龙芯中科
LOONGSON TECHNOLOGY

# 1B 开发板用户手册

| 文件状态： | 文件标识： | 1B 开发板用户手册 |
|---|---|---|
| [ ] 草稿 | 保密级别： | □高 □中 ■低 |
| [√] 正式发布 | 文档版本： | 1.1 |
| [ ] 正在修改 | 部门名称： | 产品部 |
| | 发布日期： | 2012-01 |

# 版 本 历 史

| 版本 | 日期 | 备注 |
|---|---|---|
| 1.0 | 2011-11-11 | 创建 |
| 1.1 | 2012-01-12 | 1） 修订 1B 开发板第二版硬件相关内容；<br>2） 修订其他章节与板子不对应内容。 |
| | | |

广州龙芯中科电子科技有限公司

广州大学城外环东路 232 号国家数字家庭基地（东区）4 楼 B 区 510006
电话：020—22900688
传真：020—22900699
网站：www.loongson.cn

# 目录

**1B 开发板用户手册** ..... 1
第一章　　前言 ..... 6
1.1 说明 ..... 6
1.2 名词解释 ..... 6
第二章　　1B 开发板介绍 ..... 7
2.1 开发板简介 ..... 7
2.2 开发板外观 ..... 7
2.3 开发板硬件资源 ..... 8
2.4 硬件介绍 ..... 8
2.4.1　CPU ..... 8
2.4.2 DDR2 ..... 9
2.4.3　Nand Flash ..... 10
2.4.4　LCD ..... 10
2.4.5　USB2.0 ..... 11
2.4.6　AC97 ..... 11
2.4.7　MAC ..... 11
2.4.8　SPI ..... 12
2.4.9　UART ..... 12
2.4.10　I2C ..... 14
2.4.11　PWM ..... 14
2.4.12　CAN 总线 ..... 15
2.4.13　实时时钟 RTC ..... 15
2.4.14　蜂鸣器 ..... 16
2.4.15　按键 ..... 16
2.4.16　LED ..... 17
2.5 开发板硬件应用说明 ..... 17
第三章 1B 开发板使用说明 ..... 20
3.1 开发板快速上手指南 ..... 20
3.1.1 外部接口的连线 ..... 20
3.1.2 设置终端仿真程序 ..... 20
3.1.3 恢复与更新 Linux 系统 ..... 24
第四章 在主机上搭建 LINUX 开发环境 ..... 28
4.1 安装 Ubuntu10.04 ..... 28
4.1.1 安装 VMware-workstation ..... 28
4.1.2 新建虚拟机 ..... 30
4.1.3 安装 Ubuntu 系统 ..... 34
4.1.4 备份 Ubuntu 系统 ..... 38
4.2 使用 Ubuntu10.04 ..... 41
4.2.1 Linux 终端 ..... 41
4.2.2 初体验 ..... 42

4.2.3 常用设置.....43
4.2.4 安装 TFTP.....47
4.3 建立交叉编译环境.....51
第五章 编译 BOOTLOADER（PMON）和 LINUX.....51
5.1 编译 BOOTLOADER（PMON）.....51
5.1.1 工具与依赖库安装.....52
5.1.2 配置与编译 pmon.....53
5.2 编译 Linux 内核.....54
5.2.1 配置内核.....54
5.2.2 编译 linux 内核.....55
5.3 制作文件系统镜像.....55
5.3.1 镜像文件制作工具.....55
5.3.2 镜像文件制作工具本机安装.....55
5.3.3 制作文件系统镜像文件.....56
第六章 烧写 BOOTLOADER（PMON）和 LINUX.....57
6.1 烧写 BOOTLOADER（PMON）.....57
6.1.1 烧写 PMON.....57
6.1.2 设置 IP 地址并测试.....57
6.1.3 网口更新 PMON.....57
6.1.4 串口更新 PMON.....57
6.1.5 PMON 的内置命令.....59
6.1.6 PMON 的启动设置.....61
6.2 烧写 Linux 内核.....63
6.2.1 烧写内核.....63
6.2.2 设置启动参数.....63
6.3 烧写文件系统镜像.....63
6.3.1 烧写文件系统镜像.....63
6.3.2 设置启动参数.....63
第七章 应用程序的移植.....64
7.1 Hello World.....64
7.2 应用程序的移植方式.....64
7.2.1 复制到介质（以 U 盘为例）.....65
7.2.2 通过网络（tftp）传输文件到开发板.....65
7.2.3 置于根文件系统目录下制作文件系统镜像.....65
7.2.4 通过 NFS（网络文件系统）直接运行.....66
7.3 启动脚本.....66
第八章 应用开发实验.....66
8.1 LINUX 基础实验.....66
8.1.1 实验一 shell 编程.....66
8.1.2 实验二 文件操作实验.....67
8.1.3 实验三 多线程实验.....68
8.1.4 实验四 多进程实验.....71
8.1.5 实验五 进程间通信实验.....71

8.1.6 实验六 网络编程实验.....75
8.2 1B 开发板外设测试实验.....81
8.2.1 AD 转换.....81
8.2.2 PWM.....82
8.2.3 蜂鸣器.....82
8.2.4 按键.....83
8.2.5 SD 卡.....84
8.2.6 U 盘.....85
8.2.7 音频.....85
8.2.8 网卡.....86
8.2.9 RTC 时钟.....86
8.2.10 串口.....87
8.3 LINUX GUI 实验.....87
8.3.1 实验一 QT3.....87
8.3.2 实验二 SDL.....96
8.4 LINUX 驱动程序实验.....100
8.4.1 ADC 驱动程序.....100
8.4.2 外部按键驱动.....104
8.4.3 RTC 驱动程序.....109
附录.....118
附录 1 Windows 与 Ubuntu 间文件的传输.....118
附录 2 Linux 常用命令详解.....118
附录 3 Windows 超级终端使用说明.....120
附录 4 内核配置详细说明.....125
4.1 各个驱动程序源代码位置.....125
4.2 手工定制 Linux 内核.....126
附录 5 制作根文件系统.....152
5.1 配置、编译 Busybox.....152
5.2 构建根文件系统.....164
附录 6 Minicom 使用指南.....167
6.1 安装 minicom.....167
6.2 配置 minicom.....168
6.3 使用 minicom.....170
附录 7 使用 EJTAG 烧写 BOOTLOADER（PMON）.....172
7.1 编译安装 ejtag.....172
附录 8 NFS 网络文件系统搭建.....173
8.1 安装 NFS.....173
8.2 配置 NFS.....175
8.3 本机测试.....177
8.4 使用 NFS.....177

# 第一章 前言

## 1.1 说明

**说明 1：**
主机终端的命令表示方式为：在命令前加“#”符号。
如创建目录：#mkdir -p /loongson/buildfs

开发板的串口终端命令表示方式为：当进入 BIOS（PMON）操作后在命令前加“PMON>”符号；当进入文件系统后在命令前加“$”符号。
如在 BIOS（PMON）查看环境变量：PMON>set
如 PWM 的测试例子程序：$./test-pwm

**说明 2：**
所有需要 su 权限的命令，但开头无“sudo”的命令行的权限均为“su”超级用户。使用“sudo su”
命令可以转变为超级用户权限。

**说明 3：**
文档“注意”部分的字体用红色表示，“提示”部分的字体用蓝色表示，“注释”部分的字体用橙色表示。

**说明 4：**
需要使用源码包的步骤，在开始就注明了源码包位置。
比如：源码包位置：Loongson_1B/BSP/Pmon/1b-pmon.tar.gz

## 1.2 名词解释

| 名词 | 含义 |
| --- | --- |
| PMON | 一个兼有 BIOS 和 boot loader 部分功能的开放源码软件，引导系统启动 |
| | |
| | |
| | |

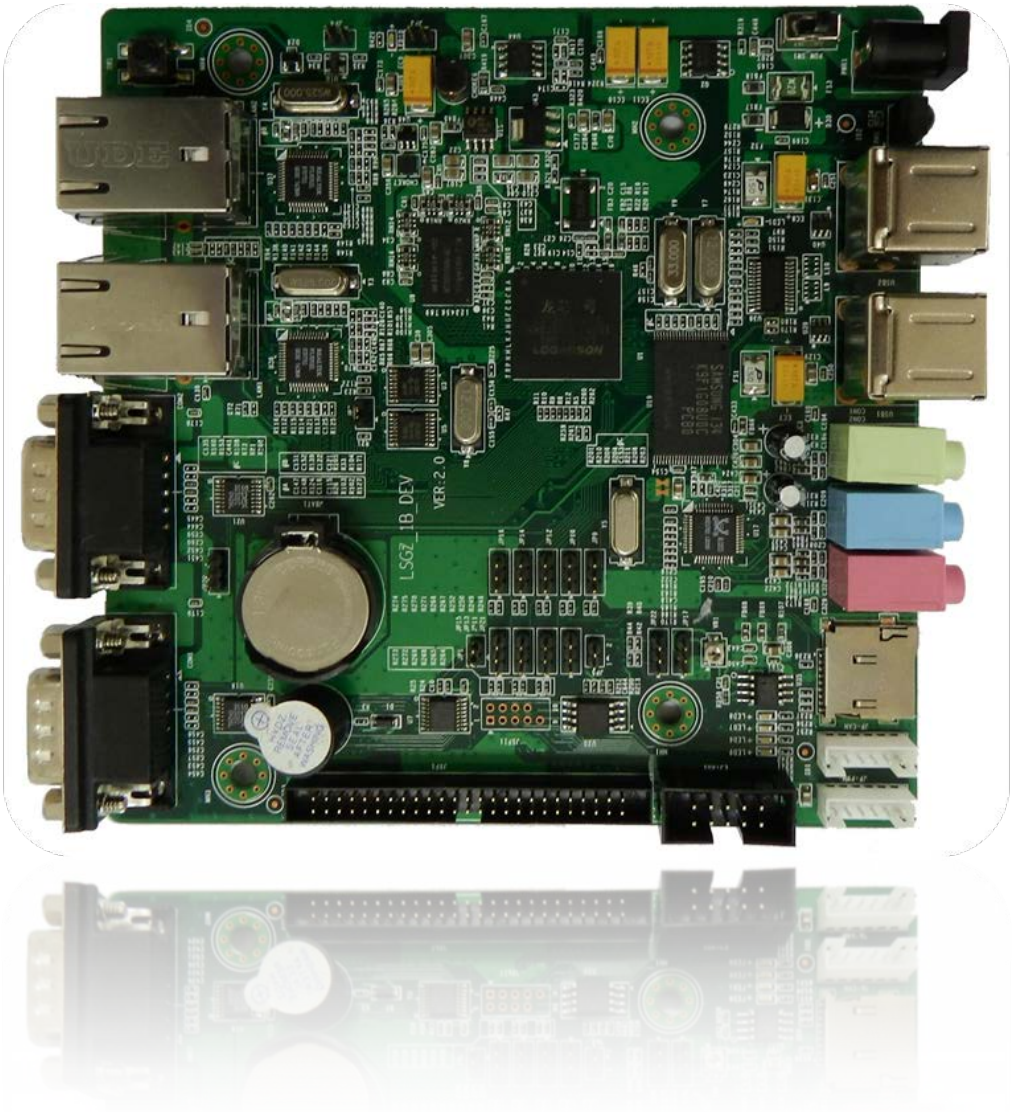
# 第二章 1B 开发板介绍

## 2.1 开发板简介

龙芯 1B 开发板采用的是基于 MIPS 精简指令集的国产龙芯 1 号系列的主控芯片。从芯片设计到板级设计，都尽量实现国产最大化，是一款应用国产技术较多，原生中文技术支持较好的开发板。

开发板主要由广州龙芯中科电子科技有限公司研发，采用 4 层 PCB 板，贴片零件全部由专业贴片机完成，不仅保证了信号的质量，同样也保证了元件的稳定可靠。在设计上，工程师尽量把芯片的各项功能通过复用或直联的方式显示出来，方便客户设计验证。

## 2.2 开发板外观



## 2.3 开发板硬件资源

| 硬件资源 | |
|---|---|
| 名称 | 描述 |
| 处理器 | 龙芯 1B，主频 200-233MHz，可配置到 266MHz |
| 存储器 | SPI FLASH，512K-byte，W25X40BVSSIG<br>SLC nand flash 一片，128MB，K9F1G08U0C-PCB0<br>64MB DDRII SDRAM，K4T51163QI-HCF7 |
| 调试接口 | 10 针标准 EJTAG 接口 |
| I/O 接口 | 音频接口，立体声音频 LINE_OUT/LINE_IN/MIC_IN 接口（ALC655）<br>5 个串口，一个 4 线串行接口，四个 2 线串行接口，波特率高达 115200bps<br>10M/100M 自适应网口两个（RTL8201EL-GR，带发送和接收指示灯）<br>内部实时时钟（带后备纽扣电池）<br>USB 2.0 HOST 接口 x4<br>一个红外线数据接收头<br>Micro SD 卡接口一个<br>4 路 PWM 接口<br>2 路标准 CAN 接口 |
| 显示 | 40 针 LCD 接口引出了 LCD 控制器和触摸屏的全部信号<br>使用 4.3 寸 LCD，最高分辨率支持 480X272<br>触摸屏使用一片 SPI 转换芯片 XPT2046 |
| 按键 | 一个复位按键<br>子板上设置 16 个小按键，由 3 路 GPIO 扩展出来 |
| 电源 | 直流电源适配器供电（5V 3A），带电源指示灯 |
| 其他 | 四个高亮蓝色 LED（LED6、LED7、LED8、LED9）<br>1 个蜂鸣器<br>一片 12 位 ADC 采集芯片，并外接一个三针可调电阻，方便用户直接测试板上 AD 功能。 |

## 2.4 硬件介绍

### 2.4.1 CPU

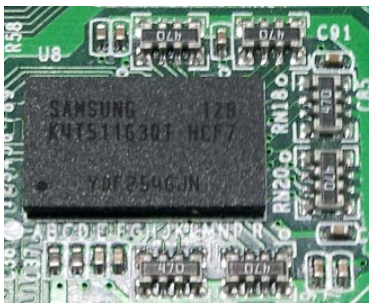
龙芯 1B 一款实现 MIPS32 兼容且支持 EJTAG 调试的双发射处理器，通过采用转移预测、寄存器重命名、乱序发射、路预测的指令 CACHE、非阻塞的数据 CACHE、写合并收集等技术来提高流水线的效率。

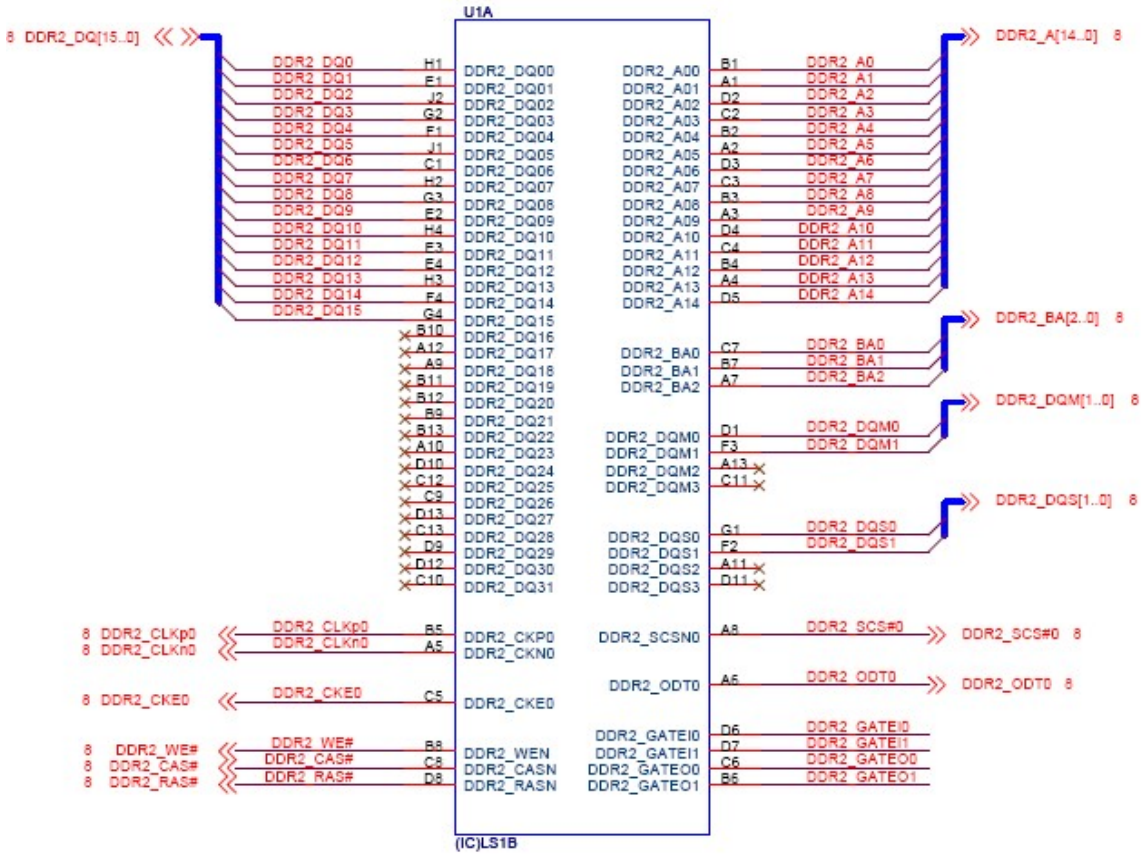
1B 芯片具有以下关键特性：

- 集成一个 GS232 双发射龙芯处理器核，指令和数据 L1 Cache 各 8KB
- 集成一个 24 位 LCD 控制器，最大分辨率可支持到 1920*1080@60Hz/16bit
- 集成 2 个 10M/100M/1000M 自适应 MAC
- 集成 1 个 32 位 133MHz DDR2 控制器，兼容 16 位 DDR2
- 集成 1 个 USB 2.0 接口，兼容 EHCI 和 OHCI
- 集成 1 个 8 位 NAND FLASH 控制器,最大支持 32GB
- 集成中断控制器，支持灵活的中断设置
- 集成 2 个 SPI 控制器，支持主从模式，支持系统启动
- 集成 AC97 控制器
- 集成 1 个全功能串口、1 个四线串口和 10 个两线串口
- 集成 3 路 I2C 控制器，兼容 SMBUS
- 集成 2 个 CAN 总线控制器
- 集成 62 个 GPIO 端口
- 集成 1 个 RTC 接口
- 集成 4 个 PWM 控制器
- 集成看门狗电路


## 2.4.2 DDR2

- 16 位 DDR2 SDRAM 存储器
- 采用一片三星的 K4T51163QI 芯片，容量为 64MB，333MHz fCK for 667Mb/sec/pin, 400MHz fCK for 800Mb/sec/pin。
- BGA 的封装，不易脱焊，长期使用更可靠
- 部分信号线采用蛇形走线，为了兼顾等长布线而设，目的使信号更稳定
- 遵守 DDR2 的行业标准（JESD79-2B）
- 接口上命令、读写数据全流水操作
- 内存命令合并、排序提高整体带宽
- 配置寄存器读写端口，可以修改内存设备的基本参数
- 内建动态延迟补偿电路（DCC），用于数据的可靠发送和接收
- 支持 33-177MHZ 工作频率





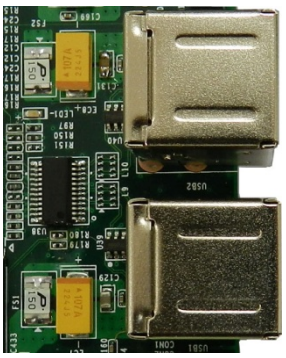

## 2.4.3 Nand Flash


- 采用三星的 K9F1G08UOC-PCB0 存储芯片，容量达 128M
- 支持最大单颗 NAND FLASH 为 32Gbit
- 共 4 个片选 CS
- 数据宽度 8bit
- 支持 SLC

## 2.4.4 LCD

- 屏幕分辨率可达 1920*1080
- 硬件光标
- 伽玛校正
- 最高像素时钟 172MHz
- 支持线性显示缓冲
- 上电序列控制
- 支持 16 位/24 位 LCD，高达 1600 万色

## 2.4.5 USB2.0

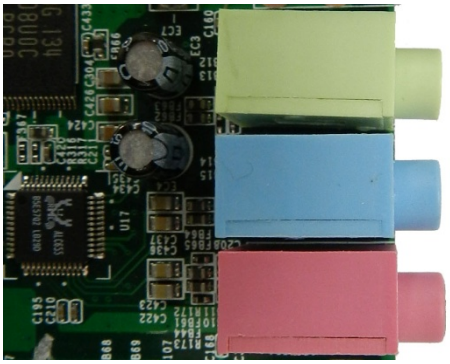


- 共 4 个，通过 USB HUB（MW7220）扩展
- 1 个独立的 USB2.0 的 HOST ports 及 PHY
- 兼容 USB1.1 和 USB2.0
- 内部 EHCI 控制和实现高速传输可达 480Mbps
- 内部 OHCI 控制和实现全速和低速传输 12Mbps 和 1.5Mbps

## 2.4.6 AC97

AC97 音频接口（ALC655）

- 具有 Line-out、Line-in、Mic-in 三种接口
- 支持 16，18 和 20 位采样精度，最高速率可达 48KHz（开发板使用 ALC655，固定采样速率为 48KHz）
- 支持麦克风输入
- 三种接口分不同颜色，方便用户使用；其中绿色是 LINE OUT、蓝色是 Line-in、粉红色是 Mic-in。



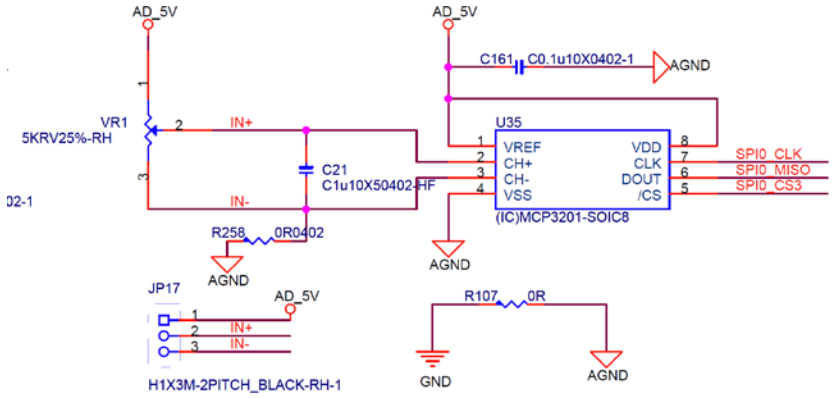
## 2.4.7 MAC

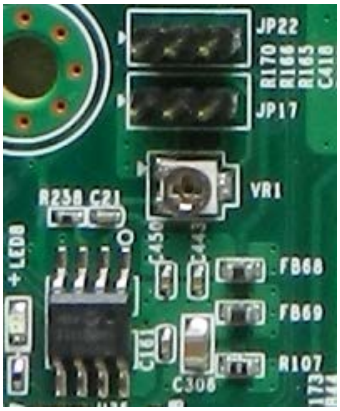
- 两路 10/100Mbps 自适应以太网控制器
- 双网卡均兼容 IEEE 802.3
- 对外部 PHY 实现 RMII 和 MII 接口
- 半双工/全双工自适应
- 半双工时，支持碰撞检测与重发（CSMA/CD）协议
- 支持 CRC 校验码的自动生成与校验

## 2.4.8 SPI

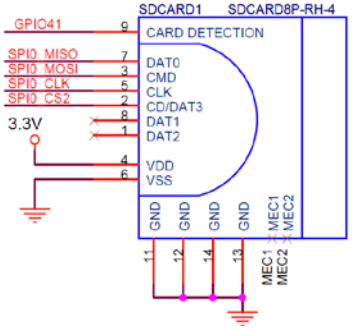
- 支持 2 路 SPI 接口
- 支持系统启动
- 极性和相位可编程的串行时钟
- 可在等待模式下对 SPI 进行控制

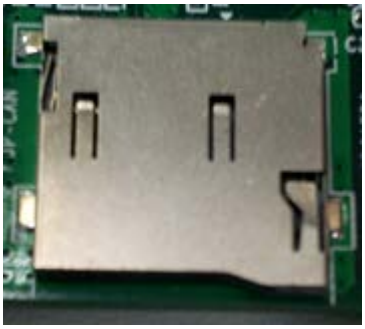
本开发板的 AD 采集及 SD 卡均采用 SPI 控制方式。AD 采集芯片则采用 MCP3201，12 位串行 AD。如图，JP4 的 1 脚为信号输入，2 脚为模拟地。




SD 卡采用 SPI 接口与 AD 采集共用一个 SPI 口，采用分时复用的模式。采用标准的 Micro SD 卡插槽。

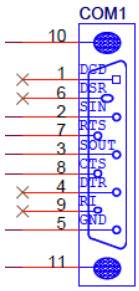
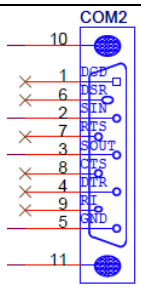


## 2.4.9 UART

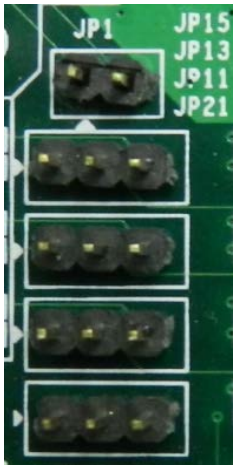
- 集成 1 个四线串口、1 个四线串口和 4 个两线串口
- 在寄存器与功能上兼容 NS16550A
- 全双工异步数据接收/发送
- 可编程的数据格式
- 16 位可编程时钟计数器
- 支持接收超时检测
- 带仲裁的多中断系统

主板提供的 RS-232 串行通讯接口（COM1，COM2）为 DB9 接口，板上其他串口（JP20，JP21，JP22）为插针式 3 针接口需要使用转换电缆。

| COM1 接口 | Pin | Signal | Pin | Signal |
|---|---|---|---|---|
|  | 1 | NC | 7 | RTS |
| | 2 | UART0_RX | 8 | CTS |
| | 3 | UART0_TX | 9 | NC |
| | 4 | NC | 10 | UGND |
| | 5 | GND | 11 | UGND |
| | 6 | NC | | |
| **COM2 接口** | **Pin** | **Signal** | **Pin** | **Signal** |
|  | 1 | NC | 7 | NC |
| | 2 | UART2_RX | 8 | NC |
| | 3 | UART2_TX | 9 | NC |
| | 4 | NC | 10 | UGND |
| | 5 | GND | 11 | UGND |
| | 6 | NC | | |

JP21，JP22，JP23 管脚定义：

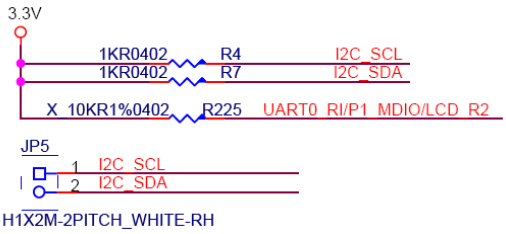
| 接口 | PIN1 | PIN2 | PIN3 |
|---|---|---|---|
| JP20 | UART3_TX | GND | UART3_RX |
| JP21 | UART4_TX | GND | UART4_RX |
| JP22 | UART5_TX | GND | UART5_RX |



## 2.4.10 I2C

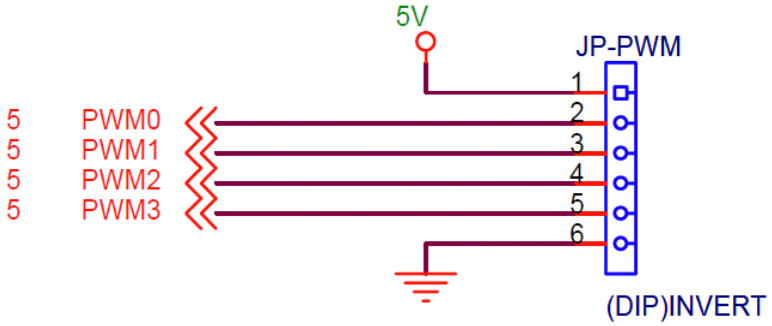
- 兼容 SMBUS（100Kbps）
- 与 PHILIPS I2C 标准相兼容
- 履行双向同步串行协议
- 只实现主设备操作
- 能够支持多主设备的总线
- 总线的时钟频率可编程
- 可以产生开始/停止/应答等操作
- 能够对总线的状态进行探测
- 支持低速和快速模式
- 支持 7 位寻址和 10 位寻址
- 支持时钟延伸和等待状态


## 2.4.11 PWM

- 4 路独立 PWM
- 提供 4 路可配置 PWM 输出,
- 数据宽度 24 位
- 定时器功能
- 计数器功能

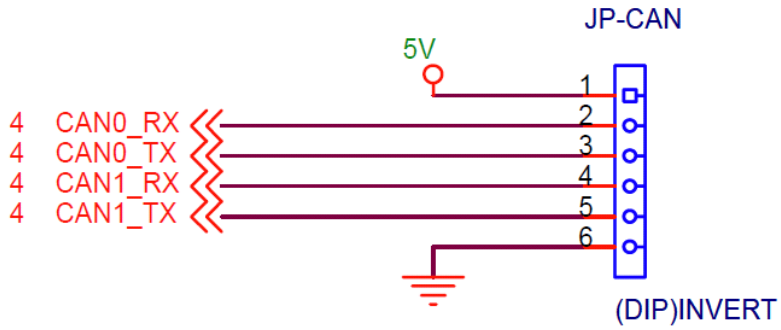
如下图，JP-PWM 接口,在两端加上电源及地，方便调试及应用




## 2.4.12 CAN 总线

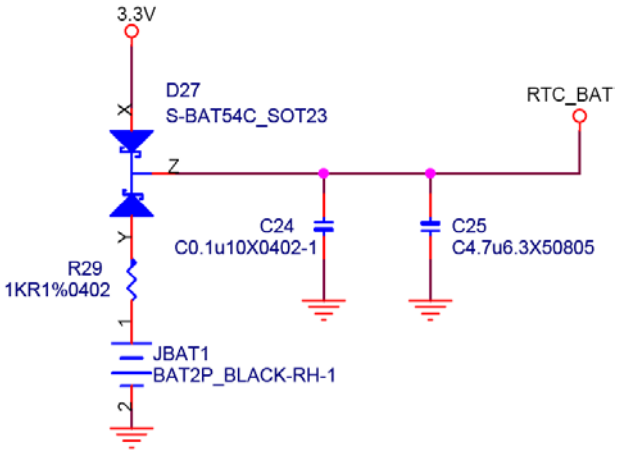
- 支持 2 个独立 CAN 总线接口
- 每路 CAN 接口均支持 CAN2.0A/B 协议
- 支持 CAN 协议扩展

如下图，JP-CAN 接口,在两端加上电源及地，方便调试及应用



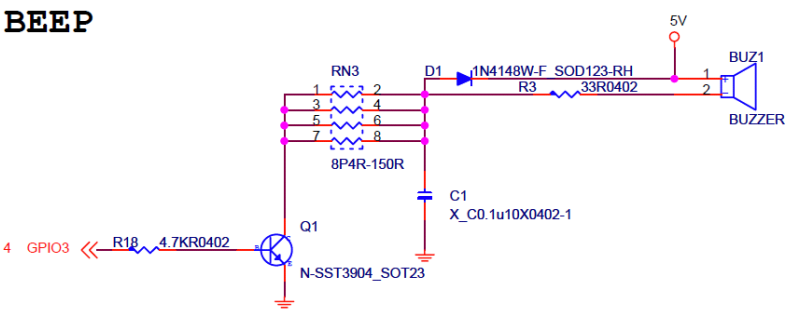

## 2.4.13 实时时钟 RTC

- 硬件支持上电时采用系统电源，断电时使用备份电源
- 计时精确到 0.1 秒
- 可产生 3 个计时中断
- 支持定时开关机功能


## 2.4.14 蜂鸣器

具有完善的保护设计，有效防止过流及电流倒灌

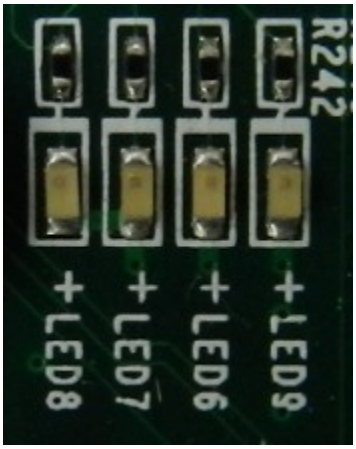



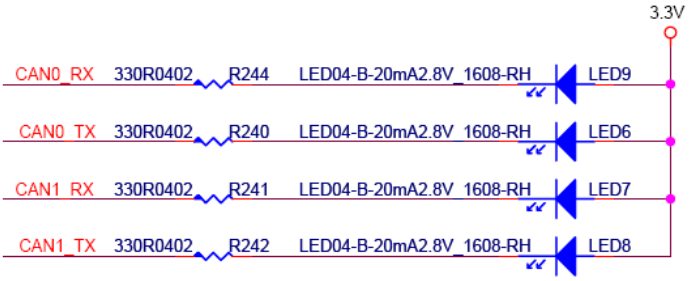

## 2.4.15 按键

16 个按键通过串行转并行来实现，节省 IO 口

### 2.4.16 LED

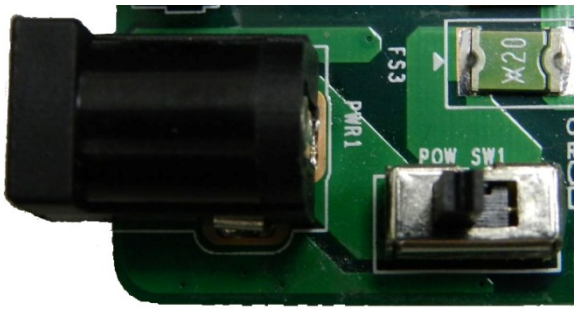
四路独立的 LED 灯

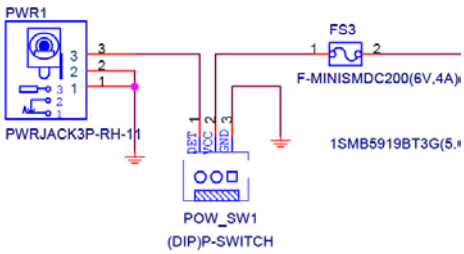


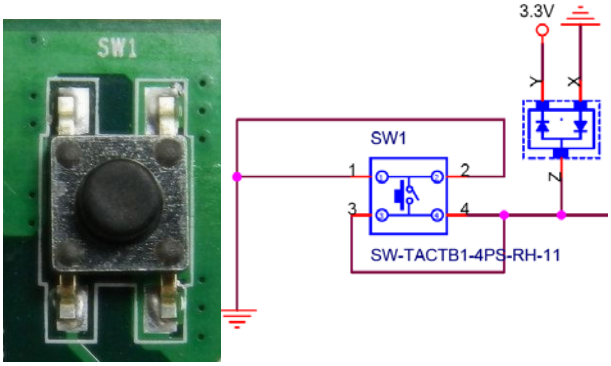

## 2.5 开发板硬件应用说明

（1）电源接口及电源开关（5V 直流输入，开关向前为 ON，反之为 OFF）

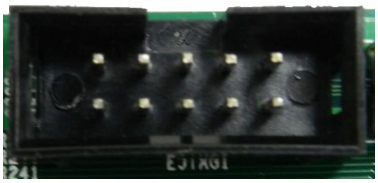


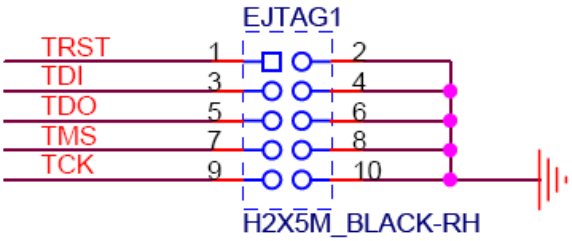




（2）系统复位按键，按下即可以实现系统复位

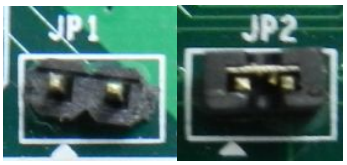



（3）E-JTAG 接口，接上 E-JTAG 则可进行在线调试






（4）复用跳线




| FUNCTION | JP1 | JP2 |
|---|---|---|
| UART0 | 0 | 0 |
| LCD | 1 | 0 |
| MAC1 | X | 1 |

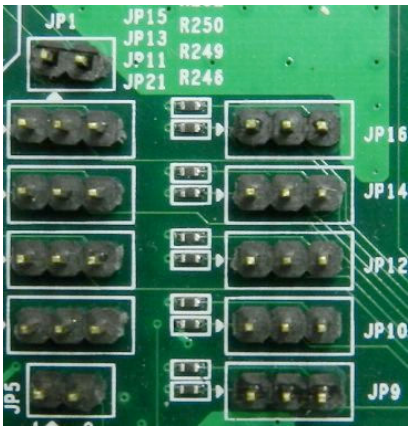
说明：

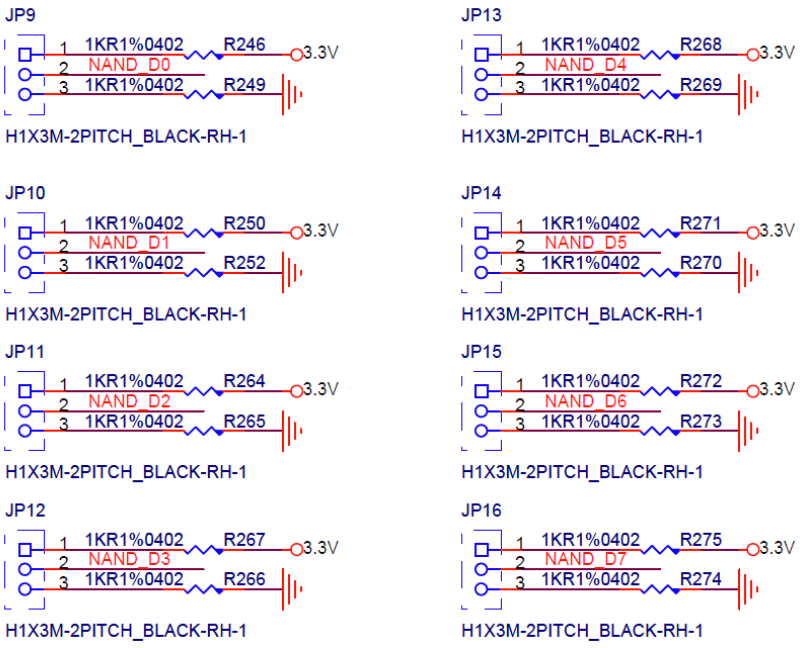
1） 如果 JP1 JP2 接口不接跳线帽，则默认电路为 UART0 工作；

2） 如果 JP1 接跳线帽上拉，JP2 不接，则电路为 LCD 工作；

3） 如果 JP2 接跳线帽上拉，则网口 1 工作。

（5）系统主频设置




说明：

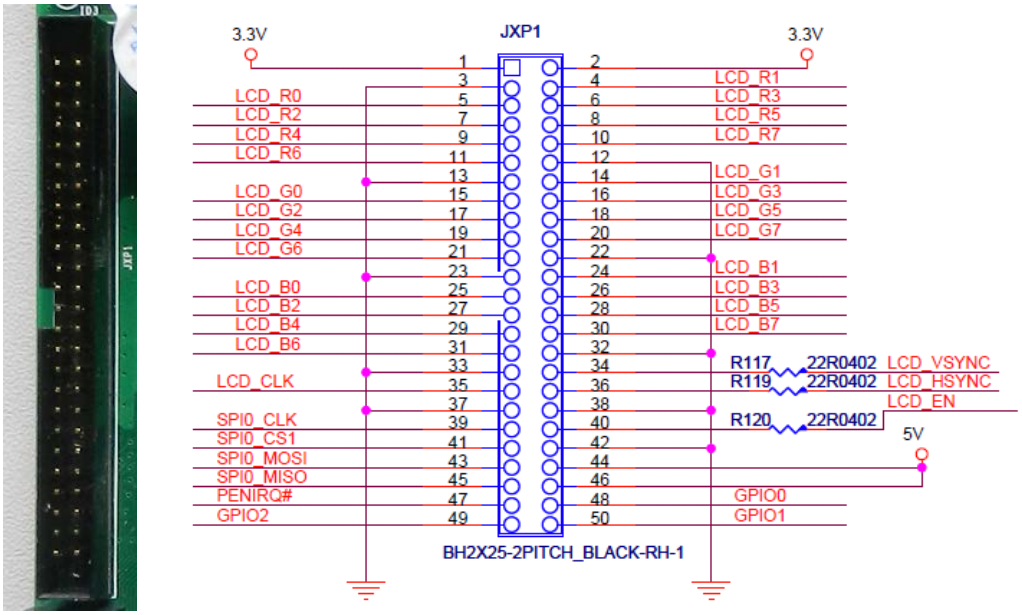
NAND_D6(JP15)设置为高电平时 CPU_clk 和 DDR2_clk 的频率为 33Mhz，这时 JP9-JP14 不起作用。

NAND_D6(JP15)设置为低电平时 CPU_clk 和 DDR2_clk 的频率可以通过 JP9-JP14 进行设置。
频率设置公式为：CPU_clk = DDR2_clk = (NAND_D[5:0]+12)*33/4MHz
如：JP9-J14 都设置为低电平，即 NAND_D[5:0]=0，则 CPU_clk = DDR2_clk = 99MHz

| NAND_D[5:0] | 倍频值 | CPU 和 DDR2 频率 |
|---|---|---|
| 000000 | 0 | 99MHz |
| 000001 | 1 | 107MHz |
| 000010 | 2 | 115MHz |
| 000011 | 3 | 123MHz |
| 000100 | 4 | …… |
| 000101 | 5 | …… |
| 000110 | 6 | …… |
| 000111 | 7 | …… |
| 001000 | 8 | …… |
| 001001 | 9 | …… |
| | | |

注意：以上是通过硬件设置 CPU 和 DDR2 频率，系统复位启动过程中，CPU 和 DDR 频率相同。PMON 的启动文件里通过软件对 CPU 和 DDR2 频率进行了重新设置，以上的硬件设置的频率只在 CPU 和 DDR2 频率还没初始化前起作用。

（6）液晶板与主板连接



# 第三章 1B 开发板使用说明

## 3.1 开发板快速上手指南

### 3.1.1 外部接口的连线

- 请使用我们提供的 5V 3A 电源适配器连接到开发板上电源输入插座
- 使用我们提供的直连串口线连接到板上标准 9 针的串口 0 与 PC 机的串口
- 使用我们提供的直连网线连接板上与 PC 机上的网口
- 如果有液晶显示屏，按照正确的方向与板上的 LCD 接口连接好数据线
- 如果有音箱或耳机，把其插头连接到板上的音频输出接口

注意：有的用户使用 USB 转串口线来扩展串口，但是有些 USB 转串口线会出现乱码现象。请使用我们提供或推荐的 USB 转串口线。

### 3.1.2 设置终端仿真程序

为了通过串口连接开发板，需要在 PC 机上使用一个终端仿真程序。

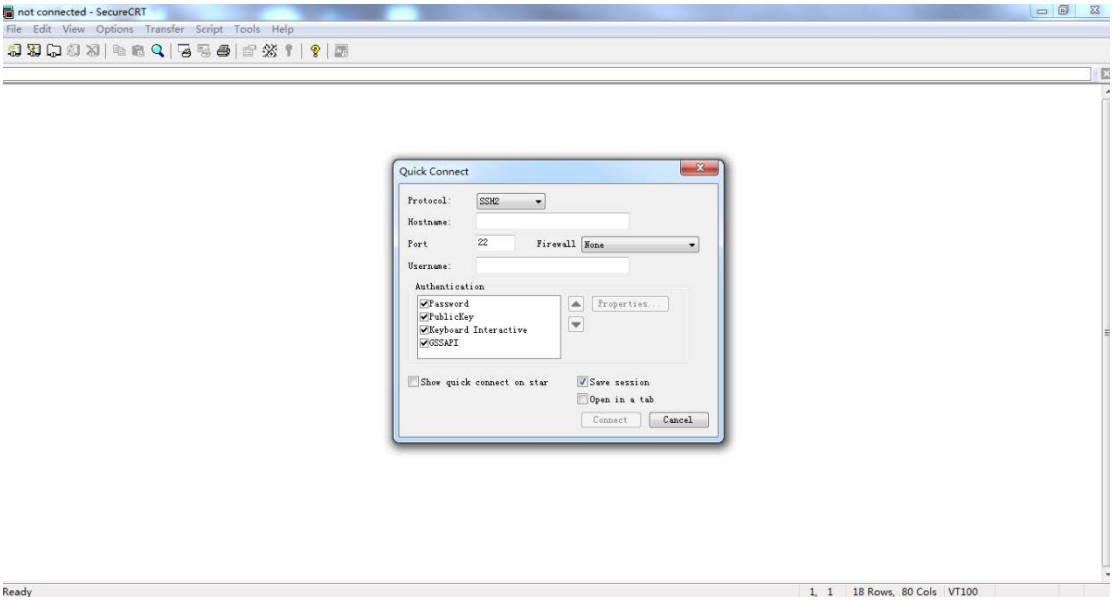
如果 PC 机是 Windows 操作系统，可以使用 SecureCRT 或者超级终端。Windows XP 上自带有超级终端软件，位于“开始-->程序-->附件-->通讯”，而 Windows 7 上需要额外安装，超级终端的设置请参考“附录 3 Windows 超级终端使用说明”。

如果 PC 机是 Linux 操作系统，可以使用 minicom（详细请参考“附录 6 Minicom 使用指南”）。

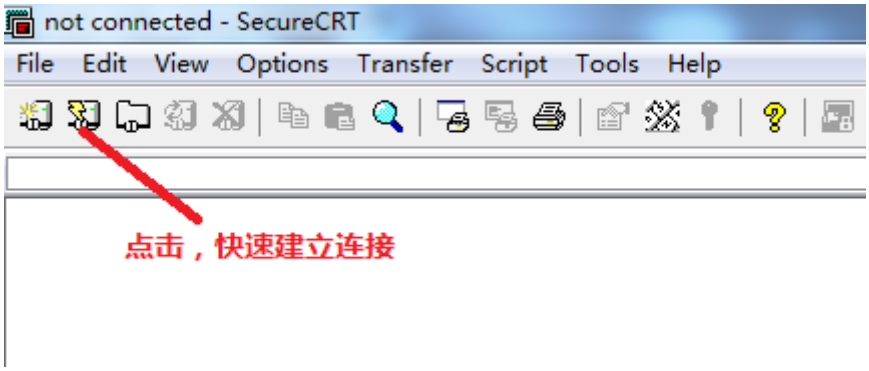
下面以 Windows 操作系统上的 SecureCRT 终端仿真软件为例。

软件工具位置：**Loongson_1B/Tools/windows_tools/SecureCRT**

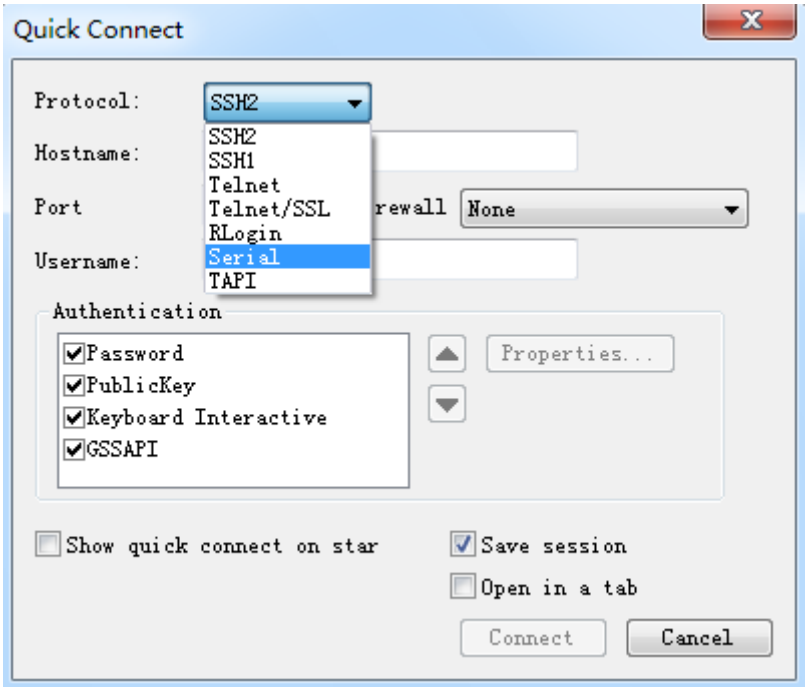
1）双击软件的文件 SecureCRT.exe，打开 SecureCRT，如下图：



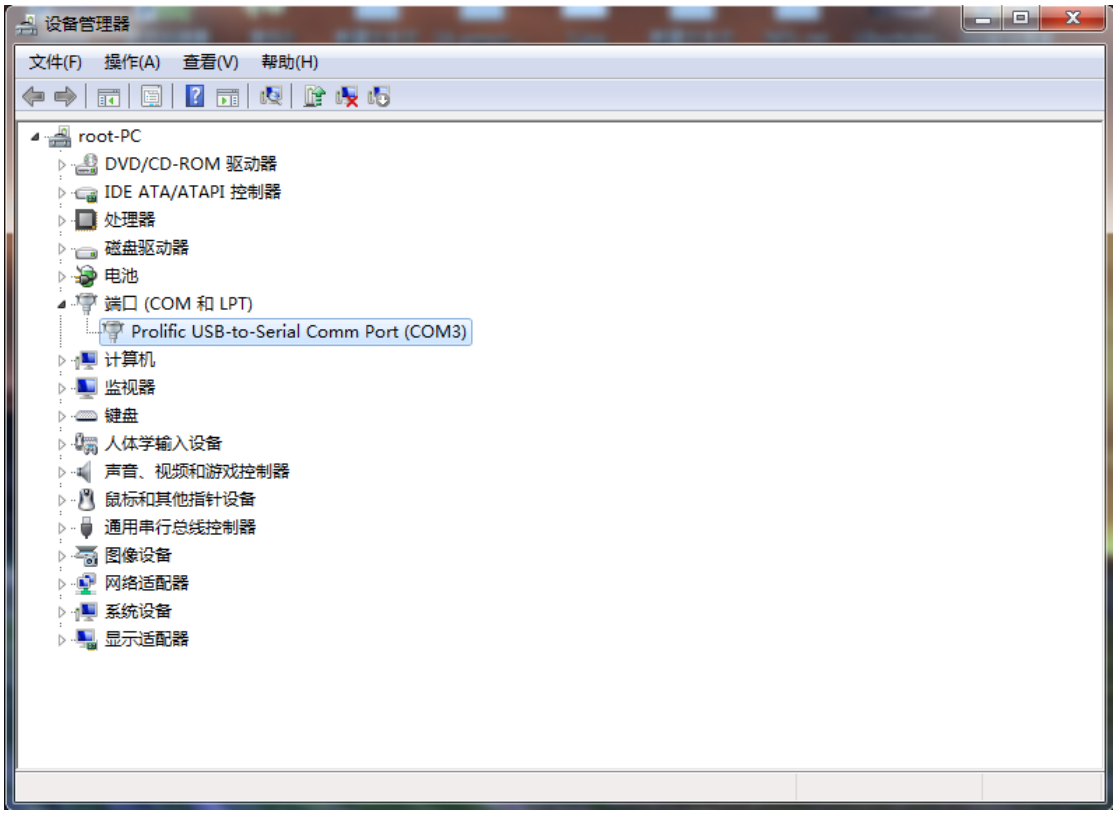

2）点击工具栏的带闪电图标，快速建立连接，如下图：

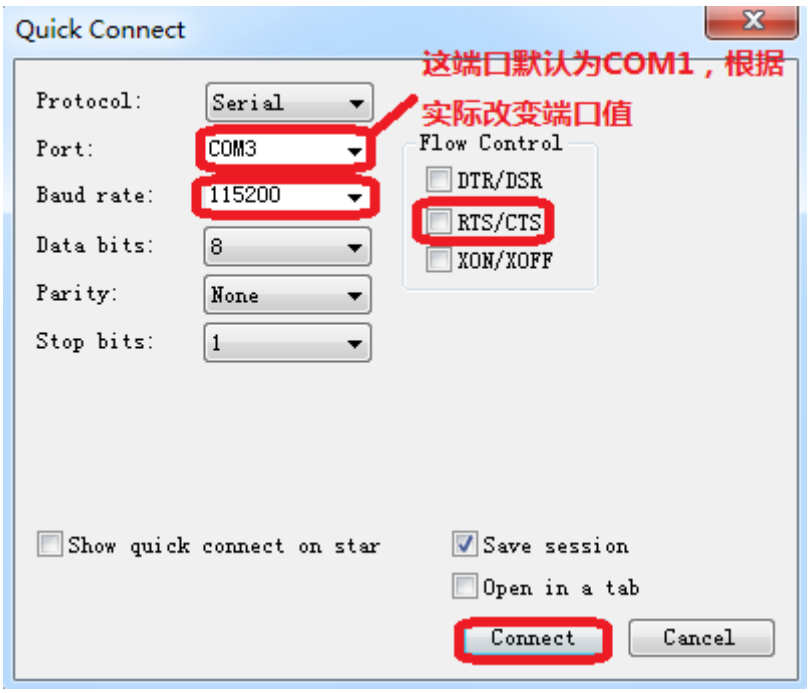



3）在 Quick Connect 对话框中，Protocol 的下拉菜单选择 Serial 协议选项，选择串口协议，如下图：

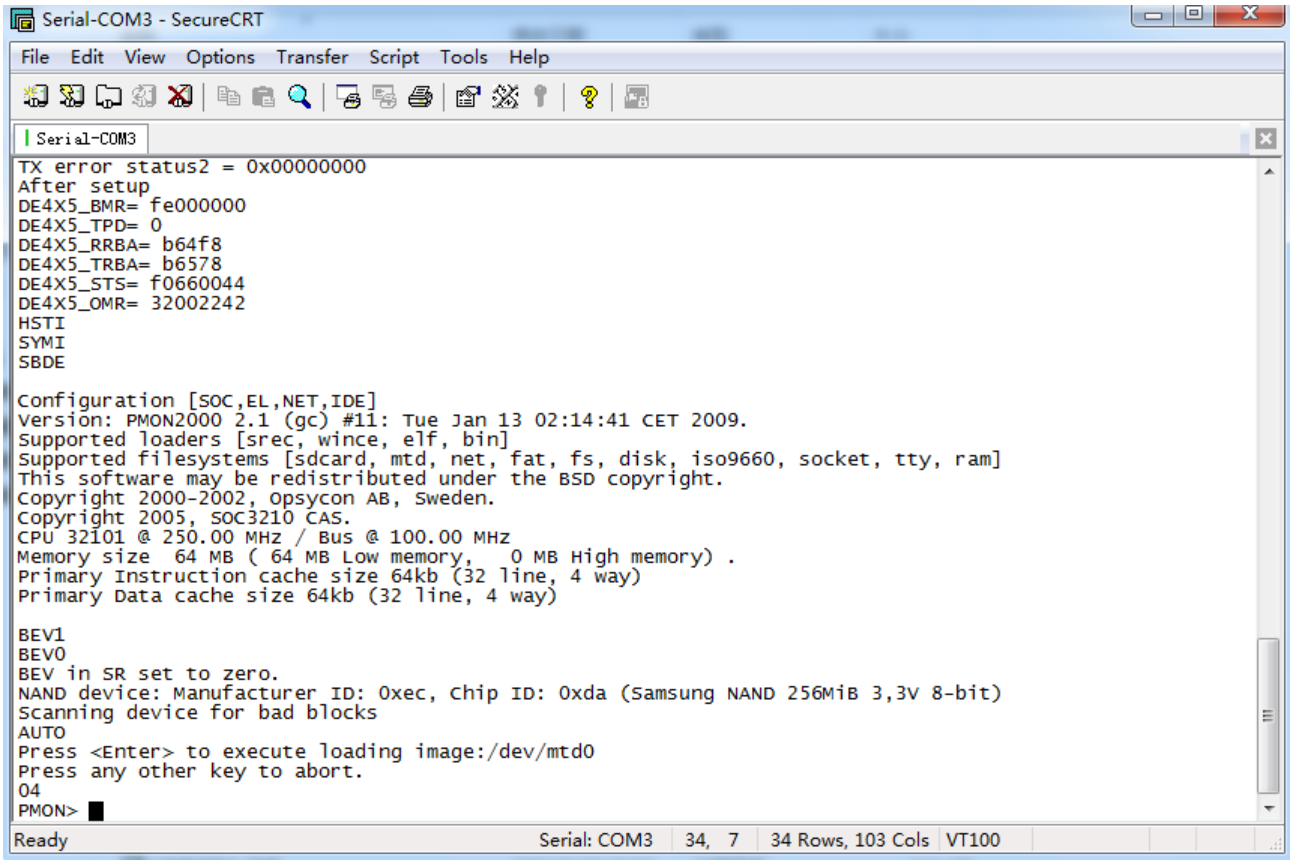



4）在 Quick Connect 对话框中，设置端口 Port 为 COM3，在“设备管理器”中可以看到串口的端口信息（注意：应根据具体信息设置正确的端口）；设置波特率 Baud rate 为 115200；去掉 Flow Control 中所有的勾选；最后 Connect，如下图：






5）开发板上电，按空格键，进入 BIOS（PMON），如下图：



### 3.1.3 恢复与更新 Linux 系统

二进制文件位置：bootloader（pmon）：Loongson_1B/BSP/Pmon/gzrom.bin
Linux 内核文件：Loongson_1B/BSP/Linux_Kernel/vmlinux
根文件系统镜像文件：Loongson_1B/BSP/Filesys/rootfs/root-cram-dyn.img

恢复与更新 Linux 系统时，需要进行以下步骤：

- 安装与设置 TFTP
- 恢复与更新 bootloader（pmon）
- 恢复与更新 Linux 内核
- 恢复与更新根文件系统

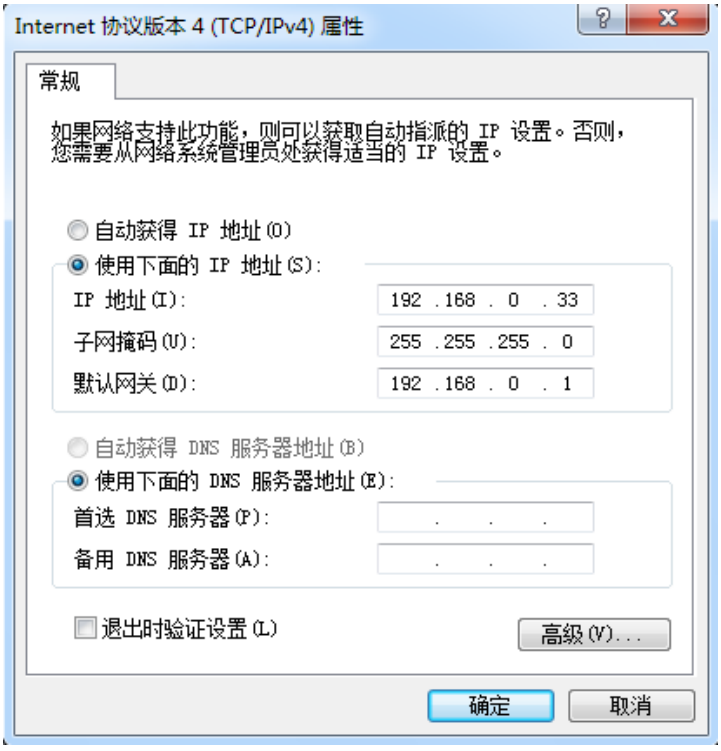
1）安装与设置 TFTP

软件工具位置：Loongson_1B/Tools/windows_tools/Tftp

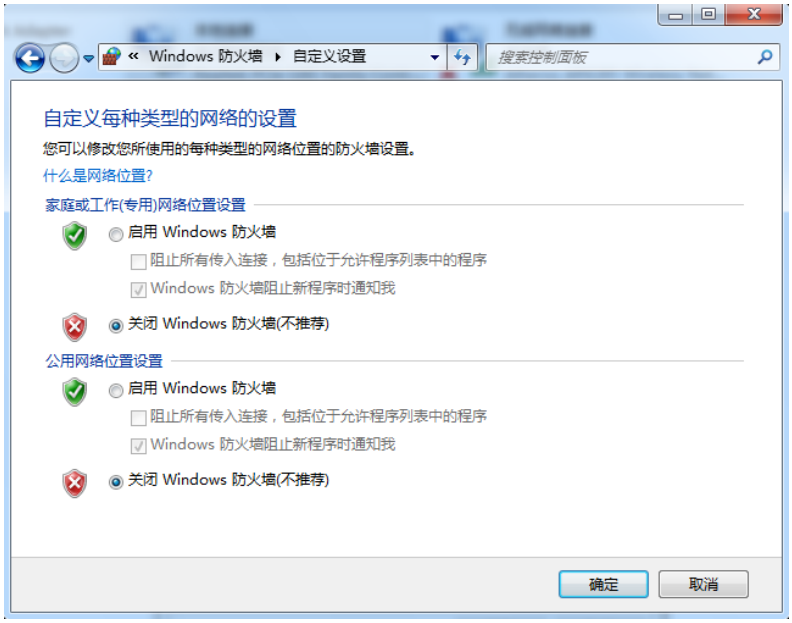
由于 1B 开发板的 Nor Flash 或 Nand Flash 上数据的更新下载通常是通过网络且 TFTP 协议来传输的，所以需要安装 TFTP。因为在开发板的 BIOS 上已经内置了 TFTP 的客户端，所以只需在 PC 机上安装 TFTP 的服务端。

