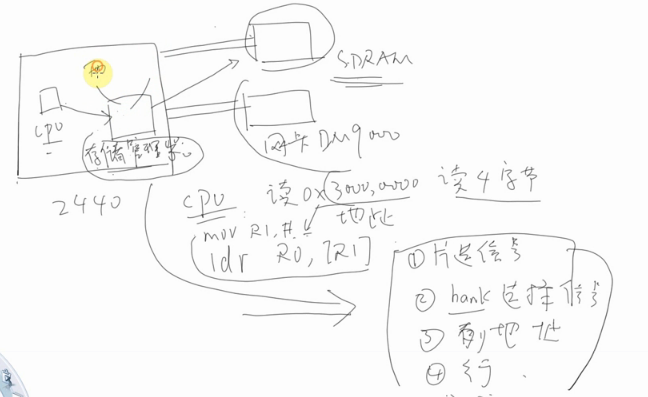
SDRAM内部是一个存储阵列，阵列就如同表格一样，将数据填写进去。先指定一个行，再指定一个列，这样就可以找到需要的单元格。这个就是SDRAM寻址的基本原理

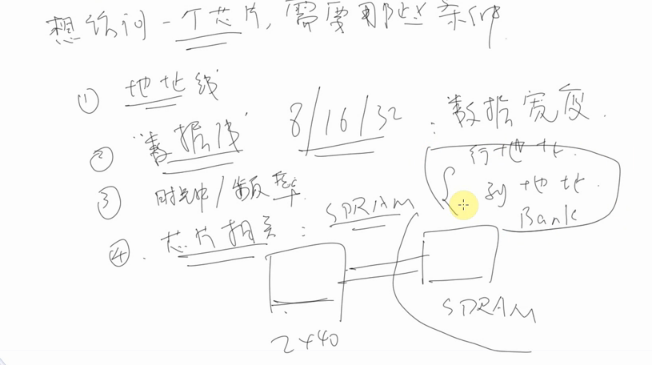
1. CPU发出片选信号，选中SDRAM

2. SDRAM有4个bank，选择需要的bank

3. 对选中的bank进行 行/列（存储单元）寻址

4. 进行数据传输



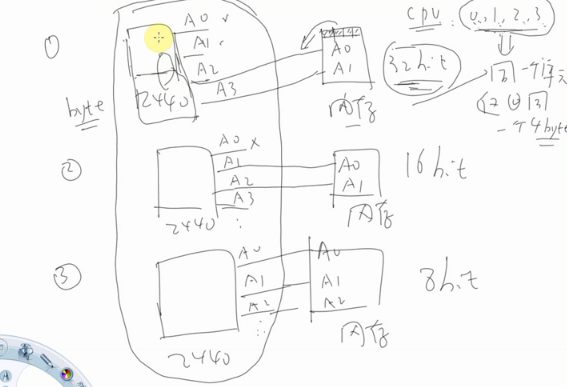


### SDRAM与CPU地址线的接法

因为CPU访问单位为byte，而SDRAM根据配置，可以配置为4、2、1byte。

对SDRAM的访问，一次返回数据就为4、2、1byte。

以SDRAM 位宽为4byte为例，如果CPU A1\A0接上，CPU地址0、1、2、3访问的都是同一个SDRAM前4byte空间，对于上面的四个地址，SDRAM返回同样的4bytes。



## NAND

XIP 芯片内执行

eXecute in place。

Nor flash支持XIP

Nor flash的接口跟RAM一样，可以实现地址随机访问

NAND 不支持XIP的原因：

<http://blog.chinaunix.net/uid-26404697-id-3152290.html>

2440 nand flash只有8个I/O接口，需要串行访问

flash可靠性需要考虑三点：

位反转，坏块，可擦除次数。

位反转：由于flash固有的电气特性，读写数据的过程中，会偶然产生一位或者几位数据错误

### Nand flash组成

最小单位为页，接着是块

Nand一般以页为单位进行读写，以块为单位进行擦除

2440为例：

1页= （512+16）字节

1块= 32页 = 16K + 512B

Nand flash 总共有4096块 = 64MB + 2MB

### Nand flash操作

发出命令==》发出地址序列==》发出操作命令（读，写）

64MB的flash需要26根地址线（2^26）,但是nand flash对外只有8个IO口（这些IO口用来传输地址跟数据，）所以要发送4个地址序列

### Nand flash代码

ldr r0, =0x30000000 @1. 目标地址=0x30000000，这是SDRAM的起始地址mov r1, #4096 @2. 源地址 = 4096，连接的时候，main.c中的代码都存在NAND Flash地址4096开始处

mov r2, #2048 @3. 复制长度= 2048(bytes)，对于本实验的main.c，这是足够了

bl nand\_read @调用C函数nand\_read

nand\_read函数的原型：

void nand\_read(unsigned char \*buf, unsigned long start\_addr, int size)

根据ATCPS规则[C语言跟汇编混合编程](#_C语言跟汇编混合编程)

