



合肥工业大学

嵌入式系统原理

实 验 指 导 书

合肥工业大学计算机与信息学院

《嵌入式系统原理》课程组

2024年6月

目 录

实验一 汇编点亮 LED.....	1
一、实验目的.....	1
二、实验内容.....	1
三、预备知识.....	1
四、实验设备及工具（包括软件调试工具）	1
五、实验原理.....	1
六、实验步骤.....	2
七、实验修改要求.....	6
实验二 查询方式检测按键.....	7
一、实验目的.....	7
二、实验内容.....	7
三、预备知识.....	7
四、实验设备及工具（包括软件调试工具）	7
五、实验原理.....	7
六、实验步骤.....	8
七、实验修改要求.....	11
实验三 按键中断实验.....	12
一、实验目的.....	12
二、实验内容.....	12
三、预备知识.....	12
四、实验设备及工具（包括软件调试工具）	12
五、实验原理.....	12
六、实验步骤.....	13
七、实验修改要求.....	18
实验四 PWM 定时器实验	19
一、实验目的.....	19
二、实验内容.....	19
三、预备知识.....	19
四、实验设备及工具（包括软件调试工具）	19
五、实验原理.....	19
六、实验步骤.....	20
七、实验修改要求.....	25
实验五 串行口通信实验.....	26
一、实验目的.....	26
二、实验内容.....	26
三、预备知识.....	26
四、实验设备及工具（包括软件调试工具）	26
五、实验原理.....	26
六、实验步骤.....	28
七、实验修改要求.....	31
实验六 熟悉 Linux 开发环境	32
一、实验目的.....	32

二、实验内容.....	32
三、预备知识.....	32
四、实验设备及工具（包括软件调试工具）	32
五、实验步骤.....	32
六、实验修改要求.....	39
实验七 多线程应用程序设计.....	40
一、实验目的.....	40
二、实验内容.....	40
三、预备知识.....	40
四、实验设备及工具（包括软件调试工具）	40
五、实验原理.....	40
六、实验步骤.....	46
实验八 串行端口程序设计.....	49
一、实验目的.....	49
二、实验内容.....	49
三、预备知识.....	49
四、实验设备及工具（包括软件调试工具）	49
五、实验原理.....	49
六、程序分析.....	51
七、实验步骤.....	55
实验九 Linux Qt 开发实验	57
一、实验目的.....	57
二、实验内容.....	57
三、预备知识.....	57
四、实验设备及工具（包括软件调试工具）	57
五、实验步骤.....	57

实验一 汇编点亮 LED

一、实验目的

学会Linux系统中开发汇编程序的步骤和方法。在此基础上,掌握通过汇编程序访问GPIO端口,以实现控制Tiny6410开发板上LED的方法。

二、实验内容

本次实验使用Fedora(合肥校区)/CentOS(宣城校区)操作系统环境,安装ARM-Linux的开发库及编译器。学习在Linux下的编程和编译过程,即创建一个新目录leds_s,使用编辑器建立start.S和Makefile文件,并使用汇编语言编写LED控制程序。编译程序,并下载文件到目标开发板上运行。

三、预备知识

1. 清楚ARM微处理器芯片S3C6410的GPIO口硬件资源及相应寄存器结构。
2. 有ARM汇编语言基础。
3. 会使用Linux下常用的编辑器。
4. 掌握Makefile的编写和使用。
5. 了解Linux下的编译程序与交叉编译的过程。

四、实验设备及工具(包括软件调试工具)

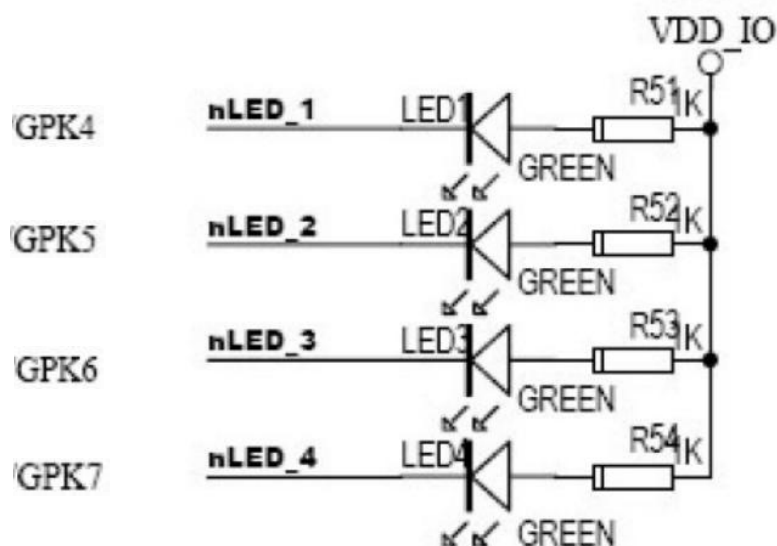
硬件: Tiny6410嵌入式实验平台。

软件: PC机操作系统Fedora/CentOS+Mini Tools+ARM-Linux开发环境

五、实验原理

1. LED硬件原理图

Tiny6410开发板上提供了4个用户可编程的LED,硬件原理图如下所示:



LED硬件原理图

可见，LED1、2、3、4分别使用的CPU GPIO端口资源为GPK_4、5、6、7。

2. 点亮LED控制原理

点亮4个LED需如下两个步骤：

- 把外设的基地址告诉CPU。对于6410来说, 内存的地址范围为 0x00000000-0x60000000, 外设的地址范围为0x70000000-0x7fffffff。
- 关闭看门狗，防止程序不断重启；设置寄存器GPKCON0，使GPK_4/5/6/7四个引脚为输出功能；往寄存器GPKDAT写0，使GPK_4/5/6/7四个引脚输出低电平, 4个LED会亮；相反，往寄存器GPKDAT写1，使GPK_4/5/6/7四个引脚输出高电平，4个LED会灭。

六、实验步骤

1. 建立工作目录leds。

首先将预先提供的实验源码复制到Windows系统桌面上，再点击【虚拟机】菜单中的【设置】，选择【选项】中的“共享文件夹”，添加Windows系统中的桌面路径为共享文件夹，然后鼠标右键复制Windows系统桌面上的leds文件夹（内含Makefile文件），接着进入虚拟机当前用户的Home（合肥校区）/root(宣城校区)目录，使用鼠标右键进行粘贴，从而将文件夹从Windows系统复制到虚拟机的系统中。

2. 编写程序源代码。

在Linux下的文本编辑器有许多，常用的是vim和Xwindow界面下的gedit等，建议在实验中使用vim（需要学习vim的操作方法，请参考相关书籍中的关于vim的操作指南）。

(1) start.S的汇编源程序如下:

```
.global _start
_start:
// 把外设的基地址告诉CPU
ldr r0, =0x70000000 //对于6410来说,内存(0x00000000~0x60000000),外设(0x70000000~0x7fffffff)
orr r0, r0, #0x13 //外设大小:256M
mcr p15,0,r0,c15,c2,4 //把r0的值(包括了外设基地址+外设大小)告诉cpu
// 关看门狗
ldr r0, =0x7E004000
mov r1, #0
str r1, [r0]
// 设置GPKCON0
ldr r1, =0x7F008800
ldr r0, =0x11110000
str r0, [r1]
mov r2, #0x1000
led_blink:
// 设置GPKDAT,使GPK_4/5/6/7引脚输出低电平,LED亮
ldr r1, =0x7F008808
mov r0, #0
str r0, [r1]
// 延时
bl delay
// 设置GPKDAT,使GPK_4/5/6/7引脚输出高电平,LED灭
ldr r1, =0x7F008808
mov r0, #0xf0
str r0, [r1]
// 延时
bl delay
sub r2, r2, #1
cmp r2, #0
bne led_blink
halt:
b halt
delay:
mov r0, #0x1000000
delay_loop:
cmp r0, #0
sub r0, r0, #1
bne delay_loop
mov pc, lr
```

(2) Makefile文件如下:

```
led.bin: start.o
arm-linux-ld -Ttext 0x50000000 -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
```

在Makefile所在目录下执行make命令时,系统会执行如下操作:

- 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;
- 执行arm-Linux-ld -Ttext 0x50000000 -o led.elf \$^,将所有.o文件链接成elf文件, -Ttext 0x50000000表示程序的运行地址是 0x50000000,即程序只有位于该地址上才能正常运行;

- 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；
- 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。

3. 编译及下载运行程序。

(1) 编译代码

确保当前用户为root用户（可使用`su root`命令切换到root用户）的条件下，在Fedora/CentOS的终端中执行如下命令：

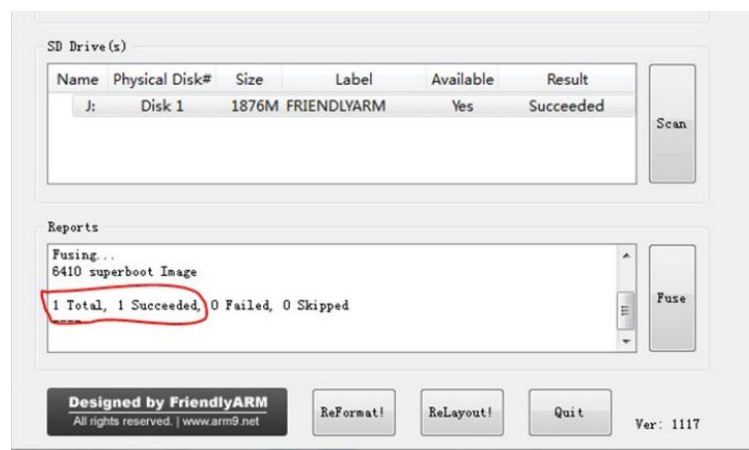
```
# cd leds
```

```
# make
```

执行 `make` 后会生成 `led.bin` 文件。

(2) 下载（烧写）和运行程序（★注意：实验领取的SD卡已提前完成Superboot-6410.bin烧写和images文件夹复制，故可跳过①、②两步操作，直接从③开始）

① 在Windows系统中，以管理员权限使用SD-Flasher程序，将引导程序Superboot-6410.bin烧入SD卡。成功烧写后的结果如下图所示：



② 在SD卡中建立一个images文件夹，并把配置文件FriendlyARM.ini复制到该文件夹中。双击打开SD卡中的该配置文件，在任意位置加入以下内容：USB-Mode = yes（注意字符的大小写）。如下图所示：

```
FriendlyARM - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
#This line cannot be removed. by FriendlyARM(www.arm9.net)

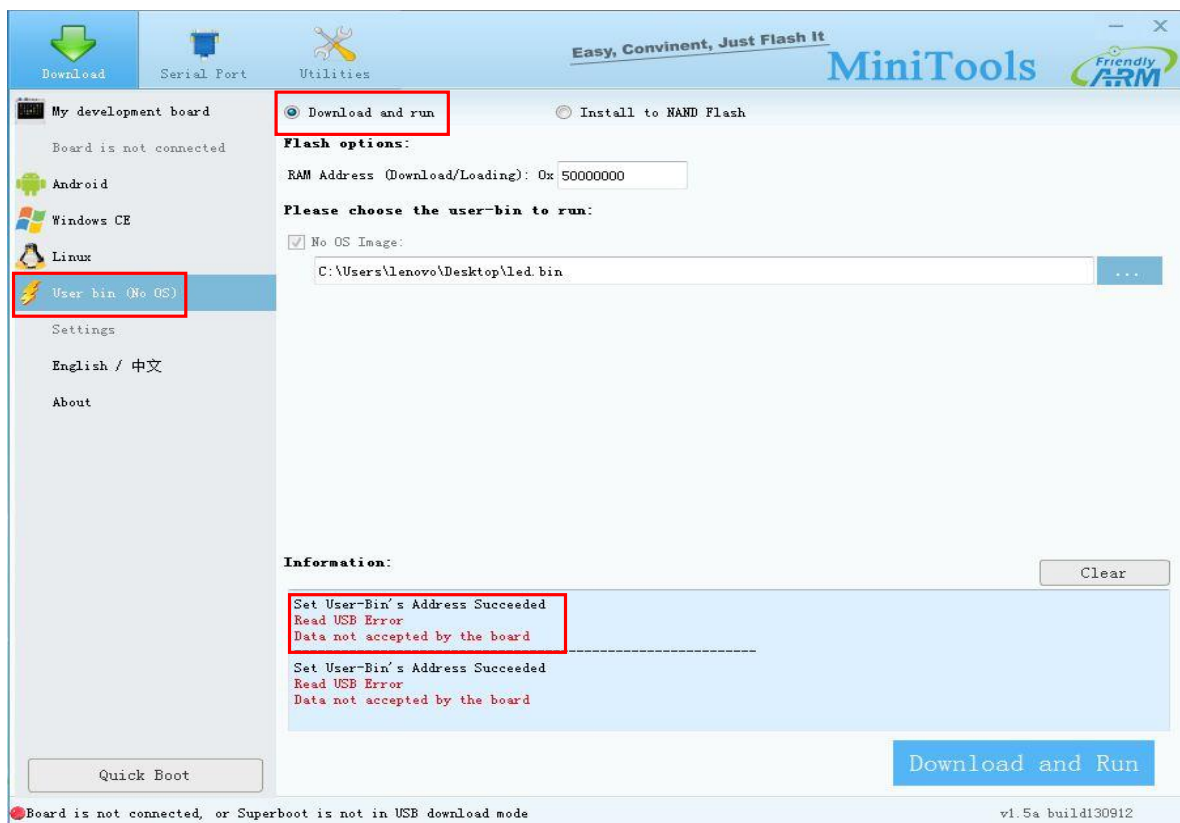
LCD-Type = S70
USB-Mode = yes
CheckOneButton=No
Action=install
OS= Linux

VerifyNandWrite=No

StatusType = Reenter LED
```

③ 在6410开发板断电的情况下，先用MicroUSB线连接开发板与PC机，并将领取（已烧写了Superboot-6410.bin）的SD卡插入开发板的SD卡槽，同时**确认启动模式**拨动开关S2拨在“**SDBOOT**”位置，然后打开开发板的电源。此时开发板将进入USB下载模式，LCD显示屏上显示“USB Mode: Waiting...”。接着，在Windows系统中以管理员身份运行MiniTools。若USB连接成功，则开发板LCD屏幕上显示“USB Mode: Connected”，同时MiniTools软件左下角显示 Board connected (S3C6410 533MHz / 256MB / 256MB(SLC) ID:ECDA1095 / 1-wire / P43(Auto))。★注意：若未显示连接成功或者发现USB连接断开，可尝试先关闭开发板电源，然后重新上电进行连接。

④ 使用Mini tool s软件下载并运行程序的方法如下：



首先在软件界面左侧选择“**User bin (No OS)**”，表示当前下载无系统的裸机程序；

接着选择“ Download and Run”，表示程序下载后直接运行；最后点击下方的【Download and Run】按钮，下载并运行程序。

★★特别说明：点击按钮后，Mini Tools左下角的开发板连接状态会变为not connected，且Information中会提示错误信息。无需理会，只需观察LED灯是否按照预定要求显示即可。

七、实验修改要求

在延时不变的条件下，将原程序中“四个LED全亮，然后全灭”的显示状态，修改为循环跑马灯效果，即“ LED1亮→LED2亮→LED3亮→LED4亮→LED3亮→LED2亮→LED1亮→LED2亮→……”。

实验二 查询方式检测按键

一、实验目的

学会Linux系统中开发C程序的步骤和方法。在此基础上，掌握通过汇编程序实现Tiny6410初始化及通过C程序实现通用输入输出端口(GPIO)数据操作的方法。

二、实验内容

本次实验使用Fedora（合肥校区）/CentOS（宣城校区）操作系统环境, 安装ARM-Linux的开发库及编译器。学习在Linux下的编程和编译过程，即创建一个新目录key_led，使用编辑器建立start.S、main.c和Makefile等文件，并使用C语言编写查询处理程序。编译程序，并下载文件到目标开发板上运行。

