安全提示

非常感谢您购买方源智能科技的产品，在打开包装箱后请首先依据物件清单检查配件，若发现物件有所损坏、或是有任何配件短缺的情况，请尽快与您的经销商联络。

- 产品使用前，务必仔细阅读产品相关说明。
- 主板与电源连接时，请确认电源电压。
- 为了保证您是使用安全，请使用平台的专用电源。
- 接触平台主板前，应将手先置于接地金属物体上一会儿，以释放身体及手中的静电。
- 为避免人体被电击或产品被损坏，在每次对主板、扩展卡进行拔插或重新配置时，须先关闭交流电源或将交流电源线从电源插座中拔掉。
- 在对平台进行搬动前，先将交流电源线从电源插座中拔掉。
- 当您需连接或拔除设备前，须确定所有的电源线事先已被拔掉。
- 设备在使用过程中出现异常情况，请找专业人员处理。

版本声明

本文档为全功能物联网教学科研平台配套实验说明。所述实验内容仅限该实验平台使用。

修订描述				
日期	修订版本	描述	编辑	备注
2017-09-27	V1.0	Alpha Edition	Cyb-bot	
2019-05-21	V1.1	文档修正	chensd	

ICS-IOT-CEP 型全功能物联网教学科研实验平台配套实验指导文档共包含三册，本册为第二册，可用于 ICS-IOT-CEP 型全功能物联网教学科研实验平台配套 Cortex-A53 智能终端部分课程实验参考。

目录

第一章. 实验环境与软件工具 4
1. 开发平台简介 5
2.1 产品概述 5
2.2 产品特点 6
2.3 平台硬件资源 6
2.4 平台软件资源 8
2.5 平台的连线使用 9
2. Linux 系统开发环境 14
2.1 嵌入式 Linux 系统开发环境 14
2.2 常见的软件工具 20
第二章. 基础应用实验 24
实验一. 实验环境使用入门 25
实验二. 多线程程序设计 29
实验三. 串口程序设计 40
实验四. SOCKET 网络编程 53
实验五. 嵌入式 SQLite 应用 63
实验六. 嵌入式 WebServer 移植 74
第三章. 基于 Qt 的 GUI 实验 82
实验一. 搭建本机 Qt 开发环境 83
实验二. 基于 QtDesigner 的程序设计 89
实验三. 搭建 Qt/Embedded ARM 环境 96
第四章. 底层系统构建实验 103
实验一. Linux 内核裁剪与编译 104
实验二. 构建根文件系统 128
第五章. 设备底层驱动实验 141
实验一. 按键中断驱动及控制 142
实验二. PWM 蜂鸣器驱动及控制 156
实验三. ADC 驱动及采样 168
实验四. LCD 设备驱动及控制 178
实验五. SD 卡接口实验 195
实验六. U 盘接口使用 213
第六章. 综合实训案例 223
实验一. 基于 RFID 无线射频的电子钱包实训案例 224
实验二. 基于无线传感网的智能家居实训案例 245
实验三. 基于无线传感网的智慧农业实训案例 255
实验四. 基于无线传感网的智能空调实训案例 267

第一章. 实验环境与软件工具

本章主要介绍基于 ICS-IOT-CEP 全功能物联网教学科研平台软硬件的资源，以及相应实验体系的开发环境（Linux 系统的开发环境）搭建过程，并着重讲解在使用该平台过程中用到的一些常见开发软件和相关服务设置。

本章内容作为 ICS-IOT-CEP 全功能物联网教学科研平台实验教学的基础内容，是顺利完成后续章节实验内容的重要基石，请用户或读者在动手实验之前务必首先仔细阅读本章内容。

1. 开发平台简介

1.1 产品概述

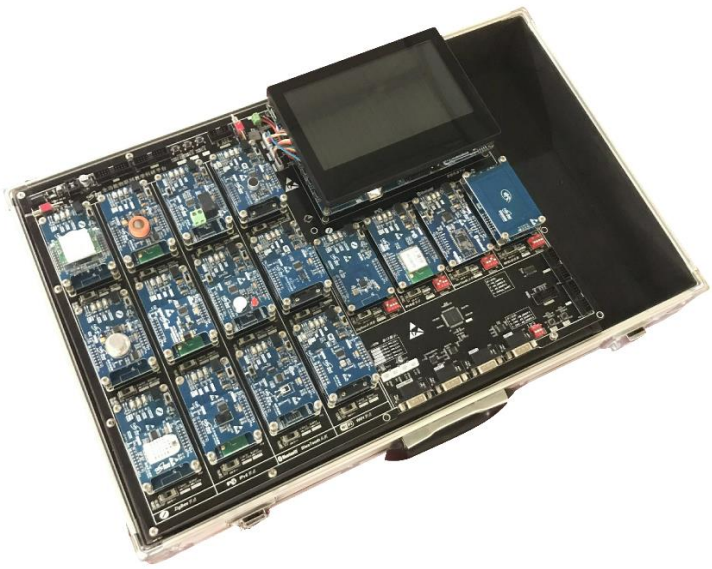


图 1.1.1.1　ICS-IOT-CEP 开发平台

全功能物联网教学科研平台是方源智能（北京）科技有限公司基于物联网多功能、全方位教学科研需求，推出的一款集无线 ZigBee、IPv6、Bluetooth、Wifi、RFID 和智能传感器等通信模块于一体的全功能物联网教学科研平台，以强大的 Cortex-A53（可支持 Linux/Android）嵌入式处理器作为核心智能终端，支持多种无线传感器通讯模块组网方式。由浅入深，提供丰富的实验例程和文档资料，便于物联网无线网络、传感器网络、RFID 技术、嵌入式系统及下一代互联网等多种物联网课程的教学和实践。

全功能物联网教学科研平台的应用结构拓朴图如下图 1.1.1.2 所示：

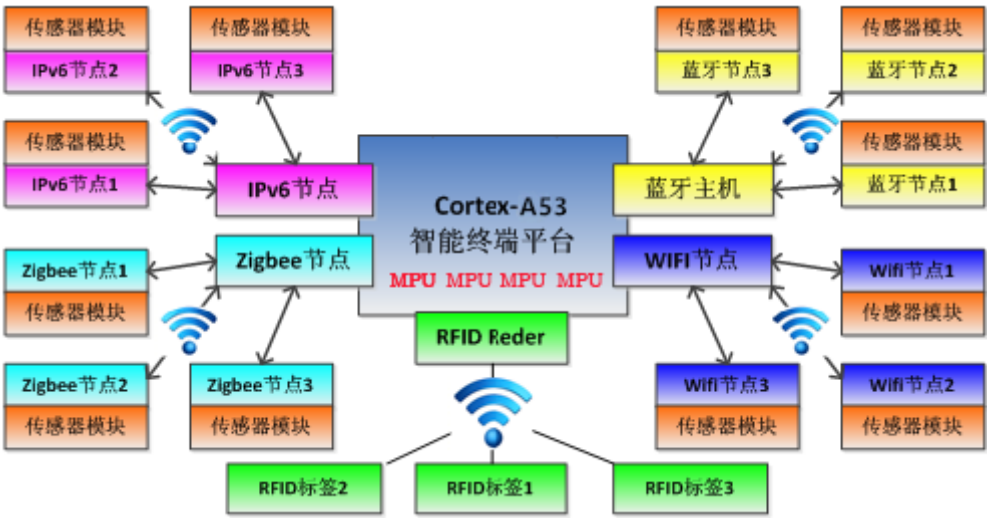


图 1.1.1.2　ICS-IOT-CEP 平台应用拓扑结构

1.2 产品特点

- 丰富快捷的无线组网功能

系统配备 ZigBee、IPv6、蓝牙、WIFI 四种无线通信节点及 RFID 读/写卡器，可以快速构成小规模 ZigBee、IPv6、蓝牙、WIFI 无线传感器通信网络。

- 丰富的传感器数据采集和扩展功能

配备温湿度、光敏、震动、三轴加速计、红外热释、烟雾等 12 种基于 MCU 的智能传感器模块，可以通过标准接口与通信节点建立连接，实现传感器数据的快速采集和通信。

- 可视化终端界面开发

基于 Qt 的跨平台图形界面开发，用户可以快速开发友好的人机界面。

1.3 平台硬件资源

全功能物联网科研教学平台硬件由 Cortex-A53 嵌入式智能终端、无线通讯模块和智能传感器模块几部分构成。

Cortex-A53 智能终端	
CPU 处理器	处理器 Samsung S5P6818,八核心处理器,基于 ARM Quad CortexM-A53,运行主频 1.4GHz 内置 Mali-400 MP 高性能图形引擎 支持流畅的 2D/3D 图形加速 最高可支持 1080p@60fps 硬件解码视频流畅播放，格式可为 MPEG4,H.263,H.264 等 最高可支持 1080p@30fps 硬件编码(Mpeg-2/VC1)视频输入
RAM 内存	1G DDR3 32bit 数据总线，单通道 运行频率: 400MHz
FLASH 存储	emmc 8GB
显示	7 寸 LCD 液晶电阻触摸屏
接口	1 路 HDMI 输出 4 路串口，RS232 *2、TTL 电平 *4 USB Host 2.0、mini USB Slave 2.0 接口 3.5mm 立体声音频(WM8960 专业音频芯片)输出接口、板载麦克风 1 路标准 TF 卡座 千兆以太网 RJ45 接口

	CMOS 摄像头接口 其中 AIN0 外接可调电阻，用于测试 I2C-EEPROM 芯片(256byte) ，主要用于测试 I2C 总线 用户按键(中断式资源引脚) *5 PWM 控制蜂鸣器 板载实时时钟备份电池
电源	电源适配器 5V(支持睡眠唤醒)

表 1.1.3.1　Cortex-A53 智能终端

无线通信节点	
ZigBee 节点（TI 方案标配）	4、处理器 CC2530，内置增强型 8 位 51 单片机和 RF 收发器，符合 IEEE802.15.4/ZigBee 标准规范，频段范围 2045M-2483.5M,板载天线
	5、存储器：256KB 闪存和 8KB RAM
	6、射频数据速率：250kbps，可编程的输出功率高达 4.5 dB
	7、用户自定制：按键 *2，LED *2
	8、供电电压：2V-3.6V，支持电池供电
	9、扩展调试接口
IPv6 节点	10、处理器 STM32W108，基于 ARM Cortex-M3 高性能的微处理器，集成了 2.4GHz IEEE 802.15.4 射频收发器，板载天线
	11、存储器：128KB 闪存和 8KB RAM，
	12、用户自定制：按键 *1，LED *2
	13、供电电压：3.7V　收发电流： 27mA/40mA，支持电池供电
	14、扩展 J-Link 调试接口
蓝牙节点	15、CC2540 蓝牙模块，板载天线
	16、处理器 STM32F103 基于 ARM Cortex-M3 内核，主频 72MHz
	17、完全兼容蓝牙 4.0 规范，硬件支持数据和语音传输，最高可支持 3M 调制模式
	18、支持 UART 透传，IO 配置
	19、扩展 J-Link 接口，外设主从开关，支持一键主从模式转换
	20、支持电池供电
WIFI 节点	21、型号：嵌入式 wifi 模块（支持 802.11b/g/n 无线标准）内置板载天线
	22、处理器 STM32F103 基于 ARM Cortex-M3 内核，主频 72MHz
	23、支持多种网络协议：TCP/IP/UD，支持 UART/以太网数据通讯接口
	24、支持无线工作在 STA/AP 模式，支持路由/桥接模式网络架构
	25、支持透明协议数据传输模式，支持串口 AT 指令
	26、扩展 J-Link 接口
	27、支持电池供电
RFID 阅读器	28、MF RC531（高集成非接触读写卡芯片）支持 ISO/IEC 14443A/B 和 MIFARE 经典协议
	29、处理器 STM8S105 高性能 8 位架构的微控制器，主频 16MHz
	30、支持 mifare1 S50 等多种卡类型
	31、用户自定制：按键 *1，LED *1
	32、最大工作距离：100mm，最高波特率：424kb/s
	33、Crypto1 加密算法并含有安全的非易失性内部密匙存储器
	34、扩展 ST-Link 接口

表 1.1.3.2　无线通讯模块

传感器模块		
处理器	● STM8S103 高性能 8 位框架结构的微控制器，主频 1MHz	
外设	● LED 灯、UART 串口及电源接口	
传感器种类	● 磁检测传感器 ● 光照传感器 ● 红外对射传感器 ● 红外反射传感器 ● 结露传感器 ● 酒精传感器	● 人体检测传感器 ● 三轴加速度传感器 ● 声响检测传感器 ● 温湿度传感器 ● 烟雾传感器 ● 振动检测传感器

表 1.1.3.3　传感器模块

外扩辅助模块	
USB-UART 扩展板	**核心芯片：** FT232RL
	功能：连接 PC 机与网络节点串口调试功能
	接口：VCC GND TXD GND RXD
电池模块	**功能：**锂电池供电,提供低电压报警,提示用户充电
	接口：3.7V
电池充电板	5V 电源适配器,双路锂电池充电
调试工具	ST-Link 仿真调试工具、J-Link 仿真调试工具

表 1.1.3.4　外扩模块

1.4　平台软件资源

Cortex-A53 智能终端平台	**操作系统：** Linux-3.4.9 + Qt4.7/Qtopia2/Qtopia4、Android 5.1 **实验内容：**可进行 Linux 系统嵌入式编程开发，包括开发环境搭建、Bootloader 开发、嵌入式操作系统移植、驱动程序调试与开发、应用程序的移植与项目开发等。
ZigBee 通信节点(ST 方案选配)	**开发环境：**IAR for STM32W108 **协议：**ZigBee pro 协议(EmberZNet 4.30 协议栈) **功能：**自动组网、无线数据传输等
ZigBee 通信节点(TI 方案标配)	**开发环境：**基于 IAR for 8051 **协议:** ZigBee pro 协议(Z-Stack2007 协议栈) **功能：**自动组网、自动路由、无线数据传输等
IPv6 通信节点	**操作系统：**Contiki 2.5 **协议：**基于 Contiki OS 在 802.15.4 平台上实现完整的 IPv6 协议(Contiki OS uIPv6 协议栈) **功能：**自动组网、自动路由、无线数据传输等
蓝牙通信节点	**协议：**完整的蓝牙通信 4.0 协议 **功能：**蓝牙模块组网、SPP 蓝牙串行服务、无线数据传输等。

WIFI 通信节点	**网络类型：** Station/AP 模式 **安全机制：** WEP/WAP-PSK/WAP2-PSK/WAPI **加密类型：** WEP64/WEP128/TKIP/AES **工作模式：** 透明传输模式，协议传输模式 **串口命令：** AT+命令结构 **网络协议：** TCP/UDP/ARP/ICMP/DHCP/DNS/HTTP **最大 TCP 连接数：** 32 **功能：** 自动组网、支持 AP 模式/AT 命令、无线数据传输等
RFID 阅读器	**功能：** 支持与节点通信、组网，支持快速 CRYPTO1 加密算法、IC 卡识别、IC 卡读写
传感器模块	**功能：** 基于 IAR for STM8 的开发环境，实现传感器数据采集与串口协议通讯

表 1.1.4.1　软件资源

1.5　平台的连线使用

1. 电源

平台使用 5V 直流电源适配器进行供电，切勿插错电源！以免损坏器件。主板上的 Cortex-A53 终端和无线通讯模块接有单独的电源开关，使用时，请有选择的打开电源。另外主板上的无线通讯模块可以单独使用平台配套锂电池供电。

注意：主板电源与锂电池不可同时使用。

2. 连线

Cortex-A53 智能终端在进行实验时，通常需要连接平台配套串口线和网线。串口线连接至 Cortex-A53 默认串口 0 接口（RS232）。

无线通讯模块在进行实验时，根节点（LCD 下方的模块）可以使用主板上的 RS232 口，子节点（LCD 左侧 12 个模块）和传感器模块默认无法使用 RS232 串口，需要时可以拔下子节点或传感器模块，通过平台配套 USB2UART 模块连接至 PC 机转接出来串口使用，此时采用 USB 线供电。

3. 启动与调试

Cortex-A53 终端的启动可以支持从 SD 卡和 FLASH 两种方式，可自动选择启动方式。

连接平台配套 5V 电源适配器，启动 Cortex-A53 智能终端电源开关，我们可以通过平台配套的串口线与 PC 机连接。PC 端需要安装相应串口终端软件，这种软件有很多，介绍通过 Xshell 软件与平台 ARM Linux 系统连接。（在光盘 Tools 目录下，选择 Xshell 进行安装，安装过程中选择“免费为学校/教育”）

打开安装完成的 Xshell 软件：

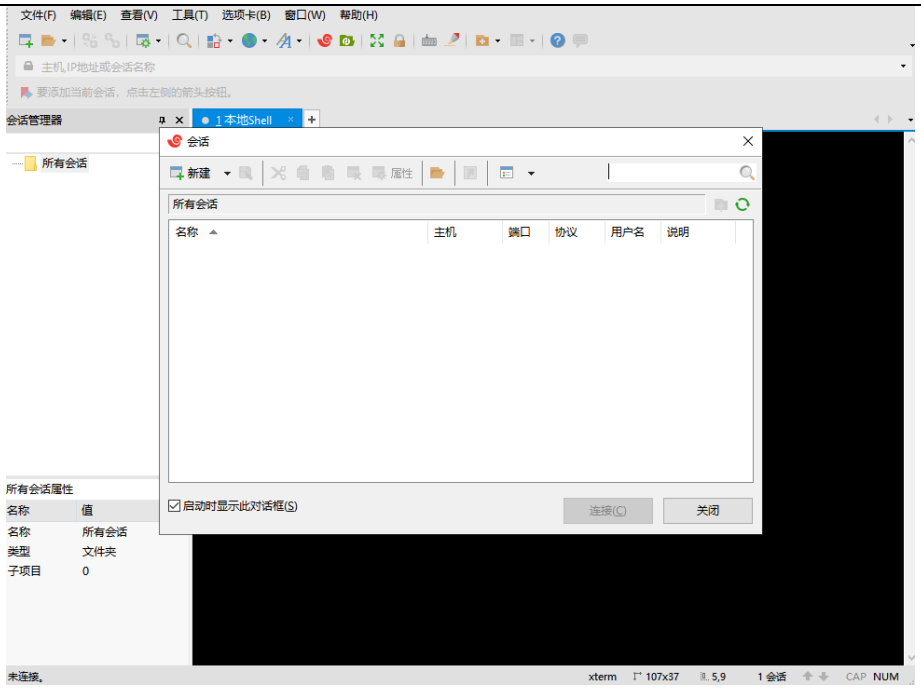


图 1.1.5.1　Xhell

在弹出的位置对话框中点击新增按钮。

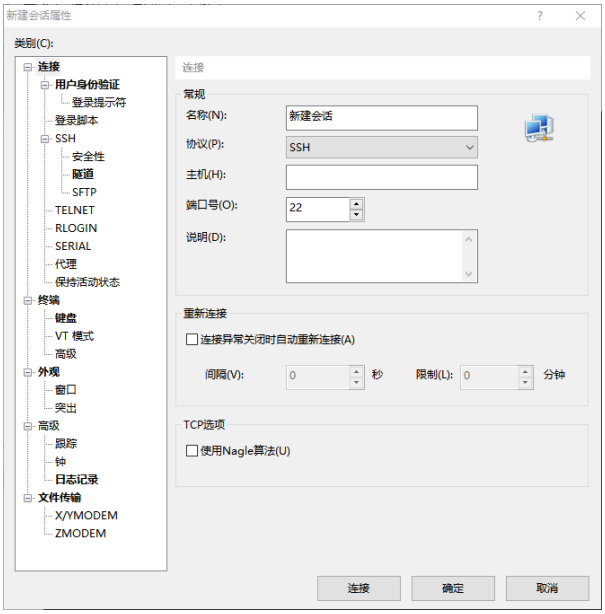


图 1.1.5.2　超级终端 电话选项

选择协议为“SERIAL”。

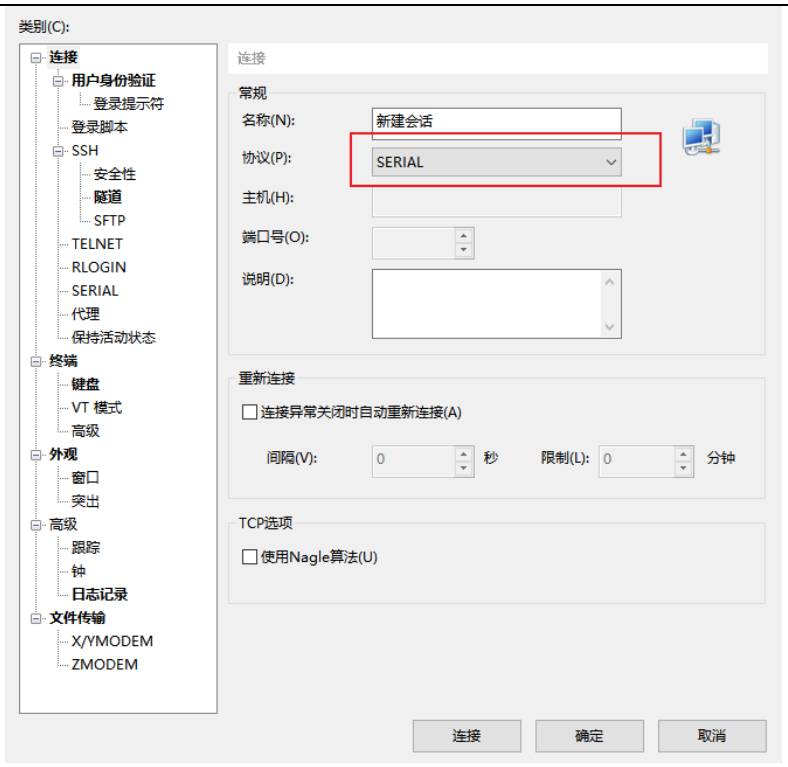


图 1.1.5.3　连接协议

选择左侧的“SERIAL”配置功能，根据 PC 机串口设备号，选择 COM1，设置串口属性：波特率 115200，数据位 8，奇偶校验 无，停止位 1，流控制 无，选择应用，选择确定。

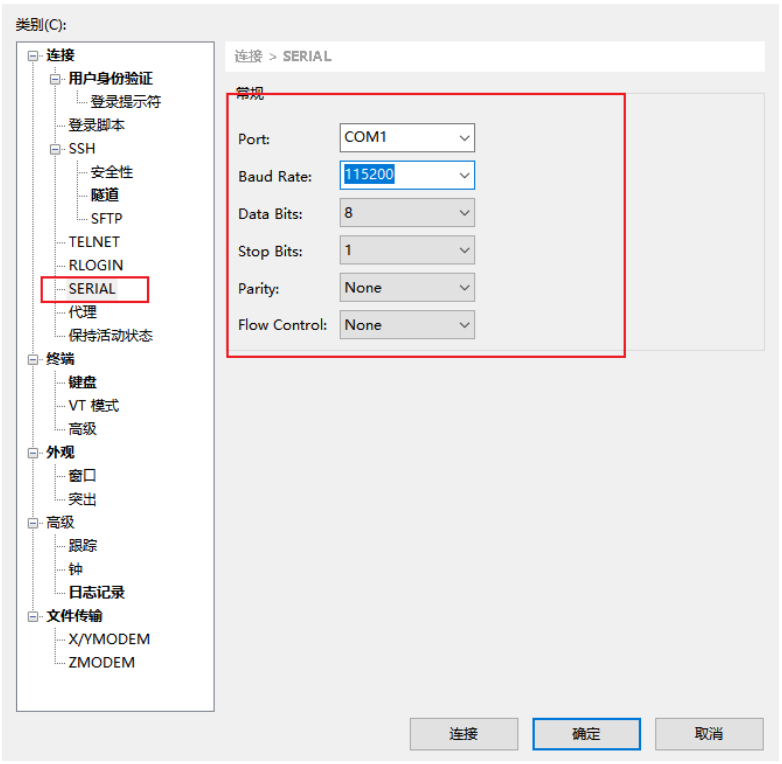


图 1.1.5.4　串口配置

双击会话中新增的条目。

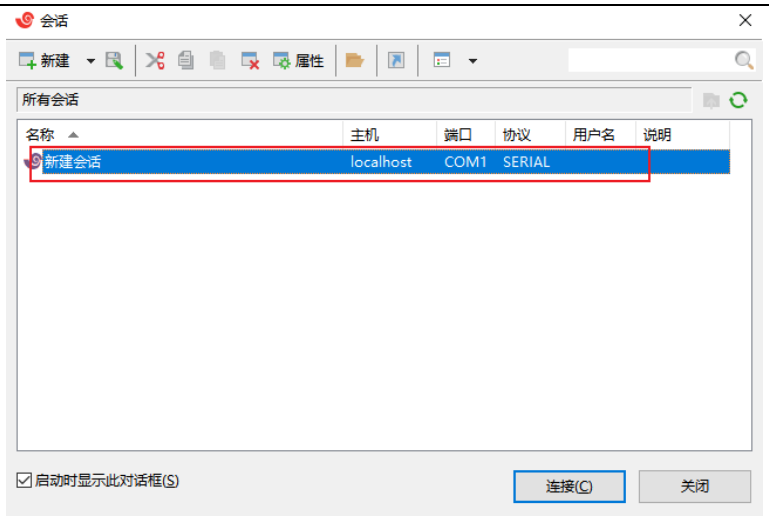


图 1.1.5.5　连接串口

之后便可以重启 ICS-IOT-CEP Cortex-A53 智能终端系统，在超级终端中登录系统了。

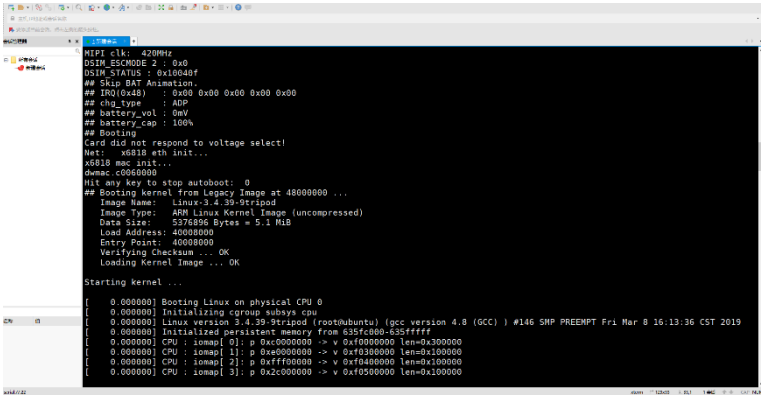


图 1.1.5.6　启动信息

主板上有 2 组共 4 个 20P 标准 JTAG 接口（主板标记为 JLink 与 STLink），其中 1 组（P20/P21）用于根节点模块的下载和调试，一组（P14/P15）用于子节点和传感器模块的下载和调试：

1）J-Link 接口用于连接 J-Link 仿真器或者 ZigBee Debuger 仿真器完成对无线通讯模块的程序下载和调试。

2）ST-Link 接口用于连接 ST-Link 仿真器完成传感器模块的程序下载和调试。

由于多个模块复用 JTAG 调试口，上述目标模块调试的选定，可以通过主板上的 2 个选择按键进行目标模块的选择（有指示灯提示）。

4. UART 串口的使用

平台默认引出 4 路 RS232 串口，其中 Debug UART 为根节点调试使用串口，其余 3 个 UART 为 Cortex-A53 终端扩展串口，用于预留扩展使用其他模块。

关于根节点串口的连接选择是复用的，可以使用相应根节点下方的 4 位拨码开关 1234，其中 123 位分别对应根节点与 Cortex-A53 相应串口的连接，4 用于根节点与 Debug UART 串口的连接。同一时间，只能有一个根节点使用 Debug UART 串口，如果串口使用出错，相应的 UART ERROR 串口指示灯会点亮。

如果要使用 Cortex-A53 终端平台的 RS232 串口连接外扩模块，需要使用主板右下角 UART-1 RS232 串口一侧的 3 位拨码开进行相应选择，同时关闭根节点上方 4 位拨码开关的设置。拨码开关（向上为打开，向下为关闭）。

5. 其他

1）USB2UART 模块用于平台配套无线模块和传感器模块的串口调试工作，需要安装平台光盘提供的驱动方可正常使用。

2）锂电池模块非标配），配有配套的电源适配器，可以通过平台配套的锂电池充电模块进行充电。

3）更详细的硬件说明，请参考平台配套硬件说明书。

2. Linux 系统开发环境

2.1 嵌入式 Linux 系统开发环境

1、嵌入式 Linux 开发环境简介

- **嵌入式 LINUX 开发环境有几个方案：**

1）基于 PC 机 WINDOWS 操作系统下的 CYGWIN；

2）在 WINDOWS 下安装虚拟机后，再在虚拟机中安装 LINUX 操作系统；

3）直接安装 LINUX 操作系统。

实际如果 PC 机器性能的条件优越，用户可以选择第二种开发方式，使用 WindowsXP 系统运行 Vmware 虚拟机，在虚拟机中运行 ubuntu 系统，而且 Windows 系统下有很多好用的开发软件，如 Xshell、Sourceinsight 等。另外由于 ICS-IOT-CEP 全功能物联网教学科研平台涉及无线通讯模块的 Windows IDE 开发环境，因此建议用户使用第二种方案。

注意：本指导全文都是采用 Vmware12 虚拟机运行 ubuntu16 系统宿主机环境。

- **交叉编译环境**

绝大多数 Linux 软件开发都是以 native 方式进行的，即本机（HOST）开发、调试，本机运行的方式。这种方式通常不适合于嵌入式系统的软件开发，因为对于嵌入式系统的开发，没有足够的资源在本机（即板子上系统）运行开发工具和调试工具。通常的嵌入式系统的软件开发采用一种交叉编译调试的方式。交叉编译调试环境建立在宿主机（即一台 PC 机）上，对应的开发板叫做目标板。

运行 Linux 的 PC（宿主机）开发时使用宿主机上的交叉编译、汇编及连接工具形成可执行的二进制代码（这种可执行代码并不能在宿主机上执行，而只能在目标板上执行），然后把可执行文件下载到目标机上运行。调试时的方法很多，可以使用串口，以太网口等，具体使用哪种调试方法可以根据目标机处理器提供的支持作出选择。宿主机和目标板的处理器一般不相同，宿主机为 Intel 处理器，而目标板如 ICS-IOT-CEP 全功能物联网教学科研平台为三星 S5P6818 处理器。

所以在进行嵌入式开发前第一步的工作就是要安装一台装有指定操作系统的 PC 机作宿主开发机，对于嵌入式 LINUX，宿主机上的操作系统一般要求为 UBUNTU LINUX。嵌入式开发通常要求宿主机配置有网络，支持 NFS（开发时 mount 所用）。然后要在宿主机上建立交叉编译调试的开发环境。环境的建立需要许多的软件模块协同工作，这将是一个比较繁杂的工作，通常都由目标产品配套的产品光盘提供支持。

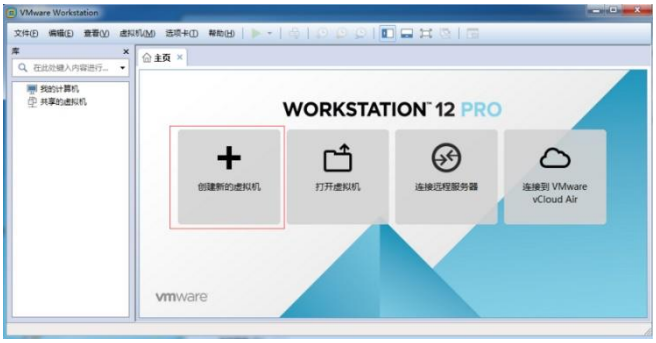
2、宿主机 ubuntu 的安装

- **安装 vmware workstation：**

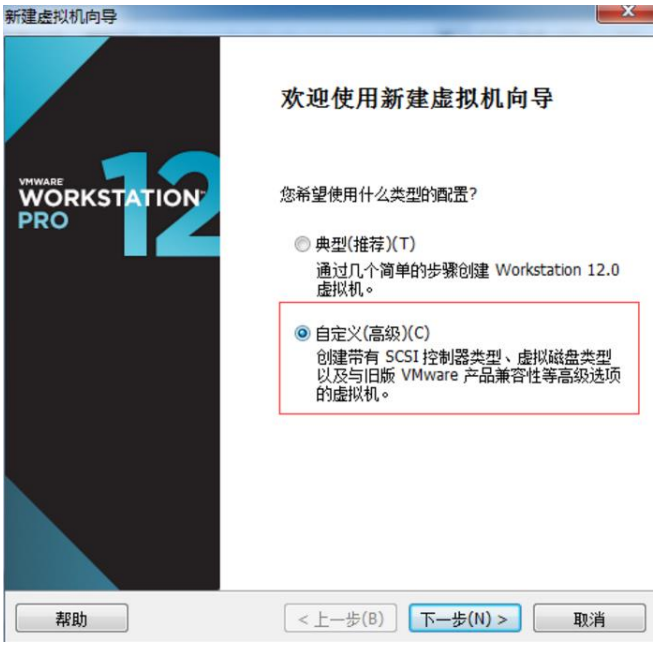
使用光盘提供或网络下载的 vmware workstation 12 安装包进行安装，安装装过程全部采用默认配置。

- **创建虚拟机**

打开安装好的 vmware workstation 12，选择创建新的虚拟机，如下图所示：

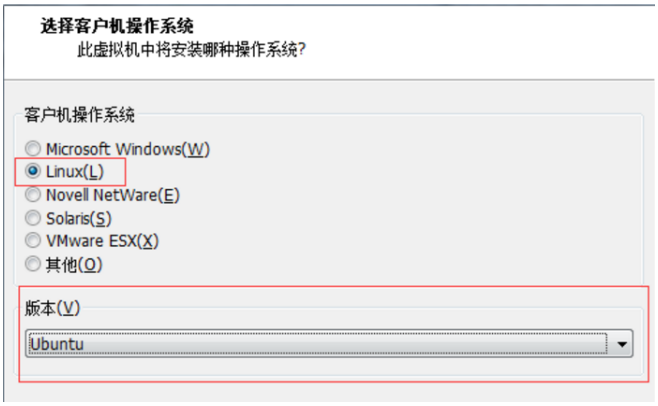


在向导界面选择自定义的配置，如下图所示：



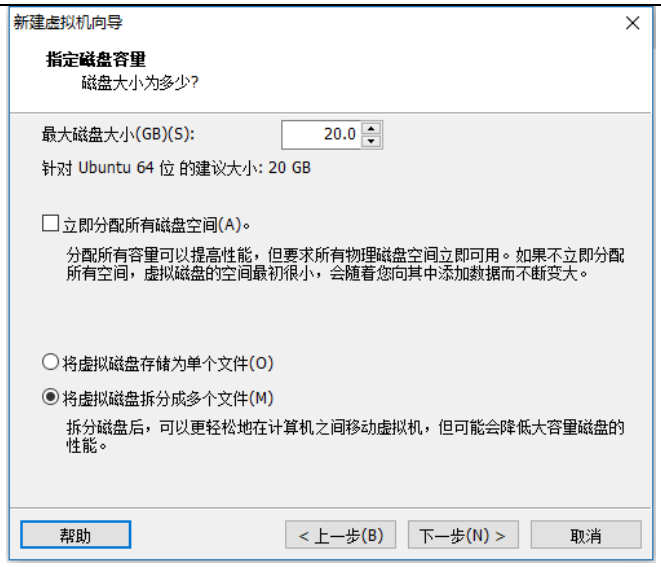
选择下一步，按照默认设置继续选择，直到出现如下配置，安装来源选择稍后安装操作系统，如下图所示：

选择下一步，选择客户机操作系统为 linux（注意这里下面的下拉选择 Ubuntu64，因为本实验系统必须使用 64 位操作系统，如下拉选择 Ubuntu，可能导致安装出错）。

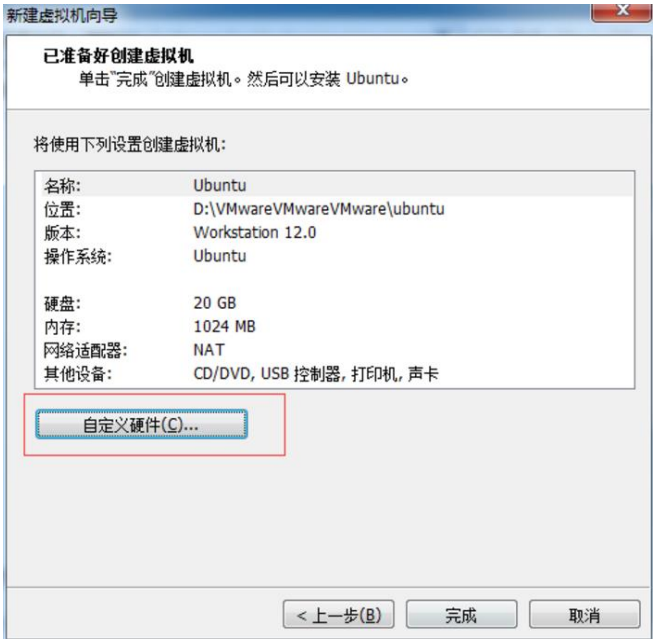


点击下一步，选择安装位置（这里必须输入一个已存在的目录，不然后面会报错的），如下图所示：

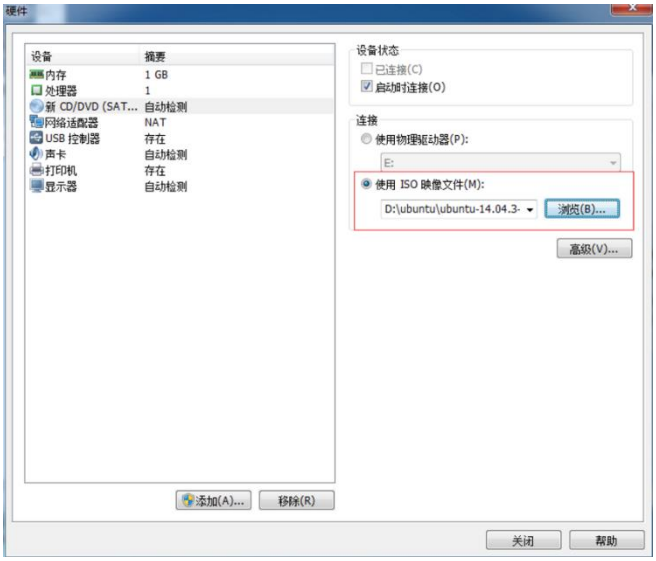
点击下一步，设置处理器和内存，可按照默认配置安装，并选择将虚拟磁盘拆分为多个文件，如下图所示：



点击下一步，到如下界面，点击自定义硬件。



在下图所示的界面中，选择下载的 ubuntu16.04 的镜像：



关闭选项卡，点击完成后，选择开启此虚拟机，按照提示步骤完成 ubuntu 的设置。

注意：通常我们安装好 ubuntu 系统后，要通过系统的 apt 工具完成必要开发软件及开发库的下载与安装。关于 apt 的使用方法，用于可以参考相关数据或网络资源。

3、产品光盘实验环境的安装

- **Linux 系统下挂载光驱：**

先打开宿主机端 Linux 系统，再将北京方源智能科技有限公司产品附带配套产品光盘插入电脑光驱中（鼠标放入 Linux 系统桌面环境中，可以避免 Windows 系统先识别光驱），如果 Linux 桌面系统自动识别并挂载了产品光盘，则可以直接拷贝使用。如果系统未识别，可以通过命令方式进行光盘挂载。

```
cbt@Cyb-Bot:~$ mount /dev/cdrom /mnt/cdrom/
```

具体系统识别光驱的设备用户可以自行在/dev/目录下查看，挂载的目录/mnt/cdrom 仅供参考。

另外用户也可以按照下节内容介绍，使用 SMB 共享方式实现光盘内容的拷贝。

- **拷贝光盘内容至宿主机 ubuntu**

上述光盘挂载成功后，用户即可以将产品光盘中的实验资料及工具拷贝安装到宿主机开发环境中了，其中我们可以使用产品光盘自带的安装脚本文件 install.sh 完成光盘的拷贝和工具安装工作。

进入光盘 Linux 目录下，运行 install.sh 脚本。

```
cbt@Cyb-Bot:~$ cd /mnt/cdrom/Linux/
cbt@Cyb-Bot: Linux$ ./install.sh
```

Install.sh 脚本实际只是完成光盘内容的拷贝、解压等简单动作。安装成功后，会在宿主机系统中根目录下创建/CBT-6818 实验目录，里面存放所有实验代码和工具。如果您的 PC Linux 系统已经安装过该产品光盘的 install.sh 脚本文件，无需再次运行安装。

- **安装交叉编译器**

不同目标硬件平台使用的交叉编译器是有严格的版本环境要求的，因此需要配合 ICS-IOT-CEP 全功能物联网教学科研平台硬件安装产品光盘中自带的配套交叉编译器。

交叉编译器的解压和安装实际在前一个步骤中已经完成，默认安装在宿主机系统的/opt/Cyb-Bot/toolschain/4.5.1/bin/目录中，用户可以进入该目录查看交叉编译器是否安装解压好。如果出现问题，可以从新手动从光盘拷贝解压出来。

用户只需将该交叉编译器的路径添加至系统默认环境变量中，以后即可以使用该交叉编译器了。

修改系统/etc/profile 文件(.开头命名的系统隐藏文件)，添加交叉编译器路径：

```
# /etc/profile: system-wide .profile file for the Bourne shell (sh(1))
```

```
# and Bourne compatible shells (bash(1), ksh(1), ash(1), ...).

if [ "$PS1" ]; then
  if [ "$BASH" ] && [ "$BASH" != "/bin/sh" ]; then
    # The file bash.bashrc already sets the default PS1.
    # PS1='\h:\w\$ '
    if [ -f /etc/bash.bashrc ]; then
      . /etc/bash.bashrc
    fi
  else
    if [ "`id -u`" -eq 0 ]; then
      PS1='# '
    else
      PS1='$ '
    fi
  fi
fi

if [ -d /etc/profile.d ]; then
  for i in /etc/profile.d/*.sh; do
    if [ -r $i ]; then
      . $i
    fi
  done
  unset i
fi
#添加如下一行路径
export PATH=$PATH:/opt/Cyb-Bot/toolschain/4.5.1/bin/
```

退出保存。

重启配置：

```
cbt@Cyb-Bot:~$ source /etc/profile
```

重新打开一个终端，即可使用 arm-linux-gcc 交叉编译器。

arm-linux-gcc 同样可以通过 which 命令查看交叉编译器的存放路径：也可以通过 arm-linux-gcc －v 命令查看交叉编译器版本。

```
cbt@Cyb-Bot:~$ which arm-linux-gcc
/opt/Cyb-Bot/toolschain/4.5.1/bin/arm-linux-gcc
```

注意：默认情况下，交叉编译器在光盘执行 install.sh 脚本时候已经安装好，用户无需手动安装。

- **安装光盘后的目录结构**

表格 1.2.1 均以 ubuntu 系统下/CBT-6818 目录为起点根目录：

目录	内容说明	
SRC	CBT-6818 系统实验代码目录	
	exp	实验工程目录
	linux	内核与文件系统源码目录
	gui	Qt 图形开发环境
IMG	CBT-6818 系统烧写镜像目录	
	uboot	bootloader 镜像文件目录
	Linux	内核、文件系统镜像文件
	fastboot	Windows 系统 fastboot 下载软件
CrossTools	CBT-6818 系统配套交叉编译器目录	
Tools	CBT-6818 系统开发常用使用软件	
install.sh	CBT-6818 系统光盘安装脚本文件	

表 1.2.1　安装目录结构

2.2常见的软件工具

1、Samba 共享

在使用 vmware 虚拟机 Linux 系统的时候，经常需要实现 Linux 系统与 windows 系统之间的文件共享。Samba 共享服务就是 Linux 系统提供的基于网络共享方式的服务。我们可以将 Linux 系统中的一个目录或文件夹设置成 samba 共享，在 windows 系统中通过合法的 Linux 用户名和密码实现访问。以下将提供该共享的设置和使用方法。

给已有的 root 用户添加 samba 共享：

1）设置 root 用户 samba 共享及密码

```
cbt@Cyb-Bot:~$ smbpasswd -a cbt
New SMB password:输入密码，无回显。
Retype new SMB password:确认密码，无回显。
Added user cbt
cbt@Cyb-Bot:~$
```

上述命令是将已存在的 root 用户，设置成 samba 共享用户，系统默认将 root 用户的工作目录/root 目录设置成 samba 共享目录。

2）修改 samba 共享目录权限(可选)

一般情况下，普通用户的 samba 共享需要设置共享目录的访问权限，我们可以通过如下命令完成：

```
cbt@Cyb-Bot:~$ chmod 777 /home/cbt
```

实际如果是 root 用户的 samba 共享，无需设置目录权限。

3）关闭系统防火墙

```
cbt@Cyb-Bot:~$ sudo service ufw stop
```

```
cbt@Cyb-Bot:~$
```

4）启动 samba 共享服务

```
cbt@Cyb-Bot:~$ sudo service smbd restart
cbt@Cyb-Bot:~$
```

访问 samba 共享目录前，请确保系统开启了该服务。

5）在 windows 系统下访问 samba 共享

使用 windows 系统中 开始->运行，输入宿主机 Linux 系统的 IP 地址即可访问 samba 共享。

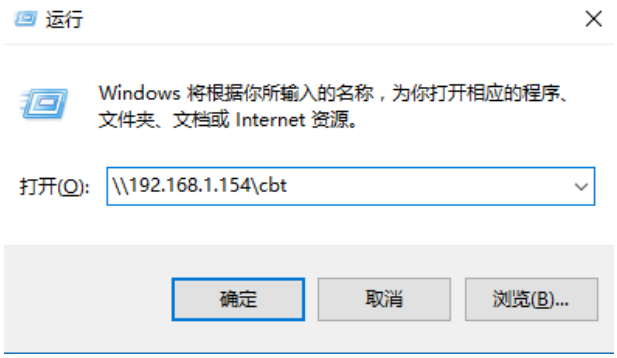


图 1.2.1 输入 Linux IP

在弹出的登录对话框中，输入上面设置的用户名和密码：

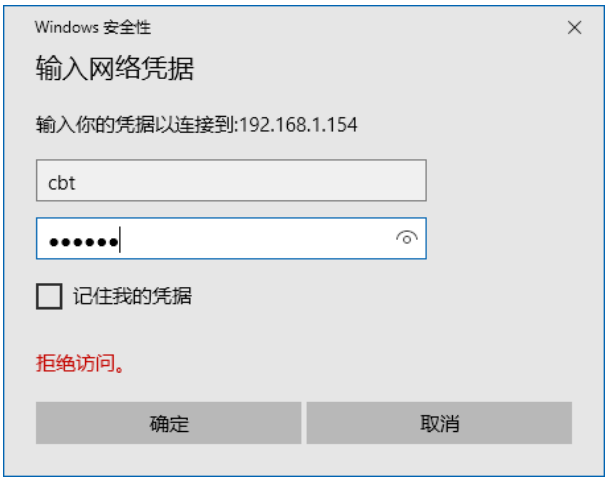


图 1.2.2 输入用户名 密码

之后便能看见 Linux 系统下 samba 共享目录：

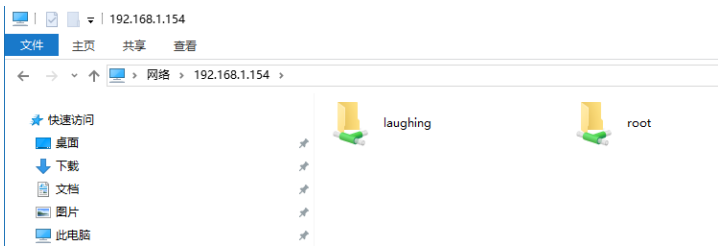


图 1.2.3 访问 samba 共享

之后便可以访问 root samba 共享文件夹了。

实际上，我们实验推荐使用高版本的 vmware12 及以上版本软件，其已经支持用复制和粘贴的方式来实现 windows 系统与虚拟机 Linux 系统间的文件拷贝了。当然实现两个系统间的文件共享方式和方法有很多，这里不再一一介绍，用户可以参考相关书籍和资料。

注意：在搭建 samba 共享服务时，请确保实验网络环境设置正确，如 ubuntu 宿主机的 IP 地址，windows 系统的 IP 地址，以及两者的防火墙设置等，强烈建议后续实验中将两者的防火墙关闭掉。

2、NFS 共享

NFS 共享也是基于网络的方式实现文件共享。在我们的实验环境中，主要是采用该种方式实现宿主机 ubuntu 系统与目标机 Cortex-A53 ARM Linux 系统直接实现文件的共享。其中我们大部分的实验都是采用这种方式进行的。NFS 共享实现了将宿主机 ubuntu 系统的目录设置才共享目录，在 ARM Linux 系统中使用 mount 挂载的方式进行访问和执行目标程序。以下将提供该共享的设置和使用方法。

1）添加 NFS 共享目录并设置权限

```
cbt@Cyb-Bot:~$ sudo vi /etc/exports
```

修改内容如下：

```
/CBT-6818 *(rw)
```

退出保存即可，该行语句表明，将系统的/CBT-6818 目录设置成共享，“*”代表任意机器都可以访问，rw 表示具有读写权限。

2）关闭系统防火墙

```
cbt@Cyb-Bot:~$ sudo service ufw stop
cbt@Cyb-Bot:~$
```

3）启动 NFS 共享服务

```
cbt@Cyb-Bot:~$ sudo service nfs-kernel-server restart
cbt@Cyb-Bot:~$
```

4）在 ARM Linux 系统中访问宿主机端 NFS 共享

```
[root@CBT-6818 /$ mount -t nfs -o nolock 192.168.1.7:/CBT-6818 /mnt/
```

mount 命令是在 CBT-6818 型实验平台上的 ARM Linux 系统的串口终端中使用的。挂载成功后，即可在 ARM 系统中访问远程宿主机端 NFS 共享实验目录了。

例如，文档实验环境 ubuntu 宿主机 IP 为 192.168.1.7，CBT-6818 型平台 ARM Linux 系统出厂默认 IP 地址为 192.168.1.230。

注意：在搭建 NFS 共享服务时，请确保实验网络环境设置正确，如 ubuntu 宿主机的 IP 地址和 ARM Linux 系统的 IP 地址保持同一个网段，并使用平台配套网线连接好宿主机和目标机系统。

后续所有实验文档介绍的 NFS 共享都统一使用的产品光盘安装后的宿主机目录/CBT-6818 目录。

3、Tftp 下载（可选）

Tftp 服务主要也是基于网络的文件传输，通常需要我们在宿主机 ubuntu 中安装 tftp-server，之后即可以使用 ARM Linux 系统的 tftp 命令从宿主机端下载文件了。当然不单单是 Linux 系统中有 tftp 服务，我们的 windows 系统下也有相应的软件支持，如光盘提供的 tftp32.exe 软件即是一款 windows 系统下的 tftp 服务器软件。以下将在宿主机 ubuntu 下 tftp 软件的设置和使用方法。

1）安装 tftp-server

如果宿主机 Linux 系统端，没有安装 tftp-server 软件，则需要利用网络进行安装，安装前要确保宿主机系统已经可以上网。

```
cbt@Cyb-Bot:~$ sudo apt-get install tftp-hpa tftpd-hpa
```

2）配置 tftp

修改/etc/default/tftp-hpa 文件，更改 tftp 下载目录和开启服务。

```
service tftp
{
        socket_type             = dgram
        protocol                = udp
        wait                    = yes
        user                    = root
        server                  = /usr/sbin/in.tftpd
        server_args             = -s /tftpboot   /*   更改默认下载目录为/tftpboot    */
        disable                 = no      /*       开启服务              */
        per_source              = 11
        cps                     = 100 2
        flags                   = IPv4
}
```

3）重启服务

```
cbt@Cyb-Bot:~$ sudo service tftpd-hpa restart
cbt@Cyb-Bot:~$
```

4）下载文件

宿主机端开启 tftp-server 服务后，将要下载的文件拷贝至/tftpboot 目录下，就可以用 ARM Linux 系统的 tftp 命令下载宿主机/tftpboot 目录下的文件了。例如：

```
[root@CBT-6818 /$ tftp -r test.txt -g 192.168.1.7
```

上面的 tftp 下载命令是在 ICS-IOT-CEP 全功能物联网教学科研平台的 Cortex-A53 智能终端的 ARM Linux 系统中使用的。

第二章. 基础应用实验

本章主要介绍在嵌入式 Linux 系统下进行常用的应用程序设计方法，包括基本的嵌入式开发方式和流程、多线程应用程序设计、串行端口设备的操作、SOCKET 客户端服务器程序设计相关编程接口、嵌入式数据库的使用以及嵌入式系统下进行视频采集等实验。通过上述的应用程序设计，使用户基本掌握在嵌入式 Linux 系统下进行应用程序设计的基本方法，熟悉嵌入式 Linux 系统下常用的编程接口与软件设计

实验一. 实验环境使用入门

1. 实验目的

- 熟悉嵌入式 Linux 开发环境，学会基于宿主机编译程序，目标机执行程序的交叉编译方法。
- 利用 arm-linux-gcc 交叉编译器编译程序，使用基于 NFS 的挂载方式进行实验，了解嵌入式开发的基本过程。

2. 实验环境

- 硬件：A53 智能网关实验平台，PC 机 Pentium 500 以上，硬盘 40G 以上，内存大于 256M。
- 软件：Vmware Workstation +ubuntu + MiniCom/超级终端 +ARM-LINUX 交叉编译开发环境。
- 实验目录：/CBT-6818/SRC/exp/basic/01_hello。

3. 实验内容

- 本次实验使用 ubuntu 操作系统环境,安装 ARM-Linux 的开发库及编译器。创建一个新目录，并在其中编写 hello.c 和 Makefile 文件。
- 学习在 Linux 下的编程和编译过程，以及 ARM 开发板的使用和开发环境的设置。将已经编译好的文件通过 NFS 方式挂载到目标开发板上运行。

4. 实验原理

♦ **嵌入式开发方式**

嵌入式开发当中通常会使用 2 种方式来运行应用程序：1、下载；2、挂载。

下载的方式：

使用 FTP、TFTP 等软件，利用宿主机与目标机的网络硬件进行，此种方法通常是将宿主机端编译好的目标机可执行的二进制文件通过网线或串口线下载固化到目标机的存储器(FLASH)中。

在目标机嵌入式设备存储资源有限的情况下受到存储容量的限制，因此，在调试阶段通常的嵌入式开发经常使用 NFS 挂载的方式进行。而在发布产品阶段才使用下载方式。

NFS 挂载方式：

利用宿主机端 NFS 服务，在宿主机端创建一定权限的 NFS 共享目录，在目标机端使用 NFS 文件系统挂载该目录，从而达到网络共享服务的目的。

这样做的好处是不占用目标机存储资源，可以对大容量文件进行访问。缺点是由于实际并没有将宿主机端文件存储到目标机存储设备上，因此掉电不保存共享文件内容。通常在嵌入式开发调试阶段，采用 NFS 挂载方式进行。

5. 实验步骤

♦ 编译源程序

1）创建实验目录

在宿主机端任意目录下建立工作目录 hello，实际光盘目录中已经给出本次实验所需全面文件及代码，存放在 01_hello 目录下。

```
cbt@Cyb-Bot:~$ mkdir  hello
cbt@Cyb-Bot:~$ cd  hello
```

2）编写程序源代码

在 Linux 下的文本编辑器有许多，常用的是 vim 和 Xwindow 界面下的 gedit 等，我们在开发过程中推荐使用 vim，用户需要学习 vim 的操作方法，请参考相关书籍中的关于 vim 的操作指南。 Kdevelope、anjuta 软件的界面与 vc6.0 类似，使用它们对于熟悉 windows 环境下开发的用户更容易上手。

实际的 hello.c 源代码较简单，如下：

```
#include <stdio.h>
main()
{
   printf("hello world \n");
}
```

我们可以是用下面的命令来编写 hello.c 的源代码，进入 hello 目录使用 vi 命令来编辑代码：

```
cbt@Cyb-Bot:~/01_hello$  vi  hello.c
```

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

3）编写 Makefile

Makefile 文件是在 Linux 系统下进行程序编译的规则文件，通过 Makefile 文件来指定和规范程序编译和组织的规则。

Makefile 文件的具体内容，用户可以参考本次实验目录下的 Makefile 中内容。

```
cbt@Cyb-Bot:~/01_hello$ cd /CBT-6818/SRC/exp/basic/01_hello/
cbt@Cyb-Bot: /CBT-6818/SRC/exp/basic/01_hello $ ls
Makefile  hello      hello.c    hello.o
```

Makefile 内容如下：

```
CC= arm-linux-gcc
EXEC = hello
OBJS = hello.o
CFLAGS +=
LDFLAGS+= –static

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

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

下面我们来简单介绍这个 Makefile 文件的几个主要部分：

- CC 指明编译器
- EXEC 表示编译后生成的执行文件名称
- OBJS 目标文件列表
- CFLAGS 编译参数
- LDFLAGS 连接参数
- all: 编译主入口
- clean： 清除编译结果

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

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

```
cbt@Cyb-Bot:~/01_hello$  vi  Makefile
```

4）编译应用程序

在上面的步骤完成后，我们就可以在 hello 目录下运行“make”来编译我们的程序了。如果进行了修改，重新编译则运行:

```
cbt@Cyb-Bot:~/01_hello$  make clean
cbt@Cyb-Bot:~/01_hello$  make
```

make clean 命令在第一次编译程序时候无需使用，在多次编译程序的时候可以用该命令来清除上次编译程序过程中生成的中间文件。这样做可以避免一些非改动的 make 编译错误提示。

注意：编译、修改程序都是在宿主机（PC 机 ubuntu）上进行，不能在 ARM 终端下进行。

♦ 运行程序（NFS 方式）

1）启动 CBT-6818 型实验系统，连好网线、串口线。通过串口终端挂载宿主机实验目

录。

在宿主机上启动 NFS 服务，并设置好共享的目录，具体配置请参照前面章节中关于嵌入式 Linux 环境开发环境的建立。在建立好 NFS 共享目录以后，我们就可以进入 ARM 串口终端建立开发板与宿主 PC 机之间的通讯了。

```
[root@CBT-6818:~]# mount -t nfs -o nolock 192.168.1.7:/CBT-6818      /mnt/
```

注意： IP 地址需要根据宿主机 PC 机的实际情况修改

成功挂接宿主机的 CBT-6818 目录后，在开发板上进入/mnt/目录便相应进入宿主机的/CBT-6818 目录，我们已经给出了编辑好的 hello.c 和 Makefile 文件，它们在/CBT-6818/SRC/exp/basic/01_hello 目录下。用户可以直接在宿主 PC 上编译生成可执行文件，并通过上面的命令挂载到开发板上，运行程序察看结果。

2）进入串口终端的 NFS 共享实验目录。

进入/mnt 目录下的实验目录，运行刚刚编译好的 hello 程序，查看运行结果。

```
[root@CBT-6818:~]# cd /mnt/SRC/exp/basic/01_hello/
[root@CBT-6818: /mnt/SRC/exp/basic/01_hello]# ls
Makefile    hello        hello.c      hello.o
```

3）执行程序。

执行程序用./表示执行当前目录下 hello 程序。

```
[root@CBT-6818 /mnt/SRC/exp/basic/01_hello]# ./hello
```

♦ **实验结果**

```
[root@CBT-6818 /mnt/SRC/exp/basic/01_hello]# ./hello
hello world
```

程序将向系统的标准输出，即串口控制台打印字符串“hello world”

注意：开发板挂接宿主计算机目录只需要挂接一次便可，只要开发板没有重启，就可以一直保持连接。这样可以反复修改、编译、调试，不需要下载到开发板。如果 NFS 挂载不稳定，可以使用挂载参数如：

mount -t nfs -o nolock,rsize=4096,wsize=4096 192.168.1.7:/CBT-6818   /mnt

实验二. 多线程程序设计

1. 实验目的

- 了解 Linux 下多线程程序设计的基本原理，学习 pthread 库函数的使用。
- 学习 Linux 系统下多线程间通讯的方法，掌握互斥锁和条件变量在多线程间通讯的使用方法。

2. 实验环境

- 硬件：A53 智能网关实验平台，PC 机 Pentium 500 以上，硬盘 40G 以上，内存大于 256M
- 软件：Vmware Workstation +ubuntu + MiniCom/超级终端 +ARM-LINUX 交叉编译开发环境
- 实验目录：/CBT-6818/SRC/exp/basic/02_pthread

3. 实验内容

- 学习 pthread.c 的源代码，熟悉几个重要的 pthread 库函数的使用。
- 在宿主机端交叉编译程序，在 ARM Linux 系统中运行程序，观察线程执行结果，分析 Linux 系统下多线程的工作特点。

4. 实验原理

4.1 多线程概述

线程 是进程的一条执行路径。每个线程共享其所附属的进程的所有的资源，包括打开的文件、页表（因此也就共享整个用户态地址空间）、信号标识及动态分配的内存等等。

线程和进程的关系是：线程是属于进程的，线程运行在进程空间内，同一进程所产生的线程共享同一物理内存空间，当进程退出时该进程所产生的线程都会被强制退出并清除。

线程技术早在 60 年代就被提出，但真正应用多线程到操作系统中去，是在 80 年代中期。传统的 Unix 也支持线程的概念，但是在一个进程（process）中只允许有一个线程，这样多线程就意味着多进程。现在，多线程技术已经被许多操作系统所支持，包括 Windows/NT，当然，也包括 Linux。

为什么有了进程的概念后，还要再引入线程呢？使用多线程到底有哪些好处？什么系统应该选用多线程？

使用多线程的理由之一是和进程相比，它是一种非常"节俭"的多任务操作方式。我们知道，在 Linux 系统下，启动一个新的进程必须分配给它独立的地址空间，建立众多的数据表

来维护它的代码段、堆栈段和数据段，这是一种"昂贵"的多任务工作方式。而运行于一个进程中的多个线程，它们彼此之间使用相同的地址空间，共享大部分数据，启动一个线程所花费的空间远远小于启动一个进程所花费的空间，而且，线程间彼此切换所需的时间也远远小于进程间切换所需要的时间。据统计，总的说来，一个进程的开销大约是一个线程开销的 30 倍左右，当然，在具体的系统上，这个数据可能会有较大的区别。

使用多线程的理由之二是线程间方便的通信机制。对不同进程来说，它们具有独立的数据空间，要进行数据的传递只能通过通信的方式进行，这种方式不仅费时，而且很不方便。线程则不然，由于同一进程下的线程之间共享数据空间，所以一个线程的数据可以直接为其它线程所用，这不仅快捷，而且方便。当然，数据的共享也带来其他一些问题，有的变量不能同时被两个线程所修改，有的子程序中声明为 static 的数据更有可能给多线程程序带来灾难性的打击，这些正是编写多线程程序时最需要注意的地方。

除了以上所说的优点外，不和进程比较，多线程程序作为一种多任务、并发的工作方式，还有以下的优点：

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

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

3）改善程序结构。一个既长又复杂的进程可以考虑分为多个线程，成为几个独立或半独立的运行部分，这样的程序会利于理解和修改。

Linux 系统下的多线程遵循 POSIX 线程接口，称为 pthread。编写 Linux 下的多线程程序，需要使用头文件 pthread.h，连接时需要使用库 libpthread.a。LIBC 中的 pthread 库提供了大量的 API 函数，为用户编写应用程序提供支持。

4.2 软件架构及流程

本实验为著名的生产者－消费者问题模型的实现，主程序中分别启动生产者线程和消费者线程。生产者线程不断顺序地将 0 到 1000 的数字写入共享的循环缓冲区，同时消费者线程不断地从共享的循环缓冲区读取数据。流程图如图 4.2.1 所示：

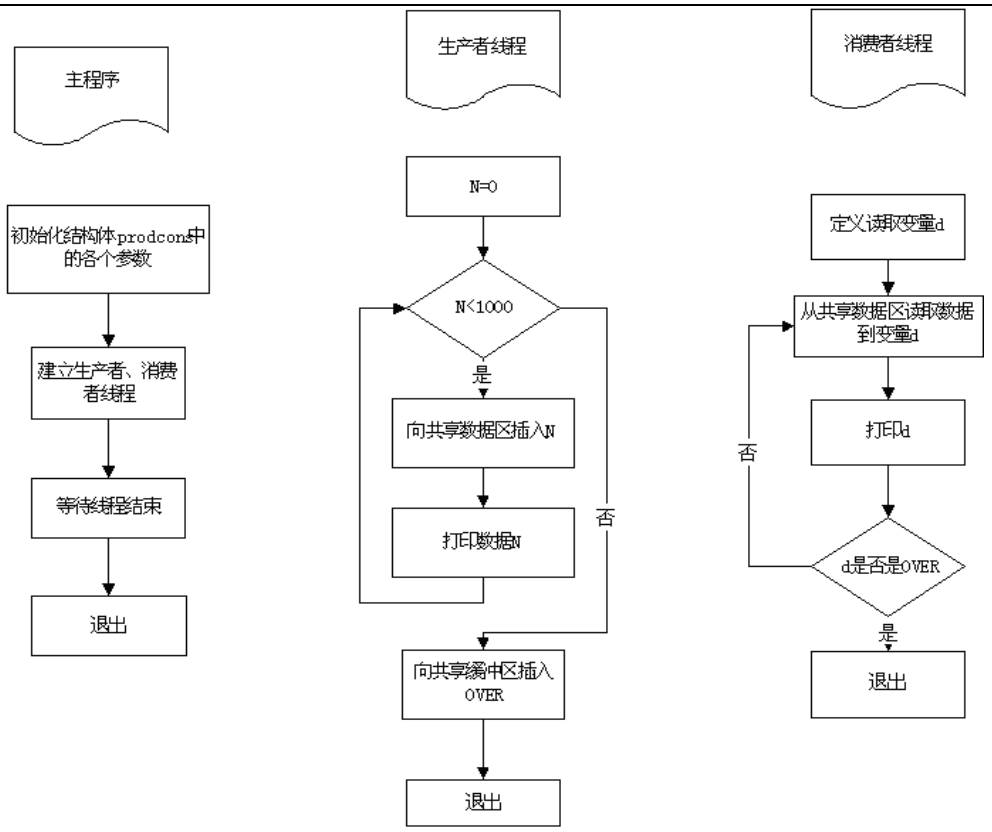


图 4.2.1 生产者-消费者实验源代码结构流程图

4.3 关键代码分析

♦ **pthread.c 源文件**

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "pthread.h"
#define BUFFER_SIZE 16
/* 设置一个整数的圆形缓冲区 */
struct prodcons {
    int buffer[BUFFER_SIZE];          /* 缓冲区数组 */
    pthread_mutex_t lock;             /* 互斥锁 */
    int readpos, writepos;            /* 读写的位置*/
    pthread_cond_t notempty;          /* 缓冲区非空信号 */
    pthread_cond_t notfull;           /*缓冲区非满信号 */
};
/*----------------------------------------------------*/
/*初始化缓冲区*/
void init(struct prodcons * b)
{
    pthread_mutex_init(&b->lock, NULL);
    pthread_cond_init(&b->notempty, NULL);
```

```
    pthread_cond_init(&b->notfull, NULL);
    b->readpos = 0;
    b->writepos = 0;
}
/*---------------------------------------------------------*/
/*  向缓冲区中写入一个整数*/
void put(struct prodcons * b, int data)
{
    pthread_mutex_lock(&b->lock);

        /*等待缓冲区非满*/
    while ((b->writepos + 1) % BUFFER_SIZE == b->readpos) {
        printf("wait for not full\n");
        pthread_cond_wait(&b->notfull, &b->lock);
        }
    /*写数据并且指针前移*/
        b->buffer[b->writepos] = data;
        b->writepos++;
        if (b->writepos >= BUFFER_SIZE) b->writepos = 0;
    /*设置缓冲区非空信号*/
        pthread_cond_signal(&b->notempty);

    pthread_mutex_unlock(&b->lock);
}
/*---------------------------------------------------------*/
/*从缓冲区中读出一个整数  */
int get(struct prodcons * b)
{
        int data;
    pthread_mutex_lock(&b->lock);

    /*  等待缓冲区非空*/
        while (b->writepos == b->readpos) {
        printf("wait for not empty\n");
        pthread_cond_wait(&b->notempty, &b->lock);
        }
        /*  读数据并且指针前移  */
        data = b->buffer[b->readpos];
        b->readpos++;
        if (b->readpos >= BUFFER_SIZE) b->readpos = 0;
        /*  设置缓冲区非满信号*/
        pthread_cond_signal(&b->notfull);

        pthread_mutex_unlock(&b->lock);
        return data;
}
```

```
/*-------------------------------------------------------*/
#define OVER (-1)
struct prodcons buffer;
/*-------------------------------------------------------*/
void * producer(void * data)
{
        int n;
        for (n = 0; n < 1000; n++) {
        printf(" put-->%d\n", n);
        put(&buffer, n);
   }
   put(&buffer, OVER);
   printf("producer stopped!\n");
   return NULL;
}
/*-------------------------------------------------------*/
void * consumer(void * data)
{
   int d;
   while (1) {
      d = get(&buffer);
      if (d == OVER ) break;
      printf("                    %d-->get\n", d);
   }
   printf("consumer stopped!\n");
   return NULL;
}
/*-------------------------------------------------------*/
int main(void)
{
        pthread_t th_a, th_b;
        void * retval;
        init(&buffer);
   pthread_create(&th_a, NULL, producer, 0);
        pthread_create(&th_b, NULL, consumer, 0);
   /* 等待生产者和消费者结束 */
        pthread_join(th_a, &retval);
        pthread_join(th_b, &retval);
        return 0;
}
```

下面我们来看一下，生产者写入缓冲区和消费者从缓冲区读数的具体流程，生产者首先要获得互斥锁，并且判断写指针+1 后是否等于读指针，如果相等则进入等待状态，等候条件变量 notfull；如果不等则向缓冲区中写一个整数，并且设置条件变量为 notempty，最后释放互斥锁。消费者线程与生产者线程类似，这里就不再过多介绍了。流程图如下图 4.2.2：

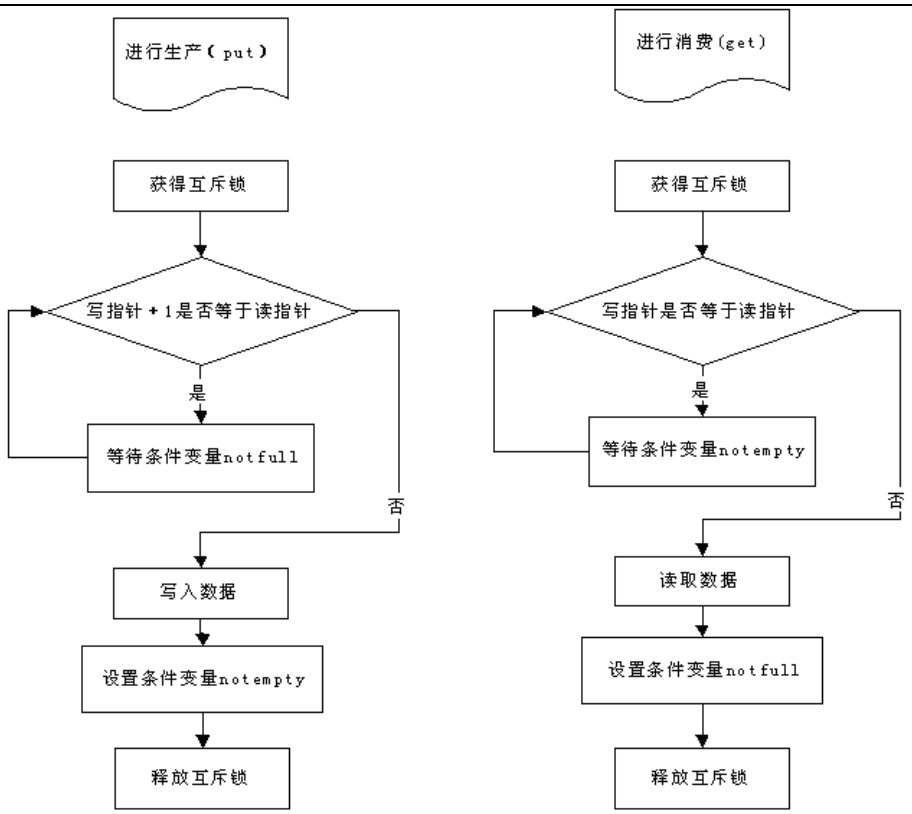


图 4.2.2 生产消费流程图

生产者写入共享的循环缓冲区函数 PUT。

```
void put(struct prodcons * b, int data)
{
  pthread_mutex_lock(&b->lock);                    //获取互斥锁
  while ((b->writepos + 1) % BUFFER_SIZE == b->readpos) {
//如果读写位置相同
        pthread_cond_wait(&b->notfull, &b->lock);
                                    //等待状态变量 b->notfull，不满则跳出阻塞
        }
        b->buffer[b->writepos] = data;                    //写入数据
        b->writepos++;
        if (b->writepos >= BUFFER_SIZE) b->writepos = 0;
        pthread_cond_signal(&b->notempty);                //设置状态变量
        pthread_mutex_unlock(&b->lock);                   //释放互斥锁
}
```

消费者读取共享的循环缓冲区函数 GET。

```
int get(struct prodcons * b)
{
        int data;
        pthread_mutex_lock(&b->lock);                     //获取互斥锁
        while (b->writepos == b->readpos) {               //如果读写位置相同
        pthread_cond_wait(&b->notempty, &b->lock);//等待状态变量 b->notempty，不空则跳出阻塞。否则无数据可读。
```

```
        }
        data = b->buffer[b->readpos];                    //读取数据
        b->readpos++;
        if (b->readpos >= BUFFER_SIZE) b->readpos = 0;
        pthread_cond_signal(&b->notfull);                //设置状态变量
        pthread_mutex_unlock(&b->lock);                  //释放互斥锁
        return data;
}
```

♦ **主要的多线程 API**

在本程序的代码中大量的使用了线程函数，如 pthread_cond_signal、pthread_mutex_init、pthread_mutex_lock 等等，这些函数的作用是什么，在哪里定义的，我们将在下面的内容中为大家做一个简单的介绍，并且为其中比较重要的函数做一些详细的说明。

线程创建函数：

```
int pthread_create (pthread_t * thread_id, __const pthread_attr_t * __attr,
                    void *(*__start_routine) (void *),void *__restrict __arg)
```

获得父进程 ID：

```
pthread_t pthread_self (void)
```

测试两个线程号是否相同：

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

线程退出：

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

等待指定的线程结束：

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

互斥量初始化：

```
pthread_mutex_init (pthread_mutex_t *,__const pthread_mutexattr_t *)
```

销毁互斥量：

```
int pthread_mutex_destroy (pthread_mutex_t *__mutex)
```

再试一次获得对互斥量的锁定（非阻塞）：

```
int pthread_mutex_trylock (pthread_mutex_t *__mutex)
```

锁定互斥量（阻塞）：

```
int pthread_mutex_lock (pthread_mutex_t *__mutex)
```

解锁互斥量：

```
int pthread_mutex_unlock (pthread_mutex_t *__mutex)
```

条件变量初始化：

```
int pthread_cond_init (pthread_cond_t *__restrict __cond,
__const pthread_condattr_t *__restrict __cond_attr)
```

销毁条件变量 COND：

```
int pthread_cond_destroy (pthread_cond_t *__cond)
```

唤醒线程等待条件变量：

```
int pthread_cond_signal (pthread_cond_t *__cond)
```

等待条件变量（阻塞）：

```
int pthread_cond_wait (pthread_cond_t *__restrict __cond, pthread_mutex_t *__restrict __mutex)
```

在指定的时间到达前等待条件变量：

```
int pthread_cond_timedwait (pthread_cond_t *__restrict __cond,
pthread_mutex_t *__restrict __mutex,    __const struct timespec *__restrict __abstime)
```

pthread 库中还有大量的 API 函数，用户可以参考其他相关书籍。下面我们对几个比较重要的函数做一下详细的说明：

pthread_create 线程创建函数。

```
int pthread_create (pthread_t * thread_id,__const pthread_attr_t *
                    void *(*__start_routine) (void *),void *__restrict __arg)
```

线程创建函数第一个参数为指向线程标识符的指针，第二个参数用来设置线程属性，第三个参数是线程运行函数的起始地址，最后一个参数是运行函数的参数。这里，我们的函数 thread 不需要参数，所以最后一个参数设为空指针。第二个参数我们也设为空指针，这样将生成默认属性的线程。当创建线程成功时，函数返回 0，若不为 0 则说明创建线程失败，常见的错误返回代码为 EAGAIN 和 EINVAL。前者表示系统限制创建新的线程，例如线程数目过多了；后者表示第二个参数代表的线程属性值非法。创建线程成功后，新创建的线程则运行参数三和参数四确定的函数，原来的线程则继续运行下一行代码。

pthread_join 函数用来等待一个线程的结束。函数原型为：

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

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

pthread_exit 函数

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

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

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

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

pthread_cond_init 函数

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

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

其中 cond 是一个指向结构 pthread_cond_t 的指针，cond_attr 是一个指向结构 pthread_condattr_t 的指针。结构 pthread_condattr_t 是条件变量的属性结构，和互斥锁一样我们可以用它来设置条件变量是进程内可用还是进程间可用，默认值是 PTHREAD_ PROCESS_PRIVATE，即此条件变量被同一进程内的各个线程使用。注意初始化条件变量只有未被使用时才能重新初始化或被释放。释放一个条件变量的函数为 pthread_cond_ destroy（pthread_cond_t cond）。

pthread_cond_wait 函数　使线程阻塞在一个条件变量上。它的函数原型为：

```
extern int pthread_cond_wait (pthread_cond_t *__restrict__cond,　pthread_mutex_t *__restrict __mutex)
```

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

pthread_cond_timedwait 函数

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

```
extern int pthread_cond_timedwait __P ((pthread_cond_t *__cond,
                                        pthread_mutex_t *__mutex, __const struct timespec *__abstime))
```

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

pthread_cond_signal 函数　它的函数原型为：

```
extern int pthread_cond_signal (pthread_cond_t *__cond)
```

它用来释放被阻塞在条件变量 cond 上的一个线程。多个线程阻塞在此条件变量上时，哪一个线程被唤醒是由线程的调度策略所决定的。要注意的是，必须用保护条件变量的互斥

锁来保护这个函数，否则条件满足信号又可能在测试条件和调用 pthread_cond_wait 函数之间被发出，从而造成无限制的等待。

5. 实验步骤

♦ **编译源程序**

1）进入实验目录：

```
cbt@Cyb-Bot:~$ cd /CBT-6818/SRC/exp/basic/02_pthread/
cbt@Cyb-Bot: /CBT-6818/SRC/exp/basic/02_pthread$ ls
Makefile    pthread    pthread.c    pthread.o
```

2）清除中间代码，重新编译

```
cbt@Cyb-Bot: /CBT-6818/SRC/exp/basic/02_pthread/$ make clean
rm -f ../bin/pthread ./pthread *.elf *.gdb *.o
cbt@Cyb-Bot: /CBT-6818/SRC/exp/basic/02_pthread/$ make
arm-linux-gcc        -c -o pthread.o pthread.c
arm-linux-gcc -static      -o ../bin/pthread pthread.o      -lpthread
arm-linux-gcc -static      -o pthread pthread.o      -lpthread
cbt@Cyb-Bot: /CBT-6818/SRC/exp/basic/02_pthread/$ ls
Makefile    pthread    pthread.c    pthread.o
cbt@Cyb-Bot: /CBT-6818/SRC/exp/basic/02_pthread/$
```

当前目录下生成可执行程序 pthread。

♦ **运行程序（NFS 方式）**

1）启动 CBT-6818 型实验系统，连好网线、串口线。通过串口终端挂载宿主机实验目录。

```
[root@CBT-6818:~]# mount -t nfs -o nolock 192.168.1.7:/CBT-6818      /mnt/
```

2）进入串口终端的 NFS 共享实验目录。

```
[root@CBT-6818:~]# cd    /mnt /SRC/exp/basic/02_pthread/
[root@CBT-6818: /mnt /SRC/exp/basic/02_pthread]# ls
Makefile      pthread       pthread.c    pthread.o
```

3）执行程序。

```
[root@CBT-6818: /mnt /SRC/exp/basic/02_pthread]# ./pthread
```

♦ **实验结果**

```
[root@ CBT-6818 /mnt//SRC/exp/basic/02_pthread/$ ./pthread
  put-->0
  put-->1
```

```
  put-->2
  put-->3
  put-->4
  put-->5
  ......
wait for not full
                        0-->get
                        1-->get
                        2-->get
                        3-->get
                        4-->get
                        5-->get
                          ......
wait for not empty
                        15-->get
......
wait for not empty
  put-->20
                        20-->get
……
……
```

实验三. 串口程序设计

1. 实验目的

- 了解在 linux 环境下串口程序设计的基本方法，掌握终端的主要属性及设置方法。
- 学习使用多线程来完成串口的收发处理。

2. 实验环境

- 硬件：A53 智能网关实验平台，PC 机 Pentium 500 以上，硬盘 40G 以上，内存大于 256M。
- 软件：Vmware Workstation +ubuntu + MiniCom/超级终端+ ARM-LINUX 交叉编译开发环境。
- 实验目录：/CBT-6818/SRC/exp/basic/03_tty。