Buf、start\_addr、size分别对应R0、R1、R2

Makefile里面的linkfile配置如下：

SECTIONS {

firtst 0x00000000 : { head.o init.o nand.o}

second 0x30000000 : AT(4096) { main.o }

}

Main.o存放的位置是nand flash偏移地址4096处，但是运行地址为0x30000000

Nnad上电的时间会将前4K自动复制到SRAM中，所以我们运行的主体程序存放在nand flash 4096以后的地方。

然后将4096以后的数据读取到0x30000000中，再去执行

## 中断

什么是中断？

某事件发生后，硬件会设置某个寄存器，CPU执行完一个指令时，查看这个寄存器，如果发现所关注的事件发生了，则中断当前的程序流程，跳转到一个固定的地址处理这个事件，最后返回继续执行被中断的程序。

## 时钟系统跟定时器

### 晶振：

水晶的特性：通电会产生机械震荡，机械变性震动会产生电。这种效应称为压电效应。

如果在极板间加交变电压，会产生机械变形震动，机械变形震动同时会产生交变电场。当外加交变电压的频率跟晶片的固有频率相等时，机械震动的幅度将急剧增加。这种现象称为压电谐振。特点是频率稳定度很高。石英表就是这个原理制作成的。

### 系统主频

晶振称为外部时钟频率，一般2440的晶振提供的频率能达到12M/20MHz，但是CPU的时钟频率需要400M/600M的样子。这时候就需要将外部的频率倍频。倍频的方式就是使用锁相环PLL

2440里面FCLK用于CPU

### 设备频率

一般对设备来说，主频显得太高。因此2440会提供两种时钟HCLK跟PCLK供设备使用。HCLK for AHB(advanced high performance bus)总线与高性能模块之间的链接, PCLK for APB(advanced peripherals bus)总线与低带宽的周边设备之间的链接。

## UART

### UART原理：

UART: universal asynchronous receiver transmitter

通用异步收发器

发送数据：CPU将并行的数据写入UART，UART按照一定的格式在一根电线上串行发出。

接受数据：UART检测另一根电线上面的信号，将串行接收到的数据保存在缓冲区，CPU即可读取UART获得数据

UART之间以全双工的方式进行数据传输，最精简的连线方式只有3根线。TxD 用于发送数据，RxD用于接收数据，Gnd 用于给双方提供参考电平

### 逻辑电平

TTL/CMOS逻辑电平表示数据

0-5V、0-3.3V、0-2.5V、0-1.8V

高电平表示1，低电平表示0

RS-232电平：

3-12V 表示0， -3 — -12表示1

所以，使用TTL\CMOS或者RS-232逻辑电平下面的串行波形是相反的

### 波特率

UART的Tx、Rx数据线都是以位为最小单位进行发送。

帧由若干位组成

发送数据前，UART需要约定好数据的传输速率，即波特率

每位所需的时间的倒数称为波特率

### 数据传输流程：

1. 平时数据线处于空闲状态（1状态）

2. 发送数据时，UART改变TxD的状态（变为0），并维持1位的时间

检测方检测到开始位后，等待1.5位的时间，然后开始一位一位传送数据

3. UART一帧中可以有5、6、7、或8位的数据，发送方一位一位改变数据线的状态， 将他们一位一位传送出去，首先发送最低位（LSB）

4. 如果使用校验功能，UART在发送完数据位后，还要发送1个校验位。（奇校验，偶校验）数据位连同校验位中，1的数目等于奇数或者偶。

5. 最后发送停止位（1），数据线恢复到1状态。停止位的长度有三种，1、1.5、2位

## I2C

Inter-integrated Circuit。 PHILIP开发的串行总线，用于连接微控制器及其外部设备

提供两条总线线路：串行数据线（SDA），串行时钟线（SCL）

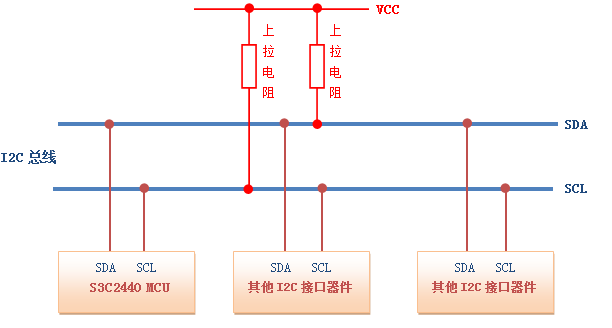
<http://blog.csdn.net/bob_fly1984/article/details/22690381>

<http://blog.csdn.net/g_salamander/article/details/8016698>

对I2C撰写驱动，涉及到：

I2C控制器驱动的撰写，I2C外挂设备驱动的撰写

### I2C总线物理拓扑结构



I2C总线在物理连接上非常简单，分别由SDA(串行数据线)和SCL(串行时钟线)及上拉电阻组成。通信原理是通过对SCL和SDA线高低电平时序的控制，来产生I2C总线协议所需要的信号进行数据的传递。在总线空闲状态时，这两根线一般被上面所接的上拉电阻拉高，保持着高电平。

### I2C总线特征

    I2C总线上的每一个设备都可以作为主设备或者从设备，而且每一个设备都会对应一个唯一的地址(可以从I2C器件的数据手册得知)，主从设备之间就通过这个地址来确定与哪个器件进行通信，在通常的应用中，我们把CPU带I2C总线接口的模块作为主设备，把挂接在总线上的其他设备都作为从设备。  
    **I2C总线上可挂接的设备数量受总线的最大电容400pF 限制，如果所挂接的是相同型号的器件，则还受器件地址位的限制。**  
   **I2C总线数据传输速率在标准模式下可达100kbit/s，快速模式下可达400kbit/s，高速模式下可达3.4Mbit/s。一般通过I2C总线接口可编程时钟来实现传输速率的调整，同时也跟所接的上拉电阻的阻值有关。**  
    I2C总线上的主设备与从设备之间**以字节(8位)为单位进行双向的数据传输**。