三、预备知识

1. 清楚ARM微处理器芯片S3C6410的GPIO口硬件资源及相应寄存器结构。
2. 有ARM汇编和C语言基础。
3. 会使用Linux下常用的编辑器。
4. 掌握Makefile的编写和使用。
5. 了解Linux下的编译程序与交叉编译的过程。

四、实验设备及工具（包括软件调试工具）

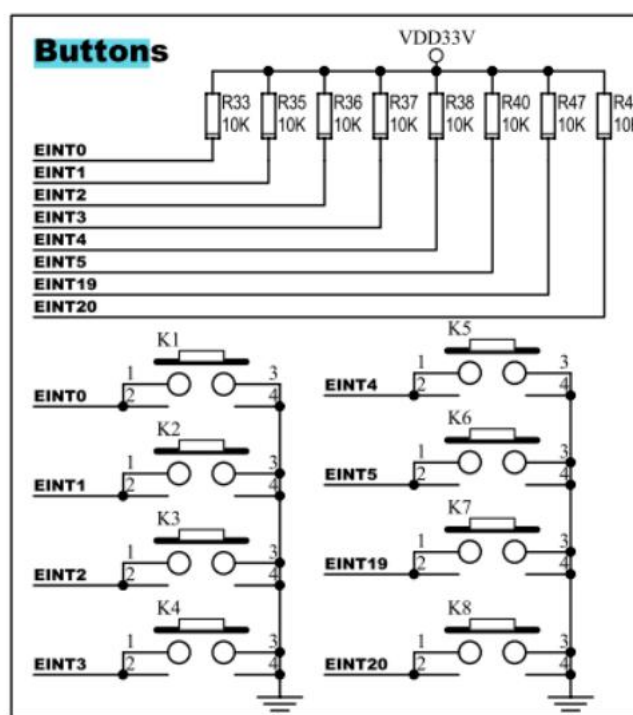
硬件：Tiny6410嵌入式实验平台。

软件：PC机操作系统Fedora/CentOS+Mini Tools+ARM-Linux开发环境

五、实验原理

1. 按键硬件原理图

Tiny6410中共有8个用户按键（☆说明：部分型号的底板上，只有K1~K4四个按键），原理图如下：



按键原理图

相关的引脚信息如下图：

按键	K1	K2	K3	K4	K5	K6	K7	K8
对应的中断	EINT0	EINT1	EINT2	EINT3	EINT4	EINT5	EINT19	EINT20
可复用为 GPIO	GPN0	GPN1	GPN2	GPN3	GPN4	GPN5	GPL11	GPL12

按键引脚图

2. 查询按键原理

- 把外设的基地址告诉CPU。对于6410来说, 内存的地址范围为 0x00000000-0x60000000, 外设的地址范围为0x70000000-0x7fffffff。
- 关闭看门狗, 防止程序不断重启; 设置寄存器GPNCON, 使接按键K1~K4的GPN0~GPN3为输入功能。设置寄存器GPKCON0, 使接LED1~LED4的GPK4~GPK7为输出功能。
- 由原理图可知, 按键按下时为低电平。读取寄存器GPNDATA, 依次判断哪个按键按下, 并点亮对应的LED灯。

六、实验步骤

1. 建立工作目录key_led。

首先将预先提供的实验源码复制到Windows系统桌面上, 再点击【虚拟机】菜单中的【设置】, 选择【选项】中的“共享文件夹”, 添加Windows系统中的桌面路径为共享文件夹, 然后鼠标右键复制Windows系统桌面上的key_led文件夹(内含源代码和Makefile文件), 接

着进入虚拟机当前用户的Home（合肥校区）/root(宣城校区)目录，使用鼠标右键进行粘贴，从而将文件夹从Windows系统复制到虚拟机的系统中。

2. 编写程序源代码。

在Linux下的文本编辑器有许多，常用的是vim和Xwindow界面下的gedit等，建议在实验中使用vim（需要学习vim的操作方法，请参考相关书籍中的关于vim的操作指南）。

(1) start.S的汇编源程序如下：

```
.global _start
_start:
    // 把外设的基地址告诉CPU
    ldr r0, =0x70000000
    orr r0, r0, #0x13
    mcr p15,0,r0,c15,c2,4

    // 关看门狗
    ldr r0, =0x7E004000
    mov r1, #0
    str r1, [r0]

    // 设置栈
    ldr sp, =0x0C002000

    // 开启icaches
#ifdef CONFIG_SYS_ICACHE_OFF
    bic r0, r0, #0x00001000 @ clear bit 12 (I) I-cache
#else
    orr r0, r0, #0x00001000 @ set bit 12 (I) I-cache
#endif
    mcr p15, 0, r0, c1, c0, 0

    // 调用C函数点灯
    bl main

halt:
    b halt
```

(2) main.c文件如下：

```
#define GPKCON0 (*(volatile unsigned long *)0x7F008800)
#define GPKDAT (*(volatile unsigned long *)0x7F008808)

#define GPNCON (*(volatile unsigned long *)0x7F008830)
#define GPNDAT (*(volatile unsigned long *)0x7F008834)

void main(void)
{
    int dat = 0;

    // 配置GPK4-7为输出功能
    GPKCON0 = 0x11111000;

    // 所有LED熄灭
    GPKDAT = 0x000000f0;

    // 配置GPN为输入功能
    GPNCON = 0;
```

```

// 轮询的方式查询按键事件
while(1)
{
    dat = GPNDAT;

    if(dat & (1<<0))          // KEY1被按下，则LED1亮，否则LED1灭
        GPKDAT |= 1<<4;
    else
        GPKDAT &= ~(1<<4);

    if(dat & (1<<1))          // KEY2被按下，则LED2亮，否则LED2灭
        GPKDAT |= 1<<5;
    else
        GPKDAT &= ~(1<<5);

    if(dat & (1<<2))          // KEY3被按下，则LED3亮，否则LED3灭
        GPKDAT |= (1<<6);
    else
        GPKDAT &= ~(1<<6);

    if(dat & (1<<3))          // KEY4被按下，则LED4亮，否则LED4灭
        GPKDAT |= 1<<7;
    else
        GPKDAT &= ~(1<<7);

}
}

```

(3) Makefile文件如下:

```

key.bin: start.o main.o
arm-linux-ld -Ttext 0x50000000 -o key.elf $^
arm-linux-objcopy -O binary key.elf key.bin
arm-linux-objdump -D key.elf > key_elf.dis
%.o : %.S
arm-linux-gcc -o $@ $< -c

%.o : %.c
arm-linux-gcc -o $@ $< -c

clean:
rm *.o *.elf *.bin *.dis -rf

```

3. 编译及下载运行程序。

(1) 编译代码

确保当前用户为root用户（可使用su root命令切换到root用户）的条件下，在Fedora/CentOS的终端中执行如下命令：

```
# cd key_led
```

```
# make
```

执行make后会生成key.bin文件。

(2) 下载（烧写）和运行程序

按实验一给出的方法下载程序。当未按下任何按键时，四个LED灯全灭，当按下KEY1~KEY4任意一个按键时，则对应的LED被点亮。

七、实验修改要求

当K1按下时，LED1和LED2被点亮；当K2按下时，LED3和LED4被点亮；当K3按下时，LED1和LED3被点亮；当K4按下时，LED2和LED4被点亮。

实验三 按键中断实验

一、实验目的

学会Linux系统中开发C程序的步骤和方法。在此基础上，掌握通过汇编程序实现Tiny6410中断控制器初始化及通过C程序实现中断处理的方法。

二、实验内容

本次实验使用Fedora（合肥校区）/CentOS（宣城校区）操作系统环境, 安装ARM-Linux的开发库及编译器。学习在Linux下的编程和编译过程，即创建一个新目录irq，使用编辑器建立start.S、main.c、irq.c和Makefile等文件，并使用C语言编写中断处理程序。编译程序，并下载文件到目标开发板上运行。

三、预备知识

1. 清楚ARM微处理器芯片S3C6410的外部中断口硬件资源及相应寄存器结构。
2. 有ARM汇编和C语言基础。
3. 会使用Linux下常用的编辑器。
4. 掌握Makefile的编写和使用。
5. 了解Linux下的编译程序与交叉编译的过程。

四、实验设备及工具（包括软件调试工具）

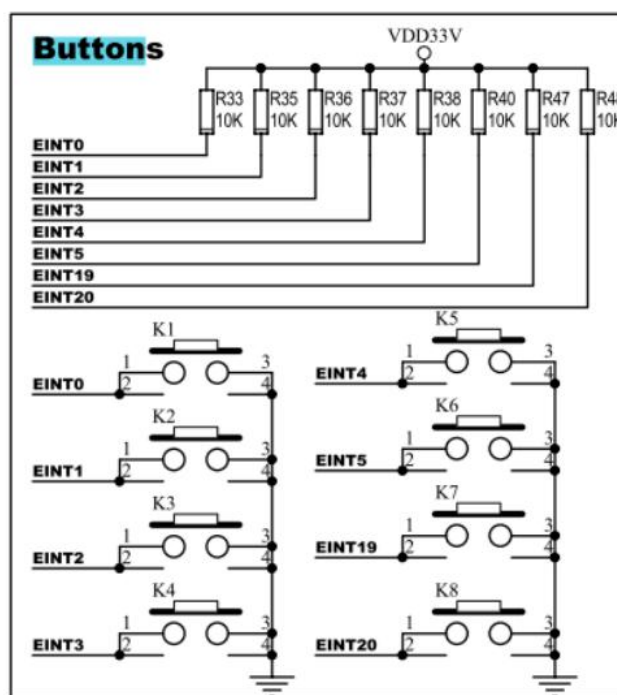
硬件：Tiny6410嵌入式实验平台。

软件：PC机操作系统Fedora/CentOS+Mini Tools+ARM-Linux开发环境

五、实验原理

1. 按键硬件原理图

Tiny6410中共有8个用户按键（☆说明：部分型号的底板上，只有K1~K4四个按键），原理图如下：



按键原理图

相关的引脚信息如下图：

按键	K1	K2	K3	K4	K5	K6	K7	K8
对应的中断	EINT0	EINT1	EINT2	EINT3	EINT4	EINT5	EINT19	EINT20
可复用为 GPIO	GPN0	GPN1	GPN2	GPN3	GPN4	GPN5	GPL11	GPL12

按键引脚图

2. 中断控制原理

- 把外设的基地址告诉CPU。对于6410来说, 内存的地址范围为 0x00000000-0x60000000, 外设的地址范围为0x70000000-0x7fffffff。
- 关闭看门狗, 防止程序不断重启; 设置**寄存器GPNCON**, 使接按键K0~K5的GPN0~GPN5为中断功能; 设置**寄存器EINTOCON0**, 使按键的中断触发信号采用下降沿触发; 设置**寄存器EINTOMASK**, 使按键对应的中断使能; 设置**VIC0INTSELECT**, 使对应中断源工作于IRQ模式。
- 在中断处理函数中, 查询**寄存器EINTPEND**, 判断哪个中断发生, 进而识别对应的按键。最后, 向**寄存器EINTPEND**对应位写“1”, 以实现清除中断标记。

六、实验步骤

1. 建立工作目录irq。

首先将预先提供的实验源码复制到Windows系统桌面上, 再点击【虚拟机】菜单中的【设置】, 选择【选项】中的“共享文件夹”, 添加Windows系统中的桌面路径为共享文件夹,

然后鼠标右键复制Windows系统桌面上的irq文件夹（内含源代码和Makefile文件），接着进入虚拟机当前用户的Home（合肥校区）/root(宣城校区)目录，使用鼠标右键进行粘贴，从而将文件夹从Windows系统复制到虚拟机的系统中。

2. 编写程序源代码。

在Linux下的文本编辑器有许多，常用的是vim和Xwindow界面下的gedit等，建议在实验中使用vim（需要学习vim的操作方法，请参考相关书籍中的关于vim的操作指南）。

(1) start.S的汇编源程序如下：

```
.global _start
.global asm_kl_irq
.extern do_irq
_start:
reset:
    // 外设地址告诉cpu
    ldr r0, =0x70000000
    orr r0, r0, #0x13
    mcr p15,0,r0,c15,c2,4

    // 关看门狗
    ldr r0, =0x7E004000
    mov r1, #0
    str r1, [r0]

    // 使能VIC
    mrc p15,0,r0,c1,c0,0
    orr r0,r0,#(1<<24)
    mcr p15,0,r0,c1,c0,0

    // 设置栈
    ldr sp, =8*1024

    // 初始化时钟
    bl clock_init

    // 初始化ddr
    bl sdram_init

    // 初始化nandflash
    bl nand_init

    // 初始化irq
    bl irq_init

    // 开中断
    //mov r0, #0x53
    //msr CPSR_cxsf, r0
    mrs r0,cpsr
    bic r0,r0,#0x80
    msr cpsr_c,r0
```

```

// 重定位, 把程序的代码段、数据段复制到它的链接地址去
adr r0, _start
ldr r1, =_start
ldr r2, =_bss_start
sub r2, r2, r1
cmp r1, r1
beq clean_bss
bl copy2ddr
cmp r0, #0
bne halt

// 清BSS, 把BSS段对应的内存清零
clean_bss:
ldr r0, =_bss_start
ldr r1, =_bss_end
mov r3, #0
cmp r0, r1
beq on_ddr
clean_loop:
str r3, [r0], #4
cmp r0, r1
bne clean_loop

on_ddr:
// 跳转
ldr pc, =main

// 中断异常
asm_k1_irq:
.word irq
irq:
/* 保存现场 */
ldr sp, =0x54000000
sub lr, lr, #4
stmfd sp!, {r0-r12, lr}
/* 处理异常 */
bl do_irq
/* 恢复现场 */
ldmfd sp!, {r0-r12, pc}^ /* ^表示把spsr恢复到cpsr */

halt:
b halt

```

(2) irq.c文件如下:

```

#include "stdio.h"
#define GPKCON0 (*(volatile unsigned long *)0x7F008800)
#define GPKDATA (*(volatile unsigned long *)0x7F008808)
#define GPNCON (*(volatile unsigned long *)0x7F008830)
#define GPNDAT (*(volatile unsigned long *)0x7F008834)
#define EINT0CON0 (*(volatile unsigned long *)0x7F008900)
#define EINT0MASK (*(volatile unsigned long *)0x7F008920)
#define EINT0PEND (*(volatile unsigned long *)0x7F008924)
#define PRIORITY (*(volatile unsigned long *)0x7F008280)
#define SERVICE (*(volatile unsigned long *)0x7F008284)
#define SERVICEPEND (*(volatile unsigned long *)0x7F008288)
#define VIC0IRQSTATUS (*(volatile unsigned long *)0x71200000)
#define VIC0FIQSTATUS (*(volatile unsigned long *)0x71200004)
#define VIC0RAWINTR (*(volatile unsigned long *)0x71200008)
#define VIC0INTSELECT (*(volatile unsigned long *)0x7120000c)
#define VIC0INTENABLE (*(volatile unsigned long *)0x71200010)
#define VIC0INTENCLEAR (*(volatile unsigned long *)0x71200014)
#define VIC0PROTECTION (*(volatile unsigned long *)0x71200020)
#define VIC0SWPRIORITYMASK (*(volatile unsigned long *)0x71200024)
#define VIC0PRIORITYDAISY (*(volatile unsigned long *)0x71200028)

#define VIC0ADDRESS (*(volatile unsigned long *)0x71200f00)

typedef void (isr) (void);
extern void asm_k1_irq();

```

```

void irq_init(void)
{
    /* 配置GPN0~5引脚为中断功能 */
    GPNCON &= ~(0xff);
    GPNCON |= 0xaa;

    /* 设置中断触发方式为：下降沿触发 */
    EINT0CON0 &= ~(0xff);
    EINT0CON0 |= 0x33;

    /* 禁止屏蔽中断 */
    EINT0MASK &= ~(0x0f);

    // Select INT_EINT0 mode as irq
    VIC0INTSELECT = 0;

    /* 在中断控制器里使能这些中断 */
    VIC0INTENABLE |= (0x3); /* bit0: eint0~3, bit1: eint4~11 */

    isr** isr_array = (isr**)(0x71200100);

    isr_array[0] = (isr*)asm_k1_irq;

    /*将GPK4-GPK7配置为输出口*/
    GPKCON0 = 0x11110000;

    /*熄灭四个LED灯*/
    GPKDATA = 0xf0;
}

