

# Linux 开发指南

基于 EasyARM-iMX283/287 开发套件

UM14031301 V1.05 Date: 2014/09/11

产品用户手册

类别	内容
关键词	EasyARM-iMX283、EasyARM-iMX287、Linux 开发
摘要	介绍在 EasyARM-iMX283/287 学习套件下的 Linux 软件开发



## 修订历史

版本	日期	原因
V1.00	2013-11-26	创建文档
V1.01	2014-3-13	修改部分文字描述, NANDflash 的分区有改变
V1.02	2014-4-10	修改 tftp 服务器所在路径
V1.03	2014-5-7	1、添加 “EasyARM-iMX283” 资源简介章节; 2、在 “在 EasyARM-iMX283 上安装 Linux 系统” 章节添加了 TF 卡烧写方案, 修改了 USB 烧写方案, 修改了网络烧写方案; 3、增加了 “系统操作和基本设置” 章节 4、在 “嵌入式开发环境构建” 章节增加了 “Linux 主机操作系统安装” 及 “Linux 主机基础应用” 内容; 5、在 “Linux 内核编译” 章节添加了 imx_ivt_linux.sb 固件制作的内容; 6、在 “嵌入式 Linux 根文件系统” 章节添加了 rootfs.tar.bz 固件制作的内容; 7、在 “嵌入式 Linux Qt 编程指南” 章节修改了 Qt Creator 工具的使用。
V1.03.01	2014-5-22	1、更正指令中 linux 被排版为 Linux 的问题; 2、处理 PDF 文件不能复制文本问题。
V1.03.02	2014-5-27	更正 ADC 操作接口指令
V1.04	2014-06-17	1、完善 3.2 章节串口登录及 SSH 登录的硬件连接描述 2、更正 4.6 章节测试工具链中引用编译器错误的问题 3、更正 4.6 章节 Makefile 示例因排版而出现的错误 4、更正 5.3 章节, 使表述更清晰易懂 5、更正 7.1 章节的语言表述及 7.4 章节的错误 6、更正 7.5 章节 Makefile 示例因排版而出现的错误 7、更正 8.4 节 setenv 的命令中的错误 8、更新第 9 章, 使表述更易懂 9、更正命令中 nand 被排版为 NAND 的错误 10、更正编译器引用, 统一使用 “arm-fsl-linux-gnueabi” 前缀 11、更正排版错误, 更新产品介绍 12、增加第 10 章 “关于 EasyARM-iMX287 开发套件” 内容
V1.05	2014-09-11	1、更新第 2 章、第 6 章及第 7 章, 取消 64MB 与 128MB 内存对应软件分开存放的描述; 2、更正程序清单 9.1 因排版引起的错误 3、修正 8.5 章节的错误 4、增加 IP、子网掩码及 DNS 设置介绍



## 目 录

1. EasyARM-iMX283 产品介绍	5
1.1    EasyARM-iMX283 简介	5
1.2    核心板资源	6
1.3    底板资源	7
1.4    Linux 平台软件开发资源	7
2. 在 EasyARM-iMX283 上安装 Linux 系统	9
2.1    NAND Flash 存储器分区	9
2.2    固件烧写前的准备	9
2.2.1    使能未签名固件启动	9
2.2.2    格式化 NAND Flash	11
2.3    TF 卡烧写方案	18
2.3.1    制作系统恢复卡	18
2.3.2    系统恢复步骤	20
2.4    USB 烧写方案	20
2.4.1    烧写 U-Boot	21
2.4.2    烧写“内核+文件系统”	23
2.5    网络烧写方案	24
2.5.1    所需条件	24
2.5.2    系统恢复步骤	24
3. 启动选择和系统基本设置操作	30
3.1    系统启动跳线设置	30
3.2    系统登录	30
3.3    网络设置	31
3.3.1    设置 IP 和子网掩码	31
3.3.2    设置默认网关	31
3.3.3    设置 DNS	31
3.3.4    注意事项	32
3.4    TF 卡使用	32
3.5    U 盘使用	33
3.6    USB Device 使用	33
3.7    LED 使用	34
3.8    蜂鸣器使用	34
3.9    LCD 背光控制	35
3.10    系统时间设置	35
4. 嵌入式开发环境构建	36
4.1    嵌入式 Linux 开发简介	36
4.2    安装 Linux 主机操作系统	37
4.3    Linux 主机基础应用介绍	52
4.4    ssh 服务器配置	54
4.5    NFS 服务器配置	55
4.6    TFTP 服务器	61



4.7	构建交叉开发环境	65
5.	功能部件编程	71
5.1	GPIO 应用编程	71
5.1.1	导出 GPIO	72
5.1.2	方向设置	72
5.1.3	输入读取	73
5.1.4	输出控制	73
5.2	ADC 接口	74
5.2.1	ADC 驱动模块的加载	74
5.2.2	操作接口	74
5.2.3	计算公式	75
5.2.4	操作示例	75
5.3	串口编程	76
5.3.1	访问串口设备	77
5.3.2	配置串口属性	78
5.3.3	操作示例	83
5.4	I <sup>2</sup> C 接口	85
5.4.1	open 调用	85
5.4.2	ioctl 调用	85
5.4.3	write 调用	86
5.4.4	read 调用	86
5.4.5	close 调用	87
5.4.6	应用程序读写 DS2460 例程	87
5.5	PWM 接口	89
5.5.1	PWM 占空比设置与输出	89
5.5.2	系统命令操作 PWM 示例	89
5.6	SPI 接口	90
5.6.1	open 调用	90
5.6.2	ioctl 调用	91
5.6.3	示例代码	93
6.	EasyARM-iMX283 的 Boot Loader	99
6.1	U-Boot 简介	99
6.2	U-Boot 源代码目录结构	99
6.3	编译 U-Boot	99
6.4	U-Boot 基本命令	100
6.4.1	预设的组合命令	102
6.4.2	通过网络启动内核	103
6.5	U-Boot Tools	103
7.	Linux 内核编译和驱动要点	104
7.1	编译内核	104
7.1.1	解压内核文件	104
7.1.2	运行 SPI 补丁	104
7.1.3	备份内核配置文件	104
7.1.4	编译内核	104



7.2	生成 imx28_ivt_linux.sb 内核固件.....	104
7.3	配置内核.....	105
7.4	内核 GPIO 使用方法 .....	110
7.5	蜂鸣器驱动.....	112
7.5.1	驱动加载.....	112
7.5.2	卸载驱动.....	113
7.5.3	Open 调用的实现.....	114
7.5.4	write 调用的实现.....	114
7.5.5	ioctl 函数的实现.....	115
7.5.6	close 调用的实现.....	115
7.5.7	编译驱动代码.....	115
7.5.8	测试程序.....	116
7.6	设置 LCD 的时序.....	118
8.	嵌入式 Linux 根文件系统 .....	120
8.1	Linux 根文件系统 .....	120
8.2	FHS 标准 .....	120
8.2.1	顶层目录.....	120
8.2.2	“/usr”目录 .....	121
8.3	BusyBox.....	121
8.4	NFS 根文件系统 .....	121
8.5	生成文件系统映像.....	122
8.5.1	生成 rootfs.ubifs 固件.....	122
8.5.2	生成 rootfs.tar.bz2 固件 .....	123
8.6	开机启动设置.....	123
9.	嵌入式 Linux Qt 编程指南 .....	125
9.1	背景知识.....	125
9.2	Qt 介绍 .....	125
9.2.1	Qt 简介 .....	125
9.2.2	Qt/E 简介 .....	125
9.3	编译环境的搭建.....	126
9.4	hellow 程序开发 .....	126
9.4.1	编译 hellow 程序 .....	126
9.4.2	在目标板上运行 hellow 程序 .....	128
9.5	qmake 与 pro 文件 .....	131
9.5.1	pro 文件例程 .....	131
9.5.2	pro 文件常见配置 .....	132
9.6	Qt 编程简单入门 .....	133
9.6.1	例程讲解 .....	133
9.6.2	信号和槽机制 .....	134
9.7	Qt SDK 的使用 .....	135
9.7.1	Qt SDK 简介 .....	135
9.7.2	Qt SDK 安装 .....	136
9.7.3	Qt Creator 配置 .....	138
9.7.4	Qt Creator 使用例程 .....	141



9.8	zylauncher 图形框架 .....	148
10.	关于 EasyARM-iMX287 开发套件 .....	153
10.1	EasyARM-iMX287 简介 .....	153
10.1.1	核心板资源 .....	153
10.1.2	底板资源 .....	154
10.2	CAN 接口的使用 .....	154
10.2.1	内核开启 CAN 驱动支持 .....	154
10.2.2	使用 CAN 设备 .....	156
10.2.3	socket CAN 编程指南 .....	161
10.3	SPI3 的使用 .....	164
10.4	双网口的使用 .....	165
	图索引 .....	168
	表格索引 .....	173
	程序清单索引 .....	174
	参考文献 .....	176
	免责声明 .....	177

## 1. EasyARM-iMX283 产品介绍

本章主要介绍用于 Linux 学习的 EasyARM-iMX283 学习套件的平台资源，包含核心板资源、底板资源及软件资源。

相关信息也可以通过<http://www.zlgmcu.com/Freescale/EasyARM-iMX283.asp>了解。

### 1.1 EasyARM-iMX283 简介

EasyARM-iMX283 是广州周立功单片机科技有限公司精心设计的一款集教学、竞赛、实验、产品设计以及功能评估于一身的入门级开发套件，其产品外观参考如图1.1所示。



图1.1 EasyARM-iMX283 产品正面

套件采用M283 核心板，标配 4.3 寸TFT液晶屏，具有丰富的硬件资源，并提供实用的 Linux 的软件支持包和完善的开发工具，大大降低了Linux学习及开发门槛，可帮助用户在短期内实现产品功能验证和开发。EasyARM-iMX283 的基本接口分布及核心板位置如图1.2所示。

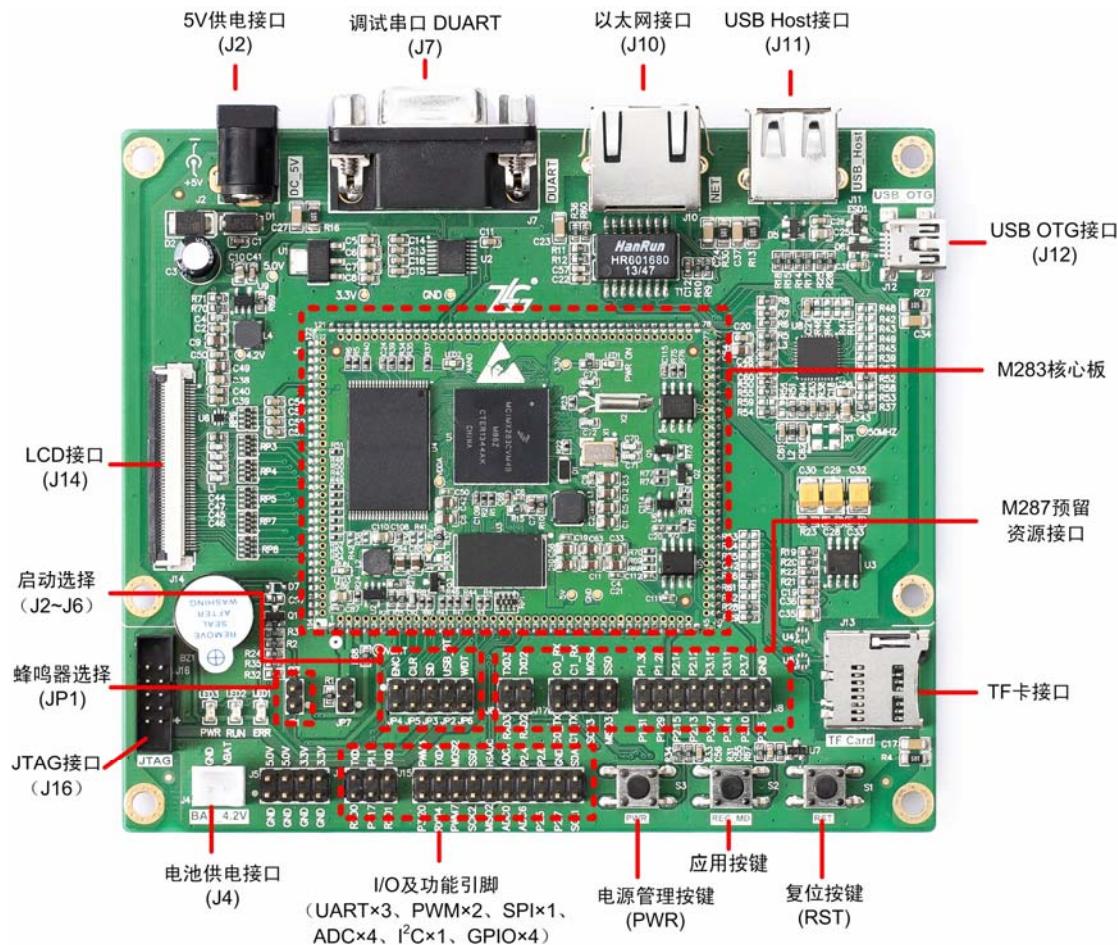


图1.2 EasyARM-iMX283 的基本接口分布及核心板位置

## 1.2 核心板资源

EasyARM-iMX283 开发套件采用“M283 核心板+底板”的组合方式，M283 核心板资源如表1.1所示。

表1.1 M283 核心板资源

项目	参数	项目	参数
CPU	MCIMX283CVM4B	PWM	3 路
主频	454MHz	USB	1 路 HOST、1 路 OTG
			(USB2.0 高速, 480Mbps)
内存 (DDR2)	64MB <sup>[1]</sup>	I <sup>2</sup> S	—
FLASH	128MB <sup>[1]</sup>	ADC	4 路
TFT 支持	最高支持 800*480 分辨率	TF 卡接口	1 路
触摸屏	四线电阻式	I <sup>2</sup> C	1 路
以太网	1 路 10/100M	看门狗	外置独立硬件看门狗
串口	6 路	硬件加密	支持
SPI	1 路	RTC	支持内部 RTC 实时时钟模块

[1]: EasyARM-iMX283 V1.00 版本的 NAND Flash 容量为 256MB, DDR2 内存为 128MB, V1.01 及之后版本 NAND Flash 容量为 128MB, DDR2 内存容量为 64MB。



### 1.3 底板资源

为方便用户灵活配置处理器相关功能复用 I/O，评估处理器相关外设的项目应用，除开发或学习所必需的外设外，核心板其他资源均通过排针方式引出。EasyARM-iMX283 开发套件硬件底板资源如下：

- 4 路串口
  - ◆ 1 路调试用的 UART（DB9 座引出）
  - ◆ 3 路应用 UART（以排针方式引出，分别是 UART0/1/4）
- 2 路 USB 2.0 接口
  - ◆ 1 路 Host 接口，支持 U 盘、USB 鼠标和键盘
  - ◆ 1 路 OTG 接口
- 1 路 TF 卡接口
- 1 路以太网接口
- 1 个蜂鸣器
- 3 个按键：1 个复位键、1 个电源管理按键及 1 个应用按键
- 1 路 16 位液晶屏接口（含触摸屏接口，支持 4 线电阻式触摸屏），默认支持 480×272 TFT 液晶屏套件
- 以排针方式引出的其他接口
  - ◆ 1 路 I<sup>2</sup>C
  - ◆ 1 路 SPI（可复用为 UART2/3）
  - ◆ 4 个 ADC 通道
  - ◆ 4 个 GPIO
  - ◆ 2 路 PWM

### 1.4 Linux 平台软件开发资源

EasyARM-iMX283 开发套件对应 Linux 平台提供的软件资源如下：

- Bootloader: u-boot-2009.08
- Linux 内核: linux-2.6.35.3
- 根目录文件系统支持: sysfs、rootfs、bdev、ext3、ext2、ramfs、nfs、jffs2、ubifs、tmpfs 等
- 交叉编译工具链: gcc-4.4.4-glibc-2.11.1-multilib-1.0
- 图形界面: Qt-4.8.0
- 提供的外设驱动
  - ◆ NAND Flash 驱动
  - ◆ SD/MMC 驱动
  - ◆ TFT LCD 驱动（默认支持 480x272 的 4.3 寸液晶）
  - ◆ 触摸屏驱动
  - ◆ SPI 驱动
  - ◆ I<sup>2</sup>C 驱动
  - ◆ 应用串口 AUART 驱动
  - ◆ ADC 驱动
  - ◆ PWM 驱动
  - ◆ LED 驱动
  - ◆ 蜂鸣器驱动
  - ◆ GPIO 驱动



- ◆ RTC 驱动
- 提供基本外设范例程序
- 提供系统开发所需的基本工具

广州周立功



## 2. 在EasyARM-iMX283 上安装Linux系统

本章主要讲述如何在 EasyARM-iMX283 上进行 Linux 系统恢复（烧写 Linux 固件）。用户可以通过 TF 卡、USB 或网络等 3 种方式进行 Linux 系统恢复，若用户不需要更新或恢复 Linux 系统，则可以跳过这一章。

如果出厂预装的就是 Linux 操作系统，在系统能正常运行的情况下，可以忽略这一章内容。如果出厂预装 WinCE 系统，或者系统在使用中被损坏，需要恢复或者更新，则务必仔细阅读本章内容。

本章所引用的固件及部分工具位于 V1.05 版本光盘目录“3.Linux\5.Linux 系统恢复\”下，本文描述的方法基于 V1.05 版本光盘提供的软件工具及固件。

V1.05 版本光盘提供的软件工具及固件兼容广州周立功发售的 64MB 及 128MB DDR2 内存的开发套件及工控核心板。

### 2.1 NAND Flash存储器分区

V1.01 及之后版本的EasyARM-iMX283 板载 128MB 的NAND Flash，其扇区大小为 128KB，Linux内核以及文件系统都安装在其中，NAND Flash的分区情况如表2.1所列。

表2.1 NAND Flash 分区信息

分区	地址范围	大小	用途
Bootloader、kernel	0x00000000-0x01400000	20M	U-Boot 及其环境变量参数、内核
rootfs	0x01400000-0x08000000 <sup>[1]</sup>	108MB	根文件系统

[1]: 对于 EasyARM-iMX283 V1.00，其板载 NAND Flash 容量为 256MB，其根文件系统地址范围为“0x01400000-0x10000000”，大小为“236MB”。

EasyARM-iMX283 在 NAND Flash 启动时，有两种方式：

- 通过 U-Boot 引导进入系统：需要在 NAND 中烧写 U-Boot、内核以及文件系统；
- 通过内核直接进入系统：只需在 NAND 中烧写内核和文件系统。直接从内核启动，可以加快系统的启动时间，但此模式下不能通过网络方式更新或烧写内核和文件系统。

### 2.2 固件烧写前的准备

V1.04 版本其之前的光盘提供的固件均使用全“0”密钥（芯片出厂初始密钥）对固件进行了数字签名，为了更好的兼容性，V1.05 版本及之后版本的光盘提供的固件均改用不经过数字签名的固件。由于 i.MX28x 系列芯片出厂默认不支持未签名固件启动，所以需要通过修改 OTP 相关的熔丝位来使能未签名的固件启动。若用户手上的开发套件或工控核心板未使能未签名固件启动，则需要先“使能未签名固件启动”。

由于 Linux 系统与 WinCE 系统对 NAND Flash 坏块的管理机制不通，若在安装 Linux 系统之前，已经在 NAND Flash 中安装过 WinCE 系统，则需要先格式化 NAND Flash，否则将导致 Linux 系统安装失败。

#### 2.2.1 使能未签名固件启动

使用 V1.05 版本及之后的光盘提供的 Mfgtool 软件下载固件时，默认的操作列表已增加了自动使能未签名固件启动的操作（注意：自动使能未签名固件启动的操作适用于签名密钥为全零的芯片，密钥已被修改且未使能未签名固件启动的芯片无法使用光盘提供的工具及固



件), 用户无须额外操作, 这里主要介绍如何通过 SD 卡引导方式来使能未签名固件的启动。

通过 SD 卡引导方式使能未签名固件启动, 需要先制作专用的启动卡, 其制作步骤如下:

- 将一张空白的 TF 通过读卡器接入电脑 (操作系统必须为 Windows XP 专业版或 Win7 旗舰版), 并记下电脑分配给其的盘符 (推荐使用 Class 4 的 TF 卡);
- 双击运行光盘资料中 “**TF 卡烧写方案\使能未签名固件启动**” 目录下的 “imx28\_BootCfg.bat” 批处理文件 (Win7 系统建议以管理员身份运行该脚本), 然后输入系统分配给 TF 卡的盘符并按回车键;
- 启动卡制作完后如图2.1所示, 此时按照移除U盘的方式移除该TF卡即可。

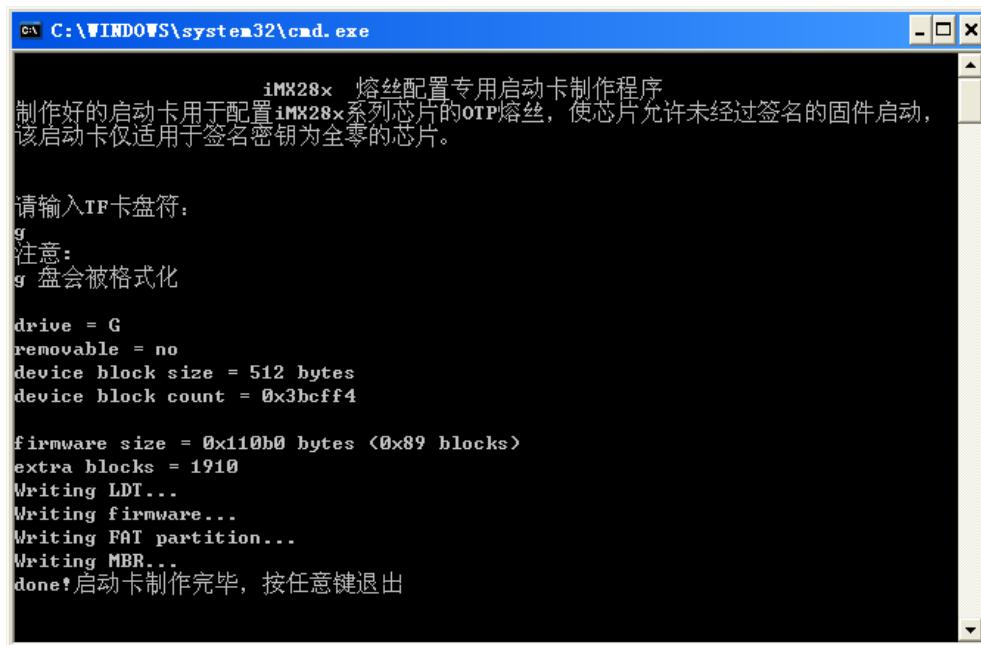


图2.1 熔丝配置专用启动卡制作完成

熔丝配置专用启动卡制作好了之后, 按如下步骤操作使能未签名固件的启动:

- 使用短路器短接开发套件上的JP1 (BZ, 使能蜂鸣器); 短接JP3 (SD, 设置为从SD卡启动); 短接JP6 (WDT, 禁用看门狗), 如图2.2所示。
- 使用串口延长线连接PC机和开发套件的DUART (J7), 如图2.2所示;
- 在 PC 机打开串口终端监听串口数据 (串口终端参数设置为 “115200,8,1,N,无<sup>[1]</sup>”)。
- 将制作好的熔丝配置专用启动卡接入开发套件的TF卡卡座, 如图2.2所示;
- 给开发套件接通电源, 熔丝配置程序运行完后, 蜂鸣器将会长鸣一声, 配置成功后串口终端是输出信息如图2.3所示。

[1]: “115200,8,1,N,无”表示“波特率为 115200, 8 位数据位, 1 位停止位, 无奇偶校验, 无流控信号”, 本文中所用的串口终端均采用此配置, 后面不再作详细说明。

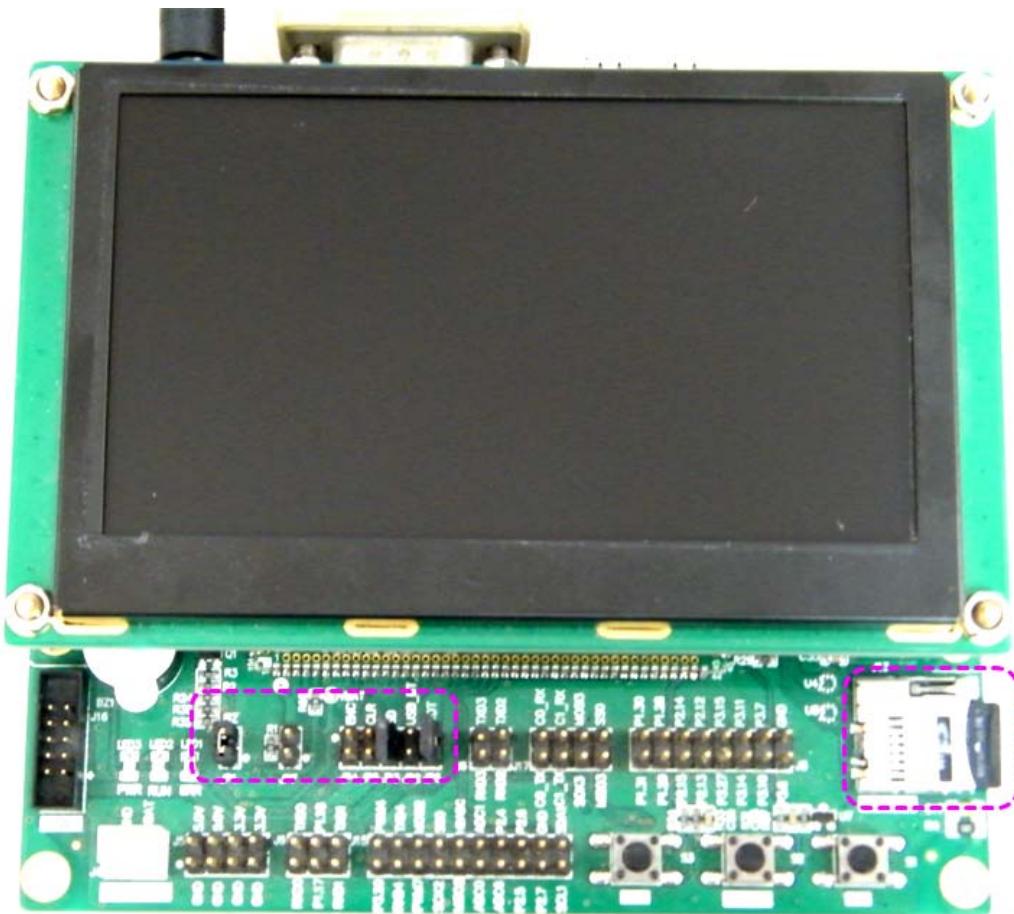


图2.2 配置为从 TF 卡启动

```
File Edit Setup Control Window Help
Aug 15 2014 17:08:24
FRAC 0x92925552
Wait for ddr ready 1bank count is 8
power 0x00820816
Frac 0x92925552
start change cpu freq
ibus 0x00000003
cpu 0x00010001
start test memory access
ddr2 0x40FFFFF00
finish simple test
*****ddr2 read write success!
finish simple test
finish boot prep,start to run ...
Application start up!
Enable unencrypted boot modes ...
Enable unencrypted boot modes.Done!
```

图2.3 使能未签名固件启动

注意：制作好的启动卡用于配置 iMX28x 系列芯片的 OTP 熔丝，使芯片允许未签名固件启动，且该启动卡仅适用于签名密钥为全零的芯片（出厂默认芯片）。“允许未签名固件启动”的熔丝位只需要且只能配置一次，被配置过的芯片无须再配置。

## 2.2.2 格式化NAND Flash

## 1. 通过USB Boot引导格式化NAND Flash

使用 USB Boot 方式格式化 NAND Flash 方法如下：

- 使用短路器短接EasyARM-iMX283 上的JP1(BZ, 使能蜂鸣器); 短接JP2(USB\_BT, 设置为USB方式启动); 短接JP6 (WDT, 禁用看门狗), 如图2.4所示。
- 使用MiniUSB通信电缆连接PC机和EasyARM-iMX283 的USB OTG接口 (J12) ; 使用串口延长线连接PC机和EasyARM-iMX283 的DUART (J7), 如图2.4所示。
- 在 PC 机打开串口终端监听串口数据 (串口终端参数设置为“115200,8,1,N,无” )。
- 给 EasyARM-iMX283 接通电源。

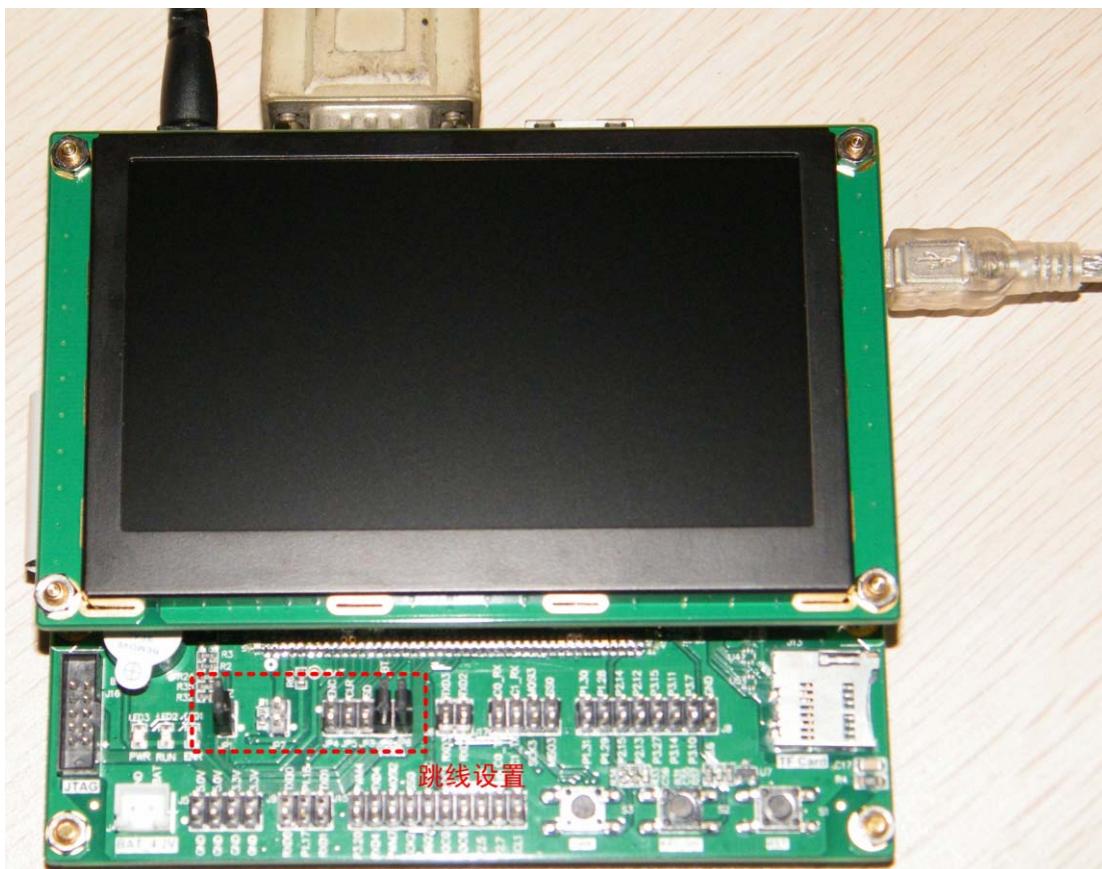


图2.4 跳线设置及接线示意图

- 请进入光盘文件中的“**USB烧写方案\NAND Flash格式化**”目录，双击“**NAND Flash格式化.bat**”脚本程序，将弹出如图2.5所示的界面，但很快将自动关闭。

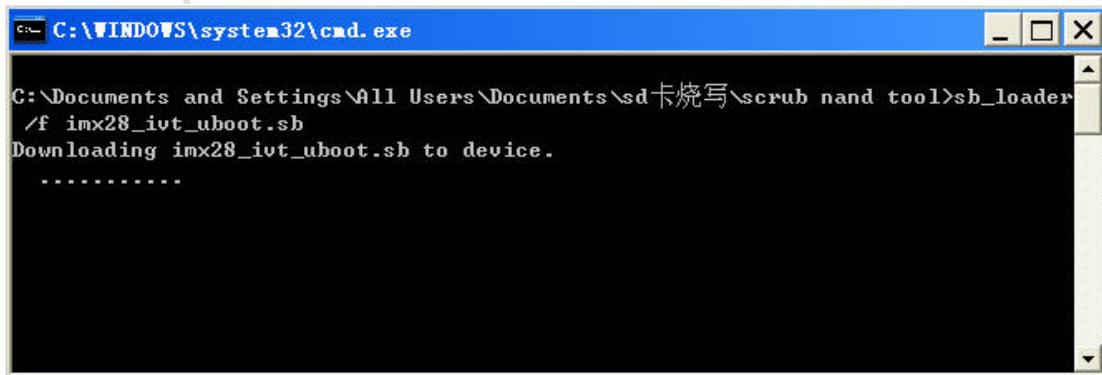




图2.5 脚本界面

这时串口终端将打印格式化输出信息，具体打印的信息可能会因具体的硬件不同而有所不同，但当看到提示“100% complete”、“OK”及“nand scrub done”等提示信息时，表示格式化成功，如图2.6所示。

如果看到串口终端的输出信息在“nand scrub done”上一行打印了“ERROR!”字样，如所示，则表示格式化失败。

串口终端输出“MTD Erase failure: -%d at:0xXXXXXXXXXXXXXXX”的提示信息，是因为在格式化过程中遇到NAND Flash的坏块(程序会自动记录和规避坏块)，NAND Flash有一定坏块是比较正常的，所以用户不用担心这个问题。

```
File Edit Setup Control Window Help
got MAC address from IIM: 00:04:00:00:00:00
FEC0
Warning: FEC0 MAC addresses don't match:
Address in SROM is      00:04:00:00:00:00
Address in environment is 02:00:92:b3:c4:a8

Hit any key to stop autoboot: 0

NAND scrub: device 0 whole chip
Warning: scrub option will erase all factory set bad blocks!
        There is no reliable way to recover them.
        Use this command only for testing purposes.
NAND Erasing at 0x0000000001700000 --  9% complete.
nand0: MTD Erase failure: -5 at:0x00000000018c0000
NAND Erasing at 0x00000000047a0000 -- 28% complete.
nand0: MTD Erase failure: -5 at:0x0000000004900000
NAND Erasing at 0x00000000070a0000 -- 44% complete.
nand0: MTD Erase failure: -5 at:0x00000000070c0000
NAND Erasing at 0x0000000008cc0000 -- 55% complete.
nand0: MTD Erase failure: -5 at:0x0000000008d40000
NAND Erasing at 0x000000000ffe0000 -- 100% complete.
OK!
nand scrub done.
MX28 U-Boot >
```

图2.6 完成 NAND Flash 格式化

```
File Edit Setup Control Window Help
Address in environment is 02:00:92:b3:c4:a8

Hit any key to stop autoboot: 0

NAND scrub: device 0 whole chip
Warning: scrub option will erase all factory set bad blocks!
        There is no reliable way to recover them.
        Use this command only for testing purposes.
NAND Erasing at 0x0000000000770000 -- 93% complete.
nand0: MTD Erase failure: -5 at:0x00000000007820000

nand0: MTD Erase failure: -5 at:0x00000000007840000
NAND Erasing at 0x00000000007fe0000 -- 100% complete.
nand_write_oob: to = 0x07820040, len = 2
Chip: 0 DMA Buf: 0x40c08048 Length: 1
status:0
nand_write_oob: to = 0x07840040, len = 2
Chip: 0 DMA Buf: 0x40c08048 Length: 1
status:0
ERROR!
nand scrub done.
MX28 U-Boot >
```

图2.7 NAND 格式化失败

若启动 ubootloader.bat 脚本时，串口终端没有反应，请检查有无下列情形：

- 串口终端通信参数是否设置好；
- MiniUSB 通信电缆是否连接正常；
- ubootloader.bat 在启动一次后，EasyARM-iMX283 必须再重新上电或按 RST 复位后，才能再一次按所选的启动模式启动；
- 设置为从USB启动的EasyARM-iMX283 在接入电脑后，在电脑的设备管理器会多一个HID设备出来，如图2.8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EasyARM-iMX283 并插拔USB连接线；
- ubootloader.bat 脚本是调用了 imx28\_ivt\_uboot\_erase.sb 文件及飞思卡尔原厂提供的 sb\_loader.exe 程序，运行 ubootloader.bat 前需要保证其所在目录下的 imx28\_ivt\_uboot\_erase.sb 文件及 sb\_loader.exe 文件正常且未被占用；
- Win7 系统建议以管理员身份运行 ubootloader.bat 脚本。



图2.8 正常的连接情况

## 2. 通过SD Boot方式格式化NAND Flash

通过 SD Boot 方式格式化 NAND Flash 需要先制作一张格式化 NAND Flash 专用的 TF 启动卡，其制作步骤如下：

- 将一张空白的 TF 通过读卡器接入电脑（操作系统必须为 Windows XP 专业版或 Win7 旗舰版），并记下电脑分配给其的盘符（推荐使用 Class 4 的 TF 卡）；



- 双击运行光盘资料中“**TF 卡烧写方案\NAND Flash 格式化**”目录下的“制作 NAND Flash 格式化启动卡.bat”批处理文件（Win7 系统建议以管理员身份运行该脚本），然后输入系统分配给 TF 卡的盘符并按回车键；
- 启动卡制作完后如图2.9所示，此时按照移除U盘的方式移除该TF卡即可。

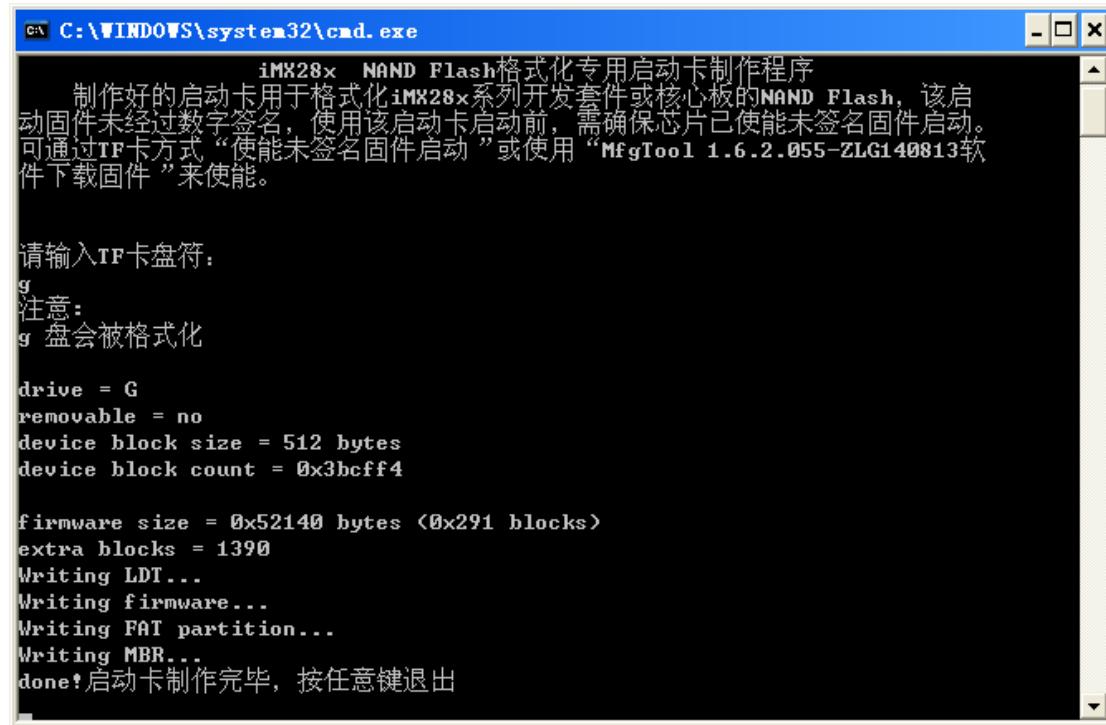


图2.9 NAND 格式化专用启动卡制作完成

格式化专用启动卡制作好了之后按如下步骤进行 NAND Flash 的格式化：

- 使用短路器短接EasyARM-iMX283 上的JP1 (BZ, 使能蜂鸣器); 短接JP3 (SD, 设置为从SD卡启动); 短接JP6 (WDT, 禁用看门狗), 如图2.2所示。
- 使用串口延长线连接PC机和EasyARM-iMX283 的DUART (J7), 如图2.2所示;
- 在 PC 机打开串口终端监听串口数据 (串口终端参数设置为 “115200,8,1,N,无”)。
- 将格式化专用启动卡接入开发套件的TF卡卡座, 如图2.2所示;

### 2.2.3 给EasyARM-iMX283 接通电源, 等待格式化程序运行完毕, 格式过程中串口终端输出信息与“2.2.1使能未签名固件启动”

使用V1.05 版本及之后的光盘提供的Mfgtool软件下载固件时, 默认的操作列表已增加了自动使能未签名固件启动的操作(注意: 自动使能未签名固件启动的操作适用于签名密钥为全零的芯片, 密钥已被修改且未使能未签名固件启动的芯片无法使用光盘提供的工具及固件), 用户无须额外操作, 这里主要介绍如何通过SD卡引导方式来使能未签名固件的启动。

通过SD卡引导方式使能未签名固件启动, 需要先制作专用的启动卡, 其制作步骤如下:

- 将一张空白的TF通过读卡器接入电脑 (操作系统必须为Windows XP专业版或Win7 旗舰版), 并记下电脑分配给其的盘符 (推荐使用Class 4 的TF卡);
- 双击运行光盘资料中“**TF卡烧写方案\使能未签名固件启动**”目录下的“imx28\_BootCfg.bat”批处理文件 (Win7 系统建议以管理员身份运行该脚本), 然后输入系统分配给TF卡的盘符并按回车键;
- 启动卡制作完后如图2.1 所示, 此时按照移除U盘的方式移除该TF卡即可。



```
iMX28x 熔丝配置专用启动卡制作程序  
制作好的启动卡用于配置iMX28x系列芯片的OTP熔丝，使芯片允许未经过签名的固件启动，  
该启动卡仅适用于签名密钥为全零的芯片。  
请输入TF卡盘符：  
注意：  
g 盘会被格式化  
drive = G  
removable = no  
device block size = 512 bytes  
device block count = 0x3bcff4  
firmware size = 0x110b0 bytes <0x89 blocks>  
extra blocks = 1910  
Writing LDT...  
Writing firmware...  
Writing FAT partition...  
Writing MBR...  
done! 启动卡制作完毕, 按任意键退出
```

图2.1 熔丝配置专用启动卡制作完成

熔丝配置专用启动卡制作好了之后，按如下步骤操作使能未签名固件的启动：

- 使用短路器短接开发套件上的JP1（BZ，使能蜂鸣器）；短接JP3（SD，设置为从SD卡启动）；短接JP6（WDT，禁用看门狗），如图2.2 所示。
- 使用串口延长线连接PC机和开发套件的DUART（J7），如图2.2 所示；
- 在PC机打开串口终端监听串口数据（串口终端参数设置为“115200,8,1,N,无<sup>[1]</sup>”）。
- 将制作好的熔丝配置专用启动卡接入开发套件的TF卡卡座，如图2.2 所示；
- 给开发套件接通电源，熔丝配置程序运行完后，蜂鸣器将会长鸣一声，配置成功后串口终端是输出信息如图2.3 所示。

[1]：“115200,8,1,N,无”表示“波特率为 115200，8 位数据位，1 位停止位，无奇偶校验，无流控信号”，本文中所用的串口终端均采用此配置，后面不再作详细说明。