### 总线信号时序分析

    1. 总线空闲状态  
    SDA和SCL两条信号线都处于高电平，即总线上所有的器件都释放总线，两条信号线各自的上拉电阻把电平拉高；  
    2. 启动信号START  
    时钟信号SCL保持高电平，数据信号SDA的电平被拉低(即负跳变)。启动信号必须是跳变信号，而且在建立该信号前必修保证总线处于空闲状态；  
    3. 停止信号STOP  
    时钟信号SCL保持高电平，数据线被释放，使得SDA返回高电平(即正跳变)，停止信号也必须是跳变信号。  
    4. 数据传送  
    SCL线呈现高电平期间，SDA线上的电平必须保持稳定，低电平表示0(此时的线电压为地电压)，高电平表示1(此时的电压由元器件的VDD决定)。只有在SCL线为低电平期间，SDA上的电平允许变化。  
    5. 应答信号ACK  
    I2C总线的数据都是以字节(8位)的方式传送的，发送器件每发送一个字节之后，在时钟的第9个脉冲期间释放数据总线，由接收器发送一个ACK(把数据总线的电平拉低)来表示数据成功接收。  
    6. 无应答信号NACK  
    在时钟的第9个脉冲期间发送器释放数据总线，接收器不拉低数据总线表示一个NACK，NACK有两种用途:  
    a. 一般表示接收器未成功接收数据字节；  
    b. 当接收器是主控器时，它收到最后一个字节后，应发送一个NACK信号，以通知被控发送器结束数据发送，并释放总线，以便主控接收器发送一个停止信号STOP。

#### 寻址约定

    地址的分配方法有两种:  
    1. 含CPU的智能器件，地址由软件初始化时定义，但不能与其它的器件有冲突；  
    2. 不含CPU的非智能器件，由厂家在器件内部固化，不可改变。  
    高7位为地址码，其分为两部分:  
    1. 高4位属于固定地址不可改变，由厂家固化的统一地址；

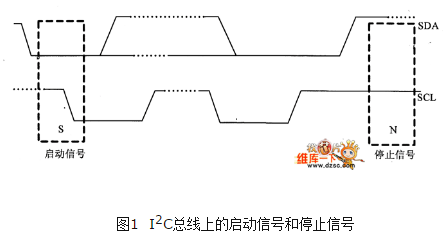
    2. 低三位为引脚设定地址，可以由外部引脚来设定(并非所有器件都可以设定)；

主控器向被控器发送的信息种类有：启动信号、停止信号、7位地址码、读／写控制位、10位地址码、数据字节、重启动信号、应答信号、时钟脉冲。  
　　被控器向主控器发送的信息种类有：应答信号、数据字节、时钟低电平。

#### 启动信号

   
　　在时钟线SCL保持高电平期间，数据线SDA上的电平被拉低（即负跳变），定义为I2C总线总线的启动信号，它标志着一次数据传输的开始。

启动信号是一种电平跳变时序信号，而不是一个电平信号。启动信号是由主控器主动建立的，在建立该信号之前I2C总线必须处于空闲状态，如图1所示。

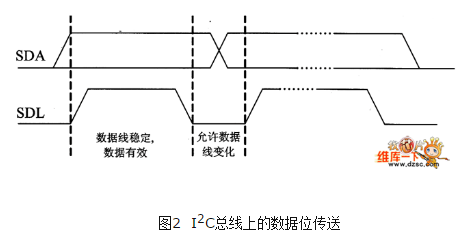


#### 停止信号

   
　　在时钟线SCL保持高电平期间，数据线SDA被释放，使得SDA返回高电平（即正跳变），称为I2C总线的停止信号，它标志着一次数据传输的终止。  
   
　　停止信号也是一种电平跳变时序信号，而不是一个电平信号，停止信号也是由主控器主动建立的，建立该信号之后，I2C总线将返回空闲状态。

#### 数据位传送

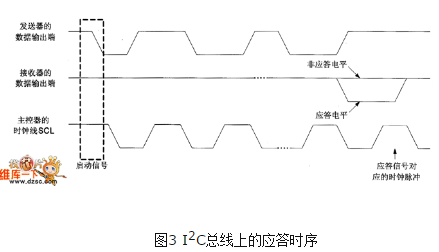
在I2C总线上传送的每一位数据都有一个时钟脉冲相对应（或同步控制），即在SCL串行时钟的配合下，在SDA上逐位地串行传送每一位数据。  
　　进行数据传送时，在SCL呈现高电平期间，SDA上的电平必须保持稳定，低电平为数据0，高电平为数据1。  
　　只有在SCL为低电平期间，才允许SDA上的电平改变状态。逻辑0的电平为低电压，而逻辑1的电平取决于器件本身的正[电源](http://www.dzsc.com/product/searchfile/2937.html)电压VDD（当使用独立电源时），如图2所示。