3. 实验内容

- 编写应用程序实现对 ARM 设备串口的读和写。
- 学习将多线程编程应用到串口的接收和发送程序设计中。

4. 实验原理

4.1 硬件接口原理

串行口是计算机一种常用的接口，具有连接线少，通讯简单，得到广泛的使用。常用的串口是 RS- 232-C 接口(又称 EIA RS-232-C)它是在 1970 年由美国电子工业协会(EIA)联合贝尔系统、调制解调器厂家及计算机终端生产厂家共同制定的用于串行通讯的标准。串口通讯指的是计算机依次以位（bit）为单位来传送数据，串行通讯使用的范围很广，在嵌入式系统开发过程中串口通讯也经常用到通讯方式之一。

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

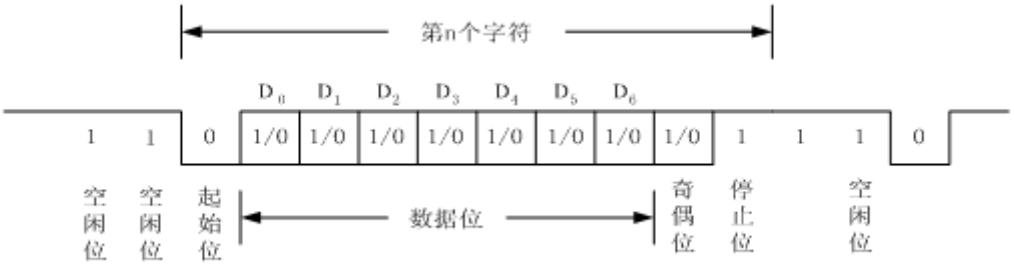


图 4.1.1 串行通信字符格式

图 4.1.1 给出异步串行通信中一个字符的传送格式。开始前，线路处于空闲状态，送出连续“1”。传送开始时首先发一个“ 0 ”作为起始位，然后出现在通信线上的是字符的二进制编码数据。每个字符的数据位长可以约定为 5 位、6 位、7 位或 8 位，一般采用 ASCII 编码。后面是奇偶校验位，根据约定，用奇偶校验位将所传字符中为“1”的位数凑成奇数个或偶数个。也可以约定不要奇偶校验，这样就取消奇偶校验位。最后是表示停止位的“1”信号，这个停止位可以约定持续 1 位、1.5 位或 2 位的时间宽度。至此一个字符传送完毕，线路又进入空闲，持续为“1”。经过一段随机的时间后，下一个字符开始传送才又发出起始位。每一个数据位的宽度等于传送波特率的倒数。微机异步串行通信中，常用的波特率为 50，95，110，150，300，600，1200，2400，4800，9600 等。

接收方按约定的格式接收数据，并进行检查，可以查出以下三种错误：

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

每一种错误都会给出相应的出错信息，提示用户处理。一般串口调试都使用空的 MODEM 连接电缆，其连接方式如下图 4.1.2：

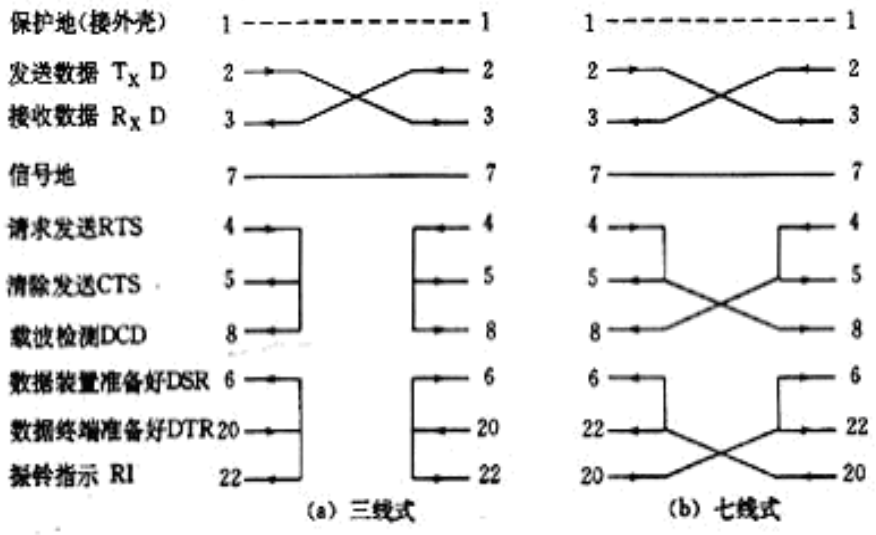


图 4.1.2 实用 RS-232C 通讯连线

4.2 软件接口介绍

Linux 对所有设备的访问是通过设备文件来进行的，串口也是这样，为了访问串口，只需打开其设备文件即可操作串口设备。在 linux 系统下面，每一个串口设备都有设备文件与其关联，设备文件位于系统的/dev 目录下面。如 linux 下的/dev/ttySAC0，/dev/ttySAC1 分别表示的是串口 0 和串口 1。

S5P6818 处理器自带 4 个串行端口控制器，用户可以参考该处理器的 datasheet 进行分析。

♦ Termios 结构体

Linux 系统下，用户应用程序很容易对串行端口设备进行属性设置，这些属性定义在结构体 struct termios 中。为在程序中使用该结构体，需要包含文件<termios.h>，该头文件定义了结构体 struct termios。

termios 函数族提供了一个常规的终端接口，用于控制非同步通信端口。这个结构包含了至少下列成员：

```
struct termio {
        unsigned short c_iflag;         /* input mode flags */
        unsigned short c_oflag;         /* output mode flags */
        unsigned short c_cflag;         /* control mode flags */
        unsigned short c_lflag;         /* local mode flags */
        unsigned char c_line;           /* line discipline */
        unsigned char c_cc[NCC];        /* control characters */
};
```

c_iflag：输入模式标志，控制终端输入方式。

键 值	说 明
IGNBRK	忽略BREAK键输入
BRKINT	如果设置了IGNBRK，BREAK键的输入将被忽略，如果设置了BRKINT ，将产生SIGINT中断
IGNPAR	忽略奇偶校验错误
PARMRK	标识奇偶校验错误
INPCK	允许输入奇偶校验
ISTRIP	去除字符的第8个比特
INLCR	将输入的NL（换行）转换成CR（回车）
IGNCR	忽略输入的回车
ICRNL	将输入的回车转化成换行（如果IGNCR未设置的情况下）
IUCLC	将输入的大写字符转换成小写字符（非POSIX）
IXON	允许输入时对XON/XOFF流进行控制
IXANY	输入任何字符将重启停止的输出
IXOFF	允许输入时对XON/XOFF流进行控制
IMAXBEL	当输入队列满的时候开始响铃，Linux在使用该参数而是认为该参数总是已经设置

表 4.2.1

c_oflag：输出模式标志，控制终端输出方式

键　值	说　明
OPOST	处理后输出
OLCUC	将输入的小写字符转换成大写字符（非POSIX）
ONLCR	将输入的NL（换行）转换成CR（回车）及NL（换行）
OCRNL	将输入的CR（回车）转换成NL（换行）
ONOCR	第一行不输出回车符
ONLRET	不输出回车符
OFILL	发送填充字符以延迟终端输出
OFDEL	以ASCII码的DEL作为填充字符，如果未设置该参数，填充字符将是NUL（'\0'）（非POSIX）
NLDLY	换行输出延时，可以取NL0（不延迟）或NL1（延迟0.1s）
CRDLY	回车延迟，取值范围为：CR0、CR1、CR2和 CR3
TABDLY	水平制表符输出延迟，取值范围为：TAB0、TAB1、TAB2和TAB3
BSDLY	空格输出延迟，可以取BS0或BS1
VTDLY	垂直制表符输出延迟，可以取VT0或VT1
FFDLY	换页延迟，可以取FF0或FF1

表 4.2.2

c_cflag：控制模式标志，指定终端硬件控制信息

键　值	说　明
CBAUD	波特率（4+1位）（非POSIX）
CBAUDEX	附加波特率（1位）（非POSIX）
CSIZE	字符长度，取值范围为CS5、CS6、CS7或CS8
CSTOPB	设置两个停止位
CREAD	使用接收器
PARENB	使用奇偶校验
PARODD	对输入使用奇偶校验，对输出使用偶校验
HUPCL	关闭设备时挂起
CLOCAL	忽略调制解调器线路状态
CRTSCTS	使用RTS/CTS流控制

表 4.2.3

c_lflag：本地模式标志，控制终端编辑功能

键　值	说　明
ISIG	当输入INTR、QUIT、SUSP或DSUSP时，产生相应的信号
ICANON	使用标准输入模式
XCASE	在ICANON和XCASE同时设置的情况下，终端只使用大写。如果只设置了XCASE，则输入字符将被转换为小写字符，除非字符使用了转义字符（非POSIX，且Linux不支持该参数）
ECHO	显示输入字符
ECHOE	如果ICANON同时设置，ERASE将删除输入的字符，WERASE将删除输入的单词
ECHOK	如果ICANON同时设置，KILL将删除当前行
ECHONL	如果ICANON同时设置，即使ECHO没有设置依然显示换行符
ECHOPRT	如果ECHO和ICANON同时设置，将删除打印出的字符（非POSIX）
TOSTOP	向后台输出发送SIGTTOU信号

表 4.2.4

c_cc[NCCS]：控制字符，用于保存终端驱动程序中的特殊字符，如输入结束符等

宏	说　明	宏	说　明
VINTR	Interrupt字符	VEOL	附加的End-of-file字符
VQUIT	Quit字符	VTIME	非规范模式读取时的超时时间
VERASE	Erase字符	VSTOP	Stop字符
VKILL	Kill字符	VSTART	Start字符
VEOF	End-of-file字符	VSUSP	Suspend字符
VMIN	非规范模式读取时的最小字符数		

表 4.2.5

为了便于通过程序来获得和修改终端参数，Linux 还提供了 tcgetattr 函数和 tcsetattr 函数。tcgetattr 用于获取终端的相关参数，而 tcsetattr 函数用于设置终端参数。

```
int tcgetattr(int fd, struct termios *termios_p);
```

该函数用来获取终端控制属性，它把串口的默认设置赋给了 termios 数据数据结构，其参数说明如下：

fd：待操作的文件描述符

termios_p：指向 termios 结构的指针

函数返回值：成功返回 0，失败返回－1。

```
int tcsetattr(int fd, int optional_actions, const struct termios *termios_p);
```

该函数用来设置终端控制属性，其参数说明如下：

fd：待操作的文件描述符

optional_actions：选项值，有三个选项以供选择：

TCSANOW：不等数据传输完毕就立即改变属性；

TCSADRAIN：等待所有数据传输结束才改变属性；

TCSAFLUSH：清空输入输出缓冲区才改变属性；

termios_p：指向 termios 结构的指针；

函数返回值：成功返回 0，失败返回－1。

4.3 软件架构及流程

Linux 操作系统从一开始就对串行口提供了很好的支持，为进行串行通讯提供了大量的函数，我们的实验主要是为掌握在 Linux 中进行串行通讯编程的基本方法。本实验的程序流程图如图 4.3 所示：

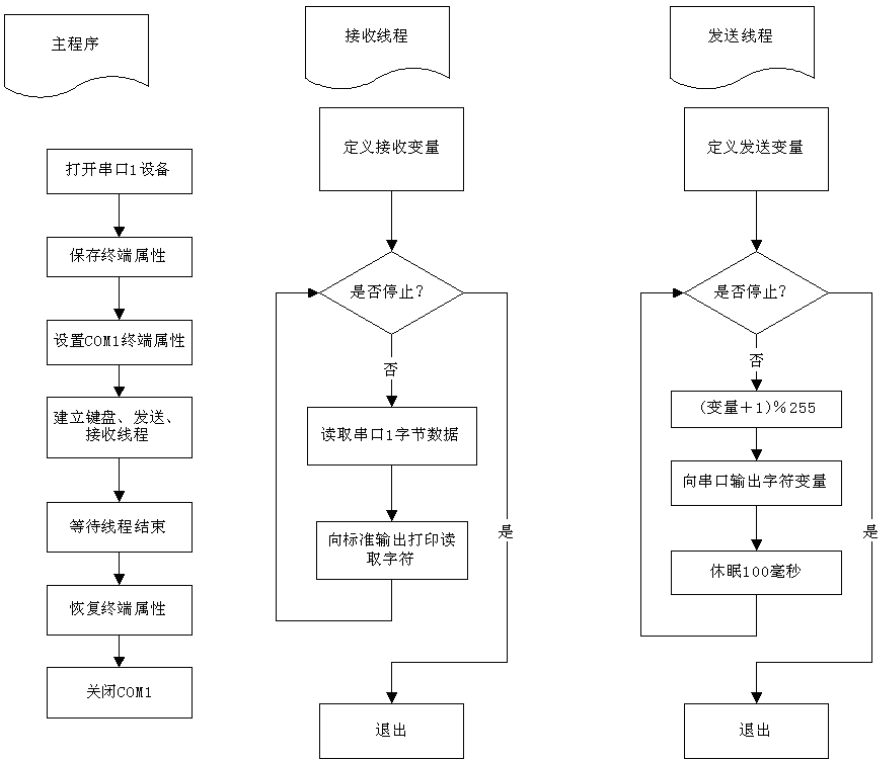


图 4.3 串口通讯实验流程图

4.1 键代码分析

本实验的代码 term.c 如下：

```
#include <termios.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/signal.h>
#include <pthread.h>

#define BAUDRATE B115200
#define COM1 "/dev/ttySAC0"                    /* 串口 0 控制台 */
#define COM2 "/dev/ttySAC1"                    /* 串口 1 */
```

```
#define ENDMINITERM1 27 /* ESC to quit miniterm */
#define ENDMINITERM2 3    /*ctl +c    to quit miniterm */
#define FALSE 0
#define TRUE 1

volatile int STOP=FALSE;
volatile int fd;
/*   终端中断信号处理函数   */
void child_handler(int s)
{
        printf("stop!!!\n");
        STOP=TRUE;
}

/*   PC 机键盘线程处理函数   */
void* keyboard(void * data)
{
        int c;
        for (;;){
                c=getchar();
        //    printf("getchar is :%d",c);
              /*     如果输入键值为回车或 CTRL+C  则置停止位  退出    */
                if( (c== ENDMINITERM1) | (c==ENDMINITERM2)){
                        STOP=TRUE;
                        break ;
                }
        }

        return NULL;
}
/*-------------------------------------------------------*/
/* modem input handler */
/*  读取终端字符处理线程函数   */
void* receive(void * data)
{
        int c;
        printf("read modem\n");

        while (STOP==FALSE)
        {
                read(fd,&c,1); /* com port */             /*   读取 1 个字符   */
                write(1,&c,1); /* stdout */              /*    打印到标准输出设备   */
        }

        printf("exit from reading modem\n");
        return NULL;
```

```
}
/*-------------------------------------------------------*/
/*  写入终端字符处理线程函数    */
void* send(void * data)
{
        int c='0';
        printf("send data\n");

        while (STOP==FALSE)
        {
                c++;
                c %= 255;
                write(fd,&c,1); /* stdout */     /*  将转换后字符写入串口终端      */
                usleep(100000);
        }

        return NULL; /* wait for child to die or it will become a zombie */
}
/*-------------------------------------------------------*/
int main(int argc,char** argv)
{

        struct termios oldtio,newtio;     /*  终端设置 termios  成员  */
        struct sigaction sa;                /*      定义信号      */
        int ok;
        pthread_t th_a, th_b, th_c;
        void * retval;
 /*    根据命令行参数个数  打开不同串口终端  默认打开串口 0 */
        if( argc > 1)
        fd = open(COM2, O_RDWR );
        else
                fd = open(COM1, O_RDWR ); //| O_NOCTTY |O_NONBLOCK);

        if (fd <0) {
                perror(COM1);
                exit(-1);
        }
        printf("\nOpen COM Port Successfull\n");
        /*    保存串口终端原有设置  */
        tcgetattr(fd,&oldtio); /* save current modem settings */
        tcflush(fd,TCIOFLUSH);
/*    设置波特率  */
        cfsetispeed(&newtio,BAUDRATE);/*设置串口输??/
        cfsetospeed(&newtio,BAUDRATE);
         /*    设置 8 位数据位  */
        newtio.c_cflag &= ~CSIZE;
```

```
        newtio.c_cflag |= CS8;
        /*   设置奇偶校验位 无 */
        newtio.c_cflag &= ~PARENB; /*set the PARENB bit to zero-------disable parity checked*/
        newtio.c_iflag &= ~INPCK;   /*set the INPCK bit to zero--------INPCK means inparitycheck(not
paritychecked)*/
        /*   设置停止位 1 */
        newtio.c_cflag &= ~CSTOPB;
        newtio.c_cc[VMIN]=1;
        newtio.c_cc[VTIME]=0;

 /* now clean the modem line and activate the settings for modem */

        tcflush(fd, TCIFLUSH);
        /*   设置 串口终端立即生效     */
        if(tcsetattr(fd,TCSANOW,&newtio) != 0)        {
                perror("\n");
                return 0;
        }
        /*   设置信号处理句柄   */
        sa.sa_handler = child_handler;
        sa.sa_flags = 0;
        sigaction(SIGCHLD,&sa,NULL); /* handle dying child */
        /*   建立串口终端处理线程 */
        pthread_create(&th_a, NULL, keyboard, 0);
        pthread_create(&th_b, NULL, receive, 0);
        pthread_create(&th_c, NULL, send, 0);

        pthread_join(th_a, &retval);
        pthread_join(th_b, &retval);
        pthread_join(th_c, &retval);
        /* 还原串口终端原有设置 */
        tcsetattr(fd,TCSANOW,&oldtio); /* restore old modem setings */
        close(fd);

        exit(0);
}
```

下面我们对这个程序的主要部分做一下简单的分析

- 头文件

```
#include      <stdio.h>       /*标准输入输出定义*/
#include      <stdlib.h>      /*标准函数库定义*/
#include      <unistd.h>      /*linux 标准函数定义*/
#include      <sys/types.h>
#include      <sys/stat.h>
#include      <fcntl.h>       /*文件控制定义*/
```

```
#include      <termios.h>      /*PPSIX 终端控制定义*/
#include      <errno.h>        /*错误号定义*/
#include     <pthread.h>  /*线程库定义*/
```

♦ **打开串口**

在 Linux 下串口文件位于/dev 下，一般在老版本的内核中串口一为/dev/ttyS0 ，串口二为 /dev/ttyS1， 在我们的开发板中串口设备位于/dev/下，串口 0 为/dev/ttySAC0,串口 1 为/dev/ttySAC1。打开串口是通过标准的文件打开函数来实现的:

```
int fd;
fd = open( "/dev/ttySAC0", O_RDWR);     /*以读写方式打开串口*/
if (-1 == fd){                                  /*  不能打开串口一*/
      perror("  提示错误！ ");
}
```

♦ **串口设置**

最基本的设置串口包括波特率设置，效验位和停止位设置。串口的设置主要是设置 struct termios 结构体的各成员值，关于该结构体的定义可以查看内核源码的 include/asm/termios.h 文件。

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

设置这个结构体很复杂，可以参考 man 手册或者由赵克佳、沈志宇编写的《UNIX 程序编写教程》，我这里就只考虑常见的一些设置：

♦ **波特率设置：**

下面是修改波特率的代码：

```
struct    termios Opt;
tcgetattr(fd, &Opt);
cfsetispeed(&Opt,B115200);                  /*设置为 115200Bps*/
cfsetospeed(&Opt,B1152200);
tcsetattr(fd,TCANOW,&Opt);
```

♦ **校验位和停止位的设置：**

无效验 8 位

```
Option.c_cflag &= ~PARENB;
Option.c_cflag &= ~CSTOPB;
Option.c_cflag &= ~CSIZE;
Option.c_cflag |= ~CS8;
```

奇效验(Odd) 7 位

```
Option.c_cflag |= ~PARENB;
Option.c_cflag &= ~PARODD;
Option.c_cflag &= ~CSTOPB;
Option.c_cflag &= ~CSIZE;
Option.c_cflag |= ~CS7;
```

偶效验(Even) 7 位

```
Option.c_cflag &= ~PARENB;
Option.c_cflag |= ~PARODD;
Option.c_cflag &= ~CSTOPB;
Option.c_cflag &= ~CSIZE;
Option.c_cflag |= ~CS7;
```

Space 效验 7 位

```
Option.c_cflag &= ~PARENB;
Option.c_cflag &= ~CSTOPB;
Option.c_cflag &= &~CSIZE;
Option.c_cflag |= CS8;
```

♦ **设置停止位：**

1 位：

```
options.c_cflag &= ~CSTOPB;
```

2 位:

```
options.c_cflag |= CSTOPB;
```

注意：如果不是开发终端之类的，只是串口传输数据，而不需要串口来处理，那么使用原始模式(Raw Mode)方式来通讯，本实验没有使用原始模式。其设置方式如下：

```
options.c_lflag    &= ~(ICANON | ECHO | ECHOE | ISIG);        /*Input*/
options.c_oflag    &= ~OPOST;                                 /*Output*/
```

♦ **读写串口**

设置好串口之后，读写串口就很容易了，把串口当作文件读写就可以了。

发送数据：

```
char    buffer[1024];
int      Length＝1024;
int      nByte;
nByte = write(fd, buffer ,Length)
```

读取串口数据：

使用文件操作 read 函数读取，如果设置为原始模式(Raw Mode)传输数据，那么 read 函数返回的字符数是实际串口收到的字符数。可以使用操作文件的函数来实现异步读取，如 fcntl，或者 select 等来操作。

```
char    buff[1024];
int      Len＝1024;
int    readByte = read(fd, buff, Len);
```

♦ **关闭串口**

关闭串口就是关闭文件。

```
close(fd);
```

5. 实验步骤

♦ **编译源程序**

1）进入实验目录：

```
cbt@Cyb-Bot:~$ cd /CBT-6818/SRC/exp/basic/03_tty/
cbt@Cyb-Bot: /CBT-6818/SRC/exp/basic/03_tty/$ ls
Makefile    term    term.c    term.o
cbt@Cyb-Bot: /CBT-6818/SRC/exp/basic/03_tty/$
```

2）清除中间代码，重新编译

```
cbt@Cyb-Bot: /CBT-6818/SRC/exp/basic/03_tty/$ make clean
rm -f ../bin/term    ./term *.elf *.elf2flt *.gdb *.o
cbt@Cyb-Bot: /CBT-6818/SRC/exp/basic/03_tty/$ make
arm-linux-gcc        -c -o term.o term.c
arm-linux-gcc -static      -o ../bin/term term.o      -lpthread
arm-linux-gcc -static      -o term term.o      -lpthread
cbt@Cyb-Bot: /CBT-6818/SRC/exp/basic/03_tty/$ ls
Makefile    term    term.c    term.o
cbt@Cyb-Bot: /CBT-6818/SRC/exp/basic/03_tty/$
```

3）当前目录下生成可执行程序 term。

♦ **运行程序（NFS 方式）**

1）启动 CBT-6818 型实验系统，连好网线、串口线。通过串口终端挂载宿主机实验目录。

```
[root@CBT-6818:~]# mount -t nfs -o nolock 192.168.1.7:/CBT-6818      /mnt/
```

2）进入串口终端的 NFS 共享实验目录。

```
[root@CBT-6818:~]# cd /mnt/SRC/exp/basic/03_tty/
[root@CBT-6818: /mnt/SRC/exp/basic/03_tty/]# ls
Makefile    term          term.c       term.o
[root@CBT-6818: /mnt/SRC/exp/basic/03_tty/]#
```

3）执行程序。

```
[root@CBT-6818: /mnt/SRC/exp/basic/03_tty/]# ./term
```

♦ **实验结果**

```
[root@CBT-6818: /mnt/SRC/exp/basic/03_tty/]#./term

Open COM Port Successfull
read modem
              send data

123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~□€ 亗
儀庠噲埘媽崕彁憭摂晼椋櫄涜溜熨、￥ウ Ж¦          辈炒刀犯购患骄坷谅媚牌侨墒颂臀闲岩釉罩棕仝
圮蒉哙徙滏玳觎盹痱篝貊鼬


123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~□€ 亗
儀庠噲埘媽崕彁憭摂晼椋櫄涜溜熨、￥ウ Ж¦          辈炒刀犯购患骄坷谅媚牌侨墒颂臀闲岩釉罩棕仝
圮蒉哙徙滏玳觎盹痱篝貊鼬

exit from reading modem

[root@CBT-6818 /mnt//SRC/exp/basic/03_tty/$
```

按住 PC 机键盘的 ESC 或者 CTRL+C，再按回车键就可以终止程序运行。

实验四. SOCKET 网络编程

1. 实验目的

- 学习在 linux 环境下使用 SOCKET 进行网络程序设计的基本方法和接口。
- 掌握基于 linux 系统下 SOCKET 套接字的进程间通讯方法。

2. 实验环境

- 硬件：A53 智能网关实验平台，PC 机 Pentium 500 以上，硬盘 40G 以上，内存大于 256M
- 软件：Vmware Workstation +ubuntu + MiniCom/超级终端 +ARM-LINUX 交叉编译开发环境
- 实验目录：/CBT-6818/SRC/exp/basic/04_socket

3. 实验内容

- 编写 socket 套接字程序，完成服务器端与客户端程序的通讯。

4. 实验原理

4.1 SOCKET 套接字

Socket 接口是 TCP/IP 网络的 API，Socket 接口定义了许多函数或例程，程序员可以用它们来开发 TCP/IP 网络上的应用程序。要学 Internet 上的 TCP/IP 网络编程，必须理解 Socket 接口。

Socket 接口设计者最先是将接口放在 Unix 操作系统里面的。如果了解 Unix 系统的输入和输出的话，就很容易了解 Socket 了。网络的 Socket 数据传输是一种特殊的 I/O，Socket 也是一种文件描述符。Socket 也具有一个类似于打开文件的函数调用 Socket()，该函数返 回一个整型的 Socket 描述符，随后的连接建立、数据传输等操作都是通过该 Socket 实现的。常用的 Socket 类型有两种：流式 Socket （SOCK_STREAM）和数据报式 Socket （SOCK_DGRAM）。流式是一种面向连接的 Socket，针对于面向连接的 TCP 服务应用；数据 报式 Socket 是一种无连接的 Socket，对应于无连接的 UDP 服务应用。

♦ **端口**

按照 OSI 七层协议的描述，传输层与网络层在功能上的最大区别是传输层提供进程通信能力。从这个意义上讲，网络通信的最终地址就不仅仅是主机地址了，还包括可以描述进程的某种标识符。为此，TCP/IP 协议提出了协议端口（protocol port，简称端口）的概念，用于标识通信的进程。

端口是一种抽象的软件结构（包括一些数据结构和 I/O 缓冲区）。应用程序（即进程）通过系统调用与某端口建立连接（binding）后，传输层传给该端口的数据都被相应进程所接收，相应进程发给传输层的数据都通过该端口输出。在 TCP/IP 协议的实现中，端口操作类似于一般的 I/O 操作，进程获取一个端口，相当于获取本地唯一的 I/O 文件，可以用一般的读写原语访问之。

类似于文件描述符，每个端口都拥有一个叫端口号（port number）的整数型标识符，用于区别不同端口。

♦ 地址

网络通信中通信的两个进程分别在不同的机器上。在互连网络中，两台机器可能位于不同的网络，这些网络通过网络互连设备（网关，网桥，路由器等）连接。因此需要三级寻址：

1）某一主机可与多个网络相连，必须指定一特定网络地址；

2）网络上每一台主机应有其唯一的地址；

3）每一主机上的每一进程应有在该主机上的唯一标识符。

通常主机地址由网络 ID 和主机 ID 组成，在 TCP/IP 协议中用 32 位整数值表示；TCP 和 UDP 均使用 16 位端口号标识用户进程。

♦ 网络字节序

不同的计算机存放多字节值的顺序不同，有的机器在起始地址存放低位字节（低价先存），有的存高位字节（高价先存）。为保证数据的正确性，在网络协议中须指定网络字节顺序。TCP/IP 协议使用 16 位整数和 32 位整数的高价先存格式，它们均含在协议头文件中。

♦ 客户机/服务器模式

在 TCP/IP 网络应用中，通信的两个进程间相互作用的主要模式是客户/服务器模式（Client/Server model），即客户向服务器发出服务请求，服务器接收到请求后，提供相应的服务。客户/服务器模式的建立基于以下两点：首先，建立网络的起因是网络中软硬件资源、运算能力和信息不均等，需要共享，从而造就拥有众多资源的主机提供服务，资源较少的客户请求服务这一非对等作用。其次，网间进程通信完全是异步的，相互通信的进程间既不存在父子关系，又不共享内存缓冲区，因此需要一种机制为希望通信的进程间建立联系，为二者的数据交换提供同步，这就是基本的客户端/服务器模式的 TCP/IP。

客户/服务器模式通信过程中采取的是主动请求方式，首先服务器方要先启动，并根据请求提供相应服务：

1）打开一通信通道并告知本地主机，它愿意在某一公认地址上（如 FTP 为 21）接收客户请求；

2）等待客户请求到达该端口；

3）接收到重复服务请求，处理该请求并发送应答信号。接收到并发服务请求，要激活一新进程来处理这个客户请求（如 UNIX 系统中用 fork、exec）。新进程处理此客户请求，

并不需要对其它请求作出应答。服务完成后，关闭此新进程与客户的通信链路，并终止。

4）返回第二步，等待另一客户请求。

5）关闭服务器

客户方：

1）打开一通信通道，并连接到服务器所在主机的特定端口；

2）向服务器发服务请求报文，等待并接收应答；继续提出请求.....

3）请求结束后关闭通信通道并终止。

从上面所描述过程可知：

1）客户与服务器进程的作用是非对称的，因此编码不同。

2）服务进程一般是先于请求而启动的。只要系统运行，该服务进程一直存在，直到正常或强迫终止。

4.2 软件接口介绍

- **创建套接字 socket()**

应用程序在使用套接字前，首先必须拥有一个套接字，系统调用 socket()向应用程序提供创建套接字的手段，其调用格式如下：

SOCKET PASCAL FAR socket(int af, int type, int protocol);

该调用要接收三个参数：af、type、protocol。参数 af 指定通信发生的区域，UNIX 系统支持的地址族有：AF_UNIX、AF_INET、 AF_NS 等，而 DOS、WINDOWS 中仅支持 AF_INET，它是网际网区域。因此，地址族与协议族相同。参数 type 描述要建立的套接字的类型。参数 protocol 说明该套接字使用的特定协议，如果调用者不希望特别指定使用的协议，则置为 0，使用默认的连接模式。根据这三个参数建立一个套接字，并将相应的资源分配给它，同时返回一个整型套接字号。因此，socket()系统调用实际上指定了相关五元组中的“协议”这一元。

- **绑定本地地址 bind()**

当一个套接字用 socket()创建后，存在一个名字空间(地址族),但它没有被命名。bind()将套接字地址（包括本地主机地址和本地端口地址）与所创建的套接字号联系起来，即将名字赋予套接字，以指定本地半相关。其调用格式如下：

int PASCAL FAR bind(SOCKET s, const struct sockaddr FAR * name, int namelen);

参数 s 是由 socket()调用返回的并且未作连接的套接字描述符(套接字号)。参数 name 是赋给套接字 s 的本地地址（名字），其长度可变，结构随通信域的不同而不同。namelen 表明了 name 的长度。如果没有错误发生，bind()返回 0。否则返回值 SOCKET_ERROR。

地址在建立套接字通信过程中起着重要作用，作为一个网络应用程序设计者对套接字地址结构必须有明确认识。

♦ **建立套接字链接 connect()与 accept()**

这两个系统调用用于完成一个完整相关的建立，其中 connect()用于建立连接。无连接的套接字进程也可以调用 connect()，但这时在进程之间没有实际的报文交换，调用将从本地操作系统直接返回。这样做的优点是程序员不必为每一数据指定目的地址，而且如果收到的一个数据报，其目的端口未与任何套接字建立“连接”。而 accept()用于使服务器等待来自某客户进程的实际连接。

connect()的调用格式如下：

int PASCAL FAR connect(SOCKET s, const struct sockaddr FAR * name, int namelen);

参数 s 是欲建立连接的本地套接字描述符。参数 name 指出说明对方套接字地址结构的指针。对方套接字地址长度由 namelen 说明。如果没有错误发生，connect()返回 0。否则返回值 SOCKET_ERROR。在面向连接的协议中，该调用导致本地系统和外部系统之间连接实际建立。

由于地址族总被包含在套接字地址结构的前两个字节中，并通过 socket()调用与某个协议族相关。因此 bind()和 connect()无须协议作为参数。

accept()的调用格式如下：

SOCKET PASCAL FAR accept(SOCKET s, struct sockaddr FAR* addr, int FAR* addrlen);

参数 s 为本地套接字描述符，在用做 accept()调用的参数前应该先调用过 listen()。addr 指向客户方套接字地址结构的指针，用来接收连接实体的地址。addr 的确切格式由套接字创建时建立的地址族决定。addrlen 为客户方套接字地址的长度（字节数）。如果没有错误发生，accept()返回一个 SOCKET 类型的值，表示接收到的套接字的描述符。否则返回值 INVALID_SOCKET。

accept()用于面向连接服务器。参数 addr 和 addrlen 存放客户方的地址信息。调用前，参数 addr 指向一个初始值为空的地址结构，而 addrlen 的初始值为 0；调用 accept()后，服务器等待从编号为 s 的套接字上接受客户连接请求，而连接请求是由客户方的 connect()调用发出的。当有连接请求到达时，accept()调用将请求连接队列上的第一个客户方套接字地址及长度放入 addr 和 addrlen，并创建一个与 s 有相同特性的新套接字号。新的套接字可用于处理服务器并发请求。

♦ **监听链接 listen()**

此调用用于面向连接服务器，表明它愿意接收连接。listen()需在 accept()之前调用，其调用格式如下：

int PASCAL FAR listen(SOCKET s, int backlog);

参数 s 标识一个本地已建立、尚未连接的套接字号，服务器愿意从它上面接收请求。backlog 表示请求连接队列的最大长度，用于限制排队请求的个数，目前允许的最大值为 5。如果没有错误发生，listen()返回 0。否则它返回 SOCKET_ERROR。

listen()在执行调用过程中可为没有调用过 bind()的套接字 s 完成所必须的连接，并建立长度为 backlog 的请求连接队列。

♦ **数据传输 send()与 recv()**

当一个连接建立以后，就可以传输数据了。常用的系统调用有 send()和 recv()。

send()调用用于向指定的已连接的数据报或流套接字上发送输出数据，格式如下：

int PASCAL FAR send(SOCKET s, const char FAR *buf, int len, int flags);

参数 s 为已连接的本地套接字描述符。buf 指向存有发送数据的缓冲区的指针，其长度由 len 指定。flags 指定传输控制方式，如是否发送带外数据等。如果没有错误发生，send()返回总共发送的字节数。否则它返回 SOCKET_ERROR。

recv()调用用于向指定的已连接的数据报或流套接字上接收输入数据，格式如下：

int PASCAL FAR recv(SOCKET s, char FAR *buf, int len, int flags);

参数 s 为已连接的套接字描述符。buf 指向接收输入数据缓冲区的指针，其长度由 len 指定。flags 指定传输控制方式，如是否接收带外数据等。如果没有错误发生，recv()返回总共接收的字节数。如果连接被关闭，返回 0。否则它返回 SOCKET_ERROR。

♦ **关闭套接字**

closesocket()关闭套接字 s，并释放分配给该套接字的资源；如果 s 涉及一个打开的 TCP 连接，则该连接被释放。closesocket()的调用格式如下：

BOOL PASCAL FAR closesocket(SOCKET s);

参数 s 待关闭的套接字描述符。如果没有错误发生，closesocket()返回 0。否则返回值 SOCKET_ERROR。

4.3 软件架构及流程

程序首先由服务器端，建立 Socket 连接，并绑定本地地监听服务请求。一旦有客户端发出连接请求则进行请求处理，最终完成数据服务。

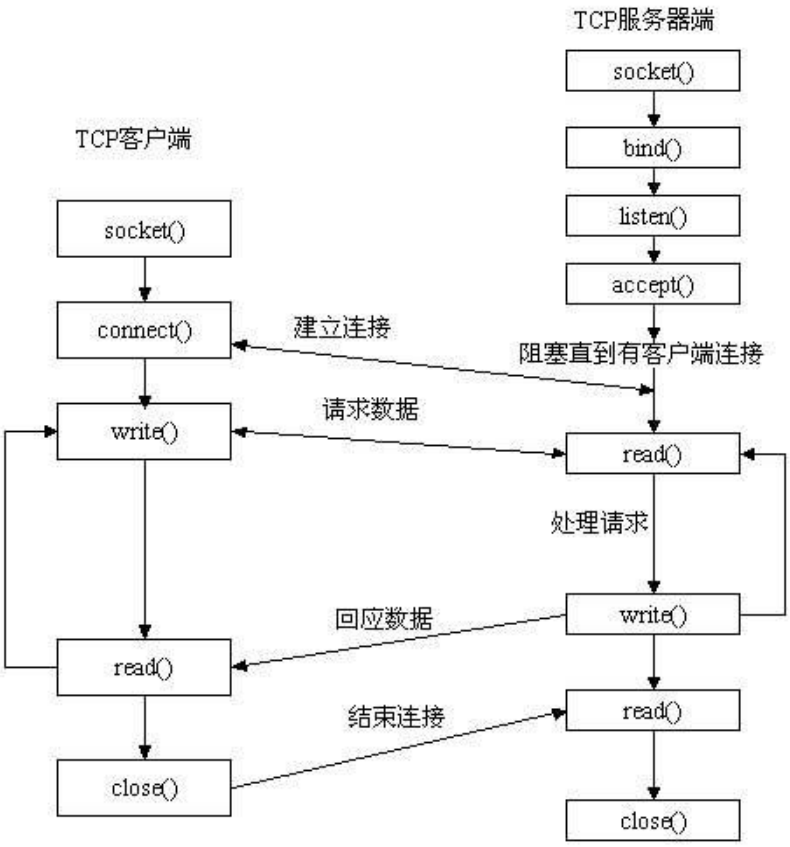


图 4.3 SOCKET 通讯 C/S 模型

4.4 关键代码分析

本实验的服务器端代码 skt_server.c 如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/wait.h>
#define SERVPORT 3333 /*服务器监听端口号 */
#define BACKLOG 10 /* 最大同时连接请求数 */

int main()
{
int sockfd,client_fd,sin_size; /*sock_fd：监听 socket；client_fd：数据传输 socket */
struct sockaddr_in my_addr; /* 本机地址信息 */
struct sockaddr_in remote_addr; /* 客户端地址信息 */

if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) //建立 socket
   {
```

```
    perror("socket");
    exit(1);
    }

my_addr.sin_family=AF_INET;
my_addr.sin_port=htons(SERVPORT);
my_addr.sin_addr.s_addr = INADDR_ANY; //表示监听任何地址
bzero(&(my_addr.sin_zero),8);

if (bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr)) == -1) //将本机地址与建立的套接字
号进行绑定
    {
    perror("bind");
    exit(1);
    }

if (listen(sockfd, BACKLOG) == -1) //开始监听
    {
    perror("listen");
    exit(1);
    }

while(1)
{
    sin_size = sizeof(struct sockaddr_in);
    if ((client_fd = accept(sockfd, (struct sockaddr *)&remote_addr, &sin_size)) == -1) //接收客户端的连接
            {
            perror("accept");
            continue;
            }
            printf("received a connection from %s\n", inet_ntoa(remote_addr.sin_addr));

    if (!fork())
            {
             /*  子进程代码段  */
                if (send(client_fd, "Hello, you are connected!\n", 26, 0) == -1) //往客户端发送消息
                perror("send");
            close(client_fd);
            exit(0);
            }
    close(client_fd);
    }
return 0;
}
```

本实验的客户端代码 skt_client.c 如下：

```
/*************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#define SERVPORT 3333
#define MAXDATASIZE 100 /*每次最大数据传输量  */

int main(int argc, char *argv[])
{
int sockfd, recvbytes; // sockfd 数据传输 socket
char buf[MAXDATASIZE];
struct hostent *host;
struct sockaddr_in serv_addr; //TCP/IP 协议的地址结构
if (argc < 2)
   {
   fprintf(stderr,"Please enter the server's hostname!\n");
   exit(1);
   }

if ((host=gethostbyname(argv[1]))==NULL) //用域名或主机名获取 IP 地址，可以直接输入 IP 地址
   {
   perror("gethostbyname 出错！ ");
   exit(1);
   }

if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)//创建套接字
   {
   perror("socket 创建出错！ ");
   exit(1);
   }

serv_addr.sin_family=AF_INET; //表示 TCP/IP 协议
serv_addr.sin_port=htons(SERVPORT); /*16 位端口号，网络字节顺序*/
serv_addr.sin_addr = *((struct in_addr *)host->h_addr); /*32 位服务器 IP 地址，网络字节顺序*/
bzero(&(serv_addr.sin_zero),8); /*保留*/
if (connect(sockfd, (struct sockaddr *)&serv_addr, sizeof(struct sockaddr)) == -1)//建立连接，连接的
serv_addr 为保存服务器地址的结构体
   {
   perror("connect error！ ");
   exit(1);
   }
```

```
if ((recvbytes=recv(sockfd, buf, MAXDATASIZE, 0)) ==-1) //接收数据
   {
   perror("recv 出错！ ");
   exit(1);
   }

buf[recvbytes] = '\0';
printf("Received: %s",buf);
close(sockfd); //关闭 socket
return 0;
}
```

5. 实验步骤

♦ 编译源程序

1）进入实验目录：

```
cbt@Cyb-Bot:~$ cd /CBT-6818/SRC/exp/basic/04_socket/
cbt@Cyb-Bot: /CBT-6818/SRC/exp/basic/04_socket $ ls
Makefile   skt_client   skt_client.c   skt_server   skt_server.c
cbt@Cyb-Bot: /CBT-6818/SRC/exp/basic/04_socket $
```

2）清除中间代码，重新编译

```
cbt@Cyb-Bot: /CBT-6818/SRC/exp/basic/04_socket $make clean
rm -f skt_client   skt_server *.elf *.gdb *.o
cbt@Cyb-Bot: /CBT-6818/SRC/exp/basic/04_socket $ make
gcc   -g -o skt_client skt_client.c
arm-linux-gcc -g -o skt_server skt_server.c
cbt@Cyb-Bot: /CBT-6818/SRC/exp/basic/04_socket $
```

当前目录下生成可执行程序 skt_server 和 skt_client。

♦ 运行程序（NFS 方式）

1）启动 CBT-6818 型实验系统，连好网线、串口线。通过串口终端挂载宿主机实验目录。

```
[root@CBT-6818:~]# mount -t nfs -o nolock 192.168.1.7:/CBT-6818     /mnt/
```

2）进入串口终端的 NFS 共享实验目录。

```
[root@CBT-6818:~]# cd /mnt/SRC/exp/basic/04_socket/
[root@CBT-6818: /mnt/SRC/exp/basic/04_socket]# ls
Makefile        skt_client      skt_ser         skt_server.c
skt_cli         skt_client.c    skt_server
```

```
[root@CBT-6818: /mnt/SRC/exp/basic/04_socket]#
```

3）执行程序。

在目标系统 ARM 端执行服务器程序：

```
[root@CBT-6818: /mnt/SRC/exp/basic/04_socket]# ./skt_server
```

4）在宿主机 ubuntu 端执行客户端程序 命令行参数输入 ARM 系统 IP 地址 192.168.1.230(出厂默认)：

```
cbt@Cyb-Bot: /CBT-6818/SRC/exp/basic/04_socket$ ./skt_client 192.168.1.230
```

此时 ARM 端服务器将出现连接提示：

```
[root@CBT-6818: /mnt/SRC/exp/basic/04_socket]# ./skt_server
```

♦ 实验结果

服务器端：

```
[root@CBT-6818: /mnt/SRC/exp/basic/04_socket]# ./skt_server
received a connection from 192.168.1.7
```

客户端：

```
cbt@Cyb-Bot: /CBT-6818/SRC/exp/basic/04_socket$ ./skt_client 192.168.1.230
Received: Hello, you are connected!
cbt@Cyb-Bot: /CBT-6818/SRC/exp/basic/04_socket$
```

本实验默认 Makfile 文件中将服务器端程序指定为交叉编译，将客户端程序指定为本机编译器，用户可以自行修改测试其他目标机编译方法。

实验五. 嵌入式 SQLite 应用

1. 实验目的

- 学习在 linux 环境下搭建嵌入式数据 sqlite 的基本方法。
- 掌握嵌入式系统下 sqlite 的基本使用方法、API 接口函数、sqlite 语法。

2. 实验环境

- 硬件：A53 智能网关实验平台，PC 机 Pentium 500 以上，硬盘 40G 以上，内存大于 256M。
- 软件：Vmware Workstation +ubuntu + MiniCom/超级终端 +ARM-LINUX 交叉编译开发环境。
- 实验目录：/CBT-6818/SRC/exp/basic/05_sqlite。

3. 实验内容

- 掌握嵌入式数据库 SQLite3.3.8 在嵌入式 Linux 中的应用。
- 编写用户应用程序对嵌入式数据库 SQLite 进行访问和交互。

4. 实验原理

4.1 SQLite 概述

数据库的目标是实现对数据的存储、检索等功能。传统的数据库产品除提供了基本的查询、添加、删除等功能外，也提供了很多高级特性，如触发器、存储过程、数据备份恢复等。但实际上用到这些高级功能的时候并不多，应用中频繁用到的还是数据库的基本功能。于是，在一些特殊的应用场合，传统的数据库就显得过于臃肿了。在这种情况下，嵌入式数据库开始崭露头角。

嵌入式数据库是一种具备了基本数据库特性的数据文件，它与传统数据库的区别是：嵌入式数据库采用程序方式直接驱动，而传统数据库则采用引擎响应方式驱动。嵌入式数据库的体积通常都很小，这使得嵌入式数据库常常应用在移动设备上。由于性能卓越，所以在高性能的应用上也经常见到嵌入式数据库的身影。

SQLite 是一种采用 C 语言开发的嵌入式数据库。SQLite 的目标是尽量简单，因此它抛弃了传统企业级数据库的种种复杂特性，只实现那些对于数据库而言非常必要的功能。尽管简单性是 SQLite 追求的首要目标，但是其功能和性能都非常出色。

SQLite 具有这样一些特点：

1） ACID 事务；

2） 零配置 – 无需安装和管理配置；

3） 储存在单一磁盘文件中的一个完整的数据库；

4） 数据库文件可以在不同字节顺序的机器间自由的共享；

5） 支持数据库大小至 2TB；

6） 足够小，大致 3 万行 C 代码，250K；

7） 比一些流行的数据库在大部分普通数据库操作要快；

8） 简单，轻松的 API；

9） 包含 TCL 绑定，同时通过 Wrapper 支持其他语言的绑定；

10） 良好注释的源代码，并且有着 90%以上的测试覆盖率；

11） 独立：没有额外依赖；

12） Source 完全的 Open，你可以用于任何用途，包括出售它；

13） 支持多种开发语言,C, PHP, Perl, Java, ASP.NET,Python；

由于 SQLite 具有功能强大、接口简单、速度快、占用空间小这样一些特殊的优点，因此特别适合于应用在嵌入式环境中。SQLite 在手机、PDA、机顶盒等设备上已获得了广泛应用。

SQLite 数据库的 SQL 语法：

SQLite 库可以解析大部分标准 SQL 语言。但它也省去了一些特性并且加入了一些自己的新特性。

SQLite 执行如下的语法。

- ALTER TABLE
- ANALYZE
- ATTACH DATABASE
- BEGIN TRANSACTION
- 注释
- COMMIT TRANSACTION
- COPY
- CREATE INDEX
- CREATE TABLE
- CREATE TRIGGER
- CREATE VIEW
- DELETE
- DETACH DATABASE

- DROP INDEX
- DROP TABLE
- DROP TRIGGER
- DROP VIEW
- END TRANSACTION
- EXPLAIN
- 表达式
- INSERT
- ON CONFLICT 子句
- PRAGMA
- REINDEX
- REPLACE
- ROLLBACK TRANSACTION
- SELECT
- UPDATE
- VACUUM

以上只是对 SQLite 实现的 SQL 语法的综述，有所忽略。

4.2 SQLite C 接口

由于早期的 SQLite 只支持 5 个 C/C++接口，因而非常容易学习和使用，但是随着 SQLite 功能的增强，新的 C/C++接口不断的增加进来，到现在有超过 150 个不同的 API 接口。这往往使初学者望而却步。幸运的是，大多数 SQLite 中的 C/C++接口是专用的，因而很少被使用到。尽管有这么多的调用接口，核心的 API 仍然相对简单和便于调用。

以下举例常用的 SQLite 数据库与用户应用程序 C/C++接口

♦ **sqlite3_open()**

该接口打开与一个 SQLite 数据库文件的连接并返回一个数据库连接对象。这通常是应用程序调用的第一个 SQLite API 接口而且也是调用其他 SQLite API 接口前需要调用的接口。许多 SQLite 接口需要一个指向数据库连接对象的指针作为它们的第一个参数，因而这些接口也可以理解成是数据库连接对象的操作接口。该接口就是创建了这样一个数据库连接对象。

♦ **sqlite3_prepare()**

该接口把一个 SQL 语句文本转换成一个预处理语句对象并返回一个指向该对象的指

针。这个接口需要一个由先前调用 sqlite3_open()返回的数据库连接对象指针以及一个预处理的 SQL 语句文本字符串为参数。这个 API 并不实际解析 SQL 语句，仅仅是为后续的解析而对 SQL 语句进行的预处理。

♦ **sqlite3_step()**

该接口用于解析一个由先前通过 sqlite3_prepare()接口创建的预处理语句，直至返回第一列结果为止。通过再次调用 sqlite3_step()可以返回下一列的结果，继续不断地调用 sqlite3_step()直至整个语句完成为止。对于那些并不返回结果的语句(例如：INSERT，UPDATE，DELETE 语句)一次调用 sqlite3_step()就完成了语句的处理。

♦ **sqlite3_column()**

该接口返回一个由 sqlite3_step()解析的预处理语句结果集中当前行的某一列数据。每次执行 sqlite3_step()都返回一个新的结果集中的一行。可以多次调用 sqlite3_column()接口返回那一行中所有列的数据。就像上面所说的那样，SQLite API 中并没有 sqlite3_column()这样的接口。取而代之的是一组用于从结果集中查询出各个列项各种数据类型数据的函数接口。在这组函数接口中，有些接口返回结果集的大小，有些返回结果集的列数。

-- sqlite3_column_blob()

-- sqlite3_column_bytes()

-- sqlite3_column_bytes16()

-- sqlite3_column_count()

-- sqlite3_column_double()

-- sqlite3_column_int()

-- sqlite3_column_int64()

-- sqlite3_column_text()

-- sqlite3_column_text16()

-- sqlite3_column_type()

-- sqlite3_column_value()

♦ **sqlite3_finalize()**

该接口销毁之前调用 sqlite3_prepare()创建的预处理语句。每一个预处理语句都必须调用这个接口进行销毁以避免内存泄漏。

♦ **sqlite3_close()**

该接口关闭一个由之前调用 sqlite3_open()创建的数据库连接。所有与该连接相关的预处理语句都必须在关闭连接之前销毁。

一个应用程序可以通过执行以下几个步骤执行一条 SQL 语句：

（1）使用 sqlite3_prepare()创建一个预处理语句。

（2）重复调用 sqlite3_step()解析执行预处理语句。

（3）对于查询操作，两次调用 sqlite3_step()之间通过 sqlite3_column()接口查询返回的结果。

（4）最后，使用 sqlite3_finalize()销毁预处理语句。

上述就是有效使用 SQLite 所需要知道的知识，其余的仅仅是些需要补充的细节而已。

♦ **sqlite3_exec()**

该接口是执行上述四个步骤的一个方便的应用调用封装接口，传递给 sqlite3_exec()的回调函数用于处理每一列返回的结果集。

4.2 键代码分析

1）consoleshell.c 源码分析

consoleshell.c 源文件主要是建立用户终端显示控制台菜单，并根据用户的输入字符，调用相应的接口函数。

```
#include "consoleshell.h"

char shell_s[]="\nshell> ";
void get_line(char *cmd);
/**
**
**控制台
**
*/
void * consoleshell(){
        int count=0;
        int i;
        char cmd[256]={0,};
        int rc = sqliteDB_open();
        char name[40],id[40];
        int price = 0;
 /*  打印终端控制台菜单  */
        printf("\n<DB control shell>");
        printf("\n [1]   select all the records in table merchandise");
        printf("\n [2]   select the the record which you known its name");
        printf("\n [3]   select the record by the id");
        printf("\n [4]   delete record");
        printf("\n [5]   add record");
        printf("\n [**] help menu");
        printf("\n [0]   print the end of the context an exit");
```

```
        while(1){
                printf(shell_s);
                fflush(stdout);
/*    获取用户终端输入     */
                get_line(cmd);

/*    根据输入批处理 SQLite 接口调用     */

                if(strncmp("1",cmd,1)==0){
                        /*   选择全部商品记录   */
                        sqliteDB_opt_select_all();
                }else if(strncmp("2",cmd,1)==0){
                        /*   根据名字查找商品记录   */
                        printf("\nenter the record name ");
                        printf("\nname:");
                        scanf("%s",name);
                        fflush(stdin);//刷新缓冲区
                        if(!sqliteDB_exist(name))
                                continue;
                        sqliteDB_opt_select(name);
                }else if(strncmp("3",cmd,1)==0){
                        /*   根据 ID 查找商品记录   */
                        printf("\nenter the record id ");
                        printf("\nid:");
                        scanf("%s",id);
                        fflush(stdin);//刷新缓冲区
                        sqliteDB_opt_select_by_id(id);
                }else if(strncmp("4",cmd,1)==0){
                        /*   根据名字删除商品记录   */
                        printf("\nplease enter the info of the record you want to delete!");
                        printf("\nname:");
                        scanf("%s",name);
                        fflush(stdin);//刷新缓冲区
                        if(!sqliteDB_exist(name))
                                continue;
                        sqliteDB_opt_select(name);
                                //删除之前应再次询问是否删除，不可回滚
                        sqliteDB_opt_delete(name);
                }else if(strncmp("5",cmd,1)==0){
                         /*   增加商品记录   */
                        printf("\nplease enter the info of the record you want to add!\nid:");
                        printf("\nenter the record id ");
                        printf("\nid:");
                        scanf("%s",id);
                        printf("\nname:");
```

```
                    scanf("%s",name);
                    fflush(stdin);//刷新缓冲区
                    printf("\nprice:");
                    scanf("%d",&price);
                    fflush(stdin);//刷新缓冲区
                    sqliteDB_opt_add(name,id,price);
                }else if(strncmp("**",cmd,2)==0){
                    /*   显示帮助控制台菜单   */
                    printf("\n<DB control shell>");
                    printf("\n [1]   select all the records in table merchandise");
                    printf("\n [2]   select the the record which you known its name");
                    printf("\n [3]   select the record by the BarCode Scanner");
                    printf("\n [4]   delete record");
                    printf("\n [5]   add record");
                    printf("\n [**] help menu");
                    printf("\n [0]   print the end of the context an exit");
                }else if(strncmp("0",cmd,1)==0){
                    /*   退出 关闭数据库   */
                    sqliteDB_close();//关闭数据库连接
                    break;
//              }else if(cmd[0] != "0"){
                }else if(strncmp("0",cmd,1)!=0){
                    system(cmd);
                }
        }
}
/*   获取用户终端输入   */
void get_line(char *cmd){
        int i=0;
        while(1){
                cmd[i]=getchar();
                if(cmd[i]==10){
                    cmd[i]=0;
                    break;
                }
                fflush(stdout);
                i++;
        }
}
```

2）SQLite.c 源程序分析

该源码文件主要是对 SQLite 数据的 C/C++接口封装，提供 consoleshell.c 中程序调用。

```
/*
*建立与数据库的连接
*/
int sqliteDB_open(){
```

```
	int rc;//操作标志
	printf("\ncreat database:test.db\n\ncreat table:table merchandise(contents:id name price)\n\nand add two records\n");
	rc = sqlite3_open("test.db", &db);
	if( rc ){
		fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(db));
		sqlite3_close(db);
		exit(1);
	}
	printf("\nOpen sucess!");
 /*  建立商品初始化 table */
	if(!sqliteDB_create_table())
		printf("\ntable exist");
	return 1;
}
/*
* 关闭与数据库的连接
*/
int sqliteDB_close(){
	if(db != 0)
		sqlite3_close(db);
}
/*
* 添加一条记录到已知或未知数据库表
*/
int sqliteDB_opt_add(char *name,char *id,int price){
	int rc;
	char *zErrMsg = 0;
	char *sql=0;//动态生成的 SQL 语句
	/*  定义 SQL 语句格式字符串  */
	char tem_sql[256]="insert into merchandise values('";
	char tem_sql0[5] = "','";
	char tem_sql1[5] = "',";
	char tem_sql2[5] = ");";
	char tem_price[20];

	sprintf(tem_price, " %d" , price);//将 int 数据转换为字符串
	/*  组合 SQL 语句格式  */
	sql = strcat(tem_sql,name);
	sql = strcat(sql,tem_sql0);

	sql = strcat(sql,id);
	sql = strcat(sql,tem_sql1);

	sql = strcat(sql,tem_price);
```

```
        sql = strcat(sql,tem_sql2);
        /*  添加记录到 SQLite  */
        rc = sqlite3_exec(db, sql, callback, 0, &zErrMsg);
        if( rc!=SQLITE_OK ){
                fprintf(stderr, "SQL error: %s\n", zErrMsg);
                sqlite3_free(zErrMsg);
        }
}
```

以上由于篇幅有限，给出部分代码，其他接口原理相同，用户可以自行分析该文件全部源码。

5. 实验步骤

♦ **编译源程序**

1）进入实验目录：

```
cbt@Cyb-Bot:~$ cd /CBT-6818/SRC/exp/basic/05_sqlite/
cbt@Cyb-Bot: /CBT-6818/SRC/exp/basic/05_sqlite$ ls
Makefile    SQLite.c   SQLite.o   consoleshell.h   linuxpatch.h   main.o    sqlite
Rules.mak   SQLite.h   consoleshell.c   consoleshell.o   main.c          readme.txt
cbt@Cyb-Bot: /CBT-6818/SRC/exp/basic/05_sqlite$
```

2）清除中间代码，重新编译。

```
cbt@Cyb-Bot: /CBT-6818/SRC/exp/basic/05_sqlite$ make clean
rm -f ./SQLite_test ./sqlite/SQLite_test *.elf *.elf2flt *.gdb *.o
cbt@Cyb-Bot: /CBT-6818/SRC/exp/basic/05_sqlite$ make
arm-linux-gcc       -c -o main.o main.c
arm-linux-gcc       -c -o SQLite.o SQLite.c
SQLite.c: In function 'sqliteDB_opt_select_all':
SQLite.c:184: warning: assignment discards qualifiers from pointer target type
SQLite.c:185: warning: assignment discards qualifiers from pointer target type
SQLite.c: In function 'sqliteDB_opt_select':
SQLite.c:226: warning: assignment discards qualifiers from pointer target type
SQLite.c:227: warning: assignment discards qualifiers from pointer target type
SQLite.c: In function 'sqliteDB_opt_select_by_id':
SQLite.c:268: warning: assignment discards qualifiers from pointer target type
SQLite.c:269: warning: assignment discards qualifiers from pointer target type
arm-linux-gcc       -c -o consoleshell.o consoleshell.c
arm-linux-gcc -static      -I ./sqlite/ -L ./sqlite/ -o SQLite_test main.o SQLite.o consoleshell.o      -lsqlite3
arm-linux-gcc -static      -I ./sqlite/ -L ./sqlite/ -o sqlite/SQLite_test main.o SQLite.o consoleshell.o      -lsqlite3
cbt@Cyb-Bot: /CBT-6818/SRC/exp/basic/05_sqlite$ ls
Makefile    SQLite.c   SQLite.o   consoleshell.c   consoleshell.o   main.c    readme.txt
Rules.mak   SQLite.h   SQLite_test   consoleshell.h   linuxpatch.h       main.o    sqlite
cbt@Cyb-Bot: /CBT-6818/SRC/exp/basic/05_sqlite$
```

当前目录下生成可执行程序 SQLite_test。

♦ **运行程序（NFS 方式）**

1）启动 CBT-6818 型实验系统，连好网线、串口线。通过串口终端挂载宿主机实验目录。

```
[root@CBT-6818 /$ mount -t nfs -o nolock 192.168.1.7:/CBT-6818      /mnt/
```

2）进入串口终端的 NFS 共享实验目录。

```
[root@CBT-6818:~]# cd /mnt/SRC/exp/basic/05_sqlite/
[root@CBT-6818:/mnt/SRC/exp/basic/05_sqlite]# ls
Makefile            SQLite.o            consoleshell.o
Rules.mak           SQLite_test         linuxpatch.h       sqlite
SQLite.c            consoleshell.c      main.c
SQLite.h            consoleshell.h      main.o
[root@CBT-6818:/mnt/SRC/exp/basic/05_sqlite]#
```

3）执行程序。

```
[root@CBT-6818:/mnt/SRC/exp/basic/05_sqlite]# ./SQLite_test
```

此时会在当前目录下创建数据库 test.db 文件，用户可以根据终端菜单来对此数据库进行操作了，同样可以使用“ls －l”命令观察数据库大小的变化

♦ **实验结果**

```
[root@CBT-6818:/mnt/SRC/exp/basic/05_sqlite]# ./SQLite_test

creat database:test.db

creat table:table merchandise(contents:id name price)

Open sucess!
<DB control shell>
  [1]   select all the records in table merchandise
  [2]   select the the record which you known its name
  [3]   select the record by the id
  [4]   delete record
  [5]   add record
  [**] help menu
  [0]   print the end of the context an exit
shell>
```

可以按照终端提示进行操作测试，如：

```
Open sucess!
<DB control shell>
```

```
  [1]    select all the records in table merchandise
  [2]    select the the record which you known its name
  [3]    select the record by the id
  [4]    delete record
  [5]    add record
  [**] help menu
  [0]    print the end of the context an exit
shell> 5     /*选择添加一条记录*/
please enter the info of the record you want to add!
id:
enter the record id
id:1    /*输入 ID 为 1 */

name:cyb-bot       /* name 为 cyb-bot */

price:100        /*  价格为 100 */

shell>
shell> 1        /*    查看已有商品记录    */

             name                  id                    price
             ------------------------------------------------------------
             cyb-bot               1                     100

shell> 0      /*    退出操作  关闭数据库  */
[root@CBT-6818:/mnt/SRC/exp/basic/05_sqlite]#
```

实验六. 嵌入式 WebServer 移植

1. 实验目的

- 学习在 linux 环境下搭建嵌入式 WEB 服务器的基本方法。
- 掌握一种常见嵌入式 WEB 服务器开源项目的移植和使用测试方法。

2. 实验环境

- 硬件：A53 智能网关实验平台，PC 机 Pentium 500 以上，硬盘 40G 以上，内存大于 256M。
- 软件：Vmware Workstation +ubuntu + MiniCom/超级终端 +ARM-LINUX 交叉编译开发环境。
- 实验目录：/CBT-6818/SRC/exp/basic/06_webserver。

3. 实验内容

- 移植一款嵌入式 WEB 服务器到 CBT-6818 系统中，并使用浏览器测试该服务器。

4. 实验原理

4.1 goAhead 服务器

♦ **goAhead 简介**

GoAhead Web 服务器是一款主要面向嵌入式系统的 WEB 服务器，它的目标也许不在于目前的 WEB 服务器市场，而是面向当嵌入式系统深入我们的工作与生活的明天，那时，它也许会成为使用最广泛的 WEB 服务器。

GoAhead Web 服务器是 GoAhead 公司的 Embedded Management Framework 产品的一部分，这个软件包主要用于解决未来嵌入式系统开发的相关问题。这款 WEB 服务器非常小巧，它的 WIN CE 版本编译后的大小还不到 60k，它的输出通常也是面向一些小屏幕设备。在性能方面，使用一颗 24MHz 的 68040 处理器，它的响应速度为 20 次/秒，使用 266MHz 的 Pentium 处理器可以达到 50 次/秒的响应速度。

仅管它的体积非常小巧，GoAhead WEB 服务器提供了不少的服务特性。它支持 ASP，嵌入的 JAVASCRPT 与内存 CGI 处理。

到目前为止，GoAhead 的源码完全免费，使用它无需交纳版税或者许可证费用。对一些硬件开发者而言，这种做法比较普遍，例如 SUN 旗下的众多开源系统。这样作的主要原因是 GoAhead 公司希望它成为未来嵌入式环境下的 WEB 服务器标准平台并看好日益增长的更智能化嵌入式设备的市场。

GoAhead Webserver 是跨平台的服务器软件，可以稳定地运行在 Windows，Linux 和 Mac OS X 操作系统之上。

♦ **goAhead 特点**

- 很小的内存消耗
- 支持认证功能 Digest Access Authentication (DAA)
- 支持安全的通信，例如 SSL（安全的套接字层）
- 支持动态 Web 页面，如 ASP 页面
- 可以使用传统的 C 语言编程定制 Web 页面里的 HTML 标签
- 支持 CGI（公共网关编程接口）
- 嵌入式的 JavaScript 脚本翻译器
- 独特的 URL 分析器
- 它基本上属于一个 HTTP1.0 标准的 WEB 服务器，对一些 HTTP1.1 的特性如（持久连接）也提供了支持。每秒 65 次 connections

4.2 goAhead 源码结构

ws031202：各种 OS 移植子目录，分别有：CE、ECOS、LINUX、LYNX、MACOSX、NW、QNX4、VXWORKS、WIN，同时包含 main 主函数。

web：存放 web 网页及说明文档。

webs.h webs.c 等服务器源程序文件（C 程序文件）。

从上面可以看到，goAhead 支持 window 系统的，有兴趣的可以在 PC 机上移植试试。

我们主要移植的是 LINUX 目录下的内容，方法也很简单，修改适当文件并交叉编译即可生成嵌入式 LINUX 系统中的 WEB 服务器。

4.3 关键代码分析

1）main.c 源码分析

main.c 源文件是整个系统的程序入口，负责完成服务器初始化，服务监听及相关接口设置的工作，这里我们只给出该主函数的流程分析，具体子函数及 SOCKET 部分，读者可以自行分析。本节实验主要目的在于讲解 goAhead 服务器的移植过程及使用测试方法。

```
static char_t           *rootWeb = T("web");        /*  默认的服务器工作目录*/
static char_t           *password = T("");          /*  默认安全密码设置  */
static int                      port = 80;          /*  默认服务器端口号*/

/************************************ Code ************************************/
/*
```

```
 *          Main -- entry point from LINUX
 */

int main(int argc, char** argv)
{
        /*内存空间分配初始化*/
        bopen(NULL, (60 * 1024), B_USE_MALLOC);
        signal(SIGPIPE, SIG_IGN);
/*
 *          初始化服务器
 */
        if (initWebs() < 0) {
                return -1;
        }
/*
        服务器事件监听循环
 */
        while (!finished) {
                if (socketReady(-1) || socketSelect(-1, 1000)) {
                        socketProcess(-1);
                }
                websCgiCleanup();
                emfSchedProcess();
        }
/*
 *          关闭服务器 SOCKET
 */
        websCloseServer();
        socketClose();
/*          释放内存空间            */
        bclose();
        return 0;
}

/******************************************************************************/
/*
 *          初始化服务器函数
 */

static int initWebs()
{
/*
 *          初始化 socket 子系统
 */
        socketOpen();
/*
```

```
        获取系统 IP 地址，并初始化
*/
        if (gethostname(host, sizeof(host)) < 0) {
                error(E_L, E_LOG, T("Can't get hostname"));
                return -1;
        }
        printf("hostname=%s\n",host);
        if ((hp = gethostbyname(host)) == NULL) {
                error(E_L, E_LOG, T("Can't get host address"));
                return -1;
        }
        memcpy((char *) &intaddr, (char *) hp->h_addr_list[0],
                (size_t) hp->h_length);
/*
设置服务器工作目录
*/
        websSetDefaultDir(webdir);
/*  转化 IP 数据字节序，编码格式等*/
        cp = inet_ntoa(intaddr);
        ascToUni(wbuf, cp, min(strlen(cp) + 1, sizeof(wbuf)));
        websSetIpaddr(wbuf);
        ascToUni(wbuf, host, min(strlen(host) + 1, sizeof(wbuf)));
        websSetHost(wbuf);

/*
   配置服务器默认打开首页及安全密码
 */
        websSetDefaultPage(T("default.asp"));
        websSetPassword(password);

/*
    通过指定端口打开服务器
 */
        websOpenServer(port, retries);
/*
    建立相应地址解析处理函数
*/
        websUrlHandlerDefine(T(""), NULL, 0, websSecurityHandler,
                WEBS_HANDLER_FIRST);
        websUrlHandlerDefine(T("/goform"), NULL, 0, websFormHandler, 0);
        websUrlHandlerDefine(T("/cgi-bin"), NULL, 0, websCgiHandler, 0);
        websUrlHandlerDefine(T(""), NULL, 0, websDefaultHandler,
                WEBS_HANDLER_LAST);
/*
    定义初始化 2 个内嵌接口，ASP 和 CGI 测试用
*/
```

```
        websAspDefine(T("aspTest"), aspTest);
        websFormDefine(T("formTest"), formTest);
/*
 *      建立默认首页地址解析处理
 */
        websUrlHandlerDefine(T("/"), NULL, 0, websHomePageHandler, 0);
        return 0;
}
```

以上由于篇幅有限，给出部分代码，用户可以自行分析该系统的全部源码。

5. 实验步骤

♦ **编译 goAhead 源码工程**

1）进入实验目录：

```
cbt@Cyb-Bot:~$ cd /CBT-6818/SRC/basic/06_webserver/
cbt@Cyb-Bot: /CBT-6818/SRC/basic/06_webserver$ ls
 webs218.tar.gz
cbt@Cyb-Bot: /CBT-6818/SRC/basic/06_webserver$
```

2）解压源码工程

```
cbt@Cyb-Bot: /CBT-6818/SRC/basic/06_webserver$ tar xzvf webs218.tar.gz
cbt@Cyb-Bot: /CBT-6818/SRC/basic/06_webserver$ cd ws031202/
cbt@Cyb-Bot: ws031202$ ls
asp.c                ECOS          handler.c    md5.h          release.htm  uemf.c     web
websSSL.h
balloc.c             ej.h          h.c          mime.c         release.txt  uemf.h     webcomp.c
websuemf.c
base64.c             ejIntrn.h     license.txt  misc.c         ringq.c      um.c       webrom.c   WIN
CE                   ejlex.c       LINUX        mocana_ssl.c   rom.c        um.h       webs.c
wsIntrn.h
cgi.c                ejparse.c     LYNX         NW             security.c   umui.c     websda.c
default.c            emfdb.c       MACOSX       page.c         sock.c       url.c      websda.h
default.css          emfdb.h       Makefile     QNX4           sockGen.c    value.c    webs.h
DMF_Brochure.pdf     form.c        md5c.c       readme.txt     sym.c        VXWORKS    websSSL.c
cbt@Cyb-Bot: /CBT-6818/SRC/basic/06_webserver/ ws031202$
```

3）修改 LINUX 目录下的 Makefile 文件，添加编译器宏定义

```
cbt@Cyb-Bot: /CBT-6818/SRC/basic/06_webserver/ ws031202$ cd LINUX/
cbt@Cyb-Bot: /CBT-6818/SRC/basic/06_webserver/ ws031202/LINUX$ vi Makefile
```

在文件前面部分加入变量 CC 和 AR 的定义。例如内容如下：

```
all:    compile
```

```
ARCH       = libwebs.a
NAME       = webs
## 添加如下定义
CC         = arm-linux-gcc
AR         = arm-linux-ar
```

注释掉 Makfile 文件最后两行代码“#”注释：

```
#.c.o:
#           cc -c -o $@ $(DEBUG) $(CFLAGS) $(IFLAGS) $<
```

保存 Makefile

4）修改 LINUNX 目录上级目录的 misc.c 文件，注释掉第 61 行的 strnlen 函数声明

```
cbt@Cyb-Bot: /CBT-6818/SRC/basic/06_webserver/ ws031202/LINUX$ vi ../misc.c
```

内容如下：

```
static int          dsnprintf(char_t **s, int size, char_t *fmt, va_list arg, int msize);
//static int        strnlen(char_t *s, unsigned int n);// 注释掉该函数声明
static void         put_char(strbuf_t *buf, char_t c);
static void         put_string(strbuf_t *buf, char_t *s, int len,
```

另外注释掉该文件中第 428 行处，该函数的定义。

```
/*
static int strnlen(char_t *s, unsigned int n)
{
        unsigned int       len;

        len = gstrlen(s);
        return min(len, n);
}
*/
```

退出保存 misc.c 文件。

5）编译 goAhead 工程

```
cbt@Cyb-Bot: /CBT-6818/SRC/basic/06_webserver/ ws031202/LINUX$ make
cbt@Cyb-Bot: /CBT-6818/SRC/basic/06_webserver/ ws031202/LINUX$ ls
libwebs.a   main.c   main.o   Makefile   webs
cbt@Cyb-Bot: /CBT-6818/SRC/basic/06_webserver/ ws031202/LINUX$
```

编译成功后，会在当前目录下生成 webs 服务器程序。

♦ **运行程序（NFS 方式）**

1）启动 CBT-6818 型实验系统，连好网线、串口线。通过串口终端挂载宿主机实验目录。

```
[root@CBT-6818:~]# mount -t nfs -o nolock 192.168.1.7:/CBT-6818      /mnt/
```

2）进入串口终端的 NFS 共享实验目录。

```
[root@CBT-6818:~]# cd /mnt/SRC/exp/basic/06_webserver/
[root@CBT-6818: /mnt/SRC/exp/basic/06_webserver]# ls
webs218.tar.gz    ws031202
```

3）运行 WEB 服务器前，执行如下命令启动 IP 地址与主机名映射(ARM 端执行)。

首先修改 ARM 系统/etc/hosts 文件内容，确保添加入 IP 地址与主机名的映射内容如下：

```
192.168.1.230      cbt
```

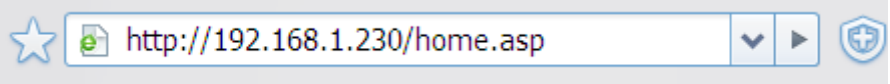
接下来设置主机名：

```
[root@CBT-6818: /mnt/SRC/exp/basic/06_webserver]# hostname -F /etc/hosts
[root@CBT-6818: /mnt/SRC/exp/basic/06_webserver]#
```

4）运行 WEB 服务器。进入到 NFS 共享目录的 ws031202/LINUX/目录下

```
[root@CBT-6818: /mnt/SRC/exp/basic/06_webserver]#    cd ws031202/LINUX/
[root@CBT-6818: /mnt/SRC/exp/basic/06_webserver/ws031202/LINUX/]# ./webs
```

5）在宿主机端使用浏览器访问 ARM 系统 IP 地址：http://192.168.1.230/home.asp



注意：出厂时候 ARM 系统默认 IP 地址为 192.168.1.230。访问服务器页面前，请确保宿主机浏览器端机器 IP 地址与服务器 ARM 端 IP 地址在同一个网段。

♦ 实验结果

浏览器默认打开 NFS 共享目录下 goAhead 工程源码中 web 目录下的页面。关于该 WEB 服务器的更多功能，用户感兴趣可以自行测试。

第三章. 基于 Qt 的 GUI 实验

Qt 是跨平台的应用程序和 UI 框架。 它包括跨平台类库、集成开发工具和跨平台 IDE。使用 Qt 您只需一次性开发应用程序，无须重新编写源代码，便可跨不同桌面和嵌入式操作系统部署这些应用程序。

通过本章的学习，用户可以掌握 Linux 系统下如何使用 Qt 的相关工具进行界面 UI 的设计工作，并通过搭建不同目标平台的 Qt 开发环境，轻松的部署属于自己的人机交互界面。最后一部分会介绍移植 Qt 图形用户程序到 ARM 设备上，并在此基础上增加了对嵌入式设备上常用的输入设备如触摸屏、鼠标等的支持。

本章所涉猎的程序，需要用户有一定的 C++语言基础来完成 Qt 程序的设计，当然我们也可以通过 Qt 配套的 IDE 开发环境以尽量避免编写代码的方式进行简单的界面设计。

实验一. 搭建本机 Qt 开发环境

1. 实验目的

- 掌握在 Linux 系统下安装搭建 Qt 本机 X11 开发环境的基本步骤和方法。
- 熟悉 Qt 开发环境中常用的工具软件和相关方法。

2. 实验环境

- 硬件：ICS-IOT-CEP 实验平台，PC 机 Pentium 500 以上，硬盘 40G 以上，内存大于 256M。
- 软件：Vmware Workstation +ubuntu + MiniCom/超级终端 +ARM-LINUX 交叉编译开发环境。
- 实验目录：qt-everywhere-opensource-src-4.7.0/examples/widgets/calculator

3. 实验内容

- 编译本机 Qt 开发环境。
- 编写基于 Qt 的 DEMO 程序或利用 Qt 自带的 DEMO，验证搭建的本机开发环境

4. 实验原理

4.1 Qt 简介

Qt 是一个 1991 年由奇趣科技开发的跨平台 C++图形用户界面应用程序开发框架。它既可以开发 GUI 程式，也可用于开发非 GUI 程式，比如控制台工具和服务器。Qt 是面向对象语言，易于扩展，并且允许组件编程。2008 年，奇趣科技被诺基亚公司收购，QT 也因此成为诺基亚旗下的编程语言工具。

基本上，Qt 同 X Window 上的 Motif，Openwin，GTK 等图形界面库和 Windows 平台上的 MFC，OWL，VCL，ATL 是同类型的东西，但是 Qt 具有下列优点:

♦ **优良的跨平台特性:**

Qt 支持下列操作系统: Microsoft Windows 95/98， Microsoft Windows NT， Linux，Solaris， SunOS， HP-UX， Digital UNIX (OSF/1， Tru64)， Irix， FreeBSD，BSD/OS， SCO， AIX， OS390，QNX 以及有帧缓冲(framebuffer)支持的嵌入式 Linux 平台，Windows CE 等等。

♦ **面向对象**

Qt 的良好封装机制使得 Qt 的模块化程度非常高，可重用性较好，对于用户开发来说是非常方便的。 Qt 提供了一种称为 signals/slots 的安全类型来替代 callback，这使得各个元件之间的协同工作变得十分简单。

♦ **丰富的 API**

Qt 包括多达 400 个以上的 C++ 类，还替供基于模板的 collections， serialization，file， I/O device， directory management， date/time 类。甚至还包括正则表达式的处理功能。

♦ **支持 2D/3D 图形渲染，支持 OpenGL**

♦ **大量的开发文档**

♦ **Qt 的用途**

♦ **Qt 的开发工具**

- GUI Designer
- 国际化工具
- HTML 帮助系统
- Visual Studio 和 Eclipse 集成
- 跨平台构建工具
- Qt Creator

4.2 Qt 主要的类

♦ **Qobject**

Qobject 是 Qt 类体系的唯一基类，是 Qt 各种功能的源头活水，就象 MFC 中的 CObject 和 Dephi 中的 Tobject。

QApplication 和 QWidget 都是 QObject 类的子类。

♦ **Qapplication**

Qapplication 类负责 GUI 应用程序的控制流和主要的设置，它包括主事件循环体，负责处理和调度所有来自窗口系统和其他资源的事件，并且处理应用程序的开始、结束以及会话管理，还包括系统和应用程序方面的设置。对于一个应用程序来说，建立此类的对象是必不可少的。

♦ **Qwidget**

Qwidget 类是所有用户接口对象的基类，它继承了 QObject 类的属性。组件是用户界面的单元组成部分，它接收鼠标、键盘和其它从窗口系统来的事件，并把它自己绘制在盘屏幕上。

QWidget 类有很多成员函数，但一般不直接使用，而是通过子类继承来使用其函数功能。如，QPushButton、QlistBox 等都是它的子类。

4.3 Qt 的事件机制

事件是由窗口系统或 qt 本身对各种事务的反应而产生的。当用户按下、释放一个键或鼠标按钮，一个键盘或鼠标事件被产生；当窗口第一次显示，一个绘图事件产生，从而告知最新的可见窗口需要重绘自身。大多数事件是由于响应用户的动作而产生的，但还有一些，比如定时器等，是由系统独立产生的。如图 4.3 所示：

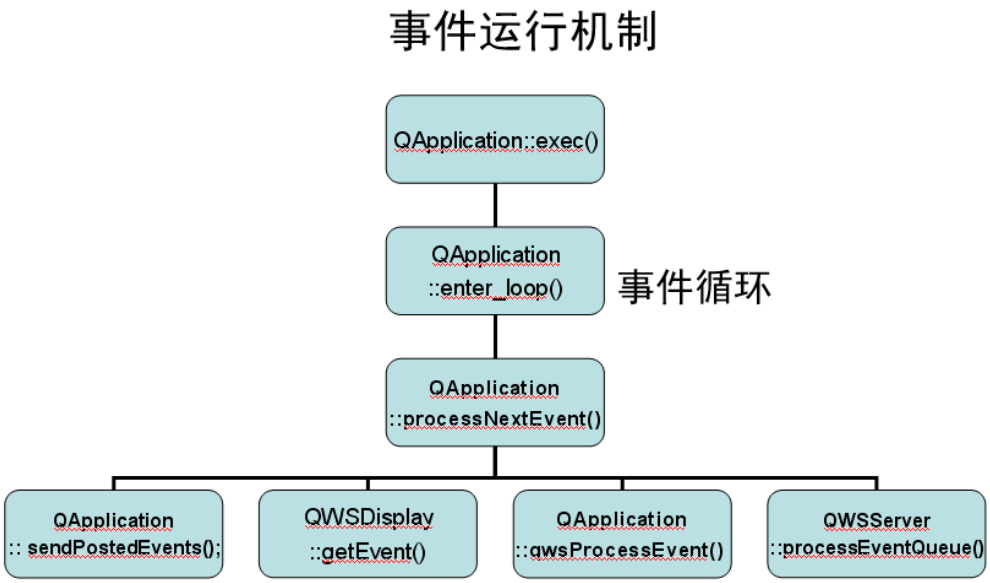


图 4.3 QT 事件机制

4.4 Qt 应用程序的编译

在 Qt 库环境搭建好的情况下，进行 Qt 应用程序的编译时相对容易的，我们可以使用配套 Qt 环境中提供的相关工具，对 Qt 的应用程序源码工程进行编译。如其中最常用的编译工具即 qmake 工具。

编译应用程序工程的方法如下：

1）使用 qmake 生成.pro 工程文件

```
[root@Intel qt4]qmake -project
```

该命令会将当前目录或工程下所有源文件进行组织，并生成工程文件。

2）通过 qmake 自动生成 Makefile 文件

```
[root@Intel qt4]qmake
```

该命令会在当前目录下生成编译规则文件 Makefile

3）make 生成可执行文件

```
[root@Intel qt4]make
```

注意：以上命令使用的 qmake 工具应为配套 Qt 环境下编译生成的，不要使用一些 Linux 桌面系统自带的 qmake 等工具。

5. 实验步骤

♦ 建立本机 Qt 实验环境目录

1）进入宿主机中系统中，建立 Qt4x11-4.7.0 实验目录，例如手动在宿主机端用户目录下建立 cbt 目录。在其中创建目录 Qt4 及子目录 Qt4x11-4.7.0。所有 QT 相关实验都放在该目录下完成，后面文章不在赘述。

```
cbt@Cyb-Bot: ~$ cd
cbt@Cyb-Bot: ~$ mkdir cbt
cbt@Cyb-Bot: ~$ cd cbt/
cbt@Cyb-Bot: ~/cbt$ mkdir Qt4/
cbt@Cyb-Bot: ~/cbt$ cd Qt4/
cbt@Cyb-Bot: ~/cbt/Qt4$ mkdir Qt4x11-4.7.0/
cbt@Cyb-Bot: ~/cbt/Qt4$ cd Qt4x11-4.7.0/
cbt@Cyb-Bot: ~/cbt/Qt4/Qt4x11-4.7.0$
```

后续所有关于 QT-X11 实验环境都建在此目录(/home/cbt/cbt/cbt/Qt4/Qt4x11-4.7.0)下进行。

2）拷贝并解压 Qt 源码包

```
cbt@Cyb-Bot: ~/cbt/Qt4/Qt4x11-4.7.0$ cp /CBT-6818/SRC/gui/qt-everywhere-opensource-src-4.7.0.tar.gz ./
cbt@Cyb-Bot: ~/cbt/Qt4/Qt4x11-4.7.0$ tar xzvf qt-everywhere-opensource-src-4.7.0.tar.gz
```

解压后会在当前目录下生成解压后的 Qt 库源码目录 qt-everywhere-opensource-src-4.7.0。

3）编译配置 Qt X11 本机环境

进入 qt-everywhere-opensource-src-4.7.0 源码包目录，执行 configure 命令，配置 Qt 本地库环境。

```
cbt@Cyb-Bot: ~/cbt/Qt4/Qt4x11-4.7.0$ cd qt-everywhere-opensource-src-4.7.0
cbt@Cyb-Bot: ~/cbt/Qt4/Qt4x11-4.7.0/qt-everywhere-opensource-src-4.7.0$ ./configure
```

4）configure 的其他具体命令行参数配置用户可以通过 --help 命令查看：（如配置失败，请输入此命令：sudo apt-get install libxtst-dev）

```
cbt@Cyb-Bot: ~/cbt/Qt4/Qt4x11-4.7.0/qt-everywhere-opensource-src-4.7.0$ ./configure --help
```

默认指定的环境安装路径为/usr/local/Trolltech/Qt-4.7.0，当然用户也可以通过命令行参数-prefix 来指定环境编译好后的安装路径，方便查找，使用编译生成的工具。

执行 configure 命令后，本机环境一般不用特使命令行参数即可，使用默认参数。当出现选择 Qt 版本许可的时候，依次输入“o”表示开源许可，再输入“yes”表示同意协议即可完成。

5）编译 Qt 本机 X11 环境。

完成上述 configure 配置后，即可输入 make 来编译该 Qt 环境.