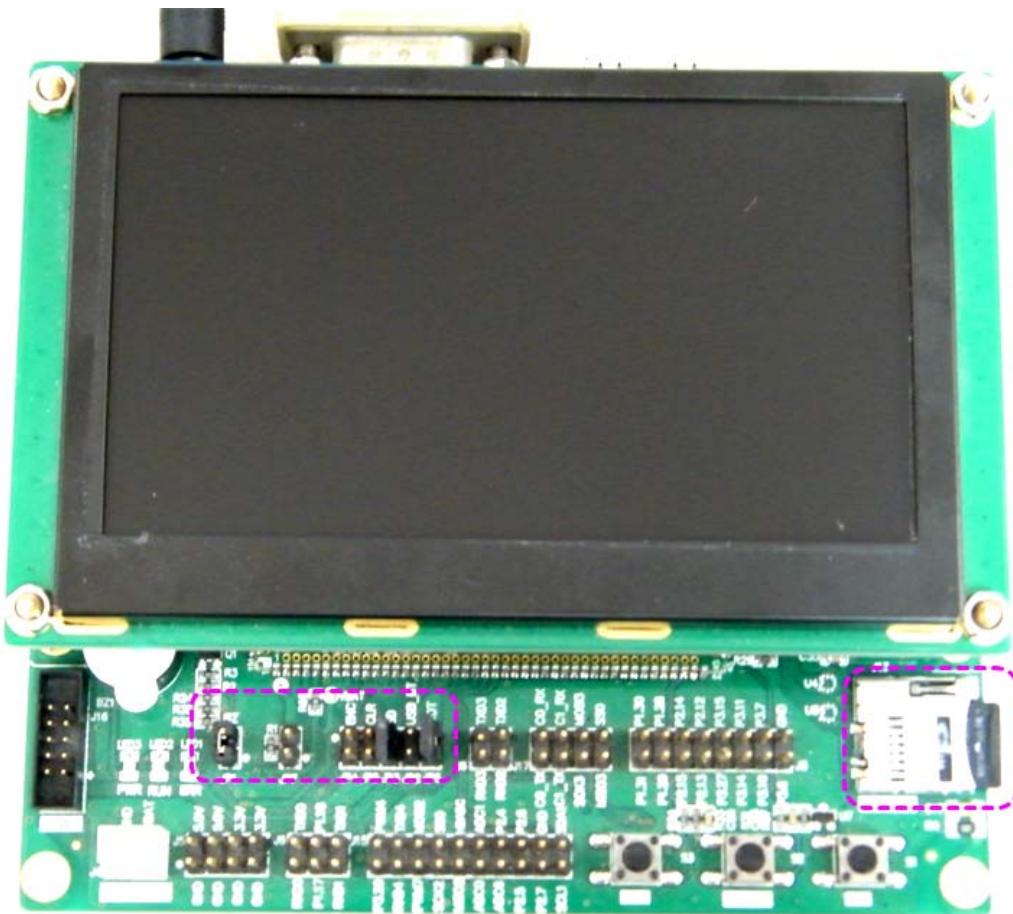


图2.2 配置为从TF卡启动

```
File Edit Setup Control Window Help
Aug 15 2014 17:08:24
FRAC 0x92925552
Wait for ddr ready 1bank count is 8
power 0x00820816
Frac 0x92925552
start change cpu freq
ibus 0x00000003
cpu 0x00010001
start test memory access
ddr2 0x40FFFFF00
finish simple test
*****ddr2 read write success!
finish simple test
finish boot prep,start to run ...
Application start up!
Enable unencrypted boot modes ...
Enable unencrypted boot modes.Done!
```

图2.3 使能未签名固件启动

注意：制作好的启动卡用于配置iMX28x系列芯片的OTP熔丝，使芯片允许未签名固件启动，且该启动卡仅适用于签名密钥为全零的芯片（出厂默认芯片）。“允许未签名固件启动”的熔丝位只需要且只能配置一次，被配置过的芯片无须再配置。

#### 2.2.4 格式化NAND Flash



- 通过USB Boot引导格式化NAND Flash”完全相同。

## 2.3 TF卡烧写方案

TF卡烧写方案的相关固件在光盘文件的“TF卡烧写方案\”目录下，目录中有如图2.10所示的内容。



图2.10 “TF 卡烧写方案”的目录内容

该目录提供了 i.MX283\_for\_kernelsb 和 i.MX283\_for\_ubootsb 两个子目录，它们均可用于烧写固件，但是有所区别：

- MX283\_for\_kernelsb 目录包含的程序将在 EasyARM-iMX283 的 NAND Flash 上烧写“内核+文件系统”，烧写完成后，系统将在内核直接启动；
- MX283\_for\_ubootsb 目录包含的程序将在 EasyARM-iMX283 的 NAND Flash 上烧写“uboot+内核+文件系统”，烧写完成后，系统将在 uboot 启动，然后引导内核启动。

这两个目录包含的程序的操作方法一样的。

整个烧写过程分两步：制作系统恢复卡和进行固件烧写操作。

### 2.3.1 制作系统恢复卡

准备一张 TF 卡（**经验证，Class2 和 Class10 不能使用，推荐使用 Class4**）和一个读卡器。请确保该 TF 卡只有一个分区，并且是 FAT32 格式。若多个分区请先使用 Windows 的磁盘管理工具删除所有分区后再重建一个主分区。

把TF卡安装入读卡器，再把读卡器插入PC机的USB端口。这时Windows将在“我的电脑”中增加了一个驱动器，如图2.11所示为增加了F盘磁盘驱动器。



图2.11 添加的驱动器

进入i.MX283\_for\_kernelsb或i.MX283\_for\_ubootsb目录，双击“制作kernelsb启动TF卡.bat”或“制作ubootsb启动TF卡.bat”脚本文件，将弹出如图2.12所示的界面。

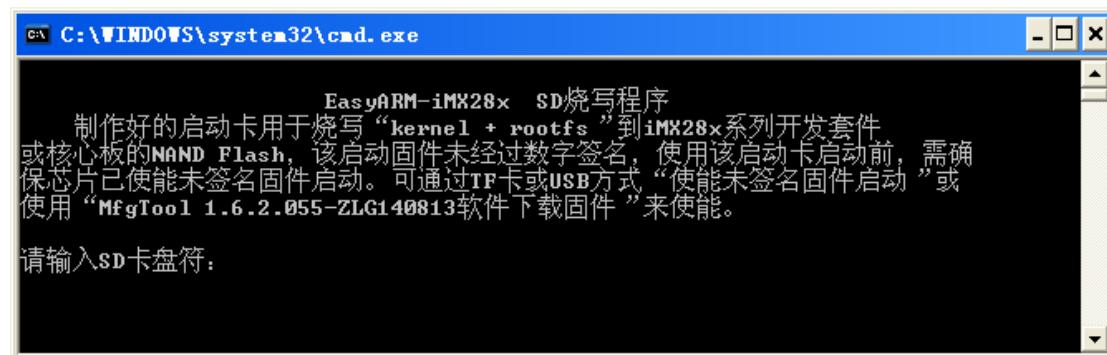




图2.12 提示用户输入读卡器的盘符

这是提示用户输入刚插入的读卡器的盘符，然后再键入Enter键。在笔者的电脑上是F盘，将显示如图2.13所示。

```
C:\WINDOWS\system32\cmd.exe

EasyARM-iMX28x SD烧写程序
制作好的启动卡用于烧写“kernel + rootfs”到iMX28x系列开发套件
或核心板的NAND Flash，该启动固件未经过数字签名，使用该启动卡启动前，需确
保芯片已使能未签名固件启动。可通过TF卡或USB方式“使能未签名固件启动”或
使用“MfgTool 1.6.2.055-ZLG140813软件下载固件”来使能。

请输入SD卡盘符:
F
注意:
    文件会被烧写在 F 盘

drive = F
removable = no
device block size = 512 bytes
device block count = 0x75a9e0

firmware size = 0x522190 bytes (0x2911 blocks)
extra blocks = 1774
Writing LDT...
Writing firmware...
Writing FAT partition...
```

图2.13 输入盘符

这时程序进入系统恢复卡制作过程，这里需要花几分钟的时间，制作完成后，将显示如图2.14所示的信息，这时请键入Enter键退出，至此，可用于烧写固件的恢复卡已经制作好。

```
C:\WINDOWS\system32\cmd.exe

EasyARM-iMX28x SD烧写程序
制作好的启动卡用于烧写“kernel + rootfs”到iMX28x系列开发套件
或核心板的NAND Flash，该启动固件未经过数字签名，使用该启动卡启动前，需确
保芯片已使能未签名固件启动。可通过TF卡或USB方式“使能未签名固件启动”或
使用“MfgTool 1.6.2.055-ZLG140813软件下载固件”来使能。

请输入SD卡盘符:
F
注意:
    文件会被烧写在 F 盘

drive = F
removable = no
device block size = 512 bytes
device block count = 0x75a9e0

firmware size = 0x522190 bytes (0x2911 blocks)
extra blocks = 1774
Writing LDT...
Writing firmware...
Writing FAT partition...
Writing MBR...
done! 已复制      1 个文件。
已复制      1 个文件。
烧写完毕，按键退出
```

图2.14 制作完成

### 2.3.2 系统恢复步骤

下面进入 EasyARM-iMX283 的系统恢复（**烧写固件**）操作，其步骤如下：

- (1) 把制作好的系统恢复卡插入到 EasyARM-iMX283 的 TF 卡卡槽。
- (2) 使用短路器短接 EasyARM-iMX283 上的JP1（**BZ**, 使能蜂鸣器）、JP3（**SD**, 设置为SD方式启动）及JP6（**WDT**, 禁用看门狗），如图2.15所示。

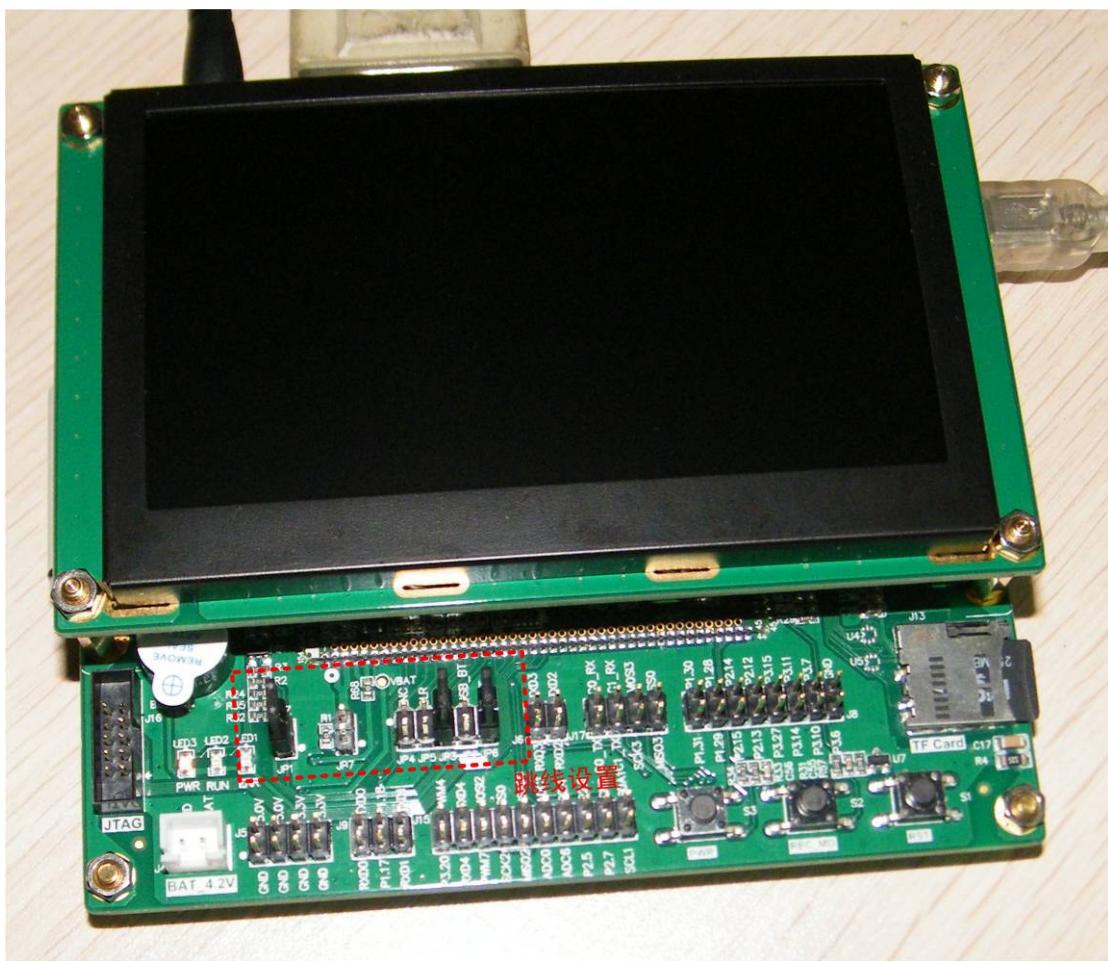


图2.15 TF 卡启动跳线设置

- (3) 使用串口延长线连接 PC 机和 EasyARM-iMX283 的 DUART (J7)。
- (4) 在 PC 机打开串口终端，监听串口数据。
- (5) 给 EasyARM-iMX283 重新上电或按 RST 键复位。这时 EasyARM-iMX283 自动进入固件烧写程序，同时在串口终端打印烧写过程信息，整个过程需要几分钟时间。

固件烧写完成后，EasyARM-iMX283 将在蜂鸣器发出“哔，哔，哔……”声音提示操作完成。这时拔出 JP3（**SD 启动选择**）的短路器，按“RST”复位键，EasyARM-iMX283 将从 NAND Flash 启动 Linux 系统。

## 2.4 USB烧写方案

这里主要介绍如何使用飞思卡尔提供的 MfgTool 工具来烧写固件。EasyARM-iMX283 的 Linux USB 烧写方案是提供了两种固件烧写方法：



- 仅烧写 U-Boot 到 NAND Flash，这种方法通常在 U-Boot 调试时用；
- 烧写“内核+文件系统”到 NAND Flash，固件烧写完成后，系统将直接在 Linux 内核启动。

说明：目前还不能支持 USB 直接烧写“uboot+内核+文件系统”；MfgTool 软件不支持 Win8 系统，请使用光盘或论坛中对应内存容量大小且版本为 1.6.2.055 的 MfgTool 软件，其他版本可能不能良好兼容 USB3.0 驱动。

#### 2.4.1 烧写U-Boot

烧写 U-Boot 步骤如下：

首先，进行硬件连接。先断开开发板电源，使用短路器短接 EasyARM-iMX283 底板的 JP6(WDT，短接禁能看门狗输出)、JP2(USB\_BT，设置为 USB 方式启动)，使用 MiniUSB 通信电缆接到 J12 接口，然后再在 J2 接上 5V 电源。

其次，选择烧写方案。运行光盘目录中的MfgTool.exe软件。打开MfgTool软件后点击菜单中的“Options”选择“Configuration...”，进入“Profiles”标签，在UTP\_UPDATE项的“选项”列中选择“NAND uboot Only”，然后点击“OK”，如图2.16所示。

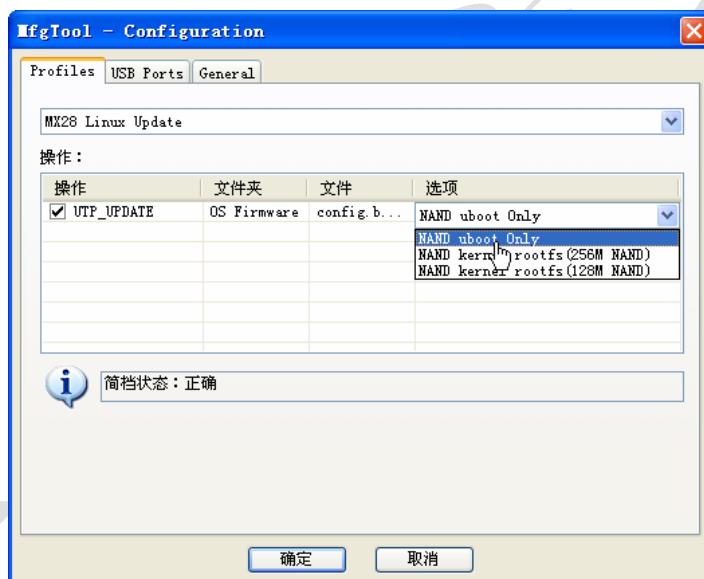


图2.16 选择 SinglechipNAND

接着，选择连接的HID设备。若软件已自动识别到连接的EasyARM-iMX283，则可以跳过这个步骤。切换到“USB Ports”标签，勾选已经连接上的的“HID-compliantdevice”（即 EasyARM-iMX283 设备）；然后点击“确定”，如图2.17所示。

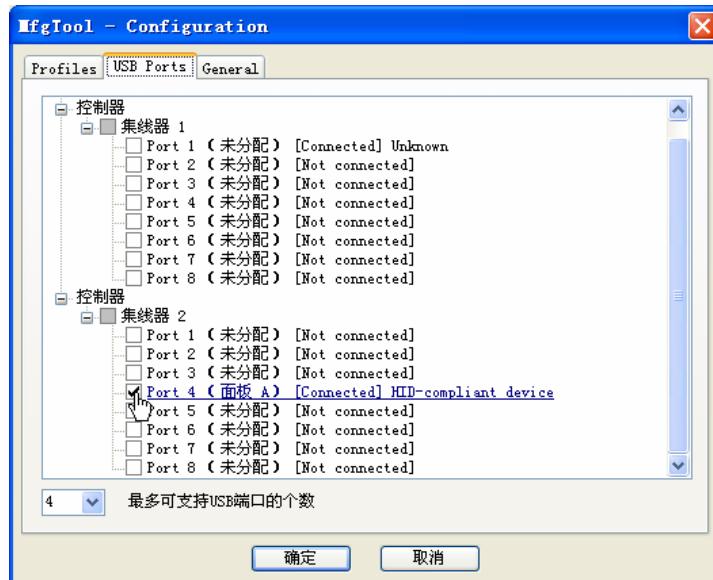


图2.17 勾选连接的 HID-compliantdevice

然后，返回到MfgTool主界面后显示软件正在监视HID-compliantdevice，如图2.18所示。此时点击“开始”进行U-Boot的烧写，如图2.19所示。直到烧写完成后点击“停止”，如图2.20所示。



图2.18 MfgTool 监视 HID-compliantdevice



图2.19 U-Boot 烧写



图2.20 烧写完成

烧写完成后，可以检验U-Boot的烧写是否正确：使用串口延长线连接EasyARM-iMX283的DUART端口和PC机，在PC机打开串口终端监听串口数据，拔掉JP2（**USB\_BT**）上的短路器，然后按底板上按RST键复位，这时EasyARM-iMX283将在NAND Flash上启动U-Boot，在串口终端打印的信息如图2.21所示。

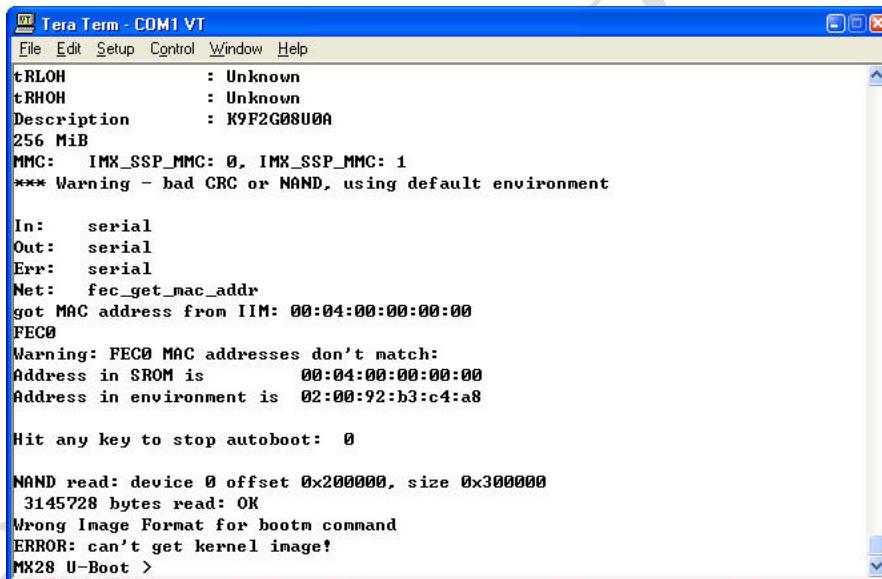


图2.21 U-Boot 启动信息。

上图出现了U-Boot读取内核错误的信息，这是正常现象。U-Boot在启动完成后，默认的操作是在NAND Flash读取内核，然后引导内核启动。但这时内核和文件系统都还没有烧写到NAND Flash，自然U-Boot也不能在NAND Flash读取到内核。

#### 2.4.2 烧写“内核+文件系统”

“内核+文件系统”的USB烧写方法和烧写U-Boot的方法是基本一样的。区别仅仅在选择烧写方案上，这里需要选择“NAND kerne rootfs (256M NAND)”或“NAND kerne rootfs (128M NAND)”，请以开发套件实际NAND容量大小为准，如图2.22所示。

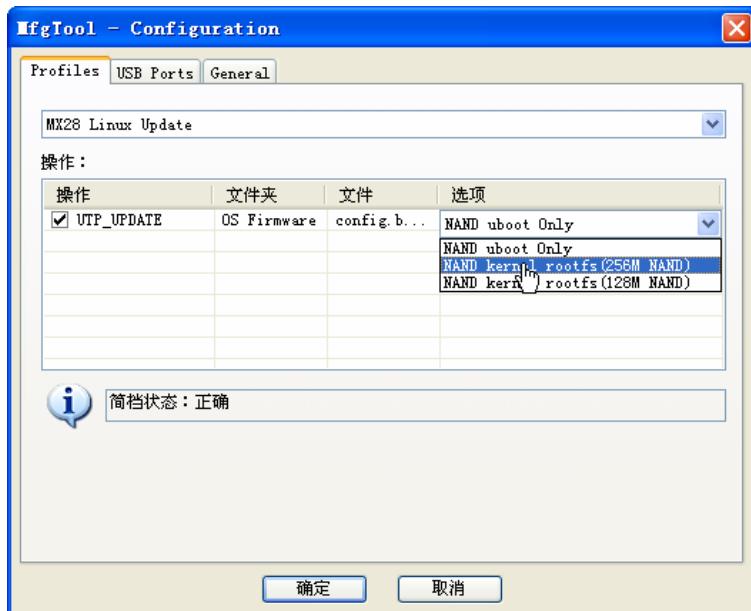


图2.22 烧写“内核+文件系统”

注意：在烧写过程中需要保持MiniUSB通信电缆的连接正常和EasyARM-iMX283 的正常供电，否则在烧写过程容易出错，若烧写失败则从2.2.2步骤开始重新操作。

## 2.5 网络烧写方案

### 2.5.1 所需条件

在使用网络烧写内核与文件系统前，需先确保 EasyARM-iMX283 的 NAND Flash 已经安装好 U-Boot（安装方法可以参考 [USB 烧写方案](#)），并且主机需要预先安装好 tftp 服务器。在 Linux 系统环境下的 tftp 服务器构建将在下文介绍，在这里以 Windows 环境下的“Cisco TFTP Server”工具软件为例进行介绍（[用户可在互联网上找到这个软件或类似功能的工具软件](#)）。

### 2.5.2 系统恢复步骤

使用网络进行 Linux 系统恢复的步骤如下：

- 使用短路器短接JP6（WDT，禁用看门狗），而JP2、JP3、JP4 及JP5 排针则保持断开。J7（DUART）接上串口，J10(NET)通过网线与主机连接，如图2.23所示；

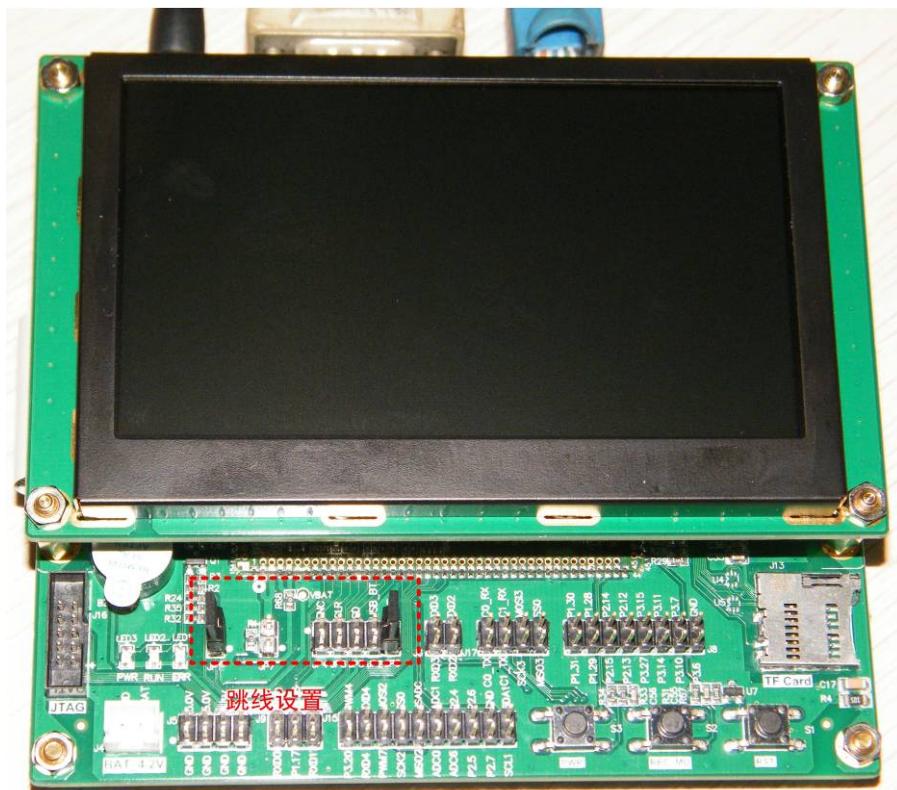


图2.23 网络烧写方案的跳线设置

按RST键复位或重新上电，系统即从NAND flash启动，运行预先安装好的U-Boot，U-Boot在串口终端输出系统的基本信息，当串口终端显示“Hit any key to stop autoboot %d”时立即输入空格键，进入U-Boot的命令行提示符模式，如图2.24所示。

```
Tera Term - COM1 VT
File Edit Setup Control Window Help
ECC Strength : 4 bits
ECC Size : 512 B
Data Setup Time : 20 ns
Data Hold Time : 10 ns
Address Setup Time: 20 ns
GPMI Sample Delay : 6 ns
tREA : Unknown
tRLOH : Unknown
tRHOB : Unknown
Description : K9F2G08U0A
256 MiB
MMC: IMX_SSP_MMC: 0, IMX_SSP_MMC: 1
In: serial
Out: serial
Err: serial
Net: fec_get_mac_addr
got MAC address from IIM: 00:04:00:00:00:00
FEC0
Warning: FEC0 MAC addresses don't match!
Address in SROM is 00:04:00:00:00:00
Address in environment is 02:00:92:b3:c4:a8
Hit any key to stop autoboot: 0
MX28 U-Boot > ■
```

图2.24 进入 U-Boot 命令行模式

把固件文件uImage及rootfs.ubifs（**在光盘文件中的“基本固件\”目录下，注意区分对应**

的DDR容量大小，不可混用）放到tftp服务器的根目录，并双击打开TFTPServer工具软件，如图2.25所示。

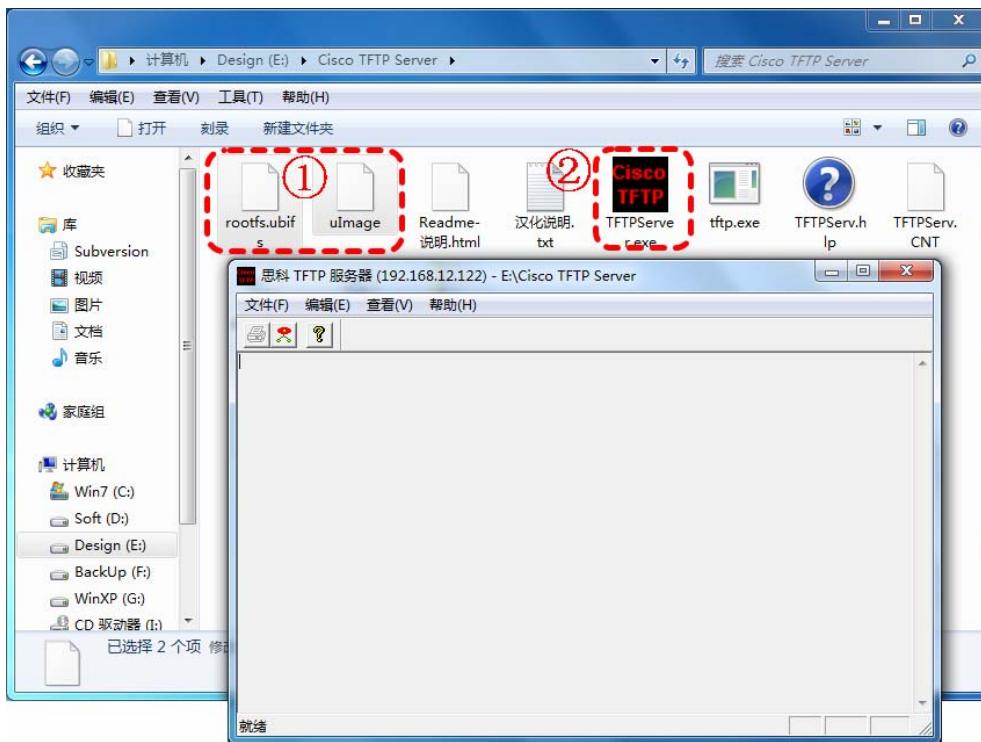


图2.25 启动 Cisco TFTP Server 工具软件

查看和设置tftp服务器的根目录，操作方法如图2.26所示，在本例中，将tftp服务器的根目录设置在Cisco TFTP Server工具软件的目录下。

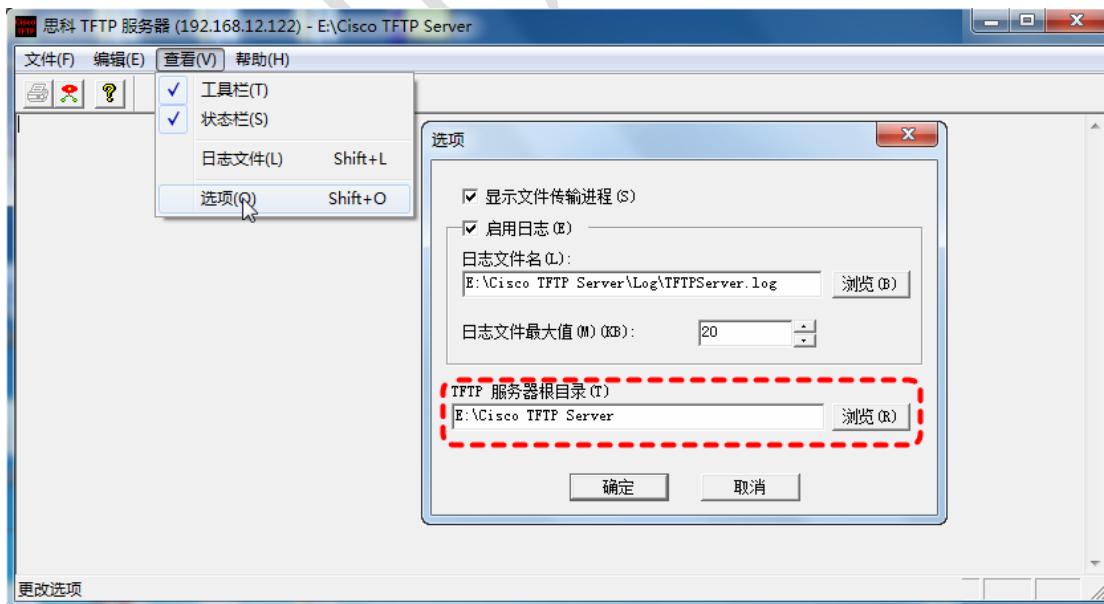


图2.26 设定 tftp 服务器根目录

通过串口终端配置 U-Boot 的 IP 和 tftp 服务器 IP（即 Cisco TFTP Server 工具软件所在系统的 IP），并保存。

其中 U-Boot 与 tftp 服务器的 IP 必须在同一个网段。本示例中主机的 IP 为 192.168.12.122，



EasyARM-iMX283 的 U-Boot 的 IP 设置为 192.168.12.124，串口终端命令如下：

```
MX28 U-Boot > setenv ipaddr 192.168.12.124  
MX28 U-Boot > setenv serverip 192.168.12.122  
MX28 U-Boot > saveenv
```

命令执行完后，串口终端如图2.27所示。

The screenshot shows the Tera Term window titled "Tera Term - COM1 VT". The terminal window displays the following text:

```
Description      : K9F2G08U0A  
256 MiB  
MMC: IMX_SSP_MMC: 0, IMX_SSP_MMC: 1  
**** Warning - bad CRC or NAND, using default environment  
  
In:   serial  
Out:  serial  
Err:  serial  
Net:  fec_get_mac_addr  
got MAC address from IIM: 00:04:00:00:00:00  
FEC0  
Warning: FEC0 MAC addresses don't match:  
Address in SROM is    00:04:00:00:00:00  
Address in environment is 02:00:92:b3:c4:a8  
  
Hit any key to stop autoboot: 0  
MX28 U-Boot > setenv ipaddr 192.168.12.124  
MX28 U-Boot > setenv serverip 192.168.12.122  
MX28 U-Boot > saveenv  
Saving Environment to NAND...  
Erasing Nand...  
Erasing at 0x1000000000000000 -- 0% complete.  
Writing to Nand... done  
MX28 U-Boot >
```

图2.27 配置并保存 U-Boot 参数

在执行烧写之前需要先确保 EasyARM-iMX283 与 tftp 服务器的网络连接畅通，可以使用 ping 命令进行测试。如果出现“host x.x.x.x is alive”这样的提示，则表示网络连接正常：

```
MX28 U-Boot > ping 192.168.12.122  
Using FEC0 device  
host 192.168.12.122 is alive #表示网络连接畅通
```

如果出现“host x.x.x.x is not alive”的提示，则表示网路有故障，请检查硬件连接或者网络配置：

```
MX28 U-Boot > ping 192.168.12.122  
Using FEC0 device  
ping failed; host 192.168.12.122 is not alive #表示网络不通
```

在 EasyARM-iMX283 与 tftp 服务器的网络连接畅通的条件下，执行如下命令即可下载内核及根文件系统：

```
MX28 U-Boot > run upsystem
```

内核与根文件系统传输需要几分钟时间，操作正常时，串口终端如图2.28所示。内核与根文件系统将自动保存到NAND flash。

run upsystem 命令执行完后系统会自动重启。系统重启后，先运行 U-Boot，然后自动启动内核。在系统启动完成后可以使用 root 用户名（密码也是 root）通过串口终端进入系统。

其中 run upsystem 命令可以拆分成 run upkernel 和 run uproofs 两条命令，分别更新内核



和文件系统。

```
FEC0
Warning: FEC0 MAC addresses don't match:
Address in SROM is      00:04:00:00:00:00
Address in environment is 02:00:92:b3:c4:a8

Hit any key to stop autoboot: 0
MX28 U-Boot > setenv ipaddr 192.168.12.124
MX28 U-Boot > setenv serverip 192.168.12.122
MX28 U-Boot > saveenv
Saving Environment to NAND...
Erasing Nand...
Erasing at 0x100000000020000 -- 0% complete.
Writing to Nand... done
MX28 U-Boot > ping 192.168.12.122
Using FEC0 device
host 192.168.12.122 is alive
MX28 U-Boot > run upsyste
Using FEC0 device
TFTP from server 192.168.12.122; our IP address is 192.168.12.124
Filename 'uImage'.
Load address: 0x42000000
Loading: T ######
```

图2.28 利用 tftp 服务器下载内核及根文件系统

但若tftp服务器不可访问（**即使网络可以ping通**），将会导致run upsyste命令执行失败，此时串口终端显示如图2.29所示，此时需要检查是否忘记打开tftp服务器或者tftp服务器软件被相关防火墙拦截。

```
Net: fec_get_mac_addr
got MAC address from IIM: 00:04:00:00:00:00
FEC0
Warning: FEC0 MAC addresses don't match:
Address in SROM is      00:04:00:00:00:00
Address in environment is 02:00:92:b3:c4:a8

Hit any key to stop autoboot: 0
MX28 U-Boot > setenv ipaddr 192.168.12.124
MX28 U-Boot > setenv serverip 192.168.12.122
MX28 U-Boot > saveenv
Saving Environment to NAND...
Erasing Nand...
Erasing at 0x100000000020000 -- 0% complete.
Writing to Nand... done
MX28 U-Boot > ping 192.168.12.122
Using FEC0 device
host 192.168.12.122 is alive
MX28 U-Boot > run upsyste
Using FEC0 device
TFTP from server 192.168.12.122; our IP address is 192.168.12.124
Filename 'uImage'.
Load address: 0x42000000
Loading: T T T T T T T T T T
```

图2.29 tftp 服务器访问超时



若tftp服务器根目录下未放置uImage及rootfs.ubifs文件，在执行run upsysterm命令时将会提示“File not found ...”，如图2.30所示，这时只需要将光盘文件中的uImage及rootfs.ubifs文件放到tftp服务器的根目录下即可。

The screenshot shows a window titled "Tera Term - COM1 VT". The menu bar includes File, Edit, Setup, Control, Window, and Help. The main window displays the following U-Boot log:

```
run upsysterm
Using FEC0 device
TFTP from server 192.168.12.122; our IP address is 192.168.12.124
Filename 'uImage'.
Load address: 0x42000000
Loading: T
TFTP error: 'File not found' !!!HFFEPEKFFEOCACACACACACACA
Starting again

Using FEC0 device
TFTP from server 192.168.12.122; our IP address is 192.168.12.124
Filename 'uImage'.
Load address: 0x42000000
Loading: *
TFTP error: 'File not found' !!!FGEFFCDCCACACACACACACACAAA
Starting again
```

图2.30 tftp 服务器根目录下没有对应的文件

注意：EasyARM-iMX283 的 U-Boot 可以连接 100Mb 的全双工、半双工的网络，不可以连接 10Mb 的全双工、半双工的网络（在进入系统后都能连接）。



### 3. 启动选择和系统基本设置操作

本章主要介绍基于 Linux 系统的 EasyARM-iMX283 学习套件的启动选择及系统基本设置操作。

#### 3.1 系统启动跳线设置

EasyARM-iMX283 底板上有 5 个跳线的设置，用于设置系统的启动方式，位置如图3.1所示。

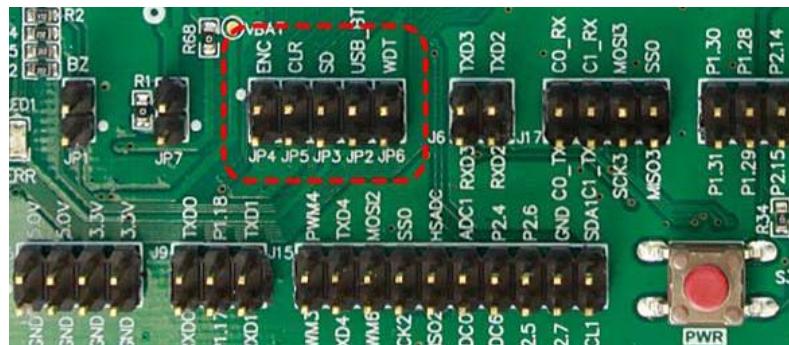


图3.1 启动跳线设置

具体描述与默认设置如表3.1所示。

表3.1 核心板配置信号功能描述

标号	说 明
JP6 (WDT)	硬件看门狗禁能控制，出厂演示的 Linux 系统未实现看门狗控制，故需短接 JP6，以禁止硬件看门狗功能，否则系统将不断重启
JP2 (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；断开后，则从 NAND 或 TF 卡启动，默认需断开
JP3 (SD)	JP2 断开后，短接 JP3 系统从 TF 卡启动，断开 JP3 则系统从 NAND 启动，默认断开
JP5 (CLR)	WinCE 下短接 JP5 将在启动时清除注册表，Linux 下暂未使用，请保持断开
JP4 (ENC)	暂未适用，请保持断开

注意：默认从 NAND 启动时，只需短接 WDT，其他全部断开。

#### 3.2 系统登录

系统启动后，会进入 zy launcher 桌面。

注意：鼠标要提前插入开发板，因为 Qt 不能动态检测鼠标。

串口登录账号：root，密码：root。

注意：需要将开发套件的 J7(DUART)通过串口线延长线连接至打开串口终端的 PC 机。

SSH 登录账号：root，密码：root。

使用 SSH 登录前，需要先在串口终端设置好开发板 IP（需连接串口延长线至 PC 主机），并通过网线将开发套件接入局域网（与 PC 主机在同一个局域网），命令如下：

```
root@EasyARM-iMX283 ~# ifconfig eth0 192.168.12.124 #此为开发板 IP，需要与电脑在同一网段
```

此时可以尝试 ping 一下 PC 机的 IP 是否连通。

在 windows 下可以使用 putty/xShell 登录开发板。在 Linux 主机可以通过 ssh 命令登录，



命令如下：

```
vmuser@Linux-host ~$ ssh root@192.168.12.124      # 192.168.12.124 为开发板IP地址
```

### 3.3 网络设置

#### 3.3.1 设置IP和子网掩码

在嵌入式 Linux 系统下，使用 ifconfig 命令可以显示或配置网络设备，其常用的组合命令格式如下：

```
ifconfig 网络端口 IP 地址 hw<HW> ether MAC 地址 netmask 掩码地址 broadcast 广播地址 [up|down]
```

具体示例指令如下：

```
ifconfig eth0 192.168.12.124 hw ether 00:11:22:33:44:55 netmask 255.255.255.0 broadcast 192.168.12.255 up
```

通过串口终端发送示例指令后，开发套件的 Linux 系统将配置网卡 eth0 的网络参数（IP 地址为：192.168.12.124，MAC 地址为：00:11:22:33:44:55，子网掩码为：255.255.255.0，广播地址为：192.168.12.255）并启动该网卡。

配置命令执行后，发送“ifconfig”命令可以查看当前的网络设备状态，如图3.2所示。

The screenshot shows a terminal window titled "Tera Term - COM1 VI". The window has a menu bar with File, Edit, Setup, Control, Window, Help. The main pane displays the following terminal session:

```
root@EasyARM-iMX283 ~# ifconfig eth0 192.168.12.124 hw ether 00:11:22:33:44:55 n
etmask 255.255.255.0 broadcast 192.168.12.255 up
root@EasyARM-iMX283 ~# ifconfig
eth0      Link encap:Ethernet HWaddr 00:11:22:33:44:55
          inet addr:192.168.12.124 Bcast:192.168.12.255 Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
          RX packets:6490 errors:0 dropped:0 overruns:0 frame:0
          TX packets:15 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:479718 (468.4 KiB)  TX bytes:1002 (1002.0 B)

root@EasyARM-iMX283 ~#
```

图3.2 配置及查看网络设备

ifconfig 命令也可以单独设置 IP 地址或其他网络参数，设置后即时生效。

#### 3.3.2 设置默认网关

添加、删除或查看网关参数使用“route”命令，如需要将默认网关设置为 192.168.12.1，其示例指令如下：

```
route add default gw 192.168.12.1
```

若需要删除该网关设置，其示例指令如下：

```
route del default gw 192.168.12.1
```

若需要查看当前网关设置，其示例指令如下：

```
route -n
```

#### 3.3.3 设置DNS

DNS 记录在“/etc/resolv.conf”配置文件中，若开发套件需要使用域名访问互联网，则需要先设定 DNS，否则访问可能不正常。可通过 vi 编辑器编辑“/etc/resolv.conf”文件的方



法设定开发套件的 DNS。使用 vi 命令打开配置文件指令如下：

```
root@EasyARM-iMX283 ~# vi /etc/resolv.conf
```

打开 “/etc/resolv.conf” 文件后在其中添加 DNS 配置，可以添加多行，若首选 DNS 及备用 DNS 分别为 192.168.0.1 和 192.168.0.2，则其示例配置如下所示：

```
# nameserver ip address  
nameserver 192.168.0.1  
nameserver 192.168.0.2
```

文件修改后，需要保存并退出 vi 编辑器，文件保存后对 DNS 的修改即时生效。

注意：关于 Linux 下怎样使用 vi 命令编辑文件，在“嵌入式开发环境构建”一章将有简单介绍。

### 3.3.4 注意事项

需要注意的是，通过串口终端配置的网络参数将会在断电或复位后丢失。如需上电自动设置 IP，可以通过如下方式实现：

在串口终端中输入如下命令，通过 vi 编辑器打开 “/etc/rc.d/init.d/start\_userapp” 文件：

```
root@EasyARM-iMX283 ~# vi /etc/rc.d/init.d/start_userapp
```

然后通过 vi 命令编辑开发板中 /etc/rc.d/init.d/start\_userapp 文件，增加网络配置命令行后保存并退出 vi 编辑器。

```
#!/bin/sh  
  
#you can add your app start_command three  
ifconfig eth0 192.168.12.124  
route add default gw 192.168.12.1  
  
#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 &
```

注意：文件修改并保存后不要直接按复位键，要执行 reboot 命令重启或者先执行 sync 同步命令，否则文件可能会没有被同步到 NAND。

## 3.4 TF卡使用

把 TF 卡插入 EasyARM-iMX283 的 TF 插槽后，Linux 系统将会自动检测到 TF 卡，并自动挂载到 /media/sd-mmcblk%d 目录下。

注意：在系统启动尚未完成阶段，插拔 TF 卡或部分其他设备可能导致系统启动故障。TF 卡推荐使用 FAT32 文件系统，其他文件系统可能不会被识别。若 TF 卡有多个分区则挂载的目录为 /media/sd-mmcblk%dp%d。