#### 应答信号

I2C总线上的所有数据都是以8位字节传送的，发送器每发送一个字节，就在时钟脉冲9期间释放数据线，由接收器反馈一个应答信号。  
　　应答信号为低电平时，规定为有效应答位（ACK简称应答位），表示接收器已经成功地接收了该字节；应答信号为高电平时，规定为非应答位（NACK），一般表示接收器接收该字节没有成功。  
　　对于反馈有效应答位ACK的要求是，接收器在第9个时钟脉冲之前的低电平期间将SDA线拉低，并且确保在该时钟的高电平期间为稳定的低电平。

　　如果接收器是主控器，则在它收到最后一个字节后，发送一个NACK信号，以通知被控发送器结束数据发送，并释放SDA线，以便主控接收器发送一个停止信号P，如图3所示。



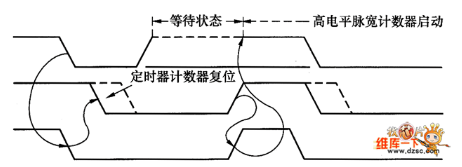
#### 插入等待时间

    
　　如果被控器需要延迟下一个数据字节开始传送的时间，则可以通过把时钟线SCL电平拉低并且保持，使主控器进入等待状态。  
　　一旦被控器释放时钟线，数据传输就得以继续下去，这样就使得被控器得到足够时间转移已经收到的数据字节，或者准备好即将发送的数据字节。  
　　带有CPU的被控器在对收到的地址字节做出应答之后，需要一定的时间去执行中断服务子程序，来分析或比较地址码，其间就把SCL线钳位在低电平上，直到处理妥当后才释放SCL线，进而使主控器继续后续数据字节的发送，如图4所示。

重启动信号  
   
　　在主控器控制总线期间完成了一次数据通信（发送或接收）之后，如果想继续占用总线再进行一次数据通信（发送或接收），而又不释放总线，就需要利用重启动Sr信号时序。  
　　重启动信号Sr既作为前一次数据传输的结束，又作为后一次数据传输的开始。利用重启动信号的优点是，在前后两次通信之间主控器不需要释放总线，这样就不会丢失总线的控制权，即不让其他主器件节点抢占总线。

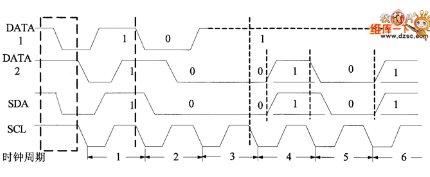
#### 时钟同步

   
　　如果在某一I2C总线系统中存在两个主器件节点，分别记为主器件1和主器件2，其时钟输出端分别为CLK1和CL【0，它们都有控制总线的能力。  
   
　　假设在某一期间两者相继向SCL线发出了波形不同的时钟脉冲序列CLK1和CLK2（时钟脉冲的高、低电平宽度都是依靠各自内部专用[计数器](http://www.dzsc.com/product/searchfile/3037.html)定时产生的），在总线控制权还没有裁定之前这种现象是可能出现的。  
   
　　鉴于I2C总线的“线与”特性，使得时钟线SCL上得到的时钟信号波形，既不像主器件1所期望的CLK1，也不像主器件2所期望的CLK2，而是两者进行逻辑与的结果。  
   
　　CLKI和CLK2的合成波形作为共同的同步时钟信号，一旦总线控制权裁定给某一主器件，则总线时钟信号将会只由该主器件产生，如图5所示。



#### 总线冲突和总线仲裁

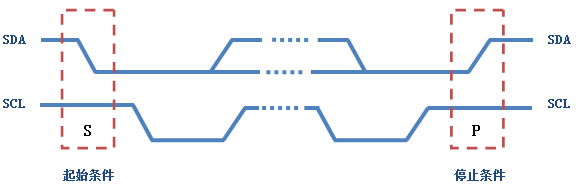
    
　　假如在某I2C总线系统中存在两个主器件节点，分别记为主器件1和主器件2，其数据输出端分别为DATA1和DATA2，它们都有控制总线的能力，这就存在着发生总线冲突（即写冲突）的可能性。  
    
　　假设在某一瞬间两者相继向总线发出了启动信号，鉴于：I2C总线的“线与”特性，使得在数据线SDA上得到的信号波形是DATA1和DATA2两者相与的结果，该结果略微超前送出低电平的主器件1，其DATA1的下降沿被当做SDA的下降沿。  
    
　　在总线被启动后，主器件1企图发送数据“101……”，主器件2企图发送数据“100101……”。  
    
　　两个主器件在每次发出一个数据位的同时都要对自己输出端的信号电平进行抽检，只要抽检的结果与它们自己预期的电平相符，就会继续占用总线，总线控制权也就得不到裁定结果。  
    
　　主器件1的第3位期望发送“1”，也就是在第3个时钟周期内送出高电平。  
    
　　在该时钟周期的高电平期间，主器件1进行例行抽检时，结果检测到一个不相匹配的电平“0”，这时主器件1只好决定放弃总线控制杈；因此，主器件2就成了总线的惟一主宰者，总线控制权也就最终得出了裁定结果，从而实现了总线仲裁的功能。  
    
　　从以上总线仲裁的完成过程可以得出：仲裁过程主器件1和主器件2都不会丢失数据；各个主器件没有优先级别之分，总线控制权是随机裁定的，即使是抢先发送启动信号的主器件1最终也并没有得到控制杈。  
    