这里安装与设置的 tftp 客户端与服务端是 Windows 上的。如果是 Linux 上的请参考“第四章 4-2-4 安装 TFTP”。

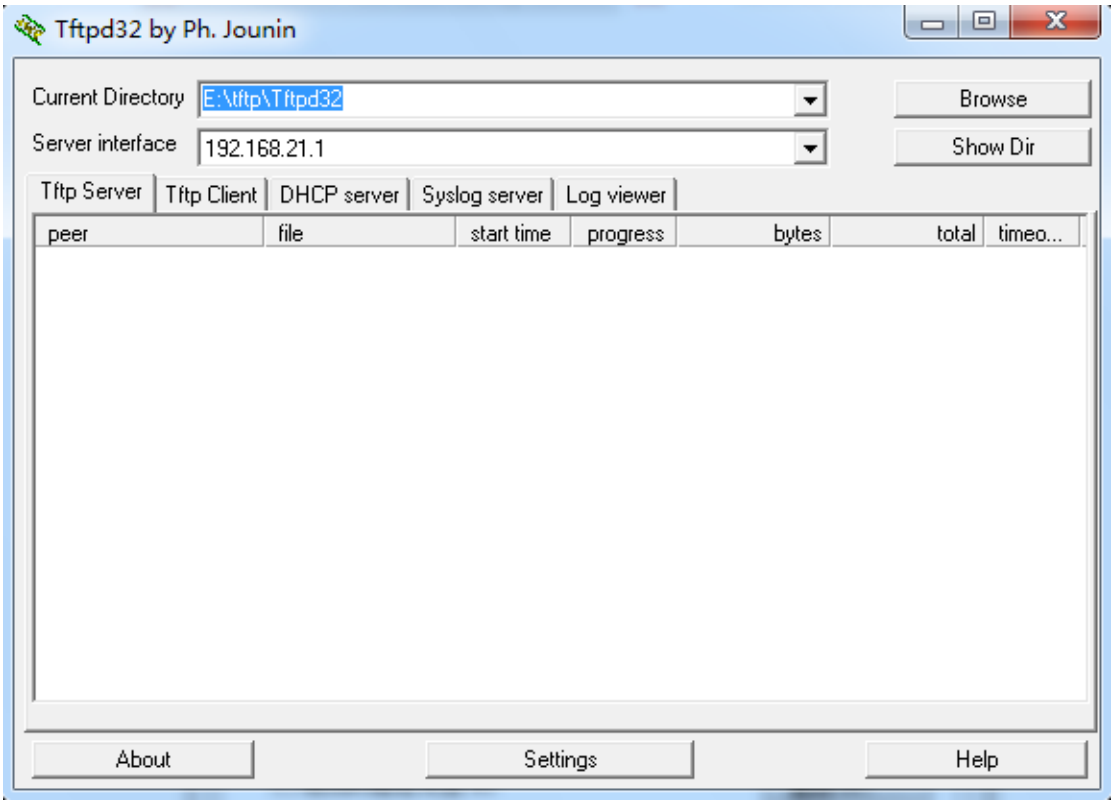
a）设置 PC 机的 IP 地址，比如“192.168.0.33”，如下图：



b）关闭 PC 机的防火墙，如下图：

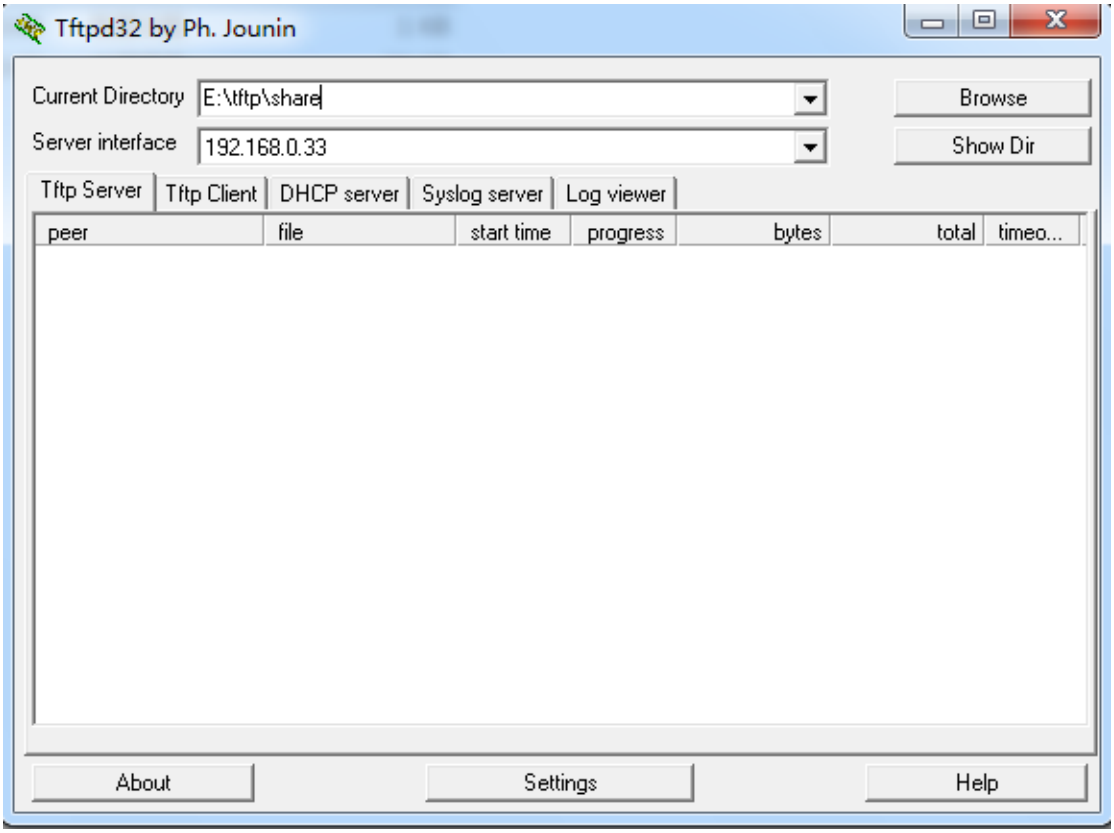



c）双击 Tftp 目录下 tftpd32.exe，打开 tftp，如下图：

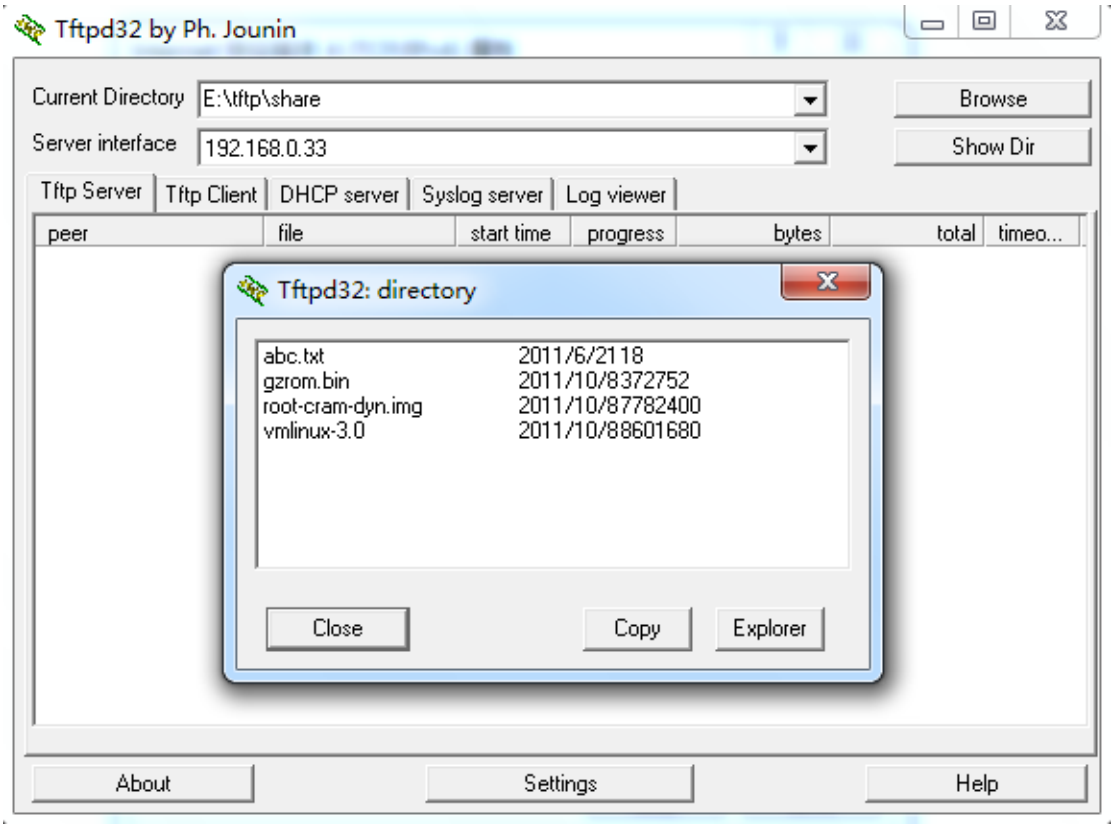



d）设置共享服务目录 Current Directory（如：E:\tftp\share）与服务端的网络接口 Server interface（如：192.168.0.33 即为本 PC 机的 IP 地址），如下图：

注意：设置服务端的网络接口 Server interface，有时需要开发板上电连通。



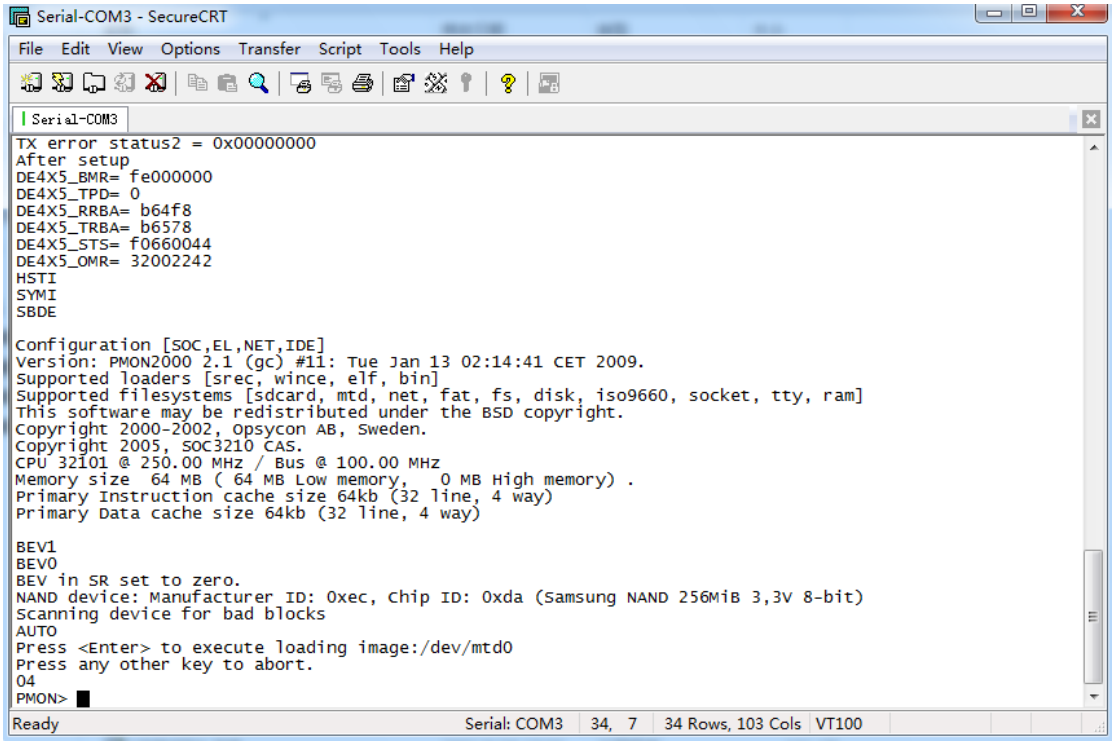

e）把需要的二进制文件 gzrom.bin、vmlinux、root-cram-dyn.img 放置于共享服务目录 E:\tftp\share 下；点击按钮“Show Dir”，可以查看，如下图：



（2）恢复与更新 bootloader（pmon）

如果是开发板上电后，进不了 BIOS（PMON）界面，需要安装完 Linux 操作系统后再参考“附录 7　使用 EJTAG 烧写 BOOTLOADER（PMON）”。

如果开发板上电，按空格键，能够进入 BIOS（PMON），如下图：




提示：需要详细了解 PMON 命令的话，请参考“6-1-4 PMON 的内置命令”。

设置开发板 IP 地址并测试：

**PMON >ifaddr syn0 192.168.0.233　　（起临时作用，断电后无效）**

**PMON >ping 192.168.0.33　　（网络测试通过，才能进行后续的恢复与更新）**

恢复与更新 bootloader（pmon）：

**PMON > load -r -f bfc00000 tftp://192.168.0.33/gzrom.bin**

（3）恢复与更新 Linux 内核

通过 TFTP 下载内核并烧到 Nand Flash：

**PMON>devcp tftp://192.168.0.33/vmlinux /dev/mtd0**

设置启动参数：

**PMON>set al /dev/mtd0**

（4）恢复与更新根文件系统

通过 TFTP 下载根文件系统镜像文件并烧到 Nand Flash：

**PMON>devcp tftp://192.168.0.33/root-cram-dyn.img /dev/mtd1**

设置启动参数：

**PMON>set append “root=/dev/mtdblock1 console=ttyS2,115200 noinitrd init=/linuxrc rootfstype=cramfs”**

# 第四章 在主机上搭建 LINUX 开发环境

在嵌入式开发过程中，通常由于目标板（开发板）没有足够的资源来运行开发和调试工具，所以需要借助建立了交叉编译环境的宿主机（本文采用 PC 机的虚拟机+Linux 操作系统方式）通过使用串口、以太网或其他方式来完成开发和调试。

## 4.1 安装 Ubuntu10.04

1B 开发板的嵌入式操作系统为 Linux，主机上也应为 Linux 操作系统。Ubuntu 是开源、免费的 Linux 操作系统，其中 10.04 版本是最新的长期支持版，操作方便、界面友好。

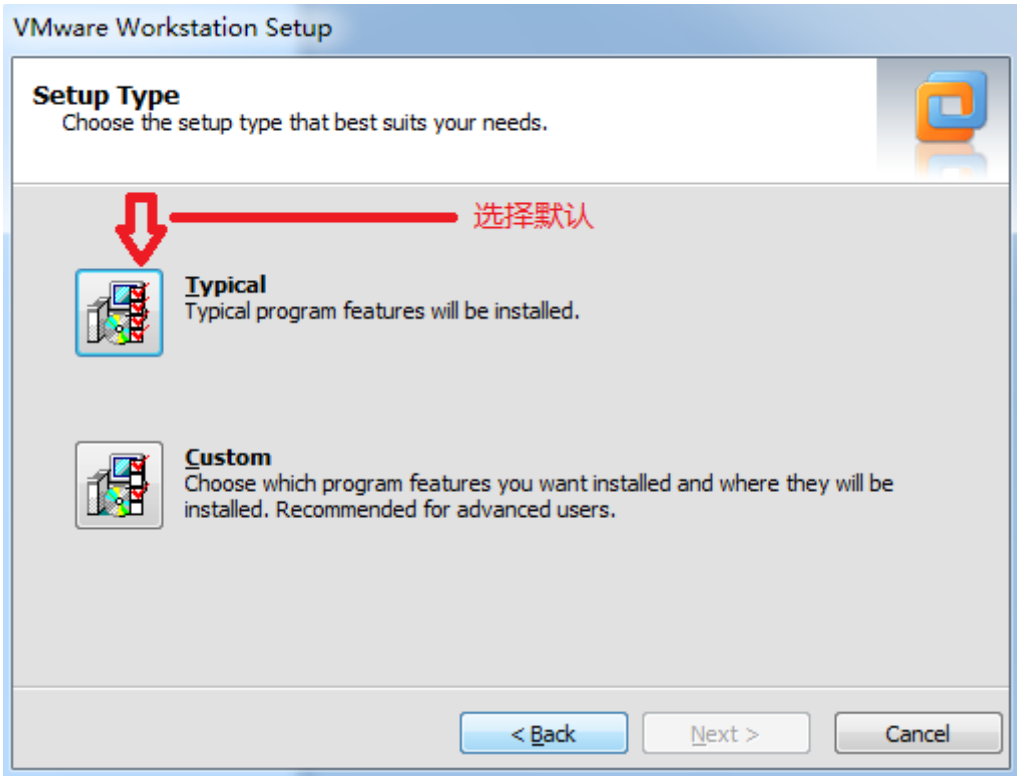
### 4.1.1 安装 VMware-workstation

软件工具位置：**Loongson_1B/Tools/windows_tools/VMware**

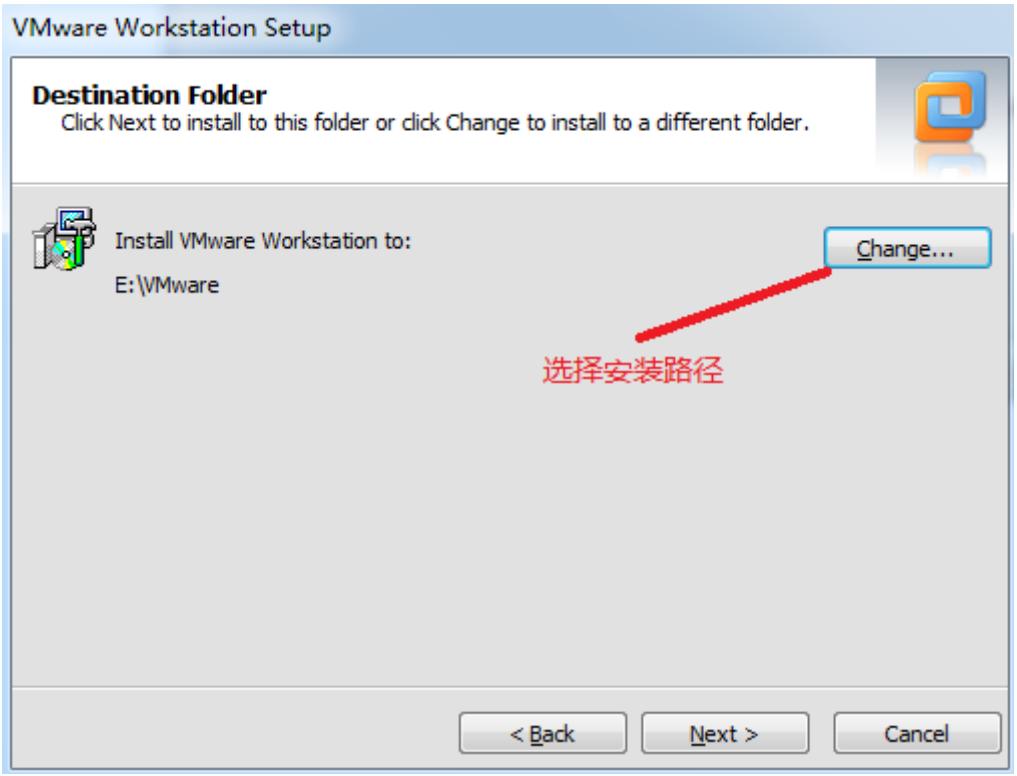
VMware Workstation 虚拟机是一个在 Windows 或 Linux 计算机上运行的应用程序，它可以模拟一个基于 x86 的标准 PC 环境。

安装步骤：

1） 在 Windows 上运行 VMware-workstation-full-7.1.3-324285.exe，选择默认设置，如图：

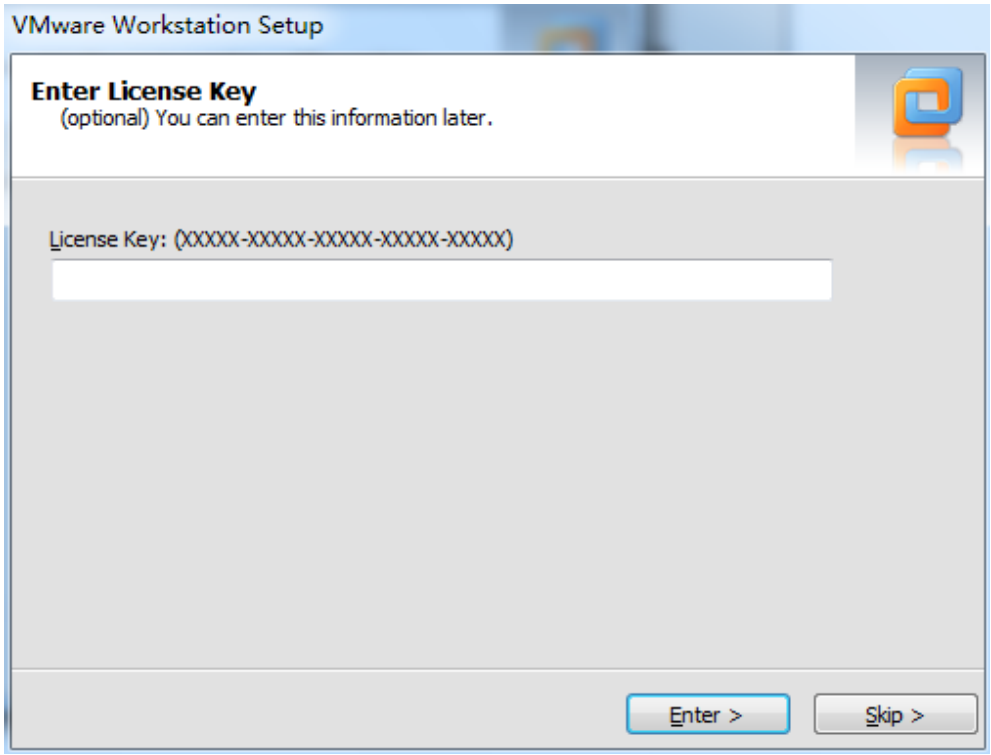


2） 修改安装路径，选择自己的安装路径，如“E:\VMware”，如图：

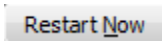



3）往下这几个步骤选择为“Next >”。

4）最终进入注册界面，输入正确注册码，如输入“假设”的注册码“UZ7WU-D3DE7-M81FY-LPP7E-P3H82”，如图：



下两步，点击“Enter”和“Restart Now”，如图：

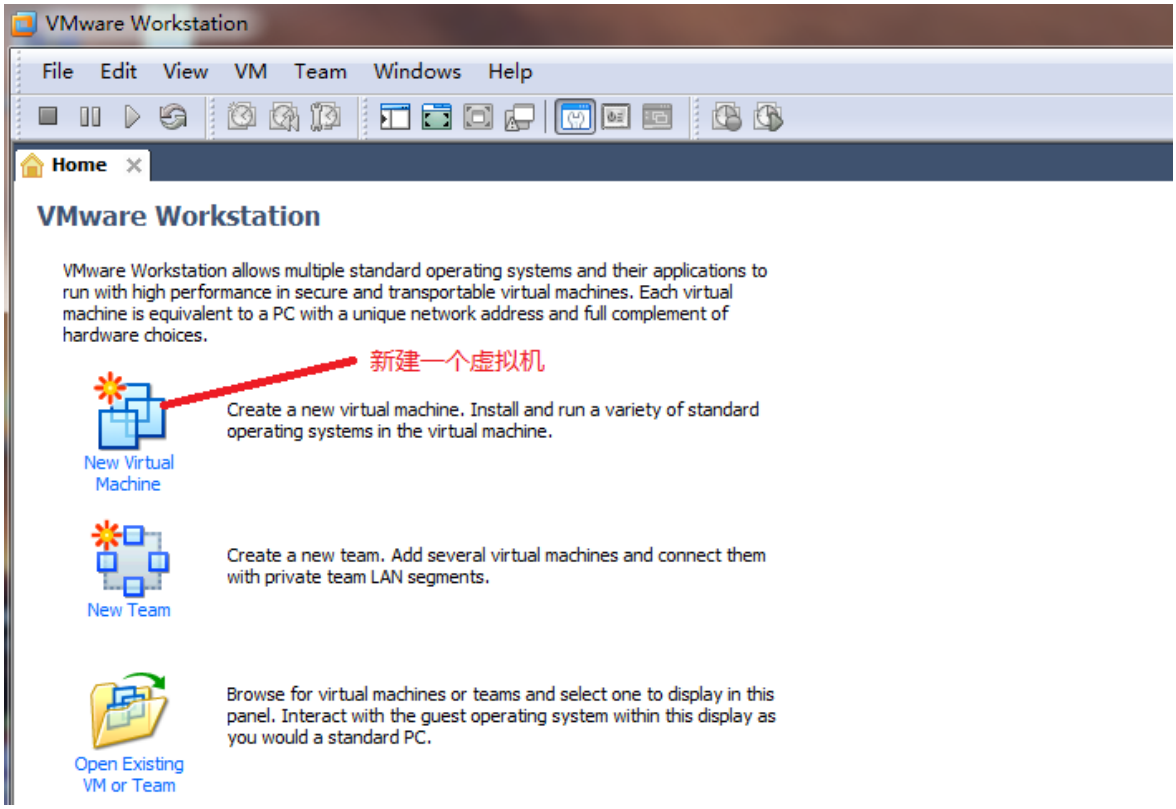




5）重启主机，完成安装。

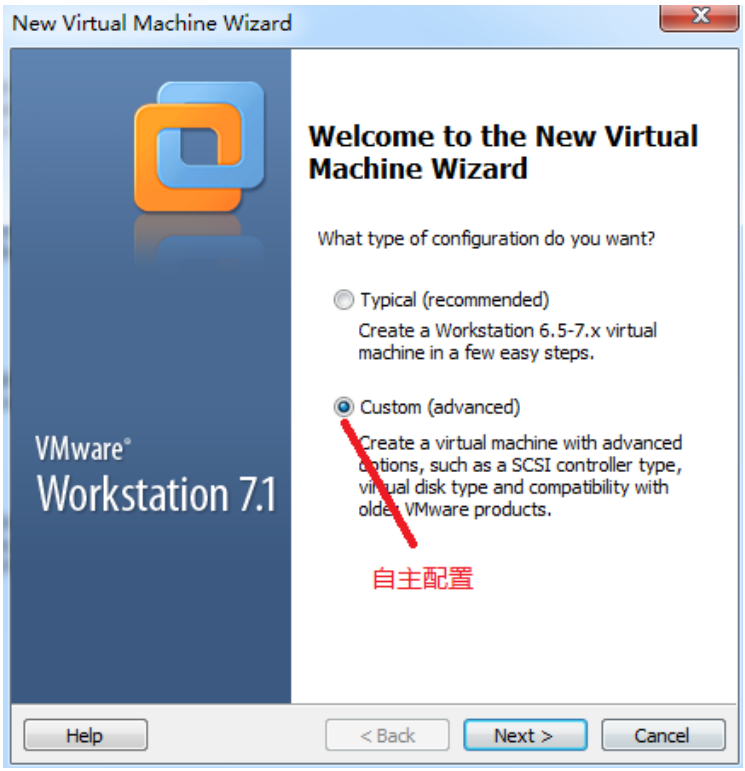
注意：建议不把 VMware-workstation 软件汉化，因为汉化后可能导致某些功能不起作用，比如可能会导致共享不能实现的问题。

### 4.1.2 新建虚拟机

（1）运行 VMware Workstation，新建一个虚拟机，在主界面“Home”上选择“New Virtual Machine”，如图：

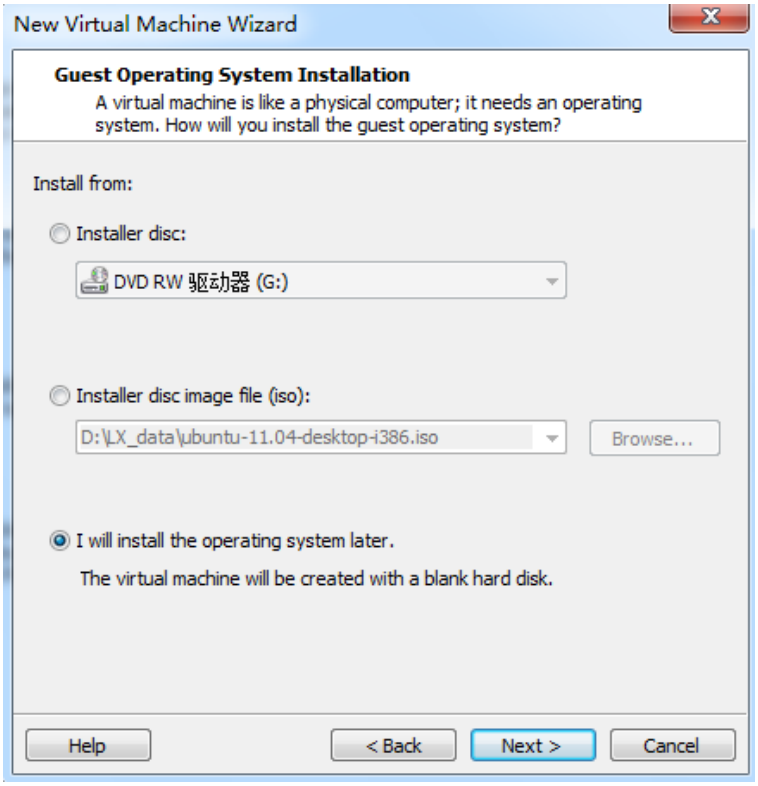



（2） 进到配置界面，选择“Custom”，如图：

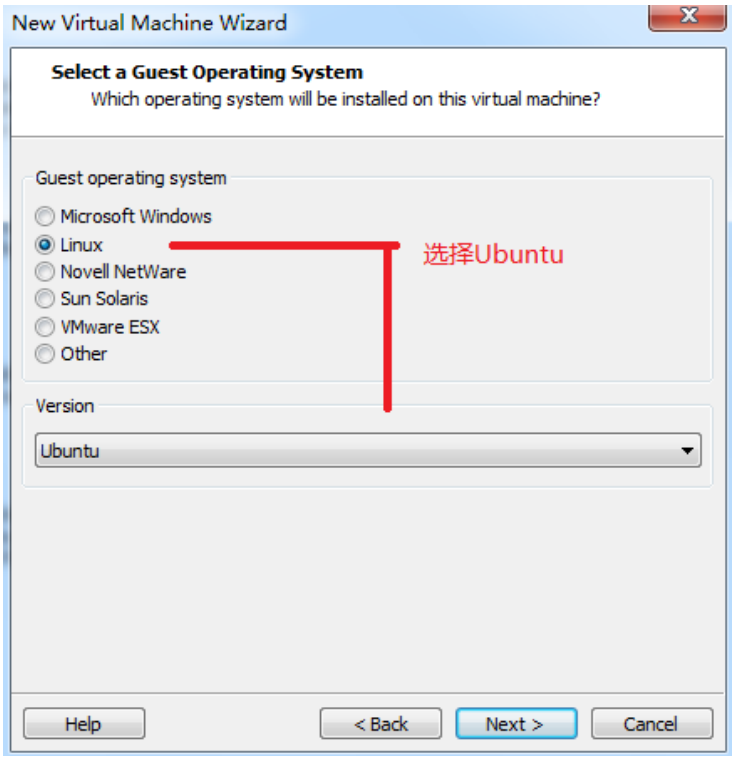



（3）往下这几个步骤选择“Next >”。

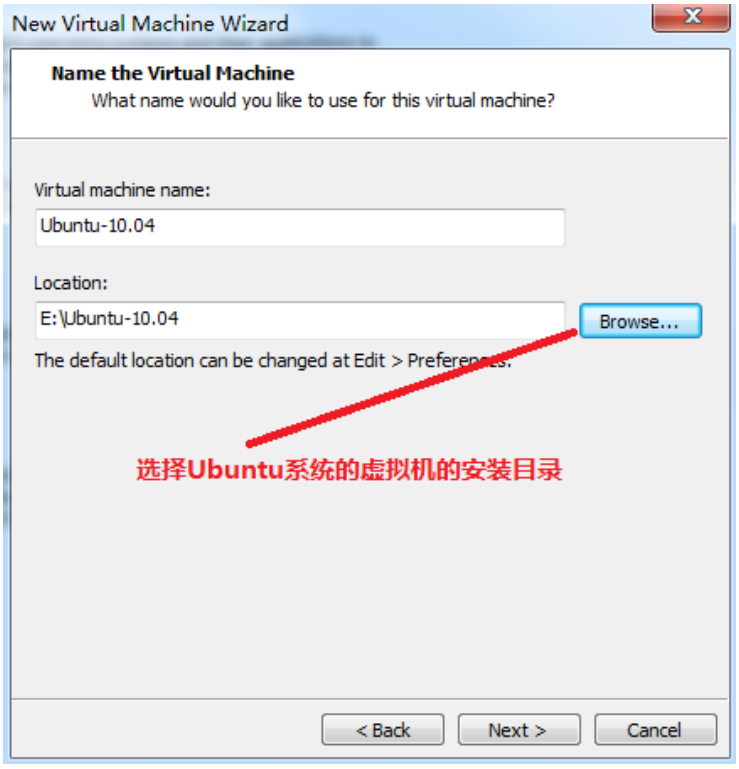
（4）最终进到选择安装系统路径步骤，选择“I will install the operating system later.”，新建完成后再安装系统，如图：



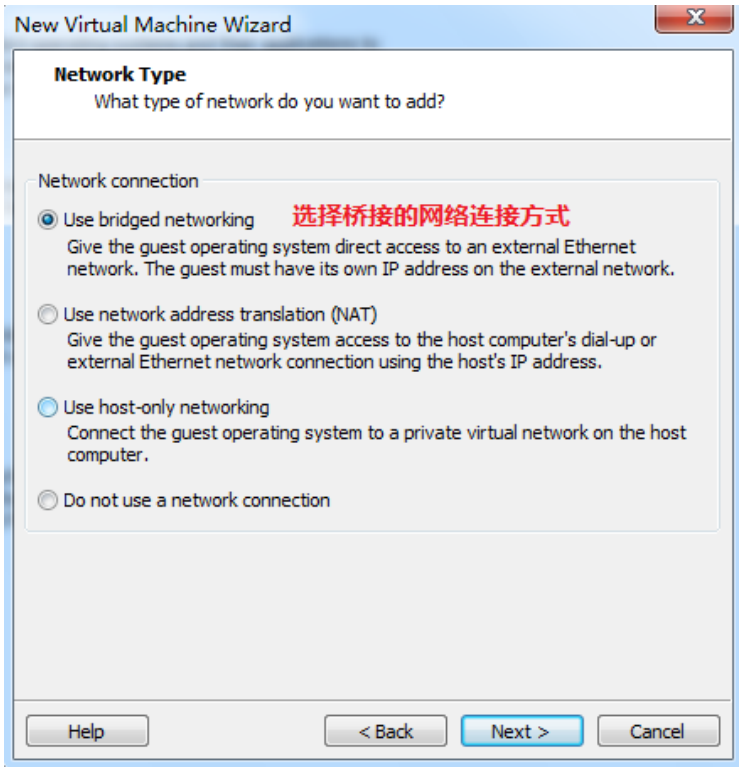
（5）系统选择配置，“Guest operating system”选择“Linux”，“Version”选择“Ubuntu”，如图：



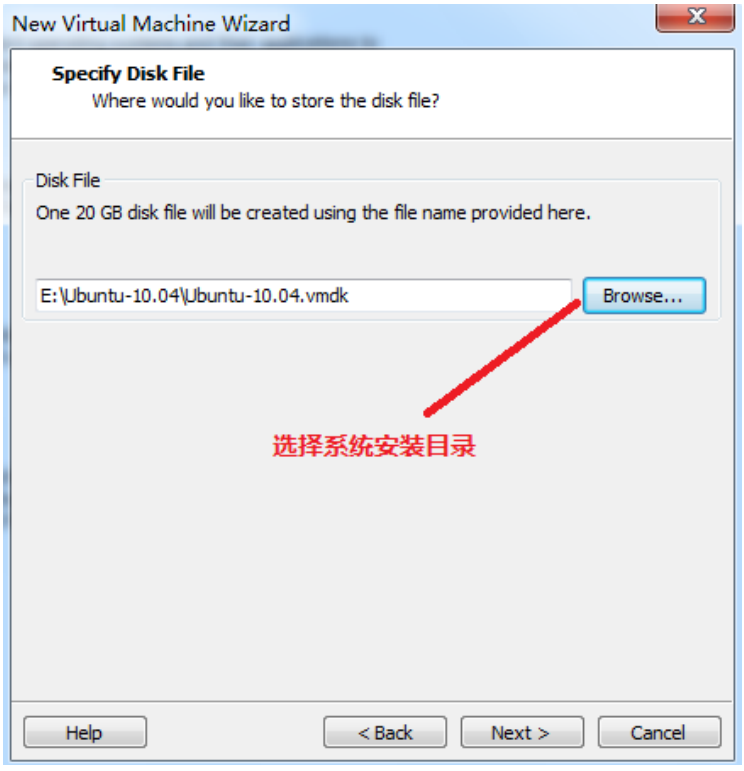

（6）修改虚拟机的名字和安装目录，比如在“Virtual machine name”输入名字“Ubuntu-10.04”，在“Location”输入安装目录“E:\Ubuntu-10.04”，如图：



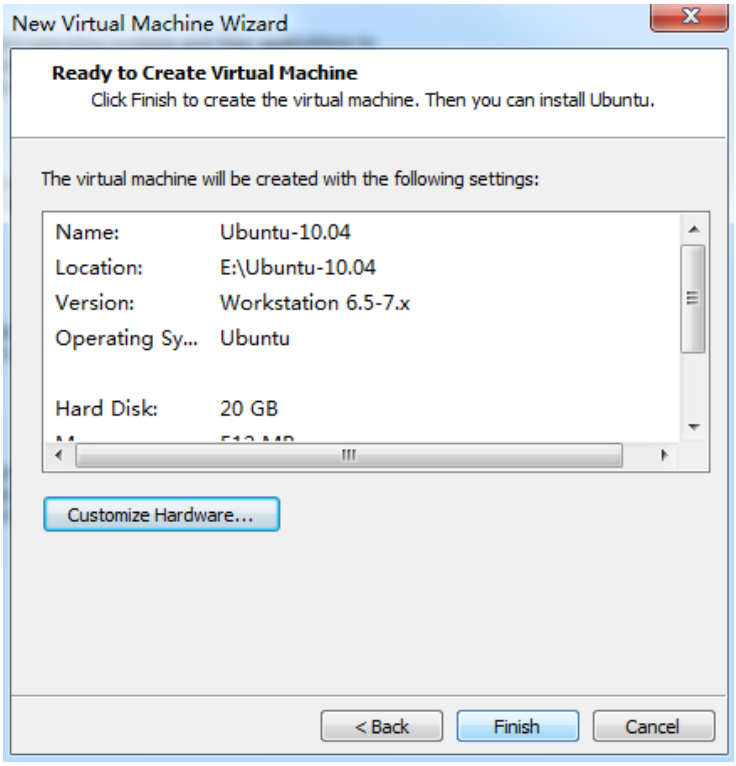
（7） 网络连接方式，选择“Use bridged networking”，如图：




（8）修改系统（此时以.vmdk 格式的文件存在）的安装目录，如输入“E:\Ubuntu-10.04\Ubuntu-10.04.vmdk”，或者选择原来虚拟机的安装目录，如图：



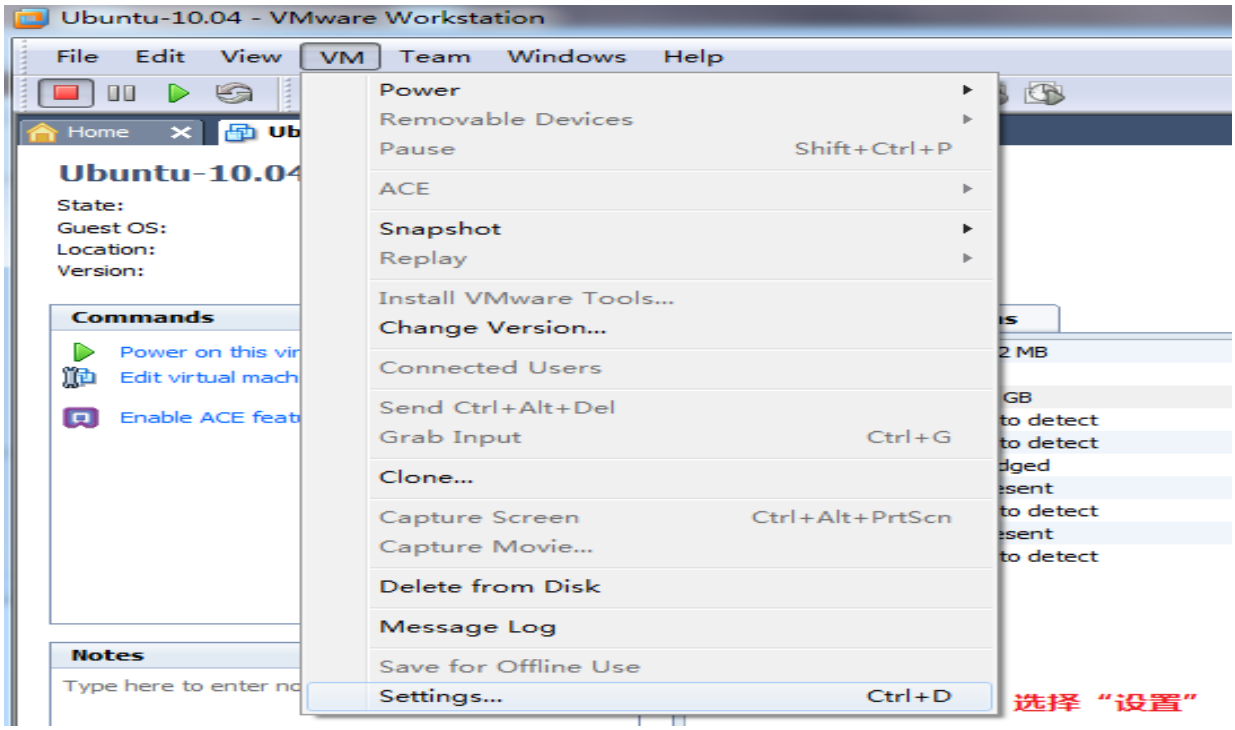
（9）新建虚拟机完成，点击“Finish”，如图。

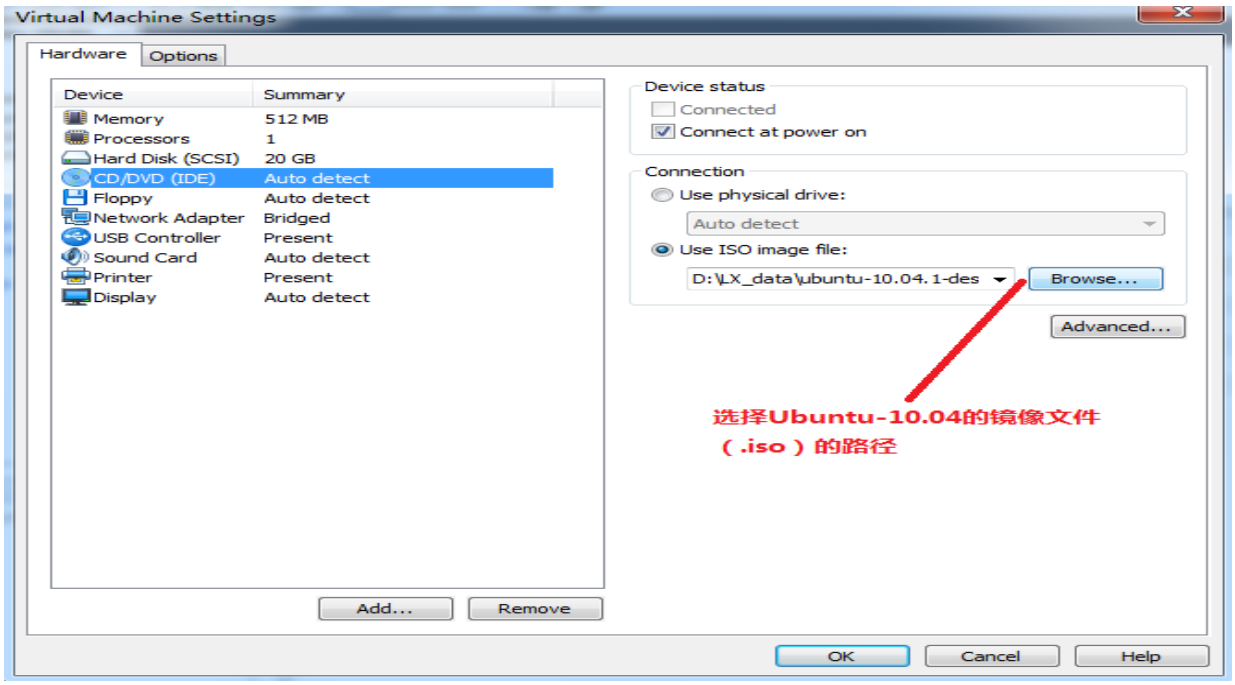



### 4.1.3 安装 Ubuntu 系统

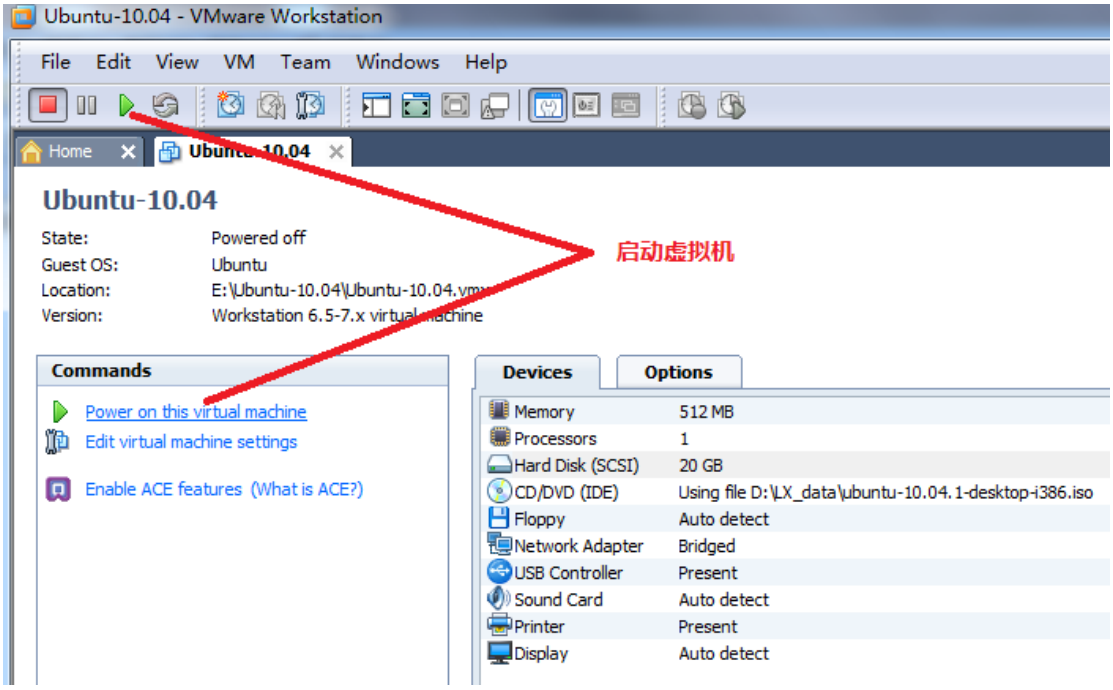
软件工具位置：Loongson_1B/Tools/windows_tools/Ubuntu-desktop-i386

（1）设置，修改 Ubuntu 系统镜像文件的路径，选择 VMware Workstation 的菜单栏的“VM”，弹出下拉菜单，选择“Setting...”；进入对话框“Virtual Machine Setting”，点击“Hardware”选项的“CD/DVD(IDE)”Device，在右边的“Device status”选项框选择“Connect at power on”，且“Connection”选择框选择“Use ISO image file”，输入或选择 Ubuntu10.04 镜像文件的路径，如“D:\ubuntu-10.04.1-desktop-i386.iso”，如图：

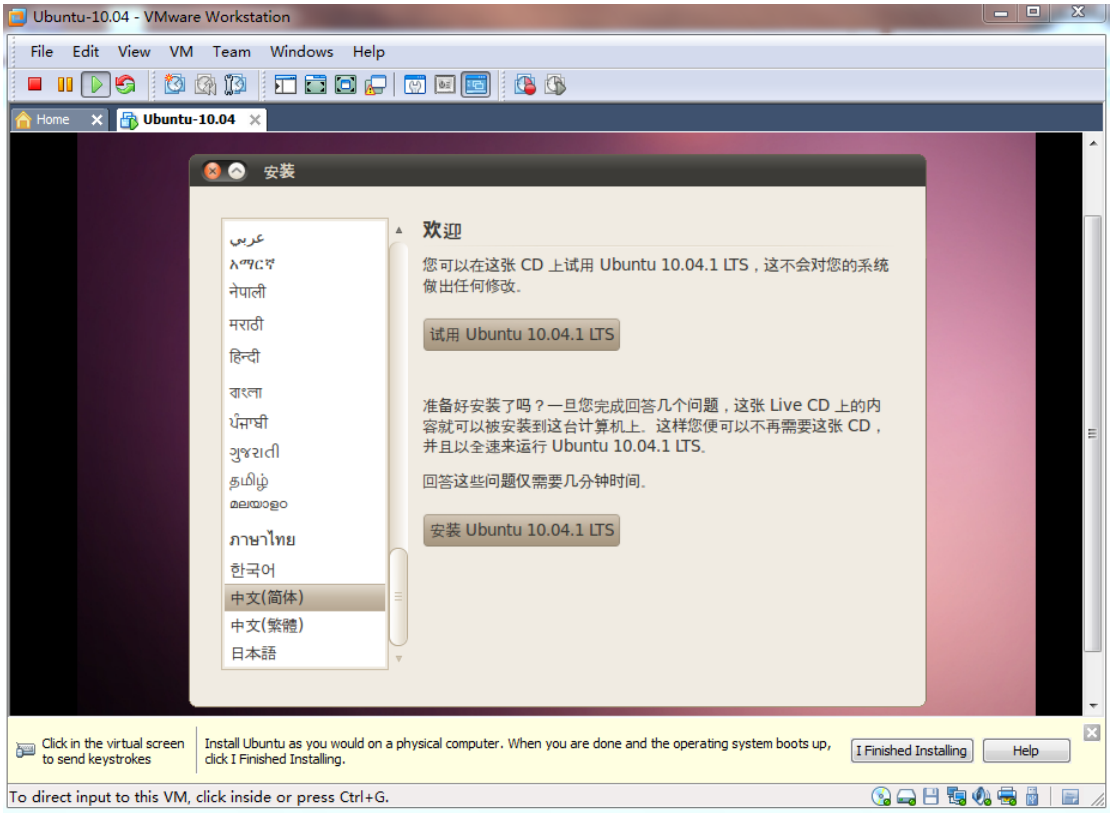



（2）启动 Ubuntu-10.04 虚拟机，点击工具栏的绿色开始按钮，或在 Ubuntu-10.04 虚拟机上“Commands”的选项“Power on this virtual machine”，如图：

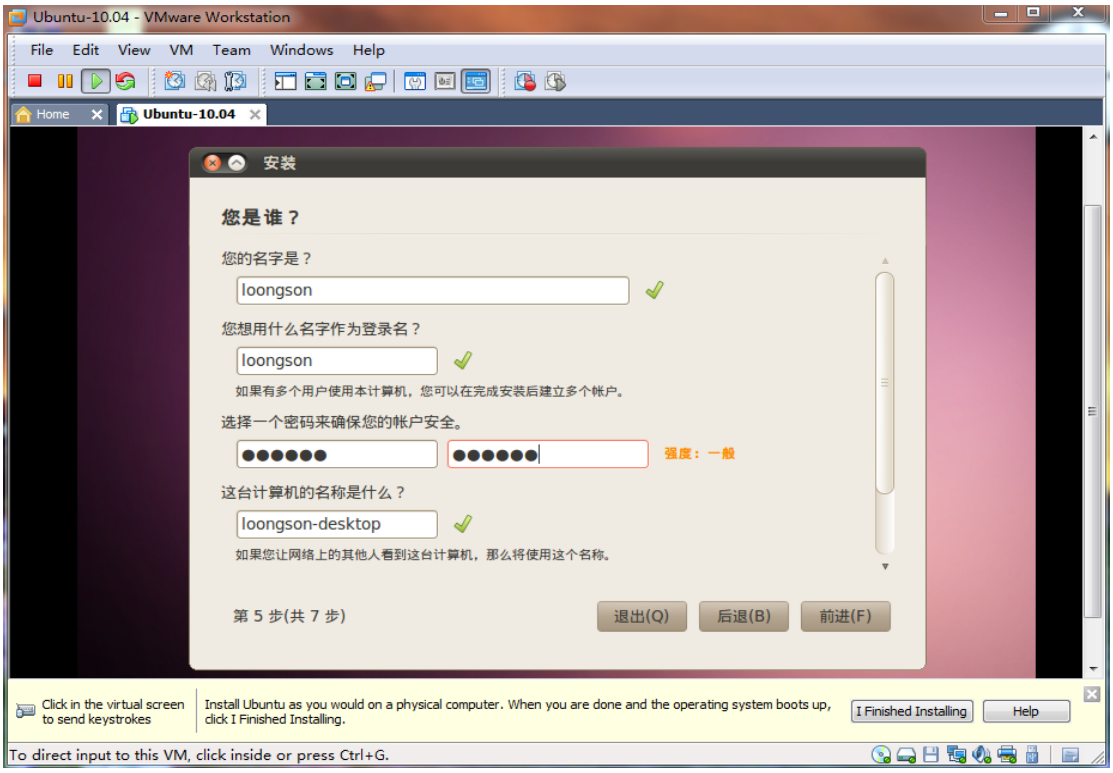



（3）进到安装 Ubuntu10.04 操作系统界面，语言字体选择“中文（简体）”，右边选择“安装 Ubuntu 10.04.1 LTS”，如图：



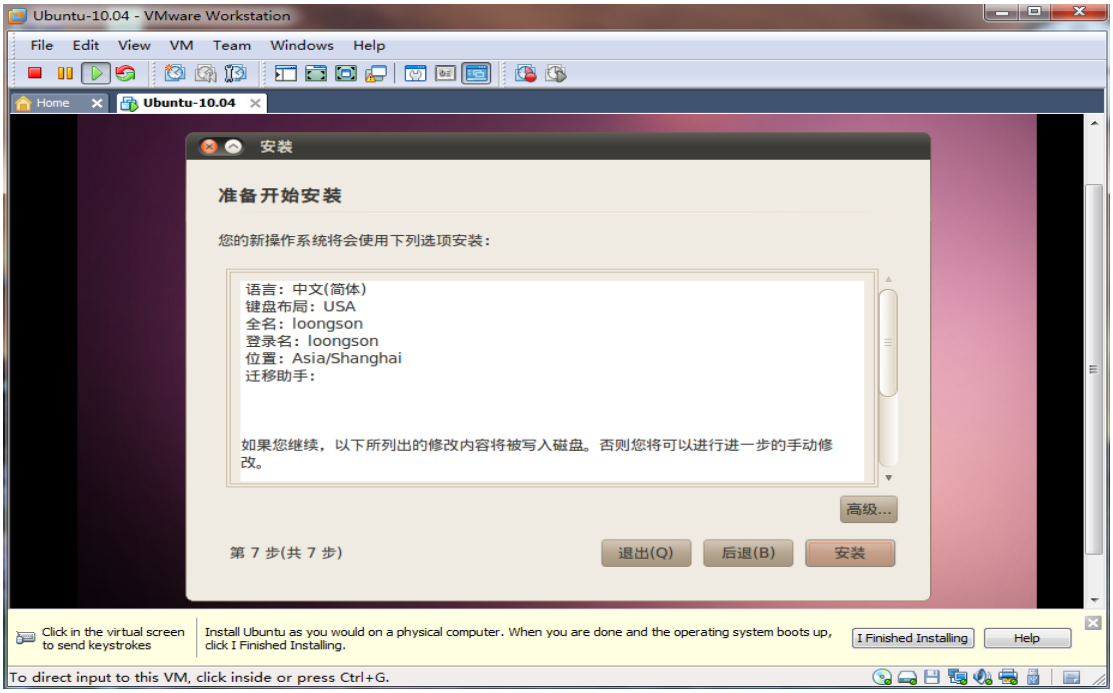

（3） 往下这几个步骤默认选择“前进（F）”。

（5）进到设置用户名和密码界面，如输入用户名“loongson”，密码输入两次，输入所有选项后点击“前进（F）”，如图：




提示：请记住自己所设置的用户名与密码。

（6）如需进行自主配置安装，点击“高级”，如果不熟悉 Ubuntu 系统安装，建议选择“安装”进行默认安装，如图：

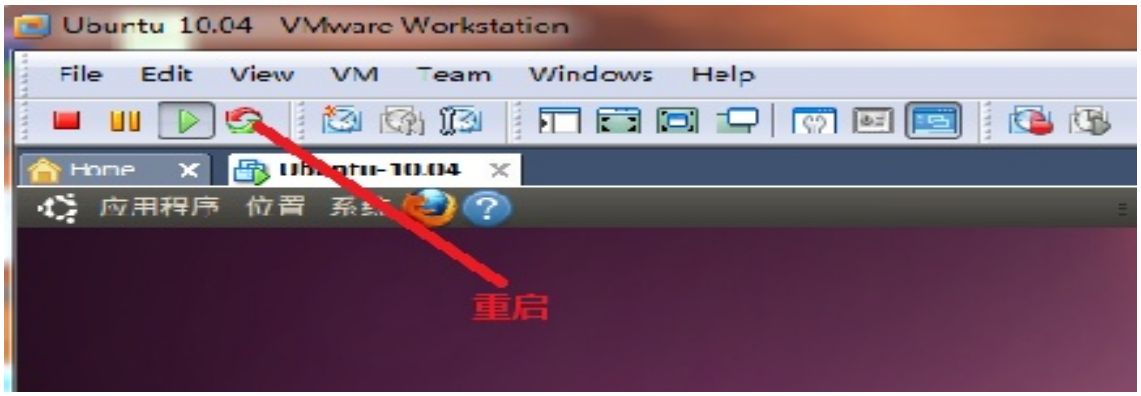


**安装过程注意：**

安装过程如果没有网络，那么安装完成得到的 Ubuntu 的语言环境为英文；如果想修改为中文的语言环境，可以在安装完成后进到系统联网更新语言包。

VMware-workstation（虚拟机）底角标注：To return to your computer , press Ctrl + Alt .
安装过程中，如果点击 VMware-workstation（虚拟机），鼠标“消失”了，按“Ctrl + Alt”重现鼠标。

（7）安装完成后，重启虚拟机，准备进入系统，点击“Ubuntu 10.04 WMware Workstation”工具栏的重启按钮（红蓝箭头圆形图按钮），如图。




## 4.1.4 备份 Ubuntu 系统

在使用 Ubuntu 操作系统过程中，可能会遇到各种问题导致不能启动或进入 Ubuntu 操作系统，此时如果有其备份文件，将会很方便地还原之前的资源，安全地保护了重要的文件资源。备份 Ubuntu 操作系统有两种方式：第一种是简单的拷贝备份虚拟机目录；第二种是使用 VMware-workstation 软件的快照备份功能。

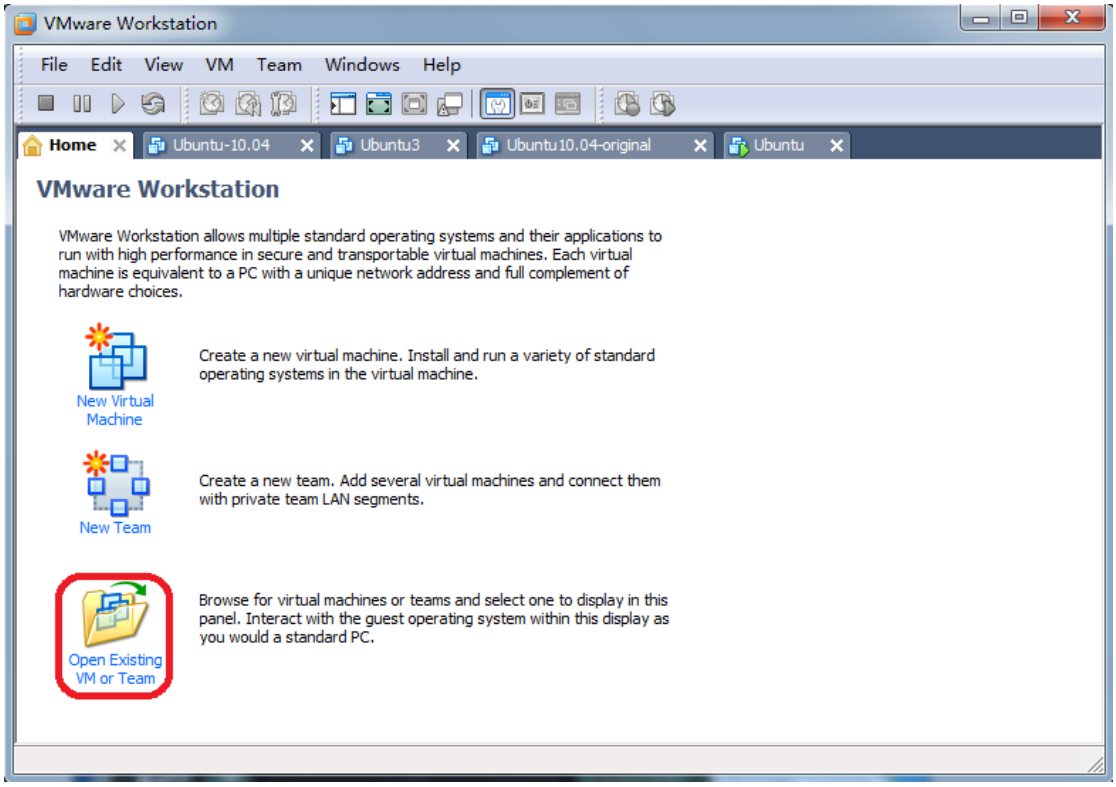
（1）拷贝虚拟机目录备份方式

a）备份 Ubuntu 操作系统的虚拟机目录

如 Ubuntu 操作系统的虚拟机目录为“E:\Ubuntu-10.04”，则把其拷贝备份为“E:\Ubuntu-10.04(备份)”。

b）打开备份 Ubuntu 操作系统的虚拟机

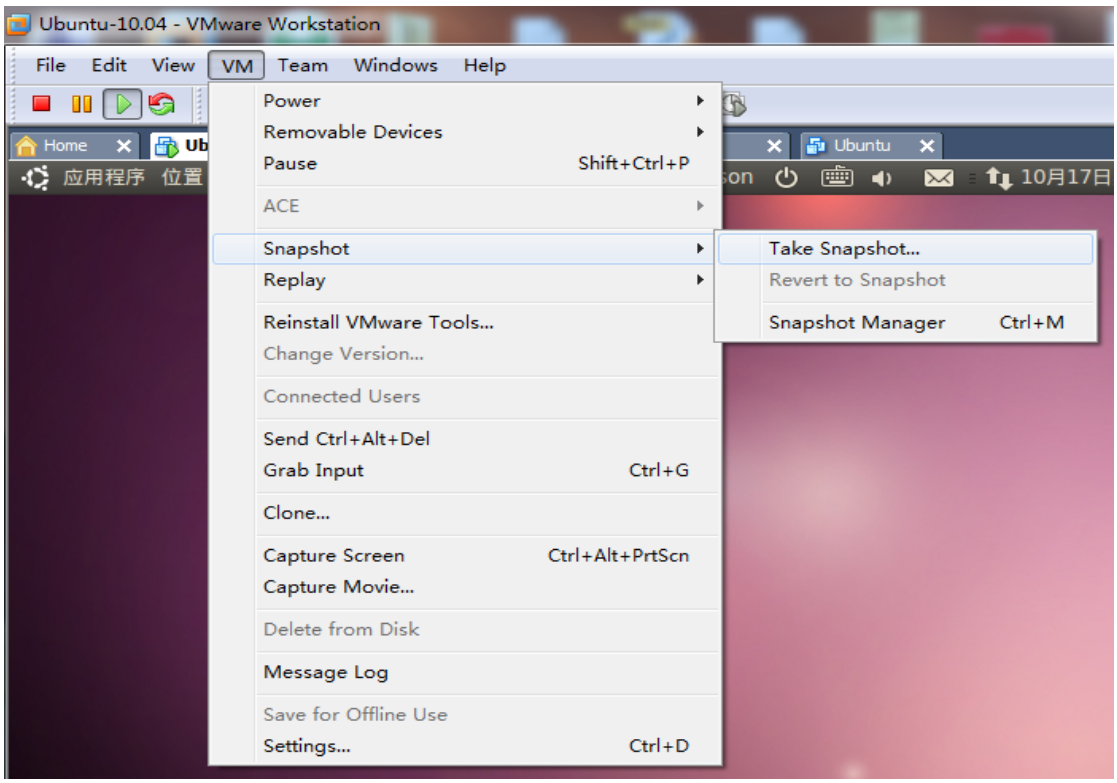
点击 VMware-workstation 的主界面 Home 下“Open Existing VM or Team”选项；弹出虚拟机打开对话框；在对话框中找到想要打开的虚拟机（如 E:\Ubuntu-10.04\Ubuntu-10.04.vmdk）；然后再启动此备份的虚拟机；如下图：



