**一. 简单原理介绍**

那个教程里面主要讲了一些简单的配置, 原理没有仔细介绍, 其实当时我也不太懂, 当时看重的是实际效果, 原理嘛, 在我们从SB到NB的过程会慢慢理解的,  
现在稍微理解了一点, 在写这篇教程之前简单介绍一下之前的存储器的地址配置的问题吧, 也和等会移植的内容有关啦, 耐心看吧:  
TQ2440开发板算起来应该是有3个片外存储器吧, 一个是 nand flash, 一个是 nor flash, (这两个应该算是ROM了), 还有一个SDRAM (就是RAM啦, SDRAM是个什么东西, 还没学数电…不知道…)

* 首先，2440的程序代码是可以存放在SDRAM里面运行的.从汇编上讲, 就是程序指针(PC指针, 程序计数器, 各种称呼,  
  指向一下条要执行的程序代码 的一个寄存器)可以指到SDRAM的内存单元.
* 然后再说nor flash, 2440的程序也可以在nor flash里执行, 原因我不清楚. 从nor flash启动的时候,  
  PC指向的0地址就是nor flash的首地址, 就是说nor flash 的首地址映射到内存的0地址。对比一下SDRAM, SDRAM在2440是映射到0x3000 0000 地址
* 然后是nand flash, nand flash 也是映射到了 0地址, 为样就和nor flash有冲突了, 所以就有了从nor 启动和从nand启动的区别了,

然后我又想说, 其实nand本身是不能运行程序的(这里的意思是说, 因为nand flash本身为了容量大和本身的接口, 是不能向CPU不断的快速的输送指令的, 我的理解就是这样, 大家可以去参考nor 和 nand 的区别的资料), 既然我说nand不能运行程序, 那个地址映射有什么意义? 是这样的, 2440从nand启动的时候, 它内部的nand flash控制器会把前4k的代码复制到内部的缓存去执行, 而这些缓存就是对应地址的0单元开始. (其实这个直接去参考2440 datasheet 上的地址映射会比看我在这里瞎说的强…)

刚才说了从nand启动的时候是把前4k的代码复制到内部的缓存去执行的, nand flash的容量远远远远大于4K呀, 不充分利用就太浪费了…如果你的程序超过4K了怎么办? 很简单, 你在前4K的程序里, 利用什么for啊while之类的把剩下的代码复制到SDRAM里面去, 复制完了以后后再用一条汇编指令 ldr (用来跳转到绝对地址的, 类似C语言的goto, 但有点区别) 跳到SDRAM里面继续执行, (因为前4K的代码在内部缓存运行的时候对应的地址是从0-4K的, SDRAM是的地址是从0x30000000开始的, 你不跳, 就等着系统崩溃吧).  
刚才说的这些工作你可以自己写代码, 也可以移植个boot loader来帮你做, 操作系统就是从boot loader开始的…这部分我没学, 不说了…

那么ldr指令是怎么跳转的? 在2440init.s里有这两句,

ldr pc, =copy\_proc\_beg

copy\_proc\_beg

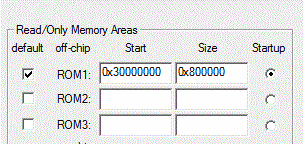
* 1
* 2

类似C语言的

goto copy\_proc\_beg

copy\_proc\_beg:

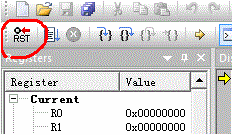
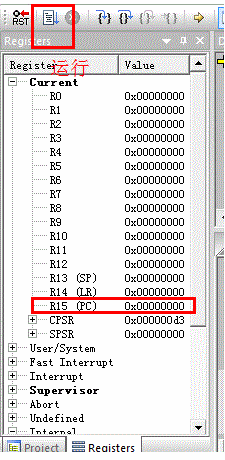
* 1
* 2

注意, 我说的是类似, 从汇编层面是讲, 是有很大的不同的, 我比较懒, 就不详细解释了  
上面的ldr 的作用就是:  
你的工程最后生成可执行的代码里面, copy\_proc\_beg 对应的地址(绝对地址, 就是说应该放在存储器的哪个位置), 会被复制到PC指针, 所以我们在nand启动的工程配置里, rom应该是这样的:  
srart(开始): 0x30000000 size(大小, 我这里是随便设置的): 0x800000  
  