　　系统实际上遵循的是“低电平优先”的仲裁原则，将总线判给在数据线上先发送低电平的主器件，而其他发送高电平的主器件将失去总线控制权，如图6所示。



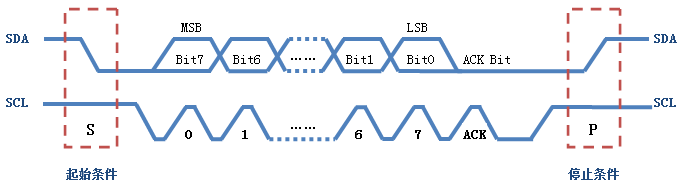
⑩总线封锁状态。  
    
　　在特殊情况下，如果需要禁止所有发生在I2C总线上的通信活动，封锁或关闭总线是一种可行途径，只要挂接于该总线上的任意一个器件将时钟线SCL锁定在低电平上即可。

### **I2C总线协议**

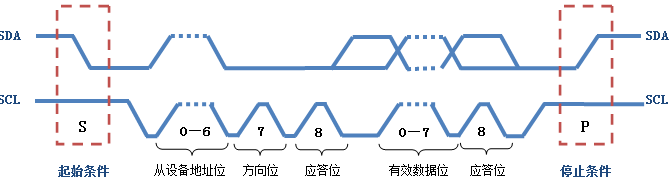
    I2C协议规定，总线上数据的传输必须以一个起始信号作为开始条件，以一个结束信号作为传输的停止条件。起始和结束信号总是由主设备产生。总线在空闲状态时，SCL和SDA都保持着高电平，当SCL为高电平而SDA由高到低的跳变，表示产生一个起始条件；当SCL为高而SDA由低到高的跳变，表示产生一个停止条件。在起始条件产生后，总线处于忙状态，由本次数据传输的主从设备独占，其他I2C器件无法访问总线；而在停止条件产生后，本次数据传输的主从设备将释放总线，总线再次处于空闲状态。如图所示：



在了解起始条件和停止条件后，我们再来看看在这个过程中数据的传输是如何进行的。前面我们已经提到过，数据传输以字节为单位。主设备在SCL线上产生每个时钟脉冲的过程中将在SDA线上传输一个数据位，当一个字节按数据位从高位到低位的顺序传输完后，紧接着从设备将拉低SDA线，回传给主设备一个应答位，此时才认为一个字节真正的被传输完成。当然，并不是所有的字节传输都必须有一个应答位，比如：当从设备不能再接收主设备发送的数据时，从设备将回传一个否定应答位。数据传输的过程如图所示：

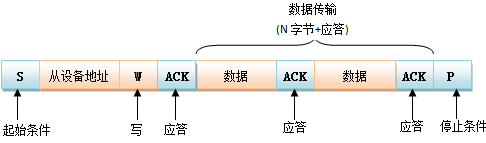


在前面我们还提到过，I2C总线上的每一个设备都对应一个唯一的地址，主从设备之间的数据传输是建立在地址的基础上，也就是说，主设备在传输有效数据之前要先指定从设备的地址，地址指定的过程和上面数据传输的过程一样，只不过大多数从设备的地址是7位的，然后协议规定再给地址添加一个最低位用来表示接下来数据传输的方向，0表示主设备向从设备写数据，1表示主设备向从设备读数据。如图所示：

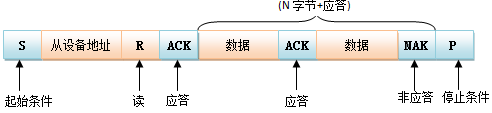


### I2C总线操作

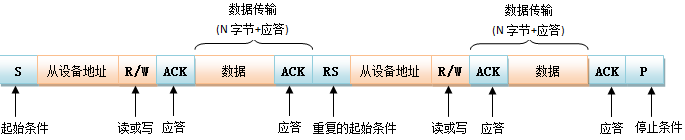
    对I2C总线的操作实际就是主从设备之间的读写操作。大致可分为以下三种操作情况：  
    第一，主设备往从设备中写数据。数据传输格式如下：



第二，主设备从从设备中读数据。数据传输格式如下：

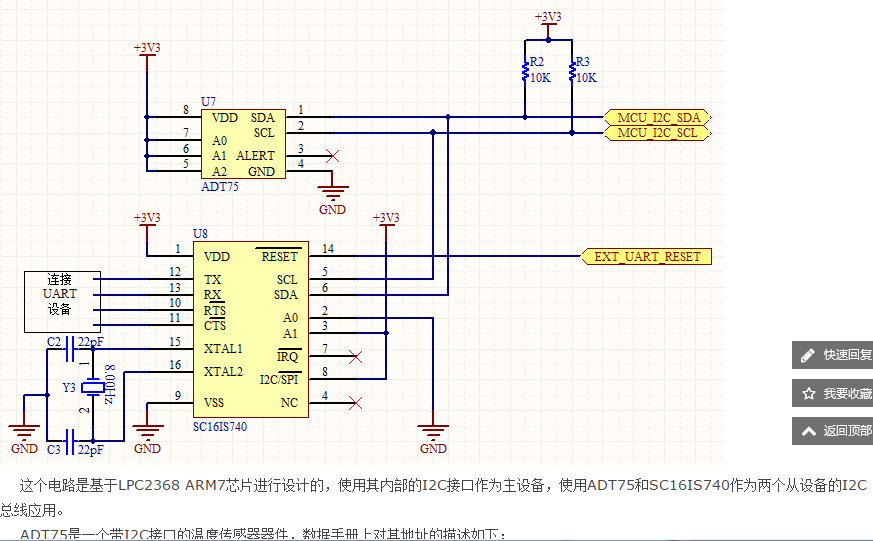


第三，主设备往从设备中写数据，然后重启起始条件，紧接着从从设备中读取数据；或者是主设备从从设备中读数据，然后重启起始条件，紧接着主设备往从设备中写数据。数据传输格式如下：



