Linux 平台下裸机程序开发指南

版本：2013-12-10

本手册适用于 Mini2451、Tiny2451、Tiny2416 开发板平台

地址：广州市天河区龙口西路龙苑大厦A1栋1705　网址：http://www.arm9.net
电话：+86-20-85201025(售前、售后咨询)　技术支持(Tel): 13719442657　传真：+86-20-85261505
E-Mail: capbily@163.com(商务或项目合作)　dev_friendlyarm@163.com (技术支持)

版 权 声 明

敬告：

在售开发板的手册会经常更新，请在 http://www.arm9home.net 网站查看最近更新，并下载最新手册，不再另行通知。

更新说明：

日期	说明
2013-12-1	本手册第一次发布，任何问题请反馈至capbily@163.com

第一章　简介

第一节　起源

对于很多嵌入式开发者和爱好者，特别是初学者，如何从底层开始了解和学习 ARM，绝非是一件容易的事！ 为此，友善之臂的工程师，花了很多时间和心血，基于 Mini2451 开发板编写了这份裸机教程，以帮助广大嵌入式爱好者更加深入地了解 S3C2451 的启动过程，同时让更多初学者能从裸机程序开始学习 S3C2451。鉴于每个人的认知水平不同，以及我们平时的开发任务比较紧张，我们并不提供关于该教程的任何技术支持。如果你对本教程的内容有任何疑问，可以到 arm9 之家论坛(http://www.arm9home.net) 反馈，并和其他网友交流讨论。需要说明的是，本教程也适用于友善之臂出品的其他型号基于 S3C2451 的开发板平台。友善之臂将对本教程作不定期的维护和补充，请时常留意论坛的更新信息，不再另行通知。友善之臂(http://www.arm9.net)保留本教程的一切解释权。

第二节　关于开发环境

学习前提：学习过简单的 C 语言和 ARM 汇编语言

开发平台：Windows XP + 虚拟机 Fedora15，使用 Eclipse 编写代码

交叉编译器：arm-linux-gcc-4.4.3
注意：交叉编译器的安装方法请参考开发板相关的用户手册。

配套硬件：本教程所有程序均在 Mini2451 上测试运行过，对于友善之臂其他型号的 S3C2451 和 S3C2416 开发板，本文档及代码仍然适用。另外，本教程使用的存储卡为大容量的 4G SDHC 卡。

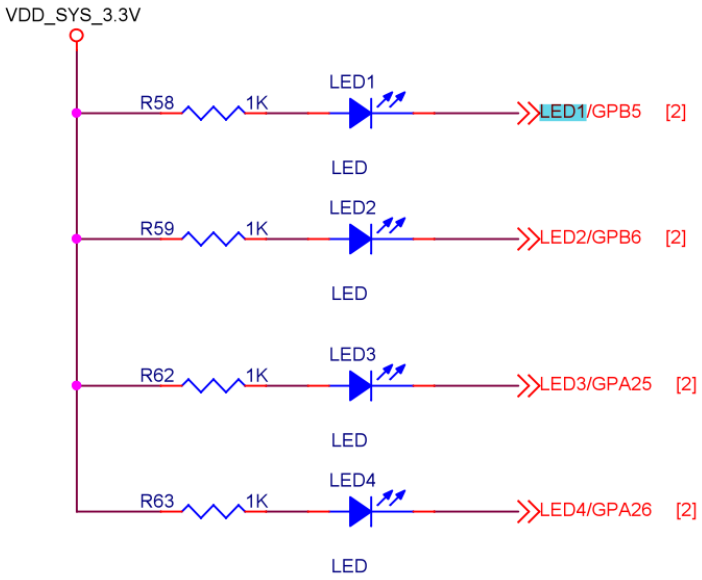
第三节　文档涉及的裸机程序

首先会从最基础的点亮 LED 讲起，先采用从 DRAM 加载裸机的方式，后面再采用 MMC/NAND 启动加载裸机的方式，逐步讲解 S3C2451 的启动过程以及如何重定位代码，让用户细致地了解 S3C2451 上电后运行的每一个步骤。用户可以通过查看书签了解本文档所涉及的硬件模块,本文档一共涉及 25 个裸机程序，并且以后将不断丰富出更多的裸机程序，也请众网友发挥各自的才智，在本文档的基础上写出更多简单实用的裸机程序并反馈给我们。目前提供的裸机示例包括：LED、按键、时钟、串口、内存、MMC 启动、NandFlash、蜂鸣器、中断、定时器、看门狗、RTC、LCD、ADC 转换。

第二章　汇编点亮 LED

第一节　查阅原理图

Mini2451 板上提供了 4 个可编程用户 LED，原理图如下：



LED 原理图

可见，LED1,2,3,4 分别使用的 CPU 端口资源为 GPB_5/6 和 GPA25/26。

第二节　程序相关讲解

完整代码见目录 1.led_s。注意：为方便用户阅读，本教程涉及的所有代码本身均含相当丰富的注释）

1. start.S

结合原理图，点亮 Mini2451 的 4 个 LED 需如下 3 个步骤：

1) 关闭看门狗，防止开发板不断重启；
2) 设置寄存器 GPBCON 和 GPACON，使 GPB_5/6 和 GPA25/26 四个引脚为输出功能,并设置寄存器 GPBSEL，使 GPB6 引脚用于 IO 功能；
3) 往寄存器 GPBDAT 的 bit5/6 和 GPADAT 的 bit25/26 写 0，使 GPB_5/6 和 GPA25/26 四个引脚输出低电平,4 个 LED 会亮；相反，往寄存器 GPBDAT 的 bit5/6 和 GPADAT 的 bit25/26 写 1，使 GPB_5/6 和 GPA25/26 四个引脚输出高电平，4 个 LED 会灭；

以上 3 个步骤即为 start.S 中的核心内容,start.S 里面涉及的汇编指令请自行学习 GNU 汇编

指令集，这里不再进行赘述。

2. Makefile

```
led.bin: start.o
    arm-linux-ld -Ttext 0x30000000 -o led.elf $^
    arm-linux-objcopy -O binary led.elf led.bin
    arm-linux-objdump -D led.elf > led_elf.dis
%.o : %.S
    arm-linux-gcc -o $@ $< -c
%.o : %.c
    arm-linux-gcc -o $@ $< -c
clean:
    rm *.o *.elf *.bin *.dis *~ -rf
```

关于 Makefile 网上的相关资料相当多，这里推荐《跟我一起写 makefile》,以方便用户后续的学习和查阅,下面我们只是进行粗略地讲解。当用户在 Makefile 所在目录下执行 make 命令时，系统会进行如下操作：

1) 执行 arm-linux-gcc -o $@ $< -c 命令将当前目录下存在的汇编文件和 C 文件编译成.o 文件；
2) 执行 arm-linux-ld -Ttext 0x30000000 -o led.elf $^将所有.o 文件链接成 elf 文件，-Ttext 0x30000000 表示程序的运行地址是 0x30000000,即程序只有位于该地址上才能正常运行；
3) 执行 arm-linux-objcopy -O binary led.elf led.bin 将 elf 文件抽取为可在开发板上运行的 bin 文件；
4) 执行 arm-linux-objdump -D led.elf > led_elf.dis 将 elf 文件反汇编后保存在 dis 文件中，调试程序时可能会用到；

第三节　编译代码和烧写运行

编译代码

在 Fedora 终端执行如下命令：

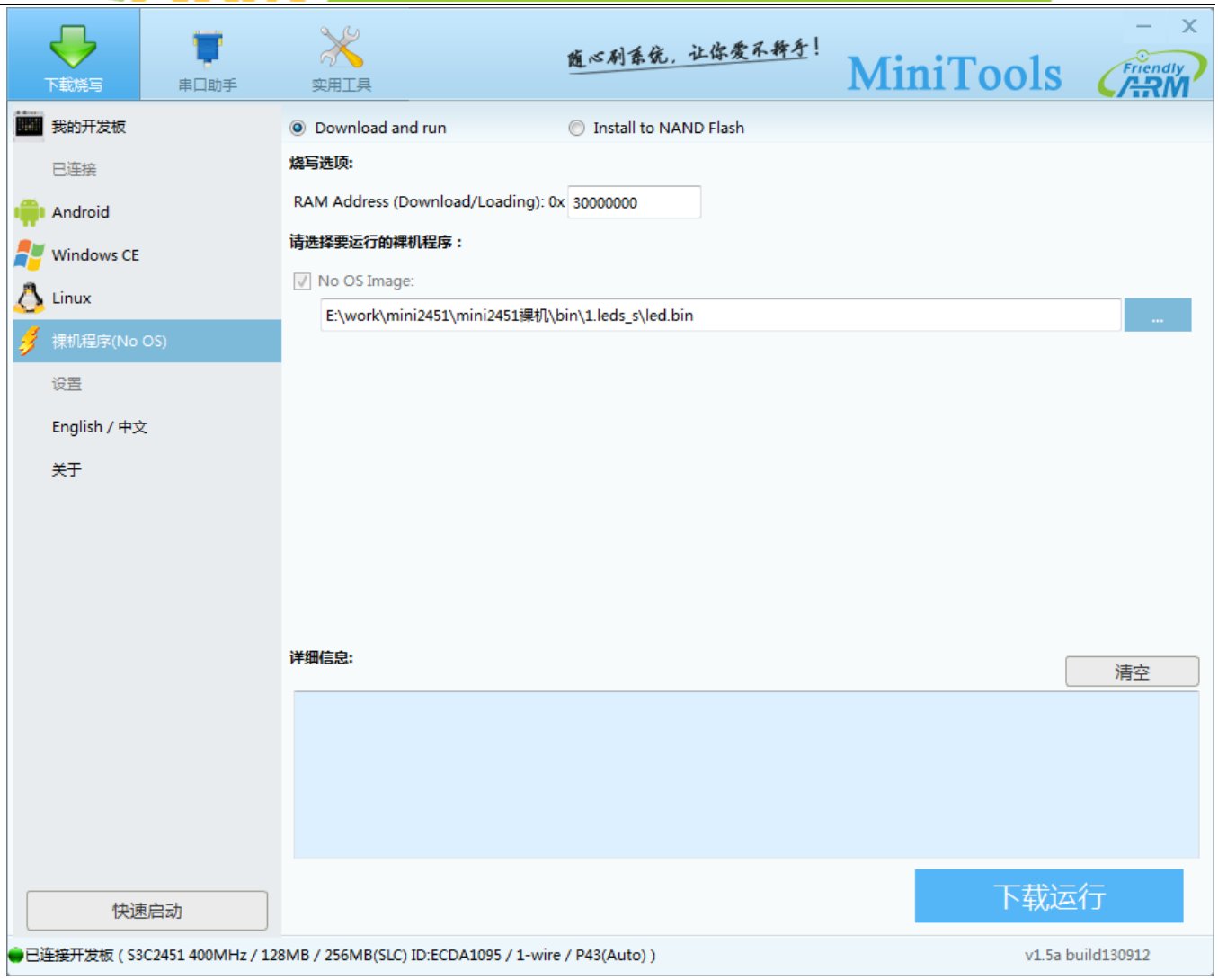
```
# cd 1.led_s
# make
```

make 成功后会生成 led.bin 文件。

烧写运行

使用 MiniTools 将 bin 文件烧写到开发板上。MiniTools 提供了两种烧写裸机程序的方式：一种是直接下载到内存 DRAM，另外一种是下载到 NAND Flash。注意：请先在 SDHC 卡中烧写带有 USB 下载功能的 Superboot。

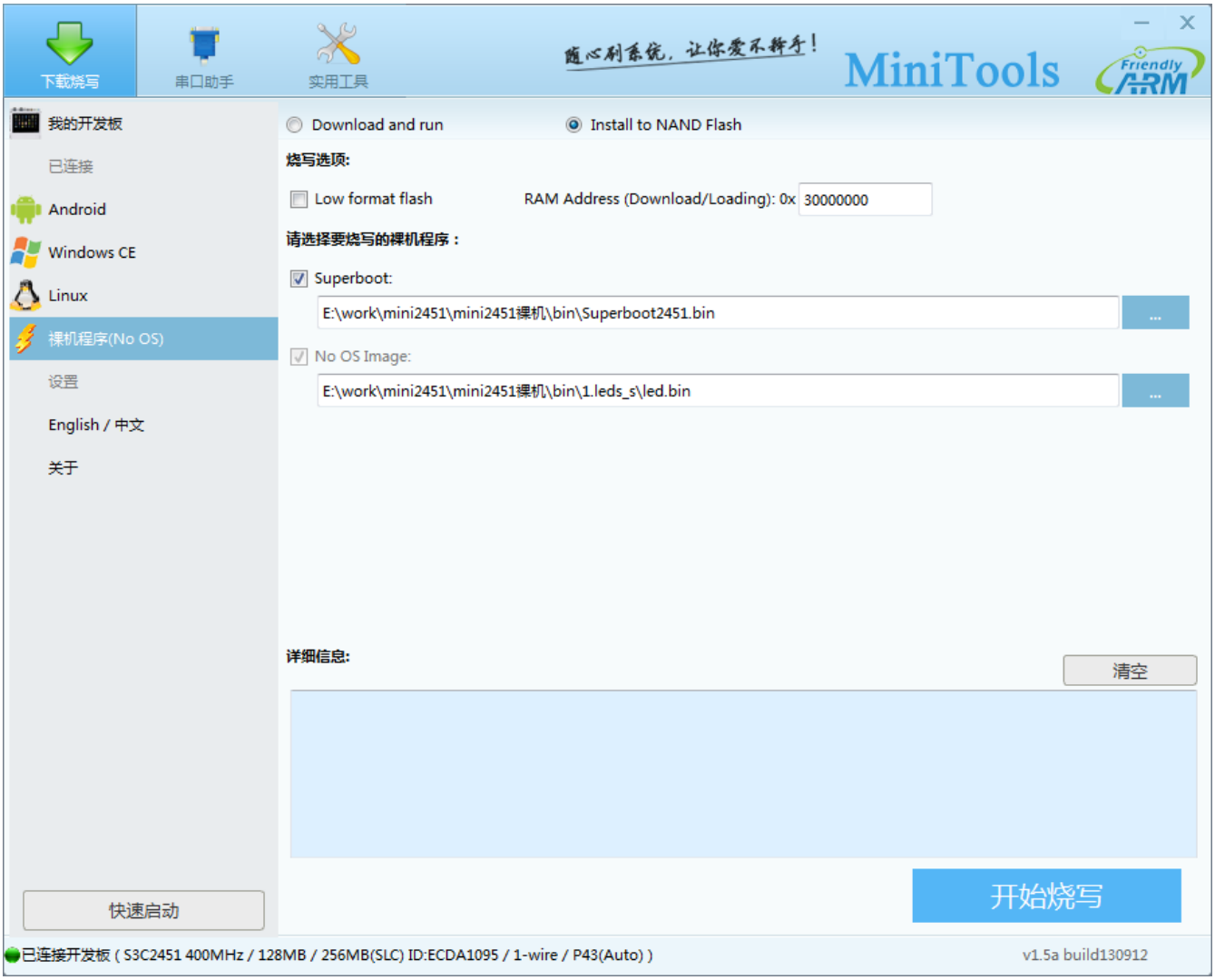
方式一 下载到 DRAM，其设置方式如下：



先选中上方的”Download and run“，设置好下载地址“RAM Address(Download/Loading)”，选择要运行的裸机程序， 再点击“下载运行”就可以了。MiniTools 首先会把裸机程序下载到 DRAM 的地址 0x30000000 处，然后跳转到该地址上运行裸机程序，所以只要 PC 上一点击“下载运行”就可以马上看到开发板上裸机程序的运行效果了。

注意：不同的 CPU 会有不同的内存起始地址，在此为 0x30000000

方式二 下载到 NANDFlash，其设置方式如下：



先选中“Install to NAND Flash”，设置好加载地址“RAM Address(Download/Loading)”，然后选择 Superboot，并选择要加载的裸机程序，最后点击“开始烧写”；MiniTools 首先把 Superboot 和裸机程序都烧写到 NAND Flash 中，以后只要选择 NAND 启动时，会先运行 NAND Flash 中的 Superboot， 然后 Superboot 就会将 NAND Flash 中的裸机程序加载到 DRAM 中运行了。

上面两种方式都可以烧写运行我们的裸机程序，由于方式一较为便捷，并且不会损坏 NAND Flash 中的原有数据，本文档所涉及的所有的程序都将统一采用这种方式进行烧写和运行。

第四节　实验现象

可以看到开发板上的 4 个 LED 同时不停地闪烁。

第三章　设置栈和 C 语言点亮 LED

第一节　为什么调用 C 函数要设置栈

栈有三个作用，包括：保存现场；传递参数:汇编代码调用 C 函数时，需传递参数；保存临时变量:包括函数的非静态局部变量以及编译器自动生成的其他临时变量；

1） 保存现场

现场，意思就相当于案发现场，总有一些现场的情况，要记录下来的，否则被别人破坏掉之后，你就无法恢复现场了。而此处说的现场，就是指 CPU 运行的时候，用到了一些寄存器，比如 r0,r1 等等，对于这些寄存器的值，如果你不保存而直接跳转到子函数中去执行，那么很可能就被其破坏了，因为其函数执行也要用到这些寄存器。因此，在函数调用之前，应该将这些寄存器等现场，暂时保持起来(入栈 push)，等调用函数执行完毕返回后(出栈 pop)，再恢复现场。这样 CPU 就可以正确的继续执行了。保存寄存器的值，一般用的是 push 指令，将对应的某些寄存器的值，一个个放到栈中，把对应的值压入到栈里面，即所谓的压栈。然后待被调用的子函数执行完毕的时候，再调用 pop，把栈中的一个个的值，赋值给对应的那些你刚开始压栈时用到的寄存器，把对应的值从栈中弹出去，即所谓的出栈。其中保存的寄存器中，也包括 lr 的值（因为用 bl 指令进行跳转的话，那么之前的 PC 的值是存在 lr 中的），然后在子程序执行完毕的时候，再把栈中的 lr 的值 pop 出来，赋值给 PC，这样就实现了子函数的正确的返回。

2） 传递参数

C 语言进行函数调用的时候，常常会传递给被调用的函数一些参数，对于这些 C 语言级别的参数，被编译器翻译成汇编语言的时候，就要找个地方存放一下，并且让被调用的函数能够访问，否则就没发实现传递参数了。对于找个地方放一下，分两种情况。一种情况是，本身传递的参数不多于 4 个，就可以通过寄存器 r0~r3 传送参数。因为在前面的保存现场的动作中，已经保存好了对应的寄存器的值，那么此时，这些寄存器就是空闲的，可以供我们使用的了，那就可以放参数。另一种情况是，参数多于 4 个时，寄存器不够用，就得用栈了。

3） 临时变量保存在栈中

包括函数的非静态局部变量以及编译器自动生成的其他临时变量。

第二节　程序相关讲解

完整代码见目录 2.led_c_sp 。

1. start.S

本章我们将使用 C 函数来实现点灯和延时的功能。在代码 2.led_c_sp 中，start.S 的作用如下：

1） 关闭看门狗；
2） 设置栈，其实就是设置 SP 寄存器，让其指向一块可用的内存。SD 启动时，S3C2451 的内部 8K 的 SRAM 被映射到 0x40000000,而 ARM 默认的栈是递减的,所以可以让 SP 指向 0x40002000；
3） 调用 C 函数 main(),实现 LED 闪烁；

2. main.c

main.c 中编写了两个 C 函数，main()实现 LED 闪烁，delay()实现延时的功能，代码如下：

```
void delay()
{
    volatile int i = 0x100000;
    while (i--);
}

int main()
{
    // 配置引脚
    volatile unsigned long *gpacon = (volatile unsigned long *)0x56000000;
    volatile unsigned long *gpbcon = (volatile unsigned long *)0x56000010;
    volatile unsigned long *gpadat = (volatile unsigned long *)0x56000004;
    volatile unsigned long *gpbdat = (volatile unsigned long *)0x56000014;
    volatile unsigned long *gpbsel = (volatile unsigned long *)0x5600001c;

    *gpbsel = 0;
    *gpacon = 0;
    *gpbcon = ( 0x5<<(2*5) );

    while (1)
    {
        *gpadat = (0x3<<25);                    // LED 灭
```

```
            *gpbdat = (0x3<<5);
            delay();
            *gpadat = 0;                        // LED 亮
            *gpbdat = 0;
            delay();
        }
        return 0;
    }
```

显然，相比起汇编，使用 C 语言来操作 LED 显得简洁容易了许多，后面的代码我们尽可能地使用 C 语言编写。

第三节　编译代码和烧写运行

编译代码

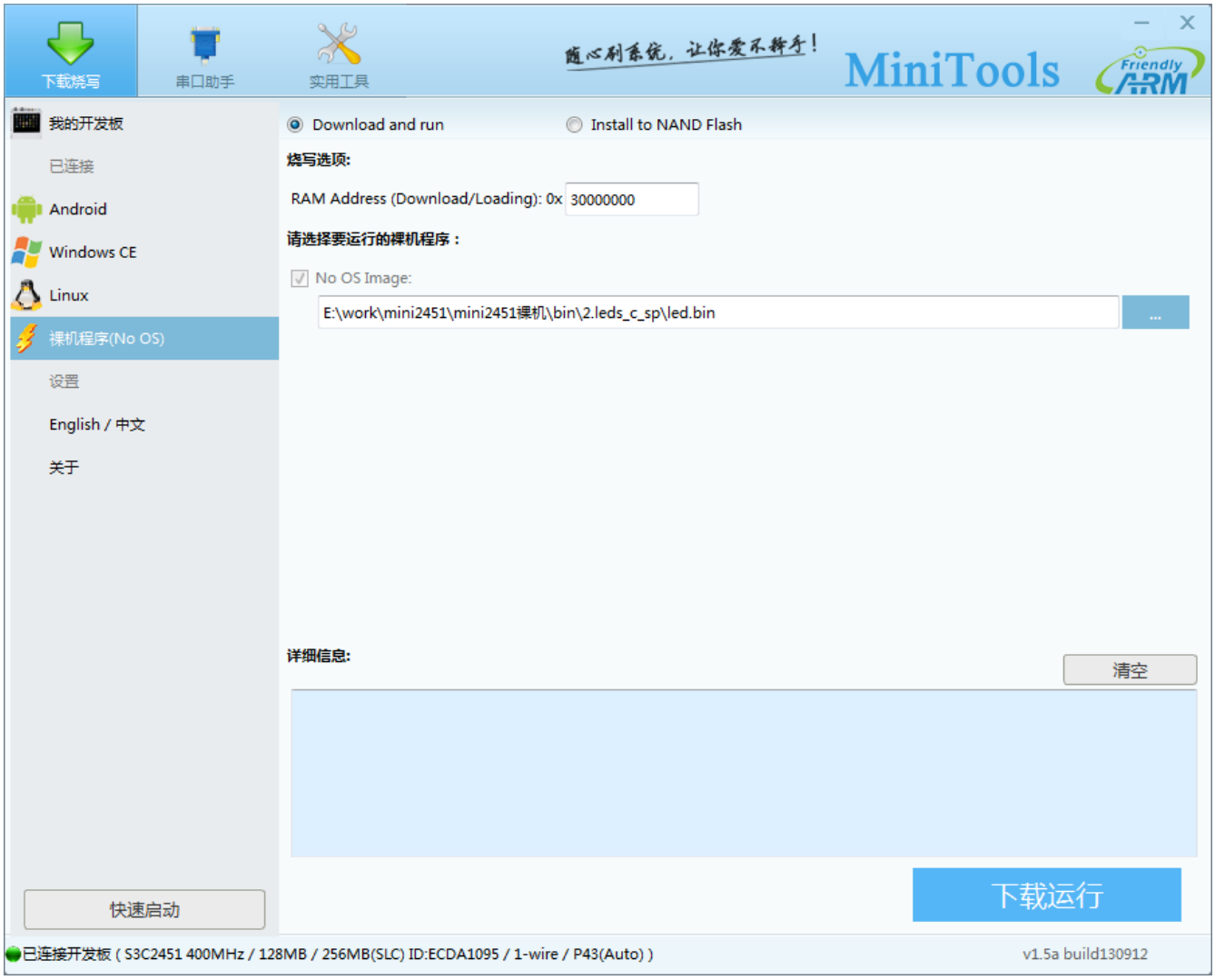
在 Fedora 终端执行如下命令：

```
# cd 2.led_c_sp
# make
```

make 成功后会生成 led.bin 文件。

烧写运行

使用 MiniTools 将 bin 文件烧写到开发板上的 DRAM，其设置方式如下：



先选中上方的”Download and run“，设置好下载地址“RAM Address(Download/Loading)”，选择要运行的裸机程序， 再点击“下载运行”就可以了。MiniTools 首先会把裸机程序下载到 DRAM 的地址 0x30000000 处，然后跳转到该地址上运行裸机程序，所以只要 PC 上一点击“下载运行”就可以马上看到开发板上裸机程序的运行效果了。

第四节　实验现象

可以看到 4 个 LED 同时不停地闪烁。设置栈以后，我们的程序就能调用 C 函数，这样将大大提升我们编写代码的速度，后面的代码我们将尽可能使用 C 语言编写。

第四章　C 语言中调用汇编函数

第一节　为什么要在 C 语言中调用汇编函数

1) 汇编执行的代码效率更高；
2) 某些操作使用汇编编写代码更方便，如对协处理器的操作；

第二节　程序相关讲解

完整代码见目录 3.led_c_call_s。

1. start.S

本章我们将使用 C 函数来实现点灯和使用汇编实现延时的功能。在代码 3.led_c_call_s 中，start.S 的作用和上一章一样，但是多了一个汇编编写的延时函数，函数名叫 delay()，代码如下：

```
delay:
delay_loop:
    cmp r0, #02
    sub r0, r0, #1
    bne delay_loop
    mov pc, lr
```

2. main.c

main.c 中的 main()函数实现了流水灯的功能，其中延时部分的代码调用的是 start.S 中的 delay()函数，参数通过 r0 传递，原理和汇编调用 C 函数是一样的，具体代码如下：

```
void delay(int count);          // 函数声明
int main()
{
    // 配置引脚
    volatile unsigned long *gpacon = (volatile unsigned long *)0x56000000;
    volatile unsigned long *gpbcon = (volatile unsigned long *)0x56000010;
    volatile unsigned long *gpadat = (volatile unsigned long *)0x56000004;
    volatile unsigned long *gpbdat = (volatile unsigned long *)0x56000014;
    volatile unsigned long *gpbsel = (volatile unsigned long *)0x5600001c;
    *gpbsel = 0;
    *gpacon = 0;
    *gpbcon = ( 0x5<<(2*5) );
    while (1)
```

```
    {
        // LED 灭
        *gpadat = (0x3<<25);
        *gpbdat = (0x3<<5);
        delay(0x100000);
        // LED 亮
        *gpadat = 0;
        *gpbdat = 0;
        delay(0x100000);
    }
    return 0;
}
```

C 语言里调用汇编函数有两个需要注意的地方：

1) 汇编文件里需用“.global 函数名”导出函数，这样其他文件才能引用这个函数；

2) C 文件里调用汇编函数前需要先声明函数，如 void delay(int count);

