**0.arm-linux-gcc[常用参数讲解gcc编译器使用方法](https://www.cnblogs.com/Rainingday/p/7435338.html)**

我们需要编译出运行在ARM平台上的代码，所**使用**的交叉编译器为 **arm-linux-gcc**。下面将**arm-linux-gcc**编译工具的一些常用命令参数介绍给大家。  
在此之前首先介绍下编译器的工作过程，在**使用**GCC编译程序时，编译过程分为四个阶段：  
1. 预处理（Pre-Processing）  
2. 编译（Compiling）  
3. 汇编（Assembling）  
4. 链接（Linking）  
Linux程序员可以根据自己的需要让 GCC在编译的任何阶段结束，以便检查或**使用**编译器在该阶段的输出信息，或者对最后生成的二进制文件进行控制，以便通过加入不同数量和种类的调试代码来为 今后的调试做好准备。和其它常用的编译器一样，GCC也提供了灵活而强大的代码优化功能，利用它可以生成执行效率更高的代码。

以文件example.c为例说明它的用法

代码位置：TQ210N/Test/arm-linux-gcc/  
0. **arm-linux-gcc** -o example example.c  
   不加-c、-S、-E参数，编译器将执行预处理、编译、汇编、连接操作直接生成可执行代码。  
    -o参数用于指定输出的文件，输出文件名为example,如果不指定输出文件，则默认输出a.out

myroot@myroot:~/TQ210N/Test/arm-linux-gcc$ arm-linux-gcc -o example example.c

myroot@myroot:~/TQ210N/Test/arm-linux-gcc$ ls

example example.c

myroot@myroot:~/TQ210N/Test/arm-linux-gcc$ arm-linux-gcc example.c

myroot@myroot:~/TQ210N/Test/arm-linux-gcc$ ls

a.out example.c

1. **arm-linux-gcc** -c -o example.o example.c  
   -c参数将对源程序example.c进行预处理、编译、汇编操作，生成example.o文件  
   去掉指定输出选项"-o example.o"自动输出为example.o,所以说在这里-o加不加都可以

myroot@myroot:~/TQ210N/Test/arm-linux-gcc$ arm-linux-gcc -c -o example.o example.c

myroot@myroot:~/TQ210N/Test/arm-linux-gcc$ ls

example.c example.o

myroot@myroot:~/TQ210N/Test/arm-linux-gcc$ arm-linux-gcc -c example.c

myroot@myroot:~/TQ210N/Test/arm-linux-gcc$ ls

example.c example.o

1. **arm-linux-gcc** -S -o example.s example.c  
      -S参数将对源程序example.c进行预处理、编译，生成example.s文件  
      -o选项同上

myroot@myroot:~/TQ210N/Test/arm-linux-gcc$ arm-linux-gcc -S -o example.s example.c

myroot@myroot:~/TQ210N/Test/arm-linux-gcc$ ls

example.c example.s

myroot@myroot:~/TQ210N/Test/arm-linux-gcc$ arm-linux-gcc -S example.c

myroot@myroot:~/TQ210N/Test/arm-linux-gcc$ ls

example.c example.s

3.**arm-linux-gcc** -E -o example.i example.c  
   -E参数将对源程序example.c进行预处理，生成example.i文件（不同版本不一样，有的将预处理后的内容打印到屏幕上）就是将#include，#define等进行文件插入及宏扩展等操作。  
myroot@myroot:~/TQ210N/Test/arm-linux-gcc$ arm-linux-gcc -E -o example.i example.c

myroot@myroot:~/TQ210N/Test/arm-linux-gcc$ ls

example.c example.i  
4.**arm-linux-gcc** -v -o example example.c  
加上-v参数，显示编译时的详细信息，编译器的版本，编译过程等。

5.**arm-linux-gcc** -g -o example example.c  
-g选项，加入GDB能够**使用**的调试信息,**使用**GDB调试时比较方便。

6.**arm-linux-gcc** -Wall -o example example.c  
-Wall选项打开了所有需要注意的警告信息，像在声明之前就**使用**的函数，声明后却没有**使用**的变量等。

7.**arm-linux-gcc** -Ox -o example example.c  
-Ox**使用**优化选项，X的值为空、0、1、2、3  
0为不优化，优化的目的是减少代码空间和提高执行效率等，但相应的编译过程时间将较长并占用较大的内存空间。

8.**arm-linux-gcc**   -I /home/include -o example example.c  
-Idirname: 将dirname所指出的目录加入到程序头文件目录列表中。如果在预设系统及当前目录中没有找到需要的文件，就到指定的dirname目录中去寻找。

9.**arm-linux-gcc**   -L /home/lib -o example example.c

-Ldirname：将dirname所指出的目录加入到库文件的目录列表中。在默认状态下，连接程序ld在系统的预设路径中(如/usr/lib)寻找所需要的库文件，这个选项告诉连接程序，首先到-L指定的目录中去寻找，然后再到系统预设路径中寻找。

10.**arm-linux-gcc** –static -o libexample.a example.c

静态链接库文件

11.arm-linux-gcc编译器：

1）预处理：将C和汇编源程序进行预处理，生成“.i”目标文件

-E参数将对源程序example.c进行预处理，生成example.i文件（不同版本不一样，有的将预处理后的内容打印到屏幕上）就是将#include，#define等进行文件插入及宏扩展等操作。

myroot@myroot:~/TQ210N/Test/arm-linux-gcc$ arm-linux-gcc -E -o example.i example.c

myroot@myroot:~/TQ210N/Test/arm-linux-gcc$ ls

example.c example.i

2）编译：将“.i”目标文件编译生成“.s”汇编文件

myroot@myroot:~/TQ210N/Test/arm-linux-gcc$ arm-linux-gcc -S -o example.s example.i

myroot@myroot:~/TQ210N/Test/arm-linux-gcc$ ls

example.c example.i example.s

3) 汇编：将编译得到的“.s”汇编文件按照给定的指令集转换成符合一定格式的机器码

myroot@myroot:~/TQ210N/Test/arm-linux-gcc$ arm-linux-gcc -c -o example.o example.s

myroot@myroot:~/TQ210N/Test/arm-linux-gcc$ ls

example.c example.i example.out example.s

4) 链接：将汇编生成的目标文件和系统库的目标文件、库文件组装起来，最终生成可以在特定处理器平台运行的可执行文件

myroot@myroot:~/TQ210N/Test/arm-linux-gcc$ arm-linux-gcc -o example.o example.out