```
cbt@Cyb-Bot: ~/cbt/Qt4/Qt4x11-4.7.0/qt-everywhere-opensource-src-4.7.0$ make
```

6）安装 Qt 本机 X11 环境。

上述编译过程成功后，可以执行 make install 命令来安装 Qt 本机环境，默认安装路径为/usr/local/Trolltech/Qt-4.7.0，会在该目录下生成相应工具(如 qmake)和库文件等。

```
cbt@Cyb-Bot: ~/cbt/Qt4/Qt4x11-4.7.0/qt-everywhere-opensource-src-4.7.0$ sudo make install
```

注意：一般情况下，Qt 库的编译需要较长时间，根据机器硬件性能，可能几个小时不等。且 Qt 环境的编译，依赖宿主机系统和 Qt 具体的版本，本实验文档仅供 ubuntu 宿主机环境和 qt4.7 的版本库，其他环境及 Qt 库版本如遇问题，请参阅网络资源来解决。

♦ 运行 Qt 本机环境自带例程

Qt 源码库的路径下自带了一系列的 examples 例程，方便用户进行学习和参考，我们可以取其中的一些例程，编译后运行测试下前面搭建的 Qt 本机环境。

1）编译 Qt 例程应用程序

以 Qt 自带的计算器例程为例，可以先编译该程序。进入 Qt 源码目录 examples/widgets/calculator 中，利用前面编译库环境生成的 qmake 工具编译该工程。

```
cbt@Cyb-Bot: ~/cbt/Qt4/Qt4x11-4.7.0/qt-everywhere-opensource-src-4.7.0$ cd examples/widgets/calculator/
cbt@Cyb-Bot: ~/cbt/Qt4/Qt4x11-4.7.0/qt-everywhere-opensource-src-4.7.0/examples/widgets/calculator$ ls
button.cpp  button.h  calculator.cpp  calculator.h  calculator.pro  main.cpp
```

先生成 Makefile 因该工程中已经生成 pro 工程文件，因此无需再使用 qmake -project 命令了。

```
cbt@Cyb-Bot:  ~/cbt/Qt4/Qt4x11-4.7.0/qt-everywhere-opensource-src-4.7.0/examples/widgets/calculator
$ /usr/local/Trolltech/Qt-4.7.0/bin/qmake
cbt@Cyb-Bot: ~/cbt/Qt4/Qt4x11-4.7.0/qt-everywhere-opensource-src-4.7.0/examples/widgets/calculator $ ls
button.cpp  button.h  calculator.cpp  calculator.h  calculator.pro  main.cpp  Makefile
```

```
cbt@Cyb-Bot: ~/cbt/Qt4/Qt4x11-4.7.0/qt-everywhere-opensource-src-4.7.0/examples/widgets/calculator $
```

编译该工程

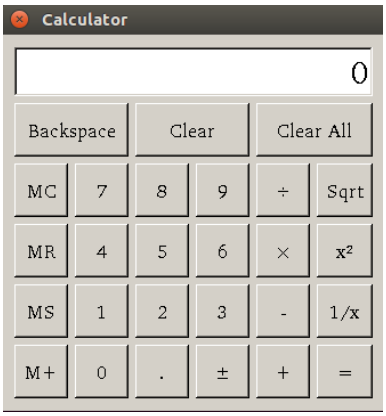
```
cbt@Cyb-Bot:      ~/cbt/Qt4/Qt4x11-4.7.0/qt-everywhere-opensource-src-4.7.0/examples/widgets/calculator
$ make
cbt@Cyb-Bot: ~/cbt/Qt4/Qt4x11-4.7.0/qt-everywhere-opensource-src-4.7.0/examples/widgets/calculator $ ls
button.cpp   calculator        calculator.debug   calculator.pro   Makefile
button.h     calculator.cpp    calculator.h       main.cpp
cbt@Cyb-Bot: ~/cbt/Qt4/Qt4x11-4.7.0/qt-everywhere-opensource-src-4.7.0/examples/widgets/calculator $
```

编译成功后，会在当前目录下生成 Qt 可执行文件 calculator。

2）运行 Qt 本机应用程序

```
cbt@Cyb-Bot:      ~/cbt/Qt4/Qt4x11-4.7.0/qt-everywhere-opensource-src-4.7.0/examples/widgets/calculator
$ ./calculator
```

3）计算器程序界面效果



实验二. 基于 QtDesigner 的程序设计

1. 实验目的

- 掌握 Qt Designer 的使用。
- 学会使用 Qt Designer 编写程序，编译，本机上运行。

2. 实验环境

- 硬件：ICS-IOT-CEP 实验平台，PC 机 Pentium 500 以上，硬盘 40G 以上，内存大于 256M
- 软件：Vmware Workstation +ubuntu + MiniCom/超级终端 +ARM-LINUX 交叉编译开发环境
- 实验目录：/home/cbt/cbt/QtDemo

3. 实验内容

- 在 Linux 下使用 Qt Designer 设计 QT 程序界面，在本机上编译并运行
- 编写 Qt 程序，学习 Qt 程序设计方法及信号与插槽的使用。

4. 实验原理

4.1 Qt Designer 简介

Qt 提供了非常强大的 GUI 编辑工具— Qt Designer，它的操作界面类似于 Windows 下的 Visual Studio，而且它还提供了相当多的部件资源。

Qt 允许程序员不通过任何设计工具，以纯粹的 C++代码来设计一个程序。但是更多的程序员更加习惯于在一个可视化的环境中来设计程序，尤其是在界面设计的时候。这是因为这种设计方式更加符合人类思考的习惯，也比书写代码要快速的多。如图 4.1 所示：

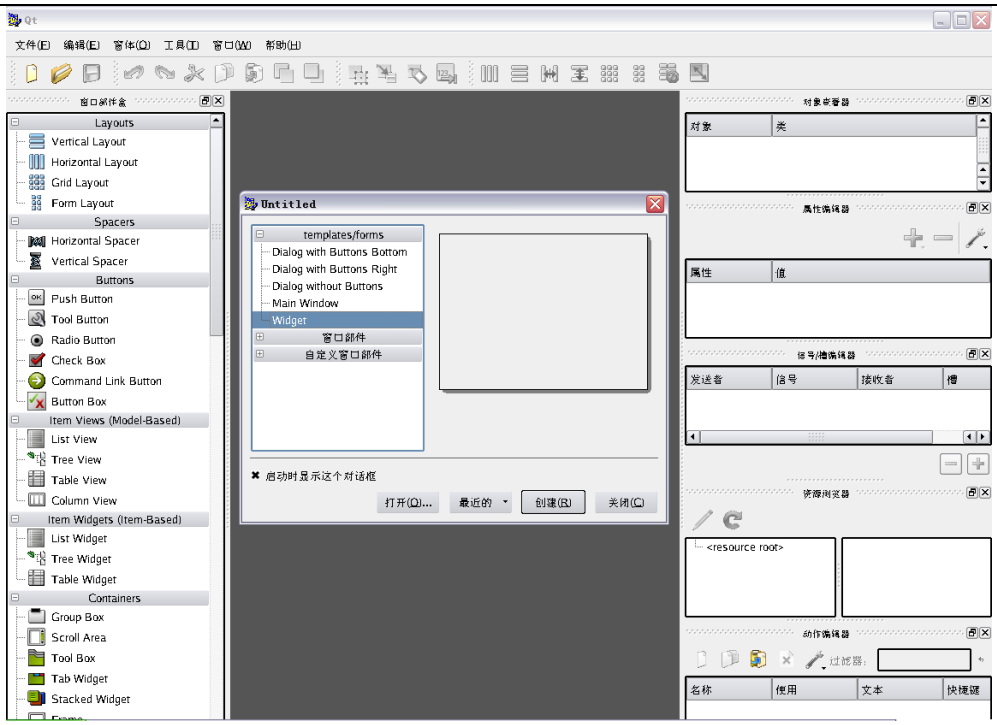


图 4.1 Qt Designer

因此 Qt 也提供了这样一个可视化的界面设计工具：Qt 设计器（Qt Designer）。其开始界面如上图所示。Qt 设计器可以用来开发一个应用程序全部或者部分的界面组件。以 Qt 设计器生成的界面组件最终被变成 C++代码，因此 Qt 设计器可以被用在一个传统的工具链中，并且它是编译器无关的。

默认情况下，Qt Designer 的用户界面是由几个顶级的窗口共同组成的。如果你更习惯于一个 MDI-style 的界面（由一个顶级窗口和几个子窗口组成的界面），可以在菜单 Edit->User Interface Mode 中选择 Docked Window 来切换界面。上图显示的就是 MDI-style 的界面风格。

4.2 Qt Designer 的一般设计方法

不管我们是使用 Qt Designer 还是编程来实现一个对话框界面，都包括以下相同的步骤：

1）创建并初始化子窗口部件。

2）将子窗口部件放置到布局当中。

3）对 Tab 的顺序进行设置。

4）放置信号和槽的连接。

5）完成对话框的通用槽的功能。

具体使用 Qt Designer 设计 QT 程序的过程如下：

1）使用 designer

- 设计界面，添加窗口组件

- 建立信号槽连接
- 编写事件处理函数
- 保存工程为.ui 文件,得到一个主窗口类

2）编写 main.cpp 文件进行主窗口类的实例化及显示

3）使用 qmake 生成.pro 工程文件

4）通过 qmake 自动生成 Makefile 文件

5）make 生成可执行文件

6）运行

5. 实验步骤

♦ 建立本机 Qt 实验环境目录

1）进入宿主机中系统中，建立 QtDemo 实验目录，例如手动在宿主机端用户目录下建立 cbt 目录。在其中创建目录 QtDemo 目录。

```
cbt@Cyb-Bot: ~$ cd
cbt@Cyb-Bot: ~$ mkdir cbt
cbt@Cyb-Bot: ~$ cd cbt/
cbt@Cyb-Bot: ~/cbt$ mkdir QtDemo
cbt@Cyb-Bot: ~/cb$ cd QtDemo
cbt@Cyb-Bot: ~/cbt/QtDemo$ ls
cbt@Cyb-Bot: ~/cbt/QtDemo$
```

后续实验环境都建在此目录(/home/cbt/cbt/QtDemo)下进行。

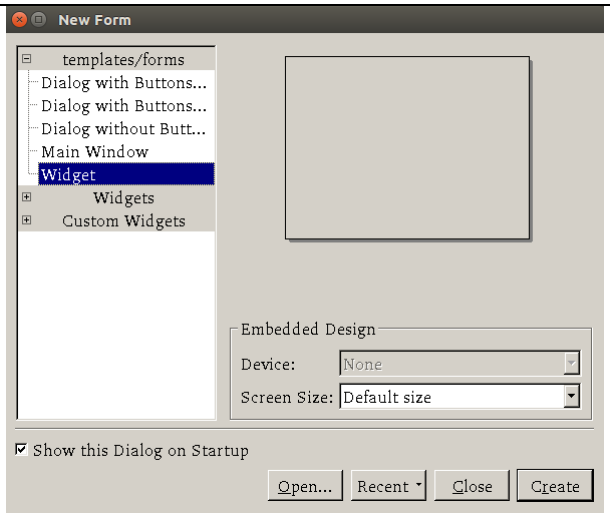
2）启动 Qt Designer，注意使用前面章节实验编译生成的环境中的 Qt Designer 工具。

```
cbt@Cyb-Bot: ~/cbt/QtDemo$ /usr/local/Trolltech/Qt-4.7.0/bin/designer
```

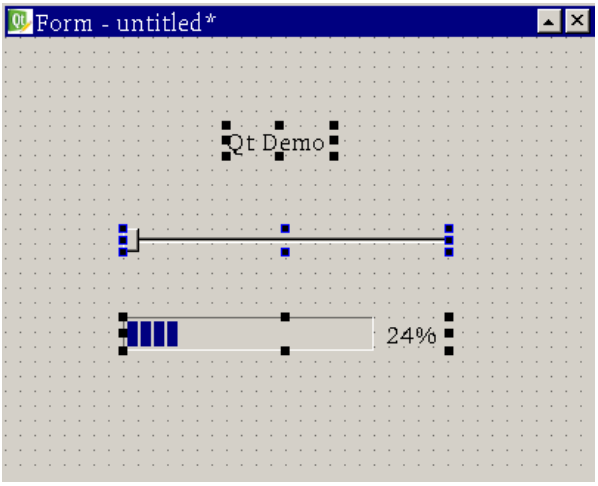
该命令使用的是绝对路径，以确保使用的是实验环境配套的工具。

3）编辑设置界面。

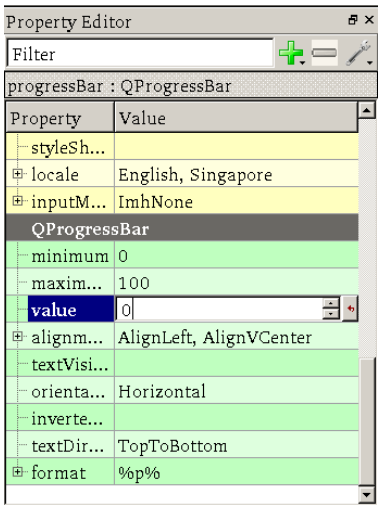
启动 designer 界面后，选择一个窗口布局 Widget 点击->Create，创建工程，如图：



拖拽几个简单的控件(label 显示标签、progressBar 进度条、horizontalSlider 水平滑动器)进行界面设计。界面布局如图：



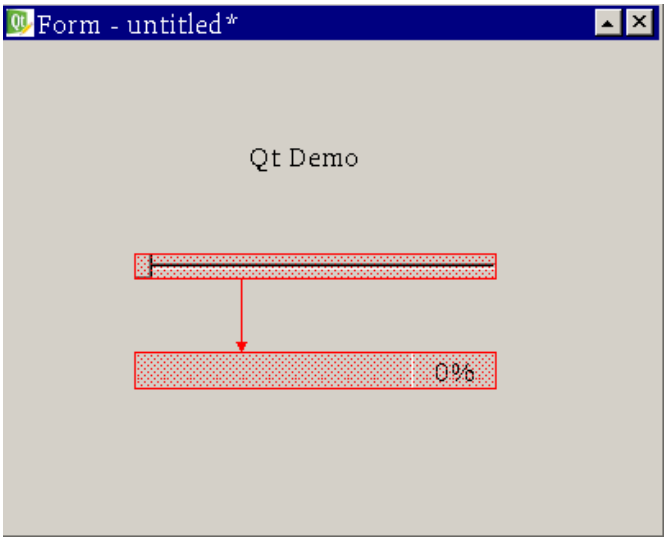
在右下角，属性编辑器中，设计标签显示内容为“QtDemo”,滚动条初始值 value 为“0”，水平滑动器初始值 value 为“0”。这样做可以保证水平滑动器滑动时候与滚动条同步。



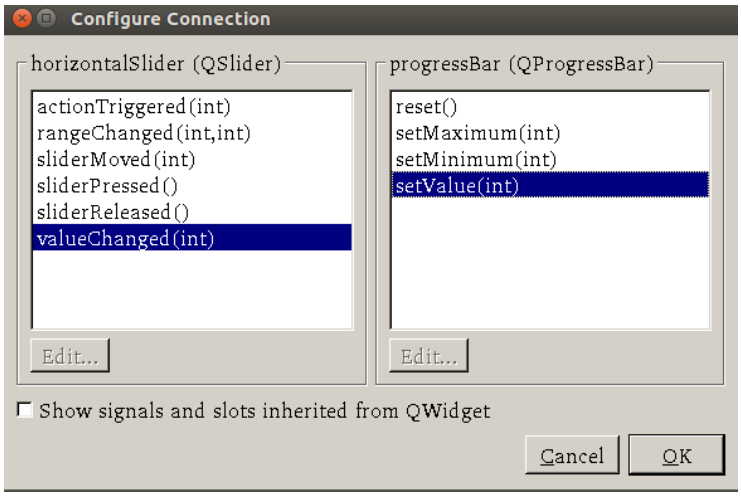
建立信号与槽连接

在 designer 的编辑菜单中选择“Edit Signals/Slots”，这样进入到信号和槽编辑模式，将

鼠标光标放置在水平滑动器上，按住鼠标左键拖拽出红色箭头，指向滚动条，即可建立起两个控件的信号与槽连接。

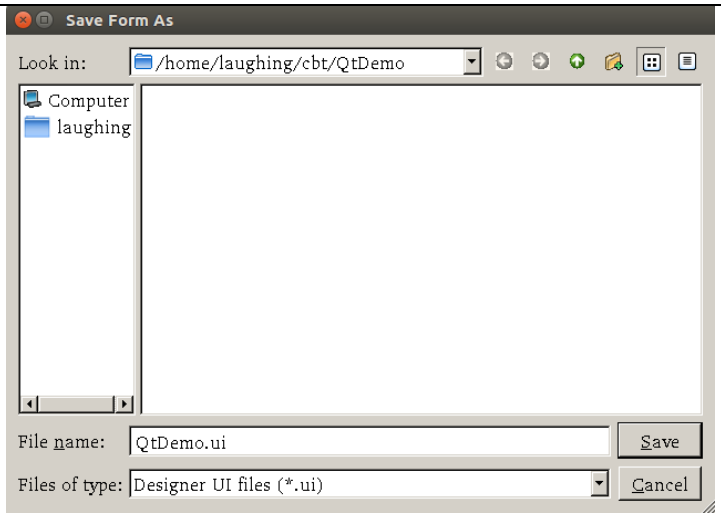


在弹出的信号与槽编辑对话框中，选择信号为“valueChanged(int)”，插槽为“setValue(int)”。注意信号与插槽都是带有相同类型参数的。



保存 UI 界面工程

退出信号与槽编辑模式，进入编辑窗口部件模式，保存当前 UI 工程，名称为“QtDemo.ui”。



备注：可以通过 Qt Designer 工具栏上的快捷方式  在不同设计模式间切换。在选择信号槽连接界面时候，左下角有个显示继承的选项要选中，才会显示 Designer 全部信号与槽。如图

4）编写 main.cpp C++主函数

```
cbt@Cyb-Bot: ~/cbt/QtDemo $ vi main.cpp
```

内容如下：

```
#include "ui_QtDemo.h"
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QWidget *widget = new QWidget;
    Ui::Form ui;
    ui.setupUi(widget);

    widget->show();
    return app.exec();
}
```

其中，包含的头文件为提前包含的，命名方式为 ui_xxx.h，xxx 代表 UI 保存的工程名字，例如本例中的 ui_QtDemo.h，此方法为 Qt 的编译技巧，限于基于 Designer 的设计方式使用。

```
cbt@Cyb-Bot: ~/cbt/QtDemo $ ls
main.cpp    QtDemo.ui
cbt@Cyb-Bot: ~/cbt/QtDemo $
```

5）使用 qmake –project 命令编译程序生成工程文件.pro

```
cbt@Cyb-Bot: ~/cbt/QtDemo$ /usr/local/Trolltech/Qt-4.7.0/bin/qmake -project
cbt@Cyb-Bot: ~/cbt/QtDemo$ ls
main.cpp    QtDemo.pro    QtDemo.ui
cbt@Cyb-Bot: ~/cbt/QtDemo$
```

6）使用 qmake 命令生成 Makefile 文件

```
cbt@Cyb-Bot: ~/cbt/QtDemo$ /usr/local/Trolltech/Qt-4.7.0/bin/qmake
cbt@Cyb-Bot: QtDemo$ ls
main.cpp  Makefile  QtDemo.pro  QtDemo.ui
cbt@Cyb-Bot: ~/cbt/QtDemo$
```

7）编译工程

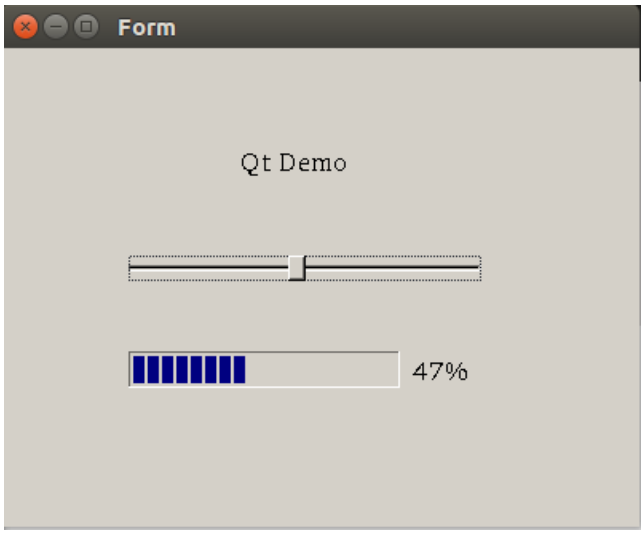
```
cbt@Cyb-Bot: ~/cbt/QtDemo$ make
/usr/local/Trolltech/Qt-4.7.0/bin/uic QtDemo.ui -o ui_QtDemo.h
g++ -c -pipe -O2 -Wall -W -D_REENTRANT -DQT_NO_DEBUG -DQT_GUI_LIB -DQT_CORE_LIB -DQT_SHARED -I/usr/local/Trolltech/Qt-4.7.0/mkspecs/linux-g++ -I. -I/usr/local/Trolltech/Qt-4.7.0/include/QtCore -I/usr/local/Trolltech/Qt-4.7.0/include/QtGui -I/usr/local/Trolltech/Qt-4.7.0/include -I. -I. -I. -o main.o main.cpp
g++ -Wl,-O1 -Wl,-rpath,/usr/local/Trolltech/Qt-4.7.0/lib -o QtDemo main.o -L/usr/local/Trolltech/Qt-4.7.0/lib -lQtGui -L/usr/local/Trolltech/Qt-4.7.0/lib -L/usr/X11R6/lib -lQtCore -lpthread
cbt@Cyb-Bot: QtDemo$ ls
main.cpp  main.o  Makefile  QtDemo  QtDemo.pro  QtDemo.ui  ui_QtDemo.h
cbt@Cyb-Bot: QtDemo$
```

编译成功后即可在当前目录下生成 Qt 本机的可执行程序 QtDemo。

8）运行 Qt 程序

```
cbt@Cyb-Bot: ~/cbt/QtDemo$ ./QtDemo
```

9）程序界面效果



可以使用鼠标拖拽水平滑动条，观察滚动条的滚动效果。

本例只是基于 QtDesigner 以后的控件及信号与槽来完成界面设计工作，如果用户设计的界面需要自定义的控件及信号与槽，则需要在 Qt 的源码中通过 C++的继承与派生的方法来实现新的类，添加自己的成员即可实现。该范畴属于 C++语言内容，此处不再赘述。有兴趣的读者可以参考相关书籍或网络资源独立完成。

实验三. 搭建 Qt/Embedded ARM 环境

1. 实验目的

- 学会 Qt/E 在 ARM 设备上的移植方法与步骤。
- 掌握在嵌入式 Linux 系统中运行 Qt/E 程序的方法。

2. 实验环境

- 硬件：ICS-IOT-CEP 实验平台，PC 机 Pentium 500 以上，硬盘 40G 以上，内存大于 256M
- 软件：Vmware Workstation +ubuntu + MiniCom/超级终端 +ARM-LINUX 交叉编译开发环境
- 实验目录：qt-everywhere-opensource-src-4.7.0/examples/widgets/analogclock

3. 实验内容

- 搭建 Qt/E for ARM 环境，移植 Qt/E 到 ICS-IOT-CEP 系统的设备上。
- 编写 Qt/E 程序，使之能够运行在嵌入式 Linux 系统中。

4. 实验原理

4.1 Qt/Embedded 简介

Qt/Embedded(简称 Qt/E)是一个专门为嵌入式系统设计图形用户界面的工具包。Qt 为各种系统提供图形用户界面的工具包，Qt/E 就是 Qt 的嵌入式版本。

使用 Qt/E，开发者可以：

- 用 Qt/E 开发的应用程序要移植到不同平台时，只需要重新编译代码，而不需要对代码进行修改。
- 可以随意设置程序界面的外观。
- 可以方便地为程序连接数据库。
- 可以使程序本地化。
- 可以将程序与 Java 集成。
- 嵌入式系统地要求是小而快速，而 Qt/E 就能帮助开发者为满足这些要求开发强壮地应用程序。
- Qt/E 是模块化和可裁剪地。开发者可以选取他所需要的一些特性，而裁剪掉所不需要的。这样，通过选择所需要的特性，Qt/E 的映像变得很小，最小只有 600K 左

右。

- 同 Qt 一样，Qt/E 也是用 C++写的，虽然这样会增加系统资源消耗，但是却为开发者提供了清洗的程序框架，使开发者能够迅速上手，并且能够方便地编写自定义的用户界面程序。

由于 QtE/是作为一种产品推出，所以它有很好的开发团体和技术支持，这对于使用 QtE/的开发者来说，方便开发过程，并增加了产品的可靠性。

4.2 Qt/Embedded 的特征

总的来说，Qt/E 拥有下面一些特征：

- 拥有同 Qt 一样的 API；开发者只需要了解 Qt 的 API，不用关心程序所用到的系统与平台
- 它的结构很好地优化了内存和资源地利用。
- 拥有自己的窗口系统：Qt/E 不需要一些子图形系统。它可以直接对底层的图形驱动进行操作。
- 模块化：开发者可以根据需要自己定制所需要的模块。
- 代码公开以及拥有十分详细的技术文档帮助开发者。
- 强大的开发工具。
- 与硬件平台无关：Qt/E 可以应用在所有主流平台和 CPU 上。支持所有主流的嵌入式 Linux，对于在 Linux 上的 Qt/E 的基本要求只不过是 Frame Buffer 设备和一个 C++编译器(如 gcc)。Qt/E 同时也支持很多实时的嵌入式系统，如 QNX 和 WindowsCE。
- 提供压缩字体格式：即使在很小的内存中，也可以提供一流的字体支持。
- 支持多种的硬件和软件的输入。
- 支持 Unicode，可以轻松地使程序支持多种语言。
- 支持反锯齿文本和 Alpha 混合的图片。

Qt/E 虽然公开代码和技术文档，但是它不是免费的，当开发者的商业化产品需要用到他的运行库时，必须向 Trolltech 公司支持 license 费用(每套 3 美金)，如果开发的东西不用于商业用途则不需要付费。

Qt/E 由于平台无关性和提供了很好的 Gui 编程接口，在许多嵌入式系统中得到了广泛的应用，是一个成功的嵌入式 GUI 产品。

4.3 Qt/Embedded 与 Qt/X11

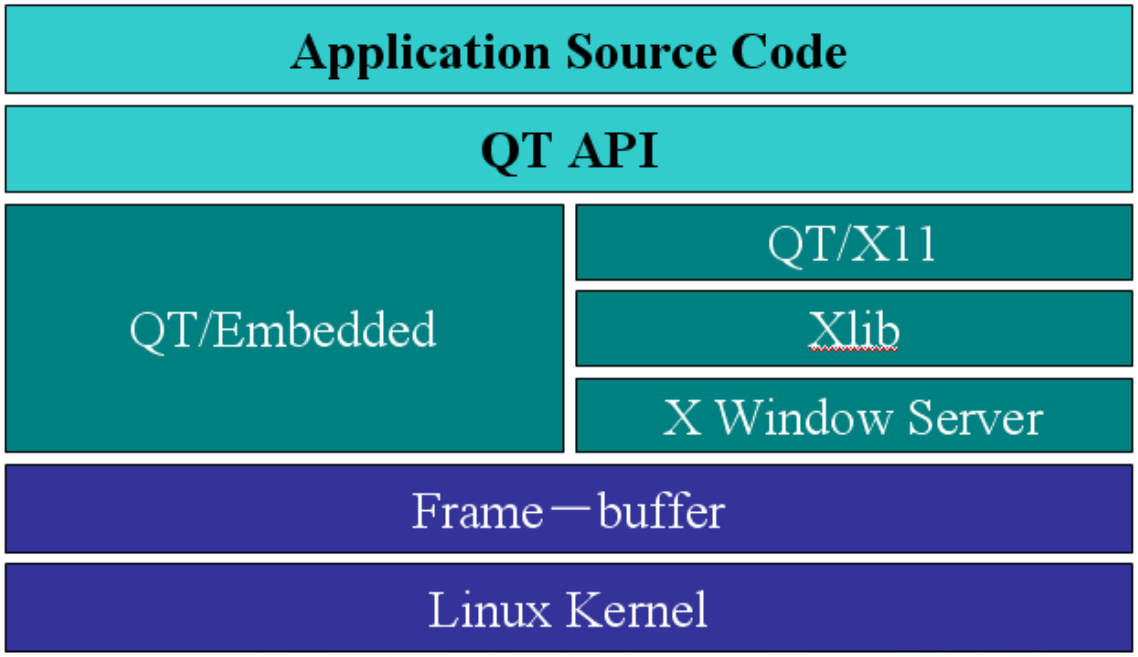


图 4.3 Qt/E 与 Qt/X11 比较

- Qt/Embedded 通过 Qt API 与 Linux I/O 设施直接交互，成为嵌入式 linux 端口。同 Qt/X11 相比，Qt/Embedded 很节省内存，其不需要一个 X 服务器或是 Xlib 库，它在底层摈弃了 Xlib，采用 framebuffer（帧缓存）作为底层图形接口。同时，将外部输入设备抽象为 keyboard 和 mouse 输入事件。
- Qt/Embedded 的应用程序可以直接写内核缓冲帧，这可避免开发者使用繁琐的 Xlib/Server 系统。

5. 实验步骤

♦ 建立 Qt/E for ARM 实验环境目录

1）进入宿主机中系统中，建立 Qt4arm-4.7.0 实验目录，例如手动在宿主机端用户目录下建立 cbt 目录。在其中创建目录 Qt4 及子目录 Qt4arm-4.7.0。所有 QT 相关实验都放在该目录下完成，后面文章不在赘述。

```
cbt@Cyb-Bot: ~$ cd
cbt@Cyb-Bot: ~$ mkdir cbt
cbt@Cyb-Bot: ~$ cd cbt/
cbt@Cyb-Bot: ~/cbt$ mkdir Qt4/
cbt@Cyb-Bot: ~/cbt $ cd Qt4/
cbt@Cyb-Bot: ~/cbt/Qt4$ mkdir Qt4arm-4.7.0/
cbt@Cyb-Bot: ~/cbt/Qt4$ cd Qt4arm-4.7.0/
cbt@Cyb-Bot: ~/cbt/Qt4/Qt4arm-4.7.0$
```

后续所有关于 Qt/E 的实验环境都建在此目录(/home/cbt/cbt/Qt4/Qt4arm-4.7.0)下进行。

2）拷贝并解压 Qt 源码包

```
cbt@Cyb-Bot: ~/cbt/Qt4/Qt4arm-4.7.0$ cp /CBT-6818/SRC/gui/qt-everywhere-opensource-src-4.7.0.tar.gz ./
cbt@Cyb-Bot: ~/cbt/Qt4/Qt4arm-4.7.0$ tar xzvf qt-everywhere-opensource-src-4.7.0.tar.gz
```

解压后会在当前目录下生成解压后的 Qt 库源码目录 qt-everywhere-opensource-src-4.7.0。

3）编译配置 Qt/E 环境

进入 qt-everywhere-opensource-src-4.7.0 源码包目录，执行 configure 命令，配置 Qt/E 库环境。

```
cbt@Cyb-Bot: ~/cbt/Qt4/Qt4arm-4.7.0$ cd qt-everywhere-opensource-src-4.7.0
cbt@Cyb-Bot: ~/cbt/Qt4/Qt4arm-4.7.0/qt-everywhere-opensource-src-4.7.0$ ./configure -opensource -embedded arm -xplatform qws/linux-arm-g++ -fast -nomake examples -nomake demos -no-webkit -qt-libtiff -qt-libmng -qt-mouse-tslib -qt-mouse-pc -no-mouse-linuxtp -no-neon
```

对于嵌入式 ARM 环境，configure 配置的时候命令行参数极为重要，请严格按照上述参数完成配置。

configure 的其他具体命令行参数配置用户可以通过 --help 命令查看：

```
cbt@Cyb-Bot: ~/cbt/Qt4/Qt4arm-4.7.0/qt-everywhere-opensource-src-4.7.0$ ./configure --help
```

默认指定的环境安装路径为/usr/local/Trolltech/QtEmbedded-4.7.0-arm，当然用户也可以通过命令行参数-prefix 来指定环境编译好后的安装路径，方便查找，使用编译生成的工具。

执行 configure 命令后，本机环境一般不用特使命令行参数即可，使用默认参数。当出现选择 Qt 版本许可的时候，依次输入“o”表示开源许可，再输入“yes”表示同意协议即可完成。

4）编译 Qt/E 环境。

完成上述 configure 配置后，即可输入 make 来编译该 Qt/E 环境.

```
cbt@Cyb-Bot: ~/cbt/Qt4/Qt4arm-4.7.0/qt-everywhere-opensource-src-4.7.0$ make
```

5）安装 Qt/E 环境。

上述编译过程成功后，可以执行 make install 命令来安装 Qt/E 环境，默认安装路径为/usr/local/Trolltech/QtEmbedded-4.7.0-arm，会在该目录下生成相应工具(如 qmake)和库文件等。

```
cbt@Cyb-Bot: ~/cbt/Qt4/Qt4arm-4.7.0/qt-everywhere-opensource-src-4.7.0$ sudo make install
```

意：一般情况下，Qt 库的编译需要较长时间，根据机器硬件性能，可能几个小时不等。且 Qt 环境的编译，依赖宿主机系统和 Qt 具体的版本，本实验文档仅供 ubuntu 宿主机环境和 qt4.7 的版本库，其他环境及 Qt 库版本如遇问题，请参阅网络资源来解决。

♦ 运行 Qt/E 环境自带例程

Qt 源码库的路径下自带了一系列的 examples 例程，方便用户进行学习和参考，我们可

以取其中的一些例程，编译后运行测试下前面搭建的 Qt/E 环境。

1）编译 Qt/E 例程应用程序

以 Qt 自带的计算器例程为例，可以先编译该程序。进入 Qt 源码目录 examples/widgets/analogclock 中，利用前面编译库环境生成的 qmake 工具编译该工程。

```
cbt@Cyb-Bot: ~/cbt/Qt4/Qt4arm-4.7.0/qt-everywhere-opensource-src-4.7.0$ cd examples/widgets/analogclock/
cbt@Cyb-Bot: ~/cbt/Qt4/Qt4arm-4.7.0/qt-everywhere-opensource-src-4.7.0/examples/widgets/analogclock$ ls
analogclock.cpp  analogclock.h  analogclock.pro  main.cpp
cbt@Cyb-Bot: analogclock$
```

先生成 Makefile 因该工程中已经生成 pro 工程文件，因此无需再使用 qmake -project 命令了。

```
cbt@Cyb-Bot: ~/cbt/Qt4/Qt4arm-4.7.0/qt-everywhere-opensource-src-4.7.0/examples/widgets/analogclock
$ /usr/local/Trolltech/QtEmbedded-4.7.0-arm/bin/qmake
cbt@Cyb-Bot: ~/cbt/Qt4/Qt4arm-4.7.0/qt-everywhere-opensource-src-4.7.0/examples/widgets/analogclock
$ ls
analogclock.cpp  analogclock.h  analogclock.pro  main.cpp  Makefile
```

编译该工程。

```
cbt@Cyb-Bot: ~/cbt/Qt4/Qt4arm-4.7.0/qt-everywhere-opensource-src-4.7.0/examples/widgets/analogclock
$ make
cbt@Cyb-Bot: ~/cbt/Qt4/Qt4arm-4.7.0/qt-everywhere-opensource-src-4.7.0/examples/widgets/analogclock
$ ls
analogclock  analogclock.cpp  analogclock.h  analogclock.pro  main.cpp  Makefile
cbt@Cyb-Bot: ~/cbt/Qt4/Qt4arm-4.7.0/qt-everywhere-opensource-src-4.7.0/examples/widgets/analogclock
$
```

编译成功后，会在当前目录下生成 Qt/E 可执行文件 analogclock。

注意：该可执行程序，只能有 ARM 处理器执行，不能再本机运行。

1）运行 Qt/E ARM 应用程序

创建宿主机端 Qt/E 的 NFS 共享目录

在前几个章节的实验的 NFS 共享目录/CBT-6818/下建立 Trolltech 目录，后续的 Qt/Embedded 实验都是在此目录下共享到 ARM 设备端执行 Qt/E 程序的。因此我们需要在该目录下搭建好 Qt/E 的环境，也就是拷贝前面编译好的 Qt/E 动态库以及应用程序到此目录下。

```
root@localhost qt-everywhere-opensource-src-4.7.0$ cd / CBT-6818/
cbt@Cyb-Bot: /CBT-6818$ sudo mkdir Trolltech
cbt@Cyb-Bot: /CBT-6818$ cd Trolltech/
cbt@Cyb-Bot: /CBT-6818/Trolltech$
```

拷贝 Qt/E 库及插件等资源到 NFS 共享目录。

```
cbt@Cyb-Bot: /CBT-6818/Trolltech$ sudo cp /usr/local/Trolltech/QtEmbedded-4.7.0-arm/ ./ -a
cbt@Cyb-Bot: /CBT-6818/Trolltech$ cd QtEmbedded-4.7.0-arm
cbt@Cyb-Bot: QtEmbedded-4.7.0-arm$ ls
bin   imports   include   lib   mkspecs   plugins   translations
cbt@Cyb-Bot: /CBT-6818/Trolltech/QtEmbedded-4.7.0-arm$
```

拷贝应用程序到 NFS 共享目录。

```
cbt@Cyb-Bot: /CBT-6818/Trolltech/QtEmbedded-4.7.0-arm $ sudo cp /home/cbt/cbt/Qt4/Qt4arm-4.7.0/qt-
everywhere-opensource-src-4.7.0/examples/widgets/analogclock/analogclock ./
cbt@Cyb-Bot: QtEmbedded-4.7.0-arm$ ls
bin   imports   include   lib   mkspecs   plugins   translations analogclock
```

ARM 端挂载 NFS 共享目录。

启动全功能物联网实验系统，连好网线、串口线。通过串口终端挂载宿主机实验目录。

```
[root@CBT-6818:~] mount -t nfs -o nolock 192.168.1.7:/CBT-6818      /mnt/
```

设置环境变量（ARM 端）

进入 QT/Embedded 实验 NFS 共享目录 Trolltech 下的 QtEmbedded-4.7.0-arm 目录中。

```
[root@CBT-6818:~]#cd /mnt/Trolltech/QtEmbedded-4.7.0-arm/
[root@CBT-6818:/mnt/Trolltech/QtEmbedded-4.7.0-arm]# ls
analogclock     imports          lib              plugins
bin             include          mkspecs          translations
[root@CBT-6818:/mnt/Trolltech/QtEmbedded-4.7.0-arm]#
```

根据此实验目录下的目录进行 ARM 端环境变量设置，如下：

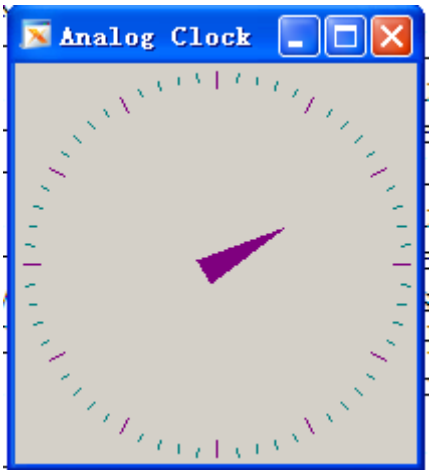
```
[root@CBT-6818:/mnt/Trolltech/QtEmbedded-4.7.0-arm]# export TSLIB_TSDEVICE=/dev/input/event1
[root@CBT-6818:/mnt/Trolltech/QtEmbedded-4.7.0-arm]# export TSLIB_CONFFILE=/etc/ts.conf
[root@CBT-6818:/mnt/Trolltech/QtEmbedded-4.7.0-arm]# export POINTERCAL_FILE=/etc/pointercal
[root@CBT-6818:/mnt/Trolltech/QtEmbedded-4.7.0-arm]# QWS_MOUSE_PROTO=tslib:/dev/input/event1
[root@CBT-6818:/mnt/Trolltech/QtEmbedded-4.7.0-arm]# export TSLIB_CONSOLEDEVICE=none
```

执行 Qt/E 程序(在 ARM 系统中的 NFS Qt/E 目录下)：

```
[root@CBT-6818:/mnt/Trolltech/QtEmbedded-4.7.0-arm]# ./analogclock -qws
```

如果第一次使用触摸屏，需要进行校准，请在终端输入 ts_calibrate 命令完成屏幕校准即可。

2）运行 Qt/E 效果图



注意：由于平台出厂时候系统文件系统中已经含有相同环境及版本的 Qt/E 动态库及相关资源，因此本实验运行的例程，实际使用的是 ARM 系统中的动态库文件来运行的。当然，用户如果熟悉 Qt/E 的运行环境，同样也可以将前面章节编译生成的 Qt/E 库和资源下载到 ARM 系统的相关目录中实验。

第四章. 底层系统构建实验

嵌入式 Linux 系统主要由 bootloader、内核(设备驱动程序)、根文件系统以及应用程序等几部分构成，相对应用程序而言，上述其它几个组成部分对学习和开发者要求更加严格，需要对目标硬件接口有深入的了解和掌握，才能移植和开发底层的软件工作。本章所涉猎的内容，可以作为高级进阶实验来进行学习和参考。

本章主要介绍了嵌入式 Linux 系统的构建过程，其中以 Cortex-A53 处理器平台上运行的 Linux 内核与根文件系统为模版，首先着重讲述了嵌入式系统中 Linux 内核的定制、裁剪、配置和编译过程，以及如何向内核中加入自定义功能模块。其次介绍了如何移植嵌入式根文件系统的过程，以及常用的系统命令和服务如何添加到文件系统中。用户通过学习本章的内容，能够对嵌入式 Linux 系统的构建结构有一个清晰的认识，并能掌握基本的嵌入式 Linux 系统搭建过程。

实验一. Linux 内核裁剪与编译

1. 实验目的

- 了解 Linux 内核相关知识与内核结构。
- 了解 Linux 内核在 ARM 设备上移植的基本步骤和方法。
- 掌握 Linux 内核裁剪与定制的基本方法。

2. 实验环境

- 硬件：A53 智能网关实验平台，PC 机 Pentium 500 以上，硬盘 40G 以上，内存大于 256M。
- 软件：Vmware Workstation +ubuntu + MiniCom/超级终端 +ARM-LINUX 交叉编译开发环境。
- 实验目录：/CBT-6818/SRC/linux。

3. 实验内容

- 分析 Linux 内核的基本结构，了解 Linux 内核在 ARM 设备上移植的一些基本步骤及常识。
- 学习 Linux 内核裁剪定制的基本配置方法，利用 ICS-IOT-CEP 全功能物联网教学科研平台 ARM 移动终端部分设备配套 Linux 内核进行自定义功能(如 helloworld 显示)的添加。并重新编译内核源码，生成内核压缩文件 uImage，下载到设备中测试。

4. 实验原理

4.1 Linux 内核相关

♦ **什么是 Linux 内核?**

在 IT 术语中，内核既是操作系统的心脏，也是它的大脑，因为内核控制着基本的硬件。内核是操作系统的核心，具有很多最基本功能，如虚拟内存、多任务、共享库、需求加载、共享的写时拷贝(copy-on-write)可执行程序和 TCP/IP 网络功能。如图 4.1.1 所示：

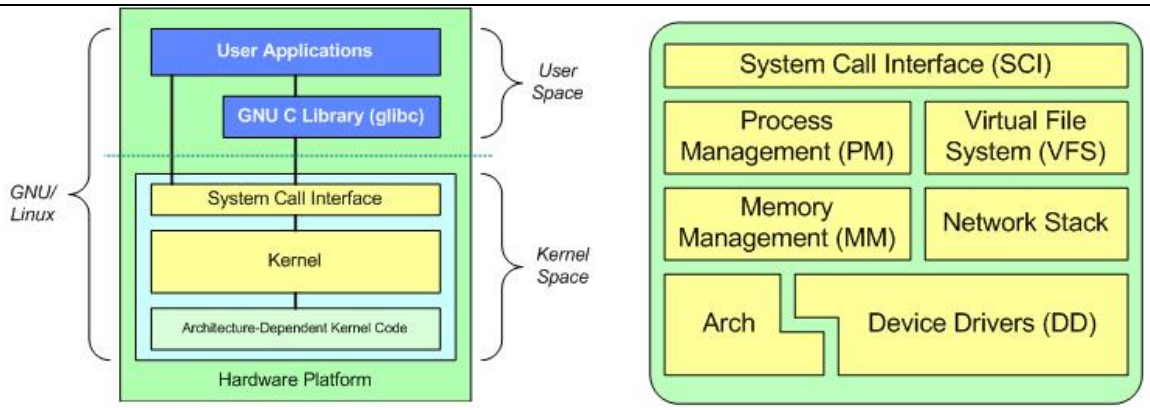


图 4.1.1 Linux 系统结构框图

最上面是用户（或应用程序）空间。这是用户应用程序执行的地方。用户空间之下是内核空间，Linux 内核正是位于这里。

Linux 内核的起源可追溯到 1991 年芬兰大学生 Linus Torvalds 编写和第一次公布 Linux 的日子。尽管到目前为止 Linux 系统早已远远发展到了 Torvalds 本人之外的范围，但 Torvalds 仍保持着对 Linux 内核的控制权，并且是 Linux 名称的唯一版权所有人。自发布 Linux 0.12 版起，Linux 就一直依照 GPL(通用公共许可协议)自由软件许可协议进行授权。

Linux 内核本身并不是操作系统，它是一个完整操作系统的组成部分。Red Hat、Novell、Debian 和 Gentoo 等 Linux 发行商都采用 Linux 内核，然后加入更多的工具、库和应用程序来构建一个完整的操作系统。

♦ Linux 内核的发展

自 1991 年 11 月由芬兰的 Linus Ttorvalds 推出 Linux 0.1.0 版内核至今，Linux 内核已经升级到 Linux3.5（写本文档时，www.kernel.org 发布的最新版 Linux 内核）。其发展速度是如此的迅猛，是目前市场上唯一可以挑战 Windows 的操作系统。如图 4.1.2 所示

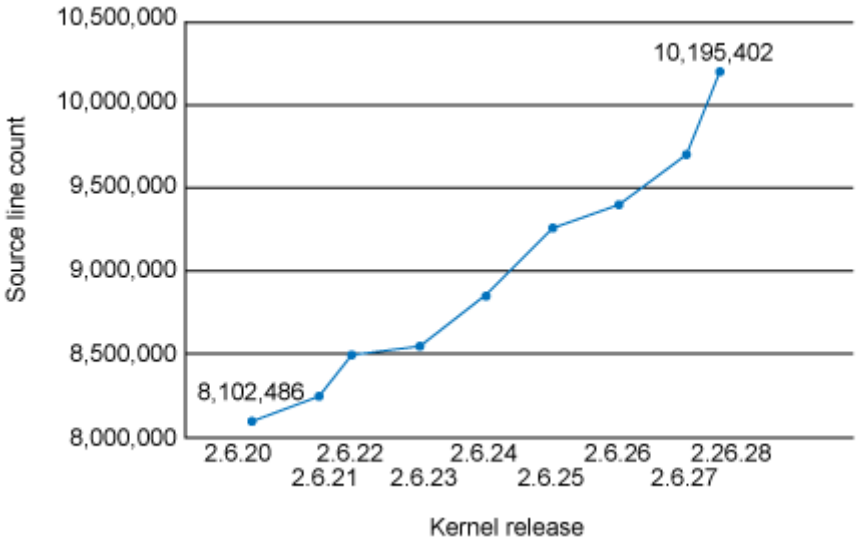


图 4.1.2 Linux 内核发展

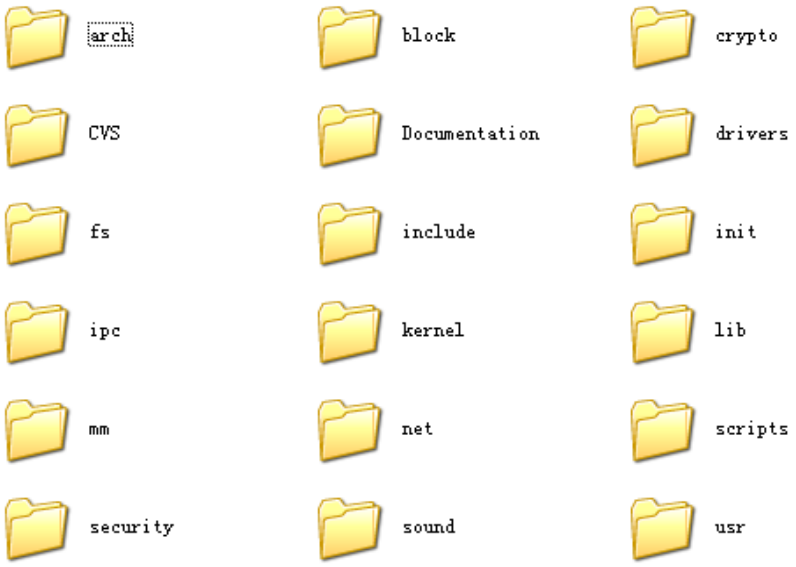
Linux 内核在其发展过程中得到分布于全世界的广大 OpenSource 项目追随者的大力支持。尤其是一些曾经参与 Unix 开发的人员，他们把应用于 Unix 上的许多应用程序移植到 Linux 上来，使得 Linux 的功能得到巨大的扩展。

目前比较稳定的版本是 Linux3.16.3。在 Linux 的版本号中，第一个数为主版本号。第二个为次版本号。第三个为修订号。次版本号为偶数表明是稳定发行版本，奇数则是在开发中的版本。

随着其功能不断加强，灵活多样的实现加上其可定制的特性以及开放源码的优势，Linux 在各个领域的应用正变得越来越广泛。目前 Linux 的应用正有舍去中间奔两头的趋势，即在 PC 机上 Linux 要真正取代 Windows，或许还有很长的路要走，但在服务器市场上它已经牢牢站稳脚跟。而随着嵌入式领域的兴起更是为 Linux 的长足发展提供了无限广阔的空间。目前专门针对嵌入式设备的 Linux 改版就有好几种。包括针对无 MMU 的 uClinx 和针对有 MMU 的标准 LINUX 在各个硬件体系结构的移植版本。基于像 S5P6818 这样的内核的 ARM－LINUX 使用了 MMU 的内存管理，对进程有保护，提高了嵌入式系统中多进程的保护能力。使用户应用程序的可靠性得以提高，降低了用户的开发难度。

4.2 内核目录结构

本实验平台运行的 Linux 内核版本为 linux-3.5，其源码目录结构如图：

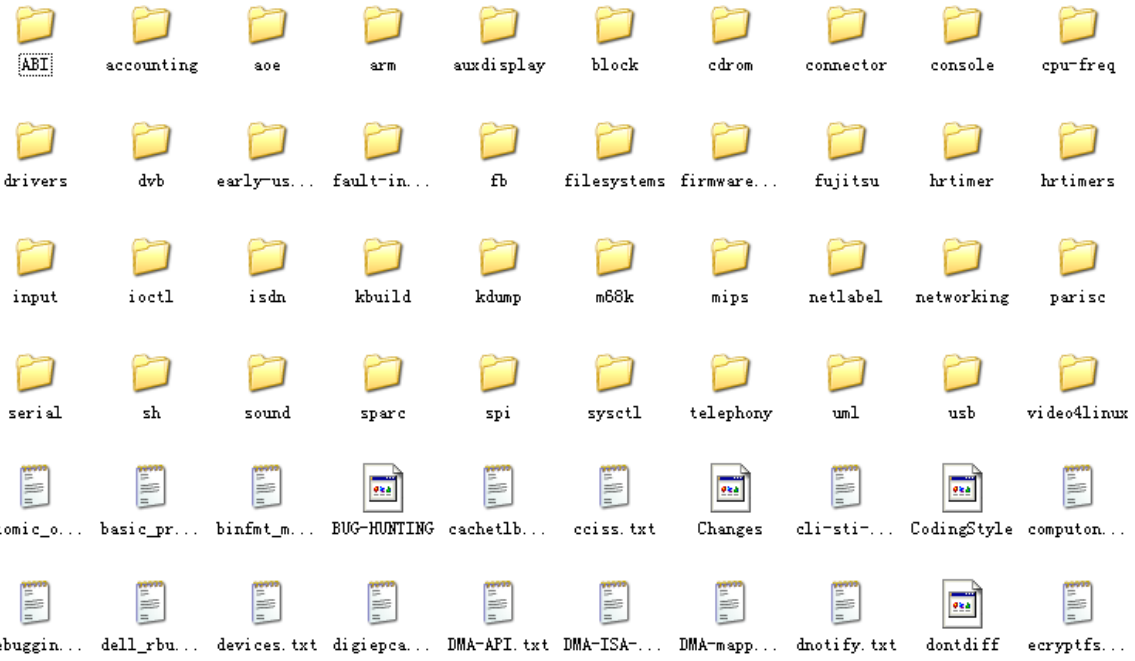


♦ **arch**

与体系结构相关的代码全部放在这里，如图所示，我们的实验设备中使用的是其中的 arm 目录。

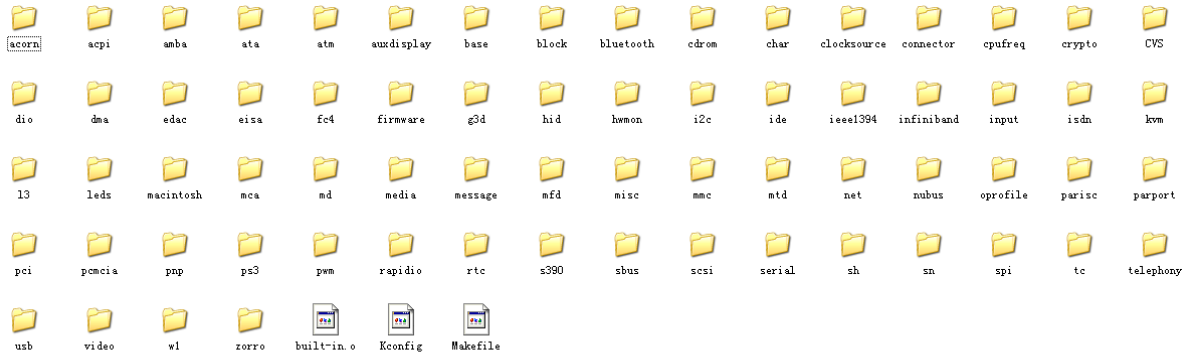
♦ **Documentation**

这里存放着内核的所有开发文档，如图所示，其中的文件会随版本的演变发生变化，通过阅读这里的文件是获得内核最新的开发资料的最好的地方。



♦ **Drivers**

此目录包括所有的驱动程序，如图所示，下面又建立了多个目录，分别存放各个分类的驱动程序源代码。下面的截图是 drivers 目录文件列表。



drivers 目录是内核中最大的源代码存放处，大约占整个内核的一多半。其中我们经常会用到的目录有：

♦ **Drivers/char**

字符设备是 drivers 目录中最为常用，也许是最为重要的目录，因为其中包含了大量与驱动程序无关的代码。通用的 tty 层在这里实现，console.c 定义了 linux 终端类型，vt.c 中定义了虚拟控制台；lp.c 中实现了一个通用的并口打印机的驱动，并保持设备无关性；kerboard.c 实现高级键盘处理，它导出 handle_scancode 函数，以便于其他与平台相关的键盘驱动使用。我们的大部分实验也是放在这个目录下。

♦ **Driver/block**

其中存放所有的块设备驱动程序，也保存了一些设备无关的代码。如 rd.c 实现了 RAM 磁盘，nbd.c 实现了网络块设备，loop.c 实现了回环块设备。

♦ **Drives/ide**

专门存放针对 IDE 设备的驱动。

♦ **Drivers/scsi**

存放 SCSI 设备的驱动程序，当前的 cd 刻录机、扫描仪、U 盘等设备都依赖这个 SCSI 的通用设备。

♦ **Drivers/net**

存放网络接口适配器的驱动程序，还包括一些线路规程的实现，但不实现实际的通信协议，这部分在顶层目录的 net 目录中实现。

♦ **Drivers/video**

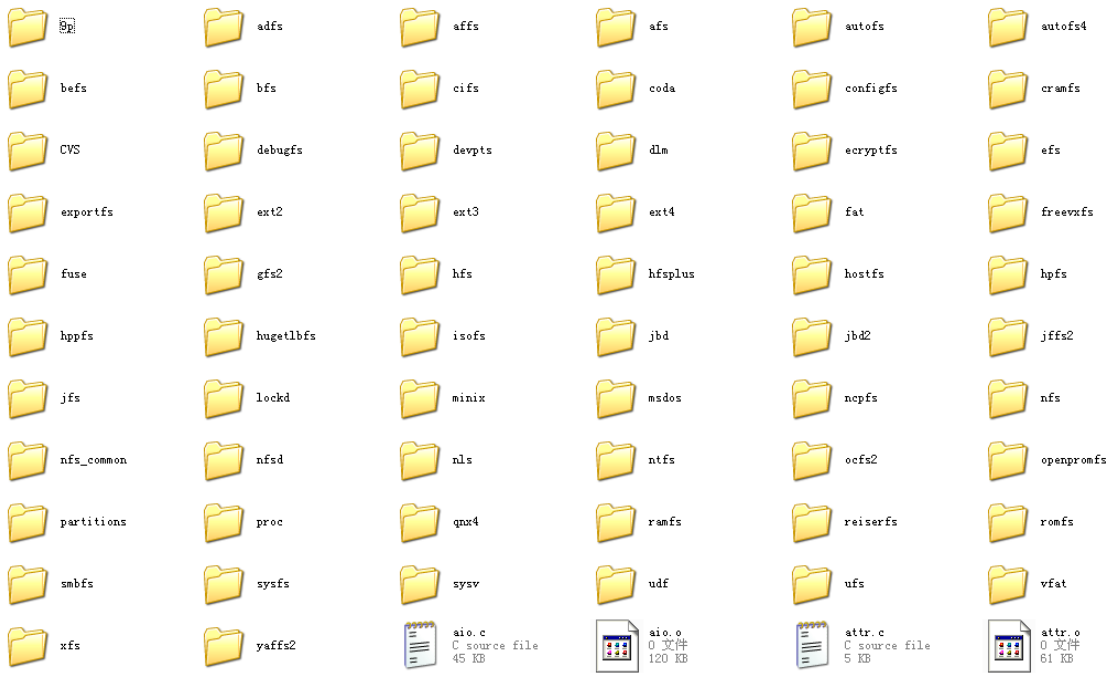
这里保存了所有的帧缓冲区视频设备的驱动程序，整个目录实现了一个单独的字符设备驱动。/dev/fb 设备的入口点在 fbmem.c 文件中，该文件注册主设备号并维护一个此设备的清单，其中记录了哪一个帧缓冲区设备负责哪个次设备号。

♦ **Drivers/media**

这里存放的代码主要是针对无线电和视频输入设备，比如目前流行的 usb 摄像头。

♦ **fs**

此目录下包括了大量的文件系统的源代码，如图所示，其中在嵌入式开发中要使用的包括：devfs、cramfs、ext2、,jffs2、romfs、yaffs、vfat、nfs、proc 等。

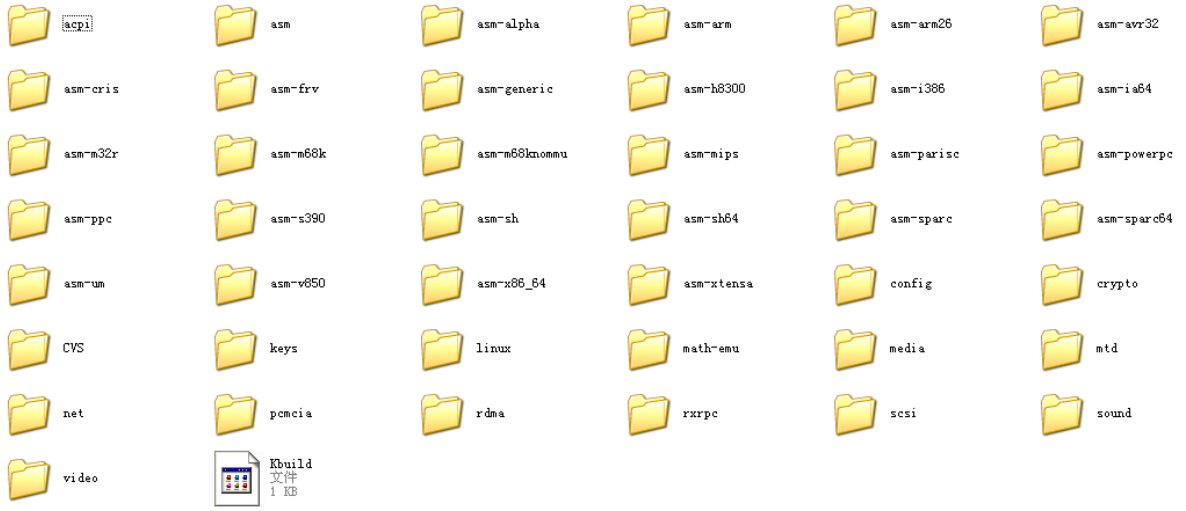


文件系统是 Linux 中非常重要的子系统，这里实现了许多重要的系统调用，比如 exec.c 文件中实现了 execve 系统调用；用于文件访问的系统调用在 open.c、read_write.c 等文件中定义，select.c 实现了 select 和 poll 系统调用，pipe.c 和 fifo.c 实现了管道和命名管道，mkdir、rmdir、rename、link、symlink、mknod 等系统调用在 namei.c 中实现。

文件系统的挂装和卸载和用于临时根文件系统的 initrd 在 super.c 中实现。Devices.c 中实现了字符设备和块设备驱动程序的注册函数；file.c、inode.c 实现了管理文件和索引节点内部数据结构的组织。Ioctl.c 实现 ioctl 系统调用。

♦ **include**

这里是内核的所有头文件存放的地方，其中的 linux 目录是头文件最多的地方，也是驱动程序经常要包含的目录。



♦ **init**

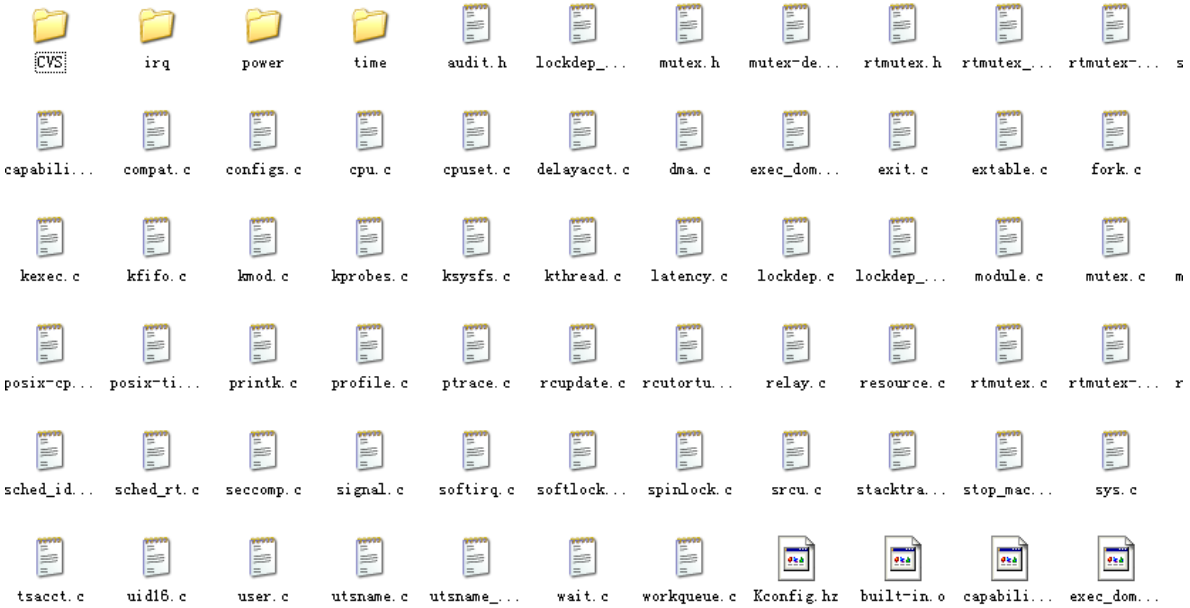
linux 的 main.c 程序，通过这个比较简单的程序，我们可以理解 LINUX 的启动流程。

♦ **ipc**

system V 的进程间通信的原语实现，包括信号量、共享内存。

♦ **kernel**

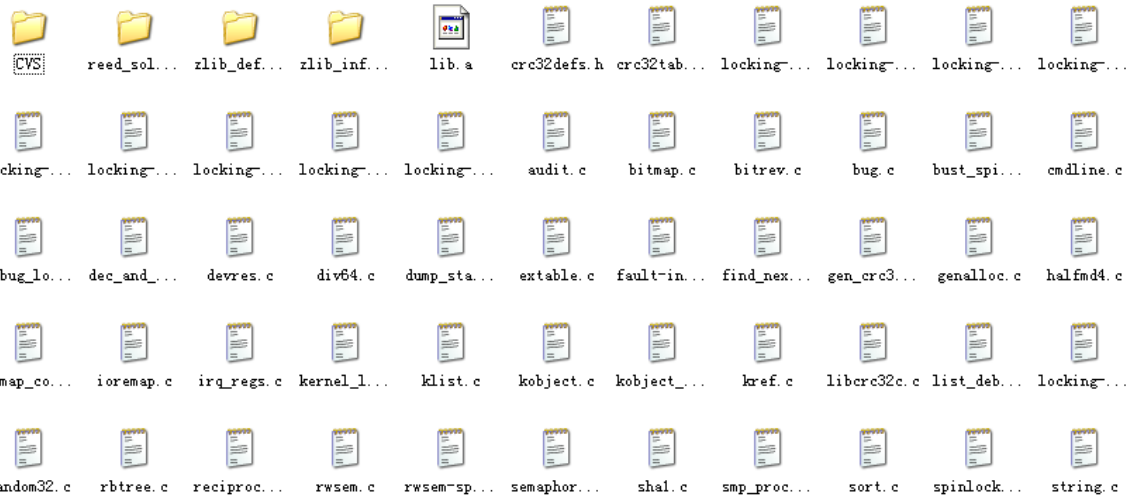
这个目录下存放的是除网络、文件系统、内存管理之外的所有其他基础设施，从下图的文件列表所示，我们大致可以看出，其中至少包括进程调度 sched.c，进程建立 fork.c，定时器的管理 timer.c，中断处理，信号处理等。



♦ **lib**

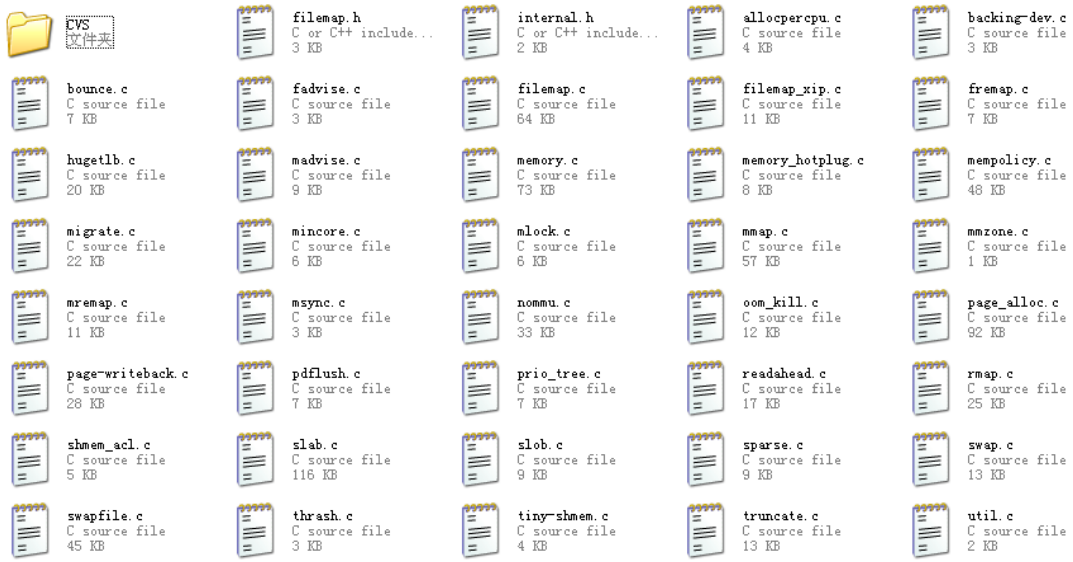
包括一些通用支持函数，类似于标准 C 的库函数。其中包括了最重要的 vsprintf 函数的实现，它是 printk 和 sprintf 函数的核心。还有将字符串转换为长整形数的 simple_atol 函

数。其文件列表如图所示。



♦ **mm**

这个目录包含实现内存管理的代码，包括所有与内存管理相关的数据结构，如图所示，其中我们在驱动中需要使用的 kmalloc 和 kfree 函数在 slab.c 中实现，mmap 定义在 mmap.c 中的 do_mmap_pgoff 函数。将文件映射到内存的实现在 filemap.c 中，mprotect 在 mprotect.c，remap 在 remap.c 中实现；vmscan.c 中实现了 kswapd 内核线程，它用于释放未使用和老化的页面到交换空间，这个文件对系统的性能起着关键的影响。



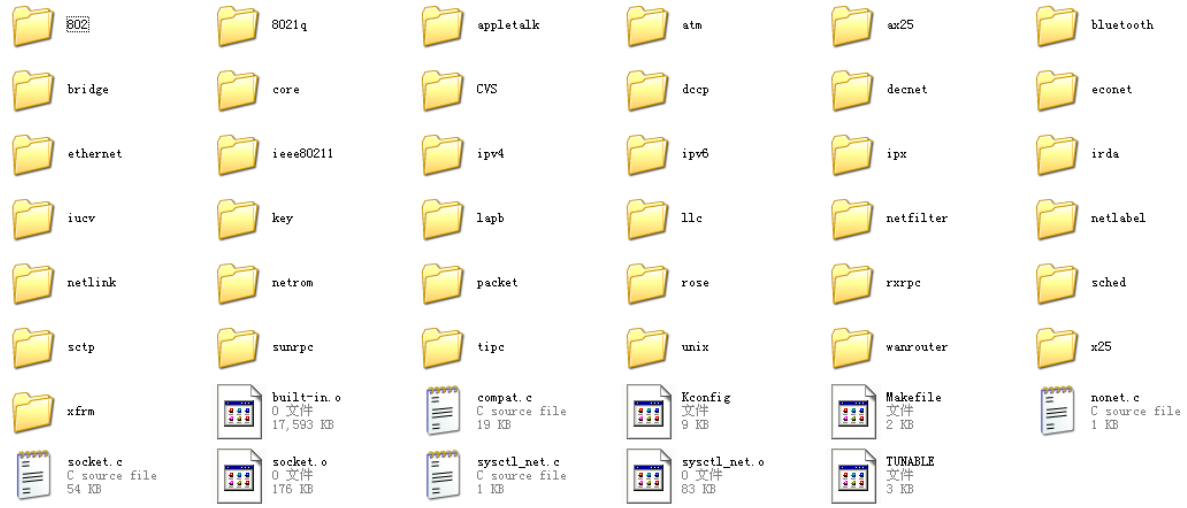
♦ **net**

这个目录包含了套接字抽象和网络协议的实现，每一种协议都建立了一个目录，我们可以看到有 26 个目录，但是其中的 core、bridge、ethernet、sunrpc、khttpd 不是网络协议。我们使用最多的是 ipv4、ipv6、802、ipx 等。Ipv4、ipv6 是 ip 协议的第 4 版本和第 6 版本。

Core 目录中实现了通用的网络功能：设备处理、防火墙、组播、别名等；

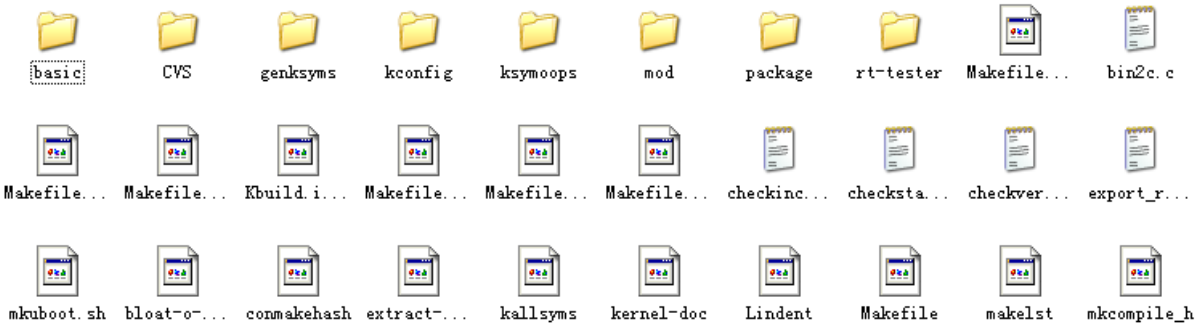
ethernet 和 bridge 实现特定的底层功能：以太网相关的辅助函数以及网桥功能。

Sunrpc 中提供了支持 NFS 服务器的函数。



♦ **script**

这个目录存放许多脚本，主要用于配置内核，其文件列表如图所示。



4.3 Linux 内核配置及裁剪

Linux 内核的裁剪与编译看上去是个挺简单的过程。只是对配置菜单的简单选择。但是内核配置菜单本身结构庞大，内容复杂。具体如何选择却难住了不少人。因此熟悉与了解该菜单的各项具体含义就显得比较重要。我们现在就对其作一些必要介绍：

Linux 内核的编译菜单有好几个版本，运行：

1）make config：进入命令行，可以一行一行的配置，这个方式不友好所以我们不具体介绍。

2）make menuconfig：进入我们熟悉的 menuconfig 菜单，相信很多人对此都不陌生。

3）make xconfig：在 2.4.X 以及以前版本中 xconfig 菜单是基于 TCL/TK 的图形库的。

所有内核配置菜单都是通过 Config.in 经由不同脚本解释器产生.config。而目前的 2.6.X 以上的内核用 QT 图形库。由 KConfig 经由脚本解释器产生。这两版本差别很大。2.6.X 的 xconfig 菜单结构清晰，使用也更方便。但基于目前 2.4.X 版本比较成熟，稳定，用的最多。所以这里我还是以 2.4.X 版本为基础介绍相关裁剪内容。同时因为 xconfig 界面比较友好，大家容易掌握。但它却没有 menuconfig 菜单稳定。有些人机器跑不起来。所以考虑最大众化角度，我们以较稳定，且不够友好的 menuconfig 为主进行介绍，它会用了，Xconfig 就没

问题。 2.4.X 版本 xconfig 配置菜单，2.4.X 版本 menuconfig 配置菜单，2.6.X 版本 xconfig 配置菜单分别如图 4.3.1，4.3.2，4.3.3 所示：

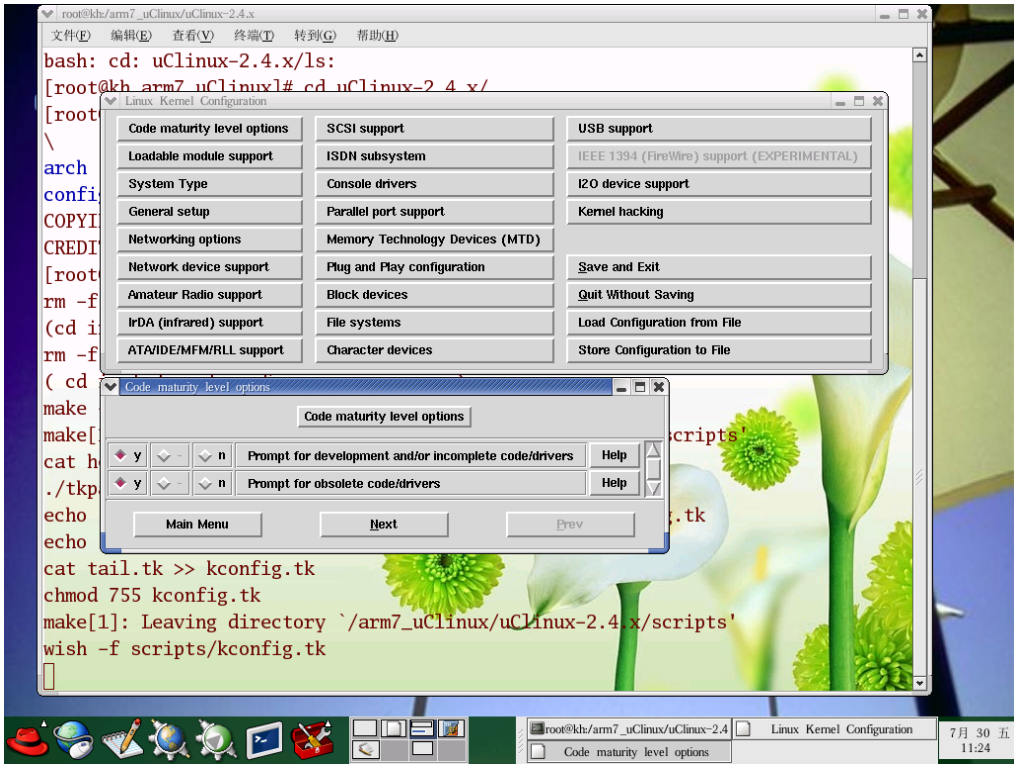


图 4.3.1　2.4.X 版本 xconfig 配置菜单

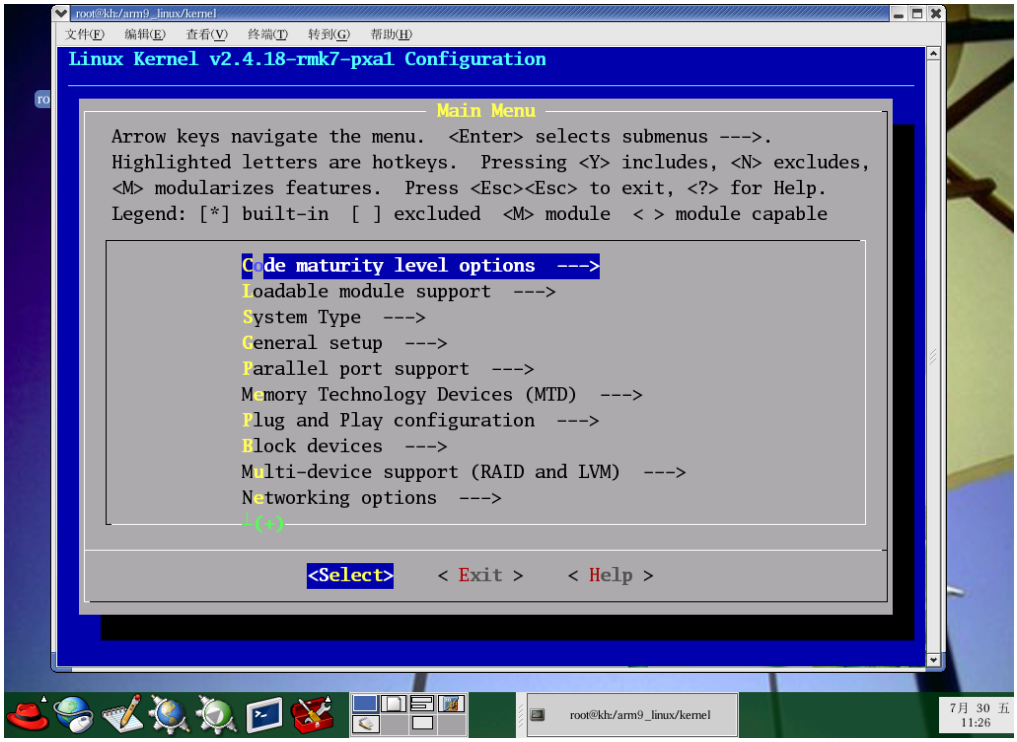


图 4.3.2 2.4.X 版本 menuconfig 配置菜单

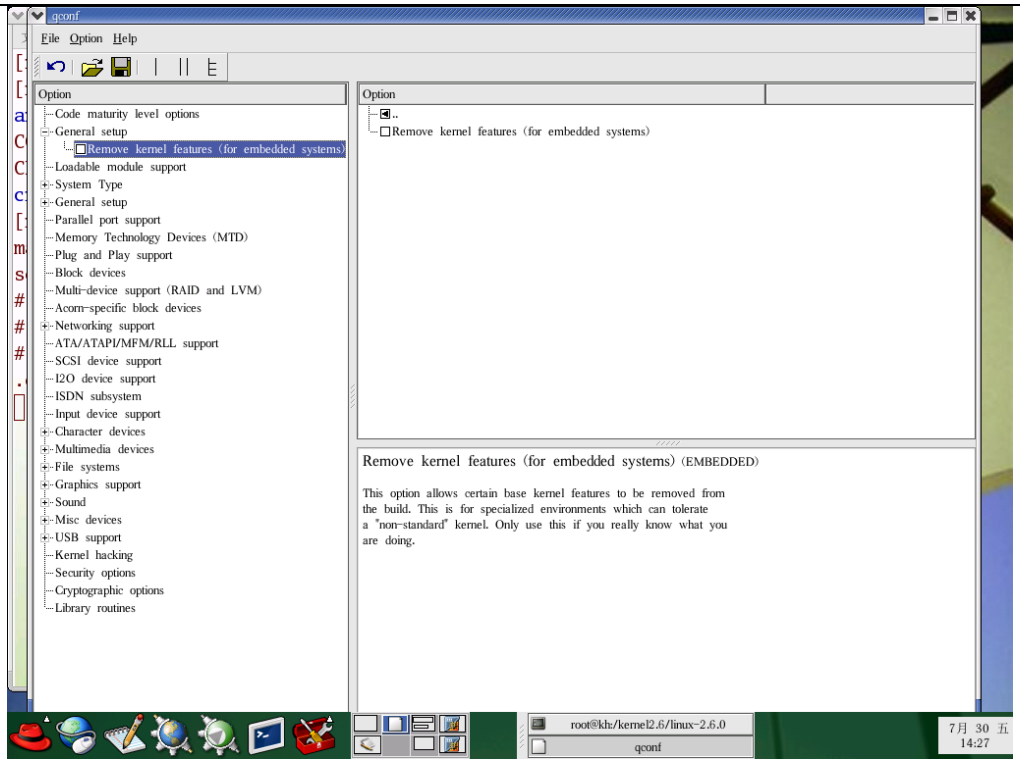


图 4.3.3 2.6.X 版本 xconfig 配置菜单

在选择相应的配置时，有三种选择方式，它们分别代表的含义如下：

Y－-将该功能编译进内核

N－-不将该功能编译进内核

M－-将该功能编译成可以在需要时动态插入到内核中的模块

如果你是使用的是 make xconfig，那使用鼠标就可以选择对应的选项。这里使用的是 make menuconfig，所以需要使用空格键进行选取。在每一个选项前都有一个括号， 有的是中括号有的是尖括号，还有圆括号。用空格键选择时可以发现，中括号里要么是空，要么是"*"，而尖括号里可以是空，"*"和"M"这表示前者对应的项要么不要，要么编译到内核里；后者则多一样选择，可以编译成模块。而圆括号的内容是要你在所提供的几个选项中选择一项。

注意：其中有不少选项是目标板开发人员加的，对于陌生选项，自己不知道该选什么时建议使用默认值)。

下面我们来看看具体配置菜单，进入内核所在目录，键入 make menuconfig 你就会看到配置菜单具有如下一些项：

以下内容仅供参考，实际内容可能根据具体内核版本有些差异。

1、Code maturity level options

代码成熟度选项，它又有子项：

1.1、prompt for development and/or incomplete code/drivers

该选项是对那些还在测试阶段的代码，驱动模块等的支持。一般应该选这个选项，除非你只是想使用 LINUX 中已经完全稳定的东西。但这样有时对系统性能影响挺大。

1.2、prompt for obsolete code/drivers

该项用于对那些已经老旧的，被现有文件替代了的驱动，代码的支持，可以不选，除非你的机器配置比较旧。但那也会有不少问题。所以该项以基本不用，在新的版本中已被替换。

2、loadable module support

动态加载模块支持选项，其子项有：

2.1、 enable module support

支持模块加载功能，应该选上。

2.2 set version information on all module symbols

该项用来支持跨内核版本的模块支持。即为某个版本的内核编译的模块可以在另一个版本的内核下使用，我们一般用不上。所以不选。

2.3 kernel module loader

如果你启用这个选项，你可以通过 kerneld 程序的帮助在需要的时候自动载入或卸载那些可载入式的模块。我们一般会选上。

3、system type

系统类型，主要是 CPU 类型，以及于此相关的内容。该项下的子项比较多，内容也比较复杂，我无法对每个 CPU 都加以说明，就以我们的开发平台作相应介绍，其它平台与此类似。

如果你是进行交叉编译，该项下的内容往往是经过相应目标平台开发人员修改的。主要是针对该平台的体系结构定义，这样可以优化系统性能。正因为目标平台的多样性所以该项定义也常常是五花八门。但开发人员一般会考虑到这些，事先设定好默认值。作为初学者按给出的默认选项就行。当然你也许想用一个原始的版本内核来建构针对你的平台的新内核，如果你的内核版本支持你目标平台所用的 CPU 那你就选上它。但不要选同系列中高于你所用的 CPU 型号否则不支持。你也可以在 Config.in 或 KConfig 中修改该项以支持你的目标平台。当然其中还有一些较复杂的事情要处理。由于本文档并不针对高级用户所以这部分内容不深入介绍。

在我们 CBT-6818 型部分平台上你在该项上看到相应的 ARM 系列 CPU。其它选项是关于该芯片及平台的一些结构定义。其它版本内核遇到的不会是这种情况，但一般包含 processor family 选项，它让我们选择 CPU 的类型，BIOS 可以自检到，留意一下你的系统的启动信息。选上正确的 CPU 类型就行。

4、General setup

4.1、support hot-plugable devieces

对可热拔插的设备的支持，看情况选择。若要对 U 盘等 USB 设备进行控制，建议选上。

下面分别是

4.2、Networking： support 网络支持，用到网络设备当然要选上。

4.3、System V IPC：支持 systemV 的进程间通讯，选上吧。

4.4、sysctl support：

该项支持在不重启情况下直接改变内核的参数。启用该选项后内核大约会增大 8K，如果你的内存太小就别选。

4.5、NWFPE math emulation

一般要选一个模拟数学协处理器，选上吧。

4.6、Power manager

电源管理，给 X86 编译内核时较有用可以选上，尤其是笔记本。给 ARM 编内核时可不选。

其它的看情况，在我们的平台上目前都用不着，不用选。

5、Networking option

网络选项，它主要是关于一些网络协议的选项。Linux 号称网络操作系统，它最强大的功能也就是在于对网络功能的灵活支持。这部分内容相当多，看情况，一般我们把以下几项选上。

5.1、 packet socket

包协议支持，有些应用程序使用 Packet 协议直接同网络设备通讯，而不通过内核中的其它中介协议。同时它可以让你在 TCP 不能用时找到一个通讯方法。

5.2、 unix domain socket

对基本 UNIX socket 的支持

5.3、 TCP/IP networking

对 TCP/IP 协议栈的支持，当然要。如果你的内核很在意大小，而且没有什么网络要就，也不跑类似 X Window 之类基于 Unix Socket 的应用那你可以不选，可节省大约 144K 空间。

至于其它众多的选项，这里挑一些对其意思简单介绍一下：

Network firewalls：是否让内核支持采用网络防火墙。如果计算机想当 firewalls server 或者是处于 TCP/IP 通信协议的网络的网路结构下这一项就选上。

Packet socket： mmapped IO 选该项则 Packet socket 可以利用端口进行快速通讯的。

IP：advanced router 如果你想把自己的 Linux 配成路由器功能这项肯定要选。选上后会带出几个子项。这些子项可以更精确配置相关路由功能。

socket filter：就是包过滤。

IP：multicasting 即网络广播协议的支持，可以一次一个 packet 送到好几台计算机的操作。

IP：syncookies 一种保护措施，将各种 TCP/IP 的通信协议加密，防止 Attacker 攻击用户的计算机，并且可以纪录企图攻击用户的计算机的 IP 地址。

IP： masquerading：这个选项可以在 Network Firewalls 选项被选后生效。

masquerading 可以将内部网络的计算机送出去的封包，通过防火墙服务器直接传递给远

端的计算机，而远端的计算机看到的就是接收到的防火墙服务器送过来的封包，而不是从内部的计算机送过来的。这样如果内部只有一台计算机可以上网，其余的机器可以通过这台机子的防火墙服务器向外连线。它是一种伪装，如果你的网络里有一些重要的信息，那你在使用 IP Masquerade 之前就要三思。因为它即可能成为你通往互联网的网关同样也可能为外边世界进入你网络的提供一条途径。

IP：ICMP masquerading： 一般 masquerading 只提供处理 TCP，UDP packets，若要让 masqurerading 也能处理 ICMP packets，就把这选项选上。

IP：always defragment：可将接收到的 packet fragments 重新组合回原来那个封包。

IP：accounting：统计 IP packet 的流量，也就是网络的流通情况。

IP：optimize as router not host：可以关闭 copy&checksum 技术，防止流量大的服务器的 IP packets 丢失。

IP：tunneling tunnel 即隧道。这里是指用另外一种协议来封装数据或包容协议类型。这样就相当于在不同的协议之间打了条隧道。使得数据包可以被不同的协议接受和解释。这样我们可在不同网域中使用 linux，且都不用改 IP 就可以直接上网了。对于嵌入式设备这点还是挺有用的。

IP：GRE tunneling 此"GRE"可不是彼"GRE"，它是(Generic Routing Encapsulation)。选该项后可以支持在 IPv4 与 IPv6 之间的通讯。

IP： ARP daemon support 即对 ARP 的支持。它是把 IP 地址解析为物理地址。

IP：Reverse ARP ：RARP(逆向地址解析)协议，可提供 bootp 的功能，让计算机从可以从网卡的 Boot Ram 启动，这对于搭建无盘工作站是很有用的，但现在硬件价格下跌好像无盘工作站用的已经不多了。

IP：Disable Path MTU Discovery：MTU 有助于处理拥挤的网络。MTU（Maximal Transfer Unit）最大的传输单位，即一次送往网络的信息大小。而 Path MTUD iscovery 的意思是当 Linux 发现一些机器的传输量比较小时，就会分送网络信息给它。如此可以增加网络的速度，所以大部分时候都选 N，也就是 Enable。

The IPX protocol：IPX 为 Netware 网络使用的通讯协议，主要是 NOVELL 系统支持的。

QoS and/or fair queueing：QoS 即(Quality Of Service)这是一种排定某种封包先送的网络线程表，可同时针对多个网络封包处理并依优先处理顺序来排序，称之为 packet schedulers。此功能特别是针对实时系统时格外重要，当多个封包同时送到网络设备时，Kernel 可以适当的决定出哪一个封包必须优先处理。因此 Kernel 提供数种 packet scheduling algorithm。

其它网络选项还有很多考虑篇幅无法给其作一一解释，如果你有兴趣可以查看相关帮助文件。这里我建议你下一个 3.5.X 内核。在其 Xconfig 中可以很方便查看各项的帮助信息。

6、Networking deveices：

网络设备支持。上面选好了网络协议了，现在选的是网络设备，其实主要就是网卡，所以关键是确定自己平台所使用的网卡芯片。该项下的子项也不少。

6.1、Dummy net driver support

哑(空)网络设备支持。它可让我们模拟出 TCP / IP 环境对 SLIP 或 PPP 的传输协议提供支持。选择它 Linux 核心增大不大。如果没有运行 SLIP 或 PPP 协议，就不用选它。

6.2、Bonding driver support bonding

技术是用来把多块网卡虚拟为一块网卡的，使他们有一个共同的 IP 地址。

6.3、Universal TUN/TAP device driver support

用于支持 TUNx/TAPx 设备的

6.3、SLIP（serial line） support

这是 MODEM 族常用的一种通讯协议，必须通过一台 Server（叫 ISP）获取一个 IP 地址，然后利用这个 IP 地址，可以模拟以太网络，使用有关 TCP / IP 的程序。

6.4、PLIP（parallel port） support

依字面上看，它是一种利用打印机的接口（平行接口），然后利用点对点来模拟 TCP / IP 的环境。它和 SLIP / PPP 全都属于点对点通讯，您可以把两台电脑利用打印机的连接接口串联起来，然后，加入此通讯协议。如此一来，这两部电脑就等于一个小小的网络了。不过，如果电脑有提供打印服务的话，这个选项最好不要打开，不然可能会有问题（因为都是用平行接口）。

6.5、PPP（point－to－point）support

点对点协议，近年来，PPP 协议已经慢慢的取代 SLIP 的规定了，原因是 PPP 协议可以获取相同的 IP 地址，而 SLIP 则一直在改变 IP 地址，在许多的方面，PPP 都胜过 SLIP 协议。

6.6、EQL (serial line load balancing) support

两台机器通过 SLIP 或 PPP 协议，使用两个 MODEM，两条电话线，进行通讯时，可以用这个 Driver 以便让 MODEM 的速度提高两倍。当然。

6.7、Token Ring driver support

对令牌环网的支持。

6.8、Ethertap network tap

6.9、Ethernet (10 or 100Mbit)

十至百兆以太网设备，我们现在该类型设备用的比较多。进入该项里头还有许多小项，它们是关于具体网络设备(一般就是网卡)的信息。选择我们平台相关的就行(如 DM9000)。

6.10、ARCnet support

它是一种网卡但不流行基本没用。其它的诸如千兆以太网，万兆以太网，无线网络，广域网，ATM，PCMCIA 卡等等网络设备的支持，要看你的具体应用而定。这里不作一一介绍。

7、Amateur Radio support

这个选项用的不多，它是用来启动无线网络的，通过无线网络我们可以利用公众频率来进行数据传输，如果你有相关无线网络通讯设备就可以用它。

8、IrDA(infrared) support

该项也属于无线通讯的一种，用于启动对红外通讯的支持。目前在 2.6.X 的内核里对它的支持内容更丰富了。

9、ATA/ATAPI/MFM/RLL support

该项主要对 ATA/ATAPI/MFM/RLL 等协议的支持。在嵌入式设备中，目前这些设备应用的还不多，但台式机及笔记本用户如果你有支持以上协议的硬盘或光驱就可选上它。在 2.6.X 内核中这方面的支持内容也比较丰富。

10、SCSI device support

如果你有 SCSI 设备(SCSI 控制卡，硬盘或光驱等)你选上这项。目前 SCSI 设备类型已经比较多，要具体区分它们你得先了解他们所使用的控制芯片类型。2.6.X 内核中对各类型 SCSI 设备已经有更具体详细的支持。

11、ISDN support ISDN (Integrated Services Digital Networks)

它是一种高速的数字电话服务。通过专用 ISDN 线路加上装在你电脑上的 ISDN 卡。利用 SLIP 或 PPP 协议进行通讯。所以你若想启动该项支持 ISDN 通讯，你还应启动前面提到的 Networking Devices 中的 SLIP 或 PPP。

12、Console drivers support

控制台设备支持。目前安装 uClinux/Linux 的设备几乎都是带控制台的，所以这项是必选项。这里头还有几个子项：

12.1、VGA text console

一般台式机选该项。支持 VGA 显示设备。

12.2、Support Frame Buffer devices

该项支持 Frame Buffer 设备。Frame Buffer 技术在 2.4.X 内核被全面采用。它通过开辟一块内存空间模拟显示设备。这样我们可以像操作具体图形设备一样来操作这块内存，直接给它输入数据，在具体显示设备上输出图形。在嵌入式设备上广泛采用 LCD 作为显示设备，所以该项显得比较重要。当该项被选上后会出现一子项让我们根据自己平台配备的具体硬件选择相应支持。这些也往往是设备开发人员给添加的。

13、parallel port support

对并行口的设备的支持。LINUX 可以支持 PLIP 协议（利用并行口的网络通讯协定），并口的打印机，ZIP 磁盘驱动器、扫描仪等。如果有打印机在选择利用并口通讯时要小心，因为它们可能会互相干扰。

14、Memory Technology Device (MTD) support

MTD 包含 flash，RAM 等存储设备。MTD 在现在的嵌入式设备中应用的相当多，也特别重要。选中该项我们可以对 MTD 进行动态支持。其下还有好多具体小项，这里做一些解释：

14.1、MTD partitioning support

选上该项可支持对 MTD 的分区操作。我们在对嵌入式设备的操作系统移植过程中往往要对 MTD 进行分区，然后在各分区放置不同的数据。以让系统能被正确引导启动。

14.2、Direct char device access to MTD devices

选该项为系统的所有 MTD 设备提供一个字符设备，通过该字符设备我们能直接对 MTD 设备进行读写以及利用 ioctl()函数来获取该 MTD 设备的相关信息。

14.3、Caching block device access to MTD devices

有许多 flash 芯片其擦除的块太大因此作为块设备使用效率被大打折扣。我们选上该项后，它支持利用 RAM 芯片作为缓存来使用 MTD 设备。这时对于 MTD 设备块设备就相当于它的一个用户。通过 JFFS 文件系统的控制，它可以模拟成一个小型块设备，具有读，写，擦，校验等一系列功能。

14.4、NAND flash device drivers

子项中有几项是关于 MTD 设备驱动的，我们的平台选择的是 NAND flash 所以我们选上它。选上后在其二级子项中我们还要选上：

14.4.1、NAND devices support

14.4.2、verify NAND pages writes

支持页校验

14.4.3、NAND flash device on ARM board

15、Plug and Play support

这是对 PNP(即插即用)设备的支持。

16、block devices

块设备，该项下也有好几个子项，主要是关于各种块设备的支持。至少把 RAM 的支持项选上。如在我们 ICS-IOT-CEP 全功能物联网教学科研平台上我们要选上 RAM disk support

17、File systems

文件系统在 Linux 中是非常重要的。该项下的子项也非常多。

17.1、Quota support

份额分配支持。选择该项则系统支持对每个用户使用的磁盘空间进行限制。

17.2、Kernel automounter support

在有 NFS 文件系统的支持下，选择该项可使得内核可以支持对一些远端文件系统的自动挂栽。

17.3、Kernel automounter version 4 support

V3 版本的升级，它兼容 V3

17.4、Reiserfs support

ReiserFS 这种文件系统以日志方式不仅把文件名，而且把文件本身保存在一个"平衡树"里。其速度与 ETX2 差不多但比传统的文件系统架构更为高效。尤其适合大目录下文件的情况。

17.5、ROM file system support

它是一个非常小的只读文件系统，主要用于安装盘及根文件系统。

17.6、JFS filesystem support

这是 IBM 的一个日志文件系统。

17.7、Second extended fs support

著名的 EXT2(二版扩展文件系统)，除非你是用 DOS 模拟器否则得选它。

17.8、Ext3 journalling file system support

它其实是 EXT2 的日志版，我们通常叫它 EXT3。

17.9、Journalling Flash file system v2(jffs2) support

Flash 日志文件系统

17.10、ISO 9660 CDROM file system support

光驱的支持

17.11、/proc file system support

这是虚拟文件系统，能够提供当前系统的状态信息。它运行时在内存生成，不占任何硬盘空间。通过 CAT 命令可以读到其文件的相关信息。

17.12、/dev file system support

它是类似于/proc 的一个文件系统，也是虚拟的，主要用于支持 devfs(设备文件系统)。把它选上，这样我们就可以不依赖于传统的主次设备号的方式来管理设备。而是由 devfs 自动管理。

17.13、NFS file system

网络文件系统。

17.13.1、NFS file system support

对网络文件系统的支持。NFS 通过 SLIP，PLIP，PPP 或以太网进行网络文件管理。它是比较重要的。

17.13.2、NFS server support

选这项可以把你的 Linux 配置为 NFS server

17.13.3、SMB file system support

SMB (Server Message Block) ，它是用于和局域网中相连的 Windows 机器建立连接的。相当于网上邻居。 这些协议都需要在 TCP/IP 被启用后才有效。

17.14、Native Language Support

就是对各国语言的支持。

18、character devices

LINUX 支持很多特殊的字符设备，所以该项下的子项也特别多。

18.1、virtual terminal

虚拟终端，选上

18.2、support for console on virtual terminal

虚拟终端控制台，也给选上。

18.3、non-standar serial port support

非标准串口设备的支持。如果你的平台上有一些非标准串口设备需要支持，就选上它。

18.4、Serial drivers

串口设置，一般选上自己开发平台相关的串口就行。

18.5、UNIX98 PTY support

PTY(pseudo terminal)伪终端，它是软件设备由主从两部分组成。从设备与具体的硬件终端绑定，而主设备则由一个进程控制向从设备写入或读出数据。其典型应用如：telnet 服务器和 xterms

18.6、I2C support

对 I2C 设备的支持。

18.7、Mice

就是对鼠标的支持。

18.8 Joysticks

对一些游戏手柄的支持。

18.9 QIC-02 tape support

对一些非 SCSI 的磁带设备支持。

18.10 watchdog card support

对看门狗定时设备的支持。

18.11 /dev/nvram support

这是一种和 BIOS 配合工作的 RAM 设备。我们常称它为"CMOS RAM"，而 NVRAM 主要是在 Ataris 机器上的称法。通过设备名/dev/nvram 可以读写该部分内存内容。它通常保存一些机器运行必需的重要数据，而且保证掉电后能继续保存。

18.12、Enhanced Real Time Clock Support

在每台 PC 机上都内建了一个时钟，它可以产生出从 1Hz 到 8192Hz 的信号。在多 CPU 的机器中这项必选。

18.13、/dev/agpgart (AGP Support)

AGP (Accelerated Graphics Port) 通过它可以沟通显卡与其它设备。如果有 AGP 设备就选上它。嵌入式系统中目前用的还不多，但台式机 AGP 设备已相当普及。

18.14、Siemens R3964 line discipline

这项主要是支持利用 Siemens R3964 的包协议进行同步通讯的。

18.15、Direct Rendering Manager (XFree86 4.1.0 and higher DRI support)

选该项后则在内核级提供对 XFree86 4.0 的 DRI(Direct Rendering Infrastructure)的支持，选择正确的显卡后，该设备能提供对同步，安全的 DMA 交换支持。选该项同时要把/dev/agpgart (AGP Support)选上。

19、USB support

即对 USB 设备的支持。 如果有相关设备就选上。

20、sound card support

关于声卡的支持，根据你自己的情况来配置。

21、kernel hacking

这里是一些有关内核调试及内核运行信息的选项。如果你正打算深入研究自己系统上运行的 LINUX 如何运作，可以在这里找到相关选项，但一般没有必要的话可以全部关掉。

4.4 内核中的 Kconfig 和 Makefile 文件

在 3.5 内核的源码树目录下一般都会有两个重要文件：Kconfig 和 Makefile。分布在各目录下的 Kconfig 构成了一个分布式的内核配置数据库，每个 Kconfig 分别描述了所属目录下源文件相关的内核配置菜单。在内核配置 make menuconfig(或 xconfig 等)时，从 Kconfig 中读出配置菜单，用户配置完后保存到.config(在内核源码顶层目录下生成)中。在内核编译时，主 Makefile 调用这个.config(隐藏文件)，就知道了用户对内核的配置情况。

Kconfig 的作用就是对应着内核的配置菜单。假如要想添加新的驱动到内核的源码中，可以通过修改 Kconfig 来增加对我们驱动的配置菜单，这样就有途径选择我们的驱动。

如果想使这个驱动被编译进内核或被内核支持，还要修改该驱动所在目录下的 Makefile 文件。该 Makefile 文件定义和组织该目录下驱动源码在内核目录树中的编译规则。这样在 make 编译内核的时候，内核源码目录顶层 Makefile 文件会递归的连接相应子目录下的 Makefile 文件，进而对驱动程序进行编译。

如上所述，添加用户驱动程序(内核程序)到内核源码目录树中，一般需要修改 Konfig 及 Makefile 两个文件。这要求我们用户要对上述连个文件的特殊语法有一定了解，如下：

♦ **Kconfig 语法：**

每行都是以关键字开始，并可以接多个参数，最常见的关键字就是 config。

语法：