```

```

void do_irq(void)
{
    int i = 0;
    //GPKDATA = 0x00;

    /* 分辨是哪个中断 */
    for (i = 0; i < 4; i++)
    {
        if (EINT0PEND & (1<<i))
        {
            GPKDATA &= ~(1<<(i+4));
        }
        else
        {
            GPKDATA |= 1<<(i+4);
        }
    }
    if (EINT0PEND & 0x8)
    {
        GPKDATA = 0xf0;
    }

    /* 清中断 */
    EINT0PEND = 0x3f;
    VIC0ADDRESS = 0;
}

```

(3) main.c文件如下:

```
#include "stdio.h"

int main()
{
    while (1)
    {
        //
    }
    return 0;
}
```

(4) Makefile文件如下:

```
CC      = arm-linux-gcc
LD       = arm-linux-ld
AR       = arm-linux-ar
OBJCOPY  = arm-linux-objcopy
OBJDUMP  = arm-linux-objdump

INCLUDEDIR := $(shell pwd)/include
CFLAGS     := -Wall -Os -fno-builtin
CPPFLAGS   := -nostdinc -I$(INCLUDEDIR)

export CC AR LD OBJCOPY OBJDUMP INCLUDEDIR CFLAGS CPPFLAGS

objs := start.o sdram.o clock.o uart.o irq.o  main.o lib/libc.a nand.o

irq.bin: $(objs)
    ${LD} -Tirq.lds -o irq.elf $^
    ${OBJCOPY} -O binary -S irq.elf $@
    ${OBJDUMP} -D irq.elf > irq.dis

.PHONY : lib/libc.a
lib/libc.a:
    cd lib; make; cd ..

%.o: %.c
    ${CC} ${CPPFLAGS} ${CFLAGS} -c -o $@ $<

%.o: %.S
    ${CC} ${CPPFLAGS} ${CFLAGS} -c -o $@ $<

clean:
    make clean -C lib
    rm -f *.bin *.elf *.dis *.o
```

3. 编译及下载运行程序。

(1) 编译代码

确保当前用户为root用户（可使用su root命令切换到root用户）的条件下，在Fedora/CentOS的终端中执行如下命令：

```
# cd irq
```

```
# make
```

执行make后会生成irq.bin文件。

(2) 下载（烧写）和运行程序

按实验一给出的方法下载程序。当按下K1按键时，LED1点亮；按下K2按键时，LED2点亮；按下K3按键时，LED3点亮；按下K4按键时，所有LED熄灭。

七、实验修改要求

当K1按下时，LED1到LED4依次被点亮；当K2按下时，LED4到LED1依次被点亮；当K3按下时，LED1到LED4依次被点亮，且每个时刻只有一个LED亮；当K4按下时，LED4到LED1依次被点亮，且每个时刻只有一个LED亮。

实验四 PWM 定时器实验

一、实验目的

学会Linux系统中开发C程序的步骤和方法。在此基础上，掌握通过C程序实现Tiny6410定时器初始化及中断处理的方法。

二、实验内容

本次实验使用Fedora（合肥校区）/CentOS（宣城校区）操作系统环境，安装ARM-Linux的开发库及编译器。学习在Linux下的编程和编译过程，即创建一个新目录timer，使用编辑器建立start.S、main.c、timer.c和Makefile等文件。编译程序，并下载文件到目标开发板上运行。

三、预备知识

1. 清楚ARM微处理器芯片S3C6410的定时器硬件资源及相应寄存器结构。
2. 有ARM汇编和C语言基础。
3. 会使用Linux下常用的编辑器。
4. 掌握Makefile的编写和使用。
5. 了解Linux下的编译程序与交叉编译的过程。

四、实验设备及工具（包括软件调试工具）

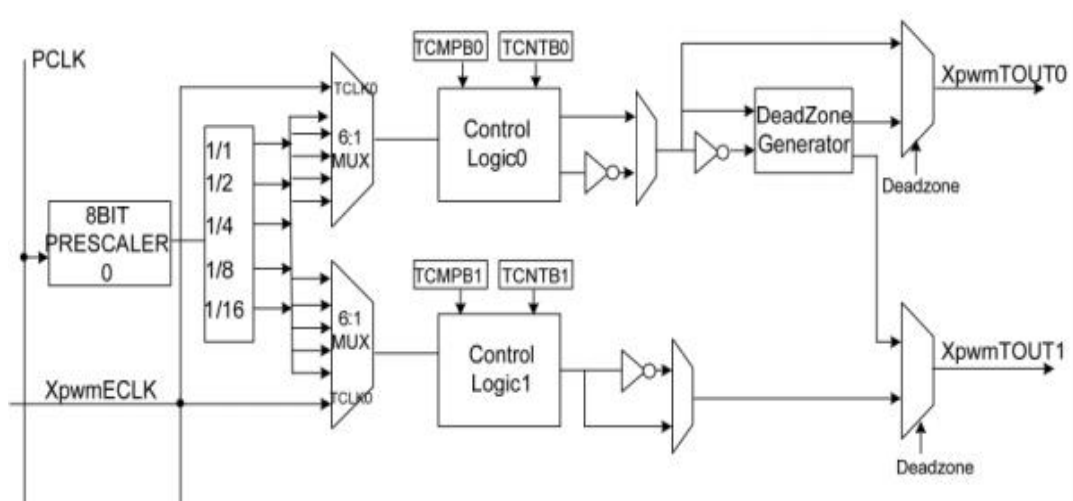
硬件：Tiny6410嵌入式实验平台。

软件：PC机操作系统Fedora/CentOS+MiniTools+ARM-Linux开发环境

五、实验原理

1. PWM定时器

S3C6410内部包含五个32位定时器0~4。这些计时器通过产生内部定时中断来与ARM微处理器交互。



定时器0和1的内部结构图

2. 中断控制原理

- 把外设的基地址告诉CPU。对于6410来说, 内存的地址范围为 0x00000000-0x60000000, 外设的地址范围为0x70000000-0x7fffffff。
- 关闭看门狗, 防止程序不断重启; 设置CPSR, 允许IRQ中断; 设置中断控制器中的寄存器VICINTENABLE, 允许定时器0中断; 设置寄存器TCFG0和TCFG1, 写入相应的两次分频系数; 设置寄存器TCNTB0和TCMPB0, 写入计数初值和PWM比较值; 设置寄存器TCON, 手动更新计数值, 自动重载计数初值, 并启动定时器; 设置寄存器TINT_CSTAT, 允许定时器0中断。
- 在中断处理函数中, 执行相应的处理。最后, 向寄存器TINT_CSTAT对应位写“1”, 以实现清除中断标记。

六、实验步骤

1. 建立工作目录timer。

首先将预先提供的实验源码复制到Windows系统桌面上, 再点击【虚拟机】菜单中的【设置】, 选择【选项】中的“共享文件夹”, 添加Windows系统中的桌面路径为共享文件夹, 然后鼠标右键复制Windows系统桌面上的timer文件夹(内含源代码和Makefile文件), 接着进入虚拟机当前用户的Home(合肥校区)/root(宣城校区)目录, 使用鼠标右键进行粘贴, 从而将文件夹从Windows系统复制到虚拟机的系统中。

2. 编写程序源代码。

在Linux下的文本编辑器有许多, 常用的是vim和Xwindow界面下的gedit等, 建议在实验中使用vim(需要学习vim的操作方法, 请参考相关书籍中的关于vim的操作指南)。

(1) start.S的汇编源程序如下:

```
.globl _start
.global asm_timer_irq
.extern do_irq
_start:

reset:
    // 外设地址告诉cpu
    ldr r0, =0x70000000
    orr r0, r0, #0x13
    mcr p15,0,r0,c15,c2,4

    // 关看门狗
    ldr r0, =0x7E004000
    mov r1, #0
    str r1, [r0]

    //Enabel VIC
    mrc p15,0,r0,c1,c0,0
    orr r0,r0,#(1<<24)
    mcr p15,0,r0,c1,c0,0

    // 设置栈
    ldr sp, =8*1024

    // 初始化时钟
    bl clock_init

    // 初始化ddr
    bl sdram_init

    // 初始化nandflash
    bl nand_init

    // 初始化中断
    bl irq_init

    // 开中断
    //mov r0, #0x53
    //msr CPSR_cxsf, r0
    mrs r0,cpsr

    // 重定位, 把程序的代码段、数据段复制到它的链接地址去
    adr r0, _start
    ldr r1, =_start
    ldr r2, =bss_start
    sub r2, r2, r1
    cmp r0,r1
    beq clean_bss
    bl copy2ddr
    cmp r0, #0
    bne halt

    // 清BSS, 把BSS段对应的内存清零
clean_bss:
    ldr r0, =bss_start
    ldr r1, =bss_end
    mov r3, #0
    cmp r0, r1
    beq on_ddr
clean_loop:
    str r3, [r0], #4
    cmp r0, r1
    bne clean_loop

on_ddr:
    // 跳转
    ldr pc, =main
```



```

        // 中断异常
asm_timer_irq:
        .word irq
irq:
        /* 1. 保存现场 */
        ldr sp, =0x54000000
        sub lr, lr, #4
        stmfd sp!, {r0-r12, lr} /* lr就是swi的下一条指令地址 */
        /* 2. 处理异常 */
        bl do_irq
        /* 3. 恢复现场 */
        ldmfd sp!, {r0-r12, pc}^ /* ^表示把spsr恢复到cpsr */

halt:
        b halt

```

(2) timer.c文件如下:

```

#define PWMTIMER_BASE (0x7F006000)
#define TCFG0 ((volatile unsigned long *) (PWMTIMER_BASE+0x00))
#define TCFG1 ((volatile unsigned long *) (PWMTIMER_BASE+0x04))
#define TCON ((volatile unsigned long *) (PWMTIMER_BASE+0x08))
#define TCNTB0 ((volatile unsigned long *) (PWMTIMER_BASE+0x0C))
#define TCMPB0 ((volatile unsigned long *) (PWMTIMER_BASE+0x10))
#define TCNT00 ((volatile unsigned long *) (PWMTIMER_BASE+0x14))
#define TCNTB1 ((volatile unsigned long *) (PWMTIMER_BASE+0x18))
#define TCMPB1 ((volatile unsigned long *) (PWMTIMER_BASE+0x1C))
#define TCNT01 ((volatile unsigned long *) (PWMTIMER_BASE+0x20))
#define TCNTB2 ((volatile unsigned long *) (PWMTIMER_BASE+0x24))
#define TCMPB2 ((volatile unsigned long *) (PWMTIMER_BASE+0x28))
#define TCNT02 ((volatile unsigned long *) (PWMTIMER_BASE+0x2C))
#define TCNTB3 ((volatile unsigned long *) (PWMTIMER_BASE+0x30))
#define TCMPB3 ((volatile unsigned long *) (PWMTIMER_BASE+0x34))
#define TCNT03 ((volatile unsigned long *) (PWMTIMER_BASE+0x38))
#define TCNTB4 ((volatile unsigned long *) (PWMTIMER_BASE+0x3C))
#define TCNT04 ((volatile unsigned long *) (PWMTIMER_BASE+0x40))
#define TINT_CSTAT ((volatile unsigned long *) (PWMTIMER_BASE+0x44))

typedef void (isr) (void);
extern void asm_timer_irq();

void irq_init(void)
{
    /* 在中断控制器里使能timer0中断 */
    VIC0INTENABLE |= (1<<23);

    VIC0INTSELECT =0;

    isr** isr_array = (isr**)(0x7120015C);

    isr_array[0] = (isr*)asm_timer_irq;

    /*将GPK4-GPK7配置为输出口*/
    GPKCON0 = 0x11110000;

    /*熄灭四个LED灯*/
    GPKDATA = 0xff;
}

// timer0中断的中断处理函数
void do_irq()
{
    unsigned long uTmp;
    GPKDATA = ~GPKDATA;
    //清timer0的中断状态寄存器
    uTmp = TINT_CSTAT;
    TINT_CSTAT = uTmp;
    VIC0ADDRESS=0x0;
}

```

```

// 初始化timer
void timer_init(unsigned long utimer,unsigned long uprescaler,unsigned long udivider,unsigned long utcntb,unsigned long utcmb)
{
    unsigned long temp0;

    // 定时器的输入时钟 = PCLK / ( {prescaler value + 1} ) / {divider value} = PCLK/(65+1)/16=62500hz

    //设置预分频系数为66
    temp0 = TCFG0;
    temp0 = (temp0 & (~0xff00ff)) | ((uprescaler-1)<<0);
    TCFG0 = temp0;

    // 16分频
    temp0 = TCFG1;
    temp0 = (temp0 & (~0xf<<4*utimer)) & (~1<<20)) | (udivider<<4*utimer);
    TCFG1 = temp0;

    // 1s = 62500hz
    TCNTB0 = utcntb;
    TCMPOB0 = utcmb;

    // 手动更新
    TCON |= 1<<1;

    // 清手动更新位
    TCON &= ~(1<<1);

    // 自动加载和启动timer0
    TCON |= (1<<0)|(1<<3);

    // 使能timer0中断
    temp0 = TINT_CSTAT;
    temp0 = (temp0 & (~1<<utimer))|(1<<(utimer));
    TINT_CSTAT = temp0;
}

```

(3) main.c文件如下:

```

#include "stdio.h"

void timer_init(unsigned long utimer,unsigned long uprescaler,unsigned long udivider,unsigned long utcntb,unsigned long utcmb);

int main()
{
    timer_init(0,65,4,62500,0);

    while (1)
    {

    }

    return 0;
}

```

(4) Makefile文件如下:

```
CC      = arm-linux-gcc
LD      = arm-linux-ld
AR      = arm-linux-ar
OBJCOPY = arm-linux-objcopy
OBJDUMP = arm-linux-objdump

INCLUDEDIR := $(shell pwd)/include
CFLAGS     := -Wall -Os -fno-builtin
CPPFLAGS   := -nostdinc -I$(INCLUDEDIR)

export CC AR LD OBJCOPY OBJDUMP INCLUDEDIR CFLAGS CPPFLAGS

objs := start.o clock.o uart.o timer.o sdram.o main.o lib/libc.a nand.o

timer.bin: $(objs)
    ${LD} -Ttimer.lds -o timer.elf $^
    ${OBJCOPY} -O binary -S timer.elf $@
    ${OBJDUMP} -D timer.elf > timer.dis

.PHONY : lib/libc.a
lib/libc.a:
    cd lib; make; cd ..

%.o:%.c
    ${CC} ${CPPFLAGS} ${CFLAGS} -c -o $@ $<

%.o:%.S
    ${CC} ${CPPFLAGS} ${CFLAGS} -c -o $@ $<

clean:
    make clean -C lib
    rm -f *.bin *.elf *.dis *.o
```

3. 编译及下载运行程序。

(1) 编译代码

确保当前用户为root用户（可使用su root命令切换到root用户）的条件下，在Fedora/CentOS的终端中执行如下命令：

```
# cd timer
```