第三节　编译代码和烧写运行

编译代码

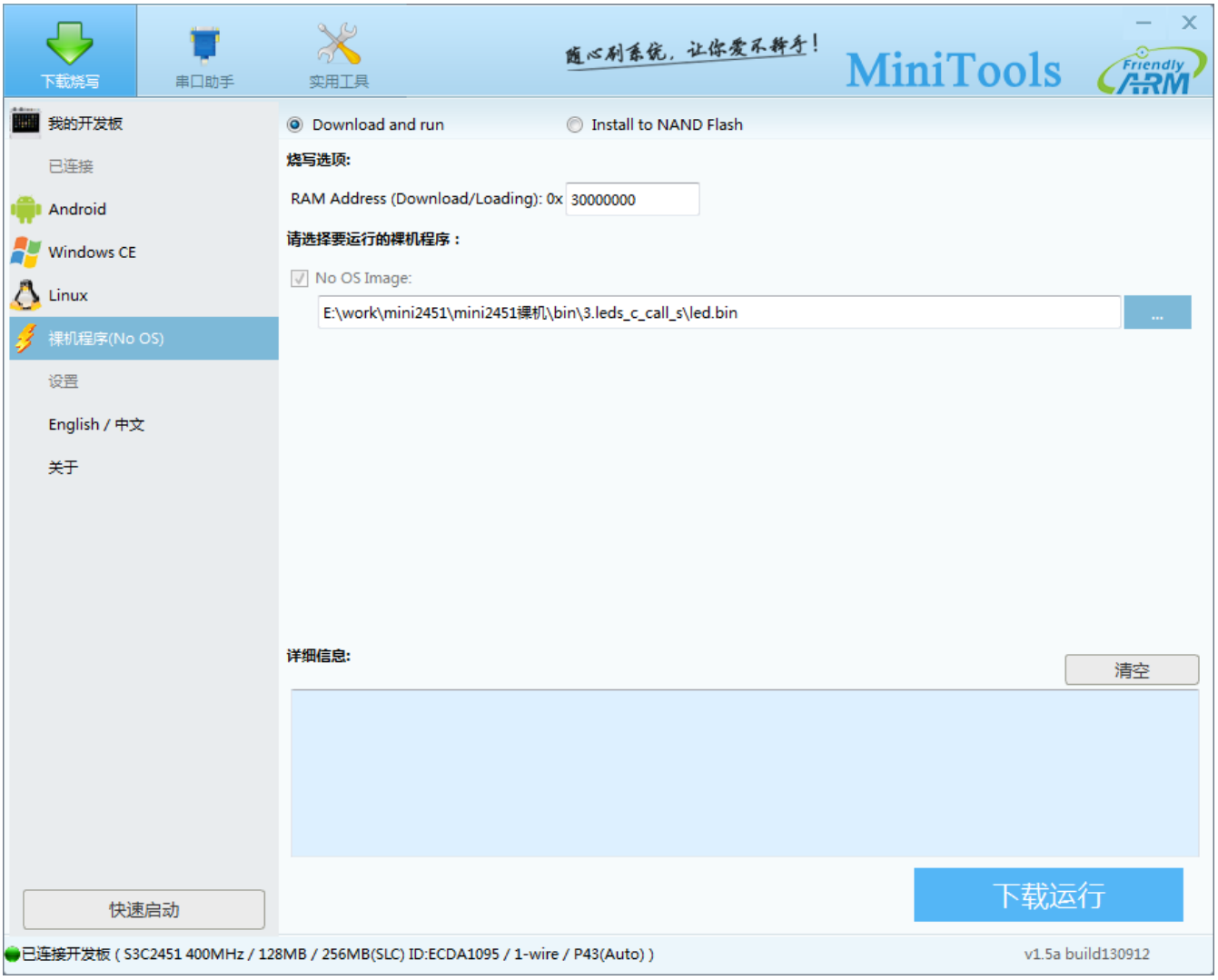
在 Fedora 终端执行如下命令：

```
# cd 3.leds_c_call_s
# make
```

make 成功后会生成 led.bin 文件。

烧写运行

使用 MiniTools 将 bin 文件烧写到开发板上的 DRAM，其设置方式如下：



先选中上方的”Download and run“，设置好下载地址“RAM Address(Download/Loading)”，选择要运行的裸机程序， 再点击“下载运行”就可以了。MiniTools 首先会把裸机程序下载到 DRAM 的地址 0x30000000 处，然后跳转到该地址上运行裸机程序，所以只要 PC 上一点击“下载运行”就可以马上看到开发板上裸机程序的运行效果了。

第四节　实验现象

代码的运行效果和上一章一样，都是 4 个 LED 同时闪烁，但是本章代码的 LED 闪烁明显比上一章的 LED 闪烁要快得多，这说明用汇编编写的代码比 C 语言编写的代码执行起来效率更高。

第五章　控制 icache

第一节　什么是 cache

基于程序访问的局限性，在主存和 CPU 通用寄存器之前设置了一类高速的、容量较小的存储器，把正在执行的指令地址附件的一部分指令或数据从主存调入这类存储器，供 CPU 在一段时间内使用，这对提高程序的运行速度有很大的作用。这类介于主存和 CPU 之间的高速小容量存储器称作高速 cache。

比较常见的 cache 包括 icache 和 dcache。icache 的使用比较简单，系统刚上电时，icache 中的内容是无效的，并且 icache 的功能是关闭的，往 CP15 协处理器中的寄存器 1 的 bit[12]写 1 可以启动 icache，写 0 可以停止 icache。icache 关闭时，CPU 每次取指都要读主存，性能非常低。因为 icache 可随时启动，越早开 icache 越好。

与 icache 相似，系统刚上电时，dcache 中的内容是无效的，并且 dcache 的功能是关闭的，往 CP15 协处理器中的寄存器 1 的 bit[2]写 1 可以启动 dcache，写 0 可以停止 dcache。因为 dcache 必须在启动 mmu 后才能被启动，而对于裸机而言，没必要开 mmu，所以本教程的程序将不会启动 dcache。

第二节　程序相关讲解

完整代码见目录 4.led_c_icache。相比代码 2.leds_c_sp，代码 4.leds_c_icache 与其的唯一区别在于在 start.S 中打开了 icaches。

1. start.S

```
#ifdef CONFIG_SYS_ICACHE_OFF
bic r0, r0, #0x00001000                    // 关闭 icache
#else
orr r0, r0, #0x00001000                    // 打开 icache
#endif
mcr p15, 0, r0, c1, c0, 0
```

当没有定义 CONFIG_SYS_ICACHE_OFF 时启动 icache，为 1 时关闭 icache。至于协处理器的相关指令，需查阅 s3c2410 的芯片手册或者《arm 体系结构与编程》一书。

第三节　编译代码和烧写运行

编译代码

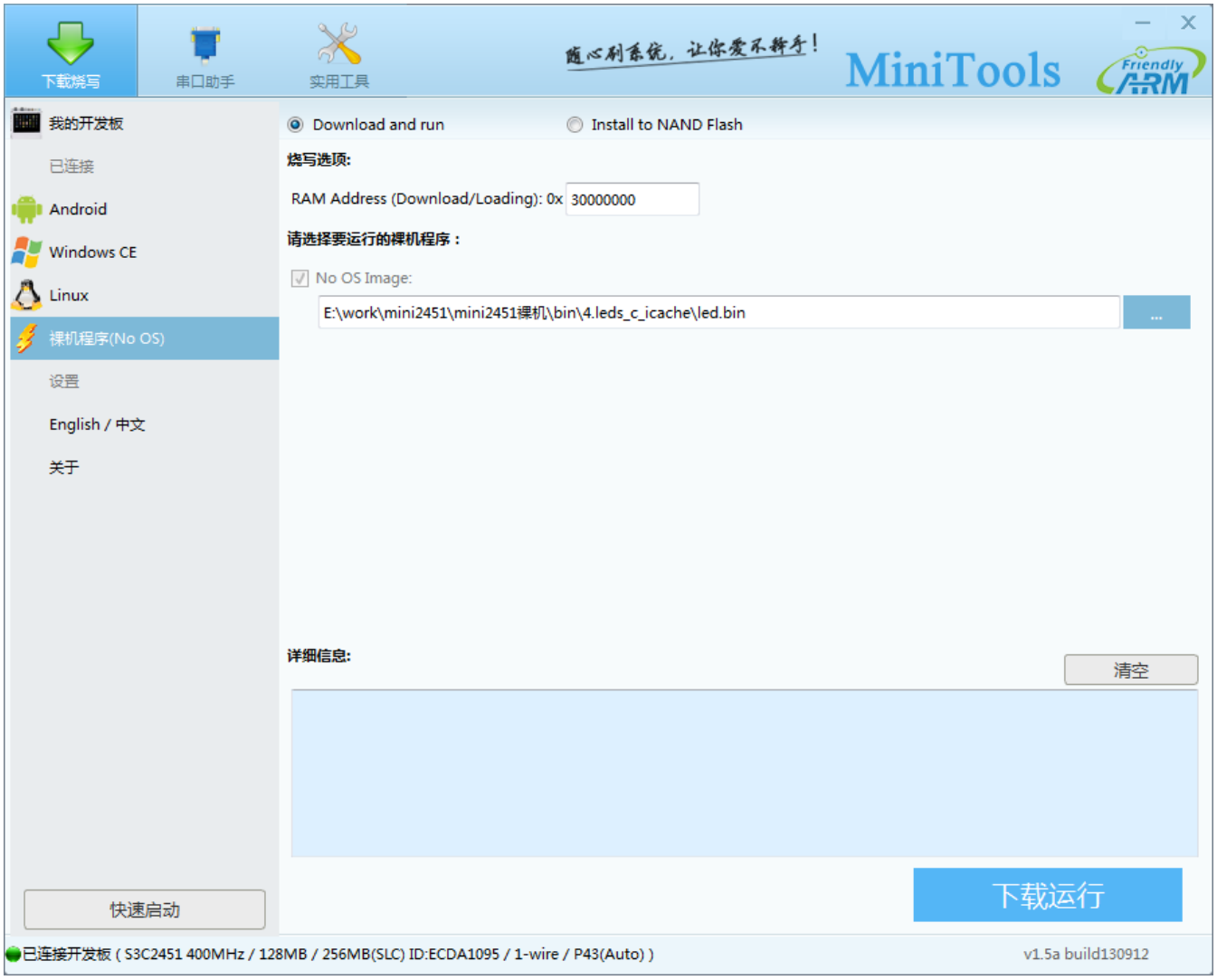
在 Fedora 终端执行如下命令：

```
# cd 4.leds_c_icache
# make
```

make 成功后会生成 led.bin 文件。

烧写运行

使用 MiniTools 将 bin 文件烧写到开发板上的 DRAM，其设置方式如下：



先选中上方的”Download and run“，设置好下载地址“RAM Address(Download/Loading)”，选择要运行的裸机程序， 再点击“下载运行”就可以了。MiniTools 首先会把裸机程序下载到 DRAM 的地址 0x30000000 处，然后跳转到该地址上运行裸机程序，所以只要 PC 上一点击“下载运行”就可以马上看到开发板上裸机程序的运行效果了。

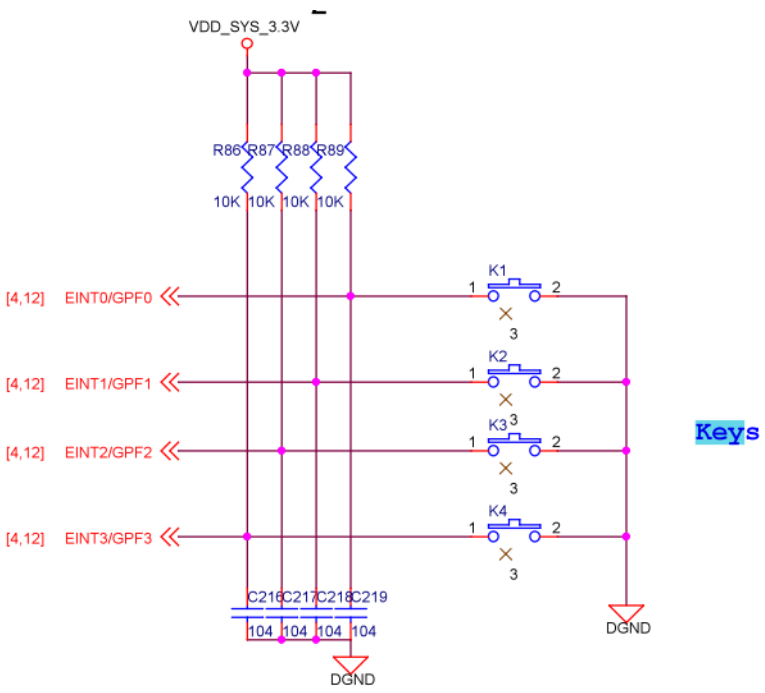
第四节　实验现象

相比程序 2.leds_c_sp， LED 闪烁变快了好几倍。由于 ichace 可以提高 CPU 的取指速度以及可以随时打开，所以程序里应该尽早的打开 icache。

第六章　查询方式检测按键

第一节　查看原理图

Mini2451 中共有 4 个用户按键，原理图如下：



按键原理图

第二节　程序相关讲解

完整代码见目录 5.key_led。启动代码与前面无异，主要区别在于 main.c。

1. main.c

代码如下：

```
void main(void)
{
    int dat = 0;
    // 所有 LED 熄灭
    GPACON = 0x0;
    GPBCON = ( 0x5<<(2*5) );
    GPADAT = (0x3<<25);
    GPBDAT = (0x3<<5);
```

```
    GPBSEL = 0x0;
    // 配置 GPF 引脚为输入功能
    GPFCON = 0;
    // 轮询的方式查询按键事件
    while(1)
    {
        dat = GPFDAT;
        if(dat & (1<<0))            // KEY1 被按下，则 LED1 亮，否则 LED1 灭
            GPBDAT |= 1<<5;
        else
            GPBDAT &= ~(1<<5);
        if(dat & (1<<1))            // KEY2 被按下，则 LED2 亮，否则 LED2 灭
            GPBDAT |= 1<<6;
        else
            GPBDAT &= ~(1<<6);
        if(dat & (1<<2))            // KEY3 被按下，则 LED3 亮，否则 LED3 灭
            GPADAT |= (1<<25);
        else
            GPADAT &= ~(1<<25);
        if(dat & (1<<3))            // KEY4 被按下，则 LED4 亮，否则 LED4 灭
            GPADAT |= 1<<26;
        else
            GPADAT &= ~(1<<26);
    }
}
```

程序很简单，首先配置 GPB/A 引脚为输出功能(输出低电平 LED 亮)以及配置 GPF 引脚为输入功能(按键被按下时，相应的引脚值为 1)，然后使用轮询的方式不断的读 GPF 引脚，当某个按键被按下时，对应的 GPF 引脚为高，此时我们点亮对应的 LED，否者让 LED 保持熄灭的状态。

第三节　编译代码和烧写运行

编译代码

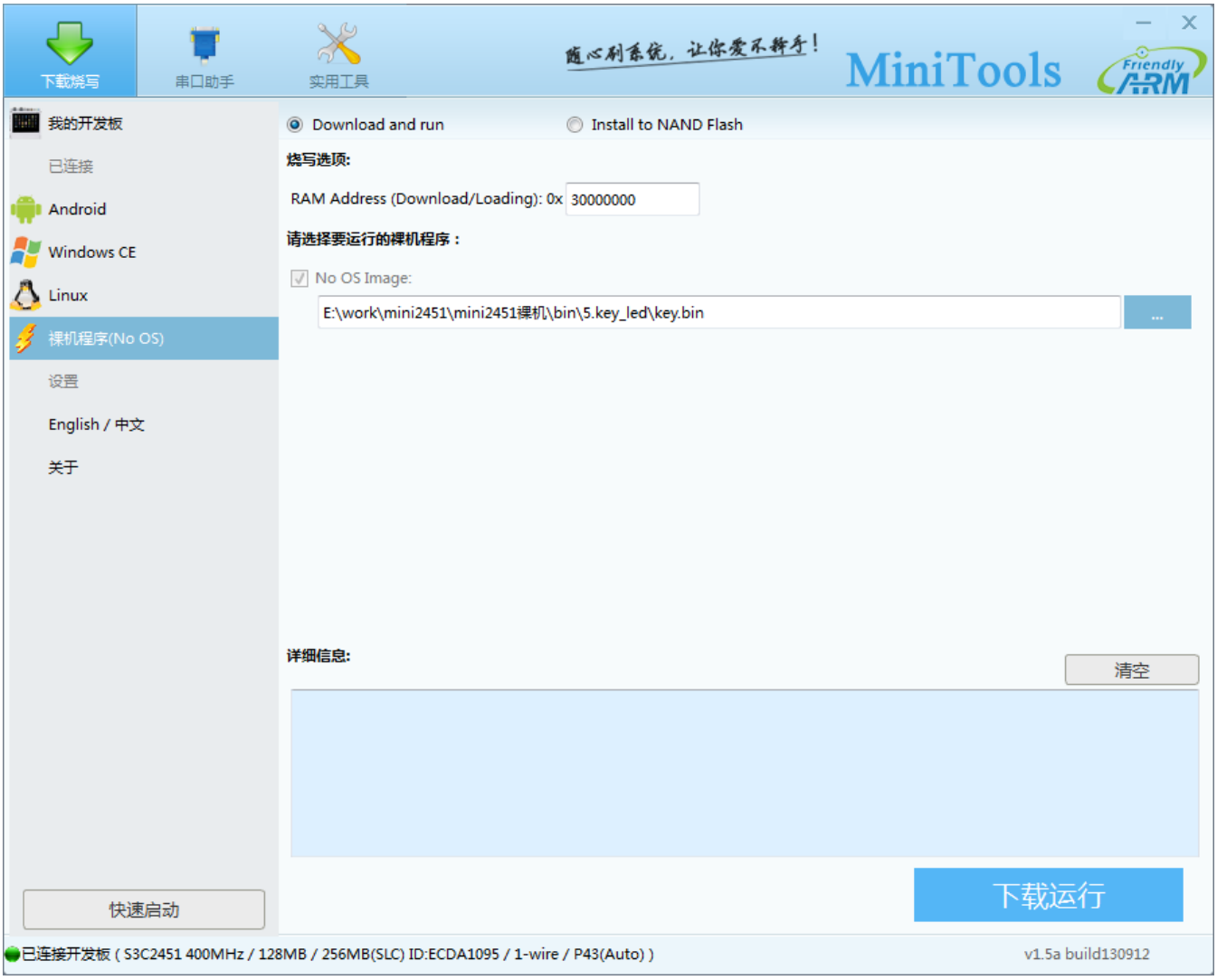
在 Fedora 终端执行如下命令：

```
# cd 5.key_led
# make
```

make 成功后会生成 key.bin 文件。

烧写运行

使用 MiniTools 将 bin 文件烧写到开发板上的 DRAM，其设置方式如下：



先选中上方的”Download and run“，设置好下载地址“RAM Address(Download/Loading)”，选择要运行的裸机程序， 再点击“下载运行”就可以了。MiniTools 首先会把裸机程序下载到 DRAM 的地址 0x30000000 处，然后跳转到该地址上运行裸机程序，所以只要 PC 上一点击“下载运行”就可以马上看到开发板上裸机程序的运行效果了。

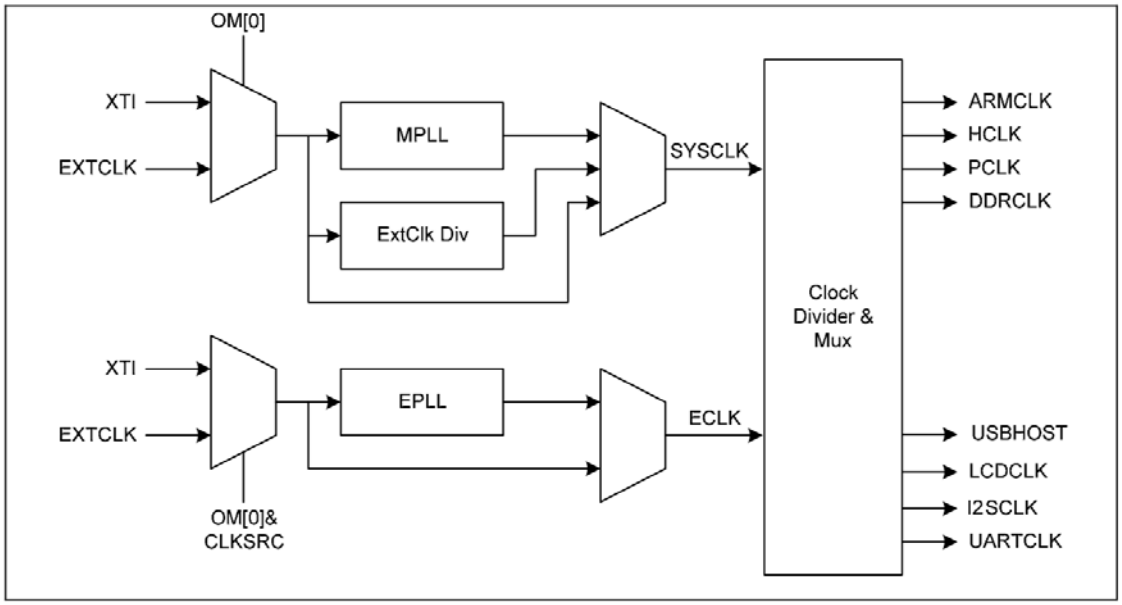
第四节　实验现象

未按下任何按键时，所有 LED 保持熄灭状态。当按下按键 KEY1 时，LED1 被点亮，松开按键 KEY1 时，LED1 熄灭。对于其余 3 个按键，也是相同情况。用查询的方式来检测按键太占 CPU 使用率了，除了检测按键，CPU 无法进行其他工作，后面的学习中断的知识后，我们可以使用中断的方式来检测按键中断，这将大大降低 CPU 的使用率。

第七章　初始化时钟

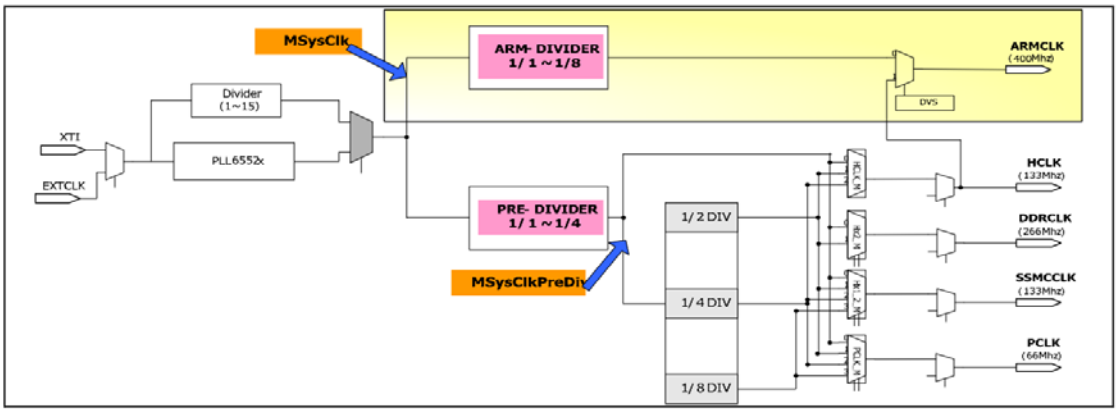
第一节　S3C2451 时钟体系

在 S3C2451 中生成所需的系统时钟信号，包括 CPU 的 ARMCLK，AXI/AHB 总线外设的 HCLK 和 APB 总线外设的 PCLK 等。在 S3C2451 中有 2 个 PLL。MPLL 用于产生 ARMCLK、HCLK、PCLK、DDRCLK 和 SSMCLK 的时钟，EPLL 用于产生 USBHOSTCLK 的时钟。每个外设块的时钟信号可能被启用或禁用，由软件控制以减少电源消耗。具体可以参考下图：



S3C2451 时钟设置参考图

对于简单的裸机程序，基本上只需要设置好 MPLL 就可以满足大多数模块的时钟要求了，具体如何设置上述各种各样的时钟，可参考下图：



MPLL 设置参考图

该图十分重要，依据上图我们就可以设置好大多数硬件部件所需的工作时钟，实际上我们并不需要设置好所有部件的工作时钟，我们只需设置好我们需要使用的硬件部件的工作时钟即可，在本章第二节中，我们将以上图为基础，通过设置时钟相关的寄存器，达到初始化时钟的目的。

第二节　程序相关讲解

完整代码见目录 6.clock_s 和 7.clock_c。本章涉及的代码有两套，包括 6.clock_s(使用汇编初始化时钟)和 7.clock_c(使用 c 语言初始化时钟)，两者的本质是一样的，初始化时钟所达到的效果也一样，想巩固汇编指令的话，可以阅读 6.clock_s 里的相关代码。由于 C 语言编写的代码思路更清晰，更便于阅读和理解，下面将以 7.clock_c 里的代码为参考进行讲解。

1. start.S

在调用 main 函数之前，调用了时钟初始化函数 clock_init，进行时钟相关的设置。

2. clock.c

时钟初始化函数 clock_init()

```
void clock_init(void)
{
    // 设置分频
    // ARMCLK = MPLLout/ARMCLK Ratio = MPLLout/(ARMDIV+1) = 800/2 = 400MHz
    // HCLK = MPLLout/HCLK Ratio = MPLLout/( (PREDIV+1) * (HCLKDIV + 1) ) = 800/(3*2)
= 133MHz
    // PCLK = HCLK/PCLKDIV = 133/2 = 66MHz
    CLKDIV0    = (CLKDIV0&~(0x1E37)) | CLK_DIV_VAL;

    // 设置 locktime
    LOCKCON0 = 0xffff;

    // 设置 MPLL
    MPLLCON = MPLL_VAL;

    // 设置时钟源
    CLKSRC |= 1<<4;
}
```

上述代码共有 4 个步骤，下面我们来一一讲解每一个步骤：

1) 设置分频

与分频相关的寄存器是 CLK_DIV0，见下图：

CLKDIV0	Bit	Description	Initial Value
RESERVED	[31:14]	-	*0x0*
DVS	[13]	Enable/disable DVS (Dynamic Voltage Scaling) feature 0 = Disable 1 = Enable (The frequency of ARMCLK is the same frequency of HCLK regardless of ARMDIV field.)	0
RESERVED	[12]	-	0
ARMDIV	[11:9]	ARM clock divider ratio ARMDIV values are recommended as below. 1/1 = 3'b000 1/2 = 3'b001 1/3 = 3'b010 1/4 = 3'b011 1/6 = 3'b101 1/8 = 3'b111	0x0
EXTDIV	[8:6]	External clock divider ratio ratio = (MPLL reference clock) / (EXTDIV*2 + 1)	0
PREDIV	[5:4]	Pre Divider for HCLK PREDIV value should be one of 0,1,2,3 Output frequency of PREDIVIDER should be less than 266MHz	0
HALFHCLK	[3]	HCLKx1_2(SSMC) clock divider ratio, 0 = HCLK, 1 = HCLK/2 User also has to configure SSMC's special register which related with half clock.	1
PCLKDIV	[2]	PCLK clock divider ratio, 0 = HCLK, 1 = HCLK / 2	1
HCLKDIV	[1:0]	HCLK clock divider ratio HCLKDIV value should be one of 0,1,3. (2'b10 is invalid)	0x0