```
config symbol
options
<!--[if                                                    !supportLineBreakNewLine]-->
<!--[endif]-->
```

symbol 就是新的菜单项，options 是在这个新的菜单项下的属性和选项。

其中 options 部分有：

1）类型定义：

每个 config 菜单项都要有类型定义，bool：布尔类型， tristate 三态：内建、模块、移除， string：字符串， hex：十六进制， integer：整型

例如

config HELLO_MODULE

bool "hello test module"

bool 类型的只能选中或不选中，tristate 类型的菜单项多了编译成内核模块的选项，假如选择编译成内核模块，则会在.config 中生成一个 CONFIG_HELLO_MODULE=m 的配置，假如选择内建，就是直接编译成内核影响，就会在.config 中生成一个 CONFIG_HELLO_MODULE=y 的配置。

2）依赖型定义 depends on 或 requires

指此菜单的出现是否依赖于另一个定义

config HELLO_MODULE

bool "hello test module"

depends on CPU_S5P6818

这个例子表明 HELLO_MODULE 这个菜单项只对 S5P6818 处理器有效，即只有在选择了 CPU_S5P6818 时，该菜单才可见(可配置)。

3）帮助性定义

只是增加帮助用关键字 help 或---help---

<!--[if !supportLineBreakNewLine]-->

<!--[endif]-->

更多详细的 Kconfig 语法可参考 Documentation /kbuild/kconfig-language.txt 文档。

♦ 内核的 Makefile 语法

内核的 Makefile 分为 5 个组成部分：

Makefile	最顶层的 Makefile
.config	内核的当前配置文档，编译时成为顶层 Makefile 的一部分
arch/$(ARCH)/Makefile	和体系结构相关的 Makefile
Makefile.*	一些 Makefile 的通用规则
kbuild Makefile	各级目录下的大概约 500 个文档，编译时根据上层 Makefile 传下来的宏定义和其他编译规则，将源代码编译成模块或编入内核。

顶层的 Makefile 文档读取 .config 文档的内容，并总体上负责 build 内核和模块。Arch Makefile 则提供补充体系结构相关的信息。（其中.config 的内容是在 make menuconfig 的时候，通过 Kconfig 文档配置的结果）在内核目录的 Documentation/kbuild 目录下有详细的介

绍有关 kernel makefile 的知识。

5. 实验步骤

♦ 编译 Linux 内核

1）进入宿主机中解压平台配套 Linux 内核源码包：

```
cbt@Cyb-Bot:~$ cd /CBT-6818/SRC/linux/
cbt@Cyb-Bot: /CBT-6818/SRC/linux$ tar jxvf cbt-6818-linux.tar.bz2
```

内核源码包的名称中包含时间信息，请以时间源码包名称为准(默认已经解压到/CBT-6818/SRC/linux/目录)。

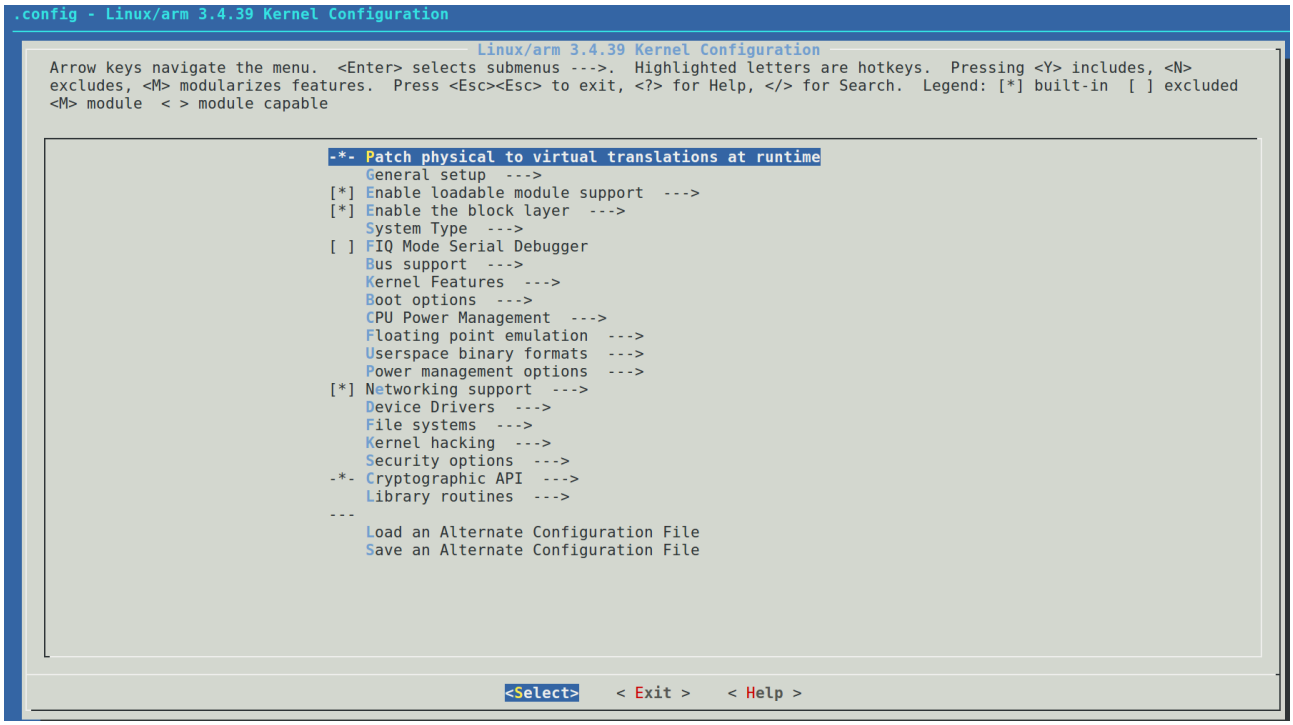
2）进入解压后的内核源码，拷贝获取内核配置文件.config

```
cbt@Cyb-Bot: /CBT-6818/SRC/linux$ cd kernel/
cbt@Cyb-Bot: /CBT-6818/SRC/linux/kernel$ cp 6818_linux_defconfig   .config
```

内核的编译依赖源码目录的.config 文件，用户可以使用出厂默认的文件，也可以生成自己的配置文件.config，平台具体出厂配置文件由 CBT-6818 标识加日期命名，具体名称参阅源码目录下该文件。

3）通过 make menuconfig 命令进入内核菜单，进行相应功能模块的订制与裁剪。

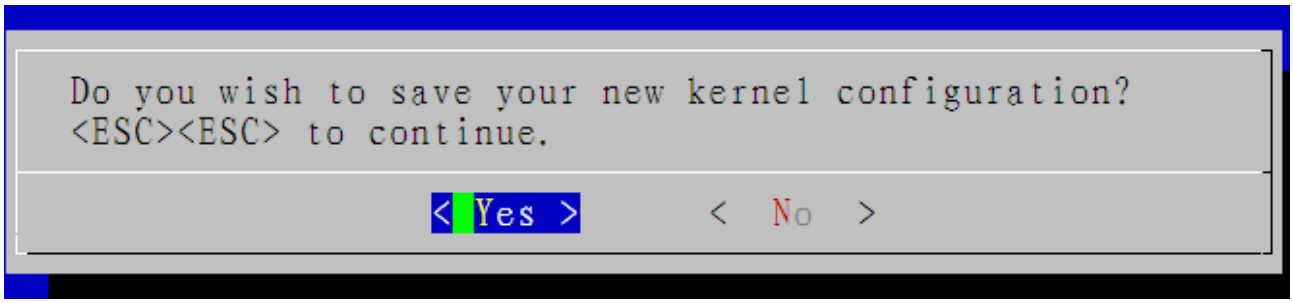
```
cbt@Cyb-Bot: /CBT-6818/SRC/linux/kernel$ make menuconfig
```



用户可以通过键盘上的方向键和空格键，对上面的菜单进行选择和控制，其中“*”代表该模块功能静态编译，“M”代表模块化编译，“空”代表不编译该模块功能。make menuconfig 命令将打开内核配置菜单，需要当前终端窗口符合一定大小范围才能正常显示，不能过大或过小，使用该命令时请注意。

该内核默认的配置已经在出厂时候完成，用户可以简单尝试进入其中目录去学习和了解相关内核模块的内容，如果对其中的模块不是很熟悉，建议不要随意修改内核配置，以免导致无法正常启动。

如果做了配置的修改，在退出时，系统会提示保存，保存后新的配置即生效。



4）编译内核

在 linux 目录下，运行编译脚本，即可完成内核的编译，最终在内核源码根目录 out/release 目录下生成内核的镜像文件 boot.img。

```
cbt@Cyb-Bot: /CBT-6818/SRC/linux/kernel$ cd ../
cbt@Cyb-Bot: /CBT-6818/SRC/linux$ ./mk -k
cbt@Cyb-Bot: /CBT-6818/SRC/linux$ ls out/release/
 boot.img
```

♦ **下载 Linux 内核**

1）硬件连接

打开 Cortex-A53 网关，将 Cortex-A53 的串口 0 用串口线与电脑连接，将 Cortex-A53 的 OTG 接口通过 miniUSB 线与电脑连接。

2）进入 bootloader 界面

电脑打开 XSHELL 程序，根据自己的电脑配置 COM 和波特率 115200，连接后启动 Cortex-A53，在启动后 3 秒读数时敲键盘任意键进入 bootloader 界面，输入 fastboot 命令。

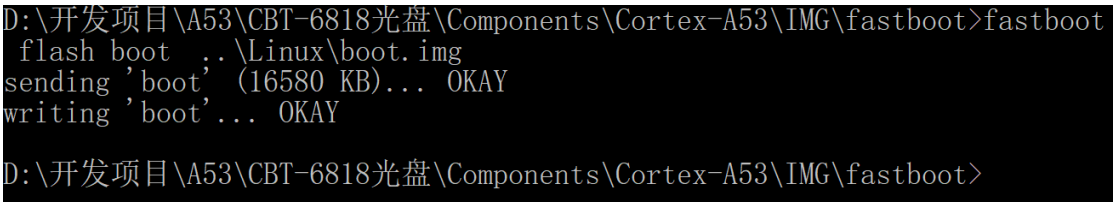
```
X6818# fastboot

Fastboot Partitions:
 mmc.2: ubootpak, img : 0x200, 0x78000
 mmc.2: 2ndboot, img : 0x200, 0x4000
 mmc.2: bootloader, img : 0x8000, 0x70000
 mmc.2: boot, fs : 0x100000, 0x4000000
 mmc.2: system, fs : 0x4100000, 0x2f200000
 mmc.2: cache, fs : 0x33300000, 0x1ac00000
 mmc.2: misc, fs : 0x4e000000, 0x800000
 mmc.2: recovery, fs : 0x4e900000, 0x1600000
 mmc.2: userdata, fs : 0x50000000, 0x0
Support fstype : 2nd boot factory raw fat ext4 emmc nand ubi ubifs
Reserved part  : partmap mem env cmd
DONE: Logo bmp 311 by 300 (3bpp), len=280854
DRAW: 0x47000000 -> 0x46000000
Load USB Driver: android
Core usb device tie configuration done
OTG cable Connected!
----------------------------------------
```

3）进入 fastboot 目录烧写

打开计算机运行进入 CMD，进入光盘资料中\Compnents\Cortex-A53\IMG\fastboot。

4）烧写内核

在命令提示符下输入命令 fastboot flash boot　..\Linux\boot.img。



实验二. 构建根文件系统

1. 实验目的

- 了解嵌入式系统中根文件系统的目录结构和重要配置文件的作用。
- 掌握嵌入式系统中根文件系统的搭建过程及常见文件系统格式工具的使用。

2. 实验环境

- 硬件：A53 智能网关实验平台，PC 机 Pentium 500 以上，硬盘 40G 以上，内存大于 256M
- 软件：Vmware Workstation +ubuntu + MiniCom/超级终端 +ARM-LINUX 交叉编译开发环境
- 实验目录：/CBT-6818/SRC/linux

3. 实验内容

- 使用 busybox 生成文件系统中的命令和工具部分，使用 mkyaffs2 工具制作 yaffs2 格式的根文件系统。
- 分析根文件系统 etc 目录下重要配置文件的格式及语法，熟悉根文件系统的启动过程。

4. 实验原理

4.1 Linux 的文件系统

Linux 的一个最重要特点就是它支持许多不同的文件系统。这使 Linux 非常灵活，能够与许多其他的操作系统共存。Linux 支持的常见的文件系统有：JFS、 ReiserFS、ext、ext2、ext3、ISO9660、XFS、Minx、MSDOS、UMSDOS、VFAT、NTFS、HPFS、NFS、SMB、SysV、PROC 等。随着时间的推移，Linux 支持的文件系统数还会增加。

Linux 是通过把系统支持的各种文件系统链接到一个单独的树形层次结构中，来实现对多文件系统的支持的。该树形层次结构把文件系统表示成一个整个的独立实体。无论什么类型的文件系统，都被装配到某个目录上，由被装配的文件系统的文件覆盖该目录原有的内容。该个目录被称为装配目录或装配点。在文件系统卸载时，装配目录中原有的文件才会显露出来。

在 Linux 文件系统中，文件用 i 节点来表示、目录只是包含有一组目录条目列表的简单文件，而设备可以通过特殊文件上的 I/O 请求被访问。

♦ **I 节点（Inodes）**

每个文件都是由被称为 i 节点的一个结构来表示的。每个 i 节点都含有对特定文件的描

述：文件类型、访问权限、属主、时间戳、大小、指向数据块的指针。分配给一个文件的数据块的地址也存储在该文件的 i 节点中。当一个用户在该文件上请求一个 I/O 操作时，内核代码将当前偏移量转换成一个块号，并使用这个块号作为块地址表中的索引来读写实际的物理块。如图 4.1.1 表示了一个 i 节点的结构：

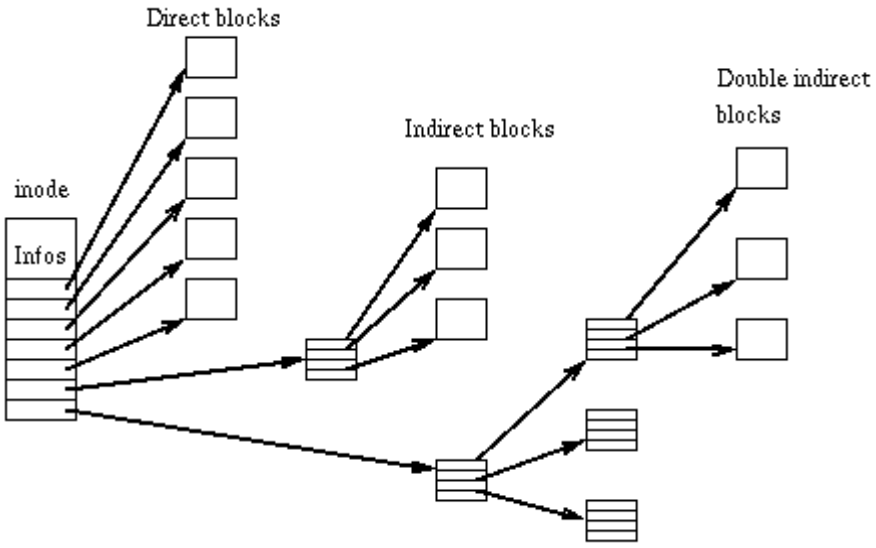


图 4.1.1 I 节点结构

♦ **目录**

目录是一个分层的树结构。每个目录可以包含有文件和子目录。 目录是作为一个特殊的文件实现的。实际上，目录是一个含有目录条目的文件，每个条目含有一个 i 节点号和一个文件名。当进程使用一个路径名时，内核代码就会在目录中搜索以找到相应的 i 节点号，在文件名被转换成了一个 i 节点以后，该 i 节点就被加载到内存中并被随后的请求所使用。如图 4.1.2 所示：

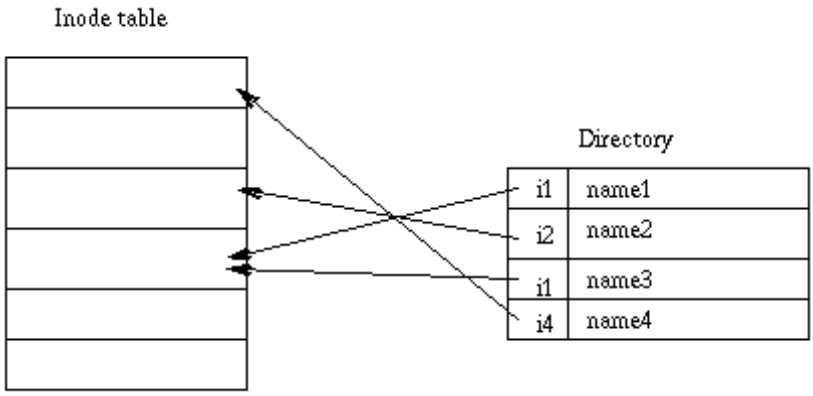


图 4.1.2 目录结构

♦ **链接（Links）**

Unix 文件系统实现了链接的概念。几个文件名可以与一个 i 节点相关联。i 节点含有一个字段，其中含有与文件的关联数目。要增加一个链接只需简单地建立一个目录项，该目录

项的 i 节点号指向该 i 节点并增加该 i 节点的连接数即可。但删除一个链接时，也即当使用 rm 命令删除一个文件名时，内核会递减 i 节点的链接计数值，如果该计数值等于零的话，就会释放该 i 节点。

这种类型的链接称为硬链接(hard link)，并且只能在单独的文件系统内使用：也即不可能创建一个跨越文件系统的硬链接。而且，硬链接只能指向文件：为了防止造成目录树的循环，不能创建目录的硬链接。

在大多数 Unix 文件系统中还有另外一种链接。符号链接(Symbolic link)仅是含有一个文件名的简单文件。在从路径名到 i 节点的转换中，但内核遇到一个符号链接时，就用该符号链接文件的内容替换链接的文件名，也即用目标文件的名称来替换，并重新开始路径名的翻译工作。由于符号链接并没有指向 i 节点，因此就有可能创建一个跨越文件系统的符号链接。符号链接可以指向任何类型的文件，甚至是一个不存在的文件。由于没有与硬链接相关的限制，因此它们非常有用。然而，它们会用掉一点磁盘空间，并且需要为它们分配 i 节点和数据块。由于内核在遇到一个符号链接时需要重新开始路径名到 i 节点的转换工作，因此会造成路径名到 i 节点转换的额外负担。

♦ 设备特殊文件(Device special files)

在 Unix 类操作系统中，设备是可以通过特殊的文件进行访问的。设备特殊文件不会使用文件系统上的任何空间，它只是对设备驱动程序的一个访问点。

存在两类设备特殊文件：字符和块设备特殊文件。前者允许以字符模式进行 I/O 操作，而后者需要通过高速缓冲功能以块模式写数据方式进行操作。当对设备特殊文件进行 I/O 请求操作，就会传递到(虚拟的)设备驱动程序中。对特殊文件的引用是通过主设备号和次设备号进行的，主设备号确定了设备的类型，而次设备号指明了设备单元。

♦ 虚拟文件系统(The Virtual File System)

Linux 内核含有一个虚拟文件系统层，用于系统调用操作文件。VFS 是一个间接层，用于处理涉及文件的系统调用，并调用物理文件系统代码中的必要功能来进行 I/O 操作。该间接机制常用于 Unix 类操作系统中，以利于集成和使用几种类型的文件系统。

当处理器发出一个基于文件的系统调用时，内核就会调用 VFS 中的一个函数。该函数会处理与结构无关的操作并且把调用重新转向到与结构相关的物理文件系统代码中的一个函数去。文件系统代码使用高速缓冲功能来请求对设备的 I/O 操作。这个方案见下图 4.1.3 所示:

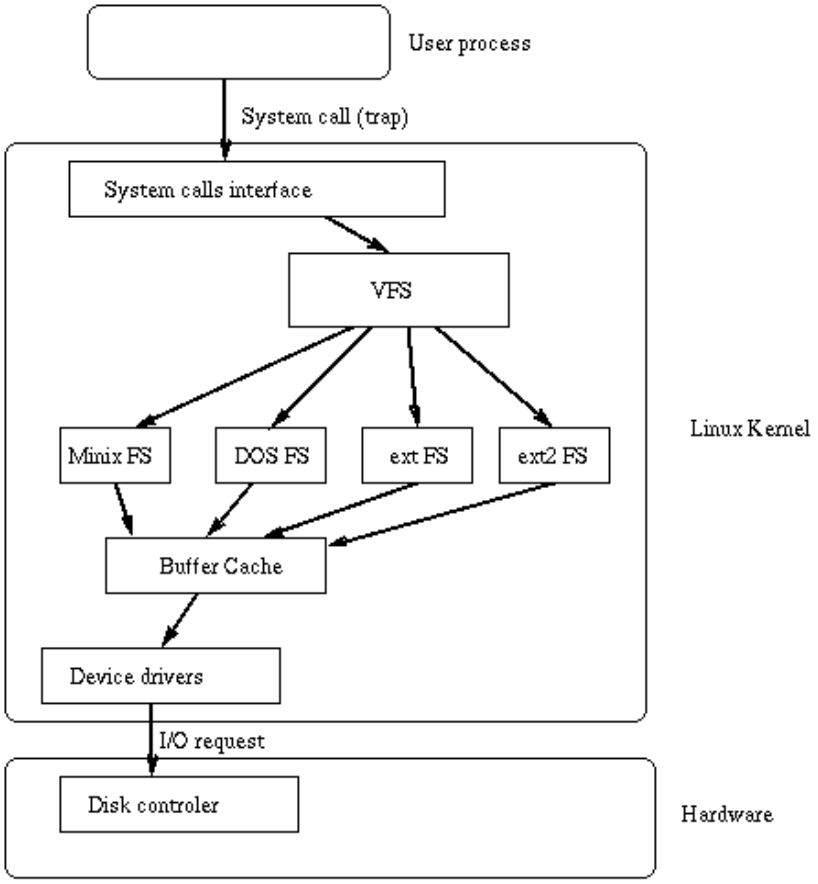


图 4.1.3 VFS 文件系统

♦ **VFS 结构**

VFS 定义了每个文件系统必须实现的函数集。该接口由一组操作集组成，涉及三类对象：文件系统、i 节点和打开文件。

VFS 知道内核所支持的文件系统的类型，它使用一个在内核配置时定义的一张表来获取这些信息。该表中的每个条目描述了一个文件系统类型：它含有文件系统类型的名称以及在加载操作时调用的函数的指针。当需要加载一个文件系统时，就会调用相应的加载函数。该函数负责从磁盘上读取超级块、初始化内部变量，并且向 VFS 返回被加载文件系统的描述符。在文件系统已被加载以后，VFS 函数就可以使用这个描述符来访问物理文件系统的子程序。

被加载文件系统的描述符含有几类数据：每个文件系统类型常用的信息、物理文件系统内核代码提供的函数指针以及物理文件系统代码私有数据。文件系统描述符中所包含的函数指针使得 VFS 能访问文件系统的内部函数。

VFS 还使用了另外两类描述符: i 节点描述符和打开文件描述符。每个描述符含有与所使用文件相关的信息以及物理文件系统代码提供的操作集。i 节点描述符含有用于任何文件操作（例如， create, unlink）的函数指针集，而文件描述符含有操作已被打开文件的函数的指针（例如， read, write）。

4.2 嵌入式系统中的根文件系统

文件系统都会被烧录在与某一存储设备上。在嵌入式设备上很少使用大容量的 IDE 硬盘作为自己的存储设备，嵌入式设备往往选用 ROM、闪存（flash memory）等作为它的主要存储设备。在嵌入式设备上选用哪种文件系统格式与闪存的特点是相关的。

♦ 闪存技术

目前市场上的 flash 从结构上大体可以分为 AND、NAND、NOR 和 DiNOR 等几种。其中 NOR 和 DiNOR 的特点为相对电压低、随机读取快、功耗低、稳定性高，而 NAND 和 AND 的特点为容量大、回写速度快、芯片面积小。现在，NOR 和 NAND FLASH 的应用最为广泛，除了在嵌入式设备上得到广泛的应用外，在 CompactFlash、Secure Digital、PC Cards、MMC 存储卡以及 USB 闪盘存储器市场都都占用较大的份额。

NOR 的特点是可在芯片内执行，这样应该程序可以直接在 flash 内存内运行，不必再把代码读到系统 RAM 中。NOR 的传输效率很高，但写入和探险速度较低。而 NAND 结构能提供极高的单元密度，并且写入和擦除的速度也很快，是高数据存储密度的最佳选择。这两种结构性能上的异同步如下：

1）NOR 的读速度比 NAND 稍快一些。

2） NAND 的写入速度比 NOR 快很多。

3）NAND 的擦除速度远比 NOR 快。

4）NAND 的擦除单元更小，相应的擦除电路也更加简单。

5）NAND 闪存中每个块的最大擦写次数量为万次，而 NOR 的擦写次数是十万次。

此外，NAND 的实际应用方式要比 NOR 复杂得多。NOR 可以直接使用，并在上面直接运行代码。而 NAND 需要 I/O 接口，因此使用时需要驱动程序。不过当今流行的操作系统对 NAND Flash 都有支持，Linux 内核也对 NAND Flash 提供了很好的支持。由于以上 flash 的特性决定了，在嵌入式设备中，我们一般会把只读属性的映象文件，如启动引导程序 blob、内核、文件系统文件存放在 NOR Flash 中，而把一些读写类的文件，如用户应用程序等存放在 NAND Flash 中,出于成本的考虑，很多厂家会选用低容量昂贵的 NOR Flash 存储启动引导程序和内核，而把文件系统存放在 NAND Flash 中。

♦ Ext2fs 文件系统

Ext2fs 是 Linux 事实上的标准文件系统，它已经取代了它的前任扩展文件系统（或 Extfs）。Extfs 支持的文件大小最大为 2 GB，支持的最大文件名称大小为 255 个字符 — 而且它不支持索引节点（包括数据修改时间标记）。Ext2fs 做得更好，它的优点是：

1）Ext2fs 支持达 4 TB 的内存。

2）Ext2fs 文件名称最长可以到 1012 个字符。

3）当创建文件系统时，管理员可以选择逻辑块的大小（通常大小可选择 1024、2048 和 4096 字节）。

Ext2fs 实现了快速符号链接：不需要为此目的而分配数据块，并且将目标名称直

接存储在索引节点（inode）表中。这使性能有所提高，特别是在速度上。

因为 Ext2 文件系统的稳定性、可靠性和健壮性，所以几乎在所有基于 Linux 的系统（包括台式机、服务器和工作站 — 并且甚至一些嵌入式设备）上都使用 Ext2 文件系统。然而，当在嵌入式设备中使用 Ext2fs 时，它有一些缺点：

1）Ext2fs 是为像 IDE 设备那样的块设备设计的，这些设备的逻辑块大小是 512 字节，1 K 字节等这样的倍数。这不太适合于扇区大小因设备不同而不同的闪存设备。

2）Ext2 文件系统没有提供对基于扇区的擦除 / 写操作的良好管理。在 Ext2fs 中，为了在一个扇区中擦除单个字节，必须将整个扇区复制到 RAM，然后擦除，然后重写入。考虑到闪存设备具有有限的擦除寿命（大约能进行 100,000 次擦除），在此之后就不能使用它们，所以这不是一个特别好的方法。

3）在出现电源故障时，Ext2fs 不是防崩溃的。

Ext2 文件系统不支持损耗平衡，因此缩短了扇区 / 闪存的寿命。（损耗平衡确保将地址范围的不同区域轮流用于写和 / 或擦除操作以延长闪存设备的寿命。）

Ext2fs 没有特别完美的扇区管理，这使设计块驱动程序十分困难。

由于这些原因，通常相对于 Ext2fs，在嵌入式环境中使用 MTD/JFFS2 组合是更好的选择。

♦ 文件系统 EXT4

Ext4(The fourth extended file system)是一种针对 ext3 系统的扩展日志式文件系统,是专门为 Linux 开发的原始的扩展文件系统(ext 或 extfs)的第四版。Linux kernel 自 2.6.28 开始正式支持 Ext4。 兼容性 Ext3 升级到 ext4 能提供系统更高的性能，消除存储限制，和获取新的功能，并且不需要重新格式化分区，ext4 会在新的数据上用新的文件结构，旧的文件保留原状。以 ext3 文件系统的方式 mount 到 ext4 上会不用新的磁盘格式，而且还能再用 ext3 来重新挂载，这样仅仅失去了 ext4 的优势。 大文件系统/文件大小 现在 ext3 支持最大 16TB 的文件系统。单个文件最大 2TB。Ext4 增加了 48 位块地址，最大支持 1EB 文件系统，和单个 16TB 的文件。1EB=1024PB 1PB=1024TB 1TB=1024GB。

Ext3 每个目录最大包含 32000 个子目录。减去.和..也就是 31998，ext4 打破了这个限制可以无限制数量的子目录 Extents 传统的 Unix 派生文件系统，使用间接块映射方案，以跟踪每块用于块相应的数据文件，这对大文件来说是低效率的。现代文件系统使用不同的方法称为“extends” 。extends 基本上指一串连续的物理快。通常叫做数据在下面的 N 个块中。一个 100MB 的文件可以分配到一 extends 中。而不需要创建一个间接映射表为 256000 个 blocks，(每个 block 4kb)大文件被分割为很多范围。Extends 改进了性能并且减少了碎片。因为 extends 鼓励在磁盘上连续布局。

Ext4 使用了一个多块分配来在一次调用中分配很多块。替换掉每次分配一个块，减少系统开销。在延时分配和 extends 上十分有用。 延迟分配 延迟分配是一个性能特性(它不修改磁盘格式)。现代文件系统例如 XFS，ZFS，btrfs or Reiser 4 并且它尽可能多的延时块的分配。相对于传统文件系统(例如 Ext3 reiser3 etc)所做的立刻分配块提供了更好的性能。

日志是磁盘中最常用的一部分，组成日志的这部分也是最容易出现硬件故障的。并且从

一个受损的日志上恢复数据会导致巨大的数据损坏。Ext4 校验和提供了改进，它允许将 EXt3 的双向提交日志格式转换为单向的，加速文件系统操作，在某些情况下达到 20%---因此可靠性和性能同时被改进。 在线碎片整理 当 延迟分配，extents 和 multiblock 分配 帮助减少碎片的时候，使用中的文件系统依然会产生碎片。为了解决这个问题，Ext4 将会支持在线碎片整理，并且有一个 e4defrag 工具可以整理个别文件在整个文件系统中。
索引节点-关联 拥有大索引节点，纳秒时间戳，快速扩展属性，索引节点保留的特点。Ext3 支持可设置的索引节点大小(通过 mkfs -l 参数)但是默认索引节点大小是 128 字节。Ext4 将默认为 256 字节。这需要适应一些额外的字段(比如纳秒时间戳或者索引节点版本)并且剩余索引节点空间会被用于存放扩展属性为那些足够小的对象来适合空间。这样使访问那些属性更快，提高那些使用扩展属性的应用程序性能 3-7 倍。在目录创建的时候保留若干索引节点在里面，预期它们在未来使用。这样改进性能，因为新文件被创建在目录中他们可以使用保留索引节点。文件创建和删除因此更高效。纳秒时间戳意味着索引节点字段例如修改时间可以用纳秒

持续预分配，这个特性在 ext3 最新的内核版本中已经可用了，由 glic 仿真来实现文件系统不支持的功能，允许应用程序预分配磁盘空间：应用程序告诉文件系统来预分配空间，文件系统预分配需要的块和数据结构。

♦ 日志闪存文件系统 JFFS2

瑞典的 Axis Communications 开发了最初的 JFFS，Red Hat 的 David Woodhouse 对它进行了改进。 第二个版本，JFFS2，作为用于微型嵌入式设备的原始闪存芯片的实际文件系统而出现。JFFS2 文件系统是日志结构化的，这意味着它基本上是一长列节点。每个节点包含有关文件的部分信息 — 可能是文件的名称、也许是一些数据。相对于 Ext2fs，JFFS2 因为有以下这些优点而在无盘嵌入式设备中越来越受欢迎：

1）JFFS2 在扇区级别上执行闪存擦除 / 写 / 读操作要比 Ext2 文件系统好。

2）JFFS2 提供了比 Ext2fs 更好的崩溃 / 掉电安全保护。当需要更改少量数据时，Ext2 文件系统

将整个扇区复制到内存（DRAM）中，在内存中合并新数据，并写回整个扇区。这意味着为了更改单个字，必须对整个扇区（64 KB）执行读 / 擦除 / 写例程 — 这样做的效率非常低。要是运气差，当正在 DRAM 中合并数据时，发生了电源故障或其它事故，那么将丢失整个数据集合，因为在将数据读入 DRAM 后就擦除了闪存扇区。JFFS2 附加文件而不是重写整个扇区，并且具有崩溃 / 掉电安全保护这一功能。

这可能是最重要的一点：JFFS2 是专门为像闪存芯片那样的嵌入式设备创建的，所以它的整个设计提供了更好的闪存管理。

因为本文主要是写关于闪存设备的使用，所以在嵌入式环境中使用 JFFS2 的缺点很少：

当文件系统已满或接近满时，JFFS2 会大大放慢运行速度。这是因为垃圾收集的问题。

♦ YAFFS 文件系统

YAFFS，Yet Another Flash File System，是一种类似于 JFFS/JFFS2 的专门为 Flash 设计的嵌入式文件系统。与 JFFS 相比，它减少了一些功能，因此速度更快、占用内存更少。

YAFFS 和 JFFS 都提供了写均衡，垃圾收集等底层操作。它们的不同之处在于：

JFFS 是一种日志文件系统，通过日志机制保证文件系统的稳定性。YAFFS 仅仅借鉴了日志系统的思想，不提供日志机能，所以稳定性不如 JAFFS，但是资源占用少。

JFFS 中使用多级链表管理需要回收的脏块，并且使用系统生成伪随机变量决定要回收的块，通过这种方法能提供较好的写均衡，在 YAFFS 中是从头到尾对块搜索，所以在垃圾收集上 JFFS 的速度慢，但是能延长 NAND 的寿命。

JFFS 支持文件压缩，适合存储容量较小的系统；YAFFS 不支持压缩，更适合存储容量大的系统。

YAFFS 还带有 NAND 芯片驱动，并为嵌入式系统提供了直接访问文件系统的 API，用户可以不使用 Linux 中的 MTD 和 VFS，直接对文件进行操作。NAND Flash 大多采用 MTD+YAFFS 的模式。MTD（ Memory Technology Devices，内存技术设备）是对 Flash 操作的接口，提供了一系列的标准函数，将硬件驱动设计和系统程序设计分开。

♦ **tmpfs**

当 Linux 运行于嵌入式设备上时，该设备就成为功能齐全的单元，许多守护进程会在后台运行并生成许多日志消息。另外，所有内核日志记录机制，象 syslogd、dmesg 和 klogd，会在 /var 和 /tmp 目录下生成许多消息。由于这些进程产生了大量数据，所以允许将所有这些写操作都发生在闪存是不可取的。由于在重新引导时这些消息不需要持久存储，所以这个问题的解决方案是使用 tmpfs。

tmpfs 是基于内存的文件系统，它主要用于减少对系统的不必要的闪存写操作这一唯一目的。因为 tmpfs 驻留在 RAM 中，所以写 / 读 / 擦除的操作发生在 RAM 中而不是在闪存中。因此，日志消息写入 RAM 而不是闪存中，在重新引导时不会保留它们。tmpfs 还使用磁盘交换空间来存储，并且当为存储文件而请求页面时，使用虚拟内存（VM）子系统。

tmpfs 的优点包括：

动态文件系统大小 — 文件系统大小可以根据被复制、创建或删除的文件或目录的数量来缩放。使得能够最理想地使用内存。

速度 — 因为 tmpfs 驻留在 RAM，所以读和写几乎都是瞬时的。即使以交换的形式存储文件，I/O 操作的速度仍非常快。

tmpfs 的一个缺点是当系统重新引导时会丢失所有数据。因此，重要的数据不能存储在 tmpfs 上。

♦ **cramfs 的特点**

在嵌入式的环境之下，内存和外存资源都需要节约使用。如果使用 RAMDISK 方式来使用文件系统，那么在系统运行之后，首先要把外存(Flash)上的映像文件解压缩到内存中，构造起 RAMDISK 环境，才可以开始运行程序。但是它也有很致命的弱点。在正常情况下，同样的代码不仅在外存中占据了空间(以压缩后的形式存在)，而且还在内存中占用了更大的空间(以解压缩之后的形式存在)，这违背了嵌入式环境下尽量节省资源的要求。

使用 cramfs 就是一种解决这个问题的方式。cramfs 是一个压缩式的文件系统，它并不需要一次性地将文件系统中的所有内容都解压缩到内存之中，而只是在系统需要访问某个位

置的数据的时侯，马上计算出该数据在 cramfs 中的位置，将其实时地解压缩到内存之中，然后通过对内存的访问来获取文件系统中需要读取的数据。cramfs 中的解压缩以及解压缩之后的内存中数据存放位置都是由 cramfs 文件系统本身进行维护的，用户并不需要了解具体的实现过程，因此这种方式增强了透明度，对开发人员来说，既方便，又节省了存储空间。

cramfs 拥有以下一些特性：

1）采用实时解压缩方式，但解压缩的时侯有延迟。

2） cramfs 的数据都是经过处理、打包的，对其进行写操作有一定困难。所以 cramfs 不支持写操作，这个特性刚好适合嵌入式应用中使用 Flash 存储文件系统的场合。

3）在 cramfs 中，文件最大不能超过 16MB。

4）支持组标识(gid)，但是 mkcramfs 只将 gid 的低 8 位保存下来，因此只有这 8 位是有效的。

5）支持硬链接。但是 cramfs 并没有完全处理好，硬链接的文件属性中，链接数仍然为 1.

6）cramfs 的目录中，没有“.”和“..”这两项。因此，cramfs 中的目录的链接数通常也仅有一个。

cramfs 中，不会保存文件的时间戳(timestamps)信息。当然，正在使用的文件由于 inode 保存在内存中，因此其时间可以暂时地变更为最新时间，但是不会保存到 cramfs 文件系统中去。

当前版本的 cramfs 只支持 PAGE_CACHE_SIZE 为 4096 的内核。因此，如果发现 cramfs 不能正常读写的时侯，可以检查一下内核的参数设置。

使用 cramfs

下面是 mkcramfs 的命令格式：

mkcramfs [-h] [-e edition] [-i file] [-n name] dirname outfile

mkcramfs 的各个参数解释如下：

-h：显示帮助信息
-e edition：设置生成的文件系统中的版本号

-i file：将一个文件映像插入这个文件系统之中(只能在 Linux2.4.0 以后的内核版本中使用)

-n name：设定 cramfs 文件系统的名字

dirname：指明需要被压缩的整个目录树

outfile：最终输出的文件
cramfsck 的命令格式：

cramfsck [-hv] [-x dir] file

cramfsck 的各个参数解释如下：

-h：显示帮助信息

-x dir：释放文件到 dir 所指出的目录中

-v：输出信息更加详细

file：希望测试的目标文件

♦ **嵌入式 Linux 中的 MTD 驱动层**

要使用 Cramfs 或 YAFFS 文件系统，离不开 MTD 驱动程序层的支持。MTD(Memory Technology Device)是 Linux 中的一个存储设备通用接口层， 虽然也可以建立在 RAM 上，但它是专为基于 Flash 的设备而设计的。MTD 包含特定 Flash 芯片的驱动程序，并且越来越多的芯片驱动正被添加进来。用户要使用 MTD，首先要选择适合自己系统的 Flash 芯片驱动。Flash 芯片驱动向上层提供读、写、擦除等基本的 Flash 操作方法。MTD 对这些操作进行封装后向用户层提供 MTD char 和 MTD block 类型的设备 MTD char 类型的设备包括 /dev/mtd0 ， /dev/mtd1 等・它们提供对 Flash 的原始字符访问;MTD block 类型的设备包括 /dev/ mtdblock0、/dev/ mtdblockl 等 MTD block 设备是将 Flash 模拟成块设备。这样可以在这些模拟的块设备上创建像 YAFFS 或 Cramfs 等格式的文件系统。

另外，MTD 支持 CFI (Common Flash Interface)接口。利用它可以在一块 Flash 存储芯片上创建多个 Flash 分区。每一个分区作为一个 MTD block 设备，可以把系统软件和数据等分配到不同的分区上， 同时可以在不同的分区上采用不同的文件系统格式。分区的方法在下文中有详细介绍。

4.3 根文件系统相关

♦ **根文件系统的目录**

如果您熟悉 Linux 操作系统环境，您应该熟悉 Linux 下的根文件系统目录结构。文件系统的顶层目录有其习惯的用法和目的，下边的列表 4.3 显示了文件系统目录结构及其习惯用法。

目录	习惯用法
bin	用户命令所在目录
dev	硬件设备文件及其它特殊文件
etc	系统配置文件，包括启动文件等
home	多用户主目录
lib	链接库文件目录
mnt	装配点,用于装配临时文件系统或其他的文件系统
opt	附加的软件套件目录
proc	虚拟文件系统，用来显示内核及进程信息
root	root 用户主目录
sbin	系统管理员命令目录
tmp	临时文件目录
usr	用户命令目录
var	监控程序和工具程序所存放的可变数据

表 4.3 根文件系统目录

对于用途单一的嵌入式系统，上边的一些用于多用户的目录可以省略，例如/home、/opt、/root 目录等。而/bin、/dev、/etc、/lib、/proc、/sbin 和/usr 目录，是几乎每个系统必备的目录，也是不可或缺的目录。

♦ **Busybox 工具集**

小型的嵌入式 Linux 系统制作 root 根文件系统时有一个常用的利器：BusyBox。Busybox 是 Debian GNU/Linux 的大名鼎鼎的 Bruce Perens 首先开发，使用在 Debian 的安装程序中。后来又有许多 Debian developers 贡献力量，这其中尤推 busybox 目前的维护者 Erik Andersen，他患有癌症，可是却是一名优秀的自由软件开发者。

Busybox 编译出一个单个的独立执行程序，就叫做 busybox。但是它可以方便的进行配置，执行 ash shell 的功能，以及几十个各种小应用程序的功能。这其中包括有一个迷您的 vi 编辑器，以及其他诸如 sed, ifconfig, mkdir, mount, ln, ls, echo, cat ... 等等这些都是一个正常的系统上必不可少的工具，但是如果我们把这些程序的原件拿过来的话，它们的体积加在一起，让人吃不消。可是 busybox 有全部的这么多功能，大小也不过 100K 左右。而且，用户还可以根据自己的需要，决定到底要在 busybox 中编译进哪几个应用程序的功能。这样的话，busybox 的体积就可以进一步缩小了。Busybox 的具体配置和编译将在实验部分介绍。

♦ **ICS-IOT-CEP 全功能物联网教学科研平台 Linux 文件系统构建方案**

1）Ext4 根文件系统：

根文件系统是系统启动时挂载的第一个文件系统，其他的文件系统需要在根文件系统目录中建立节点后再挂载。

CBT-6818ICS-IOT-CEP 全功能物联网教学科研平台设备有一个 8G 大小的 emmc，根文件系统和用户文件系统建立在该 flash 的后大半部分。该 flash 的前小半部分用来存放 bootloader 和 kernel 映像。根文件系统选用了 ext4 文件系统格式。

2）其他文件系统格式：

系统采用 YAFFS2 格式的根文件系统，为了配合系统的其他功能，如自动创建设备节点等功能，ICS-IOT-CEP 全功能物联网教学科研平台上基于根文件系统还挂接了其他几种不同的文件系统格式，如 proc 文件系统，tmp 文件系统，sys 文件系统等。

通过上述介绍，我们可以了解到，实际上一个嵌入式 Linux 系统中，通常采用一种格式的根文件系统与其他不同格式文件系统混合的方式进行构建的。

5. 实验步骤

♦ **构建 QT 文件系统**

1）进入 buildroot 目录

进入本次根文件系统实验目录。

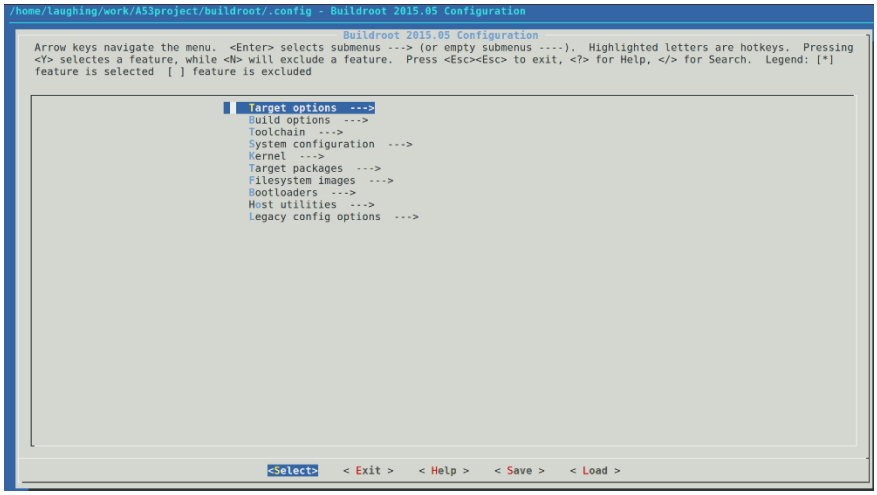
```
cbt@Cyb-Bot:~$ cd /CBT-6818/SRC/linux/buildroot
```

```
cbt@Cyb-Bot: /CBT-6818/SRC/linux/buildroot $
```

2）配置

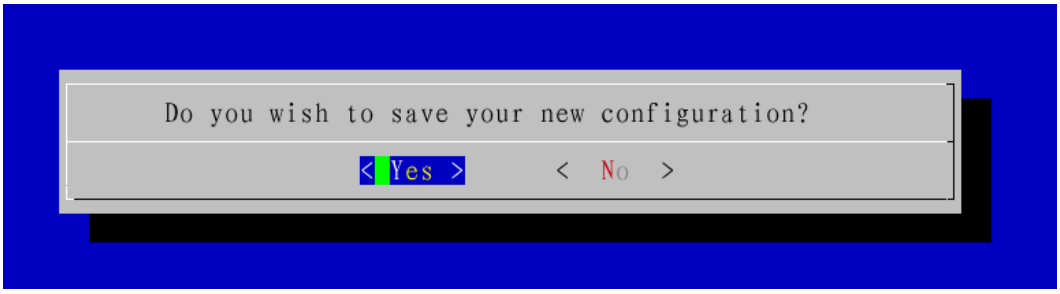
通过 make menuconfig 命令进入菜单，进行相应功能模块的订制与裁剪。

```
cbt@Cyb-Bot: /CBT-6818/SRC/linux/buildroot $make menuconfig
```



该文件系统默认的配置已经在出厂时候完成，用户可以简单尝试进入其中目录去学习和了解相关模块的内容，如果对其中的模块不是很熟悉，建议不要随意修改内核配置，以免导致无法正常启动。

如果做了配置的修改，在退出时，系统会提示保存，保存后新的配置即生效。



通过编译脚本编译 buildroot，编译完成后会在 out/release 目录下生成文件系统的 qt-rootfs.img 的镜像。

```
cbt@Cyb-Bot: /CBT-6818/SRC/linux/buildroot $ cd ../
cbt@Cyb-Bot: /CBT-6818/SRC/linux/$ ./mk -b
cbt@Cyb-Bot: /CBT-6818/SRC/linux/ $ ls out/release
qt-rootfs.img
```

♦ 下载 Linux 根文件系统

1）硬件连接

打开 Cortex-A53 网关，将 Cortex-A53 的串口 0 用串口线与电脑连接，将 Cortex-A53 的 OTG 接口通过 miniUSB 线与电脑连接

2）进入 bootloader 界面

电脑打开 XSHELL 程序，根据自己的电脑配置 COM 和波特率 115200，连接后启动 Cortex-A53，在启动后 3 秒读数时敲键盘任意键进入 bootloader 界面，输入 fastboot 命令

```
X6818# fastboot

Fastboot Partitions:
 mmc.2: ubootpak, img : 0x200, 0x78000
 mmc.2: 2ndboot, img : 0x200, 0x4000
 mmc.2: bootloader, img : 0x8000, 0x70000
 mmc.2: boot, fs : 0x100000, 0x4000000
 mmc.2: system, fs : 0x4100000, 0x2f200000
 mmc.2: cache, fs : 0x33300000, 0x1ac00000
 mmc.2: misc, fs : 0x4e000000, 0x800000
 mmc.2: recovery, fs : 0x4e900000, 0x1600000
 mmc.2: userdata, fs : 0x50000000, 0x0
Support fstype : 2nd  boot  factory  raw  fat  ext4  emmc  nand  ubi  ubifs
Reserved part  : partmap  mem  env  cmd
DONE: Logo bmp 311 by 300 (3bpp), len=280854
DRAW: 0x47000000 -> 0x46000000
Load USB Driver: android
Core usb device tie configuration done
OTG cable Connected!
----------------------------------------
```

3）进入 fastboot 目录烧写

打开计算机运行进入 cmd 目录下，进入光盘资料中\Compnents\Cortex-A53\IMG\fastboot

```
命令提示符
(c) 2017 Microsoft Corporation。保留所有权利。

C:\Users\laugh>D:

D:\>cd D:\开发项目\A53\CBT-6818光盘\Components\Cortex-A53\IMG\fastboot

D:\开发项目\A53\CBT-6818光盘\Components\Cortex-A53\IMG\fastboot>
```

4）烧写文件系统

在命令提示附下输入命令 fastboot flash system ..\Linux\qt-rootfs.img

```
D:\开发项目\A53\CBT-6818光盘\Components\Cortex-A53\IMG\fastboot>fastboot
 flash system  ..\Linux\qt-rootfs.img
sending 'system' (524288 KB)...
```

第五章. 设备底层驱动实验

Linux 中的驱动设计是嵌入式 Linux 开发中十分重要的部分，它要求开发者不仅要熟悉 Linux 的内核机制、驱动程序与用户级应用程序的接口关系、考虑系统中对设备的并发操作等等，而且还要非常熟悉所开发硬件的工作原理。这对驱动开发者提出了比较高的要求。

本章主要介绍了 Linux 系统下驱动程序的一般设计方法，以 ICS-IOT-CEP 全功能物联网教学科研平台 CBT-6818 系统配套的 Linux3.5 版本内核为基础，从典型的字符设备驱动程序设计入手到复杂的驱动程序设计等，通过相关驱动理论知识，结合具体嵌入式系统硬件设备，进行驱动程序的设计与开发。

通过本章的学习，用户可以掌握 Linux 系统下驱动程序设计的基本方法。由于驱动程序的设计与底层硬件依赖关系紧密，要求用户有一定的硬件基本知识以及掌握一定的内核程序设计开发方法。用户可以通过本产品提供的硬件数据手册以及阅读 Linux 内核源码来深入了解驱动程序的设计。

实验一. 按键中断驱动及控制

1. 实验目的

- 了解 ARM 设备外围按键设备电路结构与接口原理。
- 熟悉 Linux 系统下硬件中断驱动编程方法。

2. 实验环境

- 硬件：A53 智能网关实验平台，PC 机 Pentium 500 以上，硬盘 40G 以上，内存大于 256M
- 软件：Vmware Workstation +ubuntu + MiniCom/超级终端 +ARM-LINUX 交叉编译开发环境
- 实验目录：/CBT-6818/SRC/exp/driver/01_keypad

3. 实验内容

- 阅读 CBT-6818 系统配套硬件说明，学习 S5P6818 处理器数据手册 GPIO 相关章节，编写按键中断驱动程序及测试应用程序，实现按键设备的使用。

4. 实验原理

4.1 硬件接口原理

♦ **平台上硬件接口的连接**

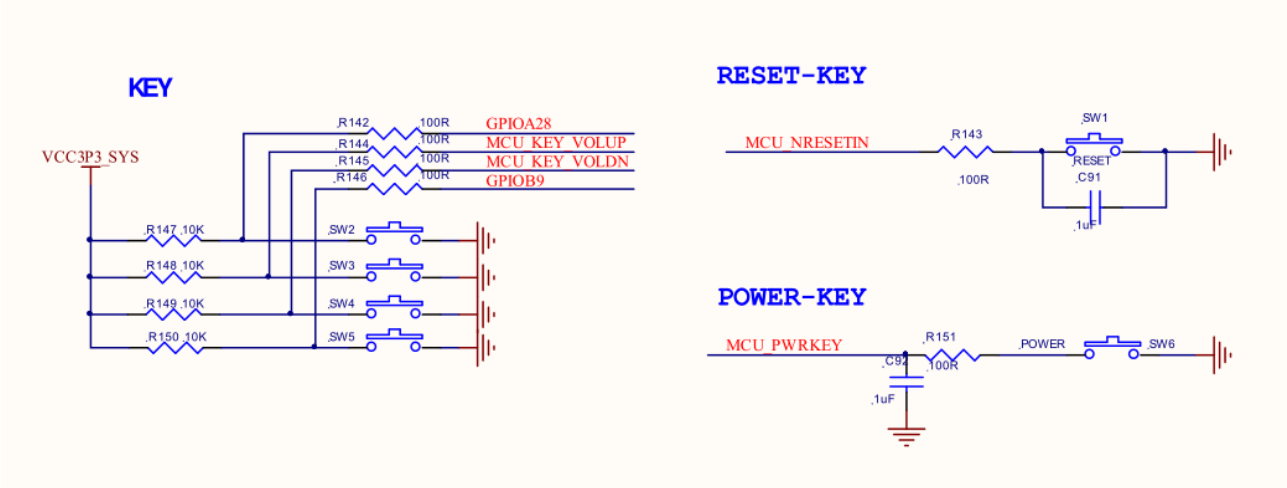


图 4.1.1 KeyPad 引脚 IO

如图 4.1.1 所示，平台上 S5P6818 处理器一共连接有 6 个按键，其共阴极接地。分别连

接到 S5P6818 处理器的终端引脚上。

♦ **S5P6818 处理器相关寄存器**

Name	Bit	Type	Description	Reset Value
GPIOXOUTENB	[31:0]	RW	GPIOx[31:0]: Specifies GPIOx In/Out mode. The Open drain pins are operated in Input/Output mode by the GPIOxOUTPUT register (GPIOxOUT) and not by this bit. 0 = Input Mode 1 = Output Mode	32'h0

表 4.1.1 S5P6818 寄存器组

如上述寄存器表所示，平台上 S5P6818 处理器连接有 6 个按键分别对应处理器 6 个 GPIO。且上述 6 个 GPIO 支持外部中断功能，因此，需要在编写驱动程序的时候初始化上述 6 个 GPIO 为相应的外部中断功能，即可使用按键设备。

4.2 软件接口介绍

♦ **Linux 混杂设备模型**

在 Linux 系统中，存在一类字符设备，他们共享一个主设备号（10），但次设备号不同，我们称这类设备为混杂设备（miscdeivce），查看/proc/device 中可以看到一个名为 misc 的主设备号为 10.所有的混杂设备形成一个链表，对设备访问时内核根据次设备号找到对应的 miscdevice 设备。

Linux 内核使用 struct miscdeivce 来描述一个混杂设备。