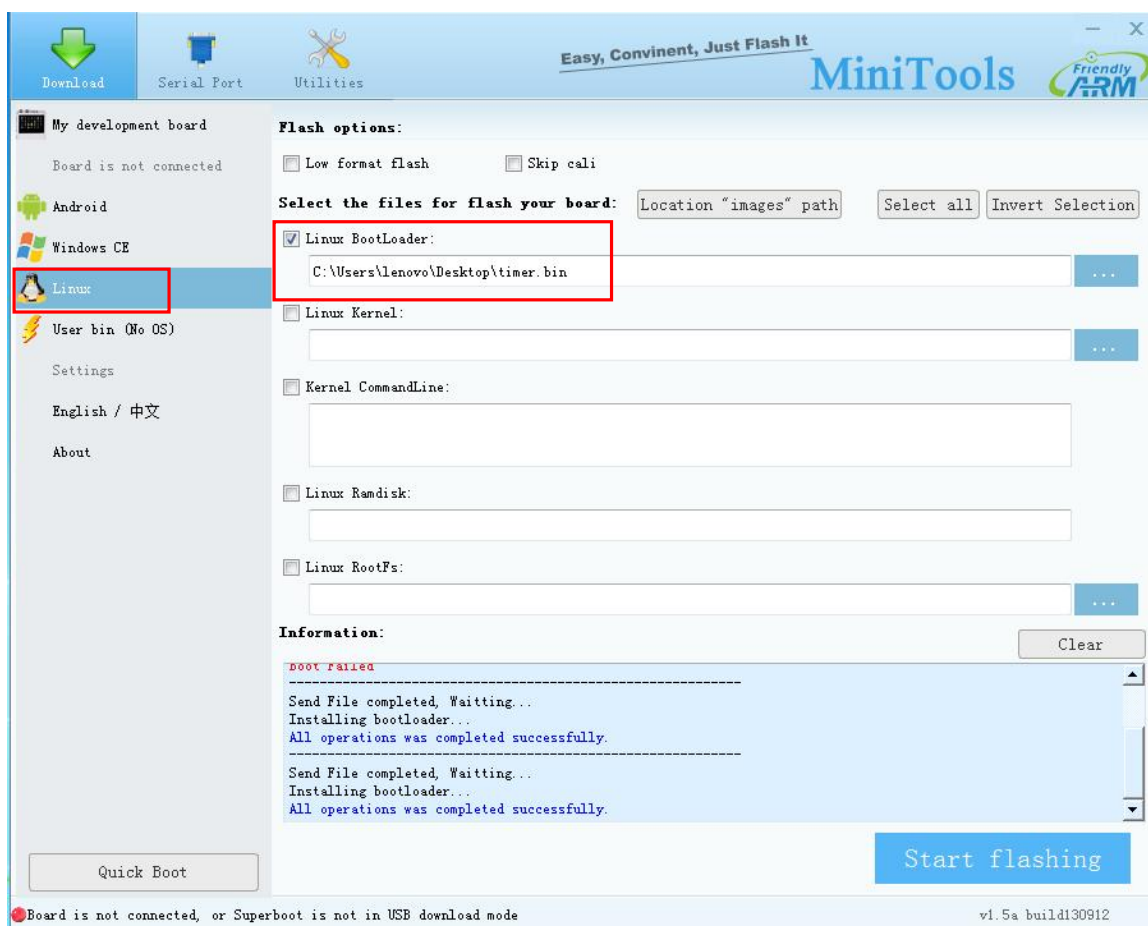
```
# make
```

执行make后会生成timer.bin文件。

(2) 下载（烧写）和运行程序

①~③按实验一给出的方法操作。

④ 使用Mini tools软件下载并运行程序的方法如下(★★★注意：本实验的程序下载和运行方法不同于之前的实验)：



首先在软件界面左侧选择“Linux”，接着勾选“Linux BootLoader”复选框，并选择需要下载的裸机程序（bin文件），表示将当前裸机程序以Linux系统启动引导程序的方式下载到Nand Flash的0地址开始的区域。最后点击下方的【Start flashing】按钮，执行所选择的裸机程序的烧写。

关闭开发板电源，将拨动开关S2拨动至“NAND”位置，然后重新开启开发板电源，则开发板运行刚刚烧入的裸机程序。初始状态，四个LED灯都为灭。每当定时器0的1秒定时达到后，四个LED切换状态（灭→亮/亮→灭）。

★★特别说明：点击按钮后，MiniTools左下角的开发板连接状态会变为not connected，且Information中会提示错误信息。无需理会，只需观察LED灯是否按照预定要求显示即可。

七、实验修改要求

每间隔2秒，四个LED按跑马灯方式循环显示（LED1亮→LED2亮→LED3亮→LED4亮→LED1亮→LED2亮→LED3亮→LED4亮→……）。

实验五 串行口通信实验

一、实验目的

学会Linux系统中开发C程序的步骤和方法。在此基础上，掌握通过C程序实现Tiny6410串口数据接收与发送的方法。

二、实验内容

本次实验使用Fedora（合肥校区）/CentOS（宣城校区）操作系统环境，安装ARM-Linux的开发库及编译器。学习在Linux下的编程和编译过程，即创建一个新目录uart_putchar，使用编辑器建立start.S、main.c、uart.c和Makefile等文件。编译程序，并下载文件到目标开发板上运行。通过串口线连接PC机与Tiny6410，实现两者之间字符的传输。

三、预备知识

1. 清楚ARM微处理器芯片S3C6410的串口硬件资源及相应寄存器结构。
2. 有ARM汇编和C语言基础。
3. 会使用Linux下常用的编辑器。
4. 掌握Makefile的编写和使用。
5. 了解Linux下的编译程序与交叉编译的过程。

四、实验设备及工具（包括软件调试工具）

硬件：Tiny6410嵌入式实验平台，串口线。

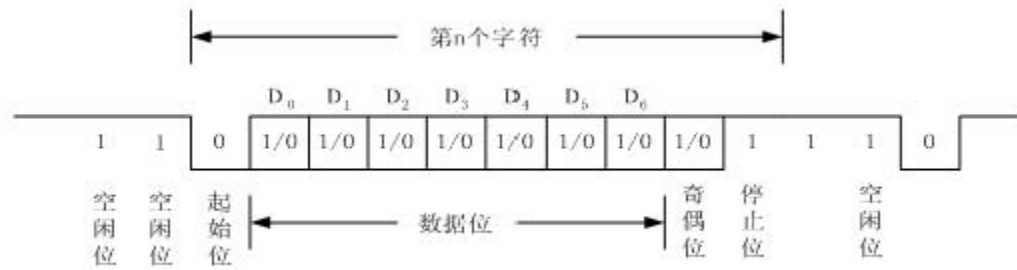
软件：PC机操作系统Fedora/CentOS+MiniTools+ARM-Linux开发环境，串口调试助手。

五、实验原理

1. 通用异步收发器UART

通用异步收发器（Universal Asynchronous Receiver and Transmitter，UART）使用异步串行通信方式传输数据。异步串行通信方式是将传输数据的每个字符一位接一位(例如先低位、后高位)地传送。数据的各不同位可以分时使用同一传输通道，因此串行通信可以减少信号连线，最少用两根线即可进行。接收方对于同一根线上一连串的数字信号，首先要分割成位，再按位组成字符。为了恢复发送的信息，双方必须协调工作。通信双方使用各自的时

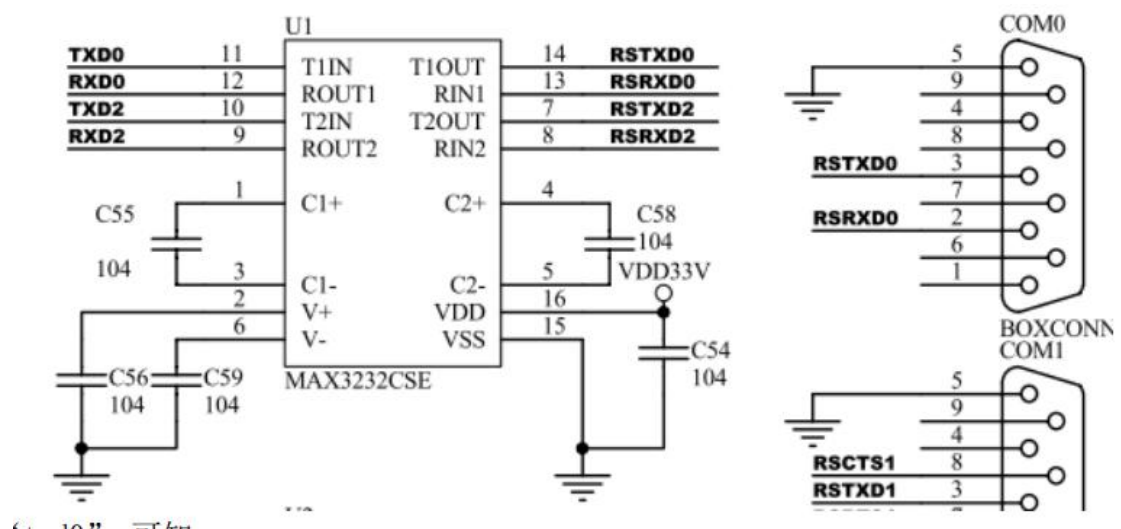
钟信号，而且允许时钟频率有一定误差，因此实现较容易。但是由于每个字符都要独立确定起始和结束（即每个字符都要重新同步），字符和字符间还可能有长度不定的空闲时间，因此效率较低。



串行通信字符格式

上图给出异步串行通信中一个字符的传送格式。开始前，线路处于空闲状态，送出连续“1”。传送开始时，首先发一个“0”作为起始位，然后出现在通信线上的是字符的二进制编码数据。每个字符的数据位长可以约定为5 位、6位、7 位或8 位，一般采用ASCII 编码。后面是奇偶校验位，根据约定，用奇偶校验位将所传字符中为“1”的位数凑成奇数个或偶数个，也可以约定不要奇偶校验，这样就取消奇偶校验位。最后是表示停止位的“1”信号，这个停止位可以约定持续1 位、1.5位或2 位的时间宽度。至此一个字符传送完毕，线路又进入空闲，持续为“1”。经过一段随机的时间后，下一个字符开始传送才又发出起始位。每一个数据位的宽度等于传送波特率的倒数。异步串行通信中，常用的波特率为50， 95， 110， 150， 300， 600， 1200， 2400， 4800， 9600， 115200 等。

S3C6410的UART提供了四个独立的异步串行通信端口。每个异步串行端口通过中断或者直接存储器存取（DMA）模式来操作，以实现在CPU和UART之间传输数据。每个UART的通道包含了两个64字节收发FIFO存储器。



RXD0	D20	XuRXD0/GPA0
TXD0	A23	XuTXD0/GPA1
	G16	
	A22	XuCTSn0/GPA2
	J15	XuRTSn0/GPA3
RXD1		XuRXD1/GPA4
TXD1	B22	XuTXD1/GPA5
CTSn1	H15	XuCTSn1/GPA6
RTSn1	C22	XuRTSn1/GPA7
RXD2	D19	XuRXD2/ExdREQ/IrRXD/GPB
TXD2	A21	XuTXD2/ExdACK/IrTXD/GPB
RXD3	J14	XuRXD3/IrRXD/ExdREQ/GPB
TXD3	B21	XuTXD3/IrTXD/ExdACK/GPB
GPB4	G15	XirSDBW/XcamFIELD/BUF_D

UART引脚连接图

2. 串口收发原理

- 把外设的基地址告诉CPU。对于6410来说, 内存的地址范围为 0x00000000-0x60000000, 外设的地址范围为0x70000000-0x7fffffff。
- 关闭看门狗, 防止程序不断重启; 对于串口0, 设置**寄存器GPA0CON**, 使GPA0和GPA1工作于串行通信功能; 设置**寄存器ULCON0、UCON0、UFCON0和UMCON0**, 写入串行通信帧格式等相关参数(8个数据位, 无奇偶校验, 1个停止位, 无流控等); 设置**寄存器UBRDIV0和UDIVSL0T0**, 写入串行通信波特率。
- 对于串口0, 查询**寄存器UFSTAT0**的低八位是否为0, 以判断是否接收到数据, 并从**寄存器URXH0**中获取收到的数据; 查询寄存器UFSTAT0的D14位, 以判断能否发送数据, 并向**寄存器UTXH0**写入发送的数据。

六、实验步骤

1. 建立工作目录uart_putchar。

首先将预先提供的实验源码复制到Windows系统桌面上, 再点击【虚拟机】菜单中的【设置】, 选择【选项】中的“共享文件夹”, 添加Windows系统中的桌面路径为共享文件夹, 然后鼠标右键复制Windows系统桌面上的uart_putchar文件夹(内含源代码和Makefile文件), 接着进入虚拟机当前用户的Home(合肥校区)/root(宣城校区)目录, 使用鼠标右键进行粘贴, 从而将文件夹从Windows系统复制到虚拟机的系统中。

2. 编写程序源代码。

在Linux下的文本编辑器有许多, 常用的是vim和Xwindow界面下的gedit等, 建议在实

验中使用vim（需要学习vim的操作方法，请参考相关书籍中的关于vim的操作指南）。

(1) start.S的汇编源程序如下：

```
.global _start
_start:

// 把外设的基地址告诉CPU
ldr r0, =0x70000000
orr r0, r0, #0x13
mcr p15,0,r0,c15,c2,4

// 关看门狗
ldr r0, =0x7E004000
mov r1, #0
str r1, [r0]

// 设置栈
ldr sp, =0x0C002000

// 开启icaches
#ifdef CONFIG_SYS_ICACHE_OFF
    bic r0, r0, #0x00001000    @ clear bit 12 (I) I-cache
#else
    orr r0, r0, #0x00001000    @ set bit 12 (I) I-cache
#endif
mcr p15, 0, r0, c1, c0, 0

// 设置时钟
bl clock_init

// 调用c函数点灯
bl main

halt:
    b halt
```

(2) uart.c文件如下：

```
#include "uart.h"

#define ULCON0    (*((volatile unsigned long *)0x7F005000))
#define UCON0     (*((volatile unsigned long *)0x7F005004))
#define UFCON0    (*((volatile unsigned long *)0x7F005008))
#define UMCON0    (*((volatile unsigned long *)0x7F00500C))
#define UTRSTAT0  (*((volatile unsigned long *)0x7F005010))
#define UFSTAT0   (*((volatile unsigned long *)0x7F005018))
#define UTXH0     (*((volatile unsigned char *)0x7F005020))
#define URXH0     (*((volatile unsigned char *)0x7F005024))
#define UBRDIV0   (*((volatile unsigned short *)0x7F005028))
#define UDIVSLOT0 (*((volatile unsigned short *)0x7F00502C))
#define GPACON    (*((volatile unsigned long *)0x7F008000))

void init_uart(void)
{
    /* 1. 配置引脚 */
    GPACON &= ~0xff;
    GPACON |= 0x22;

    /* 2. 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

    /* 3. 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;
}
```



```

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

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

```

(3) main.c文件如下:

```

#include "uart.h"

int main()
{
    char c;

    init_uart();
    while (1)
    {
        c = getchar();
        putchar(c+1);
    }

    return 0;
}

```

(4) Makefile文件如下:

```

uart.bin: start.o main.o uart.o clock.o
    arm-linux-ld -Ttext 0x50000000 -o uart.elf $^
    arm-linux-objcopy -O binary uart.elf uart.bin
    arm-linux-objdump -D uart.elf > uart.dis

%.o : %.S
    arm-linux-gcc -o $@ $< -c

%.o : %.c
    arm-linux-gcc -o $@ $< -c -fno-builtin

clean:
    rm *.o *.elf *.bin *.dis -rf

```

3. 编译及下载运行程序。

(1) 编译代码

在Fedora/CentOS的终端中执行如下命令:

```

# cd uart_putchar

# make

```

执行make后会生成uart.bin文件。

(2) 下载（烧写）和运行程序

首先按实验一给出的方法下载程序，然后使用串口线将PC机的串口与6410开发板的COM0/COM3串口进行连接，接着运行PC机上的串口调试助手软件，将波特率设置为115200，其它串口通信参数保持默认，最后打开串口，在发送区输入“ abcd” 并点击发送。若串口通信正常，则接收区会显示“ bcde” 。

七、实验修改要求

如果Tiny6410开发板收到的是大写字母，则将其转换为小写字母后发送回去；如果收到的是小写字母，则将其转换为大写字母后发送回去；如果收到的是数字，则把数字乘以2后发送回去。