芯片手册里给出了下列参考值，我们就根据这些参考值来设置该寄存器。

When PLL output frequency = 800MHz

Target frqeuency

ARMCLK = 400MHz, HCLK = 133MHz, PCLK = 66MHz, DDRCLK = 266MHz
SSMCCLK = 66MHz

Register value

ARMDIV = 4'b0001, PREDIV = 2'b10, HCLKDIV = 2'b01, PCLKDIV = 1'b1
HALKHCLK = 1'b1

- ARMCLK = 400MHZ = MPLLout/ARMCLK Ratio = MPLLout/(ARMDIV+1),经过在第四和第五步的设置后，MPLLout 会被设置为 800MHz，所以 ARMDIV = 1；
- HCLK = 133MHz = MPLLout/HCLK Ratio = MPLLout/((PREDIV+1) * (HCLKDIV + 1))，经过在第四和第五步的设置后 MOUTMPLL = 800MHz，所以 PREDIV = 2，HCLKDIV = 1；
- PCLK = 66MHz = HCLK/PCLKDIV，所以 PCLKDIV = 2；

2） 设置锁定时间

设置 PLL 后，时钟从 Fin 提升到目标频率时，需要一定的时间，即锁定时间。我们设置为最大值。

3） 设置 PLL

设置好分频和锁定时间后，我们就需要设置 PLL 了。MPLL 的启动是通过设置 MPLLCON 寄存器：

MPLLCON	Bit	Description	Initial Value
RESERVED	[31:26]	-	0x00
MPLLEN_STOP	[25]	MPLL ON/OFF in STOP mode. 0:OFF, 1:ON	0
ONOFF	[24]	MPLL ON/OFF. 0:ON, 1:OFF	1
MDIV	[23:14]	Main divider value of MPLL	0x215
RESERVED	[13:11]	-	0x0
PDIV	[10:5]	Pre-divider value of MPLL	0x6
RESERVED	[4:3]	-	0x0
SDIV	[2:0]	Post-divider value of MPLL	0x0

我们将 MPLL 的输出设置为 800MHz，由于 FOUT = MDIV * FIN / (PDIV * 2^SDIV),参考下面这个表格，我们取 MDIV=400,PDIV=3,SDIV=1。

FIN (MHz)	Target FOUT (MHz)	MDIV (decimal)	PDIV (decimal)	SDIV (decimal)	Duty
12	240	320	4	2	40~60%
12	400	400	3	2	40~60%
12	450	225	3	1	40~60%
12	500	250	3	1	40~60%
12	534	267	3	1	40~60%
12	600	300	3	1	40~60%
12	800	400	3	1	40~60%

4） 设置各种时钟开关

相关寄存器是 CLKSRC,见下图：

CLKSRC	Bit	Description	Initial Value
RESERVED	[31:19]	-	0x0_0000
SELHSSPI0	[18]	HS-SPI0 clock 0 = EPLL (divided), 1 = MPLL (divided)	0
SELHSMMC1	[17]	HSMMC1 clock 0 = EPLL (divided), 1 = EXTCLK	0
SELHSMMC0	[16]	HSMMC0 clock 0 = EPLL (divided), 1 = EXTCLK	0
RESERVED	[15:9]	-	0
SELESRC	[8:7]	Selection EPLL reference clock 10 = XTAL, 11 = EXTCLK 0x = identical to that of MPLL reference clock Do not configure SELESRC & SELEPLL register simultaneously.	00
SELEPLL	[6]	EsysClk selection 0 = EPLL reference clock, 1 = EPLL output	0
RESERVED	[5]	-	0
SELMPLL	[4]	MSYSCLK selection 0 = MPLL reference clock (produced through clock divider) 1 = MPLL output	0
SELEXTCLK	[3]	Configure MPLL reference clock divider 0 = don't use MPLL reference clock divider (means 1/1 divide ratio) 1 = use MPLL reference clock divider (See EXTDIV field of CLKDIV)	0
RESERVED	[2:0]	-	0x0

SELMPLL=1，使用 MPLL output 作为 MSYSCLK 的输入，其余位都使用默认值。

3. main.c

在 main 函数中实现 LED 闪烁的功能，与前面章节的代码大同小异。

第三节　编译代码和烧写运行

编译代码

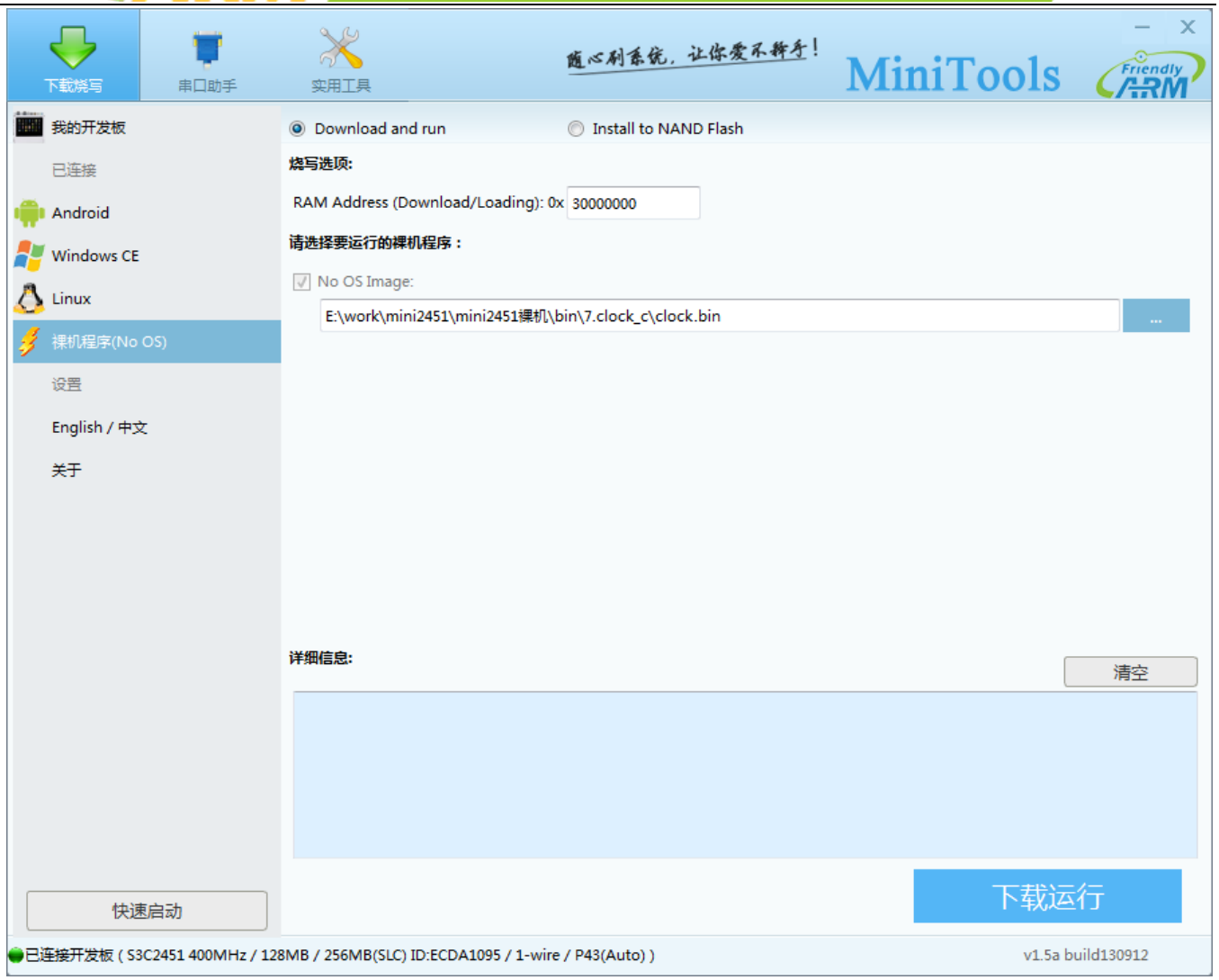
在 Fedora 终端执行如下命令：

```
# cd 7.clock_c
# make
```

make 成功后会生成 clock.bin 文件。

烧写运行

使用 MiniTools 将 bin 文件烧写到开发板上的 DRAM，其设置方式如下：



先选中上方的”Download and run“，设置好下载地址“RAM Address(Download/Loading)”，选择要运行的裸机程序， 再点击“下载运行”就可以了。MiniTools 首先会把裸机程序下载到 DRAM 的地址 0x30000000 处，然后跳转到该地址上运行裸机程序，所以只要 PC 上一点击“下载运行”就可以马上看到开发板上裸机程序的运行效果了。

第四节　实验现象

初始化时钟后，闪烁的频率并没有大幅度提升，这是因为 Superboot 已经帮我们初始化过时钟了。如果我们想体验一下不初始化系统时钟时开发板的运行速度的话，可以通过在 clock.c 中定义宏 PLL_OFF 从而关闭 PLL 的功能，这样 LED 闪烁的频率将大大降低。本章的时钟设置主要是针对 MPLL，这足以让大部分硬件正常工作了，下一章我们将初始化 UART 以达到在终端输入输出字符的目的，这将使用到我们本章所设置好的时钟。

第八章　串口设置之输入输出字符

第一节　S3C2451 UART 相关说明

通用异步收发器简称 UART，即 UNIVERSAL ASYNCHRONOUS RECEIVER AND TRANSMITTER，它用来传输串行数据。发送数据时，CPU 将并行数据写入 UART，UART 按照一定的格式在一根电线上串行发出；接收数据时，UART 检测另一根电线的信号，将串行收集在缓冲区中，CPU 即可读取 UART 获得这些数据。S3C2451 的 UART 提供了四个独立的异步串行 I/O 端口。每个异步串行 I/O 端口通过中断或者直接存储器存取（DMA）模式来操作。在 CPU 和 UART 之间传输数据的。该 UART 使用系统时钟的时间可以支持的比特率最高为 3Mb/s。每个 UART 的通道包含了两个 64 字节收发 FIFO 存储器。S3C2451 的 UART 结构图如下：

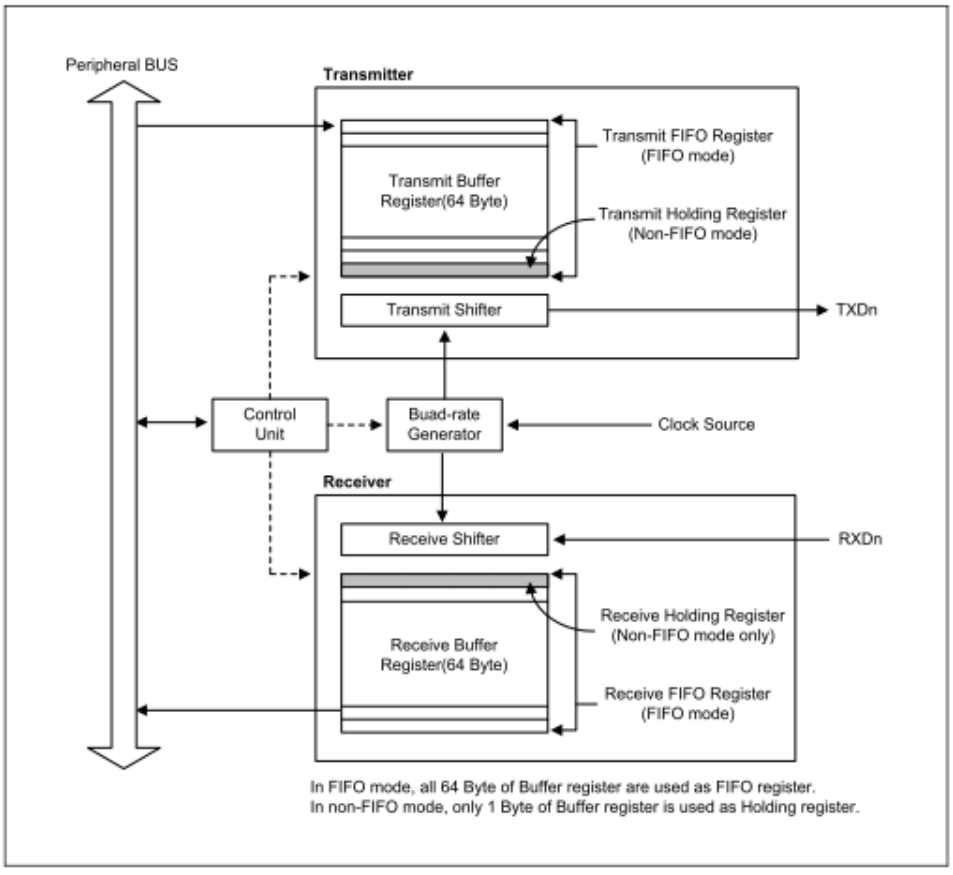
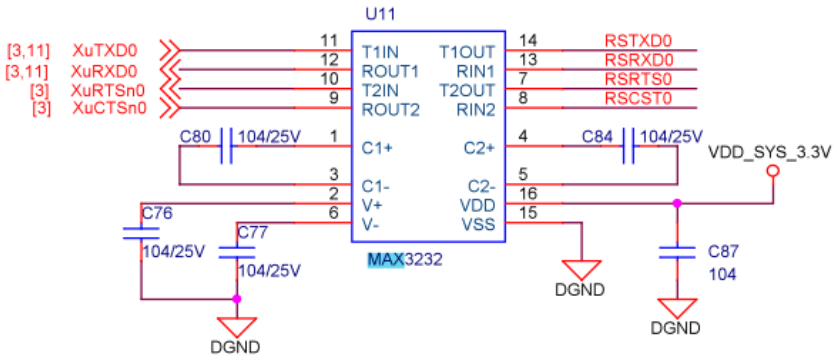


Figure 31-1. UART Block Diagram

S3C2451 的 UART 结构图

UART 使用标准的 TTL/CMCOS 逻辑电平来表示数据，为了增强数据抗干扰能力和提高传输长度，通常将 TTL/CMOS 逻辑电平转换为 RS-232 逻辑电平，查看原理图可知 Mini2451 使用的是 MAX3232SOP 芯片，使用的是相关引脚是 TX0 和 DX0：



RS-232

通过设置 UART 相关寄存器，我们就可以驱动 UART 工作，达到发送和接收字符的目的。

第二节　程序相关讲解

完整代码见目录 8.uart_putchar,对比前一个目录 7.clock_c，区别在于 main.c 和多了一个 uart.c 文件。

1. main.c

完整代码如下：

```
int main()
{
    char c;
    uart_init();          // 初始化串口
    while (1)
    {
        c = getc ();          // 接收一个字符 c
        putc(c+1);            // 发送字符 c+1
    }
    return 0;
}
```

在 start.S 中初始化时钟后执行 bl main 跳转到 main 函数，在 main 函数中，先会调用 uart_init()初始化 UART，然后使用 getchar()接收字符，再调用 putchar()将该字符+1 发送出去。

2. uart.c

uart_init()代码如下：

```
void uart_init(void)
{
    /* 配置引脚 */
    GPHCON = (GPHCON & ~0xffff ) | 0xaaaa;

    /* 设置数据格式等 */
    ULCON0 = 0x3;                    // 数据位:8, 无校验, 停止位: 1, 8n1
    UCON0  = 0x5;                    // 时钟：PCLK，禁止中断，使能 UART 发送、接收
    UFCON0 = 0x01;                   // FIFO ENABLE
    UMCON0 = 0;                      // 无流控

    /* 设置波特率 */
    // DIV_VAL = (PCLK / (bps x 16 ) ) - 1 = (66500000/(115200x16))-1 = 35.08
    // DIV_VAL = 35.08 = UBRDIVn + (num of 1’s in UDIVSLOTn)/16
    UBRDIV0   = 35;
    UDIVSLOT0 = 0x1;
}
```

上述代码共有 3 个步骤，下面我们来一一讲解每一个步骤：

1） 配置引脚用于 RX/TX 功能

参考 UART 引脚连接图，我们需要设置寄存器 GPHCON 使 GPH 引脚用于 UART 功能。

GPHCON	Bit	Description
Reserved	[31:30]	Reserved
GPH14	[29:28]	00 = Input　01 = Output 10 = CLKOUT1　11 = Reserved
GPH13	[27:26]	00 = Input　01 = Output 10 = CLKOUT0　11 = Reserved
GPH12	[25:24]	00 = Input　01 = Output 10 = EXTUARTCLK　11 = Reserved
GPH11	[23:22]	00 = Input　01 = Output 10 = nRTS1　11 = Reserved
GPH10	[21:20]	00 = Input　01 = Output 10 = nCTS1　11 = Reserved
GPH9	[19:18]	00 = Input　01 = Output 10 = nRTS0　11 = Reserved
GPH8	[17:16]	00 = Input　01 = Output 10 = nCTS0　11 = Reserved
GPH7	[15:14]	00 = Input　01 = Output 10 = RXD[3]　11 = nCTS2
GPH6	[13:12]	00 = Input　01 = Output 10 = TXD[3]　11 = nRTS2
GPH5	[11:10]	00 = Input　01 = Output 10 = RXD[2]　11 = Reserved
GPH4	[9:8]	00 = Input　01 = Output 10 = TXD[2]　11 = Reserved
GPH3	[7:6]	00 = Input　01 = Output 10 = RXD[1]　11 = reserved
GPH2	[5:4]	00 = Input　01 = Output 10 = TXD[1]　11 = Reserved
GPH1	[3:2]	00 = Input　01 = Output 10 = RXD[0]　11 = Reserved
GPH0	[1:0]	00 = Input　01 = Output 10 = TXD[0]　11 = Reserved

GPHCON 寄存器图

2） 设置数据格式等

寄存器 ULCON0 用来设置数据格式，见下图:

ULCONn	Bit	Description	Initial State
Reserved	[7]	–	0
Infrared Mode	[6]	Determine whether or not to use the Infrared mode. 0 = Normal mode operation 1 = Infrared Tx/Rx mode	0
Parity Mode	[5:3]	Specify the type of parity generation and checking during UART transmit and receive operation. 0xx = No parity 100 = Odd parity 101 = Even parity 110 = Parity forced/checked as 1 111 = Parity forced/checked as 0	000
Number of Stop Bit	[2]	Specify how many stop bits are to be used for end-of-frame signal. 0 = One stop bit per frame 1 = Two stop bit per frame	0
Word Length	[1:0]	Indicate the number of data bits to be transmitted or received per frame. 00 = 5-bits　01 = 6-bits 10 = 7-bits　11 = 8-bits	00

- Word Length = 11,8bit 的数据；
- Number of Stop Bit = 0,1bit 的停止位；
- Parity Mode = 000，无校验；
- Infrared Mode =0，使用普通模式；

UCON0 是 UART 的配置寄存器，见下图

UCONn	Bit	Description	Initial State
Clock Selection	[11:10]	Select PCLK, EXTUARTCLK(External UART clock) or divided EPLL clock for source clock of the UART. (note 5) 00, 10 (note 1) = PCLK 01 = EXTUARTCLK 11 = Divided EPLL clock (Refer to the Clock divider control register1 (CLKDIV1) in the system controller).	0
Tx Interrupt Type	[9]	Interrupt request type. 0 = Pulse (Interrupt is requested as soon as the Tx buffer becomes empty in Non-FIFO mode or reaches Tx FIFO Trigger Level in FIFO mode.)	0
Rx Interrupt Type	[8]	Interrupt request type. 0 = Pulse (Interrupt is requested the instant Rx buffer receives the data in Non-FIFO mode or reaches Rx FIFO Trigger Level in FIFO mode.)	0
Rx Time Out Enable	[7]	Enable/Disable Rx time out interrupt when UART FIFO is enabled. The interrupt is a receive interrupt. (note 2) 0 = Disable　1 = Enable	0
Rx Error Status Interrupt Enable	[6]	Enable the UART to generate an interrupt upon an exception, such as a break, frame error, parity error, or overrun error during a receive operation. 0 = Do not generate receive error status interrupt. 1 = Generate receive error status interrupt.	0
Loopback Mode	[5]	Setting loopback bit to 1 causes the UART to enter the loopback mode. This mode is provided for test purposes only. 0 = Normal operation　1 = Loopback mode	0
Send break signal	[4]	Setting this bit causes the UART to send a break during 1 frame time. This bit is auto-cleared after sending the break signal 0 = Normal transmit　1 = Send break signal	0

UCONn	Bit	Description	Initial State
Transmit Mode (note 3)	[3:2]	Determine which function is currently able to write Tx data to the UART transmit buffer register. 00 = Disable 01 = Interrupt request (note 6) or polling mode 10 = DMA request(request signal 0) 11 = DMA request(request signal 1)	00
Receive Mode	[1:0]	Determine which function is currently able to read data from UART receive buffer register. 00 = Disable (note 4) 01 = Interrupt request or polling mode 10 = DMA request(request signal 0) 11 = DMA request(request signal 1)	00

- Receive Mode = 01 ，使用中断模式或者轮询模式；
- Transmit Mode = 01，使用中断模式或者轮询模式；
- Send Break Signal = 0，普通传输；
- Loop-back Mode = 0，不使用回环方式；
- 我们采用轮询的方式接受和发送数据，不使用中断，所以 bit[6-9]均为 0；
- Clock Selection = 0，使用 PCLK 作为 UART 的工作时钟；
- 我们不使用 DMA,所以 bit[16]和 bit[20]均为 0；

UFCON0 和 UMCON0
这两个寄存器比较简单，UFCON0 用来使能 FIFO，UMCON0 用来设置无流控。

3） 设置波特率

波特率即每秒传输的数据位数，涉及两个寄存器：UBRDIV0 和 UDIVSLOT0

UBRDIV n	Bit	Description	Initial State
UBRDIV	[15:0]	Baud rate division value (When UART clock source is PCLK, UBRDIVn must be more than 0 (UBRDIVn >0))	-

NOTE: If UBRDIV value is 0, UART baudrate is not affected by UDIVSLOT value.

UDIVSLOT n	Bit	Description	Initial State
Reserved	[31:16]	Reserved	0
UDIVSLOTn	[15:0]	Select the slot where clock generator divide clock source	0x0000

波特率设置相关公式：UBRDIVn + (num of 1's in UDIVSLOTn)/16 = (PCLK / (bps x 16)) -1。PCLK = 66.5MHz，我们的波特率 bps 设置为 115200,所以 (66.6MHz/(115200 x 16)) - 1 = 35.08 = UBRDIVn + (num of 1's in UDIVSLOTn)/16,所以我们设置 UBRDIV0=35，UDIVSLOT0=0x1。

另外，getchar()和 putchar()代码如下：

```
char getchar(void)                  // 接收一个字符
{
    while ((UFSTAT0 & 0x7f) == 0);  // 如果 RX FIFO 空，等待
    return URXH0;                   // 取数据
}

void putchar(char c)                //发送一个字符
{
    while (UFSTAT0 & (1<<14));      // 如果 TX FIFO 满，等待
    UTXH0 = c;                      // 写数据
}
```

UTXHn	Bit	Description	Initial State
TXDATAn	[7:0]	Transmit data for UARTn	-

UART 数据发送寄存器

URXHn	Bit	Description	Initial State
RXDATAn	[7:0]	Receive data for UARTn	0x00

UART 数据接收寄存器

UFSTATn	Bit	Description	Initial State
Reserved	[15]		0
Tx FIFO Full	[14]	Set to 1 automatically whenever transmit FIFO is full during transmit operation 0 = 0-byte ≤ Tx FIFO data ≤ 63-byte 1 = Full	0
Tx FIFO Count	[13:8]	Number of data in Tx FIFO	0
Reserved	[7]		0
Rx FIFO Full	[6]	Set to 1 automatically whenever receive FIFO is full during receive operation 0 = 0-byte ≤ Rx FIFO data ≤ 63-byte 1 = Full	0
Rx FIFO Count	[5:0]	Number of data in Rx FIFO	0

发送/接收状态寄存器

通过读 UTRSTAT0 发送/接收状态寄存器，当 Rx FIFO FULL 为 1 或者 Rx FIFO Count>0 时说明接收到数据，读 URXH0 寄存器可以得到 8bit 的数据；当如果 TX FIFO FULL=1 时，暂停发送数据，否则写 8bit 的数据到 UTXH0。

第三节　编译代码和烧写运行

编译代码

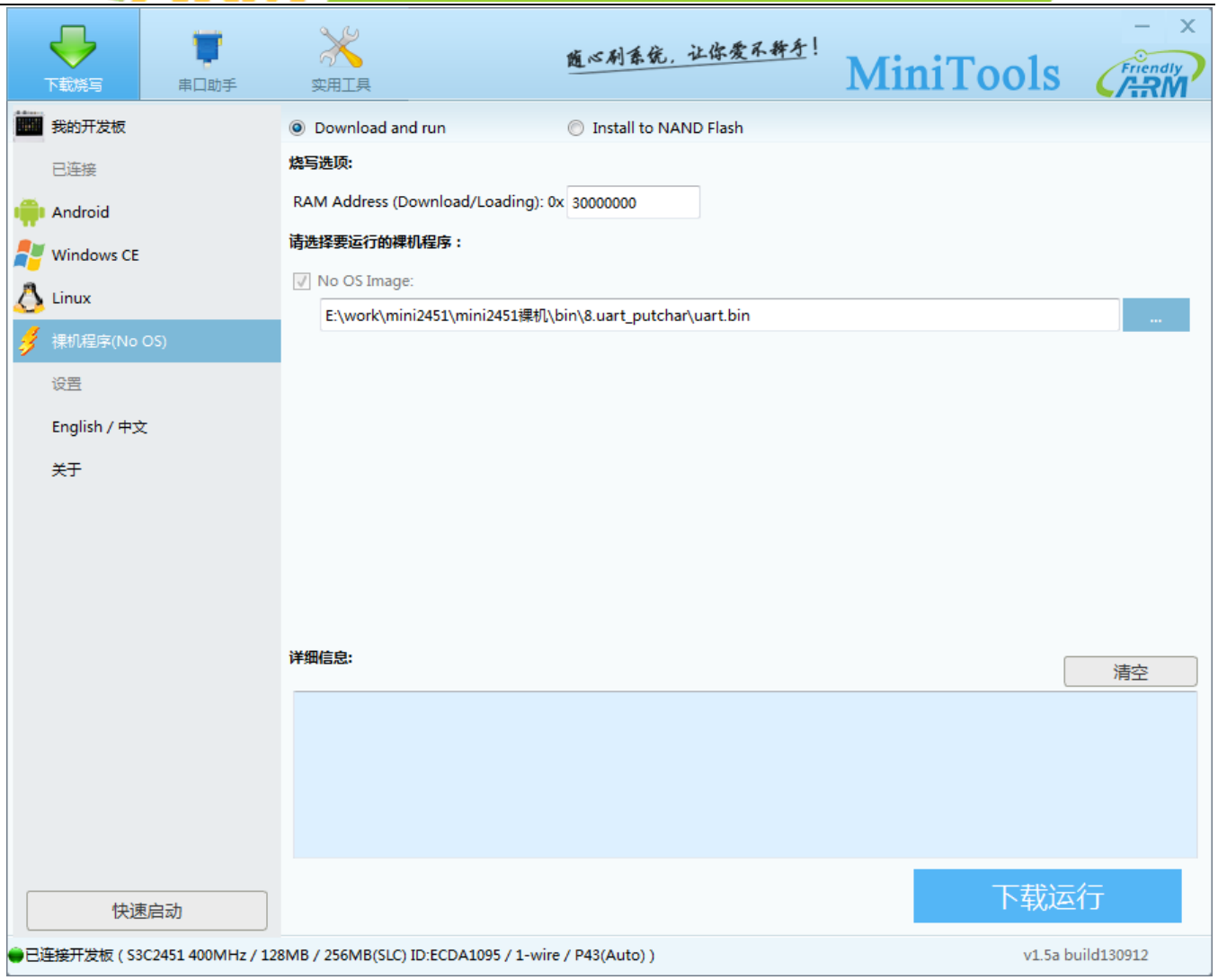
在 Fedora 终端执行如下命令：

```
# cd 8.uart_putchar
# make
```

make 成功后会生成 uart.bin 文件。

烧写运行

使用 MiniTools 将 bin 文件烧写到开发板上的 DRAM，其设置方式如下：



先选中上方的”Download and run“，设置好下载地址“RAM Address(Download/Loading)”，选择要运行的裸机程序， 再点击“下载运行”就可以了。MiniTools 首先会把裸机程序下载到 DRAM 的地址 0x30000000 处，然后跳转到该地址上运行裸机程序，所以只要 PC 上一点击“下载运行”就可以马上看到开发板上裸机程序的运行效果了。