myroot@myroot:~/TQ210N/Test/arm-linux-gcc$ ls

example.c example.i example.o example.out example.s

其他）：

①-v 显示出arm-linux-gcc编译器的配置信息，同时还会在显示编译过程的详细信息

②-g 在最后生成的可执行程序中加上调试信息

③-Wall 打开所有需要注意的警告信息

④-Ox 优化等级，（-O0 不进行优化，-O1一级优化，-O2 二级优化，-O3 最高级优化），优化等级越高，代码效率越高，但编译时间越长，推荐优化等级为2。

**以GCC在linux系统下测试：**

//gcc预处理：将C和汇编源程序进行预处理，生成“.i”目标文件

myroot@myroot:~/TQ210N/Test/arm-linux-gcc$ gcc -E -o example.i example.c

myroot@myroot:~/TQ210N/Test/arm-linux-gcc$ ls

example.c example.i

//gcc 编译：将“.i”目标文件编译生成“.s”汇编文件

myroot@myroot:~/TQ210N/Test/arm-linux-gcc$ gcc -S -o example.s example.i

myroot@myroot:~/TQ210N/Test/arm-linux-gcc$ ls

example.c example.i example.s

//gcc汇编：将编译得到的“.s”汇编文件按照给定的指令集转换成符合一定格式的机器码

myroot@myroot:~/TQ210N/Test/arm-linux-gcc$ gcc -c -o example.o example.s

myroot@myroot:~/TQ210N/Test/arm-linux-gcc$ ls

example.c example.i example.o example.s

//此时的example.o还不能执行，还不是可以执行的二进制文件，如下：

myroot@myroot:~/TQ210N/Test/arm-linux-gcc$ ./example.o

-bash: ./example.o: Permission denied

myroot@myroot:~/TQ210N/Test/arm-linux-gcc$ chmod +x example.o

myroot@myroot:~/TQ210N/Test/arm-linux-gcc$ ls

example.c example.i example.o example.s

myroot@myroot:~/TQ210N/Test/arm-linux-gcc$ ./example.o

-bash: ./example.o: cannot execute binary file: Exec format error

//将example.o转换成最终的可执行二进制文件

myroot@myroot:~/TQ210N/Test/arm-linux-gcc$ gcc -o example.out example.o

myroot@myroot:~/TQ210N/Test/arm-linux-gcc$ ls

example.c example.i example.o example.out example.s

myroot@myroot:~/TQ210N/Test/arm-linux-gcc$ ./example.out

Hello World

12.arm-linux-gcc编译器的工具：

①链接器 arm-linux-ld：

-static 阻止支持动态链接的系统上链接共享库；

-nostdlib 使编译器不链接系统标准启动文件和标准库文件，只把指定的文件传递为链接器，一般用在内核、BootLoader等

-T “-T”选项可以用于指示链接器将“.text section”、“.data section”、“.bss section”放置在特定的起始地址。命令格式如下：

arm-linux-ld -Ttext startaddress

arm-linux-ld -Tdata startaddress

arm-linux-ld -Tbss startaddress

**arm-linux-ld -Ttext 0xD0020010 -o led\_on.elf led\_on.o**

**-o**选项设置输出文件的名字为**led\_on.elf，**输入文件**led\_on.o**

②格式转换工具 arm-linux-objcopy：

ELF格式的可执行文件无法再ARM裸机平台执行；

将编译生成的ELF格式的可执行程序转换成能被处理器执行的二进制文件；

-O 指定输入输出文件的格式；

-S 不将输入文件中的符号信息和重定位信息复制到输出文件中；

-g 不把输入文件中的调试信息复制到输出文件中；

**arm-linux-objcopy被用来复制一个目标文件的内容到另一个文件中.此选项可以进行格式的转换.在实际编程的,用的最多的就是将ELF格式的可执行文件转换为二进制文件.**

**arm-linux-objcopy –O binary –S file.elf file.bin**

③反汇编工具 arm-linux-objdump

-b 指定目标码格式；

-d 将输入文件的可执行段反汇编；

-D 将输入文件的所有段反汇编；

-EB 指定输出文件字节序为大端格式；

-EL 指定输出文件字节序为小端格式；

-f 显示文件的整体头部信息；

-i或-info 显示支持的目标文件格式和CPU架构。

**arm-linux-objdump -D led\_on.elf > led\_on.dis**

# **13.[Makefile万能写法(gcc程序以及arm-linux-gcc程序)](https://www.veryarm.com/20532.html)**

在linux下使用gcc 编译时，Makefile的万能写法 ，每次只需更改要生成的目标文件名称(test)即可:

objs := $(patsubst %c, %o, $(shell ls \*.c))

test.all:$(objs)

gcc -o $@ $^

%.o:%.c

gcc -c -o $@ $<

clean:

rm -f \*.all \*.o

****在arm交叉编译时的makefile的万能写法，只需更改int.bin，以及int\_elf,int.dis名称即可****

objs := $(addsuffix .o, $(basename $(shell ls -U \*.S \*.c)))

int.bin: $(objs)

arm-linux-ld -Ttext 0x00000000 -o int\_elf $^

arm-linux-objcopy -O binary -S int\_elf $@

arm-linux-objdump -D -m arm int\_elf > int.dis

%.o:%.c

arm-linux-gcc -Wall -O2 -c -o $@ $<

%.o:%.S

arm-linux-gcc -Wall -O2 -c -o $@ $<

clean:

rm -f \*.bin \*\_elf \*.dis \*.o

a)、$(basename names.....)

原理:抽取除"names...."中每一个文件名中除后缀外的一切字符

比如：$(basename head.S hello.c helloworld.c)

结果为:head hello helloworld

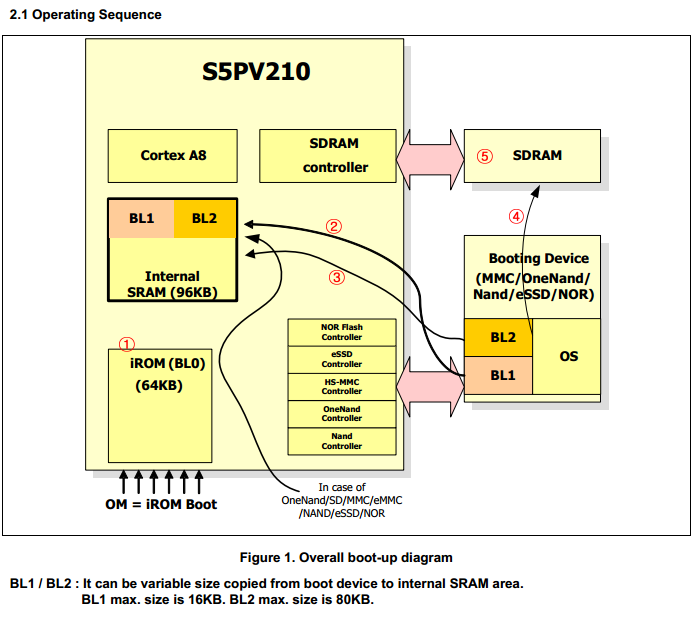
b)、$(addsuffix suffix,names...)

如：$(addsuffix .o, head hello helloworld)

 结果为:head.o hello.o hello.o helloworld.o

**1.S5PV210启动流程**

学习任何一款处理器，首先要搞清楚它的启动流程。  
参考《S5PV210\_iROM\_ApplicationNote\_Preliminary\_20091126.pdf》  
S5PV210 支持从多种设备启动： OneNand、 Nand、 MMC 等。  
下面是S5PV210 的启动框图

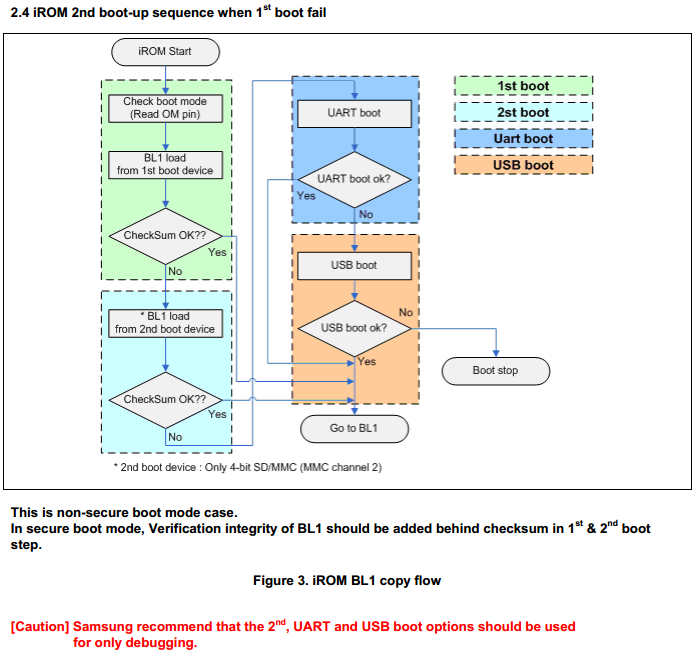


*S5PV20* 上电从 *iROM* 运行 *Samsung* 出厂时固化在里面的代码，这部分代码叫做 *BL0*（*boot loader0*），  
*BL0* 将执行如下操作：