抱歉, 我之前的教程里的生成下载到nand的bin文件的那部分配置讲错了, 当时并不理解, 不过, 小于4K的程序还是可以用的了^\_^

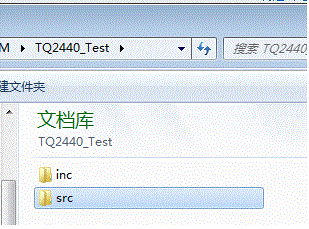
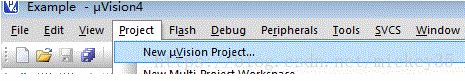
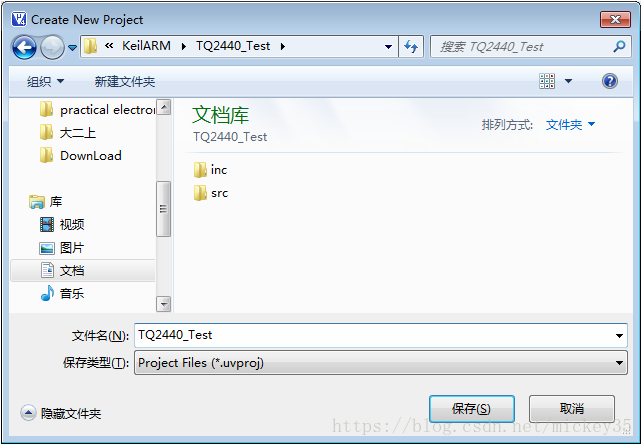
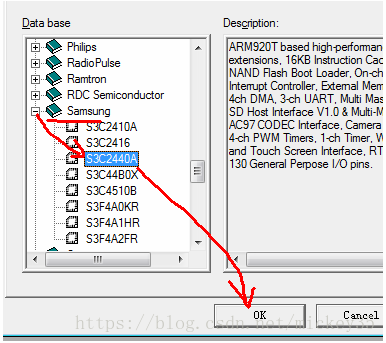
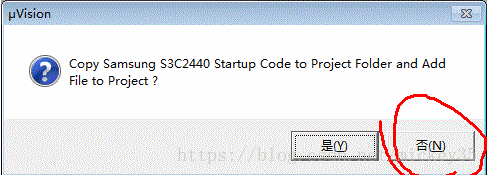
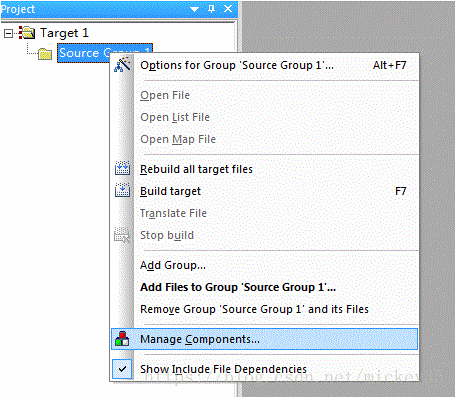
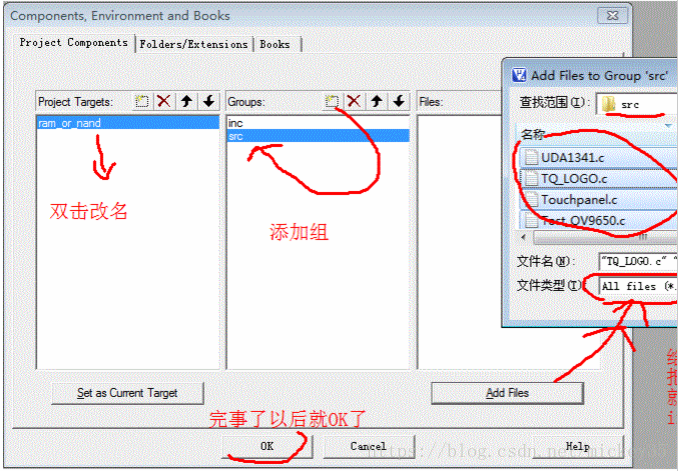