第四节　实验现象

连接好串口终端后，在 PC 键盘上输入一个字符，则串口终端会显示该字符在 ASCII 表中的下一字符，例如键盘输入“abcd”，终端上会显示“bcde”，效果如下：

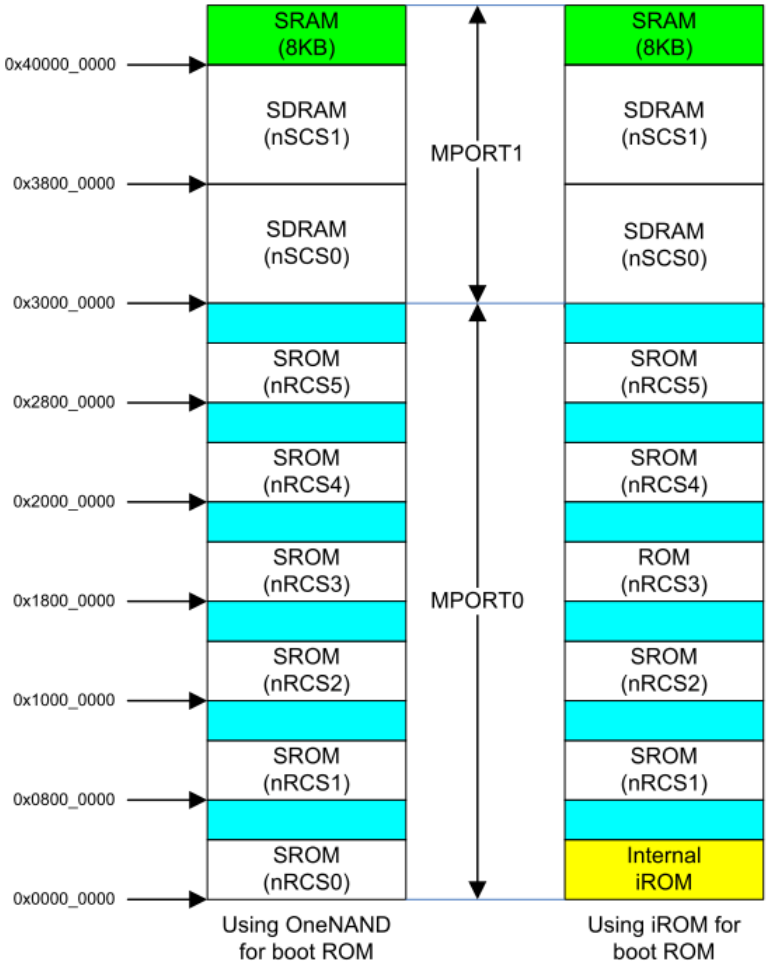
```
USB Mode: Waiting...
Hello USB Loop
USB Mode: Connected
Downloading User-Bin...
Downloading User-Bin succeed
Run UserBin
bcde
```

通过本章对 UART 的初始化，我们已经能在终端上打印字符了，这对我们调试代码有极大的帮助。但是目前代码的功能太单一，只能简单地输入或者输出一个字符，在后面的章节中，我们将为程序添加功能强大的 printf()和 scanf()功能。

第九章　S3C2451 的启动过程

第一节　IROM 和 SRAM

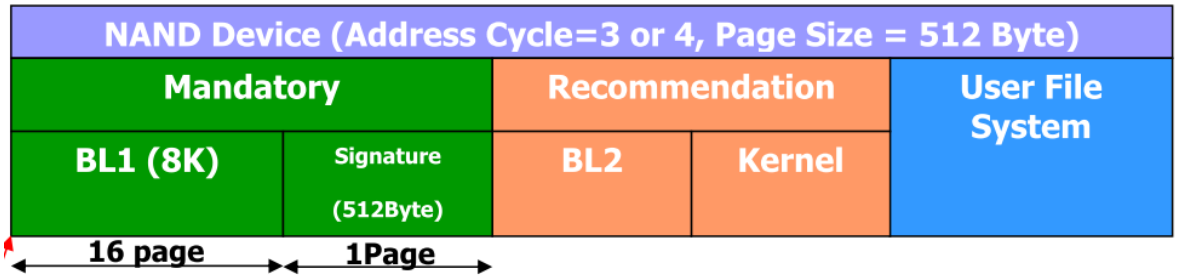
前面的程序我们都是利用 Minitools 和 Superboot 将程序直接下载到 DRAM 的，那么我们能不能抛弃各种 Bootloader 工具，直接把程序下载到 NANDFlash 或 SDHC 卡，开发板一上电就运行我们的程序？答案是肯定的，不过我们得先了解一下 S3C2451 的启动过程。S3C2451 支持两种启动方式，一是 OneNAND 启动，二是 IROM 启动。Mini2451 并没有 OneNAND，所以它使用的是第二种启动方式。S3C2451 含有一个 64K 的 IROM 和 8K 的 SRAM。IROM 和 SRAM 所处的存储空间见下图，可以看出，当采用 IROM 启动时，IROM 位于 0x0 地址处，而 SRAM 位于 0x40000000：



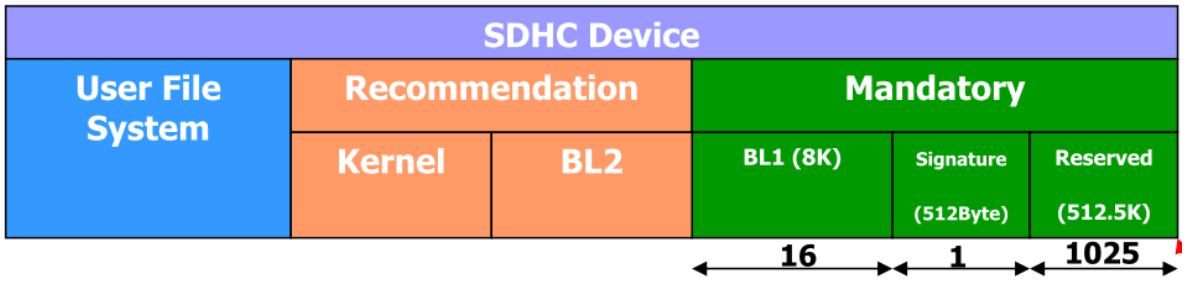
第二节　完整的启动序列

系统刚启动时，会自动运行 IROM 中的固化代码，进行一些通用的初始化，具体步骤包括：

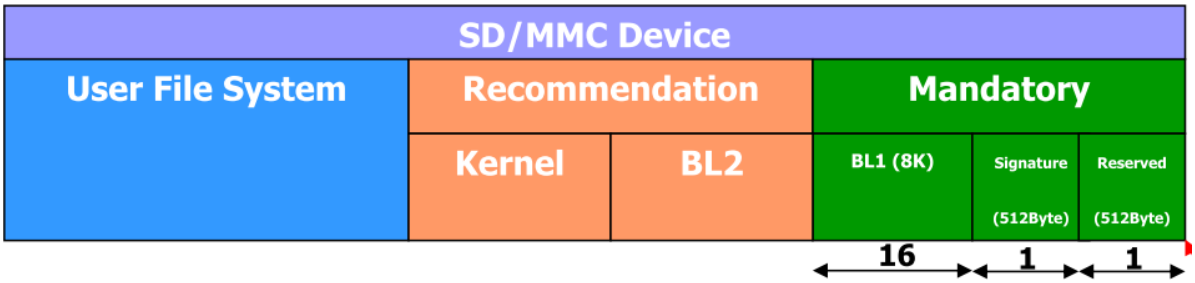
1) 关闭看门狗；
2) 初始化拷贝函数，后面的章节我们会利用这些拷贝函数来进行代码的重定位；
3) 初始化栈；
4) 初始化时钟；
5) 初始化 icache；
6) 初始化堆；
7) 如果是 NAND Flash 作为启动设备的话，将 NAND Flash 从 0 地址处开始的 8K 代码拷贝到 SRAM 的 0x40000000 处，参考下图：



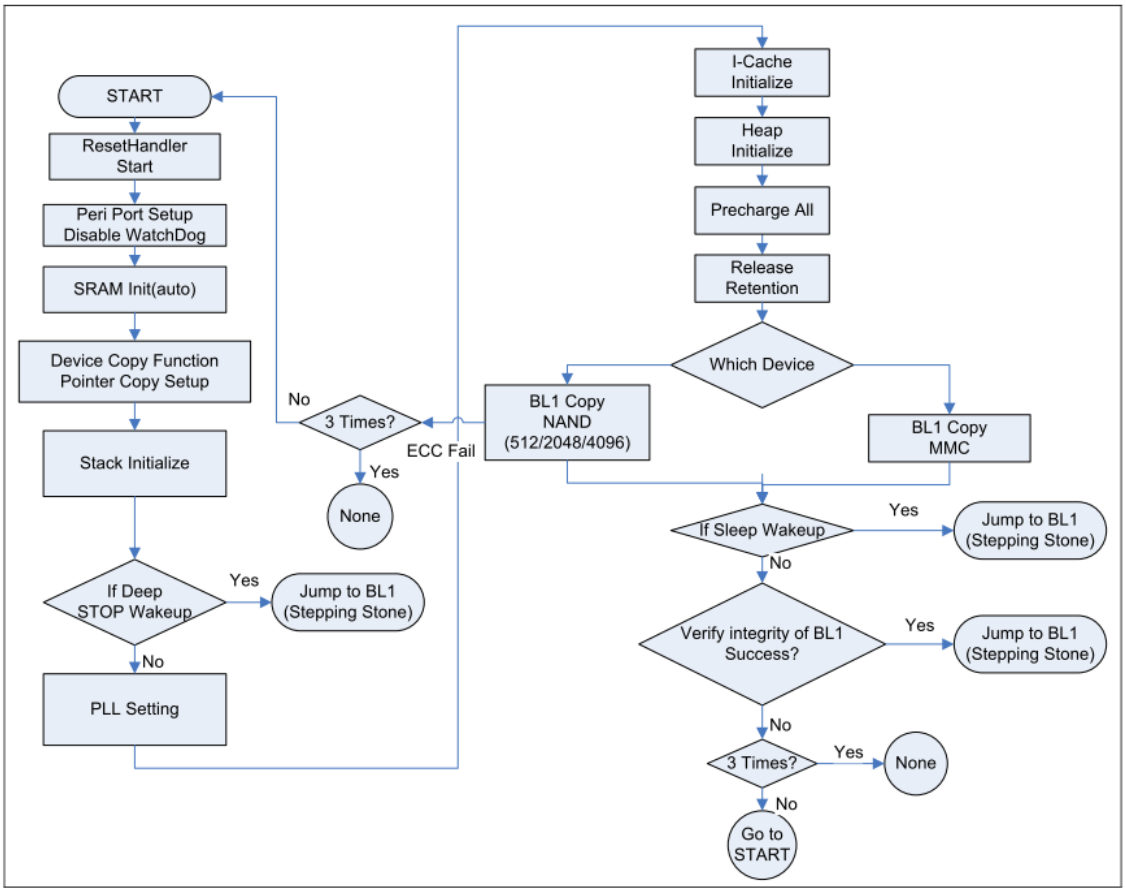
如果是大容量的 SDHC 卡作为启动设备的话，则将 SDHC 卡的倒数第(18+1024)个扇区开始的 8K 的代码拷贝到 SRAM 的 0x40000000 处，参考下图：



如果是 SD 卡作为启动设备的话，则将 SD 卡的倒数第 18 个扇区开始的 8K 的代码拷贝到 SRAM 的 0x40000000 处，参考下图：



8) 跳转到 SRAM 的 0x40000000 地址上运行我们自己的代码。



IROM 启动序列

这就是 IROM 中固化的代码所做的事，当程序跳转到 SRAM 后运行的就是我们自己编写的代码，其核心负责是将 NANDFlash 或 SDHC 卡上完整的程序重定位到 DRAM 中，最后跳转到 DRAM 中继续运行，这部分内容在后面的章节将会有更详细的讲解。

第十章　重定位代码到 SRAM+4096

第一节　两个不同的地址概念

对于程序而言，我们需要理解两个概念，一是程序当前所处的地址，即程序在运行时，所处的当前地址；二是程序应该位于的运行地址，即编译程序时，所指定的程序的链接地址。对于 S3C2451 而言，启动时只会从 NAND Flash 或 SDHC 卡等启动设备中拷贝前 8K 的代码到 SRAM 中，然后跳到 SRAM 中运行我们的代码。那么当我们的程序超过 8K 怎么办？那就需要我们在前 8K 的代码中将整个程序完完整整地拷贝到 DRAM 等其他更大存储空间，然后再跳转到 DRAM 中继续运行我们的代码，这个拷贝然后跳转的过程就叫重定位。在前面章节里，我们都是直接将 bin 文件下载到 DRAM 中运行的，所以不需要我们进行重定位。而在本章节，我们将会把 SRAM 中 0x40000000 地址处的代码重定位到 0x40001000 处，然后跳转到 0x40001000 处继续执行代码。

第二节　程序相关讲解

完整代码见目录 9.link_4096,该目录下的代码与上一章的代码的差别在于修改了 start.S 和多了链接脚本 link.lds，我们首先分析 link.lds。

1.link.lds

什么是链接脚本？链接脚本就是程序链接时的参考文件，其主要目的是描述如何把输入文件中的段（SECTION）映射到输出文件中，并控制输出文件的存储布局。链接脚本的基本命令式 SECTIONS 命令，一个 SECTIONS 命令内部包含一个或多个段，段(SECTION)是链接脚本的基本单元，它表示输入文件中的某个段是如何放置的。链接脚本的标准格式如下：

```
SECTIONS
{
    sections-command
    sections-command
}
```

下面我们配合 link.lds 进行具体讲解：

```
SECTIONS
{
    . = 0x40001000;
    .text : {
        start.o
        * (.text)
    }

    .data : {
```

```
        * (.data)
    }

    bss_start = .;
    .bss : {
        * (.bss)
    }
    bss_end  = .;
}
```

1) 在链接脚本中，单独的点号(.)代表了当前位置，. = 0x40001000;表示我们的代码的运行地址是 0x40001000；

2) link.lds 中的.text / .data / .bss 分别是 text 段、data 段、bss 段的段名，这些段名并不是固定的,是可以随便起的。.text 段包含的内容是 start.o 和其余代码中所有的 text 段,.data 段包含的内容是代码中所有的 data 段，.bss 段包含的内容是代码中所有的 bss 段。

3) bss_start 和 bss_end 保存的是 bss 段的起始地址和结束地址，在 start.S 中会被用到。

下面解释一下什么是 data、text、bss 段：

1) data 段：

数据段（datasegment）通常是指用来存放程序中已初始化的全局变量的一块内存区域。数据段属于静态内存分配。

2) text 段：

代码段通常是指用来存放程序执行代码的一块内存区域。这部分区域的大小在程序运行前就已经确定，并且内存区域通常属于只读,某些架构也允许代码段为可写，即允许修改程序。在代码段中，也有可能包含一些只读的常数变量，例如字符串常量等。

3) bss 段：

指用来存放程序中未初始化的全局变量的一块内存区域。BSS 是英文 BlockStarted by Symbol 的简称。当我们的程序有全局变量是，它是放在 bss 段的，由于全局变量默认初始值都是 0，一般需要我们手动清 bss 段。

2.start.S

在 start.S 中，有 4 个需要注意的地方：

1) 将栈顶指针 SP 指向 0x40002000，NAND Flash 启动时，S3C2451 的内部 8K 的 SRAM 被映射到 0x40000000，而 ARM 默认的栈是递减的，所以可以让 SP 指向 0x40002000；

2) 重定位，代码如下：

```
    adr r0, _start                  @ _start 的当前地址
    ldr r1, =_start                 @ _start 的链接地址
    ldr r2, =bss_start
    cmp r0, r1
    beq clean_bss
```

```
copy_loop:
    ldr r3, [r0], #4
    str r3, [r1], #4
    cmp r1, r2
    bne copy_loopp
```

首先获得_start 标号的当前地址(即 0x40000000)，然后获取_start 标号的链接地址(即 0x40001000)，因为 bin 文件中不需要保存 bss 段，所以拷贝的长度为 bss_start 的链接地址减去_start 的链接地址。

注意：adr 指令获取的是代码当前位于的地址，而 ldr 指令获取的是代码的链接地址。

3) 清 bss，代码如下：

```
clean_bss:
    ldr r0, =bss_start
    ldr r1, =bss_end
    mov r2, #0
clean_loop:
    str r2, [r0], #4
    cmp r0, r1
    bne clean_loop
```

首先获取 bss_start 的链接地址，然后获取 bss_end 的链接地址，使用 clear_loop 将该部分的内存清 0，bss_start 和 bss_end 的链接地址由 link.lds 决定。

4) 跳转，代码如下：

```
    ldr pc, =main
```

由于 ldr 指令获取的是 main 函数的链接地址，所以执行 ldr pc, =main 后，程序就跳转到 0x40001000+main 函数的 offset 的地址处了

3.Makefile

Makefile 中多了下列代码：

```
burn:
    cp *.bin ../2416.bin
    sudo ../hc_fusing_boot.sh
```

执行“make burn”命令时，首先将 bin 文件拷贝到上一级目录，并更改名字为 2416.bin，然后执行该目录下的 shell 脚本 hc_fusing_boot.sh 将 2416.bin 文件烧写到 SDHC 卡中。hc_fusing_boot.sh 的内容如下：

```
#!/bin/bash
DEV_NAME=sdb
BLOCK_CNT=`cat /sys/block/${DEV_NAME}/size`
let FIRMWARE_POSITON=${BLOCK_CNT}-16-2-1024
set -x
```

```
umount /dev/sdb1 2>/dev/null
umount /dev/sdb2 2>/dev/null
umount /dev/sdb3 2>/dev/null
umount /dev/sdb4 2>/dev/null
dd if=../2416.bin of=/dev/${DEV_NAME} bs=512 seek=${FIRMWARE_POSITON}
sync
```

它的作用是将 bin 文件烧写到 SDHC 卡上的倒数第(16+2+1024)块的位置上。这样开发板上电 IROM 里的固化代码就会拷贝 bin 文件到 SRAM 中运行了。

第三节　编译代码和烧写运行

编译代码

在 Fedora 终端执行如下命令：

```
# cd 9.link_4096
# make
```

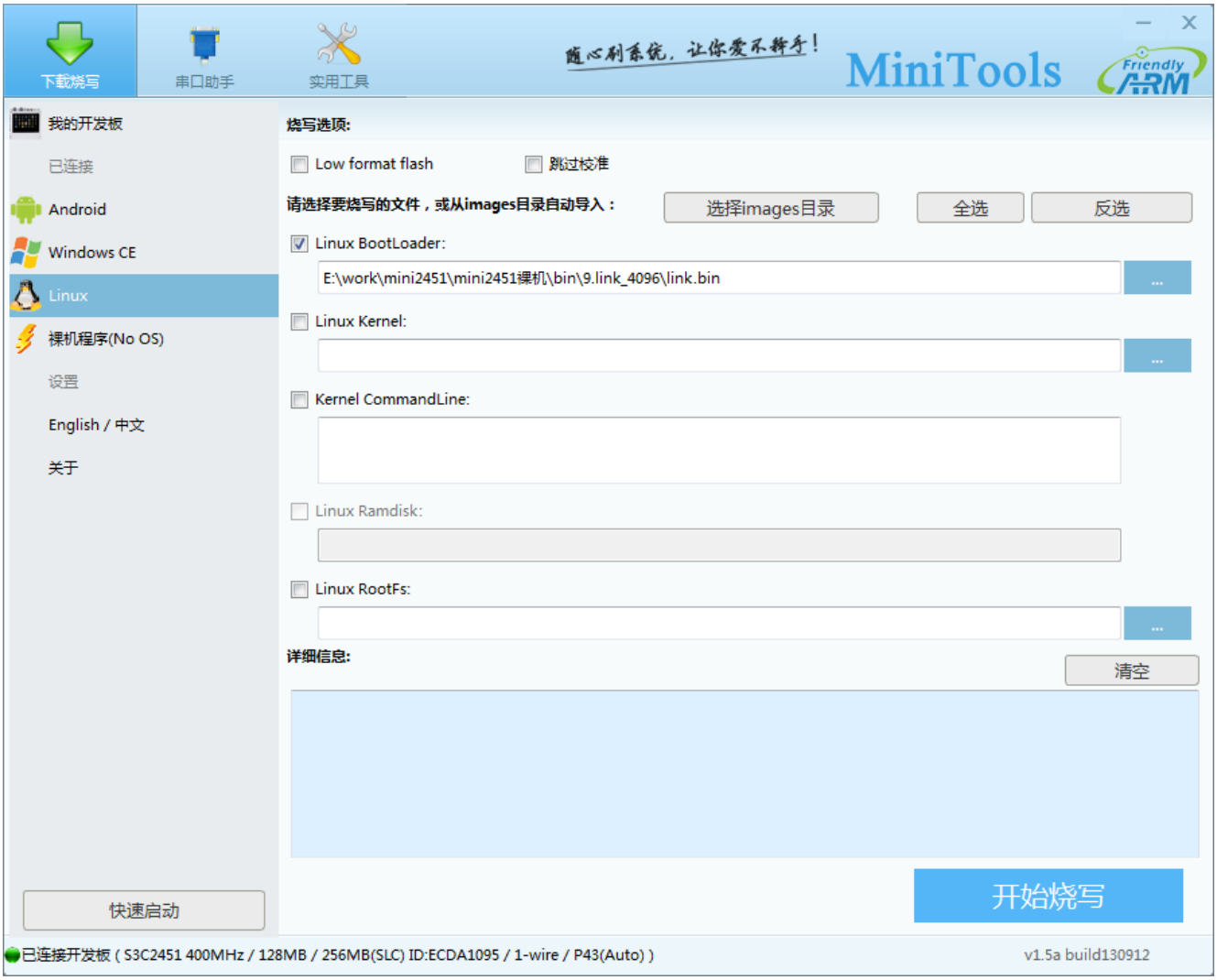
make 成功后会生成 link.bin 文件。

烧写运行

想从 NAND Flash 中加载裸机的话，则将裸机程序烧写到 NAND Flash；想从 SDHC 卡加载裸机的话，则将裸机程序烧写到 SDHC 卡中，具体的烧写方法如下：

1) 从 NAND Flash 加载运行：

使用 MiniTools 将 bin 文件烧写到开发板上的 NAND Flash 的 0 地址，其设置方式如下：



点击开始烧写，MiniTools 就会把 bin 文件烧写到 NAND Flash 的 0 地址上了。

2) 从 SDHC 卡加载运行:
将 SDHC 卡插入到 PC 中，然后进入代码所在目录，在 Fedora 终端下执行如下命令
make burn
该命令会调用脚本 hc_fusing_boot.sh 将编译好的 bin 文件烧写到 SDHC 卡上。如果执行该命令时提示无法执行该脚本文件，则需要通过 chmod 777 hc_fusing_boot.sh 命令为脚本文件增加可执行权限。对于其他脚本文件出现类似情况的话，解决办法一样。

注意:为了不破坏 NAND Flash 中的数据，本文档后面的章节除特殊情况外都将采用将裸机程序烧写到 SDHC 卡的方式。

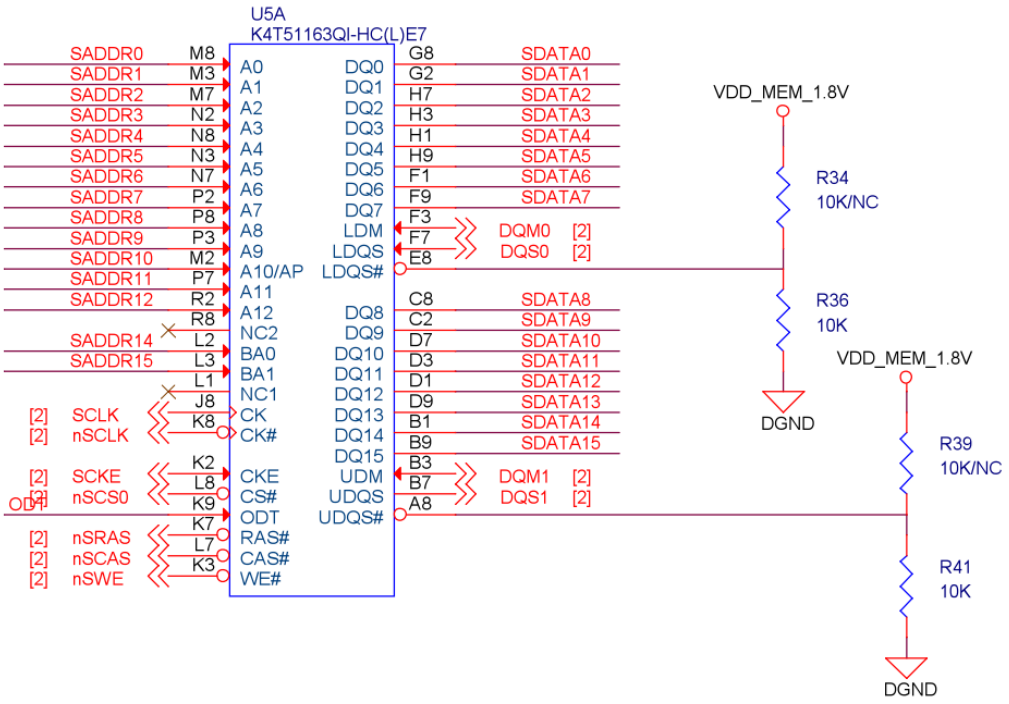
第四节　实验现象

从 NAND Flash 加载运行的话，选择 NAND 启动，从 SDHC 卡加载运行的话，则选择 SD 启动。开发板上电后，在 PC 键盘上输入一个字符，则终端会显示该字符在 ASCII 表中的下一字符，例如键盘输入“abcd”，终端上会显示“bcde”，该现象与第八章的代码运行效果一模一样，但是程序的运行过程却有了很大的区别。通过本章的学习，我们已经知道了如何对代码进行重定位，这为我们下一章节将代码重定位到 DRAM 奠定了基础。