### 3.5 U盘使用

把 U 盘插入到 EasyARM-iMX283 的 USB HOST 接口上后, Linux 系统将会检测到 U 盘, 并自动挂载到/media/usb-sda%d 目录下。

注意: U 盘推荐使用 FAT32 文件系统, 其他文件系统可能不会被识别。

### 3.6 USB Device 使用

EasyARM-iMX283 的 OTG 接口支持 USB Device 功能, 可以把开发板模拟成一个 U 盘。虚拟成 U 盘需要加载开发板上的/root/g\_file\_storage.ko 驱动, 然后创建一个 loop 类型文件, 该文件可以放置在任何目录:

```
root@EasyARM-iMX283 ~# dd if=/dev/zero of=/dev/shm/disk bs=1024 count=10240
```

该命令表示在/dev/shm/目录创建一个大小为 10M 的 disk 文件。

然后加载驱动:

```
root@EasyARM-iMX283 ~# insmod /root/g_file_storage.ko stall=0 file=/dev/shm/disk removable=1
```

使用A-MiniUSB线连接开发板OTG接口和PC机。这时在PC机中“我的电脑”里即可看见多了一个可移动磁盘的驱动器, 可通过右键菜单查看其属性或进行格式化, 如图3.3所示。

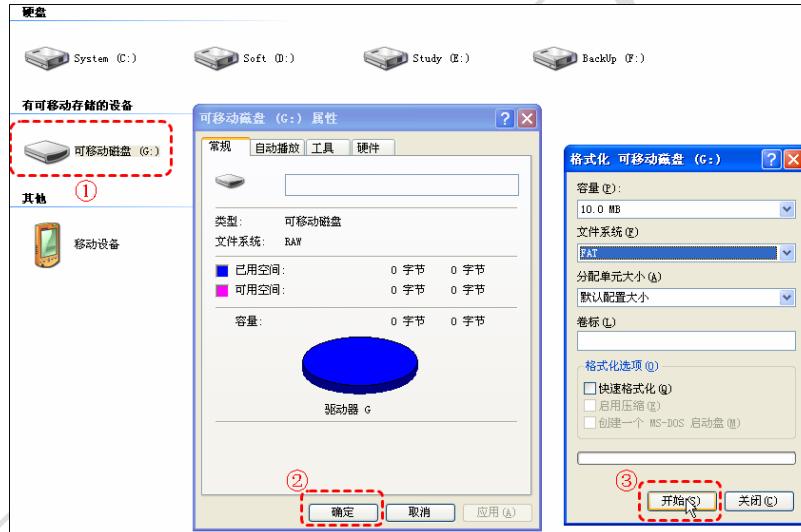


图3.3 EasyARM-iMX283 模拟出来的 U 盘

将 EasyARM-iMX283 模拟出来的 U 盘格式化, 并在该分区下创建一个文件夹, 命名为 iMX-U-Disk。若想在目标 Linux 系统下看到这个 U 盘中的文件, 需要先将这个 U 盘从电脑上移除, 然后通过串口终端把前面创建的这个 loop 类型文件挂载到一个指定目录, 如将其挂载到/mnt 目录下, 输入如下指令:

```
root@EasyARM-iMX283 ~# mount /dev/shm/disk /mnt
```

这时即可在/mnt目录下看到在电脑写入的文件(若开发套件在系统上电前已接入USB鼠标, 则可通过USB鼠标操作, 浏览到该目录; 若系统支持触摸屏, 也可以直接通过触摸屏操作, 浏览到该目录), 如图3.4所示。

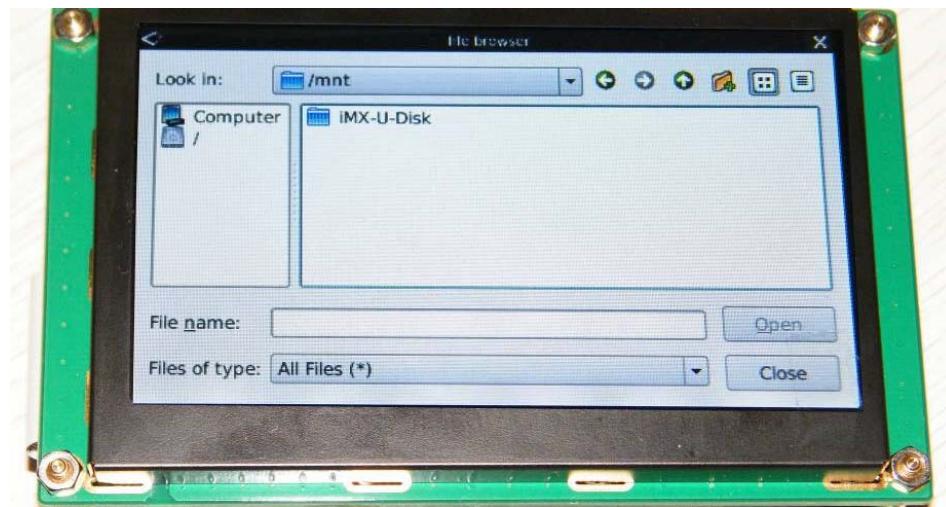


图3.4 在 Linux 系统中浏览 U 盘中的文件

### 3.7 LED使用

在 EasyARM-iMX283 上有 POWER、RUN、ERR 三个 LED。

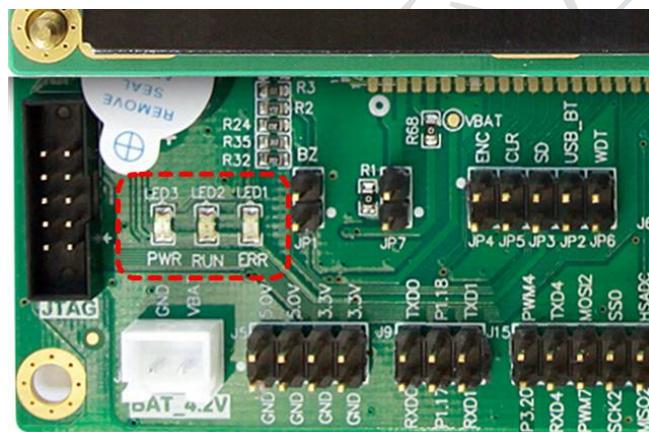


图3.5 LED 所在位置

POWER 是指示系统的电源状态；RUN 是系统心跳灯，不断闪烁表示系统正在运行。

ERR 是留给用户使用，其操作接口为 /sys/class/leds/led-err 文件。写入 1 点亮 LED，写入 0 则熄灭 LED。

操作示例：

```
root@EasyARM-iMX283 ~# echo 1 >/sys/class/leds/led-err/brightness      #控制 LED 点亮  
root@EasyARM-iMX283 ~# echo 0 >/sys/class/leds/led-err/brightness      #控制 LED 熄灭
```

### 3.8 蜂鸣器使用

EasyARM-iMX283 上的蜂鸣器控制操作文件是 /sys/class/leds/beep/brightness。写入 1 使蜂鸣器鸣叫，写入 0 停止鸣叫。

操作示例：

```
root@EasyARM-iMX283 ~# echo 1 >/sys/class/leds/beep/brightness      #控制蜂鸣器鸣叫  
root@EasyARM-iMX283 ~# echo 0 >/sys/class/leds/beep/brightness      #控制蜂鸣器停止鸣叫
```



注意：使用蜂鸣器需要短接上蜂鸣器使能排针 JP1 (BZ)，否则蜂鸣器不会鸣叫。

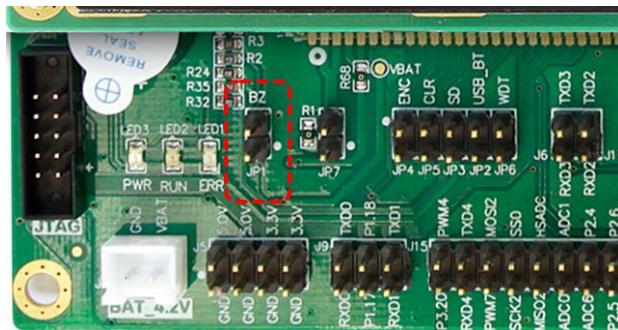


图3.6 蜂鸣器使能排针位置

### 3.9 LCD背光控制

在/sys/class/backlight/mxs-bl 目录下包含了 LCD 背光控制的属性文件：brightness。该文件可以设置的值为 0~100 之间：当设置为 0 时，背光最暗；当设置为 100 时，背光最亮，其设置命令如下：

```
root@EasyARM-iMX283 ~# echo 100 > /sys/class/backlight/mxs-bl/brightness
```

EasyARM-iMX283 开发套件出厂时的亮度默认值为 80，查看当前设定的亮度值可输入如下指令：

```
root@EasyARM-iMX283 ~# cat /sys/class/backlight/mxs-bl/brightness
```

### 3.10 系统时间设置

由于 EasyARM-iMX283 未外扩 RTC 电路，所以系统启动时将从处理器内部 RTC 获取系统的初始时间，在系统未接电池的条件下，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。由于演示的 Linux 系统未设计关机按钮，在此条件下，需要执行 halt 命令将系统“关机”进入低功耗。

设置系统 RTC 时间，使用 date 和 hwclock 命令进行，假定设置系统时间为 2014-05-07, 10:30:10，则可用如下命令：

```
root@EasyARM-iMX283 ~# date 2014.05.07-10:30:10      #设置系统时间  
Wed May 7 10:30:10 UTC 2014  
root@EasyARM-iMX283 ~# hwclock -w                  #将时间写入 RTC
```

**注意：若需要保持系统时间，在设置系统时间后，一定要使用“hwclock -w”将时间写入处理器的 RTC，同时需要在 J4 上接上电池。**



## 4. 嵌入式开发环境构建

本章介绍嵌入式 Linux 的开发环境，包括虚拟机安装、Linux 操作系统安装、必要的服务器搭建以及交叉工具链安装测试等。可能用到的下载链接如下：

Ubuntu 64-bit 12.04 for EasyARM-iMX283.iso 光盘致远服务器下载地址：

[http://www.zlgmcu.com/Freescale/pdf/EasyARM-iMX283\\_ubuntu.zip](http://www.zlgmcu.com/Freescale/pdf/EasyARM-iMX283_ubuntu.zip)

VMware Player 下载地址：

[https://my.vmware.com/web/vmware/free#desktop\\_end\\_user\\_computing/vmware\\_player/6\\_0|PLAYER-602|product\\_downloads](https://my.vmware.com/web/vmware/free#desktop_end_user_computing/vmware_player/6_0|PLAYER-602|product_downloads)

SSH Secure Shell Client 下载地址：

[http://www.onlinedown.net/softdown/20089\\_2.htm](http://www.onlinedown.net/softdown/20089_2.htm)

### 4.1 嵌入式Linux开发简介

嵌入式Linux系统由于资源有限，通常无法安装本地编译器进行本地开发，通常需要借助一台主机进行交叉开发。主机运行Linux操作系统，并安装相应的交叉编译器，将在主机编辑好的程序代码交叉编译后，通过一定方式（**如以太网或者串口**）将编译生成的可执行文件下载到目标系统运行或者调试，其基本开发环境模型如图4.1所示，而交叉开发流程则如图4.2所示。

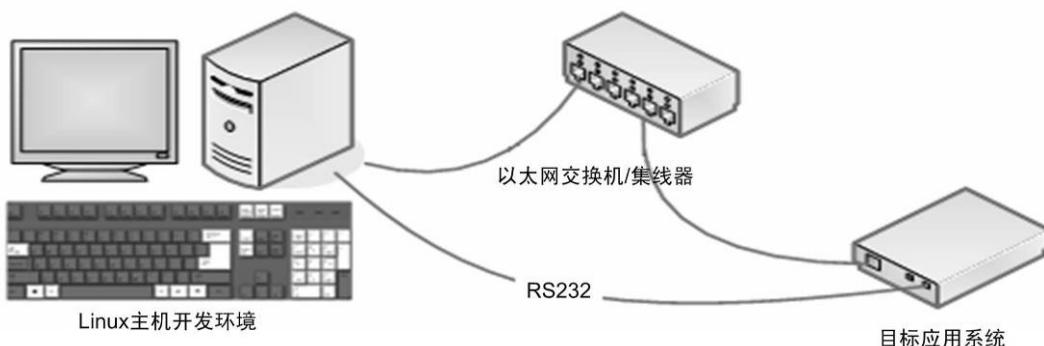


图4.1 嵌入式 Linux 开发环境模型

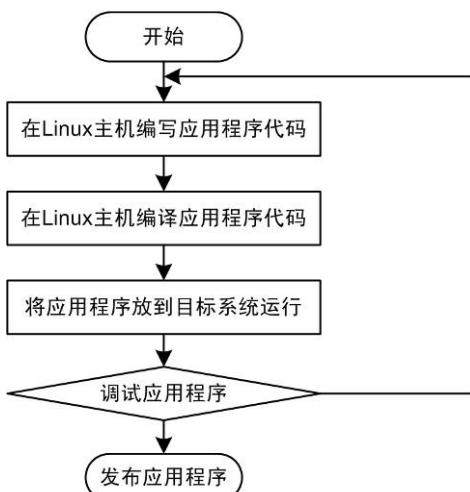


图4.2 嵌入式 Linux 交叉开发流程



主机硬件要求：至少需要有 4.5GB 空闲的硬盘空间（**实际安装完约占 4GB 空间，为了保证后期开发正常进行，建议空闲硬盘空间不低于 20GB**），1GB 的内存（**若是在虚拟机中装 Linux 系统，为了保证流畅运行，建议不低于 2GB 物理内存**），为了调试方便，还需要能联网及支持串口（**USB 转 RS232 的串口也可以**）通信。

Linux 发行版要求：推荐使用 Ubuntu-12.04 版的 64 位发行版 Linux 系统，同时还需要安装其它所需的开发软件，如交叉编译器、NFS 服务器、TFTP 服务器等，为了避免用户在主机环境搭建方面花费过多的时间，推荐用户从 ZLG 网站下载安装“Ubuntu 64-bit 12.04 for EasyARM-iMX283.iso”，里面已集成 EasyARM-iMX283 Linux 的开发所需的 NFS 服务器、tftp 服务器、交叉编译器及 UBIFS 文件系统生成工具等软件。

## 4.2 安装Linux主机操作系统

主机操作系统是嵌入式 Linux 开发的重要平台，用户可以直接在 PC 上安装 Ubuntu 操作系统，也可以在 Windows 操作系统中通过虚拟机安装 Ubuntu 操作系统，常用的虚拟机软件有 VMware、Virtual Box 和 Virtual PC 等，下文以 VMware 的 VMware Player 为例，介绍如何在虚拟机中安装 Ubuntu。

### 1. 下载待安装Ubuntu镜像文件

在浏览器中输入[http://www.zlg.cn/IPC/product\\_detail.php?id=14](http://www.zlg.cn/IPC/product_detail.php?id=14)网址，打开光盘资料下载页面，如图4.3所示，可以选择致远服务器或者其他云盘下载，下载后将文件解压到D盘根目录下，如图4.4所示。



图4.3 下载光盘镜像文件

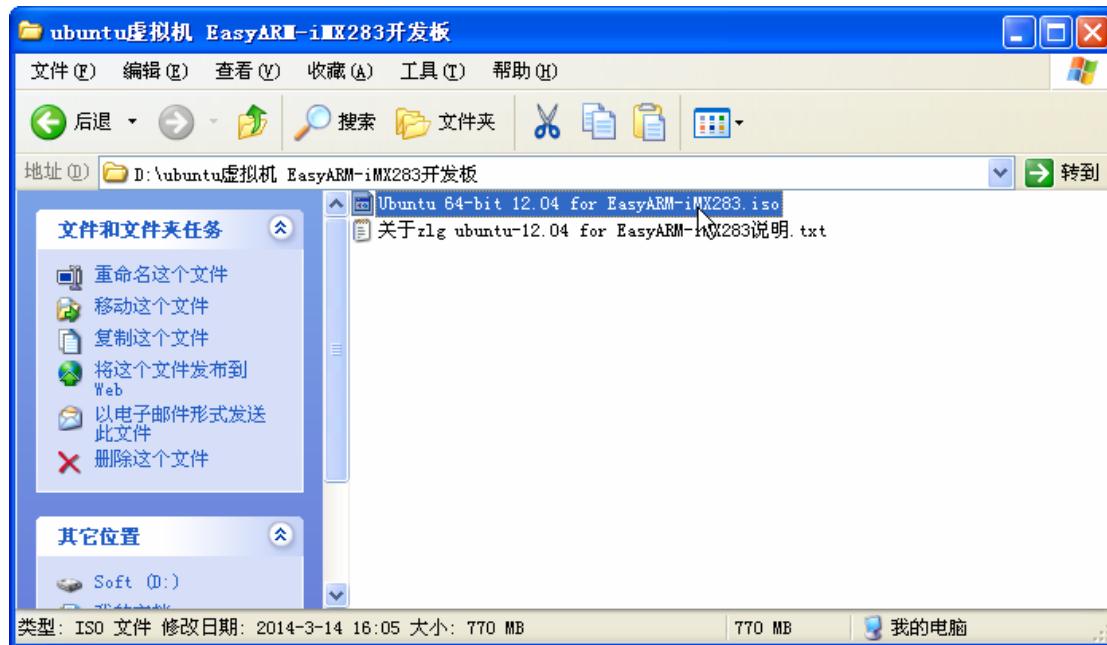


图4.4 解压下载文件得到安装镜像文件

## 2. 下载及安装VMware Player软件

用户可以在VMware的官方网站上下载到非商用的VMware Player软件，在下载页面中选择下载VMware Player for Windows 32-bit and 64-bit软件，如图4.5所示。

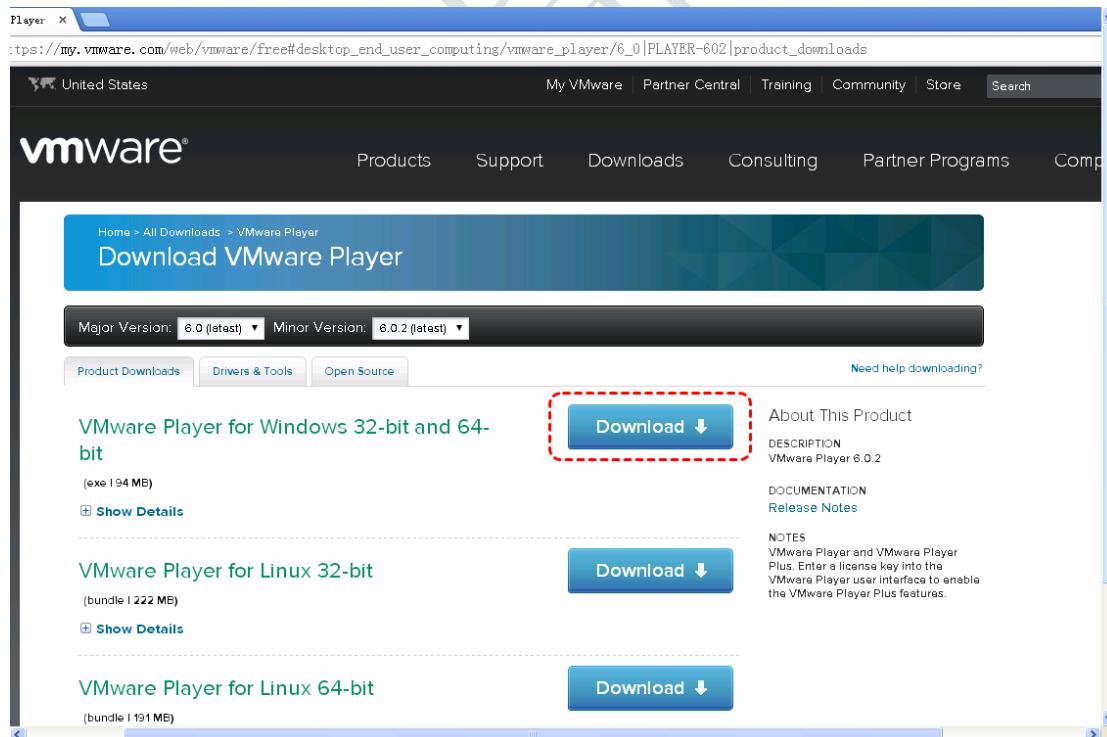


图4.5 下载 VMware Player

文件下载完成后，双击下载到的应用程序安装文件VMware-player-6.0.2-1744117.exe（具体文件名以实际下载到的文件为准），然后在弹出的对话框中选择“下一步”，如图4.6所示。



图4.6 安装 VMware Player

在弹出的对话框中选择“接受许可”，如图4.7所示，然后按默认设置一直点击“下一步”直至如图4.8所示界面，此时点击“继续”按钮即可进行VMware Player软件的安装，安装完成时如图4.9所示。



图4.7 接受许可协议



图4.8 准备安装



图4.9 完成安装

### 3. 在虚拟机中安装Linux系统

装完 VMware Player 软件后，需要重启系统，并进入 BIOS 系统，开启 CPU 对虚拟机指令的支持，若已经开启该功能的则可以直接跳过 BIOS 设置环节。不同的 PC 进入 BIOS 的方法略有不同（笔记本上通常为“刚启动时持续按 F2 键”，而台式 PC 则是“在刚启动时持续按 Del 键”，也有部分主板是需要按“F10”键的），请以实际情况为准。

当进入 BIOS 系统，找到 Intel Virtualization Technology 选项，将其配置为 Enable，如图4.10 所示。需要注意的是，不同PC的BIOS中对应的选项位置及描述可能不同，请以实际情况为准。

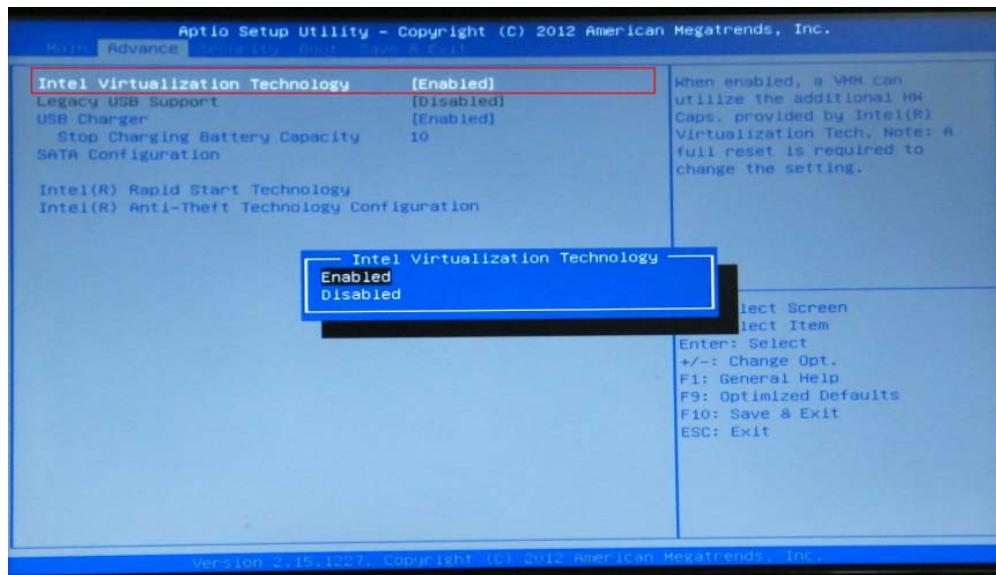


图4.10 使能“Intel Virtualization Technology”

将 Intel Virtualization Technology 设置为 Enable 后，按 F10 保存并退出 BIOS 设置。修改 BIOS 该选项后需要冷重启一次系统才能生效，即关机后再开机。

冷重启系统后，双击桌面的 VMware Player 启动快捷方式图标 “” 打开 VMware Player 软件，点击弹出窗口中的“创建新虚拟机(N)”链接，如图4.11所示。



图4.11 创建新虚拟机

在弹出的向导对话框中选择“稍后安装操作系统(S)”，然后点击“下一步”按钮，如图4.12所示。

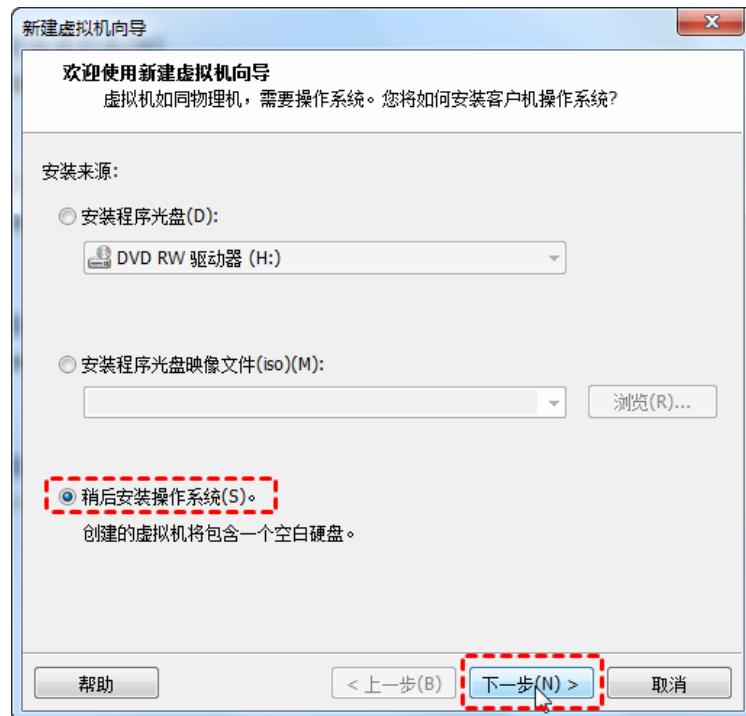


图4.12 选择“稍后安装操作系统”

再在下一步的窗口中选择“Linux(L)”客户机操作系统，然后继续点击“下一步”按钮，如图4.13所示。



图4.13 选择客户机操作系统

紧接着将虚拟机名称设置为Ubuntu 64-bit，并将其存储位置为“D:\My Virtual Machines\Ubuntu 64-bit”，如图4.14所示，建议虚拟机位置所在分区至少需要有20GB的可用空间。

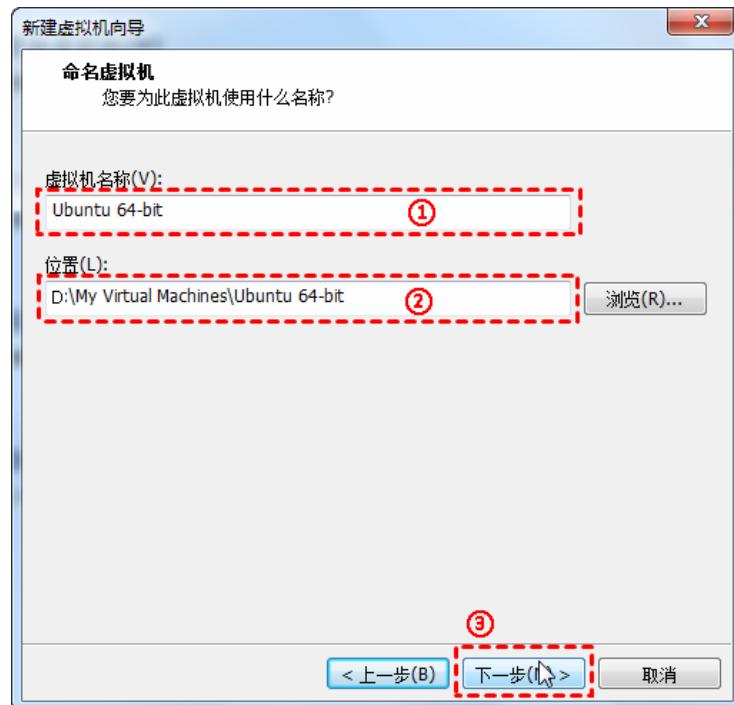


图4.14 设置虚拟机名称及存储位置

设置完虚拟机名称及存储位置后，需要按图4.15所示指定虚拟磁盘容量及磁盘文件存储方式。

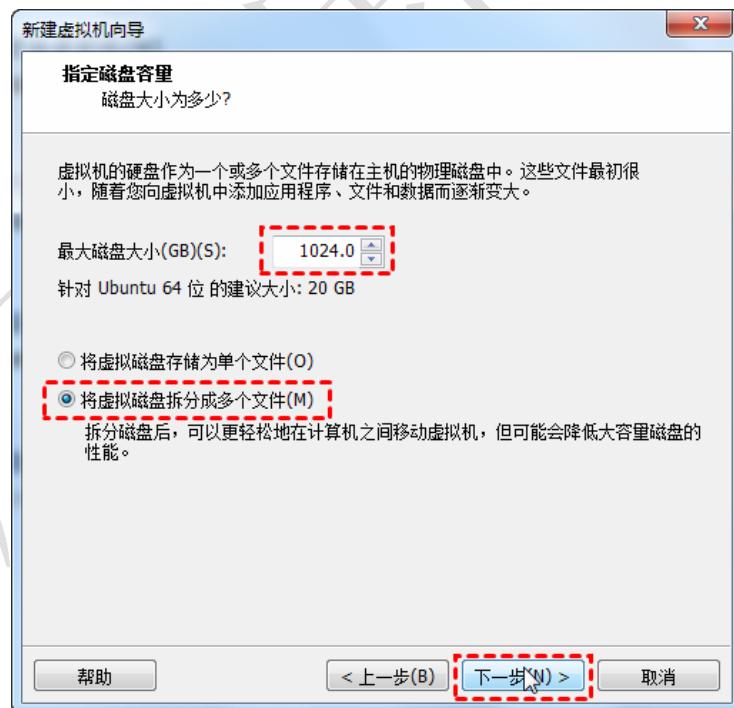


图4.15 指定虚拟磁盘容量

设置完虚拟磁盘容量后，点击“下一步”按钮，然后参考图4.16~图4.18所示对虚拟机其他硬件进行配置，其中需要将虚拟机的“网络适配器”配置为“桥接”模式，将“CD/DVD”驱动器配置为“使用ISO映像文件”，并将下载解压后待安装的Ubuntu映像文件“Ubuntu 64-bit 12.04 for EasyARM-iMX283.iso”作为虚拟机的“CD/DVD”驱动器。

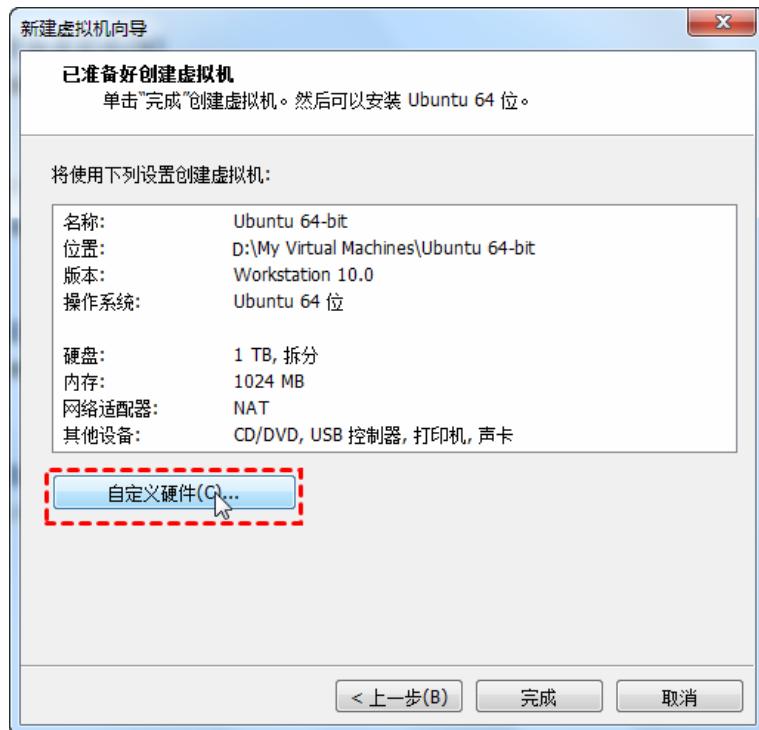


图4.16 点击“自定义硬件”

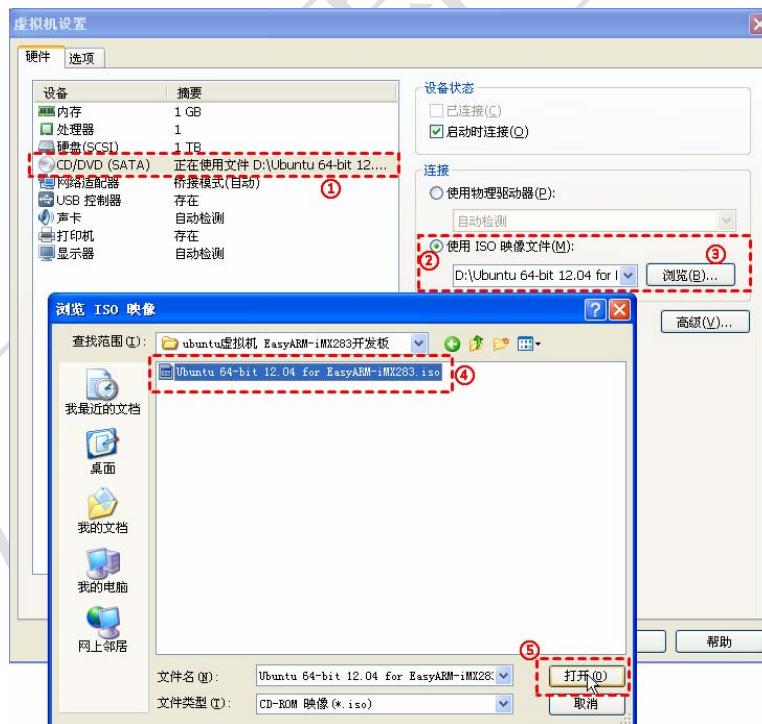


图4.17 将“CD/DVD”驱动器配置为待安装的 Ubuntu 映像文件

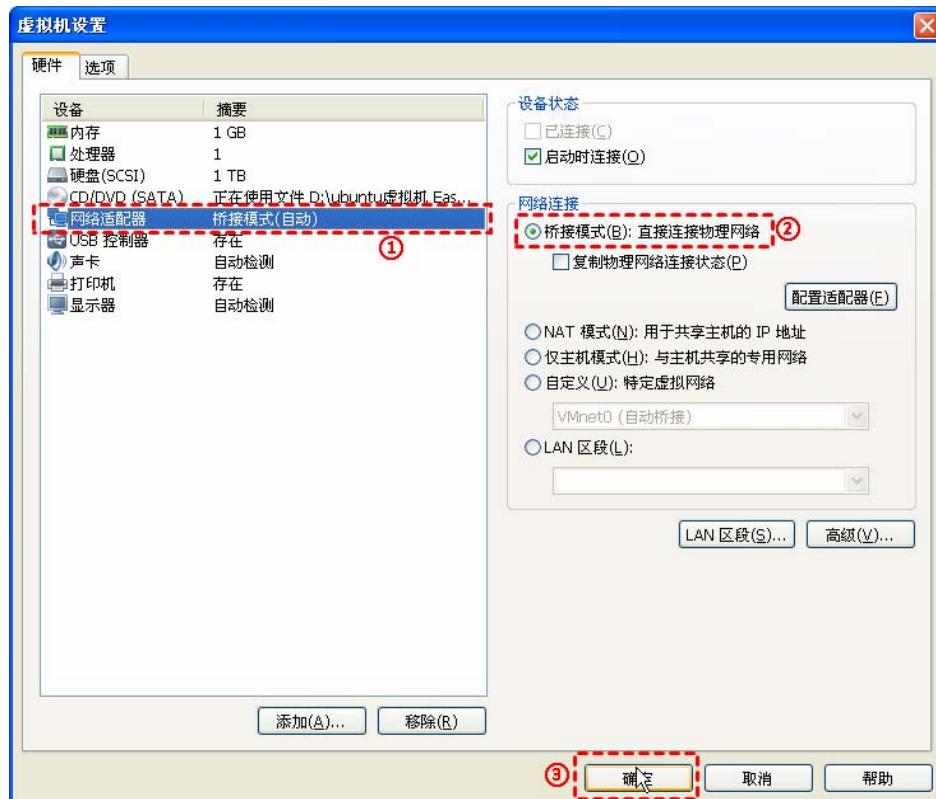


图4.18 配置“网络适配器”为“桥接模式”

当配置完虚拟机的其他硬件后，直接点击图4.18里面的“确定”按钮，然后再点击图4.19里的“完成”按钮即可。



图4.19 完成虚拟机配置



配置完虚拟机后，软件回到了VMware Player的主页界面，此时则可以看到刚刚创建的虚拟机已经位于主页的虚拟机列表了，如图4.20所示，此时选中刚刚创建的虚拟机后，再点击位于右侧的“播放虚拟机”则可以启动该虚拟机并进入Linux系统的正式安装流程。

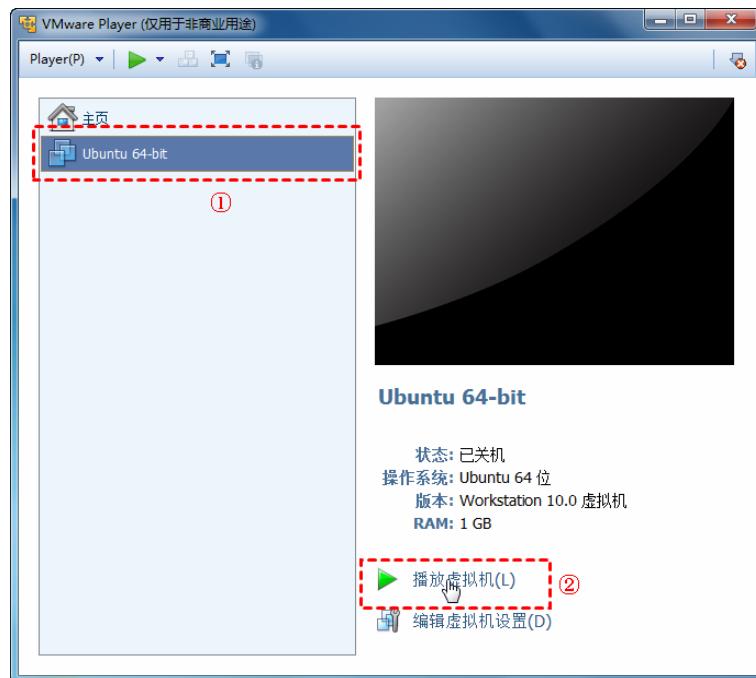


图4.20 启动虚拟机

虚拟机启动后，可能会弹出一些警告对话框（如：提示“网络未连接”或“SM总线未使能”等），除了严重影响虚拟机运行的问题（如提示主机未使能虚拟机指令，此时需要参考本节开头的介绍，到BIOS中使能虚拟机指令的支持）需要解决外，一般不需要特殊处理，虚拟机正常启动后如图4.21所示。

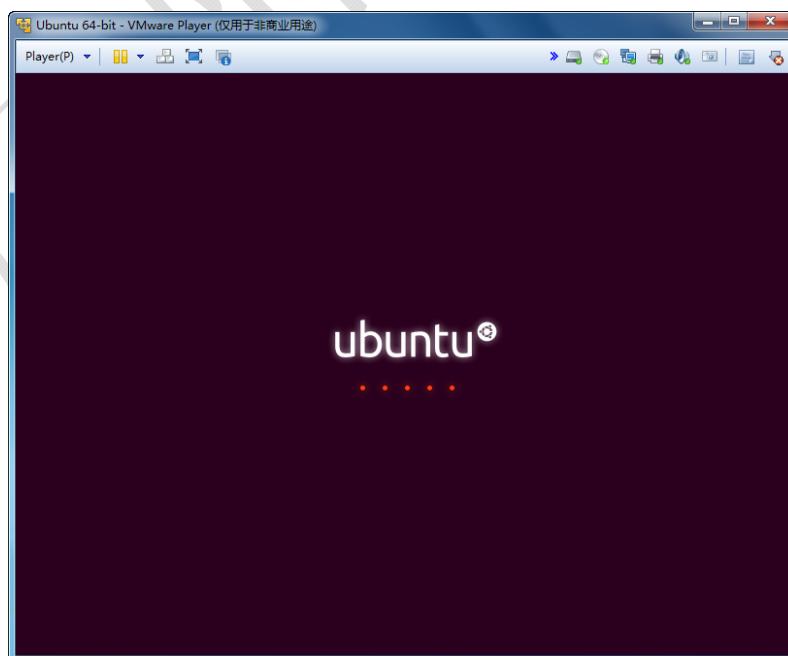


图4.21 Ubuntu 安装镜像正常启动



Ubuntu安装镜像正常启动后，按照图4.22~图4.29所示进行设置并安装Ubuntu系统，整个安装过程大概需要 20~30 分钟。



图4.22 选择系统语言



图4.23 点击“继续”按钮



图4.24 清除整个虚拟磁盘



图4.25 点击“现在安装”按钮



图4.26 选择系统时间所在时区

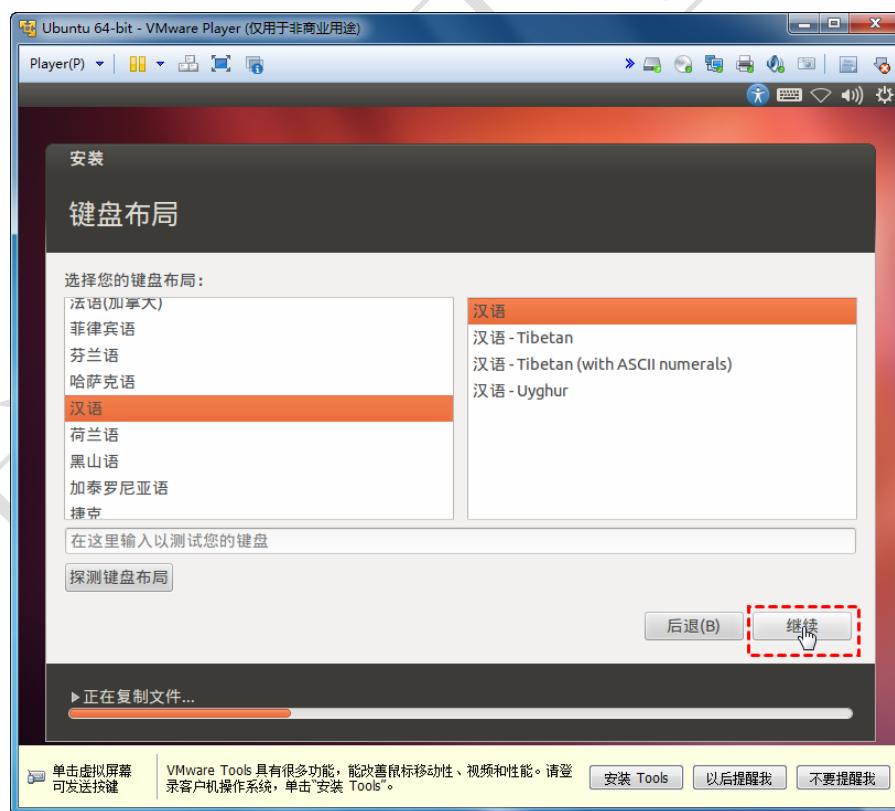


图4.27 确定键盘布局



图4.28 设置用户名及密码均为“vmuser”