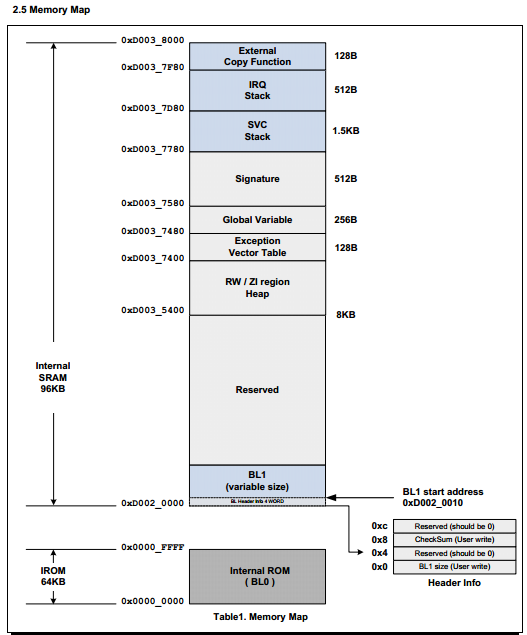
1. 禁止看门狗  
   *2.* 初始化指令*cache*

3.初始化栈、堆  
*4.* 初始化块设备拷贝函数  
*5.* 初始化 *PLL*（锁相环）、设置系统时钟  
*6.* 根据 *OM* 引脚配置，从指定的外部存储器拷贝 *BL1* 到内部 *SRAM  
7.* 校验 *BL1* 的校验和，如果校验成功则跳转到 *BL1* 的起始地址执行 *BL1*，否则进入第 *2* 个启动  
序列（见下图）

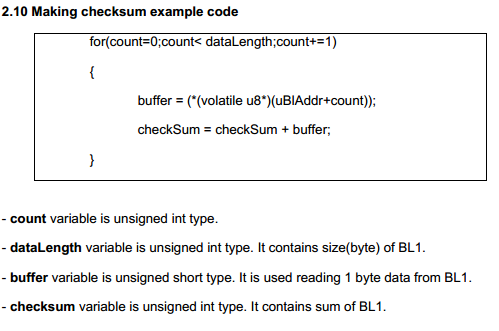
*8. BL1* 拷贝 *BL2* 到内部 *SRAM*，然后跳转到 *BL2* 的起始地址执行 *BL2  
9. BL2* 初始化 *DRAM* 控制器，加载 *OS* 到外部 *SDRAM*。  
*10. BL2* 最终跳转到 *OS* 的起始地址执行 *OS*。

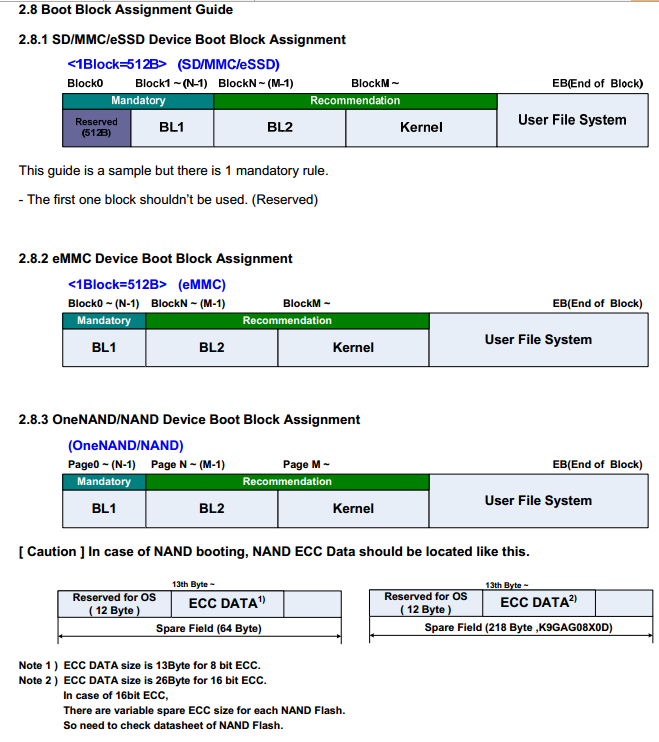


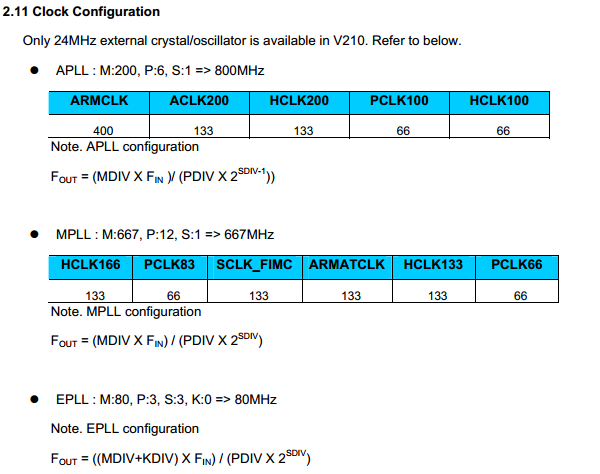
其中*BL1*最大*16KB*， *BL2*最大*80KB*，而后面我们自己移植的*u-boot*有*200* 多*Kbyte*，因此就没法按照手册给的这个流程。 我们实际的流程是：在 *BL1* 中初始化时钟、 *DRAM* 控制器，拷贝 *BL2*到外部 *DRAM*，跳转到 *DRAM* 中执行 *BL2*， *BL2* 加载 *OS* 到 *OS* 的起始地址执行 *OS*。



从上面的 *S5PV210* 启动时的内存映射图可以知道我们自己写的 *BL1* 需要 *16 Byte* 的头信息，这 *16* 字节的头信息格式在图中已经明确定义：  
 *0x0*： *BL1* 的大小  
 *0x4*： *0*（规定）  
 *0x8*：校验和  
 *0xc*： *0*（规定）  
 有了这 *16* 字节的头信息， *BL0* 就知道拷贝多大的 *BL1* 到内部 *SRAM*，并且可以验证 *BL1* 的数据是否完好无损。  
 校验和的计算方法如下：