第十一章　重定位代码到 DRAM

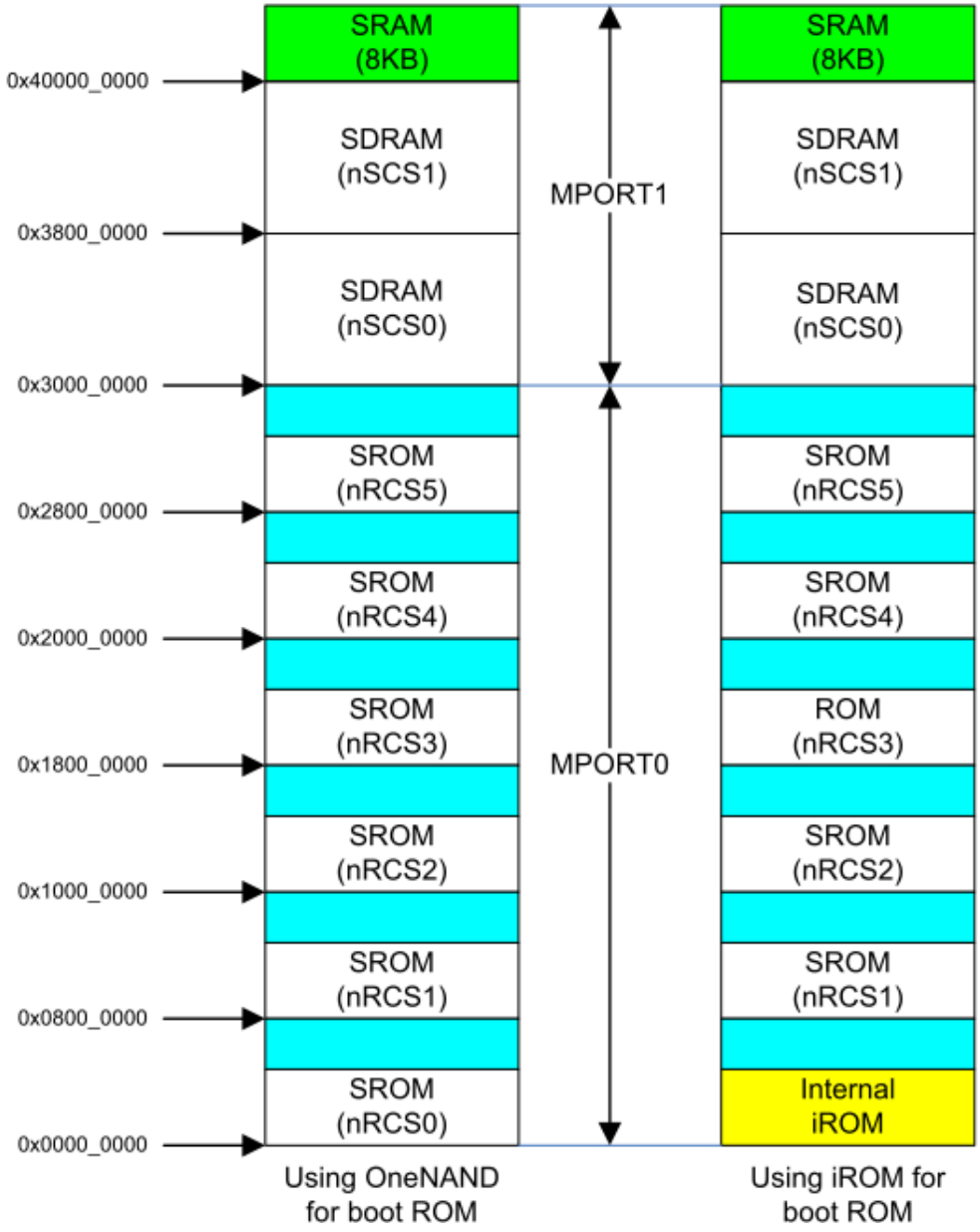
第一节　关于 DRAM

上一章我们讲解了如何对代码进行重定位，但是将代码重定位到 SRAM 中的作用不大。正确的做法是将代码重定位到容量更大的主存中，即 DRAM。S3C2451 的 DRAM 控制器支持三种内存接口：SDRAM、mobile DDR、DDR2,提供了 2 个片选信号，即支持同时挂接 2 个同类型的内存设备。查阅 Mini2451 原理图：



Mini2451 DRAM 原理图

可以看出，Mini2451 的 DRAM 挂接型号为 K4T51163QI-HC(L)E7,它是一款 DDR2 SDRAM，容量为 64MB。参考下图，可以看出 DRAM 的起始地址为 0x30000000。如何才能使用该 DRAM？对应 Mini2451 而言，我们需要初始化 DRAM 控制器。



第二节　程序相关讲解

本章涉及两个程序，包括 10.ddr 和 11.mmc_boot。程序 10.ddr 是把代码从 SRAM 中重定位到 DRAM，而程序 11.mmc_boot 是把代码从 SDHC 卡中重定位到 DRAM。我们先分析程序 10.ddr，该目录下的代码与上一章的代码的差别在于修改了 start.S 和链接脚本 ddr.lds，以及多了一个初始化 DRAM 控制器的 sdram.c。我们首先分析 ddr.lds。

1. ddr.lds

链接地址被修改为 0x30000000，程序的正常运行需要我们执行如下两个步骤：

1） 初始化 DRAM 控制器，使以 0x30000000 为起始的 64M 内存可用；

2） 将代码重定位到 0x30000000 处；

2. start.S

相比上一章的代码,本章的 start.S 多了两个步骤：

1） 调用函数 sdr_ctrl_asm_init()初始化内存，该函数的实现位于 sdram.S；

2） 重定位，把 SRAM 上 0x40000000 地址开始处的程序拷贝到内存 0x30000000 处。代码如下：

```
    adr r0, _start              // r0 = _start 的当前地址
    ldr r1, =_start             // r1 = _start 的链接地址
    ldr r2, =bss_start          // r2 - r1 = 程序的长度
    cmp r0, r1
    beq clean_bss
copy_loop:                      // 循环拷贝
    ldr r3, [r0], #4
    str r3, [r1], #4
    cmp r1, r2
    bne copy_loop
```

3. sdram.S

S3C2451 芯片手册里已经告诉我们如何初始化 DDR2 类型的 DRAM，参考下图：

3.2 DDR2 INITIALIZATION SEQUENCE

1. Setting the BANKCFG & BANKCON1, 2, 3
2. Wait 200us to allow DRAM power and clock stabilize.
3. Wait minimum of 400 ns then issue a PALL(pre-charge all) command.
   Program the INIT[1:0] to '01b'. This automatically issues a PALL(pre-charge all) command to the DRAM.
4. Issue an EMRS command to EMR(2), provide LOW to BA0, High to BA1.
   Program the INIT[1:0] of Control Register1 to '11b' & BANKCON3[31]='1b'
5. Issue an EMRS command to EMR(3), provide High to BA0 and BA1.
   Program the INIT[1:0] of Control Register1 to '11b' & BANKCON3[31:30]='11b'
6. Issue an EMRS to enable DLL and RDQS, nDQS, ODT disable.
7. Issue a Mode Register Set command for DLL reset.(To issue DLL Reset command, provide HIGH to A8 and LOW to BA0-BA1, and A13-A15.) Program the INIT[1:0] to '10b'. & BANKCON3[8]='1b'
8. Issue a PALL(pre-charge all) command.
   Program the INIT[1:0] to '01b'. This automatically issues a PALL(pre-charge all) command to the DRAM.
9. Issue 2 or more auto-refresh commands.
10. Issue a MRS command with LOW to A8 to initialize device operation.
    Program the INIT[1:0] to '10b'. & BANKCON3[8]='0b'
11. Wait 200 clock after step 7, execute OCD Calibration.
12. The external DRAM is now ready for normal operation

sdram.S 就是参考了上述相关知识并借鉴了 U-boot 进行了内存的初始化，只要严格按照上述步骤，并且参考 sdram 的芯片手册就可以驱动内存。具体设置可以参考代码，代码本身已经附有

详细的注释。

下面我们继续分析下一个程序 11.mmc_boot,该程序与 10.ddr 的区别前面已经说过了，那么 11.mmc_boot 是如何将代码从 SDHC 卡中重定位到 DRAM 中的呢？其实就是利用了 IROM 固化代码里的拷贝函数。在 start.S 中进行重定位是调用了函数 copy2ddr()，该函数的实现位于 mmc.c,具体代码如下：

```
#define    TCM_BASE                0x40004000
#define CopyMovitoMem(a,b,c,d)      (((int(*)(uint,  ushort,  uint  *,  int))(*((uint
*)(TCM_BASE + 0x8))))(a,b,c,d))
int copy2ddr()
{
    int total_blkcnt;
    uint mmc_start;
    ushort len;
    int ddr_start;

    total_blkcnt = *((volatile unsigned int*)(TCM_BASE - 0x4));
    mmc_start = total_blkcnt - 2 - 16 - 256;
    ddr_start = 0x30000000;
    len = 16;

    CopyMovitoMem(mmc_start, len, (uint *)ddr_start, 0);

}
```

可以看出，CopyMovitoMem 一个宏，该宏实质是一个函数指针，它指向了 0x40004000+0x8 地址处的一个函数，那么这个地址上有什么函数呢？参考下图：

Type	Address	Usage	Size
I-RAM	0x40000000 ~ 0x40001FFF	Stepping Stone (BL1)	8KB
SRAM	0x40002000 ~ 0x400021FF	Secure Key(512Bytes)	(64-8)KB
	0x40002200 ~ 0x40002FFF	Reserved	
	0x40003000 ~ 0x40003FFF	Heap (Reserved for global variable)	
	0x40004000 ~ 0x40010000	Device Copy Function Pointer (12Byte)	
		Stack	

位于 0x40004008 处的正是 MMC 拷贝函数,它的原型为 CopyMovitoMem(uint a, ushort b, uint *c, int d)，作用是将 MMC 卡上的第 a 个扇区起的 b 个扇区拷贝到 c 地址去。另外，total_blkcnt 指向的是 0x40003FFC 地址上的值，参考下图:

Address	Name	Usage
0x40003FFC	globalBlockSizeHide	Total block count of the MMC device.
0x40003FF8	globalSDHCInfoBit	globalSDHCInfoBit[31:16] : RCA(Relative Card Address) Data globalSDHCInfoBit[2] : SD Card globalSDHCInfoBit[1] : MMC Card globalSDHCInfoBit[0] : High Capacity Enable
0x40003FF4	globalNandECCfailureCount	Total number of ECC Fail

可以知道该地址存放了 MMC 卡的总扇区数量,需要注意的是,对于 SD 卡，存放的是 SD 卡的总扇区数，而对于 SDHC 卡，存放的是 SDHC 卡的总扇区数-1024。

函数 copy2ddr()的作用就是将 SDHC 卡倒数第(18+256+1024)个扇区开始的 256 个扇区拷贝到 DRAM 的 0x30000000 地址处。因此，对于程序 11.mmc_boot，编译出来的 bin 文件需要使用 hc_fusing_mmcboot.sh 将其烧写到 SDHC 卡的两个地方，一是 SDHC 倒数第(18+1024)个扇区处，用于作为前 8k 的启动代码，二是倒数第(18+256+1024)个扇区处，前 8k 的启动代码会把位于此处的 bin 文件拷贝到 DRAM 中。下面是 hc_fusing_mmcboot.sh 的内容:

```
#!/bin/bash
DEV_NAME=sdb
BLOCK_CNT=`cat /sys/block/${DEV_NAME}/size`
let FIRMWARE1_POSITON=${BLOCK_CNT}-16-2-1024
let FIRMWARE2_POSITON=${BLOCK_CNT}-16-2-1024-256
set -x
umount /dev/sdb1 2>/dev/null
umount /dev/sdb2 2>/dev/null
umount /dev/sdb3 2>/dev/null
umount /dev/sdb4 2>/dev/null
dd if=../2416.bin of=/dev/${DEV_NAME} bs=512 seek=${FIRMWARE1_POSITON}
dd if=../2416.bin of=/dev/${DEV_NAME} bs=512 seek=${FIRMWARE2_POSITON}
sync
```

注意：此处 SDHC 卡的设备节点为/dev/sdb,用户必须自行查看自己的 SDHC 卡的设备节点名称，并对 hc_fusing_mmcboot.sh 做出相应的修改。

第三节　编译代码和烧写运行

编译代码

1)　编译 10.ddr

在 Fedora 终端执行如下命令：

```
# cd 10.ddr
# make
```

make 成功后会生成 ddr.bin 文件。

2） 编译 11.mmc_boot
在 Fedora 终端执行如下命令：
cd 11.mmc_boot
make
make 成功后会生成 mmc.bin 文件。

烧写运行
1） 运行 10.ddr
进入代码所在目录，编译成功后在 Fedora 终端执行如下命令：
make burn
该命令会调用脚本 hc_fusing_boot.sh 将 ddr.bin 烧写到 SDHC 卡上。

2） 运行 11.mmc_boot
进入代码所在目录，编译成功后在 Fedora 终端执行如下命令：
make burn
该命令会调用脚本 hc_fusing_mmcboot.sh 将 ddr.bin 烧写到 SDHC 卡上。

第四节　实验现象

对于程序 10.ddr 和 11.mmc_boot,它们的运行效果是一样的。开发板选择 SD 启动，然后在 PC 键盘上输入一个字符，则终端会显示该字符在 ASCII 表中的下一字符，例如键盘输入“abcd”，终端上会显示“bcde”，该现象与第九章的代码运行效果一模一样。下一章将讲解如何从 NAND Flash 中拷贝代码到 DRAM 中。

第十二章 NAND Flash 控制器

第一节 关于 NAND Flash

S3C2451 的 NAND Flash 控制器有如下特点：

1) 自动导入模式：复位后，引导代码被送入 8KB 的 STEPPINGSTONE 中，引导代码移动完毕，引导代码将在 STEPPINGSTONE 中执行。 在导入期间，NAND FLASH 控制器不支持 ECC 纠正。

2) NAND FLASH 控制器 I/F：支持 512 字节、 2KB 页、4KB 页。

3) 软件模式：用户可以直接访问 NAND FLASH 控制器。例如这个特性可以用于读/擦/编程 NAND FLASH 存储器。

4) 接口：8 位 NAND FLASH 存储器接口总线。

5) 硬件 ECC 产生、检测和标志（软件纠正）。

6) 支持 SLC 和 MLC 的 NAND FLASH 控制器：1 位 、4 位、8 位 ECC NAND FLASH。

7) 特殊功能寄存器 I/F：支持字节/半字/字数据的访问和 ECC 的数据寄存器，用字来访问其他寄存器。

8) STEPPINGSTONE I/F：支持字节/半字/字的访问。

9) 8KB 内部 SRAM 缓冲器 STEPPINGSTONE，在 NAND FLASH 引导后可以作为其他用途使用。

本教程中，Mini2451 的 NAND Flash 类型为 SLC，大小为 256MB， 型号为 K9F2G08U0B。

注意：本章的内容是针对 SLC 类型的 NAND Flash，并不适用 MLC 类型的 NAND Flash。

第二节 程序相关讲解

完整代码见目录 12.nand_boot。程序的目的是开发板选择 NAND 启动，然后将 NAND Flash 中的代码读到 DRAM 中，然后跳转到 DRAM 中继续执行代码。

1. start.S

相比上一章的的 start.S 的，本章的 start.S 有两点不同：

1) 在重定位之前调用了 nand_init()初始化 NAND Flash；

2) 重定位是通过调用 copy2ddr()函数将代码从 SRAM 拷贝到 DRAM 中，copy2ddr()的具体实现位于 nand.c；

2. nand.c

1> nand_init()函数

该函数用于初始化 NAND Flash，代码如下：

```
void nand_init(void)
{
    NFCONF_REG = ( (0x2<<12)|(0xf<<8)|(0x7<<4) );
    NFCONT_REG |= (0x3<<0);
```

}

Register	Address	R/W	Description	Reset Value
NFCONF	0x4E000000	R/W	NAND Flash Configuration register	0xX000100X

NFCONF	Bit	Description	Initial State
Reserved	[31]	Reserved	0
Reserved	[30]	Should be 0	0
Reserved	[29:26]	Reserved	0000
MsgLength	[25]	Message (Data) length for 4/8 bit ECC 0 = 512-byte 1 = 24-byte	0
ECCType	[24:23]	This bit indicates what kind of ECC should be used. **00 = 1-bit ECC** **10 = 4-bit ECC** **01 = 8-bit ECC** **Note:** Don't confuse the value of 4-bit ECC and 8-bit ECC.	H/W Set (CfgBootEcc)
Reserved	[22:15]	Reserved	000000000
TACLS	[14:12]	CLE & ALE duration setting value (0~7) Duration = HCLK x TACLS	**001**
Reserved	[11]	Reserved	0
TWRPH0	[10:8]	TWRPH0 duration setting value (0~7) Duration = HCLK x (TWRPH0 + 1)	000
Reserved	[7]	Reserved	0
TWRPH1	[6:4]	TWRPH1 duration setting value (0~7) Duration = HCLK x (TWRPH1 + 1)	000
PageSize	[3]	This bit indicates the page size of NAND Flash Memory When PageSize_Ext is 1, the value of PageSize means following: **0 = 512 Bytes/page, 1 = 2048 Bytes/page** When PageSize_Ext is 0, the value of PageSize means following: **0 = 2048 Bytes/page, 1 = 4096 Bytes/page**	H/W Set (CfgAdvFlash)
PageSize_Ext	[2]	This bit indicated what kind of NAND Flash memory is used. **0 = Large Size NAND Flash** **1 = Small Size NAND Flash** This bit is determined by OM[2] pin status on reset and wake-up time from sleep mode. This bit can be changed by software later.	1

- TWRPH1/TWRPH0/TACLS 是关于访问时序的设置，需对照 NAND Flash 芯片手册设置，这里不再详细解释，分别取 TWRPH1=0x7，TWRPH0=0xF，TACLS=0x2；
- 其余位均使用默认值；

Register	Address	R/W	Description	Reset Value
NFCONT	0x4E000004	R/W	NAND Flash control register	0x000100C6

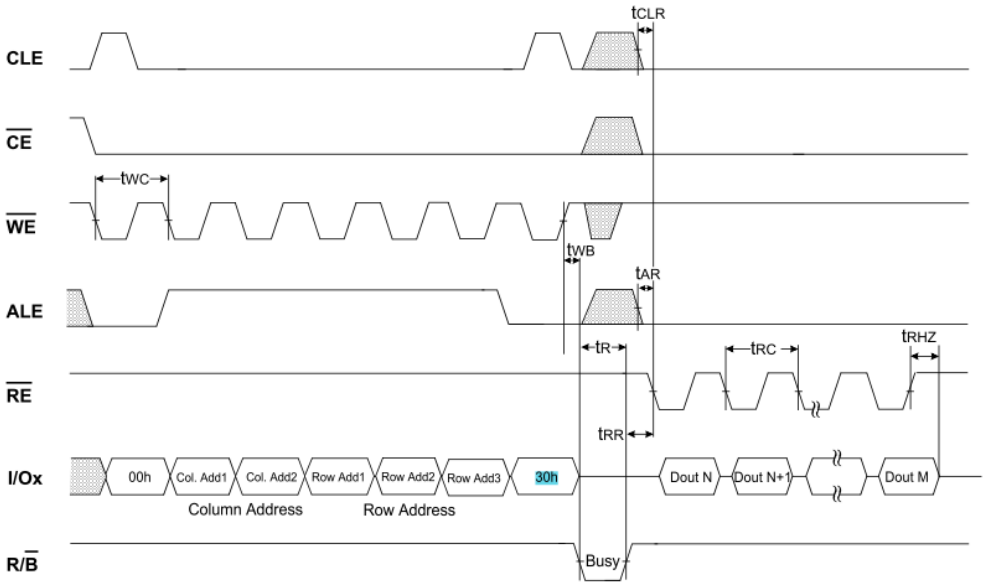
NFCONT	Bit	Description	Initial State
Reserved	[31:19]	Reserved	0
ECC Direction	[18]	4-bit, 8-bitECC encoding / decoding control 0 = Decoding 4-bit, 8bit ECC, It is used for page read 1 = Encoding 4-bit, 8-bit ECC, It is be used for page program	0
Lock-tight	[17]	Lock-tight configuration 0 = Disable lock-tight 1 = Enable lock-tight, Once this bit is set to 1, you cannot clear. Only reset or wake up from sleep mode can make this bit disable (cannot cleared by software). When it is set to 1, the area setting in NFSBLK (0x4E000020) to NFEBLK (0x4E000024) is unlocked, and except this area, write or erase command will be invalid and only read command is valid. When you try to write or erase locked area, the illegal access will be occurred (NFSTAT [5] bit will be set). If the NFSBLK and NFEBLK are same, entire area will be locked.	0
Soft Lock	[16]	Soft Lock configuration 0 = Disable lock 1 = Enable lock Soft lock area can be modified at any time by software. When it is set to 1, the area setting in NFSBLK (0x4E000020) to NFEBLK (0x4E000024) is unlocked, and except this area, write or erase command will be invalid and only read command is valid. When you try to write or erase locked area, the illegal access will be occurred (NFSTAT [5] bit will be set). If the NFSBLK and NFEBLK are same, entire area will be locked.	1
Reserved	[15:13]	Reserved. Should be written to 0.	000
EnbECCDecINT	[12]	4-bit, 8-bit ECC decoding completion interrupt control 0 = Disable interrupt 1 = Enable interrupt	0
8bit Stop	[11]	8-bit ECC encoding/decoding operation initialization	0

- MODE = 1，使能 NAND Flash；
- Reg_nCE0 = 1,暂时不选中 NAND Flash；
- 其余位使用默认值；

2> nandll_read_page()函数

该函数用于从 NAND Flash 中读一页的数据，对于 Mini2451 的 NAND Flash，一页即 2048Byte。参考 NAND Flash 的芯片手册：

4.7 Read Operation



读一页 NAND Flash 的操作如下：

1） 发片选，即设置 NFCONT 的 bit1 为 0；

2） 发读命令：0x00，即往 NFCMD 写 0x00；

3） 发 5 个周期的地址，即往 NFADDR 寄存器写入地址，因为 NFADDR 一次只能接受 8bit 的数据，所以分 5 次来写；

4） 发读命令：0x30，即往 NFCMD 写 0x30；

5） 连续读 2048 个字节，即连续读 NFDATA 2048 次；

3> copy2ddr()函数

该函数负责调用 nandll_read_page()进行读页操作，直到 bin 文件完全地从 NAND Flash 中拷贝到 DRAM 中。

第三节　编译代码和烧写运行

编译代码

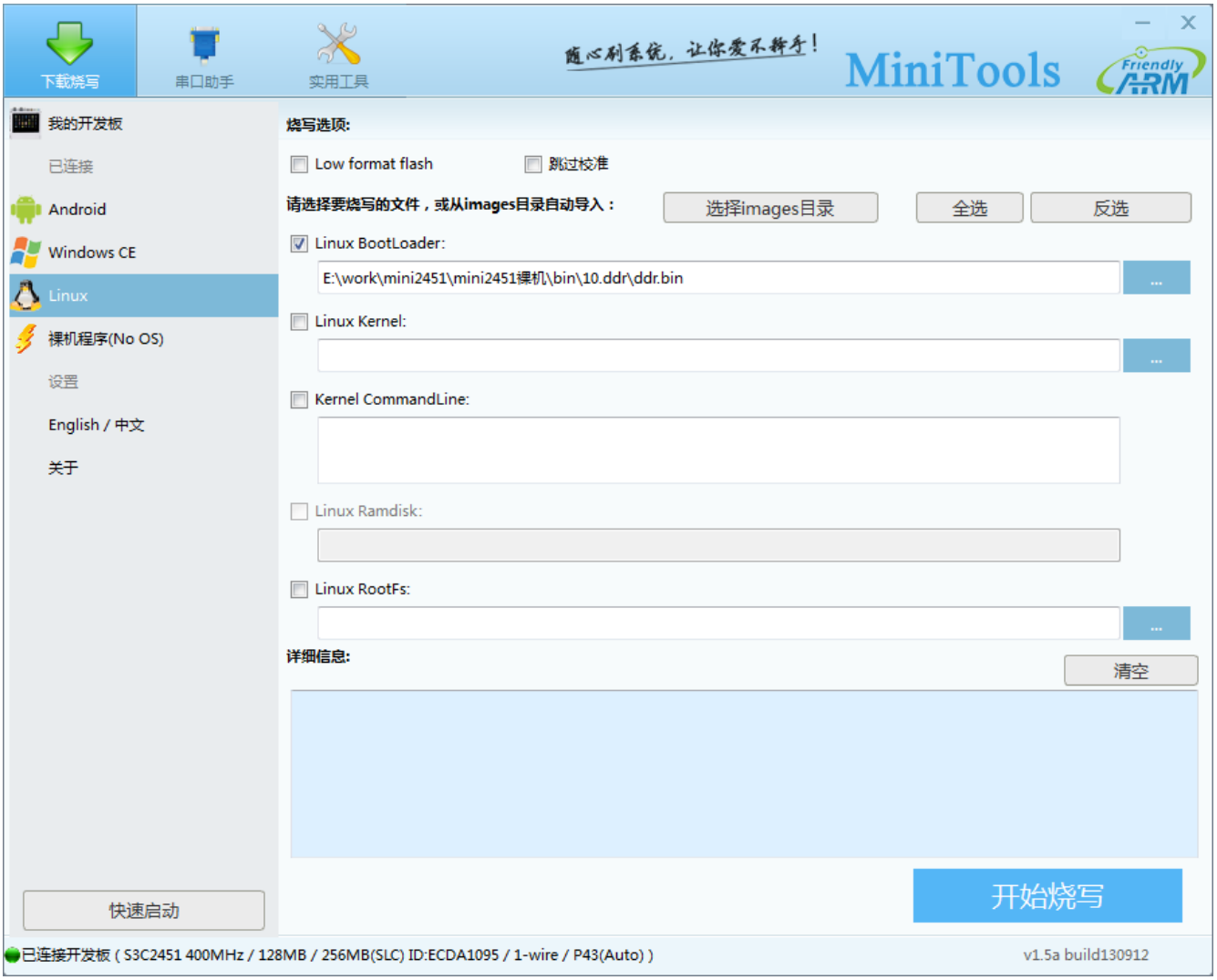
在 Fedora 终端执行如下命令：

```
# cd 12.nand_boot
# make
```

make 成功后会生成 nand.bin 文件。

烧写运行

使用 MiniTools 将 bin 文件烧写到开发板上的 NAND Flash 的 0 地址，其设置方式如下：



点击开始烧写，MiniTools 就会把 bin 文件烧写到 NAND Flash 的 0 地址上了。

第四节　实验现象

开发板选择 NAND 启动，上电，然后在 PC 键盘上输入一个字符，则终端会显示该字符在 ASCII 表中的下一字符，例如键盘输入“abcd”，终端上会显示“bcde”，该现象与上一章的代码运行效果一模一样。