第三种操作在单个主设备系统中，重复的开启起始条件机制要比用STOP终止传输后又再次开启总线更有效率。

### I2C总线硬件接口电路示例



## 板子环境搭建

### 编译Uboot

tar xjf u-boot-1.1.6.tar.bz2

cd u-boot-1.1.6/

patch -p1 < ../u-boot-1.1.6\_jz2440.patch

make 100ask24x0\_config

Make

创建Uboot压缩包

tar cjf u-boot-1.1.6\_jz2440.tar.bz2 u-boot-1.1.6 //.tar.bz2格式的压缩包容量小

解压.gz文件

tar zxvf

### 编译内核

$ tar xjf linux-2.6.22.6.tar.bz2

$ cd linux-2.6.22.6

$ patch -p1 < ../linux-2.6.22.6\_jz2440.patch

$ mv ../4.3寸LCD\_mach-smdk2440.c arch/arm/mach-s3c2440/mach-smdk2440.c // 替换4.3寸源码

$ cp config\_ok .config

$ make uImage

生成的Uimage路径如下：

/work/system/linux-2.6.22.6/arch/arm/boot

### Tftp download image

设置板子IP：

set ipaddr 192.168.1.1

set serverip 192.168.1.100

save

mtd //查看分区

tftp 30000000 stdio.bin

nand erase bootloader

nand write 30000000 bootloader

烧写内核：

tftp 30000000 uImage

nand erase kernel

nand write.jffs2 30000000 kernel

烧写文件系统：

tftp 30000000 fs\_qtopia.yaffs2

nand erase root

nand write.yaffs 30000000 0x00260000 $(filesize)

## Uboot

### Makefile分析

http://blog.chinaunix.net/uid-11319766-id-3050283.html

Makefile分为两个阶段：

1. make 100ask24x\_config

2. make all

如果只是输入make的情况下，会寻找makefile里面的第一个目标并执行他，也就是all。所以

make = make all

ifeq ($(OBJTREE)/include/config.mk,$(wildcard $(OBJTREE)/include/config.mk))

这段makefile的目的是查找config.mk是否存在，

当执行了make 100ask24x\_config后，才会生成这个文件。否则提示

@echo "System not configured - see README" >&2

#### ($(OBJTREE)

Makefile可以指定目标文件输出到一个文件夹中，使用指令

make 100ask24x0\_config O=output

make O=output

指定输出到output文件夹，这个文件夹makefile会自动创建。

#### Makefile第一阶段

100ask24x0\_config : unconfig

@$(MKCONFIG) $(@:\_config=) arm arm920t 100ask24x0 NULL s3c24x0

可以看到目标依赖是 unconfig，目标动作是调用MKCONFIG 脚本

$(MKCONFIG)

MKCONFIG := $(SRCTREE)/mkconfig

export MKCONFIG

MKCONFIG即是uboot源文件夹下面的mkconfig脚本

这个脚本的目的是进行一些配置

$(@:\_config=) 实质是 $(VAR:XX=YY)将VAR中XX部分替换为YY。

$@表示目标，也就是100ask24x0\_config，括号内不需要$,拓展开来的脚本就是:

mkconfig 100ask24x0 arm arm920t 100ask24x0 NULL s3c24x0

脚本的最终目的就是做如下几件事情：

1. 开发板名称BOARD\_NAME = $1

2. 创建平台，开发板相关的头文件的超链接

Ln –s asm-$2 asm

Ln –s arch-$6 asm-$2/arch

Ln –s proc-armv asm-$2/proc

3. 创建config.mk，内容为：

ARCH=$2

CPU= $3

..

4. 创建开发板相关的头文件config.h

#include <configs/$1.h>

#### 第二阶段

执行第一阶段后，

ifeq ($(OBJTREE)/include/config.mk,$(wildcard $(OBJTREE)/include/config.mk))为真

# load ARCH, BOARD, and CPU configuration

include $(OBJTREE)/include/config.mk

将需要的编译器，link file进行配置

选择需要编译的OBJS、LIBS

注意：

OBJ的顺序是有要求的，第一个编译的obj需要放在第一位。也就是start.o

ALL = $(obj)u-boot.srec $(obj)u-boot.bin $(obj)System.map $(U\_BOOT\_NAND)

all: $(ALL)