图4.29 系统安装完成后点击“现在重启”按钮

系统安装完选择“现在重启”后，在重启过程中可以选择安装VMware tools（**用于支持虚拟机和主机之间的文件拖动交换**），如图4.30所示。

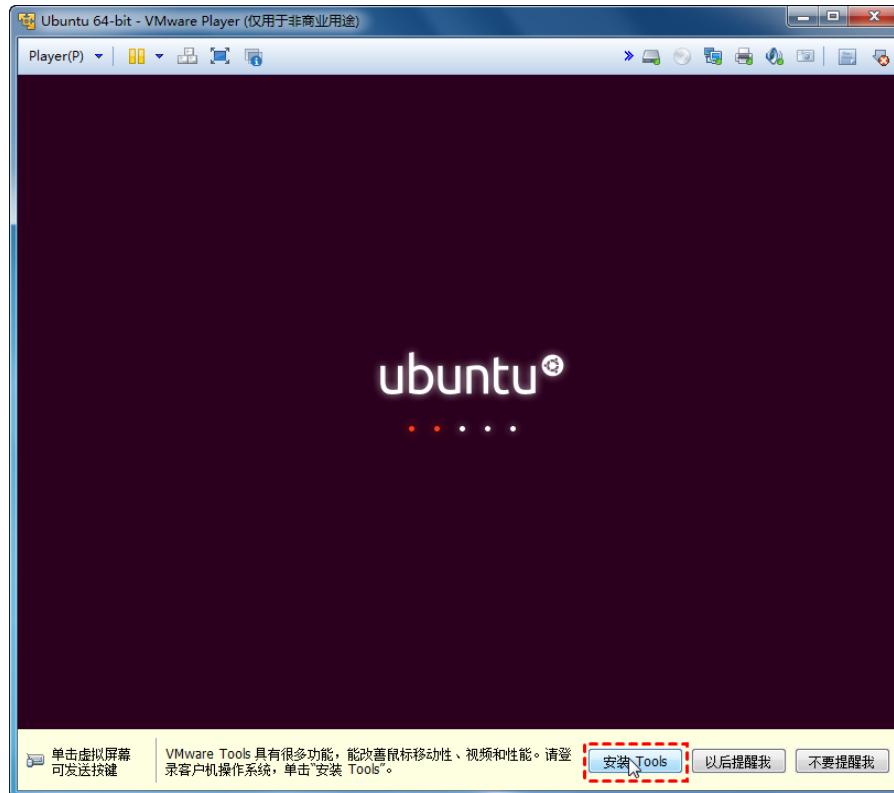


图4.30 选择安装 VMware Tools

若电脑的网络不通，则可能导致VMware Tools安装失败，此时可以选择选择“以后提醒我”或“不要提醒我”，跳过这个安装步骤，如图4.31所示。

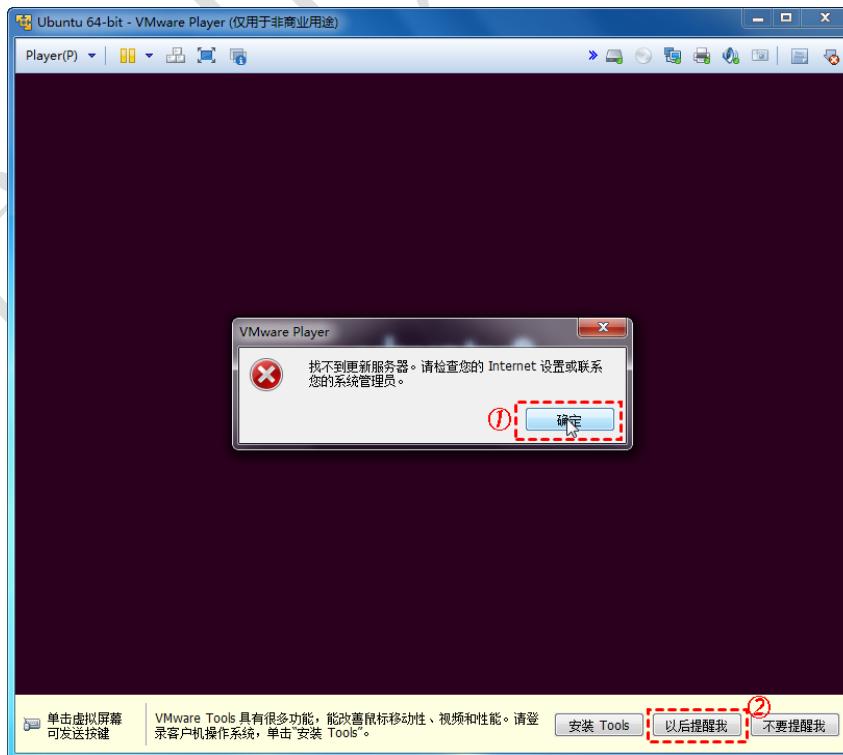


图4.31 VMware Tools 安装因没有网络连接而失败



虚拟机中的Linux系统成功启动后，将进入Linux桌面，如图4.32所示。至此，已成功完成在虚拟机中的Linux系统安装。

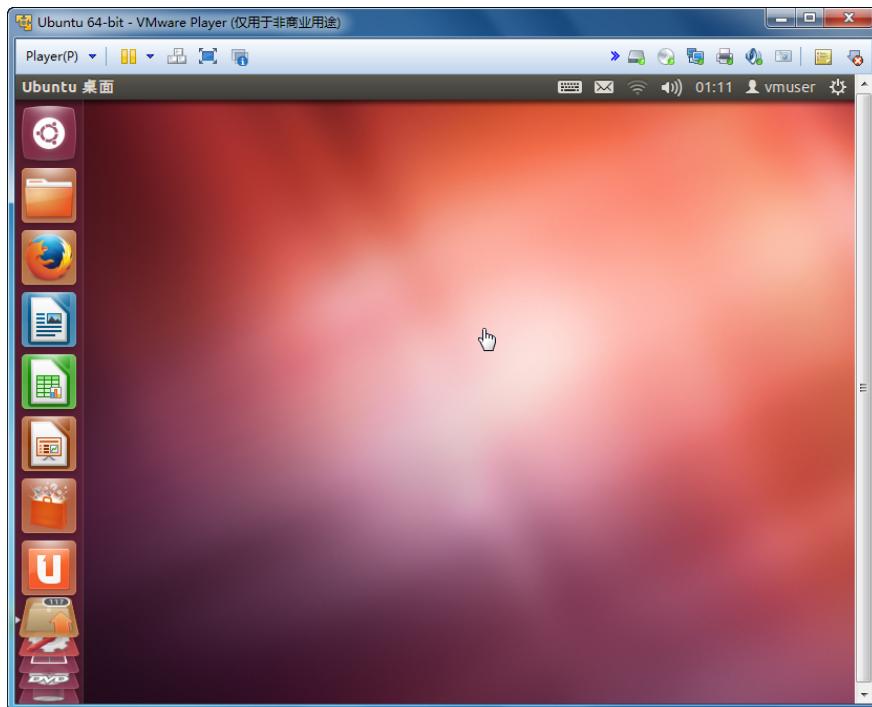


图4.32 Linux 启动后的桌面

### 4.3 Linux主机基础应用介绍

#### 1. Linux终端

这里所讨论的“终端”是指Linux下的虚拟终端，登录到Ubuntu桌面后，按“Ctrl+Alt+T”组合即可打开Linux下的虚拟终端，终端打开后其初始界面如图4.33所示，光标前的字符串“vmuser@Linux-host:~\$”是命令提示符，其中：

- vmuser 是当前登录的用户名；
- Linux-host 是计算机名；
- ~表示当前目录是当前用户主目录（即/home/vmuser 目录）；
- \$表示当前用户是普通用户。

对于本文所引用的终端命令，为了表述完整而保留了对应命令提示符，但用户实际操作时则无需输入对应的命令提示符。

**注意：命令行中出现的“#”号则是注释符号，该行中“#”号及其后面的字符串不会被终端解释和执行。**

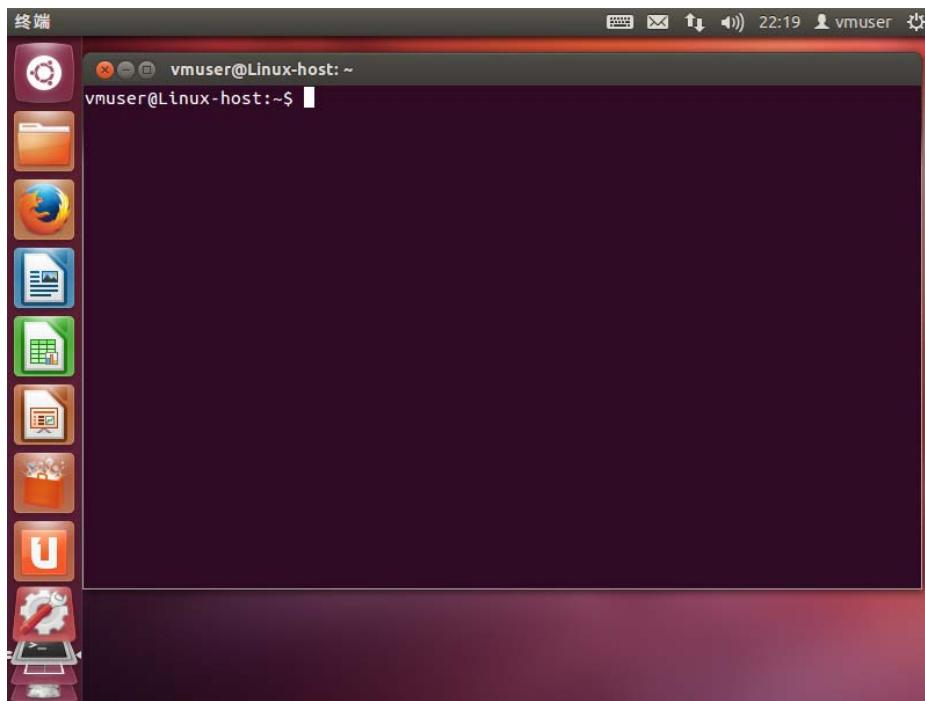


图4.33 按“Ctrl+Alt+T”组合键打开终端

Linux 中的很多工作都可以在终端中完成，许多情况下，使用终端比使用图形化的程序更便捷；不仅如此，所有的终端任务都可以写到脚本中，这样就可以自动执行。关闭已打开的终端可以直接点击终端窗口上的关闭按钮或按“Ctrl+Shift+Q”组合键，为了真正地驾驭 Linux 环境，用户需要熟练掌握如何在终端中工作，关于终端的常用命令，用户可以通过互联网查询和学习。

## 2. vi编辑器

vi 编辑器是所有 Unix 及 Linux 系统下标准的编辑器，也是 Linux 中最基本的文本编辑器。对于 Unix 及 Linux 系统的任何版本，vi 编辑器是完全相同的，因此，为了在 Linux 的世界里畅行无阻，用户也需要熟练掌握 vi 的应用。用户可以在其他任何介绍 vi 的地方进一步了解它，这里只简单地介绍一下它的用法和一小部分指令。

- vi 的基本概念

vi 应用时通常分为三种状态，分别是一般模式、插入模式及命令模式，各模式的功能区分如下：

- 一般模式：vi 处理文件时，一进入该文件，就是一般模式了，在一般模式下可以进行删除，复制，粘贴等操作，却无法进行编辑操作；
- 编辑模式：当在一般模式时按下“i、I、o、O、a、A、r、R”等字母之后就进入了编辑模式，通常在 Linux 中，按下上述字母时，左下方会出现“INSERT”或者“REPLACE”字样，只有在编辑模式才可以输入文字到文件中，要回到一般模式，按下“ESC”键即可；
- 命令行模式：在一般模式中，输入“:”或者“/”或者“?”英文字符，即可将光标移动到最下面一行，在该模式下，可以进行数据搜索，并且可以进行读取、存盘、大量删除字符等操作，离开 vi 也是通过该模式下的命令来实现。
- vi 的基本操作

在 Linux 终端中输入“vi 文件名称”后，就进入 vi 全屏幕编辑画面，指令形式如下：



```
vmuser@Linux-host:~$ vi filename
```

进入 vi 后，按字母“i”可以进入编辑模式，进入编辑模式后可以输入字符及修改文件，通过方向键则可以移动光标。

文件编辑完后需要先回到一般模式，然后才能执行其他操作，退出 vi 及保存文件则可以通过以下命令实现：

:w filename	将以指定的文件名 filename 保存文件
:wq	存盘并退出 vi
:wq!	强制存盘并退出 vi
:q!	放弃所作修改，不存盘强制退出 vi
:q	放弃修改并退出 vi

- 常用 vi 功能命令或快捷键

如没有特别说明，下面列举的常用命令或功能键均是在 vi 的一般模式或命令行模式下输入的，其中引号所引内容（不包含引号）为所需输入的命令或快捷键。

- “i”：切换进入插入模式，在光标当前位置开始输入；
- “I”：切换进入插入模式，在光标当前行首位置开始输入；
- “a”：切换进入插入模式，在光标当前位置的下一个位置开始输入；
- “A”：切换进入插入模式，在光标当前行尾位置开始输入；
- “o”：切换进入插入模式，在当前行之下新开一行；
- “O”：切换进入插入模式，在当前行之上新开一行；
- “u”：如果您误执行一个命令，可以马上按下“u”，回到上一个操作，按多次“u”可以执行多次恢复（undo）；
- “/关键字”：先按“/”键，再输入想寻找的字符，如果第一次找的关键字不是想要的，可以一直按“n”会往后继续寻找对应的关键字；
- “?关键字”：先按“?”键，再输入您想寻找的字符，如果第一次找的关键字不是想要的，可以一直按“n”会往前继续寻找对应的关键字。

注：vim 是从 vi 发展出来的一个文本编辑器，其完全兼容 vi 的功能命令及快捷键，并在代码补完、编译及错误跳转等方便编程的功能方面有良好的支持，在程序员中被广泛使用，本文演示的文本编辑在部分章节内也采用了 vim 文本编辑器。

#### 4.4 ssh服务器配置

在 Linux 主机安装 ssh 服务器是为了方便在 Windows 系统下使用 SSH Secure Shell Client 客户端软件与 Linux 主机系统进行文件共享和远程登录。

注意，若使用 SSH Secure Shell Client 客户端软件登录虚拟机中的 Linux 系统或共享文件必须配置虚拟机的以太网连接方式为 Bridged（桥接）模式，否则客户端将无法连接 ssh 服务器。

在 windows 系统下的 SSH Secure Shell Client 客户端软件在光盘资料“1.Windows 工具软件”目录下有提供，也可以在网上搜索下载得到，它的安装步骤在这里不做详细描述。

使用如下终端命令可以在 Linux 主机中安装 ssh 服务器：

```
vmuser@Linux-host:~$ sudo apt-get install openssh-server
```

ssh服务器安装成功后，终端显示如图4.34所示。



```
vmuser@Linux-host: ~
vmuser@Linux-host:~$ sudo apt-get install openssh-server
[sudo] password for vmuser:
正在读取软件包列表... 完成
正在分析软件包的依赖关系树
正在读取状态信息... 完成
将会安装下列额外的软件包:
  openssh-client
建议安装的软件包:
  libpam-ssh keychain monkeysphere openssh-blacklist openssh-blacklist-extra
  rsync molly-guard
下列软件包将被升级:
  openssh-client openssh-server
升级了 2 个软件包, 新安装了 0 个软件包, 要卸载 0 个软件包, 有 318 个软件包未被升级。
需要下载 1,281 kB 的软件包。
解压缩后会消耗掉 0 B 的额外空间。
您希望继续执行吗? [Y/n]y
获取: 1 http://cn.archive.ubuntu.com/ubuntu/ precise-updates/main openssh-server
  amd64 1:5.9p1-5ubuntu1.3 [338 kB]
获取: 2 http://cn.archive.ubuntu.com/ubuntu/ precise-updates/main openssh-client
  amd64 1:5.9p1-5ubuntu1.3 [943 kB]
下载 1,281 kB, 耗时 2 秒 (484 kB/s)
正在预设软件包 ...
(正在读取数据库 ... 系统当前共安装有 143512 个文件和目录。)
正预备替换 openssh-server 1:5.9p1-5ubuntu1.1 (使用 .../openssh-server_1%3a5.9p1-
5ubuntu1.3_amd64.deb) ...
正在解压缩将用于更替的包文件 openssh-server ...
正预备替换 openssh-client 1:5.9p1-5ubuntu1.1 (使用 .../openssh-client_1%3a5.9p1-
5ubuntu1.3_amd64.deb) ...
正在解压缩将用于更替的包文件 openssh-client ...
正在处理用于 man-db 的触发器...
正在处理用于 ureadahead 的触发器...
正在处理用于 ufw 的触发器...
正在设置 openssh-client (1:5.9p1-5ubuntu1.3) ...
正在设置 openssh-server (1:5.9p1-5ubuntu1.3) ...
ssh stop/waiting
ssh start/running, process 3497
vmuser@Linux-host:~$
```

图4.34 成功安装 ssh 服务器

## 4.5 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 Linux 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 Linux 主机共享目录下的程序，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. 安装NFS软件包

在“Ubuntu 64-bit 12.04 for EasyARM-iMX283.iso”基础上安装的 Linux 系统已经默认安装了 NFS 软件包，用户无需再执行安装，如果用户的 Linux 系统上未安装有 NFS 相关软件包，则可以在系统联网的情况下通过下面的终端命令进行安装：

```
vmuser@Linux-host ~$ sudo apt-get install nfs-kernel-server
```

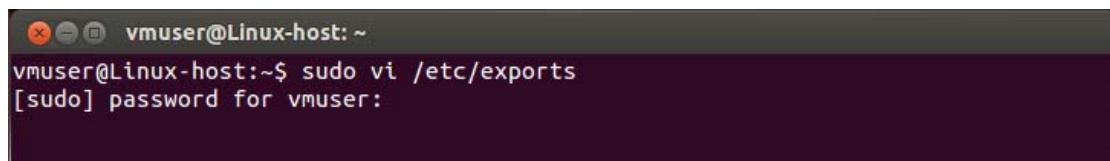


```
vmuser@Linux-host ~$ sudo apt-get install nfs-common
```

## 2. 添加EasyARM-iMX283 的NFS目录

安装完 NFS 服务器等相关软件后，需要指定用于共享的 NFS 目录，其方法为在 /etc/exports 文件里面设置对应的目录及相应的访问权限，每一行对应一个设置。下面介绍如何添加 EasyARM-iMX283 的 NFS 目录。

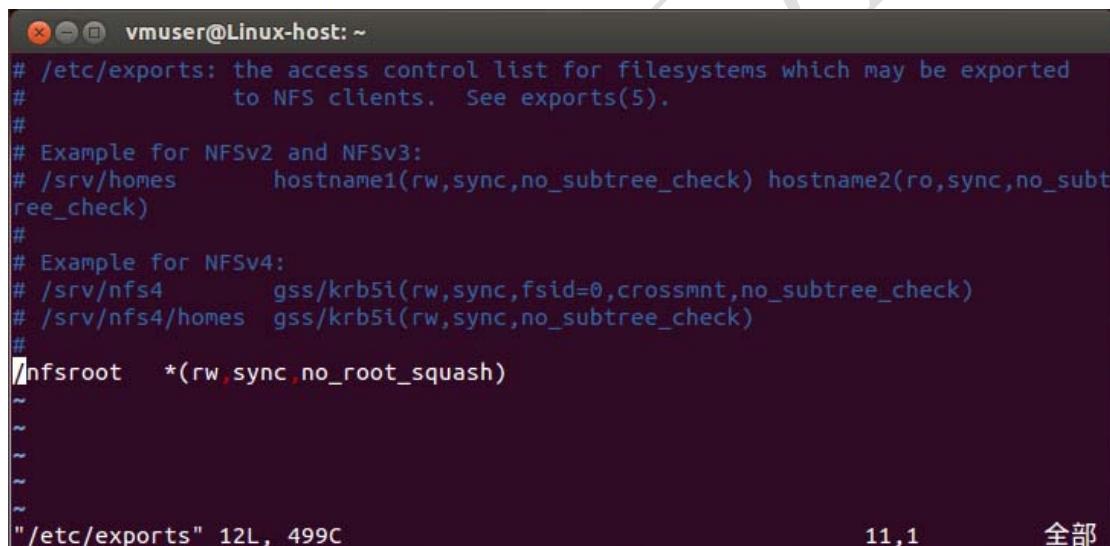
打开终端输入“sudo vi /etc/exports”指令（因为需要root权限，所以要用sudo命令），再按照提示输入“vmuser”用户的密码“vmuser”（需要注意，在Linux终端中输入密码时将不会回显输入的字符或密码替代字符），如图4.35所示，输入正确的密码后即可通过vi打开 /etc/exports文件。



```
vmuser@Linux-host:~$ sudo vi /etc/exports
[sudo] password for vmuser:
```

图4.35 通过 vi 命令打开/etc/exports 文件

/etc/exports文件打开后如图4.36所示。



```
# /etc/exports: the access control list for filesystems which may be exported
#                   to NFS clients. See exports(5).
#
# Example for NFSv2 and NFSv3:
# /srv/homes      hostname1(rw,sync,no_subtree_check) hostname2(ro,sync,no_subtree_check)
#
# Example for NFSv4:
# /srv/nfs4        gss/krb5i(rw,sync,fsid=0,crossmnt,no_subtree_check)
# /srv/nfs4/homes  gss/krb5i(rw,sync,no_subtree_check)
#
#nfsroot *(rw,sync,no_root_squash)

"/etc/exports" 12L, 499C          11,1          全部
```

图4.36 编辑“exports”文件

在此界面将键盘切换到大写模式，按下字母 G 键，光标将自动移动到文件的最后一行，再将键盘切换到小写模式，按下字母 o 键后 vi 将进入编辑模式，并自动将光标移动到文件末尾。此时可通过键盘输入以下字符串：

```
/home/vmuser/EasyARM-iMX283 *(rw,sync,no_root_squash)
```

其中“\*”表示允许任何任何网段IP的系统访问该NFS目录，字符串输完后按ESC键退出编辑模式，然后再输入vi命令“:wq”后按“回车”键，保存并退出/etc/exports文件的编辑，如图4.37所示。

```
vmuser@Linux-host: ~
# /etc/exports: the access control list for filesystems which may be exported
# to NFS clients. See exports(5).
#
# Example for NFSv2 and NFSv3:
# /srv/homes      hostname1(rw,sync,no_subtree_check) hostname2(ro,sync,no_subtree_check)
#
# Example for NFSv4:
# /srv/nfs4      gss/krb5i(rw,sync,fsid=0,crossmnt,no_subtree_check)
# /srv/nfs4/homes  gss/krb5i(rw,sync,no_subtree_check)
#
/nfsroot  *(rw,sync,no_root_squash)
/home/vmuser/EasyARM-iMX283  *(rw,sync,no_root_squash)

:wq
```

图4.37 保存并退出 vi 编辑

在/etc/exports 文件中设置了相应的 NFS 目录后，还需要到/home/vmuser/目录下创建对应的 EasyARM-iMX283 目录，否则在启动 NFS 服务的时候将会提示找不到该目录。

点击Ubuntu桌面左侧任务栏的“主文件夹”图标打开主文件夹目录（[对应目录 /home/vmuser/](#)），然后在主文件夹目录下打开右键菜单，选择“创建文件夹”并将其命名为“EasyARM-iMX283”即可，如图4.38所示。



图4.38 在主文件夹目录下创建“EasyARM-iMX283”目录

为了方便测试NFS是否挂载成功，还可以在“EasyARM-iMX283”目录下创建“Nfs Test”目录，如图4.39所示。



图4.39 创建测试目录

### 3. 启动 NFS 服务

在终端中执行如下命令，可以启动 NFS 服务：

```
vmuser@Linux-host ~$ sudo /etc/init.d/nfs-kernel-server start
```

执行如下命令则可以重新启动 NFS 服务：

```
vmuser@Linux-host ~$ sudo /etc/init.d/nfs-kernel-server restart
```

执行启动命令后，其操作结果如图4.40所示，表示NFS服务已正常启动。

```
vmuser@Linux-host:~$ sudo /etc/init.d/nfs-kernel-server start
[sudo] password for vmuser:
 * Exporting directories for NFS kernel daemon...
exportfs: /etc/exports [1]: Neither 'subtree_check' or 'no_subtree_check' specified for export "*:/nfsroot".
Assuming default behaviour ('no_subtree_check').
NOTE: this default has changed since nfs-utils version 1.0.x

exportfs: /etc/exports [2]: Neither 'subtree_check' or 'no_subtree_check' specified for export "*:/home/vmuser/EasyARM-iMX283".
Assuming default behaviour ('no_subtree_check').
NOTE: this default has changed since nfs-utils version 1.0.x

 * Starting NFS kernel daemon
[ OK ]
vmuser@Linux-host:~$ [ OK ]
```

图4.40 启动 NFS 服务

在 NFS 服务已经启动的情况下，如果修改了/etc/exports 文件，需要重启 NFS 服务，以刷新 NFS 的共享目录。

### 4. 测试NFS服务器

NFS 服务启动后，可以在 Linux 主机上进行自测或者通过 EasyARM-iMX283 进行测试，测试的基本方法为：将已经设定好的 NFS 共享目录 mount 到另外一个目录下（**既可以是 Linux 主机的目录也可以是目标板 Linux 系统的目录**），看能否成功。

#### 1) Linux 主机自测试

假定Linux主机IP为 192.168.12.123，NFS共享目录为“/home/vmuser/EasyARM-iMX283”



可使用如下命令进行测试，如图4.41所示：

```
vmuser@Linux-host~$ sudo mount -t nfs 192.168.12.123:/home/vmuser/EasyARM-iMX283 /mnt -o noblock
```

The screenshot shows a terminal window titled 'vmuser@Linux-host: ~'. The user runs the command 'sudo mount -t nfs 192.168.12.123:/home/vmuser/EasyARM-iMX283 /mnt -o noblock'. A password prompt '[sudo] password for vmuser:' appears, followed by a blank line for the password entry.

图4.41 挂在 NFS 目录测试

如果指令运行没有出错，则NFS挂载成功，在/mnt目录下应该可以看到 /home/vmuser/EasyARM-iMX283 目录下的内容，如图4.42所示。

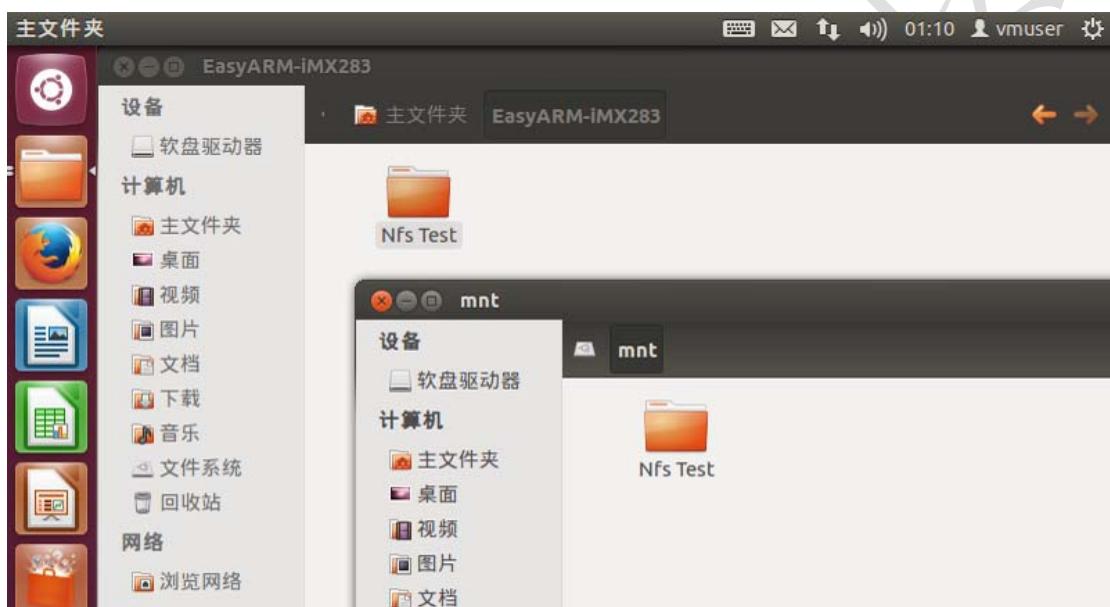


图4.42 NFS 挂载成功

## 2) 目标 Linux 系统挂载 NFS 测试

在 Linux 主机上测试通过后，则可以在目标 Linux 系统上进行测试，若用户在目标板上挂载成功 NFS，则用户以后可以轻松地使用此方法来验证所开发的 Linux 应用程序。在目标板上测试 NFS 的步骤如下：

首先，在目标板掉电的条件下，给目标板接上调试串口线、网线以及 USB 鼠标。

将 EasyARM-iMX283 学习板的 DUART 通过串口线与 PC 机的串口相连，在 PC 机中打开串口终端监听串口数据，通过 DUART 可以在串口终端上配置目标 Linux 系统的相关参数。

将目标板通过网线与 Linux 主机相连或接入局域网，当目标板与 Linux 主机直接相连时，若 Linux 主机网卡不支持以太网端口 MDI/MDIX 自适应特性，则需要采用交叉网线来连接。

USB 鼠标需要在系统启动前接入目标板，否则可能导致进入 QT 界面后无法使用鼠标（**Linux 系统支持 USB 鼠标的热插拔，但目前移植到目标板上的 QT 还不支持 USB 鼠标的热插拔**）。

然后给目标板上电，再设定目标板的 IP 地址并检查网络是否畅通。



在目标板 Linux 系统上挂载 NFS，要求目标板 Linux 系统的 IP 地址和 NFS 服务器(即 Linux 主机)的 IP 地址能互相 ping 通，并且没有防火墙禁止 NFS 连接，为简单起见，建议将两个 IP 放在同一个局域网网段中。

给目标板系统上电后，需留意串口终端的内容变化，当串口终端出现如图4.43所示的“EasyARM-iMX283 login:”时，以root用户名（密码与用户名相同）登录，登录后通过如下命令配置目标Linux系统的IP地址（掉电不保存）：

```
root@EasyARM-iMX283 ~# ifconfig eth0 192.168.12.124
```

IP 地址配置成功后可以通过如下命令检查目标系统与 Linux 主机的网络连接是否畅通：

```
root@EasyARM-iMX283 ~# ping 192.168.12.123
```

注意：网络畅通后才能进行下一步操作，若网络不通，则需要检查相关网络设置。网络畅通时，可以在串口终端中看到 Linux 主机对 ping 命令的响应，按“Ctrl+C”组合键可以退出 ping 命令测试。

接着，在目标 Linux 系统上挂载 NFS。

目标板与 Linux 主机系统网络畅通后，在串口终端中输入如下命令可将主机的 NFS 共享目录“EasyARM-iMX283”挂载到目标系统的“/mnt”目录下：

```
root@EasyARM-iMX283 ~# mount -t nfs 192.168.12.123:/home/vmuser/EasyARM-iMX283 /mnt -o noblock
```

命令执行成功后，串口终端窗口显示如图4.43所示，其中红线标注部分是通过终端输入的命令。

注意，执行mount命令时的当前路径（如图4.43所示的执行命令的当前路径为主目录“~”）不能与挂载的目标路径相同，否则可能导致挂载失败。

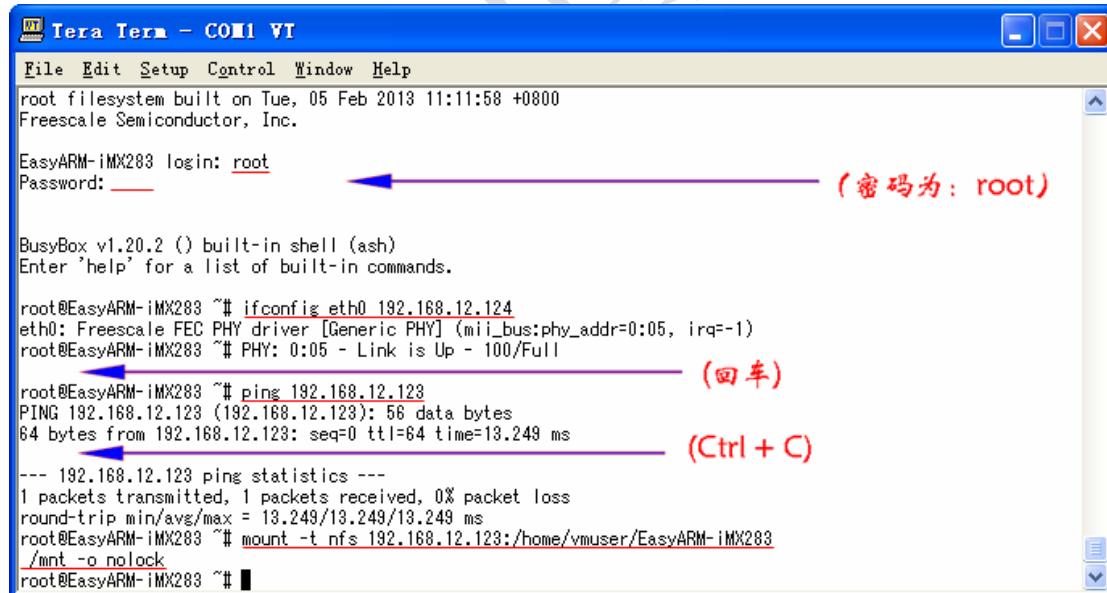


图4.43 通过 DUART 配置目标板 Linux 系统

NFS 在目标系统上挂载成功后，可以在串口终端中输入如下命令查看/mnt 目录下的文件信息：

```
root@EasyARM-iMX283 ~# ls /mnt
```

此时，亦可通过USB鼠标操作，浏览目标板系统上的/mnt目录，NFS挂载成功后/mnt目录下的内容与Linux主机对应的NFS共享目录里的内容应是相同的，如图4.44所示。



图4.44 NFS 挂载成功

## 4.6 TFTP服务器

### 1. 安装tftp软件

在“Ubuntu 64-bit 12.04 for EasyARM-iMX283.iso”基础上安装的 Linux 系统已经默认安装了 tftp 服务器需要安装 tftpd-hpa 软件，用户无需再安装 tftpd-hpa，但用户还需再安装 tftp-hpa 软件，如果用户的 Linux 系统上未安装 tftpd-hpa 或 tftp-hpa 软件，则可以在系统联网的情况下通过下面的终端命令进行安装：

```
vmuser@Linux-host ~$ sudo apt-get install tftpd-hpa tftp-hpa
```

软件安装成功后，终端显示如图4.45所示。



```
vmuser@Linux-host: ~
vmuser@Linux-host:~$ sudo apt-get install tftpd-hpa tftp-hpa
[sudo] password for vmuser:
正在读取软件包列表... 完成
正在分析软件包的依赖关系树
正在读取状态信息... 完成
tftpd-hpa 已经是最新的版本了。
下列【新】软件包将被安装：
  tftp-hpa
升级了 0 个软件包，新安装了 1 个软件包，要卸载 0 个软件包，有 318 个软件包未被升
级。
需要下载 19.8 kB 的软件包。
解压缩后会消耗掉 77.8 kB 的额外空间。
您希望继续执行吗？[Y/n]y
0% [执行中]
获取：1 http://cn.archive.ubuntu.com/ubuntu/ precise/main tftp-hpa amd64 5.2-1ub
untu1 [19.8 kB]
下载 19.8 kB, 耗时 0秒 (26.1 kB/s)
Selecting previously unselected package tftp-hpa.
(正在读取数据库 ... 系统当前共安装有 143512 个文件和目录。)
正在解压缩 tftp-hpa (从 .../tftp-hpa_5.2-1ubuntu1_amd64.deb) ...
正在处理用于 man-db 的触发器...
正在设置 tftp-hpa (5.2-1ubuntu1) ...
vmuser@Linux-host:~$
```

图4.45 安装 tftpd-hpa 及 tftp-hpa 软件

## 2. 配置tftp服务器

tftp 软件安装后，默认是关闭 tftp 服务的，需要更改 tftp 配置文件/etc/default/tftpd-hpa，可通过终端输入如下命令进行修改：

```
vmuser@Linux-host ~$ sudo vi /etc/default/tftpd-hpa
```

具体如图4.46所示。

```
vmuser@Linux-host: ~
vmuser@Linux-host:~$ sudo vi /etc/default/tftpd-hpa
[sudo] password for vmuser: ■
```

图4.46 通过 vi 打开 “/etc/default/tftpd-hpa” 文件

用vi编辑器打开tftpd-hpa文件后按字母“i”进入编辑模式，按如下参数修改后按Esc键退出修改模式，然后通过“:wq”命令退出并保存修改，如图4.47所示。

```
TFTP_USERNAME="tftp"
TFTP_DIRECTORY="/tftpboot"          #实际应用时可以改成用户指定的目录
TFTP_ADDRESS="0.0.0.0:69"
TFTP_OPTIONS="-l -c -s"
```



```
vmuser@Linux-host: ~
# /etc/default/tftpd-hpa

TFTP_USERNAME="tftp"
TFTP_DIRECTORY="/tftpboot"
TFTP_ADDRESS="0.0.0.0:69"
TFTP_OPTIONS="-l -c -s"

~
~
~
~

:wq
```

图4.47 退出并保存“tftp-hpa”文件修改

其中 TFTP\_DIRECTORY 指定 tftp 服务器的路径为 “/tftpboot” 目录，为了使用方便，可设置最宽松的访问权限。当然，如果用户的 Linux 系统下尚未创建该目录的话，则需要通过 mkdir 命令创建该目录。目录创建及权限设置命令如下：

```
vmuser@Linux-host ~$ sudo mkdir /tftpboot
vmuser@Linux-host ~$ sudo chmod -R 777 /tftpboot
vmuser@Linux-host ~$ sudo chown -R nobody /tftpboot
```

终端执行完命令后如图4.48所示。

```
vmuser@Linux-host: ~
vmuser@Linux-host:~$ sudo mkdir /tftpboot
[sudo] password for vmuser:
vmuser@Linux-host:~$ sudo chmod -R 777 /tftpboot
vmuser@Linux-host:~$ sudo chown -R nobody /tftpboot
vmuser@Linux-host:~$
```

图4.48 终端执行完“/tftpboot”目录权限设置

### 3. 启动tftp服务

tftp 服务器安装配置完成后，需要先重启 Linux 主机系统，然后才能启动 tftp 服务，启动 tftp 服务的终端命令如下：

```
vmuser@Linux-host ~$ sudo service tftpd-hpa start
```

终端执行完命令后如图4.49所示。

```
vmuser@Linux-host: ~
vmuser@Linux-host:~$ sudo service tftpd-hpa start
[sudo] password for vmuser:
tftpd-hpa start/running, process 2779
vmuser@Linux-host:~$
```

图4.49 启动 tftp 服务

### 4. 测试tftp服务器

在tftp服务器目录/tftpboot下创建一个测试文件tftpTestFile，并通过文本编辑器修改其文

件内容后保存，如图4.50~图4.52所示。

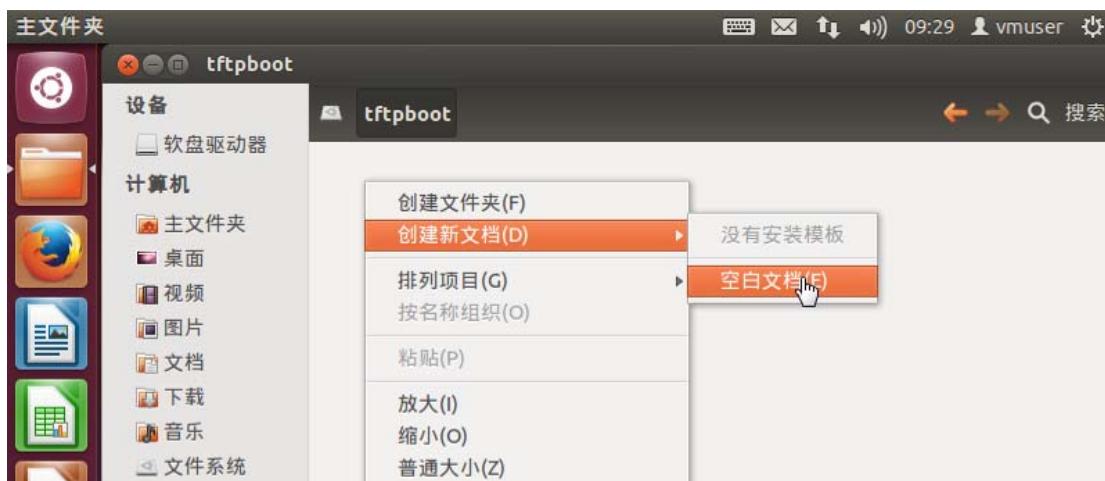


图4.50 在“/tftpboot”目录下创建测试文档 tftpTestFile



图4.51 用文本编辑器打开测试文档

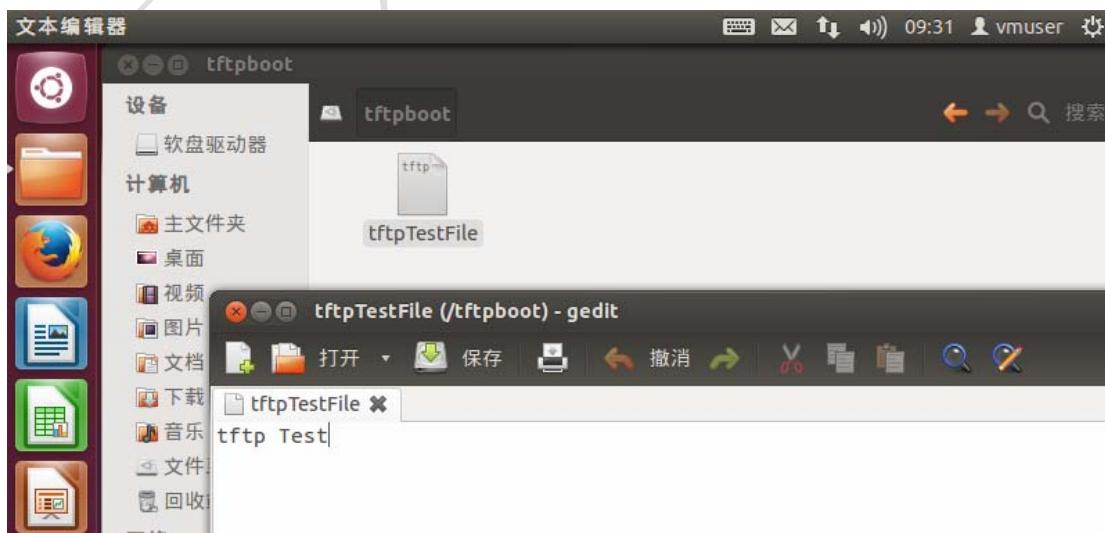


图4.52 修改测试文档内容并保存



测试文档准备好了之后，打开终端，输入以下测试命令（**192.168.12.123** 为当前 Linux 主机的 IP 地址）：

```
vmuser@Linux-host ~$ tftp 192.168.12.123  
tftp> get tftpTestFile  
tftp> q
```

若tftp服务器架设成功，终端运行命令后如图4.53所示，位于tftp服务器目录下的tftpTestFile文件将被下载到终端执行命令时的当前目录下，如图4.53所示。

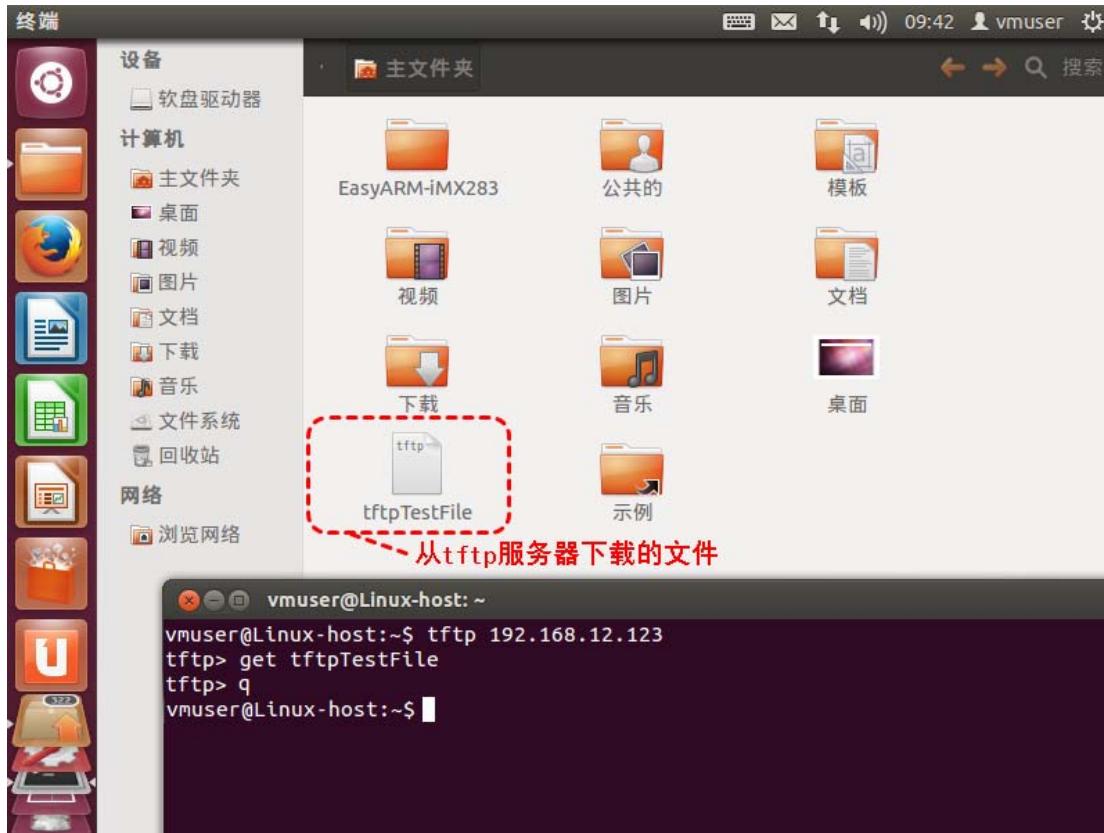


图4.53 测试 tftp 服务器

至此，tftp 服务器已经配置并测试成功，若用户操作结果与上述现象不同，则需要检查相关操作步骤是否安装文档操作。

**注意：Linux 下的文件名是严格区分大小写的，若文件名拼写错误，也有可能导致无法得到正确的测试结果。**

## 4.7 构建交叉开发环境

### 1. 工具链介绍

用户可以自行制作交叉编译器，但是比较繁琐，并且可能会因为配置等原因或者测试不充分带来一些隐患，而且编译器的隐患会给后续的开发带来很多障碍。为了避免这些情况的出现，建议使用经过测试的已经编译好的交叉工具链。