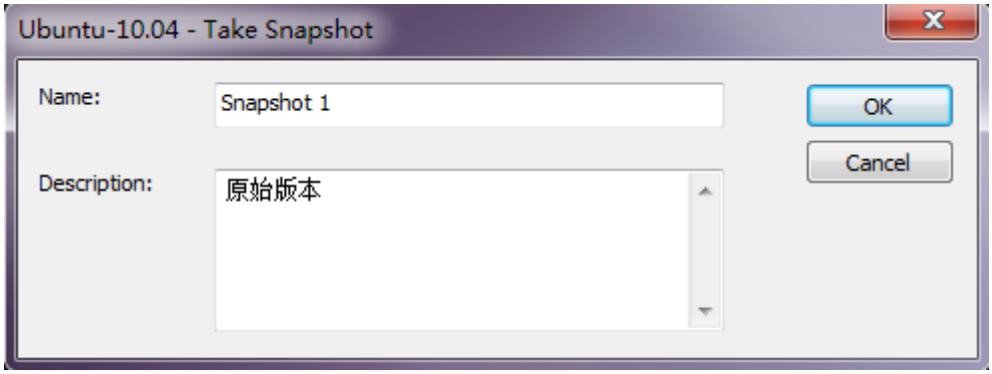

（2）快照功能备份方式

a）创建快照备份

点击 Ubuntu-10.04-VMware-workstation 中菜单栏的 VM 菜单；选择下拉菜单的 Snapshot 选项，再点击其右侧下拉菜单的“Take Snapshot...”选项，如下图：

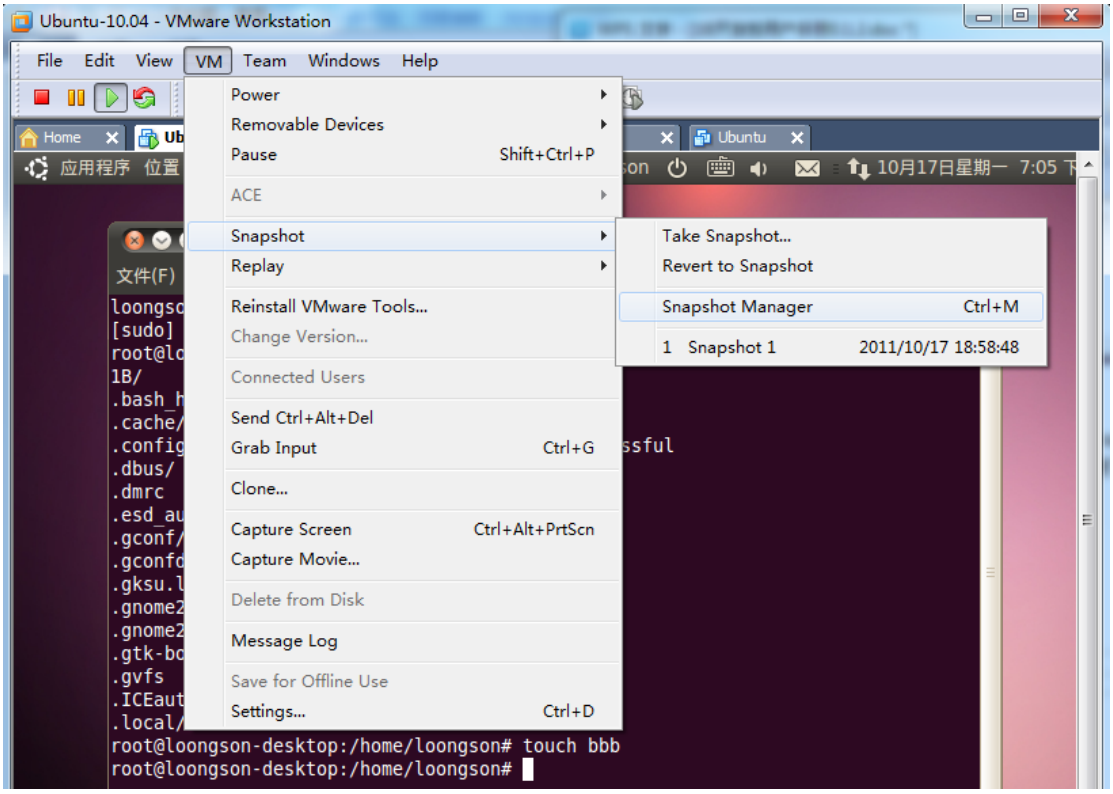


弹出 Ubuntu-10.04-Take Snapshot 对话框，在 Name 选项中输入快照的名字“Snapshot 1”，在 Description 选项输入此快照的描述信息“原始版本”以便记忆，点击“OK”完成快照的创建，如下图：

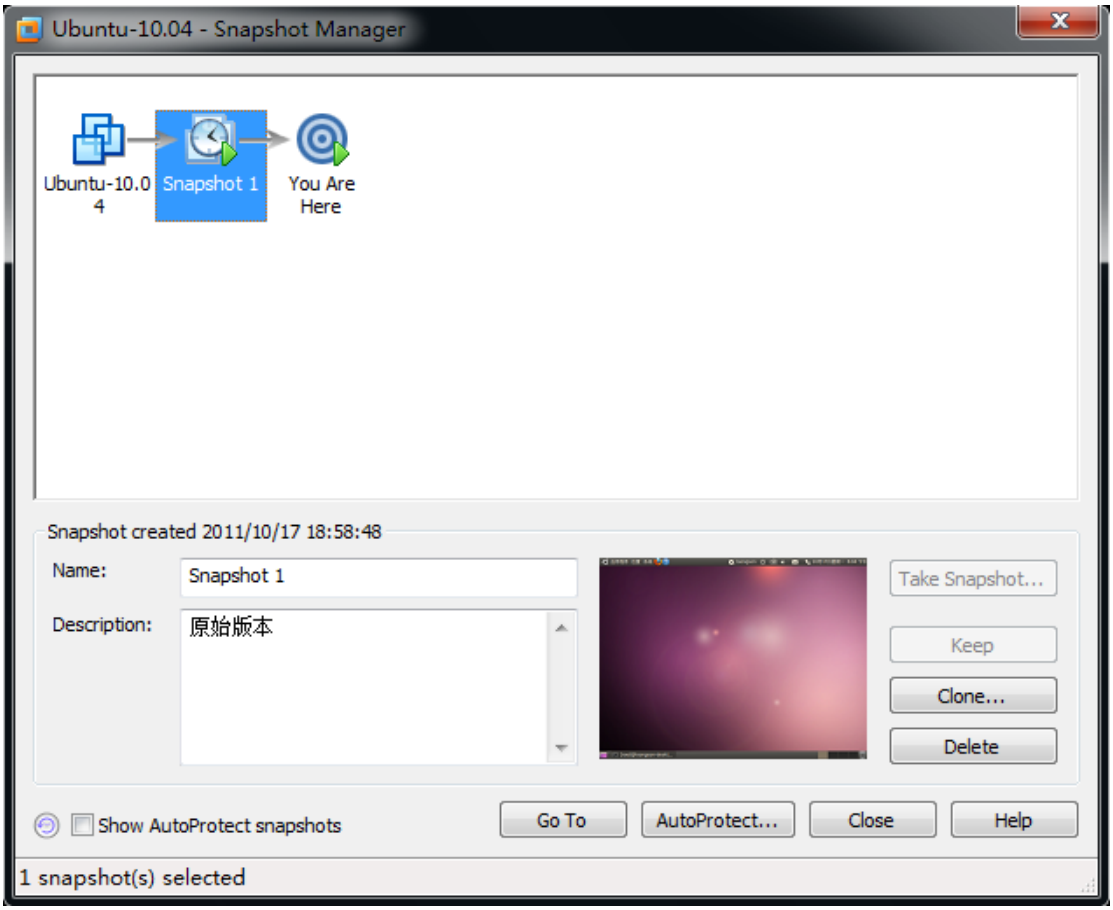



b）打开快照备份

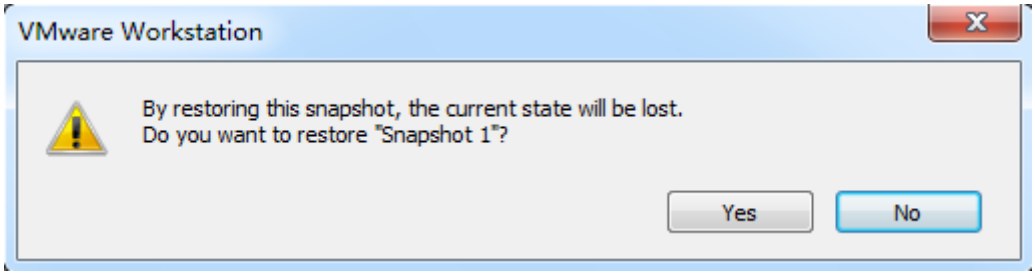
点击 Ubuntu-10.04-VMware-workstation 中菜单栏的 VM 菜单；选择下拉菜单的 Snapshot 选项，再点击其右侧下拉菜单的“Snapshot Manager”选项；如下图：




弹出 Ubuntu-10.04-Snapshot Manager 对话框，在主图形选项界面中选择所需要的快照（如选择“Snapshot 1”）；点击“Go To”按钮打开所需要的快照；如下图：




弹出提示信息，点击“Yes”按钮，如下图：



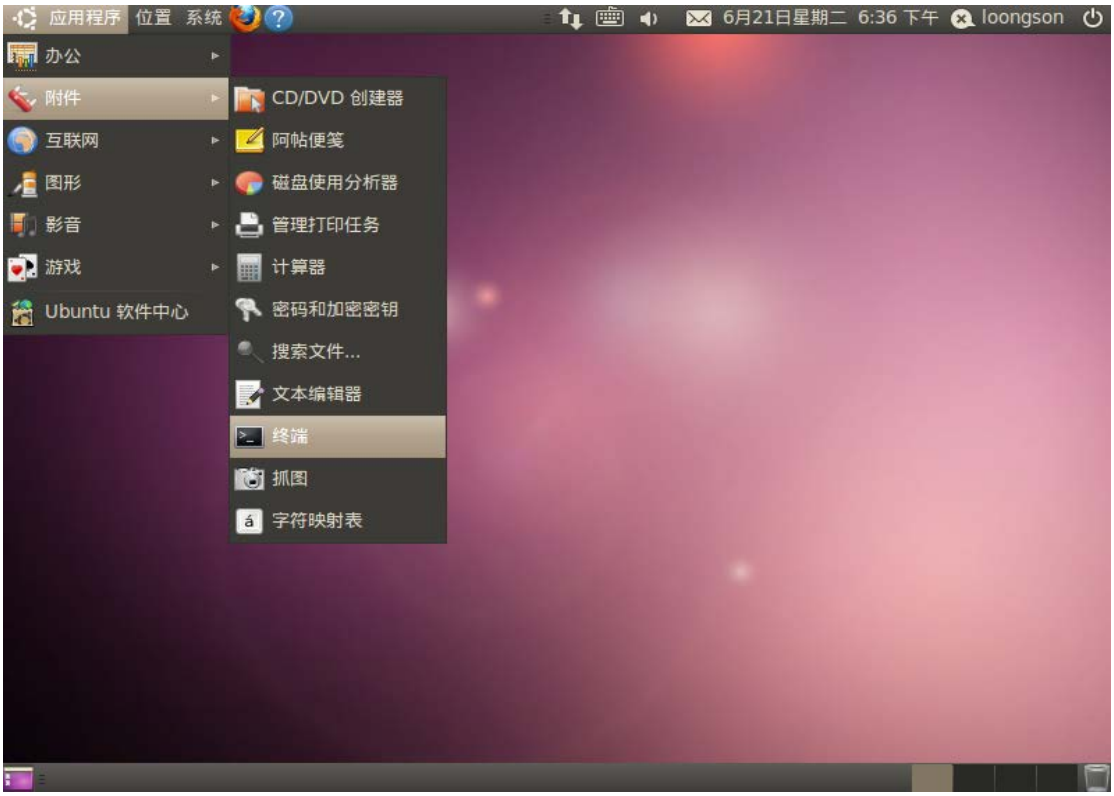

最后将会还原此快照备份时的资源。

## 4.2 使用 Ubuntu10.04

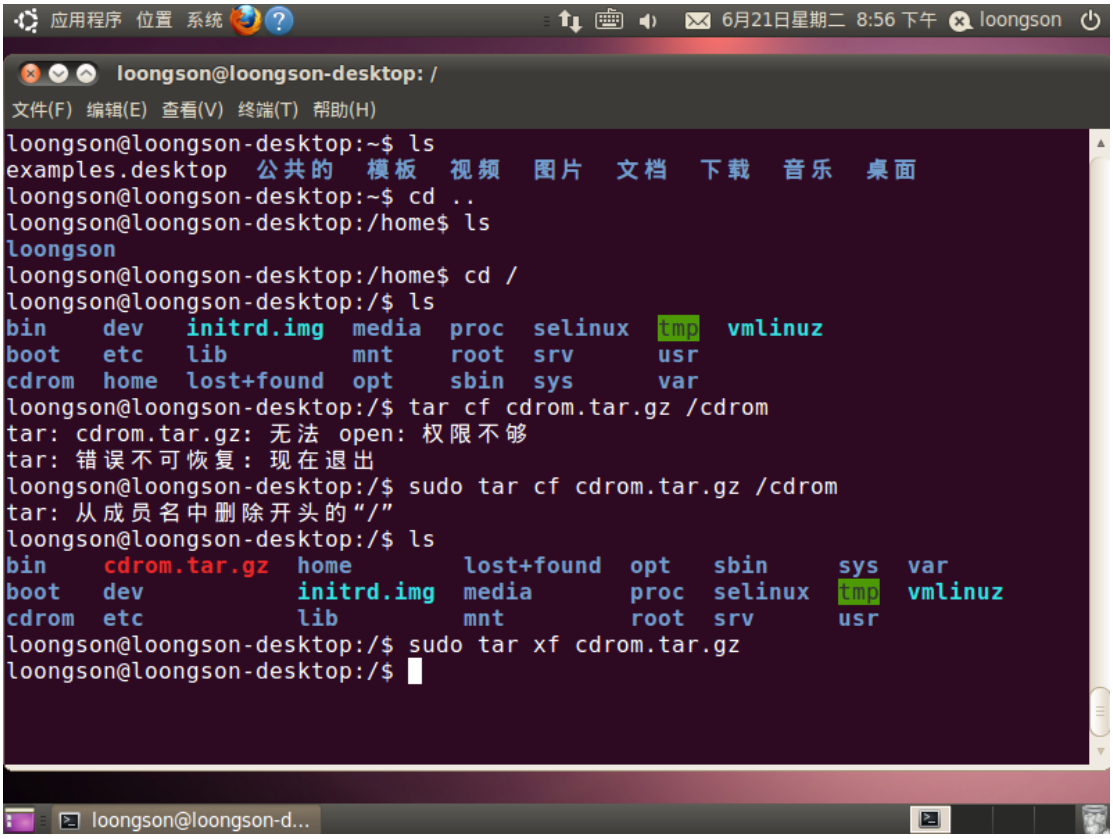
### 4.2.1 Linux 终端

Linux 中的所有管理任务都可以在终端中完成。Linux 终端使用命令行模式，许多情况下，使用终端比使用图形化的程序更快捷，而且还可能实现额外的功能。

启动终端，进入系统点击“应用程序”菜单，弹出下拉菜单，选择“附件”，弹出侧边下拉菜单，选择“终端”，如图：


应用程序 位置 系统
6月21日星期二 6:36 下午 loongson
办公
附件
互联网
图形
影音
游戏
Ubuntu 软件中心
CD/DVD 创建器
阿帖便笺
磁盘使用分析器
管理打印任务
计算器
密码和加密密钥
搜索文件...
文本编辑器
终端
抓图
字符映射表

## 4.2.2 初体验


应用程序 位置 系统
6月21日星期二 8:56 下午 loongson
loongson@loongson-desktop: /
文件(F) 编辑(E) 查看(V) 终端(T) 帮助(H)
loongson@loongson-desktop:~$ ls
examples.desktop 公共的 模板 视频 图片 文档 下载 音乐 桌面
loongson@loongson-desktop:~$ cd ..
loongson@loongson-desktop:/home$ ls
loongson
loongson@loongson-desktop:/home$ cd /
loongson@loongson-desktop:/$ ls
bin dev initrd.img media proc selinux tmp vmlinuz
boot etc lib mnt root srv usr
cdrom home lost+found opt sbin sys var
loongson@loongson-desktop:/$ tar cf cdrom.tar.gz /cdrom
tar: cdrom.tar.gz: 无法 open: 权限不够
tar: 错误不可恢复：现在退出
loongson@loongson-desktop:/$ sudo tar cf cdrom.tar.gz /cdrom
tar: 从成员名中删除开头的“/”
loongson@loongson-desktop:/$ ls
bin cdrom.tar.gz home lost+found opt sbin sys var
boot dev initrd.img media proc selinux tmp vmlinuz
cdrom etc lib mnt root srv usr
loongson@loongson-desktop:/$ sudo tar xf cdrom.tar.gz
loongson@loongson-desktop:/$
loongson@loongson-d...

简单常用命令（只适用于“搭建 LINUX 开发环境”） ls、cd、sudo、cp、tar

| 命令 | 说明 | 例子 | 例子含义 |
|---|---|---|---|
| ls | 文件列表 | ls | 以默认方式显示当前目录文件列表 |
| | | ls -a | 显示所有文件包括隐藏文件 |
| | | ls -l | 显示文件属性，包括大小，日期，符号连接，是否可读写及可执行 |
| cd | 目录切换 | cd dir | 切换到当前目录下的 dir 目录 |
| | | cd / | 切换到根目录 |
| | | cd .. | 切换到到上一级目录 |
| sudo | 超级用户模式 | sudo tar xfzv file.tgz | 在普通用户模式下将文件 file.tgz 解压 |
| | | sudo su | 切换到超级用户模式 |
| cp | 复制 | cp source target | 将文件 source 复制为 target |
| | | cp /root/source . | 将/root 下的文件 source 复制到当前目录 |
| | | cp -av soure_dir target_dir | 将整个目录复制，两目录完全一样 |
| | | cp -fr source_dir target_dir | 将整个目录复制，并且是以非链接方式复制，当 source 目录带有符号链接时，两个目录不相同 |
| tar | 压缩与解压 | tar xfzv file.tgz | 将文件 file.tgz 解压 |
| | | tar cfzv file.tgz source_path | 将文件 source_path 压缩为 file.tgz |

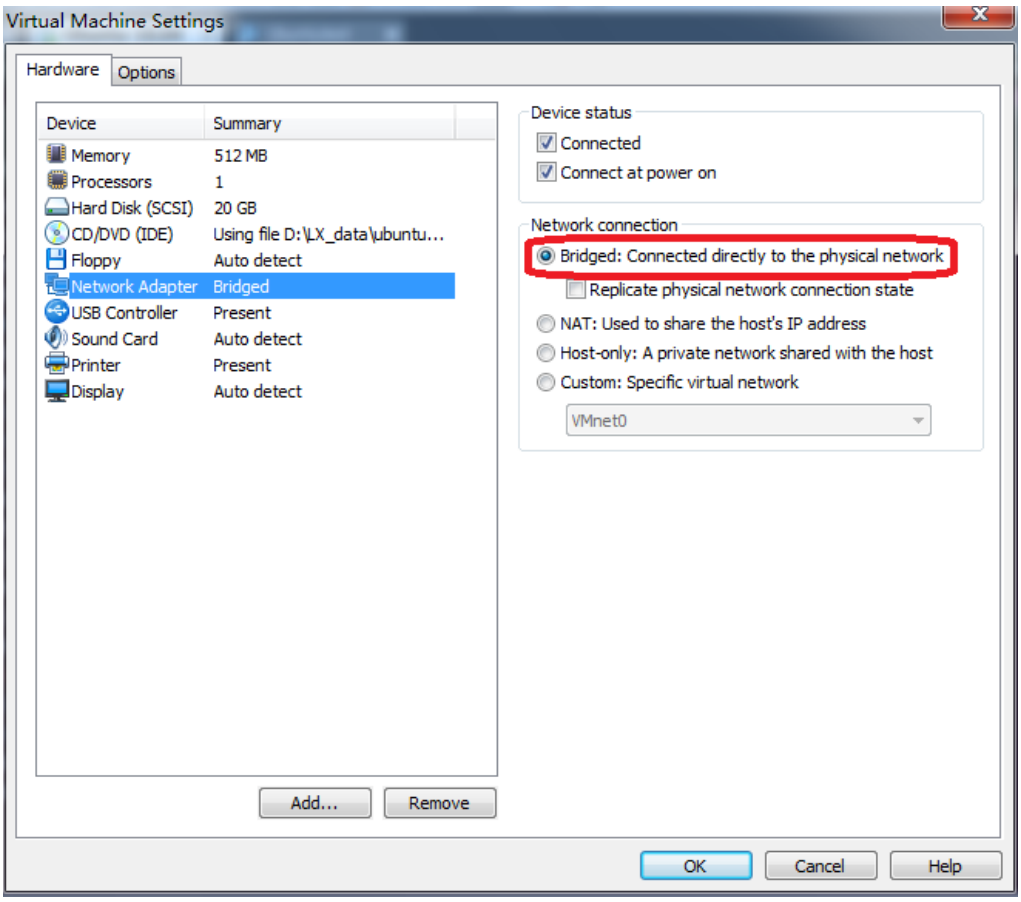
提示：需要了解更多，请参考“附录 2 Linux 常用命令详解”。

### 4.2.3 常用设置

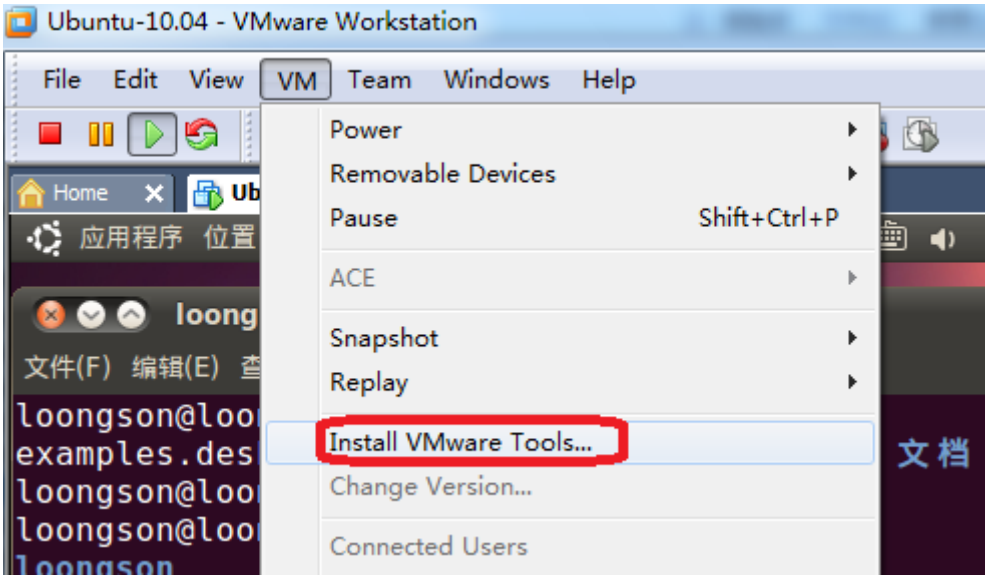
（1）Windows 与 Ubuntu 间文件的访问

参考“附录 1 Windows 与 Ubuntu 间文件的传输”

在 Ubuntu 10.04 WMware Workstation 中，选择“VM” --->“ Settings...”,“Network Adapter” --->“Networkconnection”，确认网络连接方式为“Bridged”，如图：



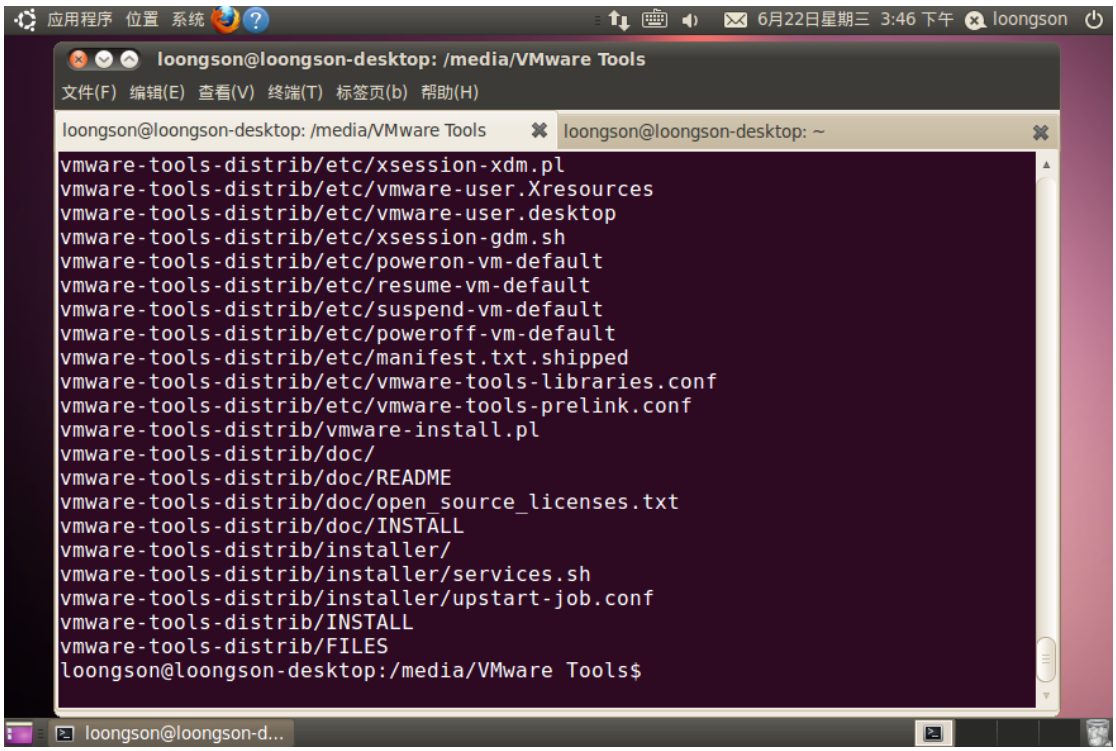

（2）安装虚拟机工具（VMware Tools），选择“VM”--->“Install VMware Tools...”，如图：




（3）进入终端，输入完命令行，按回车键“Enter”，如下：

```
#cd /media/
```

```
#ls
#cd VMware\ Tools/
#ls
#sudo tar zxvf VMwareTools-8.4.5-324285.tar.gz -C /tmp/
```

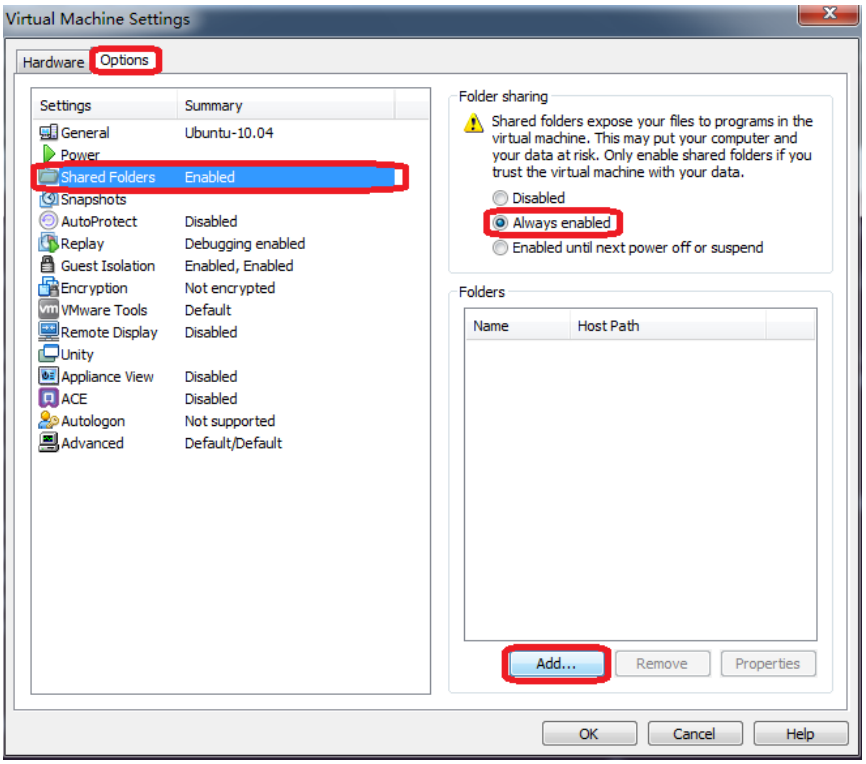



```
#cd /tmp/
#ls
#cd vmware-tools-distrib/
#ls
#sudo ./vmware-install.pl
```

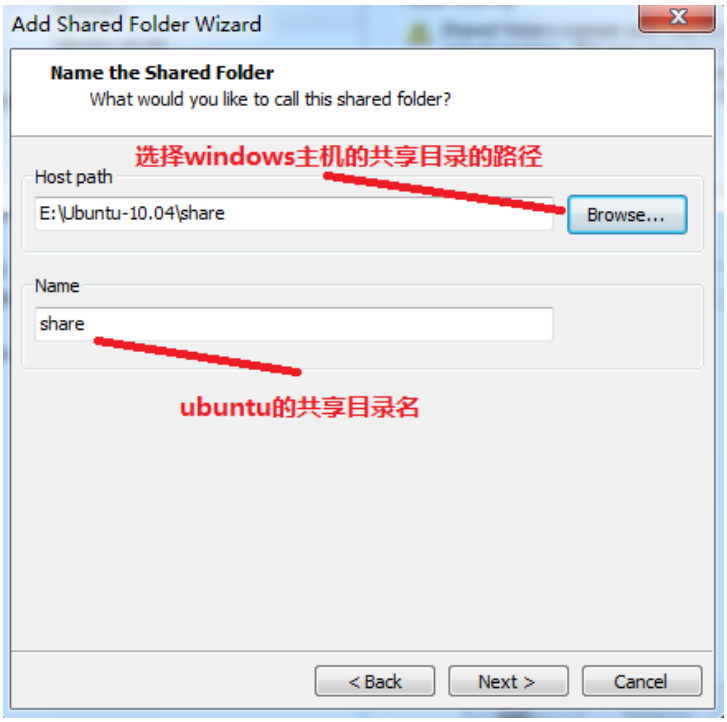
（4）安装过程中，需要选择的话，如果是[yes]，则输入 yes 再按回车键，其他的默认按回车键。直到执行完成。

（5）建立共享关系，添加共享目录。

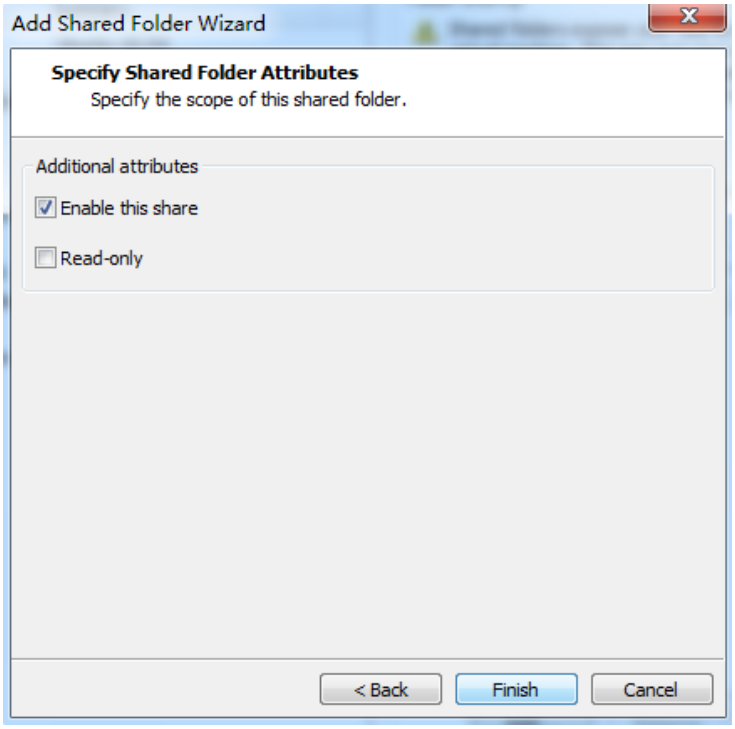
在 Ubuntu 10.04 WMware Workstation 中，选择“VM” --->“ Settings...”，选择“Options”选项，“Shared Folders”--->“Folder sharing”，选择“Always enabled”；在“Folders”选项下单击“Add...”添加共享目录，如下图：




（6）弹出对话框“Add Shared Folder Wizard”，在“Host path”选项输入或选择共享目录，如“E:\Ubuntu 10.04\share”Windows 共享目录路径；在“Name”选项下输入“share”Ubuntu 共享目录名，完成后点击“Next >”，如下图：




（7）最后，“Finish”--->“OK”，添加成功，退出对话框，如下图：




完成后，Windows 上的文件放于“E:\Ubuntu-10.04\share”，Ubuntu 可以通过访问“/mnt/hgfs/share/”来共享 Windows 上的文件；反之，Ubuntu 上的文件放于“/mnt/hgfs/share/”，Windows 可以通过访问“E:\Ubuntu-10.04\share”来共享 Ubuntu 上的文件。这样就实现了 Windows 与 Ubuntu 间文件的共享与传输。

另外，还有一种较简单的传输方式——**直接拖拽：**

在安装了 VMware Tools 之后，可以打开 Ubuntu 桌面的“位置”，进到“文件浏览器”（如：主文件夹），直接把 Windows 上的文件拖拽进到“文件浏览器”内。但需要以 root 用户（需要输入命令“passwd root”添加密码）登录，否则在没有权限的目录下不能传输文件。

提示：第一，如果不能实现共享，有可能是 VMware Tools 没有安装成功，或者是已经把 VMware-workstation 软件汉化了，汉化补丁破坏了实现共享的功能。第二，建议不要把压缩文件解压于共享目录（如/mnt/hgfs/share/）下。创建链接时会报错。

### 4.2.4 安装 TFTP

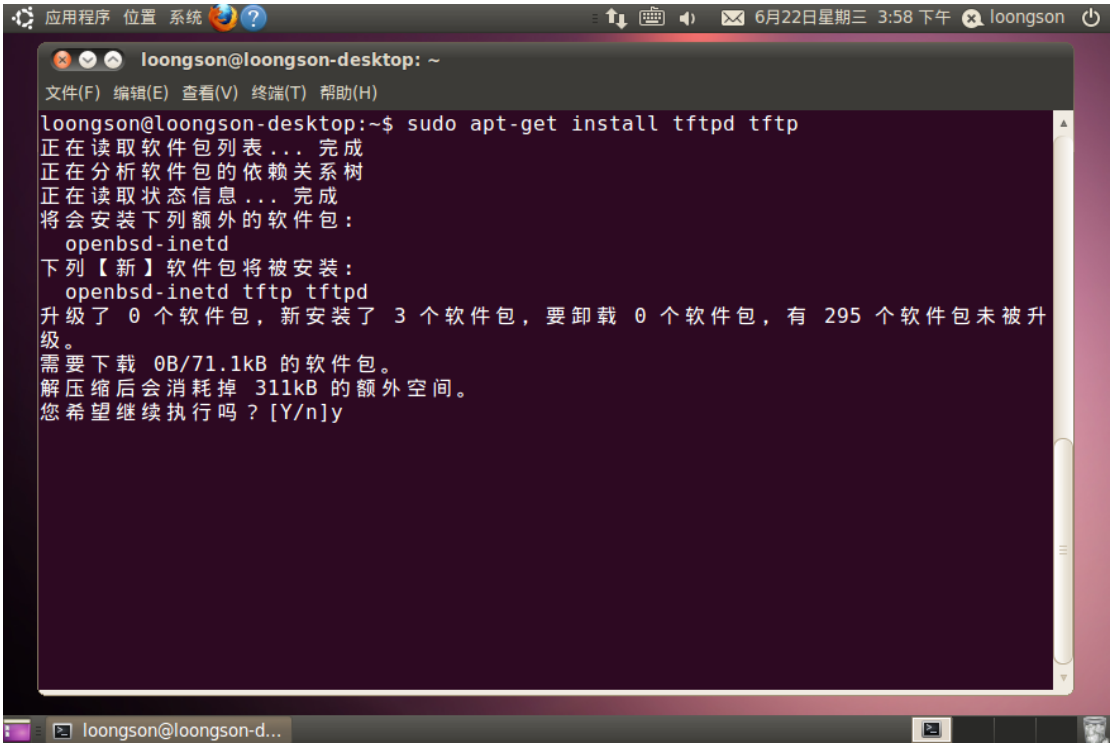
安装 TFTP 有两种方式：有互联网时使用 apt-get 命令来安装（默认方式）；无互联网时使用源码包来安装。

（1）使用 apt-get 安装（有互联网）

apt-get 命令主要用于自动从互联网的软件仓库中搜索、安装、升级、卸载软件或操作系统。

安装服务器与客户端

```
#sudo apt-get install tftpd tftp
```




安装 Xinetd

a）安装 xinetd：

**#sudo apt-get install xinetd**

b）建立配置文件：

在/etc/xinetd.d/下建立一个配置文件 tftp

**#sudo vim tftp**

在文件中输入以下内容：

```
service tftp
{
    socket_type = dgram
    protocol = udp
    wait = yes
    user = root
    server = /usr/sbin/in.tftpd
    server_args = -s /tftpshare    /*与 tftp 存放文件夹路径一致*/
    disable = no
    per_source = 11
    cps = 100 2
    flags = IPv4
}
```

保存退出。

注意：等号“=”两边需要保留空格。

c）建立 tftp 服务文件目录（上传文件与下载文件的位置），并且更改其权限

```
#sudo mkdir /tftpshare
#sudo chmod 777 /tftpshare
```

d）重新启动服务

```
#sudo /etc/init.d/xinetd restart
```

把需要 tftp 传输服务的文件放于“/tftpshare”下。

（2）使用源码包安装（无互联网）

源码包位置：**Loongson_1B/Tools/other_tools/tftp/tftp-install.tar.gz**

```
#tar zxf tftp-install.tar.gz
#cd tftp-install
#source install_tftp.sh
```

为了较清晰地表现安装流程，脚本文件 install_tftp.sh 的内容如下：

```
#!bin/bash
#安装 TFTP

#解压 tftp 和 xinetd 源码包
echo "======= tar tftp and xinetd ======="
tar xf tftp-hpa-5.1.tar.gz
tar xf xinetd-2.3.14.tar.gz

#安装 tftp
echo "=======       install tftp        ======="
cd tftp-hpa-5.1
./configure && make && make install

#安装 xinetd
echo "=======       install xinetd      ======="
cd ../xinetd-2.3.14
./configure && make && make install

#创建配置文件
echo "======= create configuration file ======="
cd ..
touch /etc/xinetd.conf
echo "service tftp
    {
```

```
        socket_type    = dgram
        protocol       = udp
        wait            = yes
        user            = root
        server          = /usr/sbin/in.tftpd
        server_args    = -s /tftpshare -c
        disable         = no
        per_source     = 11
        cps             = 100 2
        flags           = IPv4
}" > /etc/xinetd.conf

#创建共享服务目录
echo "=======    create shared dir    ======="
mkdir -p /tftpshare
chmod 777 /tftpshare

#启动 xinetd 服务
echo "=======    xinetd running      ======="
xinetd -f /etc/xinetd.conf

#重启系统时启动 xinetd 服务
echo "====== xinetd running forever ======="
echo "xinetd -f /etc/xinetd.conf" >> /root/.bashrc
```

（3）本机测试

创建测试文件

```
#touch /tftpshare/test                #创建测试文件 test
#echo "hello" >> /tftpshare/test      #文件内容为“hello”
```

TFTP 测试

```
#tftp localhost   #在本机开启 tftp 服务
tftp>get test     #通过 tftp 协议把服务目录 tftpshare 的文件 test 下载
  到当前目录
tftp>q            #退出
#cat test         #查看当前目录里的 test 文件的内容
hello             #内容
```

TFTP 服务可用。

## 4.3 建立交叉编译环境

源码包位置：**Loongson_1B/Tools/toolchain/gcc-3.4.6-2f.tgz**

（1）在 Windows 上把 gcc-3.4.6-2f.tgz 复制到共享目录“E:\Ubuntu-10.04\share”，在 Ubuntu 上再到“/mnt/hgfs/share/”中把交叉编译工具链 gcc-3.4.6-2f.tgz 复制到“/tmp”或自己创建管理的目录。然后进到该目录，解压。

复制到“/tmp”：

**#cp /mnt/hgfs/share/gcc-3.4.6-2f.tar.gz /tmp**

解压到/opt 目录下：

**#tar zxvf gcc-3.4.6-2f.tar.gz -C /opt**

（2）设置工具链的路径到系统环境变量

**#export PATH=/opt/gcc-3.4.6-2f/bin:$PATH**

输入命令 mipsel-linux-gcc -v ，看到如下显示，则交叉编译环境已经建立。

```
root@loongson-desktop:/home/loongson/1B/toolschain# mipsel-linux-gcc -v
从 /opt/gcc-3.4.6-2f/bin/../lib/gcc/mipsel-linux/3.4.6/specs 读取 specs
配置为： ../gcc-3.4.6/configure --prefix=/opt/gcc-3.4.6-2f/ --target=mipsel-lin
ux --host=i686-linux --enable-threads=posix --enable-shared --disable-checking
 --enable-languages=c,c++,f77 -v
线程模型： posix
gcc version 3.4.6
```

提示：

这条命令的作用只在当前终端有效，即交叉编译环境只在当前终端起作用。需要在整个系统建立交叉编译环境，可以把/opt/gcc-3.4.6-2f/bin: 添加到/root/.bashrc 文件最后一行。

也可以运行命令行 echo "export PATH=/opt/gcc-3.4.6-2f/bin:$PATH" >> /root/.bashrc。

*（如果工具链不能正常工作，建议重新解压到/home/cpu 目录下）*

然后打开新终端切换到超级用户权限（sudo su），运行 echo $PATH 查看验证。

# 第五章 编译 BOOTLOADER（PMON）和 LINUX

## 5.1 编译 BOOTLOADER（PMON）

源码包位置：**Loongson_1B/BSP/Pmon/1b-pmon.tar.gz**

PMON 是一个兼有 BIOS 和 bootloader 部分功能的开放源码软件，多用于嵌入式系统。基于龙芯的系统采用 PMON 作为类 BIOS 兼 bootloader，并在其基础上做了很多完善工作，支持 BIOS 启动配置，内核加载，程序调试，内存寄存器显示、设置以及内存反汇编等等。

### 5.1.1 工具与依赖库安装

工具与依赖库的安装有两种方式：有互联网时使用 apt-get 命令来安装（默认方式）；无互联网时使用源码包来安装。

（1）使用 apt-get 命令安装（有互联网）

a）因为编译 PMON 过程需要使用到工具 pmoncfg，该工具 1b-pmon/tools/pmoncfg 目录下，编译该工具又需要依赖下面的工具：

```
#apt-get install bison
#apt-get install flex
```

b）解压 1b-pmon.tar.gz，编译生成 pmoncfg 工具：

```
#tar zxf 1b-pmon.tar.gz
#cd 1b-pmon
#cd 1b-pmon/tools/pmoncfg
#make
```

c）编译完成后会在当前目录下生成 pmoncfg，拷贝该工具至用户工具目录或交叉编译工具链的 bin 目录（参看 4-3 建立交叉编译环境）下。（推荐拷贝至交叉编译工具链目录中）

```
#cp pmoncfg /opt/gcc-3.4.6-2f/bin
```

d）pmon 编译还依赖于工具 makedepend：

```
#apt-get install xutils-dev
```

（2）使用源码包安装（无互联网）

源码包位置：**Loongson_1B/Tools/pmon_tools/pmon-dependtools.tar.gz**

提示：需要把源码包 pmon-dependtools.tar.gz 放在与 1b-pmon.tar.gz 相同的目录下。

```
#tar zxf pmon-dependtools.tar.gz
#source pmon-dependtools.sh
```

pmon-dependtools.sh 的内容为：

```
#!/bin/bash
#PMON tools install

tar zxf m4_1.4.13.orig.tar.gz              #configure 时需要
cd m4-1.4.13
./configure && make && make install
cd ..
```

```
tar zxf bison-2.4.3.tar.gz
tar zxf flex_2.5.35.orig.tar.gz
tar zxf xutils-dev_7.5+2.tar.gz

cd bison-2.4.3
./configure && make && make install
cd ..

cd flex-2.5.35
./configure && make && make install
cd ..

cd xutils-dev-7.5+2/makedepend
./configure && make && make install
cd ../..


tar zxf 1b-pmon.tar.gz
cd 1b-pmon/tools/pmoncfg
make

cp pmoncfg /usr/bin/

mkdir -p /opt/gcc-3.4.6-2f/bin
cp pmoncfg /opt/gcc-3.4.6-2f/bin
cd ../../..
```

### 5.1.2 配置与编译 pmon

（1）建立交叉编译环境（第四章 4-3 建立交叉编译环境）。

（2）解决库与工具依赖以后，开始编译 pmon：

```
#cd 1b-pmon/zloader.ls1b
```

（3）编译 bin 格式的 pmon

```
#make cfg all tgt=rom
```

执行后就在当前目录下生成了 gzrom.bin。

## 5.2 编译 Linux 内核

源码包位置：**Loongson_1B/BSP/Linux_Kernel/1b-linux-3.0.tar.gz**

Linux 内核很庞大，linux 初学者以及致力于 linux 应用软件开发的技术人员，熟悉内核的好的开始就是对内核进行配置，得到符合自己需求的经过裁剪的内核，并将编译后的内核下载到开发板中运行使用。

本篇内容是为想要对内核进行个性配置的人员准备的，不涉及到代码编写，学习 linux 不必一切从“零”开始，一切可从学会配置、编译、下载运行开始。

Linux 内核的编译分为两个步骤，一、内核配置；二、内核编译。开发包默认提供一个配置文件，用户可以根据此配置文件对内核进行裁剪或者增加新的功能。

下面以 Linux-2.6.21 内核为例，描述整个 Linux 系统的配置编译过程：

### 5.2.1 配置内核

（1）建立交叉编译环境

请参考“第四章 4-3 建立交叉编译环境”。

（2）进入 Linux 源代码树根目录：

```
#tar zxf 1b-linux-3.0.tar.gz
#cd 1b-linux-3.0
```

（3）安装图形化配置“make menuconfig”依赖的工具 Ncurses：

安装 Ncurses 有两种方式：有互联网时使用 apt-get 命令来安装（默认方式）；无互联网时使用源码包来安装。

a）使用 apt-get 命令安装（有互联网）

```
#apt-get install libncurses5-dev
```

b）使用源码包安装（无互联网）

源码包位置：Loongson_1B/Tools/other_tools/ncurses/ncurses-5.7.tar.gz

```
#tar zxf ncurses-5.7.tar.gz
#cd ncurses-5.7
#./configure && make make install
```

（4）图形化配置：

提示：调整终端窗口到合适大小或者最大化。

```
#make menuconfig
```

详细请参考《附录 4 内核配置详细说明》。

### 5.2.2 编译 linux 内核

```
#make
```

最终生成的内核映像文件就在内核源代码的根目录下，名为 vmlinux。

## 5.3 制作文件系统镜像

制作方法：先制作根文件系统，再利用不同的文件系统镜像制作工具对根文件系统目录制作成对应类型的文件系统镜像。

提示：制作根文件系统，请参看“附录 5 制作根文件系统”。

### 5.3.1 镜像文件制作工具

源码包位置：**Loongson_1B/Tools/mkfs_tools**

| | | |
|---|---|---|
| （1） | cramfs-1.1.tar.gz | （制作 cramfs 文件系统工具 ） |
| （2） | mtd-utils-1.0.0.tar.gz | （制作 jffs2 文件系统工具） |
| （3） | yaffs2-d43e901.tar.gz | （制作 yaffs2 文件系统工具） |
| （4） | zlib-1.2.3.tar.gz | （依赖工具） |

### 5.3.2 镜像文件制作工具本机安装

mkfs.cramfs 为 Ubuntu 自带的制作 cramfs 格式文件系统的工具。

在主机上编译安装镜像文件制作工具，无须交叉编译。可以使用 apt-get 安装，也可使用源码安装，下面使用源码包安装。

（1）安装依赖工具 zlib：

```
#tar zxf zlib-1.2.3.tar.gz
#cd zlib-1.2.3
#./configure && make && make install
```

（2）制作 cramfs 文件系统工具 mkcramfs：

```
#tar zxf cramfs-1.1.tar.gz
#cd cramfs-1.1
#make
```

当前目录生成 mkcramfs，可以拷贝到/usr/bin：

```
#cp mkcramfs /usr/bin
```

（3）制作 jffs2 文件系统工具 mkfs.jffs2

```
#tar zxf mtd-utils-1.0.0.tar.gz
#cd mtd-utils-1.0.0
#make
```

当前目录下生成 mkfs.jffs2，可以拷贝到/usr/bin：

```
#cp mkfs.jffs2 /usr/bin
```

（4）制作 yaffs2 文件系统工具 mkyaffs2image

```
#tar zxf yaffs2-d43e901.tar.gz
#cd yaffs2-d43e901/utils
#make
```

在当前目录 yaffs2-d43e901/utils 下生成 mkyaffs2image，可以拷贝到/usr/bin：

```
#cp mkyaffs2image /usr/bin
```

### 5.3.3 制作文件系统镜像文件

（1）Cramfs：

```
#mkcramfs /root/rootfs rootfs-cramfs.img
#chmod 777 rootfs-cramfs.img          //修改文件系统权限，防止出现无法烧写的情况
```

或者使用自带的工具：

```
#mkfs.cramfs /root/rootfs rootfs-cramfs.img
#chmod 777 rootfs-cramfs.img          //修改文件系统权限，防止出现无法烧写的情况
```

（2）Jffs2：

```
#mkfs.jffs2 -r /root/rootfs -o rootfs-jffs2.img -e 0x20000 --pad=0x2000000 -n
#chmod 777 rootfs-jffs2.img           //修改文件系统权限，防止出现无法烧写的情况
```

mkfs.jffs2 各参数的意义：

-r：指定要生成 image 的目录名。

-o：指定输出 image 的文件名。

-e：每一块要擦除的 block size，不同的 flash，其 block size 会不一样。这里为 128KB。

--pad：用 16 进制来表示所要输出文件的大小，也就是 rootfs-jffs2.img 的大小，如果实际大小不足此设定的大小，则用 0xFF 补足。

-n，-no-cleanmarkers：指明不添加清楚标记（nandflash 有自己的校检块，存放相关的信息）。

（3）Yaffs2：

```
#mkyaffs2image /root/rootfs rootfs-yaffs2.img
#chmod 777 rootfs-yaffs2.img          //修改文件系统权限，防止出现无法烧写的情况
```

# 第六章 烧写 BOOTLOADER（PMON）和 LINUX

## 6.1 烧写 BOOTLOADER（PMON）

### 6.1.1 烧写 PMON

开发板上 Nor Flash 中尚没有 PMON 时，需要为这空白的 Nor Flash 芯片烧写 PMON。详解参考“附录 7 使用 EJTAG 烧写 PMON”。

### 6.1.2 设置 IP 地址并测试

设置 IP 地址：

**PMON>ifaddr syn0 192.168.x.xxx （IP 地址起临时作用，断电后无效）**

或者：

**PMON>set ifconfig syn0:192.168.x.xxx**

**PMON>reboot （重启后，IP 地址固定存在）**

网络测试：

**PMON>ping PC 机的 IP 地址**

如：**PMON>ping 192.168.x.xxx**

注意：开发板与 PC 机的 IP 地址的前 3 个段需要相同。

### 6.1.3 网口更新 PMON

PMON 中内置 TFTP 协议，据此可以采用 TFTP 对 PMON 进行更新。采用 TFTP 更新方式，首先需要宿主机存在 TFTP 服务端软件（在“第四章 4-2-4 安装 TFTP”中已符合）。

（1）把交叉编译得到的 PMON 拷贝到服务目录“/tftpshare”。

（2）更新 PMON：

**PMON > load -r -f 0xbfc00000 tftp://192.168.0.244/gzrom.bin**

更新完成后重启。

### 6.1.4 串口更新 PMON

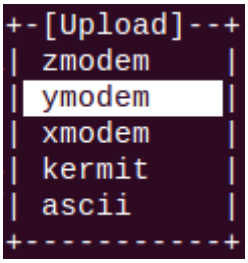
（2） 启动 ymodem 协议从串口下载 PMON：

**PMON>ymodem base=0x81800000**

提示：使用串口工具提供的 ymodem 协议下载 Pmon，该地址可以变动，与本身跑的 Pmon 不冲突即可，建议大于 0x80800000。

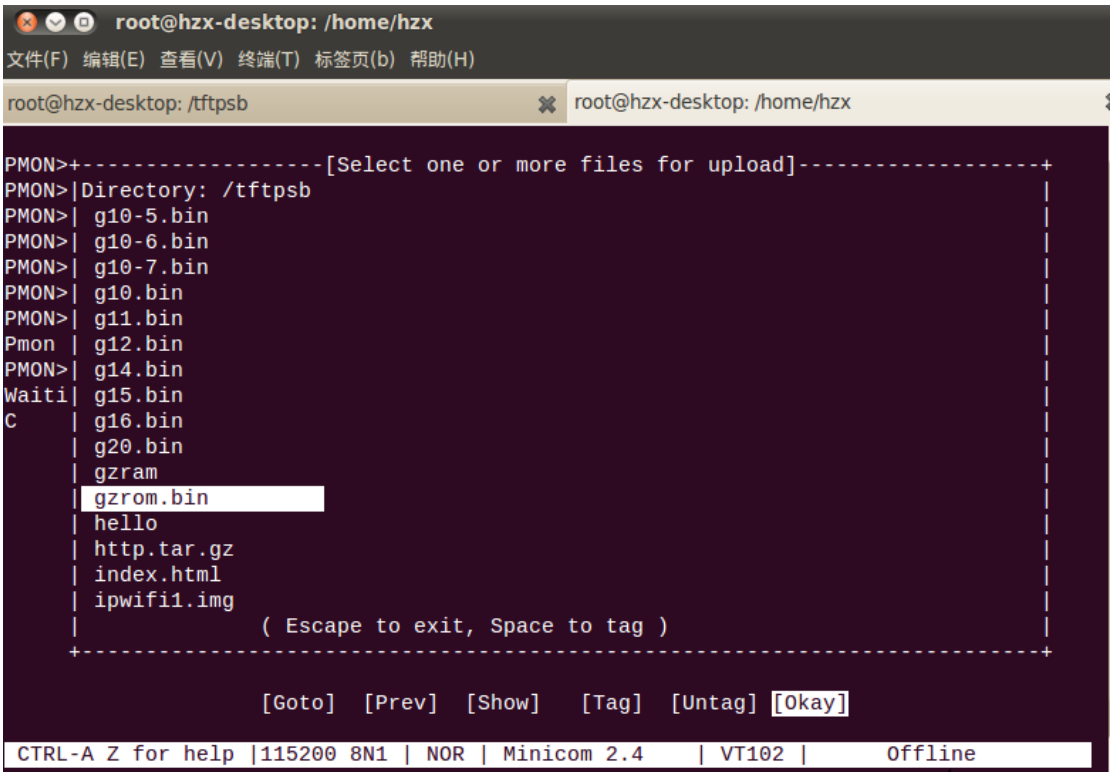
（3） 通过 ymodem 协议传输 PMON 二进制文件：

在 minicom 中，使用如下快捷键：先同时按下 Ctrl +A 键，松开后再按 S 键。弹出如下图选择框：




（4） 选择要传输的 PMON 二进制文件：

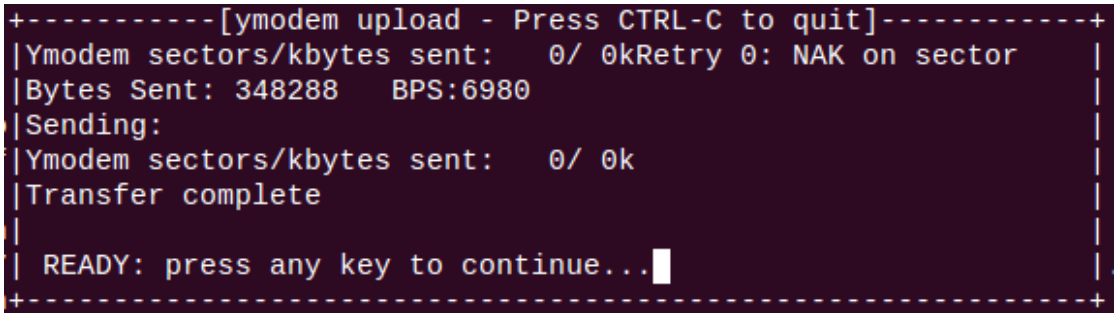
按回车键进入选择文件界面，双按空格键进入目录，单按空格键选择文件。




（5） 传输 PMON 二进制文件：

按回车键，传输文件。


传输完成，按任意键退出传输界面。




（6） 烧写到 NandFlash：

**PMON>load -r -f 0xbfc00000 /dev/ram/ymodem**

烧写完成后，PMON 更新成功，重启系统应用新的 PMON。

## 6.1.5 PMON 的内置命令

PMON 中内置了很多命令，下面举例部分命令：

| 类型 | 命令 | 说明 | 例子 | 例子含义 |
|---|---|---|---|---|
| 帮助 | h | 查看帮助信息 | h | 列出所有可以使用命令 |
| | | | h ping | 查看 ping 命令的用法 |
| 调试 | d1 | 读某个地址的值(读一个 byte) | d1 0x80300000 | 查看地址 0x80300000 处的值 |
| | d2 | 读某个地址的值(读一个 halt word) | d1 0x80300000 | 查看地址 0x80300000 处的值 |
| | d4 | 读某个地址的值(读一个 word) | d4 0x80300000 | 查看地址 0x80300000 处的值 |
| | m1 | 在某个地址处写入一个值(写入一个 byte 大小) | m1 0x80300000 0x12 | 在地址 0x80300000 处写入 0x12 |

| | | | | |
|---|---|---|---|---|
| | m2 | 在某个地址处写入一个值(写入一个 halt word 大小的值) | m2 0x80300000 0x1234 | 在地址 0x80300000 处写入 0x1234 |
| | m4 | 在某个地址处写入一个值(写入一个 word 大小的值) | m4 0x80300000 0x12345678 | 在地址 0x80300000 处写入 0x12345678<br>读出此地址值 PMON>d4 0x80300000；可以看到读出来的值为 0x12345678 |
| 内存 | mt | 内存测试命令 | mt | 测试板的内存是否正常 |
| | load | 下载 linux 内核到内存 | load<br>tftp://192.168.3.18/vmlinux | 通过网络从 IP 为 192.168.3.18 的主机上下载内核 vmlinx 到内存 |
| 测试外设 | ac97_read | 测试 ac97（录音） | ac97_read | 录音，有 5s 钟的时间录音；与 ac97_test 配合着测试 ac97 设备是否正常 |
| | ac97_test | 测试 ac97,放出刚才录进去的声音 | ac97_test | 播放刚才录进去的声音；和 ac97_read 配合着用 |
| 网络 | ifaddr | 设置板的 ip 地址(只当次有效，断电后会丢失) | ifaddr syn0 192.168.3.25 | 设置板的 ip 地址为 192.168.3.25 |
| | ping | 测试网络 | ping 192.168.3.1 | 测试与 192.168.3.1 网口是否连通 |
| 环境管理 | set | 设置环境变量；设置的参数会保存到 norflash 高位地址，在 pmon 一开始运行时就会自动去调用 | set | 列出所有已经设置好的环境变量 |
| | | | set ifconfig syn0:192.168.3.88 | 设置开发板的 IP 地址，重启开发板后 IP 地址固定存在 |
| | | | set al /dev/mtd0 | 自动从 nandflash 的 mtd0 分区 load 内存，设置板在一上电时自动执行 load 内核到内存操作 |
| | | | set append 'console=ttyS2' | 设置板的运行的启动参数 |
| | env | 查看板上已经设好的环境变量 | env | 列出所以环境变量 |

| | | | | |
|---|---|---|---|---|
| FLASH | devcp | PMON 上的拷贝下载，通常拷贝从网络下载的文件到 Nor FlashNand Flash 中 | devcp<br><br>tftp://192.168.3.18/vmlinux /dev/mtd0 | 从网络下载 vmlinux 到内存中并拷贝到 nandflash 中 |
| | mtd_erase | 擦除 Nand Flash 某分区的数据 | mtd_erase /dev/mtd1 | 擦除 Nand Flash 分区 1 的数据 |
| 系统管理 | reboot | 重启 PMON | reboot | 重启 PMON |
| 其他 | devls | 查看设备列表 | devls -n | 查看网络设备 |

提示：更多的，请在 PMON 中输入“h”，浏览帮助。

## 6.1.6 PMON 的启动设置

系统上电启动按空格键后即可进入 PMON 设置界面。在 PMON 的命令行上可以输入命令设置启动参数，参数被烧到 Flash 里面，重新启动后生效。

设置显示分辨率:

```
相应内核启动参数加上 video=ls1bfb:480x272-16@60 （以 480x272 分辨率为例）
如果使用 vga 接口的显示器，则启动参数为 video=ls1bfb:vga800x600-16@60 （以 800x600 分辨率为例），由于开发板没有 vga 接口，所以不使用该参数。
```

配置网卡:

```
ifconfig syn0 10.0.0.2
可以用 ping 命令测试网卡
ping 10.0.0.1
```

命令行设置从网卡启动:

```
ifconfig syn0 10.0.0.2
load tftp://10.0.0.1/vmlinux
g console=ttyS2,115200 rdinit=/sbin/init initcall_debug=1
```

命令行烧 nandflash（用于更新 PMON）:

```
ifconfig syn0 10.0.0.2
devcp tftp://10.0.0.1/gzom.bin /dev/mtd0
```

命令行从 nand 启动:

```
load /dev/mtd0
g console=ttyS2,115200 rdinit=/sbin/init initcall_debug=1
```

设置自动启动:
环境变量 ifconfig 用来每次启动的时候自动设置网卡地址
set ifconfig syn0:10.0.0.2:255.255.255.0

设置从不同介质启动内核（假设内核名称为 vmlinux）：

| | |
|---|---|
| set al /dev/fs/yaffs2@mtd1/boot/vmlinux | 从 yaffs2 分区里面的 boot 目录中的 vmlinux 来引导 |
| set al /dev/mtd0 | 从 nandflash 的第一个分区引导 |
| set al /dev/fs/ext2@usb0/boot/vmlinux | 如果从 usb 光盘引导 |
| set al tftp://10.0.0.3/vmlinux | 从 tftp 服务器引导 |
| Set al http://10.0.0.3/vmlinux | 从 http 引导 |
| Set al nfs://10.0.0.3/vmlinux | 从 nfs 引导 |
| set al /dev/ram@0xbe000000,0x1000000 | 从地址 0xbe000000 引导 |

设置内核启动参数：

| | |
|---|---|
| set append 'root=/dev/mtdblock2 console=tty' | 从 nand 的第二个分区作为根文件系统 |
| set append ‘ root=/dev/nfs nfsroot=192.168.1.1:/mnt/hdb1/nfs ip=192.168.1.89:::::eth0 console=tty’ | nfs 服务器 192.168.1.1 的/mnt/hdb1/nfs 作为根文件系统，网卡 eth0,ip 192.168.1.89 |
| set append ‘rdinit=/sbin/init console=tty’ | 内核里面自带的 ramdisk 作为系统 |

设置自动启动的延迟时间：

| | |
|---|---|
| set bootdelay 3 | 延迟时间 3 秒 |

设置 PMON 系统时间：

| | |
|---|---|
| set TZ +8 | 设置时区+8 区 |
| set date 200805011200.01 | 设置日期 2008-5-01 12:00:01 |

设置网卡：

| | |
|---|---|
| Set ethaddr 00:01:02:03:04:05 | 设置网卡 MAC 地址是 00:01:02:03:04:05 |
| set ifconfig syn0:10.0.0.89 | 设置 pmon 启动后网卡（syn0 代表 1b gmac 网络控制器,注：可以用 devls 列出 PMON 设备）ip 为 10.0.0.89 |