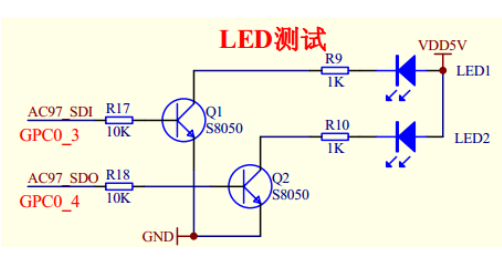




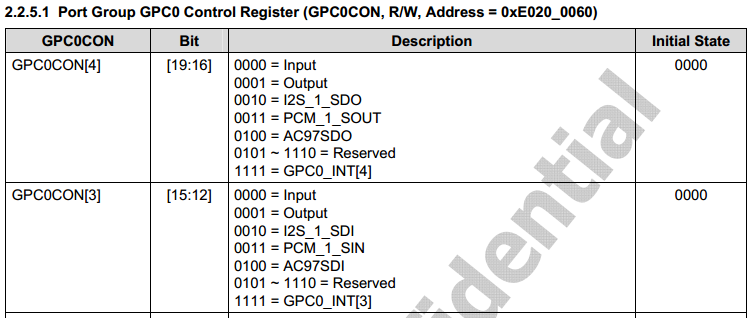


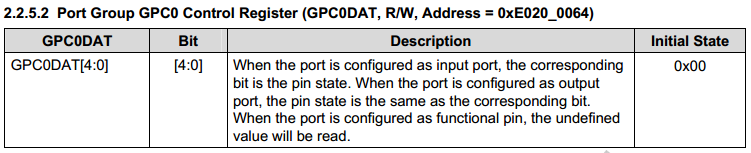
**2 点亮一个LED**

首先查看 *TQ210* 的底板原理图



两个*LED*分别接到*S5PV210*的*GPC0\_3*和*GPC0\_4* 引脚。  
这里用了 *NPN* 三极管，具有放大电流的作用，增大驱动能力， 我们只要给它的基极一个高电平，三  
极管的集电极和发射极就会导通，产生大电流驱动 *LED* 发光





要点亮*LED1*，我们需要配置寄存器 *GPC0CON* 的*[15:12]*为 *0b0001*，使 *GPC0\_3* 为输出模式，同时配  
置寄存器 *GPC0DAT[3]=1*，使 *GPC0\_3* 引脚输出高电平。  
源代码位置： TQ210N*/TQ210\_NoOS/1-led\_on/led\_on.S*

/\* led\_on.S \*/

.global \_start /\* 声明一个全局的标号 \*/

\_start:

ldr r0, =0xE0200060 /\* GPC0CON寄存器 \*/

ldr r1, =0x00001000

str r1, [r0] /\* 设置GPC0\_3为输出,GPC0[15:12] = 0b0001 \*/

ldr r0, =0xE0200064 /\* GPC0DAT寄存器 \*/

ldr r1, =0x00000008

str r1, [r0] /\* 设置GPC0\_3为高电平 \*/

halt:

b halt /\* 死循环 \*/

问：为什么需要后面的死循环？  
答：*CPU* 一旦从某个地址运行，它就会从这个地址往后依次取指运行，当运行完我们的代码，它不  
会停止，还会往后继续取指运行，但是后面的指令是未知的， *CPU* 运行后不知道会是什么结果，可能正常执行，也可能出现异常，所以我们应该让 *CPU* 一直在那里死循环。

*Makefile* 的内容如下：

led\_addhead.bin:led\_on.bin

../../Tools/checksum/s5pv210addheader $^ $@

led\_on.bin: led\_on.o

arm-linux-ld -Ttext 0xD0020010 -o led\_on.elf $^

arm-linux-objcopy -O binary -S led\_on.elf $@

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

led\_on.o : led\_on.S

arm-linux-gcc -c $< -o $@

clean:

rm \*.o \*.elf \*.bin \*.dis

这里指定了程序的链接地址（运行地址）为 *0xD0020010*， 从上一节可以知道，这个地址为 *BL1* 的起  
始地址。执行 *make* 后最终生成二进制文件 *led\_on.bin*。  
我们还需要为 *led\_on.bin* 添加 *16* 字节的头信息。

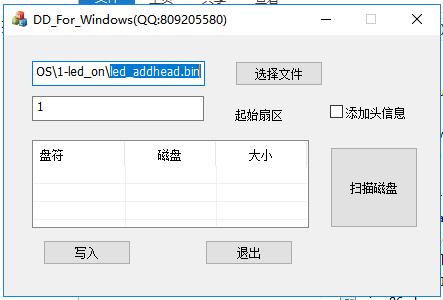
我们需要编写 s5pv210addheader.c，用来为 *led\_on.bin* 添加头信息，首先编译 s5pv210addheader.c

程序位置：TQ210N/Tools/checksum/s5pv210addheader.c  
*gcc* s5pv210addheader.c *-o* s5pv210addheader制作目标文件  
直接在Makefile文件中添加生成目标文件led\_addhead.bin

led\_addhead.bin:led\_on.bin

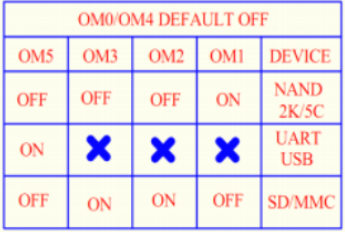
../../Tools/checksum/s5pv210addheader $^ $@

我们需要将生成的 led\_addhead.bin文件烧写到 *SD* 卡的扇区 *1*，将 *SD* 卡插入读卡器，将读卡器插入电脑，在 *Windows* 下对 *SD* 卡进行烧写，首先将 *SD* 卡插入笔记本的 *SD* 卡槽，然后运行 *DD\_For\_Windows.exe*



默认选择的文件为 *E:\210.bin*，单击选择文件找到你要烧写的文件，输入起始扇区 *1*，右边有个选项，  
是否添加头信息， 选中后，这个软件会添加 *BL1* 的头信息，这时就要选择原始的没有添加头信息的二进制文件 *led\_on.bin*，单击写入完成烧写。

现在将 *SD* 卡取出插到 *TQ210* 开发板上， 然后拨动启动选择开关，选择从 *SD* 启动



给开发板上电， *LED* 已经点亮。

使用 *C* 语言实现流水灯，源码位置： TQ210N*/TQ210\_NoOS/1-led\_on\_c/*

/\* start.S \*/

.global \_start /\* 声明一个全局的标号 \*/

\_start:

bl main /\* 跳转到C函数去执行 \*/

halt:

b halt /\* 死循环 \*/

/\* main.c \*/

#define GPC0CON \*((volatile unsigned int \*)0xE0200060)

#define GPC0DAT \*((volatile unsigned int \*)0xE0200064)

void delay(volatile unsigned int t)

{

volatile unsigned int t2 = 0xFFFF;

while (t--)

for (; t2; t2--);

}

int main()

{

int toggle = 0;

GPC0CON &= ~(0xFF << 12);

GPC0CON |= 0x11 << 12; // 配置GPC0\_3和GPC0\_4为输出

while (1)

{

GPC0DAT &= ~(0x3 << 3); // 熄灭LED1和LED2

if (toggle)

GPC0DAT |= 1 << 3; // 点亮LED1

else

GPC0DAT |= 1 << 4; // 点亮LED2

toggle = !toggle;

delay(0x50000);

}

return 0;

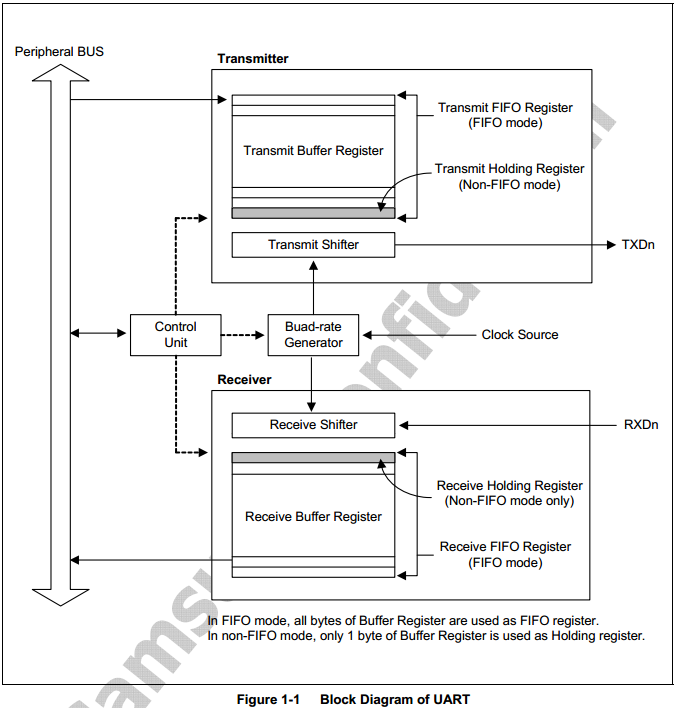
}

按照上面的烧写方法，将程序烧写到 *SD* 卡，给开发板上电， *2* 个 *LED* 开始交替闪烁。  
问： 运行 *C* 语言需要栈，为什么在 *start.S* 中没有设置栈？  
答： *S5PV210*上电运行 *iROM* 中的代码已经设置好栈，栈顶地址为 *0xD0037F80*，参考 *1* 节。