光盘资料中已经包含编译好的交叉工具链，在光盘的“3.Linux\2.工具软件\2.Linux 工具软件”目录下，以压缩包形式提供，具体的文件名为“gcc-4.4.4-glibc-2.11.1-multilib-1.0\_EasyARM-iMX283.tar.bz2”。光盘资料可以通过 U 盘的方式拷贝到 Linux 主机中，也可使用 SSH Secure Shell Client 通过 SSH 的方式把光盘资料拷贝到 Linux 主机。



## 2. 安装工具链

在“Ubuntu 64-bit 12.04 for EasyARM-iMX283.iso”基础上安装的 Linux 系统已经默认安装了开发 EasyARM-iMX283 所需的交叉编译工具链，用户无需再执行安装，如果用户的 Linux 系统上未安装这个工具链，则可以按照以下步骤进行安装：

### 3) 安装 32 位的兼容库和 libncurses5-dev 库

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 32 兼容库需要从 ubuntu 的源库中下载，所以需要在 Linux 主机系统联网的条件下，通过终端使用如下命令安装：

```
vmuser@Linux-host ~$sudo apt-get install ia32-libs
```

若 Linux 主机系统没有安装 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
```

在终端中使用如下命令则可以安装 libncurses5-dev 库。

```
vmuser@Linux-host ~$sudo apt-get install libncurses5-dev
```

如果没有安装此库，在使用 make menuconfig 时出现如下所示的错误：

```
*** 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
```

### 4) 安装交叉编译工具链

将光盘资料中的“gcc-4.4.4-glibc-2.11.1-multilib-1.0\_EasyARM-iMX283.tar.bz2”文件通过 U 盘的方式拷贝到 Linux 主机的“/tmp”目录下，然后执行如下命令进行解压安装交叉编译工具链：

```
vmuser@Linux-host ~$ cd /tmp
vmuser@Linux-host ~$ sudo tar -jxvf gcc-4.4.4-glibc-2.11.1-multilib-1.0_EasyARM-iMX283.tar.bz2 -C /opt/
vmuser@Linux-host /tmp$ # 输入 vmuser 用户的密码 “vmuser”
```

执行完解压命令后，交叉编译工具链将被安装到“/opt/gcc-4.4.4-glibc-2.11.1-multilib-1.0”目录下。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，还需将该路径添加到 PATH 环境变量中，其方法为：修改“/etc/profile”文件，具体操作方法如下：

在终端中输入如下指令

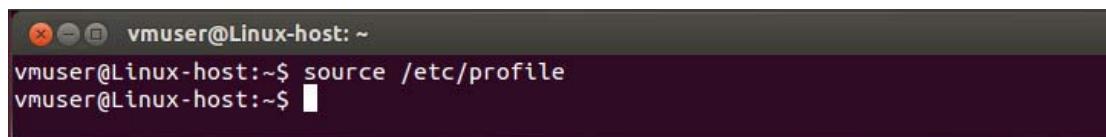
```
vmuser@Linux-host ~$ sudo vi /etc/profile # 若提示输入密码，则输入“vmuser”
```

用 vi 编辑器打开“/etc/profile”文件后，在文件末尾增加如下一行内容：

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

文件修改并保存后，再在终端中输入如下指令，更新环境变量，使设置生效，如图4.54所示。

```
vmuser@Linux-host ~$ source /etc/profile
```



```
vmuser@Linux-host: ~
vmuser@Linux-host:~$ source /etc/profile
vmuser@Linux-host:~$
```

图4.54 更新环境变量

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，如图4.55所示。



```
vmuser@Linux-host: ~
vmuser@Linux-host:~$ arm-fsl-linux-gnueabi-
arm-fsl-linux-gnueabi-addr2line      arm-fsl-linux-gnueabi-gprof
arm-fsl-linux-gnueabi-ar            arm-fsl-linux-gnueabi-ld
arm-fsl-linux-gnueabi-as            arm-fsl-linux-gnueabi-ldd
arm-fsl-linux-gnueabi-c++          arm-fsl-linux-gnueabi-nm
arm-fsl-linux-gnueabi-cc            arm-fsl-linux-gnueabi-objcopy
arm-fsl-linux-gnueabi-c++filt       arm-fsl-linux-gnueabi-objdump
arm-fsl-linux-gnueabi-cpp          arm-fsl-linux-gnueabi-populate
arm-fsl-linux-gnueabi-ct-ng.config   arm-fsl-linux-gnueabi-ranlib
arm-fsl-linux-gnueabi-g++          arm-fsl-linux-gnueabi-readelf
arm-fsl-linux-gnueabi-gcc          arm-fsl-linux-gnueabi-run
arm-fsl-linux-gnueabi-gcc-4.4.4     arm-fsl-linux-gnueabi-size
arm-fsl-linux-gnueabi-gccbug        arm-fsl-linux-gnueabi-strings
arm-fsl-linux-gnueabi-gcov         arm-fsl-linux-gnueabi-strip
arm-fsl-linux-gnueabi-gdb          vmuser@Linux-host:~$ arm-fsl-linux-gnueabi-
```

图4.55 测试交叉编译器是否安装正确

### 3. 测试工具链

在“~/EasyAMR-iMX283”目录下创建一个hello文件夹，并在该文件夹下创建hello.c文件（**创建方法为：右键该目录下空白处，在弹出的右键菜单中选择“创建新文档”，再在子菜单下选择“空白文档”，然后将创建的空白文档重命名为hello.c**），然后右击hello.c文件，选择“使用文本编辑器打开”菜单打开hello.c文件，然后输入如程序清单4.1所示内容。

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;
}
```

输入完程序代码后保存并关闭 hello.c 文件，然后按“Ctrl+Alt+T”启动终端，输入以下命令对 hello.c 进行编译并查看编译后生成文件的属性：

```
vmuser@Linux-host ~$ cd /home/vmuser/EasyARM-iMX283/hello      #浏览到程序文件所在目录
vmuser@Linux-host ~/EasyARM-iMX283/hello$ arm-fsl-linux-gnueabi-gcc hello.c -o hello #编译 hello.c 文件
vmuser@Linux-host ~/EasyARM-iMX283/hello$ file hello           #查看编译生成的 hello 文件属性
```



hello.c文件编译后将输出hello文件，终端执行命令及输出文件如图4.56所示。

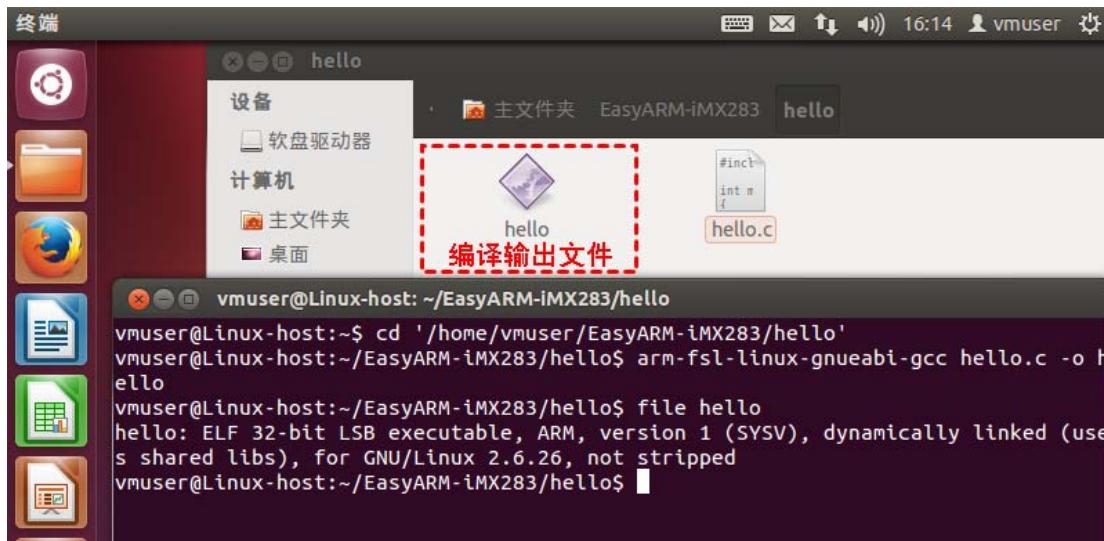


图4.56 编译“hello.c”文件

通过终端命令运行的结果可知，hello 文件是 ARM 格式的可执行文件，说明 arm-fsl-linux-gnueabi- 工具链已经可以正常使用了。

下面可以尝试通过前面学习到的 NFS 挂载方法，将 hello 文件放到 EasyARM-iMX283 上面去运行，其基本操作步骤如下：

- 5) 给目标板接上调试串口线及网线，并打开串口终端监听串口数据，然后再给目标系统上电。
- 6) 目标系统（**Linux 系统**）上电后，通过串口终端登录目标系统（**用户名及密码均为“root”**），并设置目标系统的 IP 为 192.168.12.124（**建议与 Linux 主机位于同一网段，并能互相 ping 通，否则可能导致 NFS 挂载失败**），并测试目标系统与 Linux 主机的网络是否畅通。
- 7) 目标系统与 Linux 主机网络畅通后，通过发送如下命令可将主机的 NFS 共享目录“EasyARM-iMX283”挂载到目标系统的/mnt 目录下：

```
root@EasyARM-iMX283 ~# mount -t nfs 192.168.12.123:/home/vmuser/EasyARM-iMX283 /mnt -o nolock
```

- 8) NFS 挂载成功后，通过“cd”命令浏览到 hello 文件所在目录，通过“./hello”命令运行 hello 文件，具体命令如下：

```
root@EasyARM-iMX283 ~# cd /mnt/hello
```

```
root@EasyARM-iMX283 /mnt/hello# ./hello
```

hello程序运行后，将通过串口终端打印五行字符，测试结果如图4.57所示。



BusyBox v1.20.2 () built-in shell (ash)  
Enter 'help' for a list of built-in commands.  
root@EasyARM-iMX283 ~# ifconfig eth0 192.168.12.124 设置目标板IP地址  
eth0: Freescale FEC PHY driver [Generic PHY] (mini\_bus:phy\_addr=0:05, irq=-1)  
root@EasyARM-iMX283 ~# PHY: 0:05 - Link is Up - 100/Full  
root@EasyARM-iMX283 ~# ping 192.168.12.123 检查NFS服务器是否可以ping通  
PING 192.168.12.123 (192.168.12.123): 56 data bytes  
64 bytes from 192.168.12.123: seq=0 ttl=64 time=13.249 ms  
--- 192.168.12.123 ping statistics ---  
1 packets transmitted, 1 packets received, 0% packet loss  
round-trip min/avg/max = 13.249/13.249/13.249 ms  
root@EasyARM-iMX283 ~# mount -t nfs 192.168.12.123:/home/vmuser/EasyARM-iMX283 /mnt -o nolock  
root@EasyARM-iMX283 ~# cd /mnt/hello  
root@EasyARM-iMX283 /mnt/hello# ./hello Hello程序执行结果  
Hello 0  
Hello 1  
Hello 2  
Hello 3  
Hello 4  
root@EasyARM-iMX283 /mnt/hello#

图4.57 运行测试程序

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，具体命令如下：

```
root@EasyARM-iMX283 /mnt/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。如程序清单4.2所示是针对这个hello工程编写的一个简单Makefile文件内容。

程序清单4.2 应用程序 Makefile 范例

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

用户可以采用与创建hello.c文件相同的方式，将程序清单4.2所示内容保存为Makefile文件，并且放置在hello.c文件所在目录下，如图4.58所示（已删除之前编译生成的hello文件）。



图4.58 创建 makefile 文件

有了合适的Makefile文件，只需在终端输入make命令即可编译程序，终端命令执行的结果及编译生成的文件如图4.59所示。

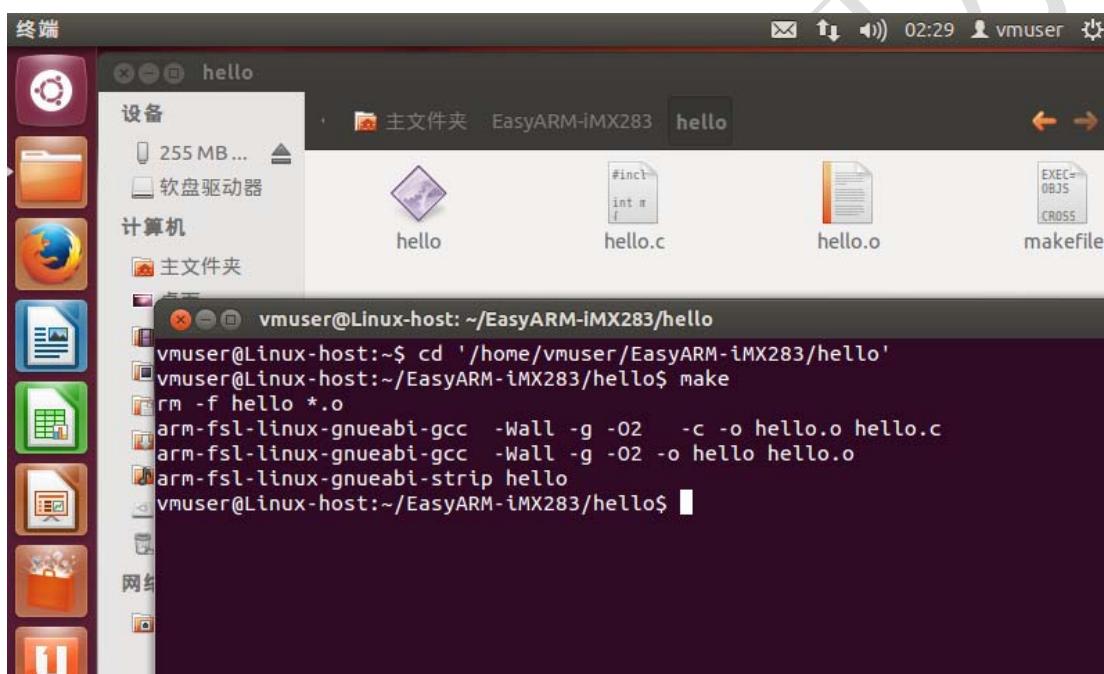


图4.59 利用 make 命令编译



## 5. 功能部件编程

本章主要介绍 EasyARM-iMX283 的 ADC、GPIO、UART、I2C、SPI、PWM 编程。对于本章的学习，首先需要设置好交叉编译器编译器，并能交叉编译普通的应用程序，而对于部分驱动源码（如 LRADC），需要读者提前编译好内核，并在 Makefile 文件中指定内核路径，才可以继续编译驱动源码。

所有例程源码放在“3.Linux\4.开发示例\2、功能部件”目录下。



图5.1 所有功能部件例程

默认编程规则：

涉及到设备操作的，需包含以下头文件：

```
#include<fcntl.h>
#include<sys/ioctl.h>
```

使用了 printf 或 sleep 函数的，需包含以下头文件：

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

注：下文部分代码将不再赘述头文件的使用问题。

### 5.1 GPIO应用编程

EasyARM-iMX283 开发平台的 Linux 系统实现了通用 GPIO 的驱动，用户通过系统命令即能控制 GPIO 的输出和读取其输入值。EasyARM-iMX283 底板引出 P2.4、P2.5、P2.6、P2.7 四个 GPIO 功能引脚，使用这些 GPIO 前，需要先将这些 GPIO 通过命令导出。

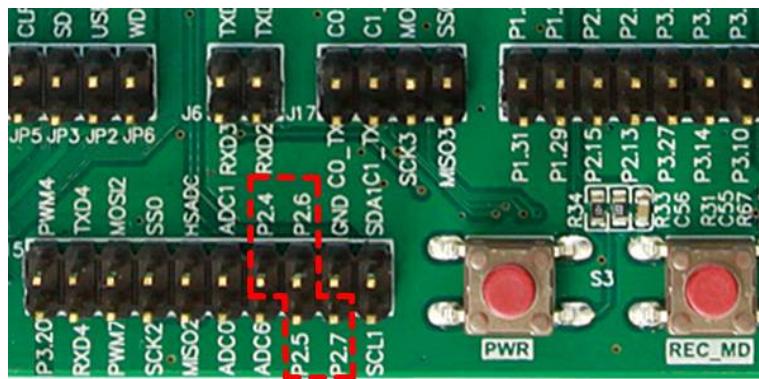


图5.2 开发板引出的 283 可用 GPIO

注意：J8 的一排 GPIO 口仅仅 i.MX287 芯片可用，i.MX283 不可用，参考光盘资料：1. 数据手册\EasyARM-iMX283 硬件设计指导手册 V1.02.pdf。

### 5.1.1 导出GPIO

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。iMX283 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：

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

BANK 为 GPIO 引脚所在的 BANK，N 为引脚所在的 BANK 的序号。

以 PP2.4 为例，其序号为  $2*32+4=68$ 。因此导出 P2.4 的 GPIO 功能需要进行如下操作：

```
root@EasyARM-iMX283 /# cd /sys/class/gpio/  
root@EasyARM-iMX283 /sys/class/gpio# ls  
export      gpiochip128  gpiochip64  unexport  
gpiochip0  gpiochip32  gpiochip96  
root@EasyARM-iMX283 /sys/class/gpio# echo 68 >export  
root@EasyARM-iMX283 /sys/class/gpio# ls  
export      gpiochip0  gpiochip32  gpiochip96  
gpio68      gpiochip128  gpiochip64  unexport root  
root@EasyARM-iMX283 /sys/class/gpio# cd gpio68/  
root@EasyARM-iMX283 /sys/devices/virtual/gpio/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。

### 5.1.2 方向设置

GPIO 导出后默认为输入功能。direction 具有方向设置功能：向 direction 文件写入“in”字符串，表示设置为输入功能；向 direction 文件写入“out”字符串，表示设置为输出功能。在读 direction 文件时，也是通过读到的这些字符串表示当前 GPIO 的方向设置。方向查看和设置通过以下命令进行设置：



```
root@EasyARM-iMX283 /sys/devices/virtual/gpio/gpio68# cat direction      #查看方向
in
root@EasyARM-iMX283 /sys/devices/virtual/gpio/gpio68# echo out >direction    #方向输入
root@EasyARM-iMX283 /sys/devices/virtual/gpio/gpio68# cat direction      #设置为输出
out
```

### 5.1.3 输入读取

当 GPIO 被设为输入时，value 文件记录 GPIO 引脚的输入电平状态：1 表示输入的是高电平；0 表示输入的是低电平。通过查看 value 文件可以读取 GPIO 的电平：

```
root@EasyARM-iMX283 /sys/devices/virtual/gpio/gpio68# cat value
1                                         #输入高电平
root@EasyARM-iMX283 /sys/devices/virtual/gpio/gpio68# cat value
0                                         #输入低电平
```

### 5.1.4 输出控制

当 GPIO 被设为输出时，通过向 value 文件写入 0 或 1（0 表示输出低电平；1 表示输出高电平）可以设置输出电平的状态：

```
root@EasyARM-iMX283 /sys/devices/virtual/gpio/gpio68# echo 0 >value      #输出低电平
root@EasyARM-iMX283 /sys/devices/virtual/gpio/gpio68# echo 1 >value      #输出高电平
```

在C语言代码中，可使用标准的read和write调用来控制GPIO，如程序清单5.1所示。

程序清单5.1 C 语言操作 GPIO 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>

#define DEV_PATH      "/sys/devices/virtual/gpio/gpio68/value"

int main(void)
{
    int fd = 0;
    char value;

    fd = open(DEV_PATH, O_RDWR);
    if (fd < 0) {
        perror(DEV_PATH);
    }

    lseek(fd, 0, SEEK_SET);
```



```
    read(fd, &value, 1);
    printf("get value:%c \n", value);

    close(fd);
    return 0;
}
```

## 5.2 ADC接口

EasyARM-iMX283 在 J15 排针提供了 ADC0、ADC1、ADC6、HSADC 四路 ADC 电压模拟量采集接口，HSADC 为 2M 采样率的高速 ADC，可用于摄像头数据采集，Freescale 尚未提供可用驱动，故本节不做讲解。

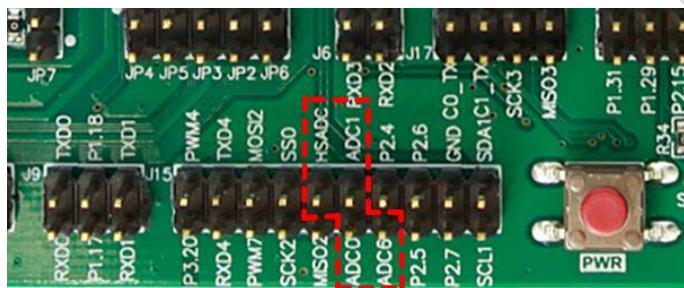


图5.3 ADC 接口示意图

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。

### 5.2.1 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@EasyARM-iMX283 ~# insmod /root/lradc.ko
adc module init!
```

注意：V1.02 及之前版本的光盘中的 Linux 文件系统内不带/root/lradc.ko 文件，如果用户发现目标系统中没有此文件，请下载最新的光盘资料，并安装光盘中的文件系统或按新光盘资料重新恢复 Linux 系统。

### 5.2.2 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（**开启硬件除 4 电路**）；



- 20: 读取 ADC0 电压值 (开启硬件除 2 电路);
- 21: 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);

### 5.2.3 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 20、21、26, 计算公式为:  $V = 2 \times 1.85 \times (Val / 4096)$

### 5.2.4 操作示例

如程序清单5.2所示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5.2 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--) {
        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\n", 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\n", 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\n", 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\n", val);
        printf("\n");
    }
    close(fd);
}

```

### 5.3 串口编程

EasyARM-iMX283 提供 3 路 TTL 电平的应用串口接口：UART0、UART1、UART4（不包括 RS232 接口的 Debug UART），如图 5.4 所示。其中 UART0 和 UART1 有 DR 可控制传输方向，因此 UART0 和 UART1 可以通过外接 RS485 模块而成 485 接口。

**注意：**在 EasyARM-iMX283 中，UART2 和 UART3 功能引脚被复用为其他功能，在此的 TXD2、RXD2、TXD3 及 RXD3 引脚仅在使用 M287 核心板时可用。

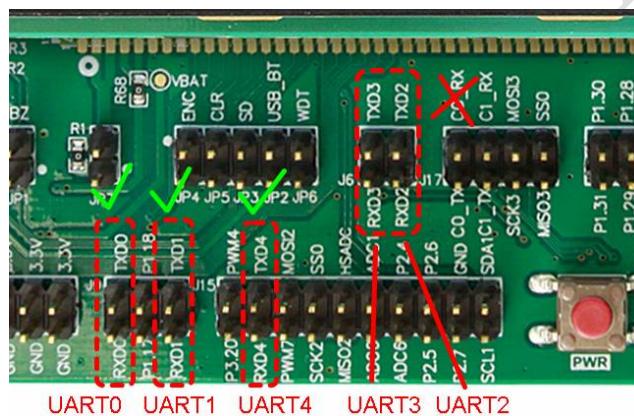


图 5.4 EasyARM-iMX283 可用的应用串口

硬件接口和设备文件节点的对应关系如表 5.1 所示（需要注意：EasyARM-iMX283 对应的 /dev/ttySP2 和 /dev/ttySP3 设备文件节点是不可实现串口通信的）。

当 UART0 和 UART1 作为 RS485 接口使用时，串口驱动将在收发数据时通过 P1.17 和 P1.18 自动控制 DR 线电平输出，无需用户的干预。

表 5.1 硬件接口与设备文件对应关系

串口标号	设备文件节点
UART0	/dev/ttySP0
UART1	/dev/ttySP1
UART4	/dev/ttySP4

**注：**若用户需要独立使用 P1.17 及 P1.18 的 GPIO 功能，则需修改 “linux-2.6.35.3/arch/arm/mach-mx28/mx28evk\_pins.c” 文件，将 mx28evk\_fixed\_pins 结构体数组中关于 RS485\_DIR 的定义注释掉，如程序清单 5.3 所示。

#### 程序清单 5.3 禁用 RS485 方向控制引脚

```

static struct pin_desc mx28evk_fixed_pins[] = {
    .....// 不列举数组其他元素，请以实际文件为准
{
    //.name      = "RS485_DIR",
    //.id        = PINID_LCD_D17,
    //.fun       = PIN_GPIO,
    //.strength   = PAD_8MA,
    //.voltage   = PAD_3_3V,
}

```



```
//.drive      = 1,  
},  
{  
    //.name     = "RS485_DIR",  
    //.id       = PINID_LCD_D18,  
    //.fun      = PIN_GPIO,  
    //.strength = PAD_8MA,  
    //.voltage  = PAD_3_3V,  
    //.drive    = 1,  
},  
.....// 不列举数组其他元素, 请以实际文件为准  
}
```

### 5.3.1 访问串口设备

#### 1. 打开串口

在EasyARM-iMX283 的Linux系统中，串口设备文件名是“/dev/ttySP%d”（其中%d=0、1、4）。打开串口设备文件的操作如程序清单5.4所示。

程序清单5.4 打开串口设备

```
int fd;  
fd = open("/dev/ttySP0", O_RDWR | O_NOCTTY | O_NDELAY);  
if (fd < 0) {  
    printf("open SPI device error\n");  
}
```

相对于普通设备操作，当调用 open()函数打开串口设备文件时，除了需要用到 O\_RDWR 选项标志外，通常还需要使用 O\_NOCTTY 和 O\_NDELAY 选项标志。

O\_NOCTTY 选项是告诉 Linux “本程序不作为串口的‘控制终端’”。如果不使用该选项，会有一些输入字符影响进程运行（如一些产生中断信号的键盘输入字符等）。

O\_NDELAY 表示让程序忽略 DCD 信号线。如果不使用该选项，进程可能在 DCD 信号线被拉低时进入休眠。

#### 2. 发送数据

使用标准的write系统调用向串口写入数据，如程序清单5.5所示。

程序清单5.5 向串口设备写入数据

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

#### 3. 读取数据

使用标准的read系统调用读入串口数据，如程序清单5.6所示。

程序清单5.6 读入串口数据



```
int len;  
unsigned char pBuf[0xff];  
len = read(fd, pBuf, 0xff);
```

当串口设备工作在原始数据模式时，read 函数返回串口缓冲区里的数据，如果缓冲区没有数据可读，read 函数会阻塞直到新的数据到来。为了使 read 调用能立即返回，可以采用下面操作：

```
fctl(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（非阻塞方式）。若要恢复正常状态，可以再次调用 fcntl() 函数并不带 FNDELAY 选项：

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

这些操作通常在调用完 open() 函数（带 O\_NDELAY 选项）打开串口设备后执行。

#### 4. 关闭串口

```
close(fd);
```

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

### 5.3.2 配置串口属性

本节主要讲解串口的波特率、数据位、校验方式等属性的获取和设置。

#### 1. 属性描述

串口属于终端设备，其接口属性用 termios 结构描述，如程序清单 5.7 所示。

程序清单 5.7 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 */  
};
```

粗略而言，控制标志影响到 RS-232 串行线（如：忽略调制解调器的状态线、每个字符需要一个或两个停止位等），输入标志由终端设备驱动程序用来控制字符的输入（如：剥除输入字节的第 8 位，允许输入奇偶校验等），输出控制则控制驱动程序输出（如：执行输出处理、将换行符映射为 CR/LF 等），本地标志影响驱动程序和用户之间的接口（如：本地回显的开和关等），c\_cc 数组则包含了所有可以更改的特殊字符。

#### 1) 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 5.2 列出了 c\_cflag 可用的部分选项。

表 5.2 c\_cflag 部分可用选项

标志	说 明	标志	说 明
CBAUD	波特率位屏蔽	CSIZE	数据位屏蔽
B0	0 位/秒（挂起）	CS5	5 位数据位
B110	100 位/秒	CS6	6 位数据位



续上表

标 志	说 明	标 志	说 明
B134	134 位/秒	CS7	7 位数据位
B1200	1200 位/秒	CS8	8 位数据位
B2400	2400 位/秒	CSTOPB	2 位停止位, 否则为 1 位
B4800	4800 位/秒	CREAD	启动接收
B9600	9600 位/秒	PARENBT	进行奇偶校验
B19200	19200 位/秒	PARODD	奇校验, 否则为偶校验
B57600	57600 位/秒	HUPCL	最后关闭时断开
B115200	115200 位/秒	CLOCAL	忽略调制调解器状态行
B460800	460800 位/秒	—	—

c\_cflag 成员的 CREAD 和 CLOCAL 选项通常是要启用的, 这两个选项使驱动程序启动接收字符装置, 同时忽略串口信号线的状态。

## 2) 输入标志

c\_iflag 成员负责控制串口输入数据的处理, 表 5.3 所示是 c\_iflag 的部分可用标志。

表 5.3 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 PARENBT (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);
```

注意: IGNPAR 选项在一些场合的应用带有一定的危险性, 它指示串口驱动程序忽略奇偶校验错误, 也就是说, IGNPAR 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);
```

## 3) 输出标志

c\_oflag 成员管理输出过滤, 如表 5.4 所示是 c\_oflag 成员的部分选项标志。



表5.4 c\_oflag 标志

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

- 启用输出处理

启用输出处理需要在 c\_oflag 成员中启用 OPOST 选项，其操作方法如下：

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

- 使用原始输出

使用原始输出，就是禁用输出处理，使数据能不经过处理、过滤地完整地输出到串口接口。当 OPOST 被禁止，c\_oflag 其它选项也被忽略，其操作方法如下：

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

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，如表5.5所示是c\_lflag的部分可用标志。

表5.5 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清
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，其操作方法如下：

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

5) 控制字符组

`c_cc` 数组的长度是 NCCS，一般介于 15-20 之间。`c_cc` 数组的每个成员的下标都用一个宏表示，表 5.6 列出了 `c_cc` 的部分下标标志名及其对应说明。

表 5.6 `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	需读取的最小字节数
VTIME	与“VMIN”配合使用，是指限定的传输或等待的最长时间

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 是指一次 `read` 调用期望返回的最小字节数。TIME 与 MIN 组合使用，其具体含义分以下四种情形：

- 当  $MIN > 0$ ,  $TIME > 0$  时

TIME 为接收到第一个字节后允许的数据传输或等待的最长分秒数（1 分秒 = 0.1 秒）。定时器在收到第一个字节后启动，在计时器超时之前，若已收到 MIN 个字节，则 `read` 返回 MIN 个字节，否则，在计时器超时后返回实际接收到的字节。

**注意：因为只有在接收到第一个字节时才启动，所以至少可以返回 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，则如同在 `read` 后数据立即被接到一样。**

- 当  $MIN > 0$ ,  $TIME = 0$  时

MIN 个字节完整接收后，`read` 才返回，这可能会造成 `read` 无限期地阻塞。

- 当  $MIN = 0$ ,  $TIME > 0$  时

TIME 为允许等待的最大时间，计时器在调用 `read` 时立即启动，在串口接到 1 字节数据或者计时器超时后即返回，如果是计时器超时，则返回 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。

## 2. 属性设置

使用函数tcgetattr和tcsetattr可以获取和设置串口termios结构属性，如程序清单5.8所示。

程序清单5.8 设置和获取 termios 结构属性

```
#include <termios.h> /* 使用终端接口函数需要使用此头文件*/  
int tcgetattr(int fd, struct termios *termpptr);  
int tcsetattr(int fd, int opt, const struct termios *termpptr);
```

其中：fd 为串口设备文件描述符，termpptr 参数在 tcgetattr 函数中是用于存放串口设置的 termios 结构体，opt 是整形变量，使用方法如下：

- 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 (Flush)。

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 可以指定在什么时候新的串口属性才起作用。

上述两函数执行时，若成功则返回 0，若出错则返回-1。

掌握了如何获取和设置串口的属性结构后，下面将介绍串口主要属性的修改，即修改 termios 结构体的成员。

termios 结构体的各个成员的各个选项中除需要用屏蔽标志的选项外（如波特率选项、数据位选项等），都是按位表示的，对这些选项的设置或清除可以直接用“^”或“&”逻辑运算来完成。

需要用屏蔽标志的选项则的设置则需要先用“&”运算清除原设置，再用“^”运算设置新选项。例如，为了设置字符长度，需先用字符长度屏蔽标志 CSIZE 将表示字符长度的位清 0，然后再将对应位设置为 CS5、CS6、CS7 或 CS8。

### 6) 设置波特率

串口的输入和输出波特率可分别用cfsetispeed()和cfsetospeed()函数来设置，如程序清单 5.9 所示。

程序清单5.9 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5.10所示。

程序清单5.10 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;  
}
```

#### 7) 设置数据位

设置数据位不需要专用的函数，只需要在设置数据位之前用数据位屏蔽标志（CSIZE）把对应数据位清零，然后再设置新的数据位即可，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 */
```

#### 8) 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;
```

### 5.3.3 操作示例

如程序清单 5.11 所示是原始模式下串口操作示例。

程序清单 5.11 串口操作示例

```
#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/ttySP0"

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;
}
```

注：测试上述代码时，需要把/dev/mSP0 接口的 RXD 和 TXD 用杜邦线短接起来。当程序执行时，程序在串口会把发出去的数据读回来，并打印出来。

## 5.4 I<sup>2</sup>C接口

EasyARM-iMX283 底板以排针形式引出了 I<sup>2</sup>C1 接口，在核心板上 I<sup>2</sup>C0 接口与 DS2460 加密芯片连接（目前提供的 Linux 系统尚未支持 DS2460 加密功能）。Linux 系统实现了 I<sup>2</sup>C0 和 I<sup>2</sup>C1 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。

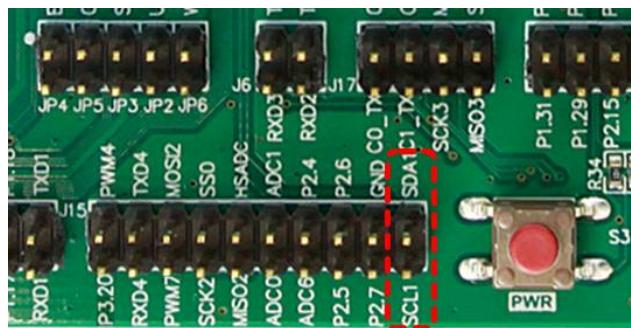


图5.5 I<sup>2</sup>C0 接口位置图

### 5.4.1 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5.12所示。

程序清单5.12 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");
}
```

### 5.4.2 ioctl调用

当使用 I<sup>2</sup>C 驱动操作 I<sup>2</sup>C 从机器件时，需要先把 I<sup>2</sup>C 从机器件地址和地址长度设置到 I<sup>2</sup>C 驱动设备文件接口。

#### 1. 设置从机地址

使用 I2C\_SLAVE 命令调用 ioctl() 函数，可以设置 I<sup>2</sup>C 从机地址，命令参数就是 I<sup>2</sup>C 从机地址右移一位后的值。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(GiFd, I2C_SLAVE, 0xA0 >> 1);
```

注意：地址需要右移一位，是因为地址的 Bit0 是读写控制位，在驱动中会将从机地址命令参数左移一位，并补上读写控制位。

#### 2. 设置地址长度

使用 I2C\_TENBIT 命令调用 ioctl() 函数，可以设置 I<sup>2</sup>C 从机地址的长度，当命令参数为



1 时，表示 I<sup>2</sup>C 从机地址长度为 10 位，当命令参数为 0 时，表示 I<sup>2</sup>C 从机地址长度为 8 位。

当需要把 I<sup>2</sup>C 从机地址长度设置为 8 位时，可参考如下代码：

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

注：I<sup>2</sup>C的ioctl调用使用到的命令在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5.13所示。

程序清单5.13 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_RETRYES      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
```

#### 5.4.3 write调用

当设置好 I<sup>2</sup>C 从机的地址后，就可以调用 write()函数向 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()函数调用后，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，然后发送器件的地址，地址发送完并接收到应答后再将 buf 缓冲区中的数据发出，数据发送完后 I<sup>2</sup>C 主机发出总线结束信号。

#### 5.4.4 read调用

当设置好 I<sup>2</sup>C 从机的地址后，就可以调用 read()函数从 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()函数调用后，I<sup>2</sup>C 主机向 I<sup>2</sup>C 从机器件发出总线始起信号，然后发送器件地址，地址发送完并接收到从机应答后，接着向 I<sup>2</sup>C 从机发出读数据时钟，驱动程序将接收到的数据存入 buff 中，数据读取完后 I<sup>2</sup>C 主机发出总线结束信号。



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。

#### 5.4.5 close 调用

当 I<sup>2</sup>C 驱动操作完成后，请调用 close() 函数关闭之前打开的 I<sup>2</sup>C 驱动设备文件：

```
close(fd);
```

#### 5.4.6 应用程序读写DS2460 例程

EasyARM-iMX283 的核心板上有 DS2460 加密芯片，与 i.MX283 处理的 I<sup>2</sup>C0 总线相连，如图 5.6 所示。

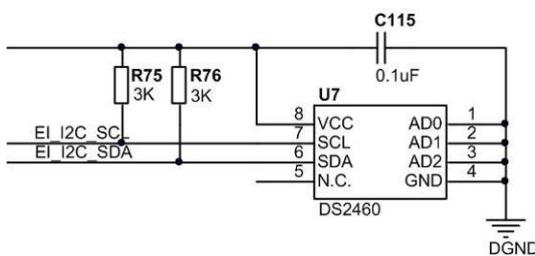


图 5.6 DS2460 连接原理图

DS2460 的 SCL 和 SDA 引脚分别和 i.MX283 处理的 I<sup>2</sup>C0\_SCL 和 I<sup>2</sup>C0\_SDA 引脚相连，所以使用 “/dev/i2c-0” 文件设备节点可以控制该芯片。如图 5.6 所示 DS2460 芯片的从机地址是 0x80。DS2460 的内部地址从 0x00 ~ 0x3F 是内部 SRAM 地址空间。

如程序清单 5.14 所示的测试程序从 DS2460 的内部地址空间 0x00 ~ 0x0F 分别写入 1、2、3……16，然后再在这片地址空间读取出数据并打印出来显示。

程序清单 5.14 DS2460 测试程序

```
#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 "iic.h"  
  
#define I2C_ADDR 0x80  
#define DATA_LEN 17  
int main(void)  
{  
    unsigned int uiRet;
```



```
int i;

unsigned char tx_buf[DATA_LEN];
unsigned char rx_buf[DATA_LEN];
unsigned char addr[2] ;

addr[0] = 0x00;
GiFd = open("/dev/i2c-0", O_RDWR);
if (GiFd == -1)
    perror("open serial 0\n");
uiRet = ioctl(GiFd, I2C_SLAVE, I2C_ADDR >> 1);
if (uiRet < 0) {
    printf("setenv address faile ret: %x \n", uiRet);
    return -1;
}
tx_buf[0] = addr[0];
for (i = 1; i < DATA_LEN; i++) {
    tx_buf[i] = i;
}
write(GiFd, tx_buf, DATA_LEN);
write(GiFd, addr, 1);
read(GiFd, rx_buf, DATA_LEN - 1);
printf("read from ds2460's eeprom:");
for(i = 0; i < DATA_LEN - 1; i++) {
    printf(" %x", rx_buf[i]);
}
printf("\n");
return 0;
}
```

在光盘资料中有该程序代码的实现文件，用户可以直接编译测试。在 EasyARM-iMX283 的 /root 目录下有该测试程序，用户可以直接进行测试。请登录 EasyARM-iMX283 的 Linux 系统，进入命令行终端，输入如下指令可以测试 I2C 的读写：

```
root@EasyARM-iMX283 ~# ./i2c_ds2460_test
```

如图5.7所示表示测试成功，打印出了 16 进制的数字 1~16。

The screenshot shows a terminal window titled "Tera Term - COM1 VI". The window displays a series of kernel boot messages followed by a root password prompt and a BusyBox shell.

```
UBIFS: mounted UBI device 0, volume 0, name "rootfs"
UBIFS: file system size: 218652672 bytes (213528 KiB, 208 MiB, 1722 LEBs)
UBIFS: journal size: 10919936 bytes (10664 KiB, 10 MiB, 86 LEBs)
UBIFS: media format: w4/r0 (latest is w4/r0)
UBIFS: default compressor: lzo
UBIFS: reserved for root: 4952683 bytes (4836 KiB)
VFS: Mounted root (ubifs filesystem) on device 0:15.
Freeing init memory: 160K

arm-none-linux-gnueabi-gcc (Freescale MAD -- Linaro 2011.07 -- Built at 2011/08/
10 09:20) 4.6.2 20110630 (prerelease)
root filesystem built on Tue, 05 Feb 2013 11:11:58 +0800
Freescale Semiconductor, Inc.