以上是一些通用的设置，具体到 1b 参考板的缺省参数设置是：

| | |
|---|---|
| set al /dev/mtd0 | 从 nand 加载内核 |
| set append ‘ root=/dev/mtdblock2 console=tty’ | 文件系统位于 nand 第二个分区 |
| set bootdelay 3 | 启动延迟 3 秒 |

## 6.2 烧写 Linux 内核

### 6.2.1 烧写内核

（1）设置 IP 地址并测试通过“参考 6-1-2 设置 IP 地址并测试”。
（2）把编译得到的 Linux 内核拷贝到服务目录“/tftpshare”。
（3）通过 TFTP 下载内核并烧到 Nand Flash：

```
PMON>devcp tftp://192.168.x.xxx/vmlinux /dev/mtd0
```

### 6.2.2 设置启动参数

```
PMON>set al /dev/mtd0
```

al(autoload)自动加载，即自动加载/dev/mtd0 设备的内容到内存区

## 6.3 烧写文件系统镜像

### 6.3.1 烧写文件系统镜像

（1）设置 IP 地址并测试通过“参考 6-1-2 设置 IP 地址并测试”。
（2）把制作得到的文件系统镜像文件拷贝到服务目录“/tftpshare”。
（3）通过 TFTP 下载文件系统镜像并烧到 Nand Flash：
烧写 cramfs 文件系统镜像：

```
PMON>devcp tftp://192.168.x.xxx/rootfs-cramfs.img /dev/mtd1
```

并且，烧写 jffs2 文件系统镜像：

```
PMON>devcp tftp://192.168.x.xxx/rootfs-jffs2.img /dev/mtd1
```

另外，烧写 yaffs2 文件系统镜像：

```
PMON>mtd_erase /dev/mtd1
PMON>devcp tftp://192.168.x.xxx/rootfs-yaffs2.img /dev/mtd1 yaf nw
```

### 6.3.2 设置启动参数

对于 cramfs 文件系统：

```
PMON>set append 'root=/dev/mtdblock1 console=ttyS2,115200 noinitrd init=/linuxrc rootfstype=cramfs video=ls1bfb:480x272-16@60'
```

或者 jffs2 文件系统：

```
PMON>set append 'root=/dev/mtdblock1 console=ttyS2,115200 noinitrd init=/linuxrc rw rootfstype=jffs2 video=ls1bfb:480x272-16@60'
```

或者 yaffs2 文件系统：

```
PMON>set append 'root=/dev/mtdblock1 console=ttyS2,115200 noinitrd init=/linuxrc rw rootfstype=yaffs2 video=ls1bfb:480x272-16@60'
```

最后，重启：

```
PMON>reboot
```

# 第七章 应用程序的移植

## 7.1 Hello World

（1）编辑源代码

```
#vi hello.c
```

代码清单如下：

```
#include<stdio.h>
int main(void)
{
    printf("Hello, world ！ \n");
    return 0;
}
```

（2）交叉编译 hello

使用以下命令编译：

```
#mipsel-linux-gcc –o hello hello.c
```

将生成 hello 可执行文件。

可以查看文件属性：

```
#file hello
```

```
root@loongson-desktop:/home/loongson# file hello
hello: ELF 32-bit LSB executable, MIPS, MIPS-II version 1 (SYSV), dynamically
 linked (uses shared libs), for GNU/Linux 2.4.0, not stripped
```

## 7.2 应用程序的移植方式

应用程序的移植方式目前主要有四种：

第一种：复制到介质（以 U 盘为例）

第二种：通过网络（tftp）传输文件到开发板（文件较小，推荐使用）

第三种：置于根文件系统目录下制作文件系统镜像，再烧进开发板（文件很大，可以使用）

第四种：通过 NFS（网络文件系统）直接运行

### 7.2.1 复制到介质（以 U 盘为例）

把 U 盘插入 PC 的 USB，然后执行以下命令把 hello 复制到 U 盘

```
$fdisk -l
$mount /dev/sdb1 /mnt
$cp hello /mnt
$umount /mnt
```

把 U 盘取出再插入到目标板的 USB HOST 端口，按照以下命令操作：

```
$fdisk -l
$mount /dev/sda1 /mnt                    //挂接 U 盘
$cp /mnt/hello /bin                      //把 hello 复制到 bin 目录
$./hello                                 //执行 hello
```

### 7.2.2 通过网络（tftp）传输文件到开发板

通过网络下载程序的主要步骤是：通过 tftp 协议通过网络将远程主机上的文件下载到目标板上，并修改执行权限，如下：

在 PC 端执行：

打开 tftp 服务（“4-2-4 安装 tftp”，安装完成后默认打开）。

改变服务共享目录/tftpshare 的权限：

```
#chmod 777 /tftpshare
```

并把 hello 放到 tftp 的共享目录下：

```
#cp hello /tftpshare
```

测试网络连同是否完好 ：

#ping tftp 服务器端的 IP 地址

```
$ping 192.168.0.244
```

通过 tftp 协议登陆 远程主机并下载 hello 到本地目标板 ：

#tftp –r 要下载文件名 -g TFTP 服务器端的 IP 地址

```
$tftp -r hello -g 192.168.0.244
```

传输完毕后，更改 hello 的可执行权限 ：

```
$chmod u+x hello
```

执行 hello：

```
$./hello
```

### 7.2.3 置于根文件系统目录下制作文件系统镜像

详解参考“5-3-3 制作文件系统镜像文件”。

### 7.2.4 通过 NFS（网络文件系统）直接运行

详解参考“附录 8　NFS 网络文件系统搭建”。

## 7.3 启动脚本

想要应用程序在启动进入文件系统后，能自动地配置好相应的环境或能灵活地操作，需要相应的启动脚本。

在应用程序相同目录下创建 SHELL 脚本文件：

```
vi hello.sh
```

内容为：

```
./hello&                                    //在后台运行 hello
```

运行启动脚本：

在“附录 4 制作根文件系统——步骤 6 创建系统配置文件”的“etc/profile”最后一行顶格添加“sh hello.sh 的绝对路径”。

```
sh /hello.sh
```

这样在启动开发板后，系统将会自动地后台运行 hello。

# 第八章 应用开发实验

## 8.1 LINUX 基础实验

### 8.1.1 实验一 shell 编程

```
# vi show.sh
```

内容为：

```
#!/bin/bash
i=0
    while [ $i -lt 10 ]
```

```
do
    for j in '-' '\\' '|' '/'
    do
        echo -ne "\033[1D$j"
        sleep 1
    done
        ((i++))
done
```

### 8.1.2 实验二 文件操作实验

源码：

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

#define BUFFER_SIZE 1024

int main(int argc,char **argv)
{

    int from_fd,to_fd;
    int bytes_read,bytes_write;
    char buffer[BUFFER_SIZE];
    char *ptr;

    if(argc!=3)
    {
        fprintf(stderr,"Usage:%s fromfile tofile\n\a",argv[0]);
        exit(1);
    }

    if((from_fd=open(argv[1],O_RDONLY))==-1)
    {
        fprintf(stderr,"Open %s Error:%s\n",argv[1],strerror(errno));
        exit(1);
```

```
    }

    if((to_fd=open(argv[2],O_WRONLY|O_CREAT,S_IRUSR|S_IWUSR))==-1)
    {
            fprintf(stderr,"Open %s Error:%s\n",argv[2],strerror(errno));
            exit(1);
    }

    while(bytes_read=read(from_fd,buffer,BUFFER_SIZE))
    {
        if((bytes_read==-1)&&(errno!=EINTR)) break;
        else if(bytes_read>0)
                {
            ptr=buffer;
            while(bytes_write=write(to_fd,ptr,bytes_read))
            {
                    if((bytes_write==-1)&&(errno!=EINTR))break;
                    else if(bytes_write==bytes_read) break;
                    else if(bytes_write>0)
                    {
                    ptr+=bytes_write;
                    bytes_read-=bytes_write;
                    }
                    }
                if(bytes_write==-1)break;

                }
    }

    close(from_fd);
    close(to_fd);
    exit(0);
}
```

### 8.1.3 实验三 多线程实验

源码：

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
```

```
void *thread_function(void *arg);

int main(int argc,char **argv)
{
    int res;
    pthread_t a_thread;
    void *thread_result;
    int i;

    res = pthread_create(&a_thread,NULL,thread_function,NULL);
    if (res != 0)
    {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }

    for(i=0; i<5; i++)
    {
        printf("this is main ......\n");
        sleep(1);
    }

    printf("Cancelling thread...\n");
    res = pthread_cancel(a_thread);
    if (res != 0)
    {
        perror("thread cancelation failed");
        exit(EXIT_FAILURE);
    }

    printf("waiting for thread to finish...\n");

#if 1
    res = pthread_join(a_thread,&thread_result);
    if (res != 0)
    {
        perror("thread join failed");
        exit(EXIT_FAILURE);
    }
#endif
    printf("the progam is end\n");
```

```
    return 0;
}

void *thread_function(void *arg)
{
    int i,res,j;
    printf("it is thread \n");
    //sleep(1);

    res = pthread_setcancelstate(PTHREAD_CANCEL_DISABLE,NULL);
    if (res != 0)
    {
        perror("thread pthread_setcancelstate failed");
        exit(EXIT_FAILURE);
    }

    sleep(2);

    printf("thread cancel type is disable,can't cancel this thread\n");
    for (i = 0;i < 3;i ++)
    {
        printf("thread is running (%d)...\n",i);
        sleep(1);
    }
    printf("thread is running (%d)...\n",i);
    res = pthread_setcancelstate(PTHREAD_CANCEL_ENABLE,NULL);
#if 1
    if (res != 0)
    {
        perror("thread pthread_setcancelstate failed");
        exit(EXIT_FAILURE);
    }
#endif
//  sleep(5);
    printf("thread is change to cancel enable\n");
    printf("thread is change \n");
    sleep(100);
    pthread_exit(0);

}
```

### 8.1.4 实验四 多进程实验

源码：

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>

int main(int argc,char ** argv)
{
    pid_t pid;
    int res;

    if ((pid = fork()) < 0)
        printf("error");
    else if (pid == 0)
    {
        printf("this child \n");
        res = execl("/bin/ls","ls","-l","./",(char *)0);
        if (res == -1)
        {
            perror("execl");
            exit(EXIT_FAILURE);
        }
        exit(1);
    }
    else
    {
        sleep(3);
        waitpid(pid,NULL);
        printf("father ok!\n");
    }

    return 0;
}
```

### 8.1.5 实验五 进程间通信实验

接收端：

```
#include <stdio.h>
```

```
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>

int main(int argc,char ** argv)
{
    int running = 1;
    char *shm_p = NULL;
    int shmid;
    int semid;
    int value;
    struct sembuf sem_b;
    sem_b.sem_num = 0;
    sem_b.sem_flg = SEM_UNDO;

    if ((semid = semget((key_t)123456,1,0666 | IPC_CREAT)) == -1)
    {
        perror("semget");
        exit(EXIT_FAILURE);
    }
    shmid = shmget((key_t)654321,(size_t)2048,0600 | IPC_CREAT);
    if (shmid == -1)
    {
        perror("shmget");
        exit(EXIT_FAILURE);
    }
    shm_p = shmat(shmid,NULL,0);
    if (shm_p == NULL)
    {
        perror("shmat");
        exit(EXIT_FAILURE);
    }
    while (running)
    {
        if ((value = semctl(semid,0,GETVAL)) == 1)
        {
            printf("read data operate\n");
            sem_b.sem_op = -1;
            if (semop(semid,&sem_b,1) == -1)
```

```
            {
                fprintf(stderr,"semaphore_p failed\n");
                exit(EXIT_FAILURE);
            }
            printf("%s\n",shm_p);
        }
        if (strcmp(shm_p,"end") == 0)
            running--;
    }
    shmdt(shm_p);
    if (shmctl(shmid,IPC_RMID,0) != 0)
    {
        perror("shmctl");
        exit(EXIT_FAILURE);
    }
    if (semctl(semid,0,IPC_RMID,0) != 0)
    {
        perror("semctl");
        exit(EXIT_FAILURE);
    }

    return 0;
}
```

发送端：

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/shm.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <string.h>

int main(int argc,char ** argv)
{
    int running = 1;
    int shid;
    int semid;
    int value;
    void * sharem = NULL;
    struct sembuf sem_b;
```

```
sem_b.sem_num = 0;
sem_b.sem_flg = SEM_UNDO;

if ((semid = semget((key_t)123456,1,0666|IPC_CREAT)) == -1)
{
    perror("semget");
    exit(EXIT_FAILURE);
}
if (semctl(semid,0,SETVAL,0) == -1)
{
    printf("sem init error");
    if (semctl(semid,0,IPC_RMID,0) != 0)
    {
        perror("semctl");
        exit(EXIT_FAILURE);
    }
    exit(EXIT_FAILURE);
}
shid = shmget((key_t)654321,(size_t)2048,0600|IPC_CREAT);
if (shid == -1)
{
    perror("shmget");
    exit(EXIT_FAILURE);
}
sharem = shmat(shid,NULL,0);
if (sharem == NULL)
{
    perror("sharem");
    exit(EXIT_FAILURE);
}
while (running)
{
    if ((value = semctl(semid,0,GETVAL)) == 0)
    {
        printf("write data opreate\n");
        printf("please input something:");
        scanf("%s",sharem);
        printf("you input is :%s\n",sharem);
        sem_b.sem_op = 1;
        if (semop(semid,&sem_b,1) == -1)
        {
            fprintf(stderr,"semaphore_p failed\n");
            exit(EXIT_FAILURE);
```

```
                }
            }
            if (strcmp(sharem,"end") == 0)
                running--;
        }
        shmdt(sharem);

        return 0;
}
```

可以把其中一端作为后台程序（添加“&”）。

### 8.1.6 实验六 网络编程实验

服务端：

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define MYPORT 1234        // the port users will be connecting to

#define BACKLOG 5          // how many pending connections queue will hold

#define BUF_SIZE 200

int fd_A[BACKLOG];        // accepted connection fd
int conn_amount;         // current connection amount

void showclient()
{
        int i;
        printf("client amount: %d\n", conn_amount);
        for (i = 0; i < BACKLOG; i++) {
                printf("[%d]:%d    ", i, fd_A[i]);
        }
        printf("\n\n");
```

```
}

int main(void)
{
    int sock_fd, new_fd;    // listen on sock_fd, new connection on new_fd
    struct sockaddr_in server_addr;       // server address information
    struct sockaddr_in client_addr; // connector's address information
    socklen_t sin_size;
    int yes = 1;
    char buf[BUF_SIZE];
    int ret;
    int i;

    if ((sock_fd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }

    if (setsockopt(sock_fd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) == -1) {
        perror("setsockopt");
        exit(1);
    }

    server_addr.sin_family = AF_INET;                // host byte order
    server_addr.sin_port = htons(MYPORT);         // short, network byte order
    server_addr.sin_addr.s_addr = INADDR_ANY; // automatically fill with my IP
    memset(server_addr.sin_zero, '\0', sizeof(server_addr.sin_zero));

    if (bind(sock_fd, (struct sockaddr *)&server_addr, sizeof(server_addr)) == -1) {
        perror("bind");
        exit(1);
    }

    if (listen(sock_fd, BACKLOG) == -1) {
        perror("listen");
        exit(1);
    }

    printf("listen port %d\n", MYPORT);

    fd_set fdsr;
    int maxsock;
    struct timeval tv;
```

```
conn_amount = 0;
sin_size = sizeof(client_addr);
maxsock = sock_fd;
while (1) {
    // initialize file descriptor set
    FD_ZERO(&fdsr);
    FD_SET(sock_fd, &fdsr);

    // timeout setting
    tv.tv_sec = 30;
    tv.tv_usec = 0;

    // add active connection to fd set
    for (i = 0; i < BACKLOG; i++) {
        if (fd_A[i] != 0) {
            FD_SET(fd_A[i], &fdsr);
        }
    }

    ret = select(maxsock + 1, &fdsr, NULL, NULL, &tv);
    if (ret < 0) {
        perror("select");
        break;
    } else if (ret == 0) {
        printf("timeout\n");
        continue;
    }

    // check every fd in the set
    for (i = 0; i < conn_amount; i++) {
        if (FD_ISSET(fd_A[i], &fdsr)) {
            ret = recv(fd_A[i], buf, sizeof(buf), 0);
            if (ret <= 0) {             // client close
                printf("client[%d] close\n", i);
                close(fd_A[i]);
                FD_CLR(fd_A[i], &fdsr);
                fd_A[i] = 0;
            } else {                    // receive data
                if (ret < BUF_SIZE)
                    memset(&buf[ret], '\0', 1);
                printf("client[%d] send:%s\n", i, buf);
            }
```

```
                }
            }

            // check whether a new connection comes
            if (FD_ISSET(sock_fd, &fdsr)) {
                new_fd = accept(sock_fd, (struct sockaddr *)&client_addr, &sin_size);
                if (new_fd <= 0) {
                    perror("accept");
                    continue;
                }

                // add to fd queue
                if (conn_amount < BACKLOG) {
                    fd_A[conn_amount++] = new_fd;
                    printf("new connection client[%d] %s:%d\n", conn_amount,
                            inet_ntoa(client_addr.sin_addr), ntohs(client_addr.sin_port));
                    if (new_fd > maxsock)
                        maxsock = new_fd;
                }
                else {
                    printf("max connections arrive, exit\n");
                    send(new_fd, "bye", 4, 0);
                    close(new_fd);
                    break;
                }
            }
            showclient();
        }

        // close other connections
        for (i = 0; i < BACKLOG; i++) {
            if (fd_A[i] != 0) {
                close(fd_A[i]);
            }
        }

        exit(0);
}
```

客户端：

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
```

```
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define SERVERIP "192.168.1.100"
#define SERVERPORT 1234
#define MAXDATASIZE 256
#define STDIN 0

int main(void)
{
    int sockfd;
    int recvbytes;
    char buf[MAXDATASIZE];
    char send_str[MAXDATASIZE];

    struct sockaddr_in serv_addr;
    fd_set rfd_set, wfd_set, efd_set;

    struct timeval timeout;
    int ret;

    if( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1 )
    {
            perror("socket");
            exit(1);
        }

    bzero(&serv_addr, sizeof(struct sockaddr_in));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(SERVERPORT);
    inet_aton(SERVERIP, &serv_addr.sin_addr);

    if(connect(sockfd, (struct sockaddr *)&serv_addr, sizeof(struct sockaddr)) == -1 )
    {
        perror("connect");
        exit(1);
    }

    //fcntl(sockfd, F_SETFD, O_NONBLOCK);
```

```
printf("");

while(1)
{
    FD_ZERO(&rfd_set);
    FD_ZERO(&wfd_set);
    FD_ZERO(&efd_set);

    FD_SET(STDIN, &rfd_set);
    FD_SET(sockfd, &rfd_set);
    FD_SET(sockfd, &efd_set);

    timeout.tv_sec = 30;
    timeout.tv_usec = 0;
    ret = select( sockfd+1, &rfd_set, &wfd_set, &efd_set, &timeout);

    if(ret == 0)
    {
        continue;
    }

    if(ret < 0)
    {
        perror("select error:");
    }

    if(FD_ISSET(STDIN, &rfd_set))
    {
        fgets(send_str, 256, stdin);
        send_str[strlen(send_str)-1] = '\0';
        if( strncmp("quit", send_str, 4) == 0 )
        {
            close(sockfd);
            exit(0);
        }
        send(sockfd, send_str, strlen(send_str), 0);
    }

    if(FD_ISSET(sockfd, &rfd_set))
    {
        recvbytes = recv(sockfd, buf, MAXDATASIZE, 0);
        if(recvbytes == 0)
        {
```

```
                close(sockfd);
                exit(0);
            }
            buf[recvbytes] = '\0';
            printf("Server: %s\n", buf);
            fflush(stdout);

        }

        if(FD_ISSET(sockfd, &efd_set))
        {
            printf("efd_set\n");
            close(sockfd);
            exit(0);
        }
    }
}
```

可以把其中一端作为后台程序（添加“&”）。

## 8.2 1B 开发板外设测试实验

### 8.2.1 AD 转换

| | |
|---|---|
| 驱动源程序所在目录 | driver/spi/ |
| 驱动程序名 | mcp3201.c |
| 设备名 | /dev/mcp3201 |
| 测试程序源代码目录 | Examples/Drivers/ADC |
| 测试程序代码名 | test-mcp3201.c |
| 测试程序可执行文件名 | test-mcp3201 |

test-mcp3201 是基于控制台下的 adc 控制器，使用方法也很简单，只需要不停的从 mcp3201 的输出引脚读取采样到的数据就可以了。该命令将驱动 mcp3201 工作，不断的读取采样到的数据。

```
$./test-mcp3201
Starting mcp3201
The value is 0x0 0x0
The value is 0x0 0x0
The value is 0x0 0x0
```

### 8.2.2 PWM

| | |
|---|---|
| 驱动源程序所在目录 | driver/char/ |
| 驱动程序名 | ls1b-pwm.c |
| 设备名 | /dev/ls1f-pwm |
| 测试程序源代码目录 | Examples/Drivers/PWM |
| 测试程序代码名 | test-pwm.c |
| 测试程序可执行文件名 | test-pwm |

test-pwm 是基于控制台下的 pwm 控制器，使用方法也很简单，只需要在测试程序中设置好我们需要的占空比，然后通过 itocl 指令选择使用哪一路 pwm 进行工作就可以了。
测试结果如下：

```
$./test-pwm
cys: ready to open!
open ok!
Input the selected pwm number: 1
Resource start=0x1fe5c010, end = 0x1fe5c01f
Input the selected pwm number: 2
Resource start=0x1fe5c020, end = 0x1fe5c02f
Trying to free nonexistent resource <000000001fe5c020-000000001fe5c02e>
Input the selected pwm number: 3
Resource start=0x1fe5c030, end = 0x1fe5c03f
Trying to free nonexistent resource <000000001fe5c030-000000001fe5c03e>
Input the selected pwm number: 4
```

### 8.2.3 蜂鸣器

| | |
|---|---|
| 驱动源程序所在目录 | driver/char/ |
| 驱动程序名 | buzzer.c |
| 设备名 | buzzer_gpio |
| 测试程序源代码目录 | Examples/Drivers/buzzer |
| 测试程序代码名 | buzzer_test.c |
| 测试程序可执行文件名 | buzzer |
| 说明：蜂鸣器驱动已经被编译到缺省内核中，因此不用使用 insmod 方式加载 | |

开发板用到的资源：

| |
|---|
| gpio3 |

在启动 linux 内核后，到测试目录 Driver_Test 目录下执行当前目录下的可执行文件 buzzer，则蜂鸣器开始鸣叫。

```
$ ./buzzer
```

```
prot is 3, value is 1
```

### 8.2.4 按键

程序源代码说明

| | |
|---|---|
| 驱动源程序所在目录 | drivers/input/keyboard |
| 驱动程序名 | 74LV165_button.c<br>74LV165_button.h |
| 设备名 | ls1b_buttons |
| 测试程序源代码目录 | Examples/Drivers/button |
| 测试程序代码名 | 74LV165_test.c |
| 测试程序可执行文件名 | 74LV165_test |
| 说明：按键驱动已经被编译到缺省内核中，因此不用使用 insmod 方式加载 | |

开发板用到的资源：

| | |
|---|---|
| gpio0 | KEY_DATA |
| gpio1 | KEY_EN |
| gpio2 | KEY_SCL |

按键驱动在硬件上没有接外部中断，且通过两片并行输入串行输出的移位芯片 74LV165 连接，因此，按键通过轮询的方式实现。应用程序直接读取/dev/ls1b_buttons 设备节点来对按键值进行读取。在启动 linux 内核后，到测试目录 Driver_Test 目录下执行当前目录下的可执行文件 74LV165_test。

```
$ ./74LV165_test
Welcome to use 74LV165 driver
init 74LV165 is done
74LV165_CFG value is 0x000F
74LV165_OE value is 0x0001
74LV165_PL value is 0x0000
74LV165_CLK value is 0x0000
```

按下相应的按键后打印相应的按键编码：

```
BReadBuf value is 0x0000ffff
data is 0xFFFF
```

其中第一行为内核打印的按键编码，第二行为用户程序读到的按键编码。

## 8.2.5 SD 卡

（1）进入 linux 系统后，在终端 minicom 中输入命令 cd /dev 进入 dev 目录，使用 ls -l 命令看是否存在 SD 卡的设备文件节点/dev/mmc0。

```
$ ls -l /dev/
crw-r--r--    1 1000     1000      253,     0 Nov 28    2009 74HC165_button
crw-r--r--    1 1000     1000        5,     1 Nov 28    2009 console
lrwxrwxrwx      1 1000       1000                9 Nov 28    2009 dsp -> /dev/dsp0
crw-r--r--    1 1000     1000       14,     3 Nov 28    2009 dsp0
crw-r--r--    1 1000     1000       14,    19 Nov 28    2009 dsp1
crw-r--r--    1 1000     1000       14,    35 Nov 28    2009 dsp2
crw-r--r--    1 1000     1000       14,    51 Nov 28    2009 dsp3
crw-r--r--    1 1000     1000       29,     0 Nov 28    2009 fb
crw-r--r--    1 1000     1000       29,     0 Nov 28    2009 fb0
crw-r--r--    1 1000     1000       29,     1 Nov 28    2009 fb1
crw-r--r--    1 1000     1000       29,     2 Nov 28    2009 fb2
brw-r--r--    1 1000     1000      249,     0 Nov 28    2009 mmc0
```

没有则创建一个：

```
$ mknod /dev/mmc0 b 249 0
```

（2）创建 SD 卡临时挂载目录：

```
$ mkdir /mnt/sd
$ ls /mnt/
  sd
```

（3）挂载 SD 卡文件系统：即把 SD 的设备节点挂载到/mnt/sd 目录下，这样在此目录下读写文件了。

```
$ mount -t vfat /dev/mmc0 /mnt/sd/
-->fs/namespace.c:sys_mount:1557
-->fs/namespace.c:do_mount:1383
-->dev:/dev/mmc0,    dir:/mnt/sd, type:vfat, flags:0xc0ed0000.
$ ls /mnt/sd/
1.bmp          14.bmp         6.bmp          pic2.bmp       sys10.bmp      sys5.bmp
10.bmp         2.bmp          7.bmp          pic3.bmp       sys11.bmp      sys6.bmp
11.bmp         3.bmp          8.bmp          record.dat     sys12.bmp      sys7.bmp
12.bmp         4.bmp          9.bmp          st1616.bin     sys2.bmp       sys8.bmp
13.bmp         5.bmp          pic1.bmp       sys1.bmp       sys3.bmp       sys9.bmp
```

现在就可以像操作 U 盘一样操作 SD 卡了。

### 8.2.6 U 盘

（1）进入 linux 系统后，在终端 minicom 中输入命令 fdisk

```
$ fdisk -l
Disk /dev/mtdblock0: 0 MB, 524288 bytes
255 heads, 63 sectors/track, 0 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
Disk /dev/mtdblock0 doesn't contain a valid partition table
Disk /dev/sda: 4023 MB, 4023385600 bytes
255 heads, 63 sectors/track, 489 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
Device Boot      Start         End      Blocks   Id System
/dev/sda1   *           1         490     3929056 b Win95 FAT32
Partition 1 has different physical/logical endings:
phys=(488, 254, 63) logical=(489, 37, 59)
```

可以看到 U 盘被识别为设备节点 /dev/sda1。

（2）再执行 U 盘挂载命令 / # mount –t vfat /dev/sda1 /mnt，即把该设备节点/dev/sda1 挂载到/mnt 目录下，就可以看到 U 盘内容；执行 U 盘卸载命令，/ # umount /mnt，就可以安全卸载 U 盘。

```
$ mount –t vfat /dev/sda1 /mnt
$ ls /mnt/
 malloc          ram12.tgz       ucos            arm-gcc
led_key.pdf     mplayer         ramdis~1.gz     usb-ba~1
ls1b_dev.pdf    mplaye~1        source~1        nbench
$ umount /mnt
$ ls /mnt
```

### 8.2.7 音频

（1）把 Examples/Drivers/ac97-test 目录拷贝到根文件系统。

（2）进入 linux 系统后，在终端 minicom 中输入命令 ls，可以看到 ac97-test 这个音频测试目录。

```
$ ls
ac97-test     etc     lost+found    proc     tmp
bin           fbtest1       memtest.sh    root          tsh.sh
captrue       init          memtester     sbin          ubench
dev           lib           mnt           sys           usr
env.sh        linuxrc       opt           test_uvc      var
```

（3）进入 ac97-test 目录，并查看目录内容为。

```
$ cd ac97-test/
$ ls
 ac97_test.sh    mp3player        mplayer          uvcvideo.ko     wait.mp3
```

（4）打开当前目录下 mplayer 这播放器，播放自己所指定的音频资料。

```
$ ./mplayer    wait.mp3
```

### 8.2.8 网卡

（1）确认网口的 LED 绿灯亮，说明网络是连通的

（2）在调试终端里设置网络 ip 地址：

```
$ ifconfig eth0 192.168.1.100
```

其中 eth0 是该网口的名称，如果设置另一个网口 IP 地址，那么就根据该网口的实际名称来设置。设置好了用 ping 命令来测试网络数据是否正常传输：

```
$ ping 192.168.1.100
```

其中 192.168.1.100 是在同一网段的电脑的 ip 地址，可以 ping 开发机。看 time 的值，越小网速越快。结束 ping 操作后，看提示信息，有没有发生网络丢包。

### 8.2.9 RTC 时钟

设置并保存系统实时时钟。

Linux 中更改时间的方法一般使用 date 命令，为了把 lonngs 1B 内部带的时钟与 linux 系统时钟同步，一般使用 hwclock 命令，下面是它们的使用方法：

（1）设置时间 2011-08-24 14:59

```
$ date -s 082414592011
```

（2）把刚刚设置的时间存入 loongson 1B 内部的 RTC

```
$ hwclock   –w
```

（3）开机时使用 hwclock -s 命令可以恢复 linux 系统时钟为 RTC，一般把该语句放入/etc/init.d/rcS 文件自动执行。

```
$ hwclock   –s
```

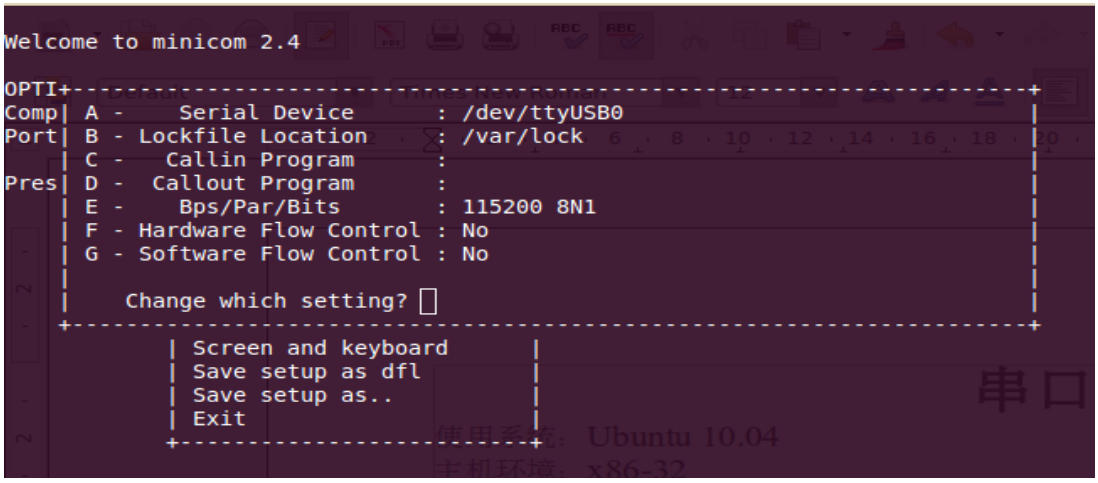
（4）读取 RTC 的时间

```
$ hwclock
```

注意：hwclock 命令需要读取/dev/rtc 设备节点，lonngs 1B 的 RTC 驱动注册后会在/dev 目录下建立/dev/rtc0 节点，可以使用命令 ln -s rtc0 rtc 建立一个 rtc 的设备节点，这样 hwclock 命令就可以读取 RTC 的时间。

### 8.2.10 串口

（1）在 Linux 系统中打开终端(gnome-terminal)，在命令行中输入“minicom -s”（如果系统没安装的话，请安装 apt-get install minicom）。主机的 minicom 设置如下：




（2）把串口线一端连接到开发板上，另一端连接到主机上。打开烧好程序的开发板电源开关，等进入系统之后命令行之后运行“minicom ttyS1 -s 115200 ”，就可以与主机进行相互通信。其他几个串口也类似(ttyS0、ttyS2、ttyS3 等等)。

## 8.3 LINUX GUI 实验

### 8.3.1 实验一 QT3

**(1). 安装 qt3 与 designer**

使用 apt-get 命令安装

```
apt-get install libqt3-mt
apt-get install qt3-designer
```

**(2). 使用 QT 编程开发**

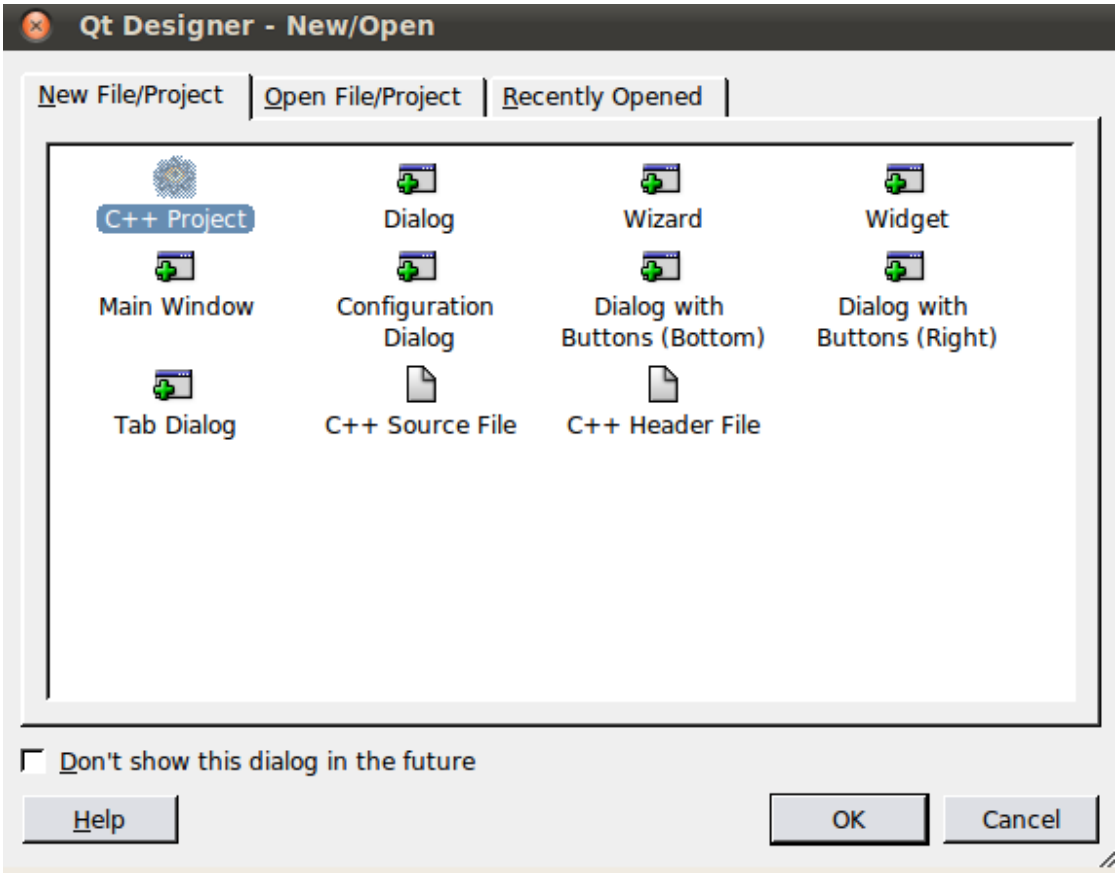
a）建立项目工作目录

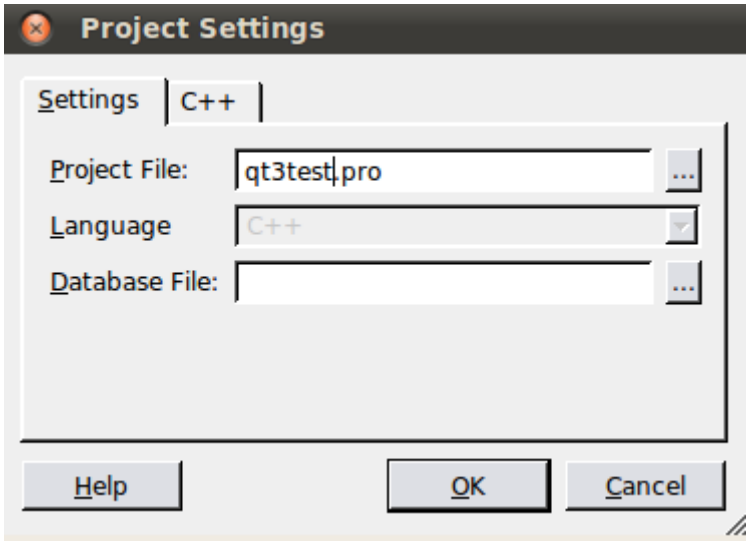
```
mkdir /root/qt3test
```

b）使用 designer 工具，建立项目的界面布局

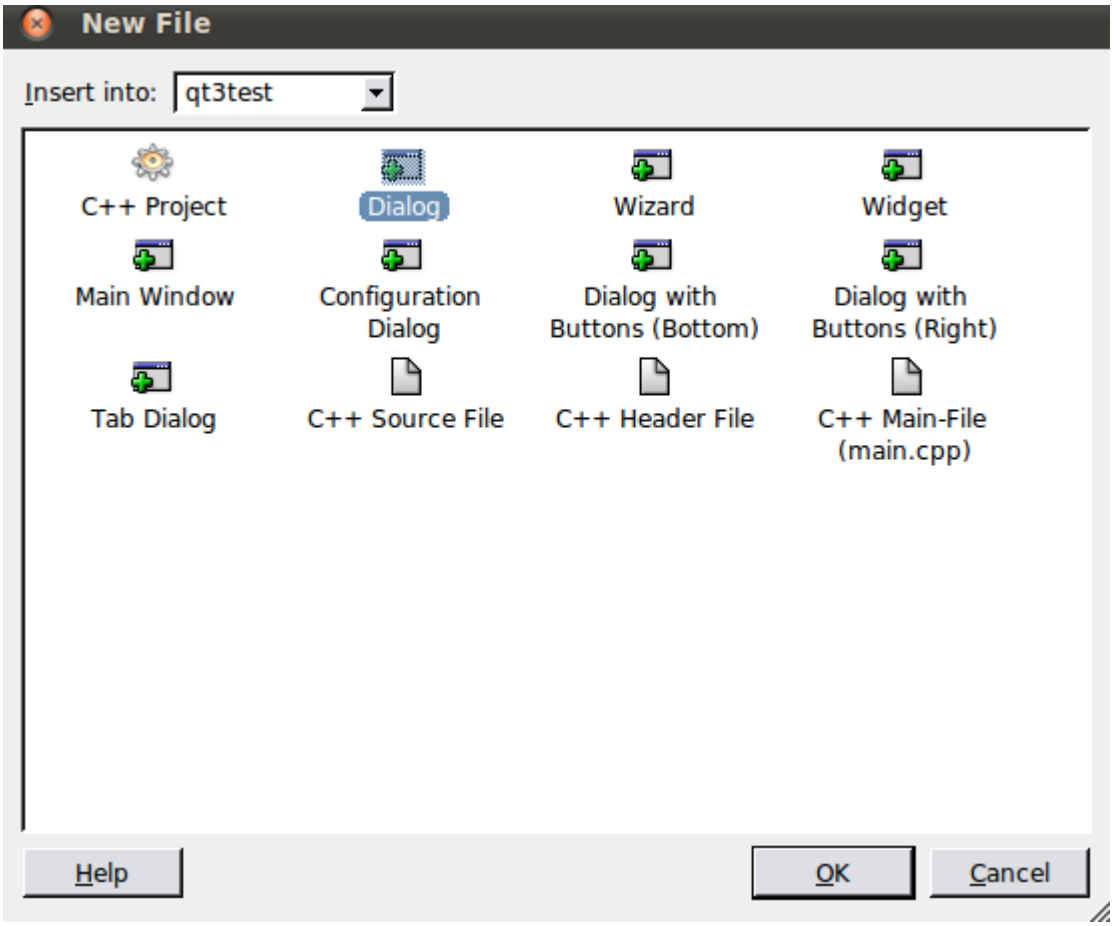
```
cd /root/qt3test
designer
```

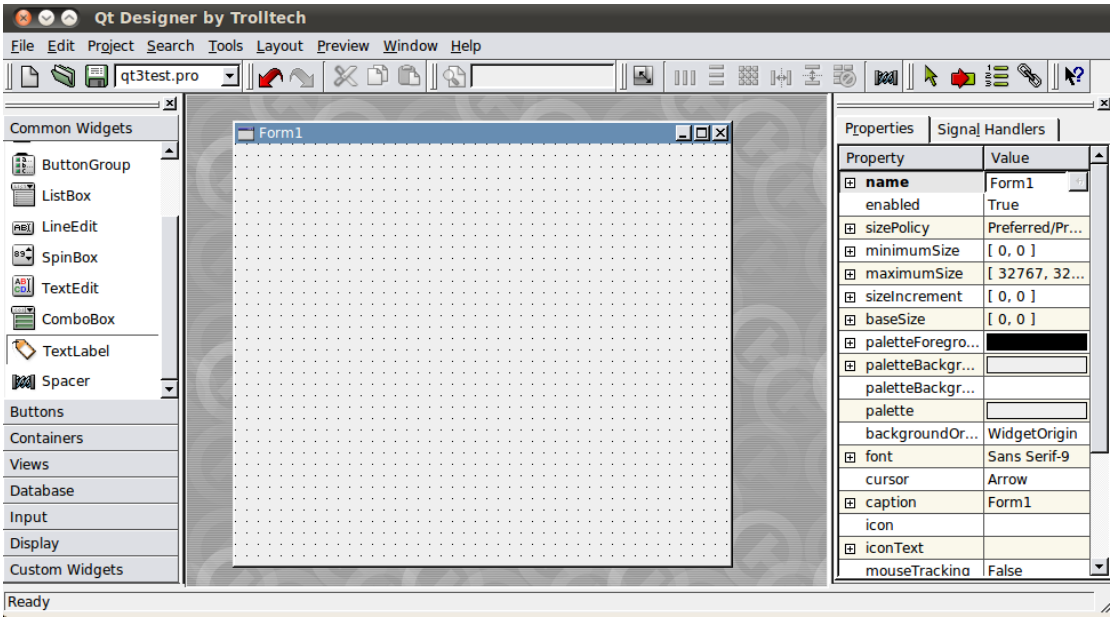
先新建项目工程（名字与工作目录名一样）：



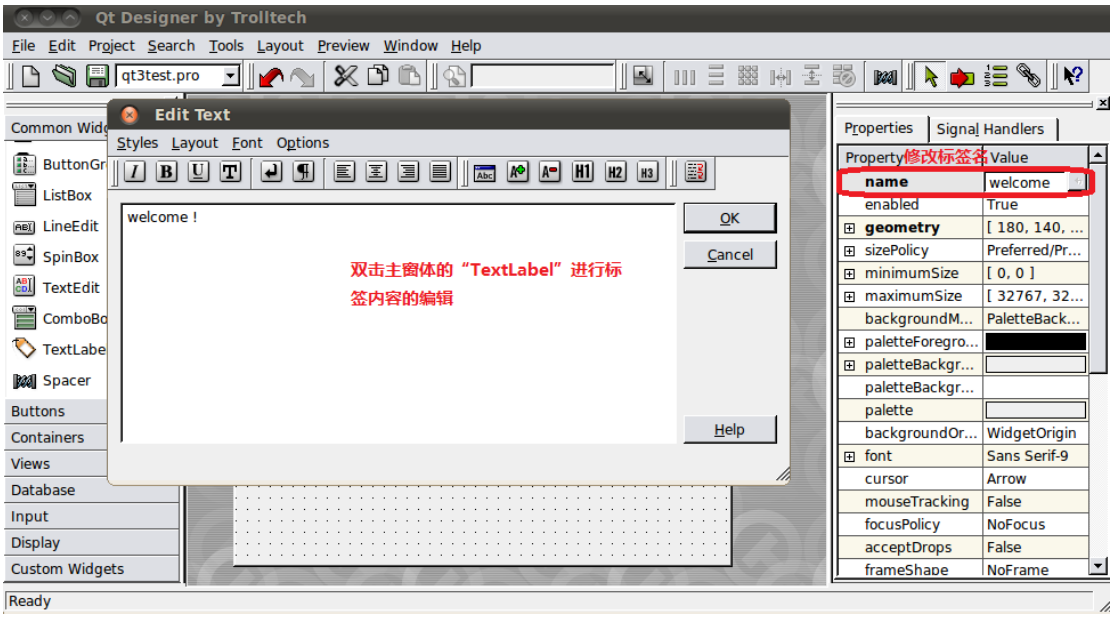




建立主界面窗体：

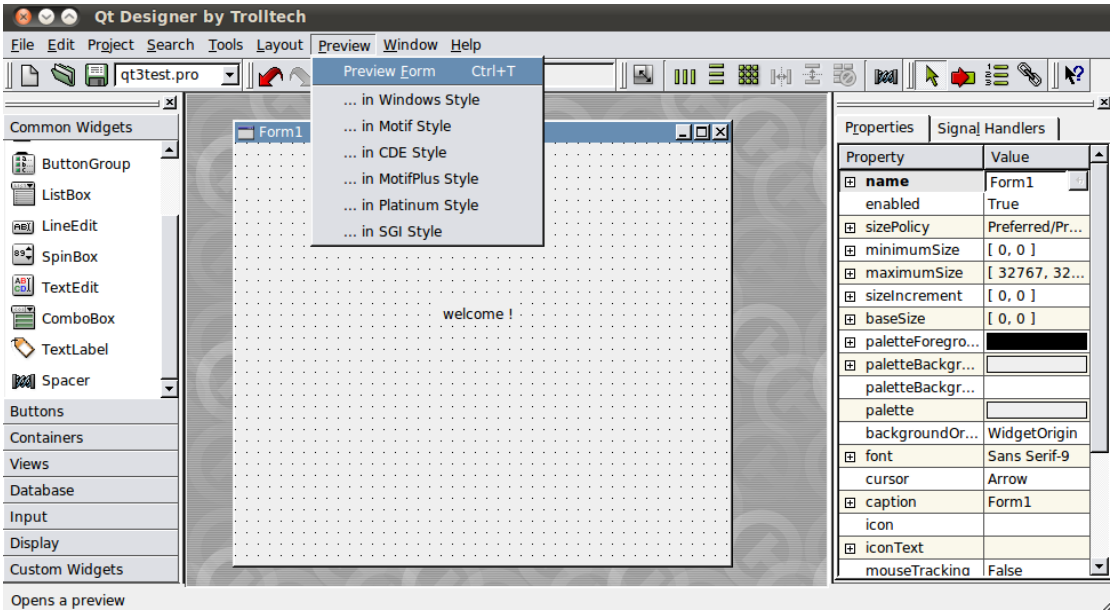


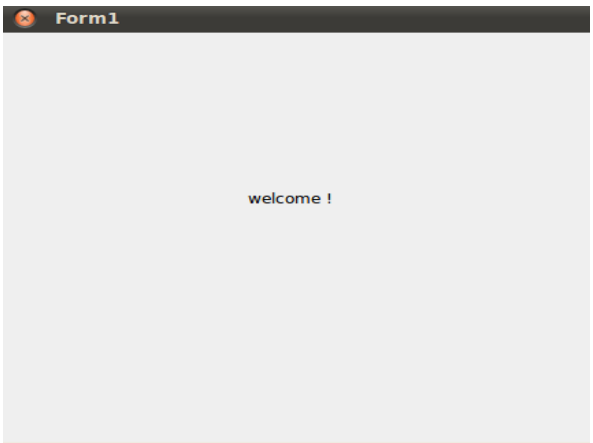




使用标签“TextLabel”在主界面窗体上标签“welcome !”:

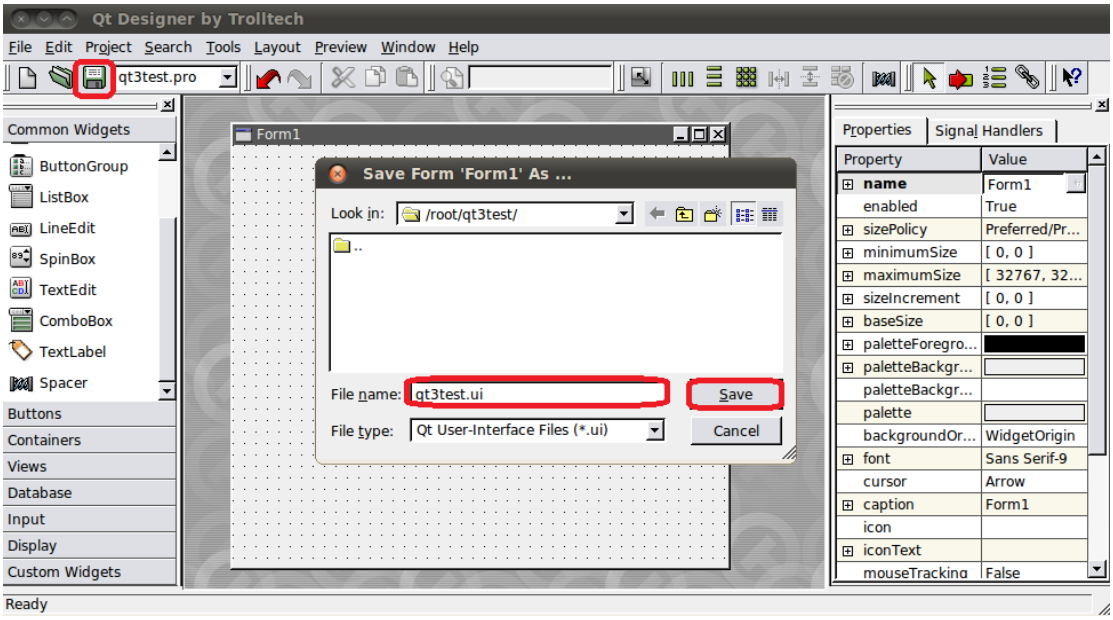



完成项目后可以预览演示：

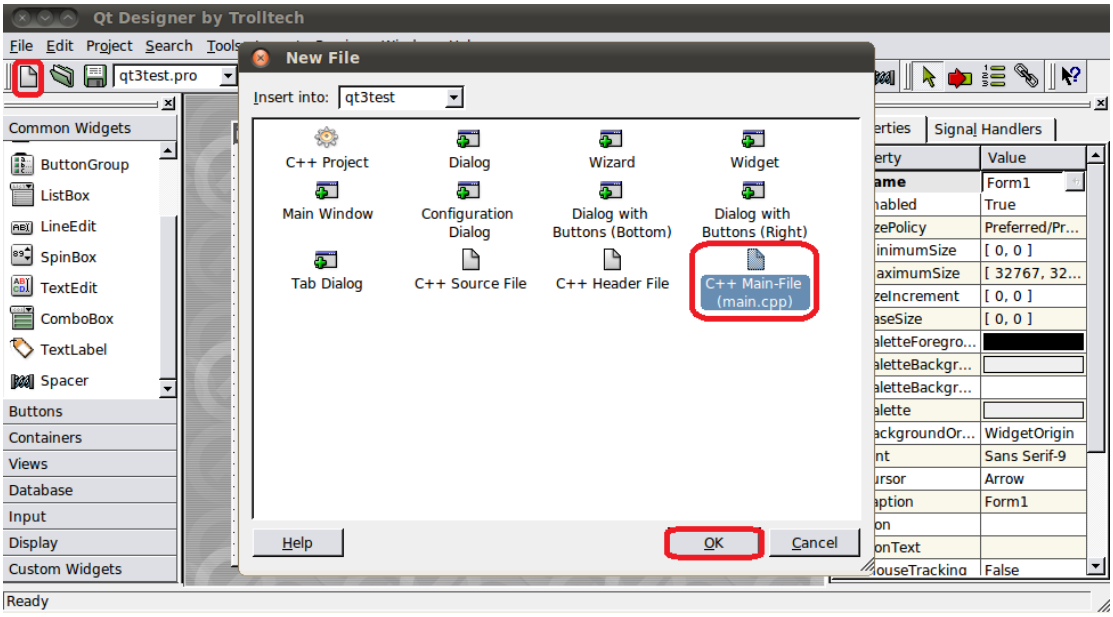


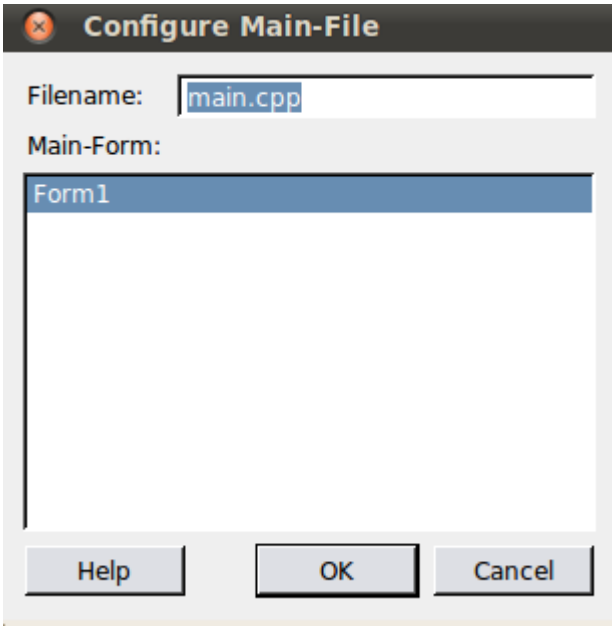



保存：

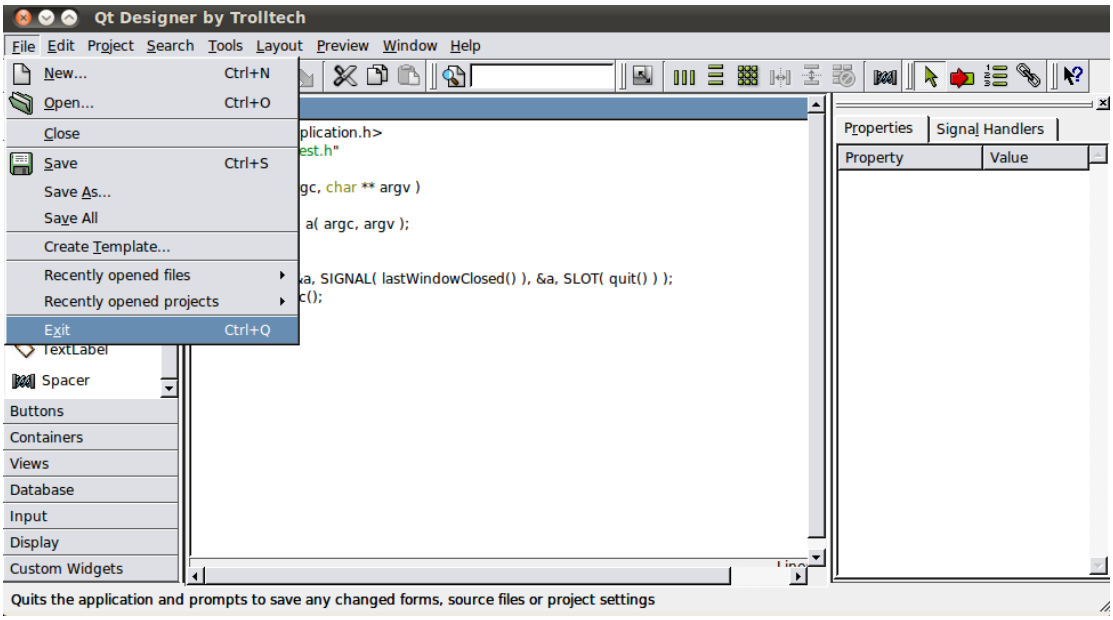



新建包含主函数的文件：






保存退出：




c）使用 uic 工具把项目工程文件与“.ui”文件转换为方便我们编程开发的“.h”和“.cpp”文件：

```
uic qt3test.ui -o qt3test.h
uic qt3test.ui -i qt3test.h -o qt3test.cpp
ls
```

```
root@loongson-desktop:~/qt3test# ls
main.cpp  qt3test.cpp  qt3test.h  qt3test.pro  qt3test.ui
```

d）先把“.ui”和“.pro”文件剪切移到到新建的“.uipro”目录备份，再利用 qmake 工具生成新的工程文件和 Makefile 文件：

```
mkdir .uipro
mv qt3test.ui qt3test.pro .uipro/
qmake -project
qmake
ls
```

```
root@loongson-desktop:~/qt3test# ls
main.cpp  Makefile  qt3test.cpp  qt3test.h  qt3test.pro
```

e）真正的 QT 编程开发（需要 C++编程知识）：

打开“qt3test.h”，“qt3test.cpp”，进行功能的扩充。或另外添加其他的“.h”和“.cpp”文件，此时需要使用 qmake 工具重新生成新的工程文件和 Makefile 文件。

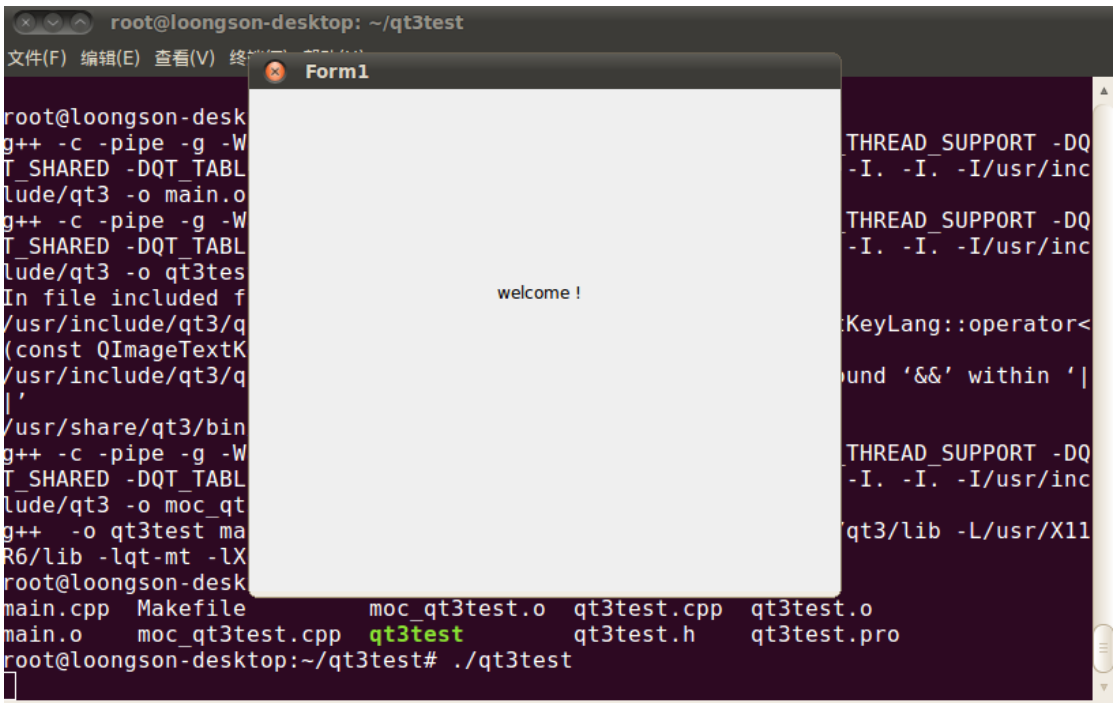
f）编译：

如果遇到提示缺少“g++”，命令安装更新 apt-get install g++。

```
make
```

g）运行测试：

```
./qt3test
```



**(3). 交叉编译 QT 应用程序**

软件工具位置：**Loongson_1B/Tools/gui_tools/qte3**

a）建立交叉编译环境（详见“第四章 4-3”）
b）交叉编译 qt-embedded-free
把 qt-embedded-free-3.3.8.tar.bz2 放于目录/root 下

1.解压 qt-embedded-free-3.3.8.tar.bz2，重命名“qt-embedded-free-3.3.8”为“qte”。

```
# tar xf qt-embedded-free-3.3.8.tar.bz2
# mv qt-embedded-free-3.3.8    qte
```

2.拷贝编译需要的本机已经安装的 uic、moc

```
# cp /usr/bin/uic      qte/bin
# cp /usr/bin/moc    qte/bin
```

3.设置 qte 的环境变量

```
# cd qte
# pwd          比如显示“/root/test-qt/qte”
# export QTEDIR = /root/test-qt/qte
# export LD_LIBRARY_PATH = $QTEDIR/lib:$LD_LIBRARY_PATH
# export PATH = $QTEDIR/bin:$PATH
# export QTDIR = $QTEDIR:$QTDIR
```

4.配置 qmake

在 mkspecs/qws 目录下新建 Cross 目录

```
# cd mkspecs/qws/
# mkdir gcc-3.4.6-2f
# cd gcc-3.4.6-2f
# cp ../linux-mips-g++/qmake.conf ./
# cp ../linux-mips-g++/qplatformdefs.h ./
```

修改 qmake.conf，将编译器指定为 Cross

```
QMAKE_CC = mipsel-linux-gcc
QMAKE_CXX = mipsel-linux-g++
QMAKE_LINK = mipsel-linux-g++
QMAKE_LINK_SHLIB = mipsel-linux-g++
```

5.配置、编译

```
# ./configure -qt-gif -xplatform qws/gcc-3.4.6-2f -thread -embedded mips -qvfb -no-cups -depths 4,8,16,24,32
# make
```

c）交叉编译 qt 应用程序“qt3test”

在 qt3test 目录下创建交叉编译配置环境脚本：

```
# vi env.sh
```

内容为：

```
export QTDIR=/root/qte
export QMAKEDIR=$QTDIR/qmake
export LD_LIBRARY_PATH=$QTDIR/lib:$LD_LIBRARY_PATH
export PATH=$QMAKEDIR/bin:$QTDIR/bin:/opt/gcc-3.4.6-2f/bin:$PATH
export QMAKESPEC=qws/linux-mips-g++
```

```
# source env.sh
# qmake -project
# qmake
# make
```