$(obj)u-boot.hex: $(obj)u-boot

$(OBJCOPY) ${OBJCFLAGS} -O ihex $< $@

$(obj)u-boot.srec: $(obj)u-boot

$(OBJCOPY) ${OBJCFLAGS} -O srec $< $@

$(obj)u-boot.bin: $(obj)u-boot

$(OBJCOPY) ${OBJCFLAGS} -O binary $< $@

$(obj)u-boot.img: $(obj)u-boot.bin

./tools/mkimage -A $(ARCH) -T firmware -C none \

-a $(TEXT\_BASE) -e 0 \

-n $(shell sed -n -e 's/.\*U\_BOOT\_VERSION//p' $(VERSION\_FILE) | \

sed -e 's/"[ ]\*$$/ for $(BOARD) board"/') \

-d $< $@

$(obj)u-boot.dis: $(obj)u-boot

$(OBJDUMP) -d $< > $@

$(obj)u-boot: depend version $(SUBDIRS) $(OBJS) $(LIBS) $(LDSCRIPT)

UNDEF\_SYM=`$(OBJDUMP) -x $(LIBS) |sed -n -e 's/.\*\(\_\_u\_boot\_cmd\_.\*\)/-u\1/p'|sort|uniq`;\

cd $(LNDIR) && $(LD) $(LDFLAGS) $$UNDEF\_SYM $(\_\_OBJS) \

--start-group $(\_\_LIBS) --end-group $(PLATFORM\_LIBS) \

-Map u-boot.map -o u-boot

all的依赖就是$(ALL),目标uboot的依赖，第一个为depend

depend dep:

for dir in $(SUBDIRS) ; do $(MAKE) -C $$dir \_depend ; done

这里有两个目标，depend或者dep，这两个目标用到一个就可以。

For循环里面做的事情是

$(MAKE) -C $$dir \_depend

执行$(SUBDIRS)每一个文件夹里面的makefile中的\_depend目标。

2440的uboot的$(SUBDIRS)里面没有\_depend目标，所以进入文件夹后又出来。打印log如下：

make -C tools

make[1]: Entering directory `/work/system/u-boot-1.1.6/examples'

make[1]: Nothing to be done for `\_depend'.

make[1]: Leaving directory `/work/system/u-boot-1.1.6/examples'

---------

make -C examples

make[1]: Entering directory `/work/system/u-boot-1.1.6/post'

make[1]: Nothing to be done for `\_depend'.

make[1]: Leaving directory `/work/system/u-boot-1.1.6/post'

---------

make -C post

make[1]: Entering directory `/work/system/u-boot-1.1.6/post/cpu'

make[1]: Nothing to be done for `\_depend'.

make[1]: Leaving directory `/work/system/u-boot-1.1.6/post/cpu'

### Uboot启动流程

<http://blog.chinaunix.net/uid-21602837-id-4081060.html>

#### 第一阶段

（一）**编译地址**： 32位的处理器，它的每一条指令是4个字节，以4个字节存储顺序，进行顺序执行，CPU是顺序执行的，只要没发生什么跳转，它会**顺序进行执行行**， 编译器会对每一条指令分配一个编译地址，这是编译器分配的，在编译过程中分配的地址，我们称之为编译地址。  
（二）**运行地址**：是指程序指令真正运行的地址，是由**用户指定**的，用户将运行地址**烧录到哪里**，**哪里就是运行的地址**。

个可执行的image必须有一个入口点，并且只能有一个**全局入口点**，所以要通知编译器这个入口在哪里。由此我们可以找到程序的入口点是在**/board/lpc2210/u-boot.lds**中指定的，其中ENTRY(\_start)说明程序从\_start开始运行，而他指向的是cpu/arm7tdmi/start.o文件。

.globl \_start

\_start: b reset

ldr pc, \_undefined\_instruction

ldr pc, \_software\_interrupt

ldr pc, \_prefetch\_abort

ldr pc, \_data\_abort

ldr pc, \_not\_used

ldr pc, \_irq

ldr pc, \_fiq

\_undefined\_instruction: .word undefined\_instruction

\_software\_interrupt: .word software\_interrupt

\_prefetch\_abort: .word prefetch\_abort

\_data\_abort: .word data\_abort

\_not\_used: .word not\_used

\_irq: .word irq

\_fiq: .word fiq

##### 首先配置中断向量表。

Linkfile指定了程序的入口ENTRY(\_start)

OUTPUT\_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")

/\*OUTPUT\_FORMAT("elf32-arm", "elf32-arm", "elf32-arm")\*/

OUTPUT\_ARCH(arm)

ENTRY(\_start)

SECTIONS

{

. = 0x00000000;

. = ALIGN(4);

.text :

{

cpu/arm920t/start.o (.text)

board/100ask24x0/boot\_init.o (.text)

\*(.text)

}

. = ALIGN(4);

.rodata : { \*(.rodata) }

. = ALIGN(4);

.data : { \*(.data) }

. = ALIGN(4);

.got : { \*(.got) }

. = .;

\_\_u\_boot\_cmd\_start = .;

.u\_boot\_cmd : { \*(.u\_boot\_cmd) }

\_\_u\_boot\_cmd\_end = .;

. = ALIGN(4);

\_\_bss\_start = .;

.bss : { \*(.bss) }

\_end = .;

}

上电后，程序会进入reset中断，执行reset

.balignl 16,0xdeadbeef

表示接下来的代码需要16字节对齐，不足的部分以0xdeadbeef补上

##### \_TEXT\_BASE:

.word TEXT\_BASE

这个地址的来源：

源文件夹下面的makefile中有

$(obj)u-boot: depend version $(SUBDIRS) $(OBJS) $(LIBS) $(LDSCRIPT)