EasyARM-iMX283 login: root
Password:

BusyBox v1.20.2 () built-in shell (ash)
Enter 'help' for a list of built-in commands.

root@EasyARM-iMX283 ~# ./i2c_ds2460_test
read from ds2460's eeprom: 1 2 3 4 5 6 7 8 9 a b c d e f 10
root@EasyARM-iMX283 ~#
```

图5.7 测试 DS2460

## 5.5 PWM接口

EasyARM-iMX283 底板引出了 PWM\_3、PWM\_4、PWM\_7 三路 PWM 输出引脚，其中 PWM\_3 用于液晶屏背光控制，PWM\_4 和 PWM\_7 在 JP15 排针上引出。

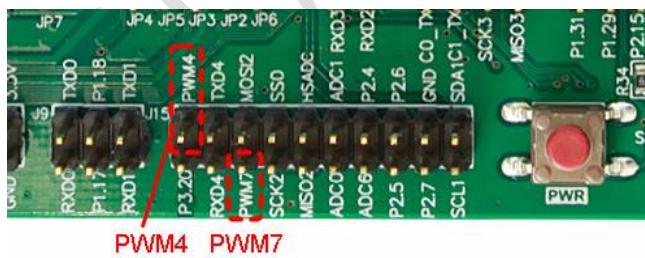


图5.8 板上3路PWM输出位置图

### 5.5.1 PWM占空比设置与输出

PWM\_4 和 PWM\_7 两个通道 PWM 以 backlight 类型的驱动的形式存在，输出的频率为 937.5Hz，可以通过系统命令进行查看/设置 PWM 的周期和占空比，也可以通过应用程序进行操作。

### 5.5.2 系统命令操作PWM示例

PWM\_3、PWM\_4 和 PWM\_7 对应的驱动设备文件分别为位于/sys/class/backlight/目录下的 mxs-bl、easy283-pwm.4 及 easy283-pwm.7 文件。通过系统命令进行查看及设置液晶背光 PWM\_3 占空比示例指令如下：

```
root@EasyARM-iMX283 ~# cd /sys/class/backlight/  
root@EasyARM-iMX283 /sys/class/backlight# ls
```



```
easy283-pwm.4  easy283-pwm.7  mxs-bl
root@EasyARM-iMX283 /sys/class/backlight# cd mxs-bl
root@EasyARM-iMX283 /sys/devices/platform/mxs-bl.0/backlight/mxs-bl# cat max_brightness
100
root@EasyARM-iMX283 /sys/devices/platform/mxs-bl.0/backlight/mxs-bl# cat brightness
100
root@EasyARM-iMX283 /sys/devices/platform/mxs-bl.0/backlight/mxs-bl# echo 10 >brightness
root@EasyARM-iMX283 /sys/devices/platform/mxs-bl.0/backlight/mxs-bl# cat brightness
10
root@EasyARM-iMX283 /sys/devices/platform/mxs-bl.0/backlight/mxs-bl#
```

操作相关指令解释如下：

- cat max\_brightness，用于查看可设置的占空比的最大值；
- cat brightness，查看当前占空比的值；
- echo 10 >brightness，设置 PWM 的占空比；实际输出的 PWM 波形的占空比为 brightness / max\_brightness，本例中为 10/100。

指令执行完后可以观察到液晶背光亮度被调低了（系统上电后默认的背光 PWM 占空比为 100%）。

## 5.6 SPI接口

EasyARM-iMX283 以排针引脚的方式引出了 SPI2 接口，其片选信号引脚为 SS0，SPI3 在 i.MX287 上有，i.MX283 中没有。

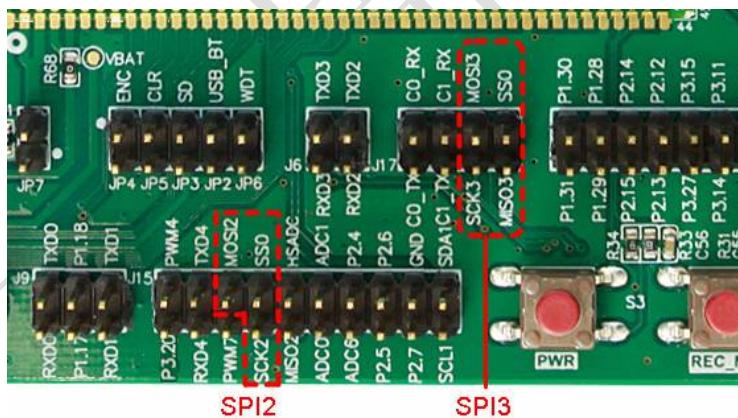


图5.9 开发板上的 SPI 接口位置图

SPI2 驱动设备文件名：/dev/spidev1.0。

需要注意的是 iMX283 处理器的 SPI 控制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内核目录下 linux-2.6.35.3/include/linux/spi/spidev.h 文件中。

注：对 SPI 的编程需要加入#include <spidev.h>。

### 5.6.1 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5.15所示。



## 程序清单5.15 打开 SPI 设备文件

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

## 5.6.2 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5.7所示。

表5.7 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5.8所示。

表5.8 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5.9所示。



表5.9 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5.10所示。

表5.10 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5.11所示。

表5.11 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 spo_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5.16所示。

程序清单5.16 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 iMX283 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。

### 5.6.3 示例代码

程序清单5.17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5.10所示:

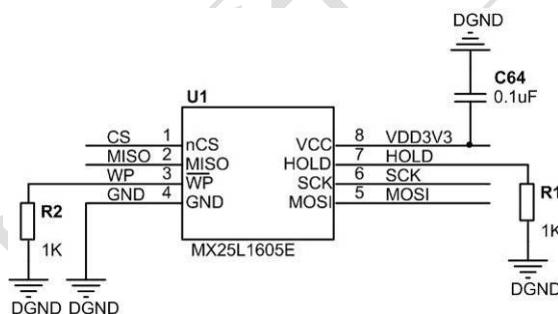


图5.10 SPI Flash 信号连接图

测试时需要将上图中四个悬空的引脚CS、MISO、MOSI、SCK分别连接开发板的SS0、MISO2、MOSI2、SCK2 四个管脚, VDD3V3 及DGND分别连接至开发套件的 3.3V及GND 上。访问MX25L1635E的测试代码如程序清单5.17所示。

#### 程序清单5.17 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]))\n\nstatic void pabort(const char *s)\n{\n    perror(s);\n    abort();\n}\n\nstatic const char *device = "/dev/spidev1.0";\nstatic uint8_t mode = 0;\nstatic uint8_t bits = 8;\nstatic uint32_t speed = 50000;\nstatic uint16_t delay;\n#define WRITE 0\n\nstatic void transfer(int fd)\n{\n    int ret;\n    int i = 10000;\n    uint8_t tx[] = {\n        0x9f, 0x00, 0x00, 0x00, 0x01, 0x02,\n    };\n    uint8_t rx[ARRAY_SIZE(tx)] = {0, };\n    struct spi_ioc_transfer tr_txrx[] = {\n        {\n            .tx_buf = (unsigned long)tx, /* 发送数据缓存区 */\n            .rx_buf = 0, /* 半双工通信，接收缓存区置为 0 */\n            .len = 1, /* 发送命令码，1 个字节 */\n            .delay_usecs = delay,\n            .speed_hz = speed,\n            .bits_per_word = bits,\n        },\n        {\n            .rx_buf = (unsigned long)rx, /* 接收数据缓存区 */\n            .len = 3, /* 接收数据长度为 3 */\n            .delay_usecs = delay,\n            .speed_hz = speed,\n            .bits_per_word = bits,\n        }\n    };\n    ret = ioctl(fd, SPI_IOC_MESSAGE(2), &tr_txrx[0]); /* 发送 2 条消息 */\n    if (ret == 1) {\n        pabort("can't review spi message");\n    }\n    for (ret = 0; ret < tr_txrx[1].len; ret++) {\n
```



```
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 'T':
    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;
}
```

程序源码也可以参见光盘中的“开发示例”对应目录下的“spidev\_test.c”文件，将程序编译成可执行文件（光盘中的示例文件为“spidev\_test”）并将其拷贝至开发套件的“/root/”目录下，然后通过串口终端输入以下命令即可运行程序，程序运行后返回结果显示如图5.11所示。

```
root@EasyARM-iMX283 ~# ./spidev_test
```

```
spi mode: 0
bits per word: 8
max speed: 50000 Hz (50 KHz)

C2 25 15
```

图5.11 SPI示例执行结果



## 6. EasyARM-iMX283 的Boot Loader

本章主要介绍 EasyARM-iMX283 的 Boot Loader。EasyARM-iMX283 是使用了 U-Boot 作为 Boot Loader。若 EasyARM-iMX283 设置了 NAND Flash 启动方式 (JP4、JP5、JP3、JP2 跳线断开连接,JP6 用短路器连接), 系统上电后将会在 NAND Flash 中读取 U-Boot 到 SDRAM 运行, 然后引导 Linux 的启动。

本章所引用的源码位于光盘目录 “3.Linux\6.源代码” 下。

### 6.1 U-Boot简介

U-Boot, 全称 Universal Boot Loader, 是遵循 GPL 条款的开源项目。从 FADSROM、8xxROM、PPCBoot、ARMBoot 逐步发展演化而来。U-Boot 不仅支持嵌入式 Linux 系统的引导, 它还支持 NetBSD、VxWorks、QNX、RTEMS、ARTOS、LynxOS 嵌入式操作系统。U-Boot 除了支持 PowerPC 系列的处理器外, 还能支持 MIPS、x86、ARM、NIOS、XScale 等多种常用系列的处理器。

### 6.2 U-Boot源代码目录结构

U-Boot源代码结构清晰, 按照CPU以及开发板进行安排。进入U-Boot源码目录, 可以看到如下的一些目录 (省略了其它文件), 常见的重要目录的说明如表6.1所示。

api	api_examples	board	common	cpu	disk	doc	drivers
examples	fs	include	lib_arm	lib_avr32	lib_blackfin	libfdt	
lib_generic	lib_i386	lib_m68k	lib_microblaze	lib_mips	lib_nios	lib_nios2	lib_ppc
lib_sh	lib_sparc	nand_spl	net	onenand_ipl	post	tools	

表6.1 U-Boot 重要目录说明

目 录	说 明
board	存放了 U-Boot 所支持的开发板的相关代码, 目前支持几十个不同体系架构的开发板, 如 evb4510、pxa255_idp、omap2420h4 等
common	U-Boot 命令实现代码目录
cpu	包含了不同处理器相关的代码, 按照不同的处理器进行分类, 如 arm920t、arm926ejs、i386、mios2 等
drivers	U-Boot 所支持外设的驱动程序。按照不同类型驱动进行分类如 spi、mtd、net 等等
fs	U-Boot 所支持的文件系统的代码。目前 U-Boot 支持 cramfs、ext2、fat、fdos、jffs2、reiserfs
include	U-Boot 头文件目录, 里面还包含各种不同处理器的相关头文件等, 以 asm-体系架构这样的目录出现。另外, 不同开发板的配置文件也在这个目录下: include/configs/开发板.h
lib_xxx	不同体系架构的一些库文件目录
net	U-Boot 所支持网络协议相关代码, 如 bootp、nfs 等
tools	U-Boot 工具源代码目录。其中的一些小工具如 mkimage 就非常实用

### 6.3 编译U-Boot

请把光盘资料中对应 DDR2 容量的 bootloader.tar.bz2 文件复制到 Linux 主机的工作目录, 然后解压该压缩包:

```
$ tar -jxvf bootloader.tar.bz2
```



然后得到一个 bootloader 目录。在 bootloader 目录下有 elftosb、u-boot-2009.08 和 imx-bootlets-src-10.12.01 等三个目录。u-boot-2009.08 目录内有 U-Boot 的源代码。把 U-Boot 源码编译后得到 U-Boot 文件。然后 U-Boot 文件需要 imx-bootlets-src-10.12.01 目录下的工具进一步编译成 imx28\_ivt\_uboot.sb 文件（**用于烧写到 NAND flash 的文件**）。elftosb 目录下提供了 32bit 和 64bit Linux 系统下适用的 elftosb 转换工具。

生成适用于 EasyARM-iMX283 的 U-Boot 文件需要按如下步骤进行操作：

首先，进入 u-boot-2009.08 目录，清除原有的编译文件，其对应的终端命令如下：

```
$ cd bootloader/u-boot-2009.08  
$ make ARCH=arm CROSS_COMPILE=arm-fsl-linux-gnueabi- distclean
```

其次，需要配置 U-Boot 的平台为 mx28\_evk\_config，对应的终端命令如下：

```
$ make ARCH=arm CROSS_COMPILE=arm-fsl-linux-gnueabi- mx28_evk_config  
Configuring for mx28_evk board...
```

然后，执行编译，对应的终端命令如下：

```
$make ARCH=arm CROSS_COMPILE=arm-fsl-linux-gnueabi-
```

编译完成后将在 u-boot-2009.08 目录的根目录下得到 U-Boot 文件。但是 U-Boot 文件并不能作为固件在 EasyARM-iMX283 平台的 NAND Flash 中直接启动。U-Boot 文件需要使用 imx-bootlets-src-10.12.01 目录下的工具进一步编译成带电源配置的 imx28\_ivt\_uboot.sb 固件文件。

把 u-boot 复制到 imx-bootlets-src-10.12.01 目录下：

```
$ cp u-boot .. /imx-bootlets-src-10.12.01
```

进行 u-boot 转换前需要先将 elftosb 目录下的“elftosb\_32bit 或 elftosb\_64bit”文件改名为“elftosb”并复制到“/usr/bin/”目录下（请以用户搭建的 Linux 上位机系统位宽为准）。复制完后需要将 elftosb 赋予可执行的权限。

```
$ cd .. /elftosb/  
$ mv elftosb_64bit elftosb      # 若用户的 Linux 上位机系统是 32bit 的，则选择 elftosb_32bit 文件  
$ sudo cp elftosb /usr/bin/  
$ sudo chmod 777 /usr/bin/elftosb
```

进入 imx-bootlets-src-10.12.01 目录，然后执行编译命令：

```
$ cd .. /imx-bootlets-src-10.12.01  
$ sudo cp elftosb /usr/bin/  
$ ./build
```

编译完成后 imx-bootlets-src-10.12.01 目录下的 imx28\_ivt\_uboot.sb 文件就是可以烧写到 NAND Flash 的固件文件。具体的烧写方法请参考 2.2 小节“**烧写 U-Boot**”的内容。

## 6.4 U-Boot 基本命令

在 U-Boot 启动阶段，在串口终端按任意键（**如空格键**）进入 U-Boot 的命令行，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)
```

#### 6.4.1 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)
```

#### 6.4.2 通过网络启动内核

当把内核的固件文件uImage放在PC机的tftp服务器的根目录时（[如“网络烧写方案”介绍的“Cisco TFTP Server”工具软件目录](#)），就可以通过网络来启动内核，方便内核调试。假设tftp服务器的IP为 192.168.12.122，EasyARM-iMX283 的IP可以设置为 192.168.12.124，那么在U-Boot中执行下面指令：

```
MX28 U-Boot > setenv ipaddr 192.168.12.124  
MX28 U-Boot > setenv serverip 192.168.12.122  
MX28 U-Boot > run settftpboot
```

然后重启即可。这样在 EasyARM-iMX283 每次开机时，U-Boot 都会在 tftp 服务器中下载 uImage 内核文件到 DRAM，然后在 DRAM 中启动内核。

### 6.5 U-Boot Tools

U-Boot 提供了一些有用的小工具，在 U-Boot 源代码的 tools 目录下。这些工具都是在主机上使用的。编译完毕，可以将这些小工具复制到系统目录如/usr/bin 目录下，以方便使用。

其中的 mkimage 工具，在编译内核的时候需要用到，务必复制到系统/usr/bin 目录下（[如使用 ZLG 网官提供的 ubuntu，不需这一步](#)），或者将 U-Boot 的 tools 目录添加到系统目录中。该工具可以生成 U-Boot 格式的文件，以配合 U-Boot 使用。

先进入 tools 目录，复制 mkimage 到/usr/bin 目录

```
$ cd tools/  
$ cp mkimage /usr/bin/
```



## 7. Linux内核编译和驱动要点

本章主要介绍了 EasyARM-iMX283 配套的 Linux 内核源码的编译、简单剪裁、固件制作。同时介绍了一些驱动的使用和编写。

本章所引用的源码位于光盘目录“3.Linux\6.源代码”下。

### 7.1 编译内核

参考6.5节内容准备好mkimage文件，并复制到/usr/bin/目录下。

```
$ sudo cp mkimage /usr/bin/
```

#### 7.1.1 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
```

#### 7.1.2 运行SPI补丁

若用户使用的是 EasyARM-iMX287 开发套件或者 M287 核心板，则解压完内核后需要先运行 SPI3 补丁，否则可以直接跳过这一步骤。

将“spi3\_for\_v1\_03.patch”文件拷贝至“linux-2.6.35.3”目录，然后在该目录下执行 patch 命令：

```
$ patch -p1 < spi3_for_v1_03.patch
```

#### 7.1.3 备份内核配置文件

注意：默认的内核配置文件为.config，如需修改内核配置，请提前备份，具体方法为在“linux-2.6.35.3”目录下执行以下命令：

```
$ cp .config EasyARM-iMX283_defconfig
```

恢复为原来的内核配置时只需拷贝回原来的 config 文件即可：

```
$ cp EasyARM-iMX283_defconfig .config
```

#### 7.1.4 编译内核

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

## 7.2 生成imx28\_iwt\_linux.sb内核固件

imx28\_iwt\_linux.sb 内核固件是可以让系统直接从 Linux 内核中启动，而不需要 U-Boot 的引导，从而减少了系统的开机时间。

制作 imx28\_iwt\_linux.sb 固件首先需求制作出来 zImage 文件。这需要进入内核代码目录输入“make zImage”命令进行编译：

```
$ cd linux-2.6.35.3  
$ make zImage
```



编译完成后将在arch/arm/boot/目录下生成zImage文件。把这个zImage文件复制到imx-bootlets-src-10.12.01目录下（见6.3节）。进入imx-bootlets-src-10.12.01目录，然后输入“./build”命令：

```
$ cd imx-bootlets-src-10.12.0  
$ ./build
```

命令执行完成后，将imx-bootlets-src-10.12.01目录下生成imx28\_ivt\_linux.sb固件。

Linux内核启动时，是需要传入启动参数的；Linux内核是由U-Boot引导启动时，启动参数是由U-Boot传递的。但是若系统直接从Linux内核启动，启动参数是由谁的传递呢？其实imx28\_ivt\_linux.sb固件中不单包含了Linux内核代码，还包含了一段引导代码。当系统从imx28\_ivt\_linux.sb固件启动时，是先执行这一段引导代码。这段引导代码任务包含了初始化电源和一部分硬件设备，以及给即将启动的Linux内核传递启动参数。

那么启动参数用户如何来自定义呢？

在imx-bootlets-src-10.12.01目录下linux\_prep/cmdlines/iMX28\_EVK.txt文件内容如下：

```
gpmi=g console=ttyAM0,115200n8 console=tty0 ubi.mtd=1 root=ubi0:rootfs rootfstype=ubifs fec_mac=ethact  
linux_prep/board/iMX28_EVK.c文件的cmdline_def变量(在该文件的最后一行)的值为:  
char cmdline_def[] = "gpmi=g console=ttyAM0,115200n8 ubi.mtd=1 root=ubi0:rootfs rootfstype=ubifs  
fec_mac=ethact";
```

当用户需要修改imx28\_ivt\_linux.sb的启动参数时，这两个地方也要修改。

生成imx28\_ivt\_linux.sb固件后，可以进行烧写：

- 如果使用USB方式烧写到EasyARM-iMX283的NAND Flash时，请把该固件文件替换到MtgTool程序的“Profiles\MX28 Linux Update\OS Firmware\files”目录；
- 如果使用SD方式烧写到EasyARM-iMX283的NAND Flash时，请把该固件文件替换到“SD烧写方案\i.MX283\_for\_kernelsb”目录。

### 7.3 配置内核

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7.1所示。

```
$ make menuconfig
```

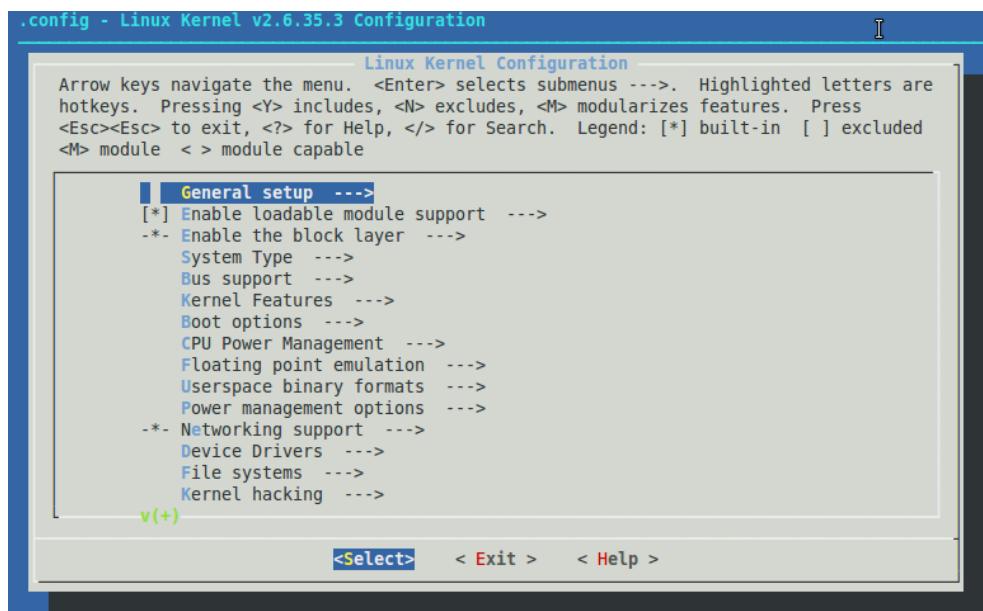


图7.1 内核配置主菜单

在该界面中用户可以随意配置内核。键盘上的“上”和“下”方向键是控制选择光标在菜单的移动，当光标移动到某一项菜单项时，按下 Enter 键即进入该项菜单下的子菜单；键盘上的“左”和“右”方向键是控制选择 Select、Exit、Help。当选择 Exit，然后按下 Enter 键时，将返回上一层菜单；当选择 Help 时，然后按下 Enter 键时，是查看该菜单的帮助内容。

当启用一项目时，请在该项键入 y。这会把该项功能静态编译到内核，如下所示：

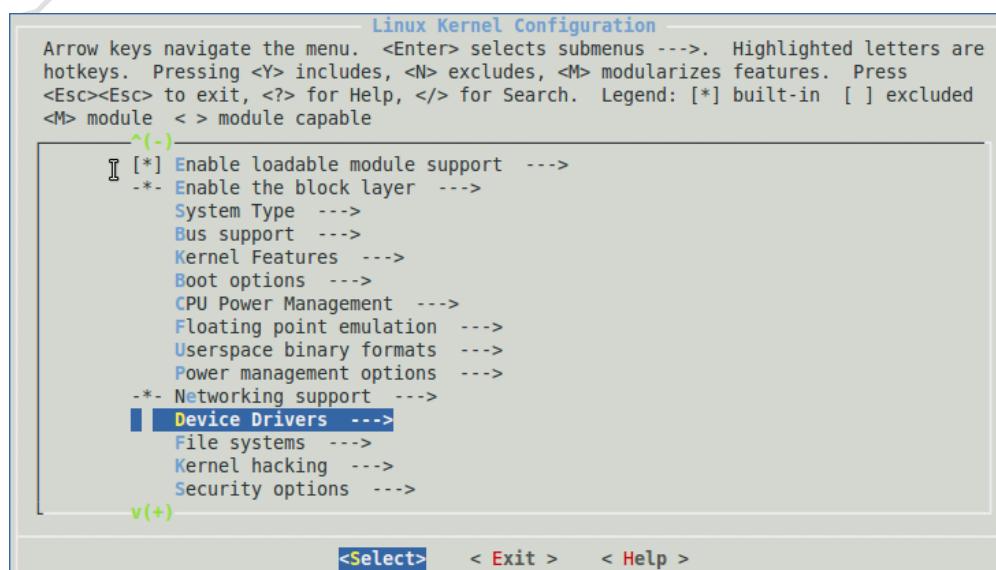
[\*] Network device support --->

当在该项键入 M 时，这会把该项功能动态编译成内核模块，在相应目录生成\*.ko 文件，如下图所示。

<M> Serial ATA and Parallel ATA drivers --->

### 1. 启用TF卡驱动

进入Device Drivers子菜单，如图7.2所示：





## 图7.2 进入 Device Drivers 子菜单

选中Block devices，如图7.3所示：

```
Device Drivers
Arrow keys navigate the menu. <Enter> selects submenus --->. Highlighted letters are
hotkeys. Pressing <Y> includes, <N> excludes, <M> modularizes features. Press
<Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [*] built-in [ ] excluded
<M> module < > module capable

Generic Driver Options --->
< > Connector - unified userspace <-> kernelspace linker --->
<*> Memory Technology Device (MTD) support --->
< > Parallel port support --->
[*] Block devices --->
[*] Misc devices --->
< > ATA/ATAPI/MFM/RLL support (DEPRECATED) --->
      SCSI device support --->
```

图7.3 选中 Block devices 支持

进入MMC/SD/SDIO card support子菜单，如图7.4所示：

```
Device Drivers
Arrow keys navigate the menu. <Enter> selects submenus --->. Highlighted letters are
hotkeys. Pressing <Y> includes, <N> excludes, <M> modularizes features. Press
<Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [*] built-in [ ] excluded
<M> module < > module capable

^(-)
[*] Voltage and Current Regulator Support --->
<*> Multimedia support --->
      Graphics support --->
< > Sound card support --->
[*] HID Devices --->
[*] USB support --->
<*> MMC/SD/SDIO card support --->
< > Sony MemoryStick card support (EXPERIMENTAL) --->
[*] I²C Support --->
```

图7.4 进入 MMC/SD 配置子菜单

将MMC/SD/SDIO card support子菜单内各模块选中情况设置为图7.5所示：

```
MMC/SD/SDIO card support
Arrow keys navigate the menu. <Enter> selects submenus --->. Highlighted letters are
hotkeys. Pressing <Y> includes, <N> excludes, <M> modularizes features. Press
<Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [*] built-in [ ] excluded
<M> module < > module capable

-- MMC/SD/SDIO card support
[ ] MMC debugging
[*] Assume MMC/SD cards are non-removable (DANGEROUS)
      *** MMC/SD/SDIO Card Drivers ***
<*> MMC block device driver
[*] Use bounce buffer for simple hosts
< > SDIO UART/GPS class support
< > MMC host test driver
      *** MMC/SD/SDIO Host Controller Drivers ***
< > Secure Digital Host Controller Interface support
< > MMC/SD/SDIO over SPI
[ ] Freescale i.MX Secure Digital Host Controller Interface PIO mode
<*> MXS MMC support
```

图7.5 TF 卡模块配置

## 2. 启用USB host

进入Device Drivers子菜单，如图7.6所示：

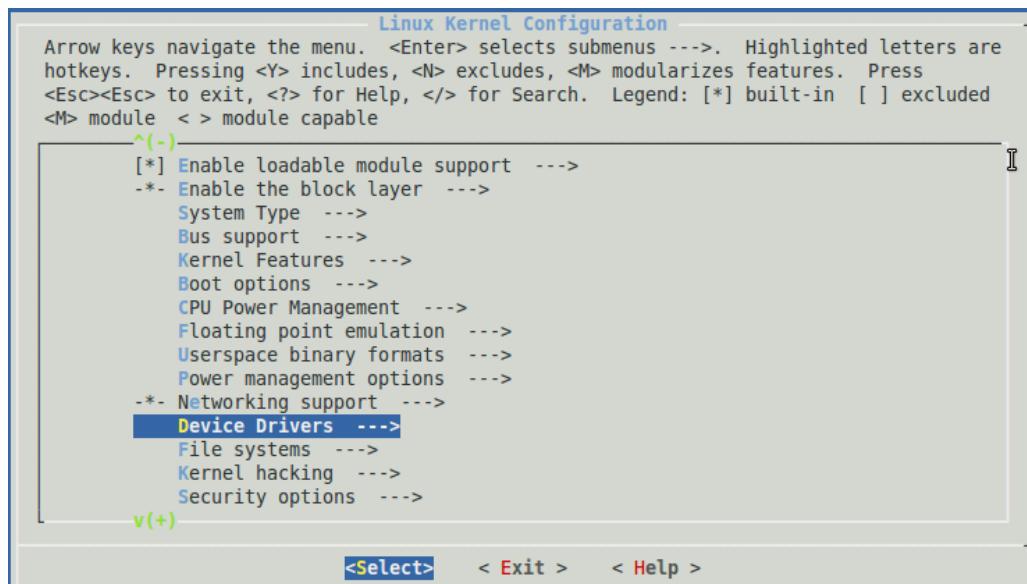


图7.6 进入 Device Drivers 子菜单

进入USB support子菜单，如图7.7所示：

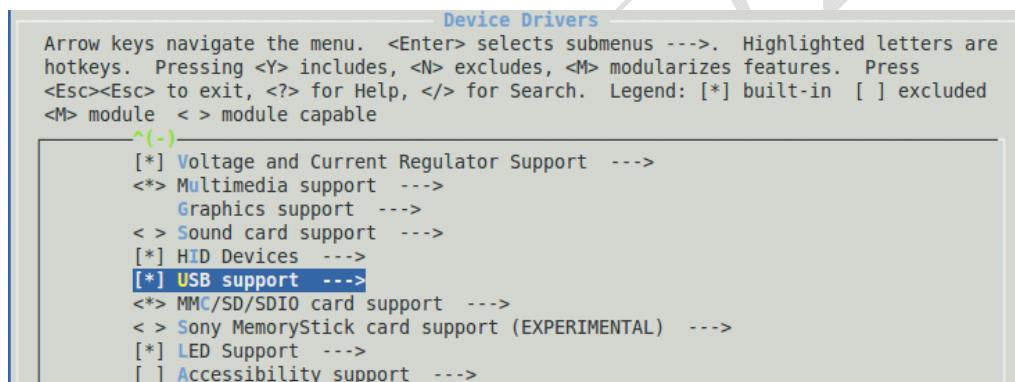


图7.7 进入 USB support 子菜单

选中如图7.8所示的两个模块：

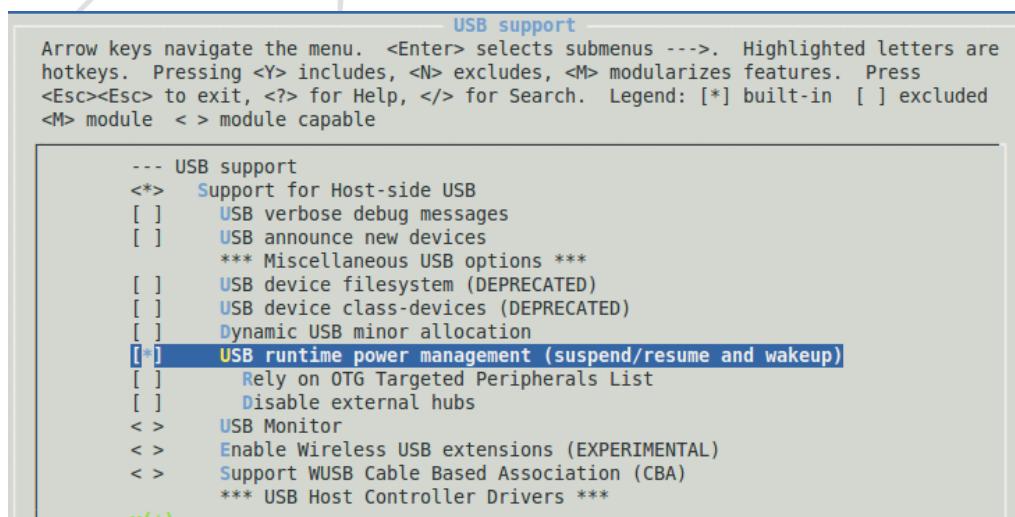


图7.8 选中 Host 和 USB 电源管理两个模块

按图7.9和图7.10所示，选中相应的模块：

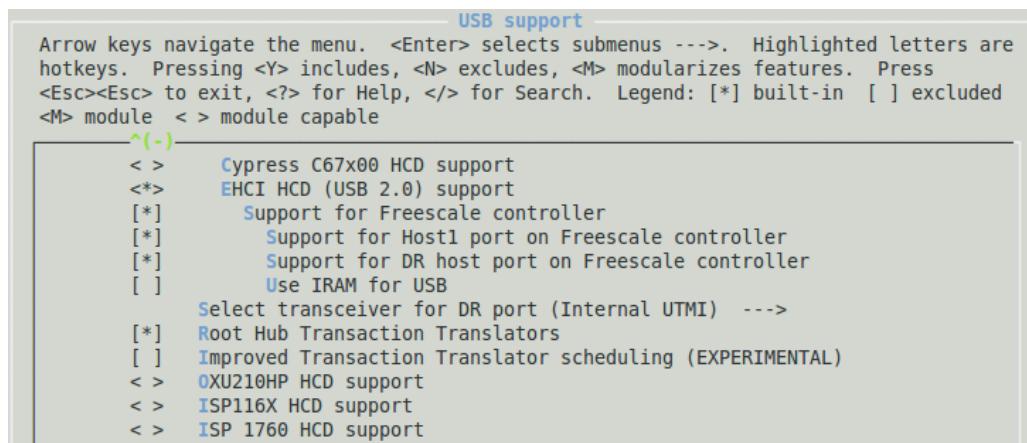


图7.9 按图中选中各个 USB 子模块

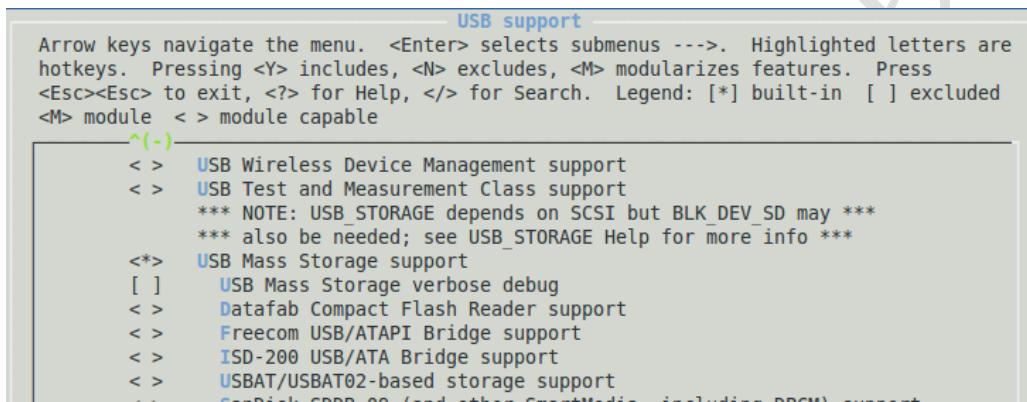


图7.10 选中 U 盘存储模块

### 3. 启用I<sup>2</sup>C

在Device Drivers菜单中，进入I2C support子菜单，如图7.11所示：

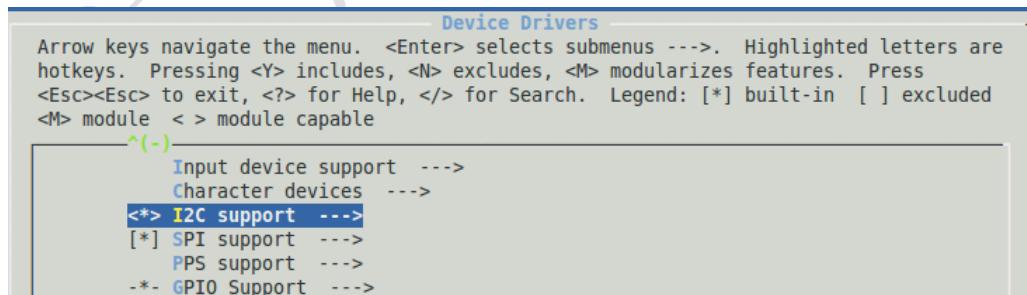


图7.11 进入 I2C support 子菜单

按图7.12所示选中各个模块：

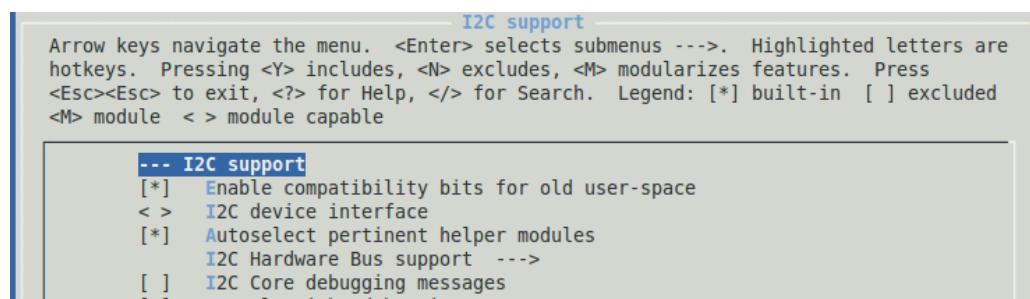


图7.12 选中图中两个 I2C 子模块

进入I2C Hardware Bus support子菜单，如图7.13所示：

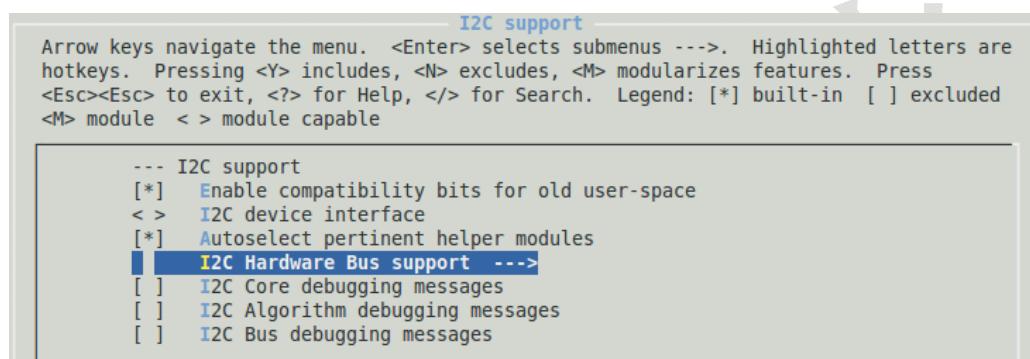


图7.13 进入 I2C Hardware Bus support 子菜单

按图7.14所示选中各个子模块：

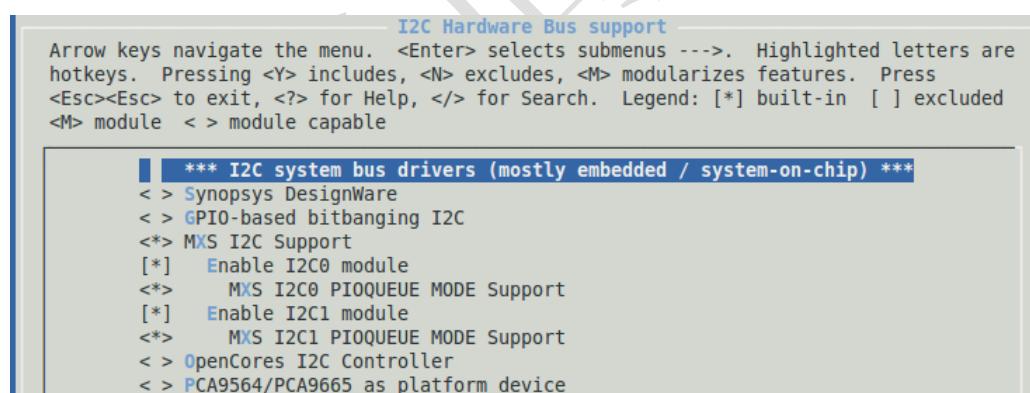


图7.14 选中 I2C 各个子模块

## 7.4 内核GPIO使用方法

i.MX283 处理器的大部分引脚都实现了功能复用，当需要把某一引脚用作 GPIO 时，需要先将其配置成 GPIO 功能模式，然后才能操作该 GPIO。

下面以 GPIO1\_17 为例，介绍 GPIO 的配置和操作。从 IMX283 芯片手册中可以得知 GPIO1\_17 的引脚名字是 LCD\_D17，在内核代码的/arch/arm/mach-mx28/mx28\_pins.h 文件中查阅到对 PINID\_LCD\_D17 引脚的定义如下：

```
#define PINID_LCD_D17          MXS_PIN_ENCODE(1, 17)
```

修改./arch/arm/mach-mx28/mx28evk\_pins.c 文件中的 static struct pin\_desc mx28evk\_fixed\_pins[] 数组中对引脚的描述就能设置引脚的功能。



如需要把 LCD\_D17 引脚配置成 GPIO 功能，并设置其属性（[如输出电压，输出电流等，具体可设置的属性需要参考 MCIMX28RM 手册](#)），则需要对 PINID\_LCD\_D17 的描述做如下的修改：

```
{  
    .name      = "RS485_DIR",                      //自定义的名称  
    .id        = PINID_LCD_D17,                     //引脚 ID  
    .fun       = PIN_GPIO,                          //fun 选择 PIN_GPIO 功能  
    .strength  = PAD_8MA,  
    .voltage   = PAD_3_3V,  
    .drive     = 1,  
},
```

PIN\_GPIO 定义在 “arch/arm/plat-mxs/include/mach/pinctrl.h” 文件中，如程序清单 7.1 所示。PIN\_GPIO 的值为 3，对应 LCD\_D17 引脚的 GPIO 功能模式（[关于引脚模式配置的更多介绍，可查阅 i.MX283 芯片用户手册《IMX28RM.pdf》](#)）。

程序清单 7.1 引脚的功能定义

```
enum pin_fun {  
    PIN_FUN1 = 0,  
    PIN_FUN2,  
    PIN_FUN3,  
    PIN_GPIO,  
};
```

引脚设置成 GPIO 工作模式，并设置好其属性后，调用 GPIOLIB 的通用函数来控制这个 GPIO 了。首先调用 gpio\_request 获得这个 GPIO 的控制权：

```
gpio_request(MXS_PIN_TO_GPIO(PINID_LCD_D17), "RS485DIR");
```

当需要 LCD\_D17 作输出功能使用时，调用 gpio\_direction\_output 设置该 GPIO 为输出模式：

```
gpio_direction_output(MXS_PIN_TO_GPIO(PINID_LCD_D17), 0); //设置为输出模式，初始化输出低电平
```

或

```
gpio_direction_output(MXS_PIN_TO_GPIO(PINID_LCD_D17), 1); //设置为输出模式，初始化输出高电平
```

然后调用 gpio\_set\_value 来设置输出高电平或低电平：

```
gpio_set_value(MXS_PIN_TO_GPIO(PINID_LCD_D17), 0);           // 控制输出低电平
```

或

```
gpio_set_value(MXS_PIN_TO_GPIO(PINID_LCD_D17), 1);           // 控制输出高电平
```

当需要 LCD\_D17 作为输入功能使用时，调用 gpio\_direction\_input 设置该 GPIO 为输入模式：

```
gpio_direction_input (MXS_PIN_TO_GPIO(PINID_LCD_D17));
```

然后调用 gpio\_get\_value 获得 GPIO 的电平输入状态：

```
gpio_get_value (MXS_PIN_TO_GPIO(PINID_LCD_D17));
```

返回值为“1”表示输入状态为高电平；返回值为“0”表示输入状态为低电平。

## 7.5 蜂鸣器驱动

EasyARM-iMX283 的底板上板载有一个蜂鸣器，其原理图如图7.15 所示。当把JP1 用短路器短接后，在BUZZER输入高电平时，蜂鸣器鸣叫；当在BUZZER输入低电平时，蜂鸣器停止鸣叫。

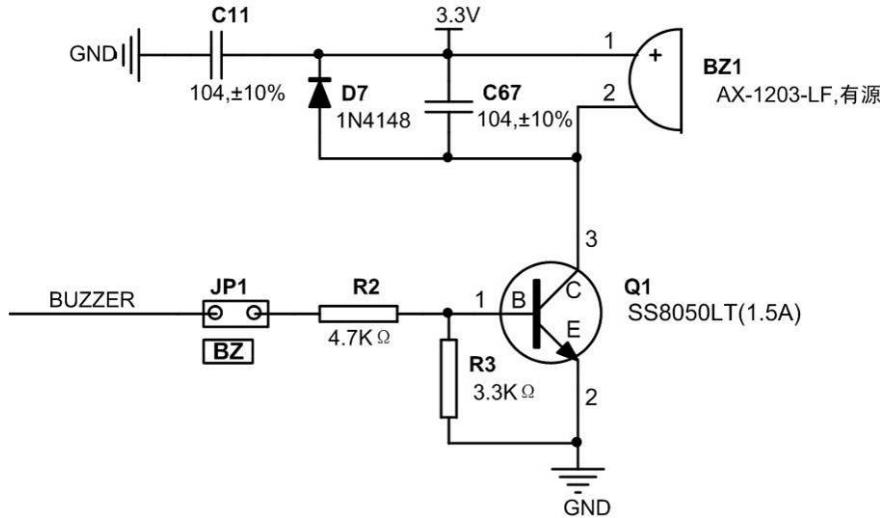


图7.15 蜂鸣器在原理图

根据原理图 BUZZER 是连接到 i.MX283 处理器的上 LCD\_D21 引脚，该引脚是可以复用为 GPIO1\_21。该引脚定义在内核原码的 arch/arm/mach-mx28/mx28\_pins.h 文件，如下所示：

```
#define PINID_LCD_D21          MXS_PIN_ENCODE(1, 21)
```

根据这个定义，并结合上一节的([内核 GPIO 使用方法](#))内容，使用 gpio\_direction\_output() 函数便可控制 LCD\_D21 引脚输出高/低电平，以实现对蜂鸣器的控制，如下所示：

```
gpio_direction_output(MXS_PIN_TO_GPIO(PINID_LCD_D21), 1);      //输出高电平，蜂鸣器鸣叫  
gpio_direction_output(MXS_PIN_TO_GPIO(PINID_LCD_D21), 0);      //输出低电平，蜂鸣器停止鸣叫
```

蜂鸣器驱动代码在光盘中的 beep 目录中的 beep.c 文件。在该文件引用了内核原码中 arch/arm/mach-mx28/mx28\_pins.h 文件，如下所示：

```
#include <../arch/arm/mach-mx28/mx28_pins.h>
```

下面详细介绍这个 beep.c 驱动程序代码文件。

### 7.5.1 驱动加载

这个蜂鸣器驱动模块文件是 beep.ko。使用下面的指令把驱动模块加载到内核：

```
root@EasyARM-iMX283 ~# insmod beep.ko
```

在运行该命令时，beep.c文件中的module\_init 宏指向的gpio\_init函数将被调用，以实现对蜂鸣器驱动的初始化工作。gpio\_init函数的实现如程序清单7.2所示。

程序清单7.2 驱动的初始化操作

```
static int __init gpio_init(void)  
{  
    misc_register(&gpio_mscdev);
```



```
    printk(DEVICE_NAME" up.\n");
    return 0;
}
module_init(gpio_init);
```

在该函数中，调用了misc\_register函数注册一个miscdevice类型的gpio\_misctdev对象。该对象的实现如程序清单7.3所示。

程序清单7.3 gpio\_misctdev 的实现

```
static struct miscdevice gpio_misctdev = {
    .minor  = MISC_DYNAMIC_MINOR,
    .name    = DEVICE_NAME,
    .fops   = &gpio_fops,
};
```

miscdevice 类型的对象是描述 misc 类型的设备驱动。Linux 对大多数的驱动做了分类：块设备驱动、网络驱动、I<sup>2</sup>C 驱动、USB 驱动、SPI 驱动、音频驱动等，但是一些不好分类的都归纳为 misc 类型设备驱动。这个蜂鸣器驱动是属于自定义的，所以属于 misc 设备驱动。

misc 设备的驱动设备文件的主设备号都是 10，其子设备号是由 minor 成员指定。若 minor 成员赋值为 MISC\_DYNAMIC\_MINOR，那么系统对其次设备号进行动态分配。为避免驱动注册时与已注册的 misc 类驱动的次设备号相冲突，一般采用次设备号动态分配方式。

gpio\_misctdev 对象的 name 成员赋值为 DEVICE\_NAME。DEVICE\_NAME 的定义为：

```
#define DEVICE_NAME "imx283_beep"
```

当 gpio\_misctdev 对象被注册后，将在文件系统的生成设备文件节点：

```
/dev/imx283_beep
```

gpio\_misctdev 对象的 fops 成员指向的 gpio\_fops 对象，是一个 file\_operations 类型文件操作列表。其实现如程序清单7.4所示。

程序清单7.4 gpio\_fops 的实现

```
static struct file_operations gpio_fops = {
    .owner        = THIS_MODULE,
    .open         = gpio_open,
    .write        = gpio_write,
    .release      = gpio_release,
    .ioctl        = gpio_ioctl,
};
```

文件操作列表的实现是驱动程序实现的关键部分，它主要用于提供设备驱动文件相关操作的实现方法。例如，在应用程序对驱动设备文件执行如下操作时，其对应的成员指向的函数将被调用：

- 执行 open 操作时，文件操作列表的 open 成员指向的函数将被调用；
- 执行 write 操作时，文件操作列表的 write 成员指向的函数将被调用；
- 执行 ioctl 操作时，文件操作列表的 ioctl 成员指向的函数将被调用；
- 执行 close 操作时，文件操作列表的 release 成员指向的函数将被调用。

## 7.5.2 卸载驱动



在 Shell 终端执行下面指令时，将卸载蜂鸣器驱动模块：