实验六 熟悉 Linux 开发环境

一、实验目的

熟悉Linux开发环境，学会基于Tiny6410的Linux开发环境的配置和使用。使用arm-Linux-gcc编译，并使用minicom实现串口方式下载和调试程序。

二、实验内容

本次实验使用Fedora（合肥校区）/CentOS（宣城校区）操作系统环境, 安装ARM-Linux的开发库及编译器。创建一个新目录hello，并在其中编写hello.c和Makefile文件。学习在Linux下的编程和编译过程，以及ARM开发板的使用和开发环境的设置。下载已经编译好的文件到目标开发板上运行。

三、预备知识


1. 有C语言基础。
2. 会使用Linux下常用的编辑器。
3. 掌握Makefile的编写和使用。
4. 了解Linux下的编译程序与交叉编译的过程。

四、实验设备及工具（包括软件调试工具）

硬件：Tiny6410嵌入式实验平台。

软件：PC机操作系统Fedora/CentOS+minicom+ARM-Linux 开发环境

五、实验步骤

1. 登录Windows系统，使用管理员权限打开VMware虚拟机软件。确认虚拟机中已安装Fedora(合肥校区)/CentOS（宣城校区）系统，否则请通过镜像文件安装。
2. 确认虚拟机串口已打开（**★注意：**虚拟机右下角显示图标且不为灰色），否则通过【编辑此虚拟机】选项，使用“添加”功能添加串行通信端口，并确认使用物理串行端口中为“自动选择串口”选项。（**★★特别说明：**对于PC机自身没有物理RS232串口，而是通过USB转串线虚拟出串口的情况，需要首先把USB转串线插上电脑USB口，并在Windows的设备管理器中确认驱动安装成功，再打开VMware软件进行串口添加，否则虚拟机会提示没有物理串口。）

3. 通过【启动此虚拟机】，启动Fedora（合肥校区）/CentOS_QT_6410（宣城校区）系统。
合肥校区请选择knight用户，并输入登录密码knight；宣城校区启动后自动以root用户登录。

4. 打开终端（terminal）

(1) ★若当前不是root用户，需要切换为root用户。（若当前已经是root用户，则跳过此步）

```
[Knight@localhost hello]$ su root
Password:
[root@localhost hello]#
```

切换方法：终端输入su root，再输入密码knight即可（合肥校区）。

(2) 建立hello工作目录。（若已存在hello目录，则跳过该步骤。可通过ls命令查看是否存在hello目录。）

首先将预先提供的实验源码复制到Windows系统桌面上，再点击【虚拟机】菜单中的【设置】，选择【选项】中的“共享文件夹”，添加Windows系统中的桌面路径为共享文件夹，然后鼠标右键复制Windows系统桌面上的hello文件夹（内含Makefile文件），接着进入虚拟机当前用户的Home（合肥校区）/root(宣城校区)目录，使用鼠标右键进行粘贴，从而将文件夹从Windows系统复制到虚拟机的系统中。

也可以使用mkdir命令自行创建hello工作目录。如下所示：

```
[root@zxt smile]# mkdir hello
[root@zxt smile]# cd hello
```

此时新建的hello工作目录，会在对应的目录下出现，说明此次操作成功（务必记清楚所创建目录的位置），如下图所示：

```
[root@localhost home]# mkdir hello
[root@localhost home]# ls
hello rdy
[root@localhost home]# cd hello
```

5. 编写程序源代码。

hello.c 源代码较简单，如下：

```
#include <stdio.h>

int main()
{
    printf("Hello,World!\n");

    return 0;
}
```

可以用下面的命令来编写hello.c的源代码，进入hello目录使用vim命令来编辑代码(也可以使用gedit命令来编辑hello.c文件：gedit hello.c)：

```
[root@zxt hello]# vi hello.c
```

按“**I**”键或者“**A**”键进入编辑模式，将上面的代码录入进去，完成后按Esc键进入命令状态，再用命令“**:wq**”保存并退出。这样便在当前目录下建立了一个名为hello.c的文件。

6. 编写Makefile。（若hello工作目录中已有Makefile文件，则跳过此步）

要使上面的hello.c程序能够运行，必须要编写一个Makefile文件，Makefile文件定义了一系列的规则，它指明了哪些文件需要编译，哪些文件需要先编译，哪些文件需要重新编译等等更为复杂的命令。使用它带来的好处就是自动编译，只需要敲一个“make”命令整个工程就可以实现自动编译，当然本次实验只有一个文件，它还不能体现出使用Makefile的优越性，但当工程比较大文件比较多时，不使用Makefile几乎是不可能的。

下面介绍本次实验用到的Makefile文件。

```
CC=arm-linux-gcc
EXEC=hello
OBS=hello.o
CFLAGS+=
LDFLAGS+=

all:$(EXEC)
$(EXEC):$(OBS)
    $(CC) $(LDFLAGS) -o $@ $(OBS)

clean:
    rm -f $(EXEC) *.elf *.gdb *.o
```

★注意：“\$(CC) \$(LDFLAGS) -o \$@ \$(OBS)”和“-rm -f \$(EXEC) *.elf *.gdb *.o”前空白由一个Tab制表符生成，不能单纯由空格来代替。

与上面编写hello.c的过程类似，用vim来创建一个Makefile文件并将代码录入其中。

```
[root@zxt hello]# vi Makefile
```

7. 编译应用程序。

在上面的步骤完成后，就可以在hello目录下运行“make”命令来编译程序。如果进行了修改，则重新编译需要运行：

```
[root@zxt hello]# make clean
[root@zxt hello]# make
```

★注意：必须在root用户下进行编译，否则会提示编译器未找到错误。

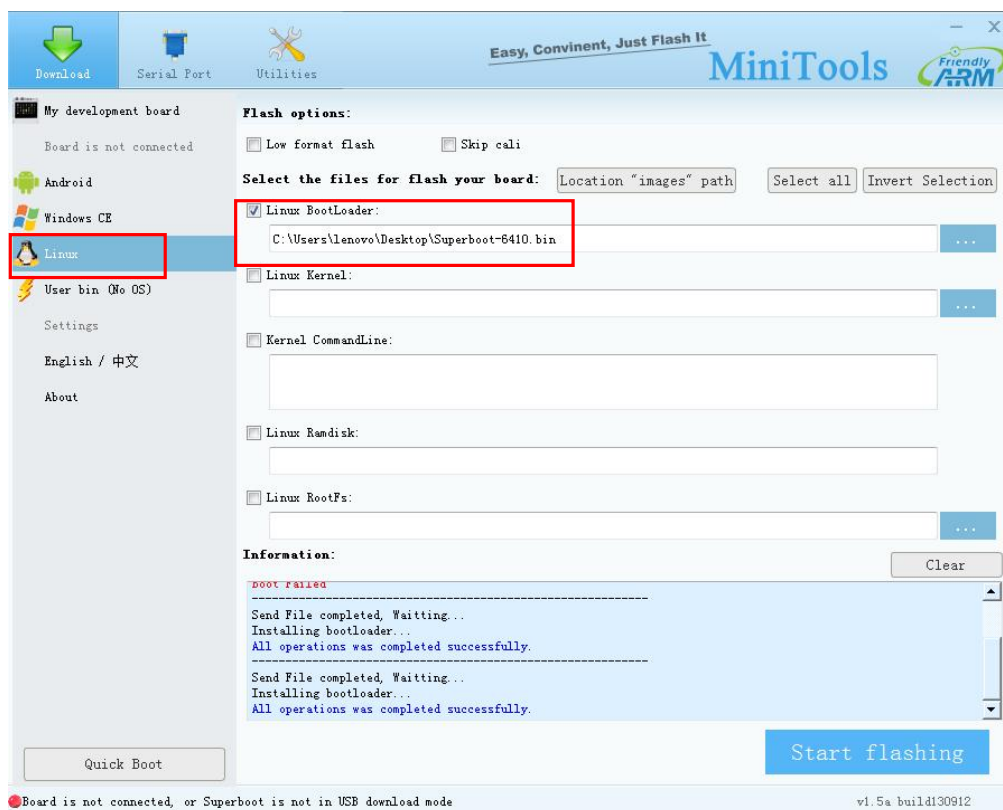
8. 下载调试。

(1) 首先使用串口线连接开发板的串口（COM0）和PC机的RS232串口（或使用USB转串线虚拟出的串口）；然后将Tiny6410开发板的启动模式拨动开关S2拨至“Nand”位置，表示从Nand Flash启动系统；再将电源拨动开关S1拨至“ON”，启动嵌入式Linux系统运行，此时开发板LCD屏幕左上角会显示一个企鹅图标，待系统初始化完毕显示主页面。

★注意：由于实验四（PWM定时器实验）的程序下载时覆盖了Nand Flash中原有的Bootloader程序，因此在完成实验四后再做本实验时，开发板中原有的嵌入式Linux系统将无法启动。此时，需要按如下操作将Bootloader程序重新写入Nand Flash中：

① 在开发板断电的情况下，先用MicroUSB线连接开发板与PC机，并将领取（已烧写了Superboot-6410.bin）的SD卡插入开发板的SD卡槽，同时确认启动模式拨动开关S2拨在“SDBOOT”位置，然后打开开发板的电源。此时开发板将进入USB下载模式，LCD显示屏上显示“USB Mode: Waiting...”。接着，在Windows系统中以管理员身份运行MiniTools。若USB连接成功，则开发板LCD屏幕上显示“USB Mode: Connected”，同时MiniTools软件左下角显示

② 使用Mini tool s软件向开发板的Nand Flash中烧写Bootloader程序。

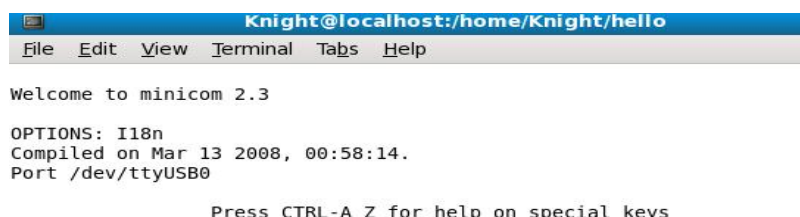


首先在软件界面左侧选择“Linux”，接着勾选“Linux BootLoader”复选框，并选择Superboot-6410.bin文件（图中的文件路径仅供参考，具体以实验电脑实际存放位置为准），

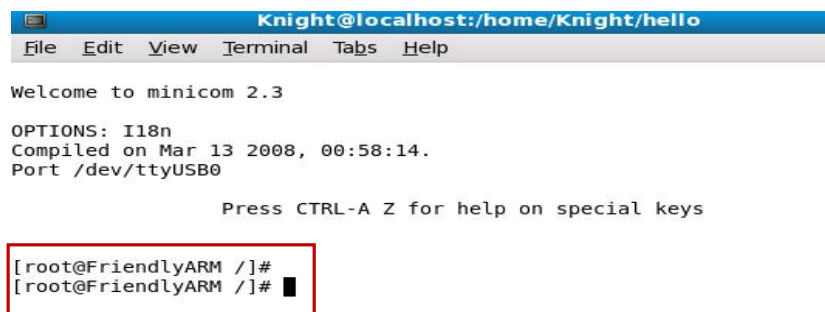
表示将Linux系统启动引导程序烧写到Nand Flash的0地址开始的区域。最后点击下方的【Start flashing】按钮，执行Bootloader的烧写。

关闭开发板电源，将拨动开关S2拨动至“NAND”位置，然后重新开启开发板电源，启动嵌入式Linux系统运行，此时开发板LCD屏幕左上角会显示一个企鹅图标。

(2) 确认当前为root用户后，在虚拟机的终端中输入minicom（非root用户可以尝试使用sudominicom）命令，若提示Device /dev/ttyUSB0 access failed: No such file or directory错误信息，则首先在命令行输入minicom -s，并用键盘方向键选择Serial port setup选项后回车，然后按“A”键将原路径中的ttyUSB0或者tty0，统一修改为ttyS0（合肥校区）或ttyS1（宣城校区）（★注意：字母“S”需要大写，且前面的/dev/路径不能删除），然后回车返回主界面，再选择Save setup as dfl后回车，最后选择Exit minicom退出并重新执行minicom命令。成功打开串口后出现下图：



回车后，出现下图：

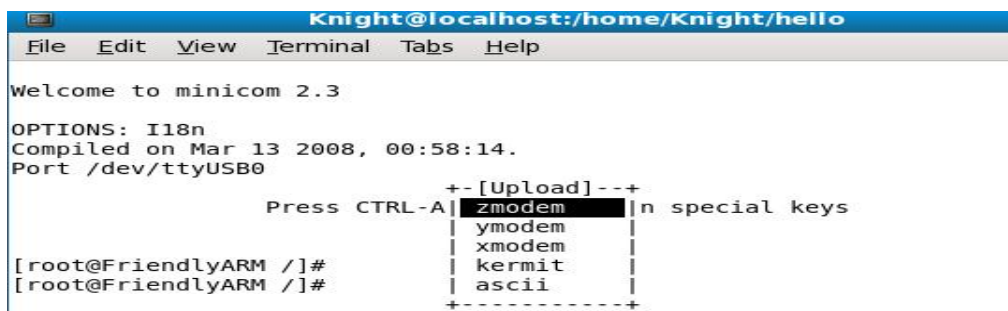


★★特别说明：必须出现红框中的路径（开发板中嵌入式Linux的路径）才能进行正常的下载，若没有出现，表明串口没有正常连接，需要检查硬件连线及相关串口参数。

★注意：关闭minicom请先按Ctrl+A，再按X。不要使用界面右上角的×退出，否则会在下一次执行minicom命令时提示串口被锁定（例如Device /dev/ttyS0 is Locked），拒绝访问。若出现该问题，可通过以下两种方式之一解决：

- ① 输入cd /var/lock，进入lock目录；输入rm -rf LCK...ttyS0（被锁定的串口文件具体名称，可通过ls命令查看），删除被锁定的串口文件，再重新执行minicom（或 sudo minicom）；
- ② 重启虚拟机解。

(3) 按Ctrl+A, 再按S, 出现下图:



```
Knicht@localhost:/home/Knicht/hello
File Edit View Terminal Tabs Help

Welcome to minicom 2.3
OPTIONS: I18n
Compiled on Mar 13 2008, 00:58:14.
Port /dev/ttyUSB0

Press CTRL-A +- [Upload] --+
                        | zmodem | n special keys
                        | ymodem |
                        | xmodem |
                        | kermit  |
                        | ascii   |
                        +-----+

[root@FriendlyARM /]#
[root@FriendlyARM /]#
```

选择第一个zmodem, 回车后, 出现下图:



```
Knicht@localhost:/home/Knicht/hello
File Edit View Terminal Tabs Help

We+-----[Select one or more files for upload]-----+
|Directory: /root|
|OP| [...] |
|Co| [.designer] |
|Po| [.gconf] |
|   | [.gconfd] |
|   | [.gnome2] |
|   | [.gnome2_private] |
|   | [.qt] |
|[r] .bash_history |
|[r] .bash_logout |
|   .bash_profile |
|   .bashrc |
|   .bashrc~ |
|   .cshrc |
|   .designerrc |
|   .designerrc~b |
|               | ( Escape to exit, Space to tag ) |
+-----+

[Goto] [Prev] [Show] [Tag] [Untag] [Okay]

CTRL-A Z for help |115200 8N1 | NOR | Minicom 2.3 | VT102 | Offline
```

回车后, 出现下图:



```
Knicht@localhost:/home/Knicht/hello
File Edit View Terminal Tabs Help

We+-----[Select one or more files for upload]-----+
|Directory: /root|
|OP| [...] |
|Co| [.designer] |
|Po| [.gconf] |
|   | [.gconfd] |
|   | [.gnome2] |
|   | [.gnome2_private] |
|   | [.qt] |
|[r] .bash_history |
|[r] .bash_logout |
|   .bash_profile |
|   .bashrc |
|   .bashrc~ |
|   .cshrc |
|   .designerrc |
|   .designerrc~b |
|               | ( Escape to exit, Space to tag ) |
+-----+

[No file selected - enter filename:
|> |

[Goto] [Prev] [Show] [Tag] [Untag] [Okay]

CTRL-A Z for help |115200 8N1 | NOR | Minicom 2.3 | VT102 | Offline
```