即得到需要移植的可执行程序“qt3test”。

**(4). 移植 QT 应用程序**

a）把 qt3test 所需的动态链接库与字体库拷贝到文件系统根目录下

```
#mkdir /root/rootfs/qte
#cp /root/qte/lib /root/rootfs/qte
```

b）把交叉编译得到的可执行程序‘qt3test’拷贝于文件系统根目录

```
#cp /root/qt3test/qt3test /root/rootfs
```

c）编写启动脚本

```
#cd /root/rootfs
#vi qt3test.sh
```

内容为：

```
export QTDIR=/qte
export LD_LIBRARY_PATH=$QTDIR/lib:$LD_LIBRARY_PATH
```

d）退出，制作文件系统镜像

```
$mkfs.cramfs rootfs rootfs-qt3test.img
```

e）开发板上电，烧写到开发板

```
PMON> devcp    tftp://192.168.0.244/rootfs-qt3test.img    /dev/mtd1
```

f）进到文件系统，运行程序

```
#sh qt3test.sh
# ./music-player -qws -fn unifont
```

至此移植完成。

## 8.3.2 实验二 SDL

软件工具位置：**Loongson_1B/Tools/gui_tools/SDL/SDL_DEV.tar.gz**

（1）交叉编译过程

a）建立交叉编译环境（详见“第四章 4-3”）。
b）建立编译库安装目录并解压 SDL 开发源码工具包：

```
#mkdir /opt/mipsel
#export PATH=/opt/gcc-3.4.6/bin:$PATH
#mkdir /home/SDL
#cd /home/SDL
#tar zxf SDL_DEV.tar.gz
```

c）编译 zlib-1.2.5

| 软件名称 | zlib |
|---|---|
| 功能描述 | zlib 是提供数据压缩用的函式库，最初是为 libpng 函式库所写的，后来普遍为许多软件所使用 |
| 下载地址 | http://www.zlib.net/ |
| 软件版本 | zlib-1.2.5.tar.gz |
| 依赖关系 | 默认 |

```
#export CC=mipsel-linux-gcc
#tar xf zlib-1.2.5.tar.gz
#cd zlib-1.2.5
# ./configure --prefix=/opt/mipsel --shared
# make
# make install
#export CC=gcc
```

d）编译 jpeg-7

| 软件名称 | libjpeg |
|---|---|
| 功能描述 | libjpeg 是一个用 c 语言编写支持 jpeg 解码、编码的库 |
| 下载地址 | http://linux.softpedia.com/get/Programming/Libraries/libjpeg-160.shtml |
| 软件版本 | jpegsrc.v7.tar.gz |
| 依赖关系 | 默认 |

```
#tar xf jpegsrc.v7.tar.gz
#cd jpeg-7/
#./configure --host=mipsel-linux --build=i686-pc-linux --prefix=/opt/mipsel
#make AR=mipsel-linux-ar RANLIB=mipsel-linux-ranlib CXX=mipsel-linux-g++
#make install
```

e）编译 freetype-2.4.3

| | |
|---|---|
| 软件名称 | freetype-2.4.3 |
| 功能描述 | freetype 是一个用 c 语言实现的字体栅格化引擎制作的一个库 |
| 下载地址 | http://download.savannah.gnu.org/releases/freetype/ |
| 软件版本 | freetype-2.4.3.tar.gz |
| 依赖关系 | 默认 |

```
#tar xf freetype-2.4.6.tar.gz
#cd freetype-2.4.6
#./configure --prefix=/opt/mipsel --host=mipsel-linux
  --build=i686-pc-linux
#make
#make install
```

f）编译 libiconv

| | |
|---|---|
| 软件名称 | libiconv |
| 功能描述 | libiconv 库为需要做转换的程序，实现了一个字符编码到另一个字符编码的转换 |
| 下载地址 | http://www.gnu.org/software/libiconv/#downloading |
| 软件版本 | libiconv-1.13.1.tar.gz |
| 依赖关系 | 默认 |

```
#tar xf libiconv-1.13.1.tar.gz
#cd libiconv-1.13.1
#./configure --host=mipsel-linux --build=i686-pc-linux --prefix=/opt/mipsel
# make AR=mipsel-linux-ar RANLIB=mipsel-linux-ranlib CXX=mipsel-linux-g++
# make install
```

g）编译 libpng-1.4.2

| | |
|---|---|
| 软件名称 | libpng |
| 功能描述 | libpng 是多种应用程序所使用的解析 PNG 图形格式的函数库 |
| 下载地址 | http://www.libpng.org/pub/png/libpng.html |
| 软件版本 | libpng-1.4.2.tar.gz |
| 依赖关系 | 默认 |

```
#tar xf libpng-1.4.2.tar.gz
#cd libpng-1.4.2
#./configure --host=mipsel-linux --build=i686-pc-linux --prefix=/opt/mipsel
   LDFLAGS="-L/opt/mipsel/lib -lz" CFLAGS="-I/opt/mipsel/include"
# make
# make install
```

h）编译 SDL-1.2.13

| | |
|---|---|
| 软件名称 | SDL-1.2.13 |
| 功能描述 | SDL 是一个自由的跨平台的多媒体开发包，适用视频音频和其他应用的软件 |
| 下载地址 | http://www.libsdl.org/download-1.2.php |
| 软件版本 | SDL-1.2.13.tar.gz |
| 依赖关系 | 默认 |

```
#tar xf SDL-1.2.13.tar.gz
#cd SDL-1.2.13
#vim ./src/joystick/linux/SDL_sysjoystick.c
```

将#include <limits.h> 改为#include <linux/limits.h>

```
# ./configure --host=mipsel-linux --prefix=/opt/mipsel --build=i686-pc-linux
--disable-static --disable-nasm --disable-video-x11 --disable-x11-shared --disable-dga
--disable-video-dga --disable-video-x11-dgamouse --disable-video-x11-vm
--disable-video-x11-xv --disable-video-x11-xinerama --disable-video-x11-xme
--disable-video-x11-xrander --disable-video-x11-dpms --disable-video-svga
--disable-video-directfb --enable-input-tslib --disable-esd --disable-esdtest
--disable-esd-shared --enable-pulseaudio=no --enable-pulseaudio-shared=no --without-x
# make
# make install
```

SDL/test 范例编译

```
#cd test
#./configure --host=mipsel-linux –prefix=/opt/mipsel --without-x
#make
```

i）编译 SDL_image-1.2.10

| | |
|---|---|
| 软件名称 | SDL_image-1.2.10 |
| 功能描述 | SDL_image 是用于处理图形文件的开源函数库 |
| 下载地址 | http://www.libsdl.org/projects/SDL_image/ |
| 软件版本 | SDL_image-1.2.10.tar.tar |
| 依赖关系 | 默认 |

```
#tar xf SDL_image-1.2.10.tar.gz
#cd SDL_image-1.2.10
# ./configure --host=mipsel-linux --build=i686-pc-linux --prefix=/opt/mipsel
--with-sdl-prefix=/opt/mipsel --enable-jpg --enable-png
#make
#make install
```

j）编译 SDL_ttf

| | |
|---|---|
| 软件名称 | SDL_ttf-2.0.9 |
| 功能描述 | SDL_ttf 是让应用程序能渲染汉字的扩展库 |
| 下载地址 | http://www.libsdl.org/projects/SDL_ttf/ |

| | |
|---|---|
| 软件版本 | SDL_ttf-2.0.9.tar.gz |
| 依赖关系 | 默认 |

```
#tar xf SDL_ttf-2.0.9.tar.gz
#cd SDL_ttf-2.0.9
#./configure --host=mipsel-linux --build=i686-pc-linux --prefix=/opt/mipsel
 --with-sdl-prefix=/opt/mipsel --with-freetype-prefix=/opt/mipsel --without-x
 LDFLAGS="-L/opt/mipsel/lib -lSDL -liconv -lfreetype"
#make
#make install
```

（2）移植过程

a）将/opt/mipsel/lib/下交叉编译的动态库放入文件系统库目录 lib 中。

b）编译测试例子 testbitmap。

1. 进入到 SDL-1.2.13 的 test 目录下

```
#cd /home/SDL/SDL-1.2.13/test/
#vim testbitmap.c
```

2. 将图形窗口大小修改为 320x240,与开发板显示屏大小一致。

即修改 99 行：

```
screen=SDL_SetVideo      Mode(640,480,video_bpp,videoflags)为
screen=SDL_SetVideoMode(320,240,video_bpp,videoflags)
```

3. 向 SDL 事件轮询循环加入延时指令，降低占用 CPU 的资源（注意：其他的例子也需要）。

即在 180 行（即 while ( SDL_PollEvent(&event) )循环外）添加如下：

```
SDL_Delay(10);
```

4. 编译：

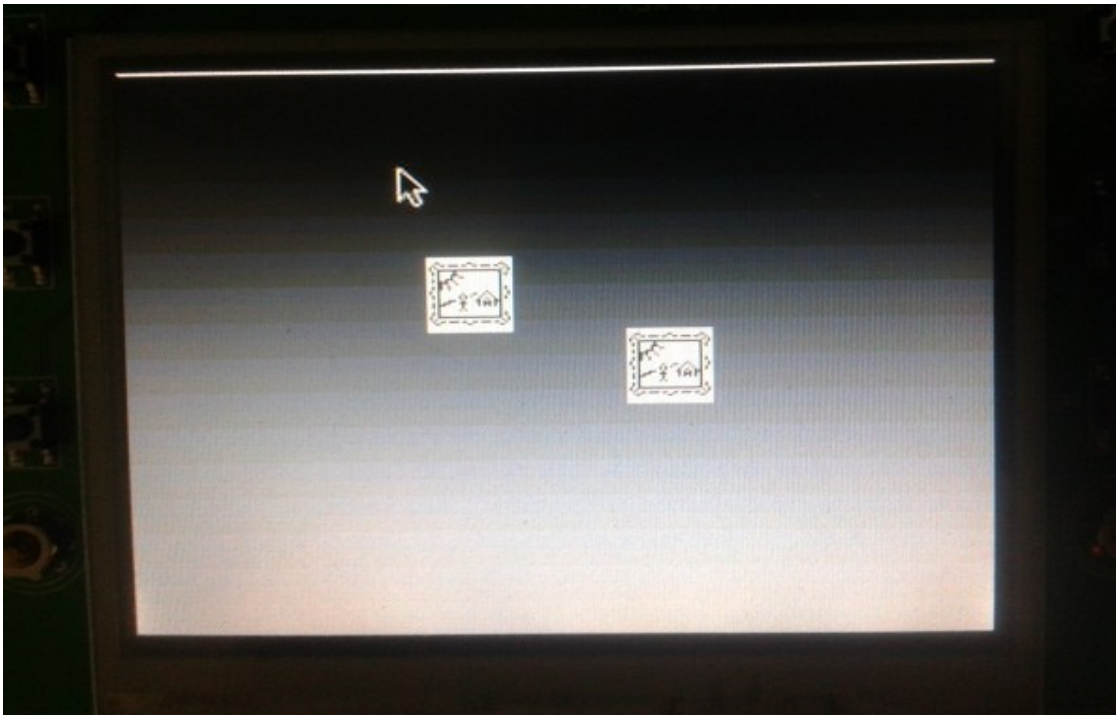
```
#make
```

c）将程序 testbitmap 和图片 picture.xbm 放入文件系统中相同目录下。

d）开发板上运行：

```
$./testbitmap
```

结果如下图所示：



## 8.4 LINUX 驱动程序实验

### 8.4.1 ADC 驱动程序

说明

程序源代码说明：

| | |
|---|---|
| 驱动源代码所在目录 | drivers/spi/ |
| 驱动程序名称 | mcp3201.c |
| 设备号 | mcp3201 属于杂项设备，设备自动生成 |
| 设备名 | /dev/mcp3201 |
| 测试程序源代码目录 | Examples/Drivers/ADC |
| 测试程序名称 | test-mcp3201.c |
| 测试程序可执行文件名称 | test-mcp3201 |

要写实际的驱动,就必须了解相关的硬件资源,比如用到的寄存器,物理地址,中断等,在这里,mcp3201 是一个很简单的例子,它用到了如下硬件资源。

开发板上所用到的 1 个 ADC 的硬件资源：

| ADC | 对应的 IO 寄存器名称 | 对应的 CPU 引脚 |
|---|---|---|
| Mcp3201 | Spi0 | G16 |

要操作所用到的 IO 口,就要设置它们所用到的寄存器,我们可以调用一些现成的函数或者宏，在我们的驱动中使用的是 mcp3201_open()来进行控制的。

函数 mcp3201_open()的定义在我们的驱动程序中，接下来的驱动程序代码将给出示意。

在下面的驱动代码中，你将看到调用 mcp3201_open()函数来对设备实施初始化并打开，除此之外，还需要调用一些和设备驱动密切相关的基本函数，如注册设备 misc_register，填写驱动函数结构 file_operations，以及像 Hello,Module 中那样的 module_init 和 module_exit 函数等。

有些函数并不是必须的,随着你对 Linux 驱动开发的进一步了解和阅读更多的代码,你自然明白。下面是我们为 mcp3201 编写的驱动代码清单。

驱动程序清单

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/device.h>
#include <linux/interrupt.h>
#include <linux/interrupt.h>
#include <linux/spi/spi.h>
#include <linux/spi/flash.h>
#include <asm/gpio.h>
#include <asm/semaphore.h>
#include <sb2f_board.h>
#include <linux/delay.h>
#include <linux/miscdevice.h>
#include <linux/errno.h>
#include <linux/fs.h>

#define  SB2F_BOARD_SPI0_BASE 0xbfe80000
struct spi_device *adc;

static int mcp3201_open(struct inode *inode, struct file *file) {
    //取消片选 CS3
    writeb(0xff, SB2F_BOARD_SPI0_BASE + REG_SPCSR);

    //复位 SPI 控制寄存器工作
    writeb(0x10, SB2F_BOARD_SPI0_BASE + REG_SPCR);

    //重置状态寄存器 SPSR
```

```
    writeb(0xc0, SB2F_BOARD_SPI0_BASE + REG_SPSR);

    //设置外部寄存器
    writeb(0x02, SB2F_BOARD_SPI0_BASE + REG_SPER);

    //禁用 SPI FLASH 读
    writeb(0x30, SB2F_BOARD_SPI0_BASE + REG_SPPR);

    //配置 SPI 时序
    writeb(0xd3, SB2F_BOARD_SPI0_BASE + REG_SPCR);

    //设置片选 CS3
    writeb(0x7f, SB2F_BOARD_SPI0_BASE + REG_SPCSR);
    return 0;
}

static int mcp3201_close(struct inode *inode, struct file *file) {
    //取消对 CS3 的片选
    writeb(0xff, SB2F_BOARD_SPI0_BASE + REG_SPCSR);
    return 0;
}

static ssize_t mcp3201_read(struct file *file, char __user *buf, size_t count, loff_t *ptr)
{
    ssize_t retval;
    unsigned char val[2] = {0x0};
    unsigned char tx_buf[1] = {0x0};
    unsigned char rx_buf[2] = {0};

    //发送 0 个字节的命令到 spi 从机，并从 spi 从机中读取 2 字节的数据到缓冲区
    retval = spi_write_then_read(adc, tx_buf, 0, rx_buf, 2);

    if (retval < 0) {
        dev_err(&adc->dev, "error %d reading SR\n", (int)retval);
        return retval;
      }

    //对从 mcp3201 读来的数据，按照其 datasheet 的要求取出编码
    val[0] = rx_buf[1] >> 1;
    val[0] |= (rx_buf[0] & 0x01) << 7;
    val[1] = (rx_buf[0] >> 1) & 0xf;

    //将从 mcp3201 取出的数据合理编码，并传送到用户空间供用户使用
```

```
    if (copy_to_user(buf, val, 2))
    {
        printk("Copy data to userspace error!\n");
        return -EFAULT;
    }
    return 0;
}

static int __devinit mcp3201_probe(struct spi_device *spi)
{
    struct spi_device *spi0;
    spi0 = spi;
    adc = spi0;
    return 0;
}

static int __devexit mcp3201_remove(struct spi_device *spi)
{
    writeb(0xff, SB2F_BOARD_SPI0_BASE + REG_SPCSR);
    return 0;
}

static struct spi_driver mcp3201_driver = {
    .driver = {
    .name    = "mcp3201",
    .bus= &spi_bus_type,
    .owner   = THIS_MODULE,
    },
    .probe    = mcp3201_probe,
    .remove = __devexit_p(mcp3201_remove),
};

static const struct file_operations mcp3201_ops = {
        .owner = THIS_MODULE,
        .open = mcp3201_open,
        .release = mcp3201_close,
        .read = mcp3201_read,
};

static struct miscdevice mcp3201_miscdev = {
        MISC_DYNAMIC_MINOR,
        "mcp3201",
        &mcp3201_ops,
```

```
};

static int mcp3201_init(void)
{
    if (misc_register(&mcp3201_miscdev)) {
        printk(KERN_WARNING "buzzer:Couldn't register device 10, %d.\n", 255);
        return -EBUSY;
    }

    return spi_register_driver(&mcp3201_driver);
}

static void mcp3201_exit(void)
{
    spi_unregister_driver(&mcp3201_driver);
}

module_init(mcp3201_init);
module_exit(mcp3201_exit);

MODULE_LICENSE("GPL");
```

### 8.4.2 外部按键驱动

说明

程序源代码说明：

| | |
|---|---|
| 驱动源程序所在目录 | driver/input/ |
| 驱动程序名 | 74LV165_button.c<br>74LV165_button.h |
| 设备名 | ls1b_buttons |
| 测试程序源代码目录 | |
| 测试程序代码名 | 74LV165_test.c |
| 测试程序可执行文件名 | 74LV165_test |
| 说明：按键驱动已经被编译到缺省内核中，因此不用使用 insmod 方式加载 | |

开发板用到的资源：

| | |
|---|---|
| gpio0 | KEY_DATA |
| gpio1 | KEY_EN |
| gpio2 | KEY_SCL |

按键驱动在硬件上没有接外部中断，且通过两片并行输入串行输出的移位芯片 74LV165 连

接，因此，按键通过轮询的方式实现。应用程序直接读取/dev/ls1b_buttons 设备节点来对按键值进行读取。在启动 linux 内核后，在根目录下执行当前目录下的可执行文件 74LV165_test。

驱动程序清单

```
#include<linux/module.h>
#include<linux/init.h>
#include<linux/fs.h>
#include<linux/timer.h>
#include<linux/ioctl.h>
#include<linux/io.h>
#include<linux/types.h>
#include<linux/kernel.h>
#include<linux/ioport.h>
#include<linux/errno.h>
#include<asm/uaccess.h>
#include<linux/miscdevice.h>
#include<linux/wait.h>
#include<linux/interrupt.h>
#include <linux/irq.h>
#include <asm/irq.h>
#include "74LV165_button.h"

#define BUF_MAXSIZE 16
#define TIMER_DELAY 5

static    unsigned int    BReadBuf = 0;


static    int delay(int time){
    while(--time);
    return 0;
}

static void prepare_for_read(void){

    int reg = 0;

    /* CP =0 */
    LS1B_74LV165_READ(reg, LS1B_74LV165_CP);
    LS1B_74LV165_WRITE(LS1B_74LV165_CP, reg & (~(1 << 2)));

    /* PL = 0 */
    LS1B_74LV165_READ(reg, LS1B_74LV165_PL);
```

```
    LS1B_74LV165_WRITE(LS1B_74LV165_PL, reg & ~(1 << 1));

    /* delay 100 */
    delay(10000);

    /* PL = 1 */
    LS1B_74LV165_READ(reg, LS1B_74LV165_PL);
    LS1B_74LV165_WRITE(LS1B_74LV165_PL, reg | (1 << 1));


}

static void scanf_keyboard(void){

    int reg = 0, val = 0;
    int time;

    delay(10);
    LS1B_74LV165_READ(reg, LS1B_74LV165_DATA);
    val = ((~reg) & (1 << 0));
    BReadBuf = val >> 0;

    for(time = 1; time < 16; time ++){
        /* delay */
          delay(100);

          /* CP = 1 */
          LS1B_74LV165_READ(reg, LS1B_74LV165_CP);
          LS1B_74LV165_WRITE(LS1B_74LV165_CP, reg | (1 << 2));

          delay(100);
          LS1B_74LV165_READ(reg, LS1B_74LV165_DATA);
          val = ((~reg) & (1 << 0));
          BReadBuf |= val << time;

          /* CP = 0 */
          LS1B_74LV165_READ(reg, LS1B_74LV165_CP);
          LS1B_74LV165_WRITE(LS1B_74LV165_CP, reg & (~(1 << 2)));
      }

}

static    void button_read(void){
```

```
            prepare_for_read();

            /* scanf keyboard */
        scanf_keyboard();

            if(BReadBuf){
                    printk("\nBReadBuf value is 0x%08x\n",BReadBuf);
            }

    }

    static    int LS1B_74LV165_button_open(struct inode *inode, struct file *filp){
            int reg;

        printk("Welcome to use 74LV165 driver\n");

        //enalbe pin
        LS1B_74LV165_EN_GPIO(GPIO_KEY_DATA);
        LS1B_74LV165_EN_GPIO(GPIO_KEY_EN);
        LS1B_74LV165_EN_GPIO(GPIO_KEY_SCL);

            //enable output
        LS1B_74LV165_OEN_GPIO(GPIO_KEY_SCL);
        LS1B_74LV165_OEN_GPIO(GPIO_KEY_EN);
        LS1B_74LV165_IEN_GPIO(GPIO_KEY_DATA);

        printk("init 74LV165 is done\n");

        return 0;
    }

    static    ssize_t LS1B_74LV165_button_read(struct file    *filp, char __user *buf, size_t count,
loff_t *oppos){

        button_read();

        if(BReadBuf){
            copy_to_user(buf,&BReadBuf,count);
            BReadBuf= 0;
            delay(10000000);
            return count;
            }
```

```
        return -EFAULT;
    }

    static int LS1B_74LV165_button_ioctl(struct inode *inode, struct file *filp, unsigned int cmd, unsigned long arg){
        return 0;
    }

    static    struct file_operations ls1b_74Llv165_button_fops = {
        .open = LS1B_74LV165_button_open,
        .read = LS1B_74LV165_button_read,
        .ioctl = LS1B_74LV165_button_ioctl,
    };

    static struct miscdevice ls1b_74lv165_button = {
        .minor = LS1B_BUTTON_MINOR,
        .name    = "ls1b_buttons",
        .fops    = &ls1b_74Llv165_button_fops,
    };

    static    int __init    LS1B_74LV165_button_init(void){
        int ret;
        printk("======================button init=========================\n");
        ret = misc_register(&ls1b_74lv165_button);
        if(ret < 0){
            printk("74LV165_button can't get major number !\n");
            return ret;
        }

    #ifdef CONFIG_DEVFS_FS
        devfs_button_dir = devfs_mk_dir(NULL,"LS1B_74LV165_button",NULL);
        devfs_buttonraw=
devfs_register(devfs_kbd_dir,"0raw",DEVFS_FL_DEFAULT,kbdMajor,KBDRAW_MINOR,S_IFCHR|S_IRUSR|S_IWUSR,&LS1B_74LV165_button_fops,NULL);
    #endif
        return 0;
    }

    static    void    __exit    LS1B_74LV165_button_exit(void){
        misc_deregister(&ls1b_74lv165_button);
    }
```

```
module_init(LS1B_74LV165_button_init);
module_exit(LS1B_74LV165_button_exit);
```

其中 module_init（）和 module_exit（）函数注册了 LS1B_74LV165_button_init 及 LS1B_74LV165_button_exit 函数。当模块加载时将会调用这两个函数。

在 LS1B_74LV165_button_init（）函数中调用 misc_register（）函数注册按键驱动为 misc 设备。并将 struct miscdevice 的数据结构传递给该函数，在该结构中对 minor 字段及 name 字段进行赋值，这样该驱动会在/dev 目录下创建名为 ls1b_buttons 的设备文件，其中主设备号为 10，次设备号为 minor 的值，即为 143。

在 struct miscdevice 数据结构中，还有一个 fops 的字段，该字段为指向 struct file_operations 数据结构的指针。其中包括 open、read、ioctl 等字段，当用户程序调用 open、write、ioctl 函数时，最终调用该驱动的相应函数。

在 LS1B_74LV165_button_open（）函数中使能芯片 74LV165 用到的 3 个 gpio 口，并将端口设置为输出。当用户调用 read 函数时，运行驱动程序的 LS1B_74LV165_button_read（）函数，该函数调用 button_read（），读取按键编码。在 button_read（）函数中调用 prepare_for_read() 根据芯片 74LV165 的时序将按键信息写入芯片 74LV165 的移位寄存器，然后调用 scanf_keyboard(）函数根据芯片 74LV165 的时序读写移位寄存器中的按键编码。最后通过 LS1B_74LV165_button_read（）函数中的 copy_to_user（）函数将按键编码返回给用户空间。

## 8.4.3 RTC 驱动程序

说明

程序源代码说明：

| | |
|---|---|
| 驱动源代码所在目录 | /driver/rtc |
| 驱动程序名称 | rtc-gs2fsb.c |
| 该驱动的主设备号 | RTC 设备，设备号将自动生成 |
| 设备名 | /dev/rtc0 |
| 测试程序可执行文件名称 | date hwclock |

RTC 模块寄存器位于 0xbfe64000——0xbfe67fff 的 16KB 地址空间内，其基地址为 0xbfe64000，所有寄存器位宽均为 32 位。详细的寄存器描述可参考 loongson 1B 的数据手册。

驱动程序清单

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/rtc.h>
#include <linux/io.h>

#define   RTC_TOYIM     0x00
#define   RTC_TOYWLO    0x04
#define   RTC_TOYWHI    0x08
```

```
#define    RTC_TOYRLO    0x0c
#define    RTC_TOYRHI    0x10
#define    RTC_TOYMH0    0x14
#define    RTC_TOYMH1    0x18
#define    RTC_TOYMH2    0x1c
#define    RTC_CNTL       0x20
#define    RTC_RTCIM      0x40
#define    RTC_WRITE0    0x44
#define    RTC_READ0     0x48
#define    RTC_RTCMH0    0x4c
#define    RTC_RTCMH1    0x50
#define    RTC_RTCMH2    0x54

struct rtc_sb2f{
    struct rtc_device *rtc;
    void __iomem          *regs;
    unsigned long         alarm_time;
    unsigned long         irq;
    spinlock_t            lock;
};

static int sb2f_rtc_read_register(struct rtc_sb2f *rtc, unsigned long reg)
{
    int data;
    data = (*(volatile unsigned int *)(rtc->regs + reg));
    return data;
}

static int sb2f_rtc_write_register(struct rtc_sb2f *rtc, unsigned long reg, int data)
{
    (*(volatile unsigned int *)(rtc->regs + reg)) = data;
}

static int sb2f_rtc_readtime(struct device *dev, struct rtc_time *tm)
{
    struct rtc_sb2f    *rtc = dev_get_drvdata(dev);
    unsigned long now, now1;

    now = sb2f_rtc_read_register(rtc, RTC_TOYRLO);

    tm->tm_sec = ((now >> 4) & 0x3f);
    tm->tm_min = ((now >> 10) & 0x3f);
    tm->tm_hour = ((now >> 16) & 0x1f);
```

```
        tm->tm_mday = ((now >> 21) & 0x1f);
        tm->tm_mon = ((now >> 26) & 0x3f) -1;

        now1 = sb2f_rtc_read_register(rtc, RTC_TOYRHI);

        tm->tm_year = (now1 -1900) ;
        return 0;
}

static int sb2f_rtc_settime(struct device *dev, struct rtc_time *tm)
{
        struct rtc_sb2f *rtc = dev_get_drvdata(dev);
        unsigned long now;
        int ret;

        now = ((tm->tm_sec << 4) | (tm->tm_min << 10) | (tm->tm_hour << 16) |
                        (tm->tm_mday << 21) | (((tm->tm_mon+1)<< 26) ));
        spin_lock_irq(&rtc->lock);
        sb2f_rtc_write_register(rtc, RTC_TOYWLO, now);
//set year
        sb2f_rtc_write_register(rtc, RTC_TOYWHI, (tm->tm_year+1900) );
        spin_unlock_irq(&rtc->lock);

        return 0;
}

static int sb2f_rtc_readalarm(struct device *dev, struct rtc_wkalrm *alrm)
{
        struct rtc_sb2f *rtc = dev_get_drvdata(dev);
        unsigned long time;

        spin_lock_irq(&rtc->lock);
        time = sb2f_rtc_read_register(rtc, RTC_TOYMH0);
        spin_unlock_irq(&rtc->lock);

        alrm->time.tm_sec = (time & 0x3f);
        alrm->time.tm_min = ((time >> 6) & 0x3f);
        alrm->time.tm_hour = ((time >> 12) & 0x1f);
        alrm->time.tm_mday = ((time >> 17) & 0x1f);
        alrm->time.tm_mon = ((time >> 22) & 0x1f);
        alrm->time.tm_year = ((time >> 26) & 0x3f);

        return 0;
```

```
}


static int sb2f_rtc_setalarm(struct device *dev, struct rtc_wkalrm *alrm)
{
    struct rtc_sb2f *rtc = dev_get_drvdata(dev);
    int time;

    time =(( alrm->time.tm_sec & 0x3f) | ((alrm->time.tm_min & 0x3f) << 6)
            | ((alrm->time.tm_hour & 0x1f) << 12) | ((alrm->time.tm_mday
            & 0x1f) << 17) | ((alrm->time.tm_mon & 0xf) << 22) |
            ((alrm->time.tm_year & 0x3f) << 26));

    spin_lock_irq(&rtc->lock);
    sb2f_rtc_write_register(rtc, RTC_TOYMH0, time);
    spin_unlock_irq(&rtc->lock);

    return 0;
}

static int sb2f_rtc_ioctl(struct device *dev, unsigned int cmd,
            unsigned long arg)
{

    switch(cmd) {
        case RTC_PIE_ON:
            break;
        case RTC_PIE_OFF:
            break;
        case RTC_UIE_ON:
            break;
        case RTC_UIE_OFF:
            break;
        case RTC_AIE_ON:
            break;
        case RTC_AIE_OFF:
            break;
        default:
            return -ENOIOCTLCMD;
    }

    return 0;
```

```
}

static irqreturn_t sb2f_rtc_interrupt(int irq, void *dev_id)
{
    struct rtc_sb2f *rtc = (struct rtc_sb2f *)dev_id;/*接收申请中断时传递过来的 rtc_dev 参数
*/
    unsigned long events = 0;

    spin_lock(&rtc->lock);
    events = RTC_AF | RTC_IRQF;
    /*节拍时间中断到来的时候，去设定 RTC 中节拍时间的相关信息，具体设定的方法，
RTC 核心部分已经在 rtc_update_irq 接口函数中实现，函数定义实现在 interface.c 中*/
    rtc_update_irq(rtc->rtc, 1, events);
    spin_unlock(&rtc->lock);
}

static struct rtc_class_ops sb2f_rtc_ops = {
    .ioctl          = sb2f_rtc_ioctl,
    .read_time      = sb2f_rtc_readtime,
    .set_time       = sb2f_rtc_settime,
    .read_alarm     = sb2f_rtc_readalarm,
    .set_alarm      = sb2f_rtc_setalarm,
};


static int __init sb2f_rtc_probe(struct platform_device *pdev)
{
    struct resource *regs;/*定义一个资源，用来保存获取的 RTC 的资源*/
    struct rtc_sb2f *rtc;
    int       irq = -1;
    int       ret;

    rtc = kzalloc(sizeof(struct rtc_sb2f), GFP_KERNEL);
    if (!rtc) {
        dev_dbg(&pdev->dev, "out of memory\n");
        return -ENOMEM;
    }

    /*获取 RTC 平台设备所使用的 IO 端口资源，注意这个 IORESOURCE_MEM 标志和 RTC
平台设备定义中的一致*/
    regs = platform_get_resource(pdev, IORESOURCE_MEM, 0);
    if (!regs) {
        dev_dbg(&pdev->dev, "no nmio resource defined\n");
```

```
            ret = -ENXIO;
            goto out;
        }
        printk("the regs->start is 0x%x", regs->start);

        //在系统定义的 RTC 平台设备中获取 TICK 节拍时间中断号
        irq = platform_get_irq(pdev, 0);
        printk("the irq is %d\n", irq);
        rtc->irq = irq;

        //把 I/O 资源 IORESOURCE_MEM 起始地址到结束地址范围的 资源映射到虚拟地址
ioremap 定义在 io.h 中。
        //注意：IO 空间要映射后才能使用，以后对虚拟地址的操作就是对 IO 空间的操作
        rtc->regs = ioremap(regs->start, regs->end - regs->start + 1);
        if (!rtc->regs) {
            ret = -ENOMEM;
            dev_dbg(&pdev->dev, "could not map i/o memory\n");
            goto out;
        }

        //初始化自旋锁
        spin_lock_init(&rtc->lock);

    //  next set control register
    //  sb2f_rtc_write_register(rtc, RTC_CNTL, 0x800);
        (*(volatile int *)(0xbfe64040)) = 0x2d00;          //TOY 和 RTC 控制寄存器
                                                            //0010 1101 0000 0000
        //申请中断 (这个可以放到 open()函数里)
        //res->start 中断号(在具体的平台设备 platform_device 定义时赋值)
        //s3c2410wdt_irq 中断处理函数  IRQF_DISABLED 中断处理属性 快速慢速共享
        //pdev->name 这个传递给 request_irq 的字串用在 /proc/interrupts 来显示中断的拥
有者
        //pdev 用作共享中断线的指针 被驱动用来指向它自己的私有数据区(来标识哪个设备
在中断).
        ret = request_irq(irq, sb2f_rtc_interrupt, IRQF_SHARED, "rtc", rtc);
        if (ret) {
            dev_dbg(&pdev->dev, "could not request irq %d", irq);
            goto out_iounmap;
        }

        //注册 RTC 设备
        /* register RTC and exit */
        /*将 RTC 注册为 RTC 设备类，RTC 设备类在 RTC 驱动核心部分中由系统定义好的，
```

注意 rtcops 这个参数是一个结构体，该结构体的作用和里面的接口函数实现在第③步中。

```
        rtc_device_register 函数在 rtc.h 中定义，在 drivers/rtc/class.c 中实现*/
    rtc->rtc = rtc_device_register(pdev->name, &pdev->dev, &sb2f_rtc_ops, THIS_MODULE);
    /*里的 IS_ERR(),它就是判断 kthread_run()返回的指针是否有错,如果指针并不是指向最后一个 page,那么没有问题,申请成功了,如果指针指向了最后一个 page,那么说明实际上这不是一个有效的指针,这个指针里保存的实际上是一种错误代码.而通常很常用的方法就是先用 IS_ERR()来判断是否是错误,然后如果是,那么就调用 PTR_ERR()来返回这个错误代码.*/
    if (IS_ERR(rtc->rtc)) {
        dev_dbg(&pdev->dev, "could not register rtc device\n");
        //PTR_ERR()只是返回错误代码,也就是提供一个信息给调用者,如果你只需要知道是否出错,而不在乎因为什么而出错
        ret = PTR_ERR(rtc->rtc);
        goto out_free_irq;
    }

    /*将 RTC 设备类的数据传递给系统平台设备。
      platform_set_drvdata 是定义在 platform_device.h 的宏，如下：
      #define platform_set_drvdata(_dev,data)      dev_set_drvdata(&(_dev)->dev, (data))
      而 dev_set_drvdata 又被定义在 include/linux/device.h 中，如下：
       static inline void dev_set_drvdata (struct device *dev, void *data){
            dev->driver_data = data;
       }*/
    platform_set_drvdata(pdev, rtc);

    /*device_init_wakeup 该函数定义在 pm_wakeup.h 中，定义如下：
     static inline void device_init_wakeup(struct device *dev, int val){
          dev->power.can_wakeup = dev->power.should_wakeup = !!val;}
    显然这个函数是让驱动支持电源管理的,这里只要知道,can_wakeup 为 1 时表明这个设备可以被唤醒,设备驱动为了支持 Linux 中的电源管理,有责任调用 device_init_wakeup()来初始化 can_wakeup，而 should_wakeup 则是在设备的电源状态发生变化的时候被 device_may_wakeup()用来测试,测试它该不该变化，因此 can_wakeup 表明的是一种能力，而 should_wakeup 表明的是有了这种能力以后去不去做某件事。好了，我们没有必要深入研究电源管理的内容了，要不就扯远了，电源管理以后再讲*/
    device_init_wakeup(&pdev->dev, 1);

    //头文件 include/linux/device.h 中所提供的宏（包括 dev_printk()、dev_dbg()、dev_warn()、dev_info()和 dev_err()）来决定何种类型的设备访问消息需要被记录。 printk()
    dev_info(&pdev->dev, "SB2F RTC    at %08lx irq %ld \n", (unsigned long)rtc->regs, rtc->irq);
    return 0;
out_free_irq:
    free_irq(irq, rtc);
out_iounmap:
```

```
        iounmap(rtc->regs);
    out:
        kfree(rtc);
        return ret;
    }

    /*注意：这是使用了一个__devexit。
      我们还是先来讲讲这个：
      在 Linux 内核中，使用了大量不同的宏来标记具有不同作用的函数和数据结构，
      这些宏在 include/linux/init.h 头文件中定义，编译器通过这些宏可以把代码优化放到合适的内存位置，
      以减少内存占用和提高内核效率。__devinit、__devexit 就是这些宏之一，在 probe()和remove()函数中
      应该使用__devinit 和__devexit 宏。又当 remove()函数使用了__devexit 宏时，则在驱动结构体中一定要
      使用__devexit_p 宏来引用 remove()，所以在第①步中就用__devexit_p 来引用 rtc_remove*/
    static int __exit sb2f_rtc_remove(struct platform_device *pdev)
    {
        /*从系统平台设备中获取 RTC 设备类的数据*/
        struct rtc_sb2f *rtc = platform_get_drvdata(pdev);

        device_init_wakeup(&pdev->dev, 0);

        free_irq(rtc->irq, rtc);
        iounmap(rtc->regs);/*释放 RTC 虚拟地址映射空间*/
        rtc_device_unregister(rtc->rtc);/*注销 RTC 设备类*/
        kfree(rtc);/*销毁保存 RTC 平台设备的资源内存空间*/
        platform_set_drvdata(pdev, NULL);/*清空平台设备中 RTC 驱动数据*/

        return 0;
    }

    MODULE_ALIAS("platform:sb2f-rtc");

    static struct platform_driver sb2f_rtc_driver = {
        .remove      = __exit_p(sb2f_rtc_remove),
        .driver      = {
            .name    = "sb2f-rtc",
            .owner   = THIS_MODULE,
        },

    };
```

```
static int __init sb2f_rtc_init(void)
{
    /*将 RTC 注册成平台设备驱动*/
    return platform_driver_probe(&sb2f_rtc_driver, sb2f_rtc_probe);
}

module_init(sb2f_rtc_init);

static void __exit sb2f_rtc_exit(void)
{
    /*注销 RTC 平台设备驱动*/
    platform_driver_unregister(&sb2f_rtc_driver);
}
module_exit(sb2f_rtc_exit);

MODULE_AUTHOR("<ninglichen@loongson.cn>");
MODULE_DESCRIPTION("Real Time clock for loongson2fsb");
MODULE_LICENSE("GPL");
```

# 附录

## 附录 1　Windows 与 Ubuntu 间文件的传输

通常 linux 操作系统在插入 U 盘后，U 盘会自动挂载在“/media”目录下生成一个新的目录。这个目录就相当于 U 盘分区，可以进行传输数据。

也可以手动挂载。如下：

```
#fdisk -l

Disk /dev/sda: 21.5 GB, 21474836480 bytes
255 heads, 63 sectors/track, 2610 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x0004444f

   Device Boot      Start         End      Blocks   Id  System
/dev/sda1   *           1        2497    20051968   83  Linux
/dev/sda2            2497        2611      916481    5  Extended
/dev/sda5            2497        2611      916480   82  Linux swap / Solaris

Disk /dev/sdb: 4002 MB, 4002938880 bytes
68 heads, 3 sectors/track, 38324 cylinders
Units = cylinders of 204 * 512 = 104448 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x04030201

   Device Boot      Start         End      Blocks   Id  System
/dev/sdb1               1       38325     3909116    b  W95 FAT32
```

```
#mount /dev/sdb1    /mnt
```

卸载：

```
#umount /mnt     或者    #umount /dev/sdb1
```

## 附录 2　Linux 常用命令详解

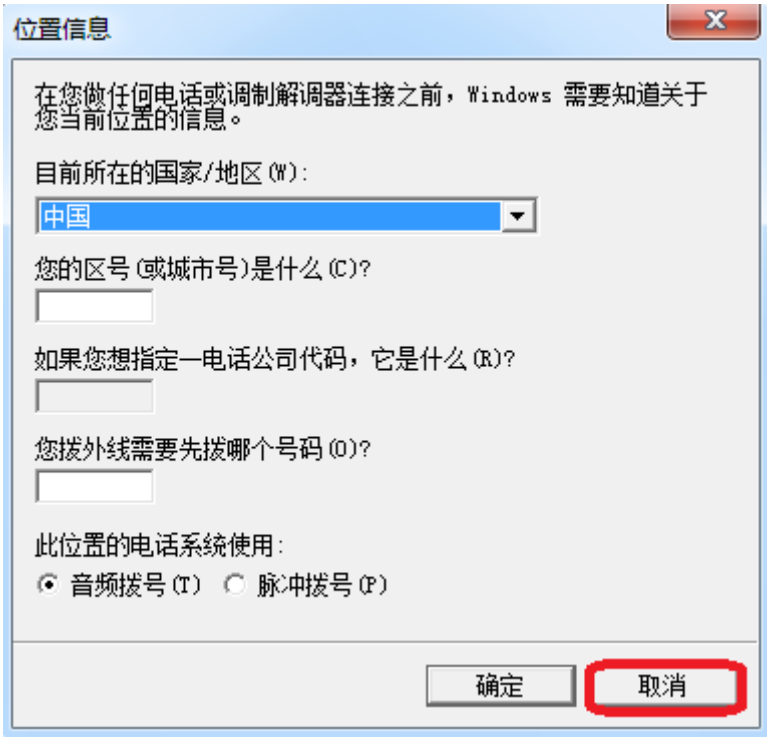
| 类型 | 命令 | 说明 | 例子 | 例子含义 |
|---|---|---|---|---|
| 文件处理 | rm | 删除 | rm file | 删除某一个文件 |
| | | | rm -fr dir | 删除当前目录下叫 dir 的整个目录 |
| | mv | 移动 | mv source target | 将文件 source 更名为 target |
| | mkdir | 创建目录 | mkdir -p fater/dir | 创建目录 dir，即使上层目录 fater 存在也不返回出错 |
| | diff | 比较 | diff dir1 dir2 | 比较目录 1 与目录 2 的文件列表是否相同，但不比较文件的实际内容，不同则列出 |
| | | | diff file1 file2 | 比较文件 1 与文件 2 的内容是否相同，如果是文本格式的文件，则将不相同的内容显示，如果是二进制代码则只表示两个文件是不同的 |
| | du | 容量查看 | du | 计算当前目录的容量 |
| | | | du -sm /root | 计算/root 目录的容量并以 M 为单位 |
| | cat | 文件内容查看 | cat file | 显示文件的内容 |
| | | | cat file \| more | 显示文件的内容并传输到 more 程序实现分页显示，使用命令 less file 可实现相同的功能 |
| | find | 查找 | find /path -name file | 在/path 目录下查找看是否有文件 file |
| | grep | 搜索 | grep -ir "chars" | 在当前目录的所有文件查找字串 chars，并忽略大小写，-i 为大小写，-r 为下一级目录 |
| 系统管理 | date | 时间日期 | date | 显示当前日期时间 |
| | | | date -s 20:30:30 | 设置系统时间为 20:30:30 |
| | | | date -s 2012-3-5 | 设置系统时期为 2012-3-5 |
| | export | 设置环境变量 | export LC_ALL=zh_CN.GB2312 | 将环境变量 LC_ALL 的值设为 zh_CN.GB2312 |
| | mount | 挂载 | mount -t yaffs /dev/mtdblock/0 /mnt | 把/dev/mtdblock/0 装载到 /mnt 目录 |
| | dmesg | 启动信息显示 | dmesg | 显示 kernle 启动及驱动装载信息 |
| | chmod | 改变文件权限 | chmod a+x file | 将 file 文件设置为可执行，脚本类文件一定要这样设置一个，否则得用 bash file 才能执行 |
| | | | chmod 666 file | 将文件 file 设置为可读写 |

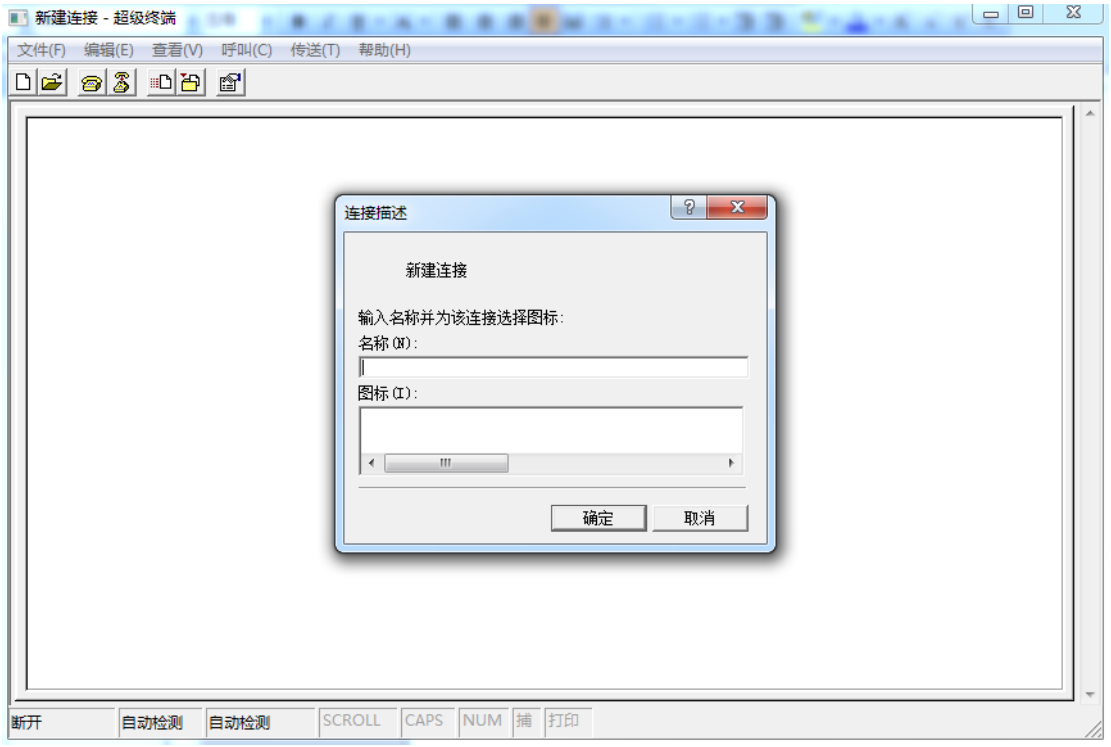
| | | | | |
|---|---|---|---|---|
| | ps | 进程查看 | ps | 显示当前系统进程信息 |
| | | | ps -ef | 显示系统所有进程信息 |
| | kill | 杀死进程 | kill -9 500 | 将进程编号为 500 的程序杀死 |
| 网络管理 | ifconfig | 检查并设置网络 | ifconfig eth0 192.168.1.100 | 设置网络 IP 地址为 192.168.1.100 |
| | | | ifconfig eth0 down | 暂时关闭网卡 |
| | route | 设置网关 | route | 显示当前路由设置情况 |
| | | | route add default gw 192.168.1.1 | 设置 192.168.1.1 为默认路由 |
| | | | route del default | 删除默认路由 |
| | ping | 测试网络 | ping –c 3 192.168.3.1 | 表示向 192.168.3.1 连续发送 3 次数据包，以验证网络是否连接正常 |
| 其他 | echo | 回显 | echo message | 显示一串字符 |
| | | | echo "message message2" | 显示不连续的字符串 |
| | more | 分页查看 | more | 分页命令，一般通过管道将内容传给它，如 ls \| more |
| | mknod | 创建节点 | mknod /dev/tty1 c 4 1 | 创建字符设备 tty1,主设备号为 4，从设备号为 1，即第一个 tty 终端 |
| | vi | 编辑 | vi file | 编辑文件 file |

# 附录 3　Windows 超级终端使用说明

软件工具位置：**Loongson_1B/Tools/windows_tools/超级终端**

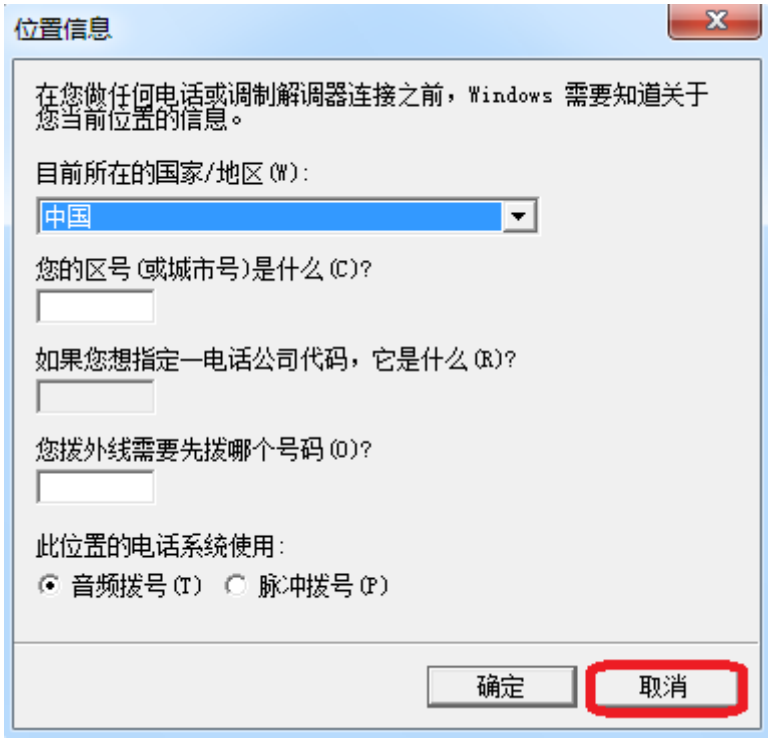
（1）打开超级终端：


位置信息
在您做任何电话或调制解调器连接之前，Windows 需要知道关于您当前位置的信息。
目前所在的国家/地区(W):
中国
您的区号(或城市号)是什么(C)?
如果您想指定一电话公司代码，它是什么(R)?
您拨外线需要先拨哪个号码(O)?
此位置的电话系统使用:
音频拨号(T)
脉冲拨号(P)
确定
取消

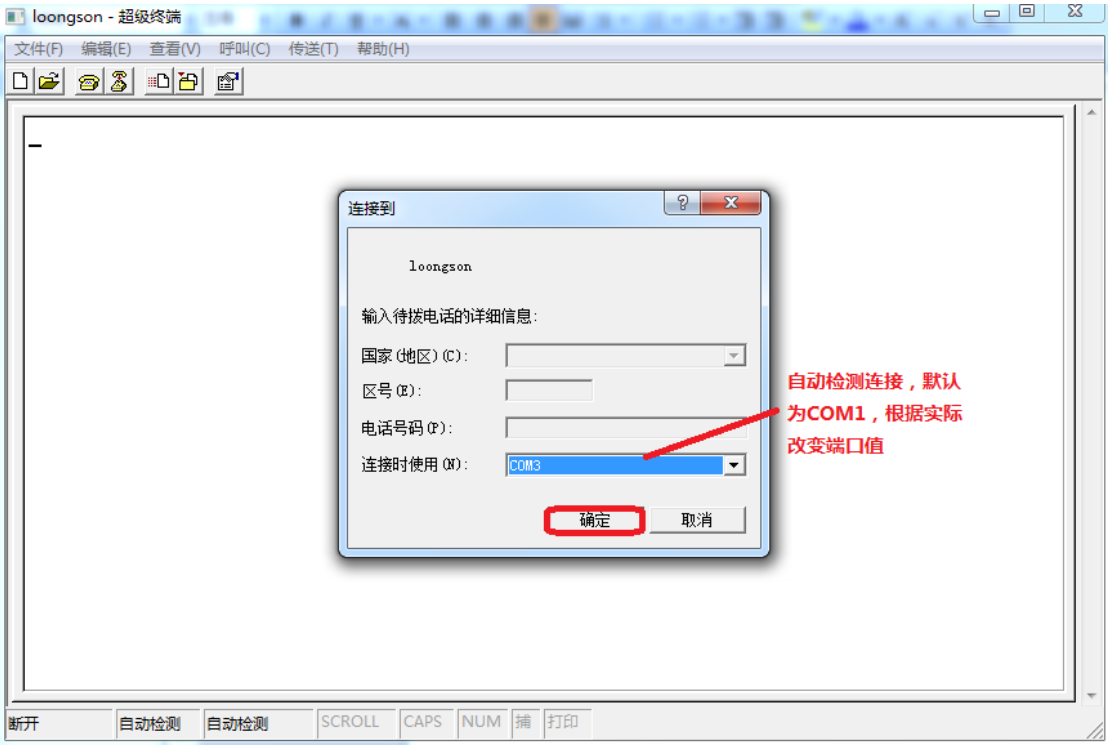

新建连接 - 超级终端
文件(F) 编辑(E) 查看(V) 呼叫(C) 传送(T) 帮助(H)
连接描述
新建连接
输入名称并为该连接选择图标:
名称(N):
图标(I):
确定
取消
断开
自动检测
自动检测
SCROLL
CAPS
NUM
捕
打印

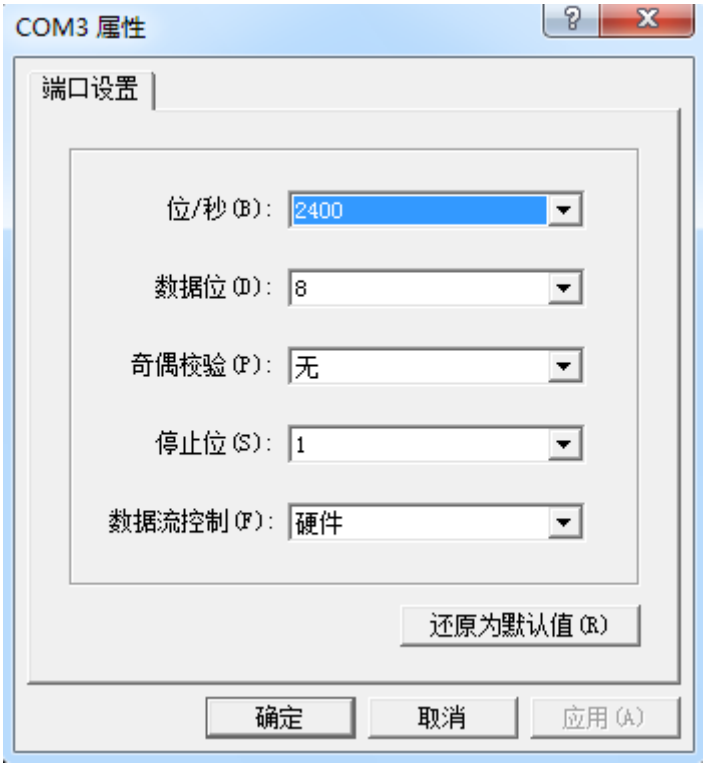
（2）新建连接，修改名称：

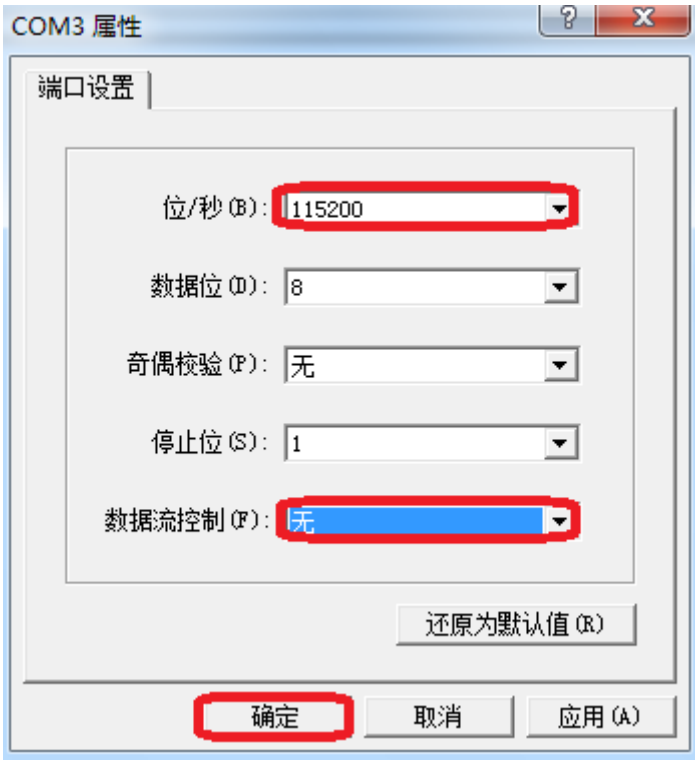

连接描述
新建连接
输入名称并为该连接选择图标:
名称(N):
loongson
图标(I):
确定
取消


位置信息
在您做任何电话或调制解调器连接之前，Windows 需要知道关于您当前位置的信息。
目前所在的国家/地区(W):
中国
您的区号(或城市号)是什么(C)?
如果您想指定一电话公司代码，它是什么(R)?
您拨外线需要先拨哪个号码(O)?
此位置的电话系统使用:
音频拨号(T)
脉冲拨号(P)
确定
取消

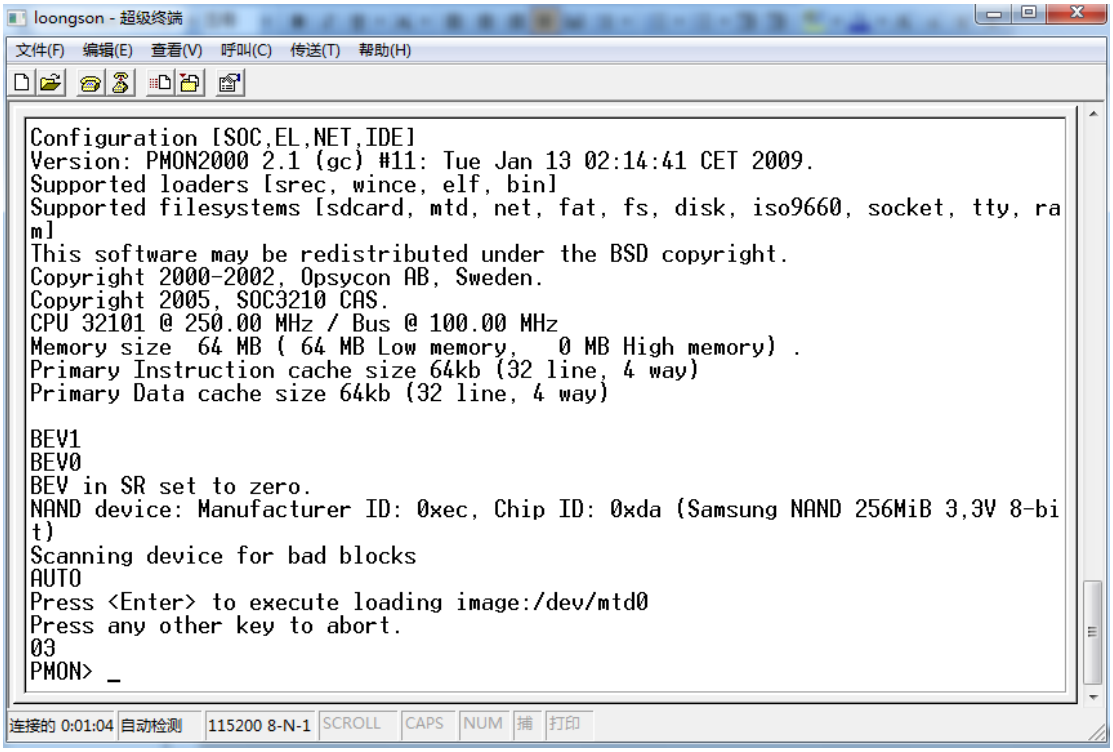
（3）设置：


loongson - 超级终端
文件(F) 编辑(E) 查看(V) 呼叫(C) 传送(T) 帮助(H)
连接到
loongson
输入待拨电话的详细信息:
国家(地区)(C):
区号(E):
电话号码(P):
连接时使用(N):
COM3
确定
取消
自动检测连接，默认为COM1，根据实际改变端口值
断开
自动检测
自动检测
SCROLL
CAPS
NUM
捕
打印


COM3 属性
端口设置
位/秒(B): 2400
数据位(D): 8
奇偶校验(P): 无
停止位(S): 1
数据流控制(F): 硬件
还原为默认值(R)
确定
取消
应用(A)


COM3 属性
端口设置
位/秒(B): 115200
数据位(D): 8
奇偶校验(P): 无
停止位(S): 1
数据流控制(F): 无
还原为默认值(R)
确定
取消
应用(A)

（4）开发板上电：


loongson - 超级终端
文件(F) 编辑(E) 查看(V) 呼叫(C) 传送(T) 帮助(H)
Configuration [SOC,EL,NET,IDE]
Version: PMON2000 2.1 (gc) #11: Tue Jan 13 02:14:41 CET 2009.
Supported loaders [srec, wince, elf, bin]
Supported filesystems [sdcard, mtd, net, fat, fs, disk, iso9660, socket, tty, ram]
This software may be redistributed under the BSD copyright.
Copyright 2000-2002, Opsycon AB, Sweden.
Copyright 2005, SOC3210 CAS.
CPU 32101 @ 250.00 MHz / Bus @ 100.00 MHz
Memory size 64 MB ( 64 MB Low memory, 0 MB High memory) .
Primary Instruction cache size 64kb (32 line, 4 way)
Primary Data cache size 64kb (32 line, 4 way)
BEV1
BEV0
BEV in SR set to zero.
NAND device: Manufacturer ID: 0xec, Chip ID: 0xda (Samsung NAND 256MiB 3,3V 8-bit)
Scanning device for bad blocks
AUTO
Press <Enter> to execute loading image:/dev/mtd0
Press any other key to abort.
03
PMON> _
连接的 0:01:04 自动检测 115200 8-N-1 SCROLL CAPS NUM 捕 打印

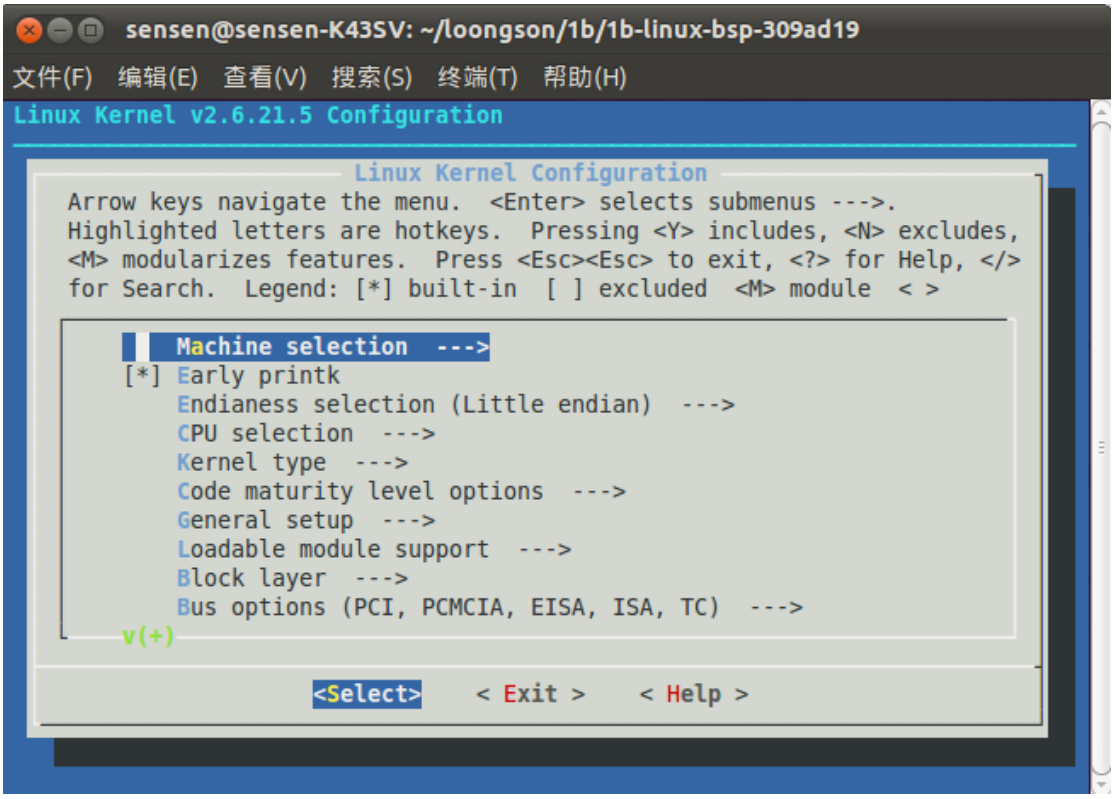
# 附录 4　内核配置详细说明

## 4.1 各个驱动程序源代码位置

- **GMAC** 网卡驱动
  1b-linux-3.0/drivers/net/sb2f_gmac/synopGMAC_network_interface.c
- 串口
  1b-linux-3.0/drivers/8250.c
- 实时时钟 **RTC** 驱动
  1b-linux-3.0/drivers/ rtc-gs2fsb.c
- 按键驱动
  1b-linux-3.0/drivers/input/keyboard/74LV165_button.c
- 触摸屏驱动
  1b-linux-3.0/drivers/i2c/chips/tsc2003.c
- **USB** 鼠标、键盘源代码
  1b-linux-3.0/drivers/usb/host/ehci-sb2f.c 与 ohci-sb2f.c
- 优盘支持驱动
  1b-linux-3.0/drivers/usb/storage
- **SD** 卡驱动源代码目录
  1b-linux-3.0/drivers/block/sb2fsd.c 与 sb2fsb_spi.c
- **Nand Flash** 驱动
  1b-linux-3.0/drivers/mtd/nand/sb2f-nand.c
- **AC97** 音频驱动目录
  1b-linux-3.0/sound/oss/sb2f-ac97.c
- **LCD** 驱动
  1b-linux-3.0/drivers/video/ sb2f-fb.c
- **I2C-EEPROM** 驱动
  1b-linux-3.0/drivers/i2c/
- 蜂鸣器驱动
  1b-linux-3.0/drivers/char/buzzer-bsp.c
- 看门狗驱动
  1b-linux-3.0/drivers/char/watchdog/gs2fsb_wdt.c
- **AD** 转换驱动
  1b-linux-3.0/drivers/spi/mcp3201.c
- **PWM** 驱动
  1b-linux-3.0/drivers/char/ls1b-pwm.c