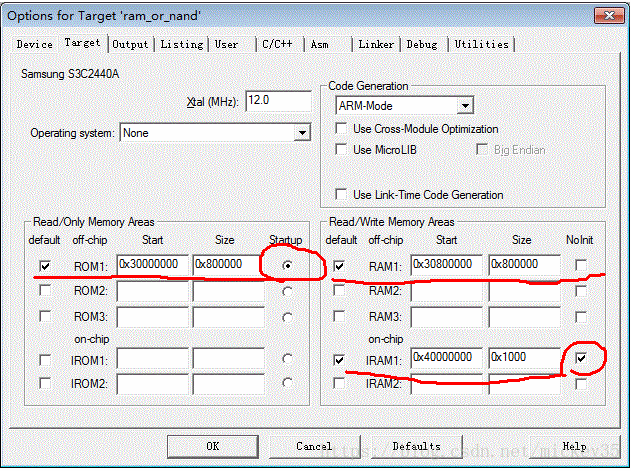
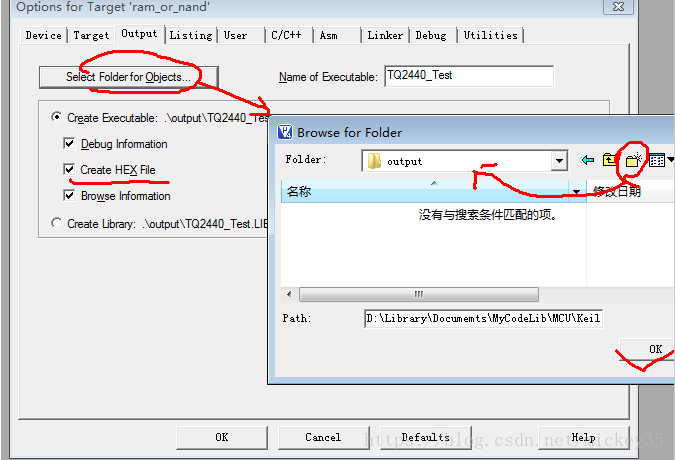
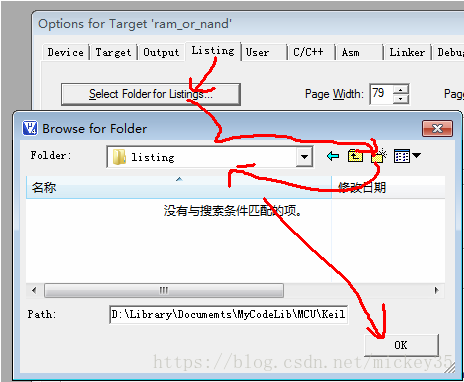
补充: 其实教程1那里的只有把ROM配置成从0x0开始才是可以正常运行的, 原因是用了KEIL给的启动代码, 有一句是这样的  
**IMPORT \_\_main ; 相当于extern void main()**  
**LDR R0, =\_\_main ; 跳到main函数的绝对地址去执行**  
如果rom 设置 0x3000 0000 开始的话, 教程1的代码里又没有进行搬移的工作, 跳到SDRAM里去, 没有代码, 程序就不能正常运行了