```
struct miscdevice   {
        int minor;
        const char *name;
        const struct file_operations *fops;
        struct list_head list;
        struct device *parent;
        struct device *this_device;
        const char *nodename;
        mode_t mode;
};
```

minor 是这个混杂设备的次设备号，若由系统自动配置，则可以设置为 MISC_DYNANIC_MINOR，name 是设备名.使用时只需填写 minor 次设备号，*name 设备名，*fops 文件操作函数集即可。

Linux 内核使用 misc_register 函数注册一个混杂设备，使用 misc_deregister 移除一个混杂设备。注册成功后，linux 内核为自动为该设备创建设备节点，在/dev/下会产生相应的节点。

注册函数：

```
int misc_register(struct miscdevice * misc)
```

输入参数：struct miscdevice

返回值：0 表示注册成功。负数 表示未成功。

卸载函数：

```
int    misc_deregister(struct miscdevice *misc)
```

输入参数：struct miscdevice

返回值：0 表示注册成功。负数 表示未成功。

4.3 关键代码分析

驱动程序(kernel/drivers/input/nxp_io_key.c)

```
#define      KEY_STAT_PRESS              (0)
#define      KEY_STAT_RELEASE      (1)
#define      DELAY_WORK_JIFFIES          (1)
#define      PWR_KEY_RESUME_DELAY   (500)      /* ms */

struct key_code {
   struct delayed_work kcode_work;
   struct workqueue_struct *kcode_wq;
   struct key_info *info;
   unsigned int io;
   unsigned int keycode;
   unsigned int val;
   unsigned int keystat;            /* current detect mode */
   unsigned int detect_high;      /* detect edge */
   int irq_disabled;
};

struct key_info {
   struct input_dev *input;
   int    keys;
   struct key_code *code;
   struct delayed_work       resume_work;
   int    resume_delay_ms;
};

#define      CHECK_PWR_KEY_EVENT(io)       nxp_check_pm_wakeup_dev("power key", io)

struct input_dev *key_input = NULL;
EXPORT_SYMBOL_GPL(key_input);

void nxp_key_power_event(void)
```

```
{
    if (key_input) {
            input_report_key(key_input, KEY_POWER, 1);
            input_sync(key_input);
            input_report_key(key_input, KEY_POWER, 0);
            input_sync(key_input);
        }
}
EXPORT_SYMBOL(nxp_key_power_event);

static void nxp_key_event_wq(struct work_struct *work)
{
    struct key_code *code = (struct key_code *)work;
    struct key_info *key = code->info;
    unsigned int keycode = code->keycode;
    int press = 0;
    u_long flags;

    local_irq_save(flags);

    press = gpio_get_value_cansleep(code->io);
    if (code->detect_high)
            press = !press;

    local_irq_restore(flags);

    if(press != code->keystat) {
            code->keystat = press;
            if (KEY_STAT_PRESS == press) {
                    input_report_key(key->input, keycode, 1);
                    input_sync(key->input);
            } else {
                    input_report_key(key->input, keycode, 0);
                    input_sync(key->input);
            }
            pr_debug("key io:%d, code:%4d %s\n", code->io, keycode,
                    (KEY_STAT_PRESS==press?"DN":"UP"));
    }
}

static irqreturn_t nxp_key_irqhnd(int irqno, void *dev_id)
{
    struct key_code *code = dev_id;

    queue_delayed_work(code->kcode_wq,
                    &code->kcode_work, DELAY_WORK_JIFFIES);
```

```
    return IRQ_HANDLED;
}

static void nxp_key_resume_work(struct work_struct *work)
{
    struct key_info *key = container_of(work, struct key_info, resume_work.work);
    struct key_code *code = key->code;
    int i = 0;

    for (i = 0; key->keys > i; i++, code++) {
        if (code->keycode == KEY_POWER &&
            code->irq_disabled) {
            code->irq_disabled = 0;
            enable_irq(gpio_to_irq(code->io));
            break;
        }
    }
}

static int nxp_key_suspend(struct platform_device *pdev, pm_message_t state)
{
    struct key_info *key = platform_get_drvdata(pdev);
    struct key_code *code = key->code;
    int i = 0;

    for (i = 0; key->keys > i; i++, code++) {
        if (code->keycode == KEY_POWER &&
            code->irq_disabled) {
            code->irq_disabled = 0;
            enable_irq(gpio_to_irq(code->io));
            break;
        }
    }

    return 0;
}

static int nxp_key_resume(struct platform_device *pdev)
{
    struct key_info *key = platform_get_drvdata(pdev);
    struct key_code *code = key->code;
    int i = 0;

    PM_DBGOUT("%s\n", __func__);
```

```
    for (i = 0; key->keys > i; i++, code++) {
        if (code->keycode == KEY_POWER &&
            CHECK_PWR_KEY_EVENT(code->io)) {

            int delay = key->resume_delay_ms ? key->resume_delay_ms : PWR_KEY_RESUME_DELAY;

            input_report_key(key->input, KEY_POWER, 1);
            input_sync(key->input);
            input_report_key(key->input, KEY_POWER, 0);
            input_sync(key->input);

            code->irq_disabled = 1;
            disable_irq(gpio_to_irq(code->io));

            schedule_delayed_work(&key->resume_work, msecs_to_jiffies(delay));
        }
    }

    return 0;
}

static int nxp_key_probe(struct platform_device *pdev)
{
    struct nxp_key_plat_data * plat = pdev->dev.platform_data;
    struct key_info *key = NULL;
    struct key_code *code = NULL;
    struct input_dev *input = NULL;
    int i, keys;
    int ret = 0;

    pr_debug("%s (device name:%s, id:%d)\n", __func__, pdev->name, pdev->id);

    key = kzalloc(sizeof(struct key_info), GFP_KERNEL);
    if (! key) {
        pr_err("fail, %s allocate driver info ...\n", pdev->name);
        return -ENOMEM;
    }

    keys = plat->bt_count;
    code = kzalloc(sizeof(struct key_code)*keys, GFP_KERNEL);
    if (NULL == code) {
        pr_err("fail, %s allocate key code ...\n", pdev->name);
        ret = -ENOMEM;
        goto err_mm;
    }
```

```
input = input_allocate_device();
if (NULL == input) {
        pr_err("fail, %s allocate input device\n", pdev->name);
        ret = -ENOMEM;
        goto err_mm;
}

key->input = input;
key->keys = keys;
key->code = code;
key->resume_delay_ms = plat->resume_delay_ms;
key_input = key->input;

INIT_DELAYED_WORK(&key->resume_work, nxp_key_resume_work);

input->name    = "Nexell Keypad";
input->phys = "nexell/input0";
input->id.bustype = BUS_HOST;
input->id.vendor = 0x0001;
input->id.product = 0x0002;
input->id.version = 0x0100;
input->dev.parent = &pdev->dev;
input->keycode = plat->bt_code;
input->keycodesize = sizeof(plat->bt_code[0]);
input->keycodemax = plat->bt_count * 2;     // for long key
input->evbit[0] = BIT_MASK(EV_KEY);

if (plat->bt_repeat)
         input->evbit[0] |= BIT_MASK(EV_REP);

input_set_capability(input, EV_MSC, MSC_SCAN);
input_set_drvdata(input, key);

ret = input_register_device(input);
if (ret) {
        pr_err("fail, %s register for input device ...\n", pdev->name);
        goto err_mm;
}

for (i=0; keys > i; i++, code++) {
        code->io = plat->bt_io[i];
        code->keycode = plat->bt_code[i];
        code->detect_high = plat->bt_detect_high ? plat->bt_detect_high[i]: 0;
        code->val = i;
        code->info = key;
        code->keystat = KEY_STAT_RELEASE;
```

```
        code->kcode_wq = create_singlethread_workqueue(pdev->name);
        if (!code->kcode_wq) {
            ret = -ESRCH;
            goto err_irq;
        }

        ret = request_irq(gpio_to_irq(code->io), nxp_key_irqhnd,
                          (IRQF_SHARED | IRQ_TYPE_EDGE_BOTH), pdev->name, code);
        if (ret) {
            pr_err("fail, gpio[%d] %s request irq...\n", code->io, pdev->name);
            goto err_irq;
        }

        __set_bit(code->keycode, input->keybit);
        INIT_DELAYED_WORK(&code->kcode_work, nxp_key_event_wq);
        pr_debug("[%d] key io=%3d, code=%4d\n", i, code->io, code->keycode);
    }

    platform_set_drvdata(pdev, key);

    return ret;

err_irq:
    for (--i; i >= 0; i--) {
        cancel_work_sync(&code[i].kcode_work.work);
        destroy_workqueue(code[i].kcode_wq);
        free_irq(gpio_to_irq(code[i].io), &code[i]);
    }
    input_free_device(input);

err_mm:
    if (code)
        kfree(code);

    if (key)
        kfree(key);

    return ret;
}

static int nxp_key_remove(struct platform_device *pdev)
{
    struct key_info *key = platform_get_drvdata(pdev);
    struct key_code *code = key->code;
    int i = 0, irq;
```

```
    pr_debug("%s\n", __func__);

    input_free_device(key->input);

    for (i = 0; i < key->keys; i++) {
        cancel_work_sync(&code[i].kcode_work.work);
        destroy_workqueue(code[i].kcode_wq);
        irq = gpio_to_irq(code[i].io);
        free_irq(irq, &code[i]);
    }

    if (code)
        kfree(code);

    if (key)
        kfree(key);

    return 0;
}

static struct platform_driver key_plat_driver = {
    .driver         = {
        .owner    = THIS_MODULE,
        .name     = DEV_NAME_KEYPAD,
    },
    .probe          = nxp_key_probe,
    .remove         = nxp_key_remove,
    .suspend  = nxp_key_suspend,
    .resume         = nxp_key_resume,
};

static int __init nxp_key_init(void)
{
    return platform_driver_register(&key_plat_driver);
}

static void __exit nxp_key_exit(void)
{
    platform_driver_unregister(&key_plat_driver);
}

module_init(nxp_key_init);
module_exit(nxp_key_exit);

MODULE_AUTHOR("jhkim <jhkim@nexell.co.kr>");
```

```
MODULE_DESCRIPTION("Keypad driver for the Nexell board");
MODULE_LICENSE("GPL");
```

以上程序主要通过按键中断和内核定时器完成对按键设备驱动程序的设计。配套的应用程序及其他接口用户可以通过阅读实验源码自行分析。

5. 实验步骤

♦ **在内核中添加按键设备支持**

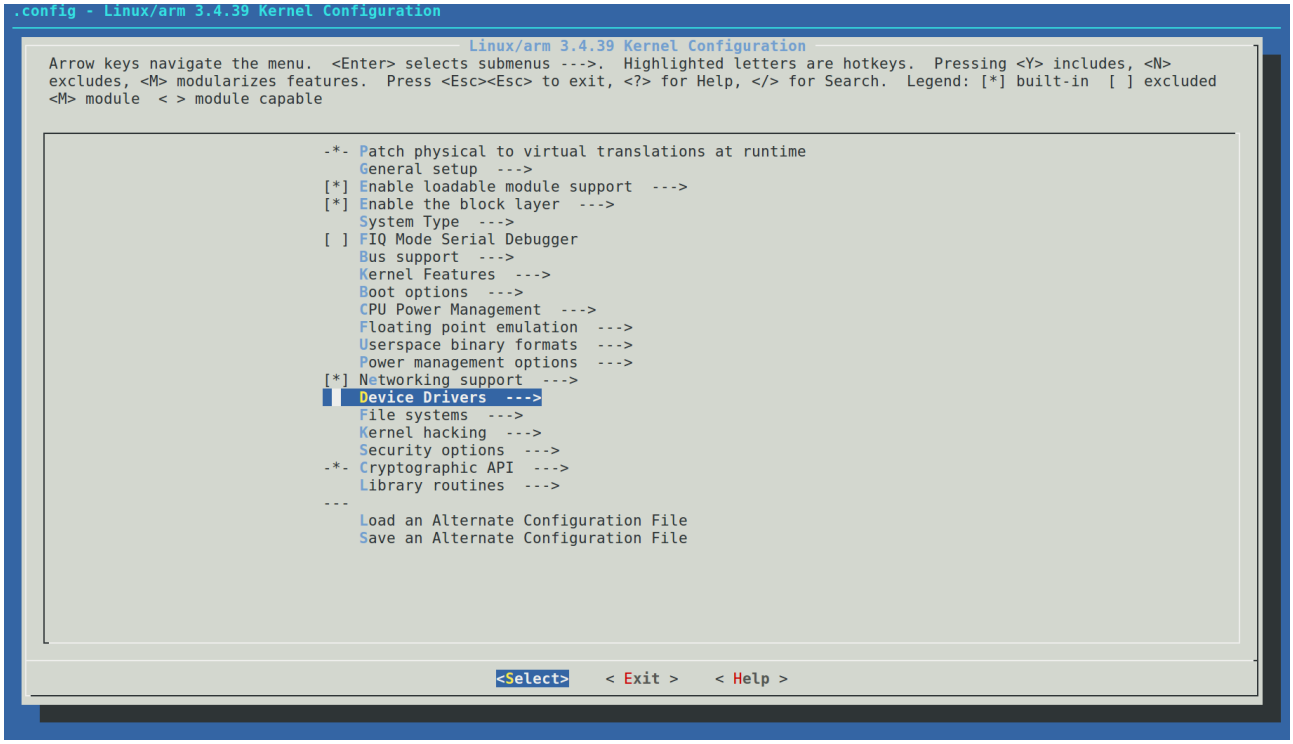
1）进入宿主机中 CBT-6818 型光盘内核目录：

```
cbt@Cyb-Bot:~$ cd /CBT-6818/SRC/linux/kernel
```

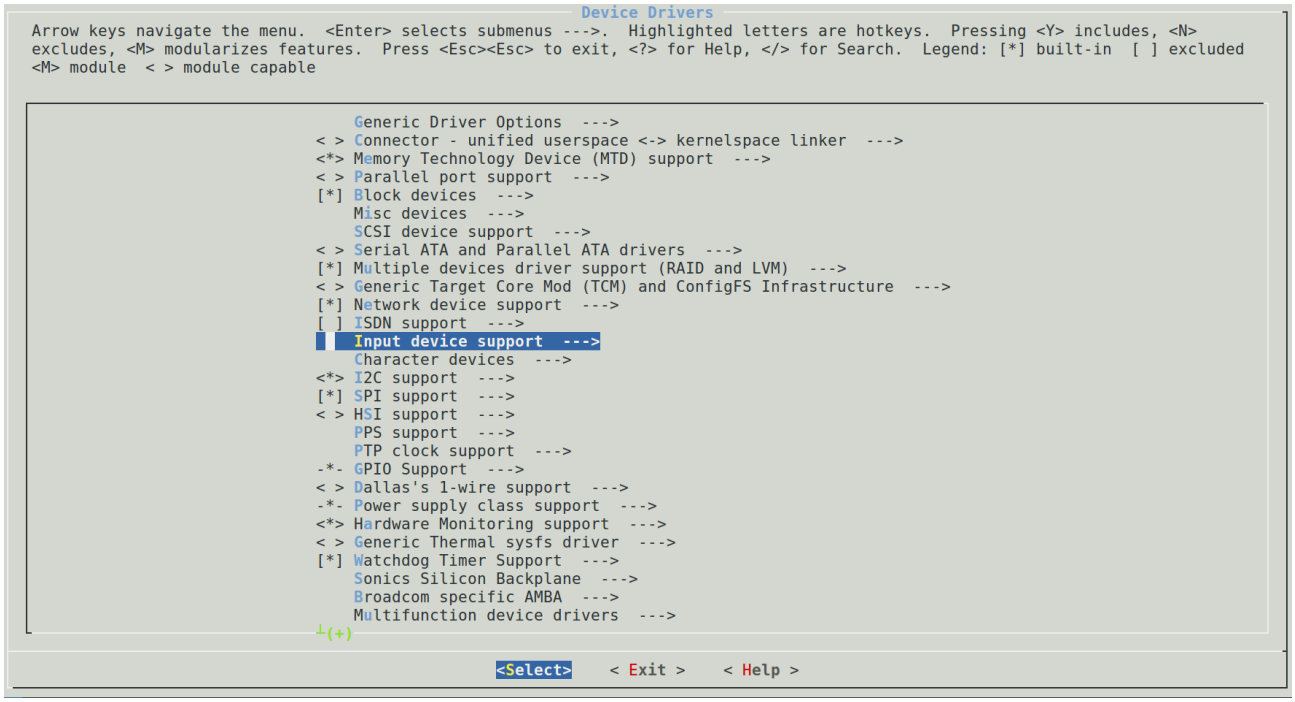
2）运行 make menuconfig 命令配置内核对按键设备的相关支持

```
cbt@Cyb-Bot: /CBT-6818/SRC/linux/kernel $ make menuconfig
```

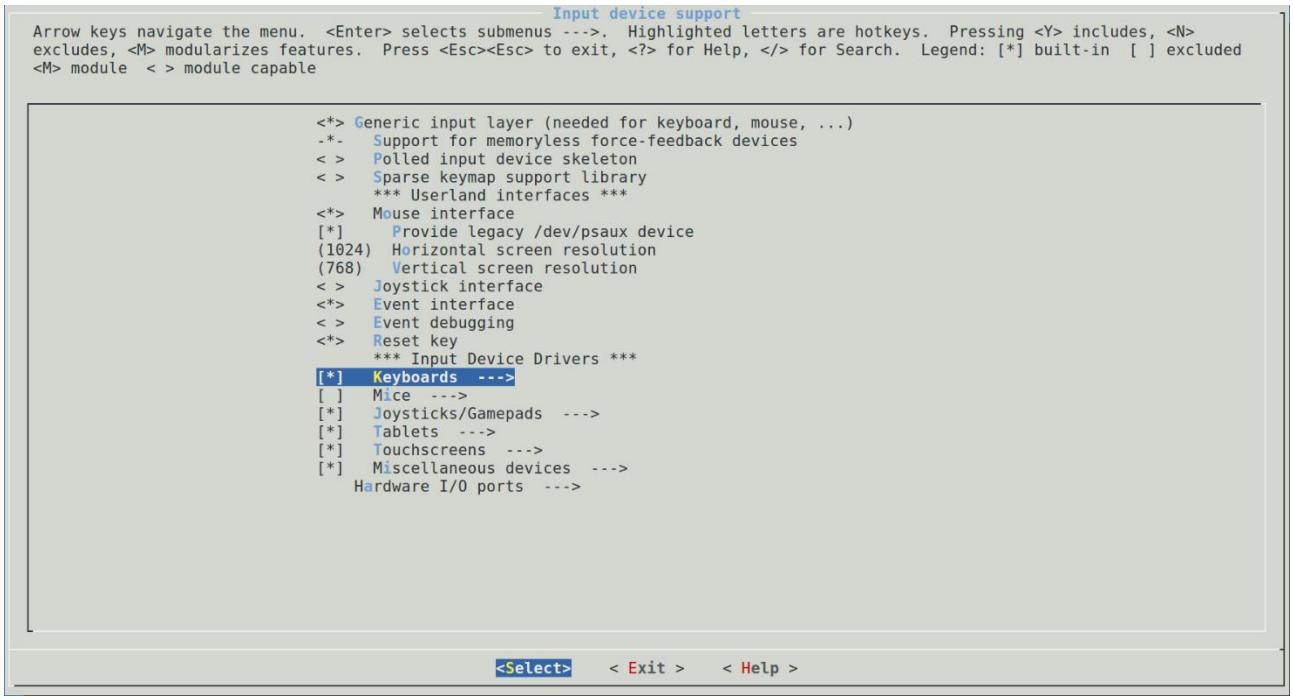
选择 Device Drivers ---->选项，如图



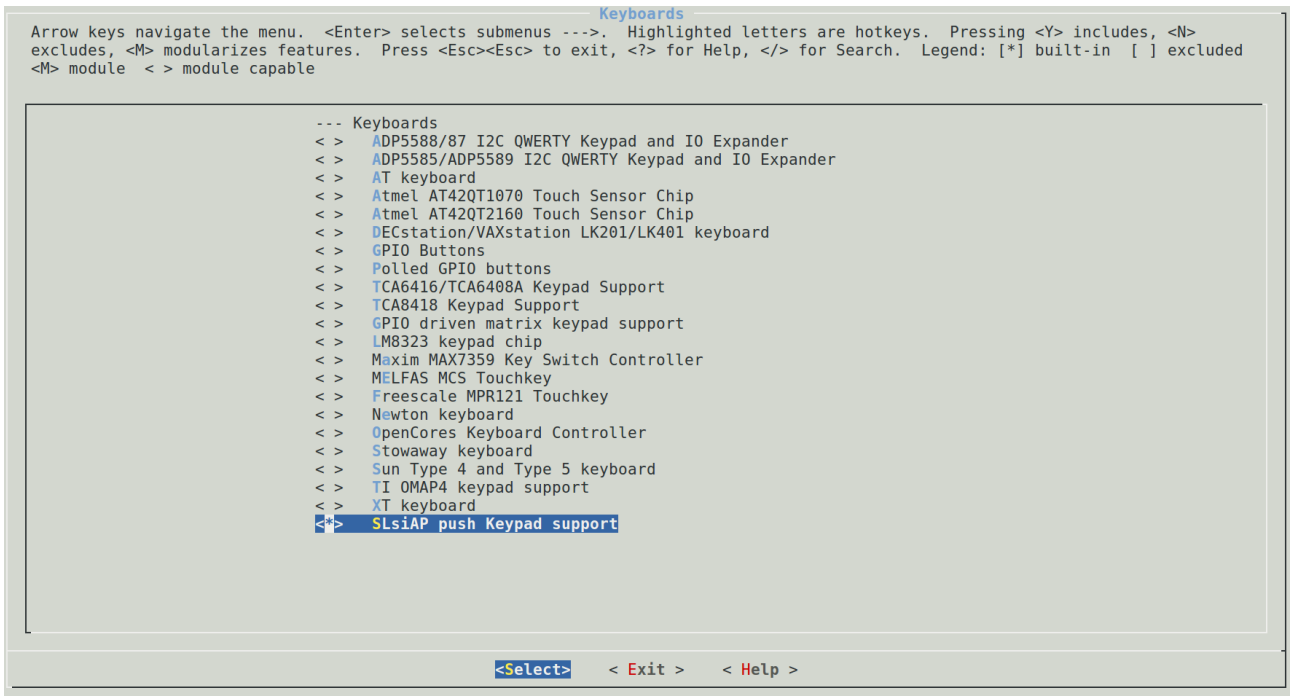
选择 Input device support ---> 如图：



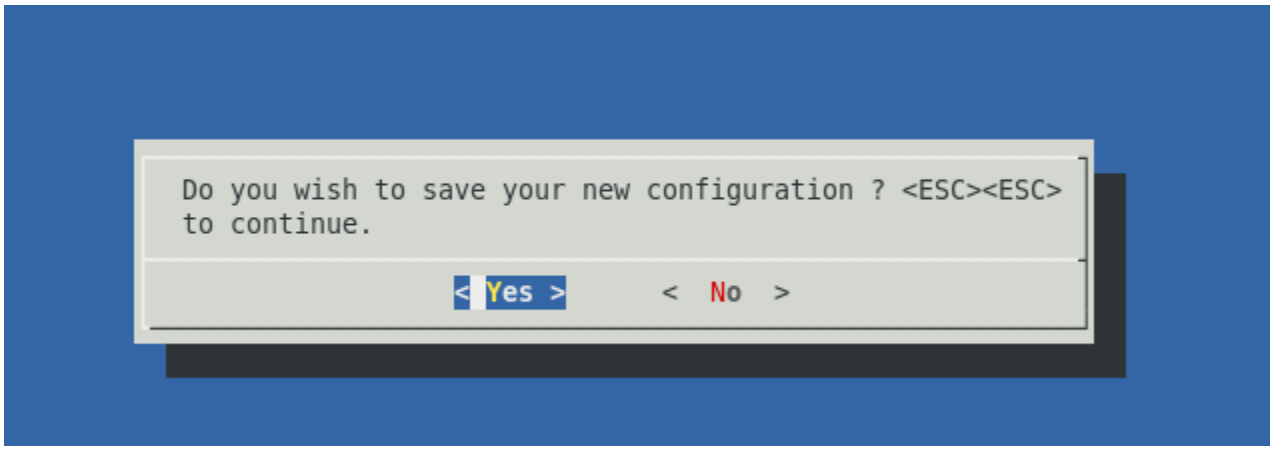
选择 Keyboard ---> 如图：



选择<*>SLsiAP push Keypad suport 如图：



退出保存设置，如图：



3）重新编译内核,运行 make 命令

```
cbt@Cyb-Bot: /CBT-6818/SRC/linux/kernel $ cd ../
cbt@Cyb-Bot: /CBT-6818/SRC/linux$ ./mk -k
  CHK     include/linux/version.h
  CHK     include/generated/utsrelease.h
make[1]: “include/generated/mach-types.h” is up to date
  CALL    scripts/checksyscalls.sh
  CHK     include/generated/compile.h
.....
```

最终在 linux 源码目录的 out/release 目录下生成新的 ARM-LINUX 内核镜像文件 boot.img

4）按照 ICS-IOT-CEP 全功能物联网教学科研平台光盘配套烧写文档将新生成的内核镜像文件 zImage 烧写到 ARM 设备中，这里不在赘述。

备注：以上在内核中添加对按键设备的支持的步骤，在 ARM 系统设备出厂自带内核中已经默认添加进来了，用户可以省略以上步骤。以上步骤在于重现系统的构造。

♦ 编译测试应用程序

1）进入实验目录：

```
cbt@Cyb-Bot:~$ cd /CBT-6818/SRC/exp/driver/01_keypad/
cbt@Cyb-Bot: /CBT-6818/SRC/exp/driver/01_keypad $ ls
keypad_buttons.c    keypad_test    Makefile
cbt@Cyb-Bot: /CBT-6818/SRC/exp/driver/01_keypad $
```

2）清除中间代码，重新编译。

```
cbt@Cyb-Bot: /CBT-6818/SRC/exp/driver/01_keypad $ make clean
rm -rf *.o keypad_test
cbt@Cyb-Bot: /CBT-6818/SRC/exp/driver/01_keypad $ make
arm-linux-gcc -Wall -O2 keypad_buttons.c -o keypad_test
cbt@Cyb-Bot: /CBT-6818/SRC/exp/driver/01_keypad $ ls
driver    keypad_buttons.c    keypad_test    Makefile
cbt@Cyb-Bot: /CBT-6818/SRC/exp/driver/01_keypad $
```

当前目录下生成测试程序 keypad_test。

♦ 运行程序（NFS 方式）

1）启动 CBT-6818 型实验系统，连好网线、串口线。通过串口终端挂载宿主机实验目录。

```
[root@CBT-6818 /$ mount -t nfs -o nolock 192.168.1.7:/CBT-6818/    /mnt
```

2）进入串口终端的 NFS 共享实验目录。

```
[root@CBT-6818 /$ cd /mnt/SRC/exp/driver/01_keypad/
[root@CBT-6818 /mnt/SRC/exp/driver/01_keypad$ ls
Makefile      keypad_buttons.c    keypad_test
[root@CBT-6818 /mnt/SRC/exp/driver/01_keypad$
```

3）执行应用程序测试该驱动及设备

```
[root@CBT-6818 /mnt/SRC/exp/driver/01_keypad$ ./keypad_test
```

♦ 实验结果

```
[root@CBT-6818 /mnt/SRC/exp/driver/01_keypad$ ./keypad_test
key POWER Pressed
key POWER Released
key 1 Pressed
key 1 Released
key 2 Pressed
```

```
key 2 Released
key 3 Pressed
key 3 Released
key 4 Pressed
key 4 Released
```

实验二. PWM 蜂鸣器驱动及控制

1. 实验目的

- 了解 ARM 设备外围按键设备电路结构与接口原理。
- 熟悉 Linux 系统下 PWM 驱动的编程方法。

2. 实验环境

- 硬件：A53 智能网关实验平台，PC 机 Pentium 500 以上，硬盘 40G 以上，内存大于 256M
- 软件：Vmware Workstation +ubuntu + MiniCom/超级终端 +ARM-LINUX 交叉编译开发环境
- 实验目录：/CBT-6818/SRC/exp/driver/02_pwm

3. 实验内容

- 阅读 CBT-6818 系统配套硬件说明，学习 S5P6818 处理器数据手册 GPIO 相关章节，编写 PWM 驱动程序及测试应用程序，实现使用 PWM 调节蜂鸣器音频的功能。

4. 实验原理

4.1 硬件接口原理

♦ **平台上硬件接口的连接**

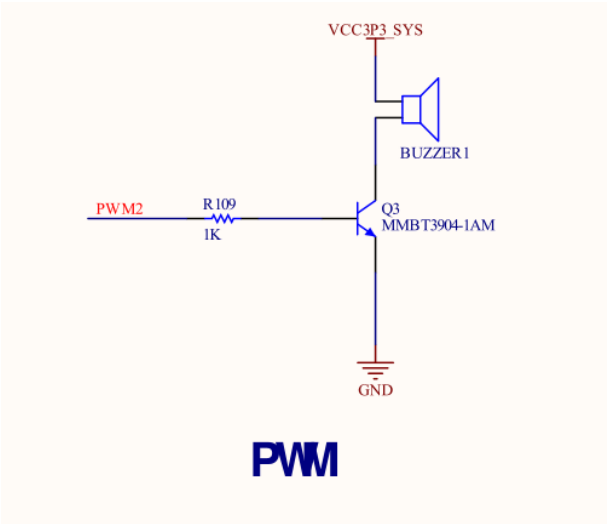


图 4.1.1 PWM 蜂鸣器引脚 IO

如图 4.1.1 所示，平台上 S5P6818 处理器将 PWM2 引脚连接到了一个板载的蜂鸣器

BUZZER。通过程序设置 PWM 电压脉宽占空比即可调试板载蜂鸣器的声响。

♦ **S5P6818 处理器相关寄存器**

Name	Bit	Type	Description	Reset Value
GPIOXOUTENB	[31:0]	RW	GPIOx[31:0]: Specifies GPIOx In/Out mode. The Open drain pins are operated in Input/Output mode by the GPIOxOUTPUT register (GPIOxOUT) and not by this bit. 0 = Input Mode 1 = Output Mode	32'h0

表 4.1.1 S5P6818 GPD0 寄存器组

如上述寄存器表所示，平台上 S5P6818 处理器的 PWM2 对应处理器 GPIO C 组第 15 个引脚，该 IO 具有 TOUT0 PWM 功能，因此，需要在编写驱动程序的时候初始化上述 GPIO 为相应的 PWM 定时器功能，即可使用 PWM 控制蜂鸣器设备。

4.2 软件接口介绍

♦ **Linux 混杂设备模型**

在 Linux 系统中，存在一类字符设备，他们共享一个主设备号（10），但次设备号不同，我们称这类设备为混杂设备（miscdeivce），查看/proc/device 中可以看到一个名为 misc 的主设备号为 10.所有的混杂设备形成一个链表，对设备访问时内核根据次设备号找到对应的 miscdevice 设备。

Linux 内核使用 struct miscdeivce 来描述一个混杂设备。

```
struct miscdevice   {
        int minor;
        const char *name;
        const struct file_operations *fops;
        struct list_head list;
        struct device *parent;
        struct device *this_device;
        const char *nodename;
        mode_t mode;
};
```

minor 是这个混杂设备的次设备号，若由系统自动配置，则可以设置为 MISC_DYNANIC_MINOR，name 是设备名.使用时只需填写 minor 次设备号，*name 设备名，*fops 文件操作函数集即可。

Linux 内核使用 misc_register 函数注册一个混杂设备，使用 misc_deregister 移除一个混杂设备。注册成功后，linux 内核为自动为该设备创建设备节点，在/dev/下会产生相应的节点。

注册函数：

```
int misc_register(struct miscdevice * misc)
```

输入参数：struct miscdevice。

返回值：0 表示注册成功。负数 表示未成功。

卸载函数：

```
int    misc_deregister(struct miscdevice *misc)
```

输入参数：struct miscdevice。

返回值：0 表示注册成功。

负数 表示未成功。

♦ Linux 内核中的 PWM 接口

Linux 内核中已经支持标准的平台 PWM 设备，并更加具体的平台定义和实现了相应的接口完成 PWM 控制。

Linux 内核使用 struct pwm_device 结构来管理 PWM 设备，其定义如下：

```
struct pwm_device {
        struct list_head        list;
        struct platform_device  *pdev;

        struct clk              *clk_div;
        struct clk              *clk;
        const char              *label;

        unsigned int            period_ns;
        unsigned int            duty_ns;

        unsigned char           tcon_base;
        unsigned char           running;
        unsigned char           use_count;
        unsigned char           pwm_id;
};
```

并根据不同硬件平台提供相应的函数支持，如下：

申请一个 PWM 设备

```
struct pwm_device *pwm_request(int pwm_id, const char *label);
```

释放一个 PWM 设备

```
void pwm_free(struct pwm_device *pwm);
```

配置 PWM 设备

```
int pwm_config(struct pwm_device *pwm, int duty_ns, int period_ns);
```

开启 PWM 设备

```
int pwm_enable(struct pwm_device *pwm);
```

关闭 PWM 设备

```
void pwm_disable(struct pwm_device *pwm);
```

只需要会使用上述内核接口，可以很容易实现对 S5P6818 处理器 PWM 外设的控制。

4.3 关键代码分析

驱动程序(kernel/drivers/char/6618_pwm.c)。

```
#define DEVICE_NAME                 "pwm"
#define PWM_NUM                     (3)
#define PWM_IOCTL_SET_FREQ          (0x1)
#define PWM_IOCTL_STOP              (0x0)
#define NS_IN_1HZ                   (1000000000UL)

static int is_pwm_working[PWM_NUM] = {0, 0, 0};
static struct pwm_device *matrix_pwm[PWM_NUM];
static struct semaphore lock;

static int pwm_set_freq(int index, int pin, unsigned long freq)
{
    int ret=0;
    int period_ns = NS_IN_1HZ / freq;
    //int duty_ns = period_ns    / 1000 * duty;

    if ((ret = pwm_config(matrix_pwm[index], period_ns / 2, period_ns))) {
            printk("fail to pwm_config\n");
            return ret;
    }
    if ((ret = pwm_enable(matrix_pwm[index]))) {
            printk("fail to pwm_enable\n");
            return ret;
    }
    return ret;
}

static void pwm_stop(int index, int pin)
{
    pwm_config(matrix_pwm[index], 0, NS_IN_1HZ / 100);
    pwm_disable(matrix_pwm[index]);
}
```

```
static int matrix_pwm_hw_init(int index, int pin)
{
    int ret;

    ret = gpio_request(pin, DEVICE_NAME);
    if (ret) {
            printk("request gpio %d for pwm failed\n", pin);
            return ret;
    }

    gpio_direction_output(pin, 0);
    matrix_pwm[index] = pwm_request(index, DEVICE_NAME);
    if (IS_ERR(matrix_pwm[index])) {
            printk("request pwm%d failed\n", index);
            gpio_free(pin);
            return -ENODEV;
    }

    pwm_stop(index, pin);
    return ret;
}

static void matrix_pwm_hw_exit(int index,int pin)
{
    pwm_stop(index, pin);
    pwm_free(matrix_pwm[index]);
    gpio_free(pin);
}

static int matrix_pwm_open(struct inode *inode, struct file *file) {
    if (!down_trylock(&lock))
            return 0;
    else
            return -EBUSY;
}

static int matrix_pwm_close(struct inode *inode, struct file *file) {
    up(&lock);
    return 0;
}

static long matrix_pwm_ioctl(struct file *filep, unsigned int cmd,unsigned long arg)
{
```

```
int pwm_id;
int param[3];
int gpio;
int freq;
int duty;
int ret;

switch (cmd) {
    case PWM_IOCTL_SET_FREQ:
        if(arg == 0)
            return -EINVAL;

        /*if (copy_from_user(param, (void __user *)arg, sizeof(param)))
            return -EINVAL;
        gpio = param[0];
        freq = param[1];
        duty = param[2];
        if (gpio == (PAD_GPIO_D + 1)) {
            pwm_id = 0;
        } else if (gpio == (PAD_GPIO_C + 13)) {
            pwm_id = 1;
        } else if (gpio == (PAD_GPIO_C + 14)) {
            pwm_id = 2;
        } else {
            printk("Invalid pwm id\n");
            return -EINVAL;
        }

        if (duty < 0 || duty > 1000) {
            printk("Invalid pwm duty\n");
            return -EINVAL;
        }*/

        pwm_id = 2;
        gpio = (PAD_GPIO_C + 14);
        if (is_pwm_working[pwm_id] == 1) {
            matrix_pwm_hw_exit(pwm_id, gpio);
        }

        if ((ret = matrix_pwm_hw_init(pwm_id, (PAD_GPIO_C + 14)))) {
            return ret;
        }
        is_pwm_working[pwm_id] = 1;
        if ((ret = pwm_set_freq(pwm_id, gpio, arg))) {
            return ret;
        }
```

```
            break;
        case PWM_IOCTL_STOP:
        default:
        /*  if (copy_from_user(&gpio, (void __user *)arg, sizeof(gpio)))
                return -EINVAL;
            if (gpio == (PAD_GPIO_D + 1)) {
                pwm_id = 0;
            } else if (gpio == (PAD_GPIO_C + 13)) {
                pwm_id = 1;
            } else if (gpio == (PAD_GPIO_C + 14)) {
                pwm_id = 2;
            } else {
                printk("Invalid pwm id\n");
                return -EINVAL;
            }*/
            pwm_id = 2;
            gpio = (PAD_GPIO_C + 14);
            if (is_pwm_working[pwm_id] == 1) {
                matrix_pwm_hw_exit(pwm_id, gpio);
                is_pwm_working[pwm_id] = 0;
            }
            break;
        //default:
        //  printk("%s unsupported pwm ioctl %d", __FUNCTION__, cmd);
        //  break;
    }

    return 0;
}

static struct file_operations matrix_pwm_ops = {
    .owner              = THIS_MODULE,
    .open               = matrix_pwm_open,
    .release        = matrix_pwm_close,
    .unlocked_ioctl     = matrix_pwm_ioctl,
};

static struct miscdevice matrix_misc_dev = {
    .minor = MISC_DYNAMIC_MINOR,
    .name = DEVICE_NAME,
    .fops = &matrix_pwm_ops,
};

static int __init matrix_pwm_init(void)
{
    int ret;
```

```
    ret = misc_register(&matrix_misc_dev);
    printk("Matirx PWM init\n");

    sema_init(&lock, 1);
    return ret;
}

static void __exit matrix_pwm_exit(void) {
    misc_deregister(&matrix_misc_dev);
    printk("Matirx PWM exit\n");
}

module_init(matrix_pwm_init);
module_exit(matrix_pwm_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("FriendlyARM Inc.");
MODULE_DESCRIPTION("FriendlyARM Matrix PWM Driver");
```

以上程序主要通过内核提供的 PWM 设备驱动接口及 MISC 型设备模型完成对 PWM 设备驱动程序的设计。配套的应用程序及其他接口用户可以通过阅读实验源码自行分析。

5. 实验步骤

♦ **在内核中添加 PWM 设备支持**

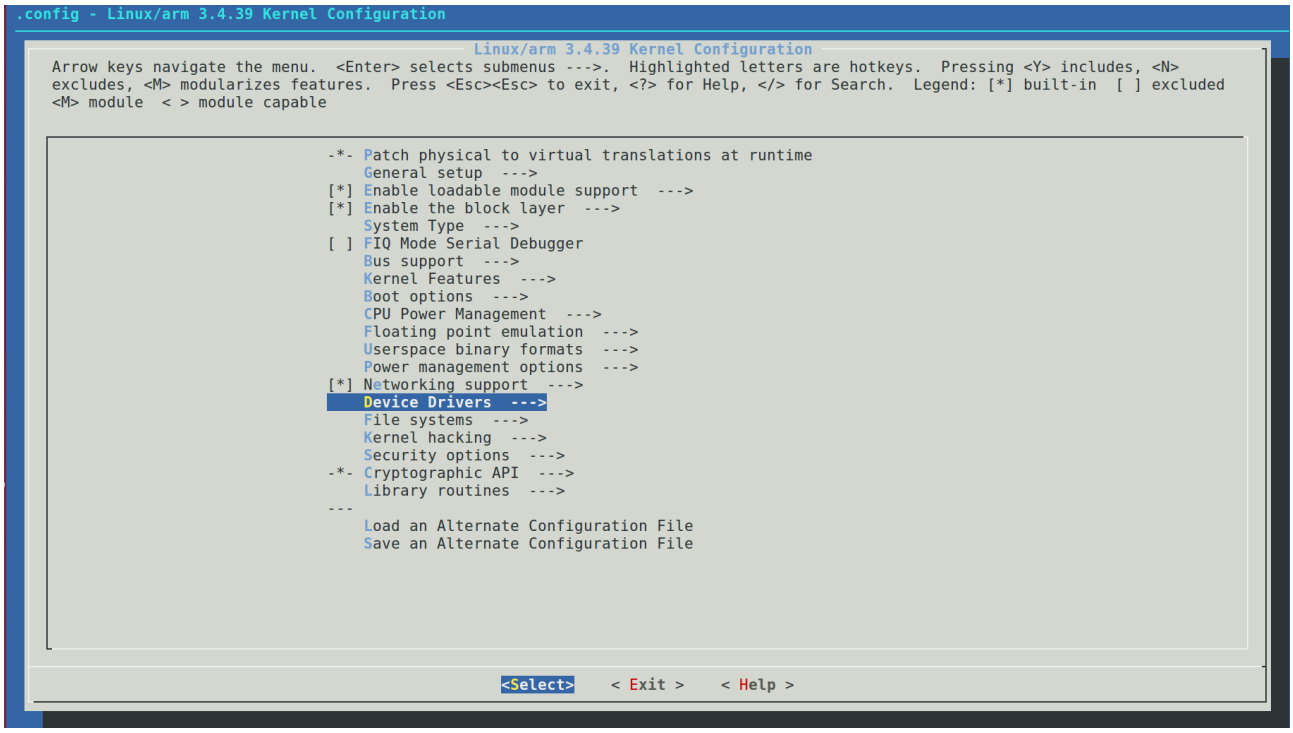
1）进入宿主机中 CBT-6818 型光盘内核目录：

```
cbt@Cyb-Bot:~$ cd /CBT-6818/SRC/linux/kernel
```

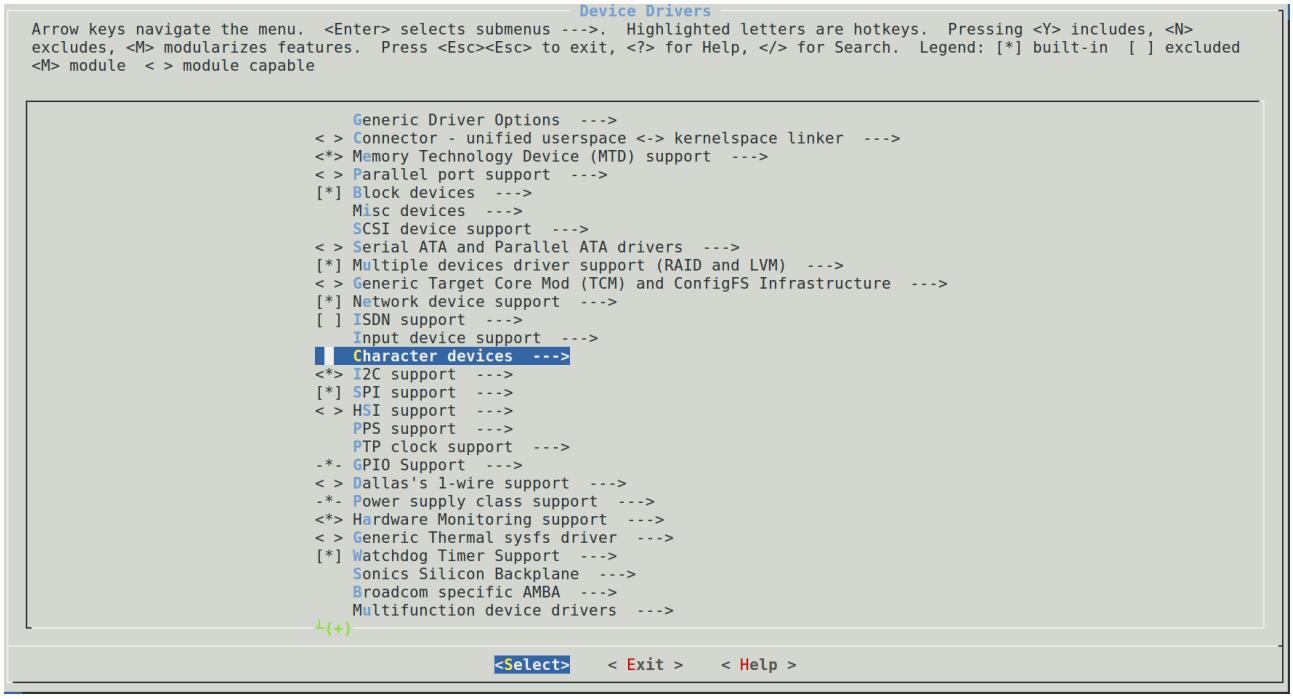
2）运行 make menuconfig 命令配置内核对按键设备的相关支持。

```
cbt@Cyb-Bot: /CBT-6818/SRC/linux/kernel $
cbt@Cyb-Bot: /CBT-6818/SRC/linux/kernel $ cp 6818_linux_config    .config
cbt@Cyb-Bot: /CBT-6818/SRC/linux/kernel $ make menuconfig
```

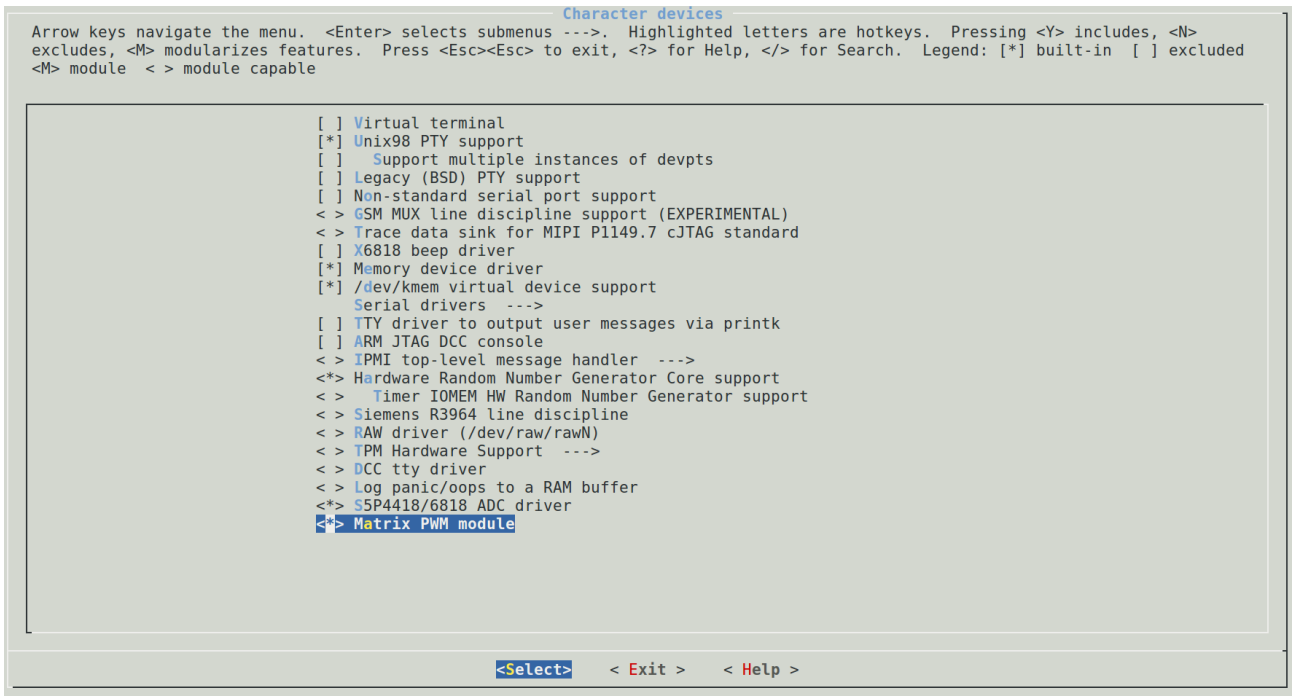
选择 Device Drivers ---->选项，如图



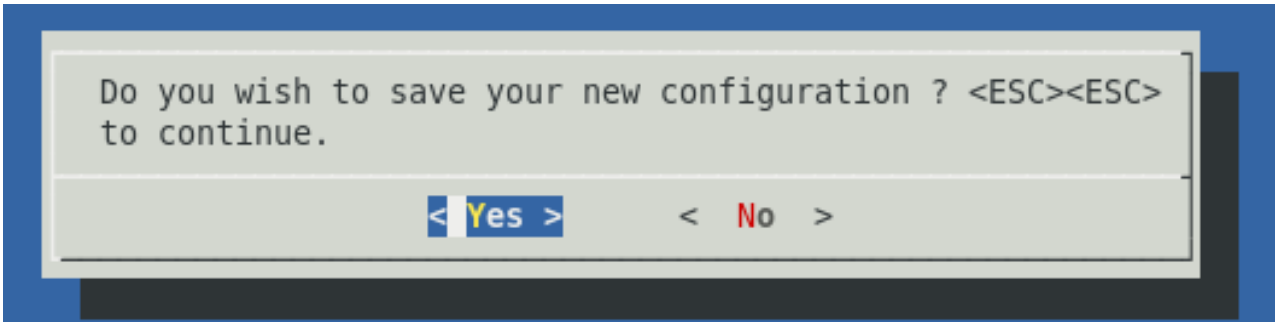
选择 Character devices ---> 如图：



选择<*>Matrix PWM module 如图：



退出保存设置，如图：



3）重新编译内核,运行 make 命令

```
cbt@Cyb-Bot: /CBT-6818/SRC/linux/kernel $ make
  CHK     include/linux/version.h
  CHK     include/generated/utsrelease.h
make[1]: “include/generated/mach-types.h”是最新的。
  CALL    scripts/checksyscalls.sh
  CHK     include/generated/compile.h
.....
```

最终在 linux 源码目录的 out/release 目录下生成新的 ARM-LINUX 内核镜像文件 boot.img

```
cbt@Cyb-Bot: /CBT-6818/SRC/linux/kernel $ ls ../out/release/boot.img
boot.img
cbt@Cyb-Bot: /CBT-6818/SRC/linux/kernel $
```

4）按照 ICS-IOT-CEP 全功能物联网教学科研平台光盘配套烧写文档将新生成的内核镜像文件 zImage 烧写到 ARM 设备中，这里不在赘述。

备注：以上在内核中添加对 PWM 设备的支持的步骤，在 ARM 系统设备出厂自带内核中已经默认添

加进来了，用户可以省略以上步骤。以上步骤在于重现系统的构造。

♦ 编译测试应用程序

1）进入实验目录：

```
cbt@Cyb-Bot:~$ cd /CBT-6818/SRC/exp/driver/02_pwm/
cbt@Cyb-Bot: /CBT-6818/SRC/exp/driver/02_pwm $ ls
  Makefile    pwm_test    pwm_test.c
cbt@Cyb-Bot: /CBT-6818/SRC/exp/driver/02_pwm $
```

2）清除中间代码，重新编译

```
cbt@Cyb-Bot: /CBT-6818/SRC/exp/driver/02_pwm $ make clean
rm -rf *.o pwm_test
cbt@Cyb-Bot: /CBT-6818/SRC/exp/driver/02_pwm $ make
arm-linux-gcc -Wall -O2 pwm_test.c -o pwm_test
cbt@Cyb-Bot: /CBT-6818/SRC/exp/driver/02_pwm $ ls
Makefile    pwm_test    pwm_test.c
cbt@Cyb-Bot: /CBT-6818/SRC/exp/driver/02_pwm $
```

当前目录下生成测试程序 pwm_test。

♦ 运行程序（NFS 方式）

1）启动 CBT-6818 型实验系统，连好网线、串口线。通过串口终端挂载宿主机实验目录。

```
[root@CBT-6818 /$ mount -t nfs -o nolock 192.168.1.7:/CBT-6818/    /mnt/
```

2）进入串口终端的 NFS 共享实验目录。

```
[root@CBT-6818 /$ cd /mnt/SRC/exp/driver/02_pwm/
[root@CBT-6818 /mnt/SRC/exp/driver/02_pwm$ ls
Makefile        driver          pwm_test        pwm_test.c
[root@CBT-6818 /mnt/SRC/exp/driver/02_pwm$
```

3）执行应用程序测试该驱动及设备

```
[root@CBT-6818 /mnt/SRC/exp/driver/02_pwm$ ./pwm_test
```

♦ 实验结果

```
[root@CBT-6818 /mnt/SRC/exp/driver/02_pwm$ ./pwm_test

BUZZER TEST ( PWM Control )
Press +/- to increase/reduce the frequency of the BUZZER
Press 'ESC' key to Exit this program

              Freq = 1000
```

```
        Freq = 990
        Freq = 980
        Freq = 970
        Freq = 960
        Freq = 950
        Freq = 940
        Freq = 930
        Freq = 920
        Freq = 910
        Freq = 900
[root@CBT-6818 /mnt/SRC/exp/driver/02_pwm$
```

此时可以通过串口终端在键盘端输入“+”或“-”调节 PWM 输出，并观察蜂鸣器音量频率，按下键盘“ESC”退出程序。

实验三. ADC 驱动及采样

1. 实验目的

- 了解 ARM 处理器片上 ADC 控制器的使用方法。
- 熟悉 Linux 系统下 ADC 驱动的编程方法及测试。

2. 实验环境

- 硬件：A53 智能网关实验平台，PC 机 Pentium 500 以上，硬盘 40G 以上，内存大于 256M
- 软件：Vmware Workstation +ubuntu + MiniCom/超级终端 +ARM-LINUX 交叉编译开发环境
- 实验目录：/CBT-6818/SRC/exp/driver/03_adc

3. 实验内容

- 阅读 CBT-6818 系统配套硬件说明，学习 S5P6818 处理器数据手册 ADC 相关章节，编写 ADC 驱动程序及测试应用程序，实现使用板载 ADC 电位器调节电压的功能。

4. 实验原理

4.1 硬件接口原理

♦ **平台上硬件接口的连接**

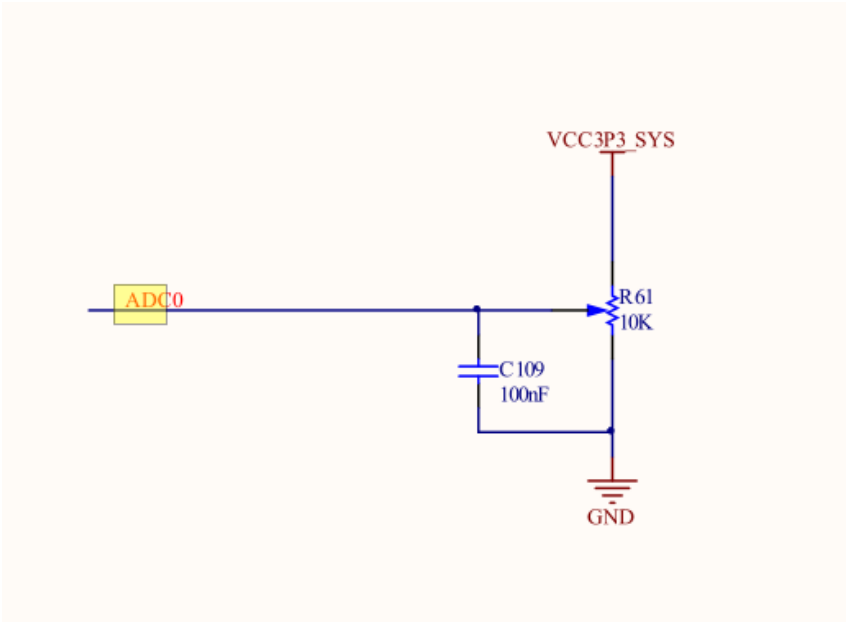


图 4.1.1 ADC 接口

如图 4.1.1 所示，平台上 S5P6818 处理器自带 4 路 10 位或 12 为精度的 AD 控制器，其中通道 0 引出，外接可调电阻，用于 AD 的测试，参考电压值为 3.3V。

♦ S5P6818 处理器相关寄存器

21.5.1.1 ADCCON

- Base Address: 0xC005_3000h
- Address = Base Address + 3000h, Reset Value = 0x0000_0004

Name	Bit	Type	Description	Reset Value
RSVD	[31:14]	R	Reserved	18'h0
ADC_DATA_SEL	[13:10]	RW	These bits select ADCDATA. 0000 = ADCDATA 5clk delayed by PCLK 0001 = ADCDATA 4clk delayed by PCLK 0010 = ADCDATA 3clk delayed by PCLK 0011 = ADCDATA 2clk delayed by PCLK 0100 = ADCDATA 1clk delayed by PCLK 0101 = ADCDATA which is not delayed else = ADCDATA 4clk delayed by PCLK	4'h0
TOT_ADC_CLK_Cnt	[9:6]	RW	These bits control the Start Of Conversion (SOC) timing. SOC signal is synchronized by (ADCCLK x TOT_ADC_CLK_Cnt).	4'h0
ASEL	[5:3]	RW	These bits select ADCIN. S5P6818 has four ADCINs and can select one of them. 000 = ADCIN_0 001 = ADCIN_1 010 = ADCIN_2 011 = ADCIN_3 100 = ADCIN_4 101 = ADCIN_5 110 = ADCIN_6 111 = ADCIN_7	3'b0
STBY	[2]	RW	A/D Converter Standby Mode. If this bit is set as "0", power is actually applied to the A/D converter. 0 = ADC Power On 1 = ADC Power Off(Standby)	1'b1
RSVD	[1]	–	Reserved	1'b0
ADEN	[0]	RW	A/D Conversion Start When the A/D conversion ends, this bit is cleared to "0". Read > Check the A/D conversion operation. 0 = Idle 1 = Busy Write > Start the A/D conversion. 0 = None 1 = Start A/D conversion	1'b0

表 4.1.1 ADCCON 寄存器

21.5.1.2 ADCDAT

- Base Address: 0xC005_3000h
- Address = Base Address + 3004h, Reset Value = 0x0000_0000

Name	Bit	Type	Description	Reset Value
RSVD	[31:12]	–	Reserved	20'h0
ADCDAT	[11:0]	R	These bits are 12-bit data converted via the ADC.	12'h0

表 4.1.2 ADCDAT 寄存器

如上述寄存器表所示，平台上 S5P6818 处理器 ADCCON 控制寄存器负责 AD 的相关参数配置，如预分频值、转换精度、通道，AD 转换完成标志等，通过软件配置好 ADCCON 寄存器后即可读取 ADCDAT 数据寄存器获取相应通道的 AD 转换结果。

4.2 软件接口介绍

♦ Linux 混杂设备模型

在 Linux 系统中，存在一类字符设备，他们共享一个主设备号（10），但次设备号不同，我们称这类设备为混杂设备（miscdeivce），查看/proc/device 中可以看到一个名为 misc 的主设备号为 10.所有的混杂设备形成一个链表，对设备访问时内核根据次设备号找到对应的 miscdevice 设备。

Linux 内核使用 struct miscdeivce 来描述一个混杂设备。

```
struct miscdevice   {
    int minor;
    const char *name;
    const struct file_operations *fops;
    struct list_head list;
    struct device *parent;
    struct device *this_device;
    const char *nodename;
    mode_t mode;
};
```

minor 是这个混杂设备的次设备号，若由系统自动配置，则可以设置为 MISC_DYNANIC_MINOR，name 是设备名.使用时只需填写 minor 次设备号，*name 设备名，*fops 文件操作函数集即可。

Linux 内核使用 misc_register 函数注册一个混杂设备，使用 misc_deregister 移除一个混杂设备。注册成功后，linux 内核为自动为该设备创建设备节点，在/dev/下会产生相应的节点。

注册函数：

```
int misc_register(struct miscdevice * misc)
```

输入参数：struct miscdevice

返回值：0 表示注册成功。负数 表示未成功。

卸载函数：

```
int    misc_deregister(struct miscdevice *misc)
```

输入参数：struct miscdevice

返回值：0 表示注册成功。

负数 表示未成功。

4.4 关键代码分析

驱动程序(linux-3.5/drivers/char/4412_adc.c)

```
#define DEVICE_NAME         "adc"/*设备名称*/

static void __iomem *base_addr;/*地址指针*/
 /*ADC 设备结构*/
typedef struct {
        struct mutex lock;
        struct s3c_adc_client *client;
        int channel;
} ADC_DEV;
static ADC_DEV adcdev;


 /*ADC 转换结束中断处理函数  */
static irqreturn_t adcdone_int_handler(int irq, void *dev_id)
{
        if (__ADC_locked) {/*并发处理*/
                adc_data = ADCDAT0 & 0x3ff;/*读取 ADC 结果，10 位精度*/

                ev_adc = 1;
                wake_up_interruptible(&adcdev.wait);/*唤醒读等待队列*/

                /* clear interrupt */
                __raw_writel(0x0, base_addr + S3C_ADCCLRINT);/*清 ADC 中断标志*/
        }


        return IRQ_HANDLED;
}
 /*ADC 设备驱动 READ 接口函数实现*/
static ssize_t exynos_adc_read(struct file *filp, char *buffer,
                size_t count, loff_t *ppos)
{
        char str[20];
```

```
        int value;
        size_t len;

        value = exynos_adc_read_ch();

        len = sprintf(str, "%d\n", value);
        if (count >= len) {
                int r = copy_to_user(buffer, str, len);/*向用户空间传递 AD 转换结果*/

                return r ? r : len;
        } else {
                return -EINVAL;
        }
}

 /*ADC 设备 OPEN 函数 */
static int exynos_adc_open(struct inode *inode, struct file *filp)
{
/*初始化 ADC 通道和分频值*/
        exynos_adc_set_channel(0);

        DPRINTK("adc opened\n");
        return 0;
}

 /* 释放 ADC 设备驱动函数 */
static int exynos_adc_release(struct inode *inode, struct file *filp)
{
        DPRINTK("adc closed\n");
        return 0;
}

static struct file_operations adc_dev_fops = {
        owner:    THIS_MODULE,
        open:     exynos_adc_open,
        read:     exynos_adc_read,
        unlocked_ioctl: exynos_adc_ioctl,
        release:          exynos_adc_release,
};
 /* 初始化 MISC 设备结构 */
static struct miscdevice misc = {
        .minor    = MISC_DYNAMIC_MINOR,
        .name     = "adc",
        .fops     = &adc_dev_fops,
};
/* 驱动初始化函数*/
```

```
static int __init exynos_adc_init(void)
{
        return platform_driver_register(&exynos_adc_driver);
}

static void __exit exynos_adc_exit(void)
{
        platform_driver_unregister(&exynos_adc_driver);
}
module_init(exynos_adc_init);
module_exit(exynos_adc_exit)
```

以上程序主要通过初始化 ADC 控制寄存器，，最终将 ADCDAT 中数据传递给用户空间，驱动中主要实现了设备的 read 接口函数和中断处理函数。其他部分代码及测试应用程序用户可以通过阅读实验源码自行分析。

5. 实验步骤

♦ **在内核中添加按键设备支持**

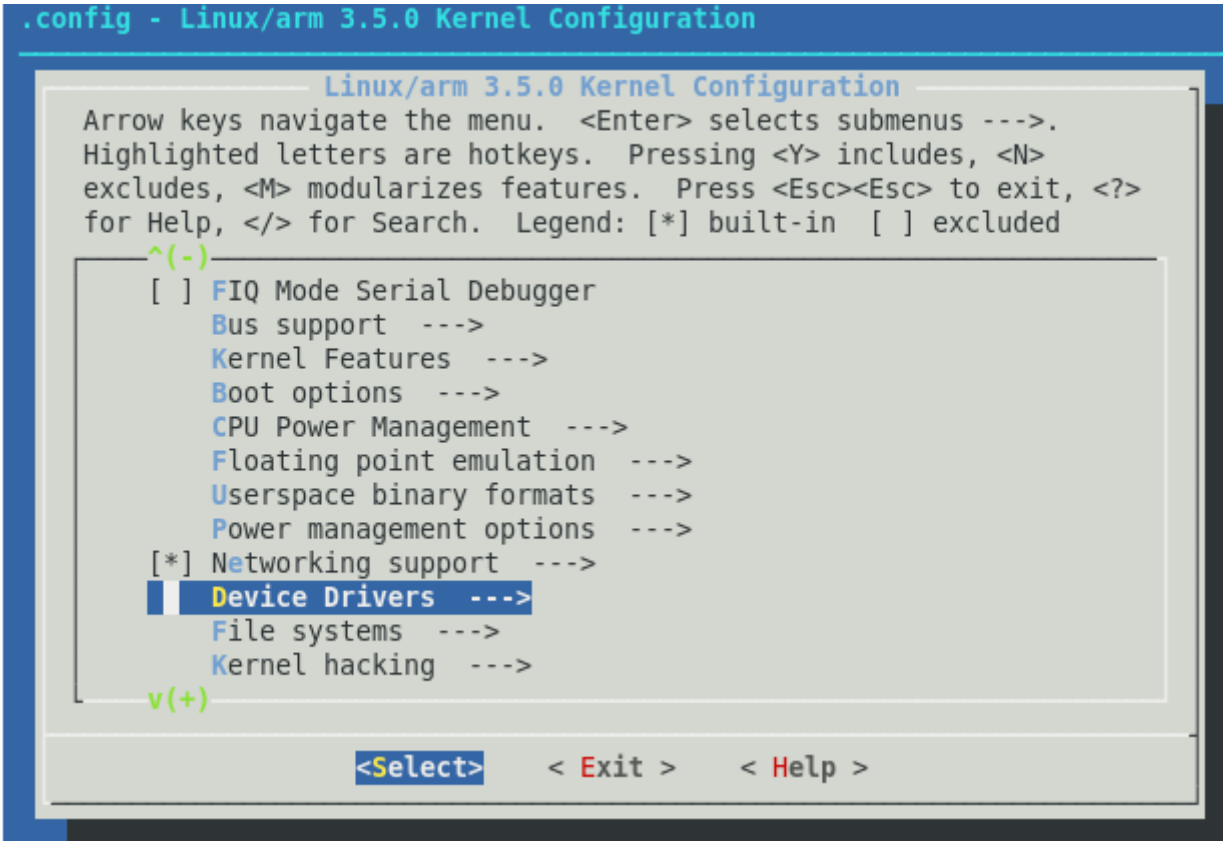
1）进入宿主机中 CBT-6818 型光盘内核目录：

```
cbt@Cyb-Bot:~$ cd /CBT-6818/SRC/linux/linux-3.5
```

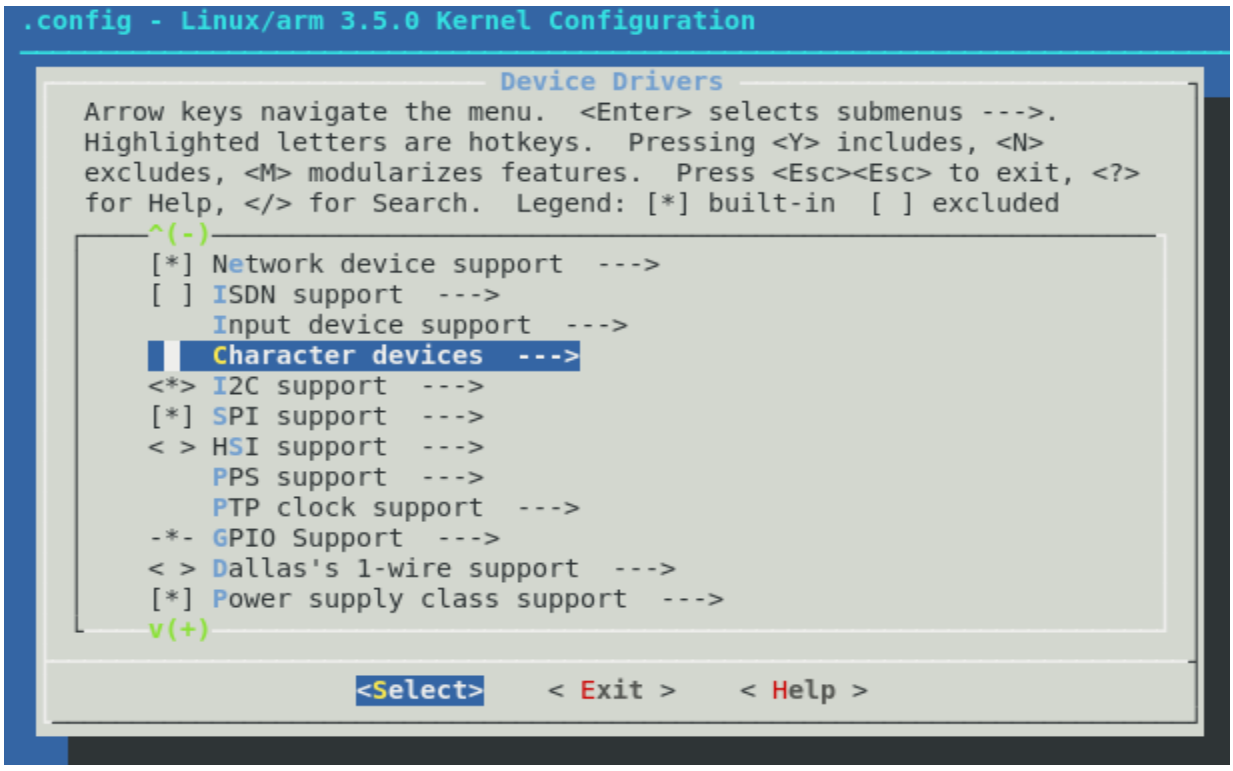
2）运行 make menuconfig 命令配置内核对按键设备的相关支持

```
cbt@Cyb-Bot: linux-3.5$
cbt@Cyb-Bot: linux-3.5$ cp cbt4412_linux_defconfig  .config
cbt@Cyb-Bot: linux-3.5$ make menuconfig
```

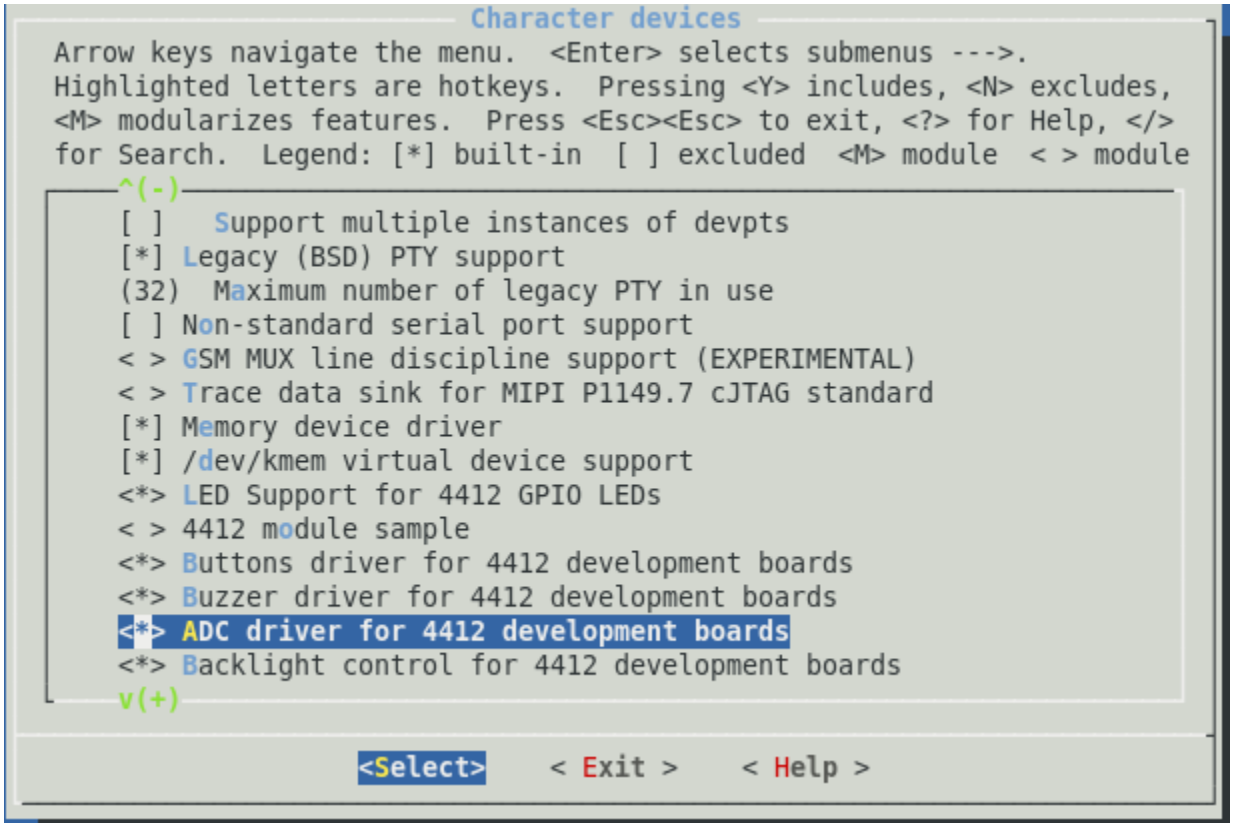
选择 Device Drivers --->选项，如图



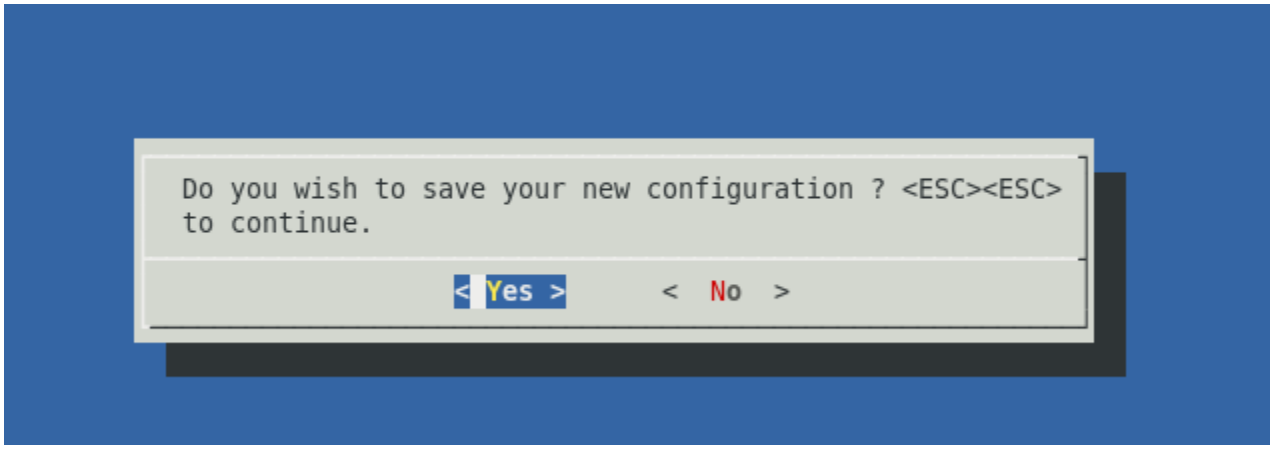
选择 Character devices ---> 如图：



选择<*> ADC driver for 4412 Development Boards 如图：



退出保存设置，如图：



3）重新编译内核,运行 make 命令

```
cbt@Cyb-Bot: linux-3.5$ make
  CHK     include/linux/version.h
  CHK     include/generated/utsrelease.h
make[1]: “include/generated/mach-types.h”是最新的。
  CALL    scripts/checksyscalls.sh
  CHK     include/generated/compile.h
.....
```

最终在内核源码目录的 arch/arm/boot 目录下生成新的 ARM-LINUX 内核镜像文件 zImage

```
cbt@Cyb-Bot: linux-3.5$ ls arch/arm/boot/zImage
```

```
arch/arm/boot/zImage
cbt@Cyb-Bot: linux-3.5$
```

4）按照 ICS-IOT-CEP 全功能物联网教学科研平台光盘配套烧写文档将新生成的内核镜像文件 zImage 烧写到 ARM 设备中，这里不在赘述。

备注：以上在内核中添加对 ADC 设备的支持的步骤，在 ARM 系统设备出厂自带内核中已经默认添加进来了，用户可以省略以上步骤。以上步骤在于重现系统的构造。

♦ 编译测试应用程序

1）进入实验目录：

```
cbt@Cyb-Bot: \$ cd /CBT-6818/SRC/exp/driver/03_adc/
cbt@Cyb-Bot: 03_adc$ ls
adc_test    adc_test.c    driver    Makefile
cbt@Cyb-Bot: 03_adc$
```

2）清除中间代码，重新编译

```
cbt@Cyb-Bot: 03_adc$ make clean
rm -rf *.o adc_test
cbt@Cyb-Bot: 03_adc$ make
arm-linux-gcc -Wall -O2 adc_test.c -o adc_test
cbt@Cyb-Bot: 03_adc$ ls
adc_test    adc_test.c    driver    Makefile
cbt@Cyb-Bot: 03_adc$
```

当前目录下生成测试程序 adc_test。

♦ 运行程序（NFS 方式）

1）启动 CBT-6818 型实验系统，连好网线、串口线。通过串口终端挂载宿主机实验目录。

```
[root@CBT-6818 /$ mount -t nfs -o nolock 192.168.1.7:/CBT-6818/    /mnt/
```

2）进入串口终端的 NFS 共享实验目录。

```
[root@CBT-6818 /$cd /mnt//SRC/exp/driver/03_adc/
cbt@Cyb-Bot: 03_adc$ ls
adc_test    adc_test.c    driver    Makefile
cbt@Cyb-Bot: 03_adc$
```

3）执行应用程序测试该驱动及设备

