



# EPC-28X/M28X 用户手册

工控产品

UM2014052401

V1.00

Date: 2015/01/28

产品用户手册



类别	内容
关键词	EPC-283、EPC-287、M283、M287，Linux，Ethernet、CAN、RS232、LCD，USB
摘要	使用指南

## 修订历史

版本	日期	原因
V0.90	2014/05/24	创建文档
V1.06	2014/10/20	完善了文档
V1.07	2014/11/25	更新了产品图片
V1.08	2015/01/29	修改内容，增加 EPC-28x-L 相关
V1.09	2015/04/17	修改了 I2C 小节，I2C 被 RTC 芯片使用了 修改了 ADC 小节，新增 HSADC 说明，更新图片 修改了 SPI 小节，更新图片
V1.10	2015/09/17	增加了 uboot 命令小结 修改了 SPI3 引线图片 规范了文档 增加了 USB 烧写方案文档

## 目录

1	产品简介.....	1
1.1	关于产品.....	1
1.2	功能特点.....	3
1.3	硬件资源.....	3
1.4	产品命名.....	4
1.5	EPC-28xC-L 驱动软件.....	4
2	基本操作.....	6
1.5.1	启动选择.....	6
1.5.2	串口连接设置.....	6
1.5.3	串口登录.....	9
1.5.4	SSH 远程登录.....	10
1.6	关机.....	10
1.7	Qt 演示程序.....	11
1.8	输入设备.....	11
1.8.1	触摸屏和 USB 鼠标.....	11
1.8.2	USB 键盘.....	11
1.9	查看系统信息.....	11
1.10	系统设置.....	12
1.10.1	网络设置.....	12
1.10.2	系统时钟.....	12
1.11	文件传输.....	13
1.11.1	SSH 文件传输.....	13
1.11.2	NFS 文件传输.....	14
1.12	U 盘使用.....	14
1.13	TF 卡使用.....	15
1.14	U-Boot 交互.....	15
	DRAM: 64 MB.....	15
	NAND: 256 MB.....	15
1.14.1	预设的组合命令.....	17
1.15	LCD 待机设置.....	19
1.16	LCD 背光调节.....	19
1.17	串口测试.....	19
3	文件系统.....	21
1.18	分区描述.....	21
1.19	支持的文件系统.....	21
1.20	安装第三方软件.....	22
1.21	程序开机自启动.....	22
1.21.1	应用程序的自动升级.....	23
1.22	修改文件系统.....	24
4	应用程序开发.....	25
1.23	应用程序开发环境构建.....	25
1.23.1	嵌入式 Linux 开发一般方法.....	25

1.23.2	安装主机操作系统.....	26
1.23.3	构建交叉开发环境.....	26
1.23.4	NFS 服务器配置 .....	27
1.24	Hello 程序.....	28
1.25	I <sup>2</sup> C 接口 .....	29
1.25.1	open 调用 .....	30
1.25.2	ioctl 调用.....	30
1.25.3	write 调用.....	31
1.25.4	read 调用 .....	31
1.25.5	close 调用.....	31
1.26	I/O 端口使用 .....	31
1.27	蜂鸣器使用.....	33
1.28	ADC .....	33
1.28.1	ADC 驱动模块的加载 .....	34
1.28.2	操作接口 .....	34
1.28.3	计算公式.....	35
1.28.4	操作示例.....	35
1.29	串口编程.....	36
1.29.1	访问串口设备 .....	36
1.29.2	配置串口接口属性.....	37
1.29.3	获得和设置串口信号线状态 .....	45
1.30	Socket CAN 编程 .....	46
1.30.1	初始化 CAN 网络接口 .....	46
1.30.2	socket can 编程 .....	46
1.30.3	示例程序.....	50
1.31	看门狗使用.....	54
1.31.1	概述.....	54
1.31.2	范例.....	54
1.32	SPI 接口 .....	55
1.32.1	open 调用 .....	56
1.32.2	ioctl 调用.....	56
1.32.3	示例代码.....	58
5	QT 4 编程 .....	65
1.33	背景知识.....	65
1.34	Qt 介绍.....	65
1.34.1	Qt 简介 .....	65
1.34.2	Qt/E 简介 .....	65
1.35	编译环境的搭建.....	65
1.35.1	编译 Qt-4.7.3 源码包.....	65
1.35.2	编译环境的设置.....	65
1.36	Hello world .....	66
1.36.1	编译 hello 程序.....	66
1.36.2	在目标板上运行 hello 程序 .....	67
1.37	qmake 与 pro 文件.....	68

1.37.1	pro 文件例程 .....	68
1.37.2	pro 文件常见配置 .....	69
1.38	桌面版本的 Qt SDK 使用简介 .....	70
1.38.1	桌面版本 Qt SDK 简介 .....	70
1.38.2	桌面版本 Qt SDK 的安装 .....	71
1.38.3	Qt Creator 配置 .....	71
1.38.4	Qt Creator 使用例程 .....	71
1.38.5	移植 hello world .....	75
1.39	zylauncher 图形框架 .....	75
6	<b>系统恢复和更新</b> .....	80
1.40	TF 恢复系统 .....	80
1.40.1	制作 Nand Flash 格式化启动盘 .....	80
1.40.2	格式化 Nand Flash .....	81
1.40.3	制作系统安装引导盘 .....	81
1.40.4	安装 Linux 系统 .....	82
1.41	USB 恢复系统 .....	83
1.41.1	执行 USB 烧写 .....	83
7	<b>免责声明</b> .....	87

## 1 产品简介

### 1.1 关于产品

M28x-128(64)LI(C)及 EPC-28x 系列产品是广州致远电子股份有限公司以 Freescale ARM9 MCIMX283/287 处理器为核心开发的嵌入式 ARM 工控产品，具有高性能、接口齐全、低功耗、成本低等特点。

图 1.1是M28x-128(64)L(W)I工控核心板系列产品（以下简称核心板）图片，EPC-28xC-L(W)工控主板系列产品（以下简称EPC）是在M28x系列产品的基础上，完善外围电路，功能框图如图 1.2。产品图片如图 1.3。

Freescale 的两种 ARM9 CPU：MCIMX283 和 MCIMX287 管脚兼容，但接口数量略有不同，如 MCIMX287 有两路以太网，MCIMX283 只有一路以太网，所以本文把使用 MCIMX283 芯片的核心板及 EPC 产品简称为 283 产品，使用 MCIMX287 芯片的核心板及 EPC 产品简称为 287 产品。

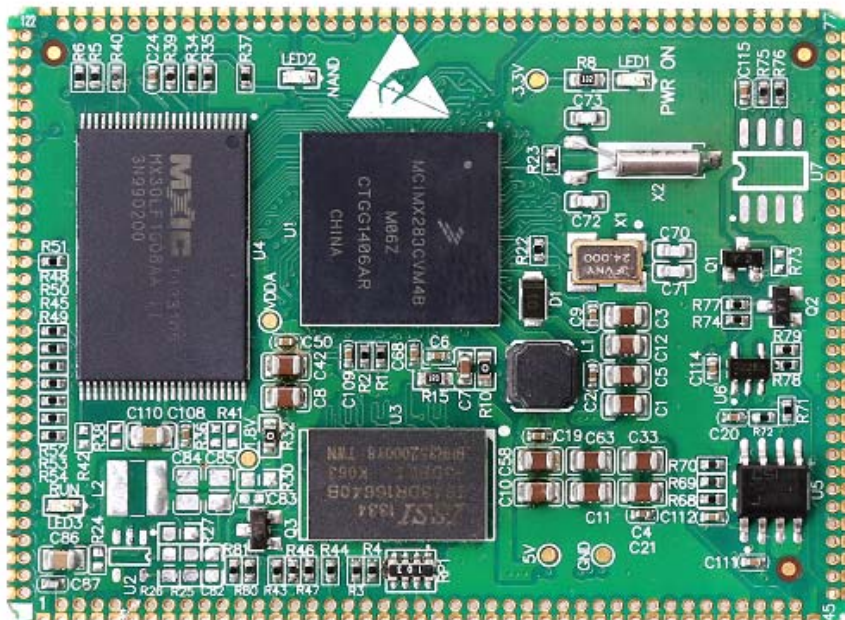


图 1.1 M283-128LI 工控核心板



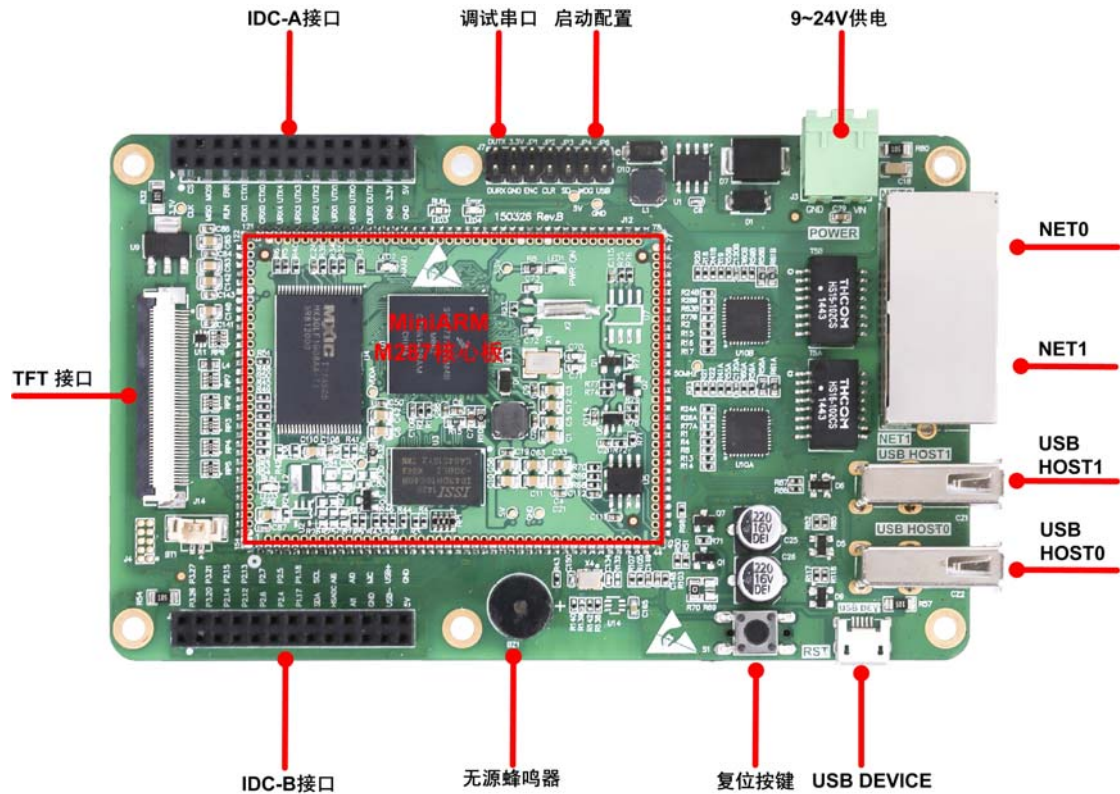


图 1.2 EPC-287C-L 功能框图

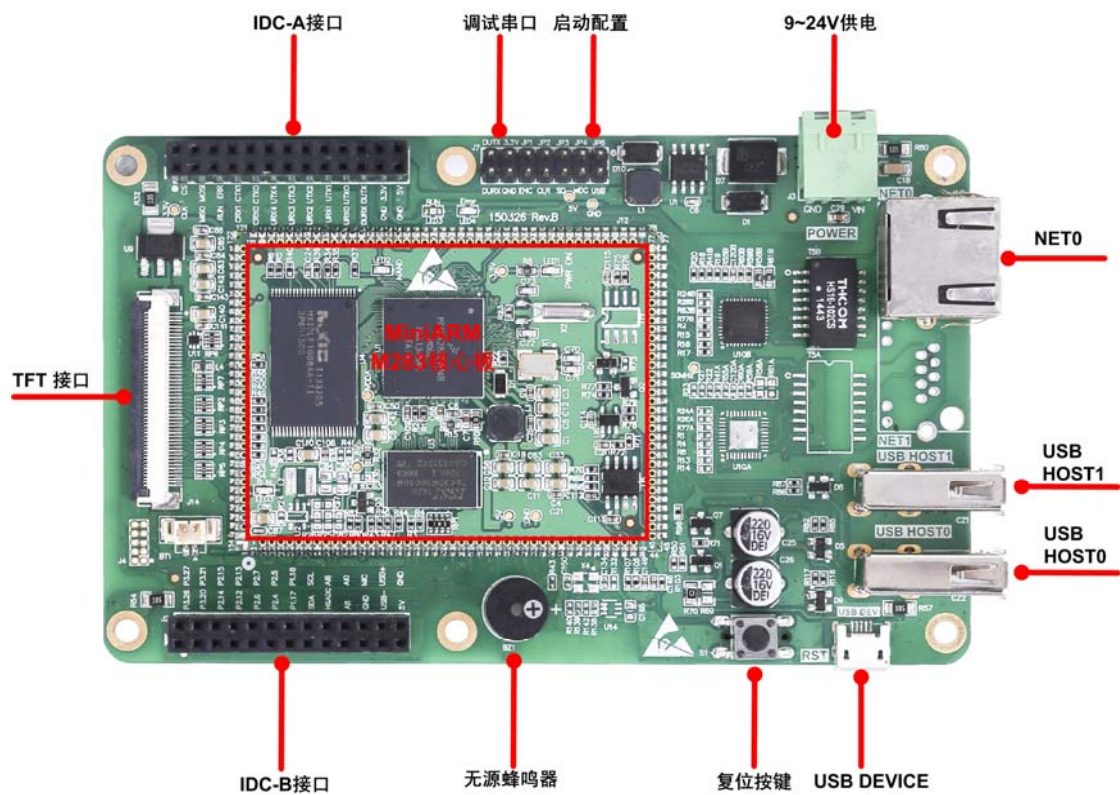


图 1.3 EPC-283C-L 功能框图

## 1.2 功能特点

- 处理器使用 Freescale MCIMX283/287，运行频率：454MHz；
- 内置 64/128MB DDR2；
- 内置 128MB 数据 NAND FLASH；
- 内置 TCP/IP 协议栈；
- 支持多种文件系统，支持 SD/MMC 卡、U 盘读写；
- 速率高达 480Mbps 的 USB HOST、USB OTG；
- 集成多达 3 路 ADC、1 路 I<sup>2</sup>C、1 路 SPI；
- 集成多达 6 路 UART 、多达 2 路 CAN、多达 2 路以太网接口；
- 集成 WDT 的复位监控电路；
- 支持多种显示分辨率，最高达 800×480 像素液晶接口；
- 支持多种升级方式；
- 6 层 PCB 工艺，尺寸 45mm×60mm；
- 低工作电压（M28x 系列）：5V±2%；
- 宽工作电压（EPC-28x 系列）：9~24V；
- 内置加密模块，保护用户程序安全；
- 可堆叠，使用我司的扩展接口。

## 1.3 硬件资源

表 1.1 M28x/EPC-28x 产品硬件资源

产品名称	M283-128(64)LI(C)	M287-128(64)LI(C)	EPC-283	EPC-287
处理器名字	MCIMX283	MCIMX287	MCIMX283	MCIMX287
主频 MHz	454	454	454	454
处理器核心	ARM926EJ-S™	ARM926EJ-S™	ARM926EJ-S™	ARM926EJ-S™
工作温度	工业级/商业级	工业级/商业级	商业级	商业级
内存	64/128MB	64/128 MB	128 MB	128 MB
NAND Flash	128MB			
LCD 接口 <sup>注1</sup>	高达 800×480			
触摸屏	四线电阻式			
以太网	1 路内置控制器	2 路内置控制器	1 路	2 路
USB 接口	1 路 USB Host/1 路 USB OTG	1 路 USB Host/1 路 USB OTG	2 路 <sup>注2</sup> USB Host/1 路 USB OTG	2 路 <sup>注2</sup> USB Host/1 路 USB OTG
串口 <sup>注3</sup>	4 路	4 路	4 路	6 路
SPI	1	2	1	2
I <sup>2</sup> C	1			
CAN	——	2	——	2
SDMMC	1			
A/D <sup>注4</sup>	3 路			
最多 GPIO	4 个	18 个	4 个	18 个



看门狗	外置独立看门狗
-----	---------

注 1: 800x480 是 LCD 接口的最大分辨率

注 2: 其中一路与 OTG 复用

注 3: 串口的数量 = 1 路调试串口 + n 路应用串口

注 4: ADC 的采样速率为 485kbps

## 1.4 产品命名

M28x/EPC-28x产品命名规则，如图 1.4图 1.5所示。

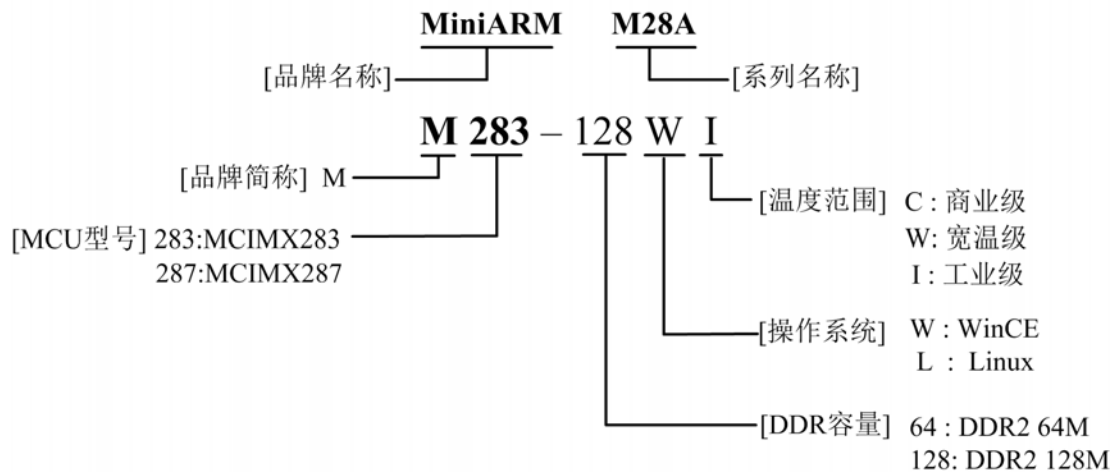


图 1.4 M28x 系列命名规则

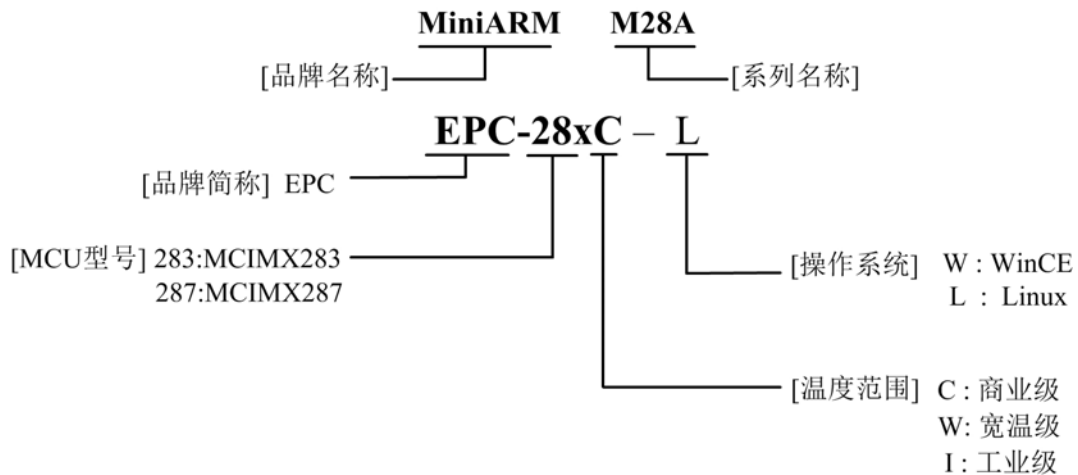


图 1.5 EPC-28x 系列命名规则

## 1.5 EPC-28xC-L驱动程序

本产品运行 Linux-2.6.35 内核，包含如下驱动程序：

- Nand Flash 驱动程序；
- 显示驱动程序；

- 触摸屏驱动程序；
- SD/MMC 卡驱动程序；
- USB Host 驱动程序，支持 USB 键盘、鼠标和 U 盘；
- CAN-Bus 驱动程序（Socket CAN 接口）；
- 10/100 自适应以太网驱动程序；
- RS-232C 接口驱动程序；
- RTC 驱动程序；
- GPIO 驱动程序；
- ADC 驱动程序；
- I2C 驱动程序；
- SPI 驱动程序；
- 蜂鸣器驱动程序；
- LED 驱动程序；
- SDIO WIFI 驱动程序；
- USB 转串口驱动程序；
- 看门狗驱动程序。

## 2 基本操作

本文是基于 EPC-28x 系列产品的系统操作说明。此处所述“系统”是指基于 EPC-28x 硬件并安装了本公司开发软件系统，即“安装了 Linux 系统的 EPC-28x”产品，如无特别说明，本文的操作也同样适用于 M28x 系列产品。

开机和登录

系统默认 IP: 192.168.1.136

系统默认登录用户名和密码均为: root

### 1.5.1 启动选择

本产品支持从NAND或者TF卡启动，可通过设置相应的外围电路进行选择。具体到评估套件对应的是底板上的 5 个跳线设置，位置如图 2.1红圈所示。



图 2.1 启动跳线设置

跳线设置具体描述如表 2.1所示。

表 2.1 核心板配置信号功能描述

标 号	说 明
WDT	短接时，禁止看门狗，断开时，开启看门狗监控
USB_BT	短接后，进入 USB 下载模式，配合 MFGTool 或 sb_loader 下载文件到 DDR 运行，如要从 NAND/SD 启动，必须断开
SD	USB_BT 断开时，SD 短接后，系统会从 TF 卡启动，断开时，系统从 NAND 启动
CLR	保留引脚，请保持断开
ENC	保留引脚，请保持断开

### 1.5.2 串口连接设置

将调试串口如图 2.2外接RS232 模块后与电脑串口相连，设置串口 115200-8N1，无流控。

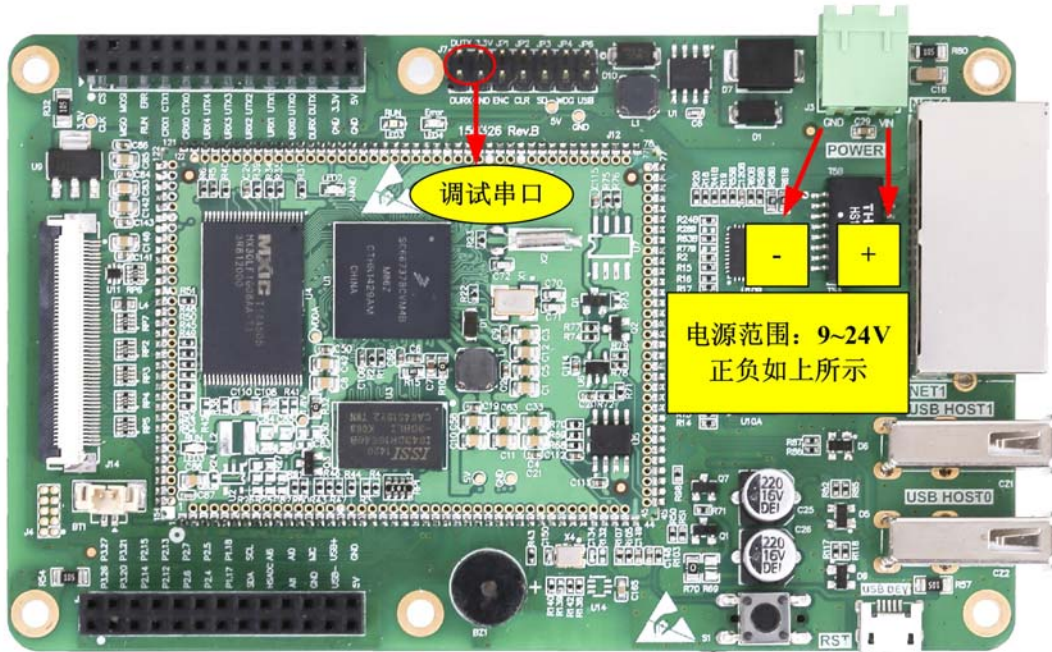


图 2.2 串口及电源配置

在Windows环境下，可以使用超级终端或者其它串口软件。下面以超级终端为例进行介绍。选择【开始】→【程序】→【附件】→【通讯】→【超级终端】，打开“新建连接”的界面，在“名称”一栏填写连接的名称如“Linux”，如图 2.3所示。



图 2.3 新建连接

点击“确定”，在如图 2.4所示的界面，选择正确的串口。

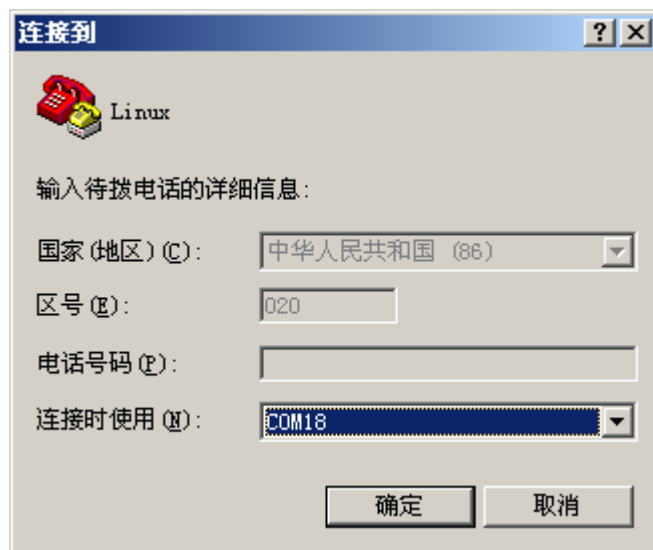


图 2.4 选择串口

点击“确定”进入串口属性设置界面，在这里设定串口的波特率、数据位等参数，具体设置为“115200，8N1，无流控制”，如图 2.5所示。



图 2.5 串口属性设置

确认设置无误后点击“确定”，将得到如图 2.6所示的超级终端界面。

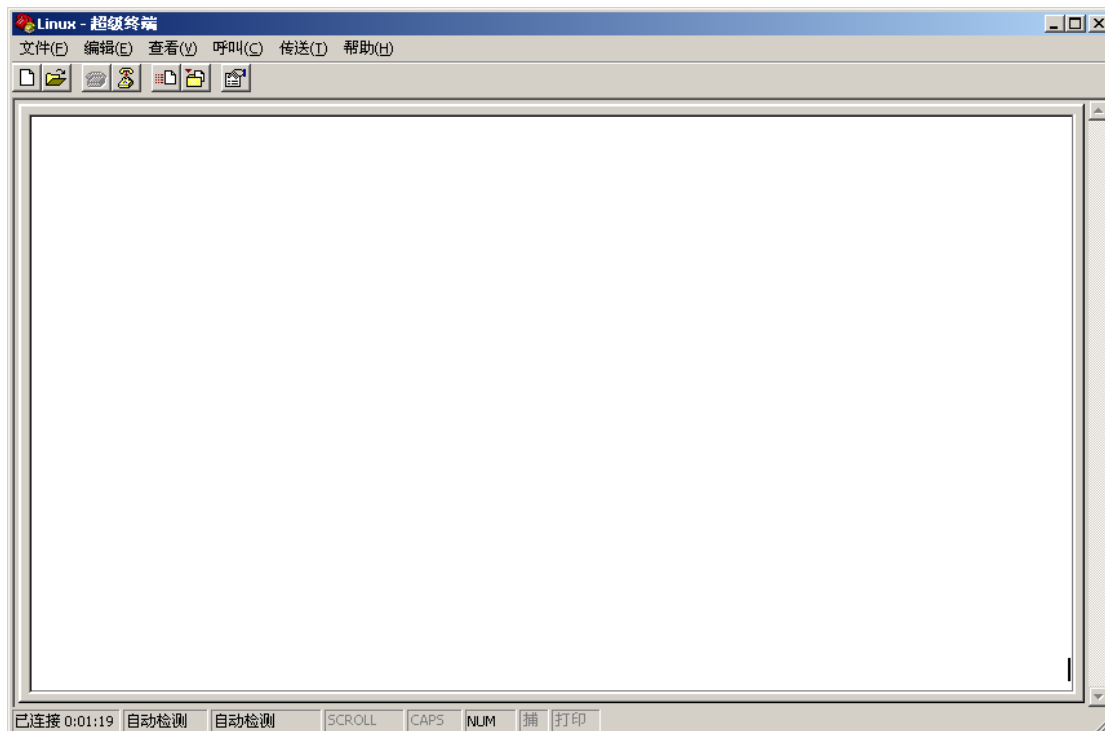


图 2.6 设定完成的超级终端界面

### 1.5.3 串口登录

接好串口打开串口软件，给EPC上电，在U-Boot阶段不要按任何按键，系统将启动并进入Linux系统，在超级终端可以看到启动信息。系统启动完毕，进入Linux Shell登录界面，如图 2.7所示。