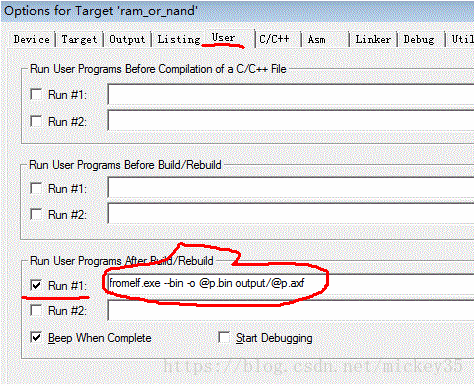
这样, 编译器生成以后, 标号copy\_proc\_beg对应的地址是0x3\*\*\*\*\*\*\*, 也就是在SDRAM里面, 然后你在运行前4k的程序的时候PC指针的内容是0x00000\*\*\*, 完成之前说的代码从nand flash到SDRAM的搬移工作后, 用一条 ldr pc, =copy\_proc\_beg 指令, 把PC指针的内容改成0x3\*\*\*\*\*\*\*, 就跳转到SDRAM里面运行了

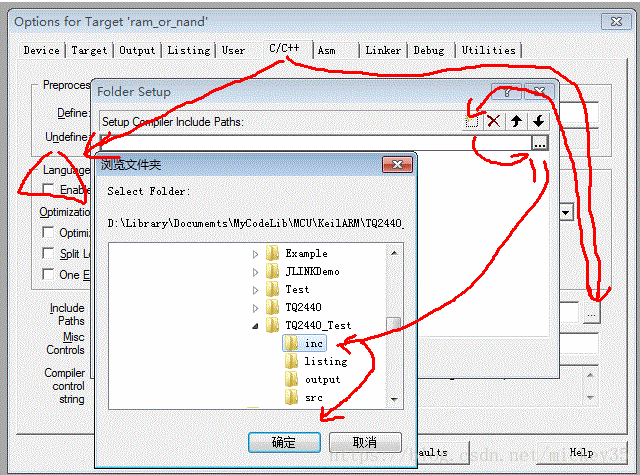
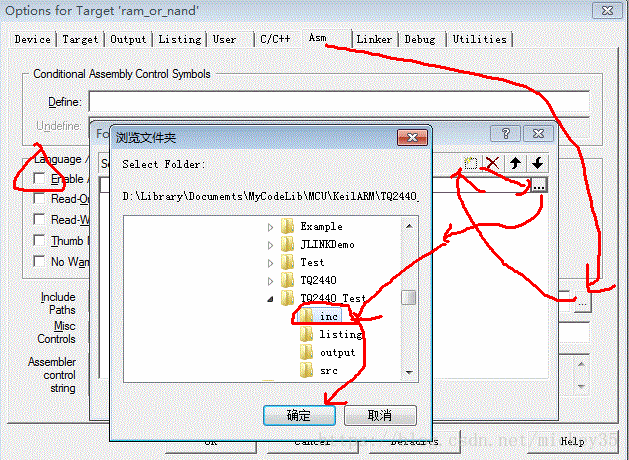
基本的原理讲到这里, 洗澡去, 等会再进入正题…顺便想一下有没有需要补充的…

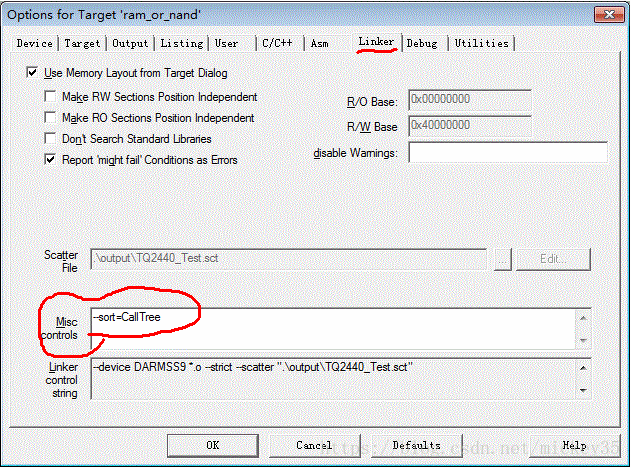
补充:  
用JLINK在SDRAM调试的时候, ROM配置也是  
srart(开始): 0x30000000 size(大小, 我这里是随便设置的): 0x800000  
然后你按下调试键的时候, 代码由J-LINK帮你复制到SDRAM了, 然后再把PC指到0X30000000开始执行, 这J-LINK的这些工作是由一个初始化文件(教程1里配置的时候从keil的安装目录那边复制过来的文件)指导完成的. 如果你在调试的时候按了这个键  
  
就把单片机给复位了, 然后PC指针指向的地址就是0, 如果你再点运行,  
除非你的nand flash 或 nor flash里面有程序, 要不然是不会正常运行的, 即使有程序, 也不能正常调试, 你要把PC指针改回0x30000000才可以让2440继续跑你要调试的程序, (双击PC又边的数字 0x00000000就可以改他的值了)  
  