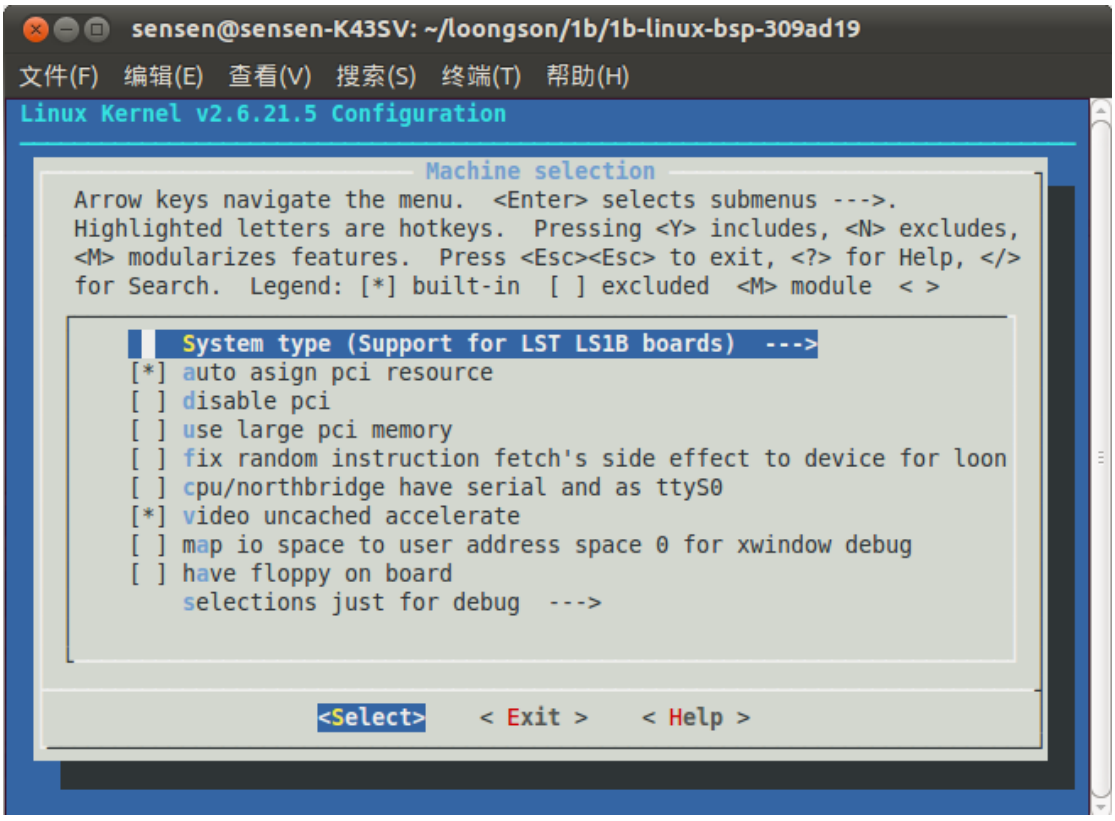
## 4.2 手工定制 Linux 内核

运行 make menuconfig 后，进入内核配置主菜单 Linux Kernel Configuration，如下图：

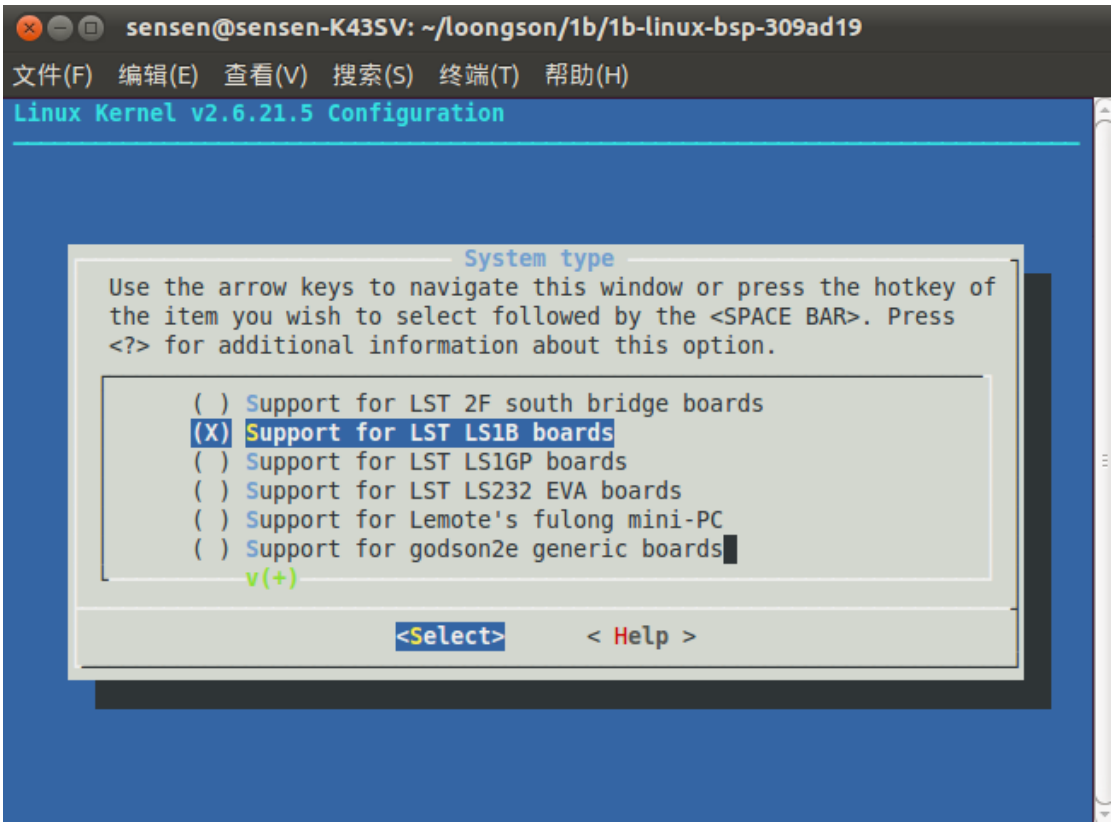



### 4.2.1 配置 CPU 平台选项

在主菜单里面,选择 Machine Selection,按回车进入，然后选择 System type 选项进入：

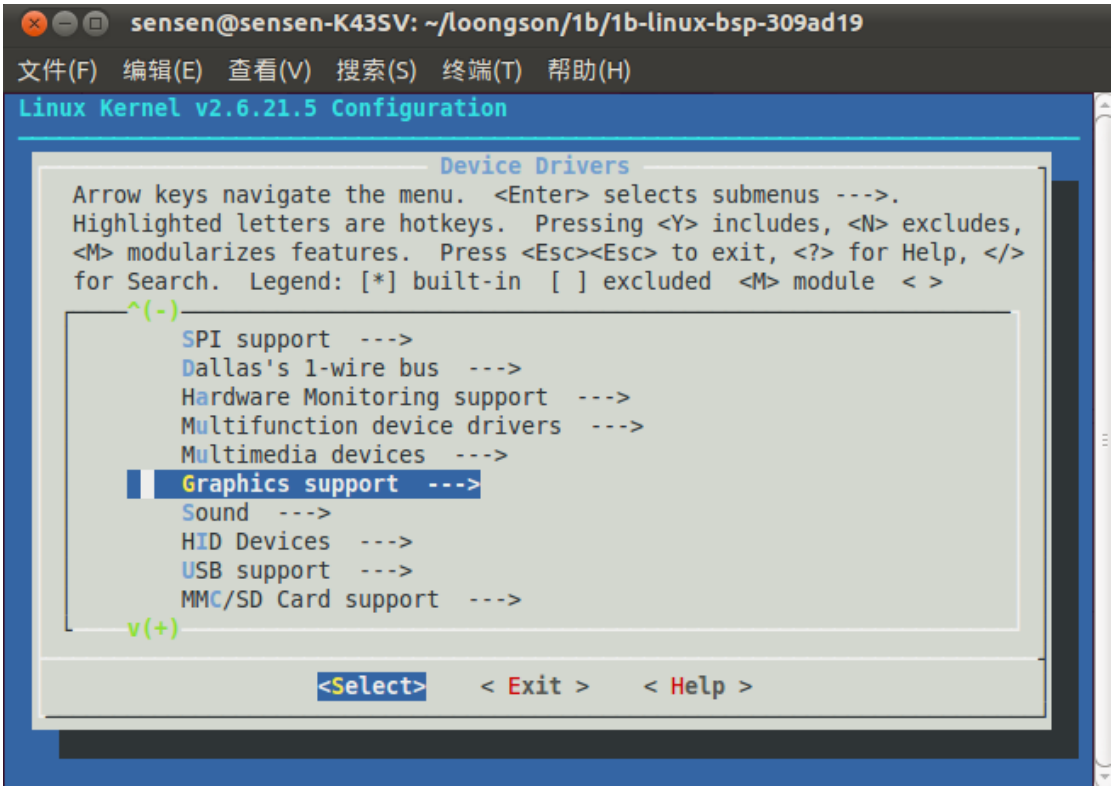



在此可以看到对不同系列的板的支持的选项，使用下文箭头进行导航，选择 Support for LST LS1B boards 选项。

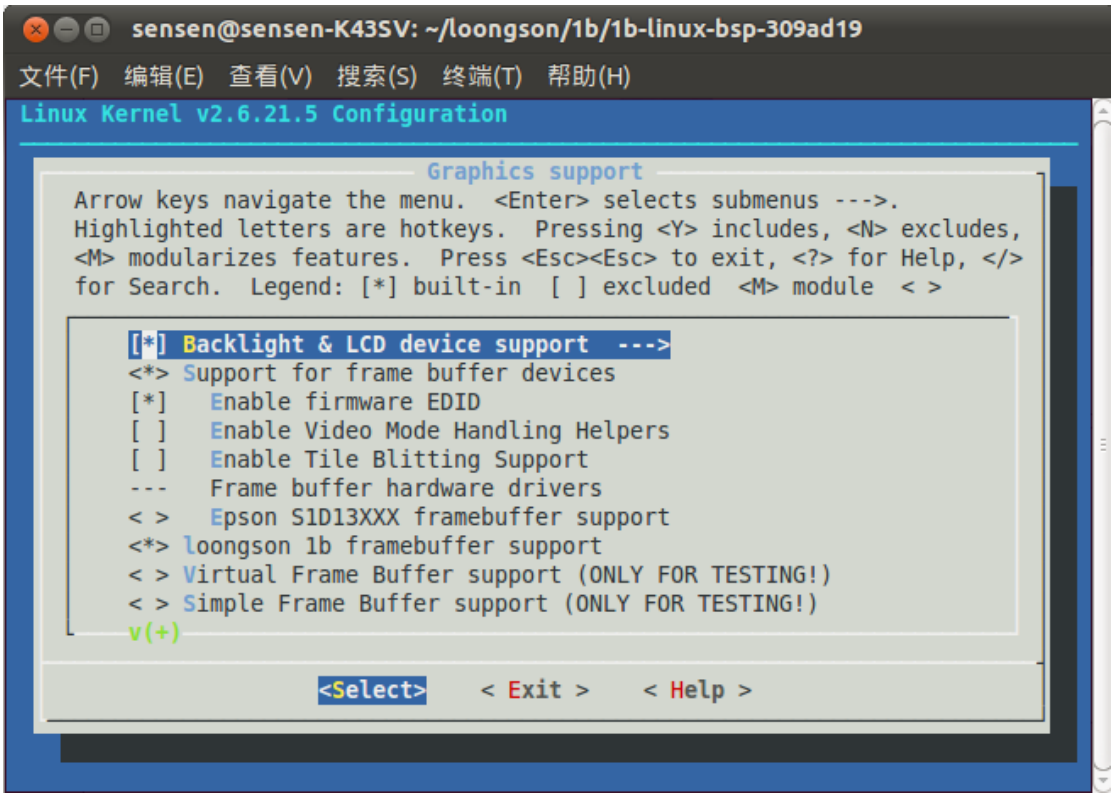


### 4.2.2 配置各个尺寸的 LCD 驱动以及背光控制支持

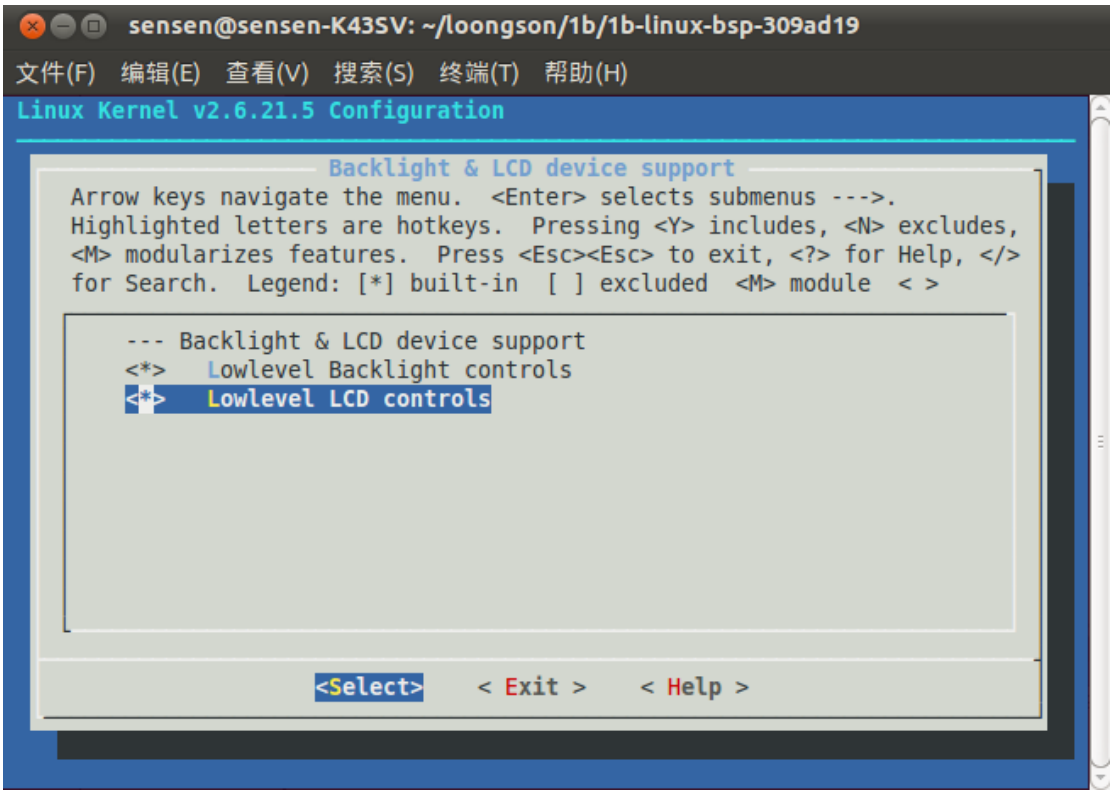
在主菜单里面,选择 Device Drivers,按回车进入,并找到如图选项,按回车进入:




找到如图选项,再按回车进入:



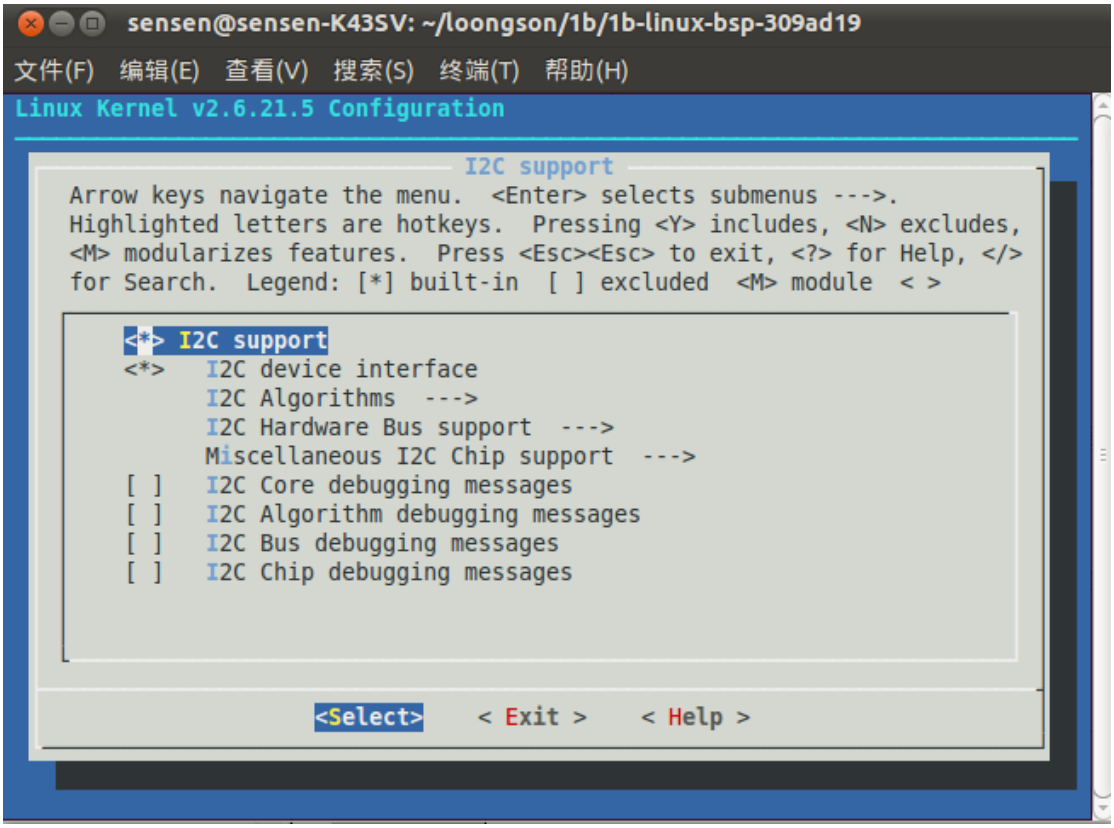
出现类似如图界面,并找到如图选项,选中如图 Backlight(背光控制)和其它用(*)号标注的选项后，选择 Backlight 选项并进入：



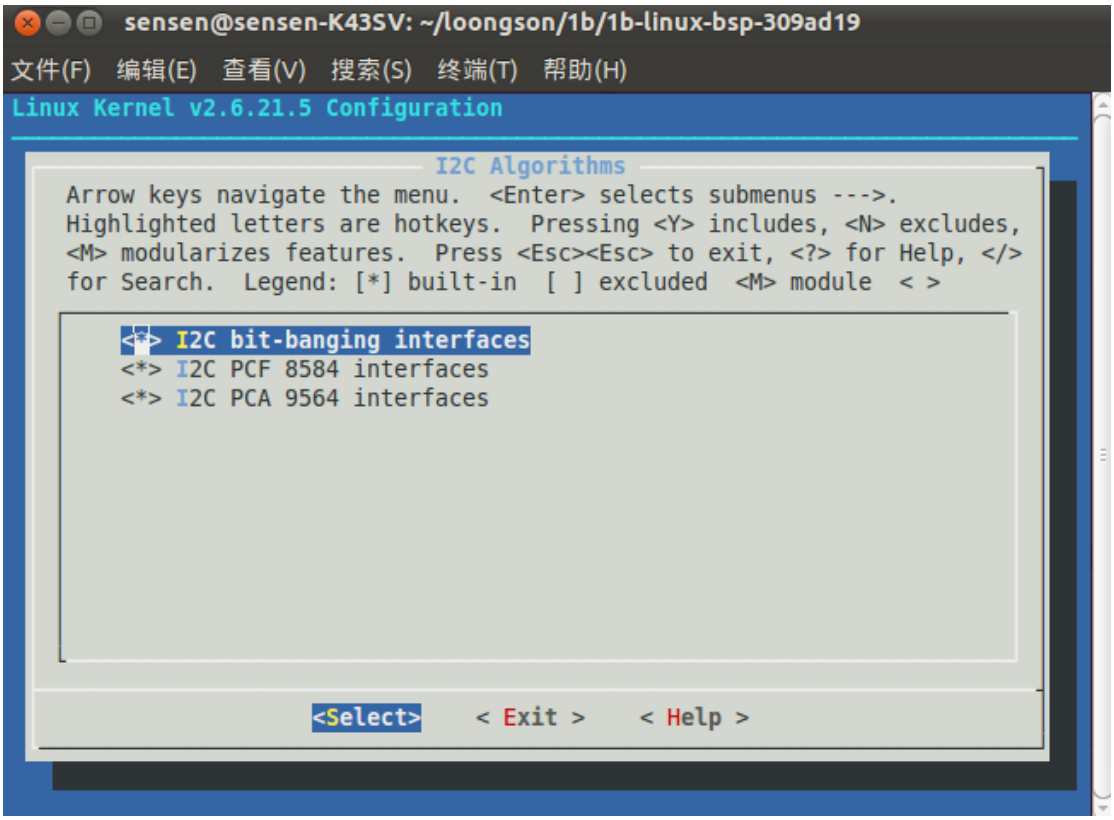

选择完毕,一直按照下方的提示返回到 Device Drivers 配置菜单。

### 4.2.3 配置触摸屏

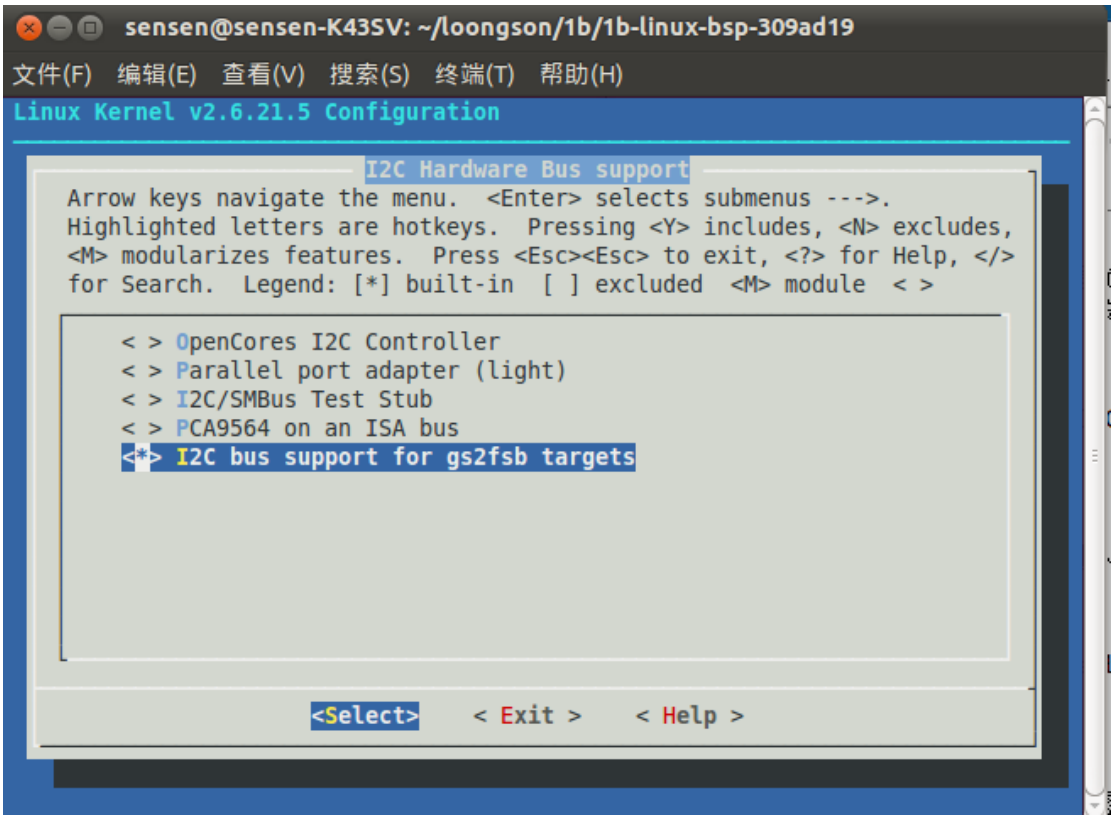
在 Device Drivers 菜单里面,选择 I2C support --->，按回车进入，按照下图进行选择后：



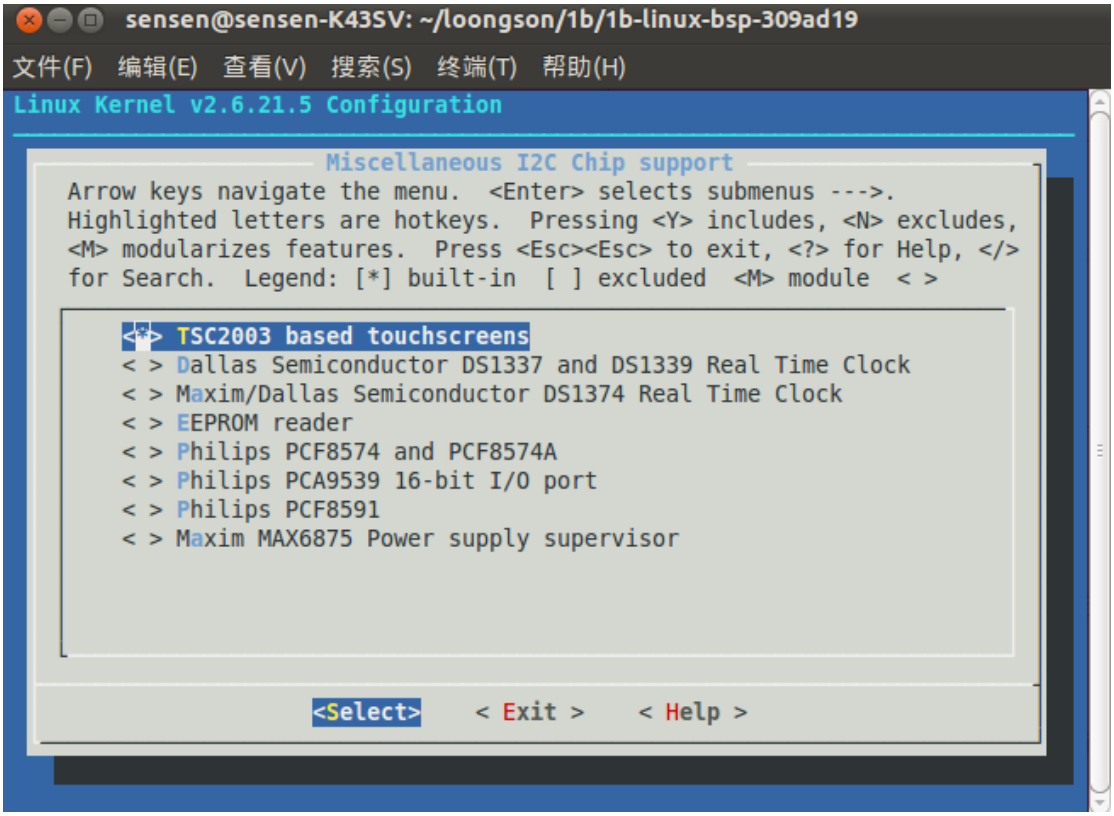

选择 I2C Algorithms--->，按回车进入，按照下图进行选择：




按<Exit>返回上一层，选择 I2C Hardware Bus support，按回车进入，按照下图选择：



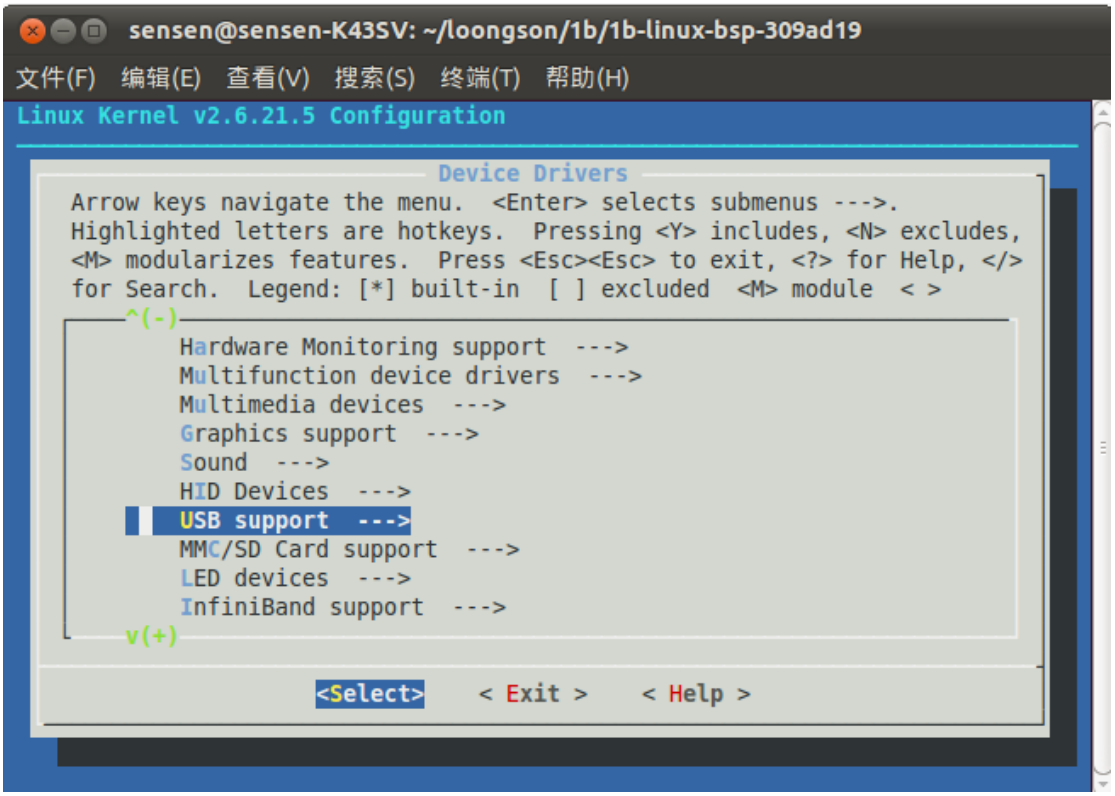

按<Exit>返回上一层，选择 Miscellaneous I2C Chip support --->，按回车进入，按照下图选择：



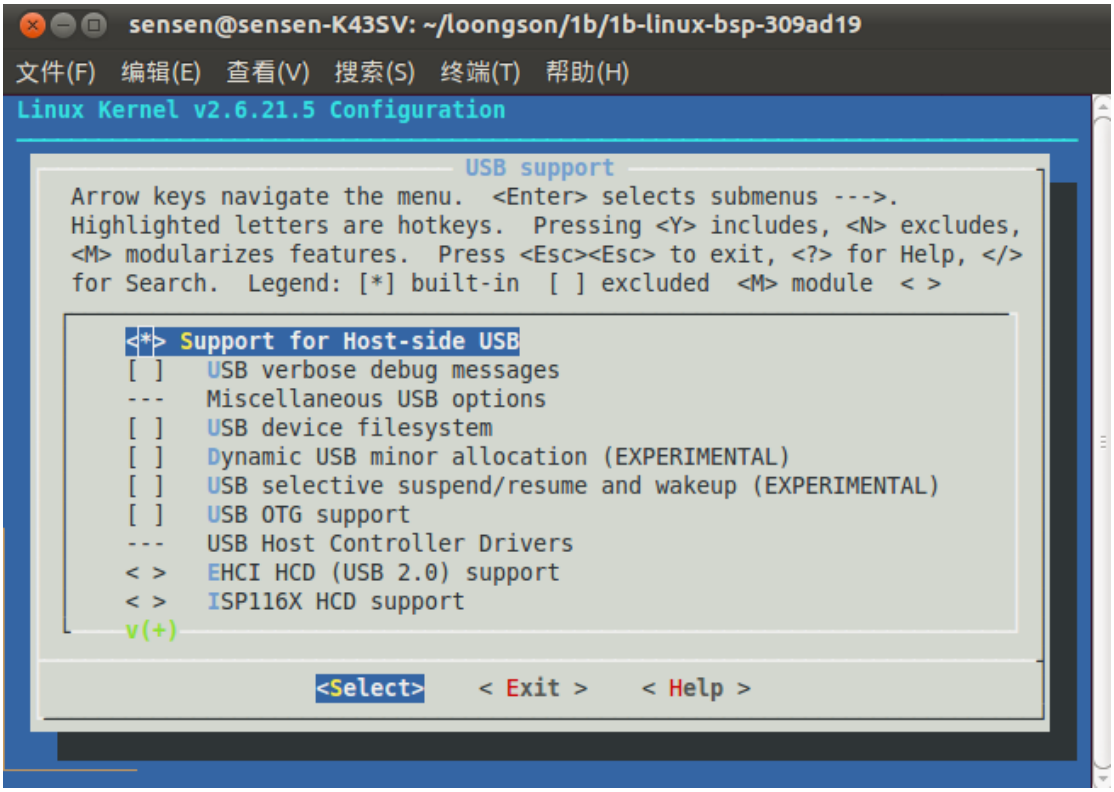
然后选择<Exit>返回 Device Drivers 菜单。

### 4.2.4 配置 USB 鼠标和键盘

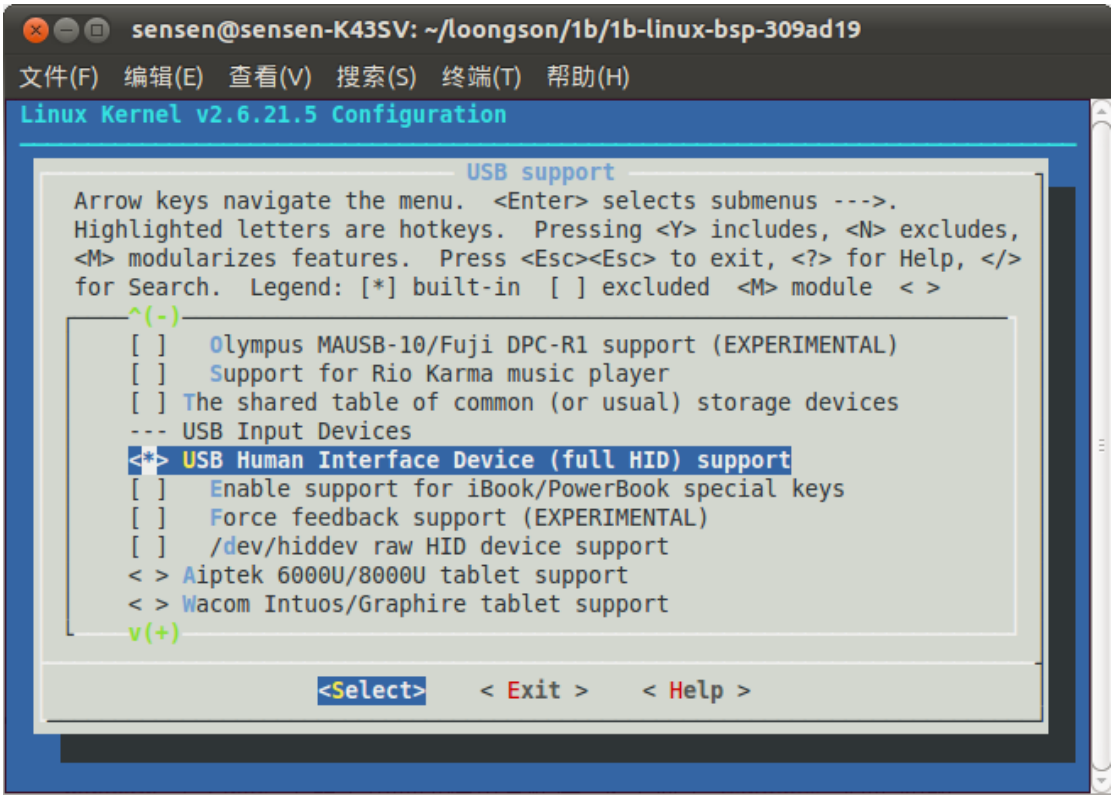
在 Device Drivers 菜单里面,找到如图选项,并选择进入：




选择如图“*”号所指示的选项：




在其弹出的下拉列表中选中如图“*”号所示的选项：

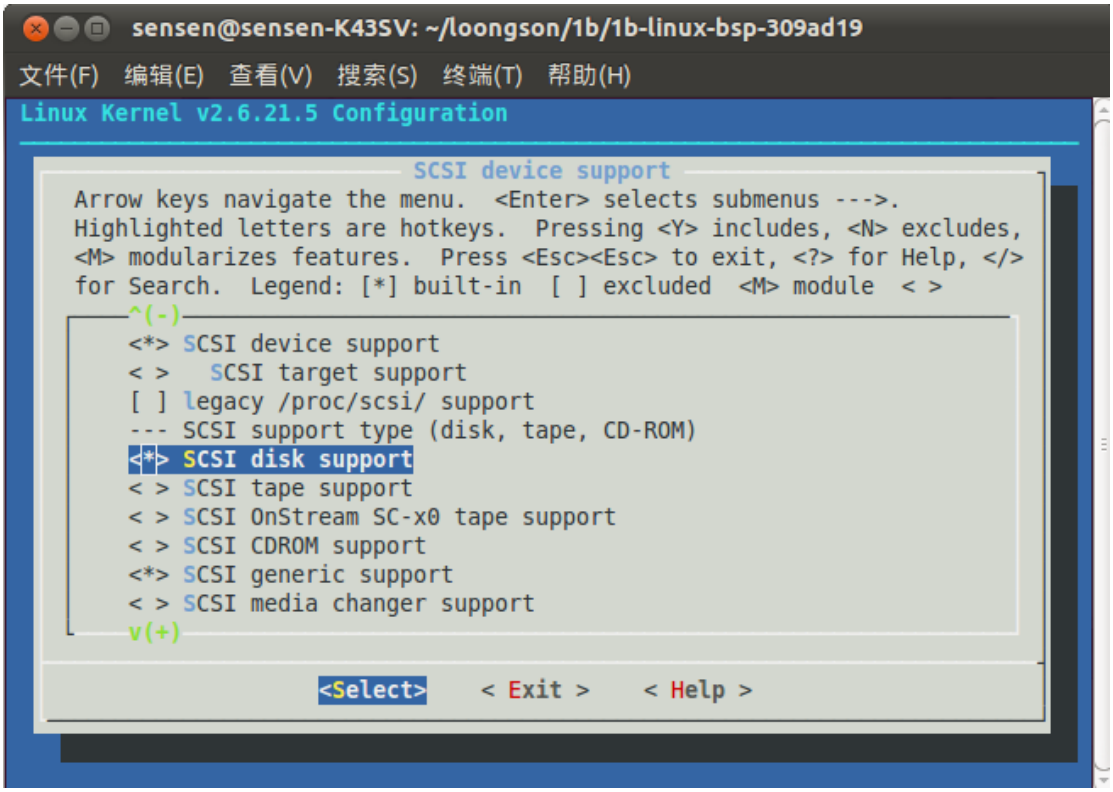



这样就选择配置了 USB 键盘和鼠标,然后选择<Exit>返回 Deice Drivers 菜单。

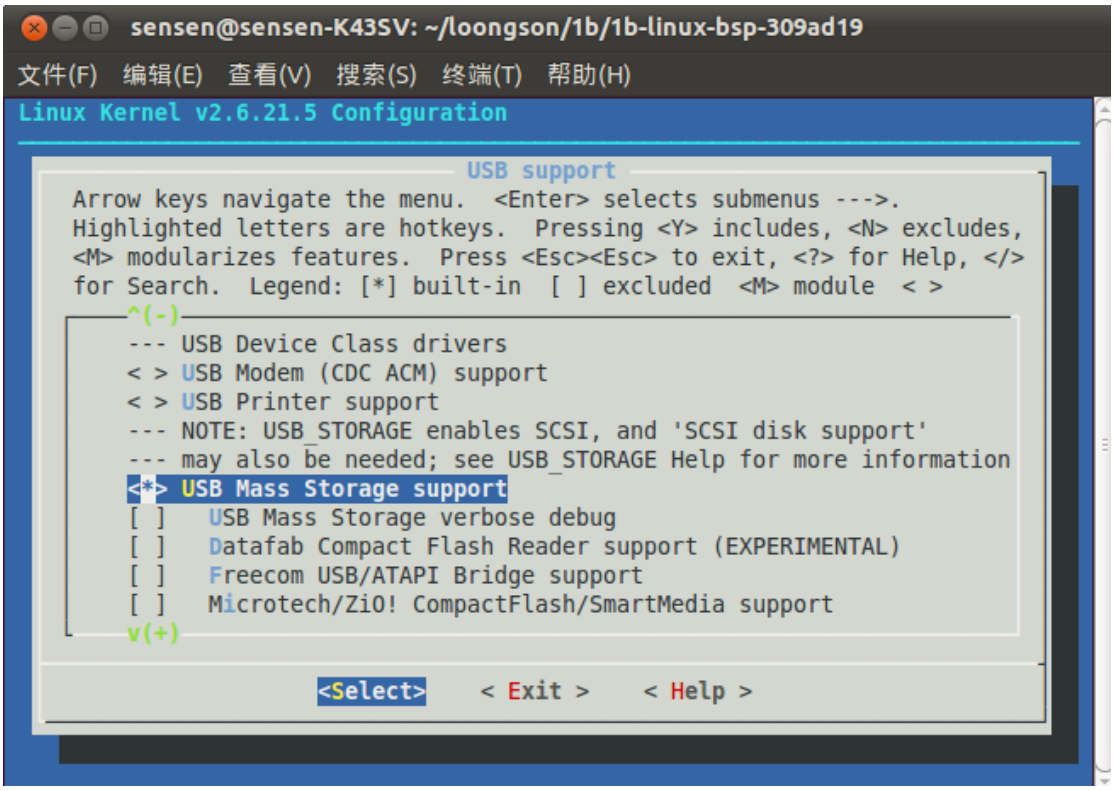
### 4.2.5 如何配置优盘的支持

因为优盘用到了 SCSI 命令,所以我们先增加 SCSI 支持。

在 Device Drivers 菜单里面,选择 SCSI device support,按回车进入：



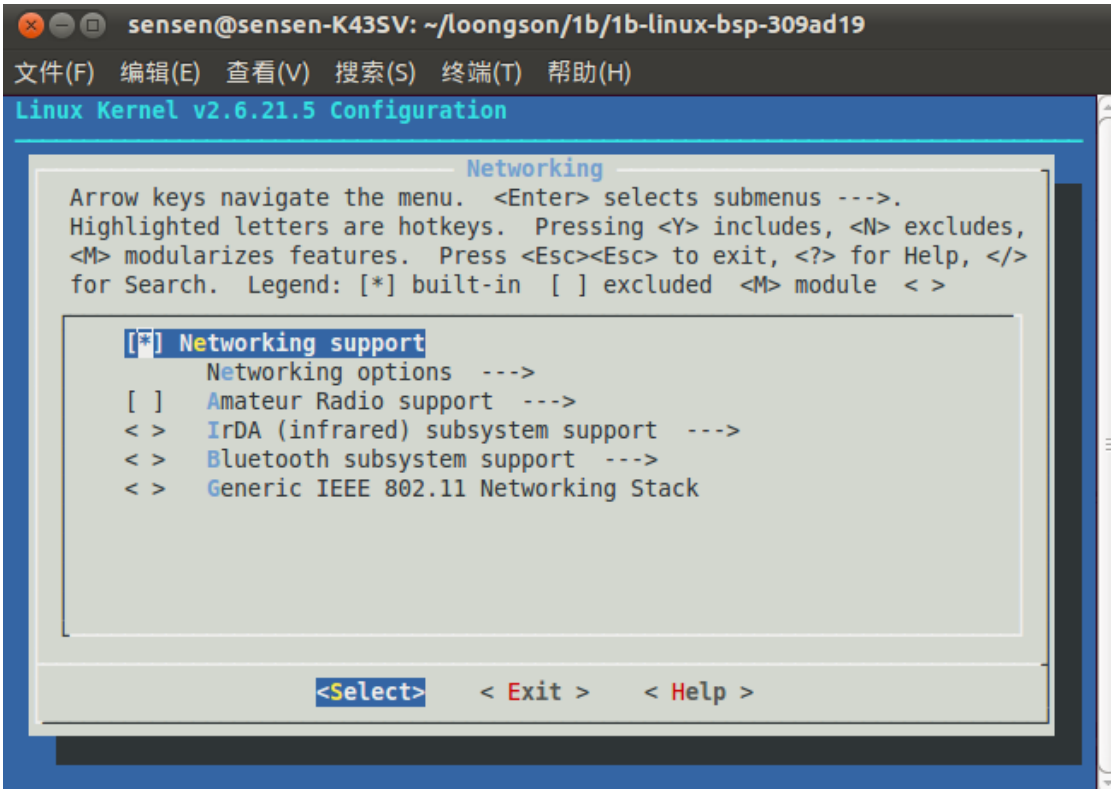

返回 Device Drivers 菜单,再选择 USB support,按回车进入 USB support 菜单找到并选中 <*> USB Mass Storage support 。



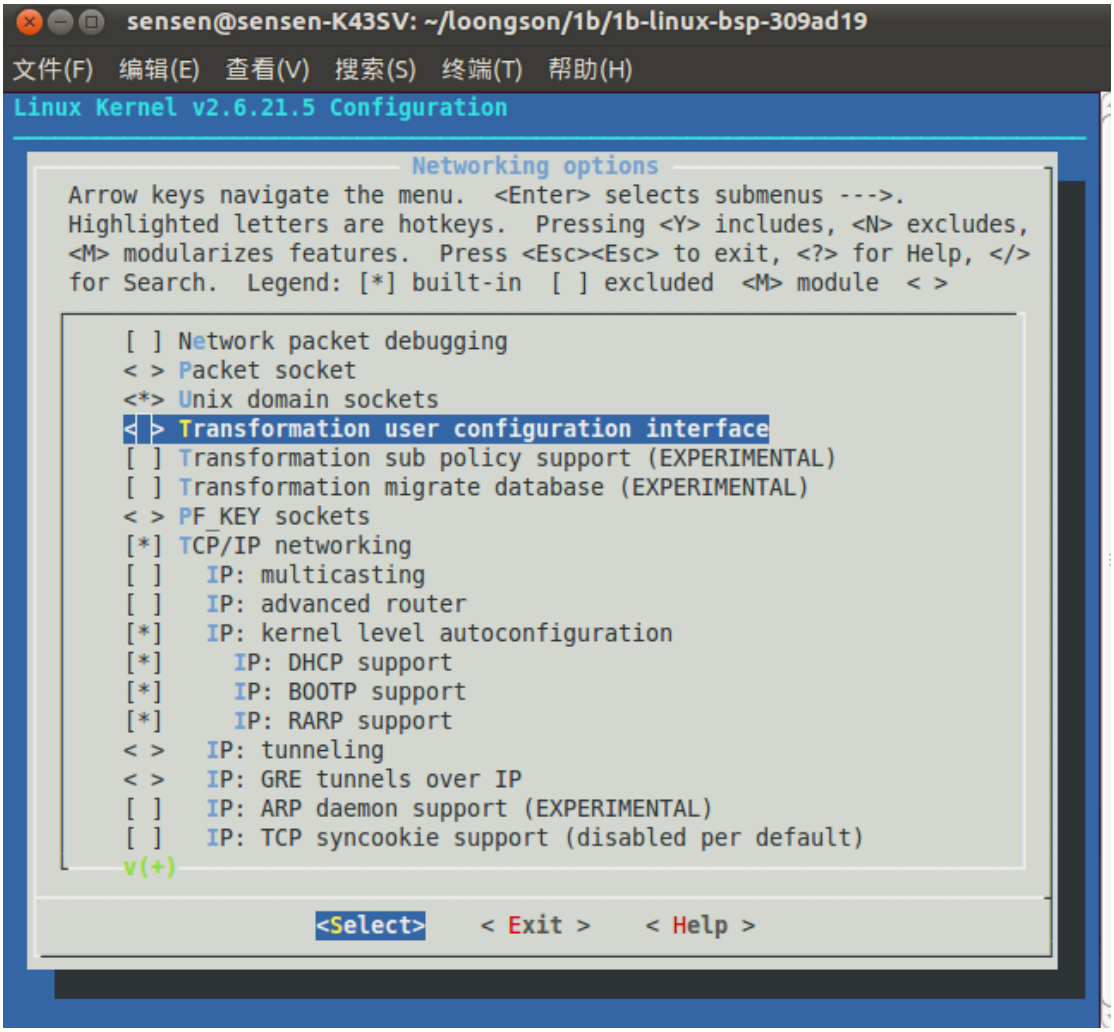

然后选择<Exit>返回 Device Drivers 菜单。

### 4.2.6 配置网卡驱动

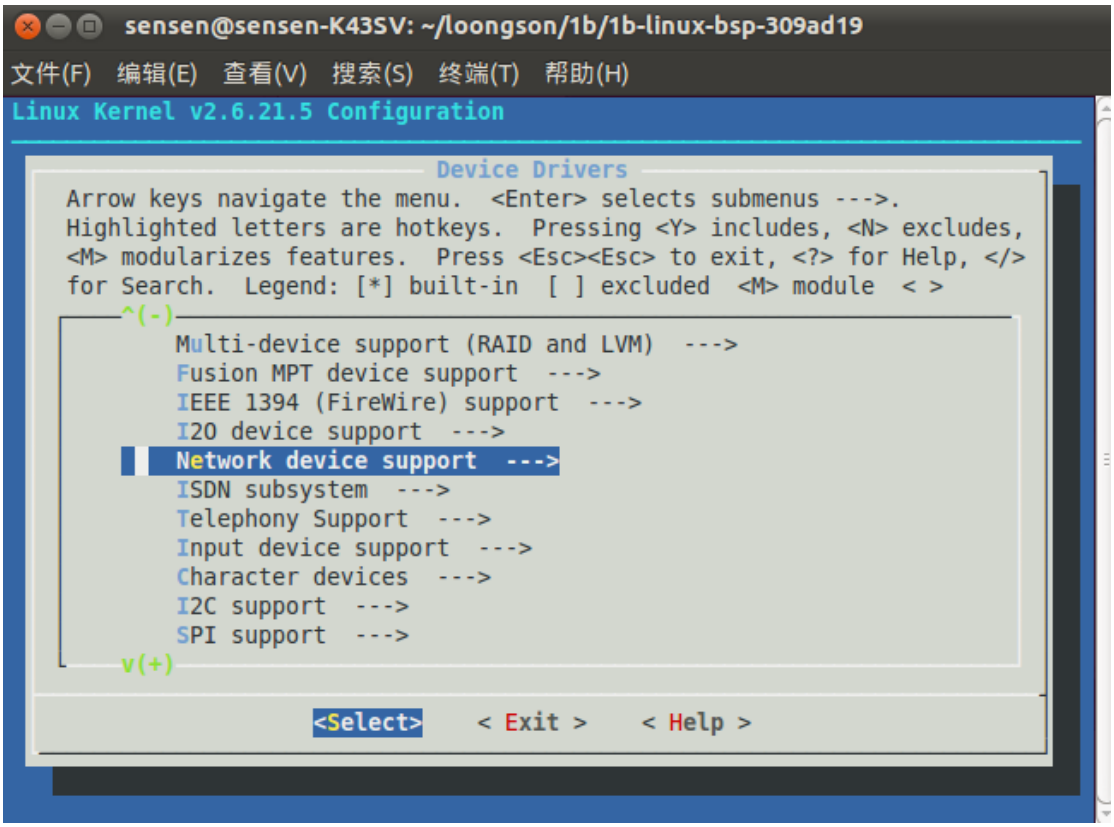
要配置网卡驱动,首先要配置网络协议支持，在主菜单中,选择 Netwoking,回车进入并选中 Networking support 选项：