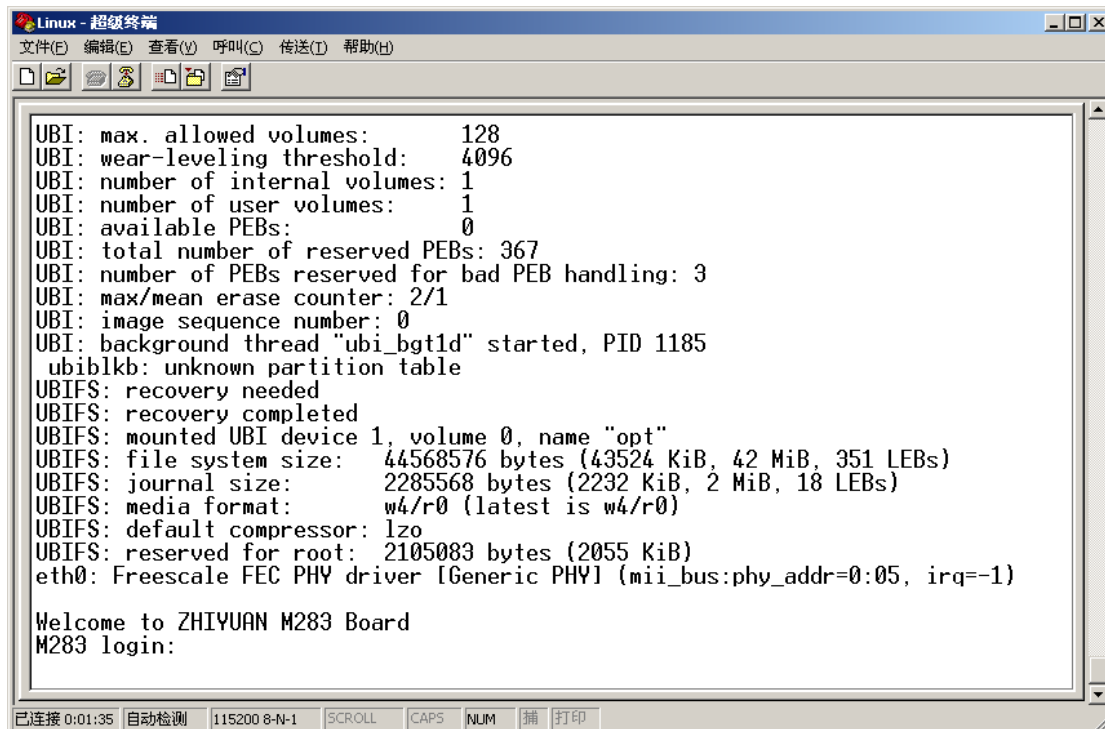


图 2.7 通过超级终端登录 Linux 系统



#### 1.5.4 SSH远程登录

本产品系统支持 SSH 远程登录，可以使用任何支持 ssh 协议的软件进行登录。下面以 Putty 软件通过 ssh 协议进行登录为例进行介绍。

本产品有 1 路以太网，默认 IP 为 192.168.1.136，简单起见，在保证网络通畅的情况下，进行 SSH 远程登录时，给 PC 主机设置或者添加一个与本产品同一网段的 IP 地址。

启动Putty软件，在“Host Name”栏填入评估套件的IP地址 192.168.1.136，选中SSH协议，如图 2.8所示。

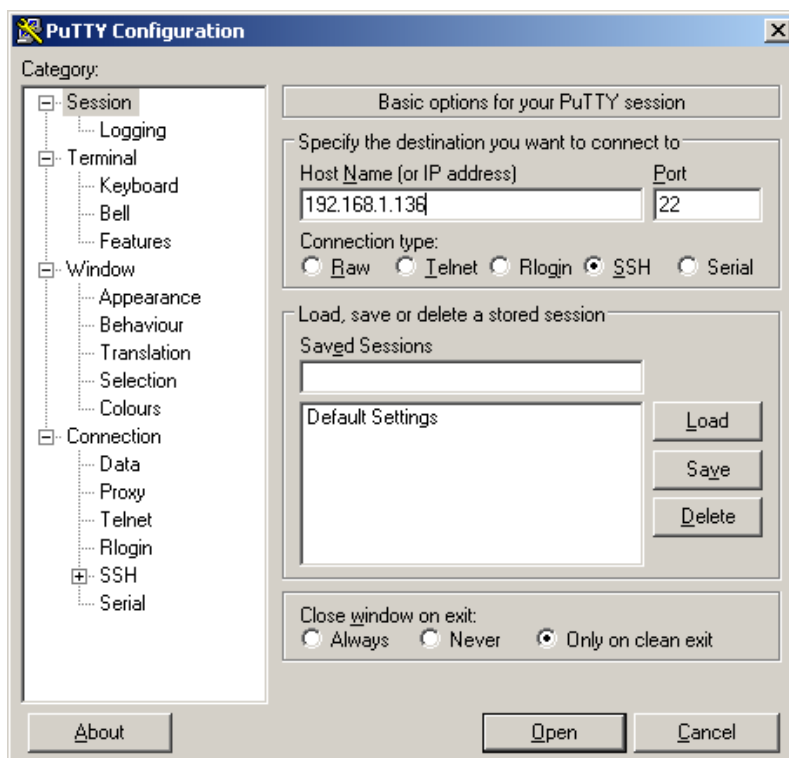


图 2.8 选中使用 SSH 协议

点击“Open”，在登录界面输入用户名和密码“root”，登录Linux系统，如图 2.9所示。

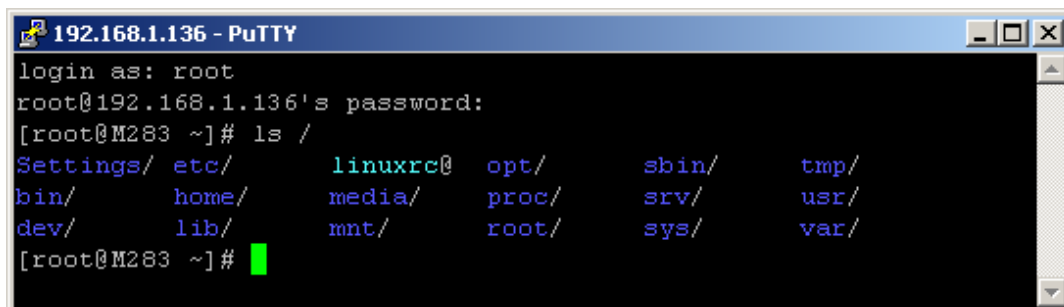


图 2.9 通过 SSH 登录 Linux 系统

## 1.6 关机

一般情况下直接关闭电源即可。如果有数据存储操作，为了确保数据完全写入，可输入 sync 命令，完成数据同步后关闭电源；或者输入“poweroff”命令，等串口终端出现“Power down.”提示后再关闭电源。

```
[root@M283 ~]# poweroff
[root@M283 ~]# Power down.
```

## 1.7 Qt演示程序

本产品支持 Qt 4.7.3，演示程序（需要外接液晶）的路径是：  
/usr/share/zhiyuan/zylauncher/start\_zylauncher。

## 1.8 输入设备

系统支持触摸屏输入，也支持 USB 键盘和鼠标等外接设备。

### 1.8.1 触摸屏和USB鼠标

如果在启动 Qt 演示程序之前没有插入 USB 鼠标，则默认使用触摸屏作为输入设备，通过触摸屏可进行相关操作。

如果插入了 USB 鼠标后启动 QT，则可同时使用触摸屏和 USB 进行操作。

### 1.8.2 USB键盘

插上 USB 键盘后启动演示界面，即可以使用 USB 键盘进行操作。

## 1.9 查看系统信息

查看系统内核版本，使用 `uname` 命令：

```
[root@M283 ~]# uname -a
Linux M283 2.6.35.3-571-gcca29a0-g6143692-dirty #117 PREEMPT Wed May 28 09:08:50 CST 2014 armv5tej
GNU/Linux
```

查看系统内存使用情况，使用 `free` 命令：

```
[root@M283 ~]# free
```

	total	used	free	shared	buffers
Mem:	124700	38428	86272	0	0
-/+ buffers:		38428	86272		
Swap:	0	0	0		

查看系统磁盘使用情况，使用 `df` 命令：

```
[root@M283 ~]# df
```

Filesystem	Size	Used	Available	Use%	Mounted on
ubi0:rootfs	54.4M	37.6M	16.8M	69%	/
tmpfs	60.9M	20.0K	60.9M	0%	/tmp
tmpfs	60.9M	52.0K	60.8M	0%	/var
tmpfs	60.9M	0	60.9M	0%	/media
ubi1:opt	38.6M	24.0K	36.6M	0%	/opt

查看 CPU 等其它信息：

```
[root@M283 ~] # cat /proc/cpuinfo
Processor       : ARM926EJ-S rev 5 (v5l)
BogoMIPS        : 226.09
Features        : swp half thumb fastmult edsp java
CPU implementer : 0x41
CPU architecture: 5TEJ
```

```
CPU variant      : 0x0
CPU part         : 0x926
CPU revision     : 5

Hardware         : Freescale MX28EVK board
Revision        : 0000
Serial          : 0000000000000000
```

更多的系统信息请使用相应的命令进行查看。

## 1.10 系统设置

### 1.10.1 网络设置

#### 1. IP地址

临时改变 IP 可以使用 ifconfig 命令进行，如：

```
[root@M283 ~]# ifconfig eth0 192.168.1.136
```

如果要永久改变网卡设置可把这条命令写在启动脚本内。

#### 2. MAC地址

本产品在出厂时为 MAC 分配了广州致远通过 IEEE 标准协会购买的合法的 MAC 地址（致远的 MAC 地址以 00-14-97 开头），贴在核心板的标签上。

MAC地址查询网址：<http://standards.ieee.org/develop/regauth/oui/public.html>。

查询范例：查询 00-14-97 开头的 MAC 地址所属厂商：

Search the Public OUI/'company\_id' Listing

Search for: 00-14-97

Search!

clear field

[Download a copy](#) of the OUI Public Listing (Updated daily)

查询结果如下：

Here are the results of your search through the public section of the IEEE Standards OUI database report for **00-14-97**:

<b>00-14-97</b> (hex)	ZHIYUAN Eletronics co.,ltd.
001497 (base 16)	ZHIYUAN Eletronics co.,ltd.
	2 Floor, NO.3 Building, Huangzhou Industrial Estate, Chebei Road,
	Tianhe
	Guangzhou Guangdong 510660
	CHINA

### 1.10.2 系统时钟

查看系统时间，使用 date 命令：

```
[root@M283 ~]# date
```

```
Thu Jan 1 00:46:05 UTC 1970
```

设置系统时钟，先使用 date 命令设置好时间，然后使用 hwclock -w 命令将时间写入硬件 RTC（如果硬件没有接外部 RTC 芯片，那么这条命令就会报错）。

本产品支持外扩 RTC 电路（EPC-28x 系列已支持，M28x 系列需要自己设计电路），该系统启动时将从处理器内部 RTC 获取系统的初始时间，在该系统底板 J4 位置未接电池的条件下，遇到复位或重新上电时，系统时间将会恢复为初始 RTC 时间。若需要保持系统时间，则需要在 J4 位置装上电压为 3.1~4.2V 的电池。

例如：设置硬件 RTC 时间为 2014-05-28, 10:11:25，则可以使用如下命令：

```
[root@M283 ~] # date 2014.05.28-10:11:25
Wed May 28 10:11:25 UTC 2014
[root@M283 ~]# hwclock -w
```

## 1.11 文件传输

### 1.11.1 SSH文件传输

本产品支持 SSH 远程登录,如果所使用的 SSH 软件带有文件传输功能,还可以通过 SSH 进行文件传输。下面以带文件传输的 SSH Secure File Transfer 为例进行介绍。

SSH Secure File Transfer的运行界面如图 2.10所示。

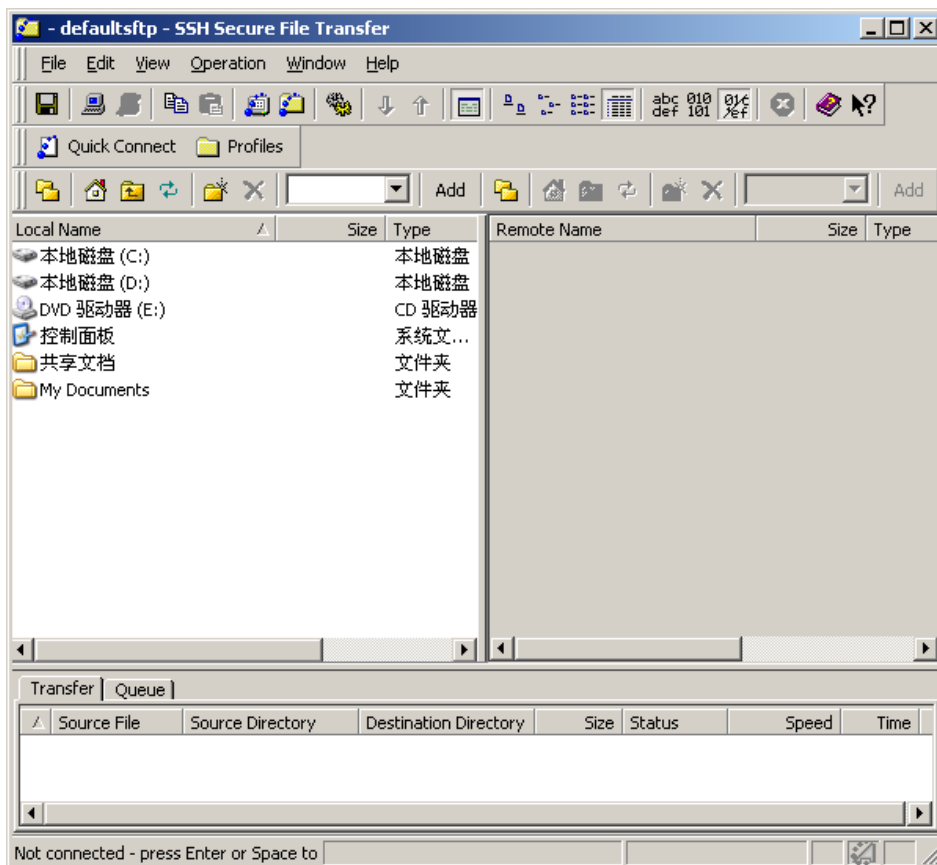


图 2.10 SSH Secure File Transfer 运行界面

点击界面的“Quick Connect”按钮,在远程连接界面相应栏中填入本产品的IP地址和登录所使用的用户名,如图 2.11所示。

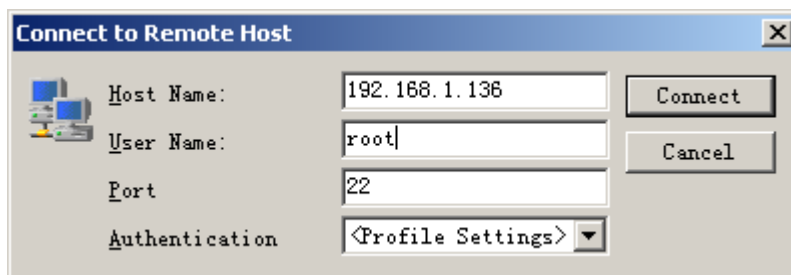


图 2.11 填写主机和用户名

最后点击“Connect”按钮，登录系统，并可进行文件传输。图 2.12所示为切换到系统根目录(/)下的视图。

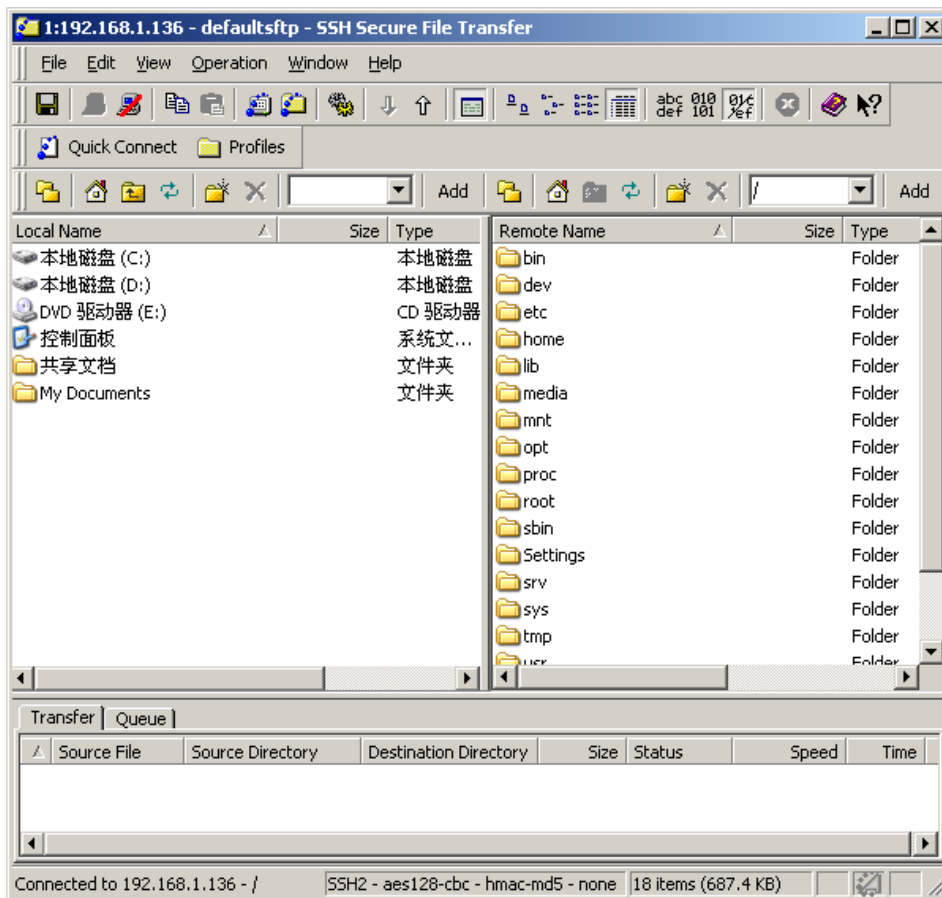


图 2.12 SSH 文件按浏览视图

通过 SSH Secure File Transfer 可以进行文件传输，由于系统保护机制，只有/opt 目录可写，所以进行文件传输需要先在 SSH 中将远程目录切换到/opt 目录。

### 1.11.2 NFS文件传输

评估套件只能作为 NFS 客户端，可以通过 NFS 方式登录到 NFS 服务器，进行文件传输。使用 mount 命令，格式如下：

```
[root@M283 ~]# mount -t nfs nfs-server-ip:nfs-share-directory /mountpoint -o nolock
```

例如：

```
[root@M283 ~]# mount -t nfs 192.168.1.138:/home/chenxibing/nfs /mnt -o nolock
```

NFS 挂载成功后，可以使用 cp 命令将 NFS 服务器的文件复制到系统本地目录。例如：

```
[root@M283 ~]# cp /mnt/hello/hello /opt
```

## 1.12 U盘使用

本产品带有 USB Host，可支持 U 盘、USB 读卡器或者 USB 接口的硬盘等移动存储设备。插入可用的 U 盘后，系统通常会自动挂载到/media 目录下，具体目录取决于 U 盘的分区情况，U 盘通常被识别为 SCSI 设备，设备名为/dev/sda%d 或者/dev/sdb%d，挂载为/media/sda%d 或者/media/sdb%d。

操作完毕，建议使用先 sync 命令完成同步，或者使用 umount 命令卸载 U 盘，确保数

据完全写入，再拔取 U 盘。

### 1.13 TF卡使用

插入 TF 卡，系统会进行 TF 卡自动挂载。一般会挂载在/media/mmc%d 目录下，%d 数值取决于卡的分区情况。

### 1.14 U-Boot交互

本产品采用 U-Boot 做为系统引导程序，在必要的情况下，可以进入 U-Boot，进行一些高级设置。

在 U-Boot 启动阶段，在串口终端按任意键（如空格键）即进入 U-Boot 的命令行模式（如果 U-Boot 未有等待延时，则需要在 U-Boot 启动阶段持续按下任意键才能进入 U-Boot 的命令行模式），可以输入已支持的命令对 U-Boot 进行配置。

```
U-Boot 1.3.3 (Feb 10 2009 - 10:09:52)
DRAM: 64 MB
NAND: 256 MB
In: serial
Out: serial
Err: serial
Hit any key to stop autoboot: 0
MX28 U-Boot >
```

在 U-Boot >提示符下，输入?或者 help 可以查看 U-Boot 所支持的全部命令以及介绍。

```
MX28 U-Boot > ?
? - alias for 'help'
autoscr - DEPRECATED - use "source" command instead
base - print or set address offset
bdinfo - print Board Info structure
boot - boot default, i.e., run 'bootcmd'
bootd - boot default, i.e., run 'bootcmd'
bootm - boot application image from memory
bootp - boot image via network using BOOTP/TFTP protocol
chpart - change active partition
cmp - memory compare
coninfo - print console devices and information
cp - memory copy
crc32 - checksum calculation
dhcp - boot image via network using DHCP/TFTP protocol
echo - echo args to console
fatinfo - print information about filesystem
fatload - load binary file from a dos filesystem
fatls - list files in a directory (default /)
go - start application at address 'addr'
help - print online help
iminfo - print header information for application image
imxtract - extract a part of a multi-image
```



itest	- return true/false on integer compare
loadb	- load binary file over serial line (kermit mode)
loads	- load S -Record file over serial line
loady	- load binary file over serial line (ymodem mode)
loop	- infinite loop on address range
md	- memory display
mii	- MII utility commands
mm	- memory modify (auto-incrementing address)
mmc	- MMC sub system
mmcinfo	- mmcinfo <dev num>-- display MMC info
mtdparts	- define flash/nand partitions
mtest	- simple RAM read/write test
mw	- memory write (fill)
mxs_mmc	- MXS specific MMC sub system
nand	- NAND sub-system
nboot	- boot from NAND device
nfs	- boot image via network using NFS protocol
nm	- memory modify (constant address)
ping	- send ICMP ECHO_REQUEST to network host
printenv	- print environment variables
rarpboot	- boot image via network using RARP/TFTP protocol
reset	- Perform RESET of the CPU
run	- run commands in an environment variable
saveenv	- save environment variables to persistent storage
setenv	- set environment variables
sleep	- delay execution for some time
source	- run script from memory
tftpboot	- boot image via network using TFTP protocol
ubi	- ubi commands
ubifsload	- load file from an UBIFS filesystem
ubifsls	- list files in a directory
ubifsmount	- mount UBIFS volume
version	- print monitor version

其中 NAND Flash 操作另有一系列命令，可使用 `help nand` 查看。

MX28 U-Boot > help nand

nand - NAND sub-system

Usage:

nand info	- show available NAND devices
nand device [dev]	- show or set current device
nand read	- addr off partition size
nand write	- addr off partition size
read/write 'size' bytes starting at offset 'off'	
to/from memory address 'addr', skipping bad blocks.	

```
nand erase [clean] [off size] - erase 'size' bytes from
                             offset 'off' (entire device if not specified)
nand bad - show bad blocks
nand dump[.oob] off - dump page
nand scrub - really clean NAND erasing bad blocks (UNSAFE)
nand markbad off [...] - mark bad block(s) at offset (UNSAFE)
nand biterr off - make a bit error at offset (UNSAFE)
```

#### 1.14.1 预设的组合命令

使用 U-Boot 提供的基本命令，可以完成对系统的操作，但是如果每次操作都输入一系列命令，既繁琐，也有可能出错。为此，光盘中的 U-Boot 预设了一些组合命令，可以方便快速的实现系统固化、升级更新等操作。

这些组合命令预存于 U-Boot 的环境变量中，进入 U-Boot、输入 `printenv` 可以查看已经预设的组合命令。输入“run 组合命令”即可运行这些组合命令。

更新内核，系统预设了一条 `upkernel` 的命令，能够完成从 tftp 服务器加载 uImage 内核文件，并完成相应 NAND Flash 擦除、烧写内核以及设置内核参数的工作，命令如下：

```
upkernel= tftp $(loadaddr) $(serverip):$(kernel);nand erase clean $(kerneladdr) $(kernelsize);nand write.jffs2
$(loadaddr) $(kerneladdr) $(kernelsize);
```

更新文件系统，系统预设了一条 `uprootfs` 的命令，能够完成从 tftp 服务器加载 rootfs.ubifs 文件，并完成 NAND Flash 擦除和文件烧写的工作，命令如下：

```
uprootfs= mtdparts default;nand erase rootfs;ubi part rootfs;ubi create rootfs;tftp $(loadaddr) $(rootfs);ubi write
$(loadaddr) rootfs $(filesize)
```

从 NAND Flash 加载内核并启动的预设命令是 `nand_boot`：

```
nand_boot=nand read.jffs2 $(loadaddr) $(kerneladdr) $(kernelsize);bootm $(loadaddr)
```

注意：当前发布的 U-Boot 不支持 `run upuboot` 命令，强制执行该命令将会存储的固件数据造成破坏。

### 1.15 内核编译

本章主要介绍了 Linux 内核源码的编译、配置过程。

本章所引用的源码位于光盘目录“7.源代码”下。

工控内核和开发套件内核使用的是同一个内核源码，只是配置选项不同而已，以下介绍工控内核和开发套件内核的配置过程。

#### 1.15.1 解压内核文件

请把光盘中的“linux-2.6.35.3.tar.bz2”复制到 Linux 主机硬盘的工作目录，然后解压该压缩包：

```
$ tar -jxvf linux-2.6.35.3.tar.bz2
```

解压完成之后得到“linux-2.6.35.3”目录，运行以下命令，进入该目录：

```
$ cd linux-2.6.35.3
```

#### 1.15.2 内核配置

Linux 内核源码具有高可配置性。用户可以根据自己的需要对内核进行裁减或添加自己所需要的驱动。

输入make menuconfig命令即可打开内核的配置界面如图 2.13所示。

\$ make menuconfig

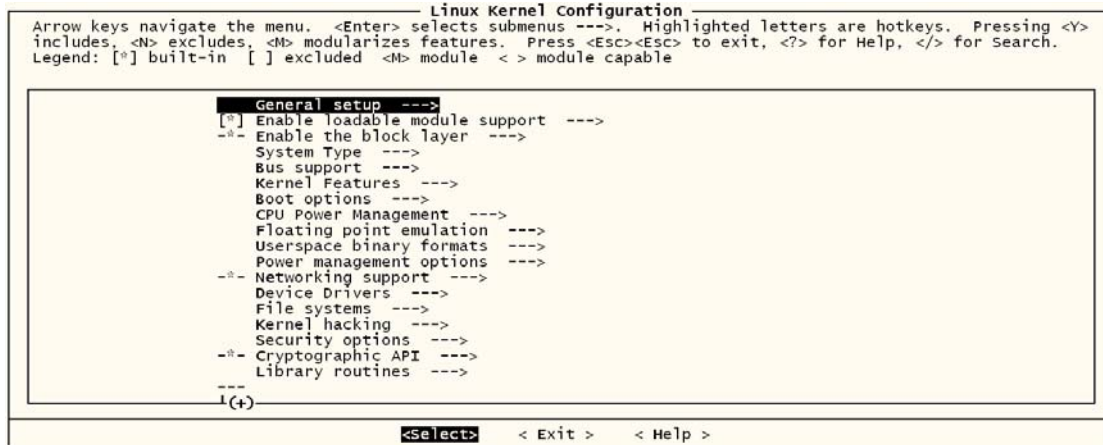


图 2.13 内核配置界面

源码目录中，工控的部分代码是不开源的，如果使用工控内核，需要将此类选项配置。配置过程如下：

选择“System Type --->”，按enter进入,如图 2.14所示。



图 2.14 内核配置

选择“Freescale i.MXS implementations --->”，按 enter 进入,如所示。



图 2.15 配置内核

下面有 5 个选项：

[\*] Convert to source for mmc module.