输入绝对路径或者通过键盘快捷键（推荐使用——双击空格键进入下一级目录，单击空格键选中文件）选择文件后, 出现下图:

六、实验修改要求

将原程序中显示的“ Hello, World! ”修改为“ Hello, HFUT! ”。然后分别通过命令行输入自己的学号和当前日期，提示符分别是“ Please enter your student ID: ”和“ Please enter the current date: ”。若输入的学号的最后一位是偶数，则按格式“ 2019123456||2021-05-08”输出信息；若是奇数，则按格式“ 2019123457|2021-05-08”输出信息。

注意：所有输出的信息必须单独显示，不能与其它字符显示在同一行。

实验七 多线程应用程序设计

一、实验目的

了解多线程程序设计的基本原理；学习pthread库函数的使用。

二、实验内容

本次实验使用Fedora（合肥校区）/CentOS（宣城校区）操作系统环境, 安装ARM-Linux的开发库及编译器。创建一个新目录thread, 并在其中编写thread.c和Makefile文件。学习并熟悉几个重要的pthread库函数的使用, 掌握共享锁和信号量的使用方法。在此基础上, 运行make产生pthread程序, 并使用mini com串口方式连接开发主机进行运行实验。

三、预备知识

1. 有C语言基础。
2. 会使用Linux下常用的编辑器。
3. 掌握Makefile的编写和使用。
4. 了解Linux下的编译程序与交叉编译的过程。

四、实验设备及工具（包括软件调试工具）

硬件：Tiny6410嵌入式实验平台。

软件：PC机操作系统Fedora/CentOS+mini com+ARM-Linux开发环境

五、实验原理

1. 多线程程序的优缺点

多线程程序作为一种多任务、并发的生活方式, 有以下的优点:

(1) 提高应用程序响应。这对图形界面的程序尤其有意义, 当一个操作耗时很长时, 整个系统都会等待这个操作, 此时程序不会响应键盘、鼠标、菜单的操作, 而使用多线程技术, 将耗时的操作置于一个新的线程, 可以避免这种尴尬的情况。

(2) 使多CPU系统更加有效。操作系统会保证当线程数不大于CPU数目时, 不同的线程运行于不同的CPU上。

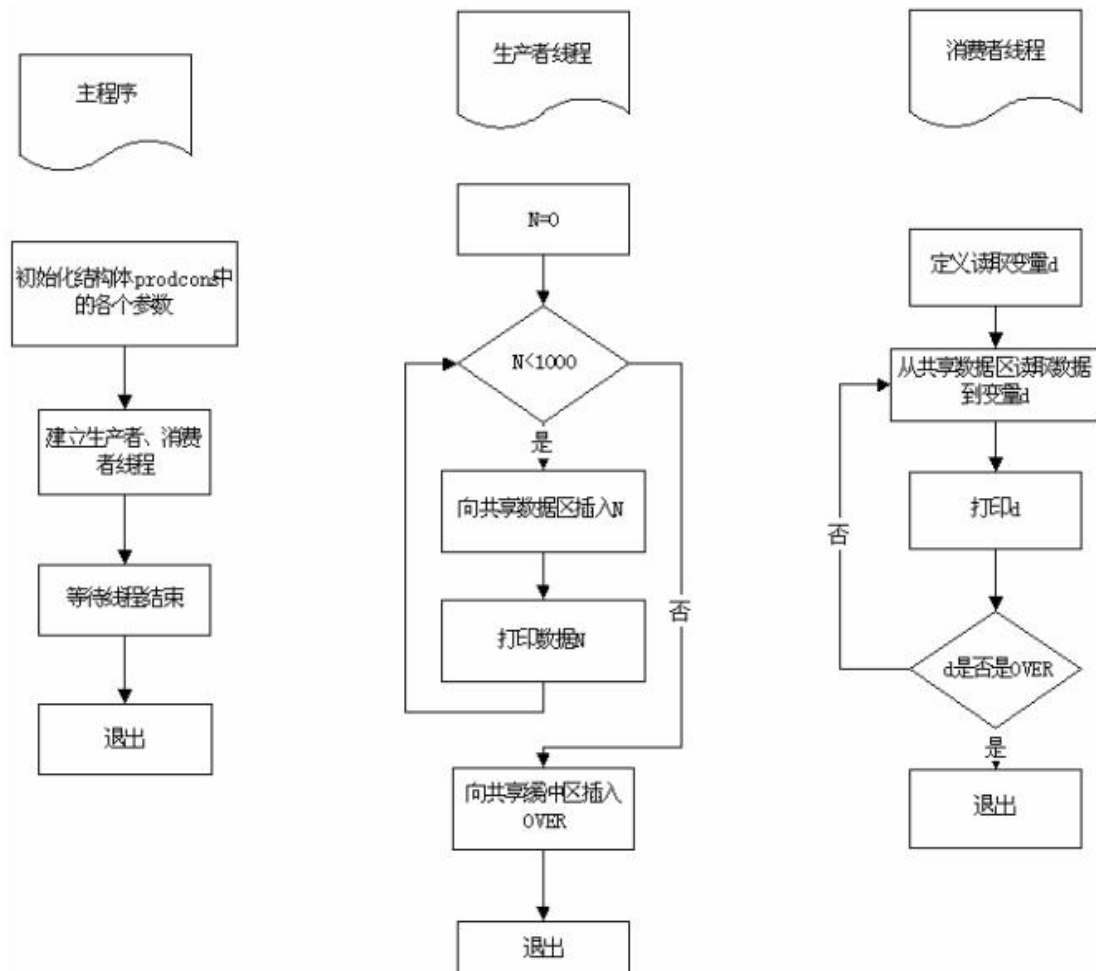
(3) 改善程序结构。一个既长又复杂的进程可以考虑分为多个线程, 成为几个独立或半

独立的运行部分，这样的程序会有利于理解和修改。

LIBC中的pthread库提供了大量的API函数，为用户编写应用程序提供支持。

2. 实验源代码结构流程图

实验为著名的生产者-消费者问题模型的实现，主程序中分别启动生产者线程和消费者线程。生产者线程不断顺序地将0到1000的数字写入共享的循环缓冲区，同时消费者线程不断地从共享的循环缓冲区读取数据。流程图如下所示：



3. 生产者写入共享的循环缓冲区函数PUT

```
Void put(struct prodcons *b,int data)
{
    Pthread_mutex_lock(&b->lock); //获取互斥锁
    While ((b->writepos+1)%BUFFER_SIZE == b->readpos) //如果读写位置相同
    {
        Pthread_cond_wait(&b->notfull, &b->lock); //等待状态变量b->notfull, 不满则跳出阻塞
    }
}
```

```

    b->buffer[b->writepos] = data; //写入数据
    b->writepos++;
    if (b->writepos >= BUFFER_SIZE) b->writepos = 0;
    pthread_cond_signal (&b->notempty); //设置状态变量
    pthread_mutex_unlock(&b->lock); //释放互斥锁
}

```

4. 消费者读取共享的循环缓冲区函数GET

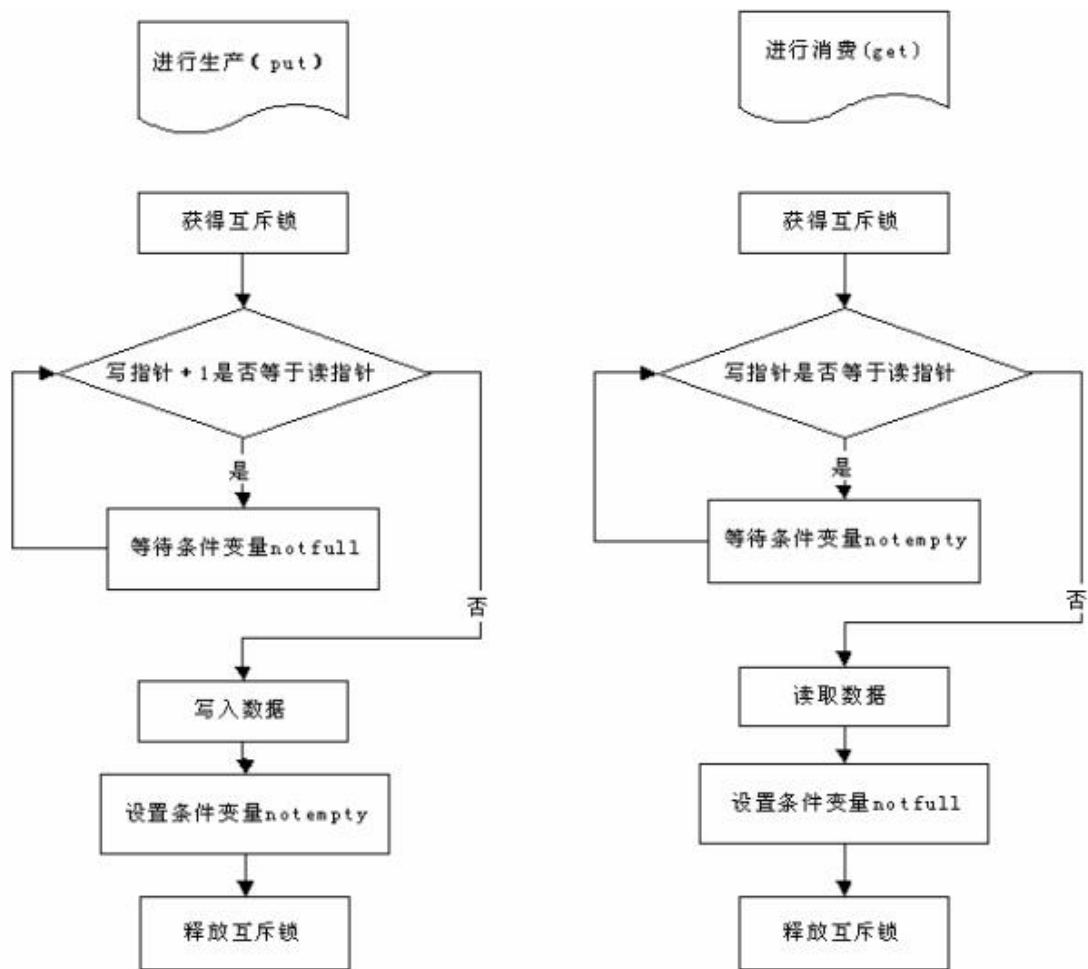
```

Int get (struct prodcons *b)
{
    Int data;
    Pthread_mutex_lock(&b->lock); //获取互斥锁
    While (b->writepos == b->readpos) // 如果读写位置相同
    {
        Pthread_cond_wait(&b->notempty, &b->lock); //等待状态变量b->notempty，不空则
跳出阻塞。否则无数据可读
    }
    Data = b->buffer[b->readpos]; //读取数据
    b->readpos++;
    if (b->readpos >= BUFFER_SIZE) b->readpos = 0;
    pthread_cond_signal (&b->notfull); //设置状态变量
    pthread_mutex_unlock(&b->lock); //释放互斥锁
    return data;
}

```

5. 生产、消费流程图：

生产消费流程图如下所示：



6. 主要的多线程API

(1) 线程创建函数

```
Int pthread_create (pthread_t * thread_id, _const pthread_attr_t *_attr, void
*(*_start_routine) (void *) , void * _restrict_arg);
```

(2) 获得父进程ID

```
Pthread_t pthread_self (void)
```

(3) 测试两个线程号是否相同

```
Int pthread_equal (pthread_t __thread1, pthread_t __thread2)
```

(4) 线程退出

```
Void pthread_exit (void * __retval)
```

(5) 等待指定的线程结束

```
Int pthread_join (pthread_t __th, void **__thread_return)
```

(6) 互斥量初始化

```
Pthread_mutex_init (pthread_mutex_t *, _const pthread_mutexattr_t *)
```

(7) 销毁互斥量

```
Int pthread_mutex_destroy (pthread_mutex_t *__mutex);
```

(8) 再试一次获得对互斥量的锁定（非阻塞）

```
Int pthread_mutex_trylock (pthread_mutex_t *__mutex);
```

(9) 锁定互斥量（阻塞）

```
Int pthread_mutex_lock (pthread_mutex_t *__mutex);
```

(10) 解锁互斥量

```
Int pthread_mutex_unlock (pthread_mutex_t *__mutex)
```

(11) 条件变量初始化

```
Int pthread_cond_init (pthread_cond_t *__restrict __cond, _const  
pthread_condattr_t *__restrict __cond_attr)
```

(12) 销毁条件变量COND

```
Int pthread_cond_destory (pthread_cond_t *__cond)
```

(13) 唤醒线程等待条件变量

```
Int pthread_cond_signal (pthread_cond_t *__cond)
```

(14) 等待条件变量（阻塞）

```
Int pthread_cond_wait (pthread_cond_t *__restrict __cond, Pthread_mutex_t  
*__restrict __mutex)
```

(15) 在指定的时间到达前等待条件变量

```
Int pthread_cond_timedwait (pthread_cond_t *__restrict __cond, Pthread_mutex_t  
*__restrict __mutex, _const struct timespec *__restrict __abstime)
```

PTHREAD库中还有大量的API函数，用户可以参考其他相关书籍。

7. 主要函数说明

(1) pthread_create 函数

```
Int pthread_create (pthread_t * thread_id, _const pthread_attr_t *_attr, void  
*(*_start_routine) (void *) , void * _restrict_arg);
```

线程创建函数第一个参数为指向线程标识符的指针，第二个参数用来设置线程属性，第三个参数是线程运行函数的起始地址，最后一个参数是运行函数的参数。这里，我们的函数thread不需要参数，所以最后一个参数设为空指针。第二个参数我们也设为空指针，这样将生成默认属性的线程。当创建线程成功时，函数返回0，若不为0则说明创建线程失败，常见

的错误返回代码为EAGAIN和EINVAL。前者表示系统限制创建新的线程，例如线程数目太多了；后者表示第二个参数代表的线程属性值非法。创建线程成功后，新创建的线程运行参数三和参数四确定的函数，原来的线程则继续运行下一行代码。

(2) pthread_join函数

用来等待一个线程的结束，函数原型为：

```
Int pthread_join(pthread_t __th,void **__thread_return)
```

第一个参数为被等待的线程标识符，第二个参数为一个用户定义的指针，它可以用来存储被等待线程的返回值。这个函数是一个线程阻塞的函数，调用它的函数将一直等待到被等待的线程结束为止，当函数返回时，被等待线程的资源被收回。

(3) pthread_exit函数

一个线程的结束有两种途径，一种是像我们上面的例子一样，函数结束了，调用它的线程也就结束了，另一种方式是通过函数pthread_exit来实现。它的函数原型为：

```
Void pthread_exit(void *_retval)
```

唯一的参数是函数的返回代码，只要pthread_join中的第二个参数thread_return不是NULL，这个值将被传递给thread_return. 最后要说明的是，一个线程不能被多个线程等待，否则第一个接收到信号的线程成功返回，其余调用pthread_join的线程则返回错误代码ESRCH。

8. 条件变量

使用互斥锁来实现线程间数据的共享和通信，互斥锁一个明显的缺点是它只有两种状态：锁定和非锁定。而条件变量通过允许线程阻塞和等待另一个线程发送信号的方法弥补了互斥锁的不足，它常和互斥锁一起使用。使用时，条件变量被用来阻塞一个线程，当条件不满足时，线程往往解开相应的互斥锁并等待条件发生变化。一旦其它的某个线程改变了条件变量，它将通知相应的条件变量唤醒一个或多个正被条件变量阻塞的线程。这些线程将重新锁定互斥锁并重新测试条件是否满足。一般说来，条件变量被用来进行线程间的同步。