第十三章　内存管理单元 MMU

第一节　关于 MMU

内存管理单元(Memory Management Unit)简称 MMU，它负责虚拟地址到物理地址的映射，并提供硬件机制的内存访问权限检查。现代的多用户多进程操作系统通过 MMU 使得各个用户进程都拥有自己独立的地址空间：地址映射功能使得各个进程拥有“看起来”一样的地址空间，而内存访问权限的检查可以保护每个进程所用的内存不会被其他进程破坏。这里不会详细讲解地址映射的知识，我们直接来看程序，程序涉及了什么知识我们就讲解什么知识，避免陷入各种复杂的理论知识里。

第二节　程序相关讲解

本章涉及的代码有两套，包括 13.leds_mmu_s(使用汇编初始化 MMU)和 14.leds_mmu_c(使用 C 语言初始化 MMU)，两者的本质是一样的，初始化 MMU 所达到的效果也一样，想巩固汇编指令的话，可以阅读 13.leds_mmu_s 里的相关代码。由于 C 语言编写的代码思路更清晰，更便于阅读和理解，下面将以 14.leds_mmu_c 里的代码为参考进行讲解。

1. mmu.lds

```
SECTIONS {
    . = 0xc0000000;

    .text : {
            start.o
            clock.o
            mmu.o
            sdram.o
            * (.text)
    }
    .rodata : {
            * (.rodata)
    }
    … // 省略
}
```

链接地址是 0xc0000000,这是个虚拟地址，从后面的代码讲解可知，虚拟地址 0xc0000000 对应物理地址 0x30000000。

2. start.S

本章的 start.S 有 2 个需要注意的地方：
1） 调用 C 函数 copy2ddr()进行重定位；
2） 调用了 mmu 初始化函数 mmu_init()；
copy2ddr()的定义位于 main.c 中，代码如下：

```
void copy2ddr(void)
{
    unsigned char *dest = (unsigned char *)0x30000000;
    unsigned char *src  = (unsigned char *)0x40000000;
    int i = 0;

    while (i < 8*1024)
    {
        dest[i] = src[i];
        i++;
    }

}
```

这段代码的作用就是将我们的代码从 0x40000000 地址拷贝到 0x30000000。

3. mmu.c

该文件的核心就是 mmu_init()函数，它的作用就是设置 mmu 和启动 mmu，具体的步骤如下：

1> 设置页表，代码如下：

```
    #define MMU_BASE                0x31000000          // 页表基址
    volatile unsigned long *table = (volatile unsigned long *)MMU_BASE;
    table[0x100] = 0x56000000 | MMU_SECDESC_NCNB;     // 建立页表项
    table[0x400] = 0x40000000 | MMU_SECDESC_WB;
    table[0xc00] = 0x30000000 | MMU_SECDESC_WB;
```

什么是页表？CPU 发出虚拟地址后，MMU 最终是如何找到对应的物理地址的，靠的就是页表。页表是有一个个页表项组成的，分为一级页表和二级页表，我们的程序里使用的是一级页表，即一个页表项所映射的地址空间大小为 1M。另外，页表存放于内存，负责存放页表的内存的起始地址即页表基地址，在本章程序里，页表基址是 0x31000000。

CPU 发出虚拟地址 VA，然后 MMU 根据“页表项地址 = 页表基地址+VA>>20”找到对应的页表项，最后再由页表项的高 12bit 和 VA 的低 20bit 组成 32bit 的物理地址，这样就完成了从虚拟地址到物理地址的映射。在代码中，我们根据上述规律构造了 3 个页表项，依次实现了下面 3 个映射：

1） 物理地址 0x56000000 起的 1M 空间====>虚拟地址 0x10000000 起的 1M 空间；
2） 物理地址 0x40000000 起的 1M 空间====>虚拟地址 0x40000000 起的 1M 空间；
3） 物理地址 0xc0000000 起的 1M 空间====>虚拟地址 0x30000000 起的 1M 空间；

其中，映射 1)是为了映射寄存器地址空间，在 main.c 中我们通过操作虚拟地址上的寄存器

达到点灯的目的；映射 2)是为了保证 MMU 使能前后的代码所在的地址空间一致；映射 3)是由于我们的程序是被重定位到 0x3000000 上，但是程序的链接地址是 0xc0000000，所以需要将 0x30000000 开始的 1M 内存映射到 0xc0000000。

2> 启动 MMU，参考如下步骤：

1) 使无效 ICaches 和 DCaches；
2) 使无效指令、数据 TLB；
3) TLB 即 Translation Lookaside Buffers，位于 MMU 和内存之后，是用来存储近期用到的页表项，以避免每次地址转换时都到主存取寻找。
4) 设置页表基地址；
5) 设置 domain，用于控制访问权限；
6) 使能 ICaches 和 Dcaches；
7) 使能 MMU；

4. main.c

main()函数的代码跟前面的点灯程序的代码差不多，唯一的驱动在于对寄存器的操作使用的是虚拟地址，代码如下：

```
volatile unsigned long *gpacon = (volatile unsigned long *)0x10000000;
volatile unsigned long *gpbcon = (volatile unsigned long *)0x10000010;
volatile unsigned long *gpadat = (volatile unsigned long *)0x10000004;
volatile unsigned long *gpbdat = (volatile unsigned long *)0x10000014;
volatile unsigned long *gpbsel = (volatile unsigned long *)0x1000001c;
```

第三节　编译代码和烧写运行

编译代码

在 Fedora 终端执行如下命令：

```
# cd 14.leds_mmu_c
# make
```

make 成功后会生成 mmu.bin 文件。

烧写运行

进入代码所在目录，编译成功后在 Fedora 终端执行如下命令：

```
# make burn
```

该命令会调用脚本 hc_fusing_boot.sh 将 mmu.bin 烧写到 SDHC 卡上。

第四节　实验现象

开发板选择 SD 启动，然后上电，LED 又欢快地闪烁起来了，没什么不同的，只不过这次我们使用的是虚拟地址。下一章节我们将会为我们的代码添加 printf 和 scanf 功能。

第十四章　移植 printf 和 scanf 功能

第一节　移植的途径

对于如何移植 printf 和 scanf，我们有许多选择：

1） 移植 linux 的 printk 功能，版本越新越难移植，但是功能也越强大；
2） 移植 uboot 的 printf 和 scanf 功能，实际 uboot 也是从 linux 内核中移植而来的；
3） 完全自己编写，但是功能比较弱。

在保证整个裸机其他代码部分没有任何问题，上述三种方法都是可行的。下面我们只是直接利用网友移植好的相关文件，为我们的裸机代码增加上该部分功能。

第二节　移植步骤

本次的移植是在代码 12.mmc_boot 的基础上进行修改，首先将代码 12.mmc_boot 复制一份，并改名为 15.uart_stdio，并进行如下操作：

1） 解压 printf.rar 到 15.uart_stdio 目录，解压成功后会多出 include 和 lib 两个目录，其中 include 放的是相关头文件，lib 放的是 printf 和 scanf 相关的代码。
2） 修改 15.uart_stdio 目录下的 makefile，将 lib 目录下的代码编译链接成 lib.a，然后将 lib.a 编译进 bin 中，具体修改见源码。
3） 编写 main 函数进行测试。

第三节　程序相关讲解

完整代码见目录 15.uart_stdio。

1. /lib/printf.c

1> printf()的定义如下：

```
int printf(const char *fmt, ...)
{
    int i;
    int len;
    va_list args;                           // va_list 即 char *

    va_start(args, fmt);
    len = vsprintf(g_pcOutBuf,fmt,args);    // 内部使用了 va_arg()
    va_end(args);
```

```
    for (i = 0; i < strlen(g_pcOutBuf); i++)
    {
        putc(g_pcOutBuf[i]);
    }
    return len;
}
```

2> 首先，printf 函数是个变参函数，什么是变参函数：

可变参数函数的原型声明为 type VAFunction(type arg1, type arg2, …); 参数可以分为两部分：个数确定的固定参数和个数可变的可选参数。函数至少需要一个固定参数，固定参数的声明和普通函数一样；可选参数由于个数不确定，声明时用"..."表示。固定参数和可选参数公同构成一个函数的参数列表。

3> 再来看 printf 函数，该函数涉及了 3 个十分重要的宏：

宏1:　#define va_start(ap, A) (void) ((ap) = (((char *) &(A)) + (_bnd (A,_AUPBND))))
宏2:　#define va_arg(ap, T)　(*(T *)(((ap) += (_bnd (T, _AUPBND))) - (_bnd (T,_ADNBND))))
宏3:　#define va_end(ap)　(void) 0
在这些宏中，va 就是 variable argument(可变参数)的意思;
ap:是指向可变参数表的指针;
A:指可变参数表的前一个固定参数;
T:可变参数的类型。
va_list 也是一个宏，其定义为 typedef char * va_list，实质上是一 char 型指针。

4> 下面逐个来分析这三个宏的作用：

(1) va_start 宏

- 作用:

根据 v 取得可变参数表的首指针并赋值给 ap，方法：最后一个固定参数 A 的地址 + 第一个变参对 A 的偏移地址，然后赋值给 ap，这样 ap 就是可变参数表的首地址。

- 举例:

如果有变参函数的声明是 void va_test(char a, char b,char c, …)，则它的固定参数依次是 a,b,c，最后一个固定参数 为 c，因此就是 va_start(ap, c)。

(2) va_arg 宏

- 作用:

指取出当前 ap 所指的可变参数并将 ap 指针指向下一可变参数。

(3) va_end 宏

- 作用:

结束可变参数的获取。va_end (list)实际上被定义为空，没有任何真实对应的代码，用于代码对称，与 va_start 对应。

5> 怎样得到可变参数个数?归纳起来有三种办法:

(1) 函数的第一个参数,指定后续的参数个数,如 func(int num,...);
(2) 根据隐含参数,判断参数个数,如 printf 系列的,通过字符串中%的个数判断;
(3) 特殊情况下(如参数都是不大于 0xFFFF 的 int),可以一直向低处访问堆栈,直到返回地址。

有了上述知识我们就可以看懂 printf()函数的内容了，首先 va_start(args, fmt);会将可变参数的首地址保存在 args 中,然后调用 vsprintf(g_pcOutBuf,fmt,args)进行处理,在 vsprintf()中，会调用 va_arg()逐个的取出变参，然后进行解析。如果是普通字符则无须转换，直接保存在 g_pcOutBuf；如果是字符串，则从可变参数表中拿到指向字符串的指针，将字符串的内容拷贝到 g_pcOutBuf；如果是数字，则调用 number 函数进行处理，并把解析的结果存放在 g_pcOutBuf。所有，最后只调用 putC 函数把 g_pcOutBuf 里的字符一个个的打印出来就可以了。另外，scanf 函数的原理和 printf 类似，这里不再进行解释

2. main.c

完整代码如下:

```
int main()
{
    int a, b;
    uart_init();
    printf("hello, world\n\r");

    while (1)
    {
        printf("please enter two number: \n\r");
        scanf("%d %d", &a, &b);
        printf("\n\r");
        printf("the sum is: %d\n\r", a+b);
    }

    return 0;
}
```

首先会打印“hello，world”,然后终端会等待接收两个整型数字，最后输出它们的和。

第四节　编译代码和烧写运行

编译代码

在 Fedora 终端执行如下命令:

```
# cd 15.uart_stdio
# make
```

make 成功后会生成 stdio.bin 文件。

烧写运行

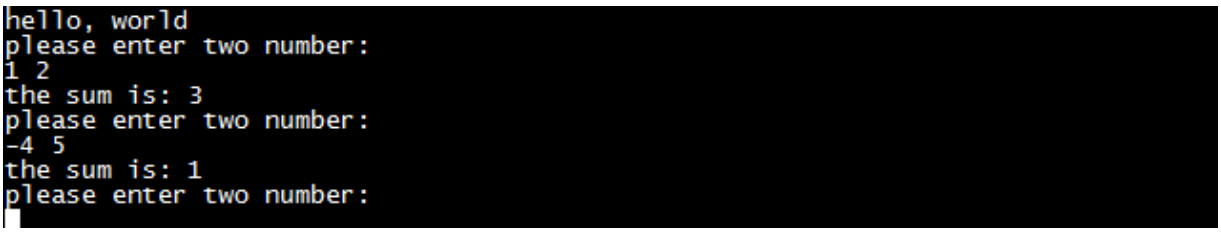
进入代码所在目录，编译成功后在 Fedora 终端执行如下命令：

```
# make burn
```

该命令会调用脚本 hc_fusing_mmcboot.sh 将 stdio.bin 烧写到 SDHC 卡上。

第五节　实验现象

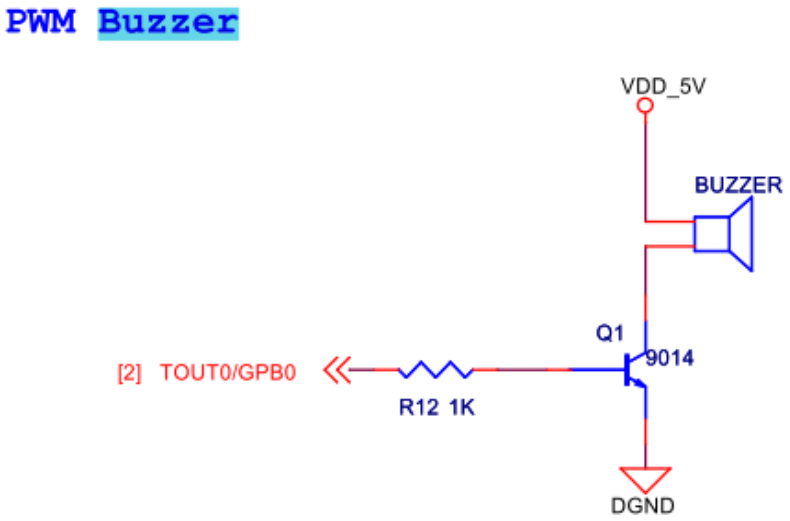
连接好串口终端后，开发板选择 SD 启动，然后上电，首先串口终端会打印“hello,world”，然后提醒你输入两个数字，成功输入两个数字后，会打印它们的和。效果图如下：



第十五章　控制蜂鸣器

第一节　查阅原理图

Mini2451 带有一个蜂鸣器，十分吵闹，本章将学习如何控制蜂鸣器。首先查阅原理图：



可以看出，蜂鸣器接在 GPB0 引脚上。

第二节　程序相关讲解

完整代码见目录 16.buzzer。蜂鸣器的操作十分简单，原理跟操作 LED 一样，都是操控 GPIO 引脚。

1. buzzer.c

完整代码如下：

```
#include "buzzer.h"
#define    GPBCON        (*(volatile unsigned long *)0x56000010)
#define    GPBDAT        (*(volatile unsigned long *)0x56000014)
void buzzer_on()
{
    GPBDAT |= 1<<0;
}

void buzzer_off()
{
    GPBDAT &= ~(1<<0);
```

```
}
void buzzer_init(void)
{
    GPBCON |= 1<<0;
    GPBCON &= ~(1<<1);
}
```

函数 buzzer_init()配置 GPIO 引脚，使 GPF14 用于输入功能；
函数 buzzer_on()使 GPF14 输出 0，蜂鸣器响；
函数 buzzer_on()使 GPF14 输出 1，蜂鸣器不响；

2. main.c

在 main.c 中，首先会调用 buzzer_init()来初始化蜂鸣器，然后打印出蜂鸣器的控制菜单并等待用户输入选择，输入 n 则调用 buzzer_on()打开蜂鸣器，输入 f 则调用 buzzer_off 关闭蜂鸣器。

第三节　编译代码和烧写运行

编译代码

在 Fedora 终端执行如下命令：

```
# cd 16.buzzer
# make
```

make 成功后会生成 buzzer.bin 文件。

烧写运行

进入代码所在目录，编译成功后在 Fedora 终端执行如下命令：

```
# make burn
```

该命令会调用脚本 hc_fusing_mmcboot.sh 将 buzzer.bin 烧写到 SDHC 卡上。

第四节　实验现象

连接好串口终端后，开发板选择 SD 启动。首先会打印蜂鸣器的操作菜单，输入 n 打开蜂鸣器，输入 f 关闭蜂鸣器，测试效果如下图：

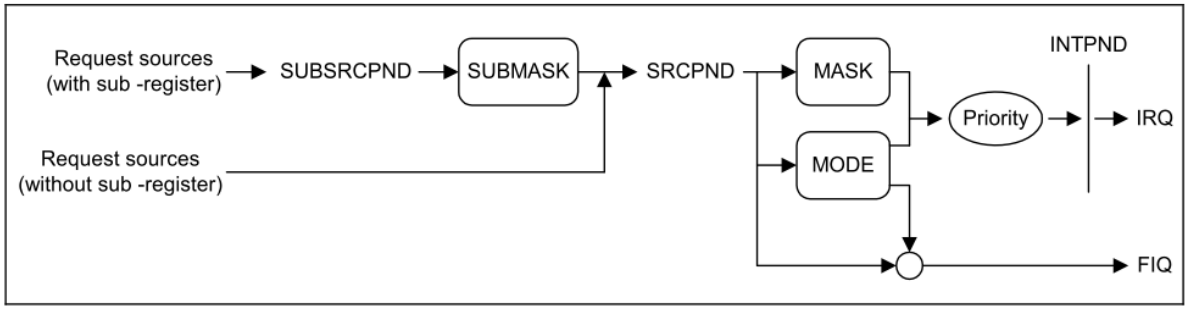
```
###############Buzzer Test###############
[n]buzzer on
[f]buzzer off
Enter your choice:n


###############Buzzer Test###############
[n]buzzer on
[f]buzzer off
Enter your choice:f
```

第十六章　中断控制器

第一节　S3C2451 的中断控制器

S3C2451 中的中断控制器接受来自 53 个中断源的请求。提供这些中断源的是内部外设，如 DMA 控制器、UART、IIC 等等。在这些中断源中，UARTn、AC97 和 EINTn 中断对于中断控制器而言是“或”关系。整体上中断处理过程如下图：



参考上图我们就可以设置好中断控制器，从而进行中断处理。

第二节　程序相关讲解

完整代码见目录 18.irq。

1. start.S

与中断相关的地方有：

1) 程序一开始时设置了中断向量表。当发生各种异常时，PC 会自动跳转到 0 地址处起始的相应的异常向量，所以我们需要把异常向量放在 0 地址处。异常向量表如下：

```
// 异常向量表
b reset                         /* 复位时,cpu 跳到 0 地址 */
b halt                          /* cpu 遇到不能识别的指令时 */
b halt                          /* swi 异常，进入 svc 模式 */
b halt                          /* 预取中止异常 */
b halt                          /* 数据访问异常 */
b halt                          /* 没用到 */
ldr pc, _irq                    /* 中断异常 */
b halt                          /* 快中断异常 */
```

2) 调用 irq_init()函数进行中断相关的初始化，该函数的定义位于 irq.c，后面会细说；

3) 开中断，设置 CPSR 寄存器，允许中断发生；代码如下：

```
mov r0, #0x53
msr CPSR_cxsf, r0
```

4) 初始化 mmu。由于 IROM 里的固化代码是将我们的程序拷贝到 SRAM 的 0x40000000 处，所以这里需要我们通过 mmu 将 0x40000000 起的 1M 的物理地址映射到虚拟地址 0x0 上，后面讲解 mmu.c 时会详细说明。

5) 中断处理，代码如下：

```
irq:
    ldr sp, =(0x31000000+0x100000)      /* 设置中断模式下的栈 */
    sub lr, lr, #4                      /* 保存现场 */
    stmdb sp!, {r0-r12, lr}
    bl do_irq                           /* 处理异常 */
    ldmia sp!, {r0-r12, pc}^            /* 恢复现场,^表示把 spsr 恢复到 cpsr */
```

当发生 IRQ 中断异常时，会根据中断向量表里的设置，跳转到该部分代码。代码的作用是设置中断模式下的栈，然后保存现场,再调用中断处理函数 do_irq()，do_irq()处理完中断后再恢复现场。do_irq ()的定义位于 irq.c，后面会详细讲解。

2. mmu.c

核心代码如下：

```
for(i=0; i<=0x102; i++)                                 // 映射寄存器
{
    table[0x4a0+i] = (0x4a000000 + i * 0x100000) | MMU_SECDESC_NCNB;
}

for(i=0; i<0x40; i++)                                   // 映射内存
{
    table[0x300+i] = (0x30000000 + i * 0x100000) | MMU_SECDESC_WB;
}
table[0x400] = 0x40000000 | MMU_SECDESC_WB;
table[0x0]   = 0x40000000 | MMU_SECDESC_WB;             // 映射中断向量表
```

mmu.c 中初始化了 mmu，我们主要注意其建立了哪些页表项。主要建立了 4 类页表项:

1) 映射寄存器所在的地址空间，保持物理地址和虚拟地址一致；
2) 映射内存所在的地址空间，保持物理地址和虚拟地址一致；
3) 映射物理地址 0x40000000 起的 1M 空间到虚拟地址 0x40000000,保持 MMU 使能前后的代码所在的空间的虚拟地址等于物理地址，目的是为了使能 mmuc 时，代码仍能正常的运行；
4) 映射物理地址 0x40000000 起的 1M 空间到虚拟地址 0x00000000, 目的是将 0x40000000 处的中断向量表映射到 0 地址处。

3. irq.c

中断初始化函数 irq_init()，代码如下：

```
void irq_init(void)
{
```

```
    GPFCON = 0xaaaa;                    // 配置 GPF 为中断功能
    EXTINT0 |= ( (0x7<<0) | (0x7<<4) | (0x7<<8) | (0x7<<12) );// 双边沿触发
    SRCPND |= 0x1f;                     // 清中断
    INTPND |= 0x1f;
    INTMSK &= ~(0xf<<0);
}
```

该函数初始化了 KEY1~4 的中断，主要步骤就是：

1) 配置引脚用于中断功能；
2) 设置中断的触发方式；
3) 清中断；
4) 取消屏蔽中断；

中断处理函数 do_irq()，代码如下：

```
void do_irq(void)
{
    int i = 0;
    // 分辨是哪个中断
    for (i = 0; i < 4; i ++)
    {
        if (SRCPND & (1<<i))
        {
            if (GPFDAT & (1<<i))
            {
                printf("K%d released\n\r", i+1);
            }
            else
            {
                printf("K%d pressed\n\r", i+1);
            }
        }
    }
    // 清中断
    SRCPND = SRCPND;
    INTPND = INTPND;
}
```

主要步骤如下：

1) 判断是哪个按键发生了中断；
2) 在串口终端打印信息；
3) 清中断，这样 CPU 才不会认为中断又发生了；

4. main.c

与前面的章节无异，就是计算两个数的和。当没中断发生时，就执行 main()里的代码，当发生中断时，则跳去执行中断处理代码，中断处理完毕后，则继续执行 main()里的代码；

第三节　编译代码和烧写运行

编译代码

在 Fedora 终端执行如下命令：

```
# cd 18.irq
# make
```

make 成功后会生成 irq.bin 文件。

烧写运行

进入代码所在目录，编译成功后在 Fedora 终端执行如下命令：

```
# make burn
```

该命令会调用脚本 hc_fusing_mmcboot.sh 将 irq.bin 烧写到 SDHC 卡上。

第四节　实验现象

连接好串口终端后，开发板选择 SD 启动。当未按下任何按键时，程序和前面的章节一样正常地计算两个数的和，当按下 KEY1~4 任意一个按键时，则会进行中断处理，处理过程只是简单的打印信息，效果如下：

```
please enter two number:
1 2
the sum of 1 + 2 is: 3
please enter two number:
K1 pressed
K1 pressed
K2 pressed
K2 released
K3 pressed
K3 released
```

第十七章　PWM 定时器

第一节　S3C2451 的 PWM 定时器

S3C2451 由五个 32 位定时器组成。这些计时器用来产生内部中断到 ARM 子系统。此外，定时器 0，1，2 和 3 包含一个 PWM 功能（脉宽调制），它可以驱动外部的 I/O 信号。定时器 0 上的 PWM 能够产生一个可选的死区发生器。它可能被用来支持大量的通用装置。定时器 4 只是一个没有输出引脚的内部定时器。相关的结构图如下：

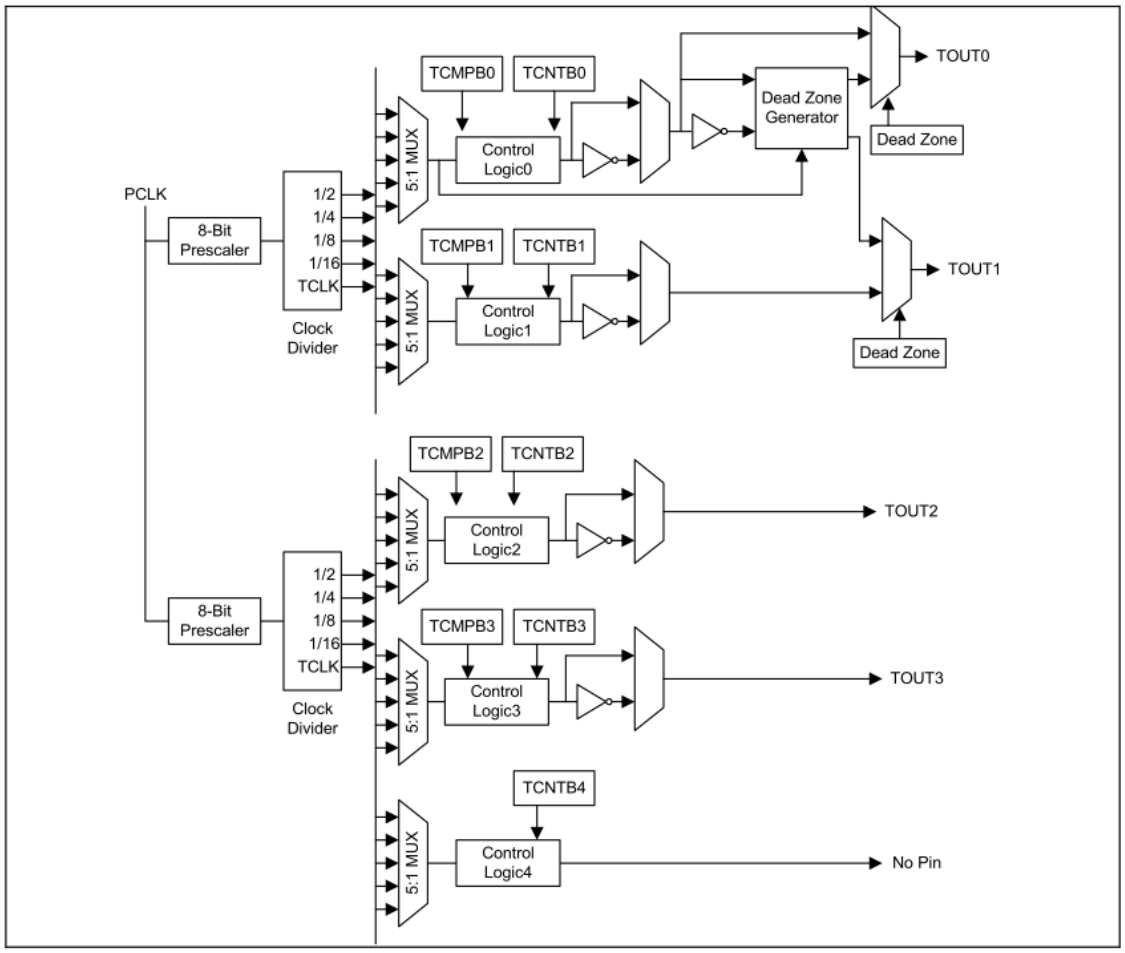


Figure 12-1. 16-bit PWM Timer Block Diagram

第二节　程序相关讲解

完整代码见目录 19.timer。

1. main.c

相关的两个步骤：

1） 调用 timer_init()初始化 Timer0；

2） 死循环，不断地打印数字；

2. timer.c

1> 中断初始化函数 irq_init()