[\*] Convert to source for IIC module.

[\*] Convert to source for fec(net) module.

[\*] Convert to source for can module.

[\*] Convert to source for nandflash module.

将光标移至这些选项上，按空格键分别将选项去掉。如果想使用学习套件内核功能，将这些选项选上即可。配置完成后保存即可。

### 1.15.3 编译内核

在“linux-2.6.35.3”目录下执行“make uImage”命令即可编译。编译完成后将在 arch/arm/boot/目录下生成 uImage 内核固件文件。

## 1.16 LCD待机设置

LCD 默认待机时间为 10 分钟。如果需要退出待机模式，可进行如下操作：

```
[root@M283 ~]# echo "0" > /sys/class/graphics/fb0/blank
```

在非待机状态下想立即进入待机模式，可进行如下操作：

```
[root@M283 ~]# echo "4" > /sys/class/graphics/fb0/blank
```

如果需要永久不进入待机模式，则需要修改内核启动参数，进入 U-Boot，在内核启动参数中增加：consoleblank=0 即可，例如：

```
bootargs='gpmi=g console=ttyAM0,115200n8 ubi.mtd=5 root=ubi0:rootfs rootfstype=ubifs consoleblank=0'
```

实际操作方法。系统启动后，进入 U-Boot 命令行。输入如下指令：

```
MX28 U-Boot> setenv bootargs 'gpmi=g console=ttyAM0,115200n8 ubi.mtd=5 root=ubi0:rootfs rootfstype=ubifs consoleblank=0'
```

```
MX28 U-Boot> saveenv
```

然后重启即可。

## 1.17 LCD背光调节

本产品支持 PWM 背光调节，可根据实际需要进行调整。操作接口为 /sys/class/backlight/mxs-bl/brightness。往 brightness 文件写入背光控制的值（PWM 占空比，有效范围 0~100），即可实现调节。例如，将亮度设置为 70%：

```
[root@M283 ~]# echo 70 > /sys/class/backlight/mxs-bl/brightness
```

## 1.18 串口测试

本产品提供了多达 5 个 TTL 电平的应用串口 UART0、UART1、UART2、UART3、UART4（不包括 RS232 接口的 Debug UART），如图 2.16 所示。

底板上的 UART2 和 UART3 所对应的硬件功能引脚，283 产品已复用为其他功能，所以在该产品中，/dev/ttySP2、/dev/ttySP3 设备是不可用的；287 产品则启用了这 2 个串口，所以 /dev/ttySP2、/dev/ttySP3 是可用的。

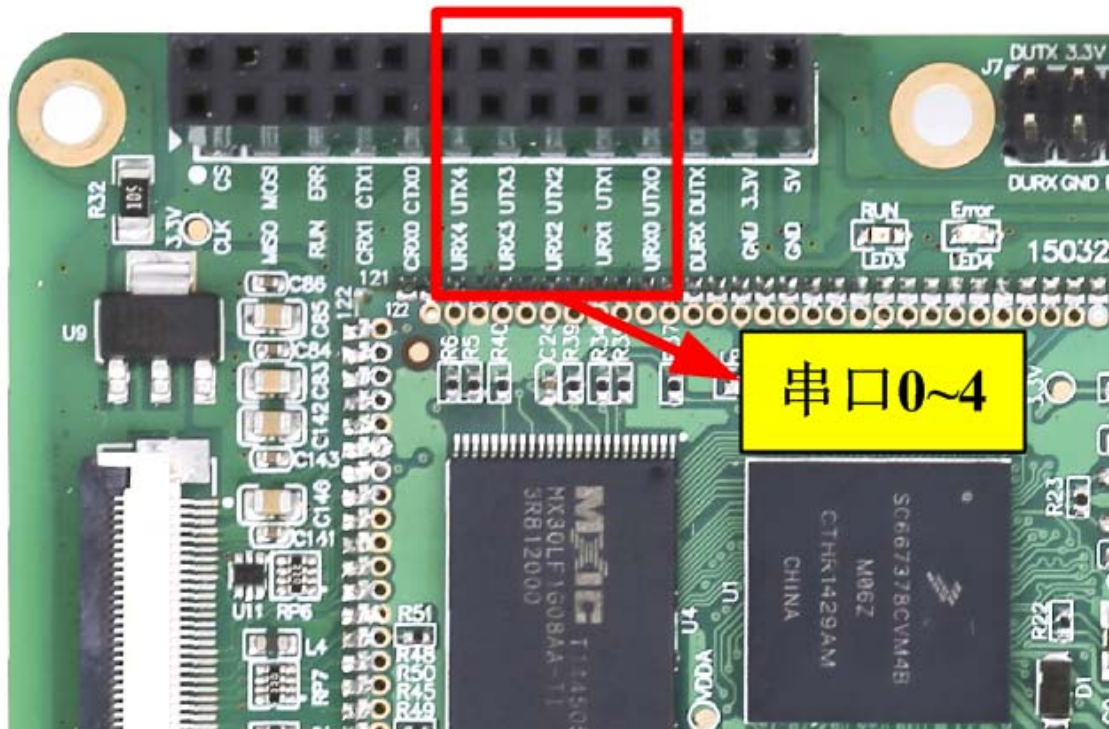


图 2.17 产品可用的应用串口

系统自带一个串口测试软件 `microcom`，可以很简便快捷的测试串口。评估系统中所提供的 `microcom` 的用法如下：

Usage: `microcom [-d DELAY] [-t TIMEOUT] [-s SPEED] [-X] TTY`

例如，需要测试串口 `ttySP1`，波特率为 38400。将本产品的 `ttySP1` 接上 TTL-RS232 转换模块后与 PC 的某个串口相连，PC 上打开串口软件。在本产品的终端输入如下命令：

```
[root@M283 ~]# microcom -t 3000 -s 38400 /dev/ttySP1
```

然后在终端输入字符，在 PC 的串口软件可以看到所发送的字符，反之亦然。`-t 3000` 表示在 3000ms 内没有输入，程序就自动退出。



### 3 文件系统

#### 1.19 分区描述

系统板载 128MB 字节 NAND Flash，一共分为 7 个 MTD 分区，查看 /proc/mtd 文件可看到各 MTD 分区信息：

```
[root@M283 ~] # cat /proc/mtd
dev:   size   erasesize  name
mtd0: 00c00000 00020000 "reserve"
mtd1: 00080000 00020000 "reserve"
mtd2: 00080000 00020000 "reserve"
mtd3: 00080000 00020000 "reserve"
mtd4: 00080000 00020000 "reserve"
mtd5: 04000000 00020000 "rootfs"
mtd6: 02e00000 00020000 "opt"
```

各分区的大小和用途等如表 3.1 所列。

表 3.1 NAND Flash 分区和说明

分区	大小	用途
mtd0	12MB	Linux 内核 1 & Linux 内核（备份）
mtd1	512KB	U-Boot
mtd2	512KB	保留分区
mtd3	2MB	启动画面
mtd4	512KB	保留分区
mtd5	64MB	用户文件系统区域
mtd6	剩余空间	/opt 分区，可存放用户数据或者程序

建议用户将应用程序或者程序数据存放在 /opt 分区，而不要放在用户文件系统中，以免对文件系统造成破坏。

#### 1.20 支持的文件系统

系统支持多种常见文件系统，如 ext2、fat 等，具体支持哪些文件系统，可通过查看 /proc/filesystem 文件。

```
[root@M283 ~] # cat /proc/filesystems
nodev   sysfs
nodev   rootfs
nodev   bdev
nodev   proc
nodev   tmpfs
nodev   binfmt_misc
nodev   debugfs
nodev   sockfs
nodev   pipefs
nodev   anon_inodefs
nodev   rpc_pipefs
```



```
nodev devpts
      ext3
      ext2
nodev ramfs
      vfat
      msdos
      iso9660
nodev nfs
nodev nfs4
nodev mqueue
nodev selinuxfs
nodev mtd_inodefs
nodev ubifs
```

## 1.21 安装第三方软件

用户程序须安装在/opt 目录下，前面已有提及，只有/opt 目录是用户可读写的，在/opt 目录下建立合理的程序目录结构，如：

```
/opt/myapp/
|-- bin          #存放用户程序可执行文件
|-- data         #存放数据文件
|-- doc          #存放文档文件
|-- etc          #存放配置等文件
|-- lib          #存放程序所需要的额外的库文件
`-- share        #存放共享文件
```

/opt 在物理上是一个单独的 MTD 分区，专门用于安装用户程序和存放用户数据。

## 1.22 程序开机自启动

系统启动过程中会扫描/etc/init.d 目录下所有以“S”开头的文件并启动。所以只需在/etc/init.d 目录下编写一个文件名为：“S+编号+名称”的可执行脚本即可，在脚本中增加启动某个具体程序的语句即可。文件名必须以“S”开头，编号代表了启动级别，越大越晚运行，取值建议在 90~99 之间。

例如，需要开机自启动的应用程序为/opt/myapp/bin/myapp，可编写 S90myapp 脚本并增加可执行权限，放到/etc/init.d 目录下，文件内容可简单的写为：

```
#!/bin/sh
/opt/myapp/bin/myapp &
```

实际应用中，如果程序启动之前需要设置一些环境变量，或者进行其他初始化，或者加载某些外设驱动或者库等等，最好编写一个独立的程序启动脚本来完成以上工作，如启动程序的脚本为 startmyapp，内容如下（注意，脚本需要可执行权限）：

```
#!/bin/sh
insmod xxx.ko
export XXX=
export YYY=
/opt/myapp/bin/myapp
```

实现开机自启动只需在/etc/init.d/S90myapp 中调用这个脚本即可：

```
#!/bin/sh
/opt/myapp/startmyapp &
```

例如，要实现开机启动 QT 演示程序，可在/etc/init.d 目录下增加一个 S90qt 文件，其中内容如下：

```
#!/bin/sh
#start qt command, you can delete it
export TSLIB_PLUGINDIR=/usr/lib/ts/
export TSLIB_CONFFILE=/etc/ts.conf
export TSLIB_TSDEVICE=/dev/input/ts0
export TSLIB_CALIBFILE=/etc/pointercal
export QT_QWS_FONTDIR=/usr/lib/fonts
export QWS_MOUSE_PROTO=Tslib:/dev/input/ts0
/usr/share/zhiyuan/zylauncher/start_zylauncher &
```

### 1.22.1 应用程序的自动升级

有的时候应用程序需要升级，需求是：把用户的应用程序拷贝到 TF 卡内，然后插入开发板内，拷贝到开发板内并替换掉老的应用程序，完成这些步骤比较繁琐，不过，可以用一个脚本来实现这些操作，内容如下：

```
#!/bin/sh
targe_sd_file=""

isexit=`cat /proc/partitions | grep mmcblk0 | wc -l`
if [ $isexit == '0' ];
then
    echo "no sd card detected "
    exit 1
fi

sd_dri=`df -h | awk '/dev/mmcblk0/' | sed -n '1p' | awk '{print $6}'`
echo "sd cart:"
echo $sd_dri
cd $sd_dri

isexit=`ls | grep pana | wc -l`
if [ $isexit == '0' ];
then
    echo "no user data file"
    exit 1
fi

cd /opt/
echo "updater the user program start"
tar -vxf $sd_dri/pana*
sync
```

```
echo "updater the user program end"
```

上面这段脚本：先检查 TF 卡是否存在，如果存在就检查 TF 卡内是否存在文件 pana\*，如果存在文件 pana\*，就运行 tar 命令把 pana\*解压到/opt 目录内。把这段脚本保存为 S31Updater，添加上执行属性，并放置在/etc/init.d 目录内，就实现了应用程序的自动升级。在 PC 机上用命令“tar -vcf pana.tar \*”打包应用程序，然后拷贝到 TF 卡内，插入开发板，然后上电开发板，系统开机后就运行脚本 S31Updater，执行自动解压的操作，实现自动升级。

当升级成功时会打印：

```
sd card:
/media/mmcblk0p1
updater the user program start
start_userapp
updater the user program end
```

### 1.23 修改文件系统

出厂的文件系统，除了/opt 和临时目录/tmp 可写之外，其余目录都是只读系统，防止系统受到意外损坏。但是系统不可避免的需要进行某些修改，系统提供了这样的操作机制。在进行文件复制、修改等命令前加上 wr，即可修改系统的只读目录和文件，例如：

```
[root@M283 ~]# wr cp S90qt /etc/init.d
```

这样就可以将 S90qt 文件复制到只读的/etc/init.d 目录。

要编辑修改某个文件，可在 vi 前加上 wr，实现对文件的修改。例如要修改/etc/inittab 文件，则可以这样操作：

```
[root@M283 ~]# wr vi /etc/inittab
```

## 4 应用程序开发

### 1.24 应用程序开发环境构建

#### 1.24.1 嵌入式Linux开发一般方法

嵌入式Linux系统，由于系统资源的匮乏，通常无法安装本地编译器进行本地开发，而需要在借助一台主机进行交叉开发。一般情况下，主机运行Linux操作系统，在主机安装相应的交叉编译器，将在主机编辑好的程序交叉编译后，通过一定方式如以太网或者串口将程序下载到目标系统运行，或者进行调试。一般的交叉开发流程如图 4.1所示。

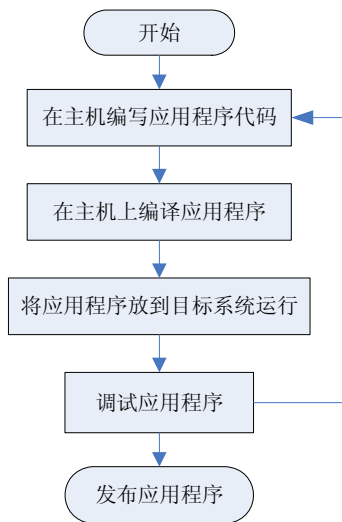


图 4.1 嵌入式 Linux 交叉开发一般流程

嵌入式Linux开发的一般模型如图 4.2所示。通常需要一台PC主机，在其中安装好各种进行交叉编译所需要的软件，通过串口和以太网和目标板相连。在主机上进行程序编辑和编译，得到的可执行文件通过串口或者以太网下载到目标板中运行或者进行调试。

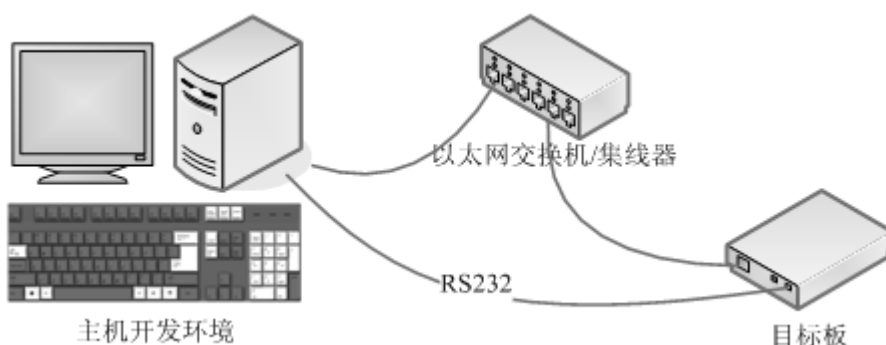


图 4.2 嵌入式 Linux 开发模型

进行嵌入式 Linux 开发，NFS（Network FileSystem）方式应该是最常用的开发方式了。主机开启 NFS 服务，作为 NFS 服务器，目标板作为 NFS 客户端，目标板通过 NFS 方式，将主机 NFS 服务器目录挂载到本地系统，像操作本地文件一样操作远程机器的文件。

对主机的要求，硬件方面，要求有串口和网口；软方面，操作系统推荐使用主流发行版，如 ubuntu 等，还需要安装开发相关的软件，同时还需要其它的软件如交叉编译器等。

## 1.24.2 安装主机操作系统

按照一般方法安装即可，操作系统强烈推荐使用 ubuntu-12.04 以上版本的 64 位发行版。

## 1.24.3 构建交叉开发环境

### 1. 工具链和安装

光盘资料中包含已经建好的交叉编译工具链，在光盘的“cross-tools”目录下，具体的文件名为“gcc-4.4.4-glibc-2.11.1-multilib-1.0\_EasyARM-iMX283.tar.bz2”。交叉编译工具可以通过 U 盘的方式，也可使用 SSH Secure Shell Client 通过 SSH 的方式拷贝到 Linux 主机。

在安装交叉编译工具之前需要先安装 32 位的兼容库和 libncurses5-dev 库，安装兼容库需要从 ubuntu 的源库中下载，所以主机必须能够上外网，使用如下命令安装：

```
[chenxibing@localhost ~]$sudo apt-get install ia32-libs
```

主机没有安装 32 位兼容库，在使用交叉编译工具的时候可能会出现错误：

```
-bash: ./arm-fsl-linux-gnueabi-gcc: 没有那个文件或目录
```

安装 libncurses5-dev，使用如下命令进行安装：

```
sudo apt-get install libncurses5-dev
```

如果没有安装此库，在使用 make menuconfig 时会如所示的错误：

```
*** Unable to find the ncurses libraries or the
*** required header files.
*** 'make menuconfig' requires the ncurses libraries.
***
*** Install ncurses (ncurses-devel) and try again.
***
make[1]: *** [scripts/kconfig/dochecklxdialog] 错误 1
make: *** [menuconfig] 错误 2
```

安装交叉编译工具链需要 root 权限。在终端执行命令：

```
[chenxibing@localhost ~]$ sudo tar -jxvf gcc-4.4.4-glibc-2.11.1-multilib-1.0_EasyARM-iMX283.tar.bz2 -C /opt/
```

交叉编译工具链将会被安装到/opt/ gcc-4.4.4-glibc-2.11.1-multilib-1.0 目录下（注意解压时必须指定解压的目录为 /opt/ 目录），交叉编译器的具体目录是 /opt/gcc-4.4.4-glibc-2.11.1-multilib-1.0/arm-fsl-linux-gnueabi/bin。为了方便使用，还需将交叉编译器路径添加到系统路径中，修改 ~/.bashrc 文件，在 PATH 变量中增加交叉编译工具链的安装路径，然后运行 ~/.bashrc 文件，使设置生效。在 ~/.bashrc 文件末尾增加一行：

```
export PATH=$PATH:/opt/gcc-4.4.4-glibc-2.11.1-multilib-1.0/arm-fsl-linux-gnueabi/bin/
```

运行 .bashrc 文件的方法，进入 ~/ 目录，输入 .bashrc 命令（点+空格.bashrc）。

在终端输入 arm-fsl-linux-gnueabi-并按 TAB 键，如果能够看到很多 arm-fsl-linux-gnueabi-前缀的命令，则基本可以确定交叉编译器安装正确。

```
[chenxibing@localhost ~]$ arm-none-linux-gnueabi-
```

arm2hpd1	arm-linux-addr2line	arm-none-linux-gnueabi-ar
arm-fsl-linux-gnueabi-addr2line	arm-linux-ar	arm-none-linux-gnueabi-as
arm-fsl-linux-gnueabi-ar	arm-linux-as	arm-none-linux-gnueabi-c++
arm-fsl-linux-gnueabi-as	arm-linux-c++	arm-none-linux-gnueabi-cc
arm-fsl-linux-gnueabi-c++	arm-linux-cc	arm-none-linux-gnueabi-c++filt
arm-fsl-linux-gnueabi-cc	arm-linux-c++filt	arm-none-linux-gnueabi-cpp

arm-fsl-linux-gnueabi-c++filt	arm-linux-cpp	arm-none-linux-gnueabi-ct-ng.config
arm-fsl-linux-gnueabi-cpp	arm-linux-ct-ng.config	arm-none-linux-gnueabi-g++
arm-fsl-linux-gnueabi-ct-ng.config	arm-linux-g++	arm-none-linux-gnueabi-gcc
arm-fsl-linux-gnueabi-g++	arm-linux-gcc	arm-none-linux-gnueabi-gcc-4.4.4
arm-fsl-linux-gnueabi-gcc	arm-linux-gcc-4.4.4	arm-none-linux-gnueabi-gccbug
arm-fsl-linux-gnueabi-gcc-4.4.4	arm-linux-gccbug	arm-none-linux-gnueabi-gcov
arm-fsl-linux-gnueabi-gccbug	arm-linux-gcov	arm-none-linux-gnueabi-gdb
arm-fsl-linux-gnueabi-gcov	arm-linux-gdb	arm-none-linux-gnueabi-gprof
arm-fsl-linux-gnueabi-gdb	arm-linux-gprof	arm-none-linux-gnueabi-ld
arm-fsl-linux-gnueabi-gprof	arm-linux-ld	arm-none-linux-gnueabi-ldd
arm-fsl-linux-gnueabi-ld	arm-linux-ldd	arm-none-linux-gnueabi-nm
arm-fsl-linux-gnueabi-ldd	arm-linux-nm	arm-none-linux-gnueabi-objcopy
arm-fsl-linux-gnueabi-nm	arm-linux-objcopy	arm-none-linux-gnueabi-objdump
arm-fsl-linux-gnueabi-objcopy	arm-linux-objdump	arm-none-linux-gnueabi-populate
arm-fsl-linux-gnueabi-objdump	arm-linux-populate	arm-none-linux-gnueabi-ranlib
arm-fsl-linux-gnueabi-populate	arm-linux-ranlib	arm-none-linux-gnueabi-readelf
arm-fsl-linux-gnueabi-ranlib	arm-linux-readelf	arm-none-linux-gnueabi-run
arm-fsl-linux-gnueabi-readelf	arm-linux-run	arm-none-linux-gnueabi-size
arm-fsl-linux-gnueabi-run	arm-linux-size	arm-none-linux-gnueabi-strings
arm-fsl-linux-gnueabi-size	arm-linux-strings	arm-none-linux-gnueabi-strip
arm-fsl-linux-gnueabi-strings	arm-linux-strip	
arm-fsl-linux-gnueabi-strip	arm-none-linux-gnueabi-addr2line	

## 2. 测试工具链

编写一个简单的应用程序文件如 `hello.c`，然后在终端输入 `arm-fsl-linux-gnueabi-gcc hello.c -o hello`，编译 `hello.c`，得到 `hello` 程序后，使用 `file` 命令查看其格式。

```
[chenxibing@localhost hello]$ arm-fsl-linux-gnueabi-gcc hello.c -o hello
[chenxibing@localhost hello]$ file hello
imx_adc_test: ELF 32-bit LSB executable, ARM, version 1 (SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.14, not stripped
```

如果得到如上信息，可知 `hello` 程序是 ARM 格式的文件，`arm-none-linux-gnueabi`-工具链已经可以正常使用了。

### 1.24.4 NFS服务器配置

NFS 即网络文件系统（Network File-System），可以通过网络，让不同机器、不同系统之间可以实现文件共享。通过 NFS，可以访问远程共享目录，就像访问本地磁盘一样。NFS 只是一种文件系统，本身并没有传输功能，是基于 RPC（远程过程调用）协议实现的，采用 C/S 架构。

嵌入式 Linux 开发中，通常需要在主机上配置 NFS 服务器，将某系统特定目录共享给目标系统访问和使用。通过 NFS，目标系统可以直接运行存放于主机上的程序，可以减少对目标系统 FLASH 的烧写，既减少了对 FLASH 损害，同时也节省了烧写 FLASH 所花费的时间。

#### 1. 添加NFS目录

修改 `/etc/exports` 文件，在其中增加 NFS 目录（需要 root 权限，请使用 `sudo` 命令）并



指定访问主机的 IP 以及访问权限。

```
chenxibing@linux-compiler: ~$ sudo vi /etc/exports
[sudo] password for chenxibing:
```

如增加/home/chenxibing/nfs 目录，并允许 IP 为 192.168.1.\*的任何系统进行 NFS 访问，增加内容如下：

```
/home/chenxibing/nfs      192.168.1.*(rw,sync,no_root_squash)
```

或者允许任何 IP 访问，则增加内容如下：

```
/home/chenxibing/nfs      *(rw,sync,no_root_squash)
```

## 2. 启动NFS服务

同样需要 root 权限，执行 `sudo /etc/init.d/nfs-kernel-server start` 或者 `restart` 命令，可以启动或者重新启动 NFS 服务：

```
chenxibing@linux-compiler: ~$ sudo /etc/init.d/nfs-kernel-server start
```

在 NFS 服务已经启动的情况下，如果修改/etc/exports 了文件，可以重启 NFS 服务，刷新 NFS 共享目录，或者输入 `exports -rv` 命令重新导出 NFS 共享目录。

```
chenxibing@linux-compiler: ~$ sudo exportfs -rv
```

## 3. 测试NFS服务器

首先可以在主机上进行自测，将已经设定好的 NFS 共享目录 mount 到另外一个目录下，看能否成功。假定主机 IP 为 192.168.1.138，NFS 共享目录为/home/chenxibing/nfs，可使用如下命令进行测试：

```
chenxibing@linux-compiler: ~$ sudo mount -t nfs 192.168.1.138:/home/chenxibing/nfs /mnt
```