```
root@EasyARM-iMX283 ~# rmmod beep
```

当执行该指令时，beep.c文件中module\_exit宏指向的gpio\_exit函数将被调用，以实现驱动模块的退出操作。gpio\_exit函数的实现如程序清单7.5所示。

程序清单7.5 gpio\_exit 函数的实现

```
static void __exit gpio_exit(void)
{
    misc_deregister(&gpio_misctdev);
    printk(DEVICE_NAME " down.\n");
}
module_exit(gpio_exit);
```

gpio\_exit 函数的主要目的是把在初始化时注册的 gpio\_misctdev 对象注销。当 gpio\_misctdev 对象注销后，文件系统中的/dev/imx283 设备文件也随之消失。

### 7.5.3 Open调用的实现

当应用程序对/dev/imx283 设备文件执行open操作时，文件操作列表的open成员指向的gpio\_open函数将执行。gpio\_open函数的实现如程序清单7.6所示。

程序清单7.6 gpio\_open 函数的实现

```
static int gpio_open(struct inode *inode, struct file *filp)
{
    gpio_request(MXS_PIN_TO_GPIO(PINID_LCD_D21), "beep");
    return 0;
}
```

该函数主要是实现了向内核申请 PINID\_LCD\_D21 这个 GPIO 端口。

当应用程序对/dev/imx283 设备文件执行 open 调用完成后，如没有出错，将获得 fd (**int** 类型的) 文件描述符。后面的操作都是对这个 fd 文件描述符进行操作。

### 7.5.4 write调用的实现

当应用程序对fd文件描述符执行write调用的时，文件操作列表的write成员指向的gpio\_write函数将被调用。gpio\_write函数的实现如程序清单7.7所示。

程序清单7.7 gpio\_write 函数的实现

```
ssize_t gpio_write(struct file *filp, const char __user *buf, size_t count, loff_t *f_pos)
{
    char data[2];

    copy_from_user(data, buf, count);
    gpio_direction_output(MXS_PIN_TO_GPIO(PINID_LCD_D21), data[0]);
    return count;
}
```

在该函数的传入参数中，其中有 buf 和 count 参数。buf 表示应用程序调用 write 时，要写入数据的缓冲区；count 表示缓冲区中有效数据的长度。



注意，buf 缓冲区是在用户空间的，而蜂鸣器驱动程序是运行在内核空间的，所以在内核空间的代码是不能直接访问用户空间的缓冲区，所以需要使用 copy\_from\_user 宏把 buf 中用户空间中要写入数据复制到 data 缓冲区中。

至于应用程序要控制 LCD\_D21 是输出高电平（值为 1）或低电平（值为 0），则保存在 data[0] 中。然后调用 gpio\_direction\_output 函数实施操作。

### 7.5.5 ioctl函数的实现

当应用程序对fd文件描述符执行ioctl调用时，文件操作列表的ioctl成员指向的gpio\_ioctl函数将被调用。gpio\_ioctl函数的实现如程序清单7.8所示。

程序清单7.8 gpio\_ioctl 的调用

```
static int gpio_ioctl(struct inode *inode, struct file *filp, unsigned int command, unsigned long arg)
{
    int data;

    switch (command) {
    case 0:
        gpio_direction_output(MXS_PIN_TO_GPIO(PINID_LCD_D21), 1);
        break;
    case 1:
        gpio_direction_output(MXS_PIN_TO_GPIO(PINID_LCD_D21), 0);
        break;
    }

    return 0;
}
```

gpio\_ioctl 的输入参数中，有 command 和 arg 参数。command 参数表示应用程序传入的控制命令；arg 参数表示控制命令的命令参数。在这里应用程序只会传入蜂鸣器鸣叫命令（命令值为 0）或蜂鸣器停止鸣叫命令（命令值为 1），而没有命令参数。

在 gpio\_ioctl 函数中调用 gpio\_direction\_output 函数分别根据命令控制 LCD\_D21 引脚输出高/低电平。

### 7.5.6 close调用的实现

当应用程序对fd文件描述符执行close调用时，文件操作列表中release成员指向的gpio\_release函数将被调用。gpio\_release函数的实现如程序清单7.9所示。

程序清单7.9 gpio\_release 函数的实现

```
static int gpio_release(struct inode *inode, struct file *filp)
{
    gpio_free(MXS_PIN_TO_GPIO(PINID_LCD_D21));
    return 0;
}
```

该函数的主要工作是释放 LCD\_D21 引脚。

### 7.5.7 编译驱动代码



在光盘的beep目录下提供了Makefile文件，可执行对本蜂鸣器驱动代码的编译工作。该Makefile的内容如程序清单7.10所示。

程序清单7.10 Makefile 文件的实现

```
# Makefile2.6
ifeq ($(KERNELRELEASE),)
#kbuild syntax. dependency relationships of files and target modules are listed here.
#gpiodrv-objs := gpiodrv.c
obj-m := beep.o
else
PWD := $(shell pwd)
KVER = 2.6.31
KDIR := /home/proton/EasyARM-iM283/kernel/linux-2.6.35.3
all:
    $(MAKE) -C $(KDIR) M=$(PWD) modules
clean:
    rm -rf *.cmd *.o *.mod.c *.ko .tmp_versions
endif
```

当需要对本驱动代码进行编译时，必须先编译光盘中提供Linux内核代码（见前面的7.1小节）。内核编译完成后，把本驱动代码的Makefile中的KDIR变量指向的内核路径改成刚编译出来的内核代码的路径，然后，执行make命令即可编译，如图7.16所示。

```
proton@proton:~/EasyARM-iM283/drivers/beep$ make
make -C /home/proton/EasyARM-iM283/kernel/linux-2.6.35.3 M=/home/proton/EasyARM-iM283/drivers/beep modules
make[1]: 正在进入目录 `/home/proton/EasyARM-iM283/kernel/linux-2.6.35.3'
  CC [M]  /home/proton/EasyARM-iM283/drivers/beep/beep.o
/home/proton/EasyARM-iM283/drivers/beep/beep.c: In function 'gpio_ioctl':
/home/proton/EasyARM-iM283/drivers/beep/beep.c:63: warning: unused variable 'data'
Building modules, stage 2.
MODPOST 1 modules
  CC      /home/proton/EasyARM-iM283/drivers/beep/beep.mod.o
  LD [M]  /home/proton/EasyARM-iM283/drivers/beep/beep.ko
make[1]:正在离开目录 `/home/proton/EasyARM-iM283/kernel/linux-2.6.35.3'
proton@proton:~/EasyARM-iM283/drivers/beep$
```

图7.16 编译驱动

编译完成后，将得到 beep.ko 驱动模块文件。

### 7.5.8 测试程序

beep目录下有一个test目录，内有本蜂鸣器驱动的测试程序代码。其中beep\_test.c文件的内容如程序清单7.11所示。

程序清单7.11 蜂鸣器测试程序代码

```
#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>
#include <asm/ioctls.h>
#include <time.h>
#include <pthread.h>

int main(void)
{
    int fd;
    char buf[1] = {0};

    fd = open("/dev/imx283_beep", O_RDWR);
    if (fd < 0) {
        perror("open /dev/imx283_gpio");
    }
    /* 测试驱动 write 调用 */
    printf("test write....\n");
    buf[0] = 1; /* 使蜂鸣器鸣叫 */
    write(fd, buf, 1);
    sleep(2);

    buf[0] = 0; /* 使蜂鸣器停止鸣叫 */
    write(fd, buf, 1);
    sleep(1);

    /* 测试驱动的 ioctl 调用 */
    printf("test ioctl.... \n");
    ioctl(fd, 0); /* 使蜂鸣器鸣叫 */
    sleep(2);
    ioctl(fd, 1); /* 使蜂鸣器停止鸣叫 */
    sleep(1);
}
```

输入 make 命令即可完成编译。编译完成后，将生成 beep\_test 文件。

在 EasyARM-iMX283 的 Linux 系统中，在/root 目录有已经编译好的 beep.ko 和 beep\_test 文件可供直接测试。登录 EasyARM-iMX283 的 Shell 命令行终端，可输入如下所示的命令进行测试：

```
root@EasyARM-iMX283/# cd /root
root@EasyARM-iMX283 ~# insmod beep.ko
imx283 up.
root@ EasyARM-iMX283 ~$./beep_test
test write....
test ioctl....
```



root@EasyARM-iMX283 ~#

## 7.6 设置LCD的时序

当用户需要更换LCD屏时，也需要修改在内核中对LCD的时序设置。在内核中，针对IMX28平台的LCD时序定义在drivers/video/mxs/lcd\_43wvf1g.c文件，如程序清单7.12所示。

程序清单7.12 LCD 时序列表

```
#define DOTCLK_H_ACTIVE 480           // 屏幕 x 轴像素长度
#define DOTCLK_H_PULSE_WIDTH 41         // 脉冲宽度
#define DOTCLK_HF_PORCH 5              // 水平前沿
#define DOTCLK_HB_PORCH 5              // 水平后延
#define DOTCLK_H_WAIT_CNT (DOTCLK_H_PULSE_WIDTH + DOTCLK_HF_PORCH)
#define DOTCLK_H_PERIOD (DOTCLK_H_WAIT_CNT + DOTCLK_HF_PORCH + DOTCLK_H_ACTIVE)

#define DOTCLK_V_ACTIVE 272           // 屏幕 y 轴像素长度
#define DOTCLK_V_PULSE_WIDTH 20         // 脉冲宽度
#define DOTCLK_VF_PORCH 5              // 垂直前沿
#define DOTCLK_VB_PORCH 5              // 垂直后沿
#define DOTCLK_V_WAIT_CNT (DOTCLK_V_PULSE_WIDTH + DOTCLK_VB_PORCH)
#define DOTCLK_V_PERIOD (DOTCLK_VF_PORCH + DOTCLK_V_ACTIVE + DOTCLK_V_WAIT_CNT)
```

上述宏定义了LCD控制器输出的时序，LCD控制器时序的具体设置流程可参考该文件中的init\_panel函数。

该文件中的static struct mxs\_platform\_fb\_entry fb\_entry变量初始化如程序清单7.13所示，用于设置屏幕的分辨率、像素时钟。

程序清单7.13 fb\_entry 的定义与初始化

```
static struct mxs_platform_fb_entry fb_entry = {
    .name = "HW480272F",           // LCD 名字
    .x_res = 272,                  // 屏幕 y 轴像素长度,
// 注意这里的 x、y 是与实际的屏的 x、y 对调的
    .y_res = 480,                  // 屏幕 x 轴像素长度
    .bpp = 16,                     // 显示颜色位数
    .dclk_f = 8000000,            // 像素时钟频率
    .lcd_type = MXS_LCD_PANEL_DOTCLK,
    .init_panel = init_panel,
    .release_panel = release_panel,
    .blank_panel = blank_panel,
    .run_panel = mxs_lcdif_run,
    .stop_panel = mxs_lcdif_stop,
    .pan_display = mxs_lcdif_pan_display,
    .bl_data = &bl_data,
};
```

用户修改程序清单7.12的宏和fb\_entry结构体的变量的值来修改LCD控制器的输出时序，实现需要的LCD的驱动。程序清单7.12的宏定义了LCD的时序，这些值都可以在LCD的



数据手册可以查阅。像素时钟fb\_entry.dclk\_f的值可以参考LCD的数据手册推荐的值进行设置，也可以通过以下计算公式计算：

```
pixclock=1012/(( DOTCLK_H_ACTIVE + DOTCLK_HF_PORCH + DOTCLK_HB_PORCH  
+DOTCLK_H_PULSE_WIDTH)  
*( DOTCLK_V_ACTIVE + DOTCLK_VF_PORCH + DOTCLK_VB_PORCH + DOTCLK_V_PULSE_WIDTH)  
* refresh)  
//refresh 一般为 60
```



## 8. 嵌入式Linux根文件系统

本章主要介绍了 EasyARM-iMX283 根文件系统的制作和 NFS 根文件系统的实现。

### 8.1 Linux根文件系统

通常情况下，Linux 内核启动后期，会寻找并挂载根文件系统。根文件系统可以存在于磁盘上，也可以是存在于内存中的映像，其中包含了 Linux 系统正常运行所必须的库和程序等等，按照一定的目录结构存放。Linux 根文件系统基本包括如下内容：

- 基本的目录结构：/bin、/sbin、/dev、/etc、/lib、/var、/proc、/sys、/tmp 等；
- 基本程序运行所需的库文件，如 glibc 等；
- 基本的系统配置文件，如 inittab、rc 等；
- 必要的设备文件，如 /dev/ttyS0、/dev/console 等；
- 基本应用程序，如 sh、ls、cd、mv 等。

### 8.2 FHS标准

理论上，Linux 根文件系统的目录结构是可以随意安排的，事实上很多 Linux 系统开发人员也这么做，但这就带来了不同开发人员之间不统一的情况存在，很容易出现混乱。后来这样的问题得到了重视，文件层次标准（**FHS**, **Filesystem Hierarchy Standard**）就在这种情况下出台的。FHS 经历了几个版本，目前最新版本是 2004 年 01 月 29 日发布的 V2.3 版本，详见 [www.pathname.com/fhs](http://www.pathname.com/fhs)。

FHS 对 Linux 根文件系统的基本目录结构做了比较详细的规定，尽管不是强制标准，但事实上，大部分 Linux 发行版都遵循这个标准。下面对 FHS V2.3 进行一些简要说明。

#### 8.2.1 顶层目录

整个根文件系统都是挂在根目录（/）下，FHS对顶层目录的要求和说明如表8.1所列。

表8.1 FHS 顶层目录

目 录	说 明
bin	基本的命令二进制文件（ <b>所有用户可用</b> ），里面不能再包含目录
boot	Boot Loader 静态文件
dev	设备文件
etc	系统配置文件，配置文件必须是静态文件，不能是二进制文件
home	用户 home 目录
lib	基本的共享库和内核模块
media	可移动介质的挂载点
mnt	临时的文件系统挂载点
opt	附加的应用程序软件包
root	root 用户目录（ <b>可选</b> ）
sbin	基本的系统命令二进制文件（ <b>仅 root 用户可用</b> ）
srv	系统服务的一些数据
tmp	临时文件



续上表

目 录	说 明
usr	该目录有二级标准
var	可变数据

### 8.2.2 “/usr” 目录

“/usr” 目录包含了系统很大一部分内容，对其中的目录结构FHS也有相应地规定，如表 8.2 所列。

表8.2 /usr 目录

目 录	说 明
bin	大部分的用户命令
include	C 程序所需要包含的头文件
lib	库文件
locale	本地层次（ <b>安装完毕后为空</b> ）
sbin	不是至关重要的系统命令（ <b>仅 root 用户可用</b> ）
share	独立数据
X11R6	X-Window 系统（ <b>可选</b> ）
games	游戏和教育相关的二进制文件
src	源代码（ <b>可选</b> ）

FHS 标准规定的相当详细，对一些目录以及深层子目录都做了详细的规定，对目录里面的文件也做了规定，更详细的情况请参考 FHS 文档。事实上，遵循 FHS 标准的 Linux 发行版也只是大体上遵循这个规范，不同发行商往往都会有一些改动。特别是在嵌入式领域，很多可选目录都可以没有，或者会增加一些新的目录。

### 8.3 BusyBox

构建根文件系统就是根据系统需要，将必要的命令或者文件集合起来，根据 FHS 的要求进行存放。根文件系统需要的命令、库文件等，可以考虑单个逐个编译得到，这是传统的 LFS（Linux From Scratch）方式。这种方式操作起来比较复杂。后来 busybox 的出现改变了这一局面。

BusyBox 项目是一个遵循 GPLv2 的开源项目（[项目主页 www.busybox.net](http://www.busybox.net)），它将 100 多种 UNIX 命令和工具集成到一个可执行文件中，而这个可执行文件只有 1M 左右，被称为“嵌入式 Linux 的瑞士军刀”。BusyBox 的出现极大的简化了 Linux 根文件系统的构建，只需对 BusyBox 进行配置和编译/交叉编译，即可得到包含了 Linux 系统运行的基本命令和库文件等。

### 8.4 NFS根文件系统

Linux 内核支持从网络加载根文件系统，这对嵌入式 Linux 开发非常有用，特别是在系统开发初期。将根文件系统放在主机上，方便文件系统的调整，也不用考虑文件系统的体积，等系统开发完毕后再进行裁剪即可。

使用 NFS 根文件系统，首先需要内核中网卡驱动已经正常工作，并且在内核已经支持网络并配置了 NFS 根文件系统支持，同时将内核参数设置为通过 NFS 启动。设置内核 NFS 启动的参数一般格式为：

```
root=/dev/nfs rw console=$(consolectf) nfsroot=$(serverip):$(rootpath)
```



```
ip=$(ipaddr):$(serverip):$(gatewayip):$(netmask):$(hostname)::off
```

其中个参数的意义如下：

consolecfg——调试串口配置；  
serverip ——NFS 服务器 IP；  
ipaddr ——本机 IP（**目标系统 IP**）；  
gateway ——网关；  
netmask ——子网掩码；  
hostname——目标板的主机名；  
rootpath ——主机 NFS 根文件系统路径。

如“使用调试串口屏 ttyAM0，NFS 服务器 IP 为 192.168.12.123，NFS 根文件系统路径为/nfsroot，目标板 IP 为 192.168.12.124”的启动参数配置为：

```
root=/dev/nfs rw console=ttyAM0,115200 nfsroot=192.168.12.123:/nfsroot  
ip=192.168.12.124:192.168.12.123:192.168.12.1:255.255.255.0:epc.zlgmcu.com:eth0:off
```

把光盘附带的 rootfs\_zlg.tar.bz2 复制 Linux 主机上的/nfsroot 目录，然后解压该包：

```
$ tar -jxvf rootfs_zlg.tar.bz2
```

解压完成后，将得到/nfsroot/rootfs 目录，这个目录将用作 NFS 根文件系统的目录。重启动 EasyARM-iMX283 进入 U-Boot 的命令行（参考“[EasyARM-iMX283 的 Boot Loader](#)”章节的“**U-Boot 基本命令**”描述进入 U-Boot 命令的方法），设置 bootargs 环境变量：

```
MX28 U-Boot > setenv bootargs 'root=/dev/nfs rw console=ttyAM0,115200n8  
nfsroot=192.168.12.123:/nfsroot/rootfs  
ip=192.168.12.124:192.168.12.123:192.168.12.1:255.255.255.0:epc.zlgmcu.com:eth0:off'  
MX28 U-Boot > saveenv #保存环境变量  
MX28 U-Boot > reset #重启 EasyARM-iMX283
```

系统重启动完成后，就进入 Linux 主机上的/nfsroot/rootfs 所在的文件系统。

## 8.5 生成文件系统映像

### 8.5.1 生成rootfs.ubifs固件

系统开发后期，对根文件系统进行裁剪后，最终需要进行固化。根文件系统映像用什么样的文件系统，需要根据实际情况进行选择。目前内核可支持的文件系统为 UBIFS。在 Linux 内核源码中配备有 UBIFS 文件系统的实现代码。

针对 EasyARM-iMX283 开发板制作 UBIFS 根文件系统映像可以按下面的方法进行。

**注意 EasyARM-iMX283 根文件所在分区的参数：分区大小为 240MB；页大小为 2048 字节（2KB）；擦除块大小为 128KB。**

(1) 准备 UBIFS 文件系统映像制作工具

制作 UBIFS 文件系统映像，需要使用 mkfs.ubifs 和 ubinize 命令。在光盘中的“3.Linux\4.开发示例\4、文件系统”目录下有 mkfs.ubifs、ubinize 程序文件。请把这两个程序文件复制到 Linux 主机下的/usr/sbin/目录下。然后添加这两个程序的可执行权限：

```
vmuser@Linux-host ~$ sudo chmod 777 /usr/sbin/mkfs.ubifs  
vmuser@Linux-host ~$ sudo chmod 777 /usr/sbin/ubinize
```

(2) 准备根文件系统和配置文件



根文件系统可以用户自己制作，也可以参考光盘上提供的根文件系统。如果使用光盘提供的根文件系统，请把光盘上“3.Linux\4.开发示例\4、文件系统”目录下的 rootfs.tar.bz2、build\_rootfs 和 ubinize.cfg 文件复制到 Linux 主机中自己的工作目录的同一个目录之下。然后执行下面命令：

```
vmuser@Linux-host ~$ sudo tar -jxvf rootfs.tar.bz2          #解压 rootfs.tar.bz2 文件  
vmuser@Linux-host ~$ sudo chmod 777 build_rootfs           #给 build_rootfs 文件添加可执行权限
```

### (3) 生成 ubi 根文件系统

build\_rootfs 文件是脚本程序文件，其内容是：

```
mkfs.ubifs -r rootfs -m 2048 -e 126976 -c 1900 -o ubifs.img  
ubinize -o ubi.img -m 2048 -p 128KiB -s 512 ubinize.cfg  
mv ubifs.img rootfs.ubifs
```

执行该脚本程序就能生成根文件系统镜像：

```
vmuser@Linux-host ~$ ./ build_rootfs
```

这时所得到的 rootfs.ubifs 文件就是所需的 ubi 根文件系统映像。把该 rootfs.ubifs 复制 tftp 服务器的根目录，就可以供 EasyARM-iMX283 更新文件系统。

### 8.5.2 生成rootfs.tar.bz2 固件

把光盘文件中的“3.Linux\4.开发示例\4、文件系统”目录的 rootfs.tar.bz2 文件复制到 Linux 主机的工作目录后。执行下面命令：

```
vmuser@Linux-host ~$ tar -jxcvf rootfs.tar.bz2      # 解压 rootfs.tar.bz2 文件  
vmuser@Linux-host ~$ cd rootfs                      # 进入 rootfs  
vmuser@Linux-host ~$ tar -cjvf rootfs.tar.bz2 *     # 生成 rootfs.tar.bz2 固件
```

rootfs.tar.bz2 固件生成后，若需要使用 USB 方式烧写到 EasyARM-iMX283 的 NAND Flash，可以替换到 MfgTool（[在光盘文件中的“3.Linux\5.Linux 系统恢复\USB 烧写方案”位置](#)）程序的“Profiles\MX28 Linux Update\OS Firmware\files”目录下；若需要使用 TF 卡方式烧写，可以替换到 i.MX283\_for\_kernelsb 或 i.MX283\_for\_ubootsb 目录中（[在光盘文件中的“3.Linux\5.Linux 系统恢复\SD 烧写方案”位置](#)）。

## 8.6 开机启动设置

当用户需要EasyARM-iMX283 在开机启动后就运行指定的应用程序或指令时，可以通过vi命令编辑/etc/rc.d/init.d/start\_userapp，将要执行的指令添加到里面。若用户有一个hellow的程序放在/home/目录中，那么设置hellow程序开机启动的方法如程序清单8.1红色部分所示。

程序清单8.1 用户启动文件

```
#!/bin/sh  
#you can add your app start_command three  
/home/hellow  
#start qt command,you can delete it 下面是启动 QT 界面的指令，若用户不需要启动 QT，可以直接删除  
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 &
```

如果程序是一个阻塞程序（**程序被运行后不会退出或返回**），则可能会导致位于其后的指令或程序无法得到执行，并且始终占用串口终端，造成其他程序（**比如 Shell**）无法通过串口终端与用户交互。对于此类应用程序，可以在其后面添加“**&**”（注意：是**“空格” + “&”**符号）让其在后台运行，如下所示：

```
/home/hellow &
```

**注：若因自启动的阻塞程序造成串口终端无法交互（如启动设置时漏了添加“&”），用户可以通过更新文件系统（uboot 下通过 tftp 服务器进行更新，更新命令为 run uprootfs）或重新安装目标板上的 Linux 系统（含文件系统）来解决。**



## 9. 嵌入式Linux Qt编程指南

本章主要介绍如何使用 EasyARM-iMX283 学习套件进行嵌入式 Linux 的 Qt 库的图形程序开发。

### 9.1 背景知识

在阅读本章前，希望读者对下面所列举的知识点有一定的了解，这样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。

### 9.2 Qt介绍

#### 9.2.1 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，已被nokia公司收购，在nokia的推动下，Qt的发展非常快速，版本不断更新。目前最新的Qt主版本为 4.8.1，所支持的平台如图9.1所示：



图9.1 Qt 支持的平台

#### 9.2.2 Qt/E简介

嵌入式 Linux 发行版本上的 Qt 属于 Qt 的 Embedded Linux 分支平台。这个分支平台一般被简称为 Qt/E。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。

Qt/E所面对的硬件平台较多，当开发人员需要在某硬件平台上移植Qt/E时，需要下载Qt源代码，利用交叉编译器编译出Qt库。接着需要将Qt库复制两份，一份放置在开发主机上，供编译使用；一份放在目标板上，供运行时动态加载使用。流程如图9.2所示：

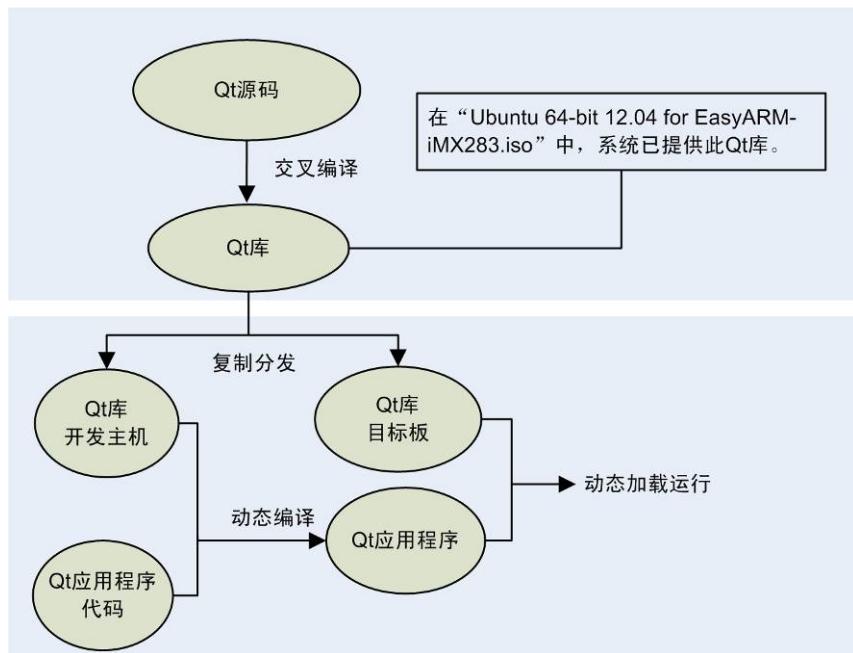


图9.2 Qt/E 编程图示

EasyARM-iMX283 提供已交叉编译好的 Qt 库 ([Qt-4.8.0](#))，用户无需编译，系统已经将 Qt 库文件集成到了交叉编译器中。用户将得到此库的两份拷贝，一份内嵌在交叉编译工具链中，供编译时链接使用。一份内嵌在目标板文件系统中，放置在系统库目录下，供 Qt 程序运行时动态加载使用。

### 9.3 编译环境的搭建

光盘资料附带的交叉编译器中已经集成了QT库 ([Qt-4.8.0](#))，用户只需按4.7章节述的方法安装交叉编译器即可。

### 9.4 hellow程序开发

#### 9.4.1 编译hellow程序

下面介绍如何开发一个简单的hellow程序，其流程如图9.3所示。

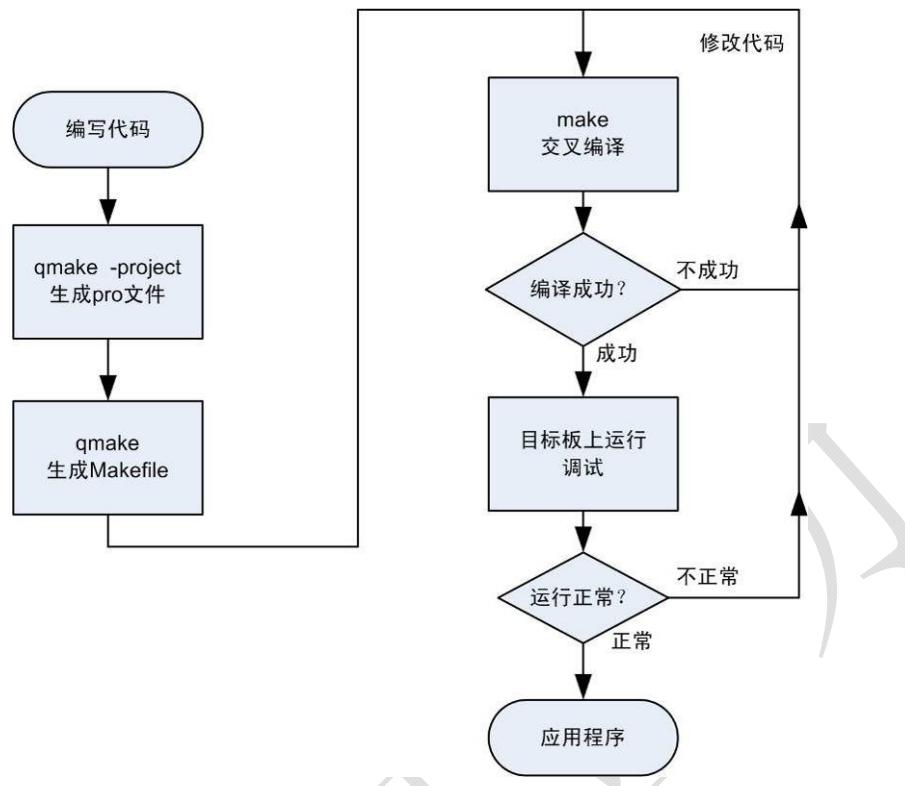


图9.3 hello 开发流程图

hellow.cpp程序代码如程序清单9.1所示。

程序清单9.1 hello 程序代码

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

将 hellow.cpp 拷贝至~/EasyARM-iMX283/hellow 目录下, 运行以下命令将生成 hellow.pro 文件:

```
vmuser@Linux-host:~/EasyARM-iMX283/hellow$ qmake -project
```

hellow.pro 文件描述整个工程所包含的源码及相应的资源文件。qmake 是 Qt 中用来管理工程的项目工具, 有关 qmake 与 pro 文件的相关信息将在下一节做详细的介绍。

执行 qmake 命令, 将根据上一步的 pro 文件, 生成 makefile 文件。

```
vmuser@Linux-host:~/EasyARM-iMX283/hellow$ qmake
```

根据 makefile 文件执行 make 命令则可以编译出可执行程序。以后需要再编译时, 也只需执行最后一步, 即 make 命令。

```
vmuser@Linux-host:~/EasyARM-iMX283/hellow$ make
```

经过上述步骤, 可以在 hellow 目录下见到 hellow 文件, 这个文件就是可执行的 Qt 程序。



### 9.4.2 在目标板上运行hellow程序

#### 1. 作为普通应用程序启动

把 hellow 文件通过 nfs 或其它方式下载到 EasyARM-iMX283 的/root/目录下，然后通过串口终端登录开发套件的 Linux 系统，并通过如下指令即可启动该程序 (**J7 (DUART)** 需要通过串口线与电脑相连)。

```
root@EasyARM-iMX283 ~# ./hellow
```

hellow程序运行后，如图9.4所示。



图9.4 运行 hellow 程序

#### 2. 作为窗口服务器启动

如果用户的 Linux 系统上电启动后没有加载任何 qt 程序(开发套件演示的 Linux 系统上电后会自动启动 zylauncher 图形框架程序 framework，该程序是一个被用作窗口服务器的 qt 程序)，则运行 hellow 程序的命令需要加上-qws 参数，具体指令如下：

```
root@EasyARM-iMX283 ~# ./hellow -qws
```

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

此外，在启动作为窗口服务器的 Qt 程序前，还需要先设定其鼠标设备。通过修改环境变量 QWS\_MOUSE\_PROTO 的值可以指定 Qt 的鼠标设备，示例命令如下：

```
export QWS_MOUSE_PROTO=[Tslib:/dev/input/event%d] [LinuxInput:/dev/input/event%d]
```

上述命令中 Tslib 字段用于指定了触摸屏设备文件，LinuxInput 字段用于指定鼠标设备文件，“[]”符号表示该字段为可选字段，实际命令中不需要输入该符号。Qt 支持多个鼠标设备，即允许 QWS\_MOUSE\_PROTO 环境变量的值由多个可选段组成，不同字段之间用空格隔开。

正常情况下，输入设备文件均位于 /dev/input 目录下，但具体的设备文件名（如 event0）对应的是触摸屏还是鼠标，则可能因实际应用而各有差异。

如程序清单9.2所示，是一段与“start\_zylauncher”文件内容相似的启动脚本示例代码，



该脚本用于自动探测接入开发套件的鼠标设备是触摸屏还是普通鼠标，并根据实际探测的结果自动设置Qt程序执行所需的环境变量，然后启动首个Qt程序。

#### 程序清单9.2 启动脚本代码

```
#!/bin/sh
SCRIPT_PATH=`cd "$(dirname "$0")"; pwd`
cd "$SCRIPT_PATH"

devs_list=`ls /dev/input/event*`
has_ts=0
QWS_MOUSE_PROTO=""
QWS_KEYBOARD=""
for dev in $devs_list
do
./input_type "$dev"
case $? in
1) QWS_MOUSE_PROTO="$QWS_MOUSE_PROTO ""Tslib:$dev"
has_ts=1
;;
2) QWS_MOUSE_PROTO="$QWS_MOUSE_PROTO ""LinuxInput:$dev"
;;
3) QWS_KEYBOARD="$QWS_KEYBOARD ""LinuxInput:$dev"
;;
*) ;;
esac
done

# delete space in head an tail
QWS_MOUSE_PROTO=`echo $QWS_MOUSE_PROTO` 
QWS_KEYBOARD=`echo $QWS_KEYBOARD` 

echo "[QWS_MOUSE_PROTO=$QWS_MOUSE_PROTO]"
echo "[QWS_KEYBOARD=$QWS_KEYBOARD]"

export "QWS_MOUSE_PROTO=$QWS_MOUSE_PROTO"
export "QWS_KEYBOARD=$QWS_KEYBOARD"

# 判断触摸屏校准文件是否存在
#if [ "$has_ts" == "1" -a ! -f /etc/pointercal ]; then
#    ts_calibrate
#fi

export TSLIB_CALIBFILE=/etc/pointercal
export TSLIB_CONFFILE=/etc/ts.conf
```

```
export TSLIB_PLUGINDIR=/usr/lib/ts

export LDPATH=/usr/lib
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$LDPATH
export QT_QWS_FONTDIR=$LDPATH/fonts
export QT_PLUGIN_PATH=$LDPATH/qt/plugins:$SCRIPT_PATH/framework/plugins
export POINTERCAL_FILE=$SCRIPT_PATH/framework/qt_pointercal

./fb_set /dev/fb0

# 这里可以修改成其他需要启动的首个 Qt 程序
./hellow -qws
```

将脚本代码保存为 qtLauncher 文件，并拷贝至开发套件 Linux 系统的“/usr/share/zhiyuan/zylauncher/”目录下，同时修改开发套件中 Linux 系统的启动文件由原来的 start\_zylauncher 更改为 qtLauncher，即修改根文件系统“/etc/rc.d/init.d/”目录下的 start\_userapp 文件，在串口终端中利用 vi 编辑器修改该文件的命令如下：

```
root@EasyARM-iMX283 ~# vi /etc/rc.d/init.d/start_userapp
```

用vi编辑器修改完开发套件根文件系统的start\_userapp文件后如图9.5所示。

图9.5 修改 start\_userapp 文件

注意：由于大多数串口终端在处理 vi 编辑器相关指令时容易出错，建议在上位机中修改该文件，然后通过文件拷贝方式覆盖开发套件上的 start\_userapp 文件，或通过挂载 NFS 根文件系统的方式进行测试。

在本例中，作为窗口服务器的hellow程序启动成功后，界面如图9.6所示。

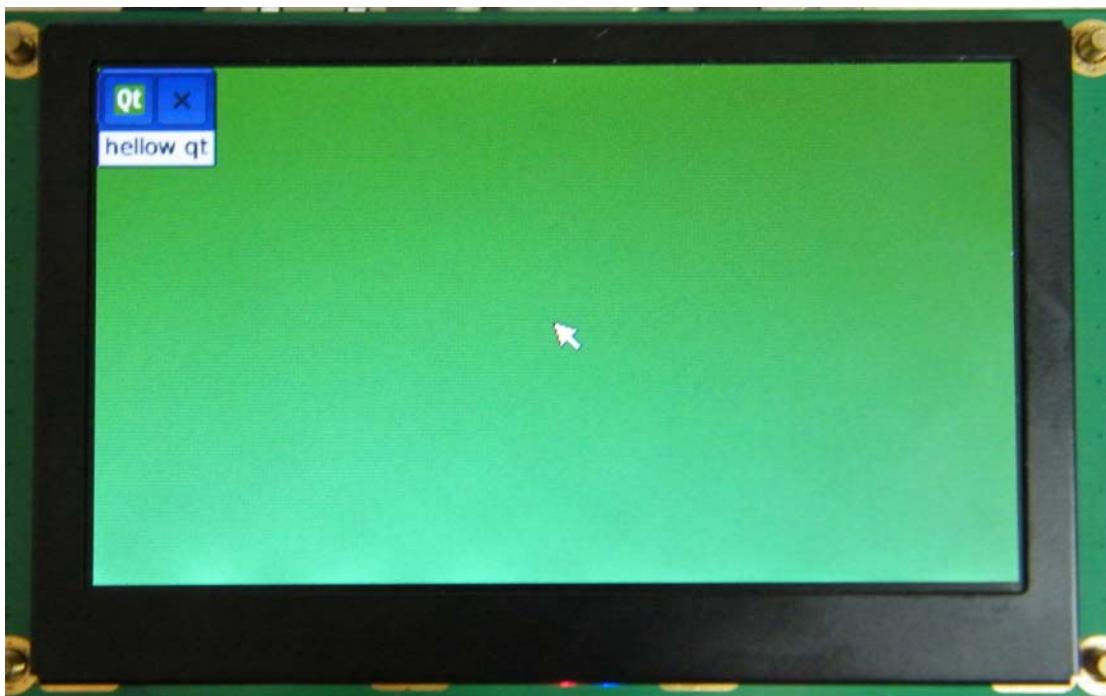


图9.6 hellow 程序运行界面

## 9.5 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，下来就简单介绍“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。

### 9.5.1 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。

假设工程项目中有如下源代码文件：

```
hello.cpp
```



```
hello.h
```

```
main.cpp
```

首先，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：

```
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：

```
qmake
```

在生成 Makefile 文件后，即可使用 make 命令进行编译。

### 9.5.2 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 desinger 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。

其中, CONFIG 变量常用的参数如下:

- 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。

## 9.6 Qt编程简单入门

### 9.6.1 例程讲解

在下面的描述中, 将说明在一个窗口中如何排列多个控件, 并学习如何利用 signal 和 slot (信号与槽) 使控件同步。

程序要求用户通过 slider 输入年龄, 并利用 lcd 显示年龄。程序中使用了三个控件: QLCDNumber, QSlider 和 QWidget。QWidget 是这个程序的主窗口。QLCDNumber 和 QSlider 被放在 QWidget 中, 所以它们是 QWidget 的 children。反过来, 也可以称 QWidget 是 QLCDNumber 和 QSlider 的 parent。QWidget 没有 parent, 因为它是程序的顶层窗口。在 QWidget 及其子类的构造函数中, 都有一个 QWidget\*参数, 用来指定它们的父控件。

源代码如程序清单9.3所示:

程序清单9.3 Qt 例程代码

```
#include <QApplication>
#include <QHBoxLayout>
#include <QSlider>
#include <QLCDNumber>
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QWidget *window = new QWidget;
    window->setWindowTitle("Enter Your Age");
    QLCDNumber * lcd = new QLCDNumber;
    QSlider *slider = new QSlider(Qt::Horizontal);
    slider->setRange(0, 99);
    QObject::connect(slider, SIGNAL(valueChanged(int)),
                     lcd, SLOT(display(int)));
    slider->setValue(35);
    QHBoxLayout *layout = new QHBoxLayout;
    layout->addWidget(lcd);
    layout->addWidget(slider);
}
```



```
window->setLayout(layout);
window->show();
return app.exec();
}
```

- 第 8, 9 行建立程序的主窗口控件，设置标题。
- 第 10 到 12 行创建主窗口的 children，并设置允许值的范围。
- 第 13 到第 14 行是 lcd 和 slider 的连接，以使 lcd 能同步显示 slider 所指示的值。当 slider 的值发生变化时，会发出 valueChanged(int)信号，lcd 的 display (int)函数就会相应地设置一个新值。
- 第 15 行将 slider 的值设置为 35，这时 slider 发出 valueChanged(int)信号，int 的参数值为 35，这个参数传递给 lcd 的 display(int)函数，将 lcd 的值也设置为 35。
- 在第 17 至 18 行，使用了一个布局管理器排列 lcd 和 slider 控件。布局管理器能够根据需要确定控件的大小和位置。Qt 有三个主要的布局管理器：
  - ◆ QBoxLayout，水平排列控件；
  - ◆ QVBoxLayout，垂直排列控件；
  - ◆ QGridLayout，按矩阵方式排列控件。
- 第 19 行，QWidget::setLayout()把这个布局管理器放在 window 上。这个语句将 lcd 和 slider 的 parent 设为 window，即布局管理器所在的控件。如果一个控件由布局管理器确定它的大小和位置，那么创建它的时候就不必指定一个明确的 parent 控件。

至此，虽然还没有看见 lcd 和 slider 控件的大小和位置，但它们已经水平排列好了，QHBoxLayout 能合理安排它们。

在 Qt 中建立用户界面就是这样简单灵活。程序员的任务就是实例化所需要的控件，按照需要设置它们的属性，把它们放到布局管理器中。同时利用 Qt 的信号和槽机制来管理用户的交互行为。

### 9.6.2 信号和槽机制

信号和槽是 Qt 编程的一个重要部分，这个机制可以在对象之间彼此并不了解的情况下将它们的行为联系起来。在上面的例程中，已经连接了信号和槽，发送了信号，触发槽函数的响应，下面将更深入介绍这个机制。

槽和普通的 c++ 成员函数很像。它们可以是虚函数 (**virtual**)，也可被重载 (**overload**)，可以是公有的 (**public**)，保护的 (**protective**)，也可是私有的 (**private**)，它们可以象任何 c++ 成员函数一样被调用，可以传递任何类型的参数。不同在于一个槽函数能和一个信号相连接，只要信号发出了，这个槽函数就会自动被调用。信号和槽函数间的连接通过 **connect** 实现。**connect** 函数语法如下：

```
connect(sender, SIGNAL(signal), receiver, SLOT(slot));
```

sender 和 receiver 是 QObject 对象 (QObject 是所有 Qt 对象的基类) 指针，signal 和 slot 是不带参数的函数原型。SIGNAL() 和 SLOT() 宏的作用是把他们转换成字符串。

在前面的例子中，已经连接了信号和槽，而在实际使用中还需要考虑如下一些规则：

- 1) 一个信号可以连接到多个槽：

```
connect(slider, SIGNAL(valueChanged(int)),spinBox, SLOT(setValue(int)));
connect(slider, SIGNAL(valueChanged(int)),this, SLOT(updateStatusBarIndicator(int)));
```



当信号发出后，槽函数都会被调用，但是调用的顺序是随机的，不确定的。

- 2) 多个信号可以连接到一个槽：

```
connect(lcd, SIGNAL(overflow()), this, SLOT(handleMathError()));  
connect(calculator, SIGNAL(divisionByZero()), this, SLOT(handleMathError()));
```

任何一个信号发出，槽函数都会执行。

- 3) 一个信号可以和另一个信号相连：

```
connect(lineEdit, SIGNAL(textChanged(const QString &)), this, SIGNAL(updateRecord(const QString &)));
```

第一个信号发出后，第二个信号也同时发送。除此之外，信号与信号连接上和信号和槽连接相同。

- 4) 连接可以被删除：

```
disconnect(lcd, SIGNAL(overflow()), this, SLOT(handleMathError()));
```

这个函数很少使用，一个对象删除后，Qt 自动删除这个对象的所有连接。

信号和槽函数必须有着相同的参数类型，这样信号和槽函数才能成功连接：

```
connect(ftp, SIGNAL(rawCommandReply(int, QString &)), this, SLOT(processReply(int, QString &)));
```

如果信号里的参数个数多于槽函数的参数，多余的参数被忽略：

```
connect(ftp, SIGNAL(rawCommandReply(int, const QString &)), this, SLOT(checkErrorCode(int)));
```