(1) pthread_cond_init 函数

条件变量的结构为pthread_cond_t，函数pthread_cond_init() 被用来初始化一个条件变量，它的原型为：

```
Intt pthread_cond_init(pthread_cond_t *cond,__const pthread_condattr_t *cond_attr)
```

其中cond是一个指向结构pthread_cond_t的指针，cond_attr是一个指向结构

pthread_condattr_t的指针。结构pthread_condattr_t是条件变量的属性结构，和互斥锁一样我们可以用它来设置条件变量是进程内可用还是进程间可用，默认值是PTHREAD_PROCESS_PRIVATE,即此条件被同一进程内的各个线程使用。注意初始化条件变量只有未被使用时才能重新初始化或被释放。释放一个条件变量的函数为pthread_cond_destroy(pthread_cond_t cond)

(2) pthread_cond_wait函数

使线程阻塞在一个条件变量上。它的函数原型为：

```
Extern int pthread_cond_wait(pthread_cond_t *_restrict__cond, Pthread_mutex_t *_restrict __mutex )
```

线程解开mutex指向的锁并被条件变量cond阻塞。线程可以被函数pthread_cond_signal和函数pthread_cond_broadcast唤醒，但是要注意的是，条件变量只是起阻塞和唤醒线程的作用，具体的判断条件还需用户给出，例如一个变量是否为0等等，这一点我们从后面的例子中可以看到。线程被唤醒后，它将重新检查判断条件是否满足，如果还不满足，一般说来线程应该仍阻塞在这里，被等待下一次唤醒。这个过程一般用while语句实现。

(3) pthread_cond_timewait函数

另一个用来阻塞线程的函数是pthread_cond_timewait()，它的原型为：

```
Extern int pthread_cond_timewait __P(pthread_cond_t *_cond, pthread_mutex_t *___mutex, __const struct timespec *__abstime);
```

它比函数pthread_cond_wait()多了一个时间参数，经历abstime段时间后，即使条件变量不满足，阻塞也被解除。

(4) pthread_cond_signal函数

原型为：Extern int pthread_cond_signal (pthread_cond_t *__cond);

它用来释放被阻塞在条件变量cond上的一个线程。多个线程阻塞在此条件变量上时，哪一个线程被唤醒是由线程的调度策略所决定的。要注意的是，必须用保护条件变量的互斥锁来保护这个函数，否则条件满足信号又可能在测试条件和调用pthread_cond_wait函数之间被发出，从而造成无限制的等待。

六、实验步骤

1. 使用vi编辑器或其他编辑器编写源代码。若实验已提供源代码，则需阅读并理解代码的含义。

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <pthread.h>

int task1(int *cnt)
{
    while(*cnt < 5)
    {
        sleep(1);
        (*cnt)++;
        printf("task1 cnt = %d.\n", *cnt);
    }

    return (*cnt);
}

int task2(int *cnt)
{
    while(*cnt < 5)
    {
        sleep(2);
        (*cnt)++;
        printf("task2 cnt = %d.\n", *cnt);
    }

    return (*cnt);
}

int main(int argc, char **argv)
{
    int result;
    int t1 = 0;
    int t2 = 0;
    int rt1, rt2;
    pthread_t thread1, thread2;

    /* create the first thread. */
    result = pthread_create(&thread1, PTHREAD_CREATE_JOINABLE, (void
*)task1, (void *)&t1);
    if(result)
    {
        perror("pthread_create: task1.\n");
        exit(EXIT_FAILURE);
    }

```

```

    }

    /* create the second thread. */
    result = pthread_create(&thread2, PTHREAD_CREATE_JOINABLE, (void
*)task2, (void *)&t2);
    if(result)
    {
        perror("pthread_create: task2.\n");
        exit(EXIT_FAILURE);
    }

    pthread_join(thread1, (void *)&rt1);
    pthread_join(thread2, (void *)&rt2);

    printf("total %d times.\n", t1+t2);
    printf("return value of task1: %d.\n", rt1);
    printf("return value of task2: %d.\n", rt2);

    exit(EXIT_SUCCESS);
}

```

2. 运行make产生pthread可执行文件

```

[ root@os thread]# make clean
rm -f thread *.elf *.gdb *.o
[ root@os thread]# make
arm-linux-gcc -c -o thread.o thread.c
arm-linux-gcc -lpthread -o thread thread.o
[ root@os thread]# ls
Makefile Makefile~ thread thread.c thread.o
[ root@os thread]# █

```

3. 首先按实验六给出的方法通过串口下载程序到目标开发板上, 然后运行pthread程序, 并观察运行结果的正确性。如下图所示:

```

[ root@FriendlyARM /root]# chmod +x thread
[ root@FriendlyARM /root]# ./thread
task1 cnt = 1.
task2 cnt = 1.
task1 cnt = 2.
task1 cnt = 3.
task2 cnt = 2.
task1 cnt = 4.
task1 cnt = 5.
task2 cnt = 3.
task2 cnt = 4.
task2 cnt = 5.
total 10 times.
return value of task1: 5.
return value of task2: 5.
[ root@FriendlyARM /root]# █

```

实验八 串行端口程序设计

一、实验目的

了解在Linux环境下串行程序设计的基本方法。掌握终端的主要属性及设置方法，熟悉终端I/O函数的使用。学习使用多线程来完成串口的收发处理。

二、实验内容

本次实验使用Fedora（合肥校区）/CentOS（宣城校区）操作系统环境, 安装ARM-Linux的开发库及编译器。创建一个新目录serialPort，并在其中编写serialPort.c和Makefile文件。学习终端I/O函数的使用方法，并学习将多线程编程应用到串口的接收和发送程序设计中。

三、预备知识

1. 有C语言基础。
2. 会使用Linux下常用的编辑器。
3. 掌握Makefile的编写和使用。
4. 了解Linux下的编译程序与交叉编译的过程。

四、实验设备及工具（包括软件调试工具）

硬件：Tiny6410嵌入式实验平台。

软件：PC机操作系统Fedora/CentOS+minicom+ARM-Linux 开发环境

五、实验原理

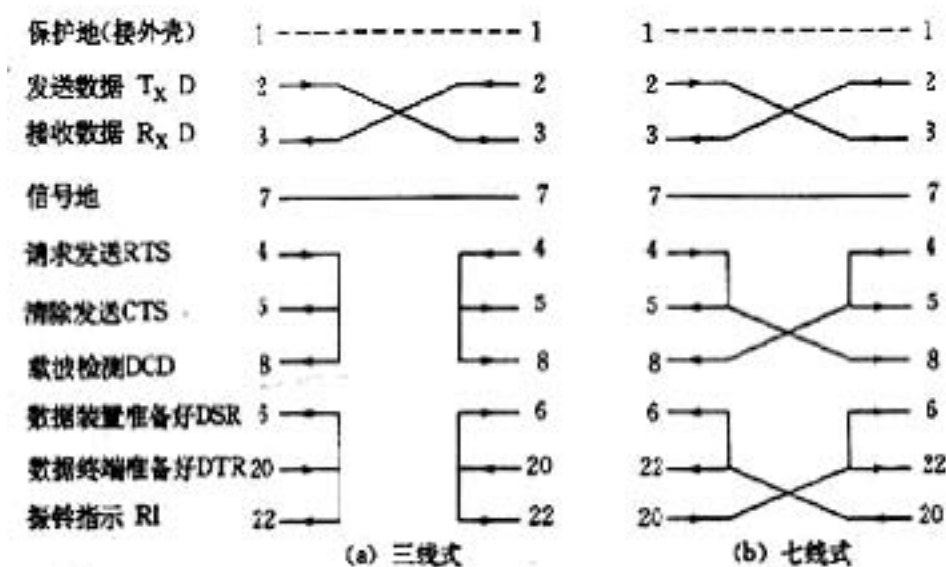
异步串行I/O方式是将传输数据的每个字符一位接一位(例如先低位、后高位)地传送。数据的各不同位可以分时使用同一传输通道，因此串行I/O可以减少信号连线，最少用一对线即可进行。接收方对于同一根线上一连串的数字信号，首先要分割成位，再按位组成字符。为了恢复发送的信息，双方必须协调工作。在微型计算机中大量使用异步串行I/O方式，双方使用各自的时钟信号，而且允许时钟频率有一定误差，因此实现较容易。但是由于每个字符都要独立确定起始和结束(即每个字符都要重新同步)，字符和字符间还可能有长度不定的空闲时间，因此效率较低。



上图给出异步串行通信中一个字符的传送格式。开始前，线路处于空闲状态，送出连续“1”。传送开始时首先发一个“0”作为起始位，然后出现在通信线上的是字符的二进制编码数据。每个字符的数据位长可以约定为5位、6位、7位或8位，一般采用ASCII编码。后面是奇偶校验位，根据约定，用奇偶校验位将所传字符中为“1”的位数凑成奇数个或偶数个。也可以约定不要奇偶校验，这样就取消奇偶校验位。最后是表示停止位的“1”信号，这个停止位可以约定持续1位、1.5位或2位的时间宽度。至此一个字符传送完毕，线路又进入空闲，持续为“1”。经过一段随机的时间后，下一个字符开始传送才又发出起始位。每一个数据位的宽度等于传送波特率的倒数。微机异步串行通信中，常用的波特率为50, 95, 110, 150, 300, 600, 1200, 2400, 4800, 9600等。接收方按约定的格式接收数据，并进行检查，可以查出以下三种错误：

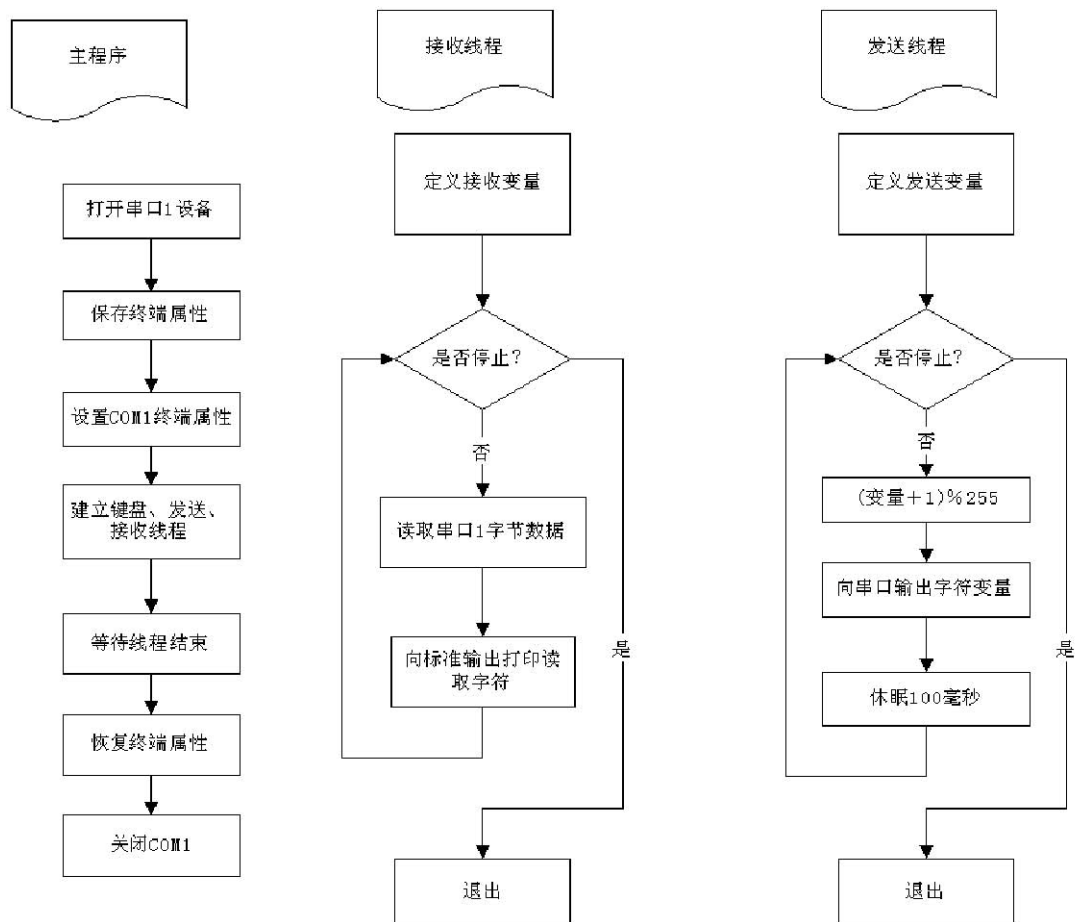
- 奇偶错：在约定奇偶检查的情况下，接收到的字符奇偶状态和约定不符。
- 帧格式错：一个字符从起始位到停止位的总位数不对。
- 溢出错：若先接收的字符尚未被微机读取，后面的字符又传送过来，则产生溢出错。

每一种错误都会给出相应的出错信息，提示用户处理。一般串口调试都使用空的MODEM连接电缆，其连接方式如下：



六、程序分析

Linux 操作系统从一开始就对串行口提供了很好的支持，为进行串行通讯提供了大量的函数，我们的实验主要是为掌握在Linux 中进行串行通讯编程的基本方法。本实验的程序流程图如下：



本实验的代码如下：

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <errno.h>

int set_opt(int,int,int,char,int);
void leds_control(int);

void main(int argc, char* argv[])
{
    int fd, read_num = 0, flag = 0;
```

```

//char *uart3 = "/dev/ttySAC3";
unsigned char buffer[1024],buffer_test[1024];
memset(buffer, 0, 1024);
memset(buffer_test, 0, 1024);

if(argc < 2)
{
    printf("usage: ./uarttest /dev/ttySAC3\n");
    return;
}

char *uart_out = "please input\r\n";
char *head_out = " output is \r\n";

if((fd = open(argv[1], O_RDWR|O_NOCTTY|O_NDELAY))<0)
{
    printf("open %s is failed\n", argv[1]);
    return;
}
else{
    set_opt(fd, 115200, 8, 'N', 1);
    write(fd,uart_out, strlen(uart_out));

while(1){
    memset(buffer, 0, 1024);
    read_num = read(fd, buffer, 100);
    if(read_num>0){
        write(fd,"\r\n",2);
        write(fd,buffer,strlen(buffer));

/*  flag++;
    buffer_test[flag] = buffer[0];
    if(buffer_test[flag]=='\n'){
        write(fd,"\r\n",2);
        write(fd,buffer_test,strlen(buffer_test));
        flag = 0;
    }*/
    write(fd,"\r\n",2);
    memset(buffer,0,read_num);
    sprintf(buffer,"\n output num is %d\r\n",read_num);

    write(fd,buffer,strlen(buffer));
    }
}
}
}

```

```

int set_opt(int fd,int nSpeed, int nBits, char nEvent, int nStop)
{
    struct termios newtio,oldtio;
    if ( tcgetattr( fd,&oldtio) != 0) {
        perror("SetupSerial 1");
        return -1;
    }
    bzero( &newtio, sizeof( newtio ) );
    newtio.c_cflag |= CLOCAL | CREAD;
    newtio.c_cflag &= ~CSIZE;

    switch( nBits )
    {
        case 7:
            newtio.c_cflag |= CS7;
            break;
        case 8:
            newtio.c_cflag |= CS8;
            break;
    }

    switch( nEvent )
    {
        case 'O':
            newtio.c_cflag |= PARENB;
            newtio.c_cflag |= PARODD;
            newtio.c_iflag |= (INPCK | ISTRIP);
            break;
        case 'E':
            newtio.c_iflag |= (INPCK | ISTRIP);
            newtio.c_cflag |= PARENB;
            newtio.c_cflag &= ~PARODD;
            break;
        case 'N':
            newtio.c_cflag &= ~PARENB;
            break;
    }
    switch( nSpeed )
    {
        case 2400:
            cfsetispeed(&newtio, B2400);
            cfsetospeed(&newtio, B2400);
            break;
        case 4800:
            cfsetispeed(&newtio, B4800);
            cfsetospeed(&newtio, B4800);