如果指令运行没有出出错，则 NFS 挂载成功，在/mnt 目录下应该可以看到/home/chenxibing/nfs 目录下的内容。

启动评估套件并进入 Linux。将目标板接入局域网或者通过交叉网线与主机直接相连，设定目标板的 IP，使之与主机在同一网段，然后进行远程 mount 操作。

```
[root@M283 ~]# ifconfig eth0 192.168.1.136
```

```
[root@M283 ~]# ping 192.168.1.138
```

```
[root@M283 ~]# mount -t nfs 192.168.1.138:/home/chenxibing/nfs /mnt -o nolock
```

在进行远程挂载之前，最好先用 `ping` 命令检查网络通信是否正常，只有在能 `ping` 通的情况下，才能进行正常挂载，否则请检查网络。如果在已经 `ping` 通的情况下，远程挂载出现错误，请检查主机和目标机的其它设置。

NFS 基于 RPC 协议，进行 NFS 挂载，客户端需要运行 `portmap` 服务，如果出现“RPC: Timed out”的错误，则有可能是目标板尚未启动 `portmap` 服务。解决方法就是在目标板启动 `portmap` 服务：

```
[root@M283 ~]# portmap&
```

## 1.25 Hello程序

使用熟悉的文本编辑器，在NFS目录下，编写一个简单的程序，往终端打印“Hello”字符串，程序清单 4.1是一个简单范例。

程序清单 4.1 Hello 程序清单

```
#include <stdio.h>
int main(void)
```

```
{
    int i;
    for (i=0; i<5; i++) {
        printf("Hello %d!\n", i);
    }
    return 0;
}
```

启动终端，进入 `hello` 程序文件所在目录，输入编译命令对 `hello.c` 进行编译：

```
chenxibing@linux-compiler: hello$ a arm-arago-linux-gnueabi-gcc hello.c -o hello
```

编译完毕，将得到 `hello` 文件。

启动系统，进行 NFS 挂载，并进入 `hello` 程序所在目录，运行 `hello` 程序。

```
[root@M283 ~]# mount -t nfs 192.168.1.138:/home/chenxibing/nfs /mnt -o nolock
[root@M283 ~]# cd /mnt/hello
[root@M283 hello]# ./hello
Hello 0!
Hello 1!
Hello 2!
Hello 3!
Hello 4!
```

如果需要固化 `hello` 程序，只需使用 `cp` 命令将 `hello` 文件复制到本地目录即可。

```
[root@M283 hello]# cp hello /root/
```

这是一个非常简单的程序，并且只有一个文件，所以可以采用直接输入命令进行交叉编译，如果工程较大，文件较多，这种方式就不可取了，通常需要编写 `Makefile` 文件，通过 `make` 程序来进行工程管理。程序清单 4.2 所示是一个简单的 `Makefile` 文件。

程序清单 4.2 应用程序 `Makefile` 范例

```
EXEC    = hello
OBSJ    = hello.o
CROSS   = arm-arago-linux-gnueabi-
CC       = $(CROSS)gcc
STRIP    = $(CROSS)strip
CFLAGS  = -Wall -g -O2
all:    clean $(EXEC)
$(EXEC):$(OBSJ)
        $(CC) $(CFLAGS) -o $@ $(OBSJ)
        $(STRIP) $@
clean:
        -rm -f $(EXEC) *.o
```

有了合适的 `Makefile` 文件，只需在终端输入 `make` 命令即可编译程序。`Makefile` 编写有详细规则，请参考其它书籍或者资料。

## 1.26 I<sup>2</sup>C 接口

M283 底板以排针形式引出了 I<sup>2</sup>C1 接口，Linux 系统实现了 I<sup>2</sup>C 总线的驱动，用户通过应用程序即可进行 I<sup>2</sup>C 总线的通信。

注：I2C-1 已经用于 RTC 芯片（PCF8563），客户如果想用做其他用途，需要联系销售来处理。

### 1.26.1 open调用

在使用I<sup>2</sup>C驱动操作接口时，先调用open函数打开I<sup>2</sup>C驱动设备文件，获得文件描述符，如程序清单 4.3所示。

程序清单 4.3 打开 I<sup>2</sup>C 设备文件

```
int fd;
fd = open("/dev/i2c-1", O_RDWR);
if (fd < 0) {
    perror("open i2c-1 \n");
}
```

### 1.26.2 ioctl调用

当使用 I<sup>2</sup>C 驱动操作 I<sup>2</sup>C 从机时，需要设置 I<sup>2</sup>C 从机地址以及地址长度。

#### 1. I2C\_SLAVE命令设置从机地址

I2C\_SLAVE 宏定义为：

```
#define I2C_SLAVE 0x0703
```

调用 ioctl 使用 I2C\_SLAVE 命令，可以设置 I<sup>2</sup>C 从机地址，参数就是 I<sup>2</sup>C 从机地址的值。当需要把 I<sup>2</sup>C 从机地址设置为 0xA0 时，示例代码如下：

```
ioctl(fd, I2C_SLAVE, I2C_ADDR >> 1); // 注意要右移一位，因为第 0 位是读写标记位
```

#### 2. I2C\_TENBIT命令设置从机地址长度

I2C\_TENBIT 宏的定义为：

```
#define I2C_TENBIT 0x0704
```

调用 ioctl 使用 I2C\_TENBIT 命令，可以设置 I<sup>2</sup>C 从机地址的长度。参数可为 1 或 0：当为 1 时，表示 I<sup>2</sup>C 从机地址长度为 10 位；当为 0 时，表示 I<sup>2</sup>C 从机地址长度为 8 位。

如把 I<sup>2</sup>C 从机地址长度设置为 8 位：

```
ioctl(fd, I2C_TENBIT, 0);
```

注：I<sup>2</sup>C 的 ioctl 调用使用到的命令在 kernel/Linux-2.6.35.3/include/Linux/i2c-dev.h 头文件中有定义，因此必须在程序中使用#include<i2c-dev.h>。

i2c-dev.h如程序清单 4.28所示。

程序清单 4.4 ioctl 命令定义

```
/* /dev/i2c-X ioctl commands. The ioctl's parameter is always an
 * unsigned long, except for:
 *
 * - I2C_FUNCS, takes pointer to an unsigned long
 *
 * - I2C_RDWR, takes pointer to struct i2c_rdwr_ioctl_data
 *
 * - I2C_SMBUS, takes pointer to struct i2c_smbus_ioctl_data
 */
#define I2C_RETRIES 0x0701 /* number of times a device address should */
/* be polled when not acknowledging */
#define I2C_TIMEOUT 0x0702 /* set timeout in units of 10 ms */
```

```
#define I2C_SLAVE          0x0703          /* Use this slave address          */
/* NOTE: Slave address is 7 or 10 bits, but          */
/* 10-bit addresses are NOT supported! (due          */
/* to code brokenness)                               */
#define I2C_SLAVE_FORCE 0x0706          /* Use this slave address, even if it is already          */
/* in use by a driver!                               */
#define I2C_TENBIT        0x0704          /* 0 for 7 bit addrs, != 0 for 10 bit          */
#define I2C_FUNCS          0x0705          /* Get the adapter functionality mask          */
#define I2C_RDWR           0x0707          /* Combined R/W transfer (one STOP only)          */
#define I2C_PEC            0x0708          /* != 0 to use PEC with SMBus          */
#define I2C_SMBUS          0x0720          /* SMBus transfer          */
```

### 1.26.3 write调用

当设置好 I<sup>2</sup>C 从机的地址后，就可以调用 write 向 I<sup>2</sup>C 从机器件写入数据。示例代码如下：

```
write(fd, buf, len);          // len 为 buf 缓冲区的长度
```

当 write 调用完成后，I<sup>2</sup>C 主机会向 I<sup>2</sup>C 从机器件发出 I<sup>2</sup>C 总线始起信号；在发出数据之前会先发出通过 ioctl 设置的从机地址，然后把 buf 缓冲区中的数据发出；最后发出 I<sup>2</sup>C 总线结束信号。

### 1.26.4 read调用

当设置好 I<sup>2</sup>C 从机的地址后，就可以调用 read 从 I<sup>2</sup>C 从机器件读入数据。示例代码如下：

```
read(fd, buf, len);          // len 表示要读数据的长度
```

当 read 调用完成后，I<sup>2</sup>C 主机向 I<sup>2</sup>C 从机发出总线始起信号；在读取数据之前会先发出通过 ioctl 设置的从机地址，从机器件在接收到从机地址后返回 ACK 信号，然后向 I<sup>2</sup>C 主机发送数据，驱动程序将接收到的数据存入 buf 中，最后发出 I<sup>2</sup>C 总线结束信号。

对于类似于 EEPROM 之类具有子地址的 I<sup>2</sup>C 接口的器件，在发送或读取数据之前需要先发送 I<sup>2</sup>C 子地址。

```
write(fd, addr, 1);          //发送要读取的数据的子地址
read(fd, rx_buf, 16);        //读取数据
```

EEPROM 的读写操作例程请参考开发示例中的 I<sup>2</sup>C 接口部分的代码。

### 1.26.5 close调用

当 I<sup>2</sup>C 驱动操作完成后，请使用 close 调用关闭之前打开的 I<sup>2</sup>C 驱动设备文件：

```
close(fd);
```

## 1.27 I/O端口使用

IMX28 系列处理器的 IO 端口分为 7 个 BANK，其中 BANK0~4 具有 GPIO 功能，每个 BANK 具有 32 个 I/O。本产品部分 BANK 的引脚不支持 GPIO 功能，具体需要参考

《IMX28CEC\_Datasheet.pdf》手册。在导出 GPIO 功能引脚时，需要先计算 GPIO 引脚的排列序号，其序号计算公式：

$$GPIO\text{序号} = BANK \times 32 + N$$

BANK 为 GPIO 引脚所在的 BANK，N 为引脚所在的 BANK 的序号。系统实现了通用 GPIO 的驱动，GPIO 的全部操作通过/sys/class/gpio 目录下的文件来完成，该目录下提供了 GPIO 的相关操作接口，用户通过读写这些文件，即能控制 GPIO 的输出和读取其输入值。

283 产品引出 P2.4、P2.5、P2.6、P2.7 四个 GPIO 功能引脚，287 产品则增加了 P1.28、P1.29、P1.30、P1.31、P2.12、P2.13、P2.14、P2.15、P3.6、P3.7、P3.10、P3.11、P3.14、P3.15，使用这些 GPIO 前，需要先将这些 GPIO 通过命令导出。

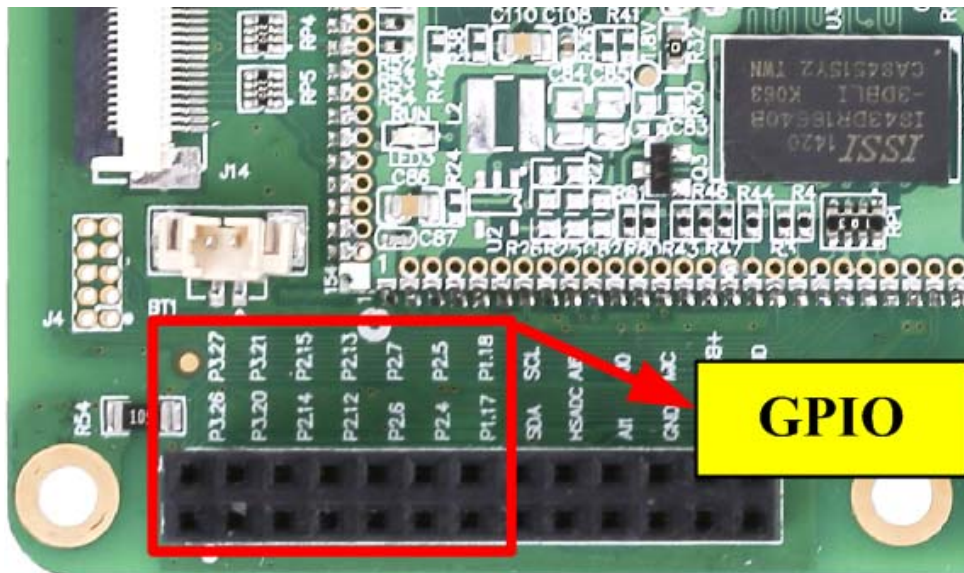


图 4.3 GPIO 引脚分布

以 P2.4 为例，其序号为  $2 \times 32 + 4 = 68$ 。因此导出 P2.4 的 GPIO 功能需要进行如下操作：

```
[root@M283 ~]# cd /sys/class/gpio
[root@M283 gpio]# ls
export      gpiochip128  gpiochip64  unexport
gpiochip0   gpiochip32   gpiochip96
[root@M283 gpio]# echo 68 >export
[root@M283 gpio]# ls
export      gpiochip0   gpiochip32   gpiochip96
gpio68      gpiochip128  gpiochip64   unexport root
[root@M283 gpio]# cd gpio68
[root@M283 gpio68]# ls
active_low  direction  edge  power  subsystem  uevent  value
```

通过以上操作后在/sys/class/gpio 目录下生成 gpio68 文件夹，文件夹下包含了 P2.4 引脚的属性文件，用于对 P2.4 GPIO 功能引脚进行操作。

以此类推可以导出其它 GPIO 功能引脚，对于已经被占用做其它功能的引脚无法导出其 GPIO 功能，导出时候会提示资源占用。

GPIO 导出后默认为输入，通过向 direction 文件写入“in”或者“out”字符串，可以设置 I/O 口的方向为输入或者输出；而读 direction 文件时，读到的字符串表示当前 I/O 口的方向设置。

```
[root@M283 gpio68]# cat direction    ←查看方向
in                                    ←方向输入
```

```
[root@M283 gpio68]# echo out >direction    ←设置为输出
[root@M283 gpio68]# cat direction          ←查看方向
out
```

当 GPIO 被设为输入时，通过读取 value 的内容可以获得输入电平的状态，读到 0 表示输入低电平，1 表示输入高电平；

当 GPIO 被设为输出时，通过向 value 写入 0 或 1 可以设置输出电平的状态，0 表示输出低电平，1 表示输出高电平。写操作仅在 I/O 口被设置为输出时有效。

除了在 Shell 下用 echo/cat 这样的命令操作 GPIO 外，还可以用 C 的标准文件操作函数 read, write 来对 /sys/class/gpio 中的文件进行操作，以 P2.4(gpio68)为例：

```
int fd;
fd = open("/sys/class/gpio/gpio68/value", O_RDWR);
if (fd < 0) {
printf("Open file /sys/class/gpio/gpio68/value failed!\n");
return -1;
}
for (i=0; i<5; i++) {
write(fd, "1", 1);
sleep(1);
write(fd, "0", 1);
sleep(1);
}
close(fd);
```

## 1.28 蜂鸣器使用

本产品的蜂鸣器控制文件是 /sys/class/leds/beep/brightness，写入 1 使蜂鸣器鸣叫，写入 0 停止鸣叫。注意底板的 JP1（BZ）需要接上跳线帽，否则蜂鸣器不会响。操作示例：

```
[root@M283 ~]# echo 1 >/sys/class/leds/beep/brightness ←控制蜂鸣器鸣叫
[root@M283 ~]# echo 0 >/sys/class/leds/beep/brightness ←控制蜂鸣器停止鸣叫
```

## 1.29 ADC

本产品的排针提供了 ADC0、ADC1、ADC6 三路 ADC 电压模拟量采集接口。此外，本产品还提供了一路 HSADC（高速 ADC），若客户需要使用，请联系销售。



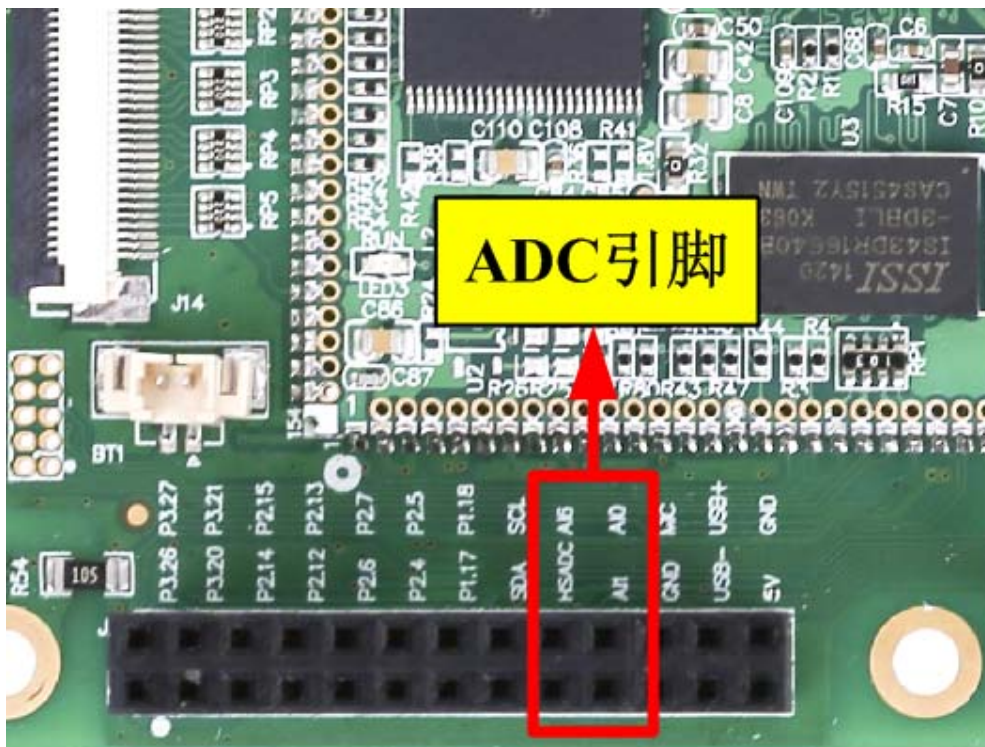


图 4.4 ADC 接口示意图

ADC0、ADC1、ADC6 三路通道内部含有一个除 2 模拟电路，在未开启内部除 2 电路时，其量程为 0~1.85V，开启除 2 电路时，量程为 0~3.7V。ADC 参考源来自内部参考电压 1.85V。另外，驱动提供了一个内部读取电池电压的接口，此通道内部有除 4 电路。

### 1.29.1 ADC驱动模块的加载

ADC 驱动以动态加载模块的形式提供，因此在 ADC 操作之前要先安装 lradc 驱动模块。

```
[root@M283 ~]# insmod /root/lradc.ko
adc module init!
```

### 1.29.2 操作接口

本着简单易用原则，ADC 使用字符设备的文件操作，操作仅使用了 ioctl 函数，读取电压操作代码如下：

```
int iRes, fd;
fd = open("/dev/magic-adc", 0);
ioctl(fd, cmd, &iRes);
```

其中，iRes 为读取的电压 AD 值，cmd 为操作命令，共有 7 个，分别如下：

- 10: 读取 ADC0 电压值；
- 11: 读取 ADC1 电压值；
- 16: 读取 ADC6 电压值；
- 17: 读取电池电压值；
- 21: 读取 ADC0 电压值（开启硬件除 2 电路）；
- 22: 读取 ADC1 电压值（开启硬件除 2 电路）；
- 26: 读取 ADC6 电压值（开启硬件除 2 电路）；



### 1.29.3 计算公式

对于命令 10、11、16，计算公式为： $V = 1.85 \times (Val/4096)$

对于命令 17，计算公式为： $V = 4 \times 1.85 \times (Val/4096)$

对于命令 21、22、26，计算公式为： $V = 2 \times 1.85 \times (Val/4096)$

### 1.29.4 操作示例

例程每一行打印四个数据：通道 0 开启除二电路采样值、通道 1 关闭除二电路采样值、通道 6 开启除二电路采样值、电池电压测量值，共打印 100 行后退出程序。

程序清单 4.5 ADC 操作示例

```
#include<stdio.h>                                /* using printf()          */
#include<stdlib.h>                                /* using sleep()           */
#include<fcntl.h>                                 /* using file operation     */
#include<sys/ioctl.h>                             /* using ioctl()            */
int main(int argc, char *argv[])
{
    int fd;
    int iRes;
    int iTime = 100;
    double val;
    fd = open("/dev/magic-adc", 0);
    if (fd < 0) {
        printf("open error by APP- %d\n", fd);
        close(fd);
        return 0;
    }
    while(iTime-->0) {
        sleep(1);
        ioctl(fd, 20, &iRes);                      /* CH0 开启硬件除 2 电路并读取数据 */
        val = (iRes * 3.7) / 4096.0;
        printf("CH0:%.2f\t", val);
        ioctl(fd, 11, &iRes);                      /* CH1 关闭硬件除 2 电路并读取数据 */
        val = (iRes * 1.85) / 4096.0;
        printf("CH1:%.2f\t", val);
        ioctl(fd, 26, &iRes);                      /* CH6 开启硬件除 2 电路并读取数据 */
        val = (iRes * 3.7) / 4096.0;
        printf("CH6:%.2f\t", val);
        ioctl(fd, 17, &iRes);                      /* 电池电压测量默认开启除 4 电路 */
        val = (iRes * 7.4) / 4096.0;
        printf("Vbat:%.2f\t", val);
        printf("\n");
    }
    close(fd);
}
```

## 1.30 串口编程

本产品上有 3 个用户串口，各串口在Linux系统中的设备名称如表 4.1所列。

表 4.1 串口列表

标号	功能	备注
DUART	DEBUGUART(/dev/ttyAM0)	调试串口，不能做它用
RXD0-TXD0	UART0 (/dev/ttySP0)	TTL 电平
RXD1-TXD1	UART1 (/dev/ttySP1)	TTL 电平
RXD2-TXD2	UART2 (/dev/ttySP2)	TTL 电平，仅 287 产品有效
RXD3-TXD3	UART3 (/dev/ttySP3)	TTL 电平，仅 287 产品有效
RXD4-TXD4	UART5 (/dev/ttySP4)	TTL 电平

### 1.30.1 访问串口设备

#### 1. 打开串口设备文件

在使用串口设备之前，我们需要先打开串口设备文件，获得串口设备文件描述符fd。在本产品的Linux系统中，串口设备文件名是“/dev/ttySPn”（其中 283 产品 n=0, 1, 4； 287 产品 n=0, 1, 2, 3, 4）。我们可以使用标准的open函数打开串口设备文件，如程序清单 4.6所示。

程序清单 4.6 打开串口设备

```
fd = open("/dev/ttySPn", O_RDWR | O_NOCTTY | O_NDELAY);
if (fd < 0) {
    perror(cSerialName);
    exit(0);
}
```

#### 2. open选项

当打开串口设备文件时，我们除了用到 O\_RDWR 选项标志外，还使用到 O\_NOCTTY 和 O\_NDELAY 选项标志：

```
fd = open("/dev/ttySP1", O_RDWR | O_NOCTTY | O_NDELAY);
```

O\_NOCTTY 选项标志是告诉 Linux：本程序是不作为串口端口的“控制终端”。如果不作这样特别指出，会有一些输入字符（如一些产生中断信号的键盘输入字符等）影响进程。

O\_NDELAY 标志表示程序忽略 DCD 信号线。如果不加这标志选项，进程可能在 DCD 信号线被拉低时进入休眠。

#### 3. 向串口设备写入数据

向串口设备写入数据是相当容易的，仅是使用标准的write系统调用，如程序清单 4.7所示。

程序清单 4.7 向串口设备写入数据

```
n = write(fd, "hollow zlg \r", 14);
if (n < 0) {
    printf("write data to serial failed! \n");
}
```

#### 4. 从串口设备读取数据

当串口设备工作在原始数据模式时，每次 `read` 系统调用都返回串口驱动缓冲区里实际可用的数据。如果缓冲区没有数据可用，`read` 系统调用等待到数据到来。这可能会堵塞进程。为了使 `read` 调用能立即返回，可以采用下面操作：

```
fcntl(fd, F_SETFL, FNDELAY);
```

`FNDELAY` 选项会使 `read` 函数在串口没有数据到来的情况下立即返回。若要恢复正常状态，可以再次调用 `fcntl()` 而不带 `FNDELAY` 选项：

```
fcntl(fd, F_SETFL, 0);
```

这些操作通常在完成 `open`（带 `O_NDELAY` 选项）串口设备后执行。

## 5. 关闭串口设备

关闭串口设备仅需 `close` 系统调用：

```
close(fd);
```

关闭一个串口设备也通常会引起串口 `DTR` 信号线电平置高，使大部分的 modem 设备挂起。

### 1.30.2 配置串口接口属性

串口设备的波特率、数据位、校验方式等属性，是通过终端接口配置实现的。

#### 1. 终端接口

终端属性用 `termios` 结构描述，如程序清单 4.8 所示。

程序清单 4.8 `termios` 结构

```
struct termios {
tcflag_t  c_cflag          /* 控制标志 */
tcflag_t  c_iflag;         /* 输入标志 */
tcflag_t  c_oflag;         /* 输出标志 */
tcflag_t  c_lflag;         /* 本地标志 */
    tcflag_t  c_cc[NCCS];   /* 控制字符 */
};
```

简单来说，输入控制标志由终端设备驱动程序用来控制字符的输入（剥除输入字节的第 8 位，允许输入奇偶校验等等），输出控制则控制驱动程序输出，控制标志影响到 `RS-232` 串行线，本地标志影响驱动程序和用户之间的接口（串口作为用户终端时）。

#### 2. 获得和设置属性

使用函数 `tcgetattr` 和 `tcsetattr` 可以获得或设置 `termios` 结构，如程序清单 4.9 所示。

程序清单 4.9 设置和获得 `termios` 结构函数

```
#include <termios.h>
int tcgetattr(int fd, struct termios *termpptr);
int tcsetattr(int fd, int opt, const struct termios *termpptr);
```

上述两函数执行时，若成功中则返回 0，若出错则返回 -1。这两个函数都有一个指向 `termios` 结构的指针作为其参数，它们返回当前串口的属性，或者设置该串口的属性。

在串口驱动程序里，有输入缓冲区和输出缓冲区。在改变串口属性时，缓冲区中的数据可能还存在，这时需要考虑到更改后的属性什么时候起作用。`tcsetattr` 的参数 `opt` 使我们指定在什么时候新的串口属性才起作用。`opt` 可以指定为下列常量中的一个：

- TCSANOW 更改立即发生。
- TCSADRAIN 发送了所有输出后更改才发生。若更改输出参数则应用此选项。
- TCSAFLUSH 发送了所有输出后更改才发生。更进一步，在更改发生时未读的所有输入数据被删除（刷清）。

### 3. 控制标志

c\_cflag成员控制着波特率、数据位、奇偶校验、停止位以及流控制。表 4.2列出了c\_cflag可用的部分选项。

表 4.2 c\_cflag 部分可用选项

标志	说明
CBAUD	波特率位屏蔽
B0	0 位/秒（挂起）
B110	100 位/秒
B134	134 位/秒
B1200	1200 位/秒
B2400	2400 位/秒
B4800	4800 位/秒
B9600	9600 位/秒
B19200	19200 位/秒
B57600	57600 位/秒
B115200	115200 位/秒
B460800	460800 位/秒
CSIZE	数据位屏蔽
CS5	5 位数据位
CS6	6 位数据位
CS7	7 位数据位
CS8	8 位数据位
CSTOPB	2 位停止位，否则为 1 位
CREAD	启动接收
PARENB	进行奇偶校验
PARODD	奇校验，否则为偶校验
HUPCL	最后关闭时断开
CLOCAL	忽略调制调解器状态行

c\_cflag 成员有两选项通常是要启用的：CLOCAL 和 CREAD。这会程序启动接收字符装置，同时忽略串口信号线的状态。

#### ● 标志选项

termios 各成员的各个选项标志（除屏蔽标志外）都用一位或几位表示（设置或清除）表示，而屏蔽标志则定义多位，它们组合在一起，于是可以定义多个值。屏蔽标志有一个定义名，每个值也有一个名字。例如，为了设置字符长度，首先用字符长度屏蔽标志 CSIZE 将表示字符长度的位清 0，然后设置下列值之一：CS5、CS6、CS7 或 CS8。