**3.串口**

*S5PV210* 包含 *4* 个异步收发器（*UART*），提供 *4* 个独立的异步串行输入*/*输出（*I/O*）端口。所有端口  
可工作于中断模式或 *DMA* 模式。提供高达 *3Mbps* 的位速率。每个 *UART* 包含 *2* 个 *FIFO* 用于接收和发送数据。具有可编程的波特率、红外收发、 *1* 位或 *2* 位停止位、 *5~8* 位数据位、校验。

**KEY FEATURES OF UNIVERSAL ASYNCHRONOUS RECEIVER AND TRANSMITTER**• RxD0, TxD0, RxD1, TxD1, RxD2, TxD2, RxD3 and TxD3 with DMA-based or interrupt-based operation  
• UART Ch 0, 1, 2 and 3 with IrDA 1.0  
• UART Ch 0 with 256-byte FIFO, Ch 1 with 64-byte FIFO, Ch2 and 3 with 16-byte FIFO  
• UART Ch 0, 1 and 2 with nRTS0, nCTS0, nRTS1, nCTS1, nCTS2 and nRTS2 for Auto Flow Control  
• Supports handshake transmit/receive

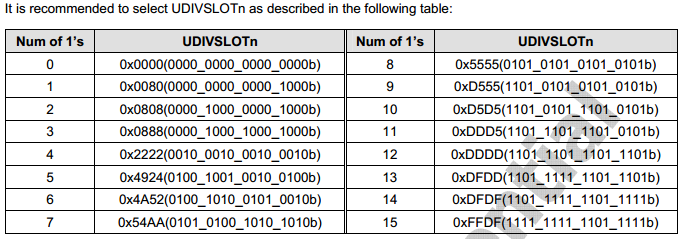


数据发送：要发送的数据帧是可编程的。它包含 *1* 位起始位， *5~8* 位数据位， *1* 个可选校验位， *1*或 *2* 位停止位，这些都通过 *ULCONn* 寄存器来设置。在 *FIFO* 模式下发送器将要发送的数据发送给 *Tx FIFO*，在非*FIFO* 模式下，发送器将要发送的数据发送给 *Tx* 保持寄存器。  
 数据接收：和数据发送类似。

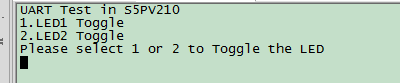
串口编程操作步骤如下：  
 *1*、配置时钟，选择时钟源  
 *2*、配置 *ULCONn* 寄存器：设置数据位、停止位、 校验位、模式  
 *3*、配置 *UCONn* 寄存器：设置数据接收和发送模式、时钟源  
 *4*、设置 *UFCONn*：启用或静止 *FIFO  
 5*、配置 *UBRDIVn* 和 *UDIVSLOTn*：计算波特率  
 *6*、发送数据：等待发送器为空，将要发送的 *8* 位数据赋给发送缓存寄存器 *UTXHn  
 7*、接收数据：等待接收缓冲区有数据可读，从接收缓存寄存器 *URXHn* 中取出数据

波特率计算：  
*DIV\_VAL = UBRDIVn + (num of 1's in UDIVSLOTn)/16  
DIV\_VAL = (PCLK / (bps x 16)) -1*

或者  
*DIV\_VAL = (SCLK\_UART / (bps x 16)) -1*比如配置波特率为 *115200bps*，时钟源选择 *PCLK=66MHz  
DIV\_VAL = (66000000/(115200 x 16))-1 = 35.8 - 1 = 34.8  
UBRDIV0 = 34* （*DIV\_VAL* 的整数部分）  
*(num of 1's in UDIVSLOTn)/16 = 0.8* （*DIV\_VAL* 的小数部分）  
*(num of 1's in UDIVSLOTn) = 12  
UDIVSLOT0 = 0xDDDD (*查表*)*



实验现象： 按数字 *1* 改变 *LED1* 的状态；按数字 *2* 改变 *LED2* 的状态。



源代码位置： TQ210N*/TQ210\_NoOS/2-uart*由于我们在 *uart.c* 中使用了和 *C* 库同名的函数： *putchar*、 *getchar*、 *puts*，为了不和 *C* 库中的同名函数发送冲突，需要给 *gcc* 加一个选项*-fno-builtin*，不使用内建函数。  
问：为什么我们没有进行时钟配置相关的操作？  
答：因为 *S5PV210* 在启动时，运行 *iROM* 里的代码已经为我们初始化了时钟，其中 *PCLK=66MHz*，  
*1* 节有说明。

**4 实现 printf 函数**

*printf* 的原型如下

***int printf(const char \*format, ...);***它带有一个固定的参数 *format*，紧接着是可选参数，所谓可选参数就是参数个数和类型都不固定，  
也可以没有。  
*C* 语言参数传递的入栈顺序是**从右到左。**  
比如在 *ARM Linux* 系统下， *ATPCS* 规定为递减堆栈，假设栈顶 *SP=0x24000000*假设有这么一个函数 *void foo(int a, int b, int c);*则参数 *c* 先压入栈，地址为 *SP-4=0x23FFFFFC;*紧接着 *b* 压入栈，地址为 *SP-4=0x23FFFFF8;*紧接着 *a* 压入栈，地址为 *SP-4=0x23FFFFF4;*读取可变参数的过程其实就是在堆栈中，使用指针*,*遍历堆栈段中的参数列表*,*从低地址到高地址一  
个一个地把参数内容读出来的过程。  
标准 *C* 为我们提供了相关的宏来处理可选参数，这些宏在 *stdarg.h* 中定义。在 *Linux* 发行版系统（比  
如 *ubuntu*）中执行 *man stdarg* 可查看相关帮助信息。

#include <stdarg.h>

void va\_start(va\_list ap, last);

type va\_arg(va\_list ap, type);

void va\_end(va\_list ap);

void va\_copy(va\_list dest, va\_list src);

*va\_list ap:*实际上就是 *int \*ap*；

*va\_start:*初始化 *ap*，使 *ap* 指向第一个可选参数， *last* 就是调用函数的第一个固定参数；

*va\_arg:*解析出一个可变参数，并使 *ap* 指向下一个可选参数；  
*va\_end :*使 *ap* 指针无效。  
注意：在同一个函数中，每次对 *va\_start* 的调用都必须有对应的 *va\_end* 调用与之匹配。  
通过 *man stdarg* 查询的帮助信息中有一个例子可以参考：

EXAMPLE

The function foo takes a string of format characters and prints out the argument

associated with each format character based on the type.

#include <stdio.h>

#include <stdarg.h>

void

foo(char \*fmt, ...)