```
cbt@Cyb-Bot: 03_adc$ ./adc_test
```

♦ 实验结果

```
[root@CBT-6818 03_adc$ ./adc_test
```

```
press Ctrl-C to stop
ADC Value(Valtage): 2.2V
cbt@Cyb-Bot: 03_adc$ .
```

此时可以通过选择 ICS-IOT-CEP 全功能物联网教学科研平台智能终端上的 ADC 电阻器来改变电阻值，观察 ADC 转换后的电压输出数值。

实验四. LCD 设备驱动及控制

1. 实验目的

- 了解 LCD 基本概念与原理及 Linux 下 LCD 的 Framebuffer 结构原理。
- 了解用总线方式驱动 LCD 模块，掌握 ARM 内置的 LCD 控制器驱动。

2. 实验环境

- 硬件：A53 智能网关实验平台，PC 机 Pentium 500 以上，硬盘 40G 以上，内存大于 256M。
- 软件：Vmware Workstation +ubuntu + MiniCom/超级终端 +ARM-LINUX 交叉编译开发环境。
- 实验目录：/CBT-6818/SRC/exp/driver/04_lcd。

3. 实验内容

- 掌握 linux 内核中显示设备驱动实现的方法，编写 LCD 显示测试程序，实现 LCD 显示绘图。

4. 实验原理

4.1 硬件接口原理

♦ **LCD（Liquid Crystal Display）原理**

液晶得名于其物理特性：它的分子晶体，以液态存在而非固态。这些晶体分子的液体特性使得它具有两种非常有用的特点：

1、如果让电流通过液晶层，这些分子将会以电流的流向方向进行排列，如果没有电流，它们将会彼此平行排列。

2、如果提供了带有细小沟槽的外层，将液晶倒入后，液晶分子会顺着槽排列，并且内层与外层以同样的方式进行排列。

液晶的第三个特性是很神奇的：液晶层能使光线发生扭转。液晶层表现的有些类似偏光器，这就意味着它能够过滤除了那些从特殊方向射入之外的所有光线。此外，如果液晶层发生了扭转，光线将会随之扭转，以不同的方向从另外一个面中射出。

液晶的这些特点使得它可以被用来当作一种开关——即可以阻碍光线，也可以允许光线通过。液晶单元的底层是由细小的脊构成的，这些脊的作用是让分子呈平行排列。上表面也是如此，在这两侧之间的分子平行排列，不过当上下两个表面之间呈一定的角度时，液晶随着两个不同方向的表面进行排列，就会发生扭曲。结果便是这个扭曲的螺旋层使通过的光线

也发生扭曲。如果电流通过液晶，所有的分子将会按照电流的方向进行排列，这样就会消除光线的扭转。如图 4.1.1 所示，如果将一个偏振滤光器放置在液晶层的上表面，扭转的光线通过(如 A)，而没有发生扭转的光线(如 B)将被阻碍。因此可以通过电流的通断改变 LCD 中的液晶排列，使光线在加电时射出，而不加电时被阻断。也有某些设计为了省电的需要，有电流时，光线不能通过，没有电流时，光线通过。

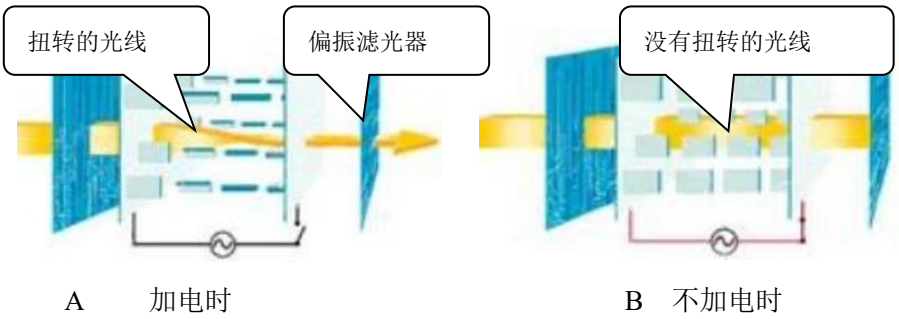


图 4.1.1 光线穿过与阻断示意图

LCD 显示器的基本原理就是通过给不同的液晶单元供电，控制其光线的通过与否，从而达到显示的目的。因此，LCD 的驱动控制归于对每个液晶单元的通断电的控制，每个液晶单元都对应着一个电极，对其通电，便可使光线通过（也有刚好相反的，即不通电时光线通过，通电时光线不通过）。

♦ 电致发光

LCD 的发光原理是通过控制加电与否来使光线通过或挡住，从而显示图形。光源的提供方式有两种：透射式和反射式。笔记本电脑的 LCD 显示屏即为透射式，屏后面有一个光源，因此外界环境可以不需要光源。而一般微控制器上使用的 LCD 为反射式，需要外界提供光源，靠反射光来工作。电致发光（EL）是液晶屏提供光源的一种方式。电致发光的特点是低功耗，与二极管发光比较而言体积小。

电致发光（EL）是将电能直接转换为光能的一种发光现象。电致发光片是利用此原理经过加工制作而成的一种发光薄片，如图 4.1.2 所示。其特点是：超薄、高亮度、高效率、低功耗、低热量、可弯曲、抗冲击、长寿命、多种颜色选择等。因此，电致发光片被广泛应用于各种领域。

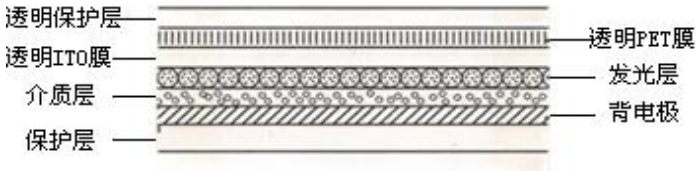


图 4.1.2 电致发光片的基本结构

♦ LCD 的驱动控制

市面上出售的 LCD 有两种类型：

一种是带有驱动电路的 LCD 显示模块，这种 LCD 可以方便地与各种低档单片机进行接口，如 8051 系列单片机，但是由于硬件驱动电路的存在，体积比较大。这种模式常常使用总线方式来驱动。

另一种是 LCD 显示屏，没有驱动电路，需要与驱动电路配合使用，特点是体积小，但却需要另外的驱动芯片。也可以使用带有 LCD 驱动能力的高档 MCU 驱动，如 ARM 系列的 S5P6818、PXA270 等。如图 4.1.3 所示：

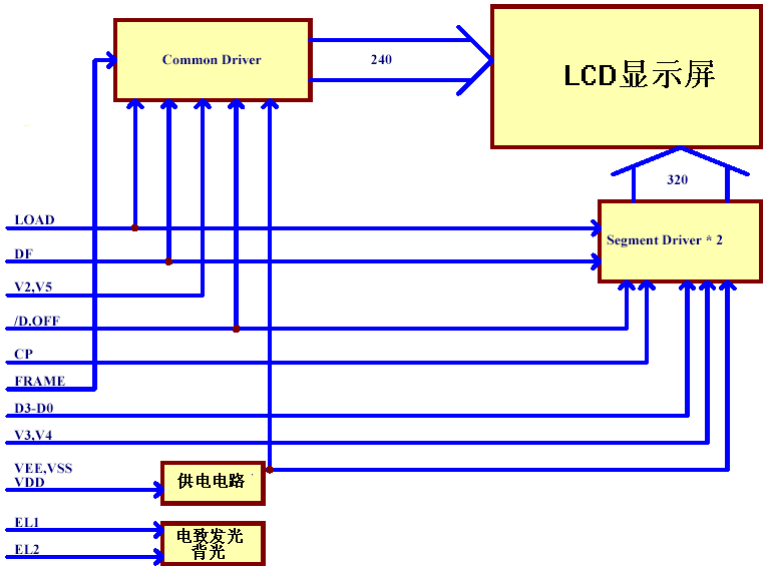


图 4.1.3 不带驱动电路的 LCD 结构

1）总线驱动方式

一般带有驱动模块的 LCD 显示屏使用这种驱动方式，由于 LCD 已经带有驱动硬件电路，因此模块给出的是总线接口，便于与单片机的总线进行接口。驱动模块具有八位数据总线，外加一些电源接口和控制信号。而且自带显示缓存，只需要将要显示的内容送到显示缓存中就可以实现内容的显示。由于只有八条数据线，因此常常通过引脚信号来实现地址与数据线复用，以达到把相应数据送到相应显示缓存的目的。表 5.2-1 为一个典型的显示模块（HY－12864B）提供的总线接口。

Pin No	Description
1	GND
2	Power supply for Logic
3	Power supply for LCD
4	Register selection (H:Data registor, L:Instruction registor)
5	Read/write selection (H:Read,L:Write)
6	Enable signal for chip
7-14	Data Bus line
15	Chip Select Signal for Left Half of the Screen
16	Chip Select Signal for RIGHT Half of the Screen
17	Reset signal
18	Negative voltage output
19	Power supply for Backlight
20	Power supply for Backlight

表 4.1.1 典型带驱动液晶模块的总线接口

2）控制器扫描方式

S5P6818 处理器中具有内置的 LCD 控制器，它具有将显示缓存（在系统存储器中）中的 LCD 图象数据传输到外部 LCD 驱动电路的逻辑功能。支持 TFT（主动矩阵或叫有源矩阵）LCD 屏，并支持黑白和彩色显示。对于不同尺寸的 LCD，具有不同数量的垂直和水平象素、数据接口的数据宽度、接口时间及刷新率，而 LCD 控制器可以进行编程控制相应的

寄存器值，以适应不同的 LCD 显示板。

♦ **平台 LCD 部分硬件电路图**

如图 4.1.4 所示：

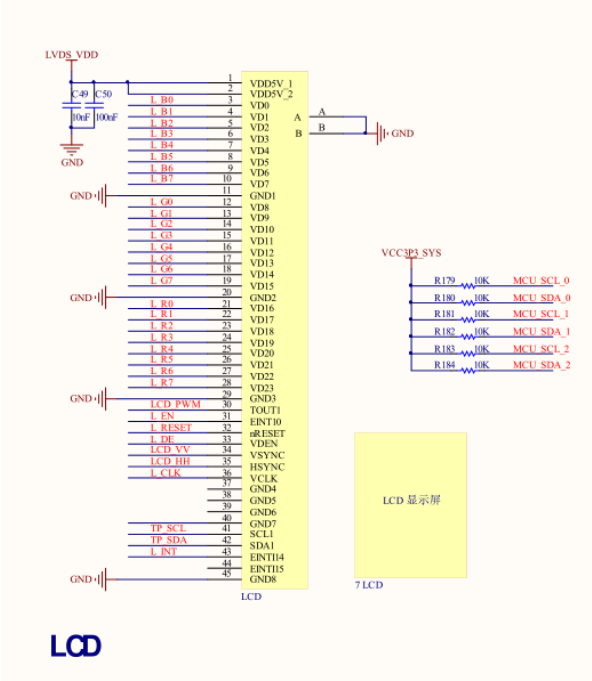


图 4.1.4　LCD 的硬件电路图

4.2 软件接口介绍

♦ **Linux 的帧缓冲设备**

帧缓冲（framebuffer）是 Linux 为显示设备提供的一个接口，把显存抽象后的一种设备，他允许上层应用程序在图形模式下直接对显示缓冲区进行读写操作。这种操作是抽象的，统一的。用户不必关心物理显存的位置、换页机制等等具体细节。这些都是由 Framebuffer 设备驱动来完成的。帧缓冲驱动的应用广泛，在 linux 的桌面系统中，Xwindow 服务器就是利用帧缓冲进行窗口的绘制。尤其是通过帧缓冲可显示汉字点阵，成为 Linux 汉化的唯一可行方案。

帧缓冲设备对应的设备文件为/dev/fb*，如果系统有多个显示卡，Linux 下还可支持多个帧缓冲设备，最多可达 32 个，分别为/dev/fb0 到/dev/fb31，而/dev/fb 则为当前缺省的帧缓冲设备，通常指向/dev/fb0。当然在嵌入式系统中支持一个显示设备就够了。帧缓冲设备为标准字符设备，主设备号为 29，次设备号则从 0 到 31。分别对应/dev/fb0 到/dev/fb31。

驱动程序端：

♦ **FrameBuffer 在 Linux 中的实现和机制**

Framebuffer 对应的源文件在 linux/drivers/video/目录下。总的抽象设备文件为 fbcon.c，在这个目录下还有与各种显示设备驱动相关的源文件。

FrameBuffer 设备驱动基于如下两个文件：
linux/include/linux/fb.h
linux/drivers/video/fbmem.c

1、在 fb.h 中定义了 framebuffer 所使用的重要数据结构：

1).Struct fb_var_screeninfo 描述显卡的特性的。通常是被用户设置的。

2).Struct fb_fix_screeninfo 定义了显卡的硬件特性，是不能改变的。

3).Struct fb_cmap 描述设备无关的颜色映射信息。可以通过 FBIOGETCMAP 和 FBIOPUTCMAP 对应的 ioctl 操作设定或获取颜色映射信息.

4).Struct fb_info 定义了当前显卡 framebuffer 设备状态，一个显卡可能有两个 framebuffer，在这种情况下，就需要两个 fb_info 结构。这个结构是唯一在内核空间可见的。在这个结构中有一个 fb_ops 指针，指向驱动设备工作所需的函数集。

5).struct fb_ops 用户应用可以使用 ioctl()系统调用来操作设备，这个结构就是用以支持 ioctl()的这些操作的。（注: fb_ops 结构与 file_operations 结构不同，fb_ops 是底层操作的抽象,而 file_operations 是提供给上层系统调用的接口，可以直接调用.）ioctl()系统调用在文件 fbmem.c 中实现，通过观察可以发现 ioctl()命令与 fb_ops’s 中函数的关系:

FBIOGET_VSCREENINFO fb_get_var

FBIOPUT_VSCREENINFO fb_set_var

FBIOGET_FSCREENINFO fb_get_fix

FBIOPUTCMAP fb_set_cmap

FBIOGETCMAP fb_get_cmap

FBIOPAN_DISPLAY fb_pan_display

如果我们定义了 fb_XXX_XXX 方法，用户程序就可以使用 FBIOXXXX 宏的 ioctl()操作来操作硬件

2、 fbmem.c 处于 Framebuffer 设备驱动技术的中心位置.它为上层应用程序提供系统调用也为下一层的特定硬件驱动提供接口；那些底层硬件驱动需要用到这儿的接口来向系统内核注册它们自己. fbmem.c 为所有支持 FrameBuffer 的设备驱动提供了通用的接口。

1) 全局变量

struct fb_info *registered_fb[FB_MAX];

int num_registered_fb;

这两变量记录了所有 fb_info 结构的实例，fb_info 结构描述显卡的当前状态，所有设备对应的 fb_info 结构都保存在这个数组中，当一个 FrameBuffer 设备驱动向系统注册自己时，其对应的 fb_info 结构就会添加到这个结构中，同时 num_registered_fb 为自动加 1.

2）fbmem.c 实现了如下函数.

register_framebuffer(struct fb_info *fb_info);

unregister_framebuffer(struct fb_info *fb_info);

这两个是提供给下层 FrameBuffer 设备驱动的接口，设备驱动通过这两函数向系统注册或注销自己。几乎底层设备驱动所要做的所有事情就是填充 fb_info 结构然后向系统注册或注销它。

应用程序端：

对/dev/fb 设备的操作，应用程序的操作主要有这几种：

1）读/写（read/write）/dev/fb：相当于读/写屏幕缓冲区。

例如：

用 cp /dev/fb0 tmp 命令可将当前屏幕的内容拷贝到一个文件中，而命令 cp tmp > /dev/fb0 则将图形文件 tmp 显示在屏幕上。

2）映射（map）操作：由于 Linux 工作在保护模式，每个应用程序都有自己的虚拟地址空间，在应用程序中是不能直接访问物理缓冲区地址的。为此，Linux 在文件操作 file_operations 结构中提供了 mmap 函数，可将文件的内容映射到用户空间。对于帧缓冲设备，则可通过映射操作，可将屏幕缓冲区的物理地址映射到用户空间的一段虚拟地址中，之后用户就可以通过读写这段虚拟地址访问屏幕缓冲区，在屏幕上绘图了。

例如：

mmap 用法：

```
#include <sys/mman.h>
void *mmap(void *start, size_t length, int prot, int flags,int fd, off_t offset);
int munmap(void *start, size_t length);
```

参数：

start：映射区的开始地址。

length：映射区的长度。

prot：期望的内存保护标志，不能与文件的打开模式冲突。是以下的某个值，可以通过 or 运算合理地组合在一起

PROT_EXEC：页内容可以被执行

PROT_READ：页内容可以被读取

PROT_WRITE：页可以被写入

PROT_NONE：页不可访问

flags：指定映射对象的类型，映射选项和映射页是否可以共享。它的值可以是一个或者多个以下位的组合体

MAP_FIXED：使用指定的映射起始地址，如果由 start 和 len 参数指定的内存区重叠于现存的映射空间，重叠部分将会被丢弃。如果指定的起始地址不可用，操作将会失败。并且起始地址必须落在页的边界上。

MAP_SHARED：与其它所有映射这个对象的进程共享映射空间。对共享区的写入，相当于输出到文件。直到 msync()或者 munmap()被调用，文件实际上不会被更新。

MAP_PRIVATE：建立一个写入时拷贝的私有映射。内存区域的写入不会影响到原文件。这个标志和以上标志是互斥的，只能使用其中一个。

MAP_NORESERVE：不要为这个映射保留交换空间。当交换空间被保留，对映射区修改的可能会得到保证。当交换空间不被保留，同时内存不足，对映射区的修改会引起段违例信号。

MAP_LOCKED：锁定映射区的页面，从而防止页面被交换出内存。

MAP_GROWSDOWN：用于堆栈，告诉内核 VM 系统，映射区可以向下扩展。

MAP_ANONYMOUS：匿名映射，映射区不与任何文件关联。

MAP_ANON：MAP_ANONYMOUS 的别称，不再被使用。

MAP_FILE：兼容标志，被忽略。

MAP_32BIT：将映射区放在进程地址空间的低 2GB，MAP_FIXED 指定时会被忽略。当前这个标志只在 x86-64 平台上得到支持。

MAP_POPULATE：为文件映射通过预读的方式准备好页表。随后对映射区的访问不会被页违例阻塞。

MAP_NONBLOCK：仅和 MAP_POPULATE 一起使用时才有意义。不执行预读，只为已存在于内存中的页面建立页表入口。

fd：有效的文件描述词。如果 MAP_ANONYMOUS 被设定，为了兼容问题，其值应为-1。

offset：被映射对象内容的起点。

返回说明：

成功执行时，mmap()返回被映射区的指针，munmap()返回 0。失败时，mmap()返回 MAP_FAILED[其值为(void *)-1]，munmap 返回-1。errno 被设为以下的某个值

EACCES：访问出错

EAGAIN：文件已被锁定，或者太多的内存已被锁定

EBADF：fd 不是有效的文件描述词

EINVAL：一个或者多个参数无效

ENFILE：已达到系统对打开文件的限制

ENODEV：指定文件所在的文件系统不支持内存映射

ENOMEM：内存不足，或者进程已超出最大内存映射数量

EPERM： 权能不足，操作不允许

ETXTBSY：已写的方式打开文件，同时指定 MAP_DENYWRITE 标志

SIGSEGV：试着向只读区写入

SIGBUS：试着访问不属于进程的内存区

3）I/O 控制：对于帧缓冲设备，对设备文件的 ioctl 操作可读取/设置显示设备及屏幕的

参数，如分辨率，显示颜色数，屏幕大小等等。ioctl 的操作是由底层的驱动程序来完成的。

例如：

```
ioctl(fbfd, FBIOGET_FSCREENINFO, &finfo)
```

获取 fb_var_screeninfo 结构的信息，在 linux/include/linux/fb.h 定义。

```
ioctl(fbfd, FBIOGET_VSCREENINFO, &vinfo)
```

获取 fb_fix_screeninfon 结构的信息。在 linux/include/linux/fb.h 定义。

在应用程序中，操作/dev/fb 的一般步骤如下：

1）打开/dev/fb 设备文件。

2）用 ioctrl 操作取得当前显示屏幕的参数，如屏幕分辨率，每个像素点的比特数。根据屏幕参数可计算屏幕缓冲区的大小。

3）将屏幕缓冲区映射到用户空间（mmap）。

4）映射后就可以直接读写屏幕缓冲区，进行绘图和图片显示了。

典型程序段如下：

```
#include <linux/fb.h>
int main()
{
  int fb = 0;
void *fb_mem;

struct fb_var_screeninfo vinfo;
  struct fb_fix_screeninfo finfo;
  long int screensize = 0;
/*打开设备文件*/
  fb = open("/dev/fb0", O_RDWR);
/*取得屏幕相关参数*/
  ioctl(fb, FBIOGET_FSCREENINFO, &finfo);
  ioctl(fb, FBIOGET_VSCREENINFO, &vinfo);
/*计算屏幕缓冲区大小*/
  screensize = vinfo.xres * vinfo.yres * vinfo.bits_per_pixel / 8;
/*映射屏幕缓冲区到用户地址空间*/
fb_mem=(char*)mmap(0,screensize,PROT_READ|PROT_WRITE,MAP_SHARED, fb, 0);
/*下面可通过 fbp 指针读写缓冲区*/
……
/*释放缓冲区，关闭设备*/
munmap(fb_mem, screensize);
close(fb);
}
```

4.3 关键代码分析

```
struct fb_var_screeninfo vinfo;
struct fb_fix_screeninfo finfo;
char *frameBuffer = 0;

//打印 fb 驱动中 fix 结构信息，注：在 fb 驱动加载后，fix 结构不可被修改。
void
printFixedInfo ()
{
      printf ("Fixed screen info:\n"
                  "\tid: %s\n"
                  "\tsmem_start: 0x%lx\n"
                  "\tsmem_len: %d\n"
                  "\ttype: %d\n"
                  "\tline_length: %d\n"
                  "\n",
                  finfo.id, finfo.smem_start, finfo.smem_len, finfo.type,
              finfo.line_length);
}

//打印 fb 驱动中 var 结构信息，注：fb 驱动加载后，var 结构可根据实际需要被重置
void
printVariableInfo ()
{
      printf ("Variable screen info:\n"
                  "\txres: %d\n"
                  "\tyres: %d\n"
                  "\tbits_per_pixel: %d\n"
                  "\tred: offset: %2d, length: %2d, msb_right: %2d\n"
                  "\tgreen: offset: %2d, length: %2d, msb_right: %2d\n"
                  "\tblue: offset: %2d, length: %2d, msb_right: %2d\n"
                  "\theight: %d\n"
                  "\twidth: %d\n"
                  "\n",
                  vinfo.xres, vinfo.yres, vinfo.bits_per_pixel,
                  vinfo.red.offset, vinfo.red.length,vinfo.red.msb_right,
         vinfo.green.offset, vinfo.green.length,vinfo.green.msb_right,
                  vinfo.blue.offset, vinfo.blue.length,vinfo.blue.msb_right,
                  vinfo.height, vinfo.width);
}

//画大小为 width*height 的同色矩阵，8alpha+8reds+8greens+8blues
void
```

```
drawRect_rgb32 (int x0, int y0, int width, int height, int color)
{
	const int bytesPerPixel = 4;
	const int stride = finfo.line_length / bytesPerPixel;

	int *dest = (int *) (frameBuffer)
		+ (y0 + vinfo.yoffset) * stride + (x0 + vinfo.xoffset);

	int x, y;
	for (y = 0; y < height; ++y)
	{
		for (x = 0; x < width; ++x)
		{
			dest[x] = color;
		}
		dest += stride;
	}
}

//画大小为 width*height 的同色矩阵，5reds+6greens+5blues
void
drawRect_rgb16 (int x0, int y0, int width, int height, int color)
{
	const int bytesPerPixel = 2;
	const int stride = finfo.line_length / bytesPerPixel;
	const int red = (color & 0xff0000) >> (16 + 3);
	const int green = (color & 0xff00) >> (8 + 2);
	const int blue = (color & 0xff) >> 3;
	const short color16 = blue | (green << 5) | (red << (5 + 6));

	short *dest = (short *) (frameBuffer)
		+ (y0 + vinfo.yoffset) * stride + (x0 + vinfo.xoffset);

	int x, y;
	for (y = 0; y < height; ++y)
	{
		for (x = 0; x < width; ++x)
		{
			dest[x] = color16;
		}
		dest += stride;
	}
}
/*  绘制矩形图案*/
void
drawRect (int x0, int y0, int width, int height, int color)
```

```
{
    switch (vinfo.bits_per_pixel)
    {
    case 32:
        drawRect_rgb32 (x0, y0, width, height, color);
        break;
    case 16:
        drawRect_rgb16 (x0, y0, width, height, color);
        break;
    default:
        printf ("Warning: drawRect() not implemented for color depth %i\n",
                vinfo.bits_per_pixel);
        break;
    }
}

int
main (int argc, char **argv)
{
    const char *devfile = "/dev/fb0";
    long int screensize = 0;
    int fbFd = 0;


    /* Open the file for reading and writing */
    fbFd = open (devfile, O_RDWR);
    if (fbFd == -1)
    {
        perror ("Error: cannot open framebuffer device");
        exit (1);
    }

    //获取 finfo 信息并显示
    if (ioctl (fbFd, FBIOGET_FSCREENINFO, &finfo) == -1)
    {
        perror ("Error reading fixed information");
        exit (2);
    }
    printFixedInfo ();
    //获取 vinfo 信息并显示
    if (ioctl (fbFd, FBIOGET_VSCREENINFO, &vinfo) == -1)
    {
        perror ("Error reading variable information");
        exit (3);
    }
    printVariableInfo ();
```

```
    /* Figure out the size of the screen in bytes */
    screensize = finfo.smem_len;

    /* Map the device to memory */
    frameBuffer =
        (char *) mmap (0, screensize, PROT_READ | PROT_WRITE, MAP_SHARED,
                       fbFd, 0);
    if (frameBuffer == MAP_FAILED)
    {
        perror ("Error: Failed to map framebuffer device to memory");
        exit (4);
    }

    /* Clear the device of framebuffer */
    memset(frameBuffer, 0xffffffff, screensize);

    printf ("Will draw 3 rectangles on the screen,\n"
            "they should be colored red, green and blue (in that order).\n");
    drawRect (vinfo.xres / 8, vinfo.yres / 8,
              vinfo.xres / 4, vinfo.yres / 4, 0xffff0000);
    drawRect (vinfo.xres * 3 / 8, vinfo.yres * 3 / 8,
              vinfo.xres / 4, vinfo.yres / 4, 0xff00ff00);
    drawRect (vinfo.xres * 5 / 8, vinfo.yres * 5 / 8,
              vinfo.xres / 4, vinfo.yres / 4, 0xff0000ff);

    //sleep (5);
    printf (" Done.\n");

    munmap (frameBuffer, screensize);      //解除内存映射，与 mmap 对应

    close (fbFd);
    return 0;
}
```

以上程序可实现在 LCD 显示设备上绘制三个矩形图案出来。其他接口用户可以通过阅读实验源码自行分析。

5. 实验步骤

♦ **在内核中添加 framebuffer 设备支持**

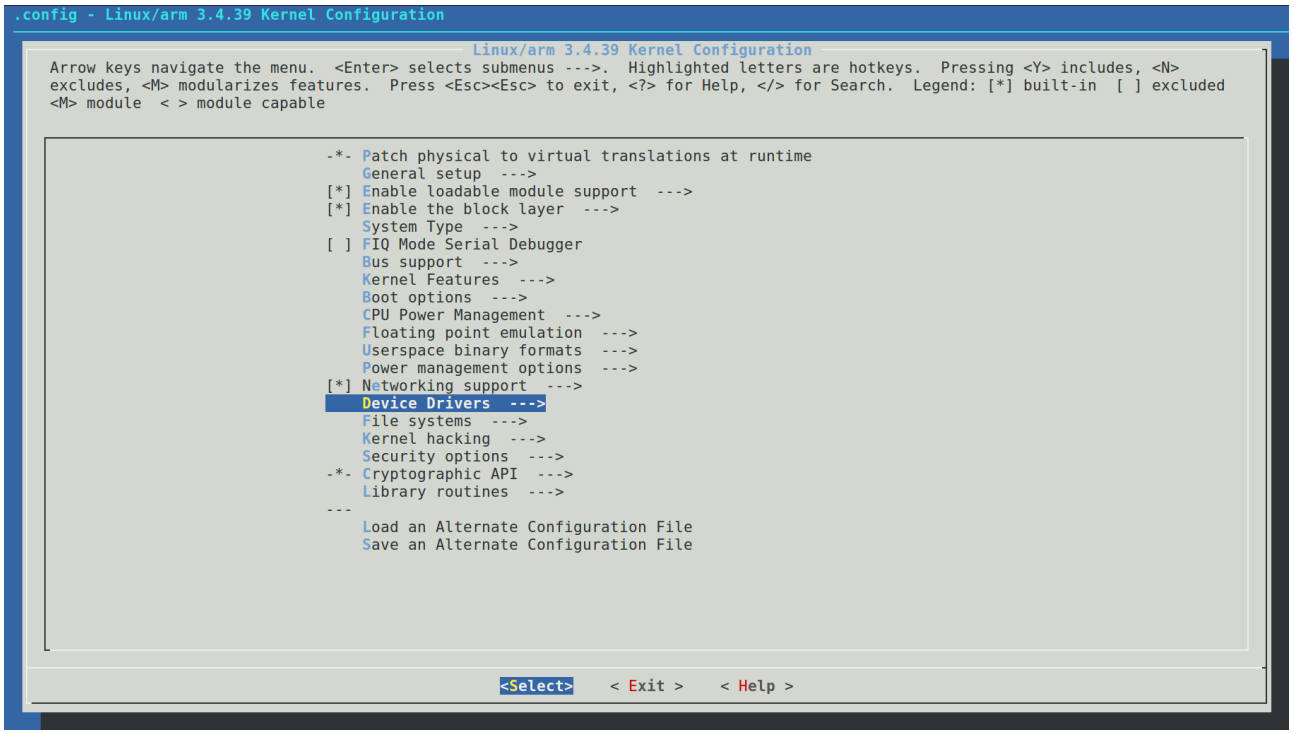
1）进入宿主机中 CBT-6818 型光盘内核目录：

```
cbt@Cyb-Bot:~$ cd /CBT-6818/SRC/linux/kernel
```

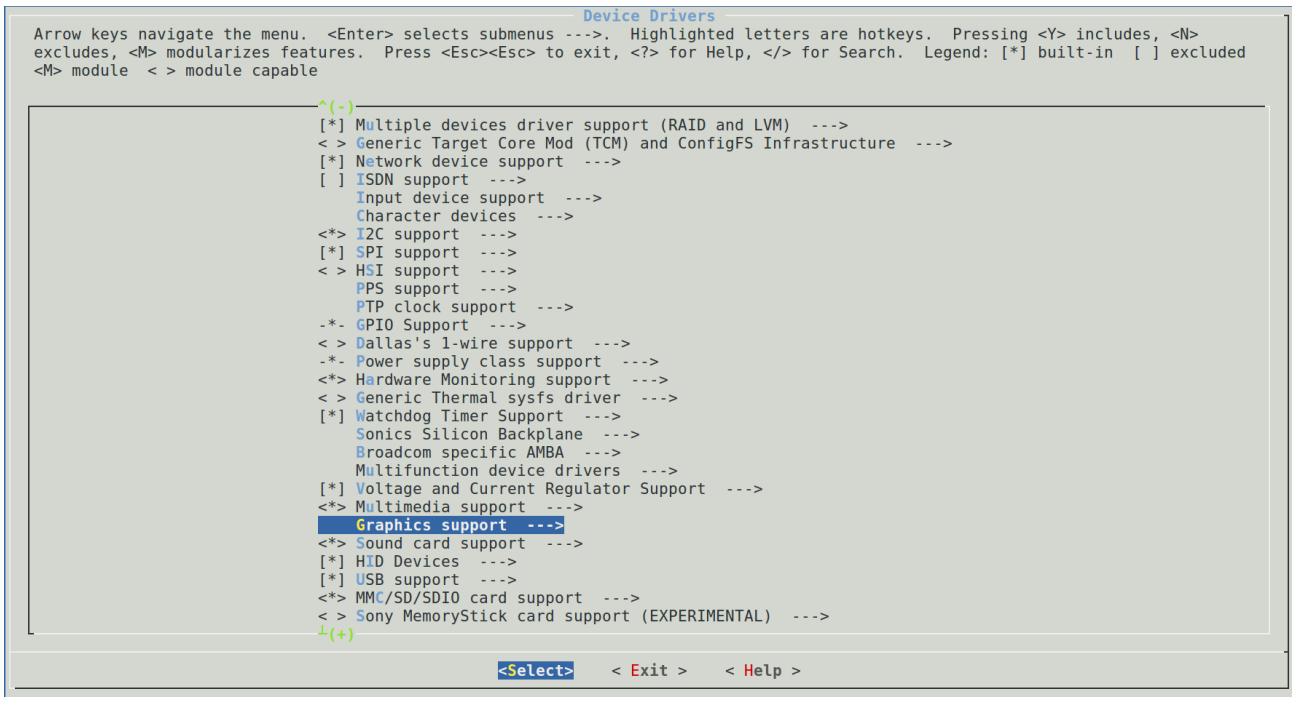
2）运行 make menuconfig 命令配置内核对 framebuffer 的相关支持

```
cbt@Cyb-Bot: /CBT-6818/SRC/linux/kernel $
cbt@Cyb-Bot: /CBT-6818/SRC/linux/kernel $ cp 6618_linux_config    .config
cbt@Cyb-Bot: /CBT-6818/SRC/linux/kernel $ make menuconfig
```

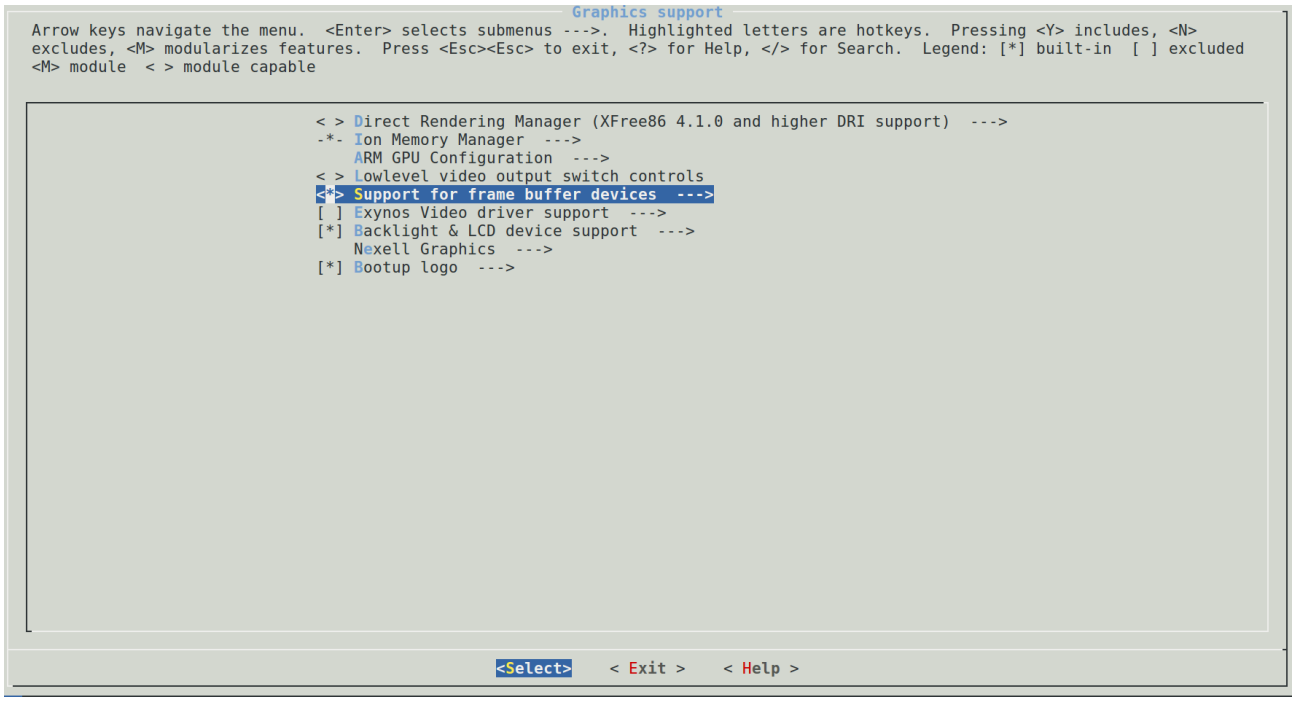
选择 Device Drivers ---->选项，如图



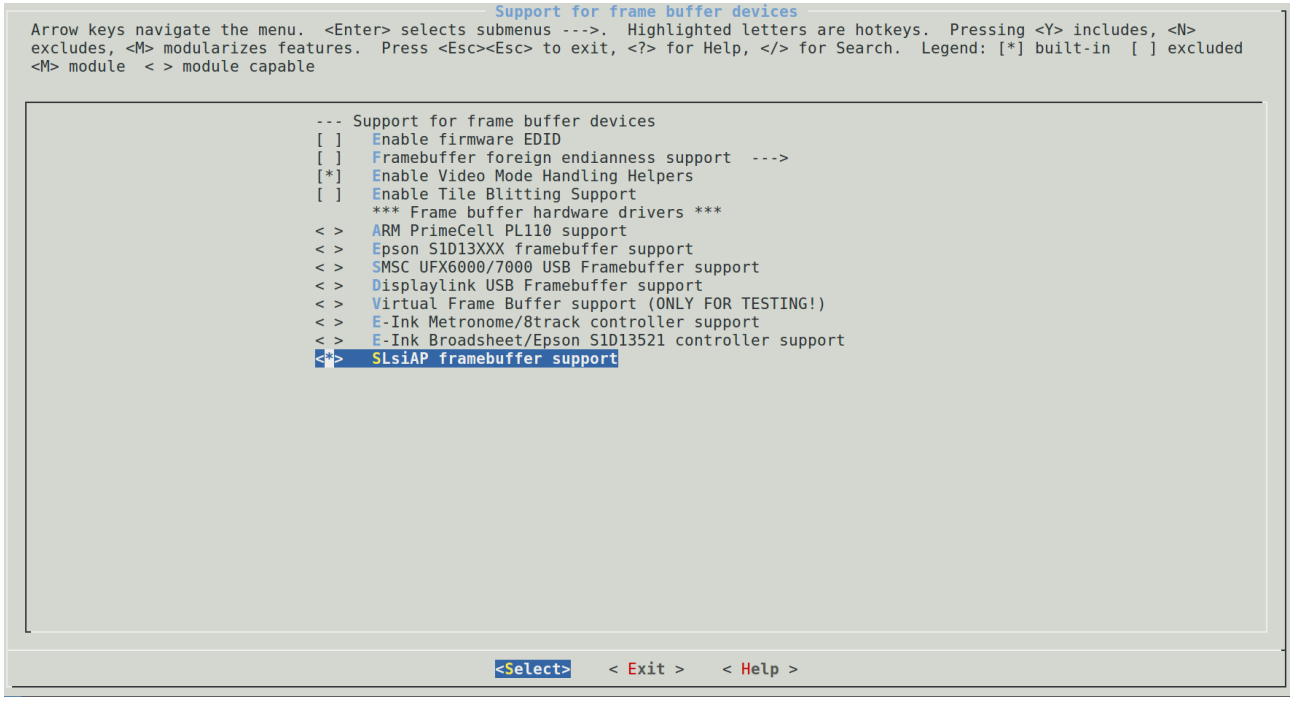
选择 Graphics support ---> 支持，如图：



选择 Support for frame buffer devices ---> 支持，如图：



选中 <*>SLsiAP dramebuffer support 选项，如图：



退出保存设置，如图：

3）重新编译内核,运行 make 命令

```
cbt@Cyb-Bot: /CBT-6818/SRC/linux/kernel $ make
  CHK     include/linux/version.h
  CHK     include/generated/utsrelease.h
make[1]: “include/generated/mach-types.h”是最新的。
  CALL    scripts/checksyscalls.sh
  CHK     include/generated/compile.h
.....
```

最终在 linux 源码目录的 out/release 目录下生成新的 ARM-LINUX 内核镜像文件 boot.img

```
cbt@Cyb-Bot: /CBT-6818/SRC/linux/kernel $ ls ../out/release/boot.img
boot.img
cbt@Cyb-Bot: /CBT-6818/SRC/linux/kernel $
```

4）按照烧写文档将新生成的内核镜像文件 boot.img 烧写到设备中，这里不在赘述。

备注：以上在内核中添加对 framebuffer 设备的支持的步骤，在 ARM 系统设备出厂自带内核中已经默认添加进来了，用户可以省略以上步骤。以上步骤在于重现系统的构造。

♦ 编译测试应用程序

1）进入实验目录：

```
cbt@Cyb-Bot: ~$ cd /CBT-6818/SRC/exp/driver/04_lcd/
cbt@Cyb-Bot: /CBT-6818/SRC/exp/driver/04_lcd $ ls
lcdtest   lcdtest.c   Makefile
```

2）清除中间代码，重新编译

```
cbt@Cyb-Bot: /CBT-6818/SRC/exp/driver/04_lcd $ make clean
rm -f *.o a.out lcdtest *.gdb
cbt@Cyb-Bot: /CBT-6818/SRC/exp/driver/04_lcd $ make
arm-linux-gcc       lcdtest.c     -o lcdtest
```

```
cbt@Cyb-Bot: /CBT-6818/SRC/exp/driver/04_lcd $
```

当前目录下生成测试程序 lcdtest。

♦ 运行程序（NFS 方式）

1）启动 CBT-6818 型实验系统，连好网线、串口线。通过串口终端挂载宿主机实验目录。

```
[root@CBT-6818 /$ mount -t nfs -o nolock 192.168.1.7:/CBT-6818/    /mnt/
```

2）进入串口终端的 NFS 共享实验目录。

```
[root@CBT-6818 /$ cd /mnt/SRC/exp/driver/04_lcd/
[root@CBT-6818 /mnt/SRC/exp/driver/04_lcd$ ls
Makefile      lcdtest       lcdtest.c
```

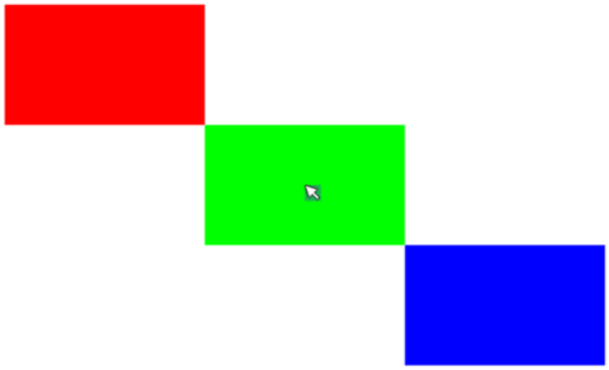
3）执行应用程序测试该驱动及设备

```
[root@CBT-6818 /mnt/SRC/exp/driver/04_lcd$ ./lcdtest
```

♦ 实验结果

```
[root@CBT-6818 /mnt/SRC/exp/driver/04_lcd$ ./lcdtest
Fixed screen info:
        id: s3cfb
        smem_start: 0x29723000
        smem_len: 3072000
        type: 0
        line_length: 3200

Variable screen info:
        xres: 800
        yres: 480
        bits_per_pixel: 32
        red: offset: 16, length:    8, msb_right:    0
        green: offset:    8, length:    8, msb_right:    0
        blue: offset:    0, length:    8, msb_right:    0
        height: 96
        width: 154

Will draw 3 rectangles on the screen,
they should be colored red, green and blue (in that order).
  Done.
[root@CBT-6818 /mnt/SRC/exp/driver/04_lcd$
```

此时观察 ARM 系统 LCD 显示屏矩形绘图显示结果。



实验五. SD 卡接口实验

1. 实验目的

- 学习 SD 卡规范的概念及驱动的基本流程。
- 掌握 Linux 内核中关于 MMC/SD 卡设备的配置。

2. 实验环境

- 硬件：A53 智能网关实验平台，PC 机 Pentium 500 以上，硬盘 40G 以上，内存大于 256M。
- 软件：Vmware Workstation +ubuntu + MiniCom/超级终端 +ARM-LINUX 交叉编译开发环境。

3. 实验内容

- 根据所提供的 SD 卡原理图，SD 卡的读写时序，SD 规范理解 SD 的驱动函数。
- 在 Linux 内核源码中配置对 MMC/SD 卡设备的支持，并在系统下挂载 SD 卡设备测试。

4. 实验原理

4.1 硬件接口原理

♦ **SD 卡简介**

SD 卡是 Secure Digital Card 卡的简称，直译成汉语就是“安全数字卡”，是由日本松下公司、东芝公司和美国 SANDISK 公司共同开发研制的全新的存储卡产品。SD 存储卡是一个完全开放的标准（系统），多用于 MP3、数码摄像机、数码相机、电子图书、AV 器材等等，尤其是被广泛应用在超薄数码相机上。SD 卡在外形上同 MultiMedia Card 卡保持一致，大小尺寸比 MMC 卡略厚，容量也大很多。并且兼容 MMC 卡接口规范。SD 卡最大的特点就是通过加密功能，可以保证数据资料的安全保密。它还具备版权保护技术，所采用的版权保护技术是 DVD 中使用的 CPRM 技术（可刻录介质内容保护）。

SD 卡通信基于 9 芯的接口（Clock, Command,4xDat，3xPower lines），最大的操作频率是 25MHz。SD 卡规范包括多个文档，各文档之间的结构如图 4.1.1 所示：

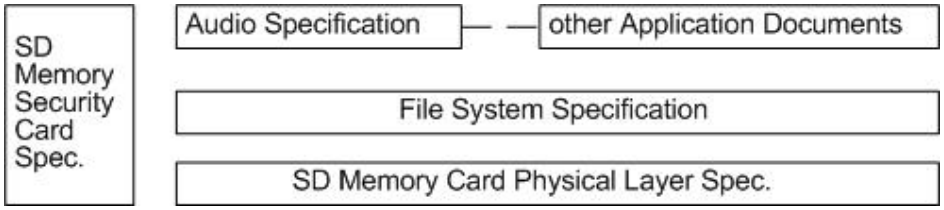


图 4.1.1 文档结构

♦ **SD 卡的总线拓扑**

SD 卡系统支持两种通信协议：SD 和 SPI 方式。模式的选择对主机是透明的，由 SD 卡自动检测复位命令的模式，在此后的通信过程中始终使用此种通信方式。SD 卡在结构上使用一主多从星型拓扑结构。拓扑图如图 4.1.2 所示：

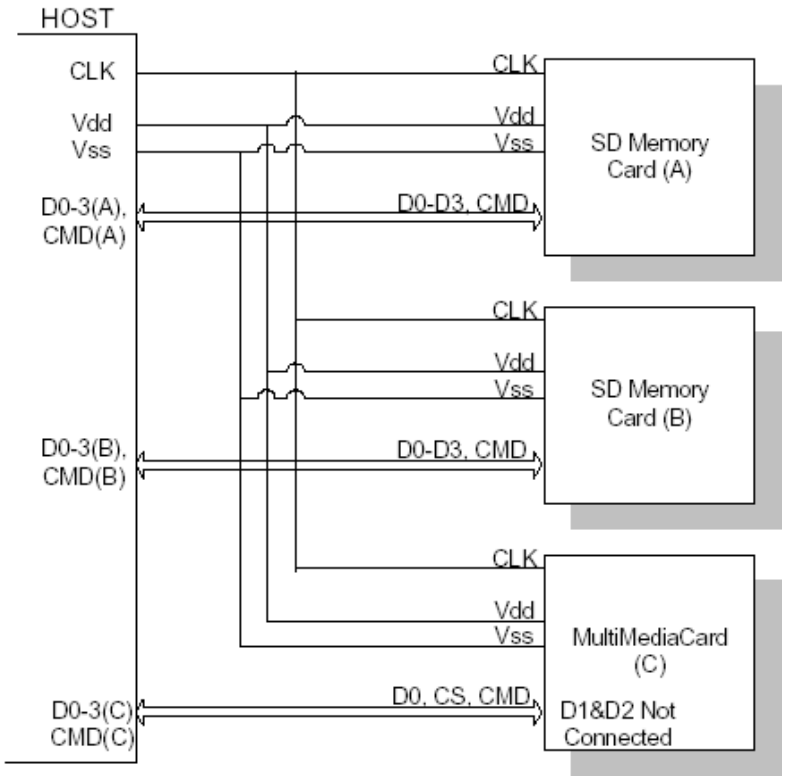


图 4.1.2 SD 卡系统的总线拓扑图

♦ **SD 总线信号**

CLK：时钟信号；

CMD：命令/相应信号；

DAT0-DAT3：双向数据传输信号；

VDD，VSS1，VSS2：电源和地信号；

其原理图如图 4.1.3 所示：

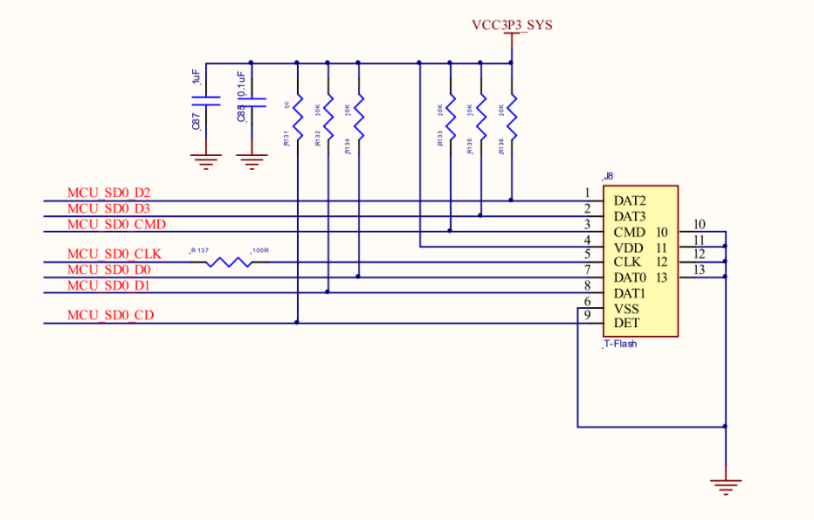


图 4.1.3 SD 卡原理图

从电路原理图上可以看出，SD 分别使用 S5P6818 处理自带的控制器端口作为 4 根数据信号线、使用的信号线和时钟信号线也同样连接在处理器内部的 MMC 控制器相应的信号线上。

♦ SD 总线协议

SD 总线上的通信基于位流的方式，在位流中实现传输命令和数据，包含起始位和停止位。

CMD：命令发起一个操作过程。命令可分为地址方式（主机到单个 SD 卡）或者广播方式（主机到所有的 SD 卡）。

Response：是卡对前一个命令的回应，通过 CMD 线传输。

DAT：通过数据线传输。

SD 卡传输数据的单位是块，块数据之后是 CRC 位段。SD 卡传输定义单块和多块的传输。其中，多块传输在快速写入中优于单块传输。在数据传输的过程中，可以使用单数据线（DAT0）或者多数据线（DAT0－DAT3）。

在 CMD 线上，数据传输的次序是先传输高位后传低位。

♦ 读块时序

读块时序如图 4.1.4 所示：

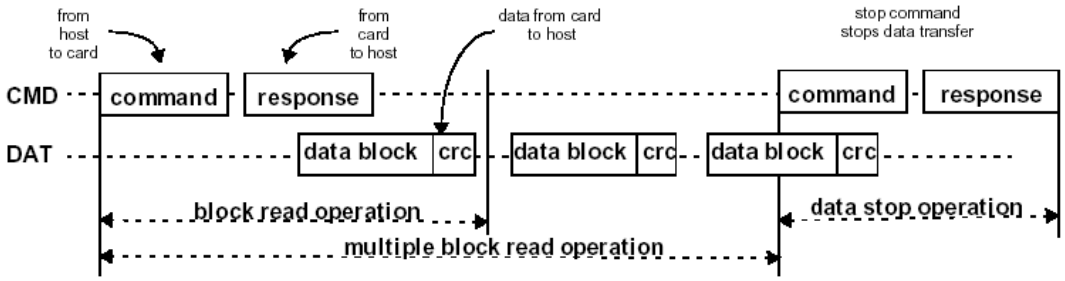


图 4.1.4 读块时序图

♦ **写块时序**

写块时序图如图 4.1.5 所示：

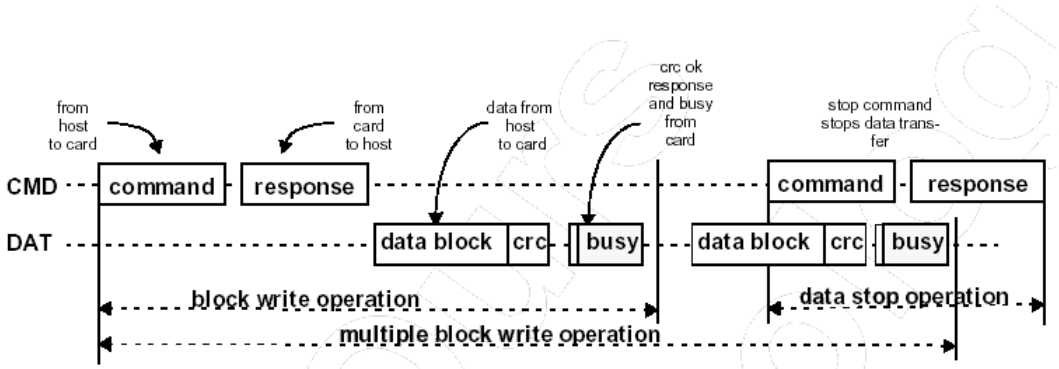


图 4.1.5 写块时序图

♦ **SD 卡外型和接口**

标准 SD 的外形尺寸是 24mm×32mm×2.1mm，如图 4.1.6 所示：

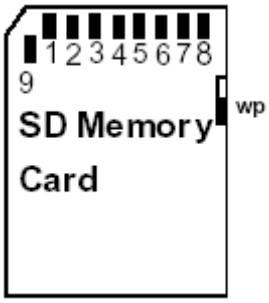


图 4.1.6 SD 卡的外形和接口

表 SD 卡引脚定义如表 4.1.1 所示：

Pin #	SD Mode			SPI Mode		
	Name	Type[1]	Description	Name	Type	Description
1	CD/DAT3[2]	I/O/PP[3]	Card Detect / Data Line [Bit 3]	CS	I	Chip Select (neg true)
2	CMD	PP	Command/Response	DI	I	Data In
3	V_{SS1}	S	Supply voltage ground	VSS	S	Supply voltage ground
4	V_{DD}	S	Supply voltage	VDD	S	Supply voltage
5	CLK	I	Clock	SCLK	I	Clock
6	V_{SS2}	S	Supply voltage ground	VSS2	S	Supply voltage ground
7	DAT0	I/O/PP	Data Line [Bit 0]	DO	O/PP	Data Out
8	DAT1	I/O/PP	Data Line [Bit 1]	RSV		
9	DAT2	I/O/PP	Data Line [Bit 2]	RSV		

表 4.1.1 表 SD 卡引脚定义

S：供电，I：输入，O：输出，使用推挽驱动，PP：IO 使用推挽方式。

♦ **SD 卡命令介绍**

SD 卡的命令有四种类型：

- 无响应广播命令
- 带响应广播命令。各个卡的响应同时进行，这种类型的命令仅用于所用的 CMD 线是分立的一命令和响应会在每根 CMD 线上单独进行。
- 带地址命令一DAT 上无数据传输
- 带地址命令一DAT 上有数据传输

SD 卡的命令格式如表 4.1.2 所示：

Bit position	47	46	[45:40]	[39:8]	[7:1]	0
Width (bits)	1	1	6	32	7	1
Value	'0'	'1'	x	x	x	'1'
Description	start bit	transmission bit	command index	argument	CRC7	end bit

表 4.1.2 SD 卡的命令格式

♦ SD 卡寄存器

SD 卡的寄存器描述如表 4.1.3 所示：

名称	位宽	描述
CID	128	卡 ID 号寄存器，每个卡唯一（必有）
RCA	16	卡相对地址寄存器，卡在系统中的局部地址。在初始化的过程中由卡申请，最终由主机确定（必有）
DSR	16	驱动电压配置寄存器，配置卡的输出驱动。（可选）
CSD	128	SD 特定数据寄存器，存储关于卡的操作条件。（必有）
SCR	64	SD 配置寄存器，存储关于卡的特征和性能（必有）
OCR	32	操作条件寄存器（必有）

表 4.1.3 SD 卡的寄存器描述

SD 卡接口定义了六个寄存器：OCR，CID，CSD，RCA，DSR 和 SCR。这些寄存器仅可以通过相应的命令来读取。OCR，CID，CSD 和 SCR 寄存器包含卡的状态信息，而 RCA 和 DSR 寄存器存储卡的实际配置参数。

♦ OCR 寄存器

32 位操作条件寄存器保存有 SD 卡的 VDD 电压配置。另外，该寄存器包含一个状态信息位。当上电过程结束后，状态信息位就会被置位。设置 SD 卡 OCR 寄存器的作用是为了操作不支持全操作电压范围的 SD 卡。OCR 寄存器的定义如表 4.1.4 所示：

OCR bit position	VDD voltage window
0-3	reserved
4	1.6-1.7
5	1.7-1.8
6	1.8-1.9
7	1.9-2.0
8	2.0-2.1
9	2.1-2.2
10	2.2-2.3
11	2.3-2.4
12	2.4-2.5
13	2.5-2.6
14	2.6-2.7
15	2.7-2.8
16	2.8-2.9
17	2.9-3.0
18	3.0-3.1
19	3.1-3.2
20	3.2-3.3
21	3.3-3.4
22	3.4-3.5
23	3.5-3.6
24-30	reserved
31	card power up status bit (busy)[1]

表 4.1.4 OCR 寄存器定义列表

♦ **CID 寄存器**

SD 卡标识寄存器长度 128 位。包括若干卡识别信息。每个 SD 卡都有唯一的一个标识。CID 寄存器的结构如表 4.1.5 所示：

Name	Field	Width	CID-slice
Manufacturer ID	MID	8	[127:120]
OEM/Application ID	OID	16	[119:104]
Product name	PNM	40	[103:64]
Product revision	PRV	8	[63:56]
Product serial number	PSN	32	[55:24]
reserved	--	4	[23:20]
Manufacturing date	MDT	12	[19:8]
CRC7 checksum	CRC	7	[7:1]
not used, always '1'	-	1	[0:0]

表 4.1.5 CID 数据域定义

♦ **CSD 寄存器**

SD 卡相关数据寄存器－提供如何访问卡中内容的信息。CSD 定义数据格式，错误校正类型，最大数据访问时间，是否使用 DSR 寄存器。通过使用命令 CMD27 来改变该寄存器中的可更改内容。如表 4.1.6 所示：

Name	Field	Width	Cell Type	CSD-slice
CSD structure	CSD_STRUCTURE	2	R	[127:126]
reserved	-	6	R	[125:120]
data read access-time-1	TAAC	8	R	[119:112]
data read access-time-2 in CLK cycles (NSAC*100)	NSAC	8	R	[111:104]
max. data transfer rate	TRAN_SPEED	8	R	[103:96]
card command classes	CCC	12	R	[95:84]
max. read data block length	READ_BL_LEN	4	R	[83:80]
partial blocks for read allowed	READ_BL_PARTIAL	1	R	[79:79]
write block misalignment	WRITE_BLK_MISALIGN	1	R	[78:78]
read block misalignment	READ_BLK_MISALIGN	1	R	[77:77]
DSR implemented	DSR_IMP	1	R	[76:76]
reserved	-	2	R	[75:74]
device size	C_SIZE	12	R	[73:62]
max. read current @V_{DD} min	VDD_R_CURR_MIN	3	R	[61:59]
max. read current @V_{DD} max	VDD_R_CURR_MAX	3	R	[58:56]
max. write current @V_{DD} min	VDD_W_CURR_MIN	3	R	[55:53]
max. write current @V_{DD} max	VDD_W_CURR_MAX	3	R	[52:50]
device size multiplier	C_SIZE_MULT	3	R	[49:47]
erase single block enable	ERASE_BLK_EN	1	R	[46:46]
erase sector size	SECTOR_SIZE	7	R	[45:39]
write protect group size	WP_GRP_SIZE	7	R	[38:32]
write protect group enable	WP_GRP_ENABLE	1	R	[31:31]
reserved for MultiMediaCard compatibility		2	R	[30:29]
write speed factor	R2W_FACTOR	3	R	[28:26]
max. write data block length	WRITE_BL_LEN	4	R	[25:22]
partial blocks for write allowed	WRITE_BL_PARTIAL	1	R	[21:21]
reserved	-	5	R	[20:16]
File format group	FILE_FORMAT_GRP	1	R/W(1)	[15:15]
copy flag (OTP)	COPY	1	R/W(1)	[14:14]
permanent write protection	PERM_WRITE_PROTECT	1	R/W(1)	[13:13]
temporary write protection	TMP_WRITE_PROTECT	1	R/W	[12:12]
File format	FILE_FORMAT	2	R/W(1)	[11:10]
reserved		2	R/W	[9:8]
CRC	CRC	7	R/W	[7:1]
not used, always`1`	-	1	-	[0:0]
copy flag (OTP)	COPY	1	R/W(1)	[14:14]

表 4.1.6 CSD 数据域定义

♦ **RCA 寄存器**

可写 16 位相对地址寄存器，由卡在 SD 识别器件发送。相对地址用于 SD 卡与主机之间进行通讯。缺省的 RCA 值是 0x0000。此值同样用于设置所有的卡进入 Stand-by 状态（CMD7）

♦ **DSR 寄存器**

上文有述，DSR 寄存器主要用于设定 SD 卡的驱动电平范围。用以提高 SD 卡的总线性能。缺省 DSR 值为 0x404。

♦ **SCR 寄存器**

提供 SD 中的配置状态。其结构如表 4.1.7 所示：

Description	Field	Width	Cell Type	SCR Slice
SCR Structure	SCR_STRUCTURE	4	R	[63:60]
SD Memory Card - Spec. Version	SD_SPEC	4	R	[59:56]
data_status_after erases	DATA_STAT_AFTER_ERASE	1	R	[55:55]
SD Security Support	SD_SECURITY	3	R	[54:52]
DAT Bus widths supported	SD_BUS_WIDTHS	4	R	[51:48]
reserved	-	16	R	[47:32]
reserved for manufacturer usage	-	32	R	[31:0]

表 4.1.7 SD 数据域定义

4. 2SD 卡详细介绍

主机和 SD 卡之间的通讯过程由主机统一控制。主机发出的命令由两种类型：广播命令和地址（点到点）命令。

根据协议，MMC/SD 卡的驱动被分为：卡识别阶段和数据传输阶段。在卡识别阶段通过命令使 MMC/SD 处于：空闲(idle)、准备(ready)、识别(ident)、等待(stby)、不活动(ina)几种不同的状态；而在数据传输阶段通过命令使 MMC/SD 处于：发送(data)、传输(tran)、接收(rcv)、程序(prg)、断开连接(dis)几种不同的状态。所以可以总结 MMC/SD 在工作的整个过程中分为两个阶段和十种状态。

SD 卡通讯中使用两种模式：卡识别模式和数据传输模式。

♦ **卡识别模式**

主机复位后进入卡识别模式来搜索新卡。卡会保持此种模式直到收到命令 SEND_RCA（CMD3）。

在此种模式中，主机复位所有的 SD 卡，校验操作电压范围，请求 SD 相对卡地址（RCA）。此种操作通过选择它们的命令线依次对每个 SD 卡进行。在卡识别模式下进行的数据通信全部在 CMD 线上传输的。

在卡识别模式的数据状态图 4.1.7：

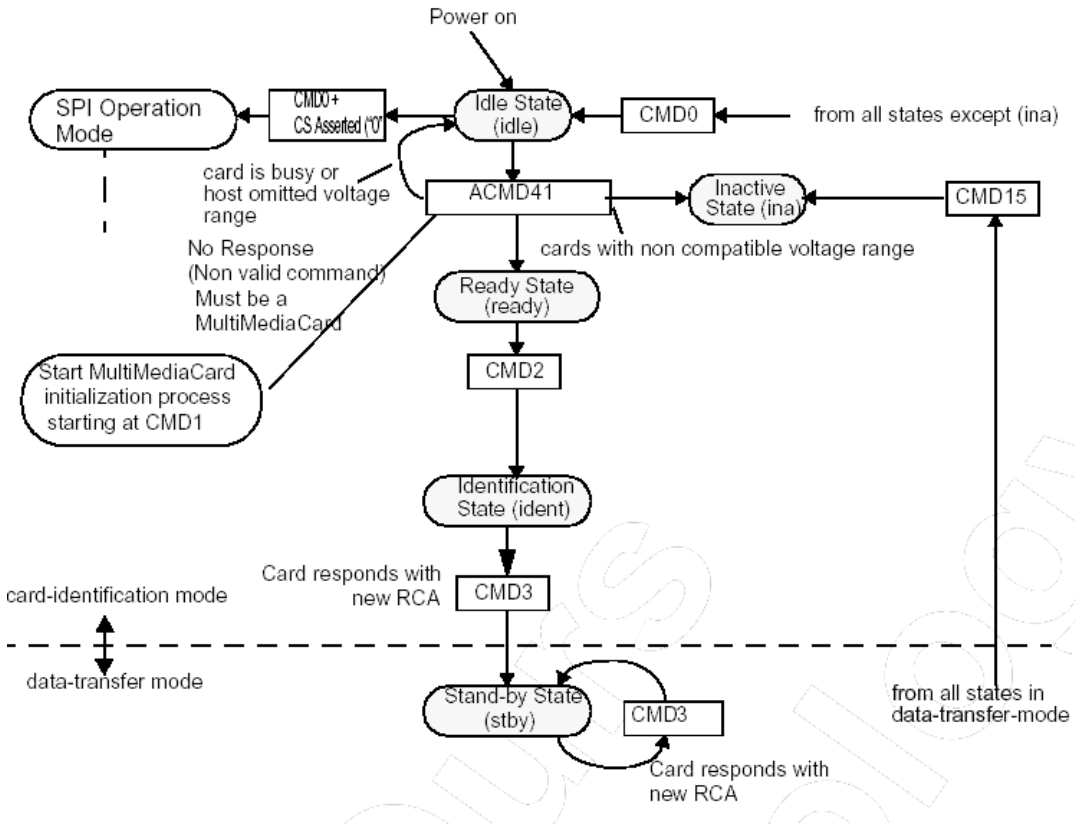


图 4.1.7 SD 卡识别过程

复位后总线进入有效的状态，主机会分别请求每个 SD 卡发出他们的有效操作条件（APP_CMD_CMD55 + ACMD41，其中，RCA=0x0000）。ACMD41 命令的响应是 OCR 寄存器的内容。不兼容的卡就会进入 Inactive 状态。此后，主机发出 ALL_SEND_CID (CMD2)命令，获取每个卡的 ID 号。当 SD 卡发送 CID 数据后，进入 Identification 状态。之后，主机发送 CMD3 (SEND_RELATIVE_ADDR)命令请求卡发送其新的 RCA 地址（发送之后，卡进入 the Stand-by State）。此时，如果主机要求 SD 卡更换 RCA 地址的话就会重复发送 CMD3 要求新的 RCA 地址。SD 卡最终发送的 RCA 地址就是实际的 RCA。

♦ **数据传输模式**

卡识别过程结束后，进入数据传输模式。在此模式中，主机发送 SEND_CSD (CMD9)命令获取卡的 CSD 寄存器的内容（例如：传输块的长度，卡存储容量）。

发送 SET_DSR (CMD4)配置所有识别卡的驱动电压范围。

发送 CMD7 来选择一个卡进入传输状态（Transfer State），在任意时刻，只有一个卡可以处于该状态。当发送的 CMD7 中的 RCA 为“0x0000”时，所有的卡进入 Stand-by 状态，如图 4.1.8 所示。

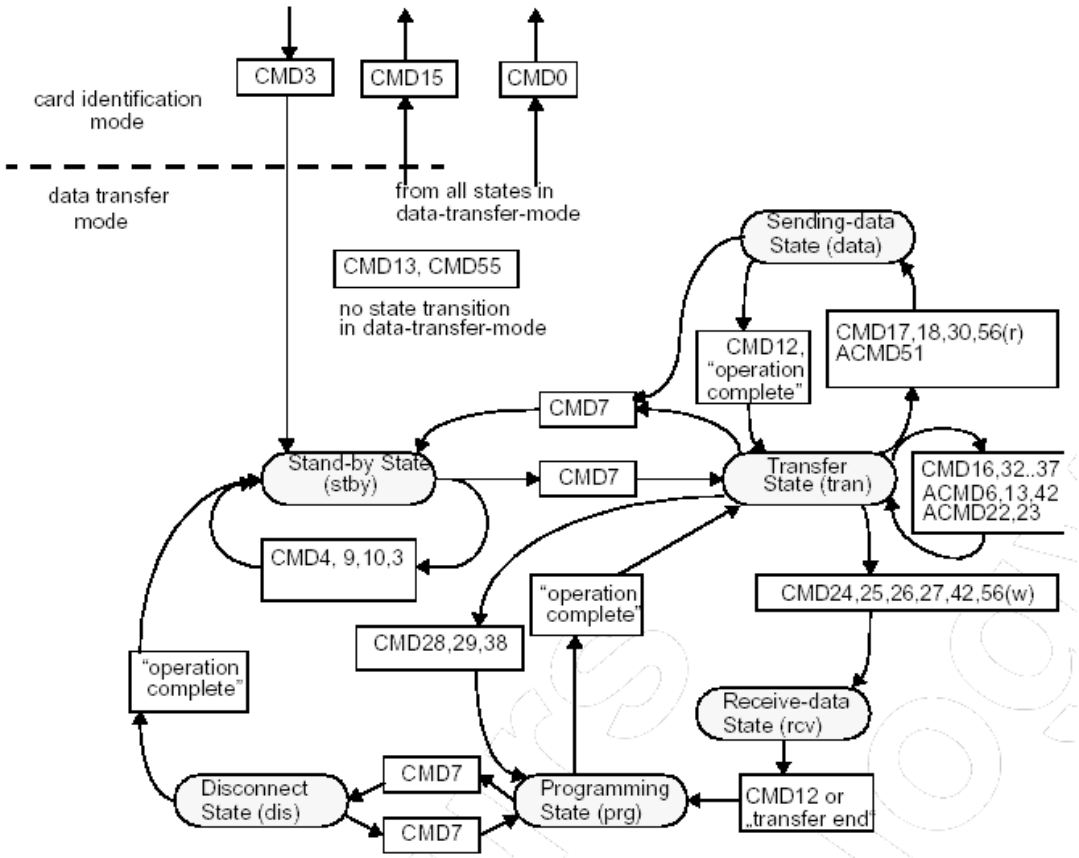


图 4.1.8 SD 卡状态图（数据传输模式）

数据传输模式摘要如下：

所有的数据读取命令在主机发出 CMD12 之后被中断。数据传输过程被中断并返回到 Transfer 状态。读取命令包括：CMD17-读取块，CMD18-读取多个块，CMD30-发送写保护，ACMD51-发送 SCR，CMD56-读入模式通用命令。

所有的数据写入命令在主机发出 CMD12 之后被中断。写命令必须在 SD 卡处于非选定状态之前停止（卡状态切换命令：CMD7）。写命令包括：CMD24、CMD25-写块，CMD26-写 CSD，CMD42-锁和解锁命令，CMD56-写模式通用命令。

当数据发送完毕后，SD 卡退出数据写入状态返回到如下的两种状态：编程状态（发送成功）或者传输状态（发送失败）。

如果写入块的操作停止后，块长和 CRC 校验有效，数据就会在 SD 卡中编程。

SD 卡提供数据缓冲的功能用于写块，如果写缓冲区满，只要卡仍然处于编程状态，数据线 DAT0 就会保持低（忙状态）。

对于命令 CSD、CID、写保护和擦除操作没有相应的数据缓冲区。

编程中不能使用参数设置命令。

参数设置命令包括：设定块长度（CMD16）、擦除快起始（CMD32）、擦除快中止（CMD33）

编程中不可以使用读命令。

擦除和编程中 CMD7 无效。擦除和编程完成后 SD 卡会进入 Disconnect 状态释放数据线。使用命令 CMD7，SD 卡可以重新选择在 Disconnect 状态。

复位 SD 卡（命令 CMD0 和 CMD15）将结束任何处于等待或正在进行的编程操作，这会破坏卡中的保存的数据的内容。

4.3 Linux 内核中 MMC/SD 设备驱动

Linux 内核中 MMC/SD 驱动可以分为 3 层：块设备层（mmc_block.c ，mmc_sysfs.c，mmc_queue.c）、MMC 协议层（mmc.c）、SD 驱动层（s3c-hsmmc.c）。其中主要源文件存放在内核源码目录的 drivers/mmc/目录下。分别有 card、core 和 host 三个文件夹。

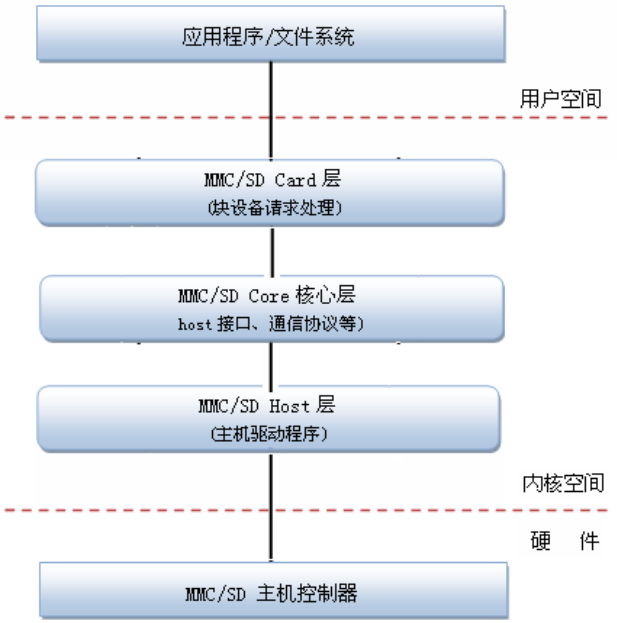


图 4.2.1 SD 卡驱动层次

整个 MMC/SD 模块中最重要的部分是 Core 核心层，他提供了一系列的接口函数，对上提供了将主机驱动注册到系统，给应用程序提供设备访问接口，对下提供了对主机控制器控制的方法及块设备请求的支持。

mmc.c 文件是内核中关于 MMC 命令及协议的功能实现，一般无需修改。

mmc_sysfs.c 文件中主要完成 MMC 总线注册及 MMC class 的注册。并提供 MMC 设备驱动相关初始化函数接口。

mmc_block.c 文件中主要完成 MMC 块设备的注册以及向 MMC 总线注册 driver 的工作。

4.4 关键代码分析

下面主要介绍下 SD 控制器驱动层，即 sdhci-s3c.c 文件。

主控制器驱动结构分配及初始化函数 sdhci_s3c_probe(位于 drivers/mmc/host/sdhci-s3c.c)：

```
static int __devinit sdhci_s3c_probe(struct platform_device *pdev)
{
// 定义和初始化 SD 卡平台数据结构
```

```
        struct s3c_sdhci_platdata *pdata = pdev->dev.platform_data;
        struct device *dev = &pdev->dev;
        struct sdhci_host *host;
        struct sdhci_s3c *sc;
        struct resource *res;
        int ret, irq, ptr, clks;

        if (!pdata) {
                dev_err(dev, "no device data specified\n");
                return -ENOENT;
        }
// 获取平台相关的数据参数：IRQ 中断源
        irq = platform_get_irq(pdev, 0);
        if (irq < 0) {
                dev_err(dev, "no irq specified\n");
                return irq;
        }
// 获取平台相关的寄存器资源
        res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
        if (!res) {
                dev_err(dev, "no memory specified\n");
                return -ENOENT;
        }
// 分配 host 结构内存空间，建立扫描工作队列
        host = sdhci_alloc_host(dev, sizeof(struct sdhci_s3c));
        if (IS_ERR(host)) {
                dev_err(dev, "sdhci_alloc_host() failed\n");
                return PTR_ERR(host);
        }
// 初始化 sc 数据结构
        sc = sdhci_priv(host);

        sc->host = host;
        sc->pdev = pdev;
        sc->pdata = pdata;
// 设置平台驱动数据
        platform_set_drvdata(pdev, host);
// 获取并激活 SD 卡设备总线时钟
        sc->clk_io = clk_get(dev, "hsmmc");
        if (IS_ERR(sc->clk_io)) {
                dev_err(dev, "failed to get io clock\n");
                ret = PTR_ERR(sc->clk_io);
                goto err_io_clk;
        }

        /* enable the local io clock and keep it running for the moment. */
```

```
        clk_enable(sc->clk_io);

        for (clks = 0, ptr = 0; ptr < MAX_BUS_CLK; ptr++) {

                struct clk *clk;
                char *name = pdata->clocks[ptr];

                if (name == NULL)
                        continue;

                clk = clk_get(dev, name);
                if (IS_ERR(clk)) {
                        dev_err(dev, "failed to get clock %s\n", name);
                        continue;
                }

                clks++;
                sc->clk_bus[ptr] = clk;
                clk_enable(clk);

                dev_info(dev, "clock source %d: %s (%ld Hz)\n",
                         ptr, name, clk_get_rate(clk));
        }

        if (clks == 0) {
                dev_err(dev, "failed to find any bus clocks\n");
                ret = -ENOENT;
                goto err_no_busclks;
        }
// 设备地址空间映射，将设备物理寄存器地址映射为内核虚拟空间地址
        sc->ioarea = request_mem_region(res->start, resource_size(res),
                                        mmc_hostname(host->mmc));
        if (!sc->ioarea) {
                dev_err(dev, "failed to reserve register area\n");
                ret = -ENXIO;
                goto err_req_regs;
        }

        host->ioaddr = ioremap_nocache(res->start, resource_size(res));
        if (!host->ioaddr) {
                dev_err(dev, "failed to map registers\n");
                ret = -ENXIO;
                goto err_req_regs;
        }

        /* Ensure we have minimal gpio selected CMD/CLK/Detect */
```

```
        if (pdata->cfg_gpio)
                pdata->cfg_gpio(pdev, pdata->max_width);

        if (pdata->get_ro)
                sdhci_s3c_ops.get_ro = sdhci_s3c_get_ro;

        host->hw_name = "samsung-hsmmc";
        host->ops = &sdhci_s3c_ops;//此处的 sdhci_s3c_ops 实现了该 sd 卡驱动私有的硬件操作函数。

        host->quirks = 0;
        host->irq = irq;


        /* Setup quirks for the controller */
        host->quirks |= SDHCI_QUIRK_NO_ENDATTR_IN_NOPDESC;
        host->quirks |= SDHCI_QUIRK_BROKEN_CARD_PRESENT_BIT;
        host->quirks |= SDHCI_QUIRK_BROKEN_TIMEOUT_VAL;
  // DMA 模式的处理
#ifndef CONFIG_MMC_SDHCI_S3C_DMA

        /* we currently see overruns on errors, so disable the SDMA
          * support as well. */
        host->quirks |= SDHCI_QUIRK_BROKEN_DMA;

        /* PIO currently has problems with multi-block IO */
        host->quirks |= SDHCI_QUIRK_NO_MULTIBLOCK;

#endif /* CONFIG_MMC_SDHCI_S3C_DMA */

        /* It seems we do not get an DATA transfer complete on non-busy
          * transfers, not sure if this is a problem with this specific
          * SDHCI block, or a missing configuration that needs to be set. */
        host->quirks |= SDHCI_QUIRK_NO_BUSY_IRQ;

        host->quirks |= (SDHCI_QUIRK_32BIT_DMA_ADDR |
                         SDHCI_QUIRK_32BIT_DMA_SIZE);

        host->quirks |= SDHCI_QUIRK_NO_HISPD_BIT;

        if (pdata->host_caps)
                host->mmc->caps = pdata->host_caps;
        else
                host->mmc->caps = 0;

        /* Set pm_flags for built_in device */
        host->mmc->pm_caps = MMC_PM_KEEP_POWER | MMC_PM_IGNORE_PM_NOTIFY;
```

```
        if (pdata->built_in)
                host->mmc->pm_flags |= MMC_PM_KEEP_POWER |
MMC_PM_IGNORE_PM_NOTIFY;

        /* to add external irq as a card detect signal */
        if (pdata->cfg_ext_cd) {
                pdata->cfg_ext_cd();

                if (pdata->detect_ext_cd())
                        host->flags |= SDHCI_DEVICE_ALIVE;
        }

        /* to configure gpio pin as a card write protection signal */
        if (pdata->cfg_wp)
                pdata->cfg_wp();
 //初始化 sdhci_host 结构体完成后，向内核注册设备驱动
        ret = sdhci_add_host(host);
        if (ret) {
                dev_err(dev, "sdhci_add_host() failed\n");

                goto err_add_host;
        }

        /* register external irq here (after all init is done) */
        if (pdata->cfg_ext_cd) {// 注册中断资源
                ret = request_irq(pdata->ext_cd, sdhci_irq_cd,
                                IRQF_SHARED, mmc_hostname(host->mmc), sc);
                if(ret)
                        goto err_add_host;
        }

        return 0;
// 以下是出错处理部分，省略
```

从上述代码架构可以看出平台 SD 卡设备驱动架构采用了内核中 platform 总线架构设计，更多详情请自行阅读内核相关目录下的源码。

5. 实验步骤

♦ 在内核中添加 MMC/SD 卡设备支持

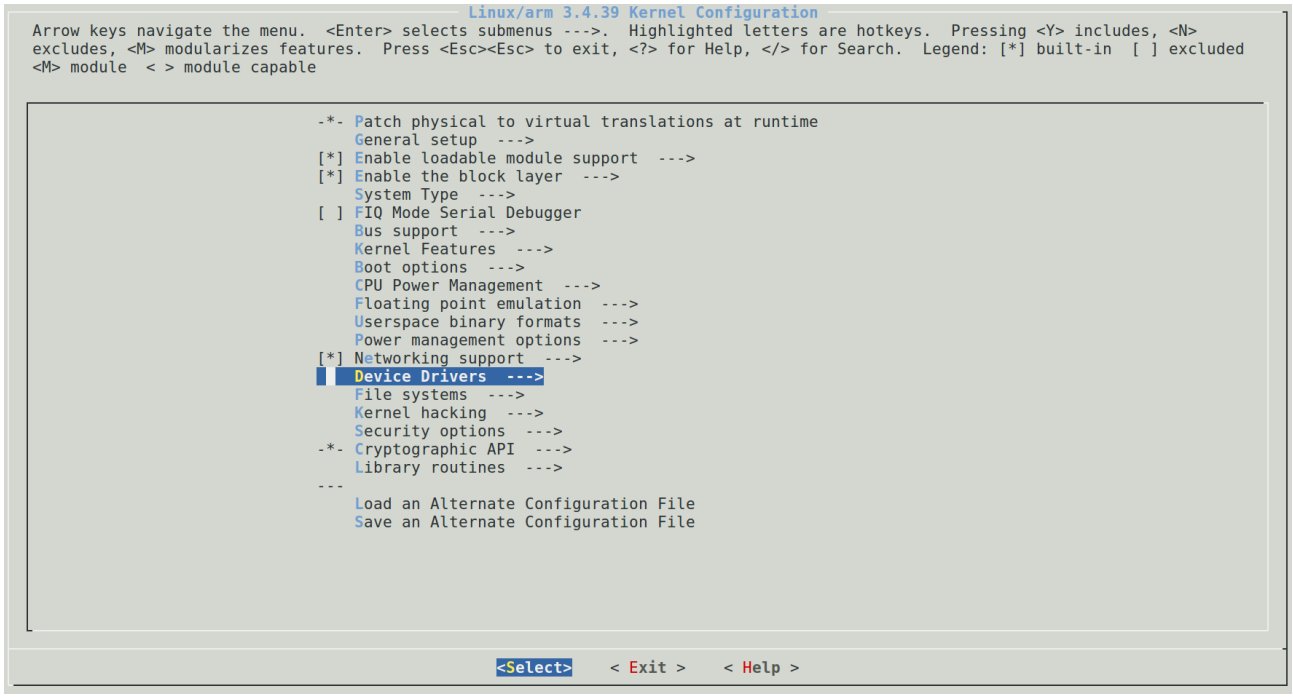
1）进入宿主机中 CBT-6818 型光盘内核目录：