程序清单 4.10例示了怎样使用屏蔽标志或设置一个值。

程序清单 4.10 tcgetattr 和 tcsetattr 实例

```
#include <termios.h>
int main(void)
{
    struct termios term;
    int fd;
    fd = open("/dev/ttySP0", O_RDWR | O_NOCTTY);
    if(fd < 0) {
        perror("SerialName");
        exit(0);
    }
    if(tcgetattr(fd, &term) < 0) {
        printf("tcgetattr error");
        exit(0);
    }
    switch(term.c_cflag & CSIZE) {
    case CS5:
        printf("5 bits/byte \n");
        break;
    case CS6:
        printf("6 bits/byte \n");
        break;
    case CS7:
        printf("7 bits/byte \n");
        break;
    case CS8:
        printf("8 bits/byte \n");
        break;
    default:
        printf("unknown bits/byte \n");
    }
    term.c_cflag &= ~CSIZE;
    term.c_cflag |= CS8;
    if(tcsetattr(fd, TCSANOW, &term) < 0) {
        printf("tcsetattr error");
        exit(0);
    }
    return 0;
}
```

- 设置波特率

cfsetispeed和cfsetospeed分别用于设置串口的输入和输出波特率，如程序清单 4.11所示。

程序清单 4.11 设置串口输入/输出波特率函数

```
#include <termios.h>
```

```
int cfsetispeed(struct termios *termpptr, speed_t speed);
int cfsetospeed(struct termios *termpptr, speed_t speed);
```

这两个函数若执行成功返回 0，若出错则返回-1。

使用这两个函数时，应当理解输入、输出波特率是存在串口设备termios结构中的。在调用任一cfset函数之前，先要用tcgetattr获得设备的termios结构。与此类似，在调用任一cfset函数后，波特率都被设置到termios结构中。为使用这种更改影响到设备，应当调用tcsetattr函数。操作方法如程序清单 4.12所示。

程序清单 4.12 设置波特率示例

```
if (tcgetattr(fd, &opt)<0) {
    return ERROR;
}
cfsetispeed(&opt, B9600);
cfsetospeed(&opt, B9600);
if (tcsetattr(fd, TCSANOW, &opt)<0) {
    return ERROR;
}
```

#### ● 设置数据位

设置数据位不需要专用的函数。在设置数据位之前，需要用数据位屏蔽标志（CSIZE）把数据位清零，然后设置数据位，如下所示：

```
options.c_cflag &= ~CSIZE;          /* 先把数据位清零 */
options.c_cflag |= CS8;             /* 把数据位设置为 8 位 */
```

#### ● 设置奇偶校验

正如设置数据位一样，设置奇偶校验是在直接在 cflag 成员上设置。下面是各种类型的校验设置方法：

##### ■ 无奇偶校验（8N1）

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

##### ■ 7 位数据位奇偶校验（7E1）

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

##### ■ 奇校验（7O1）

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

#### 4. 本地标志

本地标志c\_lflag控制着串口驱动程序如何管理输入的字符。表 4.3所示的是c\_lflag部分可用标志。

表 4.3 c\_lflag 标志

标志	说明
ISIG	启用终端产生的信号
ICANON	启用规范输入
XCASE	规范大/小写表示
ECHO	进行回送
ECHOE	可见擦除字符
ECHOK	回送 kill 符
ECHONL	回送 NL
NOFLSH	在中断或退出键后禁用刷新
IEXTEN	启用扩充的输入字符处理
ECHOCTL	回送控制字符为^(char)
ECHOPRT	硬拷贝的可见擦除方式
ECHOKE	Kill 的可见擦除
PENDIN	重新打印未决输入
TOSTOP	对于后台输出发送 SIGTTOU

##### ● 选择规范模式

规范模式是行处理的。调用 read 读取串口数据时，每次返回一行数据。当选择规范模式时，需要启用 ICANON、ECHO 和 ECHOE 选项：

```
options.c_lflag |= (ICANON | ECHO | ECHOE);
```

当串口设备作为用户终端时，通常要把串口设备配置成规范模式。

##### ● 选择原始模式

在原始模式下，串口输入数据是不经过处理的。在串口接口接收的数据被完整保留。要使串口设备工作在原始模式，需要关闭 ICANON、ECHO、ECHOE 和 ISIG 选项：

```
options.c_lflag &= ~(ICANON | ECHO | ECHOE | ISIG);
```

#### 5. 输入标志

c\_iflag成员负责控制串口输入数据的处理。表 4.4所示是c\_iflag部分可用标志。

表 4.4 c\_iflag 标志

标志	说明
INPCK	打开输入奇偶校验
IGNPAR	忽略奇偶错字符
PARMRK	标记奇偶错
ISTRIP	剥除字符第 8 位
IXON	启用/停止输出控制流起作用
IXOFF	启用/停止输入控制流起作用
IGNBRK	忽略 BREAK 条件
INLCR	将输入的 NL 转换为 CR
IGNCR	忽略 CR



ICRNL	将输入的 CR 转换为 NL
-------	----------------

- 设置输入校验

当 `c_cflag` 成员的 `PARENB`（奇偶校验）选项启用时，`c_iflag` 的也应启用奇偶校验选项。操作方法是启用 `INPCK` 和 `ISTRIP` 选项：

```
options.c_iflag |= (INPCK | ISTRIP);
```

这里有个选项值得注意：`IGNPAR`。`IGNPAR` 选项在一些场合的应用带有一定的危险性，它是指示串口驱动程序忽略串口接口接收到的字符奇偶校验出错。也就是说，`IGNPAR` 使即使奇偶校验出错的字符也通过输入。这在测试通信链路的质量时也许有用，但在通常的数据通信应用中不应使用。

- 设置软件流控制

使用软件流控制是启用 `IXON`、`IXOFF` 和 `IXANY` 选项：

```
options.c_iflag |= (IXON | IXOFF | IXANY);
```

相反，要禁用软件流控制是禁止上面的选项：

```
options.c_iflag &= ~(IXON | IXOFF | IXANY);
```

## 6. 输出标志

`c_oflag` 成员管理输出过滤。表 4.5 所示的是 `c_oflag` 成员部分选项标志。

表 4.5 `c_oflag` 标志

标志	说明
<code>BSDLY</code>	退格延迟屏蔽
<code>CMSPAR</code>	标志或空奇偶性
<code>CRDLY</code>	<code>CR</code> 延迟屏蔽
<code>FFDLY</code>	换页延迟屏蔽
<code>OCRNL</code>	将输出的 <code>CR</code> 转换为 <code>NL</code>
<code>OFDEL</code>	填充符为 <code>DEL</code> ，否则为 <code>NULL</code>
<code>OFILL</code>	对于延迟使用填充符
<code>OLCUC</code>	将输出的小写字符转换为大写字符
<code>ONLCR</code>	将 <code>NL</code> 转换为 <code>CR-NL</code>
<code>ONLRET</code>	<code>NL</code> 执行 <code>CR</code> 功能
<code>ONOCR</code>	在 0 列不输出 <code>CR</code>
<code>OPOST</code>	执行输出处理
<code>OXTABS</code>	将制表符扩充为空格

- 启用输出处理

启用输出处理是在 `c_oflag` 成员启用 `OPOST` 选项：

```
options.c_oflag |= OPOST;
```

- 使用原始输出

使用原始输出，就是禁用输出处理。使用原始输出，数据能不经过处理、过滤地完整地输出到串口接口。操作方法：

```
options.c_oflag &= ~OPOST;
```

当 `OPOST` 被禁止，`c_oflag` 其它选项也被忽略。

## 7. 控制字符组

c\_cc数组包含了所有可以更改的特殊字符。该数组长度是NCCS，一般是介于 15-20 之间。c\_cc数组的每个成员的下标都用一个宏表示。表 4.6列出了c\_cc的部分标志。

表 4.6 c\_cc 标志

标志	说明
VINTR	中断
VQUIT	退出
VERASE	擦除
VEOF	行结束
VEOL	行结束
VMIN	需读取的最小字符数
VTIME	可以等待数据的最长时间

### ● VMIN 和 VTIME

在规范模式下，调用 read 读取串口数据时，通常是返回一行数据。而在原始模式下，串口输入数据是不分行的。

在原始模式下，返回读取数据的数量需要考虑两个变量：MIN 和 TIME。MIN 和 TIME 在 c\_cc 数组中的下标名为 VMIN 和 VTIME。

MIN 说明一个 read 返回前的最小字节数。TIME 说明等待数据到达的分秒数（秒的 1/10 为分秒）。有下列四种情形：

#### ◆ 当 MIN > 0, TIME > 0 时

TIME 说明字节间的计时器，在接到第一个字节时才启动它。在该计时器超时之前，若已接到 MIN 个字节，则 read 返回 MIN 个字节。如果在接到 MIN 个字节之前，该计时器已超时，则 read 返回已接收到的字节（因为只有在接收到第一个字节时才启动，所以在计时器超时的时候，至少返回了 1 个字节）。这种情形中，在接到第一个字节之前，调用者阻塞。如果在调用 read 时数据已经可用，则这如同在 read 后数据立即被接到一样。

#### ◆ 当 MIN > 0, TIME == 0 时

已经接到了 MIN 个字节时，read 才返回。这可能会造成 read 无限期地阻塞。

#### ◆ 当 MIN == 0, TIME > 0 时

TIME 指定了一个调用 read 时启动的计时器，（与第一种情形是不同的，在第一种情形下，是在接收到第一个字节时，才启动定时器）在接到 1 字节或者该计时器超时，read 即返回。如果是计时器超时，则 read 返回 0。

#### ◆ 当 MIN == 0, TIME == 0 时

如果有数据可用，则 read 最多返回所要求的字节数。如果无数据可用，则 read 立即返回 0。

## 8. 实例

程序清单 4.13是原始模式下串口操作的实例。

程序清单 4.13 串口程序示例

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

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <errno.h>
#include <limits.h>
#define DEV_NAME "/dev/ttySP1"
int main(void)
{
    int iFd, i;
    int len;
    unsigned char ucBuf[1000];
    struct termios opt;
    iFd = open(DEV_NAME, O_RDWR | O_NOCTTY);
    if(iFd < 0) {
        perror(DEV_NAME);
        return -1;
    }
    tcgetattr(iFd, &opt);
    cfsetispeed(&opt, B115200);
    cfsetospeed(&opt, B115200);
    if (tcgetattr(iFd, &opt)<0) {
        return -1;
    }
    opt.c_lflag &= ~(ECHO | ICANON | IEXTEN | ISIG);
    opt.c_iflag &= ~(BRKINT | ICRNL | INPCK | ISTRIP | IXON);
    opt.c_oflag &= ~(OPOST);
    opt.c_cflag &= ~(CSIZE | PARENB);
    opt.c_cflag |= CS8;
    opt.c_cc[VMIN] = 255;
    opt.c_cc[VTIME] = 150;
    if (tcsetattr(iFd, TCSANOW, &opt)<0) {
        return -1;
    }
    tcflush(iFd,TCIOFLUSH);
    for (i = 0; i < 1000; i++){
        ucBuf[i] = 0xff - i;
    }
    write(iFd, ucBuf, 0xff);
    len = read(iFd, ucBuf, 0xff);
    printf("get date: %d \n", len);
    for (i = 0; i < len; i++){
        printf(" %x", ucBuf[i]);
    }
}
```

```

    }
    printf("\n");
    close(iFd);
    return 0;
}

```

测试上述代码时，需要把“/dev/ttySP1”对应的 RS-232 接口的 RXD 和 TXD 用杜邦线短接起来。当程序执行时，程序在串口把发出去的数据再读回，并打印出来。

### 1.30.3 获得和设置串口信号线状态

获得和设置串口信号线状态是通过对打开的串口设备文件描述符调用ioctl实现。程序清单 4.14是获取串口信号线状态的范例。

程序清单 4.14 设置串口信号线状态

```

#include <unistd.h>
#include <asm/termios.h>
int fd;
int status;
ioctl(fd, TIOCMGET, &status);

```

在该例子中，使用TIOCMGET命令调用ioctl把串口信号线状态存放在整型变量status中。该整型变量的各位含义如表 4.7所示。

表 4.7 TIOCM 标志

标志	说明
TIOCM_LE	DSR
TIOCM_DTR	DTR
TIOCM_RTS	RTS
TIOCM_ST	TXD
TIOCM_SR	RXD
TIOCM_CTS	CTS
TIOCM_CAR	DCD
TIOCM_CD	等于 TIOCM_CAR
TIOCM_RNG	RNG
TIOCM_RI	等于 TIOCM_RNG
TIOCM_DSR	DSR

设置串口信号线状态需要使用TIOCMSET命令，如程序清单 4.15所示。

程序清单 4.15 设置串口信号线状态

```

#include <unistd.h>
#include <termios.h>
int fd;
int status;
ioctl(fd, TIOCMGET, &status);
status &= ~TIOCM_DTR;
ioctl(fd, TIOCMSET, &status);

```

上述代码是把 DTR 信号线拉低。

### 1.31 Socket CAN编程

因为 283 产品不带 CAN 接口，287 产品自带 2 路 CAN 接口，支持 Socket CAN。本节关于 Socket CAN 的内容仅针对 287 产品。

#### 1.31.1 初始化CAN网络接口

在使用 socket can 之前，需要先设置 CAN 的波特率和激活 CAN 网络接口。可以使用下面的指令：

```
[root@M283 ~]# ifconfig can0 down
[root@M283 ~]# echo 1000000 > /sys/devices/platform/FlexCAN.0/bitrates ←设置波特率为 1M
[root@M283 ~]# ifconfig can0 up
```

完成之后，输入“ifconfig can0”命令就可以看到新添加的 CAN 网络接口：

```
[root@M283 ~]# ifconfig can0
can0      Link encap:UNSPEC  HWaddr 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
          UP RUNNING NOARP  MTU:16  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:10
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

这时我们就可以使用 socket can 了。

#### 1.31.2 socket can编程

##### 1. 创建套接字

就像 TCP/IP 协议一样，在使用 CAN 网络之前需要先打开一个套接字。CAN 的套接字使用到了一个新的协议族，所以在调用 socket(2)这个系统函数的时候需要将 PF\_CAN 作为第一个参数。当前有两个 CAN 的协议可以选择：一个是原始套接字协议；另一个是广播管理协议。可以这样来打开一个套接字：

```
s = socket(PF_CAN, SOCK_RAW, CAN_RAW);
```

或者

```
s = socket(PF_CAN, SOCK_DGRAM, CAN_BCM);
```

##### 2. 绑定CAN接口

在成功创建一个套接字之后，通常需要使用 bin(2)函数将套接字绑定在某个指定的 CAN 接口上（这和 TCP/IP 使用不同的 IP 地址不同）。在绑定（CAN\_RAW）或连接（CAN\_BCM）套接字之后，就可以在套接字上使用 read(2)/write(2)，也可以使用 send(2)/sendmsg(2)和对应的 rev\*操作。

基本的CAN帧结构和套接字地址结构定义在/include/linux/can.h，如程序清单 4.16所示。

程序清单 4.16 can\_frame 的定义

```
/*
 * 扩展格式识别符由 29 位组成。其格式包含两个部分：11 位基本 ID、18 位扩展 ID。
 * Controller Area Network Identifier structure
 */
```

```
* bit 0-28 : CAN 识别符 (11/29 bit)
* bit 29 : 错误帧标志 (0 = data frame, 1 = error frame)
* bit 30 : 远程发送请求标志 (1 = rtr frame)
* bit 31 : 帧格式标志 (0 = standard 11 bit, 1 = extended 29 bit)
*/
typedef __u32 canid_t;
struct can_frame {
    canid_t can_id; /* 32 bit CAN_ID + EFF/RTR/ERR flags */
    __u8 can_dlc; /* 数据长度: 0 .. 8 */
    __u8 data[8] __attribute__((aligned(8)));
};
```

结构体的有效数据在 data 数组中，它的字节对齐是 64bit 的，所以用户可以比较方便的在 data 中传输自己定义的结构和共用体。CAN 总线中没有默认的字节序。在 CAN\_RAW 套接字上调用 read(2)，返回给用户空间的数据是一个 struct can\_frame 的结构体。

就像PF\_PACKET套接字一样，sockaddr\_can结构体也有接口的索引，这个索引绑定了特定接口，如程序清单 4.17所示。

程序清单 4.17 struct sockaddr\_can 结构体

```
struct sockaddr_can {
    sa_family_t can_family;
    int can_ifindex;
    union {
        /* transport protocol class address info (e.g. ISOTP) */
        struct { canid_t rx_id, tx_id; } tp;
        /* reserved for future CAN protocols address information */
    } can_addr;
};
```

指定接口索引需要调用ioctl()，如程序清单 4.18所示。

程序清单 4.18 绑定接口

```
int s;
struct sockaddr_can addr;
struct ifreq ifr;

s = socket(PF_CAN, SOCK_RAW, CAN_RAW);

strcpy(ifr.ifr_name, "can0");
ioctl(s, SIOCGIFINDEX, &ifr);

addr.can_family = AF_CAN;
addr.can_ifindex = ifr.ifr_ifindex;

bind(s, (struct sockaddr *)&addr, sizeof(addr));
```

.....

为了将套接字和所有的 CAN 接口绑定，接口索引必须是 0。这样套接字就可以从所有使用的 CAN 接口接收 CAN 帧。revfrom(2)可以指定从哪个接口接收。在一个已经和所有 CAN 接口绑定的套接字上，sendto(2)可以指定从哪个接口发送。

### 3. 接收/发送帧

从一个CAN\_RAW套接字上读取CAN帧也就是读取struct can\_frame结构体，如程序清单 4.19所示。

程序清单 4.19 接收 CAN 帧

```
struct can_frame frame;

nbytes = read(s, &frame, sizeof(struct can_frame));
if (nbytes < 0) {
    perror("can raw socket read");
    return 1;
}

/* paranoid check ... */
if (nbytes < sizeof(struct can_frame)) {
    fprintf(stderr, "read: incomplete CAN frame\n");
    return 1;
}

/* do something with the received CAN frame */
```

写 CAN 帧也是类似的用到 write(2)函数：

```
nbytes = write(s, &frame, sizeof(struct can_frame));
```

如果套接字跟所有的CAN接口都绑定了（addr.can\_index = 0），推荐使用recvfrom(2)获取数据源接口信息，程序清单 4.20所示。

程序清单 4.20 获取数据源接口信息

```
struct sockaddr_can addr;
struct ifreq ifr;
socklen_t len = sizeof(addr);
struct can_frame frame;

nbytes = recvfrom(s, &frame, sizeof(struct can_frame),
    0, (struct sockaddr*)&addr, &len);

/* get interface name of the received CAN frame */
ifr.ifr_ifindex = addr.can_ifindex;
ioctl(s, SIOCGIFNAME, &ifr);
printf("Received a CAN frame from interface %s", ifr.ifr_name);
```

对于绑定了所有接口的套接字，向某个端口发送数据必须指定接口的详细信息，如程序清单 4.21所示。



程序清单 4.21 指定输出接口的详细信息

```
strcpy(ifr.ifr_name, "can0");
ioctl(s, SIOCGIFINDEX, &ifr);
addr.can_ifindex = ifr.ifr_ifindex;
addr.can_family = AF_CAN;

nbytes = sendto(s, &frame, sizeof(struct can_frame),
                0, (struct sockaddr*)&addr, sizeof(addr));
```

#### 4. 使用过滤器

在上面的介绍中，我们从 CAN 接口中接收所有的数据帧，也不管我们是不是感兴趣。如果我们只想要指定 ID 的数据帧，那我们需要使用过滤器。

##### ● 原始套接字选项 CAN\_RAW\_FILTER

CAN\_RAW套接字的接收可以使用CAN\_RAW\_FILTER套接字选项指定的多个过滤规则。过滤规则定义在/include/linux/can.h中，如程序清单 4.22所示。

程序清单 4.22 can\_filter 的定义

```
struct can_filter {
    canid_t can_id;
    canid_t can_mask;
};
```

过滤规则的匹配：

```
<接收帧 id> & mask == can_id & mask
```

启用过滤器的示例如程序清单 4.23所示。

程序清单 4.23 启用过滤器示例代码

```
/* valid bits in CAN ID for frame formats */
#define CAN_SFF_MASK 0x000007FFU /* 标准帧格式 (SFF) */
#define CAN EFF_MASK 0x1FFFFFFFU /* 扩展帧格式 (EFF) */
#define CAN_ERR_MASK 0x1FFFFFFFU /* 忽略 EFF, RTR, ERR 标志 */

struct can_filter rfilter[2];

rfilter[0].can_id = 0x123;
rfilter[0].can_mask = CAN_SFF_MASK;
rfilter[1].can_id = 0x200;
rfilter[1].can_mask = 0x700;

setsockopt(s, SOL_CAN_RAW, CAN_RAW_FILTER, &rfilter, sizeof(rfilter));
```

为了在指定的 CAN\_RAW 套接字上禁用接收过滤规则，可以这样：

```
setsockopt(s, SOL_CAN_RAW, CAN_RAW_FILTER, NULL, 0);
```

在一些极端情况下不需要读取数据，可以把过滤规则清零（所有成员为 0），这样原始套接字就会忽略接到的 CAN 帧。

- 原始套接字选项 CAN\_RAW\_ERR\_FILTER

CAN 接口驱动可以选择性的产生错误帧，错误帧和正常帧以相同的方式传给应用程序。可能产生的错误被分不同的各类，使用适当的错误掩码可以过滤它们。为了注册所有可能的错误情况，CAN\_ERR\_MASK 这个宏可以用来作为错误掩码。这个错误掩码定义在 linux/can/error.h。

使用示例如下：

```
can_err_mask_t err_mask = ( CAN_ERR_TX_TIMEOUT | CAN_ERR_BUSOFF );
setsockopt(s, SOL_CAN_RAW, CAN_RAW_ERR_FILTER,&err_mask, sizeof(err_mask));
```

### 1.31.3 示例程序

socket can的示例程序如程序清单 4.24所示。该程序是把接收到的指定ID的数据帧打印出来，然后把接收到的数据帧发送出去。

程序清单 4.24 socket can 示例程序

```
#include <stdio.h>
#include <sys/ioctl.h>
#include <arpa/inet.h>
#include <net/if.h>
#include <linux/socket.h>
#include <linux/can.h>
#include <linux/can/error.h>
#include <linux/can/raw.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <time.h>

#ifndef AF_CAN
#define AF_CAN 29
#endif

#ifndef PF_CAN
#define PF_CAN AF_CAN
#endif

static void print_frame(struct can_frame *fr)
{
    int i;
    printf("%08x\n", fr->can_id & CAN_EFF_MASK);
    //printf("%08x\n", fr->can_id);
    printf("dlc = %d\n", fr->can_dlc);
    printf("data = ");
    for (i = 0; i < fr->can_dlc; i++)
```

```
        printf("%02x ", fr->data[i]);
    printf("\n");
}

#define errout(_s)    fprintf(stderr, "error class: %s\n", (_s))
#define errcode(_d) fprintf(stderr, "error code: %02x\n", (_d))

static void handle_err_frame(const struct can_frame *fr)
{
    if (fr->can_id & CAN_ERR_TX_TIMEOUT) {
        errout("CAN_ERR_TX_TIMEOUT");
    }
    if (fr->can_id & CAN_ERR_LOSTARB) {
        errout("CAN_ERR_LOSTARB");
        errcode(fr->data[0]);
    }
    if (fr->can_id & CAN_ERR_CRTL) {
        errout("CAN_ERR_CRTL");
        errcode(fr->data[1]);
    }
    if (fr->can_id & CAN_ERR_PROT) {
        errout("CAN_ERR_PROT");
        errcode(fr->data[2]);
        errcode(fr->data[3]);
    }
    if (fr->can_id & CAN_ERR_TRX) {
        errout("CAN_ERR_TRX");
        errcode(fr->data[4]);
    }
    if (fr->can_id & CAN_ERR_ACK) {
        errout("CAN_ERR_ACK");
    }
    if (fr->can_id & CAN_ERR_BUSOFF) {
        errout("CAN_ERR_BUSOFF");
    }
    if (fr->can_id & CAN_ERR_BUSERERROR) {
        errout("CAN_ERR_BUSERERROR");
    }
    if (fr->can_id & CAN_ERR_RESTARTED) {
        errout("CAN_ERR_RESTARTED");
    }
}

#define myerr(str)    fprintf(stderr, "%s, %s, %d: %s\n", __FILE__, __func__, __LINE__, str)
```

```
static int test_can_rw(int fd, int master)
{
    int ret, i;
    struct can_frame fr, frdup;
    struct timeval tv;
    fd_set rset;

    while (1) {
        tv.tv_sec = 1;
        tv.tv_usec = 0;
        FD_ZERO(&rset);
        FD_SET(fd, &rset);

        ret = select(fd+1, &rset, NULL, NULL, NULL);
        if (ret == 0) {
            myerr("select time out");
            return -1;
        }
        /* select 调用无错返回时，表示有符合规则的数据帧到达 */
        ret = read(fd, &frdup, sizeof(frdup));
        if (ret < sizeof(frdup)) {
            myerr("read failed");
            return -1;
        }
        if (frdup.can_id & CAN_ERR_FLAG) { /* 检查数据帧是否错误 */
            handle_err_frame(&frdup);
            myerr("CAN device error");
            continue;
        }
        print_frame(&frdup); /* 打印数据帧信息 */
        ret = write(fd, &frdup, sizeof(frdup)); /* 把接收到的数据帧发送出去 */
        if (ret < 0) {
            myerr("write failed");
            return -1;
        }
    }
    return 0;
}

int main(int argc, char *argv[])
{
    int s;
    int ret;
    struct sockaddr_can addr;
```

```
struct ifreq ifr;
int master;

srand(time(NULL));
s = socket(PF_CAN, SOCK_RAW, CAN_RAW);      /* 创建套接字 */
if (s < 0) {
    perror("socket PF_CAN failed");
    return 1;
}
/* 把套接字绑定到 can0 接口 */
strcpy(ifr.ifr_name, "can0");
ret = ioctl(s, SIOCGIFINDEX, &ifr);
if (ret < 0) {
    perror("ioctl failed");
    return 1;
}

addr.can_family = PF_CAN;
addr.can_ifindex = ifr.ifr_ifindex;
ret = bind(s, (struct sockaddr *)&addr, sizeof(addr));
if (ret < 0) {
    perror("bind failed");
    return 1;
}
/* 设置过滤规则 */
if (1) {
    struct can_filter filter[2];
    /* 第 1 个规则是可以接收 ID 为 0x200 & 0xFFF 的数据帧 */
    filter[0].can_id = 0x200 | CAN_EFF_FLAG;
    filter[0].can_mask = 0xFFF;
    /* 第 2 个规则是可以接收 ID 为 0x20F & 0xFFF 的数据帧 */
    filter[1].can_id = 0x20F | CAN_EFF_FLAG;
    filter[1].can_mask = 0xFFF;

    /* 启用过滤规则，只要 CAN0 接收到的数据帧满足上面 2 个规则中的任何一个也被接受*/
    ret = setsockopt(s, SOL_CAN_RAW, CAN_RAW_FILTER, &filter, sizeof(filter));
    if (ret < 0) {
        perror("setsockopt failed");
        return 1;
    }
}

test_can_rw(s, master);    /*进入测试*/
close(s);
return 0;
```

```
}
```