```



```

        break;
    case 9600:
        cfsetispeed(&newtio, B9600);
        cfsetospeed(&newtio, B9600);
        break;
    case 115200:
        cfsetispeed(&newtio, B115200);
        cfsetospeed(&newtio, B115200);
        break;
    case 460800:
        cfsetispeed(&newtio, B460800);
        cfsetospeed(&newtio, B460800);
        break;
    case 921600:
        printf("B921600\n");
        cfsetispeed(&newtio, B921600);
        cfsetospeed(&newtio, B921600);
        break;
    default:
        cfsetispeed(&newtio, B9600);
        cfsetospeed(&newtio, B9600);
        break;
}
if( nStop == 1 )
    newtio.c_cflag &= ~CSTOPB;
else if ( nStop == 2 )
    newtio.c_cflag |= CSTOPB;
newtio.c_cc[VTIME] = 0;
newtio.c_cc[VMIN] = 0;
tcflush(fd,TCIFLUSH);
if((tcsetattr(fd,TCSANOW,&newtio))!=0)
{
    perror("com set error");
    return -1;
}
// printf("set done!\n\r");
return 0;
}

```

下面是对程序主要部分的简单分析：

(1) 头文件

```

#include <stdio.h>      /* 标准输入输出定义*/
#include <stdlib.h>     /* 标准函数库定义*/
#include <unistd.h>     /*linux 标准函数定义*/
#include <sys/types.h>  /*基本系统数据类型*/

```

```
#include <sys/stat.h> /*获取文件属性*/
#include <fcntl.h> /* 文件控制定义*/
#include <termios.h> /*PPSIX 终端控制定义*/
#include <errno.h> /* 错误号定义*/
```

(2) 串口设置最基本的设置串口包括波特率、校验位和停止位设置。串口的设置主要是设置 struct termios 结构体的各成员值，关于该结构体的定义可以查看 /arm2410s/kernel-2410s/include/asm/termios.h 文件。

```
struct termio
{
    unsigned short c_iflag; /* 输入模式标志 */
    unsigned short c_oflag; /* 输出模式标志 */
    unsigned short c_cflag; /* 控制模式标志 */
    unsigned short c_lflag; /* local mode flags */
    unsigned char c_line; /* line discipline */
    unsigned char c_cc[NCC]; /* control characters */
};
```

七、实验步骤

1. 编写源代码。若实验已提供源代码，则应阅读并理解源代码的含义。

使用vi 编辑器或其他编辑器编辑或阅读理解源代码。

2. 编译应用程序

运行make产生serial Port可执行文件，如下图所示：

```
[root@os serialPort]# ls
Makefile serialPort.c
[root@os serialPort]# make
arm-linux-gcc -c -o serialPort.o serialPort.c
arm-linux-gcc -o serialPort serialPort.o
[root@os serialPort]# ls
Makefile serialPort serialPort.c serialPort.o
[root@os serialPort]# █
```

3. 下载调试

首先按实验六给出的方法通过串口下载程序到目标开发板上，然后运行serial Port程序，如下图所示：

```

[ root@FriendlyARM /root]#
[ root@FriendlyARM /root]# ls
Applications  adc-test      led           thread
Documents     hello         qt4           tmp
Settings      i2c          serialPort
[ root@FriendlyARM /root]# chmod +x serialPort
[ root@FriendlyARM /root]# ./serialPort /dev/ttySAC
/dev/ttySAC0 /dev/ttySAC1 /dev/ttySAC2 /dev/ttySAC3
[ root@FriendlyARM /root]# ./serialPort /dev/ttySAC3

```

然后打开PC端的串口助手（具体串口通信参数如下图所示），观察运行结果的正确性。



实验九 Linux Qt 开发实验

一、实验目的

掌握Qt程序设计；掌握Qt程序交叉编译；掌握在开发板上运行Qt程序。

二、实验内容

本次实验使用Fedora（合肥校区）/CentOS（宣城校区）操作系统环境, 安装ARM-Linux的开发库及编译器。使用Qt Creator创建一个led工程，并编写led.cpp、led.h和main.cpp等源程序文件，从而实现通过Qt软件界面对目标开发板上的LED进行控制。学习Linux Qt程序的设计过程和方法。

三、预备知识

1. 有C++语言基础。
2. 会使用Linux下的UI开发工具Qt Creator。

四、实验设备及工具（包括软件调试工具）

硬件：Tiny6410嵌入式实验平台。

软件：PC机操作系统Fedora/CentOS+minicom+ARM-Linux 开发环境

五、实验步骤

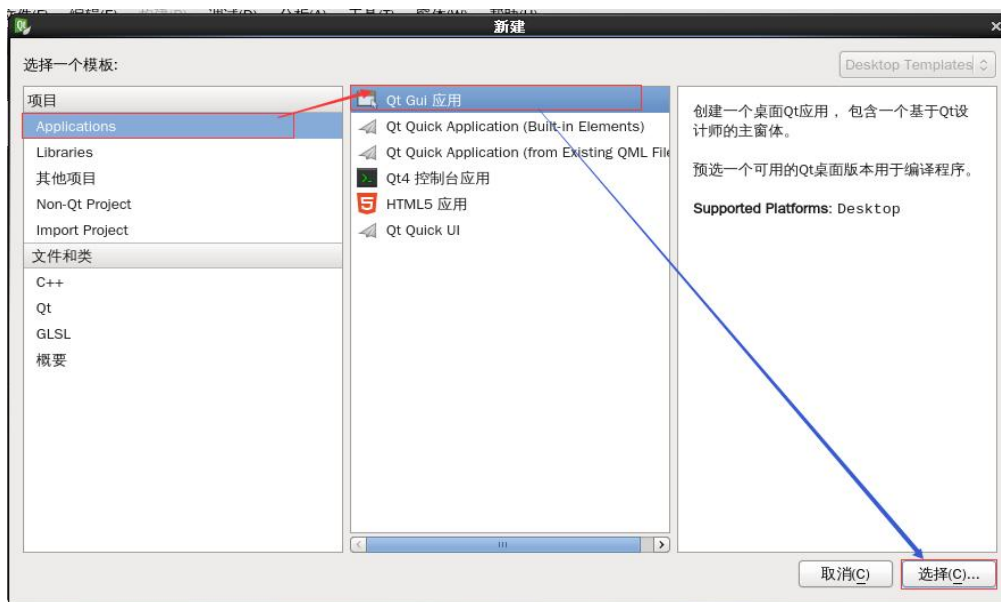
开发Qt程序使用的IDE是Qt Creator，其快捷方式在虚拟机桌面上。打开Qt Creator，其主界面如下图所示：



1. 如下图所示，点击新建文件或工程。



2. 弹出工程配置窗口，配置如下图所示。



3. 配置工程名称和路径。



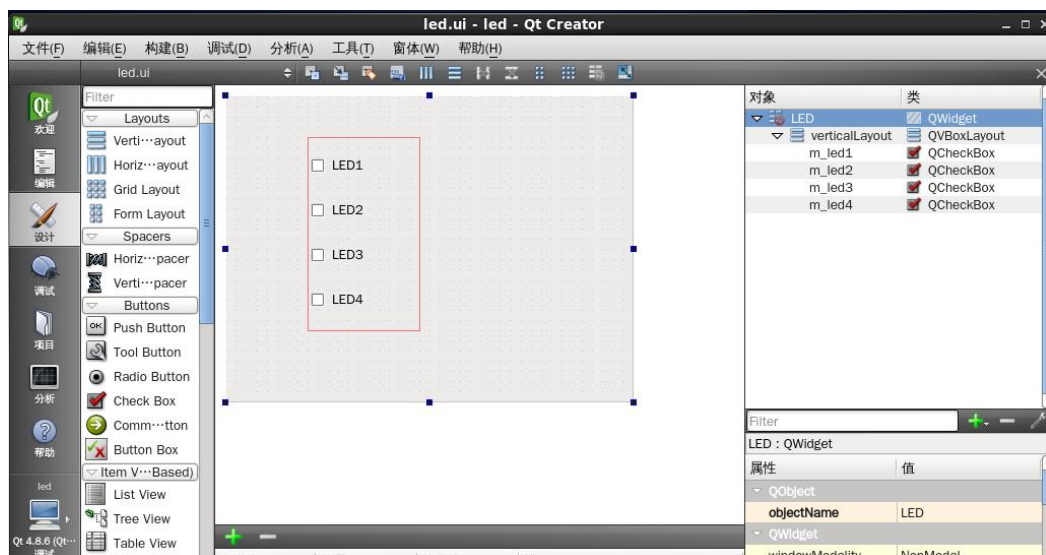
4. 配置类名称，点击下一步。



5. 点击完成。



6. 界面如下图所示。



代码如下:

(1) led.cpp

```
#include "led.h"
#include "ui_led.h"
#include <qcheckbox.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <fcntl.h>

LED::LED(QWidget *parent) :
    QWidget(parent),
    ui(new Ui::LED)
{
    ui->setupUi(this);
    ::system("kill -s STOP `pidof led-player`");
    m_fd = ::open("/dev/leds0", O_RDONLY);
    if (m_fd < 0)
    {
        m_fd = ::open("/dev/leds", O_RDONLY);
    }
    connect(ui->m_led1, SIGNAL(clicked()), this, SLOT(checkBoxClicked() ));
    connect(ui->m_led2, SIGNAL(clicked()), this, SLOT(checkBoxClicked() ));
    connect(ui->m_led3, SIGNAL(clicked()), this, SLOT(checkBoxClicked() ));
    connect(ui->m_led4, SIGNAL(clicked()), this, SLOT(checkBoxClicked() ));
    checkBoxClicked();
}

LED::~LED()
{
    delete ui;
}

void LED::checkBoxClicked()
{
    ioctl(m_fd, int(ui->m_led1->isChecked()), 0);
    ioctl(m_fd, int(ui->m_led2->isChecked()), 1);
    ioctl(m_fd, int(ui->m_led3->isChecked()), 2);
    ioctl(m_fd, int(ui->m_led4->isChecked()), 3);
}
```

(2) led.h

```
#ifndef LED_H
#define LED_H
#include <QWidget>

namespace Ui{
class LED;
}

class LED : public QWidget
{
    Q_OBJECT

public:
    explicit LED(QWidget *parent = 0);
    ~LED();

private slots:
    void checkBoxClicked();

private:
    Ui::LED *ui;
    int m_fd;
};

#endif // LED_H
```

(3) main.cpp

```
#include <QApplication>
#include "led.h"

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    LED w;
    w.show();
    return a.exec();
}
```

7. 交叉编译。

- (1) 打开命令行切换到led工程所在目录。


```
[root@os led]# ls
led.cpp led.h led.pro led.ui main.cpp
[root@os led]#
```

(2) 执行arm-qmake生成MakeFile文件。

```
[root@os led]# ls
led.cpp led.h led.pro led.ui main.cpp
[root@os led]# arm-qmake
[root@os led]# ls
led.cpp led.h led.pro led.ui main.cpp Makefile
[root@os led]#
```

(3) 然后执行make命令。

```
root@os led]# make
/usr/local/Trolltech/QtEmbedded-4.8.5-arm/bin/uic led.ui -o ui_led.h
arm-linux-g++ -c -pipe -O2 -Wall -W -D_REENTRANT -DQT_NO_DEBUG -DQT_GUI_LIB -DQT_NETWORK_LIB -DQT_CORE_LIB -DQT_SHARED -I/usr/local/Trolltech/QtEmbedded-4.8.5-arm/mkspecs/qws/linux-arm-g++ -I. -I/usr/local/Trolltech/QtEmbedded-4.8.5-arm/include/QtCore -I/usr/local/Trolltech/QtEmbedded-4.8.5-arm/include/QtNetwork -I/usr/local/Trolltech/QtEmbedded-4.8.5-arm/include/QtGui -I/usr/local/Trolltech/QtEmbedded-4.8.5-arm/include -I. -I. -o main.o main.cpp
arm-linux-g++ -c -pipe -O2 -Wall -W -D_REENTRANT -DQT_NO_DEBUG -DQT_GUI_LIB -DQT_NETWORK_LIB -DQT_CORE_LIB -DQT_SHARED -I/usr/local/Trolltech/QtEmbedded-4.8.5-arm/mkspecs/qws/linux-arm-g++ -I. -I/usr/local/Trolltech/QtEmbedded-4.8.5-arm/include/QtCore -I/usr/local/Trolltech/QtEmbedded-4.8.5-arm/include/QtNetwork -I/usr/local/Trolltech/QtEmbedded-4.8.5-arm/include/QtGui -I/usr/local/Trolltech/QtEmbedded-4.8.5-arm/include -I. -I. -o led.o led.cpp
/usr/local/Trolltech/QtEmbedded-4.8.5-arm/bin/moc -DQT_NO_DEBUG -DQT_GUI_LIB -DQT_NETWORK_LIB -DQT_CORE_LIB -DQT_SHARED -I/usr/local/Trolltech/QtEmbedded-4.8.5-arm/mkspecs/qws/linux-arm-g++ -I. -I/usr/local/Trolltech/QtEmbedded-4.8.5-arm/include/QtCore -I/usr/local/Trolltech/QtEmbedded-4.8.5-arm/include/QtNetwork -I/usr/local/Trolltech/QtEmbedded-4.8.5-arm/include/QtGui -I/usr/local/Trolltech/QtEmbedded-4.8.5-arm/include -I. -I. led.h -o moc_led.cpp
arm-linux-g++ -c -pipe -O2 -Wall -W -D_REENTRANT -DQT_NO_DEBUG -DQT_GUI_LIB -DQT_NETWORK_LIB -DQT_CORE_LIB -DQT_SHARED -I/usr/local/Trolltech/QtEmbedded-4.8.5-arm/mkspecs/qws/linux-arm-g++ -I. -I/usr/local/Trolltech/QtEmbedded-4.8.5-arm/include/QtCore -I/usr/local/Trolltech/QtEmbedded-4.8.5-arm/include/QtNetwork -I/usr/local/Trolltech/QtEmbedded-4.8.5-arm/include/QtGui -I/usr/local/Trolltech/QtEmbedded-4.8.5-arm/include -I. -I. -o moc_led.o moc_led.cpp
arm-linux-g++ -Wl, -O1 -Wl, -rpath, /usr/local/Trolltech/QtEmbedded-4.8.5-arm/lib -o led main.o led.o moc_led.o -L/usr/local/Trolltech/QtEmbedded-4.8.5-arm/lib -lQtGui -L/usr/local/Trolltech/QtEmbedded-4.8.5-arm/lib -lQtNetwork -lQtCore -lpthread
root@os led]#
```

(4) 查看生成的可执行文件。

```
root@os led]# ls
led led.h led.pro main.cpp Makefile moc_led.o
led.cpp led.o led.ui main.o moc_led.cpp ui_led.h
[root@os led]#
```

(5) 按实验六给出的方法将led文件通过串口下载到目标开发板中，并执行chmod a+x led添加可执行权限。

```
root@FriendlyARM /opt]# ls
root@FriendlyARM /opt]# ls
Qtopia kde led python
root@FriendlyARM /opt]# chmod a+x led
root@FriendlyARM /opt]#
```

(6) 在开发板中执行vi /etc/init.d/rcS编辑该配置文件。

① 注释 /bin/qttopia &, 如下图所示:

```
echo "
sleep 1

echo "
/etc/rc.d/init.d/asocd start
echo "Starting sound card..." > /dev/tty1
echo "

/sbin/ifconfig lo 127.0.0.1
/etc/init.d/ifconfig-eth0
fa-network-service

# /bin/qttopia &
echo "
echo "Starting Qttopia, please waiting..." > /dev/tty1
~
```

② 修改后保存并退出, 然后重启开发板。

(7) 重启后执行命令source /bin/setqt4env加载Qt环境配置选项。

```
[root@FriendlyARM /opt]# source /bin/setqt4env
[root@FriendlyARM /opt]#
```

(8) 执行命令./led -qws, 运行Qt程序。运行效果如下图所示:



(9) 点击复选框可控制位于核心板上的LED亮灭。