{

va\_list ap;

int d;

char c, \*s;

va\_start(ap, fmt);

while (\*fmt)

switch (\*fmt++) {

case 's': /\* string \*/

s = va\_arg(ap, char \*);

printf("string %s\n", s);

break;

case 'd': /\* int \*/

d = va\_arg(ap, int);

printf("int %d\n", d);

break;

case 'c': /\* char \*/

/\* need a cast here since va\_arg only

takes fully promoted types \*/

c = (char) va\_arg(ap, int);

printf("char %c\n", c);

break;

}

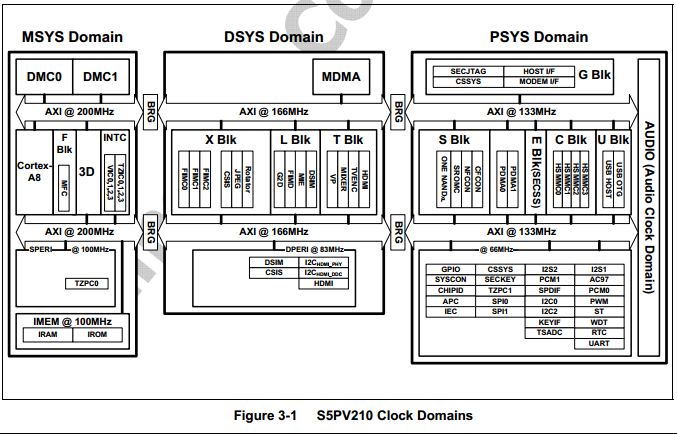
va\_end(ap);

}

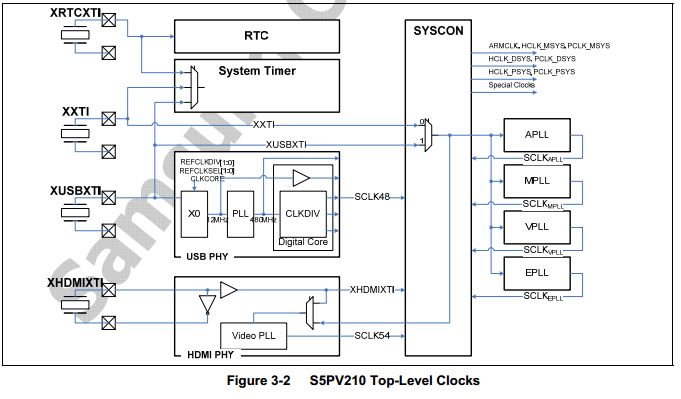
源代码位置：TQ210N*/TQ210\_NoOS/3-printf*

**5 时钟配置**

前面的操作都是使用 *S5PV210* 的默认时钟配置，其主频*(ARMCLK)*只有*400MHz*，我们需要把它提高到 *1GHz*。  
 *S5PV210* 由 *3* 个时钟域构成，分别是主系统（*MSYS*），显示系统（*DSYS*），外围系统（*PSYS*），如下图所示：



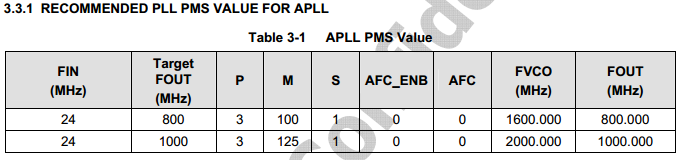
 *MSYS* 域服务对象： *Cortex A8* 核、 *DRAM* 内存控制器、 *3D*、 *iROM*、 *iRAM*、 *INTC* 等  
 *DSYS* 域服务对象： 显示相关模块，包括 *FIMC*、 *FIMD*、 *JPEG* 等  
 *PSYS* 域服务对象： 安全、 *I/O* 外围、低功耗的声音播放等  
 每个总线系统分别工作在最大 *200MHz*、*166MHz*、*133MHz*。不同的域之间通过异步总线桥（*BRG*）  
相连接  
*S5PV210* 最顶层时钟来源包括：  
 来自于外部晶振： *XRTCXTI, XXTI, XUSBXTI, and XHDMIXTI* 来自于 *CMU*（时钟管理单元）： *ARMCLK, HCLK, PCLK* 等  
 来自于 *USB PHY* 来自于 *GPIO* 引脚  
如下图所示：

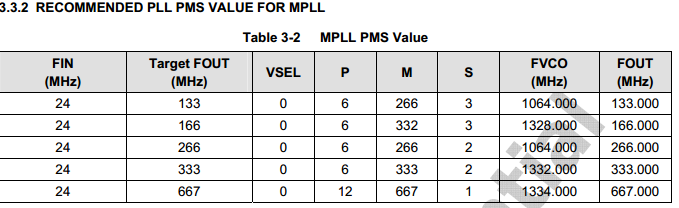


*S5PV210* 包含 *4* 个 *PLL*（锁相环）， *PLL* 就是用来将外部晶振（*XXTI*） 的输入时钟的频率放大。  
*S5PV210* 手册建议使用 *24MHz* 的输入时钟作为 *APLL*、 *MPLL*、 *VPLL*、 *EPLL* 的时钟源。  
 *APLL* 使用 *FINPLL* 作为输入时钟源产生 *30MHz ~ 1GHz* 的频率 *SCLKAPLL* *MPLL* 使用 *FINPLL* 作为输入时钟源产生 *50MHz ~ 2GHz* 的频率 *SCLKMPLL* *EPLL* 使用 *FINPLL* 作为输入时钟源产生 *10MHz ~ 600Hz* 的频率 *SCLKEPLL* *VPLL* 使用 *FINPLL* 或者 *SCLK\_HDMI27M* 作为输入时钟源产生 *10MHz ~ 600Hz* 的频率 *SCLKVPLL*，产  
生 *54MHz* 的 *video clock*上面的 *FINPLL* 即外部晶振输入时钟 *24MHz*在 *S5PV210* 中的典型应用：  
 *Cortex-A8* 和 *MSYS* 时钟域使用 *APLL*（*ARMCLK*， *HCLK\_MSYS* 和 *PCLK\_MSYS*）  
 *DSYS* 和 *PSYS* 时钟域*(HCLK\_DSYS*， *HCLK\_PSYS*， *PCLK\_DSYS* 和 *PCLK\_PSYS)*和外围时钟*(GPIO*、 串口、*SPI* 等*)*使用 *MPLL* 和 *EPLL* *video clock* 使用 *VPLL*时钟控制器允许为低速时钟避开 *PLL*。也可以使用软件编程从每个时钟块连接和断开，达到降低功  
耗的目的。  
下图列出了 *S5PV210* 的时钟关联：