在中断控制器里使能 Timer0 中断，通过往 INTMSK 的 bit10 置 0 即可。

2> Timer0 初始化函数 timer_init()

主要包括 3 个步骤：

1） 设置预分频

首先设置预分频系数为 65，相关寄存器为 TFG0,如下：

Register	Address	R/W	Description	Reset Value
TCFG0	0x51000000	R/W	Configures the two 8-bit prescalers	0x00000000

TCFG0	Bit	Description	Initial State
Reserved	[31:24]		0x00
Dead zone length	[23:16]	These 8 bits determine the dead zone length. The 1 unit time of the dead zone length is equal to that of timer 0.	0x00
Prescaler 1	[15:8]	These 8 bits determine prescaler value for Timer 2, 3 and 4.	0x00
Prescaler 0	[7:0]	These 8 bits determine prescaler value for Timer 0 and 1.	0x00

然后设置分配比为 16 分频，相关寄存器为 TCFG1,如下：

Register	Address	R/W	Description	Reset Value
TCFG1	0x51000004	R/W	5-MUX & DMA mode selection register	0x00000000

TCFG1	Bit	Description	Initial State
Reserved	[31:24]		00000000
DMA mode	[23:20]	Select DMA request channel 0000 = No select (all interrupt) 0001 = Timer0 0010 = Timer1 0011 = Timer2 0100 = Timer3 0101 = Timer4 0110 = Reserved	0000
MUX 4	[19:16]	Select MUX input for PWM Timer4. 0000 = 1/2 0001 = 1/4 0010 = 1/8 0011 = 1/16 01xx = External TCLK	0000
MUX 3	[15:12]	Select MUX input for PWM Timer3. 0000 = 1/2 0001 = 1/4 0010 = 1/8 0011 = 1/16 01xx = External TCLK	0000
MUX 2	[11:8]	Select MUX input for PWM Timer2. 0000 = 1/2 0001 = 1/4 0010 = 1/8 0011 = 1/16 01xx = External TCLK	0000
MUX 1	[7:4]	Select MUX input for PWM Timer1. 0000 = 1/2 0001 = 1/4 0010 = 1/8 0011 = 1/16 01xx = External TCLK	0000
MUX 0	[3:0]	Select MUX input for PWM Timer0. 0000 = 1/2 0001 = 1/4 0010 = 1/8 0011 = 1/16 01xx = External TCLK	0000

经过上面的设置之后，就能确定 Timer0 的输入时钟了，计算方式如下：Timer Input Clock Frequency = PCLK / ({prescaler value + 1}) / {divider value} = 66MHz/(65+1)/16=62500Hz

2） 设置计数

设置寄存器 TCNTB0=62500 和 TCMPB0=0，启动 Timer0 后，TCNTB0 会逐渐减一，直到为 0 时就产生一次中断，即 1 秒产生一次 Timer0 中断。

Register	Address	R/W	Description	Reset Value
TCNTB0	0x5100000C	R/W	Timer 0 count buffer register	0x00000000
TCMPB0	0x51000010	R/W	Timer 0 compare buffer register	0x00000000

TCMPB0	Bit	Description	Initial State
Timer 0 compare buffer register	[15:0]	Set compare buffer value for Timer 0	0x00000000

TCNTB0	Bit	Description	Initial State
Timer 0 count buffer register	[15:0]	Set count buffer value for Timer 0	0x00000000

3） 启动 Timer0

设置寄存器 TCON，先设置手动更新位，然后清除手动更新位，使用自动装载，最后启动 timer0

3> 中断处理函数 do_irq()

共 2 个步骤：

1） 打印信息；

2） 清 Timer0 的中断；

第三节　编译代码和烧写运行

编译代码

在 Fedora 终端执行如下命令：

```
# cd 19.timer
# make
```

make 成功后会生成 timer.bin 文件。

烧写运行

进入代码所在目录，编译成功后在 Fedora 终端执行如下命令：

```
# make burn
```

该命令会调用脚本 hc_fusing_mmcboot.sh 将 timer.bin 烧写到 SDHC 卡上。

第四节　实验现象

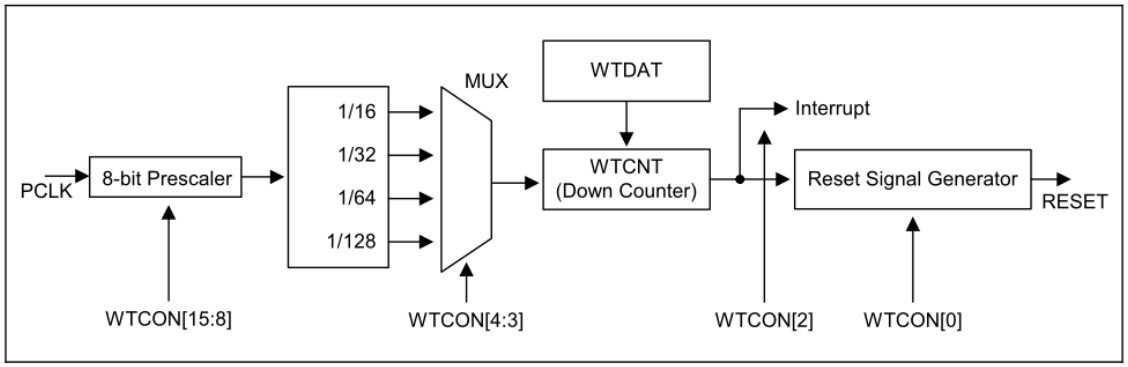
连接好串口终端后，开发板选择 SD 启动。Timer0 中断没发生时，main()会不停地打印数字，当 Timer 中断发生时，就会调用中断处理函数打印“Timer0 interrupt occur”效果如下：

```
#############timer test#############
0
1
2
3
1 Timer0 interrupt occur
4
5
6
2 Timer0 interrupt occur
7
8
```

第十八章　看门狗定时和复位

第一节　S3C2451 的看门狗定时器

S3C2451 上的看门狗定时器相当于一个普通的 16bit 的定时器，它与 PWM 定时器的区别是看门狗定时器可以产生 RESET 信号而 PWM 定时器不能，S3C2451 看门狗定时器的结构图如下：



第二节　程序相关讲解

完整代码见目录 20.watchdog。

1. main.c

需要注意两点：

1) 调用 wtd_operate()初始化看门狗；
2) 死循环，不断地打印数字；

2. watchdog.c

1> 中断初始化函数 irq_init()

在中断控制器里使能看门狗中断，代码如下：

```
    INTMSK &= (~(1<<9));
    INTSUBMASK &= (~(1<<27));
```

2> 看门狗初始化函数 wtd_operate()

wtd_operate()的完整代码如下：

```
void wtd_operate(unsigned long uenreset, unsigned long uenint, unsigned long uselectclk,
unsigned long uenwtd, unsigned long uprescaler,unsigned long uwtdat, unsigned long
uwtcnt)
{
    WTDAT = uwtdat;
```

```
    WTCNT = uwtcnt;
    WTCON= (uenreset<<0)|(uenint<<2)|(uselectclk<<3)|(uenwtd<<5)|((uprescaler)<<8);
}
```

首先设置计数相关的寄存器 WTDAT 和 WTCNT，寄存器 WTDAT 用来决定看门狗定时器的超时周期，在看门狗定时器启动后，寄存器 WTDAT 的值会自动传入寄存器 WTCNT，当 WTCNT 计数达到 0 时:如果中断被使能的话会发出中断，如果 RESET 功能被使用的话会发出复位信号，然后装载 WTDAT 的值并重新计数。

Register	Address	R/W	Description	Reset Value
WTDAT	0x53000004	R/W	Watchdog timer data register	0x8000

WTDAT	Bit	Description	Initial State
Count reload value	[15:0]	Watchdog timer count value for reload.	0x8000

Register	Address	R/W	Description	Reset Value
WTCNT	0x53000008	R/W	Watchdog timer count register	0x8000

WTCNT	Bit	Description	Initial State
Count value	[15:0]	The current count value of the watchdog timer	0x8000

寄存器 WTCON 进行相关配置，用来决定是否使能 RESET、是否使能中断、分频、是否启动定时器等，具体见下图：

Register	Address	R/W	Description	Reset Value
WTCON	0x53000000	R/W	Watchdog timer control register	0x8021

WTCON	Bit	Description	Initial State
Prescaler value	[15:8]	Prescaler value. The valid range is from 0 to 255(28-1).	0x80
Reserved	[7:6]	Reserved. These two bits must be 00 in normal operation.	00
Watchdog timer	[5]	Enable or disable bit of Watchdog timer. 0 = Disable 1 = Enable	1
Clock select	[4:3]	Determine the clock division factor. 00 = 16 01 = 32 10 = 64 11 = 128	00
Interrupt generation	[2]	Enable or disable bit of the interrupt. 0 = Disable 1 = Enable	0
Reserved	[1]	Reserved. This bit must be 0 in normal operation.	0
Reset enable/disable	[0]	Enable or disable bit of Watchdog timer output for reset signal. 1 = Assert reset signal of the S3C2416 at watchdog time-out 0 = Disable the reset function of the watchdog timer.	1

3> 中断处理函数 do_irq()

共 2 个步骤：

1) 打印信息；

2) 清中断；

3) 当发生了 5 次定时中断后，使能看门狗的 RESET 功能，此时系统会重启。

第三节　编译代码和烧写运行

编译代码

在 Fedora 终端执行如下命令：

```
# cd 20.watchdog
# make
```

make 成功后会生成 wtd.bin 文件。

烧写运行

进入代码所在目录，编译成功后在 Fedora 终端执行如下命令：

```
# make burn
```

该命令会调用脚本 hc_fusing_mmcboot.sh 将 wtd.bin 烧写到 SDHC 卡上。

第四节　实验现象

连接好串口终端后，开发板选择 SD 启动。首先会不断地打印 1、2、3、4...，当发生看门狗定时中断时，则打印“Watchdog interrupt occur”，当发生了 5 次看门狗定时中断后，看门狗复位功能会被使能，接着系统就会重启，效果如下图：

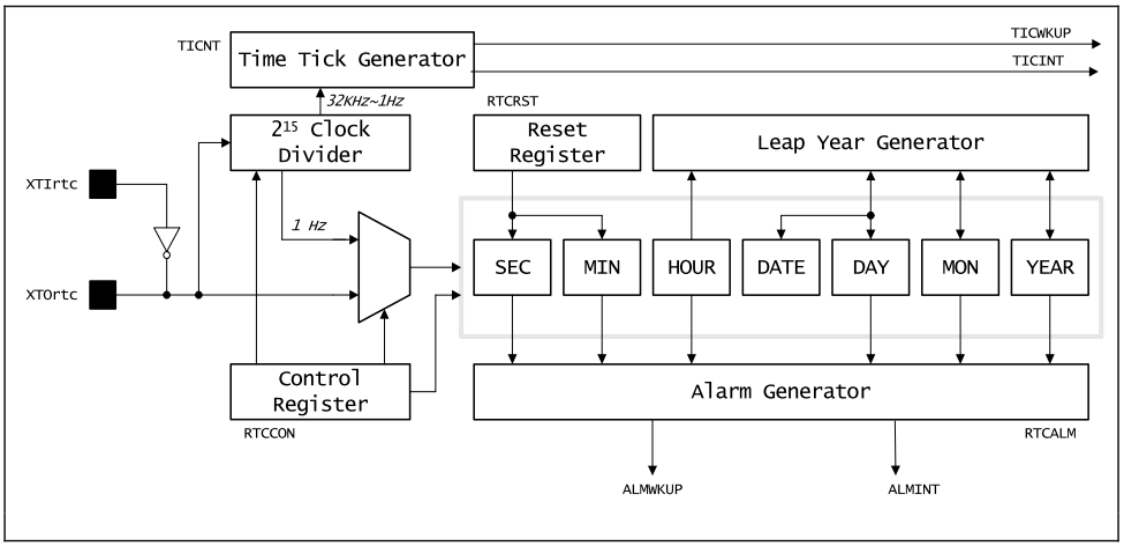
第十九章 RTC 读写时间

第一节 S3C2451 的 RTC

RTC 就是实时时钟芯片，用来在系统断电时，利用备用的锂电池继续记录时间。S3C2451 的 RTC 主要有如下特点：

1) 二进制编码数据：秒，分钟，小时，日期，日，月和年；
2) 闰年发生器；
3) 报警功能：报警中断或从断电模式中唤醒；
4) 时钟计数功能：时钟节拍中断或从断电模式中唤醒；
5) 不存在千年虫问题；
6) 独立地电源引脚 (RTCVDD)；
7) 支持毫秒标记的时间中断信号，用于 RTOS 内核时间标记；

下面的 S3C2451 的 RTC 相关结构图：



第二节 程序相关讲解

完整代码见目录 17.rtc。

1. main.c

main()函数中打印了一个菜单，共两个选择，一是打印时间，调用了函数 rtc_realtime_display()；二是重置时间，调用了函数 rtc_settime(),这两个函数的定义均位于 rtc.c；

2. watchdog.c

1> 打印时间函数 rtc_realtime_display()

代码如下：

```
void rtc_realtime_display(void)
{
    int counter = 0;
    rtc_enable(true);                       // 使能 rtc 控制器
    rtc_ticktime_enable(true);              // 使能 rtc tick timer

    while( (counter++) < 5)                 // 打印 5 次时间
    {
        rtc_print();
        delay(0x10000000/3);
    }

    rtc_ticktime_enable(false);             // 关闭 rtc 控制器
    rtc_enable(false);                      // 关闭 rtc tick timer
}
```

共 3 个步骤：

1) 使能 rtc 控制器，调用了函数 rtc_enable()，使能 rtc tick timer，调用了 rtc_ticktime_enable()。两个函数的实质都是设置寄存器 RTCCON，见下图：

Register	Address	R/W	Description	Reset Value
RTCCON	0x57000040	R/W	RTC control register	0x00

RTCCON	Bit	Description	Initial State
TICsel2	[8:5]	Tick Time clock select2. 0 = clock period of 1/16384 second select 1 = clock period of 1/8192 second select 2 = clock period of 1/4096 second select 3 = clock period of 1/2048 second select 4 = clock period of 1/128 second select 5 = clock period of 1 second select 6 = clock period of 1/1024 second select 7 = clock period of 1/512 second select 8 = clock period of 1/256 second select 9 = clock period of 1/64 second select 10 = clock period of 1/32 second select 11 = clock period of 1/16 second select 12 = clock period of 1/8 second select 13 = clock period of 1/4 second select 14 = clock period of 1/2 second select	0x0
TICsel	[4]	Tick Time clock select1. 0 = Clock period select at TICsel2 1 = Clock period of 1/32768 second	0
CLKRST	[3]	RTC clock count reset. 0 = No reset 1 = Reset	0
CNTSEL	[2]	BCD count select. 0 = Merge BCD counters 1 = Reserved (Separate BCD counters)	0
CLKSEL	[1]	BCD clock select. 0 = XTAL 1/215 divided clock 1 = Reserved (XTAL clock only for test)	0
RTCEN	[0]	RTC control enable. 0 = Disable　1 = Enable **Note:** Only BCD time count and read operation can be performed.	0

2） 打印 5 次时间，调用了函数 rtc_print()，其作用就是读寄存器 BCDYEAR、BCDMON 、BCDDATE、BCDHOUR、BCDMIN、BCDSEC、BCDDAY，分别获取了年、月、日、时、分、秒、星期几；

3） 关闭 rtc 控制器和 rtc tick timer，同样都是调用了函数 rtc_enable()和 rtc_ticktime_enable()，不过这次是关闭功能。

2> 设置时间函数 rtc_settime()

共 3 个步骤：

1） 给 year、month、date 等时间变量赋初始值；

2） 将时间转化为 BCD 码；

3） 保存 BCD 码到寄存器 BCDYEAR、BCDMON、BCDDATE 等；

第三节　编译代码和烧写运行

编译代码

在 Fedora 终端执行如下命令：

```
# cd 21.rtc
# make
```

make 成功后会生成 rtc.bin 文件。

烧写运行

进入代码所在目录，编译成功后在 Fedora 终端执行如下命令：

```
# make burn
```

该命令会调用脚本 hc_fusing_mmcboot.sh 将 rtc.bin 烧写到 SDHC 卡上。

第四节　实验现象

连接好串口终端后，开发板选择 SD 启动。串口终端会打印功能菜单，输入 s 复位时间，输入 d 显示当前时间，效果如下：

```
################rtc test###############
[d] Display rtc realtime(hour:min:sec:weekday date/month/year)
[s] Reset rtc realtime(12:0:0:Tuesday 1/1/2012)
Enter your choice:s
reset success

################rtc test###############
[d] Display rtc realtime(hour:min:sec:weekday date/month/year)
[s] Reset rtc realtime(12:0:0:Tuesday 1/1/2012)
Enter your choice:d
12 :  0 :  1     Tuesday,   5/ 1/2012
12 :  0 :  2     Tuesday,   5/ 1/2012
12 :  0 :  3     Tuesday,   5/ 1/2012
12 :  0 :  4     Tuesday,   5/ 1/2012
12 :  0 :  5     Tuesday,   5/ 1/2012
```

第二十章　LCD 绘图和打印字符

第一节　S3C2451 LCD 控制器

要使一块 LCD 正常显示文字或图像，不仅需要 LCD 驱动器，还需要相应的 LCD 控制器。LCD 控制器的主要作用是将定位在系统存储器中的显示缓冲区中的 LCD 图像数据传送到外部 LCD 驱动器，并产生必要的控制信号，例如 VSYNC、HSYSNC、VCLK。S3C2451 内部了 LCD 控制器，它结构图如下：

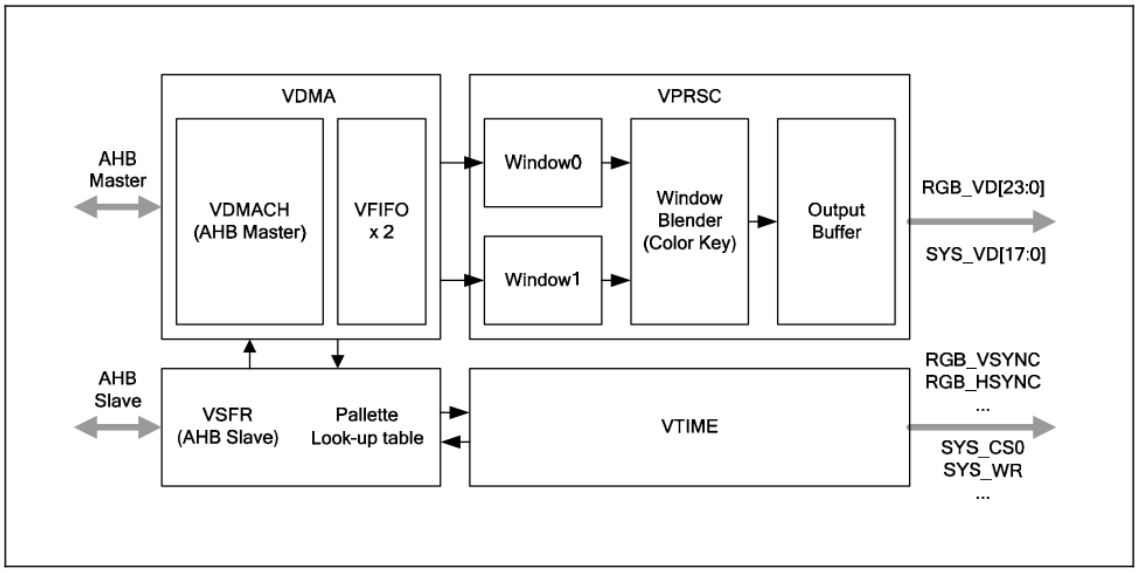


Figure 21-1. LCD Controller Block diagram

LCD 控制器由 VSFR,VDMA,VPRCS,VTIME 和视频时钟产生器组成。VSFR 包括可编程寄存器设置，用于配置显示控制器。VDMA 专用于显示 DMA，VDMA 可以将帧存储器内的视频数据转换到到 VPRCS。通过使用特殊的 DMA －VDMA，可以不使用 CPU，直接将视频数据显示在屏幕上。VPRCS 接收 VDMA 发出的视频数据，将其转换到适合的数据格式，如 8 位像素或 16 位像素后，将视频数据发送到显示设备上。VTIME 包括可编程逻辑，以支持在不同 LCD 驱动上发现的接口时序和速率的各种环境。VIIME 模块产生 RGB_VSYNC, RGB_HSYNC, RGB_VCLK, RBG_VDEN, SYS_CS, SYS_CS0 等等。

注意：本教程所测试时所用的 LCD 为友善之臂的 H43 和 P43，同样的参数设置对它们而言都是适用的。

第二节　程序相关讲解

完整代码见目录 22.lcd 和 23.lcd_putchar，我们先讲解 22.lcd。

1. main.c

需要注意下面两点：

1） LCD 初始化，调用了 lcd_init()；
2） 打印功能菜单,共 5 个选择，分别是清屏、画十字、画横向、画竖线、画圆。

2. lcd.c

1> LCD 初始化函数 lcd_init()

在 S3C2451 中，给出了 LCD 的初始化序列，见下图：

The following registers are used to configure LCD controller

1. VIDCON0: Configure Video output format and display enable/disable.
2. VIDCON1: RGB I/F control signal.
3. SYSIFCONx: i80-System I/F control signal.
4. VIDTCONx: Configure Video output Timing and determine the size of display.
5. WINCONx: Each window format setting
6. VIDOSDxA, VIDOSDxB: Window position setting
7. VIDOSDxC: Alpha value setting
8. VIDWxxADDx: Source image address setting
9. WxKEYCONx: Color key value register
10. WINxMAP: Window color control
11. WPALCON: Palette control register
12. WxPDATAxx: Window Palette Data of the each Index

参考上述序列，在 lcd.c 里函数 lcd_init()对 lcd 控制器进行了初始化，共 7 个步骤：
1） 配置相关引脚用于 LCD 功能，代码如下：

```
GPCCON = 0xAAAAAAAA;                    // 配置 GPIO 用于 LCD 相关的功能
GPDCON = 0xAAAAAAAA;
```

查看 Mini2451 的原理图可知，涉及到的 GPIO 引脚为 GPC、GPD。

2） 打开 LCD 电源和背光，代码如下：

```
GPBCON &= ~(0x3<<2);                    // 打开 LCD 电源
GPBCON |= (1<<2);
GPBDAT |= (1<<1);
GPGCON &= ~(0x3<<14);                   // 打开背光
GPGCON |= (1<<14);
GPGDAT |= (1<<7);
```

3） 配置 VIDCONx，设置接口类型、时钟、极性和使能 LCD 控制器等；
寄存器 VIDCON0：

Register	Address	R/W	Description	Reset Value
VIDCON0	0x4C800000	R/W	Video control 1 register	0x0000_0000

VIDCON0	Bit	Description	Initial State
Reserved	[31:24]	Reserved	0x00
VIDOUT	[23:22]	It determines the output format of LCD Controller 00 = RGB I/F 01 = Reserved 10 = i80-System I/F for Main LDI 11 = i80-System I/F for Sub LDI	0
L1_DATA16	[21:19]	Select the mode of output data format of i80-System I/F (Sub LDI.) (Only when, VIDOUT == 2'b11) 000 = 16-bit mode (16 bpp) 001 = 16 + 2 bit mode (18 bpp) 010 = 9 + 9 bit mode (18 bpp) 011 = 16 + 8 bit mode (24 bpp) 100 = 18-bit mode (18bpp)	000
L0_DATA16	[18:16]	Select the mode of output data format of i80-System I/F (Main LDI.) (Only when, VIDOUT == 2'b10) 000 = 16 bit mode (16 bpp) 001 = 16 + 2 bit mode (18 bpp) 010 = 9 + 9 bit mode (18 bpp) 011 = 16 + 8 bit mode (24 bpp) 100 = 18 bit mode (18bpp)	000
Reserved	[15]	Reserved	0
PNRMODE	[14:13]	Select the display mode.　(Where, VIDOUT == 2'b00) 00 = RGB Parallel format (RGB) 01 = RGB Parallel format (BGR) 10 = Serial Format (R->G->B) 11 = Serial Format (B->G->R) Select the display mode.　(Where, VIDOUT == 2'b1x) 00 = RGB Parallel format (RGB)	00

VIDCON0	Bit	Description	Initial State
CLKVALUP	[12]	Select CLKVAL_F Update timing control 0 = Always 1 = Start of a frame (Only once per frame)	0
CLKVAL_F	[11:6]	Determine the rates of VCLK. VCLK = (HCLK or LCD video Clock) / [CLKVAL+1] (CLKVAL ≥ 1)	0
VCLKEN	[5]	VCLK Enable Control 0 = Disable 1 = Enable	0
CLKDIR	[4]	Select the clock source as direct or divide using CLKVAL_F register. 0 = Direct clock (frequency of VCLK = frequency of Clock source) 1 = Divided using CLKVAL_F	0
CLKSEL_F	[3:2]	Select the Video Clock source 00 = HCLK 01 = LCD video Clock (from SYSCON EPLL) 10 = Reserved 11 = Reserved	0
ENVID	[1:0]	Video output and the LCD logics enable/disable control. 00 = Disable video signals and logics immediately. 01 = Reserved. 10 = Disable video signals and logics at the end of current frame. 11 = Enable video output and logics. ***Note :*** *If set to '10b' in the middle of displaying current frame, the value of ENVID is still '11b'. However, the LCD functions are disabled at the end of current frame and the value is changed to '10b'.*	0

- ENVID = 1，使能 LCD 控制器；
- CLKSEL_F = 0，选择时钟源为 HCLK；
- CLKDIR = 1，选择需要分频；
- VCLKEN = 1,使能 VCLK；
- CLKVAL_F = 9，分频系数为 9，即 VCLK = 133 / (9+1) = 13MHz；
- CLKVALUP = 0，总是选择 CLKVAL_F 来更新时序控制；
- PNRMODE = 0，RGB 并行；
- VIDOUT=0，使用 RGB 接口；
- 未设置的 bit 均使用默认值。