关于nor flash的设置, 没什么好讲的了, 从0开始执行, 嗯, 就这样了

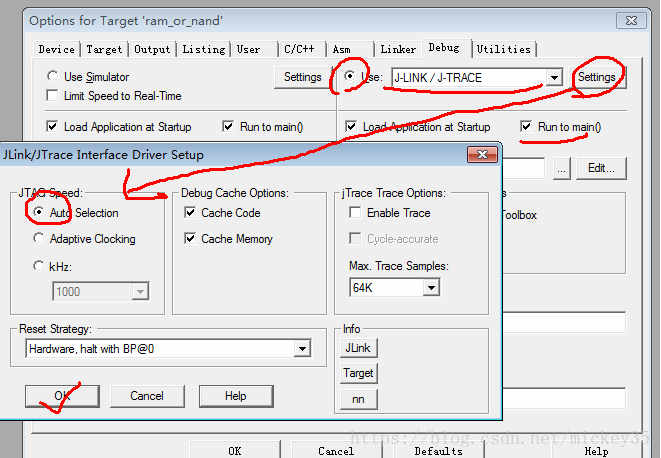
**二. 复制代码, 建立并配置工程**

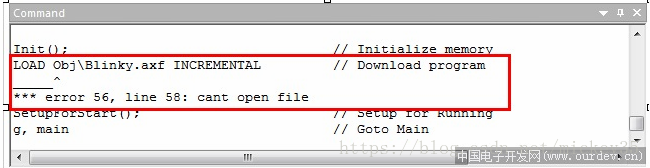
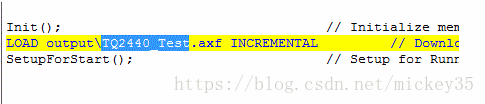
把TQ2440测试程序的源码的src和inc复制到一个文件夹下  
  
打开keil软件, project->new uVision Project…  
  
工程名就定为TQ2440\_Test了  
  
选择S3C2440…  
  
选否, 这回就不用MDK给的启动代码了, 我们是要移植程序, 直接用tq2440\_test里的2440init.s嘛,  
  
习惯改下名, 然后添加些文件…  
  
然后看图吧, 没必要详细讲, 自己摸索就会了  
  
接下来开始设置工程了:  
  
这个好像不必要, 习惯而已  
  


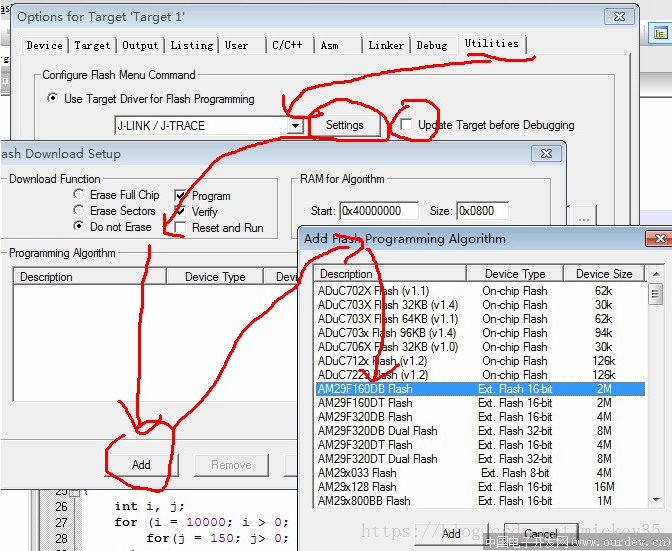
fromelf.exe –bin -o @p.bin output/@p.axf  
关于这个命令的, 我的第一个教程的帖子里传有相关的资料  


后面的配置比较重要了, 头文件的, 还有uncheck Enable Arm/Thumb interworking  
  


这个配置也很重要, 移植成功的一个重要配置, 我不太会解释, 因为MDK的链接器感觉本来就不太好用, 这里的主要目的是让nand.c,2440init.s和2440slib.s生成的目标被安排到最后生成程序的前4K代码区  
关于那个sort的, 可以自己去参考MDK的linker的说明文档  
我在上面的原理中说到，要在前面的4K的程序里面完成代码的搬移工作，这个配置的主要目的就是让nand.c（进行代码搬移工作的那些函数所在文件）生成的目标被安排到前4K的内容，你也可以尝试自己写分散加载文件，我试过的比较好的办法就是这样了，如果你有更好的配置方法，也希望你能分享你的方法哈^\_^  


调试的配置, 如果你不打算用J-LINK调试, 也可以不配置  


还要复制一个初始化文件, 文件在Keil的安装目录里找, 我这里是D:\Program Files\Keil\ARM\Boards\Samsung\S3C2440\RTX\_Blinky\Ext\_RAM.ini  
直接复制到你的工程目录下, 然后还要做一点修改, 如果不修改, 就调试不了, 会出现这样的情况  
  
工程输出的.axf文件(我也不清楚是用来干啥的)在工程目录下的Output文件夹里, 所以可以用编辑工具(记事本就行了)打开来这样修改:  