```
cbt@Cyb-Bot:~$ cd /CBT-6818/SRC/linux/kernel/
```

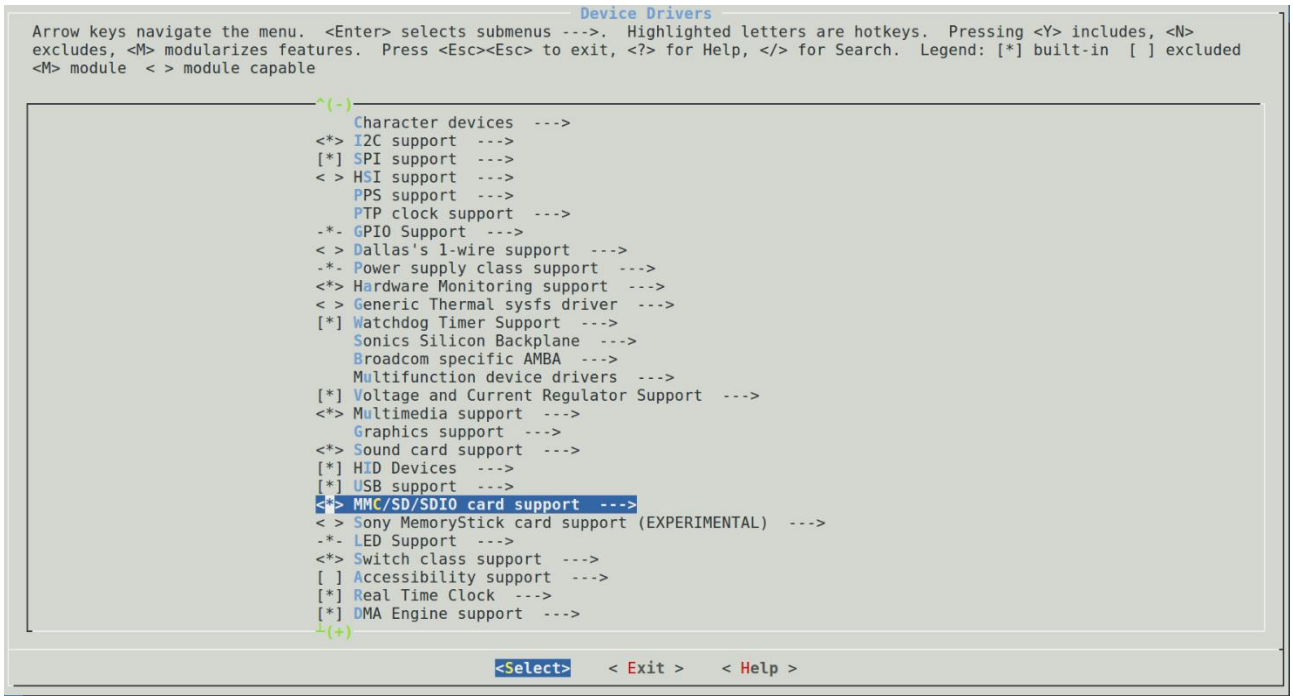
2）运行 make menuconfig 命令配置内核对 framebuffer 的相关支持

```
cbt@Cyb-Bot: /CBT-6818/SRC/linux/kernel $
cbt@Cyb-Bot: /CBT-6818/SRC/linux/kernel $ cp 6818_linux_config    .config
cbt@Cyb-Bot: /CBT-6818/SRC/linux/kernel $ make menuconfig
```

选择 Device Drivers --->选项，如图



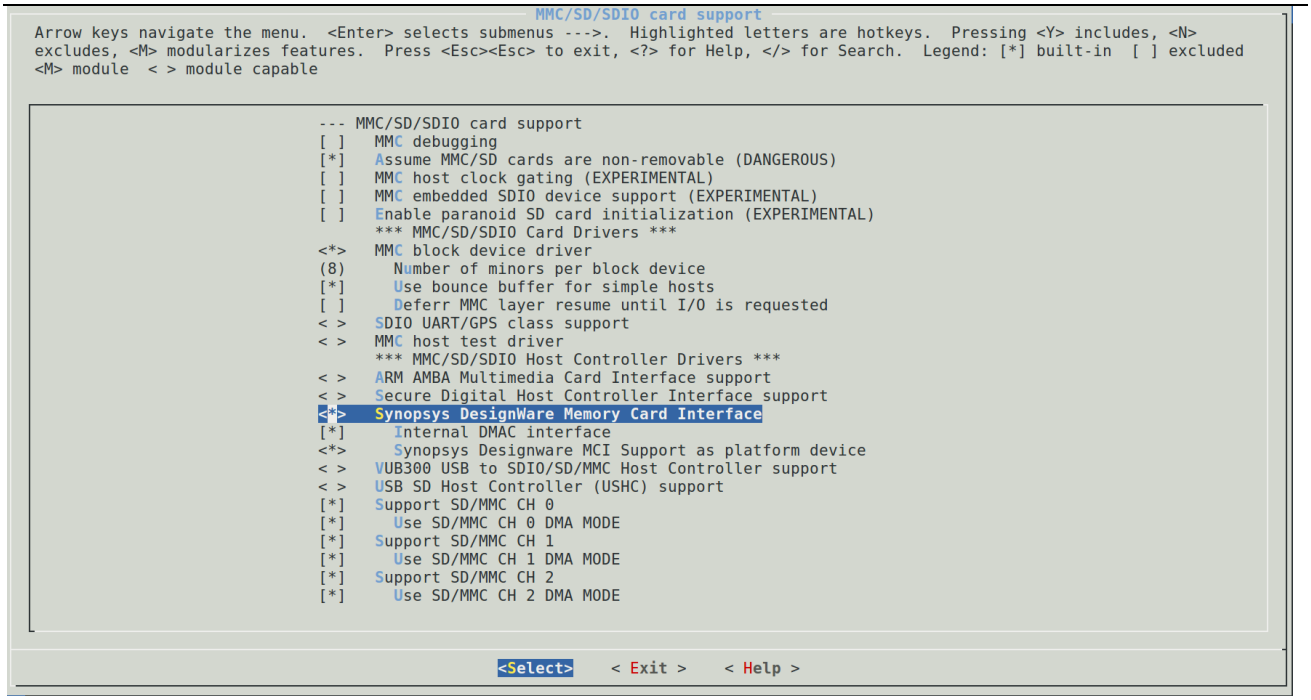
选择 <*> MMC/SD/SDIO card support ---> 支持，如图：



进入 MMC/SD/SDIO card support --->

选择 <*> Synopsys Designware Memory Card Interface

如图所示配置：



退出保存设置，如图：

3）重新编译内核,运行 make 命令

```
cbt@Cyb-Bot: /CBT-6818/SRC/linux/kernel $ make
  CHK     include/linux/version.h
  CHK     include/generated/utsrelease.h
make[1]: “include/generated/mach-types.h”是最新的。
  CALL    scripts/checksyscalls.sh
  CHK     include/generated/compile.h
.....
```

最终在 linux 源码目录的 out/release 目录下生成新的 ARM-LINUX 内核镜像文件 boot.img

```
cbt@Cyb-Bot: /CBT-6818/SRC/linux/kernel $ ls ../out/release/boot.img
boot.img
cbt@Cyb-Bot: /CBT-6818/SRC/linux/kernel $
```

4）按照 ICS-IOT-CEP 全功能物联网教学科研平台光盘配套烧写文档将新生成的内核镜像文件 zImage 烧写到 ARM 设备中，这里不在赘述。

备注：以上在内核中添加对 MMC/SD 卡设备的支持的步骤，在系统设备出厂自带内核中已经默认添加进入了，用户可以省略以上步骤。以上步骤在于重现系统的构造。

♦ 使用挂载 SD 卡测试

1）使用不小于 1G(FAT32 格式)的 TF 卡，插入平台底层的 TF 插槽内(请勿拔出，以免影响系统运行)。手动挂在 TF 卡到/tfcard/

```
[ 1949.927000] mmc_host mmc1: Bus speed (slot 0) = 25000000Hz (slot req 25000000Hz, actual 25000000HZ
div = 0)
[ 1949.931000] mmc1: new SDHC card at address 0001
[ 1949.936000] mmcblk1: mmc1:0001 ASTC 14.8 GiB
[ 1949.942000]    mmcblk1: p1
[root@CBT-6818:~$ mkdir /tfcard
[root@CBT-6818:~$ mount -t vfat /dev/mmcblk1p1 /tfcard/
```

2）卸载 SD 卡

```
[root@CBT-6818 /$ umount    /tfcard/
```

实验六. U 盘接口使用

1. 实验目的

- 了解 USB 接口的工作原理，掌握 Linux 下 USB 接口的配置和使用。
- 理解 Linux 下面对 usb 设备自动检测(hotplug)过程的基本原理。

2. 实验环境

- 硬件：A53 智能网关实验平台，PC 机 Pentium 500 以上，硬盘 40G 以上，内存大于 256M
- 软件：Vmware Workstation +ubuntu + MiniCom/超级终端 +ARM-LINUX 交叉编译开发环境

3. 实验内容

- 学习 USB 通信原理，了解 USB 通信协议的结构框架。学习在 Linux 内核中对 usb mass storage 设备的配置方法。
- 掌握在嵌入式 Linux 系统中如何使用 USB 设备并测试。

4. 实验原理

4.1 硬件接口原理

♦ **USB 设备接口**

USB 全称是"Universal Serial Bus"，意为"通用串行总线"，由 Compaq、DEC、IBM、Intel、NEC、微软以及 Northern Telecom 等公司于 1994 年 11 月共同提出的，主要目的就是为了解决接口标准太多的弊端。USB 使用一个 4 针插头作为标准插头，采用菊花瓣形式把所有外设连接起来，最多可连接 127 个外设。它采用串行方式传输数据，目前最大数据传输率为 12Mbps, 支持多数据流和多个设备并行操作，允许外设热插拔。

♦ **USB 接口规范**

USB 有两个规范，USB1.1 和 USB2.0。USB1.1 是目前较为普遍的 USB 规范，其高速方式的传输速率为 12Mbps，低速方式的传输速率为 1.5Mbps。

注意：这里的 b 是 Bit 的意思，1MB/s（兆字节/秒）=8MBPS（兆位/秒），12Mbps=1.5MB/s。

USB2.0 规范是由 USB1.1 规范演变而来的。它的传输速率达到了 480Mbps，折算为 MB 为 60MB/s，足以满足大多数外设的速率要求。USB 2.0 中的“增强主机控制器接口”

（EHCI）定义了一个与 USB 1.1 相兼容的架构。它可以用 USB 2.0 的驱动程序驱动 USB 1.1 设备。

♦ **SCSI 设备接口**

SCSI 是有别于 IDE 的一个计算机标准接口。现在大部分平板式扫描仪、CD-R 刻录机、MO 光磁盘机等渐渐趋向使用 SCSI 接口，加之 SCSI 又能提供一个高速传送通道，所以，接触到 SCSI 设备的用户会越来越多。Linux 支持很多种的 SCSI 设备，例如：SCSI 硬盘、SCSI 光驱、SCSI 磁带机。更重要的是，Linux 提供了 IDE 设备对 SCSI 的模拟（ide-scsi.o 模块），我们通常会就把 IDE 光驱模拟为 SCSI 光驱进行访问。因为在 Linux 中很多软件都只能操作 SCSI 光驱。例如大多数刻录软件、一些媒体播放软件。通常我们的 USB 存储设备，也模拟为 SCSI 硬盘而进行访问。

♦ **平台上的 USB HOST 接口**

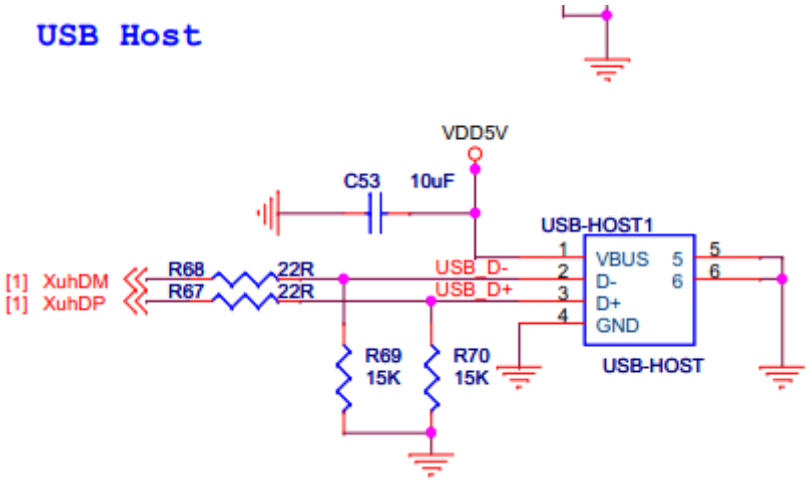


图 4.1.1 平台上的 USB HOST 连接

4.2 内核中 USB Mass Storage 驱动

linux 内核中自带的 usb mass storage 驱动位于内核源代码目录 drivers/usb/storage/下，核心文件为 usb.c、transport.c、scsiglue.c 和 protocol.c。

usb.c 文件是 Linux 内核中关于 Usb-storage 的核心文件，是整个驱动的框架代码。

transpot.c 文件实现了对于不同通讯方式的函数支持。

scsiglue.c 文件实现了 SCSI 设备的模拟函数。

protocol.c 文件实现了对于几种通讯协议的 SCSI 命令翻译函数

4.3 关键代码分析

下面主要介绍下 drivers/usb/storage/usb.c 文件中主要函数。

驱动程序入口 usb_stor_init 函数，向系统注册初始化 usb_storage_driver 结构驱动。

```
static int __init usb_stor_init(void)
{
        int retval;
        printk(KERN_INFO "Initializing USB Mass Storage driver...\n");
//注册 usb_register  驱动结构
        /* register the driver, return usb_register return code if error */
        retval = usb_register(&usb_storage_driver);
        if (retval == 0) {
                printk(KERN_INFO "USB Mass Storage support registered.\n");
                usb_usual_set_present(USB_US_TYPE_STOR);
        }
        return retval;
}
```

storage_probe 初始化函数：

```
static int storage_probe(struct usb_interface *intf,
                         const struct usb_device_id *id)
{
        struct us_data *us;
        int result;

        /*
        检查设备驱动是否为符合标准的设备类型
         */
        if (usb_usual_check_type(id, USB_US_TYPE_STOR) ||
                        usb_usual_ignore_device(intf))
                return -ENXIO;

        /*
         * Call the general probe procedures.
         *
            第一阶段检查 USB 设备，从注册过的设备标识中遍历查找。
         */
        result = usb_stor_probe1(&us, intf, id,
                        (id - usb_storage_usb_ids) + us_unusual_dev_list);

        if (result)
                return result;

        /*  第二阶段检查设备  */

        result = usb_stor_probe2(us);
        return result;
}
```

usb_stor_probe1 函数：

```
int usb_stor_probe1(struct us_data **pus,
                    struct usb_interface *intf,
                    const struct usb_device_id *id,
                    struct us_unusual_dev *unusual_dev)
{
        struct Scsi_Host *host;
        struct us_data *us;
        int result;

        US_DEBUGP("USB Mass Storage device detected\n");

        /*
             申请 scsi_host 结构空间
         */
        host = scsi_host_alloc(&usb_stor_host_template, sizeof(*us));
        if (!host) {
                dev_warn(&intf->dev,
                                "Unable to allocate the scsi host\n");
                return -ENOMEM;
        }
        host->max_cmd_len = 16;
        host->sg_tablesize = usb_stor_sg_tablesize(intf);
        *pus = us = host_to_us(host);// scsi_host  和 us_data 结构转换

        memset(us, 0, sizeof(struct us_data));
//初始化系统 us  结构相关的互斥锁、互斥量、完成量和等待队列
        mutex_init(&(us->dev_mutex));
        init_completion(&us->cmnd_ready);
        init_completion(&(us->notify));
        init_waitqueue_head(&us->delay_wait);
        init_completion(&us->scanning_done);

 //关联 USB 设备接口数据到 us_data 结构
        result = associate_dev(us, intf);
        if (result)
                goto BadDevice;

//获取初始化 us 结构设备信息
        result = get_device_info(us, id, unusual_dev);
        if (result)
                goto BadDevice;

//获取初始化 us 结构 transport、protocol 和 pipe 设置
        get_transport(us);
        get_protocol(us);
```

```
        /* Give the caller a chance to fill in specialized transport
         * or protocol settings.
        return 0;

BadDevice:
        US_DEBUGP("storage_probe() failed\n");
        release_everything(us);
        return result;
}
```

usb_stor_probe2 函数：

```
int usb_stor_probe2(struct us_data *us)
{
        struct task_struct *th;
        int result;
        struct device *dev = &us->pusb_intf->dev;

        /* Make sure the transport and protocol have both been set */
        if (!us->transport || !us->proto_handler) {
                result = -ENXIO;
                goto BadDevice;
        }
        US_DEBUGP("Transport: %s\n", us->transport_name);
        US_DEBUGP("Protocol: %s\n", us->protocol_name);

        /* fix for single-lun devices */
        if (us->fflags & US_FL_SINGLE_LUN)
                us->max_lun = 0;

        /* Find the endpoints and calculate pipe values */
        result = get_pipes(us);
        if (result)
                goto BadDevice;

 //获取初始化 us 结构 transport、protocol 和 pipe 设置
        result = usb_stor_acquire_resources(us);
        if (result)
                goto BadDevice;
        snprintf(us->scsi_name, sizeof(us->scsi_name), "usb-storage %s",
                                        dev_name(&us->pusb_intf->dev));
// 注册 scsi_host 驱动到系统中
        result = scsi_add_host(us_to_host(us), dev);
        if (result) {
                dev_warn(dev,
                                "Unable to add the scsi host\n");
                goto BadDevice;
```

```
        }


//建立 SCSI 设备扫描内核进程
        th = kthread_create(usb_stor_scan_thread, us, "usb-stor-scan");
        if (IS_ERR(th)) {
                dev_warn(dev,
                                "Unable to start the device-scanning thread\n");
                complete(&us->scanning_done);
                quiesce_and_remove_host(us);
                result = PTR_ERR(th);
                goto BadDevice;
        }

        wake_up_process(th);

        return 0;
        /* We come here if there are any problems */
BadDevice:
        US_DEBUGP("storage_probe() failed\n");
        release_everything(us);
        return result;
}
```

5. 实验步骤

♦ **在内核中添加 U 盘的设备支持**

1）进入宿主机中 CBT-6818 型光盘内核目录：

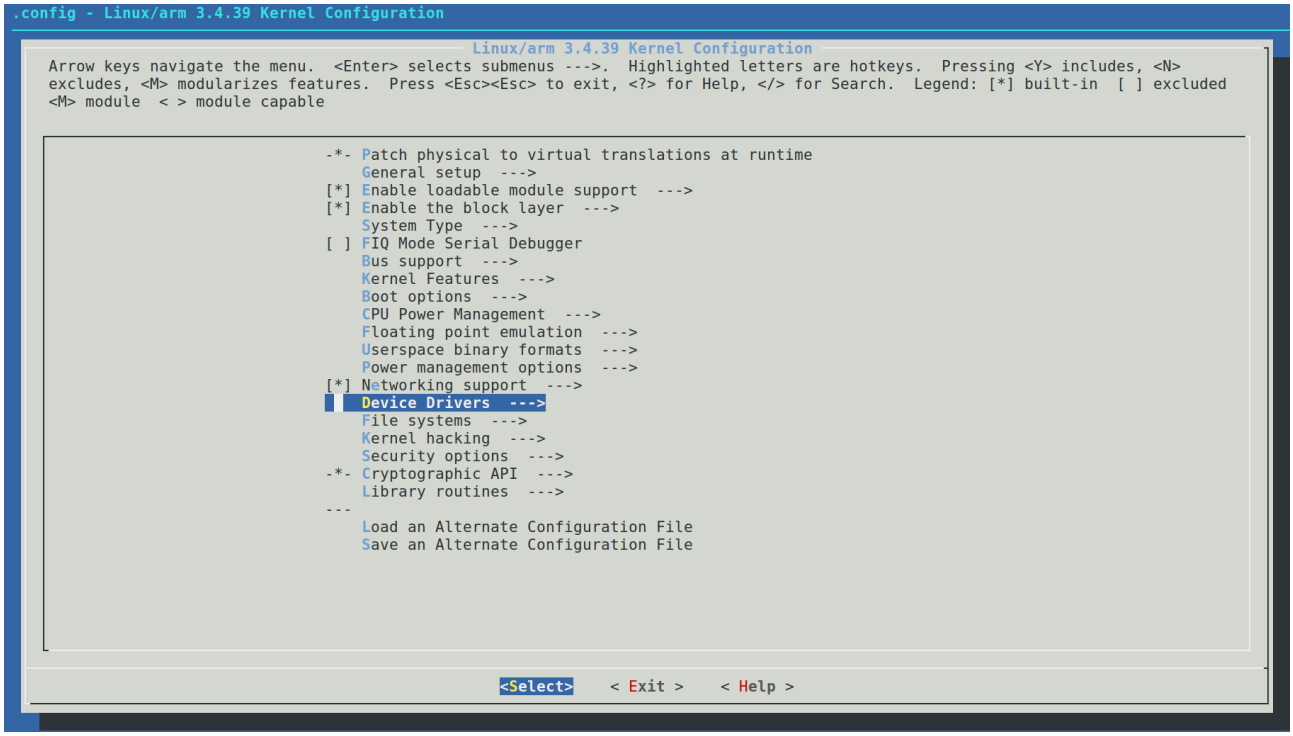
```
cbt@Cyb-Bot:~$ cd /CBT-6818/SRC/linux/kernel
```

2）运行 make menuconfig 命令配置内核对 framebuffer 的相关支持

```
cbt@Cyb-Bot: /CBT-6818/SRC/linux/kernel $
cbt@Cyb-Bot: /CBT-6818/SRC/linux/kernel $ cp 6618_linux_config    .config
cbt@Cyb-Bot: /CBT-6818/SRC/linux/kernel $ make menuconfig
```

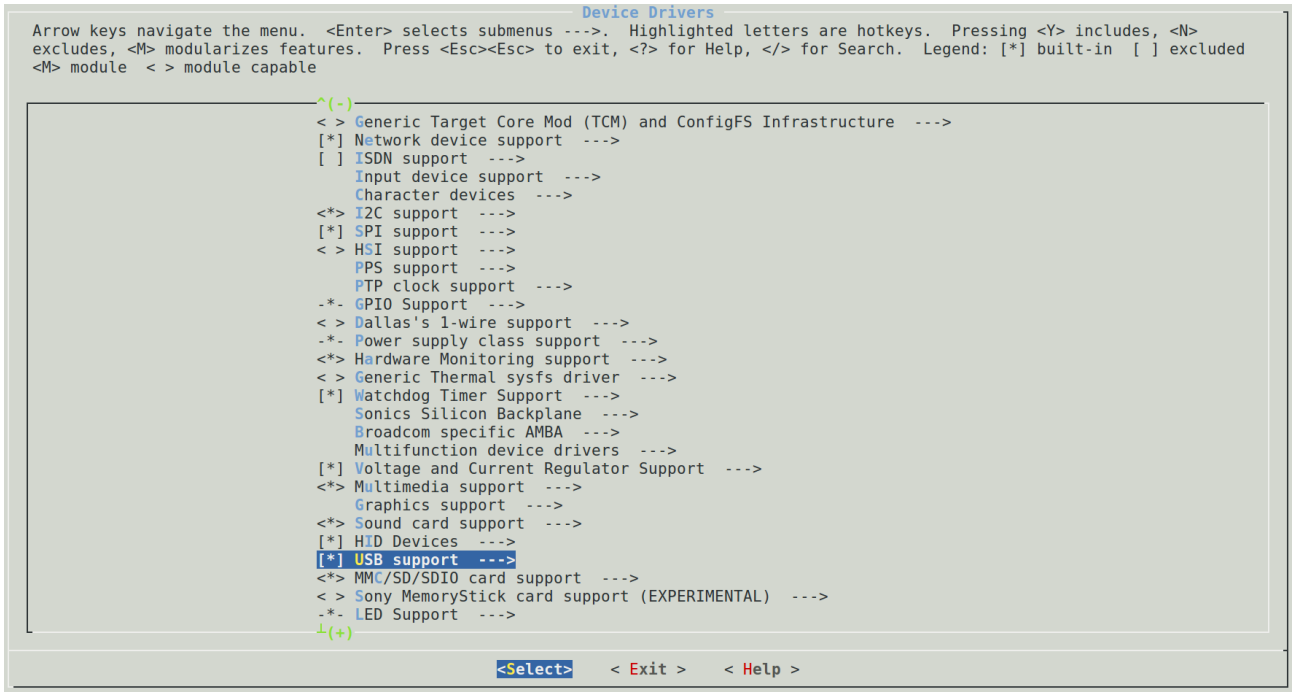
选择 Device Drivers ---->选项，如图

选择 [*] USB support ---> 支持，如图：

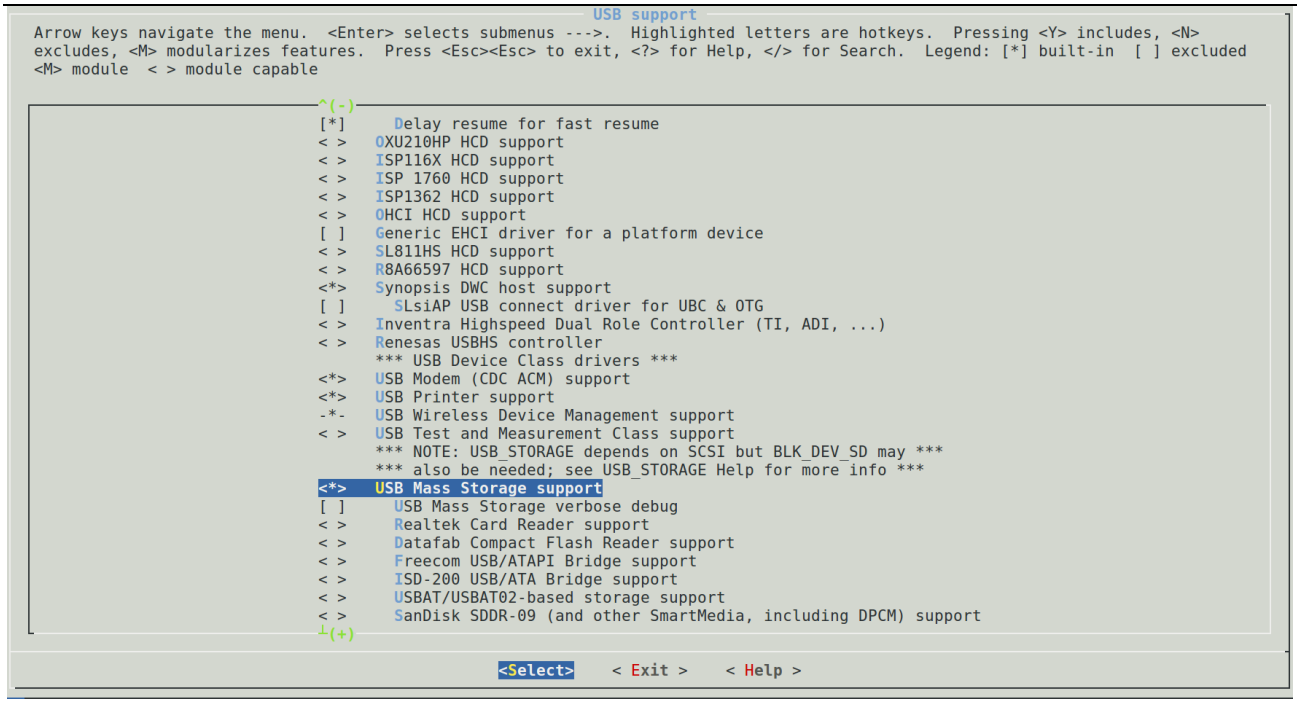


进入 [*] USB support --->

如图所示配置：



选择 <*> USB Mass Storage support 如图：



退出保存设置，如图：

3）重新编译内核,运行 make 命令

```
cbt@Cyb-Bot: /CBT-6818/SRC/linux/kernel $ make
  CHK     include/linux/version.h
  CHK     include/generated/utsrelease.h
make[1]: “include/generated/mach-types.h”是最新的。
  CALL    scripts/checksyscalls.sh
  CHK     include/generated/compile.h
.....
```

最终在 linux 源码目录的 out/release 目录下生成新的 ARM-LINUX 内核镜像文件 boot.img

```
cbt@Cyb-Bot: /CBT-6818/SRC/linux/kernel $ ls ../out/release/boot.img
boot.img
```

```
cbt@Cyb-Bot: /CBT-6818/SRC/linux/kernel $
```

4）按照烧写文档将新生成的内核镜像文件 boot.img 烧写到 ARM 设备中，这里不在赘述。

备注：以上在内核中添加对 U 盘存储设备的支持的步骤，在系统设备出厂自带内核中已经默认添加进入了，用户可以省略以上步骤。以上步骤在于重现系统的构造。

♦ 使用挂载 U 盘测试

1）启动实验系统，连好网线、串口线。系统启动后插入 U 盘（FAT32 格式），如果先插入 U 盘再启动系统，U 盘会被自动挂载到系统/udisk 目录下，终端显示如下信息：

```
[    28.468000] usb 1-1.2: new high-speed USB device number 3 using nxp-ehci
[    28.685000] usb 1-1.2: New USB device found, idVendor=13fe, idProduct=5500
[    28.687000] usb 1-1.2: New USB device strings: Mfr=1, Product=2, SerialNumber=3
[    28.694000] usb 1-1.2: Product: USB DISK 3.0
[    28.698000] usb 1-1.2: Manufacturer:
[    28.703000] usb 1-1.2: SerialNumber: 0708668D9AC44F21
[    28.708000] scsi0 : usb-storage 1-1.2:1.0
[    30.564000] scsi 0:0:0:0: Direct-Access                USB DISK 3.0        PMAP PQ: 0 ANSI: 6
[    30.567000] sd 0:0:0:0: Attached scsi generic sg0 type 0
[    30.567000] sd 0:0:0:0: [sda] 30326784 512-byte logical blocks: (15.5 GB/14.4 GiB)
[    30.568000] sd 0:0:0:0: [sda] Write Protect is off
[    30.569000] sd 0:0:0:0: [sda] No Caching mode page present
[    30.569000] sd 0:0:0:0: [sda] Assuming drive cache: write through
[    30.572000] sd 0:0:0:0: [sda] No Caching mode page present
[    30.572000] sd 0:0:0:0: [sda] Assuming drive cache: write through
[    30.573000]  sda: s
[    30.612000] sd 0:0:0:0: [sda] No Caching mode page present
[    30.615000] sd 0:0:0:0: [sda] Assuming drive cache: write through
[    30.621000] sd 0:0:0:0: [sda] Attached SCSI removable disk
[root@CBT-6818 /$
```

上述终端信息，可以知道系统识别 U 盘为/dev/sda 设备。

2）挂载 U 盘

```
[root@CBT-6818 /$ mkdir /usb/
[root@CBT-6818 /$ mount /dev/sda /usb/
```

3）查看 U 盘内容

```
[root@CBT-6818 /$ ls /usb/
CBT-6818                                Source.Insight.v3.50
demo        UltraEdit_11.00a+_SC(1).exe
testlcd.zip
[root@CBT-6818 /$
```

4）卸载 U 盘

```
[root@CBT-6818 /$ umount    /usb/
[root@CBT-6818 /$
```

注意：卸载命令不要在挂载的目标目录中执行，否则会卸载失败。

第六章. 综合实训案例

本章主要在配套 RFID、ZigBee、传感器等的基础上，设计了几个基于 QT 的综合实例。关于 RFID、ZigBee、传感器等的学习，可查阅《全功能物联网教学科研平台(ICS-IOT-CEP)实验指导书（第一册）》

通过本章的学习，用户可以掌握 Linux 系统下用户界面设计的基本方法，掌握用户界面是如何同 RFID、ZigBee、传感器等模块联动的。用户通过动手实现本章的综合实例，可以对物联网在家居等领域的应用有个明确的认识，加深物联网的学习，并可以在此基础上进行二次开发，实现更全面地功能。

实验一. 基于 RFID 无线射频的电子钱包实训案例

1. 实验目的

- 学习在 LINUX 系统下设备驱动设计的基本原理和方法。
- 掌握使用模块方式进行驱动开发调试和使用测试的过程。

2. 实验环境

- 硬件：ICS-IOT-CEP 实验平台，PC 机 Pentium 500 以上，硬盘 40G 以上，内存大于 256M
- 软件：Vmware Workstation＋ubuntu＋MiniCom/超级终端 ＋ARM-LINUX 交叉编译开发环境
- 实验目录：\Components\Cortex-A53\SRC\items\rfid

3. 实验内容

- 学习 linux 系统下设备接口驱动的编写方法，编写简单的字符设备驱动程序和测试程序。
- 实现驱动程序的编写和使用。

4. 实验原理

4.1 RFID 简介

♦ **什么是 RFID**

RFID 是 Radio Frequency Identification 的缩写，即射频识别。常称为感应式电子晶片或近接卡、感应卡、非接触卡、电子标签、电子条码，等等。

RFID 的应用非常广泛，目前典型应用有动物晶片、汽车晶片防盗器、门禁管制、停车场管制、生产线自动化、物料管理。RFID 标签有两种：有源标签和无源标签。

♦ **什么是电子标签**

电子标签即为 RFID 有的称射频标签、射频识别。它是一种非接触式的自动识别技术，通过射频信号识别目标对象并获取相关数据，识别工作无须人工干预，作为条形码的无线版本，RFID 技术具有条形码所不具备的防水、防磁、耐高温、使用寿命长、读取距离大、标签上数据可以加密、存储数据容量更大、存储信息更改自如等优点。

♦ **什么是 RFID 技术**

RFID 射频识别是一种非接触式的自动识别技术，它通过射频信号自动识别目标对象并获取相关数据，识别工作无须人工干预，可工作于各种恶劣环境。RFID 技术可识别高速运动物体并可同时识别多个标签，操作快捷方便。

短距离射频产品不怕油渍、灰尘污染等恶劣的环境，可在这样的环境中替代条码，例如用在工厂的流水线上跟踪物体。长距射频产品多用于交通上，识别距离可达几十米，如自动收费或识别车辆身份等。

♦ **RFID 无线识别电子标签基础介绍**

无线射频识别技术（Radio Frequency Idenfication，RFID）是一种非接触的自动识别技术，其基本原理是利用射频信号和空间耦合（电感或电磁耦合）或雷达反射的传输特性，实现对被识别物体的自动识别。

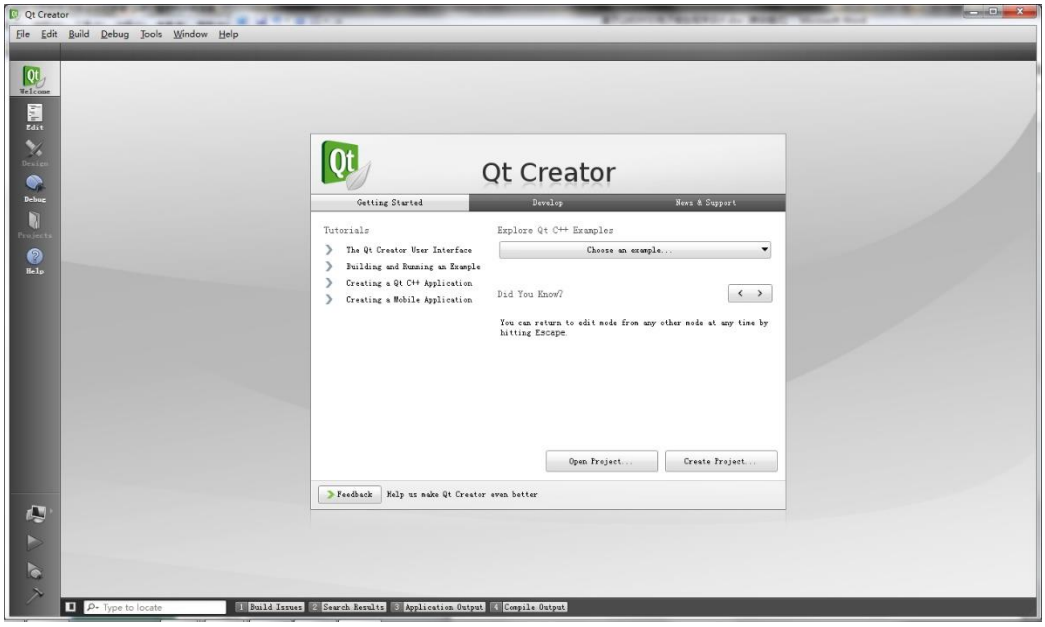
一套完整 RFID 系统由 Reader 与 Transponder 两部份组成 ,其动作原理为由 Reader 发射一特定频率之无限电波能量给 Transponder,用以驱动 Transponder 电路将内部之 ID Code 送出，此时 Reader 便接收此 ID Code。 Transponder 的特殊在于免用电池、免接触、免刷卡故不怕脏污，且晶片密码为世界唯一无法复制，安全性高、长寿命。

RFID 阅读器（读写器）通过天线与 RFID 电子标签进行无线通信，可以实现对标签识别码和内存数据的读出或写入操作。电子标签依据频率的不同可分为低频电子标签、高频电子标签、超高频电子标签和微波电子标签。依据封装形式的不同可分为信用卡标签、线形标签、纸状标签、玻璃管标签、圆形标签及特殊用途的异形标签等。

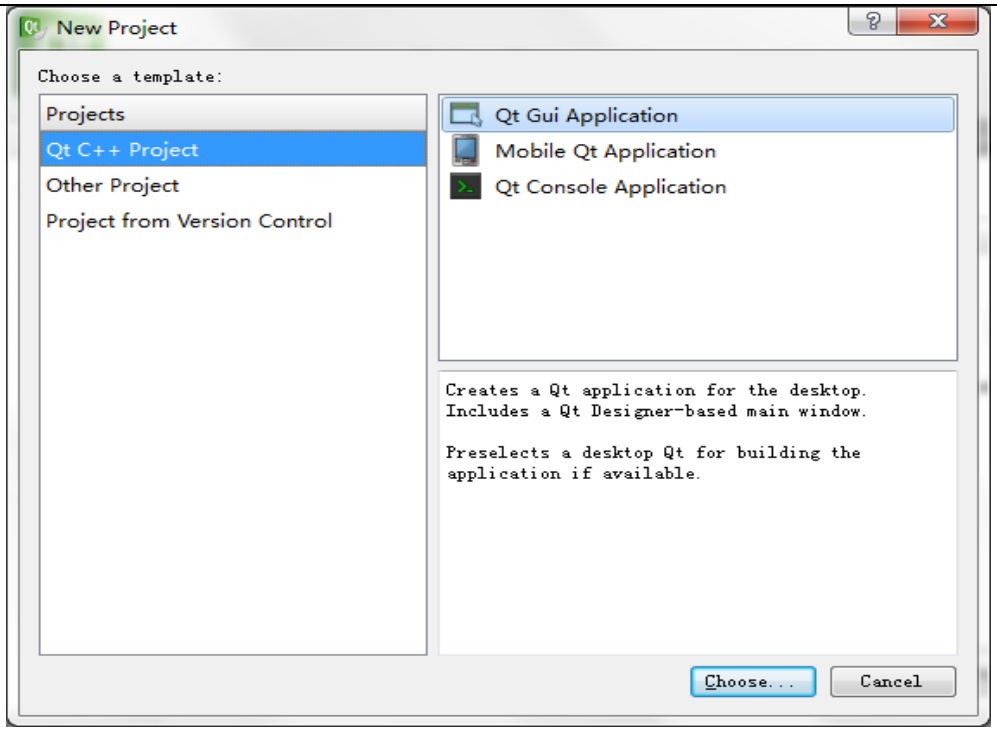
5. 实验步骤

♦ **创建工程文件**

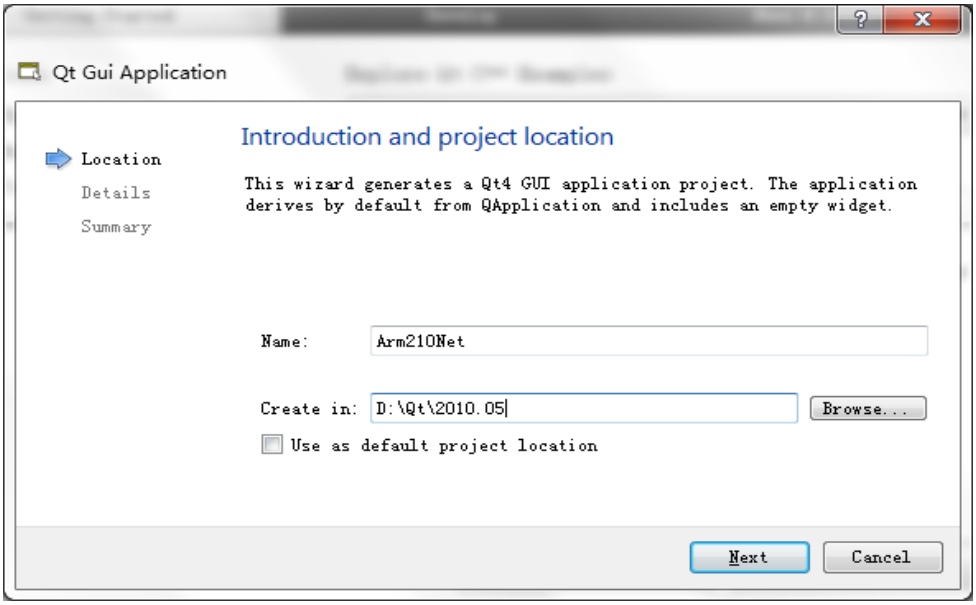
1）打开 Qt Creator 如下：



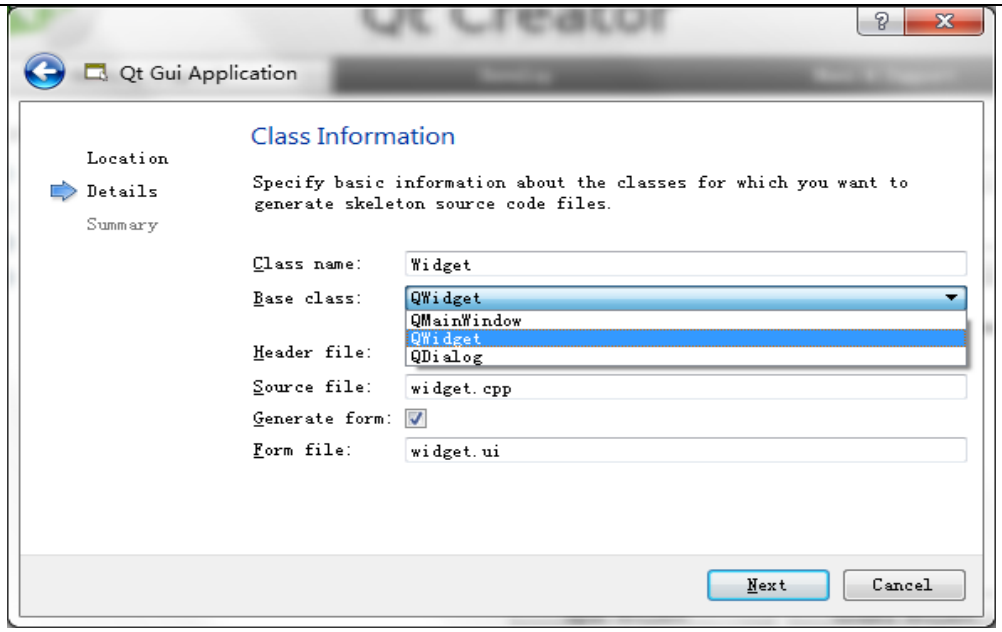
2）点击“Creator Project”按钮，选择“Qt C++ Project”à“Qt Gui Application”点击“Choose”。



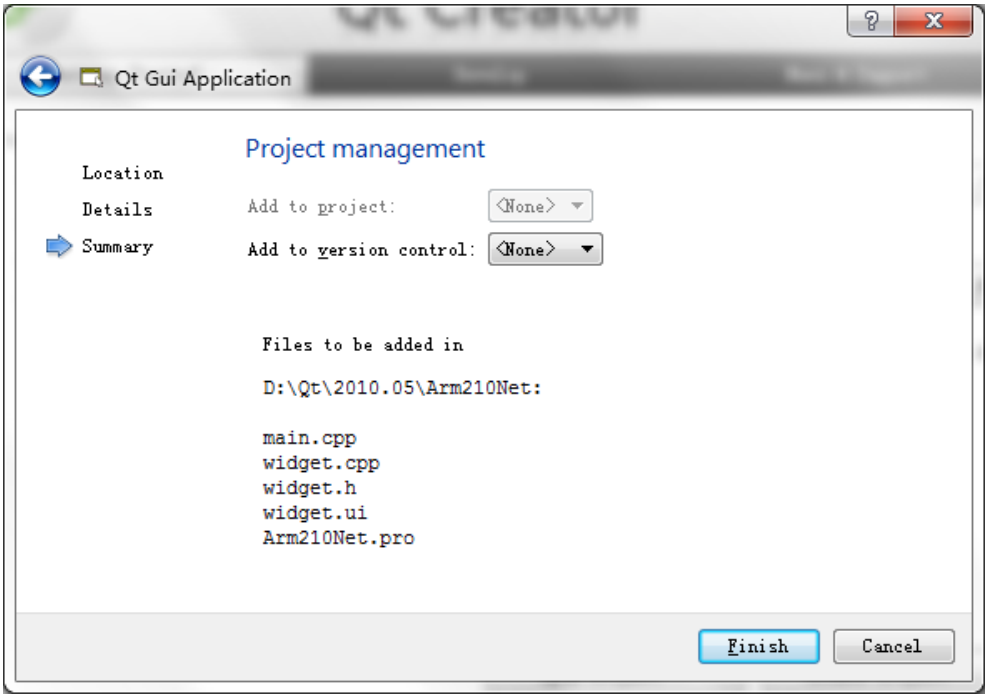
3）写好工程名字，工程的创建路径，点击“Browse”更改路径，（注：路径不能有中文）点击“Next”



4）选好要用的基类“QWidget”，点击“Next”。



5）将文件添加的了新建的工程中，我们选择默认设置，点击“Finish”。



♦ **设计 UI 窗口**

此程序包含两个界面：电子钱包主窗口、电子钱包充值和扣费窗口。将要实现的效果图如下：

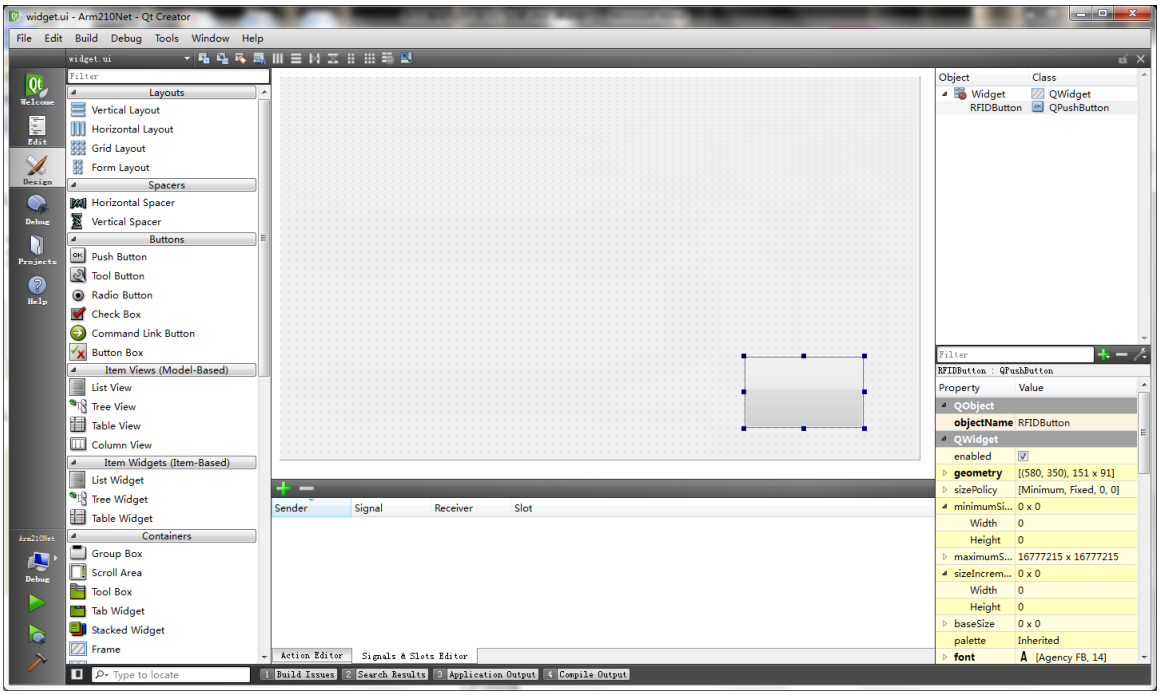
图 5.1 电子钱包主窗口

图 5.2 电子钱包计费窗口

主窗口只有一个 RFID 按键，按下 RFID 按键后便能进入计费窗口，点击返回按键则返回到主窗口接下来开始两个界面的设计：

主窗口设计：

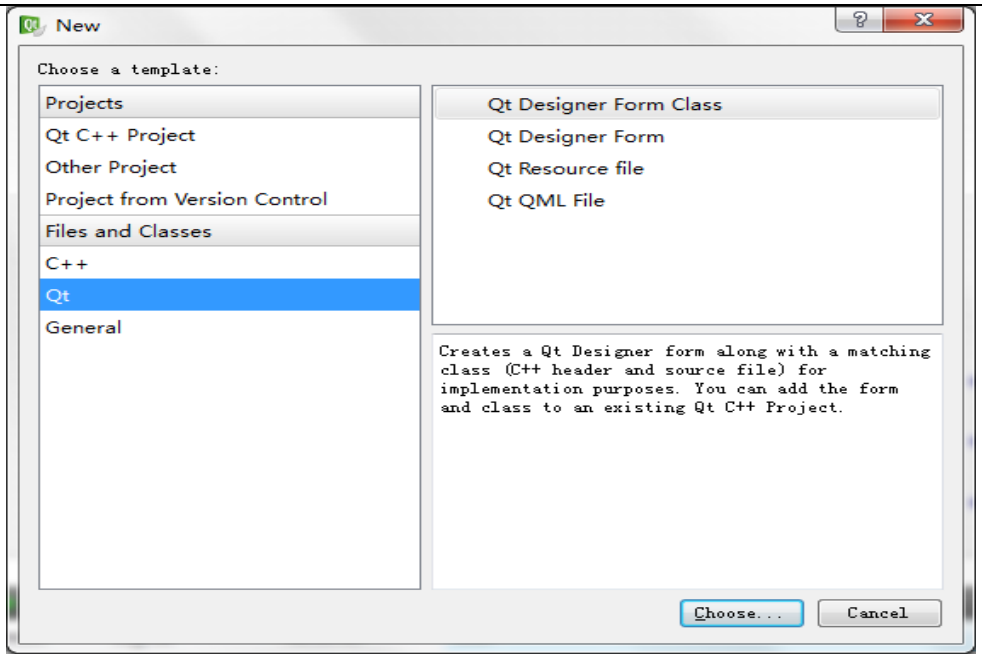
在窗口上添加一个“Push Button”按键（左键选中“Push Button”按住拖拽到主窗口），将右下角的按键信息里把“objectName”后面改为“RFIDButton”。



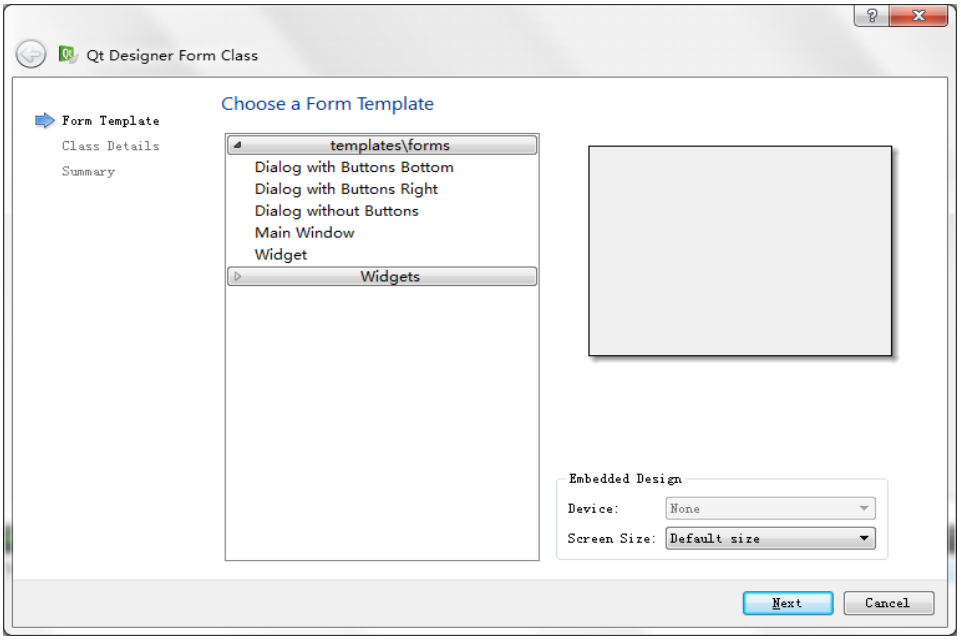
主窗口设计完毕，接下来进行电子钱包计费窗口设计。

电子钱包计费窗口设计：

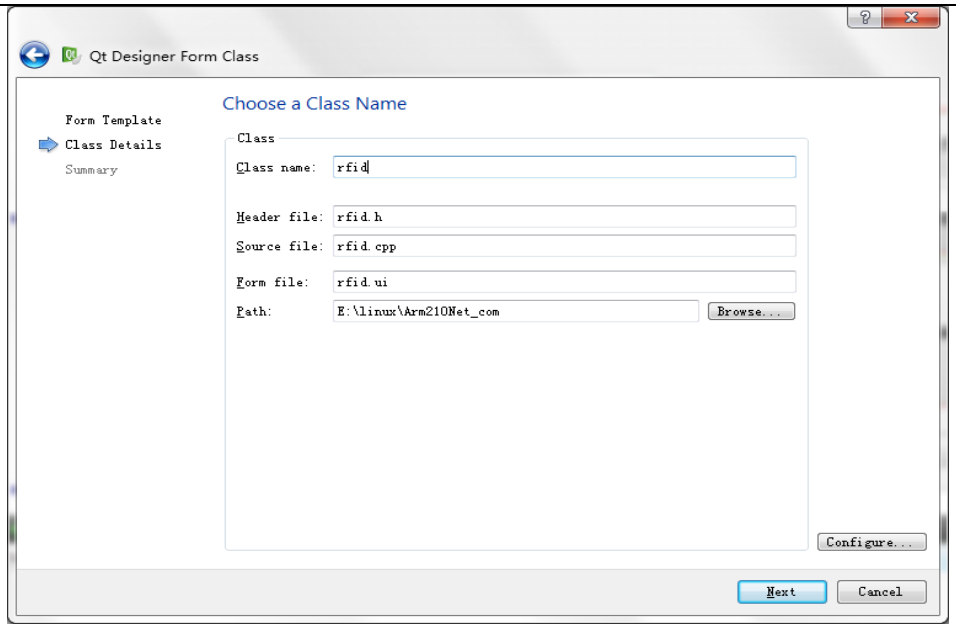
点击“File” -〉“New File or Project”，选中“Qt”右面的“Qt Designer Form Class”点击“Choose”，新建一个界面。



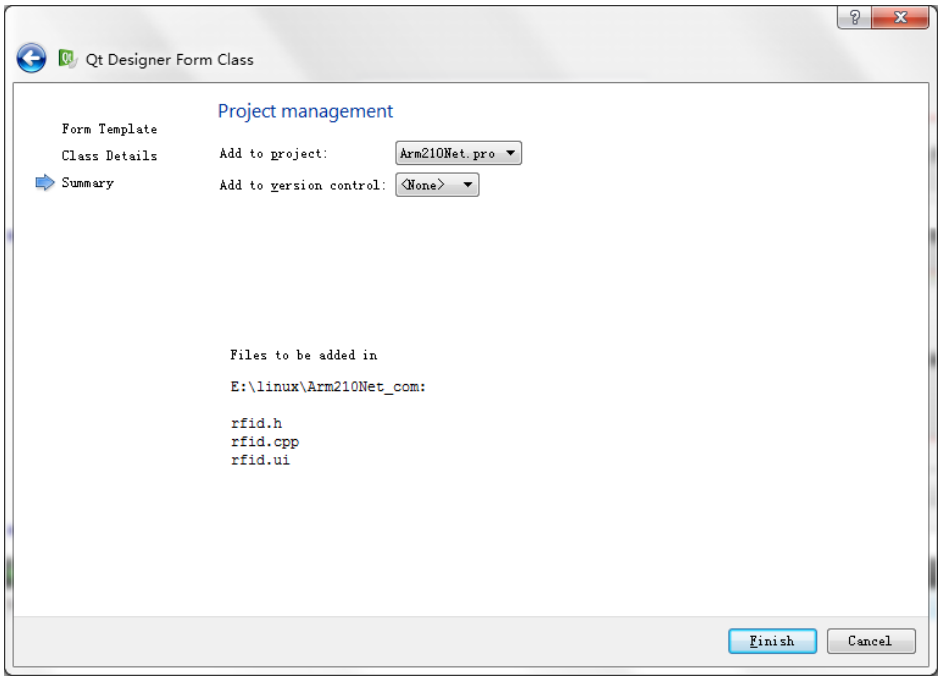
在“Qt Designer Form Class”下，选中“templates\forms”下的“Dialog without Buttons”，点击“Next”。



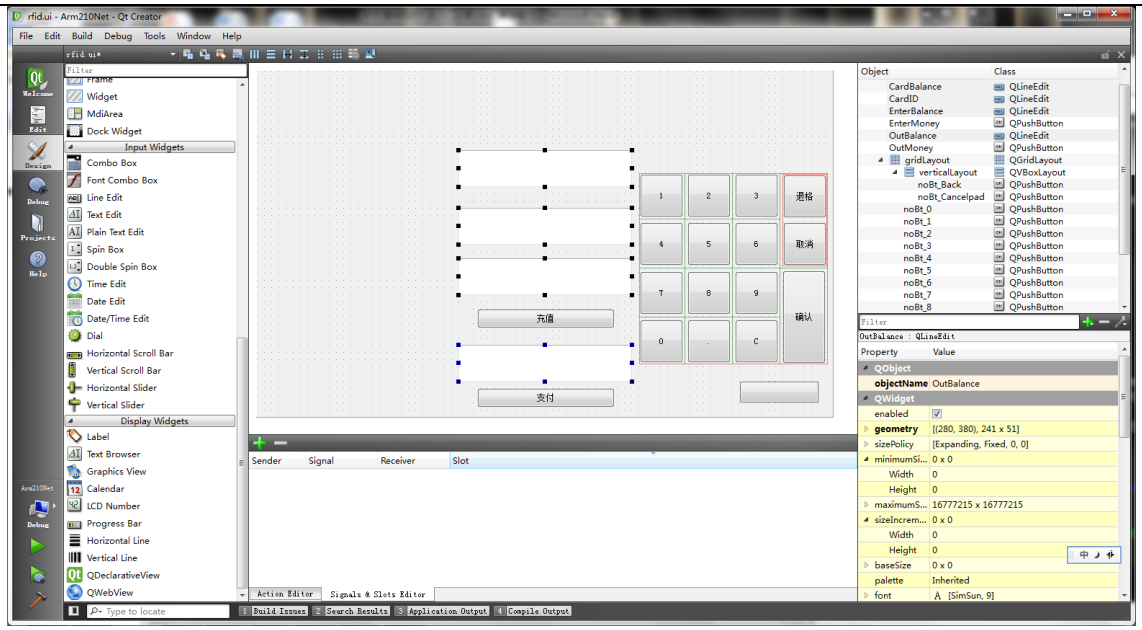
在“Qt Designer Form Class”下，在“Class name”后编写“rfid”，保存路径为自己的工程路径，点击“Next”。



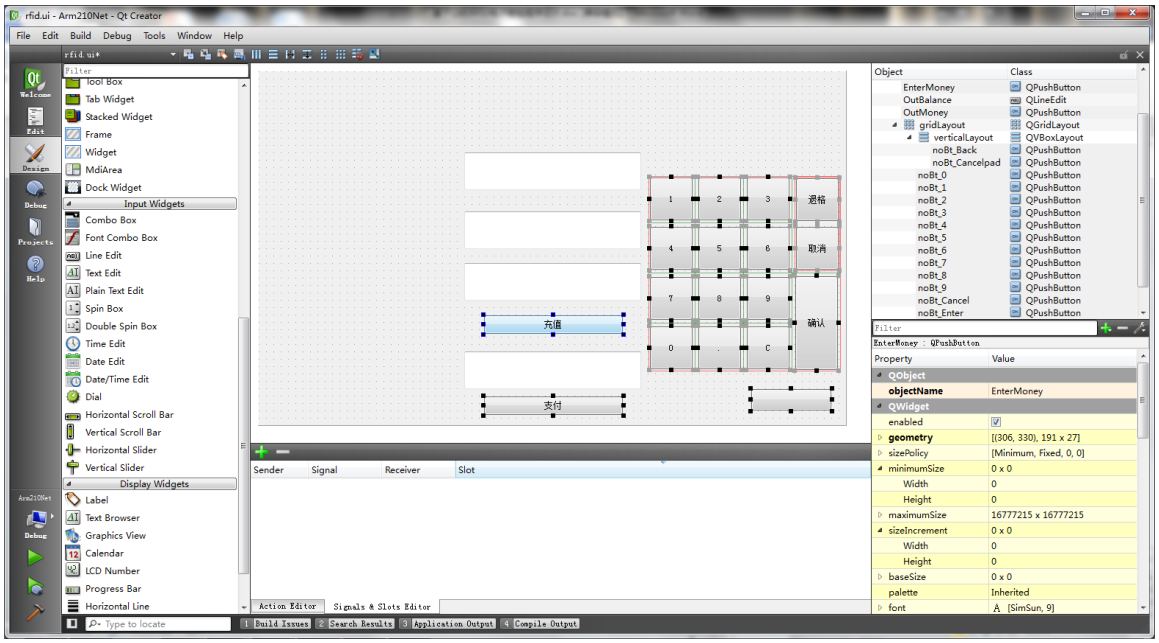
添加到工程里，我们选择默认设置，点击“Finish”。



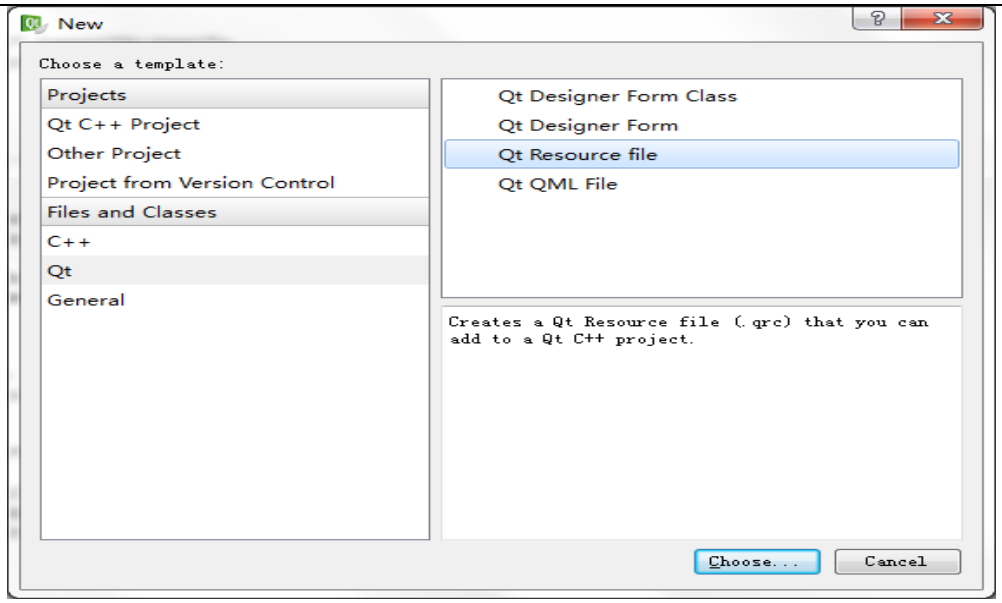
首先先拖拽四个“QLineEdit”，十八个“QPushButton”上下左右按下图依次排好，双击按键按照下图改写按键上的文字，其中右下角的为返回按键，为此按键添加图示不需文字标识，把“QLineEdit”的四个按键按照上到下的顺序将右下角的“objectName”后依次改为“CardID”IC 卡号，“CardBalance”IC 卡余额，“EnterBalance”充值，“OutBalance”扣款。



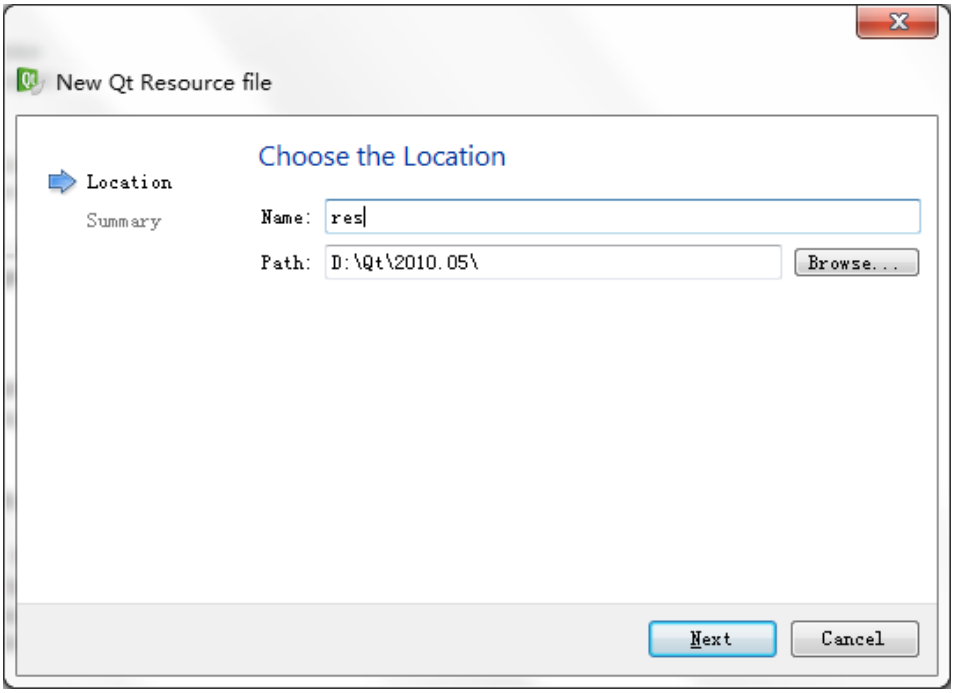
再把“QPushButton”的“objectName”后做出更改，按键“充值”改为“EnterMoney”，按键“支付”改为“OutMoney”，按键“1”改为“noBt_1”，按键“2”改为“noBt_2”，按键“3”改为“noBt_3”，按键“4”改为“noBt_4”，按键“5”改为“noBt_5”，按键“6”改为“noBt_6”，按键“7”改为“noBt_7”，按键“8”改为“noBt_8”，按键“9”改为“noBt_9”，按键“0”改为“noBt_0”，按键“退格”改为“noBt_Back”，按键“取消”改为“noBt_Cancelpad”，按键“确定”改为“noBt_Enter”，按键“.”改为“noBt_dot”，按键“C”改为“noBt_Cancel”，右下角的按键改为“returnButton”。



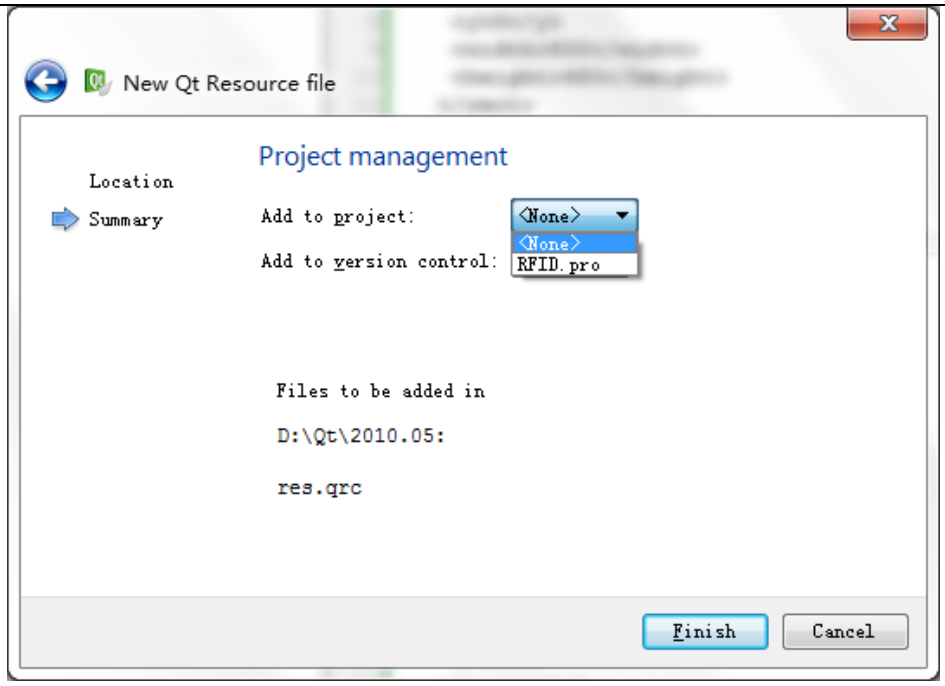
接下来要在工程里添加图片文件，点击“File”à“New File or Project”选择“Qt”右边的“Qt Resource file”点击“Choose”。



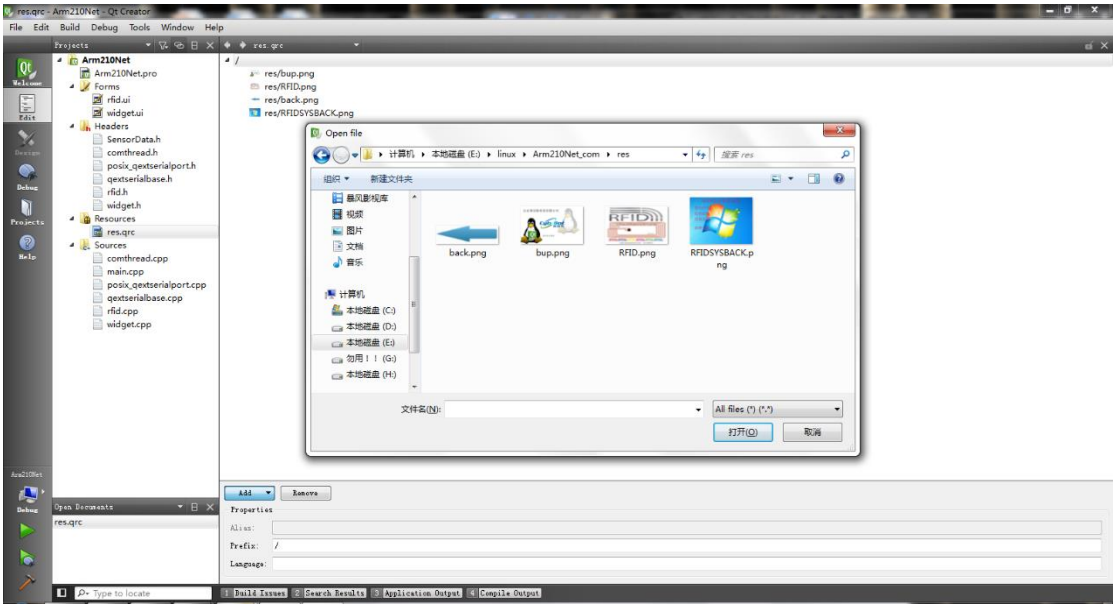
在“Name”后写上要新建的图片文件的名字，路径为你工程的路径，点击“Next”。

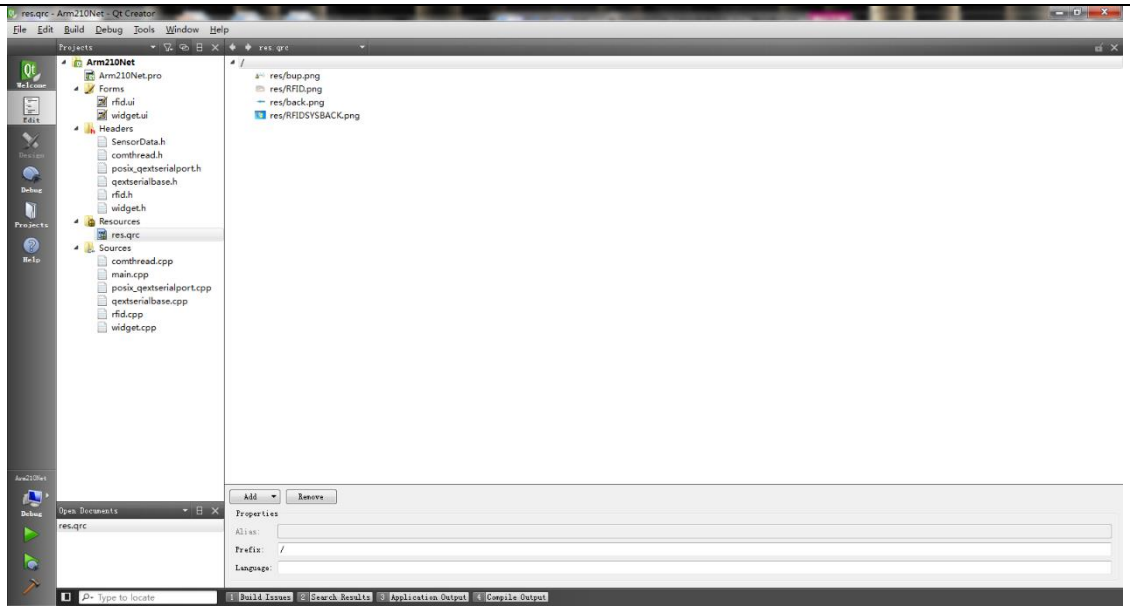


在“Add to project ”后选择你的工程“***.pro”点击“Finish”。



点击下面的“Add”选择“Add prefix”再次点击“Add”选择“Add Files”如下，选择自己要添加的所有图片。





修改程序，添加背景图片：

在 widget.cpp 里添加头文件：

```
#include <QPixmap>
```

在构造函数里添加：

```
this->setWindowFlags(Qt::Window | Qt::FramelessWindowHint);
    //设置背景图片
    QPixmap mypixmap(":/res/bup.png");//添加的图片文件路径
    QPalette palette;
palette.setBrush(backgroundRole(),QBrush(mypixmap));
setPalette(palette);
```

在 rfid.cpp 里添加头文件：

```
#include <QPixmap>
```

在构造函数里添加：

```
SetButtonBackGround(ui->returnButton,QPixmap(":/res/back.png"));//为返回按键添加图片
    /*****************************************************/
      QPixmap mypixmap(":/res/RFIDSYSBACK.png");//为子窗口添加背景图片
      QPalette palette;
palette.setBrush(backgroundRole(),QBrush(mypixmap));
  setPalette(palette);
```

♦ 程序编写

首先介绍一下程序的文件构成，有 main.cpp，rfid.cpp，widget.cpp，comthread.cpp，qextserialbase.cpp， posix_qextserialport.cpp， main.h，rfid.h，widget.h，comthread.h，qextserialbase.h， posix_qextserialport.h， SensorDate.h 这些文件构成的，下面介绍这些文件中重要的功能函数，

在 main.cpp 中添加中文支持：

```
#include <QtGui/QApplication>
    #include <QTextCodec>//支持中文的头文件

    #include "widget.h"
    int main(int argc, char *argv[])
    {
        QApplication a(argc, argv);
        //中文支持
        QTextCodec::setCodecForLocale(QTextCodec::codecForName("UTF-8"));
        QTextCodec::setCodecForTr(QTextCodec::codecForName("UTF-8"));//支持中文 Tr
        QTextCodec::setCodecForCStrings(QTextCodec::codecForName("UTF-8"));//支持中文文件名
        QFont font;
        font.setPointSize(9);//字号设置为 9
        font.setFamily(("WenQuanYi Micro Hei Mono"));
        font.setBold(false);
        a.setFont(font);
        Widget w;
        w.show();
    return a.exec();
}
```

在 UI 窗口里添加了按键，要在程序里定义每个按键的执行动作的槽函数，而且要把按键的信号和对应的槽函数连接起来。

在 rfid.h 里声明。

```
private slots:                    //定义私有的槽函数，槽函数为子界面按键的
void NumButtonsHide();
    void NumButtonsShowEnter();
    void NumButtonsShowOut();
    void Num0Click();
    void Num1Click();
    void Num2Click();
    void Num3Click();
    void Num4Click();
    void Num5Click();
    void Num6Click();
    void Num7Click();
    void Num8Click();
    void Num9Click();
    void NumdotClick();
    void NumBackClick();
    void NumCancelClick();
    void NumEnterClick();
    void NumEnterMoneyClick();
void NumOutMoneyClick();
```

```
void ReceiveRFData(QByteArray RFData);
void rfidshow();
```

接下来，要在 rfid.cpp 里写已经在头文件里声明的槽函数的执行动作：

```
void RFID::ReceiveRFData(QByteArray RFData)//接收数据显示在 UI 子窗口，
{
    qDebug()<<QString("RFIDRX:%1").arg(RFData.toHex().data());
    QString tempString = RFData.toHex().data();
    qDebug()<<QString("RFIDNUM:%1").arg(tempString.mid(18,8).toInt(0,16));
    //显示卡号
    ui->CardID->setText(tempString.mid(10,8));
    //显示余额
    ui->CardBalance->setText(QString("%1.00").arg(tempString.mid(18,8).toInt(0,16)));
  }
void RFID::NumBackClick()//返回按键的槽函数
{
     numpointer->backspace();
}
void RFID::NumEnterMoneyClick()//充值按键的槽函数
{
     QByteArray tempByte;
     tempByte.insert(0,0xFF);
     bool ok;
     int num = ui->EnterBalance->text().toInt(&ok,10);
     if(ok)
     {
         disconnect(ui->EnterBalance,SIGNAL(selectionChanged()),this,SLOT(NumButtonsShowEnter()));
         disconnect(ui->OutBalance,SIGNAL(selectionChanged()),this,SLOT(NumButtonsShowOut()));
         tempByte.insert(0,num%256);
         num/=256;
         tempByte.insert(0,num%256);
         num/=256;
         tempByte.insert(0,num%256);
         num/=256;
         tempByte.insert(0,num%256);          //CC EE FE 01 01 XX XX XX XX FF 为充值命令，XX 代
表
         tempByte.insert(0,0x01);             //将要充值的金额
         tempByte.insert(0,0x01);
         tempByte.insert(0,0xFE);
         tempByte.insert(0,0xEE);
         tempByte.insert(0,0xCC);
         emit newSendRFData(tempByte);
         ui->EnterMoney->setDisabled(true);
         ui->EnterBalance->clear();
         connect(ui->EnterBalance,SIGNAL(selectionChanged()),this,SLOT(NumButtonsShowEnter()));
         connect(ui->OutBalance,SIGNAL(selectionChanged()),this,SLOT(NumButtonsShowOut()));
```