出现如图子菜单,如图选择 Networking options 并进入，一般我们选择 TCP/IP 协议就够了，但推荐使用我们缺省配置的几个选项,如图：



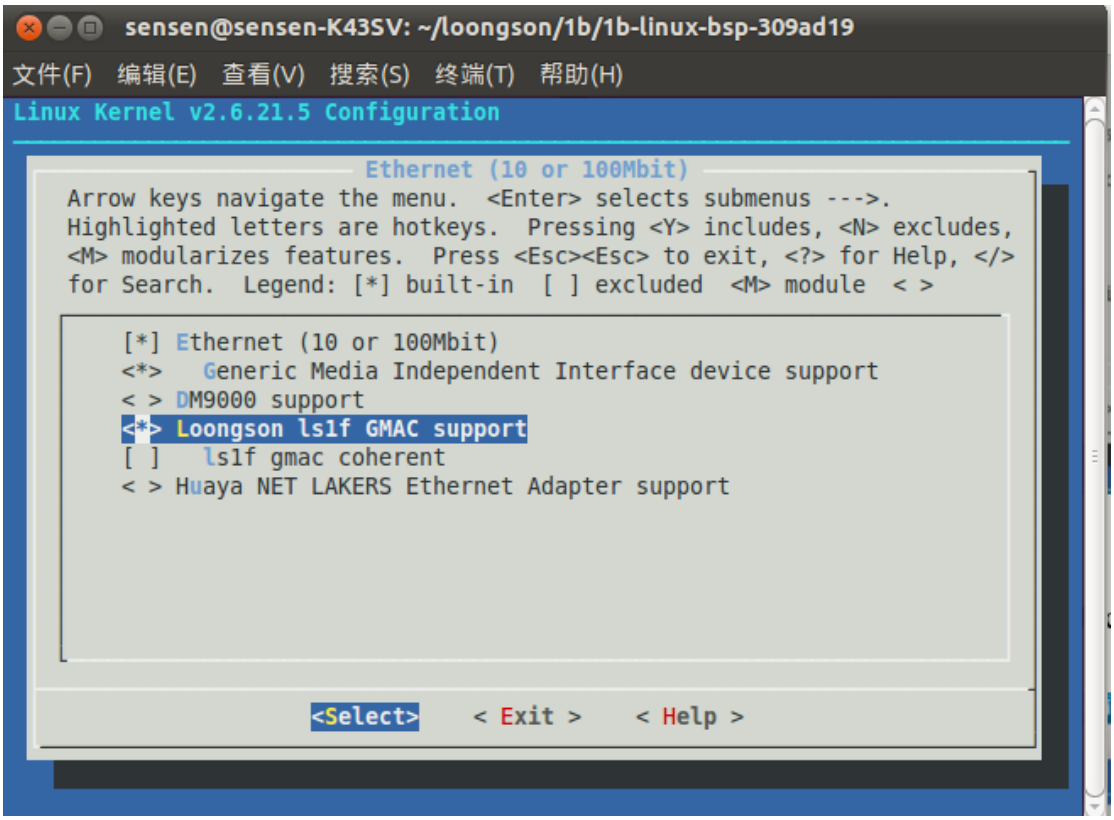

选择完毕，一直退回到主菜单，并选择进入 Device Drivers 菜单。找到 Network device support，选择进入：




找到并进入 Ethernet (10 or 100Mbit) 选项并选中：

**<*> Generic Media Independent Interface device support**

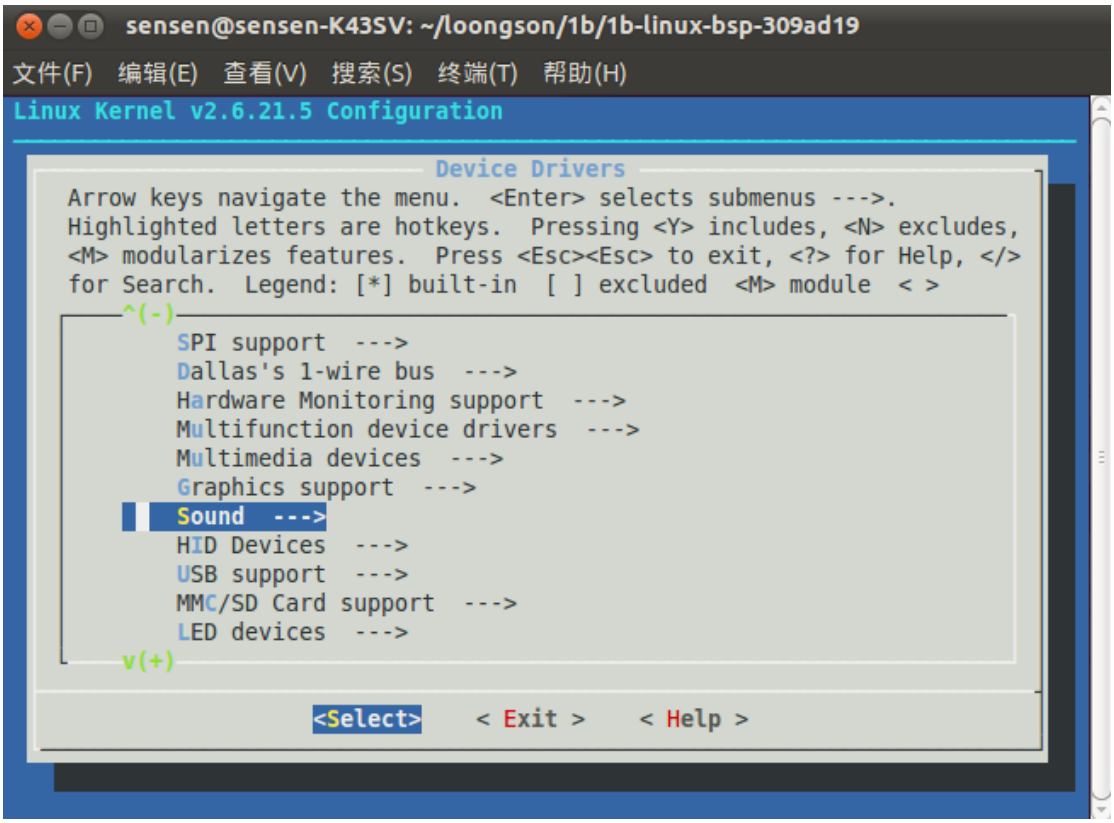
**<*> Loongson ls1f GMAC support**



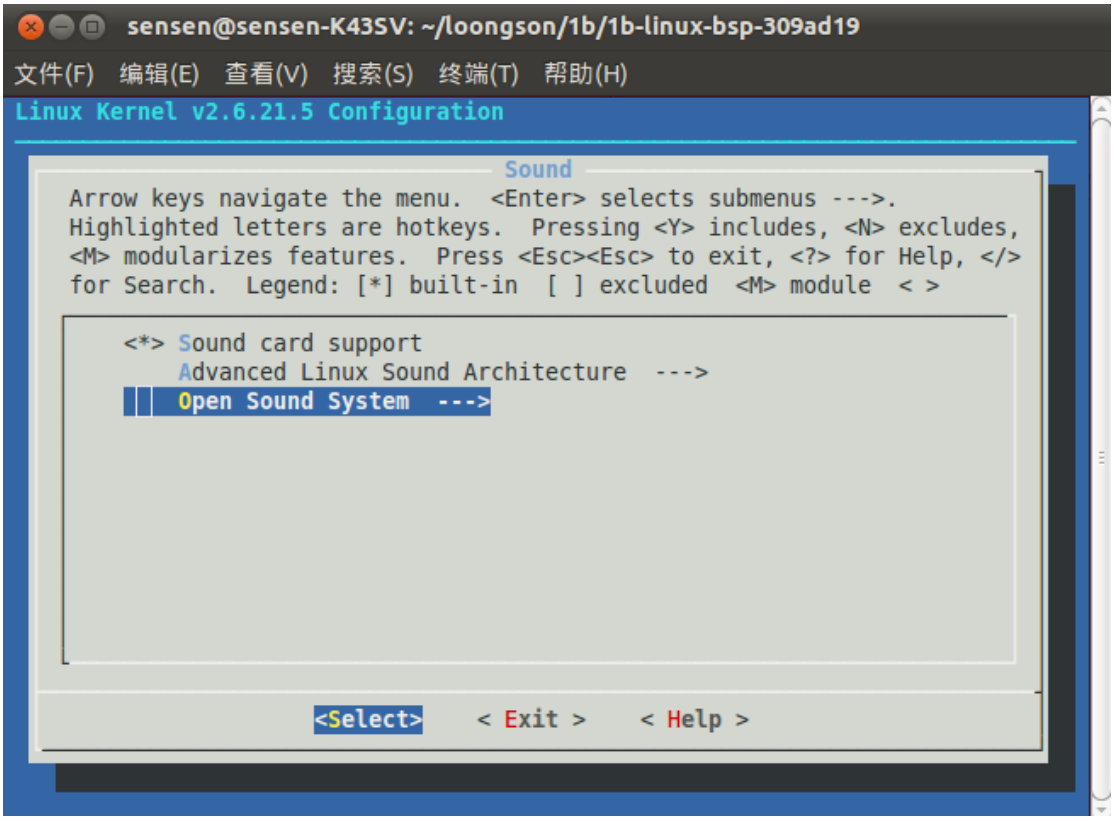
选择<Exit>一直返回到 Device Drivers 菜单。

### 4.2.7 配置音频驱动

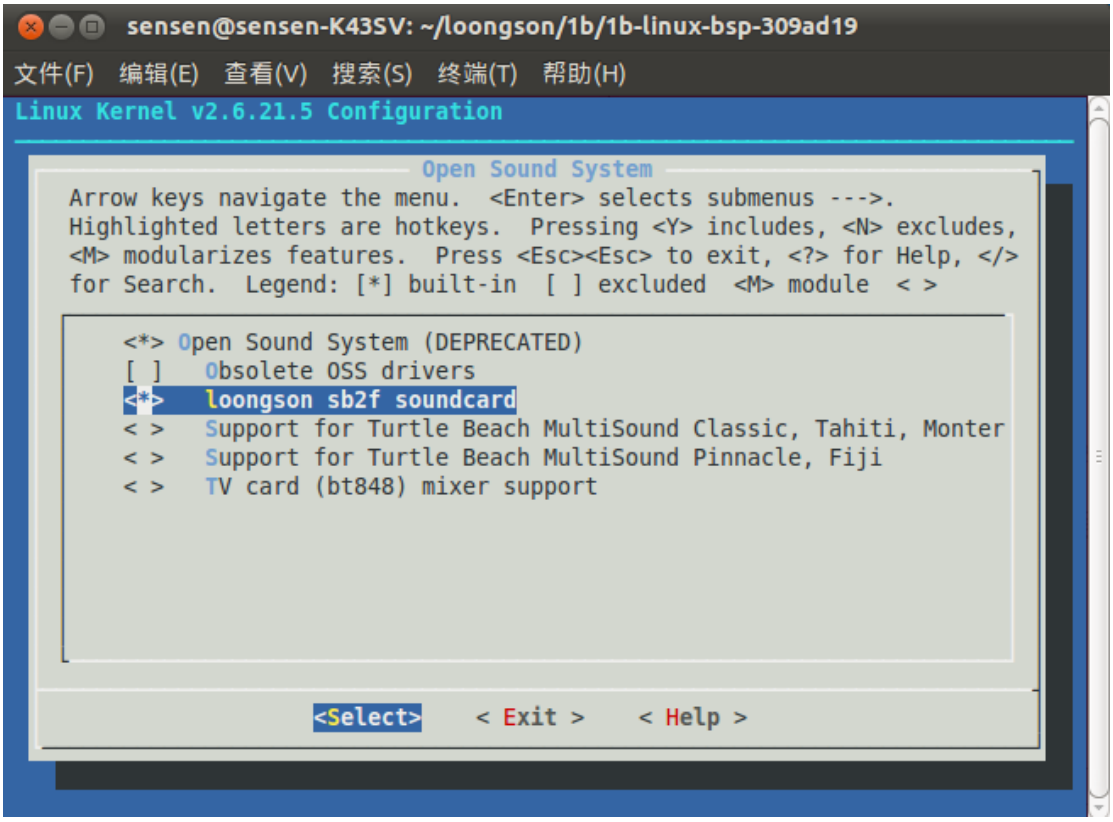
在 Device Drivers 菜单中，选择 sound,并进入：




选择 Sound card supprt，并在下拉列表里选择 Open Sound System 并进入：



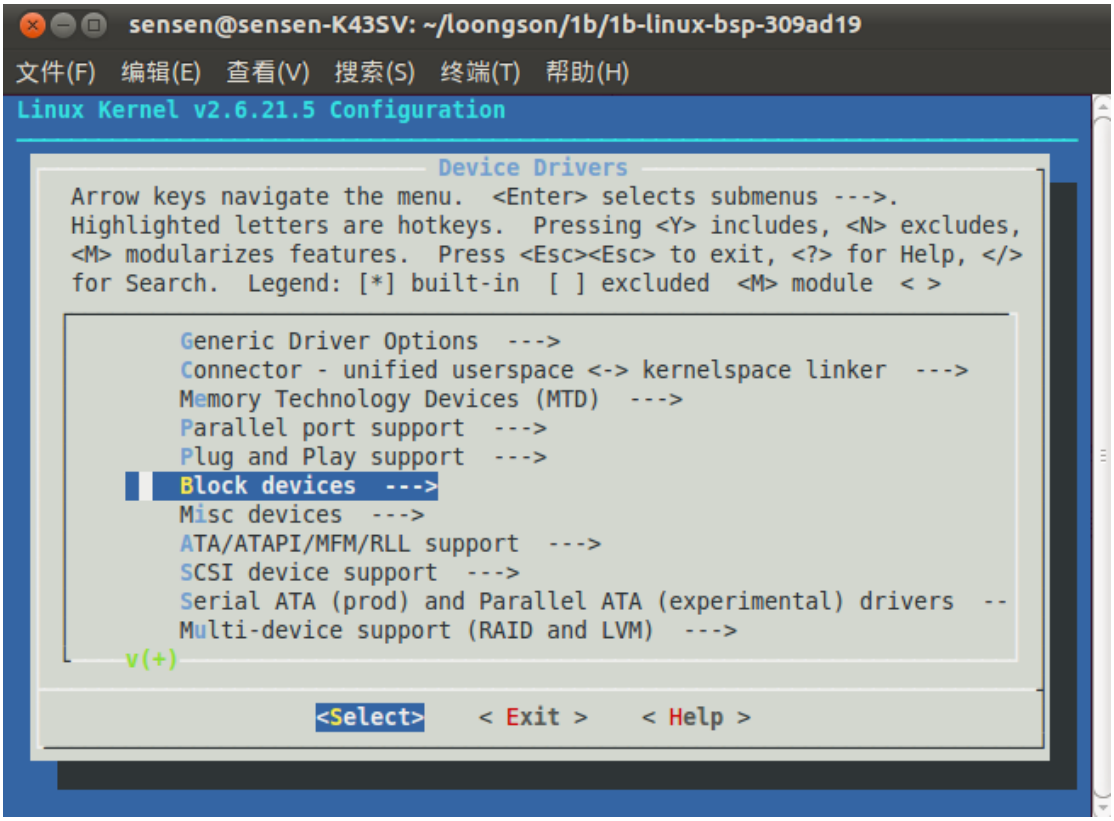

选择 Open Sound System (DEPRECATED)并在下拉列表里选择 loongson sb2f soundcard，如图所示：



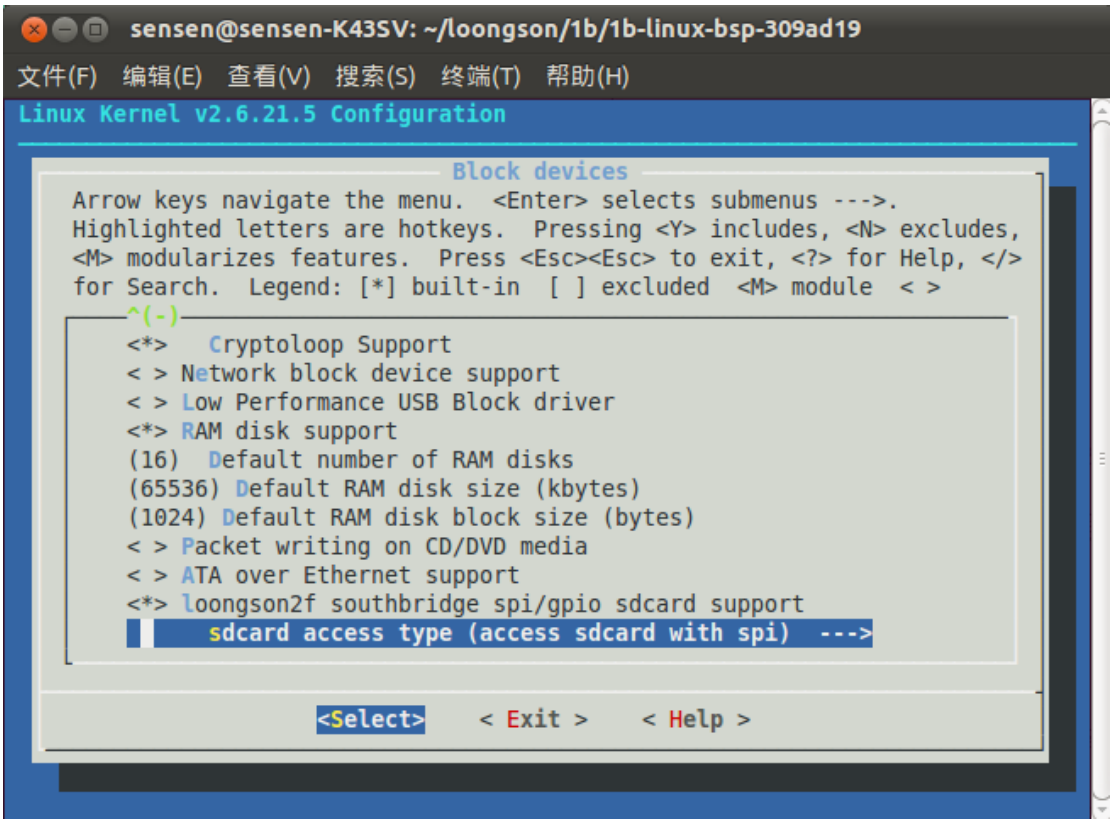

选择完毕,一直按<Exit>返回到 Device Drivers 菜单。

### 4.2.8 配置 SD 卡驱动

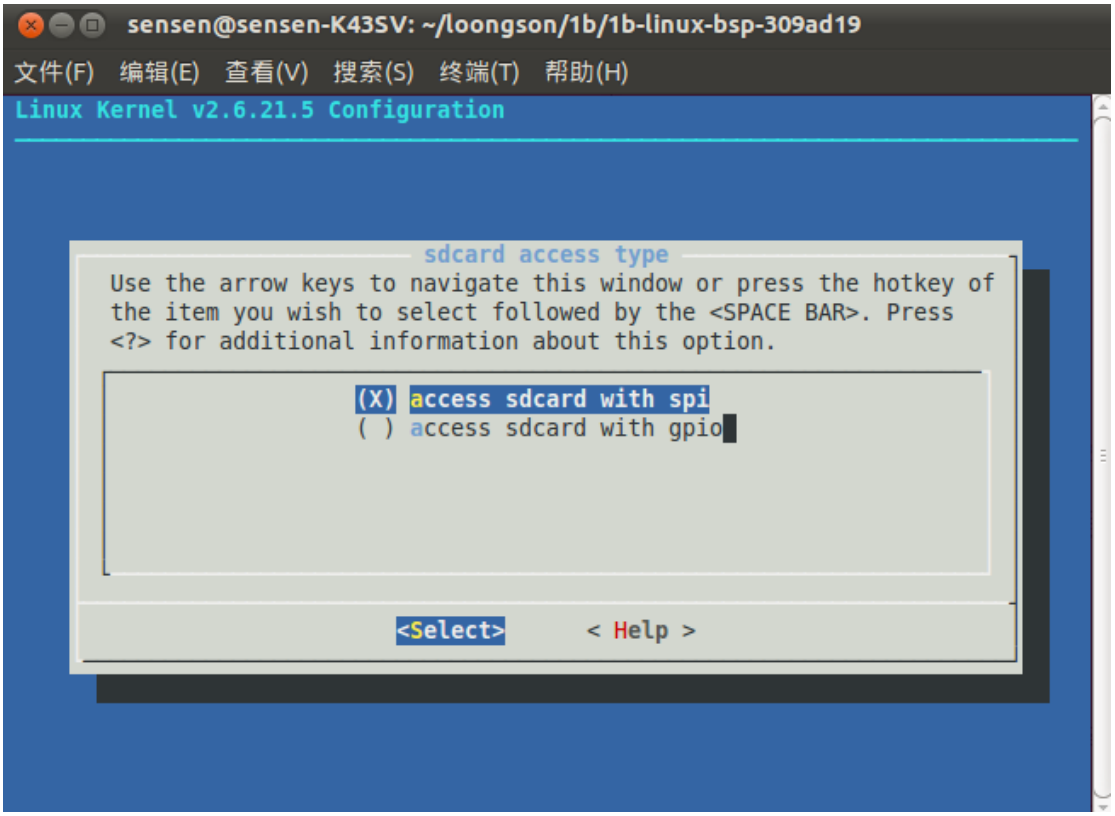
在 Device Drivers 菜单中，选择 Block devices---> 设备选项并按回车进入：




在打开的 Block devices 选项列表中选中 loongson2f southbridge spi/gpio sdcard support，并选择 sdcard access type (access sdcard with spi)项：



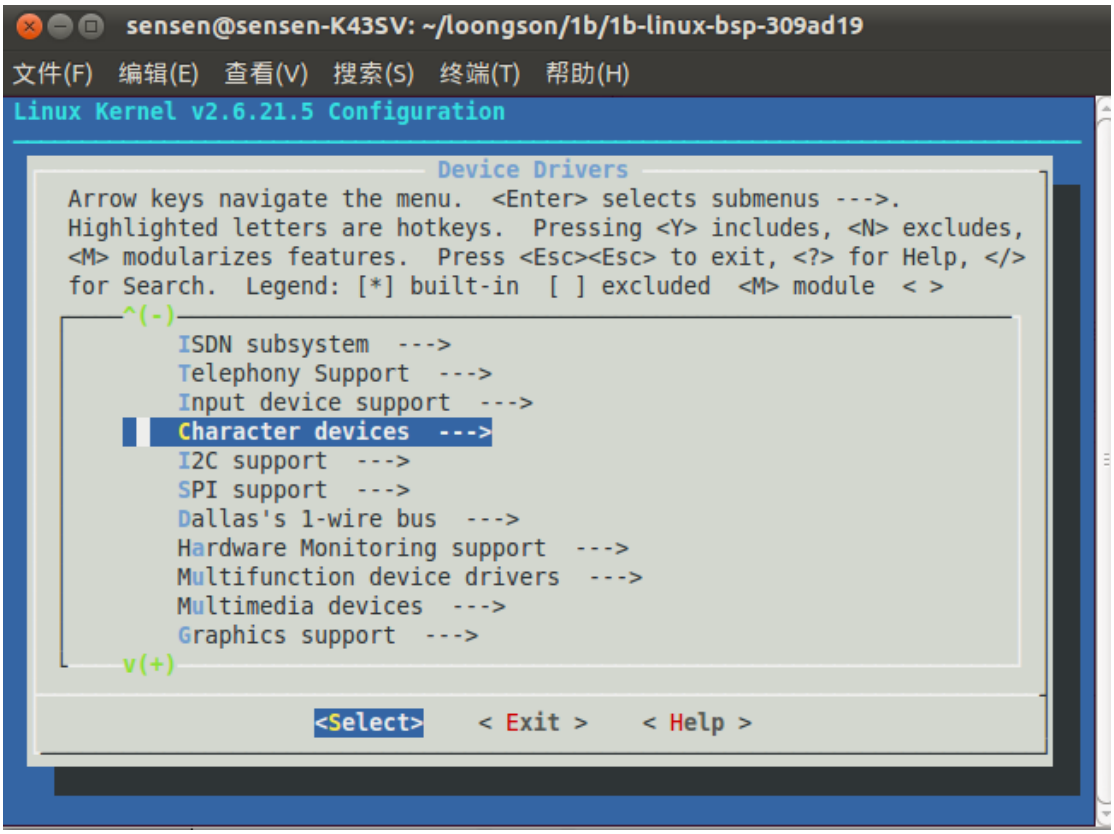

在进入后，选中 access sdcard with spi：



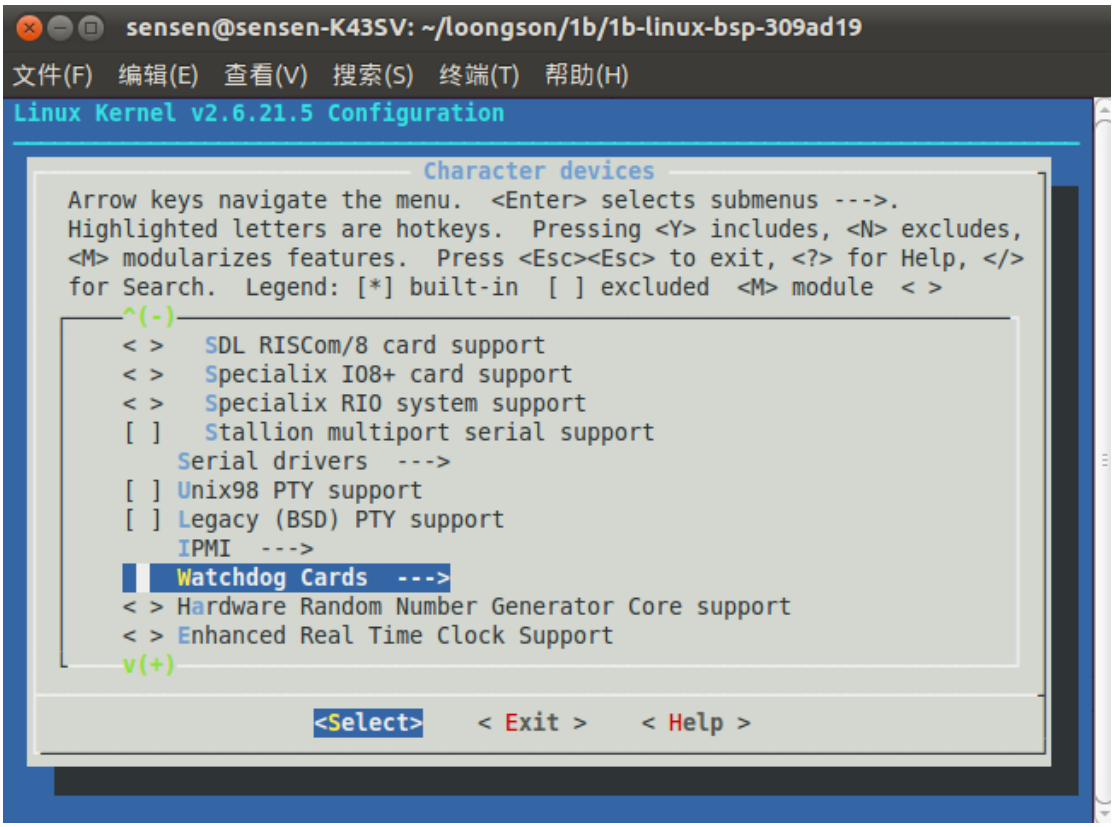

选择完毕,一直按<Exit>返回到 Device Drivers --->菜单。

### 4.2.9 配置看门狗驱动支持

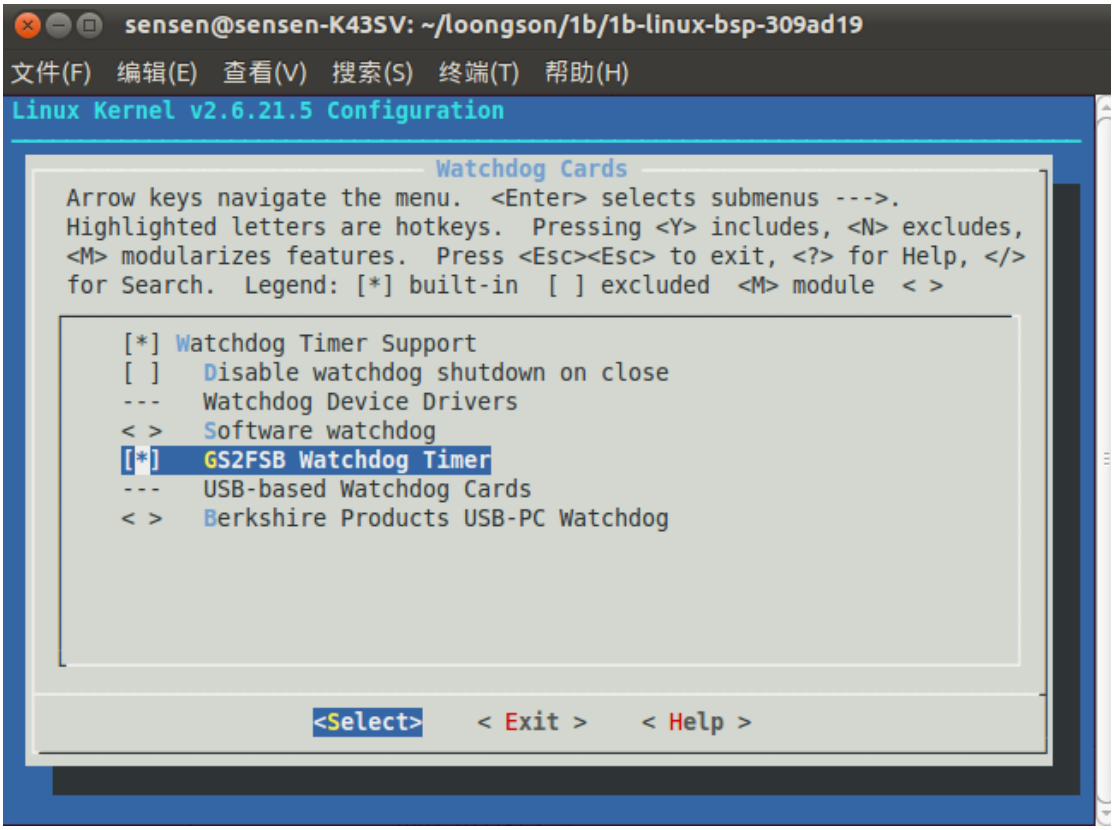
在 Device Drivers 菜单中,选择 Character devices---> 选项并按回车进入：




接着在打开的选项中选中 Watchdog Cards 选项并进入：



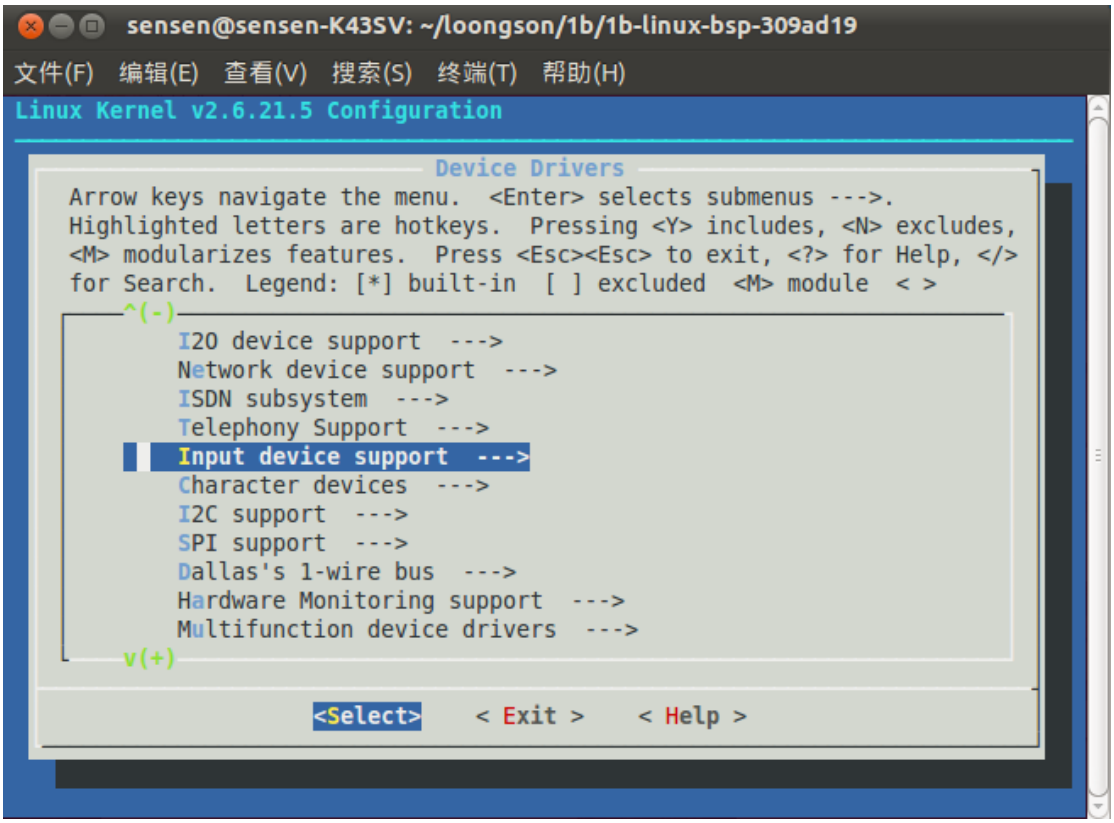

先选中 Watchdog Timer Support 选项，再选中如图所示看门狗驱动支持：



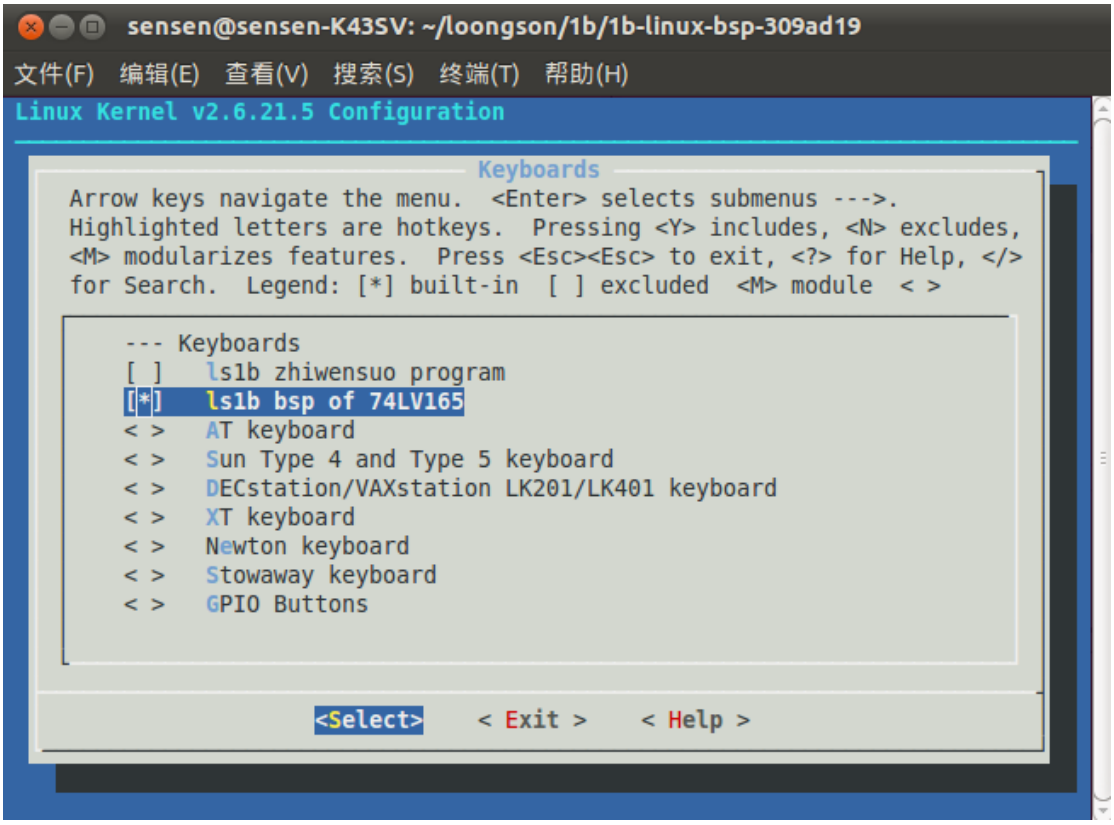

按<Exit>返回到 Device Drivers 菜单。

### 4.2.10 配置按键驱动

在 Device Drivers 菜单中，选择进入 Input device support --->，找到并选中 Keyboards 驱动支持，如图。



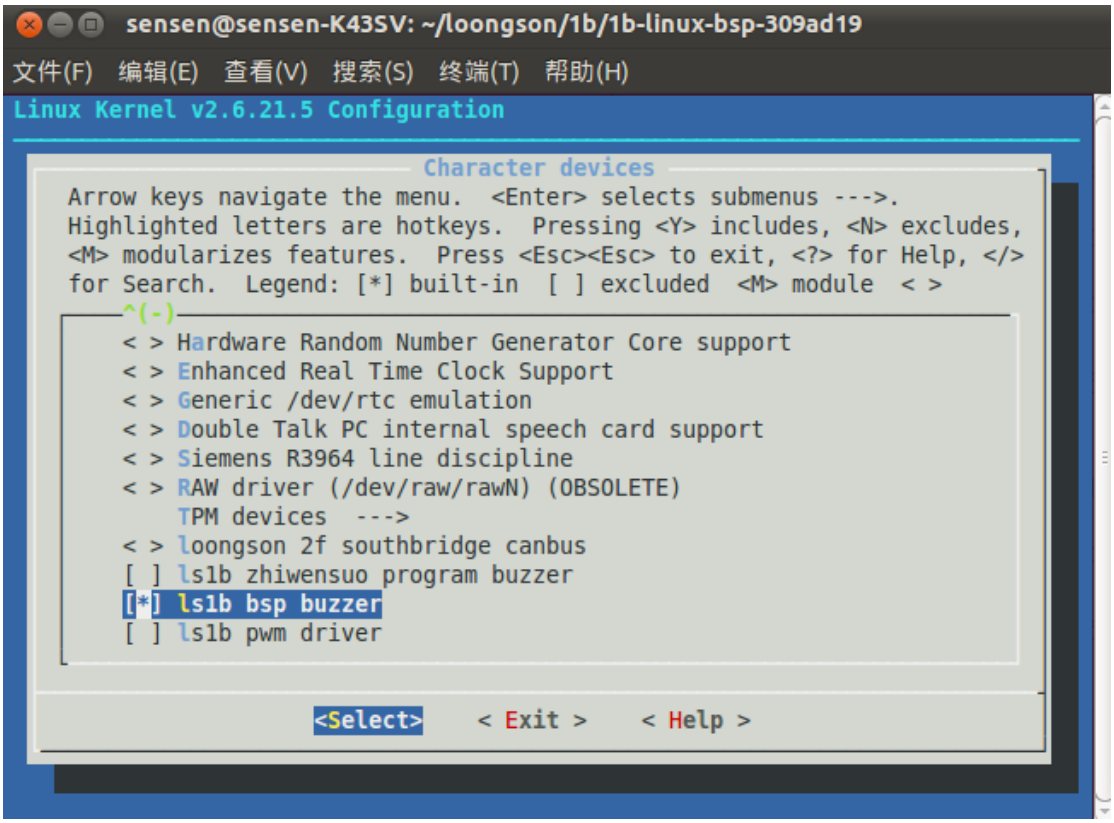

在打开的选项中选中 ls1b bsp of 74LV165 选项。




按<Exit>返回到 Device Drivers 菜单。

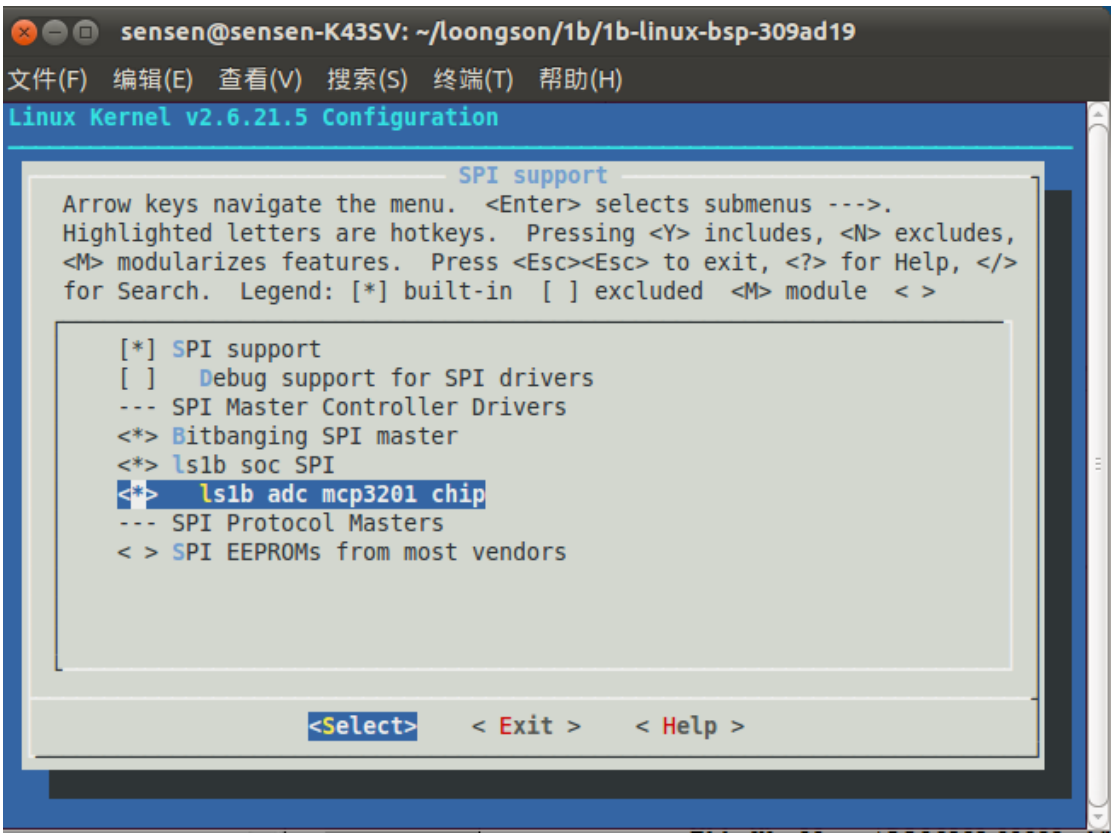
### 4.2.11 配置蜂鸣器驱动

依然在 Character devices 菜单中，找到并选中 ls1b bsp buzzer 选项，如图：



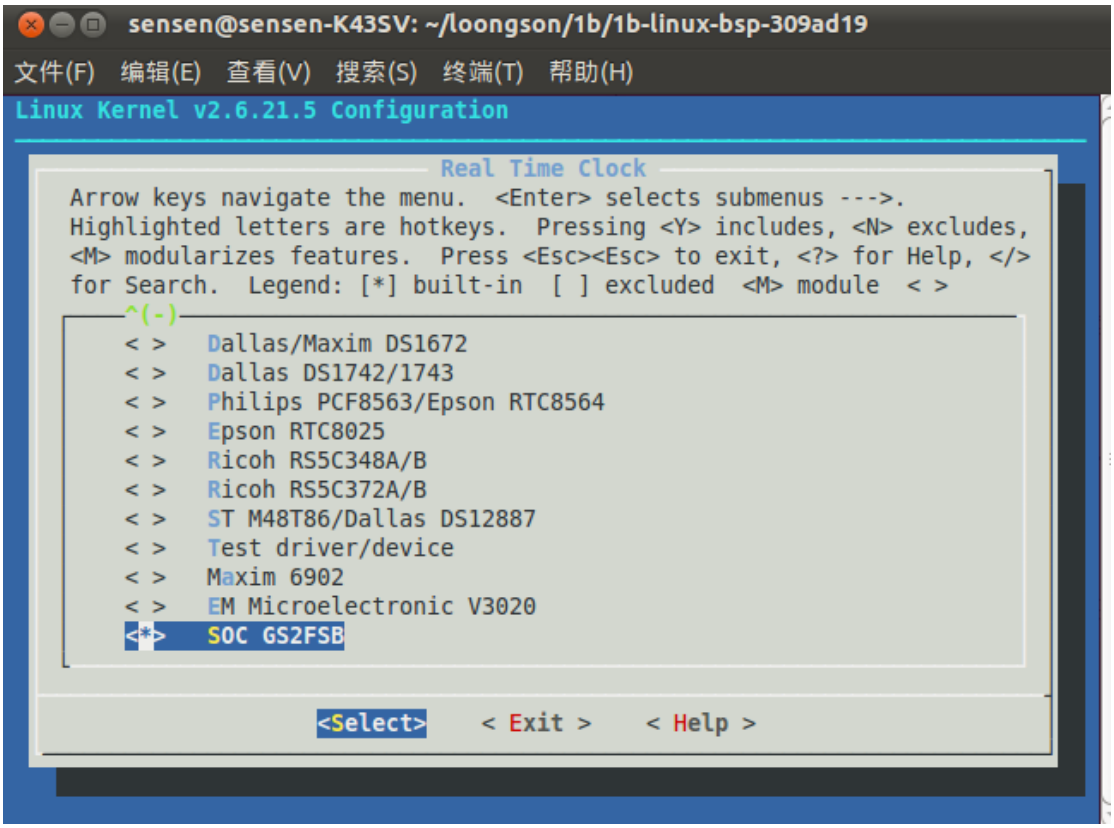

### 4.2.12 配置 AD 转换驱动

在 Device Drivers 菜单中,选择进入 SPI support --->，并按下图所示选中各个选项：



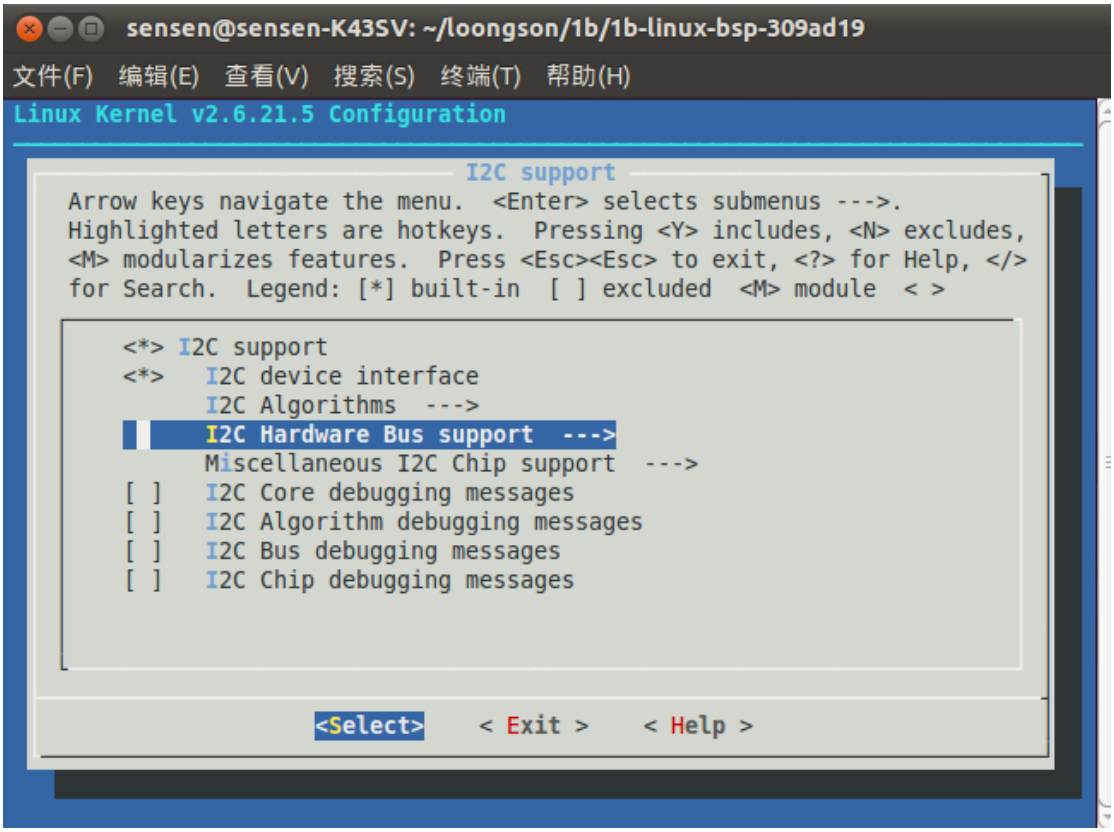

### 4.2.13 配置 RTC 驱动

依然在 Device Drivers 菜单中,选择 Real Time Clock 选项并进入，选中 RTC class 后在其下拉列表中，选中 SOC GS2FSB 选项。

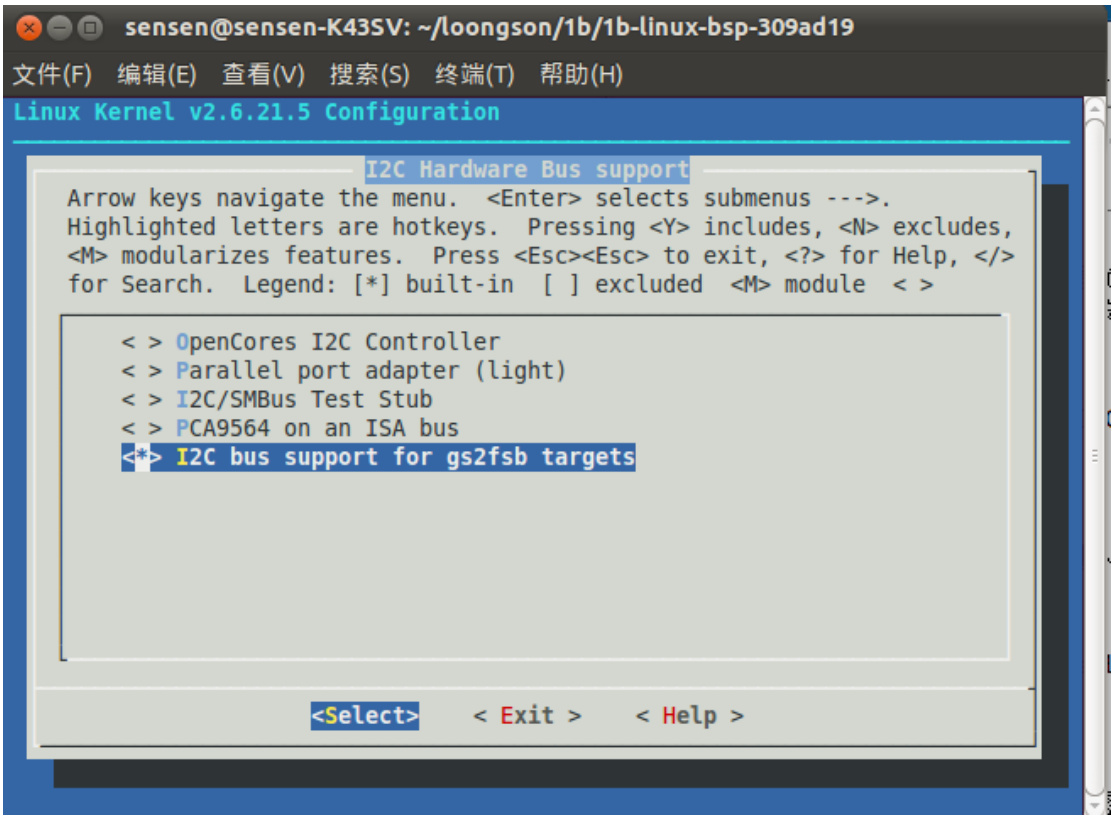



### 4.2.14 配置 I2C-EEPROM 驱动

在 Device Drivers 菜单里面,选择 I2C support --->,按回车进入，按照下图进行选择后：



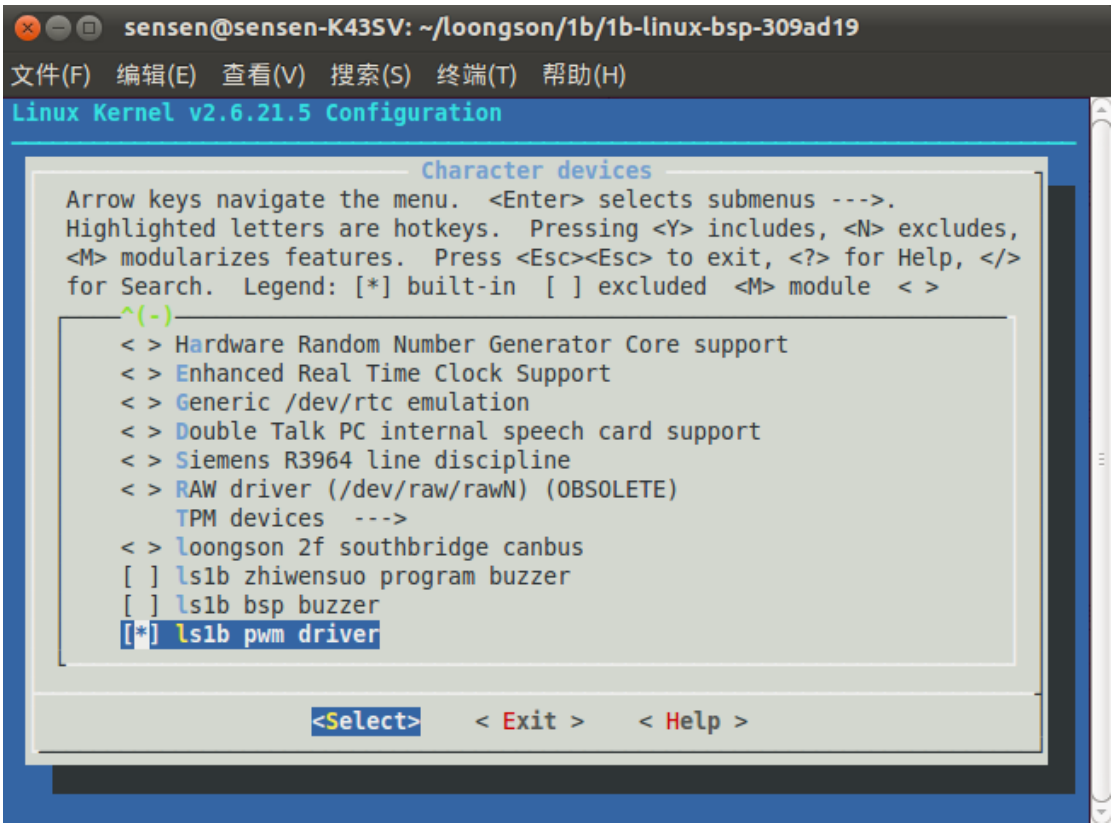

再选择 I2C Hardware Bus support，按回车进入，按照下图选择：




然后选择<Exit>返回 Device Drivers 菜单。

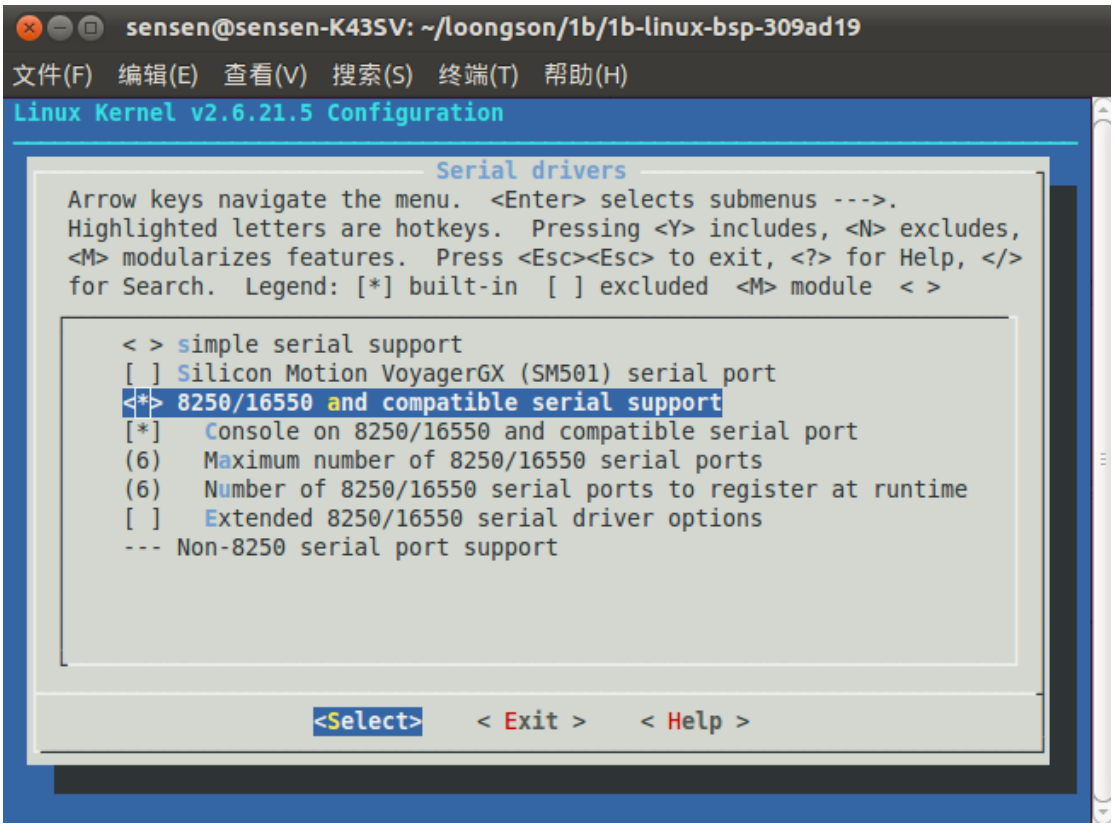
### 4.2.15 配置 PWM 驱动

在 Device Drivers 菜单里面，选择 Character devices --->，按回车进入，按照下图进行选择后。




### 4.2.16 配置串口驱动

依然在 Character devices 菜单中，选择进入 Serial drivers - - ->，并按如图所示进行选择：




# 附录 5　制作根文件系统

源码包位置：**Loongson_1B/BSP/Filesys/busybox-1.19.2-default.tar.gz**

## 5.1 配置、编译 Busybox

Busybox 是一个集成了一百多个最常用 linux 命令和工具的软件，它甚至还集成了一个 http 服务器和一个 telnet 服务器，而所有这一切功能却只有区区 1M 左右的大小。Busybox 的完全可定制性，提供了非常灵活，宜于扩展的结构。

Busybox 的配置方法类似于 linux 内核的配置。下载解压 Busybox-1.19.2.tar.gz 工具包后，进入 busybox-1.19.2 目录，运行“make menuconfig”，根据需要选择需使用的模块，保存退出后会在本地生成一个.config 文件，它指定 busybox 在编译的过程中需要包含哪些功能。

### 5.1.1 建立交叉编译环境

请参考“第四章 4-3 建立交叉编译环境”。

另外需要建立一个链接：

```
#mkdir -p /home/cpu
#cd /home/cpu
#ln -s /opt/gcc-3.4.6-2f    gcc-3.4.6-2f
```

### 5.1.2 安装图形化配置的依赖工具 Ncurses

请参考“第五章 5-2-1 配置内核 第 3 步骤”。

### 5.1.3 图形化配置

提示：调整终端窗口到合适大小或者最大化。

```
#tar zxf busybox-1.19.2-default.tar.gz
#cd busybox-1.19.2-default
#make menuconfig
```

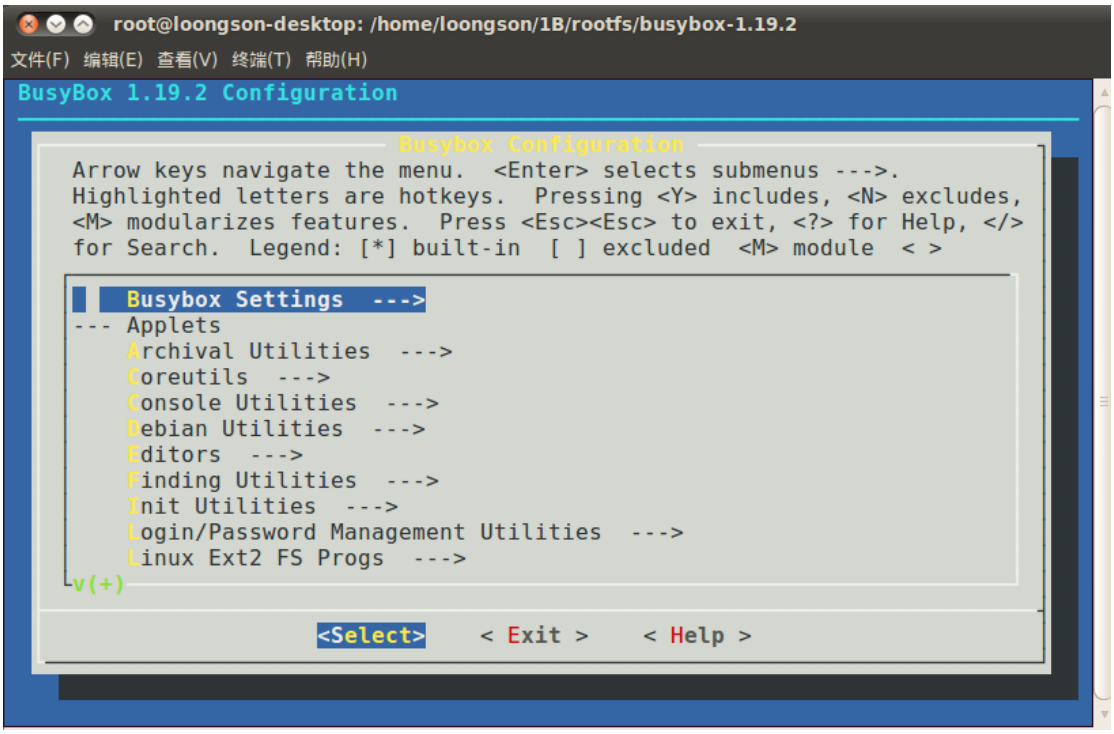
配置选择为：

```
Busybox Setings--->

    General Configuration --->  （默认）

    Build Options--->

        [ ] Build Busybox as a static binary (no shared libs)
        [*] Build shared libbusybox
        (/opt/gcc-3.4.6-2f/bin/mipsel-linux-)Cross Compiler prefix  （需要编辑）

    Installation Options("make install"behavior)--->    （默认）

    Busybox Library Turning--->

        [*] vi-style line editing commands(NEW)
        [*] Fancy shell prompts(NEW)

    Init Utilities--->  （默认）
        [*]init
        [*]Suport reading an inittab file
        [*]Run commands with leading dash with controlling tty
        [*]Support running init from within an initrd (not initramfs)


  Miscellaneous Utilities-->

    [ ] inoice
```

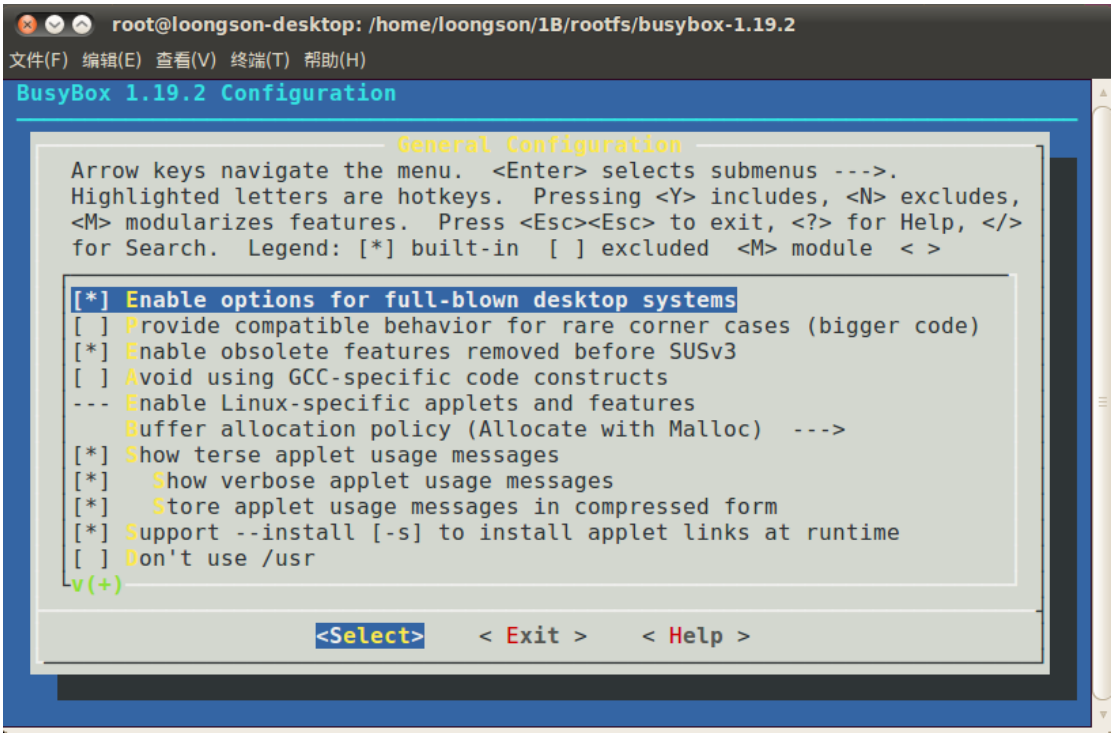
[ ] ubiattach
[ ] ubidetach
[ ] ubimkvol
[ ] ubirmvol
[ ] ubirsvol
[ ] ubiupdatevol

图例说明：

（1）进入主界面 Busybox Configuration，如下图：




（2）在主界面，选择并进入 Busybox Setings--->选项，可以看到 General Configuration ---> 选项，按默认配置即可，如下图：



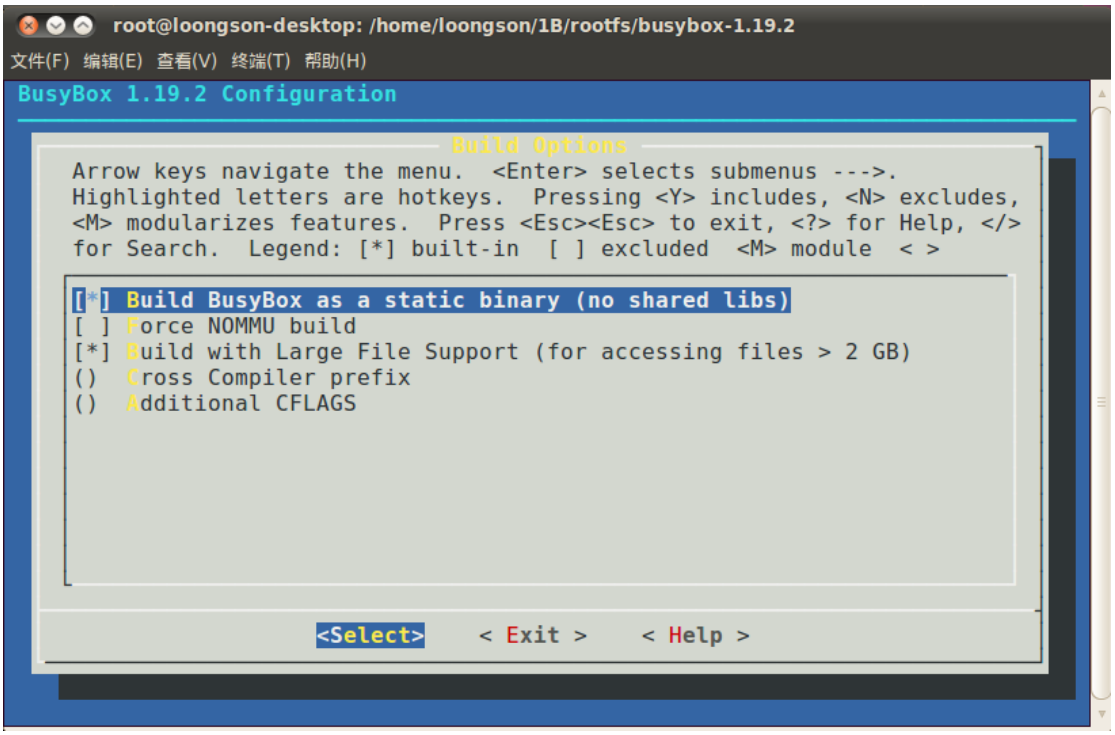

按<Exit>，返回 Busybox Settings 菜单。

（3）在 Busybox Settings 菜单，选择并进入 Build Options--->选项：

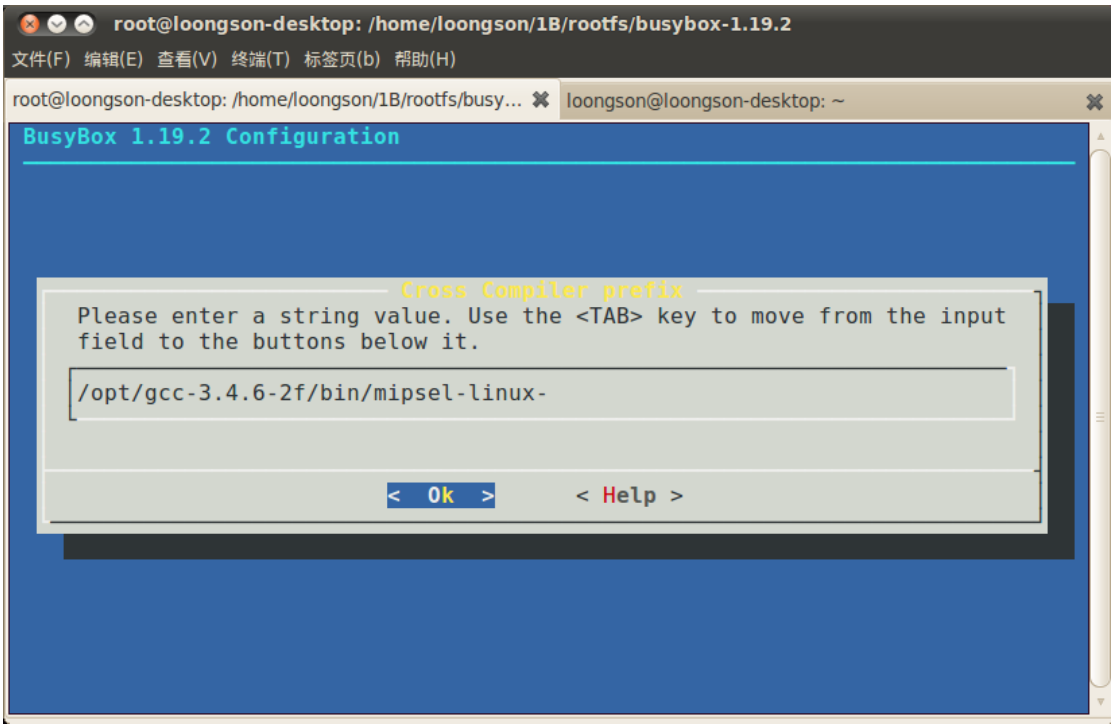
这里可选择静态编译或动态编译 busybox，可根据需要任选一种编译方式进行编译。（说明：动态编译的 busybox 在构建文件系统时需要加入一些必须的动态库，而静态编译的 busybox 不依赖动态库便可执行，构建文件系统时可省略拷贝动态库的步骤。）

a）静态编译：

选择 Build Busybox as a static binary (no shared libs)选项，如下图：

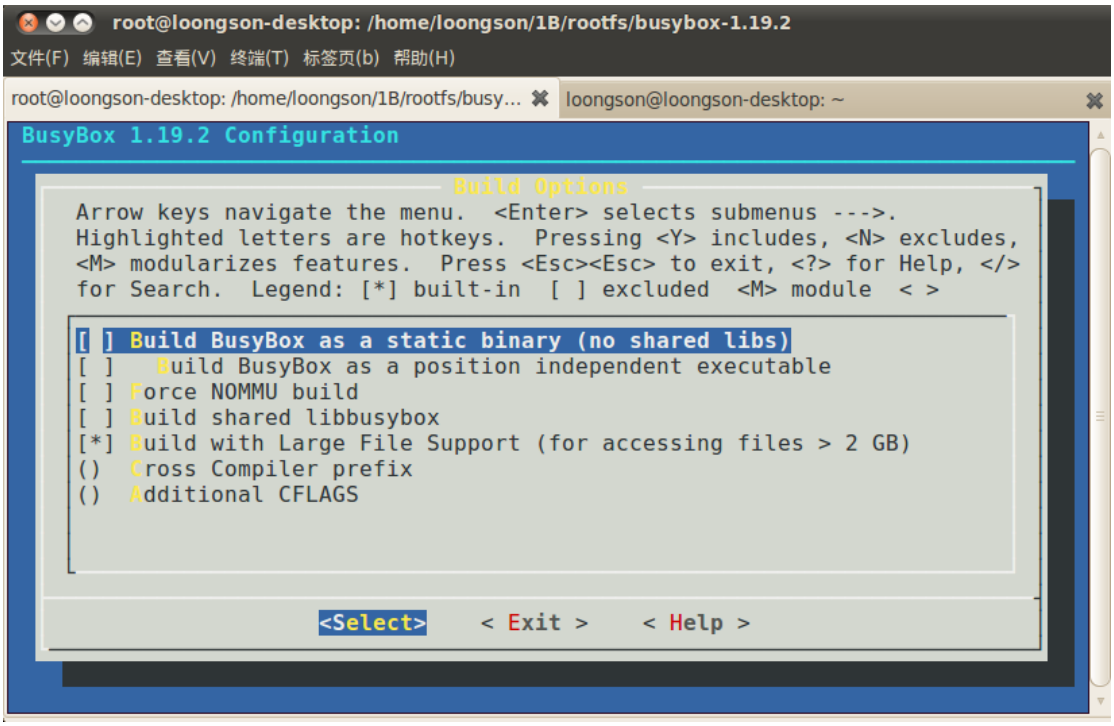



进入 Cross Compiler prefix 选项，修改本地交叉工具链的路径，如“/opt/gcc-3.4.6-2f/bin/mipsel-linux-”，如下图：