上述代码在编译后，可以使用广州致远电子股份有限公司的 USBCAN 或 CANNET 来测试。

## 1.32 看门狗使用

### 1.32.1 概述

本产品有一个可编程看门狗（/dev/watchdog），可用于监测应用程序的运行状况。看门狗有如下特点：

- （1）默认超时时间为 19 秒，超时时间可设置（1~60 秒）；
- （2）先向看门狗发送一个字符“V”后，可关闭看门狗。

#### 1. 设置超时时间

看门狗默认超时时间为 19 秒，用户可以将超时时间设置为适合的值，通过 ioctl 实现，具体命令为：WDIOC\_SETTIMEOUT，需要一个参数，即超时时间。使用示例：

```
int timeout = 10;
ioctl(wdt_fd, WDIOC_SETTIMEOUT, &timeout);
```

#### 2. 获取超时时间

通过 ioctl 以及 WDIOC\_GETTIMEOUT 命令，可以获得看门狗的超时时间。使用示例：

```
timeout = 0;
ioctl(wdt_fd, WDIOC_GETTIMEOUT, &timeout);
printf("The timeout is %d seconds\n", timeout);
```

#### 3. 喂狗操作

看门狗被打开后，必须在超时时间内进行喂狗，最简单的喂狗办法就是向看门狗写一个字符（特殊字符除外）。

#### 4. 关闭看门狗

需先写入“V”，然后使用 close 关闭看门狗。使用示例：

```
write(wdt_fd, "V", 1);
close(wdt_fd);
```

### 1.32.2 范例

程序清单 4.25 所示是一个非常简单的范例。打开看门狗后，将看门狗超时设置为 10 秒钟，然后在超时时间内周期性喂狗。运行该程序，系统不会复位。如果调整喂狗周期大于 10 秒，程序将会引起系统复位。

程序清单 4.25 看门狗用法简单范例

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <linux/watchdog.h>
#define WDT "/dev/watchdog"
```

```
int main(void)
{
    int wdt_fd = -1;
    int timeout;

    wdt_fd = open(WDT, O_WRONLY);
    if (wdt_fd == -1) {
        printf("fail to open \"WDT \"!\n");
    }
    printf(WDT " is opened!\n");

    timeout = 10;
    ioctl(wdt_fd, WDIOC_SETTIMEOUT, &timeout);

    ioctl(wdt_fd, WDIOC_GETTIMEOUT, &timeout);
    printf("The timeout was is %d seconds\n", timeout);

    #if 1 //循环喂狗
        while(1) {
            write(wdt_fd, "\0", 1);
            sleep(9); //喂狗时间为 9 秒，小于设定的 10 秒；如果喂狗时间大于超时时间，将会发生看门狗
            复位
        }
    #else //关闭看门狗
        write(wdt_fd, "V", 1);
        close(wdt_fd);
        printf( WDT " is closeed!\n");
    #endif

    return 0;
}
```

### 1.33 SPI接口

本产品以排针引脚的方式引出了 SPI2、SPI3 接口。287 产品支持 SPI3，283 产品则不支持。



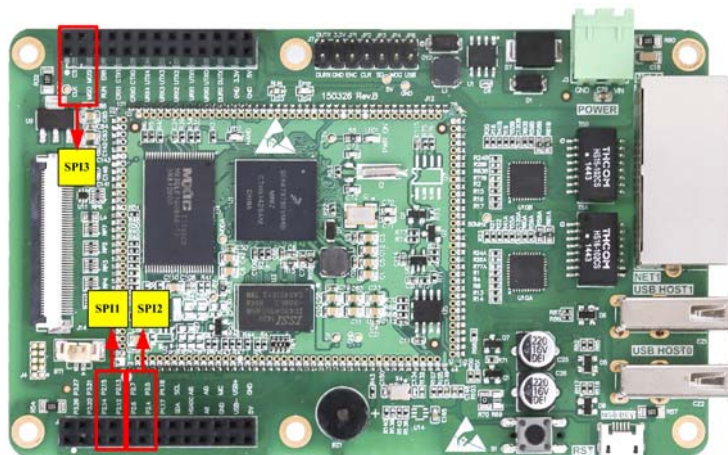


图 4.5 开发板上的 SPI 接口位置图

SPI2 驱动设备文件名: /dev/spidev1.0。SPI3 驱动设备文件名: /dev/spidev2.0。

需要注意的是本产品处理器的 SPI 控制器只支持半双工的通信方式,在发送数据时不能接收数据,在接收数据时不能发送数据。另外,其 ioctl 函数使用的参数都来自 Linux-2.6.35.3/include/Linux/spi/spidev.h 文件。

注:对 SPI 的编程需要加入#include <spidev.h>。

### 1.33.1 open调用

在使用SPI设备驱动之前,请使用open调用打开驱动设备文件,获得文件描述符,如程序清单 4.26所示。

程序清单 4.26 打开 SPI 设备文件

```
fd = open("/dev/spidev1.0", O_RDWR);
if (fd < 0) {
    printf("can not open SPI device\n");
}
```

### 1.33.2 ioctl调用

Linux 的 SPI 驱动为用户提供了相当全面的命令,通过这些命令用户可以配置 SPI 总线的时序、设置总线速率和实现全双工通信。

#### 1. 设置极性和相位

设置SPI极性及相位可以通过调用ioctl()函数时传递SPI\_IOC\_WR\_MODE命令参数实现,如表 4.8所示。

表 4.8 SPI\_IOC\_WR\_MODE 命令

命 令	SPI_IOC_WR_MODE
调用方式	ret = ioctl(fd, SPI_IOC_WR_MODE, &mode);
功能描述	设置 SPI 总线的极性和相位
参数说明	mode 类型: U32 可选值: SPI_MODE_0、SPI_MODE_1、SPI_MODE_2、SPI_MODE_3; 关于 SPI 总线极性和相位的 4 种模式请参考 SPI 协议
返回值说明	0: 设置成功 1: 设置不成功

## 2. 读取极性和相位

读取SPI极性及相位设置模式可以通过调用ioctl()函数时传递SPI\_IOC\_RD\_MODE命令参数实现，如表 4.9所示。

表 4.9 SPI\_IOC\_RD\_MODE 命令

命 令	SPI_IOC_RD_MODE
调用方式	ret = ioctl(fd, SPI_IOC_RD_MODE, &mode);
功能描述	读取 SPI 总线的极性和相位设置模式
参数说明	mode 类型: U32 参数返回值为: SPI_MODE_0、SPI_MODE_1、SPI_MODE_2、SPI_MODE_3; 关于 SPI 总线极性和相位的 4 种模式请参考 SPI 协议。
返回值说明	恒为 0: 读取成功

## 3. 设置每字的数据位长度

设置SPI总线上每字的数据位长度可以通过调用ioctl()函数时传递SPI\_IOC\_WR\_BITS\_PER\_WORD命令参数实现，如表 4.10所示。

表 4.10 SPI\_IOC\_WR\_BITS\_PER\_WORD 命令

命 令	SPI_IOC_WR_BITS_PER_WORD
调用方式	ret = ioctl(fd, SPI_IOC_WR_BITS_PER_WORD, &bits);
功能描述	设置 SPI 总线上每字的数据位长度
命令参数说明	bits 类型: U32 取值可在 1~N 之间
返回值说明	恒为 0: 设置成功

## 4. 设置最大总线速率

设置SPI总线的最大速率可以通过调用ioctl()函数时传递SPI\_IOC\_WR\_MAX\_SPEED\_HZ命令参数实现，如表 4.11所示。

表 4.11 SPI\_IOC\_WR\_MAX\_SPEED\_HZ

命 令	SPI_IOC_WR_MAX_SPEED_HZ
调用方式	ret = ioctl(fd, SPI_IOC_WR_MAX_SPEED_HZ, &speed);
功能描述	设置 SPI 总线的最大速率

命令参数说明	speed 类型: U32 单位为 Hz, 取值可在 1~N 之间
返回值说明	恒为 0: 设置成功

## 5. 数据接收/发送操作命令

在调用ioctl()函数时传递SPI\_IOC\_MESSAGE(1)命令 (SPI\_IOC\_MESSAGE(1)是一个带参数的宏, 其在spidev.h文件定义)参数则可以实现SPI总线数据的收发, 该命令介绍如表 4.12 所示。

表 4.12 SPI\_IOC\_MESSAGE(1)命令

命 令	SPI_IOC_MESSAGE(1)
调用方式	ret = ioctl(fd, SPI_IOC_MESSAGE(1), &tr);
功能描述	实现在 SPI 总线接收/发送数据操作
命令参数说明	tr 类型: struct spi_ioc_transfer 参数 tr 是 struct spi_ioc_transfer 结构的类型, 用于封装要接收/发送的数据, 详细请阅读下文。
返回值说明	0: 操作成功 1: 操作失败

使用SPI\_IOC\_MESSAGE(1)命令进行接收/发送的数据都需要使用struct spi\_ioc\_transfer 结构体来封装, 该结构体的定义如程序清单 4.27所示。

程序清单 4.27 struct spi\_ioc\_transfer 结构体的定义

```
struct spi_ioc_transfer {
    __u64      tx_buf;           //指向要发送数据的缓冲区
    __u64      rx_buf;           //指向要接收数据的缓冲区
    __u32      len;              // 发送数据和接收数据缓冲区中数据的长度
    __u32      speed_hz;         // 发送/接收这些数据需要的总线速率
    __u16      delay_usecs;
    __u8       bits_per_word;    // 发送/接收这些数据在 SPI 总线上, 每字是多少位
    __u8       cs_change;
    __u32      pad;
}
```

len 是指 tx\_buf 和 rx\_buf 所指向的缓冲区长度。

speed\_hz 不能大于 SPI\_IOC\_WR\_MAX\_SPEED\_HZ 的总线速率。

由于本产品处理器的 SPI 控制器只支持半双工, 因此 struct spi\_ioc\_transfer 结构体中的 tx\_buf 和 rx\_buf 只能设置一个有效, 另一个必须设置为 0, 否则调用 ioctl 时会返回 1 提示操作错误。

### 1.33.3 示例代码

程序清单 4.28是Linux源码中自带的SPI测试代码, 做了一些修改后可以读取 MX25L1635E型号的spiflash的ID。读取MX25L1635E ID的命令码为 0x9F, 其ID为 0XC22515, 因此在SPI接口上接上MX25L1635E器件运行此测试程序将会看到读取回来的数据为 C22515。

MX25L1635E是一款支持四线通讯的SPI Flash，通讯时钟高达 75MHz，64Mb存储容量被划分为 4K与 64K两组扇区，且支持扇区及块除擦。其电路如图 4.6所示：

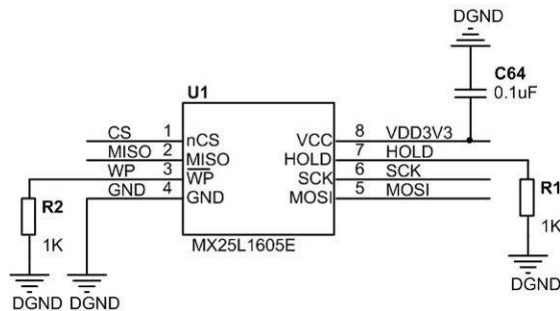


图 4.6 SPI Flash 信号连接图

上图中，SPIFI\_CS、SPIFI\_MISO\_SIO1、SPIFI\_MOSI\_SIO0、SPIFI\_SCK 四个引脚分别连接开发板的 SS0、MISO2、MOSI2、SCK2 四个管脚，出电源外，其他管脚可悬空或接地。

程序清单 4.28 SPI 测试代码

```
#include <stdint.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <Linux/types.h>
#include "spidev.h"

#define ARRAY_SIZE(a) (sizeof(a) / sizeof((a)[0]))

static void pabort(const char *s)
{
    perror(s);
    abort();
}

static const char *device = "/dev/spidev1.0";
static uint8_t mode = 0;
static uint8_t bits = 8;
static uint32_t speed = 50000;
static uint16_t delay;
#define WRITE 0
static void transfer(int fd)
{
    int ret;
    int i = 10000;
    uint8_t tx[] = {
```

```

        0x9f, 0x00, 0x00, 0x00, 0x01, 0x02,
    };
    uint8_t rx[ARRAY_SIZE(tx)] = {0, };
    struct spi_ioc_transfer tr_txx[] = {
        {
            .tx_buf = (unsigned long)tx,                /* 发送数据缓存区 */
            .rx_buf = 0,                                /* 半双工通信，接收缓存区置为 0 */
            .len = 1,                                    /* 发送命令码，1 个字节 */

            .delay_usecs = delay,
            .speed_hz = speed,
            .bits_per_word = bits,
        },
        {
            .rx_buf = (unsigned long)rx,                /* 接收数据缓存区 */
            .len = 3,                                    /* 接收数据长度为 3 */

            .delay_usecs = delay,
            .speed_hz = speed,
            .bits_per_word = bits,
        }
    };
    ret = ioctl(fd, SPI_IOC_MESSAGE(2), &tr_txx[0]);    /* 发送 2 条消息 */
    if (ret == 1) {
        perror("can't receive spi message");
    }
    for (ret = 0; ret < tr_txx[1].len; ret++) {
        if (!(ret % 6))
            puts("");
        printf("%.2X ", rx[ret]);
    }
    puts("");
}

void print_usage(const char *prog)
{
    printf("Usage: %s [-DsbdlHOLC3]\n", prog);
    puts("  -D --device    device to use (default /dev/spidev1.0)\n"
        "  -s --speed     max speed (Hz)\n"
        "  -d --delay     delay (usec)\n"
        "  -b --bpw       bits per word\n"
        "  -l --loop      loopback\n"
        "  -H --cpha      clock phase\n"
        "  -O --cpol      clock polarity\n"
        "  -L --lsb       least significant bit first\n");
}

```

```
" -C --cs-high   chip select active high\n"
" -3 --3wire     SI/SO signals shared\n");
exit(1);
}

void parse_opts(int argc, char *argv[])
{
    while (1) {
        static const struct option lopts[] = {
            { "device",  1, 0, 'D' },
            { "speed",   1, 0, 's' },
            { "delay",   1, 0, 'd' },
            { "bpw",     1, 0, 'b' },
            { "loop",    0, 0, 'l' },
            { "cpha",    0, 0, 'H' },
            { "cpol",    0, 0, 'O' },
            { "lsb",     0, 0, 'L' },
            { "cs-high", 0, 0, 'C' },
            { "3wire",   0, 0, '3' },
            { "no-cs",   0, 0, 'N' },
            { "ready",   0, 0, 'R' },
            { NULL, 0, 0, 0 },
        };

        int c;

        c = getopt_long(argc, argv, "D:s:d:b:lHOLC3NR", lopts, NULL);
        if (c == -1) {
            break;
        }

        switch (c) {
        case 'D':
            device = optarg;
            break;
        case 's':
            speed = atoi(optarg);
            break;
        case 'd':
            delay = atoi(optarg);
            break;
        case 'b':
            bits = atoi(optarg);
            break;
        case 'l':
            mode |= SPI_LOOP;
```

```

        break;
    case 'H':
        mode |= SPI_CPHA;
        break;
    case 'O':
        mode |= SPI_CPOL;
        break;
    case 'L':
        mode |= SPI_LSB_FIRST;
        break;
    case 'C':
        mode |= SPI_CS_HIGH;
        break;
    case '3':
        mode |= SPI_3WIRE;
        break;
    case 'N':
        mode |= SPI_NO_CS;
        break;
    case 'R':
        mode |= SPI_READY;
        break;
    default:
        print_usage(argv[0]);
        break;
    }
}

/*
 * 示例程序为读 MX25L1635E spiflash 的 id 功能，接上 MX25L1635E 器件并运行此测试程序，将会读取
 * 器件的 ID 为 0XC22515
 */
int main(int argc, char *argv[])
{
    int ret = 0;
    int fd;

    parse_opts(argc, argv);
    fd = open(device, O_RDWR);
    if (fd < 0) {
        pabort("can't open device");
    }
}
/*

```



```

    * spi mode
    */

    ret = ioctl(fd, SPI_IOC_WR_MODE, &mode);
    if (ret == -1) {
        pabort("can't set wr spi mode");
    }
    ret = ioctl(fd, SPI_IOC_RD_MODE, &mode);
    if (ret == -1) {
        pabort("can't get spi mode");
    }
    /*
    * bits per word
    */
    ret = ioctl(fd, SPI_IOC_WR_BITS_PER_WORD, &bits);
    if (ret == -1) {
        pabort("can't set bits per word");
    }
    ret = ioctl(fd, SPI_IOC_RD_BITS_PER_WORD, &bits);
    if (ret == -1) {
        pabort("can't get bits per word");
    }
    /*
    * max speed hz
    */
    ret = ioctl(fd, SPI_IOC_WR_MAX_SPEED_HZ, &speed);
    if (ret == -1) {
        pabort("can't set max speed hz");
    }
    ret = ioctl(fd, SPI_IOC_RD_MAX_SPEED_HZ, &speed);
    if (ret == -1) {
        pabort("can't get max speed hz");
    }
    printf("spi mode: %d\n", mode);
    printf("bits per word: %d\n", bits);
    printf("max speed: %d Hz (%d KHz)\n", speed, speed/1000);
    sleep (1);
    transfer(fd);
    close(fd);
    return ret;
}

```

执行程序后，显示如下：

```
spi mode: 0
bits per word: 8
max speed: 500000 Hz (500 KHz)

FF FF FF FF FF FF
40 00 00 00 00 95
FF FF FF FF FF FF
FF FF FF FF FF FF
FF FF FF FF FF FF
DE AD BE EF BA AD
F0 0D
```

图 4.7 SPI 示例执行效果

## 5 QT 4 编程

### 1.34 背景知识

在阅读本章前，如果对下面所列举的知识点有一定的了解，将有助于更好的理解本章内容。

- ◆ C++基础知识，了解简单的类，继承，重载等面向对象概念；
- ◆ Linux 基础知识，了解基本的 Shell 命令，懂得对 Linux 进行简单的配置；
- ◆ 嵌入式开发基础知识，了解基本的嵌入式开发流程，了解简单的嵌入式开发工具的使用，如调试串口的使用，NFS 文件系统的挂载方法；
- ◆ 交叉编译与动态库的基础知识。

### 1.35 Qt介绍

#### 1.35.1 Qt简介

Qt 是一个跨平台应用程序和 UI 开发框架。使用 Qt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。Qt原为奇趣科技公司（Trolltech, [www.trolltech.com](http://www.trolltech.com)）开发维护，现在被nokia公司收购。目前在nokia的推动下，Qt的发展非常快速，版本不断更新。本产品所用的Qt主版本为 4.7，所支持的平台如图 5.1所示。

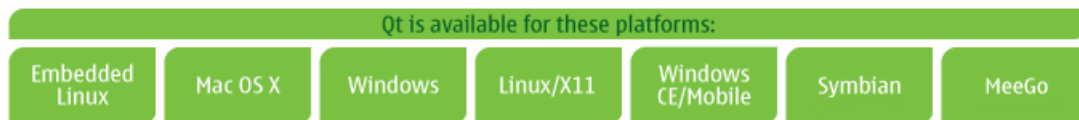


图 5.1 Qt 支持的平台

#### 1.35.2 Qt/E简介

Qt/E 在原始 Qt 的基础上，做了许多出色的调整以适合嵌入式环境。同 Qt/X11 相比，Qt/E 很节省内存，因为它不需要 X server 或是 Xlib 库，它在底层摒弃了 Xlib，采用 framebuffer 作为底层图形接口。Qt/E 的应用程序可以直接写内核帧缓冲，因此它在嵌入式 linux 系统上的应用非常广泛。

### 1.36 编译环境的搭建

#### 1.36.1 编译Qt-4.7.3 源码包

解压 qt-everywhere-opensource-src-4.7.3.tar.bz2，并进入 qt-everywhere-opensource-src-4.7.3 目录。先修改其中的 built-qt 文件的 prefix，设定合适的编译结果输出目录。然后运行 build-qt 文件，将会完成 Qt 的编译。如果主机缺少某些依赖库导致编译失败，请用 apt-get 安装相应的依赖库后重新编译。

#### 1.36.2 编译环境的设置

为了能使用编译得到的 Qt 环境，需要设置环境变量，至少需要将 qt-everywhere-opensource-src-4.7.3/bin 目录添加到系统目录中。该目录有编译 Qt 应用程序所需要的 qmake 等工具。如果不添加到系统目录，则在每次使用 qmake 之前，通过 export 命

令设置 qmake 所在路径也是可以的:

```
$ export PATH=/path/to/qt-4.7.3/bin: $PATH
```

在提供的范例文件夹下有一个 set-env 脚本，可根据实际环境进行修改，然后运行该文件即可:

```
$ . set-env (点+空格+set-env)
```

在终端中输入 qmake -v 命令，若能看到类似如下输出，则工具链安装成功。

```
chenxibing@linux-compiler: ~$ qmake -v
```

```
QMake version 2.01a
```

```
Using Qt version 4.7.3 in /home/chenxibing/work/aM3352/M3352/qt/qt-everywhere-opensource-src-4.7.3/qt/lib
```

## 1.37 Hello world

### 1.37.1 编译hello程序

下面介绍如何开发一个简单的hello world程序。其流程如图 5.2所示。

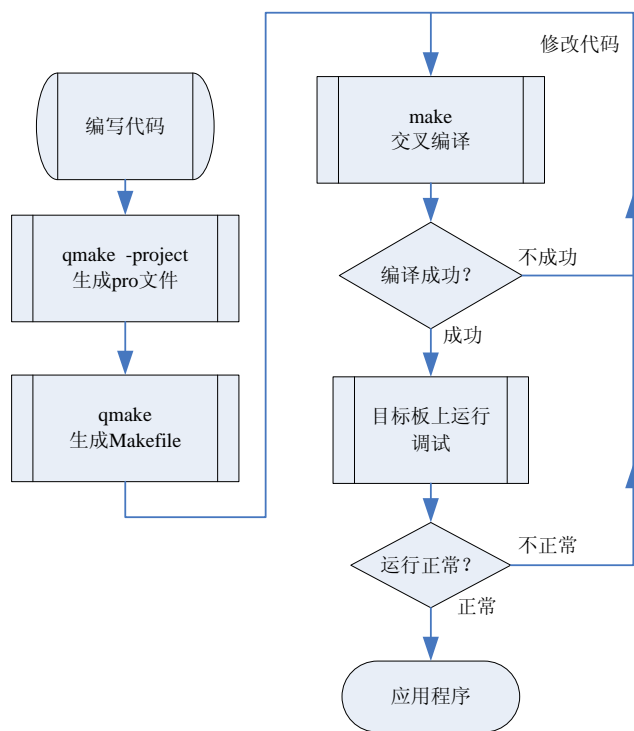


图 5.2 hello 开发流程图

hello.cpp程序代码如程序清单 5.1所示。

程序清单 5.1 hello 程序代码

```
#include <QtGui>
int main(int argc, char * argv[])
{
    QApplication app(argc, argv);
    QLabel label("hello world");
    label.show();
}
```

```
return app.exec();  
}
```

将 hello.cpp 拷贝至 hello 目录下,运行如下命令(注意,使用 qmake 之前需要先设置 qmake 的路径) :

```
chenxibing@linux-compiler: ~$ qmake -project
```

生成 hello.pro 文件。(注: qmake 是 Qt 中用来管理工程的项目工具, pro 文件则描述整个工程所包含的源码与相应的资源文件。有关于 qmake 与 pro 文件的相关信息将在下一节介绍。)

```
chenxibing@linux-compiler: ~$ qmake
```

根据上一步的 pro 文件,生成 Makefile 文件。

```
chenxibing@linux-compiler: ~$ make
```

根据 Makefile 编译出可执行程序。以后再编译时,只需执行最后一步 make。

经过上述步骤,可以在 hello 目录下见到 hello 程序,这个程序就是我们所希望得到的可执行程序。

### 1.37.2 在目标板上运行hello程序

下面介绍如何在目标板上运行 hello 程序。

建议通过 nfs 挂载 PC 主机上的目录至目标板上,便于调试开发。具体的 nfs 挂载方法请参阅相关资料。下面假设已经将 PC 上 hello 目录挂载至目标板,接下来的所有操作都在目标板上进行。

在启动 hello 程序前,需要先设定其鼠标设备。可通过如下命令指定触摸屏为 Qt 的鼠标设备。

```
[root@M283 ~]# export QWS_MOUSE_PROTO=Tslib:/dev/input/event1
```

上述命令中 Tslib 指定了触摸屏对应的设备文件,这里指定为/dev/input/event1。但其值并不固定,需要根据实际情况确定。正常情况下,所需的设备文件位于/dev/input 目录下。在命令行下输入如下命令:

```
[root@M283 ~]$ cat /dev/input/event0 | hexdump
```

点击触摸屏,如果有数据输出,那么对应的设备文件可能就是所需的设备文件。

如果需要使用 usb 鼠标,可如下设置:

```
[root@M283 ~]# export QWS_MOUSE_PROTO=LinuxInput:/dev/input/event1
```

与触摸屏类似,也需要根据具体情况确认 usb 鼠标所对应的设备文件,方法与触屏类似。

在成功设定鼠标设备后,可以如下启动 Qt 程序。

```
[root@M283 qtdemo]# ./hello -qws
```

-qws 指明这个 Qt 程序同时作为一个窗口服务器运行,在目标板上启动的第一个 Qt 程序应使用此参数启动。

在本例中,程序启动成功后,界面如图 5.3所示:



图 5.3 hello 程序运行界面

### 1.38 qmake与pro文件

qmake 是一个用来为不同平台和编译器生成 Makefile 的工具。手写 Makefile 是比较困难并且容易出错的，尤其是需要给不同的平台和编译器组合写几个 Makefile。使用 qmake，编程人员只需创建一个简单的“pro 文件”并且运行 qmake 即可生成恰当的 Makefile。

对于某些简单的项目，可以在其项目顶层目录下直接 qmake -project 自动生成“pro 文件”，例如 hello 程序。但对于一些复杂的 Qt 程序，自动生成的“pro 文件”可能并不符合要求，这时就需要程序员手动改写“pro 文件”。因此，下面就简单介绍“pro 文件”相关内容。

pro 文件主要有三种模板：

- app（应用程序模板）
- lib（库模板）
- subdirs（递归编译模板）。

在 pro 文件中可以通过以下代码指定所使用的模板。