最后设置utilities这里添加了烧写算法的, 其实在烧NOR Flash那里我才需要这个, 这个算法是不能擦除的, 但是这里我找不到合适的算法, 就用这个相近的了, 要擦除的话就用JFlash, 你也可以自己写一个算法, 参考MDK的文档, 搜flash programming algorithm之类的字眼, 你会找到的  
  
到这里, 设置完啦~接下来~要改一点代码。。。

**三. 修改部分代码**

在2440Init.S文件的

AREA Init,CODE,READONLY

* 1

把**Init**改成**RESET** (MDK默认的符号… = =)  
然后在这一行后，加入如下行：

PRESERVE8 ;8 字对齐

* 1

这样做，主要是为了排除如下错误  
 .\output\2440test.axf: Error: L6238E: 2440init.o(Init) contains invalid call from ‘~PRES8’ function to ‘REQ8’  
function Main.  
 .\output\2440test.axf: Finished: 0 information, 0 warning and 1 error messages.  
 Target not created

然后是修改一些名字:

在2440init.s中做如下替换：  
(注意: 汇编的格式我并不太懂, 所以原来的代码怎样缩进(空多少格), 你就怎样做, 要不然…出了错也不知道神马原因…)  
将如下代码段(75行附近)

IMPORT |Image$$RO$$Base| ; Base of ROM code

IMPORT |Image$$RO$$Limit| ; End of ROM code (=start of ROM data)

IMPORT |Image$$RW$$Base| ; Base of RAM to initialise

IMPORT |Image$$ZI$$Base| ; Base and limit of area

IMPORT |Image$$ZI$$Limit| ; to zero initialise

* 1
* 2
* 3
* 4
* 5

替换为：

IMPORT |Image$$ER\_ROM1$$RO$$Base| ; Base of ROM code

IMPORT |Image$$ER\_ROM1$$RO$$Limit| ; End of ROM code (=start of ROM data)

IMPORT |Image$$RW\_RAM1$$RW$$Base| ; Base of RAM to initialise

IMPORT |Image$$RW\_RAM1$$ZI$$Base| ; Base and limit of area

IMPORT |Image$$RW\_RAM1$$ZI$$Limit| ; to zero initialise

* 1
* 2
* 3
* 4
* 5

将450 行附近

BaseOfROM DCD |Image$$RO$$Base|

TopOfROM DCD |Image$$RO$$Limit|

BaseOfBSS DCD |Image$$RW$$Base|

BaseOfZero DCD |Image$$ZI$$Base|

EndOfBSS DCD |Image$$ZI$$Limit|

* 1
* 2
* 3
* 4
* 5

替换为：

BaseOfROM DCD |Image$$ER\_ROM1$$RO$$Base|

TopOfROM DCD |Image$$ER\_ROM1$$RO$$Limit|

BaseOfBSS DCD |Image$$RW\_RAM1$$RW$$Base|

BaseOfZero DCD |Image$$RW\_RAM1$$ZI$$Base|

EndOfBSS DCD |Image$$RW\_RAM1$$ZI$$Limit|

* 1
* 2
* 3
* 4
* 5

将Main.c中的如下段（18-23行）

extern char Image$$RO$$Limit[];

extern char Image$$RO$$Base[];

extern char Image$$RW$$Limit[];

extern char Image$$RW$$Base[];

extern char Image$$ZI$$Limit[];

extern char Image$$ZI$$Base[];

* 1
* 2
* 3
* 4
* 5
* 6

替换为：

extern char Image$$ER\_ROM1$$RO$$Limit[];

extern char Image$$ER\_ROM1$$RO$$Base[];

extern char Image$$RW\_RAM1$$RW$$Limit[];

extern char Image$$RW\_RAM1$$RW$$Base[];

extern char Image$$RW\_RAM1$$ZI$$Limit[];

extern char Image$$RW\_RAM1$$ZI$$Base[];

* 1
* 2
* 3
* 4
* 5
* 6

将如下代码（67 行前后）

extern char Image$$RW$$Limit[];

U32 \*pMagicNum=(U32 \*)Image$$RW$$Limit;

* 1
* 2

替换为：

extern char Image$$RW\_RAM1$$RW$$Limit[];

U32 \*pMagicNum=(U32 \*)Image$$RW\_RAM1$$RW$$Limit;