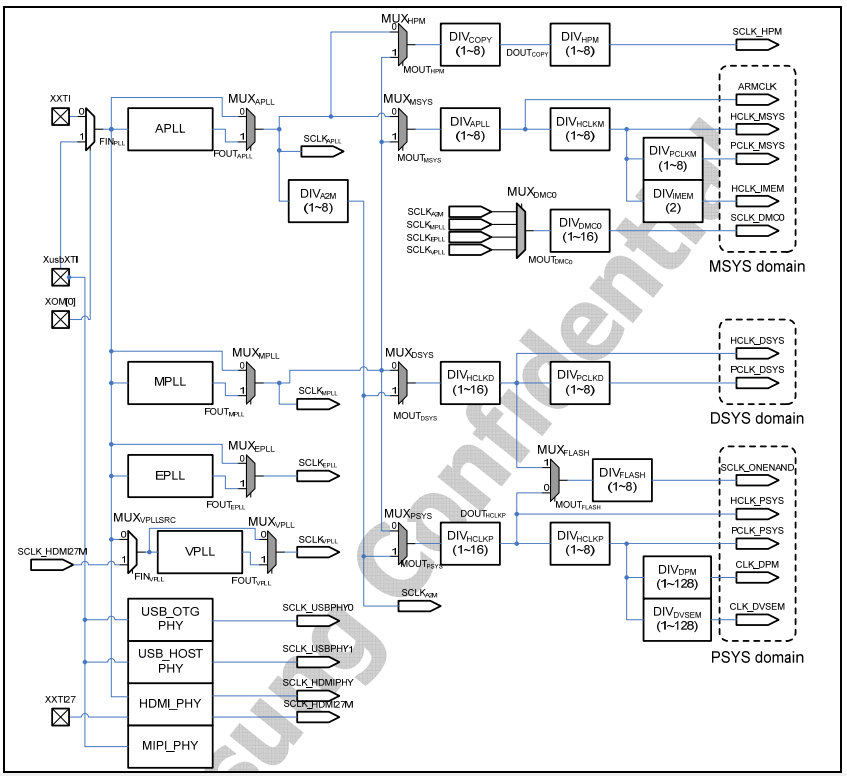
**CLOCK RELATIONSHIP**Clocks have the following relationship:  
• **MSYS clock domain**- freq(ARMCLK) = freq(MOUT\_MSYS) / n, where n = 1 ~ 8  
- freq(HCLK\_MSYS) = freq(ARMCLK) / n, where n = 1 ~ 8  
- freq(PCLK\_MSYS) = freq(HCLK\_MSYS) / n, where n = 1 ~ 8  
- freq(HCLK\_IMEM) = freq(HCLK\_MSYS) / 2  
• **DSYS clock domain**- freq(HCLK\_DSYS) = freq(MOUT\_DSYS) / n, where n = 1 ~ 16  
- freq(PCLK\_DSYS) = freq(HCLK\_DSYS) / n, where n = 1 ~ 8  
• **PSYS clock domain**  
- freq(HCLK\_PSYS) = freq(MOUT\_PSYS) / n, where n = 1 ~ 16  
- freq(PCLK\_PSYS) = freq(HCLK\_PSYS) / n, where n = 1 ~ 8  
- freq(SCLK\_ONENAND) = freq(HCLK\_PSYS) / n, where n = 1 ~ 8  
Values for the high-performance operation:  
• freq(ARMCLK) = 1000 MHz  
• freq(HCLK\_MSYS) = 200 MHz  
• freq(HCLK\_IMEM) = 100 MHz  
• freq(PCLK\_MSYS) = 100 MHz  
• freq(HCLK\_DSYS) = 166 MHz  
• freq(PCLK\_DSYS) = 83 MHz  
• freq(HCLK\_PSYS) = 133 MHz  
• freq(PCLK\_PSYS) = 66 MHz  
• freq(SCLK\_ONENAND) = 133 MHz, 166 MHz

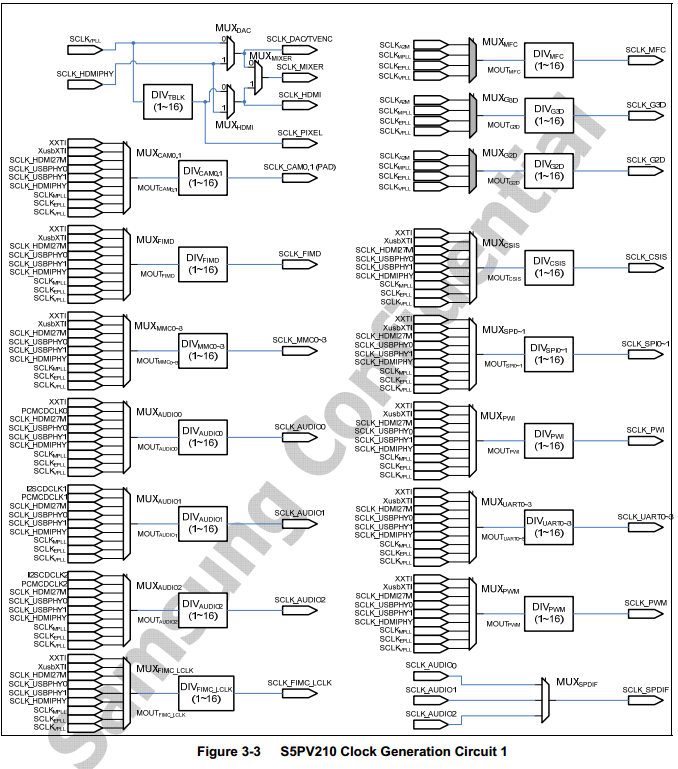
*APLL* 能够驱动 *MSYS* 域和 *DSYS* 域。产生高达 *1 GHz* 的频率  
*MPLL* 驱动 *MSYS* 域 *and DSYS* 域。产生高达 *2GHz* 的频率  
*EPLL* 主要用于产生声音相关的时钟  
*VPLL* 主要用于 *video* 系统，产生 *54 MHz* 的频率  
典型的 *APLL* 驱动 *MSYS* 域， *MPLL* 驱动 *DSYS* 域



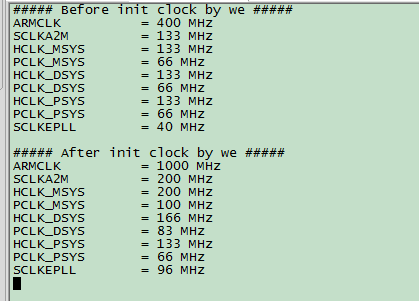
  
时钟配置步骤如下：  
*1.* 设置 *PLL* 锁定值 *//* 参考 *PLL\_LOCK* 寄存器  
*2.* 设置 *PLL* 的 *PMS* 值，并使能 *PLL //* 参考 *PLL\_CON* 寄存器和手册 *3.3.1*、 *3.3.2*、 *3.3.3*、 *3.3.4  
3.* 等待 *PLL* 锁定（即等待 *PLL* 输出稳定的频率） *//* 读取 *PLL\_CON* 寄存器的 *LOCKED* 位来判断

4.设置系统时钟源：选择 *PLL*， 而不是外部晶振 *//* 参考 *CLK\_SRC0* 寄存器  
*5.* 设置其他模块的时钟源 *//* 参考 *CLK\_SRC1~CLK\_SRC6* 寄存器  
*6.* 设置系统时钟分频值 *//* 参考 *CLK\_DIV0* 寄存器  
*7.* 设置其他模块的时钟分频值 *//* 参考 *CLK\_DIV1~CLK\_DIV7*具体配置参考下面两个框图，在后续硬件操作中，都要参这 *2* 个图，非常重要：





实验现象： 程序中实现了一个频率计算函数，分别在调用我们自己实现的时钟初始化函数前后执行  
一次，输出系统默认时钟配置信息和我们配置的时钟信息。 由于在这个函数中使用了除法操作， 需要链接 *libgcc.a* 这个静态库， 具体看*Makefile*， 同时需要实现一个函数： *void raise(int signum)*。  
实验现象：



可以屏蔽 *main.c* 中的时钟初始化函数， 可以观察到 *LED* 的闪烁频率变慢了。  
源码位置： TQ210N*/TQ210\_NoOS/4-clock*