```
TEMPLATE = app
```

如果不指定 TEMPLATE，pro 文件默认为 app 模式。项目中使用最多也是 app 模式。app 模式的 pro 文件主要用于构造适用于应用程序的 Makefile。

#### 1.38.1 pro文件例程

下面通过一个例子简单地介绍 app 模式下 pro 文件（关于 lib 与 subdirs 模式的 pro 文件，用户可以参看 qmake 的相关文档）。这个 pro 文件内容将完全手动编写。在实际的项目中，程序员可以使用 qmake -project 生成 pro 文件，再在这个 pro 文件上进行相应修改。

假设我们的项目中有如下源代码文件：

- hello.cpp
- hello.h
- main.cpp

首先，我们需要在 pro 文件中指定 cpp 文件，可以通过 SOURCES 变量指定，代码如下：

```
SOURCES += hello.cpp
```

对于每一个 `cpp` 文件，都需要如此指定。代码如下：

```
SOURCES += hello.cpp
```

```
SOURCES += main.cpp
```

也可以通过反斜线形式指定：

```
SOURCES = hello.cpp \
```

```
main.cpp
```

下来需要指定所需的 `h` 文件，通过 `HEADERS` 指定。pro 文件中代码如下：

```
HEADERS += hello.h
```

```
SOURCES += hello.cpp
```

```
SOURCES += main.cpp
```

项目生成的可执行程序文件名会自动设置，程序文件名与 `pro` 文件名一致，但在不同的平台下，其扩展名是不同的。比如 `pro` 文件名为 `hello.pro`，在 `windows` 平台下，其会生成 `hello.exe`；在 `linux` 平台下，会生成 `hello`。可以使用 `TARGET` 指定可执行程序的基本文件名。代码如下：

```
TARGET = helloworld
```

接下来最后一步便是设置 `CONFIG` 变量。由于此项目为一个 `Qt` 项目，因此要将 `qt` 添加到 `CONFIG` 变量中，以告知 `qmake` 将 `Qt` 相关的库与头文件信息添加到 `Makefile` 文件中。现在完整的 `pro` 文件内容如下所示：

```
CONFIG += qt
```

```
HEADERS += hello.h
```

```
SOURCES += hello.cpp
```

```
SOURCES += main.cpp
```

```
TARGET = helloworld
```

现在就可以利用此 `pro` 文件生成 `Makefile`，命令如下：

```
chenxibing@linux-compiler: ~$ qmake -o Makefile hello.pro
```

如果当前目录下只有一个 `pro` 文件，可以直接使用命令：

```
chenxibing@linux-compiler: ~$ qmake
```

在生成 `Makefile` 文件后，即可使用 `make` 命令进行编译。

### 1.38.2 pro文件常见配置

对于 `app` 模式的 `pro` 文件，常用的变量有下面这些：

- `HEADERS` 指定项目的头文件 (\*.h)
- `SOURCES` 指定项目的 C++ 文件 (\*.cpp)
- `FORMS` 指定需要 `uic` 处理的由 `Qt designer` 生成的 \*.ui 文件
- `RESOURCES` 指定需要 `rcc` 处理的 \*.qrc 文件
- `DEFINES` 指定预定义的 C++ 预处理器符号
- `INCLUDEPATH` 指定 C++ 编译器搜索全局头文件的路径
- `LIBS` 指定工程要链接的库。
- `CONFIG` 指定各种用于工程配置和编译的参数
- `QT` 指定工程所要使用的 `Qt` 模块（默认是 `core gui`，对应于 `QtCore` 和



QtGui)

- TARGET 指定可执行文件的基本文件名
- DESTDIR 指定可执行文件放置的目录

CONFIG 变量用于控制编译过程中的各个方面。常用参数如下:

- debug 编译出具有调试信息的可执行程序。
- release 编译不带调试信息的可执行程序, 与 debug 同时存在时, release 失效。
- qt 指应用程序使用 Qt。此选项是默认包括的。
- dll 动态编译库文件
- staticlib 静态编译库文件
- console 指应用程序需要写控制台

## 1.39 桌面版本的Qt SDK使用简介

### 1.39.1 桌面版本Qt SDK简介

Qt 是一个跨平台的图形框架, 在安装了桌面版本的 Qt SDK 的情况下, 用户可以先在 PC 主机上进行 Qt 应用程序的开发调试, 待应用程序基本成型后, 再将其移植到目标板上。

桌面版本的 Qt SDK 主要包括以下两个部分:

- 用于桌面版本的 Qt 库
- Qt Creator (集成开发环境)

Qt Creator是一个强大的跨平台IDE, 集编辑, 编译, 运行, 调试功能于一体。其代码编辑器支持关键字高亮, 上下文信息提示, 自动完成, 智能重命名等高级功能。IDE中集成的可视化界面编辑器, 可以让用户以所见即所得的方式进行图形程序的设计。其编译, 运行无需敲入命令, 直接点击按钮或使用快捷键即可完成。同时还支持图形化的调试方式, 可以以插入断点, 单步运行, 追踪变量, 查看函数堆栈等方式进行应用程序的调试开发。Qt Creator主界面如图 5.4所示。



图 5.4 Qt Creator 主界面

### 1.39.2 桌面版本Qt SDK的安装

桌面版本的 Qt SDK 支持三个平台：Windows、Linux、Mac。这里只讲述 Linux 桌面版本的 Qt SDK 的安装。其他平台下的安装可参阅官方资料。用户可以在 Qt 官方网站找到三个平台下对应的安装包。推荐通过 ubuntu 下的 apt-get 获取 Linux 版的 Qt SDK。使用如下命令获取 SDK:

```
chenxibing@linux-compiler: ~$ sudo apt-get install qt-sdk
```

### 1.39.3 Qt Creator配置

通过如下命令启动 Qt Creator。

```
chenxibing@linux-compiler: ~$ qtcreator
```

启动后将得到如图 5.4所示界面。使用之前需要先设置Qt Creator所调用的Qt版本。点击如图 5.4菜单栏上“工具”→“选项”。得如类似图 5.5界面。

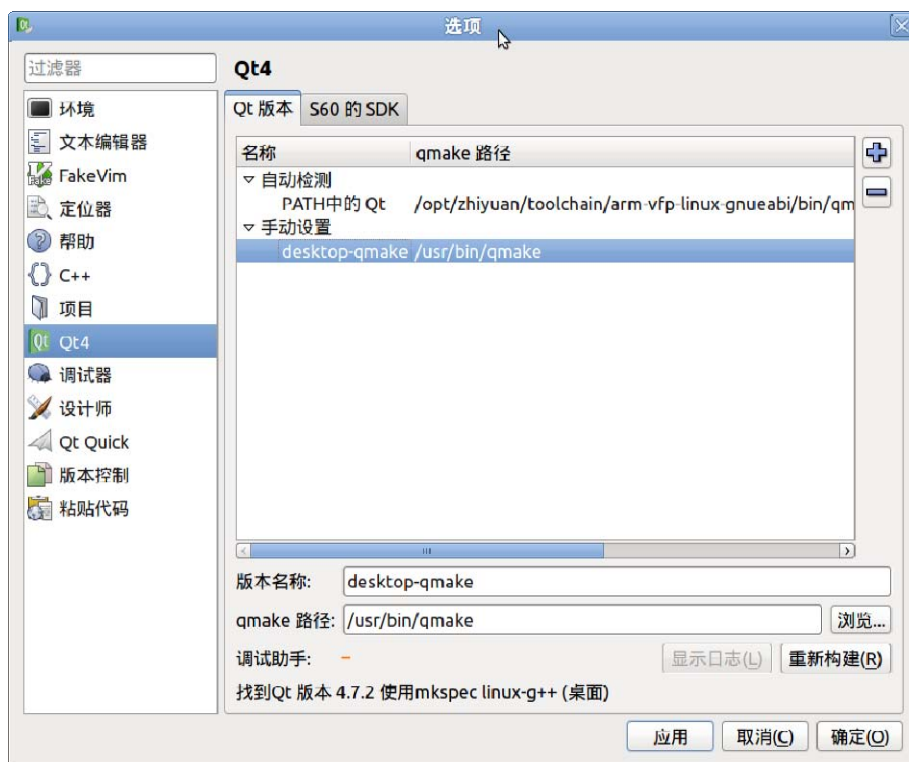


图 5.5 选项界面

在不同的环境中可能得到的结果有差异，如图 5.5所示的界面，Qt Creator自动检测到的Qt工具（qmake）是用于嵌入式环境的版本，因此需要手动添加用于桌面版本的Qt配置。单击图 5.5中的“加号”，添加用于桌面版本的Qt配置。用于桌面版本的qmake路径为 /usr/bin/qmake。这里将桌面版本的Qt配置名称设置为desktop-qmake。添加成功后按确定，返回图 5.4界面。

### 1.39.4 Qt Creator使用例程

下面将通过一个简单的例程讲解如何使用 Qt Creator 来进行 Qt 程序的开发。

单击界面中“创建项目”按钮，会得到如图 5.6所示的界面。

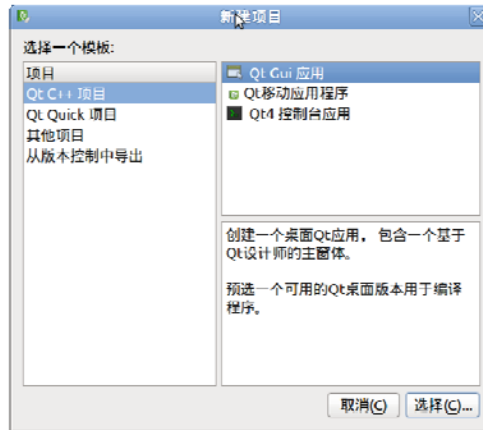


图 5.6 新建项目界面

如图 5.6所示进行项目模板的选择，使用默认的“Qt Gui应用”模板。点击选择，进入如图 5.7界面，设置项目名称与路径。接下来一路点击“下一步”直到如图 5.8界面。



图 5.7 项目介绍和位置界面



图 5.8 项目管理界面

单击“完成”后，将进入如图 5.9界面。可以看到，Qt Creator已经自动生成一个最基本Qt程序所需的代码，用户可以在此基础上做进一步的开发。

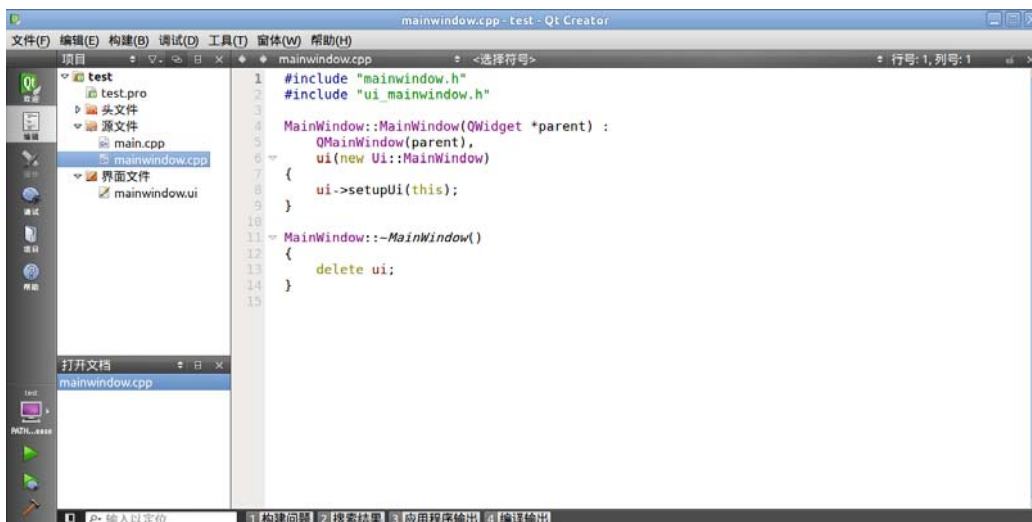


图 5.9 mainwindow.cpp 界面

单击项目侧边栏中mainwindow.ui可以启动可视化编辑器，如图 5.10所示。

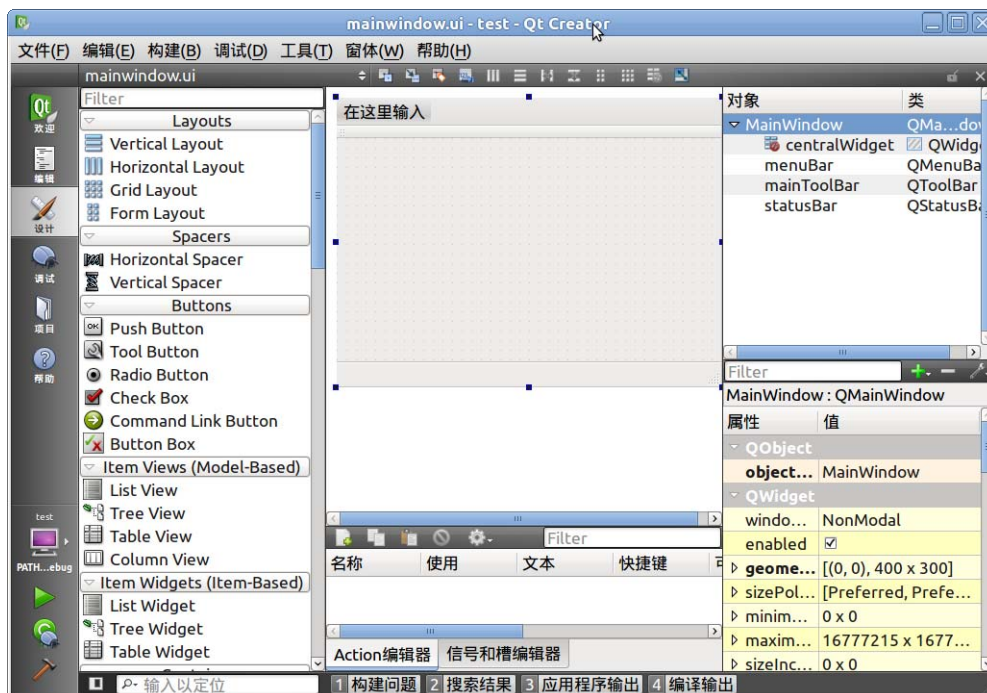


图 5.10 可视化界面编辑器

在图 5.10的界面中，通过拖动控件侧边栏中控件到程序主页面中，以所见即所得的方式设计程序界面。下面拖动一个QLabel控件到程序主页面上，并设置QLabel上文字为“Hello World”。如图 5.11：

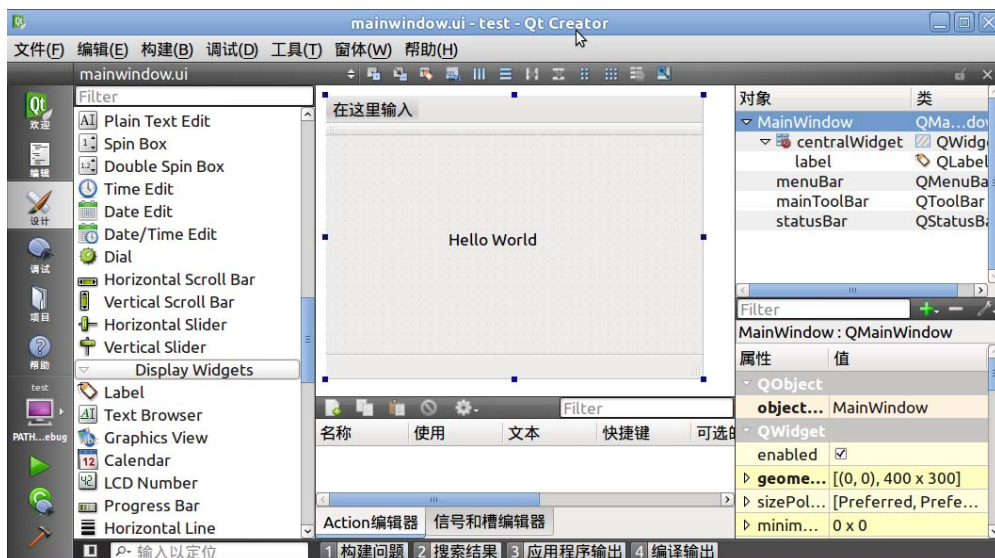


图 5.11 hello world 程序界面

接下来通过侧边栏上  选择前面所设置桌面版本的Qt。如图 5.12所示：

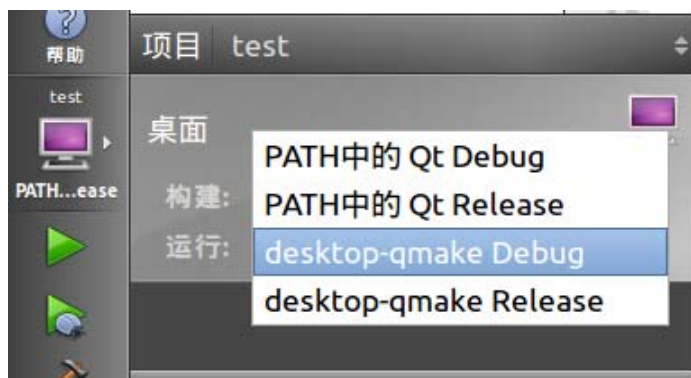



图 5.12 选择 Qt 版本

最后，点击  按钮对程序进行编译链接运行。如编译无误，将自动启动应用程序。界面如图 5.13所示：

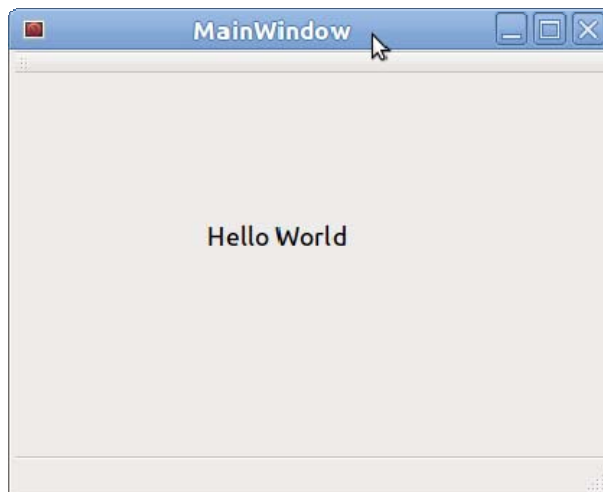




图 5.13 hello world 界面

### 1.39.5 移植hello world

由于 Qt 良好的可移植性，在桌面版本 Qt SDK 中编译运行成功的应用程序，一般只需重新交叉编译，便可在目标板上运行。在本例中，只需执行以下命令，重新交叉编译，便可获得嵌入式版本的 Qt 程序。

```
chenxibing@linux-compiler: ~$ cd /home/chenxibing/test
chenxibing@linux-compiler: ~$ qmake -project
chenxibing@linux-compiler: ~$ qmake
chenxibing@linux-compiler: ~$ make
```

### 1.40 zylauncher图形框架

zylauncher是ZLG嵌入式Linux上的一个简单的图形演示框架。整体界面采用当前流行的宫格界面。用户可以将自己的程序添加到演示框架中，直接从GUI界面启动程序。界面如图 5.14。在界面上可以放置三种类型的按钮图标。按钮图标类型如下：

- 菜单图标：点击可以切换新的宫格界面（菜单界面）。
- 程序图标：点击可以启动演示程序。
- 退出图标：点击将退出整个演示框架。

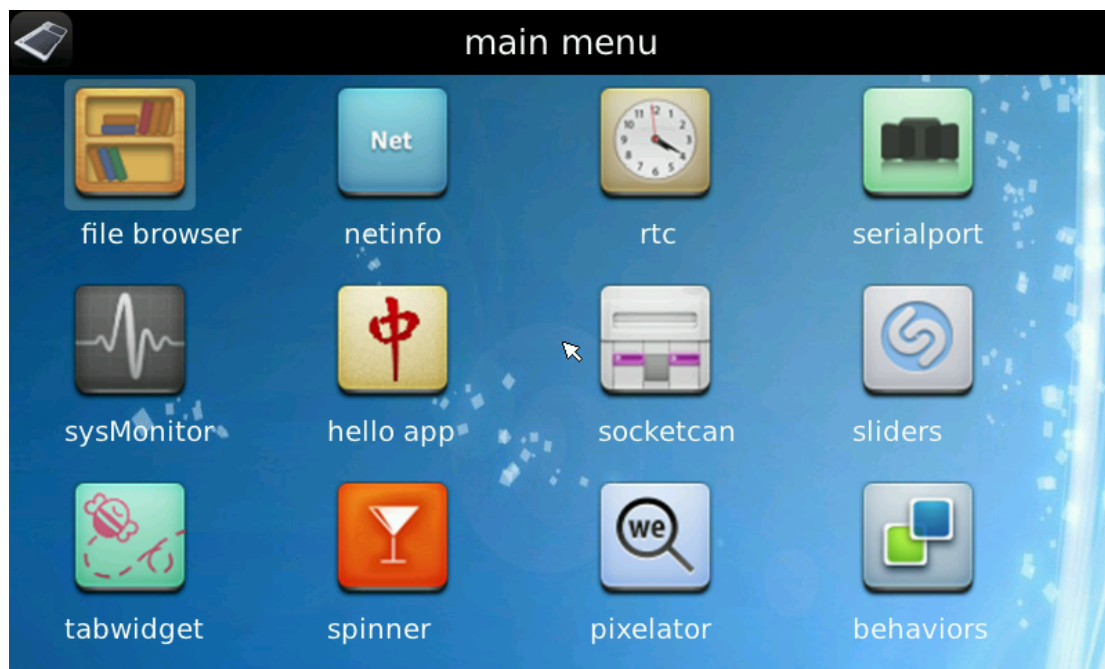


图 5.14 zylauncher 界面

演示框架主体是采用 qml 语言描述，用户可以通过修改 qml 文件，进行界面的配置。如果按钮图标过多，还可以新建菜单界面，以容纳更多的按钮图标。

演示框架目录结构如图 5.15所示：

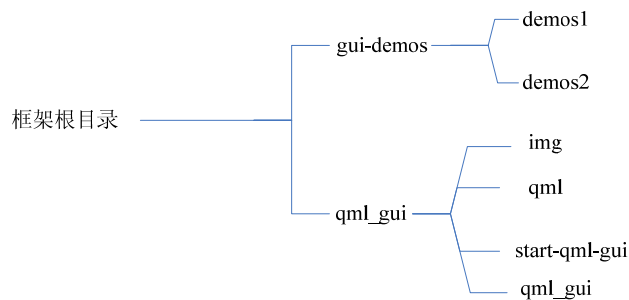


图 5.15 zylauncher 框架结构

根目录下的 **gui-demos** 中放置的是演示程序的可执行文件和相关资源文件，如果用户希望将自己的程序添加到框架中，需要把程序的可执行文件和相关资源文件放置在 **gui-demos** 目录下。

**qml\_gui** 目录下放置的是框架相关文件。**img** 下放置演示程序对应的按钮图标文件；**start-qml-gui** 与 **qml\_gui** 为框架的可执行文件，其中 **start-qml-gui** 为整个框架的启动脚本，在将框架根目录拷贝至目标板文件系统后，可使用 **start-qml-gui** 启动框架；**qml** 目录下放置着描述框架界面的 **qml** 文件，用户可以通过修改 **qml** 下文件来对界面进行配置。

下面以 **qml** 目录下的 **MainMenu.qml** 文件来讲解如何通过修改 **qml** 文件来对界面进行配置。**MainMenu.qml** 对应的是框架的主页面，即图 5.14。**MainMenu.qml** 代码如程序清单 5.2 所示：

程序清单 5.2 MainMenu.qml 文件

```

import QtQuick 1.0
import "func.js" as Logic
Rectangle {
    width: rootloader.viewWidth
    height: rootloader.viewHeight
    color: "black"
    // 界面标题
    Text {x:rootloader.titleX; y:0; text:"Main Menu"; font.pointSize:titleFontSize; color:"white"}
    // 第一行图标
    ProButton {
        x:Logic.indexToX(0); y:Logic.indexToY(0);
        imagePath: "../img/switch.png";
        imgText : "fluidlauncher";
        execText : "../gui-demos/fluidlauncher/start-fluidlauncher"
    }
    ProButton {
        x:Logic.indexToX(1); y:Logic.indexToY(0);
        imagePath: "../img/cpp.png";
        imgText : "syntax high";
        execText : "../gui-demos/syntaxhighlighter/start-syntaxhighlighter"
    }
    ProButton {

```



```

        x:Logic.indexToX(2); y:Logic.indexToY(0);
        imgPath: "../img/block.png";
        imgText : "move block";
        execText : "../gui-demos/moveblocks/start-moveblocks"
    }
    MenuButton{
        x:Logic.indexToX(3); y:Logic.indexToY(0);
        imgPath: "../img/cookie.png";
        imgText : "small_demos";
        menuName : "../SmallDemos.qml"
    }
    //      第二行图标
    ProButton {
        x:Logic.indexToX(0); y:Logic.indexToY(1);
        imgPath: "../img/clock.png";
        imgText : "clocks";
        execText : "../gui-demos/clocks/start-clocks"
    }
    ProButton {
        x:Logic.indexToX(1); y:Logic.indexToY(1);
        imgPath: "../img/qt.png";
        imgText : "pixelator";
        execText : "../gui-demos/pixelator/pixelator"
    }
    ProButton {
        x:Logic.indexToX(2); y:Logic.indexToY(1);
        imgPath: "../img/behavior.png";
        imgText : "behaviors";
        execText : "../gui-demos/behaviors/start-behaviors"
    }
    MenuButton{
        x:Logic.indexToX(3); y:Logic.indexToY(1);
        imgPath: "../img/button1.png";
        imgText : "qml_demos";
        menuName : "../QmlDemos.qml"
    }
    //      第三行图标
    ProButton {
        x:Logic.indexToX(0); y:Logic.indexToY(2);
        imgPath: "../img/stylesheet.png";
        imgText : "style sheet";
        execText : "../gui-demos/stylesheet/stylesheet"
    }
    ProButton {

```

```

        x:Logic.indexToX(1); y:Logic.indexToY(2);
        imgPath: "../img/sysMonitor.png";
        imgText : "sysMonitor";
        execText : "../gui-demos/sysMonitor/start-sysmonitor"
    }
    ProButton {
        x:Logic.indexToX(2); y:Logic.indexToY(2);
        imgPath: "../img/note.png";
        imgText : "dockWidgets";
        execText : "../gui-demos/dockwidgets/dockwidgets"
    }
    ExitButton{
        x:Logic.indexToX(3); y:Logic.indexToY(2);
        imgPath: "../img/exit.png";
        imgText : "    exit"
    }
}

```

下来关注代码:

```

ProButton {
    x:Logic.indexToX(0); y:Logic.indexToY(0);
    imgPath: "../img/switch.png";
    imgText : "fluidlauncher";
    execText : "../gui-demos/fluidlauncher/start-fluidlauncher"
}

```

**ProButton** 指示这是一个程序按钮图标，点击此图标可以启动一个演示程序，在大括号中的是有关于此按钮的属性信息。

**x,y:** 指定图标在屏上的横纵坐标。zylanucher 采用的是 3 行 4 列宫格界面，可通过 **Logic.indexToX(0)** 与 **Logic.indexToY(0)** 获得 0 行 0 列宫格格子的 **x, y** 坐标。

**imgPath:** 指定按钮的图标文件。

**imgText:** 指定按钮下方的描述文字。

**execText:** 指定按钮对应的演示程序可执行文件。

其中需要注意的一点，**imgPath** 与 **execText** 都是通过相对路径指定对应的文件，但它们的相对路径的基准是不同的。

设置 **imgPath** 时，"相对路径"相对的是"qml 文件所在路径"( **MenuButton** 中的 **menuName** 亦如此 )。

设置 **execText** 时，"相对路径"相对的是"当前工作路径"（当使用 **start-qml-gui** 启动程序时，其"当前工作路径"为 **start-qml-gui** 文件所在目录）。

接着关注代码:

```

MenuButton{
    x:Logic.indexToX(3); y:Logic.indexToY(1);
    imgPath: "../img/button1.png";
    imgText : "qml_demos";
}

```

```
menuName : "/QmlDemos.qml"  
}
```

**MenuButton** 是一个菜单按钮，点击此按钮，将切换到另一个菜单界面（宫格界面）。在大括号中的是有关于此按钮的属性信息。其属性与 **ProButton** 基本一致。唯一不同的是 **MenuButton** 没有 **ProButton** 中的 **execText** 属性，取而代之的是 **menuName** 属性。

**menuName**: 指定新菜单界面对应的 **qml** 文件。如上代码，指定 **QmlDemos.qml** 为新的菜单界面（可以在同级目录下找到 **QmlDemos.qml** 文件）。

最后关注的代码：

```
ExitButton{  
    x:Logic.indexToX(3); y:Logic.indexToY(2);  
    imagePath: "../img/exit.png";  
    imgText : "    exit"  
}
```

**ExitButton** 是一个退出按钮，点击此按钮，将退出演示框架。其属性参数较少，参数信息可参考 **ProButton**。

## 6 系统恢复和更新

### 1.41 TF恢复系统

本章主要讲述如何使用 TF 卡恢复和更新本产品的 Linux 固件，系统安装的步骤：

1. 制作 Nand Flash 格式化启动盘；
2. 格式化 Nand Flash；
3. 制作系统安装引导盘；
4. 安装 Linux 系统。

注：如果您的板子先前没有安装 WinCE 系统，那么就可以免去步骤 1 和步骤 2，直接进行步骤 3 和步骤 4。这里为了兼容一些老客户，特意保留了步骤 1 和步骤 2。

#### 1.41.1 制作Nand Flash格式化启动盘

把TF卡在PC机的读卡器上插接好，确定TF卡被格式化为FAT32，并记下其盘符，然后运行产品光盘“images\TF卡烧写方案\1-TF卡 Nand Flash格式化”目录下的sd\_os.bat，如图 6.1所示：

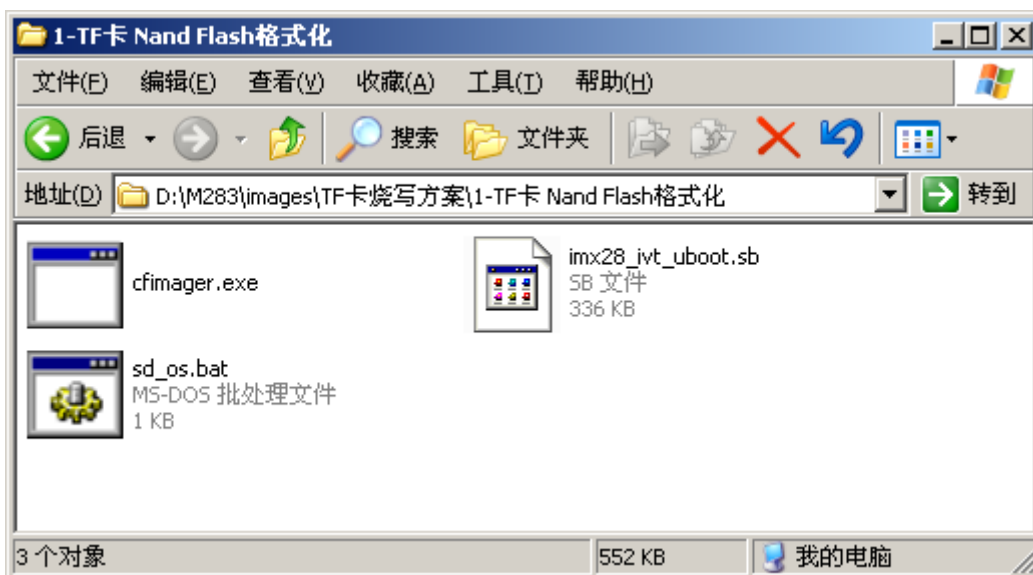


图 6.1 制作 TF 卡格式化启动盘

在图 6.2所示窗口内，输入TF卡的盘符，比如在当前系统下TF卡显示为F盘，输入F并回车，然后稍等一会，数据写入到TF卡完毕，启动盘就制作好了。

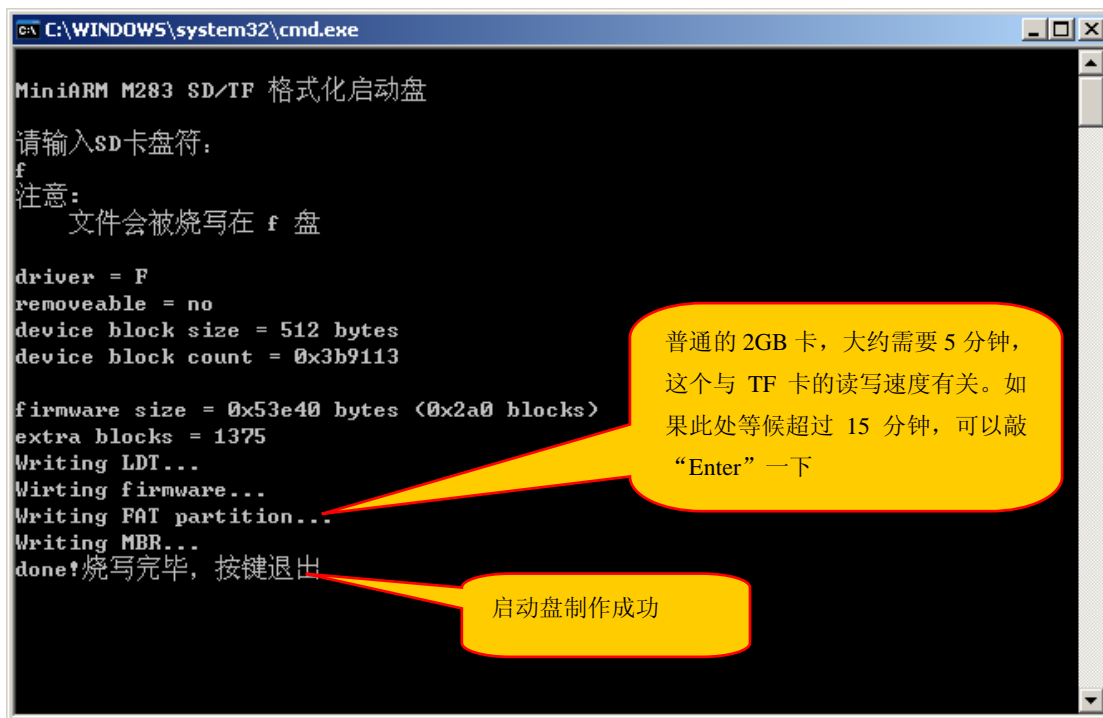


图 6.2 制作 TF 卡格式化启动盘运行结果

#### 1.41.2 格式化Nand Flash

设置系统启动模式为从TF卡启动，启动模式的设置见前文“[2.1.1 启动选择](#)”，此处不再赘述。把调试串口J7 和PC机连接好，打开PC机串口软件查看Nand Flash格式化结果，串口软件建议使用产品光盘“software”目录下的putty.exe。然后在底板的TF卡座插入在步骤 1 制作好的TF卡启动盘，再给板子上电，稍等待，板子启动后会自动格式化上面的Nand Flash。格式化完成后的结果，应该跟图 6.3所示的相类似。

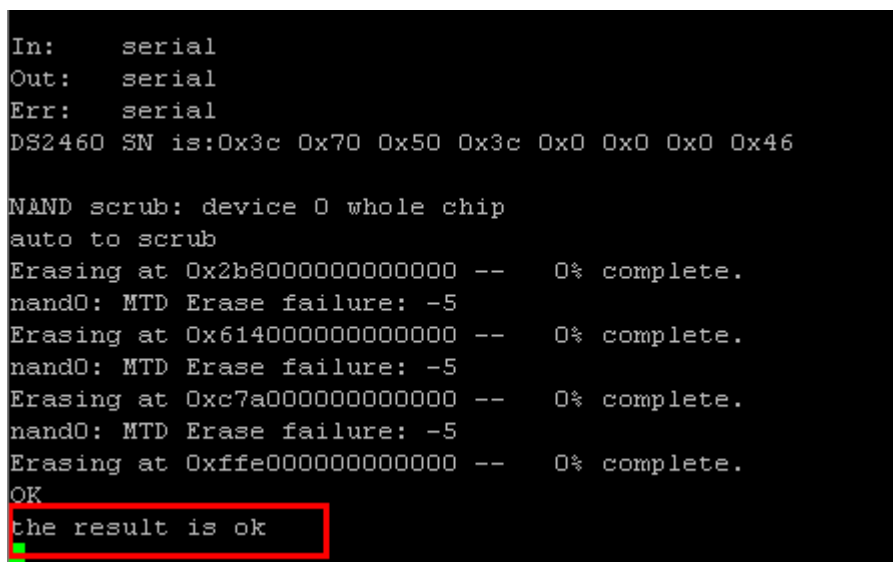


图 6.3 Nand Flash 格式化结果

#### 1.41.3 制作系统安装引导盘

跟步骤 1 的做法类似，把TF卡在PC机的读卡器上插接好，确定TF卡被格式化为FAT32，并记下其盘符，这一次运行产品光盘“images\TF卡烧写方案\2-TF卡烧写系统”目录下的sd\_os.bat，如图 6.4所示：

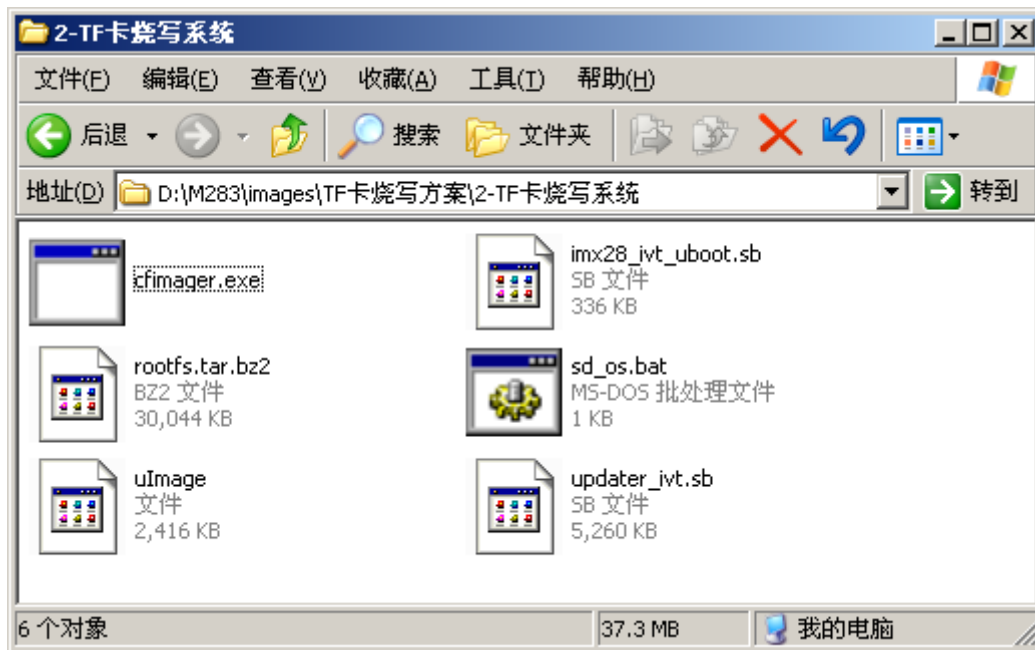


图 6.4 制作 TF 卡安装引导盘

在图 6.5所示窗口内，输入TF卡的盘符，比如在当前系统下TF卡显示为F盘，输入F并回车，然后稍等一会，数据写入到TF卡完毕，安装引导盘就制作好了。

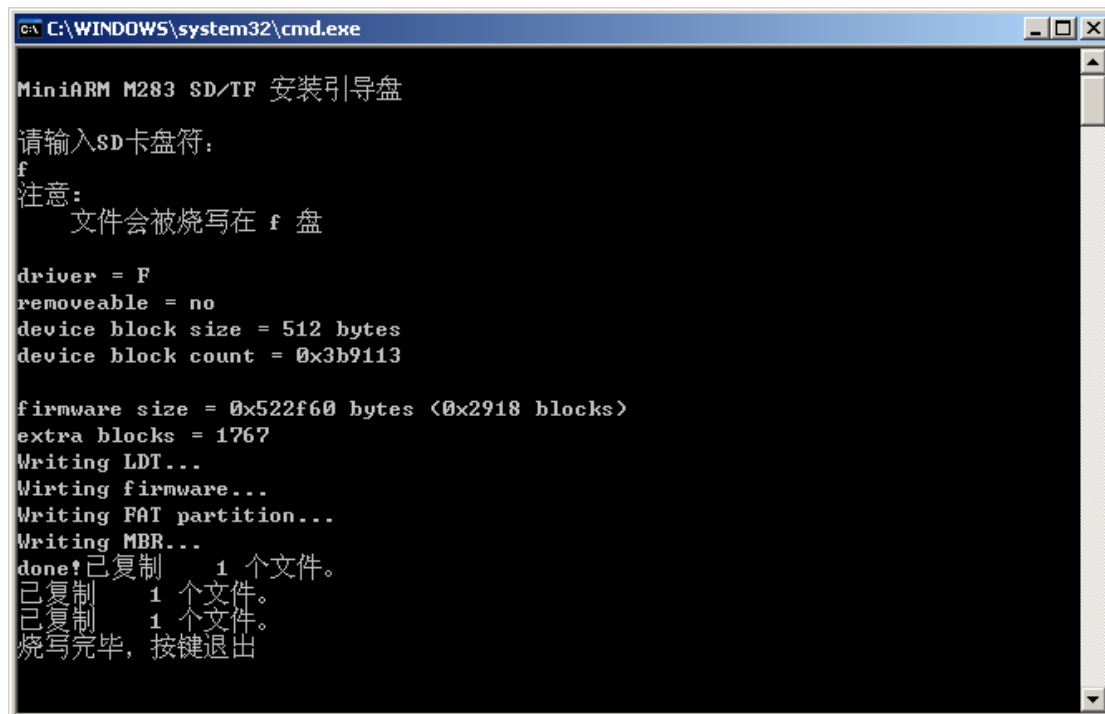


图 6.5 制作 TF 卡安装引导盘运行结果

#### 1.41.4 安装Linux系统

把安装引导盘插入底板上的卡座，确认启动模式是从 TF 卡启动，然后给板子上电或复位，板子启动后会自动安装系统，等待几分钟，成功后蜂鸣器会一直鸣叫。到此为止，整个本产品的 Linux 系统就全部安装完毕。

安装完毕后把板子启动模式设成从 Nand Flash 启动，把 TF 卡取下，重新启动板子就可以了。

## 1.42 USB恢复系统

通过USB烧写固件需要使用飞思卡尔提供的MfgTool软件。MfgTool软件在光盘的“4. 系统恢复\V1.02 烧写固件\USB烧写方案\MfgTool 1.6.2.055-ZLG140813”目录。该目录的内容如图 6.6所示，其中MfgTool.exe程序是USB固件烧写的程序。



图 6.6 MfgTool 软件

**说明：** MfgTool 软件不支持 Win8 系统。

### 1.42.1 执行USB烧写

#### 1. 硬件连接

硬件连接方法如下：

把 EPC-28x-L 设置为 USB 启动方式（使用短路器短接 JP4 和 JP6 跳线，保持 JP1、JP2、JP3 跳线的断开）；

使用 MicroUSB 线缆连接开发套件的 USB OTG 接口和主机。

#### 2. 设置MtgTool软件

双击运行MfgTool.exe软件，软件界面如图 6.7所示。



图 6.7 MtgTool 的主界面

点击主菜单中的Options→Configuration...菜单项，打开MfgTool的配置界面，如图 6.8所示。



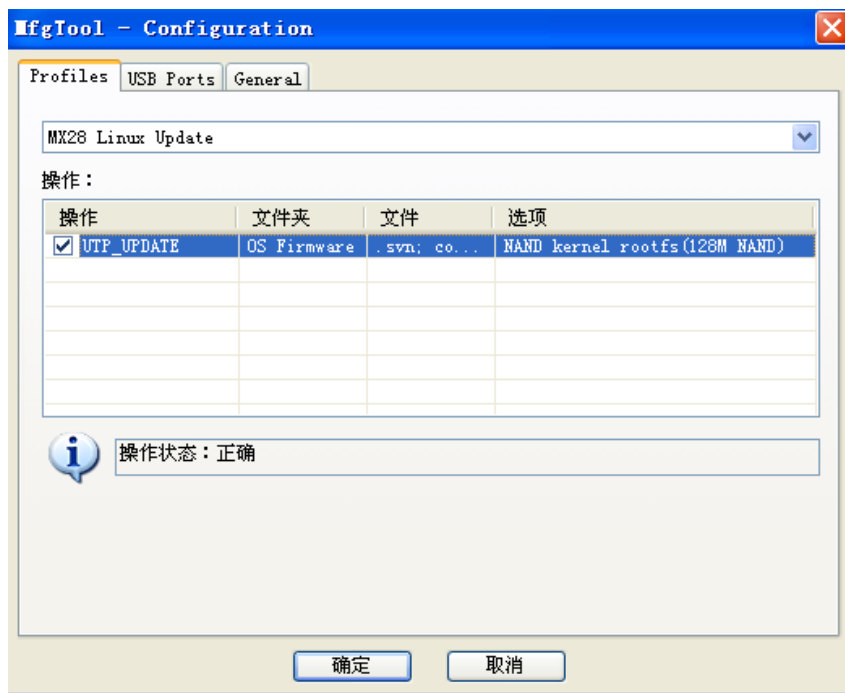


图 6.8 MfgTool 的配置界面

在MfgTool的配置界面，进入“Profiles”标签，在UTP\_UPDATE项的“选项”列中选择“NAND kernel-rootfs（128MB）”，如图 6.9所示，然后点击“确定”。

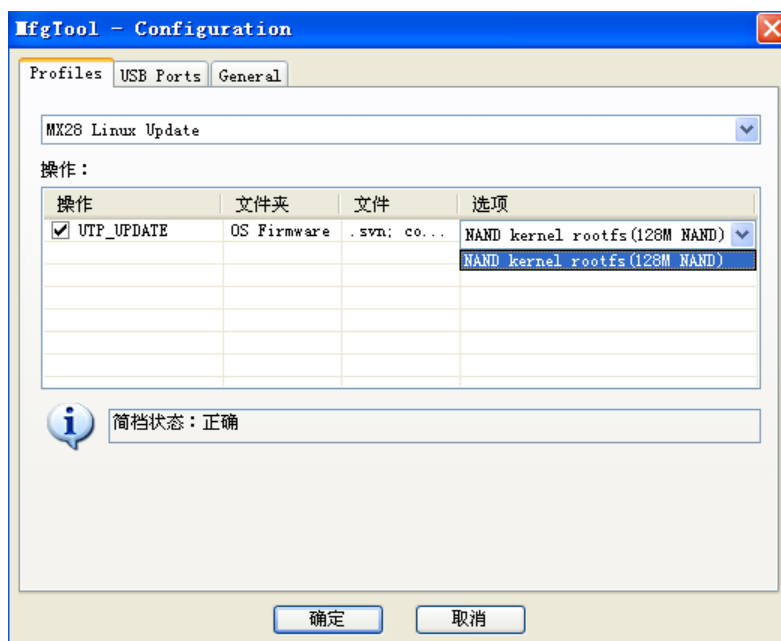


图 6.9 选择“NAND kernel-rootfs（128MB）”

切换到“USB Ports”标签，勾选已经连接上的“HID-compliantdevice”（即开发套件对应的设备）如图 6.10所示。然后点击“确定”。

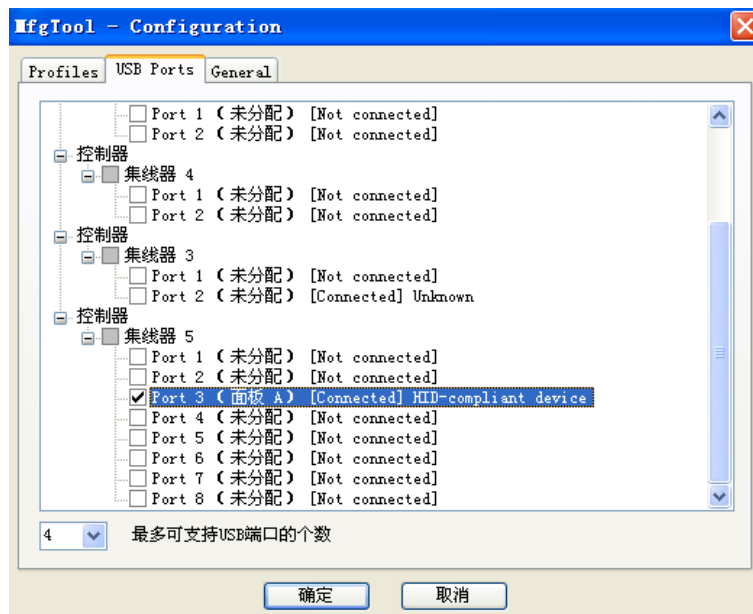


图 6.10 勾选连接的 HID-compliantdevice

### 3. 执行烧写

返回到MfgTool主界面后显示软件正在监视HID-compliantdevice，如图 6.11所示。



图 6.11 MfgTool 监视 HID-compliantdevice

此时点击“开始”执行烧写操作，如图 6.12所示。



图 6.12 执行烧写

直到烧写完成后点击“停止”，如图 6.13所示。

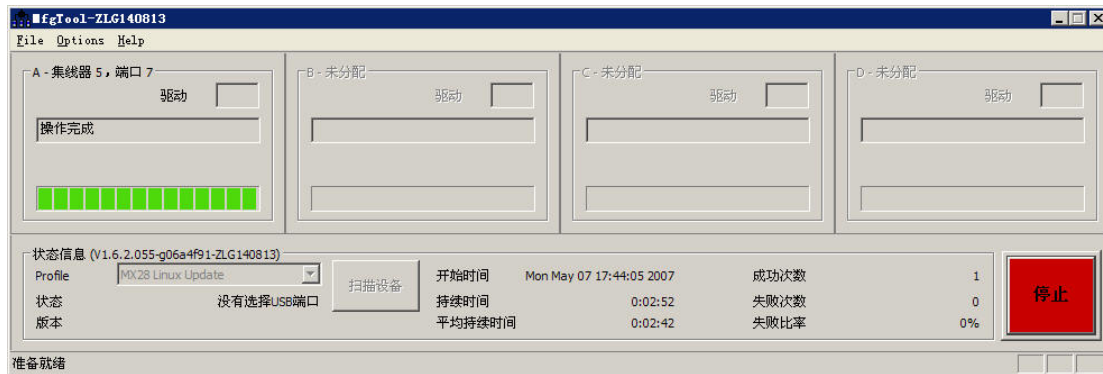


图 6.13 烧写完成

注意：在烧写过程中需要保持 MiniUSB 线缆的连接正常和开发套件的正常供电，否则在烧写过程容易出错。

烧写完成后，把 EPC-28x-L 设置为 NAND Flash 启动方式（取出 JP6 的短路器，仅短接 JP4，其他全部断开），按“RST”键复位系统。EPC-28x-L 将从 NAND Flash 启动 Linux 系统。

若使用 MfgTool 软件时出错，请检查是否有下列情形：

- MicroUSB 线缆是否连接正常；
- MfgTool 在点击“开始”烧写后，EPC-28x-L 必须再重新上电或按 RST 复位后，才能再次进行烧写；
- 设置为 USB 启动方式的 EPC-28x-L 在接入电脑后，在电脑的设备管理器会多一个 HID 设备出来，如**错误！未找到引用源。**所示，若电脑中未发现这个 HID 设备，请先检查启动模式配置及与电脑的连接是否正常，然后重新复位 EPC-28x-L 并插拔 USB 连接线；
- 在 Windowss 7 系统，建议以管理员身份运行 MfgTool 软件。

在使用 MfgTool 烧写固件的过程中，目标板将被虚拟成大容量存储设备（U 盘），如果用户的电脑系统受文件监控软件的保护，将可能无法正常进行烧写。所以，在使用 MfgTool 烧写固件前需要先关闭会对 U 盘读写进行监控的软件，如杀毒软件及防水墙等。

## 7 免责声明

本文档提供有关致远电子产品的信息。本文档并未授予任何知识产权的许可，并未以明示或暗示，或以禁止发言或其它方式授予任何知识产权许可。除致远电子在其产品的销售条款和条件中声明的责任之外，致远电子概不承担任何其它责任。并且，致远电子对致远电子产品的销售和 / 或使用不作任何明示或暗示的担保，包括对产品的特定用途适用性、适销性或对任何专利权、版权或其它知识产权的侵权责任等，均不作担保。致远电子产品并非设计用于医疗、救生或维生等用途。致远电子可能随时对产品规格及产品描述做出修改，恕不另行通知。

在订购产品之前，请您与当地的致远电子销售处或分销商联系，以获取最新的规格说明。

本文档中提及的含有订购号的文档以及其它致远电子文献可通过访问广州致远电子股份有限公司的万维网站点获得，网址是：

<http://www.zlg.cn>或致电+86-20-22644254查询。

Copyright © 2013, ZHIYUAN ElectronicsStock Co., Ltd. 保留所有权利。

## 销售与服务网络

### 广州致远电子股份有限公司

地址：广州市天河区车陂路黄洲工业区 7 栋 2 楼

邮编：510660

网址：[www.zlg.cn](http://www.zlg.cn)



全国销售与服务电话：400-888-4005

### 销售与服务网络：

#### 广州总公司

广州市天河区车陂路黄洲工业区 7 栋 2 楼

电话：(020)28267985 22644261

#### 上海分公司：上海

上海市北京东路 668 号科技京城东楼 12E 室

电话：(021)5386552153083451

#### 北京分公司

北京市海淀区知春路 108 号豪景大厦 A 座 19 层

电话：(010)62536178 62635573

#### 上海分公司：南京

南京市珠江路 280 号珠江大厦 1501 室

电话：(025)68123923 68123920

#### 深圳分公司

深圳市福田区深南中路 2072 号电子大厦 12 楼

电话：(0755)8364016983783155

#### 上海分公司：杭州

杭州市天目山路 217 号江南电子大厦 502 室

电话：(0571)89719491 89719493

#### 武汉分公司

武汉市洪山区广埠屯珞瑜路 158 号 12128 室（华中电脑数码市场）

电话：(027)87168497 87168397

#### 重庆分公司

重庆市九龙坡区石桥铺科园一路二号大西洋国际大厦（赛格电子市场）2705 室

电话：(023)68796438 68797619

#### 成都分公司

成都市一环路南二段 1 号数码科技大厦 403 室

电话：(028)85439836 85432683

#### 西安办事处

西安市长安北路 54 号太平洋大厦 1201 室

电话：(029)87881295 87881296

请您用以上方式联系我们，我们会为您安排样机现场演示，感谢您对我公司产品的关注！