```
        }
}
void RFID::NumOutMoneyClick()//支付按键的槽函数
{
    QByteArray tempByte;
     tempByte.insert(0,0xFF);
    bool ok;
    int num = ui->OutBalance->text().toInt(&ok,10);
    if(ok)
    {
        disconnect(ui->EnterBalance,SIGNAL(selectionChanged()),this,SLOT(NumButtonsShowEnter()));
        disconnect(ui->OutBalance,SIGNAL(selectionChanged()),this,SLOT(NumButtonsShowOut()));
        tempByte.insert(0,num%256);
        num/=256;
        tempByte.insert(0,num%256);
        num/=256;
        tempByte.insert(0,num%256);
        num/=256;
        tempByte.insert(0,num%256);          //CC EE FE 01 02 XX XX XX XX FF 为支付的命令，XX
代表
        tempByte.insert(0,0x02);             //支付的金额
        tempByte.insert(0,0x01);
        tempByte.insert(0,0xFE);
        tempByte.insert(0,0xEE);
        tempByte.insert(0,0xCC);
        emit newSendRFData(tempByte);
        ui->OutMoney->setDisabled(true);
        ui->OutBalance->clear();
        connect(ui->EnterBalance,SIGNAL(selectionChanged()),this,SLOT(NumButtonsShowEnter()));
        connect(ui->OutBalance,SIGNAL(selectionChanged()),this,SLOT(NumButtonsShowOut()));
    }
}
void RFID::NumEnterClick()//确认按键的槽函数
{
    disconnect(ui->EnterBalance,SIGNAL(selectionChanged()),this,SLOT(NumButtonsShowEnter()));
    disconnect(ui->OutBalance,SIGNAL(selectionChanged()),this,SLOT(NumButtonsShowOut()));
    NumButtonsHide();
    buttonpointer->setDisabled(false);
    NumButtonsHide();
    ui->EnterBalance->clearFocus();
    ui->OutBalance->clearFocus();
     connect(ui->EnterBalance,SIGNAL(selectionChanged()),this,SLOT(NumButtonsShowEnter()));
     connect(ui->OutBalance,SIGNAL(selectionChanged()),this,SLOT(NumButtonsShowOut()));
}
void RFID::NumCancelClick()//C 按键的槽函数
{
```

```
    numpointer->clear();
    disconnect(ui->EnterBalance,SIGNAL(selectionChanged()),this,SLOT(NumButtonsShowEnter()));
    disconnect(ui->OutBalance,SIGNAL(selectionChanged()),this,SLOT(NumButtonsShowOut()));
    NumButtonsHide();
    connect(ui->EnterBalance,SIGNAL(selectionChanged()),this,SLOT(NumButtonsShowEnter()));
    connect(ui->OutBalance,SIGNAL(selectionChanged()),this,SLOT(NumButtonsShowOut()));
    ui->EnterBalance->clearFocus();
    ui->OutBalance->clearFocus();
}
void RFID::NumdotClick()// . 按键的槽函数
{
    if(!numpointer->text().isEmpty())
    {
        if(numpointer->text().indexOf(".")==-1)
        {
            numpointer->insert(".");
        }
    }
}
void RFID::Num0Click()  //0 按键的槽函数 下面 1~9 按键的槽函数很相似，
{
    if(!numpointer->text().isEmpty())
    {
        if(numpointer->text().indexOf(".")==-1)
         {
            //如果输入栏没有小数点存在
            if(numpointer->text().length()>=0)
            {
                //如果第一个数字为 0 就不能出现
                 if(numpointer->text().indexOf("0")!=0)
                     numpointer->insert("0");
            }
         }
        else
        {
             // 如 果 输 入 栏 有 小 数 点 存 在 且 小 数 点 位 数 少 于  2  位
if(numpointer->text().length()-numpointer->text().indexOf(".")<3)
                 numpointer->insert("0");
        }
    }
    else
    {
         if(numpointer->text().length()==0)
         {
             numpointer->insert("0");
         }
```

```
    }
    return;
}
void RFID::Num1Click()
void RFID::Num2Click()
void RFID::Num3Click()
void RFID::Num4Click()
void RFID::Num5Click()
void RFID::Num6Click()
void RFID::Num7Click()
void RFID::Num8Click()
void RFID::Num9Click()
```

在 rfid.cpp 里将信号和槽函数连接：

```
    ui->CardID->setReadOnly(true);                  //设置按键模式，下同
    ui->CardBalance->setReadOnly(true);
    ui->EnterMoney->setDisabled(true);
    ui->OutMoney->setDisabled(true);
    ui->noBt_dot->setDisabled(true);
    ui->CardID->setAutoFillBackground(false);
    ui->CardBalance->setAutoFillBackground(false);
    ui->EnterBalance->setAutoFillBackground(false);
    ui->OutBalance->setAutoFillBackground(false);
    SetButtonBackGround(ui->returnButton,QPixmap(":/res/back.png"));      //为返回按键加图示
    /*****************************************************/
     QPixmap mypixmap(":/res/RFIDSYSBACK.png");      //为子窗口添加背景图片
     QPalette palette;
     palette.setBrush(backgroundRole(),QBrush(mypixmap));
     setPalette(palette);
    /*****************************************by cxue*/
    NumButtonsHide();
    ui->EnterBalance->setAlignment(Qt::AlignRight);
    ui->CardID->setAlignment(Qt::AlignRight);
    ui->CardBalance->setAlignment(Qt::AlignRight);
    ui->OutBalance->setAlignment(Qt::AlignRight);
    connect(ui->EnterBalance,SIGNAL(selectionChanged()),this,SLOT(NumButtonsShowEnter()));
    connect(ui->OutBalance,SIGNAL(selectionChanged()),this,SLOT(NumButtonsShowOut()));
    connect(ui->noBt_0,SIGNAL(clicked()),this,SLOT(Num0Click()));
    connect(ui->noBt_1,SIGNAL(clicked()),this,SLOT(Num1Click()));
    connect(ui->noBt_2,SIGNAL(clicked()),this,SLOT(Num2Click()));          //将子窗口上的按键
    connect(ui->noBt_3,SIGNAL(clicked()),this,SLOT(Num3Click()));          //与对应按键的槽函
    connect(ui->noBt_4,SIGNAL(clicked()),this,SLOT(Num4Click()));          //数关联。
    connect(ui->noBt_5,SIGNAL(clicked()),this,SLOT(Num5Click()));
    connect(ui->noBt_6,SIGNAL(clicked()),this,SLOT(Num6Click()));
    connect(ui->noBt_7,SIGNAL(clicked()),this,SLOT(Num7Click()));
    connect(ui->noBt_8,SIGNAL(clicked()),this,SLOT(Num8Click()));
```

```
    connect(ui->noBt_9,SIGNAL(clicked()),this,SLOT(Num9Click()));
    connect(ui->noBt_dot,SIGNAL(clicked()),this,SLOT(NumdotClick()));
    connect(ui->noBt_Back,SIGNAL(clicked()),this,SLOT(NumBackClick()));
    connect(ui->noBt_Cancelpad,SIGNAL(clicked()),this,SLOT(NumCancelClick()));
    connect(ui->noBt_Enter,SIGNAL(clicked()),this,SLOT(NumEnterClick()));
    connect(ui->EnterMoney,SIGNAL(clicked()),this,SLOT(NumEnterMoneyClick()));
    connect(ui->OutMoney,SIGNAL(clicked()),this,SLOT(NumOutMoneyClick()));
connect(ui->returnButton,SIGNAL(clicked()),this,SLOT(hide()));
```

因为此程序要在 Cortex-A53 终端上运行，所以要通过串口和 RFID 模块进行数据交互，这里使用的是 Cortex-A53 的 com1 口，串口配置如下：

在 widget.h 里：

```
ComThread *comport_1;   //选定 com1
Posix_QextSerialPort *myCom_1;
```

在 comthread.cpp 里：

```
ComThread::ComThread(Posix_QextSerialPort *com,int portno,QObject *parent=0):QThread(parent)
{

    //得到串口指针
    threadcom = com;
    //初始化读取定时器计时间隔
    timerdly = TIMER_INTERVAL;
    //初始化连续发送计时器计时间隔
    obotimerdly = OBO_TIMER_INTERVAL;
    portnum = portno;
    //设置读取计时器
    //by cxue
    entryNum = 0;
    timer = new QTimer(this);
  //计时器溢出执行 ReceiveRFIDData()
    connect(timer, SIGNAL(timeout()), this, SLOT(ReceiveRFIDData()));

    timer->start(timerdly);
}
void ComThread::ReceiveRFIDData()     //接收串口返回的数据
{
    QByteArray temp = threadcom->readAll();
    if(!temp.isEmpty())
    {
        sensormessage.clear();
        sensormessage.append(temp);
        qDebug()<<QString("RFID:%1").arg(sensormessage.toHex().data());
        temp.clear();
        if((quint8)sensormessage[0]!= 0xee && (quint8)sensormessage[1]!= 0xcc)
```

```
            {
                sensormessage.clear();
            }
            if(sensormessage.length()>COMDATAMAXLENGTH)
            {
                sensormessage.clear();
            }
            if(sensormessage.length()==COMRFIDDATAMAXLENGTH && (quint8)sensormessage[0] ==
0xee && (quint8)sensormessage[1] == 0xcc && (quint8)sensormessage[COMRFIDDATAMAXLENGTH-1]
== 0xff)
            {
                qDebug()<<QString("RFID:%1").arg(sensormessage.toHex().data());
                emit newcomRFData(sensormessage);
                sensormessage.clear();
            }
        }
        //协议解析并发送信息到相应的端口
        return;//收取串口数据
}
//打开需要的串口设备在线成当中监听串口消息
void Widget::comportinit(ComThread *&comport, Posix_QextSerialPort *&myCom ,QString devname,int
comNO)
{
        myCom = new Posix_QextSerialPort(devname, QextSerialBase::Polling);
        //设置波特率 115200
        myCom->setBaudRate((BaudRateType)19);
        //设置数据位 8
        myCom->setDataBits((DataBitsType)3);
        //设置校验 0
        myCom->setParity((ParityType)0);
        //设置停止位 1
        myCom->setStopBits((StopBitsType)0);
        //设置流控制
        myCom->setFlowControl(FLOW_OFF);
        //设置延时
        myCom->setTimeout(TIME_OUT);
        comport = new ComThread(myCom,comNO,this);

connect(comport,SIGNAL(newcomRFData(QByteArray)),rfidform,SLOT(ReceiveRFData(QByteArray)));
            connect(rfidform,SIGNAL(newSendRFData(QByteArray)),comport,SLOT(SendData(QByteArray)));
}
```

要了解计时器请参考 comthread.h：

```
//延时，TIME_OUT 是串口读写的延时
#define TIME_OUT 10
        //读取定时器计时间隔,200ms，读取定时器是我们读取串口缓存的延时
```

```
    #define TIMER_INTERVAL 200
    #define COMDATAMAXLENGTH 26
    #define COMRFIDDATAMAXLENGTH 14
    #define RFIDID 0x01
//连续发送定时器计时间隔,200ms
#define OBO_TIMER_INTERVAL 200
```

卡号和余额通过 rfid.cpp 里：

```
void RFID::ReceiveRFData(QByteArray RFData)
{
    qDebug()<<QString("RFIDRX:%1").arg(RFData.toHex().data());
    QString tempString = RFData.toHex().data();

    qDebug()<<QString("RFIDNUM:%1").arg(tempString.mid(18,8).toInt(0,16));
    //显示余额
ui->CardBalance->setText(QString("%1.00").arg(tempString.mid(18,8).toInt(0,16)));
//显示卡号
    ui->CardID->setText(tempString.mid(10,8));
}
```

♦ 第三方插件库支持

把程序下载到 Cortex-A53 终端运行，还要有第三方库来支持，程序里用到了一些头文件也需要第三方库来支持，在工程文件 Arm210Net.pro 里添加：

```
#头文件的支持
INCLUDEPATH += /usr/local/qwt-6.0.1-arm/include
    INCLUDEPATH += /usr/local/wwWidgets-arm/include
    #pc 版本 linux 下需要的库
    #LIBS += -L"/usr/local/qwt-6.0.1/lib/" -lqwt
    #LIBS += -L"/usr/local/wwWidgets/lib/" -lwwwidgets4
    #arm 版本下需要的库
LIBS += -L"/usr/local/qwt-6.0.1-arm/lib/" -lqwt
LIBS += -L"/usr/local/wwWidgets-arm/lib/" -lwwwidgets4
```

有四个第三方插件库： qwt-6.0.1.tar.gz，qwt-6.0.1-arm.tar.gz，wwWidgets.tar.gz，wwWidgets-arm.tar.gz

插件库使用说明：

- qwt-6.0.1.tar.gz 是 PC 版本的 qwt 库和相关头文件，在使用的时候将其解压到/usr/local/目录下。
- qwt-6.0.1-arm.tar.gz 是 arm 版本的 qwt 库和相关头文件，在使用的时候将其解压到 PC 和目标 A53 平台的/usr/local/目录下。
- wwWidgets.tar.gz 是 PC 版本的 wwWidgets 库和相关头文件，在使用的时候将其解压到/usr/local/目录下。

- wwWidgets-arm.tar.gz 是 arm 版本的 wwWidgets 库和相关头文件，在使用的时候将其解压到 PC 和目标 A53 平台的/usr/local/目录下。
- 特别说明：
- arm 版本使用的交叉编译器版本为 gcc 4.5.1。
- 使用的 qt 版本为 qtEmbeded4.7.0。

♦ 编译与测试

最后把工程文件夹放到 Linux 下编译，还需要在 linux 下配置环境变量，编辑 /root/.bashrc 文件，在文件内容后添加如下：

```
export PATH=/PATH=/usr/local/Trolltech/Qt-4.7.0/bin:$PATH
export PATH=/usr/local/Trolltech/QtEmbedded-4.7.0-arm/bin:$PATH
export PATH=/usr/local/qwt-6.0.1/lib:$PATH
export PATH=/usr/local/wwWidgets/lib:$PATH
export PATH=/usr/local/qwt-6.0.1-arm/lib:$PATH
export PATH=/usr/local/wwWidgets-arm/lib:$PATH
export PATH=/opt/Cyb-Bot/toolschain/4.5.1/bin:$PATH
```

:wq 保存退出，命令行执行 source /root/.bashrc，使添加的环境变量立刻生效。

进入工程文件夹执行 qmake，make 进行编译，编译时，有可能需要 make clean 命令先清除下先前的信息。编译成功后，通过 nfs 挂载方式把生成的 Arm210Net 可执行文件拷贝到 Cortex-A53 终端，在 Cortex-A53 终端挂载宿主机 linux 的文件夹时应重启 NFS 服务 service nfs restart，关掉防火墙 service iptables stop，在宿主机 linux 系统下输入命令如下：

```
cbt@Cyb-Bot: /$ service nfs restart
关闭 NFS mountd：                                          [确定]
关闭 NFS 守护进程：                                        [确定]
关闭 NFS quotas：                                          [确定]
关闭 NFS 服务：                                            [确定]
启动 NFS 服务：                                            [确定]
关掉 NFS 配额：                                            [确定]
启动 NFS 守护进程：                                        [确定]
启动 NFS mountd：                                          [确定]
cbt@Cyb-Bot: /$ service iptables stop
cbt@Cyb-Bot: /$
```

接下来回到 ARM 系统中，将现有的 UI 界面程序 kill 掉（ps 查出当前 UI 界面程序的进程 PID 号，执行 kill PID 号，如果当前没有 UI 界面程序，可省略此步），再执行./Arm210Net –qws ，就可以看到主窗口的效果：

RFID电子钱包应用实例

按下 RFID 按键，则进入子窗口，将 IC 卡放在外部 RFID 模块上就可以对 IC 卡进行充值与扣款，如下图：（RFID 模块需要烧写出厂程序）

实验二. 基于无线传感网的智能家居实训案例

1. 实验目的

- 掌握 ICS-IOT-CEP 实验平台上传感器与处理器通过串口通信的协议
- 掌握 Qt 的信号与槽机制的使用
- 学习 Qt 如何使用第三方串口插件

2. 实验环境

- 硬件：ICS-IOT-CEP 实验平台，PC 机 Pentium 500 以上，硬盘 40G 以上，内存大于 256M
- 软件：Vmware Workstation+ubuntu+xshell 终端+ARM-LINUX 交叉编译开发环境，Qtcreator 开发环境

3. 实验内容

- 分析传感器通过串口发送数据的格式，以及各个位代表的含义
- Qt 对串口类的使用
- 实现基于 ZigBee 无线传感网的智能家居应用

4. 实验原理

4.1 串口通信数据格式

这次实验使用三个 zigbee 节点传感器分别为人体检测传感器，温湿度传感器和烟雾传感器，它们实时采集数据然后通过 zigbee 的组网方式把数据发送的 zigbee 协调器端，然后协调器通过串口发送到处理器，处理器在做进一步的处理。如图：

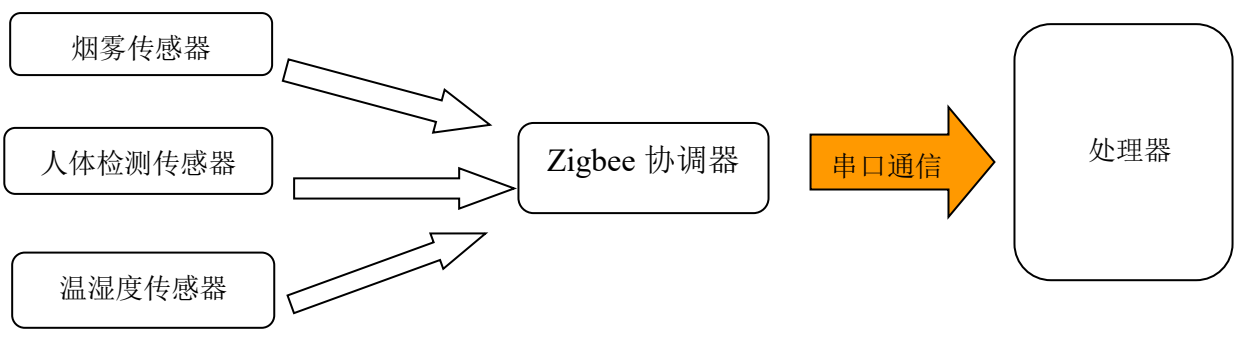


图 1 传感器数据传输方式

ICS-IOT-CEP 实验平台规定传感器数据传输格式如下：

```
u8 DataHeadH;              //包头 0xEE
u8 DataDeadL;              //包头  0xCC
u8 NetID;                  //所属网络标识  00(zigbee) 01(蓝牙)02(WiFi)03(IPv6)04(RFID)
u8 NodeAddress[4];         //节点地址
u8 FamilyAddress[4];       //根节点地址
u8 NodeState;          //节点状态   （00 未发现）  （01 已发现）
u8 NodeChannel;        //蓝牙节点通道
u8 ConnectPort;        //通信端口
u8 SensorType;         //传感器类型编号
u8 SensorID;               //相同类型传感器 ID
u8 SensorCMD;          //节点命令序号
u8 Sensordata1;        //节点数据 1
u8 Sensordata2;        //节点数据 2
u8 Sensordata3;            //节点数据 3
u8 Sensordata4;            //节点数据 4
u8 Sensordata5;        //节点数据 5
u8 Sensordata6;            //节点数据 6
u8 Resv1;              //保留字节 1
u8 Resv2;              //保留字节 2
u8 DataEnd;              //节点包尾 0xFF
```

一帧数据为定长 26 字节。在这三个传感器中我们主要通过判断 SensorType 位来判断传输过来的数据来源于哪个传感器，然后在分析相应的数据节点位的数据，最终得到我们想要的信息。我们通过对 Qt 代码得到的数据截图进行分析：

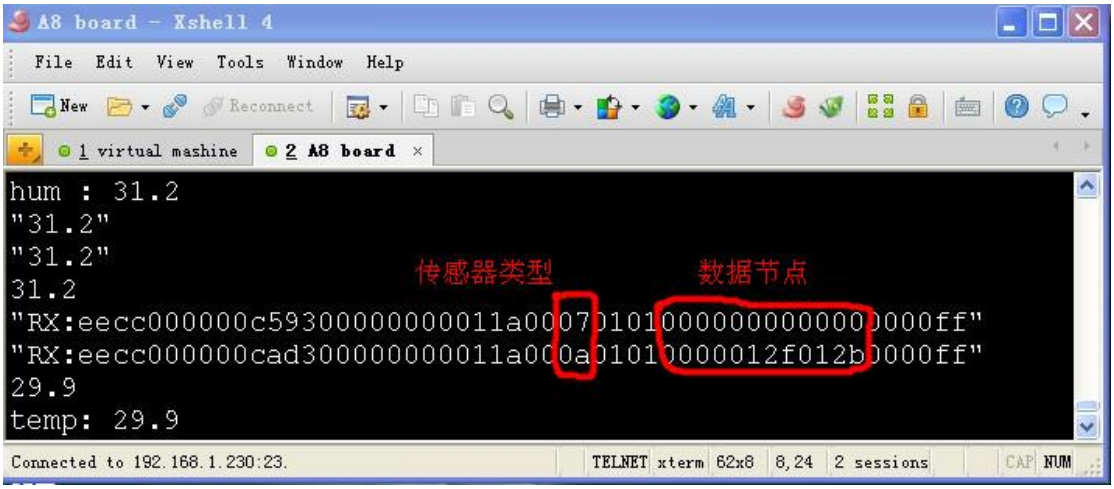


图 2 串口的数据分析

图上的红色圆框 07：代表人体检测节点，0a：代表温湿度传感器节点。人体检测的数据节点代表当前的检测状态（是否检测到人），图中表示没有检测到人。温湿度传感器的数据节点代表当前的温度和湿度百分比。

4.2 Qt 串口类的使用

1）找到串口类相关的文件如图

图 3 串口类的文件

把这几个文件放到工程文件夹下。

2）在工程里添加这个类

在工程文件中找到如下图 header 右击 会弹出四个选项 选择“add Existing Files”。

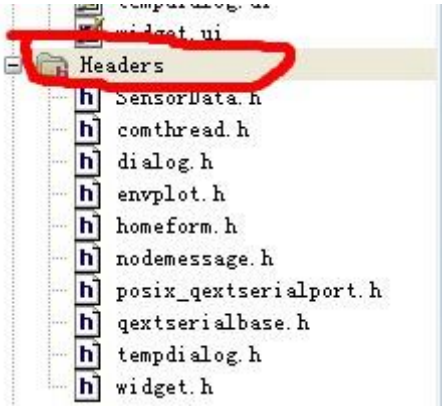


图 4 添加串口类

找到 posix_qextserialport.h 和 qextserialport.h 进行添加。添加 posix_qextserialport.cpp 和 qextserialport.cpp 如上。

3）工程中对串口类的使用

对串口的初始化

```
void Widget::comportInit(comThread *&comport, Posix_QextSerialPort *&myCom,
                                                QString devname,int comNO)
{
    myCom = new Posix_QextSerialPort(devname,
                                     QextSerialBase::Polling); //新建串口类
    if(myCom->open(QIODevice::ReadWrite)){

        //return true;
    }else{
        //return false;
    }
        myCom->setBaudRate((BaudRateType)19);     //设置波特率 115200
```

```
        myCom->setDataBits((DataBitsType)3);     //设置数据位 8
        myCom->setParity((ParityType)0);         //设置校验 0
        myCom->setStopBits((StopBitsType)0);     //设置停止位 1
        myCom->setFlowControl(FLOW_OFF);     //设置流控制
        myCom->setTimeout(10);                   //设置延时
        comport = new comThread(myCom,this);
        connect(comport,SIGNAL(sensorData(QByteArray)),nodeform,
                                    SLOT(reveiceSensordata(QByteArray)));
        comport->start();     //启动线程
}
```

读取串口的数据

```
QByteArray temp = comPort->readAll()。//temp 里存放串口发过来的数据。
```

4.3 信号与槽机制

1）信号与槽的概述

信号和槽机制是 QT 的核心机制，要精通 QT 编程就必须对信号和槽有所了解。信号和槽是一种高级接口，应用于对象之间的通信，它是 QT 的核心特性，也是 QT 区别于其它工具包的重要地方。信号和槽是 QT 自行定义的一种通信机制，它独立于标准的 C/C++语言。

2）信号

信号的声明是在头文件中进行的，QT 的 signals 关键字指出进入了信号声明区，随后即可 声明自己的信号。例如，下面是在 homeform.h 定义了两个信号：

```
signals:
    void tempSender(double temp);
    void humSender(double hum);
```

这两个信号在程序中负责发送温度和湿度的值。另外需要注意的是信号的返回值只能是 void，不要指望能从信号返回什么有用的信息。

3）槽

槽是普通的 C++成员函数，可以被正常调用，它们唯一的特殊性就是很多信号可以与其相关联。当与其关联的信号被发射时，这个槽就会被调用。槽可以有参数，但槽的参数不能有缺省值。既然槽是普通的成员函数，因此与其它的函数一样，它们也有存取权限。槽的存取权限决定了谁能够与其相关联。同普通的 C++成员函数一样，槽函数也分为三种类型，即 public slots、private slots 和 protected slots。

public slots：

在这个区内声明的槽意味着任何对象都可将信号与之相连接。这对于组件编程非常有用，你可以创建彼此互不了解的对象，将它们的信号与槽进行连接以便信息能够正确的传递。

protected slots：

在这个区内声明的槽意味着当前类及其子类可以将信号与之相连接。这适用于那些槽，它们是类实现的一部分，但是其界面接口却面向外部。

private slots：

在这个区内声明的槽意味着只有类自己可以将信号与之相连接。这适用于联系非常紧密的类。

下面是定义在 homeform.cpp 中的两个槽函数。

```
public slots:
    void setTemplabel(double temp);
    void setHumlabel(double hum);
```

4、信号与槽的关联

通过调用 QObject 对象的 connect 函数来将某个对象的信号与另外一个对象的槽函数相关联，这样当发射者发射信号时，接收者的槽函数将被调用。该函数的定义如下：

```
    bool QObject::connect ( const QObject * sender, const char * signal,
            const QObject * receiver, const char * member ) [static]
```

这个函数的作用就是将发射者 sender 对象中的信号 signal 与接收者 receiver 中的 member 槽函数联系起来。当指定信号 signal 时必须使用 QT 的宏 SIGNAL()，当指定槽函数时必须使用宏 SLOT()。下面是 homeform.cpp 中将信号与槽连接起来。

```
    connect(this,SIGNAL(humSender(double)),this,SLOT(setHumlabel(double)));
    connect(this,SIGNAL(tempSender(double)),this,SLOT(setTemplabel(double)));
```

如果发射者与接收者属于同一个对象的话，那么在 connect 调用中接收者参数可以省略。如下图：

```
    connect(this,SIGNAL(humSender(double)),SLOT(setHumlabel(double)));
    connect(this,SIGNAL(tempSender(double)),SLOT(setTemplabel(double)));
```

4.4 关键代码分析

Qt 中给窗口添加背景。

```
void Widget::setWidgetbackground(QWidget *widget,QPixmap image)
{
    QPalette palette;
    palette.setBrush(backgroundRole(),QBrush(image));
    widget->setPalette(palette);
}
```

其中参数 *widget 表示要进行改变的窗口，QPixmap image 表示图片的路径。一般图片的路径我们这样申明：

```
QPixmap myPixmap(":/rcs/mainpage.jpg");
setWidgetbackground(this,myPixmap);
```

当然给窗体添加背景不止有这一种方法。

使 pushbutton 变得漂亮

```
void Widget::setButtonbackground(QPushButton *button,QPixmap picturepath)
{
    button->setFixedSize(picturepath.width(),picturepath.height());
    button->setIcon(QIcon(picturepath)); //设置按钮图标
    button->setFlat(true);   //使按钮变平
    button->setIconSize(QSize(picturepath.width(),picturepath.height()));
    button->setToolTip("");   //光标进入后的提示信息
}
```

这个函数的功能就是能使按键变得适应图片，只要有一个漂亮的图标，就能变成一个漂亮的按键。

Qt 线程的使用。

```
  #ifndef COMTHREAD_H
#define COMTHREAD_H

#include <QThread>
#include "posix_qextserialport.h"

class comThread : public QThread
{
    Q_OBJECT
public:
    explicit comThread(Posix_QextSerialPort *&com,QObject *parent);
    void ReceiveData();
    virtual void run();    //线程执行的任务
private:
    Posix_QextSerialPort *comPort;    //声明一个串口
signals:
    void sensorData(QByteArray);    //发送串口数据
public slots:

};

#endif // COMTHREAD_H
```

这个线程的主要目的是循环检测串口是否接受到数据，如果有数据它将以信号的形式把数据发送到相应的窗口。使用线程类我们主要就是完成 run() 这个函数，这个函数来完成要执行的任务。

```
void comThread::ReceiveData()
{
    QByteArray temp = comPort->readAll();    //读取串口数据
    if(!temp.isEmpty())   //判断是否为空
```

```
    {
        if(temp.length() > COMDATAMAXLENGTH) //判断是否满足命令长度
        {
            temp.clear();
        }
        if(temp.length() == COMDATAMAXLENGTH &&
                    (quint8)temp[0] == 0xee &&
                    (quint8)temp[1] == 0xcc &&
                    (quint8)temp[COMDATAMAXLENGTH-1] == 0xff) //判断数据的包头，和包尾。
        {
            emit sensorData(temp);     //发送数据
        }
    }
}
void comThread::run()
{
    While(1)   //循环检测串口
    {
        ReceiveData();
        msleep(20);
    }
}
```

值得一提的是我们很有必要在 while 循环中添加一个延时，主要是把 cpu 时间片让给其他线程。

5. 实验步骤

1）打开 ubuntu 虚拟机 输入查看网络连接命令（确定 IP 地址）如下图：

```
 cbt@Cyb-Bot: ~$ ifconfig ens33
ens33       Link encap:以太网  硬件地址  00:0c:29:d3:58:c5
            inet 地址:192.168.1.156  广播:192.168.1.255  掩码:255.255.255.0
            inet6 地址: fe80::b2eb:d40f:4890:5b7c/64 Scope:Link
            UP BROADCAST RUNNING MULTICAST  MTU:1500  跃点数:1
            接收数据包:7507 错误:0 丢弃:0 过载:0 帧数:0
            发送数据包:1849 错误:0 丢弃:0 过载:0 载波:0
            碰撞:0 发送队列长度:1000
            接收字节:728832 (728.8 KB)  发送字节:241649 (241.6 KB)

cbt@Cyb-Bot: ~$
```

2）重启虚拟机的 smb、nfs 以及关闭防火墙(如果经常关闭虚拟机要执行此操作)。

```
cbt@Cyb-Bot: ~$ service smb restart
关闭 SMB 服务：                                            [确定]
启动 SMB 服务：                                            [确定]
```

```
cbt@Cyb-Bot: ~$ service nfs restart
关闭 NFS mountd：                                          [确定]
关闭 NFS 守护进程：                                        [确定]
关闭 NFS quotas：                                          [确定]
关闭 NFS 服务：                                            [确定]
启动 NFS 服务：

                                                           [确定]
关掉 NFS 配额：                                            [确定]
启动 NFS 守护进程：                                        [确定]
启动 NFS mountd：                                          [确定]
cbt@Cyb-Bot: ~$ service iptables stop
cbt@Cyb-Bot: ~$
```

3）通过 smb 服务器把我们的 Qt 实例源码 zigbee 目录（在光盘 Cortex-A53\Linux\SRC\item\items\智能家居\zigbee 目录）放到共享目录。

4）搭建 Qt linux 编译环境

在 linux 虚拟机上进行如下操作来查看 qmake 的路径：

```
cbt@Cyb-Bot: /$ which qmake
/usr/local/Trolltech/QtEmbedded-4.7.0-arm/bin/qmake
```

由于我们编译的 Qt 源码要在开发平台上运行使用，所以必须要使用这个路径的 qmake。如果没有这个文件路径我们需要下载平台配套编译好的 QtEmbedded-4.7.0-arm.tar.gz 压缩包，解压之后放到/usr/local/Trolltech/目录下。然后配置环境变量如下：

```
cbt@Cyb-Bot: /$ export PATH=/usr/local/Trolltech/QtEmbedded-4.7.0-arm/bin:$PATH
```

然后再次执行

```
cbt@Cyb-Bot: /$ which qmake
/usr/local/Trolltech/QtEmbedded-4.7.0-arm/bin/qmake
```

出现上面的结果证明我们的 Qt 环境已经搭建完成。

5）编译 Qt 源码程序

这时我们可以进入我们的 Qt 源码里执行如下命令：

```
cbt@Cyb-Bot: /CBT-6818/zigbee$ qmake;make
```

这个命令顺利执行完后会生成可执行文件，在编译前可能需要 make clean 命令清除下信息。

6）挂载我们的虚拟机共享目录（通过 nfs 服务）。

```
[root@Cyb-Bot /$ mount -t nfs -o nolock 192.168.1.12:/CBT-6818 /mnt
[root@Cyb-Bot /$ cd /mnt/
```

以上要根据自己的实际共享目录来操作。

7）运行 Qt 程序

在进入开发平台的/mnt 目录下找到 Qt 源码的位置并进入这个文件夹

```
[root@Cyb-Bot mnt/zigbee $ pwd
/mnt/zigbee
[root@Cyb-Bot zigbee$ ./zigbee -qws
```

♦ 程序运行效果

备注：本实验，需要将全功能物联网教学科研平台上配套的 ZigBee 模块烧写指定程序，方可由 Cortex-A53 平台完成对 ZigBee 无线传感网信息的处理，因此在做实验前，请确保平台上 ZigBee 模块已经烧写有配套的程序。

ZigBee 部分配套程序请读者参加 ZigBee 部分实验指导书《无线传感网络演示实验》章节，此处不再赘述。

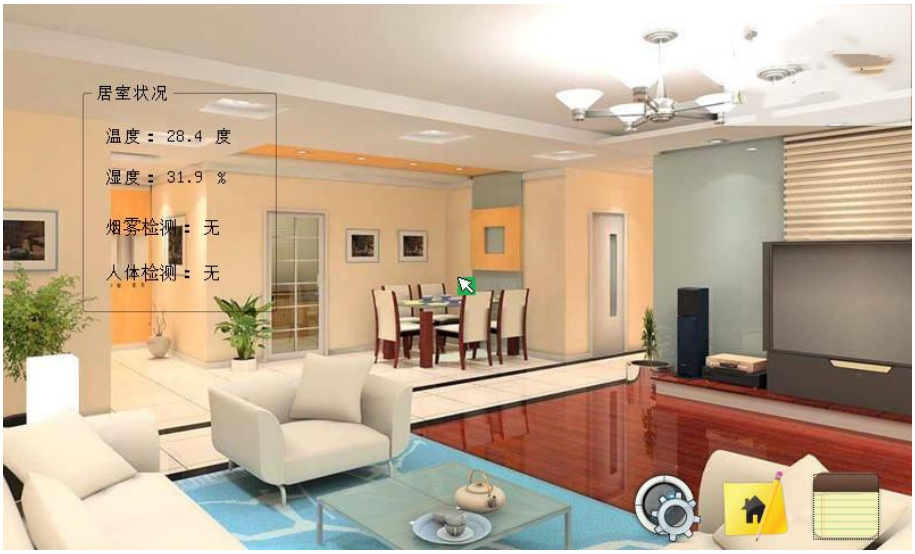


图 5 程序运行效果

图 6 传感器报警设置

图 7 传感器的布局

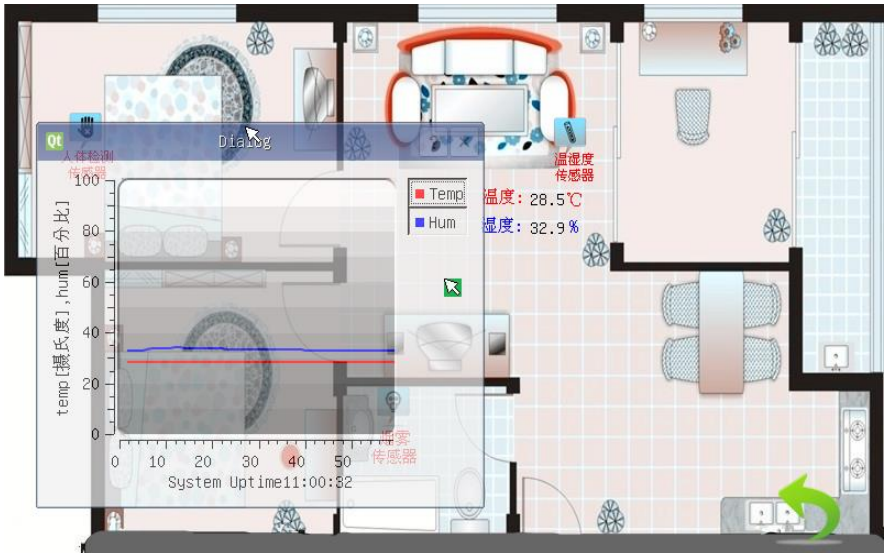


图 8 温湿度的图表显示

实验三. 基于无线传感网的智慧农业实训案例

1. 实验目的

- 掌握 ICS-IOT-CEP 实验平台上传感器与处理器通过串口通信的协议
- 掌握 Qt 的信号与槽机制的使用
- 熟悉 zigbee 的组网通信
- 学习 Qt 如何使用第三方串口插件

2. 实验环境

- 硬件：ICS-IOT-CEP 实验平台，PC 机 Pentium 500 以上，硬盘 40G 以上，内存大于 256M
- 软件：Vmware Workstation+ubuntu+xshell 终端+ ARM-LINUX 交叉编译开发环境，Qtcreator 开发环境

3. 实验内容

- 分析传感器通过串口发送数据的格式，以及各个位代表的含义
- Qt 对串口类的使用
- 实现基于 ZigBee 无线传感网的智慧农业应用

4. 实验原理

4.1 串口通信数据格式

这次实验使用三个 zigbee 节点传感器分别为光照传感器，温湿度传感器和结露传感器，它们把实时采集的数据，通过 zigbee 的组网方式，发送的 zigbee 协调器端，协调器通过串口把数据发送到处理器，处理器对接受的数据进行处理。如图：

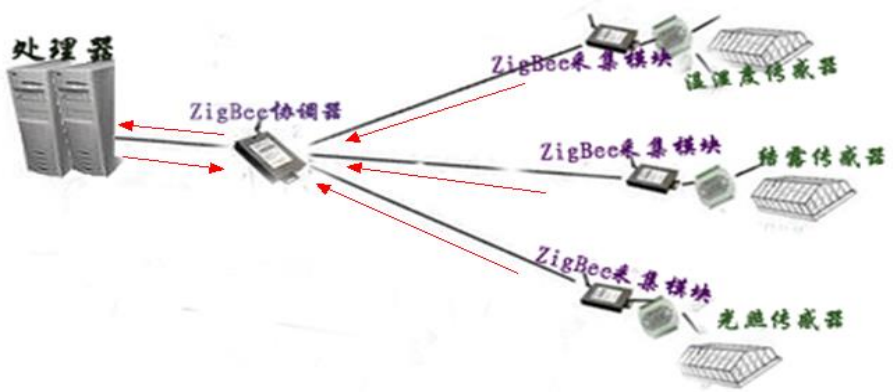


图 1 传感器数据传输方式

ICS-IOT-CEP 实验平台规定传感器数据传输格式如下：

```
u8 DataHeadH;              //包头 0xEE
u8 DataDeadL;              //包头  0xCC
u8 NetID;                  //所属网络标识  00(zigbee) 01(蓝牙)02(WiFi)03(IPv6)04(RFID)
u8 NodeAddress[4];         //节点地址
u8 FamilyAddress[4];       //根节点地址
u8 NodeState;          //节点状态   （00 未发现）  （01 已发现）
u8 NodeChannel;        //蓝牙节点通道
u8 ConnectPort;        //通信端口
u8 SensorType;         //传感器类型编号
u8 SensorID;               //相同类型传感器 ID
u8 SensorCMD;          //节点命令序号
u8 Sensordata1;        //节点数据 1
u8 Sensordata2;        //节点数据 2
u8 Sensordata3;            //节点数据 3
u8 Sensordata4;            //节点数据 4
u8 Sensordata5;        //节点数据 5
u8 Sensordata6;            //节点数据 6
u8 Resv1;              //保留字节 1
u8 Resv2;              //保留字节 2
u8 DataEnd;              //节点包尾 0xFF
```

一帧数据为定长 26 字节。在这三个传感器中我们主要通过判断 SensorType 位来判断传输过来的数据来源于哪个传感器，然后在分析相应的数据节点位的数据，最终得到我们想要的信息。我们通过对 Qt 代码得到的数据截图进行分析：

图 2 串口的数据分析

图上的红色圆框 02：代表光照传感器节点，0a：代表温湿度传感器节点，05：代表结露传感器节点；光照传感器和结露传感器的数据节点代表当前的检测状态（是否检测到光照/结露），温湿度传感器的数据节点代表当前的温度和湿度百分比。

4.2 Qt 串口类的使用

1）找到串口类相关的文件如图。

图 3 串口类的文件

把这几个文件放到工程文件夹下。

2）在工程里添加这个类

在工程文件中找到 headers 右击，选择“Add Existing Files…”，找到 posix_qextserialport.h 和 qextserialport.h 添加。同样将 posix_qextserialport.cpp 和 qextserialport.cpp 添加到 Sources 目录下。

如图：

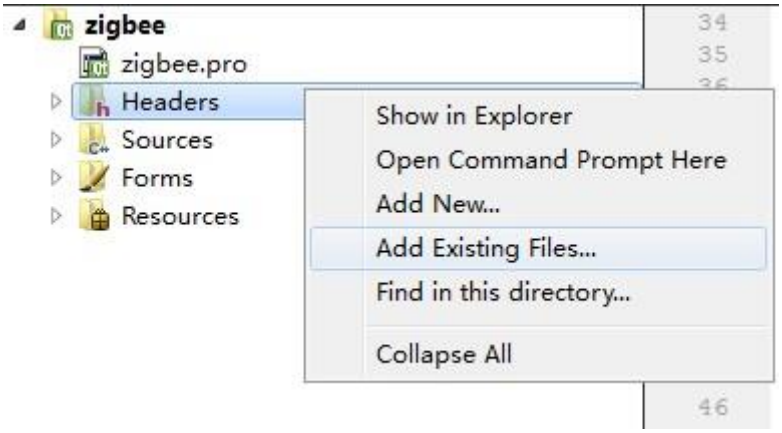


图 4 添加串口类

找到 posix_qextserialport.h 和 qextserialport.h 进行添加，同样添加 posix_qextserialport.cpp 和 qextserialport.cpp 步骤如上。

3）工程中对串口类的使用

对串口的初始化

```
void Widget::comportInit(comThread *&comport, Posix_QextSerialPort *&myCom,
                                                QString devname,int comNO)
{
    myCom = new Posix_QextSerialPort(devname,
                                     QextSerialBase::Polling); //新建串口类
    if(myCom->open(QIODevice::ReadWrite)){

        //return true;
    }else{
```

```
        //return false;
    }
        myCom->setBaudRate((BaudRateType)19);    //设置波特率 115200

        myCom->setDataBits((DataBitsType)3);    //设置数据位 8
        myCom->setParity((ParityType)0);        //设置校验 0
        myCom->setStopBits((StopBitsType)0);    //设置停止位 1
        myCom->setFlowControl(FLOW_OFF);    //设置流控制
        myCom->setTimeout(10);                  //设置延时
        comport = new comThread(myCom,this);
        connect(comport,SIGNAL(sensorData(QByteArray)),nodeform,
                                SLOT(reveiceSensordata(QByteArray)));
        comport->start();    //启动线程
}
```

读取串口的数据

```
QByteArray temp = comPort->readAll()。//temp 里存放串口发过来的数据。
```

4.3 信号与槽机制

1）信号与槽的概述

信号和槽机制是 QT 的核心机制，要精通 QT 编程就必须对信号和槽有所了解。信号和槽是一种高级接口，应用于对象之间的通信，它是 QT 的核心特性，也是 QT 区别于其它工具包的重要地方。信号和槽是 QT 自行定义的一种通信机制，它独立于标准的 C/C++语言。

2）信号

信号的声明是在头文件中进行的，QT 的 signals 关键字指出进入了信号声明区，随后即可 声明自己的信号。例如，下面是在 widget.h 定义了两个信号：

```
signals:
    void tempSender(double temp);
    void humSender(double hum);
```

这两个信号在程序中负责发送温度和湿度的值。另外需要注意的是信号的返回值只能是 void，不要指望能从信号返回什么有用的信息。

3）槽

槽是普通的 C++成员函数，可以被正常调用，它们唯一的特殊性就是很多信号可以与其相关联。当与其关联的信号被发射时，这个槽就会被调用。槽可以有参数，但槽的参数不能有缺省值。既然槽是普通的成员函数，因此与其它的函数一样，它们也有存取权限。槽的存取权限决定了谁能够与其相关联。同普通的 C++成员函数一样，槽函数也分为三种类型，即 public slots、private slots 和 protected slots。

public slots：

在这个区内声明的槽意味着任何对象都可将信号与之相连接。这对于组件编程非常有

用，你可以创建彼此互不了解的对象，将它们的信号与槽进行连接以便信息能够正确的传递。

protected slots：

在这个区内声明的槽意味着当前类及其子类可以将信号与之相连接。这适用于那些槽，它们是类实现的一部分，但是其界面接口却面向外部。

private slots：

在这个区内声明的槽意味着只有类自己可以将信号与之相连接。这适用于联系非常紧密的类。

下面是定义在 envplot.h 中的两个槽函数

```
public slots:
    void setTemp(double temp);
    void setHum(double hum);
```

4）信号与槽的关联

通过调用 QObject 对象的 connect 函数来将某个对象的信号与另外一个对象的槽函数相关联，这样当发射者发射信号时，接收者的槽函数将被调用。该函数的定义如下：

```
    bool QObject::connect ( const QObject * sender, const char * signal,
            const QObject * receiver, const char * member ) [static]
```

这个函数的作用就是将发射者 sender 对象中的信号 signal 与接收者 receiver 中的 member 槽函数联系起来。当指定信号 signal 时必须使用 QT 的宏 SIGNAL()，当指定槽函数时必须使用宏 SLOT()。下面是 widget.cpp 中将信号与槽连接起来。

```
    connect(nodeform,SIGNAL(humCheck(double)),splot,SLOT(setHum(double)));
    connect(nodeform,SIGNAL(tempCheck(double)),splot,SLOT(setTemp(double)));
```

如果发射者与接收者属于同一个对象的话，那么在 connect 调用中接收者参数可以省略。如下图：

```
    connect(this,SIGNAL(humSender(double)),SLOT(setHumlabel(double)));
    connect(this,SIGNAL(tempSender(double)),SLOT(setTemplabel(double)));
```

4.4 关键代码分析

1）widget.cpp 中主要串口显示界面和通信

```
  static int fd = -1;
static const char *DCM_DEV="/dev/pwm";

Widget::Widget(QWidget *parent) :
    QWidget(parent),
    ui(new Ui::Widget)
{
    ui->setupUi(this);
```

```
    //开启蜂鸣器设备
    if((fd=open(DCM_DEV, O_WRONLY))<0){
                        printf("Error opening %s device\n", DCM_DEV);
                        close(); }
  //设置窗口和按钮背景
    QPixmap myPixmap(":/rcs/mainpage.jpg");
    this->setWidgetbackground(this,myPixmap);

    QPixmap buttonPixmap(":/rcs/widgetButton.png");
    setButtonbackground(ui->startButton,buttonPixmap);

    QPixmap setButtonpixmap(":/rcs/setButton.png");
    setButtonbackground(ui->setButton,setButtonpixmap);
    //按钮响应窗口
    setDialog = new Dialog(this);
    nodeform = new Form(this);
    this->showFullScreen();
    nodeform->hide();
    connect(ui->startButton,SIGNAL(clicked()),nodeform,SLOT(showFullScreen()));
    //端口传送数据
    comportInit(thread_1, com_1,"/dev/ttySAC1",0);
   /**********绘制曲线并传递数据*****************/
    splot = new CpuPlot(this);
    ui->cuvlayout->addWidget(splot);
    connect(nodeform,SIGNAL(humCheck(double)),splot,SLOT(setHum(double)));
    connect(nodeform,SIGNAL(tempCheck(double)),splot,SLOT(setTemp(double)));
  /*********信号与槽的应用****************/
    connect(nodeform,SIGNAL(waterCheck(bool)),this,SLOT(receviceWaterdata(bool)));
    connect(nodeform,SIGNAL(higherCheck(bool)),this,SLOT(receviceHigherdata(bool)));
    connect(nodeform,SIGNAL(humCheck(double)),this,SLOT(receviceHumdata(double)));
    connect(nodeform,SIGNAL(tempCheck(double)),this,SLOT(receviceTempdata(double)));
}
```

2）主要应变功能函数

```
  /*接收结露数据  并根据数据 tempSwitch 做出判断，并响应  */
void Widget::receviceWaterdata(bool flag)
{
    if(setDialog->waterSwitch)
    {
        if(flag)
        {
            ui->Wlabel->setText(tr("结露"));
             QPixmap manpixmap_1(":/rcs/watercheck_1.png");
             ui->waterLabel->setPixmap(manpixmap_1);
        }
        else
```

```
            {
                ui->Wlabel->setText(tr("无结露"));
                QPixmap manpixmap_0(":/rcs/watercheck_0.png");
                ui->waterLabel->setPixmap(manpixmap_0);
            }
        }
}
/*接收光照数据  并根据数据 tempSwitch 做出判断  */
void Widget::receviceHigherdata(bool flag)
{
    if(setDialog->higherSwitch)
    {
        if(flag)
        {   //当有光照时停止报警，并更换图片及文字提示
            ui->Hlabel->setText(tr("有光照"));
            QPixmap higherpixmap_1(":/rcs/highercheck_1.png");
            ui->higherLabel->setPixmap(higherpixmap_1);
              int ret = ioctl(fd, PWM_IOCTL_STOP);
                        if(ret<0){
                            perror("stop the buzzer11111");
                            exit(1); }
        }
        else
        {    //当无光照时停止报警，并更换图片及文字提示
            ui->Hlabel->setText(tr("无光照"));
            QPixmap higherpixmap_0(":/rcs/highercheck_0.png");
            ui->higherLabel->setPixmap(higherpixmap_0);
            int ret= ioctl(fd, PWM_IOCTL_SET_FREQ, 100);
            if(ret < 0) {
                perror("the frequency of the buzzer is not right");
                exit(1);}
        }
    }
}
/*接收温度数据  并根据数据做出判断  tempSwitch*/
void Widget::receviceTempdata(double data)
{
    QString sensorData;
    tempData = data;
    qDebug()<<sensorData.setNum(data);
    ui->tempLabel->setText(sensorData.setNum(data));
    if(setDialog->tempSwitch)
    {
        if(data > setDialog->tempHeight||data < setDialog->tempLow)
        {
            int ret= ioctl(fd, PWM_IOCTL_SET_FREQ, 100);
```

```
                if(ret < 0) {
                    perror("the frequency of the buzzer is not right");
                    exit(1); }
            }
        }
        else { int ret = ioctl(fd, PWM_IOCTL_STOP);
            if(ret<0){
                perror("stop the buzzer22222");
                exit(1); }
            }
    }
/*接收湿度数据  并根据数据做出判断  tempSwitch*/
void Widget::receviceHumdata(double data)
{
    QString sensorData;
    humData = data;
    qDebug()<<sensorData.setNum(data);
    ui->humLabel->setText(sensorData.setNum(data));
    if(setDialog->tempSwitch)
    {
        if(data > setDialog->humHeight||data < setDialog->humLow)
        {
            int ret= ioctl(fd, PWM_IOCTL_SET_FREQ, 100);
            if(ret < 0) {
                perror("the frequency of the buzzer is not right");
                exit(1);    }
            }
        }
        else
        {
            int ret = ioctl(fd, PWM_IOCTL_STOP);
            if(ret<0){
                perror("stop the buzzer33333");
                exit(1);}
        }
}
```

3）Qt 线程的使用

```
#ifndef COMTHREAD_H
#define COMTHREAD_H
#include <QThread>
#include "posix_qextserialport.h"
class comThread : public QThread
{
    Q_OBJECT
public:
```

```
    explicit comThread(Posix_QextSerialPort *&com,QObject *parent);
    void ReceiveData();
    virtual void run();     //线程执行的任务
private:
    Posix_QextSerialPort *comPort;     //声明一个串口
signals:
    void sensorData(QByteArray);     //发送串口数据
public slots:
};
#endif // COMTHREAD_H
```

这个线程的主要目的是循环检测串口是否接受到数据，如果有数据它将以信号的形式把数据发送到相应的窗口。使用线程类我们主要就是完成 run() 这个函数，这个函数来完成要执行的任务。

```
void comThread::ReceiveData()
{
    QByteArray temp = comPort->readAll();     //读取串口数据
    if(!temp.isEmpty())   //判断是否为空
    {
        if(temp.length() > COMDATAMAXLENGTH) //判断是否满足命令长度
        {
            temp.clear();
        }
        if(temp.length() == COMDATAMAXLENGTH &&
                    (quint8)temp[0] == 0xee &&
                    (quint8)temp[1] == 0xcc &&
                    (quint8)temp[COMDATAMAXLENGTH-1] == 0xff) //判断数据的包头，和包尾。
        {
            emit sensorData(temp);     //发送数据
        }
    }
}
void comThread::run()
{
    While(1)   //循环检测串口
    {
        ReceiveData();
        msleep(20);
    }
}
```

值得一提的是我们很有必要在 while 循环中添加一个延时，主要是把 cpu 时间片让给其他线程。

以上篇幅有限，给出部分代码，用户可以自行分析该系统的全部代码

5. 实验步骤

1）打开 ubuntu 虚拟机输入查看网络连接命令（确定 IP 地址）如下图：

```
cbt@Cyb-Bot: ~$ ifconfig eth3
eth3      Link encap:Ethernet  HWaddr 00:0C:29:4F:66:3A
          inet addr:192.168.1.12  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::20c:29ff:fe4f:663a/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:349924 errors:0 dropped:0 overruns:0 frame:0
          TX packets:416925 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:167613886 (159.8 MiB)  TX bytes:409374631 (390.4 MiB)
          Interrupt:19 Base address:0x2024
cbt@Cyb-Bot: ~$
```

2）重启虚拟机的 smb、nfs 以及关闭防火墙(如果经常关闭虚拟机要执行此操作)。

```
cbt@Cyb-Bot: ~$ service smb restart
关闭 SMB 服务：                                            [确定]
启动 SMB 服务：                                            [确定]
cbt@Cyb-Bot: ~$ service nfs restart
关闭 NFS mountd：                                          [确定]
关闭 NFS 守护进程：                                        [确定]
关闭 NFS quotas：                                          [确定]
关闭 NFS 服务：                                            [确定]
启动 NFS 服务：                                            [确定]
关掉 NFS 配额：                                            [确定]
启动 NFS 守护进程：                                        [确定]
启动 NFS mountd：                                          [确定]
cbt@Cyb-Bot: ~$ service iptables stop
cbt@Cyb-Bot: ~$
```

3）通过 smb 服务器把我们的 Qt 实例源码 zigbee 目录（在光盘 Cortex-A53\Linux\SRC\item\智能农业\zigbee 目录下） 放到共享目录。

4）搭建 Qt linux 编译环境

在 linux 虚拟机上进行如下操作来查看 qmake 的路径：

```
cbt@Cyb-Bot: /$ which qmake
/usr/local/Trolltech/QtEmbedded-4.7.0-arm/bin/qmake
```

由于我们编译的 Qt 源码要在开发平台上运行使用，所以必须要使用这个路径 qmake。如果没有这个文 件路径我们需要下载 QtEmbedded-4.7.0-arm.tar.gz 压缩包，解压之后放到/usr/local/Trolltech/目录下。然后配置环境变量如下：

```
cbt@Cyb-Bot: /$ export PATH=/usr/local/Trolltech/QtEmbedded-4.7.0-arm/bin:$PATH
```

执行 which qmake 之后出现上面的结果证明我们的 Qt 环境已经搭建完成。

5）编译 Qt 源码程序

这时我们可以进入我们的 Qt 源码里执行如下命令：

```
cbt@Cyb-Bot: /CBT-6818/zigbee$ qmake
cbt@Cyb-Bot: /CBT-6818/zigbee$ make
```

这个命令顺利执行完后证命生成了可执行文件。

6）挂载我们的虚拟机共享目录（通过 nfs 服务）。

```
[root@CBT-6818:/mnt]# mount -t nfs -o nolock,rsize=4096,wsize=4096 192.168.1.12:/CBT-6818    /mnt/
[root@CBT-6818:/mnt]# cd /mnt/
```

以上要根据自己的实际共享目录来操作。

7）运行 Qt 程序

备注：本实验，需要将全功能物联网教学科研平台上配套的 ZigBee 模块烧写指定程序，方可由 Cortex-A53 平台完成对 ZigBee 无线传感网信息的处理，因此在做实验前，请确保平台上 ZigBee 模块已经烧写有配套的程序。ZigBee 部分配套程序请读者参加 ZigBee 部分实验指导书《无线传感网络演示实验》章节，此处不再赘述。

进入开发平台的/mnt/目录，找到 Qt 源码的位置并进入这个文件夹

```
[root@CBT-6818:/mnt/zigbee]# pwd
/mnt/zigbee
[root@Cyb-Bot zigbee]# ./zigbee -qws
```

8）程序运行效果

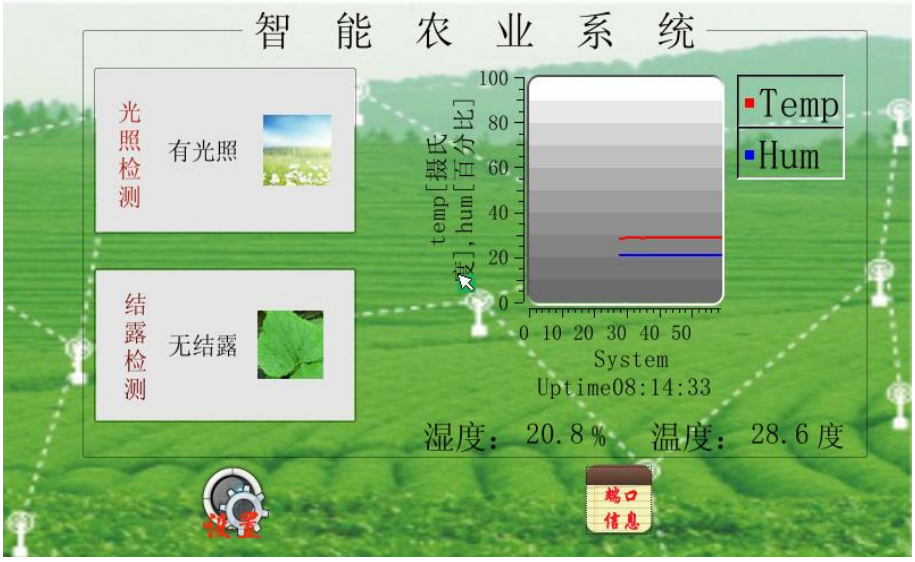


图 5 程序运行效果

左侧通过文字和图片显示光照和结露信息，右边通过曲线（有上角的 temp 和 hum 用来选择显示曲线）和数据显示温湿度

图 6 传感器报警设置

结露和光照是默认打开报警，温湿度是通过上下按钮设置上下限，每修改一次数值都需要打开报警之后才为有效数据

图 7 传感器信息的图表显示

通过表格的形式显示各个端口的传感器的详细信息。

实验四. 基于无线传感网的智能空调实训案例

1. 实验目的

- 掌握 ICS-IOT-CEP 实验平台上传感器与处理器通过串口通信的协议。
- 掌握 Qt 的信号与槽机制的使用。
- 熟悉 zigbee 的组网通信。
- 学习 Qt 如何使用第三方串口插件。

2. 实验环境

- 硬件：ICS-IOT-CEP 实验平台，PC 机 Pentium 500 以上，硬盘 40G 以上，内存大于 256M。
- 软件：Vmware Workstation +ubuntu + xshell 终端 + ARM-LINUX 交叉编译开发环境，Qtcreator 开发环境。

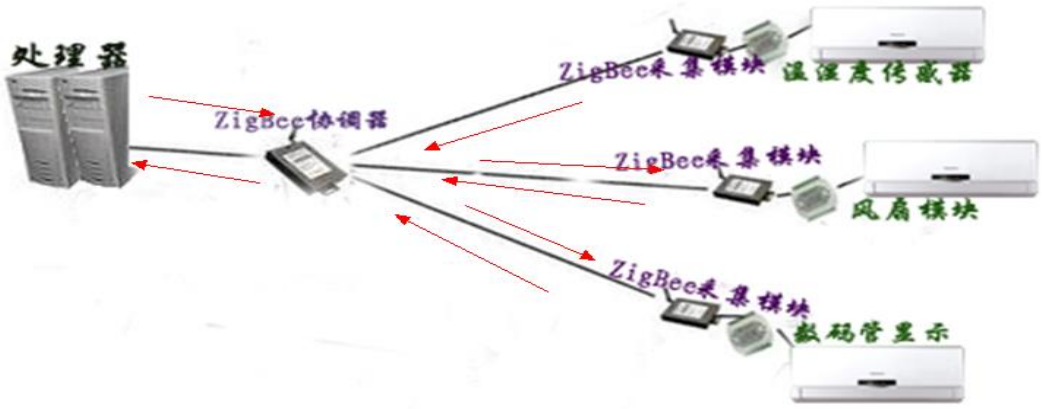
3. 实验内容

- 分析传感器通过串口发送数据的格式，以及各个位代表的含义
- Qt 对串口类的使用
- 实现基于 ZigBee 无线传感网的智能空调系统应用

4. 实验原理

4.1 串口通信数据格式

这次实验使用三个 zigbee 节点传感器分别为风扇传感器，温湿度传感器和数码管，它们把实时采集的数据，通过 zigbee 的组网方式，发送的 zigbee 协调器端，协调器通过串口把数据发送到处理器，处理器对接受的数据进行处理。如图：



ICS-IOT-CEP 实验平台规定传感器数据传输格式如下：

```
u8 DataHeadH;              //包头 0xEE
u8 DataDeadL;              //包头  0xCC
u8 NetID;                  //所属网络标识  00(zigbee) 01(蓝牙)02(WiFi)03(IPv6)04(RFID)
u8 NodeAddress[4];         //节点地址
u8 FamilyAddress[4];       //根节点地址
u8 NodeState;          //节点状态   （00 未发现）  （01 已发现）
u8 NodeChannel;        //蓝牙节点通道
u8 ConnectPort;        //通信端口
u8 SensorType;         //传感器类型编号
u8 SensorID;               //相同类型传感器 ID
u8 SensorCMD;          //节点命令序号
u8 Sensordata1;        //节点数据 1
u8 Sensordata2;        //节点数据 2
u8 Sensordata3;            //节点数据 3
u8 Sensordata4;            //节点数据 4
u8 Sensordata5;        //节点数据 5
u8 Sensordata6;            //节点数据 6
u8 Resv1;              //保留字节 1
u8 Resv2;              //保留字节 2
u8 DataEnd;              //节点包尾 0xFF
```

一帧数据为定长 26 字节。在这三个传感器中我们主要通过判断 SensorType 位来判断传输过来的数据来源于哪个传感器，然后在分析相应的数据节点位的数据，最终得到我们想要的信息。我们通过对 Qt 代码得到的数据截图进行分析：

图 2 串口的数据分析

图上的红色圆框 12：代表风扇传感器节点，0a：代表温湿度传感器节点，13：代表数码管传感器节点；风扇传感器和数码管传感器的数据节点代表当前的检测状态（是否检测到

光照/结露），温湿度传感器的数据节点代表当前的温度和湿度百分比。

4.2 Qt 串口类的使用

1）找到串口类相关的文件如图

图 3 串口类的文件

把这几个文件放到工程文件夹下。

2）在工程里添加这个类

在工程文件中找到 headers 右击，选择“Add Existing Files…”，找到 posix_qextserialport.h 和 qextserialport.h 添加。同样将 posix_qextserialport.cpp 和 qextserialport.cpp 添加到 Sources 目录下。

如图：

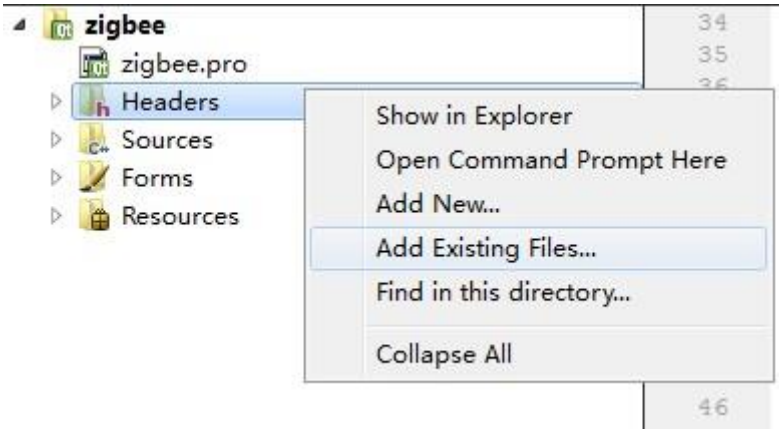


图 4 添加串口类

找到 posix_qextserialport.h 和 qextserialport.h 进行添加，同样添加 posix_qextserialport.cpp 和 qextserialport.cpp 步骤如上。

3）工程中对串口类的使用

对串口的初始化。

```
void Widget::comportInit(comThread *&comport, Posix_QextSerialPort *&myCom,
                                                QString devname,int comNO)
{
    myCom = new Posix_QextSerialPort(devname,
                                     QextSerialBase::Polling); //新建串口类
    if(myCom->open(QIODevice::ReadWrite)){
```

```
        //return true;
    }else{
        //return false;
    }
        myCom->setBaudRate((BaudRateType)19);    //设置波特率 115200

        myCom->setDataBits((DataBitsType)3);    //设置数据位 8
        myCom->setParity((ParityType)0);        //设置校验 0
        myCom->setStopBits((StopBitsType)0);    //设置停止位 1
        myCom->setFlowControl(FLOW_OFF);    //设置流控制
        myCom->setTimeout(10);                  //设置延时
        comport = new comThread(myCom,this);
        connect(comport,SIGNAL(sensorData(QByteArray)),nodeform,
                                SLOT(reveiceSensordata(QByteArray)));
        comport->start();    //启动线程
}
```

2）读取串口的数据

```
QByteArray temp = comPort->readAll()。//temp 里存放串口发过来的数据。
```

4.3 信号与槽机制

1）信号与槽的概述

信号和槽机制是 QT 的核心机制，要精通 QT 编程就必须对信号和槽有所了解。信号和槽是一种高级接口，应用于对象之间的通信，它是 QT 的核心特性，也是 QT 区别于其它工具包的重要地方。信号和槽是 QT 自行定义的一种通信机制，它独立于标准的 C/C++语言。

2）信号

信号的声明是在头文件中进行的，QT 的 signals 关键字指出进入了信号声明区，随后即可 声明自己的信号。例如，下面是在 widget.h 定义了两个信号：

```
signals:
    void tempSender(double temp);
    void humSender(double hum);
```

这两个信号在程序中负责发送温度和湿度的值。另外需要注意的是信号的返回值只能是 void，不要指望能从信号返回什么有用的信息。

3）槽

槽是普通的 C++成员函数，可以被正常调用，它们唯一的特殊性就是很多信号可以与其相关联。当与其关联的信号被发射时，这个槽就会被调用。槽可以有参数，但槽的参数不能有缺省值。既然槽是普通的成员函数，因此与其它的函数一样，它们也有存取权限。槽的存取权限决定了谁能够与其相关联。同普通的 C++成员函数一样，槽函数也分为三种类型，即 public slots、private slots 和 protected slots。

public slots：

在这个区内声明的槽意味着任何对象都可将信号与之相连接。这对于组件编程非常有用，你可以创建彼此互不了解的对象，将它们的信号与槽进行连接以便信息能够正确的传递。

protected slots：

在这个区内声明的槽意味着当前类及其子类可以将信号与之相连接。这适用于那些槽，它们是类实现的一部分，但是其界面接口却面向外部。

private slots：

在这个区内声明的槽意味着只有类自己可以将信号与之相连接。这适用于联系非常紧密的类。

下面是定义在 widget.h 中的两个槽函数

```
public slots:
    void receiverTempData(double data);
    void receiverHumData(double data);
    void receiverCoData(bool flag);
     void receiverFanData(bool flag);
```

4、信号与槽的关联

通过调用 QObject 对象的 connect 函数来将某个对象的信号与另外一个对象的槽函数相关联，这样当发射者发射信号时，接收者的槽函数将被调用。该函数的定义如下

```
bool QObject::connect ( const QObject * sender, const char * signal,
const QObject * receiver, const char * member ) [static]
```

这个函数的作用就是将发射者 sender 对象中的信号 signal 与接收者 receiver 中的 member 槽函数联系起来。当指定信号 signal 时必须使用 QT 的宏 SIGNAL()，当指定槽函数时必须使用宏 SLOT()。下面是 widget.cpp 中将信号与槽连接起来。

```
connect(form,SIGNAL(CoCheck(bool)),this,SLOT(receiverCoData(bool)));
connect(form,SIGNAL(fanCheck(bool)),this,SLOT(receiverFanData(bool)));
connect(form,SIGNAL(humCheck(double)),this,SLOT(receiverHumData(double)));
```

如果发射者与接收者属于同一个对象的话，那么在 connect 调用中接收者参数可以省略。如下图：

```
connect(this,SIGNAL(FanSender(double)),thread_1,SLOT(setWData(double )));
connect(this,SIGNAL(tempSender(double)),thread_1,SLOT(setTempData(double)));
```

4.4 关键代码分析

1）widget.cpp 中主要串口显示界面和通信

```
Widget::Widget(QWidget *parent) :
    QWidget(parent),
    ui(new Ui::Widget)
{
```

```
    ui->setupUi(this);
    this->showFullScreen();
    this->tempdata = ui->doubleSpinBox->text().toDouble();
    /*美化各个控件*/
    QPixmap widgetPixmap(":/rcs/line.png");
    this->setWidgetbackground(this,widgetPixmap);

    QPixmap labelPixmap(":/rcs/air1.png");
    ui->label_9->setPixmap(labelPixmap);

  QPixmap buttonPixmap(":/rcs/sure.png");
  QPixmap buttonPixmap_2(":/rcs/ser.png");
  setButtonbackground(ui->navButton,buttonPixmap_2);
  setButtonbackground(ui->startButton,buttonPixmap);



    QPixmap buttonPixmap_0(":/rcs/open.png");
      setButtonbackground(ui->fanButton,buttonPixmap_0);
   setButtonbackground(ui->selfButton,buttonPixmap_0);



    /*设备表格*/
    form = new Form(this);
    form->hide();
/*********信号与槽机制*******************/
    connect(ui->navButton,SIGNAL(clicked()),form,SLOT(showFullScreen()));
    comportInit(thread_1,com_1,"/dev/ttySAC1",0);
    connect(form,SIGNAL(CoCheck(bool)),this,SLOT(receiverCoData(bool)));
    connect(form,SIGNAL(fanCheck(bool)),this,SLOT(receiverFanData(bool)));
    connect(form,SIGNAL(humCheck(double)),this,SLOT(receiverHumData(double)));
    connect(form,SIGNAL(tempCheck(double)),this,SLOT(receiverTempData(double)));
    connect(this,SIGNAL(FanSender(double)),thread_1,SLOT(setWData(double )));
   // connect(form,SIGNAL(tempCheck(double)),thread_1,SLOT(setTempData(double)));
    connect(this,SIGNAL(tempSender(double)),thread_1,SLOT(setTempData(double)));

}
```

2）主要应变功能函数

```
//风扇调控模块
void Widget::receiverFanData(bool flag)
{

    double sensor;
//如果温差过大或者设置开启风扇  都需要打开风扇
  if(flag)
    {
```

```
            QPixmap labelPixmap_0(":/rcs/air_0.png");
            ui->label_9->setPixmap(labelPixmap_0);
    }
    else{   QPixmap labelPixmap(":/rcs/air1.png");
            ui->label_9->setPixmap(labelPixmap);

    }
    if(!(fanSwitch))
    {
            sensor=0x01;
            emit FanSender(sensor);
  }
else{
       if(!(selfSwitch))
            {
              if((tempData-tempdata)>10){
                        sensor=0x02;
                        emit FanSender(sensor);
                }
                else{
                        if((tempData-tempdata)>2&&(tempData-tempdata)<10)
                        {
                                sensor=0x01;
                                emit FanSender(sensor);
                        }
                        else{
                        sensor=0x00;
                        emit FanSender(sensor);
                                }
                        }
               }
            else{
                  sensor=0x00;
                  emit FanSender(sensor);
                  }

            }

}
//按钮之间的调控
void Widget::on_fanButton_clicked()
{//fanSwitch 默认是 true
    if(changState(fanSwitch)){

            QPixmap fbuttonPixmap_0(":/rcs/open.png");
            setButtonbackground(ui->fanButton,fbuttonPixmap_0);
```

```
            ui->selfButton->setDisabled(false);
        }
        else{
     QPixmap buttonPixmap_1(":/rcs/close.png");
            setButtonbackground(ui->fanButton,buttonPixmap_1);
            ui->selfButton->setDisabled(true);
        }

}
void Widget::on_startButton_clicked()
{
        if(changState(tempSwitch))
        {
            this->tempdata = ui->doubleSpinBox->text().toDouble();
            //设置的温度值
        }
        else {

        }

}
void Widget::on_selfButton_clicked()
{
        if(changState(selfSwitch))
        {
     QPixmap sbuttonPixmap_0(":/rcs/open.png");
            setButtonbackground(ui->selfButton,sbuttonPixmap_0);

//              ui->selfButton->setText("close");
   ui->fanButton->setDisabled(false);
        }
        else {
     QPixmap buttonPixmap_4(":/rcs/close.png");
            setButtonbackground(ui->selfButton,buttonPixmap_4);

//              ui->selfButton->setText(tr("open"));
   ui->fanButton->setDisabled(true);

        }

}
```

3）Qt 线程的使用

```
#ifndef COMTHREAD_H
#define COMTHREAD_H
#include <QThread>
```

```
#include "posix_qextserialport.h"
class comThread : public QThread
{
    Q_OBJECT
public:
    explicit comThread(Posix_QextSerialPort *&com,QObject *parent);
    void ReceiveData();
    virtual void run();    //线程执行的任务
private:
    Posix_QextSerialPort *comPort;    //声明一个串口
signals:
    void sensorData(QByteArray);    //发送串口数据
public slots:
};
#endif // COMTHREAD_H
```

这个线程的主要目的是循环检测串口是否接受到数据，如果有数据它将以信号的形式把数据发送到相应的窗口。使用线程类我们主要就是完成 run() 这个函数，这个函数来完成要执行的任务。

```
void comThread::ReceiveData()
{
    QByteArray temp = comPort->readAll();    //读取串口数据
    if(!temp.isEmpty())   //判断是否为空
    {
        if(temp.length() > COMDATAMAXLENGTH) //判断是否满足命令长度
        {
            temp.clear();
        }
        if(temp.length() == COMDATAMAXLENGTH &&
                    (quint8)temp[0] == 0xee &&
                    (quint8)temp[1] == 0xcc &&
                    (quint8)temp[COMDATAMAXLENGTH-1] == 0xff) //判断数据的包头，和包尾。
        {
            emit sensorData(temp);    //发送数据
        }
    }
}
void comThread::run()
{
    While(1)   //循环检测串口
    {
        ReceiveData();
        msleep(20);
    }
}
//写风扇数据，通过信号与槽的数据传递，改变风扇的状态
```

```
void comThread::setWData(double sensordata)
{




                if((quint8)temp1[14]==0x12)
                {
          (quint8)temp1[22].operator =(sensordata);
                    comPort->write(temp1);

   }

}
//写数码管数据，通过信号与槽的数据传递，分析得到的数据，将数据显示到数码管上

void comThread::setTempData(double data)
{

    if((quint8)temp1[14]==0x13)
    {
  if((data-100)<0){
              (quint8)temp1[18].operator =(0x02);
               int total=data*100;
               int first=total/1000;
               int second=(total-first*1000)/100;
               int third=((total-first*1000)-second*100)/10;
               int forth = ((total-first*1000)-second*100)-third*10;
                (quint8)temp1[22].operator =(forth);
                (quint8)temp1[21].operator =(third);
                (quint8)temp1[20].operator =(second);
                (quint8)temp1[19].operator =(first);
      comPort->write(temp1);
         }
      }
}
```

值得一提的是我们很有必要在 while 循环中添加一个延时，主要是把 cpu 时间片让给其他线程。

以上篇幅有限，给出部分代码，用户可以自行分析该系统的全部代码。

5. 实验步骤

1）打开 ubuntu 虚拟机 输入查看网络连接命令（确定 IP 地址）如下图：

```
cbt@Cyb-Bot: ~$ ifconfig eth3
```

```
eth3      Link encap:Ethernet  HWaddr 00:0C:29:4F:66:3A
          inet addr:192.168.1.12  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::20c:29ff:fe4f:663a/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:349924 errors:0 dropped:0 overruns:0 frame:0
          TX packets:416925 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:167613886 (159.8 MiB)  TX bytes:409374631 (390.4 MiB)
          Interrupt:19 Base address:0x2024

cbt@Cyb-Bot: ~$
```

2）重启虚拟机的 smb、nfs 以及关闭防火墙（如果经常关闭虚拟机要执行此操作）。

```
cbt@Cyb-Bot: ~$ service smb restart
关闭 SMB 服务：                                            [确定]
启动 SMB 服务：                                            [确定]
cbt@Cyb-Bot: ~$ service nfs restart
关闭 NFS mountd：                                          [确定]
关闭 NFS 守护进程：                                        [确定]
关闭 NFS quotas：                                          [确定]
关闭 NFS 服务：                                            [确定]
启动 NFS 服务：                                            [确定]
关掉 NFS 配额：                                            [确定]
启动 NFS 守护进程：                                        [确定]
启动 NFS mountd：                                          [确定]
cbt@Cyb-Bot: ~$ service iptables stop
cbt@Cyb-Bot: ~$
```

3）通过 smb 服务器把我们的 Qt 实例源码 zigbee 目录（在光盘 Cortex-A53\Linux\SRC\item\智能空调\Clair 目录下） 放到共享目录。

4）搭建 Qt linux 编译环境

在 linux 虚拟机上进行如下操作来查看 qmake 的路径：

```
cbt@Cyb-Bot: /$ which qmake
/usr/local/Trolltech/QtEmbedded-4.7.0-arm/bin/qmake
```

由于我们编译的 Qt 源码要在开发平台上运行使用，所以必须要使用这个路径 qmake。如果没有这个文 件路径我们需要下载 QtEmbedded-4.7.0-arm.tar.gz 压缩包，解压之后放到/usr/local/Trolltech/目录下。然后配置环境变量如下：

```
cbt@Cyb-Bot: /$ export PATH=/usr/local/Trolltech/QtEmbedded-4.7.0-arm/bin:$PATH
```

执行 which qmake 之后出现上面的结果证明我们的 Qt 环境已经搭建完成。

5）编译 Qt 源码程序

这时我们可以进入我们的 Qt 源码里执行如下命令：

```
cbt@Cyb-Bot: /CBT-6818/Clair$ qmake
```

```
cbt@Cyb-Bot: /CBT-6818/Clair$ make
```

这个命令顺利执行完后证命生成了可执行文件。

6）挂载虚拟机共享目录（通过 nfs 服务）。

```
[root@CBT-6818:~]# mount -t nfs -o nolock,rsize=4096,wsize=4096 192.168.1.12:/CBT-6818    /mnt/
[root@CBT-6818:~]# cd /mnt/
```

以上要根据自己的实际共享目录来操作。

7）运行 Qt 程序

备注：本实验，需要将全功能物联网教学科研平台上配套的 ZigBee 模块烧写指定程序，方可由 A9 平台完成对 ZigBee 无线传感网信息的处理，因此在做实验前，请确保平台上 ZigBee 模块已经烧写有配套的程序。ZigBee 部分配套程序请读者参加 ZigBee 部分实验指导书《无线传感网络演示实验》章节，此处不再赘述。

进入开发平台的/mnt/目录，找到 Qt 源码的位置并进入这个文件夹

```
[root@CBT-6818: /mnt/Clair]# pwd
/mnt/Clair
[root@CBT-6818: /mnt/Clair]# ./Clair –qws
```

8）程序运行效果

图 5 程序运行效果

左侧通过文字显示室内温湿度信息，右边为风扇的工作模式:分为人工模式和智能模式。

图 6 智能模式启动

启动智能模式之后，当温度的实际值与设置的温度值（没有设置的时候默认为 23 度）相差大于 2 小于 10 的时候，风扇中速转动，当相差大于 10 的时候，风扇以快速转动。否则不转动。

启动人工模式之后，风扇打开，之后人工确定关闭，风扇关闭。

（由左边的空调图片可以查看风扇状态）

图 7 传感器信息的图表显示

通过表格的形式显示各个端口的传感器的详细信息。