* 1
* 2

将2440Lib.c中的如下代码（19-20 行）

extern char Image$$RW$$Limit[];

void \*mallocPt=Image$$RW$$Limit;

* 1
* 2

替换为：

extern char Image$$RW\_RAM1$$RW$$Limit[];

void \*mallocPt=Image$$RW\_RAM1$$RW$$Limit;

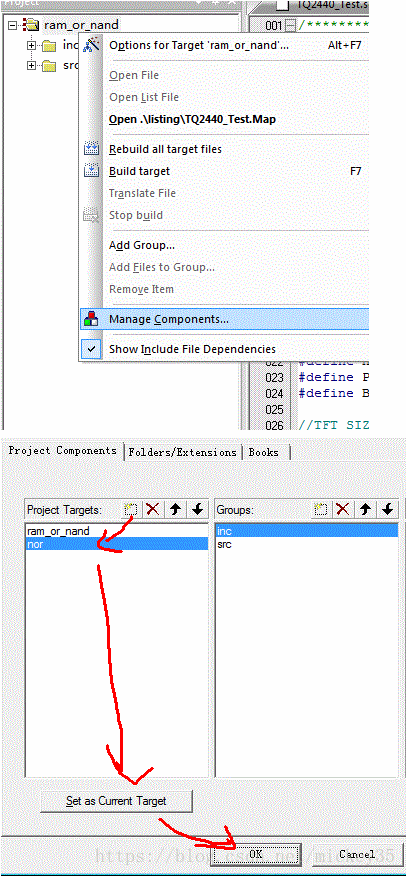
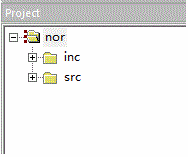
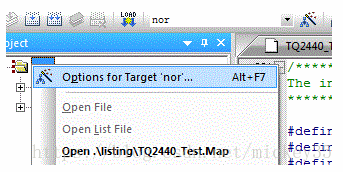
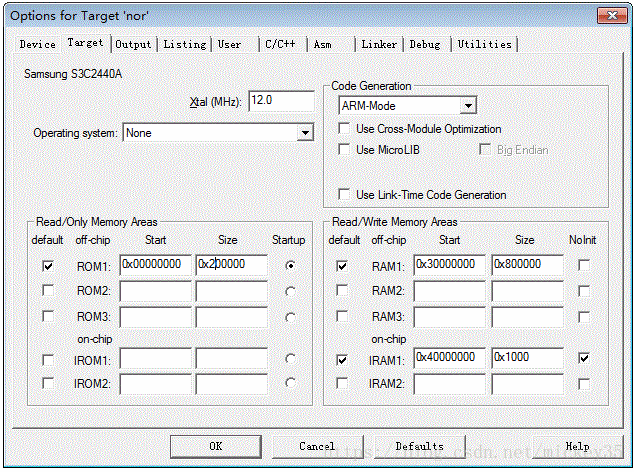
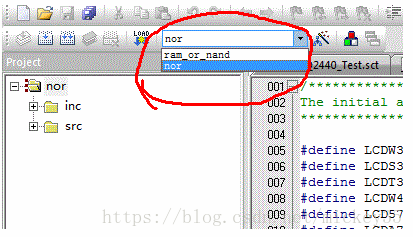
* 1
* 2

LCD\_TFT.h第14行: 我的开发板配的是4.3寸的,

#define LCD\_Type LCDW43 //设定屏的类型

* 1

编译, 生成的bin文件就可以下载到nand flash去了, 下载的方式和TQ2440裸奔教程上说的一样

如果你想用JLINK烧到nor flash  
可以这样  
  
看, 变了  
  
然后你可以很方便的重新配置  
  
因为utilities那部分刚才已经配置了, 所以现在改一下地址就行了  
  
编译, 然后就可以用J-LINK烧到NOR flash里去了 ^\_^  
(烧写之前要先擦除的哟…哥已经不想再截图了…参考教程1的内容去)  
两种配置可以很方便的切换的^\_^  


**四. 测试结果**

关于NOR烧写的, 可能程序比较大, 然后算法不合适, 我这里刚刚测试没成功过, 总是会有TIME OUT之类的信息…  
RAM调试的基本通过, 但是按键中断测试出了点问题, 这个我也不太清楚, 以后再研究吧~  
下载到NAND的测试通过了, 没发现问题, 所以我的目的已经达到了~  