UNDEF\_SYM=`$(OBJDUMP) -x $(LIBS) |sed -n -e 's/.\*\(\_\_u\_boot\_cmd\_.\*\)/-u\1/p'|sort|uniq`;\

cd $(LNDIR) && $(LD) $(LDFLAGS) $$UNDEF\_SYM $(\_\_OBJS) \

--start-group $(\_\_LIBS) --end-group $(PLATFORM\_LIBS) \

-Map u-boot.map -o u-boot

其中$(LDFLAGS)定义于源文件夹下面的config.mk

LDFLAGS += -Bstatic -T $(LDSCRIPT) -Ttext $(TEXT\_BASE) $(PLATFORM\_LDFLAGS)

ifdef BOARD

sinclude $(TOPDIR)/board/$(BOARDDIR)/config.mk # include board specific rules

endif

此处的BOARD就是脚本文件mkconfig中的参数，也就是100ask24x0

所以TEXT\_BASE的来源就是

/board/100ask24x0/config.mk，定义如下：

TEXT\_BASE = 0x33F80000

##### Start.s中相关宏的来源

首先，定义了

#include <config.h>

Config.h位于include文件夹中。

这个头文件是脚本文件mkconfig自动生成的，打开可以看到包含了另一个头文件

/\* Automatically generated - do not edit \*/

#include <configs/100ask24x0.h>

100ask24x0.h中包含了板子相关的配置信息

##### 进入SVC模式

\* set the cpu to SVC32 mode

\*/

mrs r0,cpsr

bic r0,r0,#0x1f

orr r0,r0,#0xd3

msr cpsr,r0

##### 关闭看门狗

/\* turn off the watchdog \*/

#if defined(CONFIG\_S3C2400)

# define pWTCON 0x15300000

# define INTMSK 0x14400008 /\* Interupt-Controller base addresses \*/

# define CLKDIVN 0x14800014 /\* clock divisor register \*/

#elif defined(CONFIG\_S3C2410)

# define pWTCON 0x53000000

# define INTMOD 0X4A000004

# define INTMSK 0x4A000008 /\* Interupt-Controller base addresses \*/

# define INTSUBMSK 0x4A00001C

# define CLKDIVN 0x4C000014 /\* clock divisor register \*/

#endif

#if defined(CONFIG\_S3C2400) || defined(CONFIG\_S3C2410)

ldr r0, =pWTCON

mov r1, #0x0

str r1, [r0]

##### 屏蔽中断

\* mask all IRQs by setting all bits in the INTMR - default

\*/

mov r1, #0xffffffff

ldr r0, =INTMSK

str r1, [r0]

# if defined(CONFIG\_S3C2410)

ldr r1, =0x3ff

ldr r0, =INTSUBMSK

str r1, [r0]

# endif

##### cpu\_init\_crit

lowlevel\_init

##### 配置堆栈

/\* Set up the stack \*/

stack\_setup:

ldr r0, \_TEXT\_BASE /\* upper 128 KiB: relocated uboot \*/

sub r0, r0, #CFG\_MALLOC\_LEN /\* malloc area \*/

sub r0, r0, #CFG\_GBL\_DATA\_SIZE /\* bdinfo \*/

##### 配置时钟

#ifndef CONFIG\_SKIP\_LOWLEVEL\_INIT

bl clock\_init

#endif

##### 拷贝代码到\_TEXT\_BASE

relocate: /\* relocate U-Boot to RAM \*/

adr r0, \_start /\* r0 <- current position of code \*/

ldr r1, \_TEXT\_BASE /\* test if we run from flash or RAM \*/

cmp r0, r1 /\* don't reloc during debug \*/

beq clear\_bss

ldr r2, \_armboot\_start

ldr r3, \_bss\_start //bss\_start所在的位置就是代码结束的位置，可以从link file中看到

sub r2, r3, r2 /\* r2 <- size of armboot \*/

#if 1

bl CopyCode2Ram /\* r0: source, r1: dest, r2: size \*/

##### BSS清零

clear\_bss:

ldr r0, \_bss\_start /\* find start of bss segment \*/

ldr r1, \_bss\_end /\* stop here \*/

mov r2, #0x00000000 /\* clear \*/

clbss\_l:str r2, [r0] /\* clear loop... \*/

add r0, r0, #4

cmp r0, r1

ble clbss\_l

##### 内存使用情况

根据start.s可以计算出来内存的排布：

SDRAM的地址为：

BANK6，128M

0x30000000-0x34000000

\_TEXT\_BASE = 0x33f80000

CFG\_MALLOC\_LEN = CFG\_ENV\_SIZE + 128\*1024 = 0x40000 = 256KB

0x20000

CFG\_GBL\_DATA\_SIZE = 128 = 0x80

CONFIG\_STACKSIZE\_IRQ = 4\*1024 = 0X1000

CONFIG\_STACKSIZE\_FIQ = 4\*1024 = 0X1000

12 for abort stack

SP= 0x33f80000 – 0x40000 – 0x80 – 0x1000 – 0x1000 – 0xd = 0x33f3df74

##### 跳转到第二阶段

ldr pc, \_start\_armboot