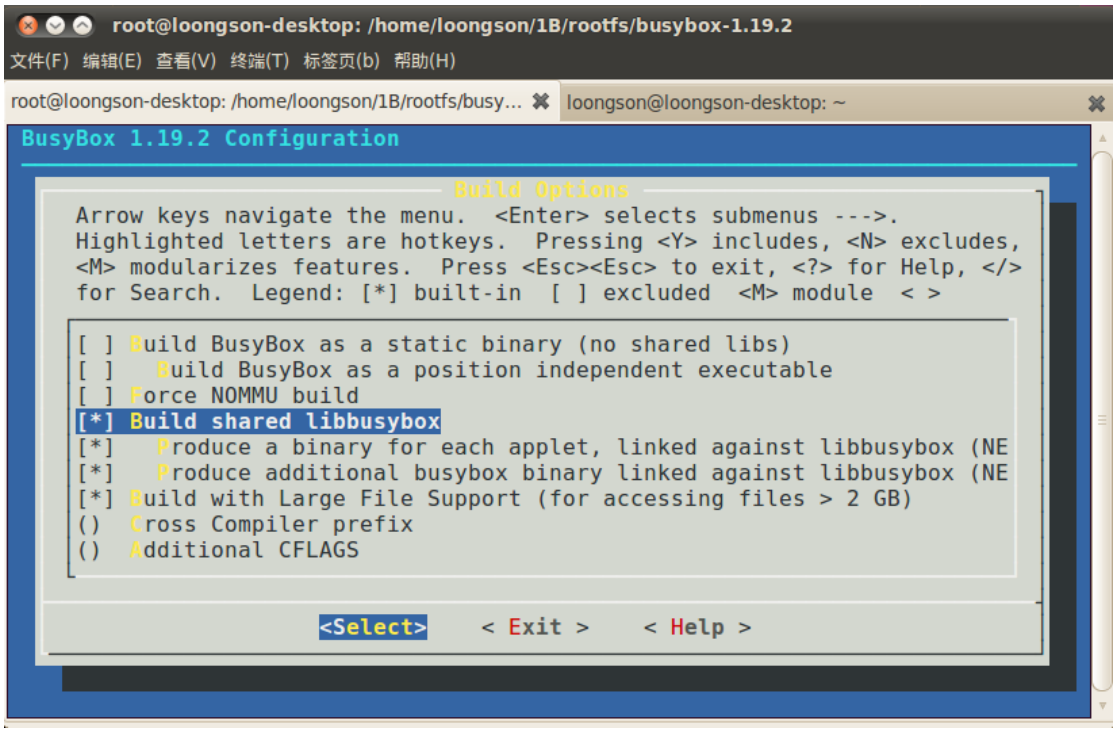



b）动态编译：

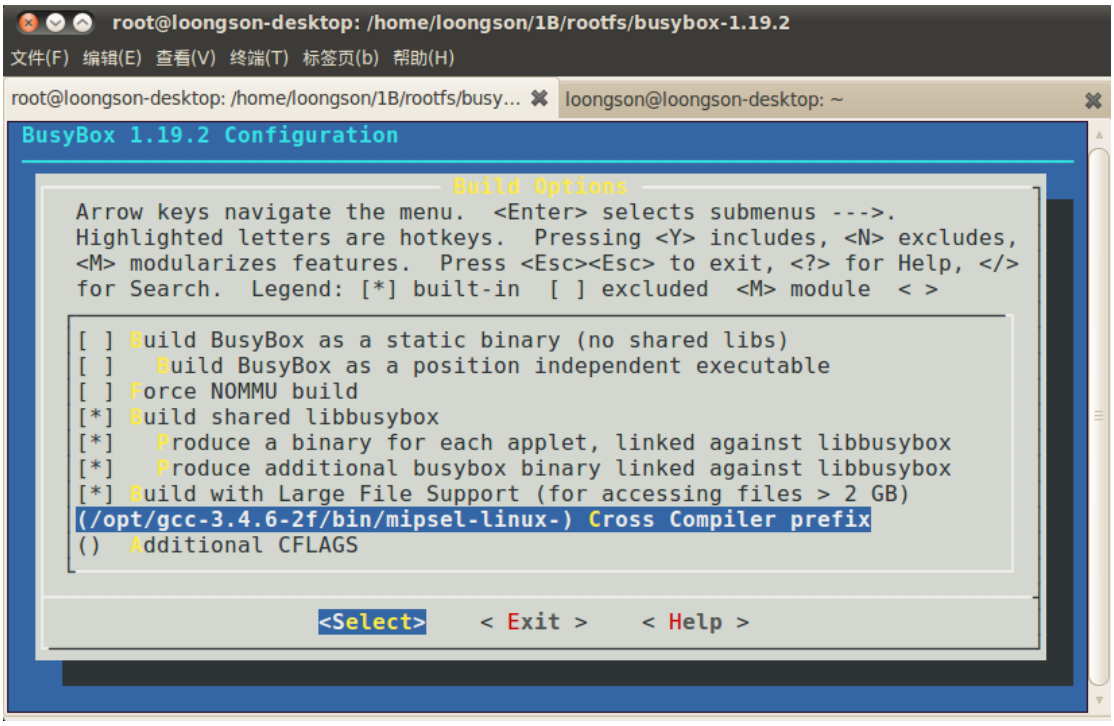
选空 Build Busybox as a static binary (no shared libs)选项，如下图：



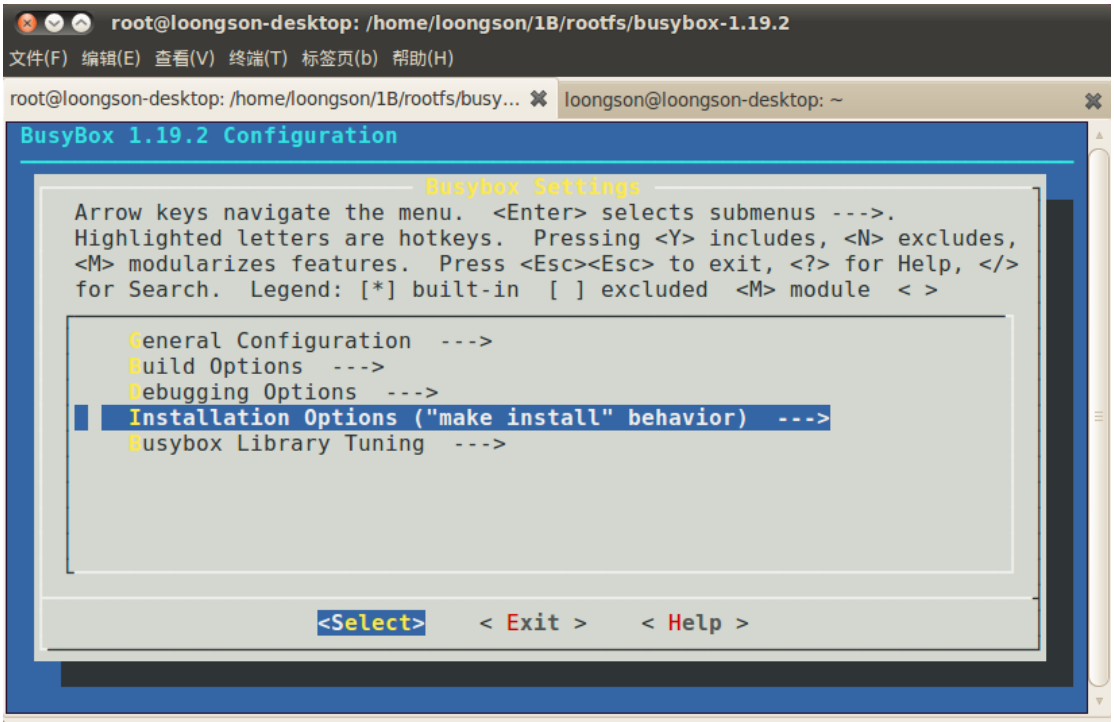

选择 Build shared libbusybox 选项，如下图：



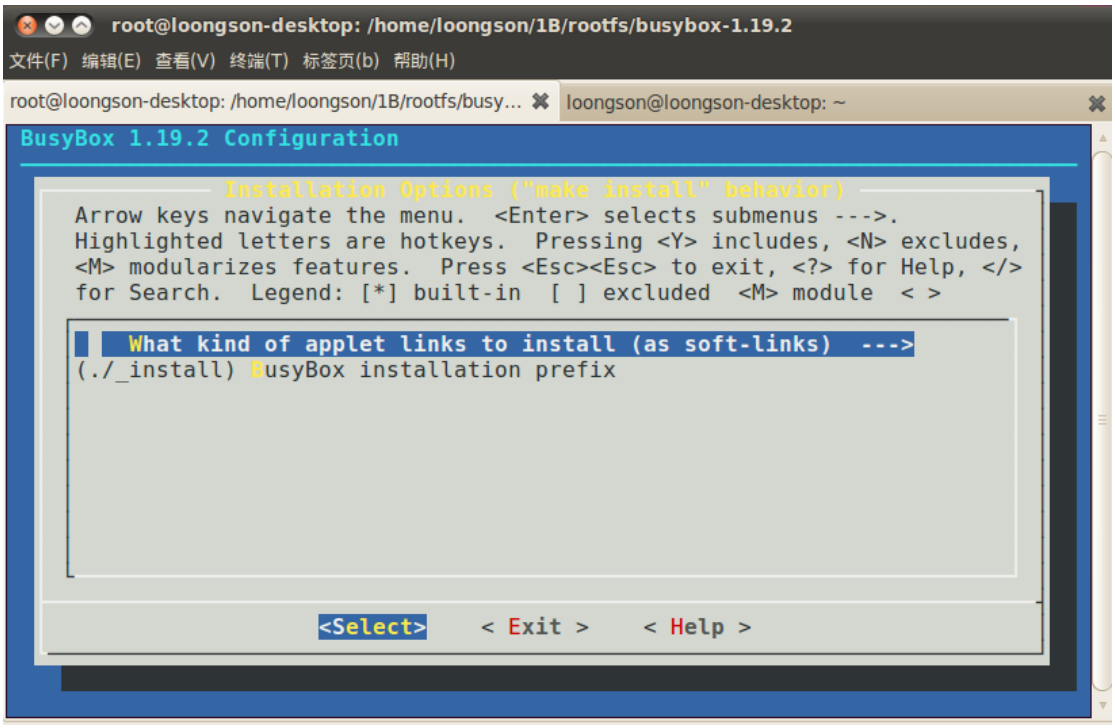

进入 Cross Compiler prefix 选项，修改本地交叉工具链的路径，如下图：



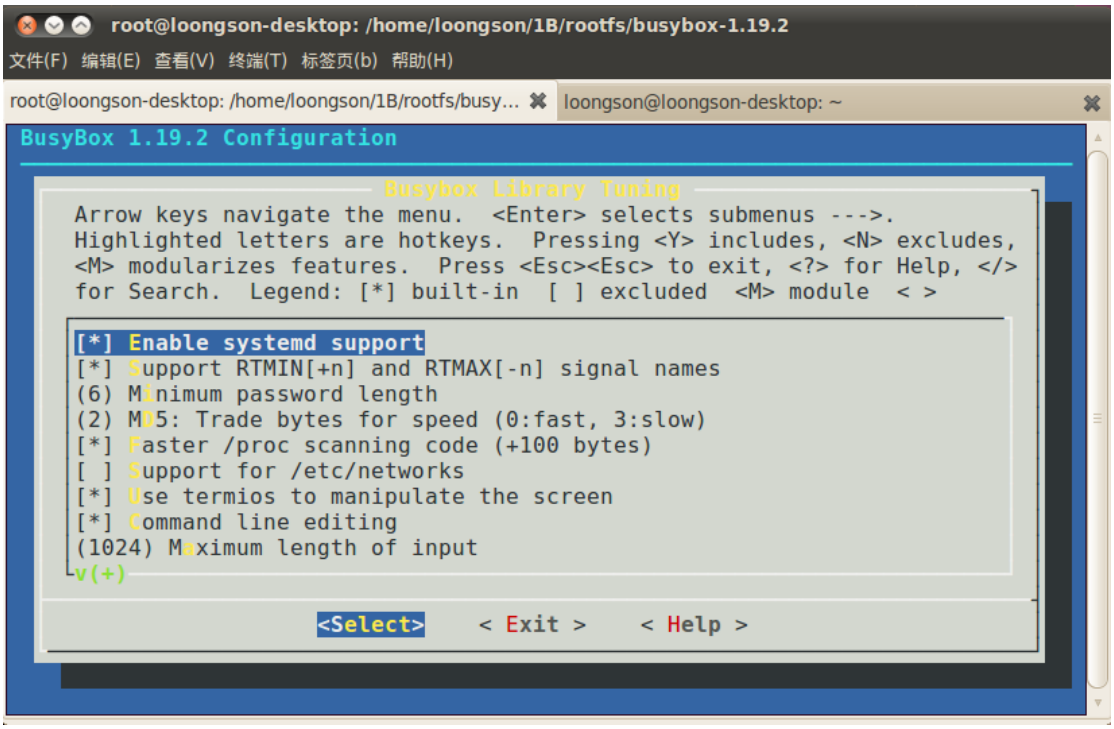

（4）按<Exit>，返回 Busybox Settings 菜单，选择 Installation Options("make install"behavior)--->选项，如下图：



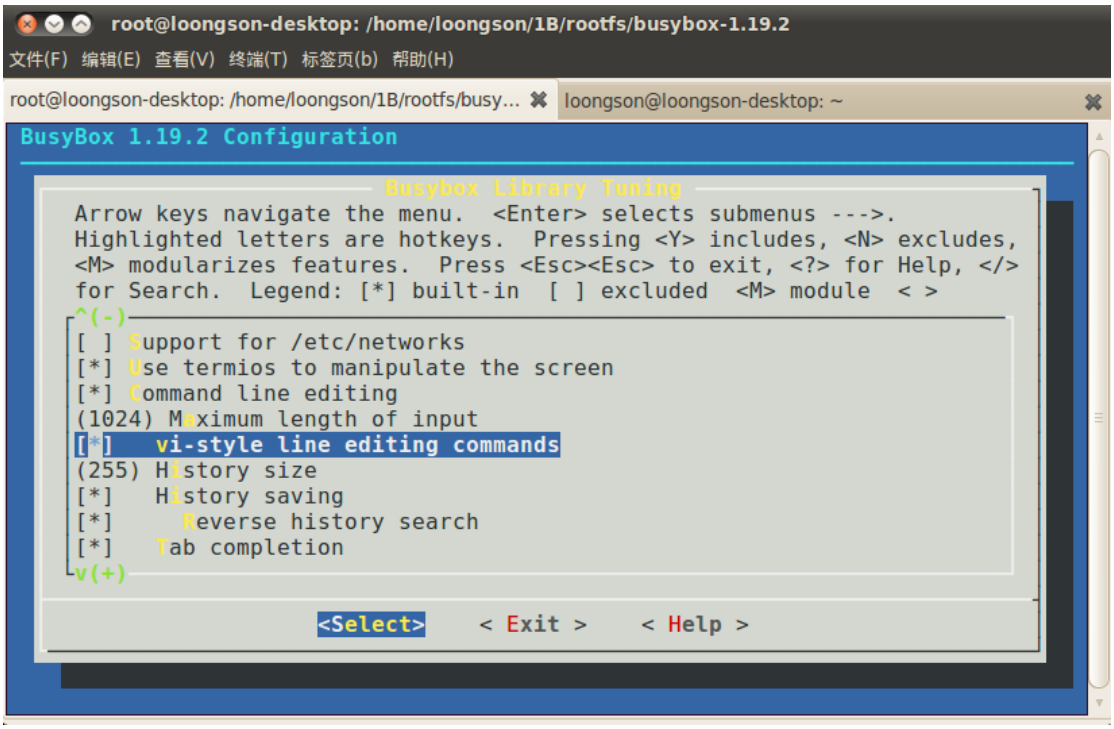

（5）进入 Installation Options("make install"behavior)--->选项，配置链接类型与安装路径，这里使用默认配置，如下图：



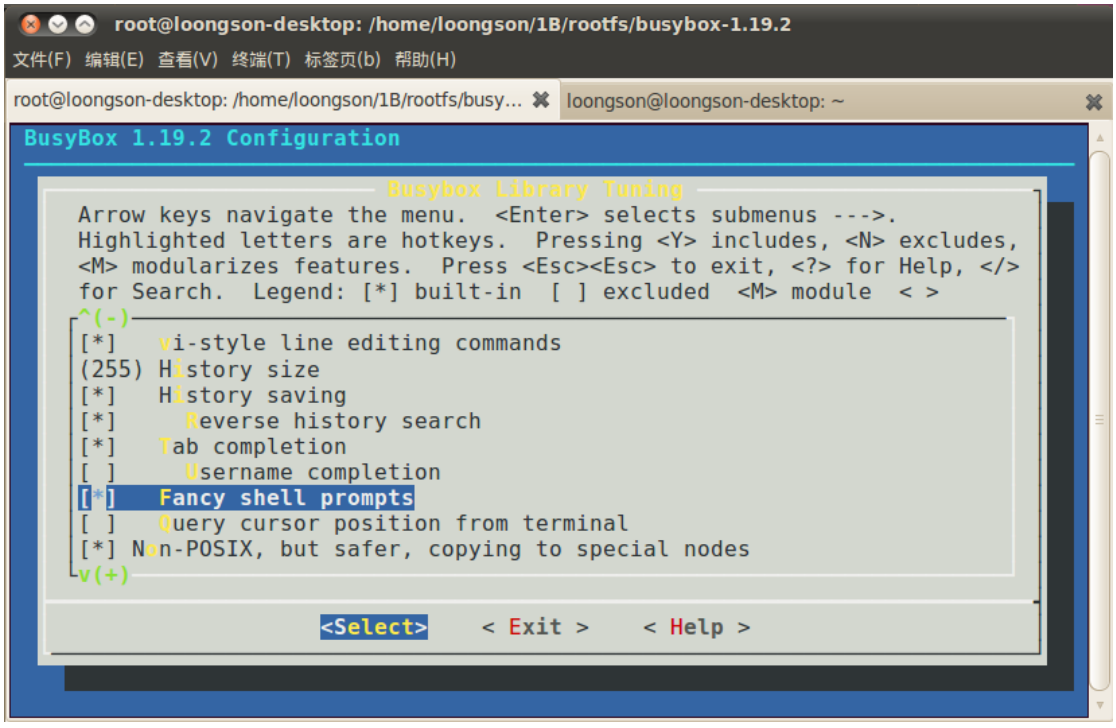

（6）按<Exit>，返回 Busybox Settings 菜单，选择并进入 Busybox Library Turing--->选项，如下图：



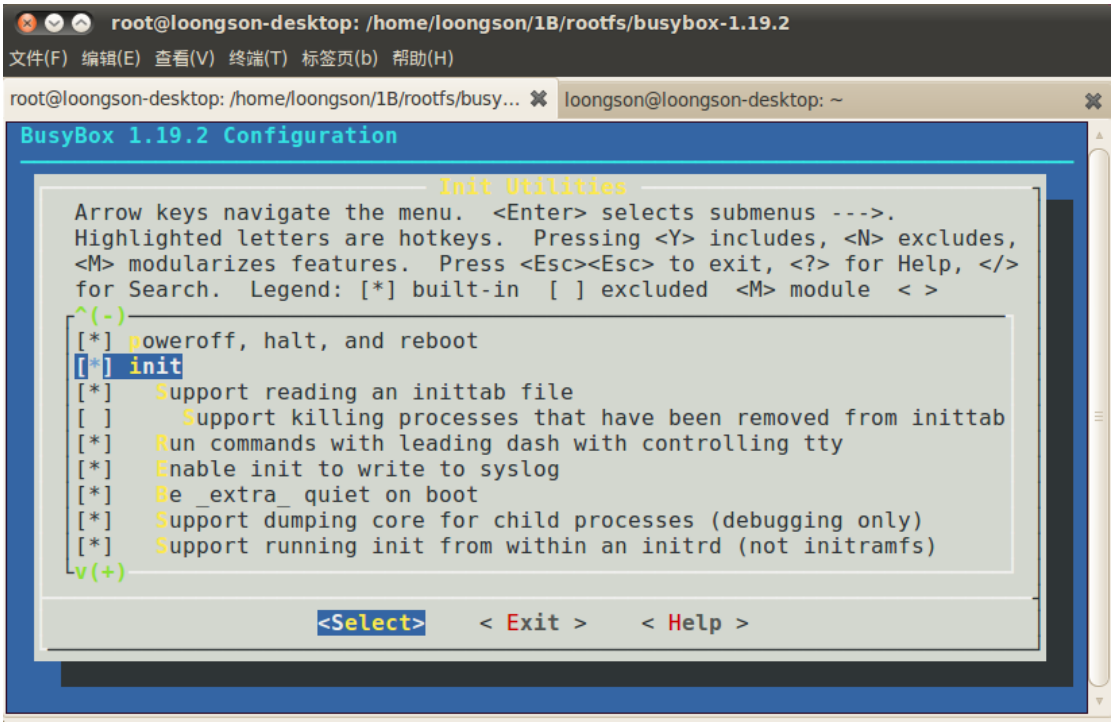

（7）选择 vi-style line editing commands(NEW)选项，如下图：




（8）选上 Fancy shell prompts(NEW)选项，如下图：




（9）按<Exit>，一直返回主界面选择并进入 Init Utilities--->选项，如下图（这里使用默认配置）：




选择如下：

[*]init

[*]Suport reading an inittab file

[*]Run commands with leading dash with controlling tty

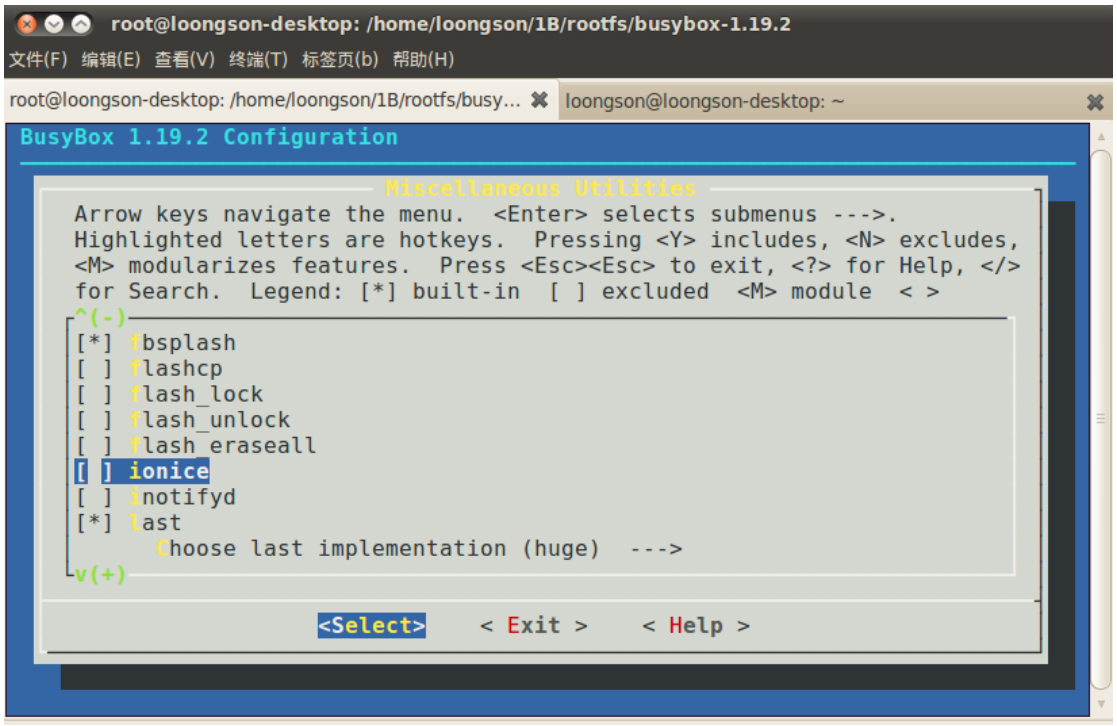
[*]Support running init from within an initrd (not initramfs)

（10）按<Exit>，返回主界面，下面是需要编译进 busybox 的功能选项，它们都是一些 linux 基本命令选项,需要哪些命令就编译进去，一般用默认的就可以了，如下图：

```
--- Applets
    Archival Utilities  --->
    Coreutils  --->
    Console Utilities  --->
    Debian Utilities  --->
    Editors  --->
    Finding Utilities  --->
    Init Utilities  --->
    Login/Password Management Utilities  --->
    Linux Ext2 FS Progs  --->
    Linux Module Utilities  --->
    Linux System Utilities  --->
    Miscellaneous Utilities  --->
    Networking Utilities  --->
    Print Utilities  --->
    Mail Utilities  --->
    Process Utilities  --->
    Runit Utilities  --->
    Shells  --->
    System Logging Utilities  --->
```

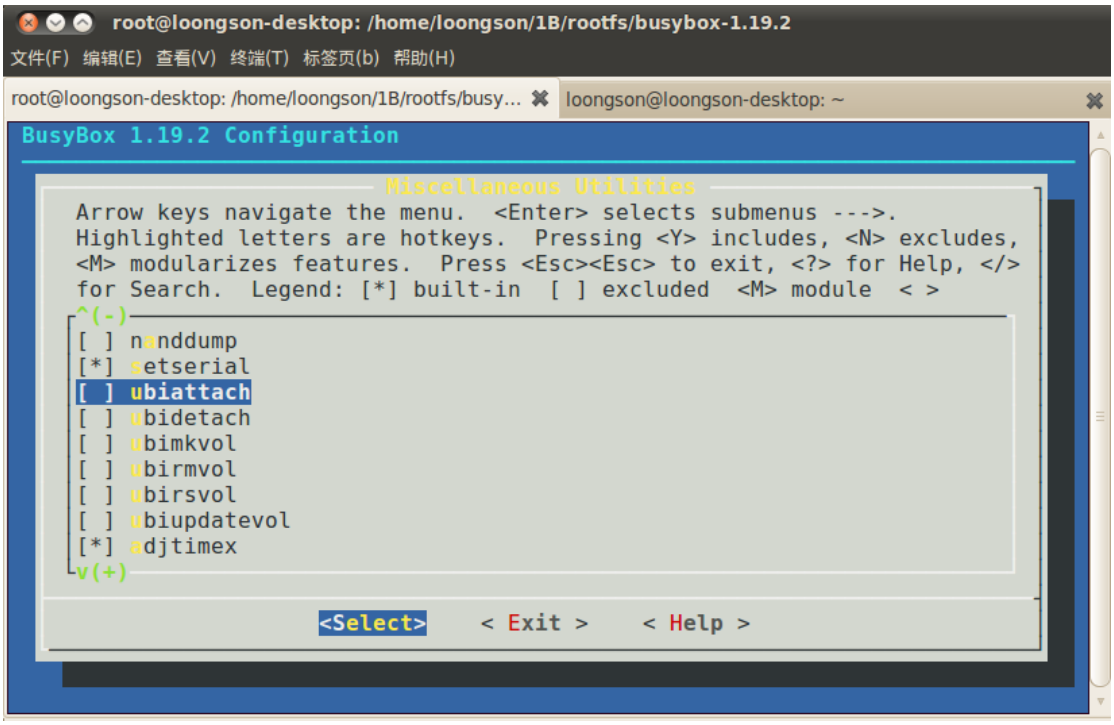
（11）以上几个步骤基本上是配置完成了，但为了解决某些系统的 inoice 出错，需要进行如下配置：

选空 Miscellaneous Utilities-->中的[]inoice 选项，如下图：

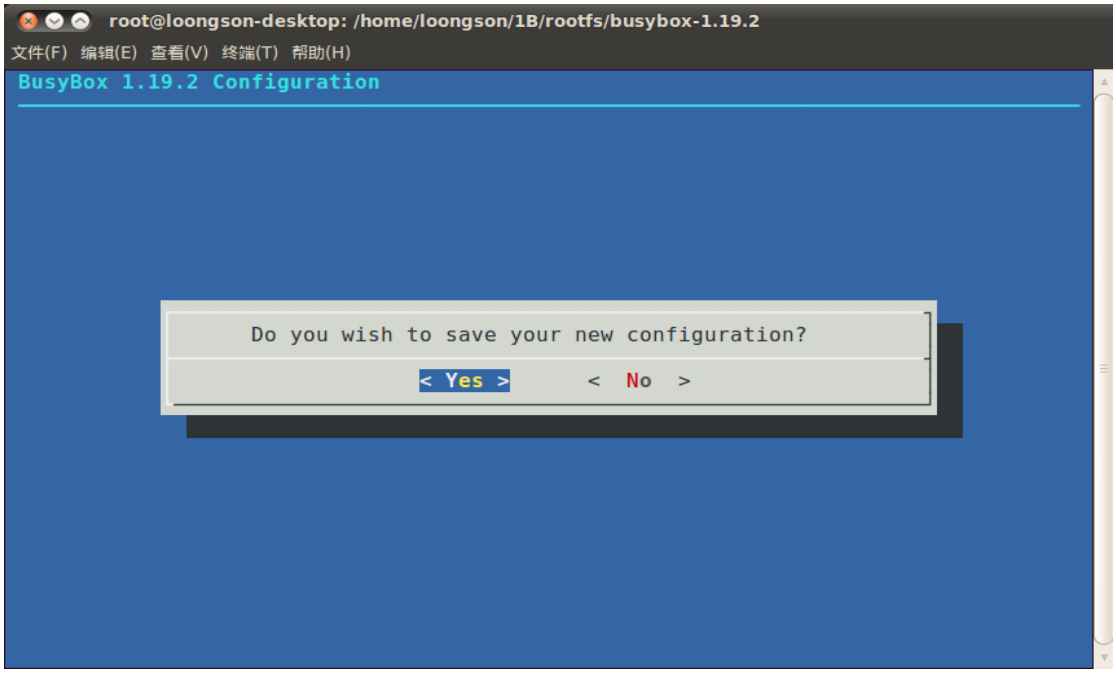



（12）但为了解决某些系统的 ubi 出错，需要进行如下配置：

选空 Miscellaneous Utilities-->中的[]ubi*选项，如下图：




（13）按<Exit>，一直返回到保存配置，按< Yes >保存，如下图：




图形化配置完成。

### 5.1.4 编译 Busybox

配置完成后，编译 Busybox：

```
#make clean all
```

安装到目录：

```
#make install
```

在 Busybox1.19.2 目录下生成子目录_inistall，里面的内容：

```
drwxr-xr-x  2 root root 4096 2011-09-21 14:48 bin/
lrwxrwxrwx  1 root root   11 2011-09-21 14:48 linuxrc -> bin/busybox*
drwxr-xr-x  2 root root 4096 2011-09-21 14:49 sbin/
drwxr-xr-x  4 root root 4096 2011-09-21 14:49 usr/
```

其中可执行文件 busybox 在 bin 目录下,其他的都是指向它的符号链接。

## 5.2 构建根文件系统

### 5.2.1 建立系统根目录

```
#mkdir /root/rootfs
#cd /root/rootfs
#mkdir dev home proc tmp var etc lib mnt sys usr etc/rc.d
```

### 5.2.2 建立系统配置文件

注意：以下的配置文件不是在 Ubuntu 操作系统的根目录下创建的。

（1）etc/inittab 文件

说明：inittab 文件是 init 进程的配置文件，系统启动后所访问的第一个脚本文件，后续启动的文件都由它指定。

```
#cd /root/ rootfs
#vi etc/inittab
```

添加如下内容：

```
::sysinit:/etc/rc.d/rc.sysinit      //指定系统启动后首先执行的文件

#Example of how to put a getty on a serial line (for a terminal)
ttyS2::respawn:-/bin/sh      //启动后进入 shell 环境
```

```
tty1::respawn:-/bin/sh

#Stuff to do when restarting the init process
::restart:/sbin/init

#Stuff to do before rebooting
::ctrlaltdel:/sbin/reboot       //捕捉 ctrl+alt+del 键，重启文件系统
::shutdown:/bin/umount -a –r       //当关机时卸载所有文件系统
::shutdown:/sbin/swapoff -a
```

（2）etc/rc.d/rc.sysinit 文件

说明：这是一个脚本文件，可以在里面添加想自动执行的命令。以下命令配置环境变量、主机名、dev 目录环境、挂接/etc/fstab 指定的文件系统、建立设备节点与设置 IP。

**#vi etc/rc.d/rc.sysinit**

添加如下内容

```
#!/bin/sh
#Set binary path
export PATH=/bin:/sbin:/usr/bin:/usr/sbin

#Set hostname
/bin/hostname    "Loongson-gz"

#Config dev environment
mount –t tmpfs –o size=64k,mode=0755 tmpfs /dev
mkdir –p /dev/pts
mount –t devpts devpts /dev/pts

# mount all filesystem defined in "/etc/fstab"
echo "#mount all....."
/bin/mount –a

echo "# starting mdev...."
echo /sbin/mdev > /proc/sys/kernel/hotplug
/sbin/mdev –s

#Set ip
ifconfig eth0 192.168.3.110 up
ifconfig lo 127.0.0.1
```

(说明：eth0 的 ip 地址可根据需要自行配置)

（3）etc/fstab 文件

说明：执行 mount –a 时挂接/etc/fstab 指定的文件系统。

**#vi etc/fstab**

添加如下内容

```
none       /tmp       ramfs      defaults 0 0
none       /var       ramfs      defaults 0 0
sysfs      /sys       sysfs      defaults 0 0
proc       /proc      proc       defaults 0 0
```

（4）etc/profile 文件

说明：inittab 中执行了这样一个语句“::respawn:-/bin/sh”。启动/bin/sh 程序时会启动 ash 的配置信息，而它就是/etc/profile，sh 会把 profile 的所有配置全部都运行一遍，因此用户可以把自己的启动程序放在这里。

**#vi etc/profile**

添加如下内容

```
#!/bin/sh
#/etc/profile:system-wide .profile file for the Bourne shells
echo "Processing /etc/profile....."

# Set search library path
export LD_LIBRARY_PATH=/lib:/usr/lib

# set user path
export PATH=/bin:/sbin:/usr/bin:/usr/sbin

#Set PS1
USER = "LOONGSON"
#LOGNAME=$USER
HOSTNAME=’/bin/hostname’
PS1='[$USER@\h:\w]\$'
echo "Done!"
```

（5）修改系统配置文件权限

```
#chmod 755 etc/*
#chmod 755 etc/rc.d/rc.sysinit
```

（6）拷贝 Busybox 文件

将安装好的 Busybox 文件拷贝到/root/rootfs/目录：

```
#cp ./Busybox1.19.2/_install/* /root/rootfs -rf
```

### 5.2.3 拷贝库文件

提示：配置 Busybox 若选择静态编译则省略此步骤。

在 x86 系统中，动态编译 Busybox 制作文件系统，需将下列几个必须库从工具链

gcc-3.4.6-2f/mipsel-linux/lib 拷贝到 lib 目录。

ld.so.1, ld-2.3.6.so, libcrypt.so.1, libc.so.6, libdl.so.2, libgcc_s.so.1, libm.so.6, libpthread.so.0, libstdc++.so.6

```
#cp /opt/gcc-3.4.6-2f/mipsel-linux/lib/库文件名  /root/rootfs/lib/
```

可以使用脚本，快速地拷贝所需的基本动态库：

```
#cd /opt/gcc-3.4.6-2f/mipsel-linux/lib
#vi cplib.sh
```

添加内容：

```
for file in libcrypt libc libdl libgcc libgcc_s libm libpthread libstdc++
do
cp -d $file.so.[*0-9] /root/rootfs/lib
done
cp -d ld*.so* /root/rootfs/lib
```

保存退出。

运行 cplib.sh SHELL 脚本：

```
#source cplib.sh
```

为了减少根文件系统的库大小，使用交叉编译工具即 gcc-3.4.6-2f 的 strip 工具来处理库文件，把二进制文件中的包含的符号表和调试信息删除掉,可有效减少库文件大小。

例：

```
#cd /opt//gcc-3.4.6-2f/bin
#./mipsel-linux-strip /root/rootfs/lib/*.so
```

至此根文件系统（目录/root/rootfs）制作完成。

# 附录 6　Minicom 使用指南

minicom 是 Linux 上最常用的终端仿真程序，它类似于 Windows 下的“超级终端”的程序，一般完全安装大部分发行版的 Linux 时都会包含它，下面介绍它的使用方法。

## 6.1 安装 minicom

如果没有安装 minicom，先安装：

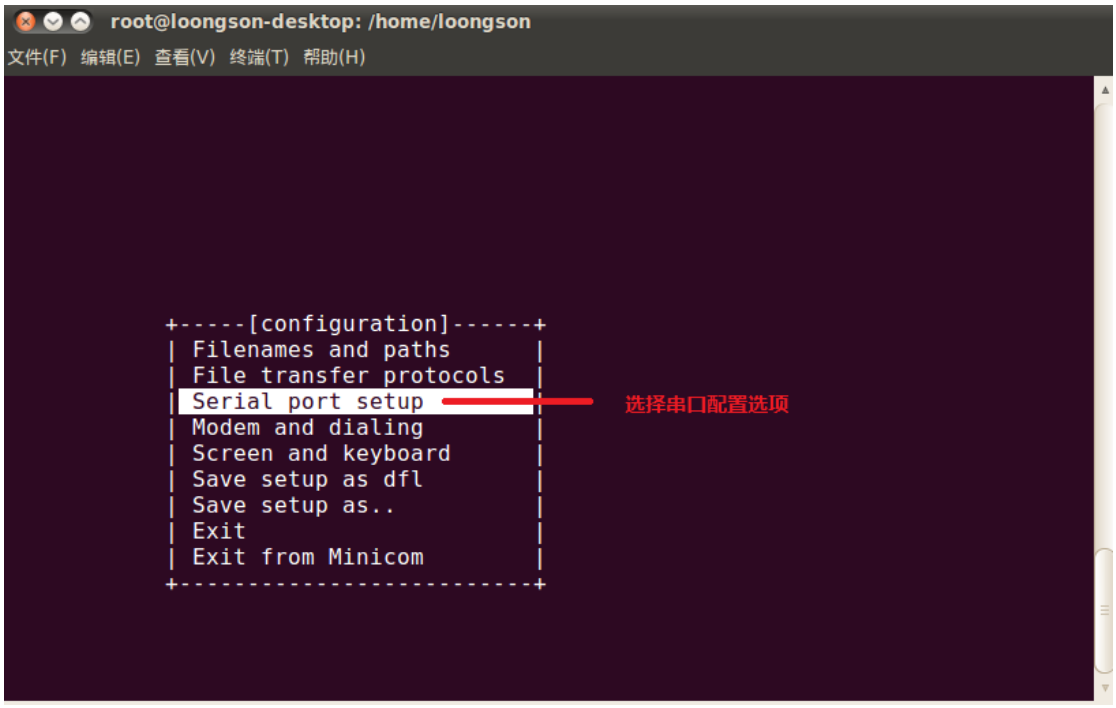
```
apt-get install minicom
```

## 6.2 配置 minicom

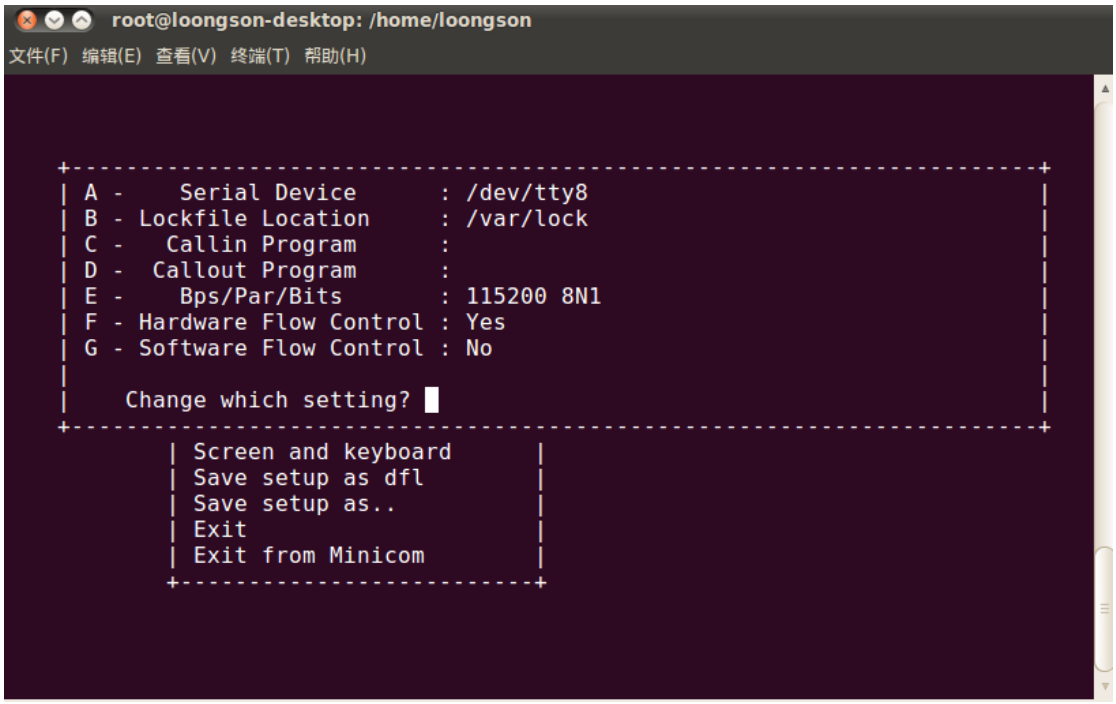
使用 minicom 之前先设置一下：

（1）Linux 终端输入

```
minicom -s
```




（2）进入“Serial port setup”：



提示：

A 对应的是键盘‘A’键，按下‘A’键配置串口设备。通常的配置是/dev/ttyS0 或/dev/tty0 或者/dev/ttyUSB0，对应于 windows 操作系统的 COM1。

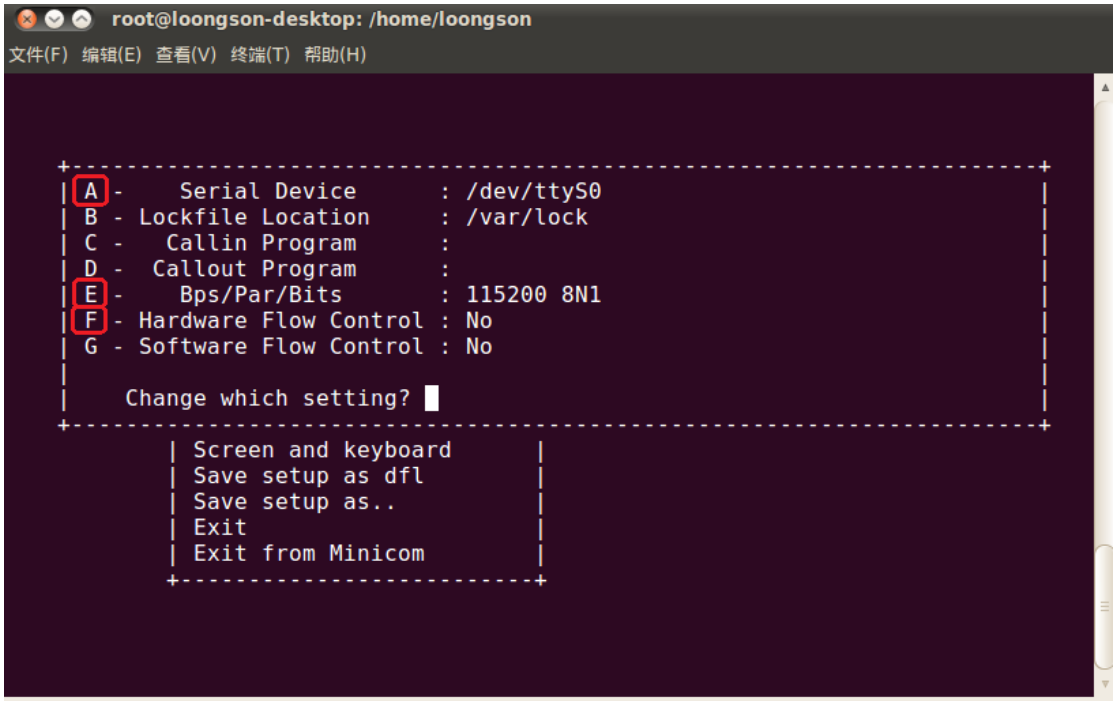
F 对应的是硬件流设置，按‘F’进行配置，通常选择 NO 。

E 对应的是键盘‘E’键，是用来配置串口波特率的，按键‘E’选择波特率 。

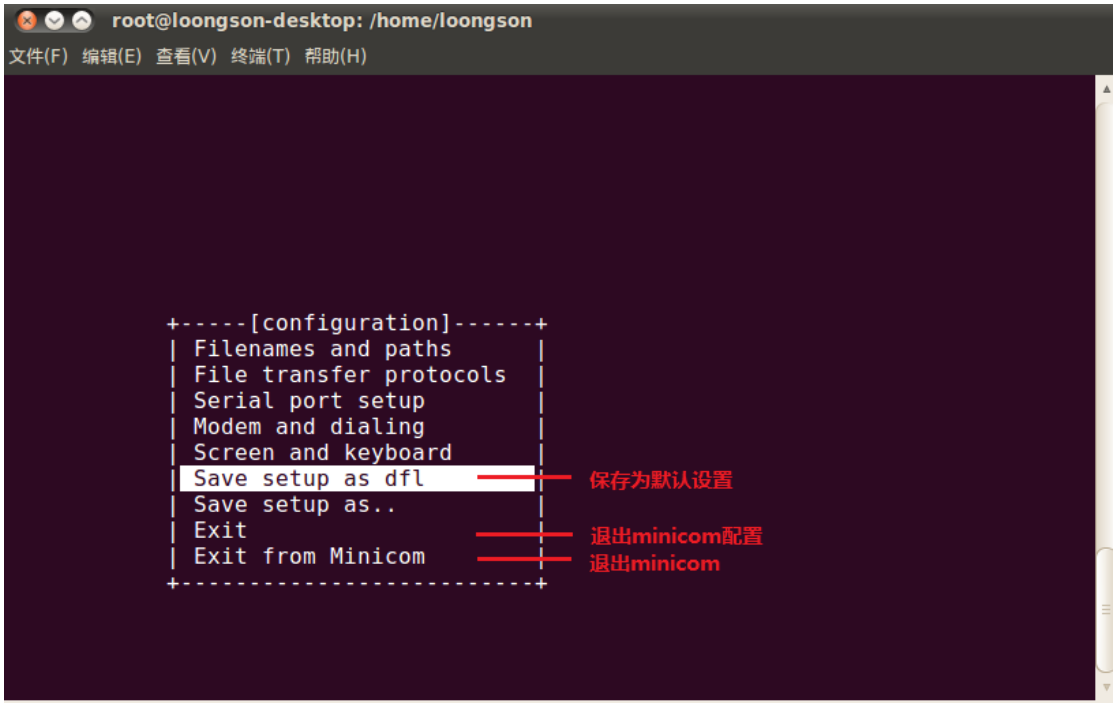
键盘‘ESC’键，退出，返回上一层。

键盘‘Enter’键，确定，保存。

（3）配置串口设备、波特率和硬件流：

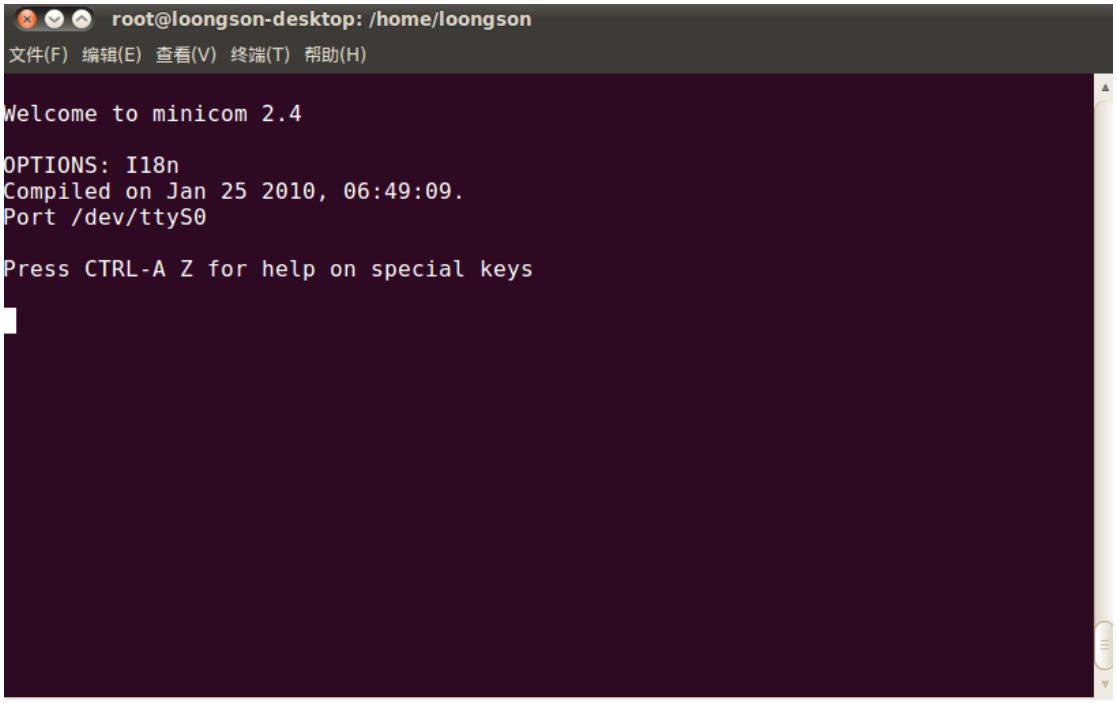


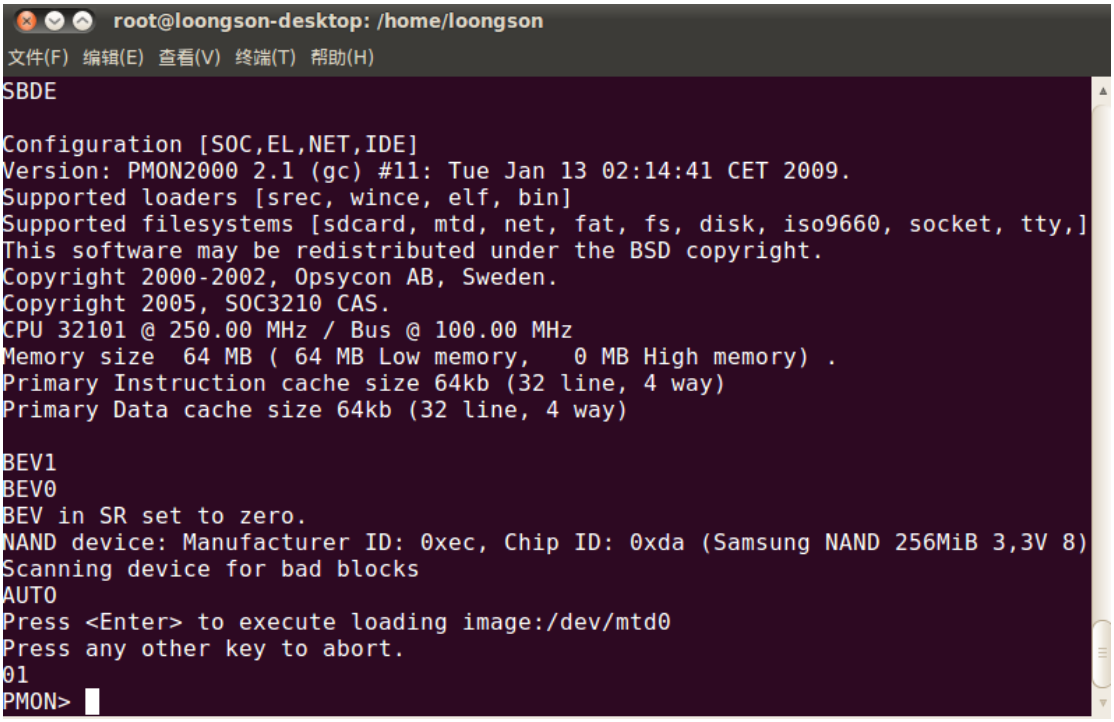

（4）保存退出到主配置界面：



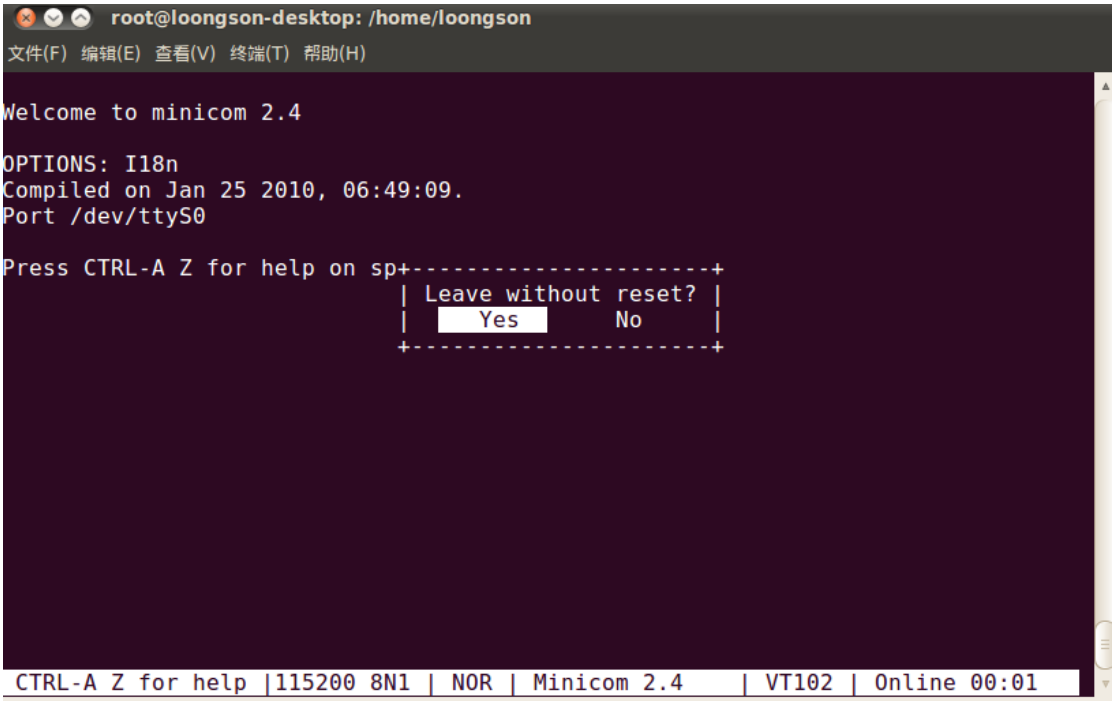

## 6.3 使用 minicom

先连接好串口线或 USB 转串口。
退出 minicom 配置，进入 minicom：






在进入 minicom 后，如果想退出，则可以按下 ctrl+A，松开 ctrl+A 后再紧接着按下 Q 就可以看到退出提示了，选择 Yes 退出。



# 附录 7　使用 EJTAG 烧写 BOOTLOADER（PMON）

源码包位置：

## 7.1 编译安装 ejtag

### 7.1.1 工具与依赖库的安装

工具与依赖库的安装有两种方式：有互联网时使用 apt-get 命令来安装（默认方式）；无互联网时使用源码包来安装。

（1）使用 apt-get 命令安装（有互联网）

安装 readline 与 libusb

```
#apt-get install libreadline5-dev
#apt-get install libusb-dev
```

（2）使用源码包安装（无互联网）

源码工具位置：**Loongson_1B/Tools/ejtag**

```
#tar xf ejtag-dependtools.tar.gz
#cd ejtag-dependtools
#source ejtag-dependtools.sh
```

脚本文件 ejtag-dependtools.sh 的内容如下：

```
#!/bin/bash
#EJTAG tools install

cp ./include/* /usr/include -rf        # 拷贝 readline 和 libusb 工具的头文件

cp ./lib/* /usr/lib -rf                # 拷贝 readline 和 libusb 工具的动态库
```

### 7.1.2 编译 ejtag

（1）解压交叉编译工具链并设置环境变量

请参考“第四章 4-3 建立交叉编译环境”。

（2）编译 EJTAG 工具

```
#tar zxvf ejtag-1b.tar.gz
#cd bioscfg/
```

```
#make clean
#make
```

## 7.2 烧写 pmon

（1）先把开发板相应的连线接好，上电运行。

（2）把从“第五章 5-1-2 配置与编译 pmon ”得到的 gzrom.bin 拷贝到 ejtag 的 bioscfg 目录，再在 bioscfg 目录内执行：

```
./ejtag_debug_usb <config-sst.pmon        （其实一般来说配置文件为 config.pmon）
```

# 附录 8 NFS 网络文件系统搭建

NFS 不是传统意义上的文件系统，而是访问远程文件系统的协议。其最主要的功能就是让网络上的 linux 主机可以共享目录及档案。我们可以将远端所分享出来的档案系统，挂载(mount) 在本地端的系统上，然後就可以很方便的使用远端的档案，而操作起来就像在本地操作一样，不会感到有甚麼不同。而使用 NFS 也有相当多的好处，例如档案可以集中管理等等，特别是对开发中的调试程序有极大的方便。下面则是 nfs 搭建的步骤：

## 8.1 安装 NFS

NFS 的安装有两种方式：有互联网时使用 apt-get 命令来安装（默认方式）；无互联网时使用源码包来安装。

（1）使用 apt-get 命令安装（有互联网）

```
#apt-get install nfs-common
#apt-get install nfs-kernel-server
```

（2）使用源码包安装（无互联网）

源码包位置：**Loongson_1B/Tools/other_tools/nfs**

```
#tar xf nfs-install.tar.gz
#cd nfs-install
#source nfs-install.sh
```

为了较清晰地表现安装流程，脚本文件 nfs-install.sh 的内容为：

```
#!/bin/bash
```

```
# NFS install

############    nfs-utils    ############

tar xf patch_2.6.orig.tar.gz
cd patch-2.6
./configure && make && make install
cd ..

tar xf tcp-wrappers_7.6.q.orig.tar.gz
cd tcp_wrappers_7.6
patch -Np1 -i ../tcp_wrappers-7.6-shared_lib_plus_plus-1.patch && sed -i -e "s,^extern char \*malloc();,/* & */," scaffold.c
#http://www.linuxfromscratch.org/patches/blfs/svn/tcp_wrappers-7.6-shared_lib_plus_plus-1.patch
make REAL_DAEMON_DIR=/usr/sbin STYLE=-DPROCESS_OPTIONS linux
make install
cd ..
tar xf libevent_1.4.13-stable.orig.tar.gz
cd libevent-1.4.13-stable
./configure && make && make install
cd ..

tar xf libnfsidmap_0.23.orig.tar.gz
cd libnfsidmap-0.23
./configure && make && make install
cd ..

tar xf util-linux_2.17.2.orig.tar.gz
cd util-linux-ng-2.17.2
./configure --without-ncurses && make && make install
cd ..

tar xf nfs-utils_1.2.0.orig.tar.gz
cd nfs-utils-1.2.0
./configure --enable-gss=no
make && make install
cd ..

############    portmap    ############

tar xf portmap_6.0.0.orig.tar.gz
cd portmap-6.0.0
```

```
make && make install
cd ..
```

## 8.2 配置 NFS

### 8.2.1 在 linux 工作服务器端配置

（1）修改配置文件

```
#vim    /etc/exports
```

添加内容为：

```
/home/xxx/nfsrootfs 192.168.*.*(rw,no_root_squash,sync)
```

内容含义：

/home/xxx/nfsrootfs：要共享的目录，需要先创建后改变权限
*：网段内所有值
rw：读写权限
sync：资料同步写入内在和硬盘
no_root_squash：nfs 客户端共享目录使用者权限

保存退出 。

（2）启动 NFS 服务

在使用了 apt-get 命令安装的平台：

```
# /etc/init.d/portmap restart
# /etc/init.d/nfs-kernel-server restart
```

在使用了源码包来安装的平台：
运行 nfs-install 目录里的启动脚本

```
#cd nfs-install
#source nfs-server.sh
```

脚本 nfs-server.sh 内容为：

```
#!/bin/bash
# NFS server

#mkdir -p /nfsrootfs
#touch /etc/exports | echo "/nfsrootfs *(sync,rw,no_root_squash)" > /etc/exports

portmap
```

```
mount -t nfsd nfsd /proc/fs/nfsd

mkdir -p /var/lib/nfs
touch /var/lib/nfs/etab

exportfs -av

rpc.mountd
rpc.statd --no-notify
rpc.nfsd
```

（3）显示共享出的目录

```
#showmount    -e
```

（4）创建服务文件目录，更改权限

```
#mkdir /home/xxx/nfsrootfs
#chmod 777 /home/xxx/nfsrootfs
```

配置服务器端完毕。

### 8.2.2 目标机一端配置

目标机一端配置，主要是重新编译 1B 开发板的内核，再把此内核更新到开发板。
（1）编译 1B 板子内核，添加 nfs 功能

```
Networking --->

    [*] Networking support --->

        Networking options --->

            [*] TCP/IP networking
                [*] IP: kernel level autoconfiguration
                    [*] IP: DHCP support
                    [*] IP: BOOTP support
                    [*] IP: RARP support
```

```
File systems --->
```

```
Network File Systems --->
    <*> NFS file system support
        [*] Provide NFSv3 client support
            [*] Provide client support for....
        [*] Provide NFSv4 client support...
        [*] Root file system on NFS
```

（2）编译内核，下载到 1B 开发板上的/dev/mtd0 分区。

## 8.3 本机测试

在 Linux 上，测试挂载 nfs 服务目录：

```
#mount -t nfs -o nolock 192.168.3.122:/home/xxx/nfsrootfs    /tmp
```

提示：

192.168.3.122 为 Linux 的 IP 地址；/home/xxx/nfsrootfs 为 NFS 服务目录。

那么在/home/xxx/nfsrootfs 对文件的操作也相当于在/tmp 的操作。比如：

```
#touch /home/xxx/nfsrootfs/test | echo "test !" > /home/xxx/nfsrootfs/test
#ls /tmp
#cat /tmp/test
test !
```

反之，也一样。说明 NFS 服务已经开启。

当然，退出 NFS 服务，可以取消挂载。如：

```
#umount /tmp
```

## 8.4 使用 NFS

1、在开发板上挂载 nfs 服务目录

（1）在宿主机，把交叉编译后的程序置于共享目录：

```
#cd /home/xxx/nfsrootfs
```

（2）在目标板中，挂载宿主机的 nfs 共享目录：

```
#mount -t nfs -o nolock 192.168.3.x:/home/xxx/nfsrootfs    /mnt
```

其中 192.168.3.x 为宿主机的 IP 地址。

这样就把共享目录挂到了/mnt 目录。

（3）使用 NFS 运　行程序：

```
#cd /mnt
```

可以直接运行当前目录已经交叉编译的程序。

（4）取消挂载：

```
#umount /mnt
```

2、建立网络文件系统

(1)在主机上，把自己当前使用的根文件系统 rootfs 移置 nfsrootfs 目录下,同时需要确保相关文件的链接路径不能有错,(当真实环境的 rootfs 使用)

```
#cp rootfs /home/xxx/nfsrootfs -rf
```

(2) 重启板子,进入 pmon 中,设置启动参数;

```
PMON> set al '/dev/mtd0'
PMON> set append 'g root=/dev/nfs rw
nfsroot=192.168.3.xxx:/home/xx/nfsrootfs/rootfs noinitrd init=/linuxrc
console=ttyS4,115200 ip=192.168.3.x:::::eth0:off'
```

提示：xxx - 主机的 ip 地址 ，/home/xx/nfsrootfs/rootfs - 自己在服务器的目录， x - 自己分 配给板子的 ip 地址。

（3）重启，进入网络文件系统。