4) 配置 VIDTCONx，设置时序和长宽等；
在讲解如何设置寄存器之前，需要先讲讲 LCD 的时序设置，见下图：

1.3.11.2 LCD RGB Interface Timing

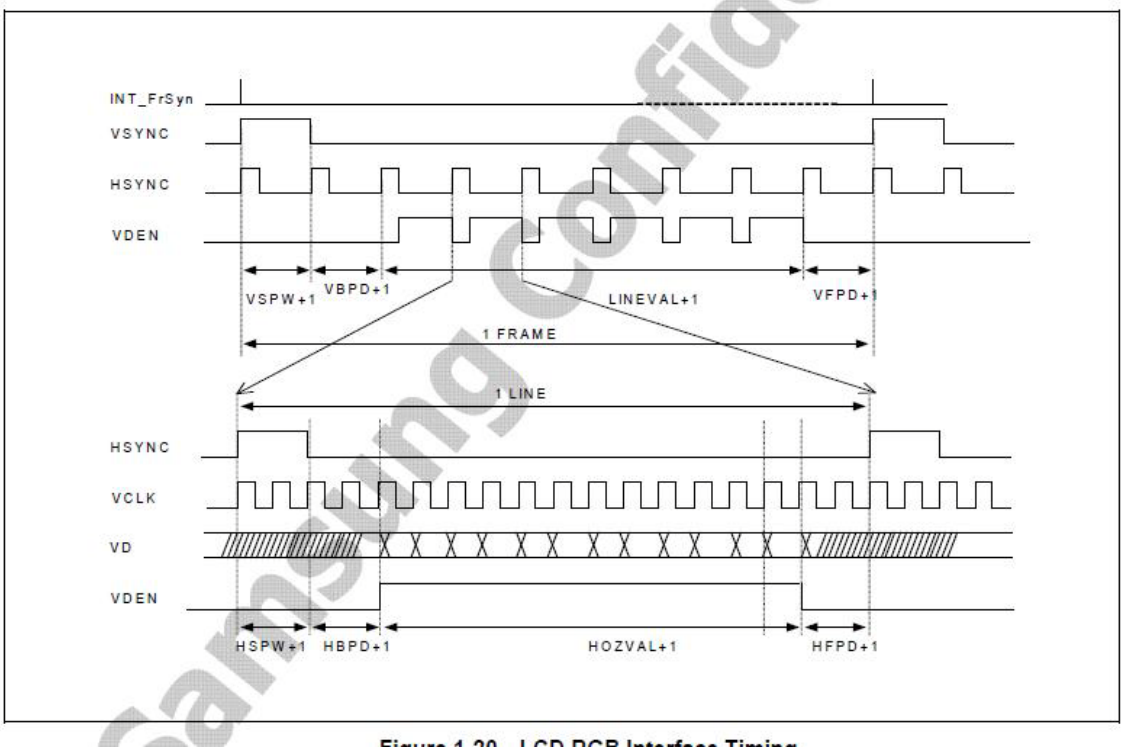


Figure 4-20 LCD RGB Interface Timing

其中各个时序的含义如下图：

- VBPD(vertical back porch)：表示在一帧图像开始时，垂直同步信号以后的无效的行数
- VFBD(vertical front porch)：表示在一帧图像结束后，垂直同步信号以前的无效的行数
- VSPW(vertical sync pulse width)：表示垂直同步脉冲的宽度，用行数计算
- HBPD(horizontal back porch)：表示从水平同步信号开始到一行的有效数据开始之间的 VCLK 的个数
- HFPD(horizontal front porth)：表示一行的有效数据结束到下一个水平同步信号开始之间的 VCLK 的个数
- HSPW(horizontal sync pulse width)：表示水平同步信号的宽度，用 VCLK 计算

每一帧的传输过程如下：
第一步：　VSYNC 信号有效时，信号宽度为(VSPW+1)个 HSNC 信号周期，即(VSPW+1)个无效行；
第二步：　VSYNC 信号脉冲之后，总共还要进过(VBPD+1)个 HSYNC 信号周期，有效的行数据才出现。所以,在 VSYNC 信号有效之后，还要进过(VSPW+1+VBPD+1)个无效的行；
第三步：　随即发出(LINEVAL+1)行的有效数据；
第四步：　最后是(VFPD+1)个无效的行；

每一行中像素的传输过程如下：

第一步：　HSNC 信号有效时，表示一行数据的开始，信号宽度为(HSPW+1)个 VCLK 信号周期，即(HSPW+1)个无效像素；
第二步：　HSYNC 信号脉冲之后，还要经过(HBPD+1)个 VCLK 信号周期，有效的像素数据才出现；
第三步：　随后发出(HOZVAL+1)个像素的有效数据；
第四步：　最后是(HFPD+1)个无效的像素；

有了上面这些知识，设置时序相关的寄存器 VIDTCON0 和 VIDTCON1 就没什么问题了，相关代码如下：

```
#define HSPW            (2)
#define HBPD            (40)
#define HFPD            (5)
#define VSPW            (2)
#define VBPD            (8)
#define VFPD            (8)
#define LINEVAL         (271)
#define HOZVAL          (479)
// 设置时序
VIDTCON0 = VBPD<<16 | VFPD<<8 | VSPW<<0;
VIDTCON1 = HBPD<<16 | HFPD<<8 | HSPW<<0;
// 设置长宽
VIDTCON2 = (LINEVAL << 11) | (HOZVAL << 0);
```

首先是寄存器 VIDTCON0,我们参考下面两张图：

Register	Address	R/W	Description	Reset Value
VIDTCON0	0x4C800008	R/W	Video time control 1 register	0x0000_0000

VIDTCON0	Bit	Description	Initial State
VBPD	[23:16]	Vertical back porch is the number of inactive lines at the start of a frame, after vertical synchronization period. (Period : VBPD +1)	0x00
VFPD	[15:8]	Vertical front porch is the number of inactive lines at the end of a frame, before vertical synchronization period. (Period : VFPD +1)	0x00
VSPW	[7:0]	Vertical sync pulse width determines the VSYNC pulse's level width by counting the number of inactive lines. (Period : VSPW +1)	0x00

6.3 Data Input Format

Parallel 24-bit RGB Input Timing Table

Parameters	Symbol	Min.	Typ.	Max.	Unit	Conditions
DCLK frequency	fclk	5	9	12	MHz	
VSYNC period time	Tv	277	288	400	Th	
VSYNC display area	Tvd		272		Th	
VSYNC back porch	Tvbp	3	8	31	Th	
VSYNC front porch	Tvfp	2	8	93	Th	
HSYNC period time	Th	520	525	800	DCLK	
HSYNC display area	Thd		480		DCLK	
HSYNC back porch	Thbp	36	40	255	DCLK	
HSYNC front porch	Thfp	4	5	65	DCLK	

- VFPD：Vertical front porch，单位是行。LCD 芯片手册时序图中 VSYNC front porch 就是 VFPD+1，因为时序图中 Th 表示一行需多少个 VCLK，所有 VFPD=8-1 行(Th)；
- VBPD：Vertical back porch，单位是行。LCD 芯片手册时序图中 VSYNC back porch 就是 VBPD+1，因为时序图中 Th 表示一行需多少个 VCLK，所有 VBPD=8-1 行(Th)；
- VSPW：Vertical sync pulse width，单位是行。LCD 芯片手册时序图中没直接给出它的值，需要自己计算：VSPW= VSYNC period time - VSYNC display area - VSYNC back porch - VSYNC front porch=288-272-8-8=0，所有 VSPW 设为 0 即可；

然后是寄存器 VIDTCON1，我们参考下面两张图：

Register	Address	R/W	Description	Reset Value
VIDTCON1	0x4C80000C	R/W	Video time control 2 register	0x0000_0000

VIDTCON1	Bit	Description	Initial state
HBPD	[23:16]	Horizontal back porch is the number of VCLK periods between the edge of HSYNC and the start of active data. (Period : HBPD +1) ***Note:*** *Set 0x10 for i80-System Interface* *When the PNRMODE (VIDCON0 [14:13]) is set to serial format the period becomes 3 times of HBPD value.* *(If HBPD is set to '0' in serial mode, the period becomes 3-VLCK)*	0000000
HFPD	[15:8]	Horizontal front porch is the number of VCLK periods between the end of active data and the edge of next HSYNC. (Period : HFPD +1) ***Note:*** *When the PNRMODE(VIDCON0[14:13]) is set to serial format the period of HFPD becomes 3 times of VCLK* *(If HFPD is set to '0' in serial mode, the period becomes 3-VLCK)*	0x00
HSPW	[7:0]	Horizontal sync pulse width determines the HSYNC pulse's level width by counting the number of the VCLK. (Period : HSPW +1) ***Note:*** *When the PNRMODE(VIDCON0[14:13]) is set to serial format the period of HSPW becomes 3 times of VCLK* *(If HSPW is set to '0' in serial mode, the period becomes 3-VLCK)*	0x00

6.3 Data Input Format

Parallel 24-bit RGB Input Timing Table

Parameters	Symbol	Min.	Typ.	Max.	Unit	Conditions
DCLK frequency	fclk	5	9	12	MHz	
VSYNC period time	Tv	277	288	400	Th	
VSYNC display area	Tvd		272		Th	
VSYNC back porch	Tvbp	3	8	31	Th	
VSYNC front porch	Tvfp	2	8	93	Th	
HSYNC period time	Th	520	525	800	DCLK	
HSYNC display area	Thd		480		DCLK	
HSYNC back porch	Thbp	36	40	255	DCLK	
HSYNC front porch	Thfp	4	5	65	DCLK	

- HBPD：Horizontal back porch，单位是 VCLK。LCD 芯片手册时序图中 HSYNC back porch 就是 HBPD +1，所有 HBPD=40-1(VCLK)；
- HFPD：Horizontal front porch，单位是 VCLK。LCD 芯片手册时序图中 HSYNC front porch 就是 HFPD +1，所有 VBPD=5-1(VCLK)；
- HSPW：Horizontal sync pulse width，单位是 VCLK。LCD 芯片手册时序图中没直接给出它的值，需要自己计算：HSPW+1 = HSYNC display area - HSYNC display area - HSYNC back porch - HSYNC front porch=525-480-40-5=0，所有 VSPW 设为 0 即可；

最后是寄存器 VIDTCON2，见下图：

Register	Address	R/W	Description	Reset Value
VIDTCON2	0x4C800010	R/W	Video time control 3 register	0x0000_0000

VIDTCON2	Bit	Description	Initial state
LINEVAL	[21:11]	These bits determine the vertical size of display	0
HOZVAL	[10:0]	These bits determine the horizontal size of display	0

- HOZVAL：HOZVAL = (Horizontal display size) -1=480-1=479
- LINEVAL：LINEVAL = (Vertical display size) - 1=272-1=271

5） 配置 WINCON0，设置 window0 的数据格式；
S3C2451 的 LCD 控制器有 overlay 功能，支持 2 个 window。这里我们只使用 window0，相关的代码设置如下：

```
WINCON0 |= 1<<0;
WINCON0 &= ~(0xf << 2);
WINCON0 |= 0xB<<2;
```

我们不需要使用很强大的功能，所有只需设置如下几个 bit 即可：

- ENWIN_F=1，使能；
- BPPMODE_F= 1011，24bpp；

6） 配置 VIDOSD0A/B/C,设置 window0 的坐标系，相关代码如下：

```
VIDOSD0A = (LeftTopX<<11) | (LeftTopY << 0);
VIDOSD0B = (RightBotX<<11) | (RightBotY << 0);
VIDOSD0C = (LINEVAL + 1) * (HOZVAL + 1);
```

设置 window0 的左上角和右下角的坐标和长宽。

7) 配置 VIDW00ADD0B0 和 VIDW00ADD1B0，设置 framebuffer 的地址，相关代码如下：

```
VIDW00ADD0B0 = FRAME_BUFFER;
VIDW00ADD1B0 = (((HOZVAL + 1)*4 + 0) * (LINEVAL + 1)) & (0xffffff);
```

2> LCD 描点函数 lcd_draw_pixel()

经过 lcd_init()初始化 LCD 控制器之后，我们就可以在 LCD 上描绘图形了，代码里的所有绘制图形的函数都是基于 lcd_draw_pixel()这个函数，它的作用是在 LCD 上描绘一个点，所有单独的点最终就可以组成图形了。在 LCD 上描点的本质就是往 FrameBuffer 中写入颜色值而已。lcd_draw_pixel()的完整代码如下：

```
void lcd_draw_pixel(int row, int col, int color)
{
    unsigned long * pixel = (unsigned long  *)FB_ADDR;
    *(pixel + row * COL + col) = color;
}
```

其中 FB_ADDR = 0x32f00000，即 framebuffer 的基地址。row 和 col 用来决定偏移，color 是颜色值，只要在 framebuffer 中对应的地址处写入颜色值就可以在 LCD 上描绘出该点。

到这里，代码 22.lcd 就讲解完了，我们现在来看 23.lcd_putchar：

1. lcd.c

相比 22.lcd，多了一个函数 lcd_draw_char()，其主要内容如下：

1) 获得字模。以传进来的参数为下标，从字模数组 fontdata_8x8 里取出对应的字模，每一个字模用 8*8bit 的数据来表示，数组 fontdata_8x8 的定义位于 font_8x8.c 中；

2) 是否需要回车换行。当遇到’\n’时表示换行，当遇到’\r’表示回车；

3) 在 16*16 个像素里描绘一个字符。font_8x8.c 里定义的每一个字模都是由 8*8bit 组成，每 1bit 对应一个像素，为 1 则描白点，为 0 则描黑点。但是为了看得更清楚，我们的代码把字体放大了两倍，即用 16*16 个像素来显示一个字符，具体方法参考代码。

4) 光标移动到下一个 16*16 的位置；

2. main.c

main()会打印一个菜单，这个菜单没有具体的功能，只是调用了 printf(),在 printf()里会调用 putc()，我们在 putc()最后面加了一行代码：lcd_draw_char()，把打印到终端的信息也同时打印到 LCD 上。

第三节　编译代码和烧写运行

编译代码

1） 对于程序 22.lcd
在 Fedora 终端执行如下命令：
cd 22.lcd
make
make 成功后会生成 lcd.bin 文件。

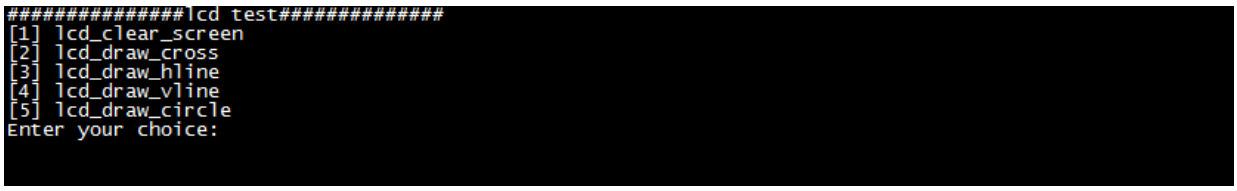
2） 对于程序 23.lcd_putchar
在 Fedora 终端执行如下命令：
cd 23.lcd_putchar
make
make 成功后会生成 lcd.bin 文件。

烧写运行

进入代码所在目录，编译成功后在 Fedora 终端执行如下命令：
make burn
该命令会调用脚本 hc_fusing_mmcboot.sh 将 lcd.bin 烧写到 SDHC 卡上。

第四节　实验现象

连接好串口终端后，开发板选择 SD 启动。对于代码 22.lcd， 首先在 PC 终端会打印如下菜单：



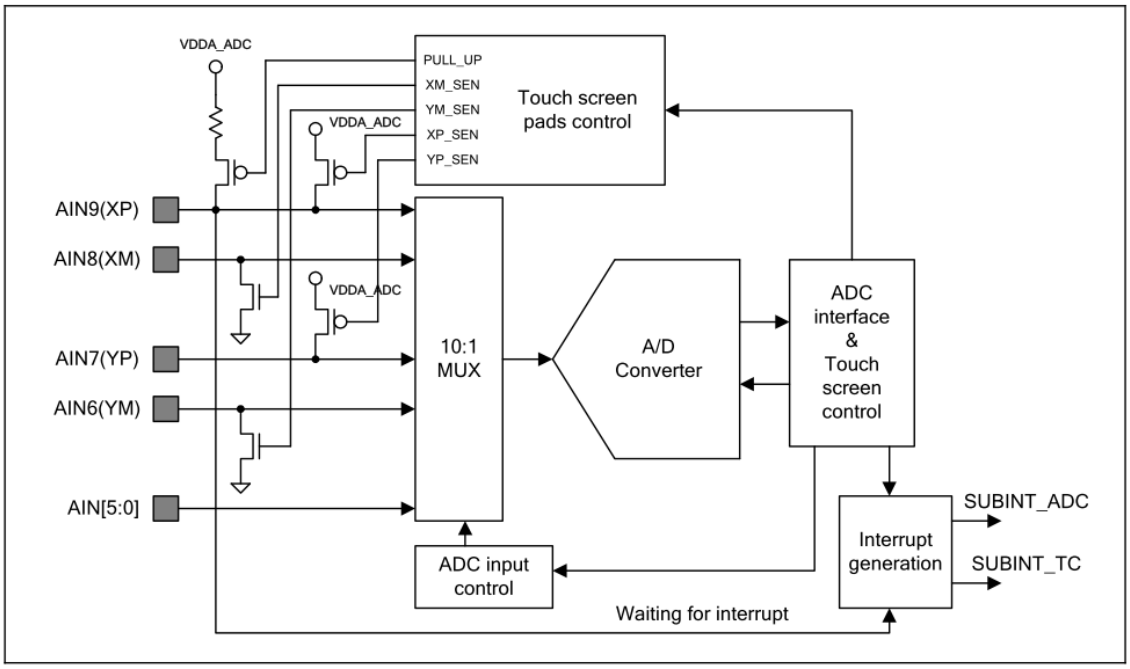
输入 1：LCD 会清屏；
输入 2：LCD 会在左上角打印一个十字；
输入 3：LCD 会在正中间画一条横线；
输入 4：LCD 会在正中间画一条竖线；
输入 5：LCD 会出现同心圆；

对于代码 23.lcd_putchar，在串口终端和 LCD 屏幕中，都会打印出功能菜单：

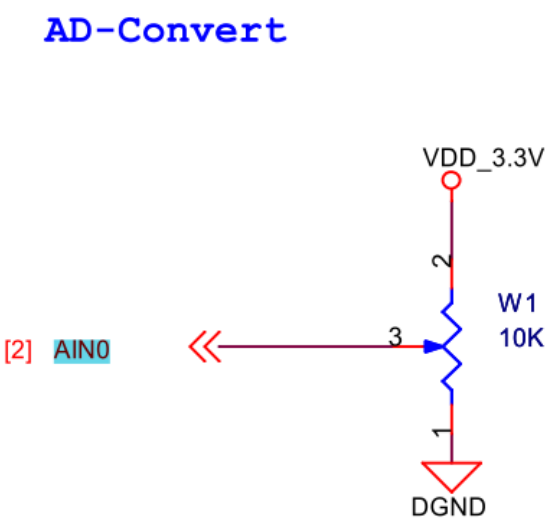
第二十一章　测试 ADC 转换

第一节　S3C2451 的 ADC

S3C2451 的 ADC 是 12 位 CMOS 模/数转换器。它是一个 10 通道模拟输入的再循环类型设备。其转换模拟输入信号为 12 位二进制数字编码，最大转换率为 5MHz A/D 转换器时钟下的 1MSPS 。本章只是涉及到初步的 ADC 转换，并不会讲解触摸屏相关知识，所有我们暂时把触摸屏的相关知识屏蔽。其结构图如下：



在 Mini2451 中，ADC 相关的原理图如下：



通道 0 的输入被接到可调电阻上，通过调节可变电阻，ADC 能转换出不同的值。

第二节　程序相关讲解

完整代码见目录 24.adc。

1. main.c

main 函数很简单，就是在一个死循环里不断地的读取 ADC 转换的值。

2. adc.c

核心函数是 read_adc(),它主要包括 5 个步骤：

1) 设置时钟。相关代码如下：

```
ADCCON = (1<<16)|(1 << 14) | (65 << 6) | (ch << 3);
```

ADCCON 寄存器

Register	Address	R/W	Description	Reset Value
ADCCON	0x58000000	R/W	ADC control register	0x3FC4

ADCCON	Bit	Description	Initial State
ECFLG	[15]	End of conversion flag (read only). 0 = A/D conversion in process 1 = End of A/D conversion	0
PRSCEN	[14]	A/D converter prescaler enable. 0 = Disable 1 = Enable	0
PRSCVL	[13:6]	A/D converter prescaler value. Data value: 5 ~ 255 Note that division factor is (N+1) when the prescaler value is N. **Note:** ADC frequency should be set less than PCLK by 5 times. (Ex. PCLK = 10MHz, ADC Frequency < 2MHz)	0xFF
Reserved	[5:4]	Reserved	0
RESSEL	[3]	A/D converter resolution selection 0 = 10-bit resolution 1 = 12-bit resolution	0
STDBM	[2]	Standby mode select. 0 = Normal operation mode 1 = Standby mode	1
READ_ START	[1]	A/D conversion starts by read. 0 = Disable start by read operation 1 = Enable start by read operation	0
ENABLE_ START	[0]	A/D conversion starts by setting this bit. If READ_START is enabled, this value is not valid. 0 = No operation 1 = A/D conversion starts and this bit is cleared after the start-up.	0

- SEL_MUX = 0，选择通道 0；
- PRSCVL = 65，时钟预分频系数为 65；
- PRSCEN = 1，使能预分频；
- RESSEL = 1，使用 12bit 的 ADC；

2) 设置模式，代码如下：
```
ADCCON &= ~((1<<2)|(1<<1));
```
设为普通转换模式,禁止 read start。

3) 启动转换。代码如下：
```
ADCCON |= (1 << 0);
```

4) 检查转换是否完成。代码如下：
```
while (ADCCON & (1 << 0));
```

5) 读数据，代码如下：

```
return (ADCDAT0 & 0xfff);
```

由于我们使用的 12bit 的模式，所有读寄存器 ADCDAT0 的前 12bit。

第三节　编译代码和烧写运行

编译代码

在 Fedora 终端执行如下命令：

```
# cd 24.adc
# make
```

make 成功后会生成 adc.bin 文件。

烧写运行

进入代码所在目录，编译成功后在 Fedora 终端执行如下命令：

```
# make burn
```

该命令会调用脚本 hc_fusing_mmcboot.sh 将 adc.bin 烧写到 SDHC 卡上。

第四节　实验现象

PC 终端上会不断的打印出数字，数字的范围是 0~4096，这是因为我们使用的是 12bit 的 ADC。有了 ADC 转换的基础，也就为触摸屏的学习打下了基础。

```
##################adc test################
adc = 0
adc = 0
adc = 36
adc = 126
adc = 585
adc = 911
adc = 1823
adc = 2013
adc = 2363
adc = 3372
adc = 3624
adc = 3664
adc = 4081
adc = 4082
adc = 4078
adc = 4084
adc = 4072
adc = 4084
adc = 4088
adc = 4081
adc = 4075
adc = 4082
adc = 4079
adc = 4081
adc = 4086
adc = 4075
adc = 4082
adc = 4073
adc = 4082
```

第二十二章　增加命令功能

第一节　关于命令功能

这里所说的命令功能类似 linux 中的 shell，输入一个命令，然后程序开始解析运行。这里我们只是象征性的实现几个简单的命令，包括：

1) help-提供帮助信息
2) md-memory display 显示内存
3) mw-memory write 写内存
4) loadb-通过串口下载 bin 文件到内存
5) go-运行内存中的 bin 文件

第二节　程序详细讲解

完整代码见目录 25.shell。

1. main.c

main()函数核心代码如下：

```
    while (1)
    {
        printf("Mini2451: ");
        gets(buf);                              // 等待用户输入命令
        argc = shell_parse(buf, argv);          // 解析命令
        command_do(argc, argv);                 // 运行命令
    }
```

注释已经一目了然了。函数 shell_parse()的定义位于 shell.c，command_do()的定义位于 command.c 中。

2. shell.c

shell_parse()核心代码如下：

```
    while (*buf)
    {
        if (*buf != ' ' && state == 0)      // 获得一个单词
        {
            argv[argc++] = buf;
            state = 1;
        }
        if (*buf == ' ' && state == 1)      // 跳过空格
```

```
        {
            *buf = '\0';
            state = 0;
        }
        buf++;
    }
```

逐个解析 buf 里的字符，结果保存在 argc 和 argv 中。例如输入命令：go 0x51000000，最终解析的结果是 argc = 2,argv[0] = “go”,argv[1] = “0x51000000”。

3. command.c

首先来看 command_do()，其大致代码如下：

```
int command_do(int argc, char * argv[])
{
    if (argc == 0)
        return -1;
    if (strcmp(argv[0], "help") == 0)          // 执行 help 命令
        help(argc, argv);
    …                                          // 省略代码
    if 2(strcmp(argv[0], "loadb") == 0)        // 执行 loadb 命令
        loadb(argc, argv);
    …                                          // 省略代码
}
```

根据不同的命令，调用不同的执行函数。例如输入 help 命令，调用的执行函数是 help()函数。下面来简单地解释各个命令的执行函数：

int help(int argc, char * argv[])：打印帮助信息。

int md(int argc, char * argv[])：读内存，读写内存都只是简单的指针操作

int mw(int argc, char * argv[])：写内存

int loadb(int argc, char * argv[])：从串口下载 bin 文件到内存，先获得文件大小，再获得下载地址，最后利用 getc()函数一个字节一个字节的接受 bin 文件。

int go(int argc, char * argv[])：执行内存中的 bin 文件，先定义一个函数指针，然后赋值调用。

第三节　编译代码和烧写运行

编译代码

在 Fedora 终端执行如下命令：

```
# cd 25.shell
# make
```

make 成功后会生成 shell.bin 文件。

烧写运行

进入代码所在目录，编译成功后在 Fedora 终端执行如下命令：

```
# make burn
```

该命令会调用脚本 hc_fusing_mmcboot.sh 将 shell.bin 烧写到 SDHC 卡上。

第四节　实验现象

PC 串口终端上会打印出“Mini2451”，输出 help 命令会打印帮助信息：

```
Tiny2416: help
help usage:
md - memory dispaly
mw - memory write
loadb - loadb filesize addr
go - go addr
Tiny2416: md
 0: ea000006 ea000022 ea000021 ea000020
10: ea00001f ea00001e ea00001d ea00001c
20: e3a00453 e3a01000 e5801000 e59fd064
30: eb00002a e28f0074 eb00004a e3a00053
40: e12ff000 e24f004c e59f104c e59f204c
50: e0422001 e1500001  a000002 eb0000a8
60: e3500000 1a00000a e59f0030 e59f1030
70: e3a03000 e1500001  a000003 e4803004
80: e1500001 1afffffc eb000158 e59fd014
90: e59ff014 eafffffe 40002000 30000000
a0: 30002304 30002b08 32100000 300005a8
b0:    4920d  4000040   57003a 80000030
c0:      313     1941 61656100  1006962
d0:        f   543405  1080206      109
e0: e3a03313 e5932024 e3c22c1e e3c22037
f0: e3822f89 e3822001 e5832024 e59f2018
Tiny2416:
```