/\* 2.设置PLL的PMS值(使用芯片手册推荐的值)，并使能PLL \*/

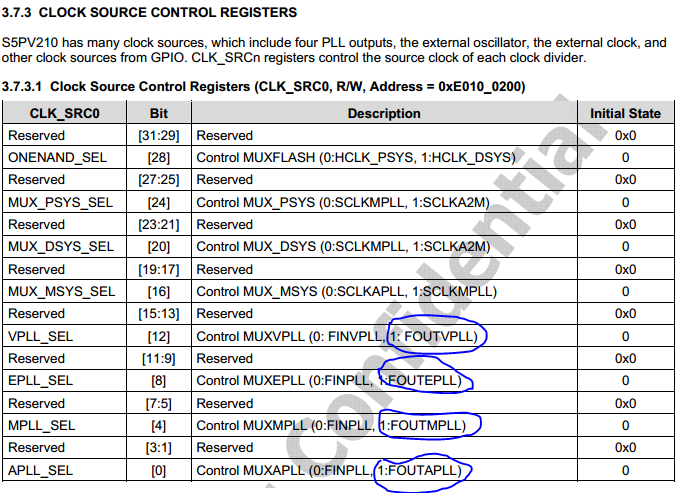
/\* P M S EN \*/

APLL\_CON0 = (3 << 8) | (125 << 16) | (1 << 0) | (1 << 31); /\* FOUT\_APLL = 1000MHz \*/

MPLL\_CON = (12 << 8) | (667 << 16) | (1 << 0) | (1 << 31); /\* FOUT\_MPLL = 667MHz \*/

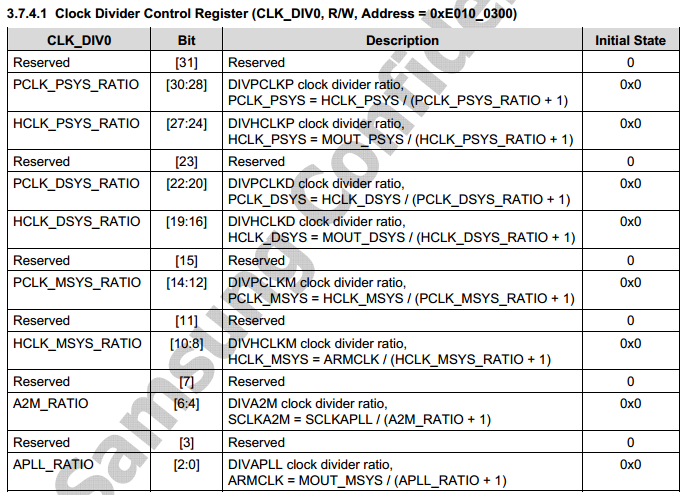
EPLL\_CON0 = (3 << 8) | (48 << 16) | (2 << 0) | (1 << 31); /\* FOUT\_EPLL = 96MHz \*/

VPLL\_CON = (6 << 8) | (108 << 16) | (3 << 0) | (1 << 31); /\* FOUT\_VPLL = 54MHz \*/



选择图上圈出的输出时钟源配置如下：

CLK\_SRC0 = (1 << 0) | (1 << 4) | (1 << 8) | (1 << 12);



/\* 6.设置系统时钟分频值 \*/

CLK\_DIV0 = (0 << 0) |/\* APLL\_RATIO = 0, freq(ARMCLK) = MOUT\_MSYS / (APLL\_RATIO + 1) = 1000MHz \*/

(4 << 4) |/\* A2M\_RATIO = 4, freq(A2M) = SCLKAPLL / (A2M\_RATIO + 1) = 200MHz \*/

(4 << 8) |/\* HCLK\_MSYS\_RATIO = 4, freq(HCLK\_MSYS) = ARMCLK / (HCLK\_MSYS\_RATIO + 1) = 200MHz \*/

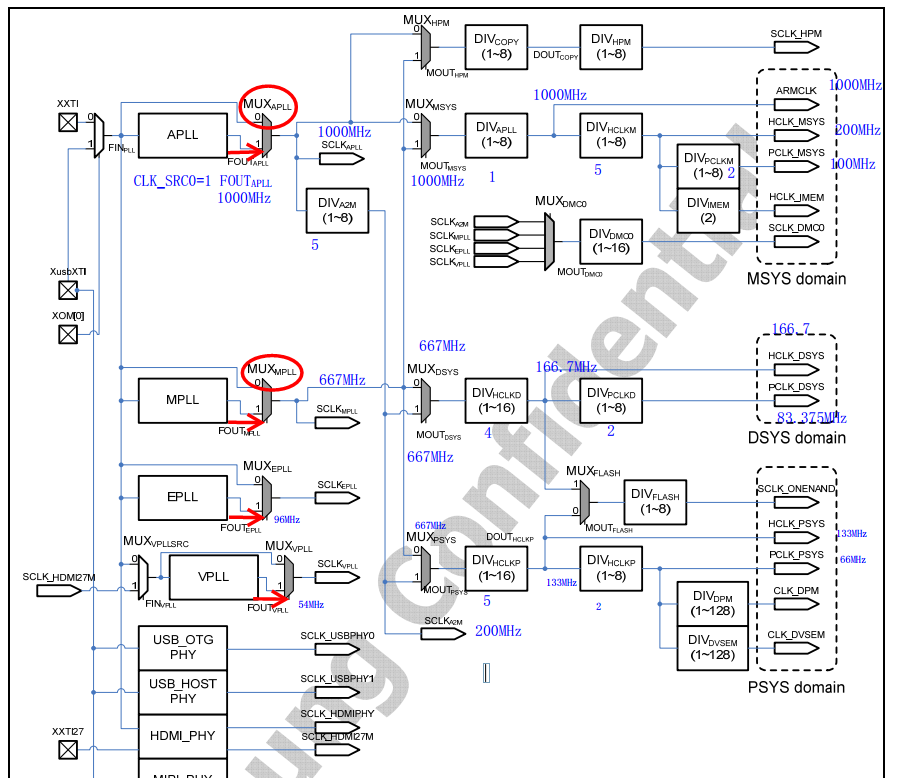
(1 << 12) |/\* PCLK\_MSYS\_RATIO = 1, freq(PCLK\_MSYS) = HCLK\_MSYS / (PCLK\_MSYS\_RATIO + 1) = 100MHz \*/

(3 << 16) | /\* HCLK\_DSYS\_RATIO = 3, freq(HCLK\_DSYS) = MOUT\_DSYS / (HCLK\_DSYS\_RATIO + 1) = 166MHz \*/

(1 << 20) | /\* PCLK\_DSYS\_RATIO = 1, freq(PCLK\_DSYS) = HCLK\_DSYS / (PCLK\_DSYS\_RATIO + 1) = 83MHz \*/

(4 << 24) | /\* HCLK\_PSYS\_RATIO = 4, freq(HCLK\_PSYS) = MOUT\_PSYS / (HCLK\_PSYS\_RATIO + 1) = 133MHz \*/

(1 << 28); /\* PCLK\_PSYS\_RATIO = 1, freq(PCLK\_PSYS) = HCLK\_PSYS / (PCLK\_PSYS\_RATIO + 1) = 66MHz \*/



**6重定位（BL1 加载 BL2 到 SRAM）**

这次试验有两个程序， *bl1.bin* 和 *bl2.bin*，我们将 *bl1.bin* 烧写到 *SD* 卡的扇区 *1*，将 *bl2.bin* 烧写到 *SD*卡的扇区 *20*，在 *bl1.bin* 中实现将 *bl2.bin* 从 *SD* 的扇区 *20* 拷贝到 *SRAM* 的 *0xD0022800* 地址处，然后跳转到 *bl2.bin* 的起始地址 *0xD0022800* 执行 *bl2.bin*。 三星固化在 *S5PV210 iROM* 中的代码提供了这个 *SD* 卡拷贝函数， 参考《*S5PV210 iROM ApplicationNote Preliminary 20091126*》 手册