如果参数类型不匹配，或者信号和槽不存在，在 debug 状态时，Qt 会在运行期间给出警告。如果信号和槽连接时包含了参数的名字，Qt 将会给出警告。

本小节的例子，使用qmake -project， qmake， make可直接生成可执行程序，无需修改pro文件。其运行界面如图9.7所示：

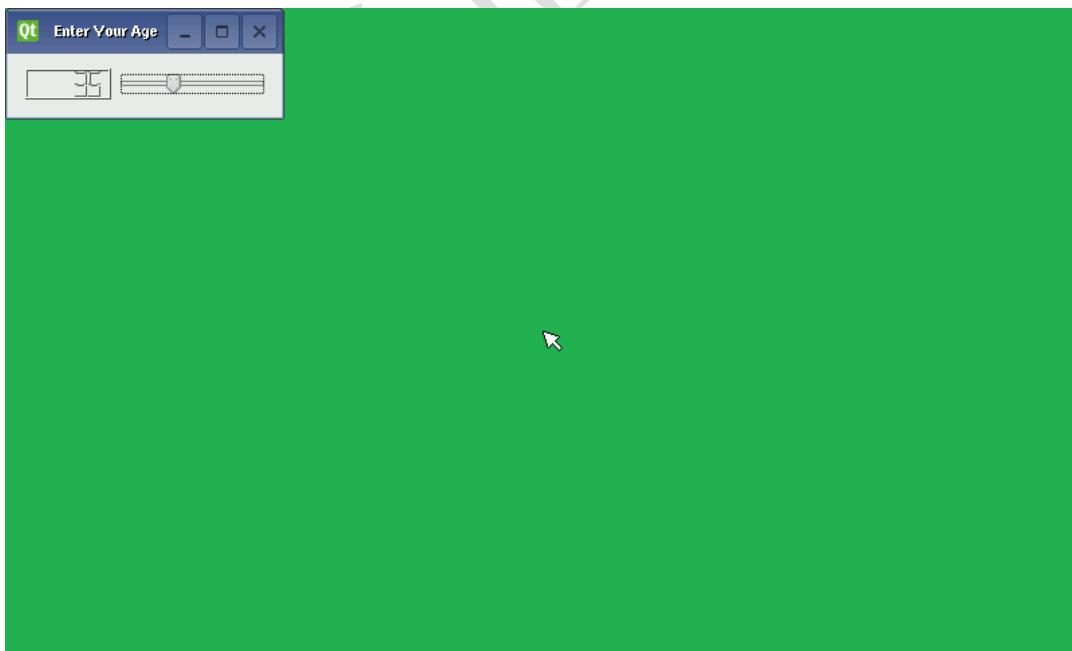


图9.7 Qt 例程界面

## 9.7 Qt SDK的使用

### 9.7.1 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 9.8 所示。

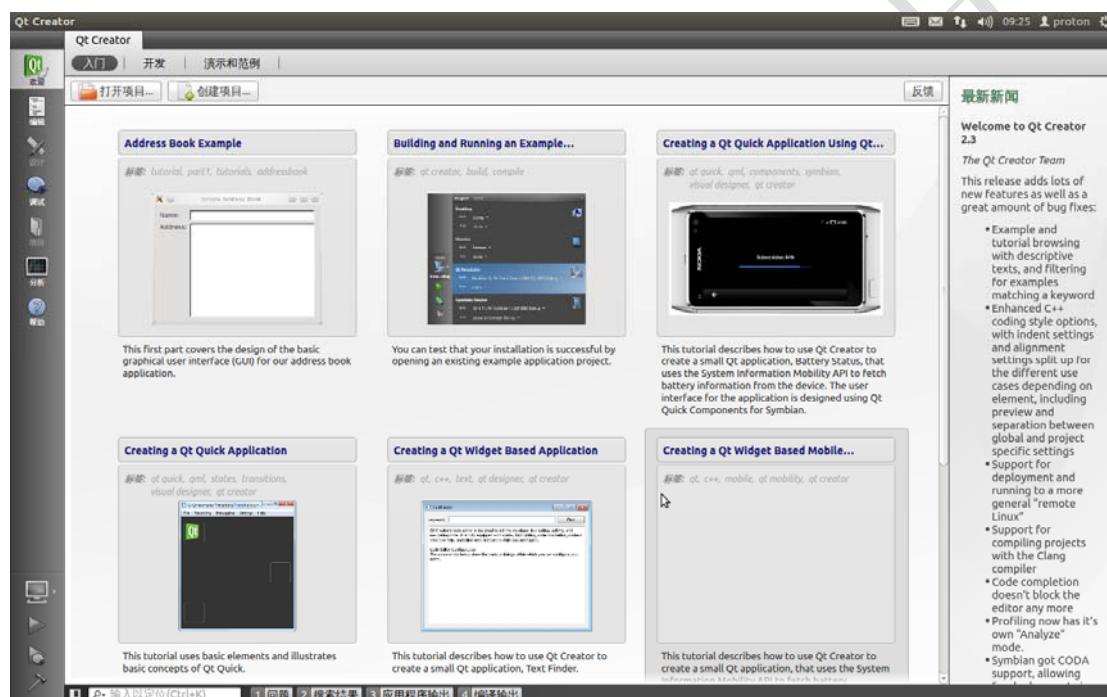


图 9.8 Qt Creator 主界面

### 9.7.2 Qt SDK 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。在 Linux 主机可以正常上网的条件下，先进行安装源的更新，否则可能导致 QT-SDK 安装失败。安装源更新命令如下：

```
vmuser@Linux-host:~$ sudo apt-get update
```

其执行过程如图 9.9 所示。



```
vmuser@Linux-host: ~
vmuser@Linux-host:~$ sudo apt-get update
[sudo] password for vmuser:
0% [正在连接 cn.archive.ubuntu.com] [正在连接 security.ubuntu.com] [正在]
```

图9.9 更新 Linux 安装源

安装源更新后，可以使用如下命令获取并安装 QT SDK：

```
vmuser@Linux-host:~$ sudo apt-get install qt-sdk
```

在QT SDK的安装过程中可能会出现如图9.10所示的警告窗口，这时只需要选中该窗口并按“回车”键即可。

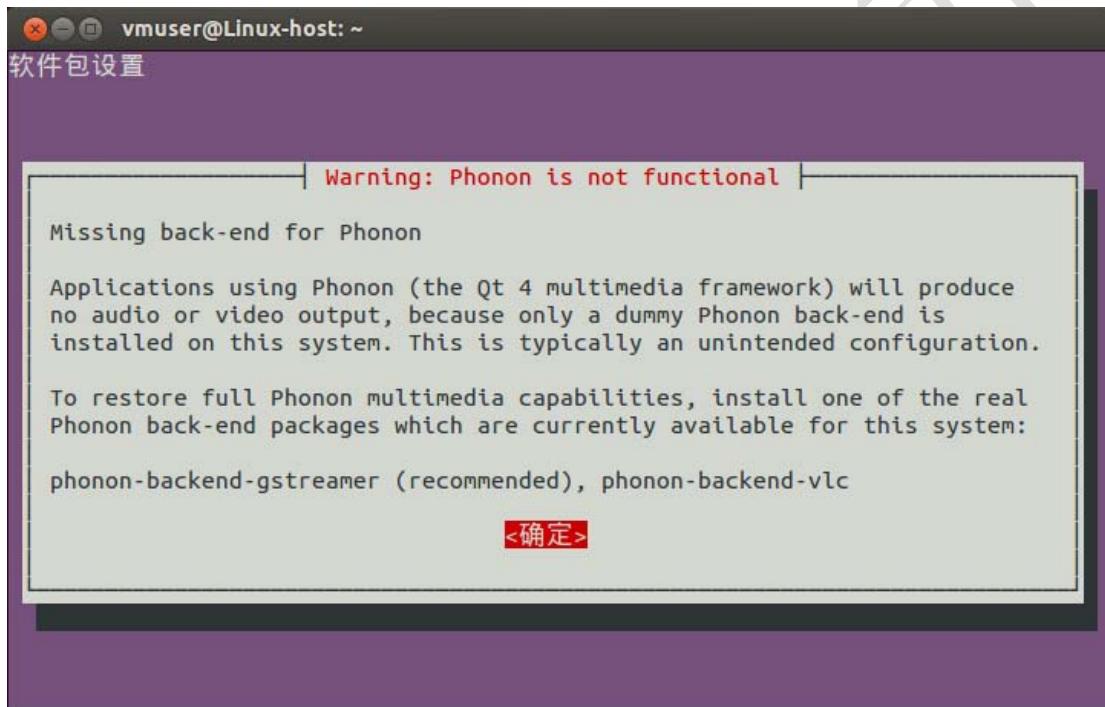


图9.10 QT SDK 安装过程中可能出现的警告窗口

当QT SDK安装完成后，终端显示如图9.11所示。



```
vmuser@Linux-host: ~
update-alternatives: 使用 /usr/bin/uic-qt4 来提供 /usr/bin/uic (uic), 于 自动模式 中。
update-alternatives: 警告: 跳过创建 /usr/share/man/man1/uic.1.gz 因为所关联文件 /usr/share/man/man1/uic-qt4.1.gz (位于链接组 uic) 不存在。
正在设置 phonon-backend-null (4:4.7.0really4.6.0-0ubuntu1) ...
正在设置 libphonon-dev (4:4.7.0really4.6.0-0ubuntu1) ...
正在设置 libqt4-declarative-gestures (4:4.8.1-0ubuntu4.8) ...
正在设置 libqt4-declarative-particles (4:4.8.1-0ubuntu4.8) ...
正在设置 libqt4-opengl-dev (4:4.8.1-0ubuntu4.8) ...
正在设置 libqtwebkit-dev (2.2.1-1ubuntu4) ...
正在设置 qt4-designer (4:4.8.1-0ubuntu4.8) ...
update-alternatives: 使用 /usr/bin/designer-qt4 来提供 /usr/bin/designer (designer), 于 自动模式 中。
正在设置 qt4-dev-tools (4:4.8.1-0ubuntu4.8) ...
update-alternatives: 使用 /usr/bin/assistant-qt4 来提供 /usr/bin/assistant (assistant), 于 自动模式 中。
update-alternatives: 使用 /usr/bin/linguist-qt4 来提供 /usr/bin/linguist (linguist), 于 自动模式 中。
正在设置 qt-sdk (2ubuntu3) ...
正在设置 qt4-demos (4:4.8.1-0ubuntu4.8) ...
正在设置 qt4-qmlviewer (4:4.8.1-0ubuntu4.8) ...
正在处理用于 libc-bin 的触发器...
ldconfig deferred processing now taking place
vmuser@Linux-host:~$
```

图9.11 完成 QT SDK 的安装

### 9.7.3 Qt Creator配置

QT SDK 安装完成后，通过如下命令可以启动 Qt Creator:

```
vmuser@Linux-host:~$ qtcreator
```

Qt Creator启动后界面如图9.8所示。如已安装交叉编译工具链，则此时系统中将同时存在两个Qt版本，一个用于桌面环境，一个用于嵌入式环境。因此首先需要设置Qt Creator所调用的Qt版本。

#### 1. 设置Qt版本

把鼠标光标移动到屏幕顶端，以显示隐藏的菜单栏，如图9.12所示。

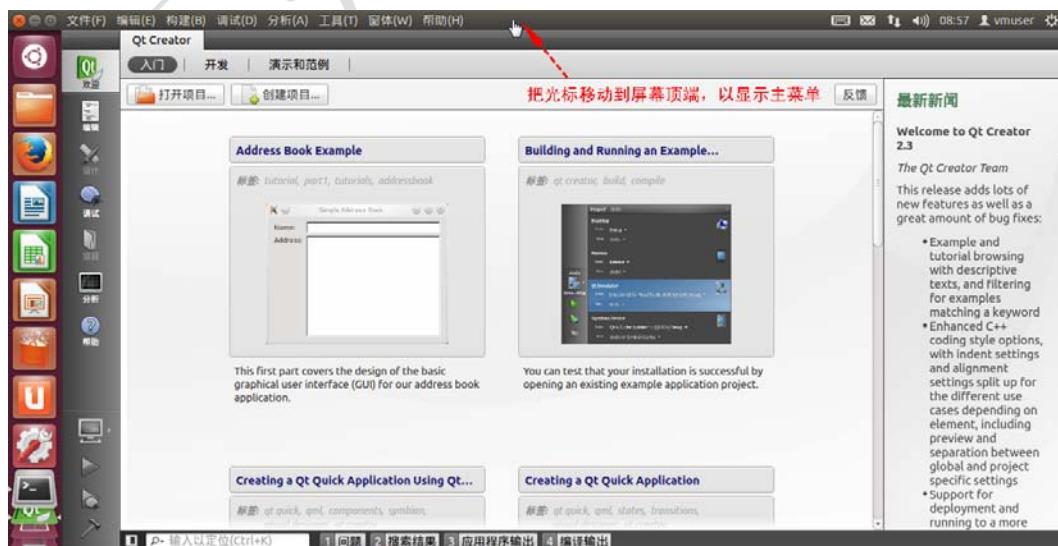


图9.12 显示 qtcreate 的主菜单

菜单栏显示出来后，点击“工具”->“选项”菜单，打开QT选项配置窗口，如图9.13所



示。

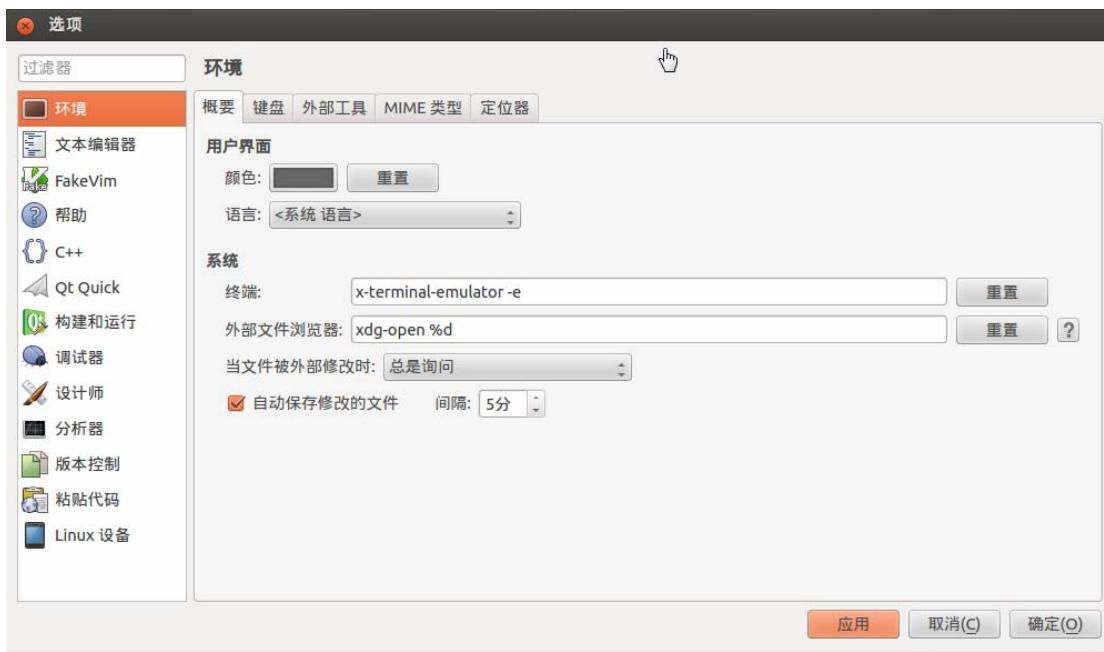


图9.13 选项界面

在打开的“选项”窗口上，点击左边的“构建和运行”选项，然后切换至“Qt 版本”标签。在这里 Qt Creator 会根据 PATH 环境变量自动探测 Qt 版本路径，若用于桌面环境及嵌入式 Linux 开发的 Qt 版本未被自动检测到，则需要手动进行添加。

点击右侧“添加”按钮，选择“usr/bin/qmake”路径下的qmake-qt4（**若该路径下没有 qmake-qt4 文件，则选择qmake文件**）文件以及“/opt/gcc-4.4.4-glibc-2.11.1-multilib-1.0/arm-fsl-linux-gnueabi/bin/”路径下的qmake文件，如图9.14及图9.15所示。



图9.14 添加 Qt 版本



图9.15 添加完 Qt 版本

## 2. 添加工具链

点击“工具链”标签，进入工具链设置。在这里已经自动检测出本地gcc的编译器，还需要将EasyARM-iMX283 的交叉编译器也添加上去，点击右边的“添加”按钮，选择“GCC”菜单，如图9.16所示。



图9.16 添加一个 GCC 项

再点击选中手动设置栏中的“GCC”选项，然后点击下边的“浏览”按钮，如图9.17所示。



图9.17 添加新的GCC项

在这里需要把“/opt/gcc-4.4.4-glibc-2.11.1-multilib-1.0/arm-fsl-linux-gnueabi/bin/”路径下的“arm-fsl-linux-gnueabi-g++”文件添加到“编译器路径”的输入框中，如图9.18所示。

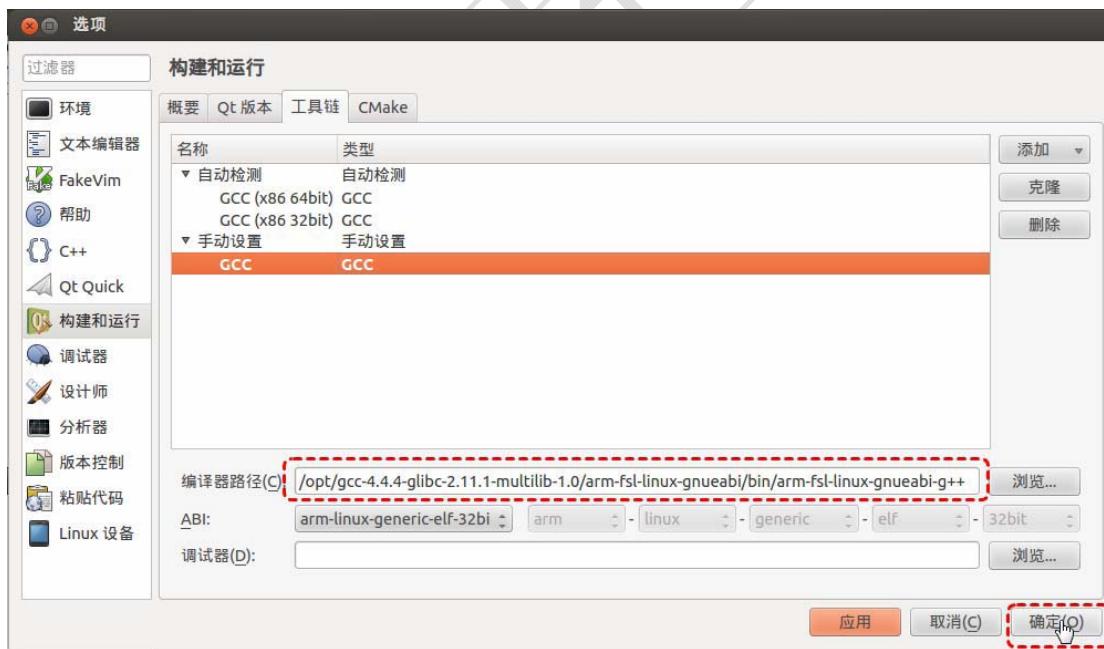


图9.18 添加交叉编译器路径

然后点击对话框中的“确定”按钮，关闭对话框。至此，Qt Creator 开发环境已配置完成。

#### 9.7.4 Qt Creator使用例程

接下来将以“Hello World”项目为例，介绍如何使用 Qt Creator 开发图形界面。

##### 1. 创建Hello World项目

启动Qt Creator，单击界面中“创建项目”按钮，在弹出的“新项目”窗口上先点击左侧的“Qt控件项目”，然后再选中右侧的“Qt Gui应用”模板，最后点击“选择”按钮，如图9.19所示。

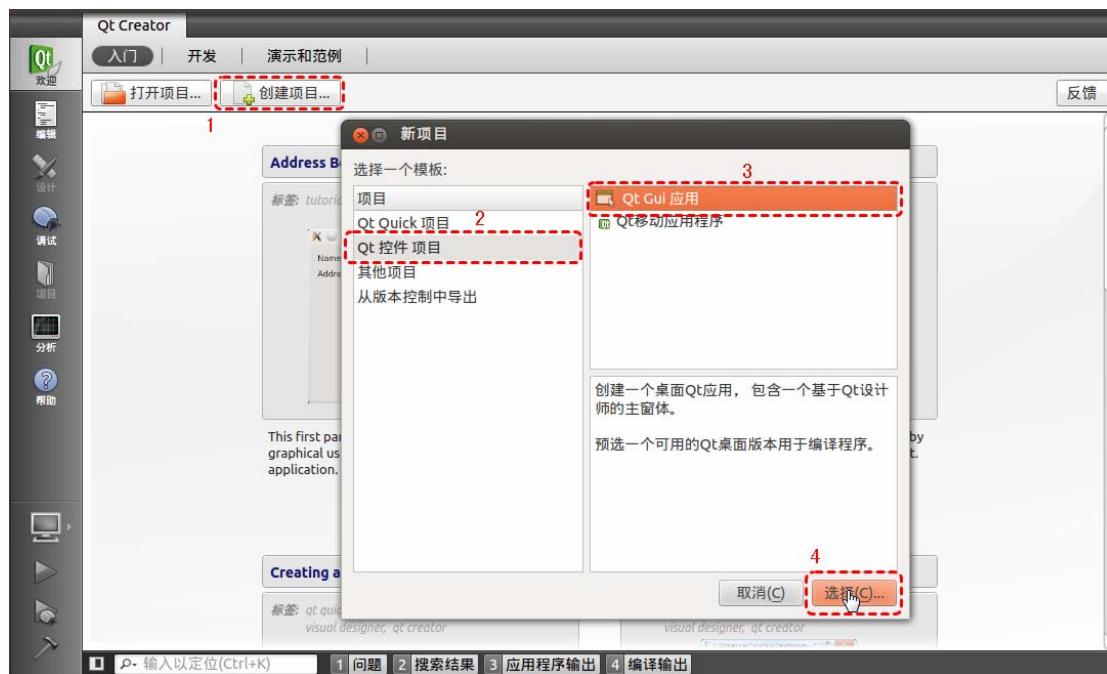


图9.19 新建 Gui 应用项目

点击“选择”按钮后，选择项目工程存储位置及设置工程名，参考图9.20所示设置即可（注意：创建路径将是项目工作目录，该路径下将会自动生成一个与工程名相同的文件夹以及编译生成的其他文件夹），然后点击“下一步”按钮。



图9.20 设置项目名称及创建路径

在本示例中，创建路径为“/home/vmuser/qt\_test/”目录。点击“下一步”按钮，进入“目标设置”界面。这里，选择按如图9.21所示的默认设置即可。



图9.21 目标设置界面

设置完成后，点“下一步”按钮进入“类信息”界面，如图9.22所示。

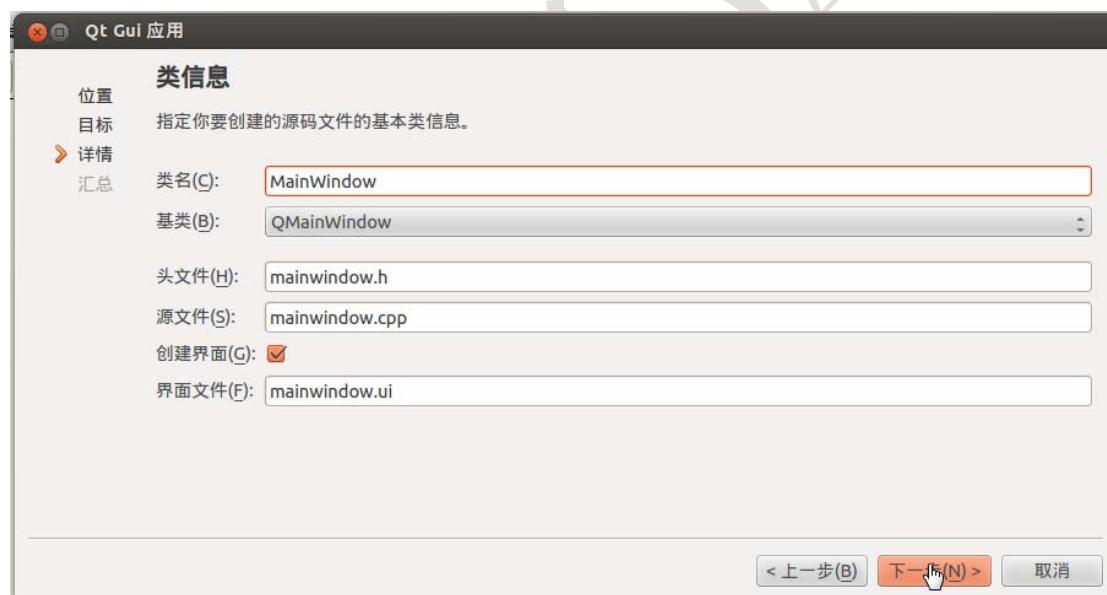


图9.22 类信息界面

在这里按默认设置即可，直接点击“下一步”按钮，进入“项目管理”界面，如图9.23所示。



图9.23 项目管理界面

单击“完成”后，可以看到创建好的qt\_test项目，在该项目中Qt Creator已经自动生成Qt程序所需的基本代码，并已自动打开了mainwindow.cpp文件，如图9.24所示。

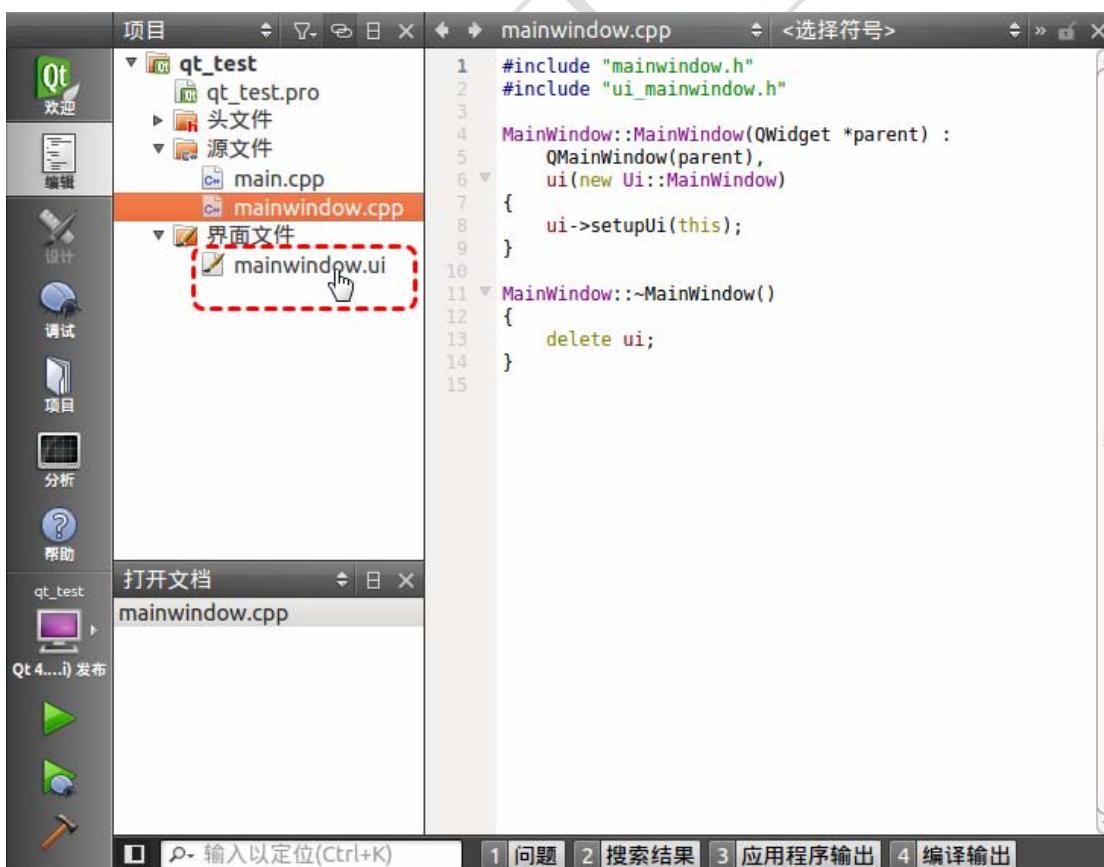


图9.24 mainwindow.cpp 界面

双击左侧项目栏中的mainwindow.ui文件则可以启动可视化编辑器，如图9.25所示。

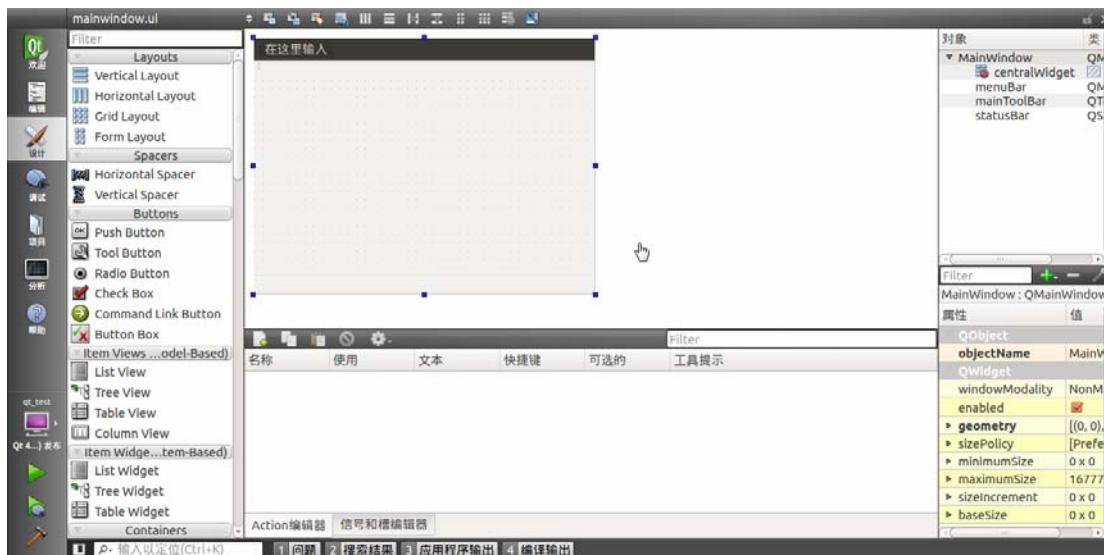


图9.25 可视化界面编辑器

在可视化编辑器中，通过拖动左侧控件栏中控件到窗体页面中，以所见即所得的方式设计应用界面。

拖动一个Label控件到窗体页面，然后双击该控件，设置Label上文字为“Hello World!”，，然后再用鼠标调整控件大小，使文字整体可见，如图9.26所示。

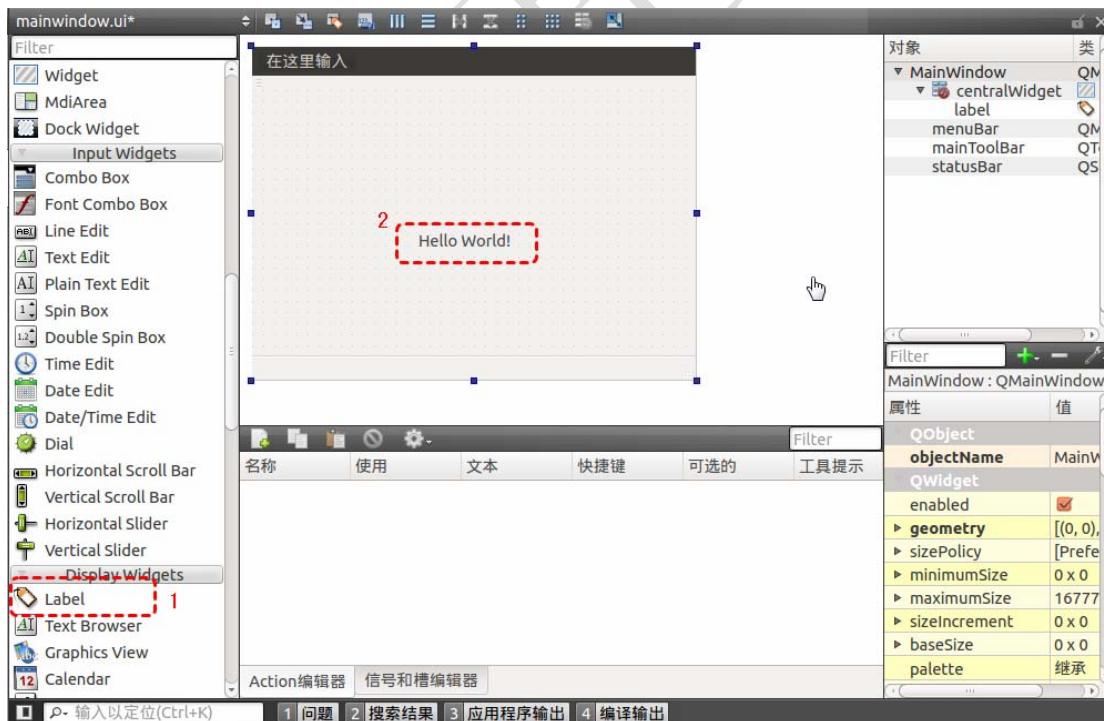


图9.26 hello world 程序界面

至此，一个简单的 Hello World 应用界面就设计完成了，下面将介绍如何编译该 Qt 项目。

## 2. 本地编译Qt应用



点击左侧栏上的发布按钮 “ ”，然后在“构建”一栏选择前面所设置的桌面版本的

Qt (如本例中的“Qt 4.8.1 在PATH (系统) 发布”), 如图9.27所示。

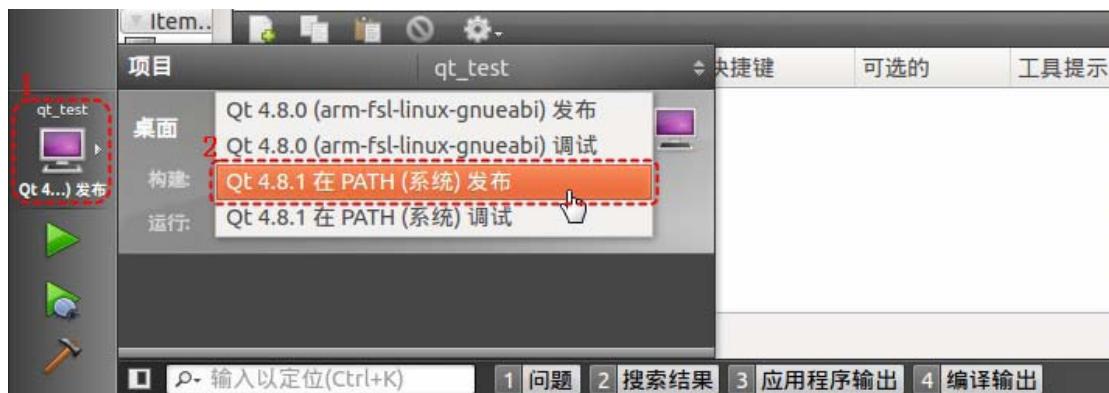


图9.27 选择 Qt 版本

选择了“Qt 4.8.1 (系统) 发布”的本地编译器后, 点击运行按钮“”对程序进行编译链接及运行。如编译无误, 将自动启动应用程序, 界面如图9.28所示。

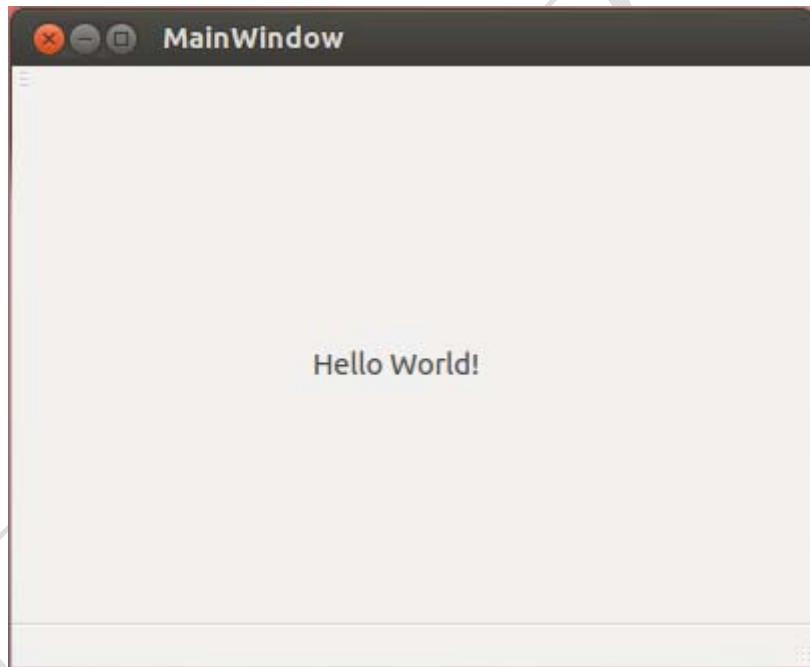


图9.28 hello world 界面

### 3. 交叉编译Qt应用

点击左侧栏上的发布按钮“”, 然后在“构建”一栏选择前面所设置的交叉编译器版本的Qt (如本例中的“Qt 4.8.0 (arm-fsl-linux-gnueabi) 发布”), 然后点击构建按钮“产品用户手册

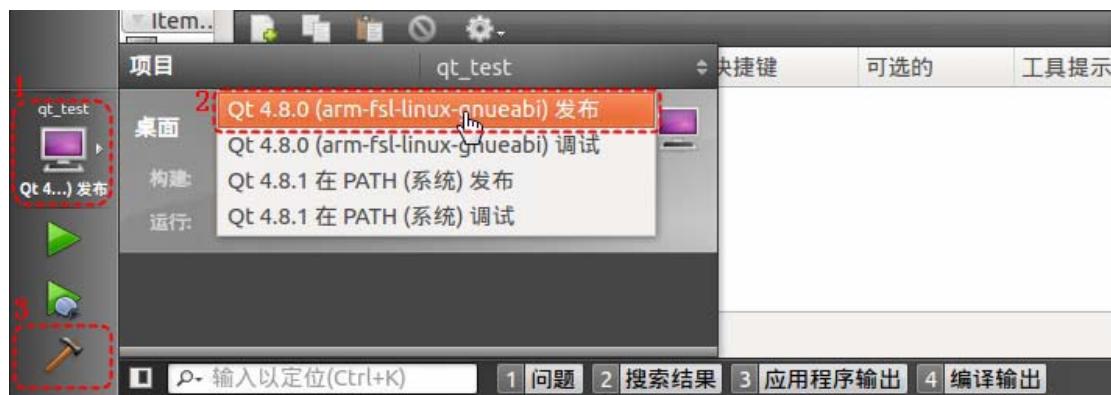


图9.29 选择交叉编进行构建译器

注意，由于这是交叉编译，所以编译出来的程序不能在本地 PC 机上运行或调试。因此不能点击 按钮运行程序，也不能点击 按钮调试程序。

编译完成后，在项目创建目录下自动生成“`qt_test-build-desktop-Qt_4_8_0_arm-fsl-linux-gnueabi`”目录。该目录下的内容如图9.30所示。

```
vmuser@Linux-host:~/qt_test/qt_test-build-desktop-Qt_4_8_0_arm-fsl-linux-gnueabi$ ls
main.o mainwindow.o Makefile moc_mainwindow.cpp moc_mainwindow.o qt_test ui_mainwindow.h
vmuser@Linux-host:~/qt_test/qt_test-build-desktop-Qt_4_8_0_arm-fsl-linux-gnueabi$ file qt_test
qt_test: ELF 32-bit LSB executable, ARM, version 1 (SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.26, not stripped
vmuser@Linux-host:~/qt_test/qt_test-build-desktop-Qt_4_8_0_arm-fsl-linux-gnueabi$
```

图9.30 交叉编译后的输出内容

在该目录中有一个 `qt_test` 的文件，经文件属性查看可知，这是一个 ARM 指令架构的可执行文件。把该文件通过 nfs 或其它方式下载到 EasyARM-iMX283 的 /root/ 目录下，然后通过串口终端登录开发套件的 Linux 系统，并通过如下指令即可启动该程序 (**J7 (DUART)** 需要通过串口线与电脑相连)。

```
root@EasyARM-iMX283 ~# ./qt_test
```

程序启动后如图9.31所示。

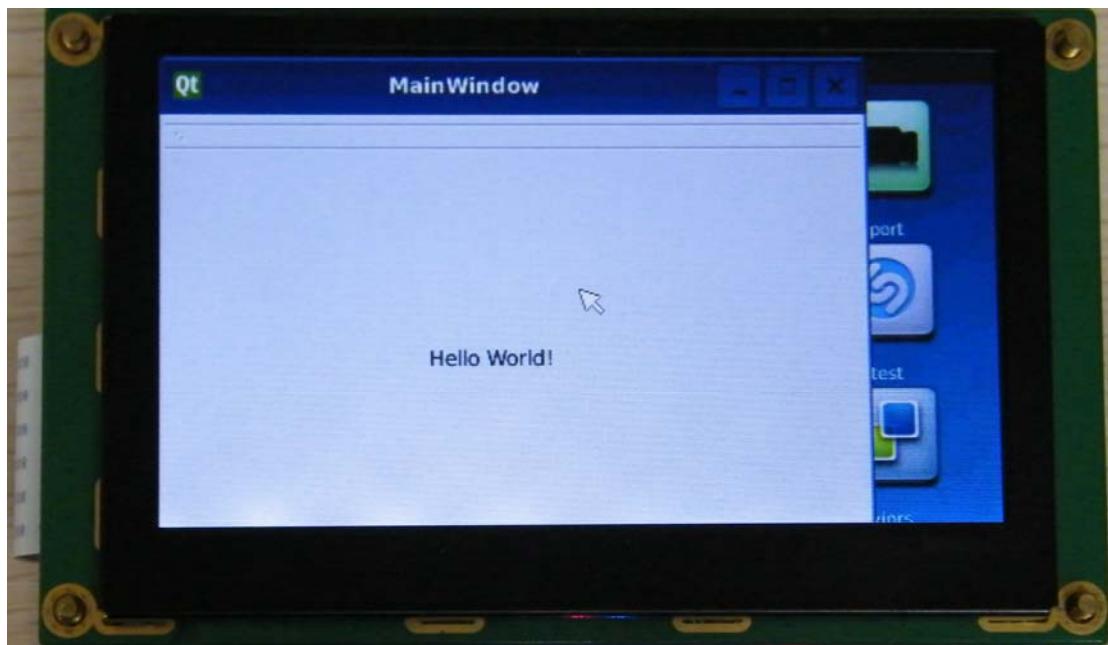


图9.31 Hello World 程序在开发套件上运行

## 9.8 zylauncher图形框架

zylauncher是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9.32。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9.32 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9.33



所示：

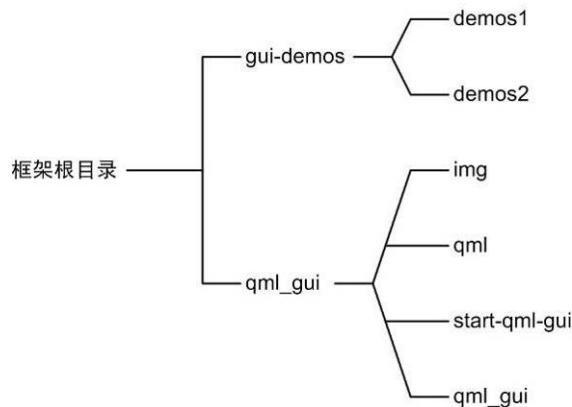


图9.33 zylauncher 框架结构

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9.32。`MainMenu.qml` 代码如程序清单9.4 所示。

程序清单9.4 `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);
        imgPath: "./img/switch.png";
        imgText : "fluidlauncher";
        execText : "../gui-demos/fluidlauncher/start-fluidlauncher"
    }
    ProButton {
        x:Logic.indexToX(1); y:Logic.indexToY(0);
    }
}
```



```
    imgPath: "../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"
}
}
```

接下来关注代码：

```
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”(**MenuBar 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);  
    imgPath: "../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。



## 10. 关于EasyARM-iMX287 开发套件

i.MX287 处理器向下兼容 i.MX283，并在 i.MX283 的基础上增加了一些其他外设，如 CAN、双以太网、SPI3、GPIO 等，而 EasyARM-iMX287 又使用了与 EasyARM-iMX283 完全相同的底板，所以 EasyARM-iMX287 将直接复用 EasyARM-iMX283 的相关软件开发资料（包含 U-Boot 及文件系统）。

本章将简单介绍 EasyARM-iMX287 的硬件资源，以及 Linux 下如何使用其相对于 EasyARM-iMX283 新增加的部分资源的应用，如 CAN、SPI3、双网口等。

### 10.1 EasyARM-iMX287 简介

EasyARM-iMX287 的基本接口分布及核心板位置如图1.2所示。

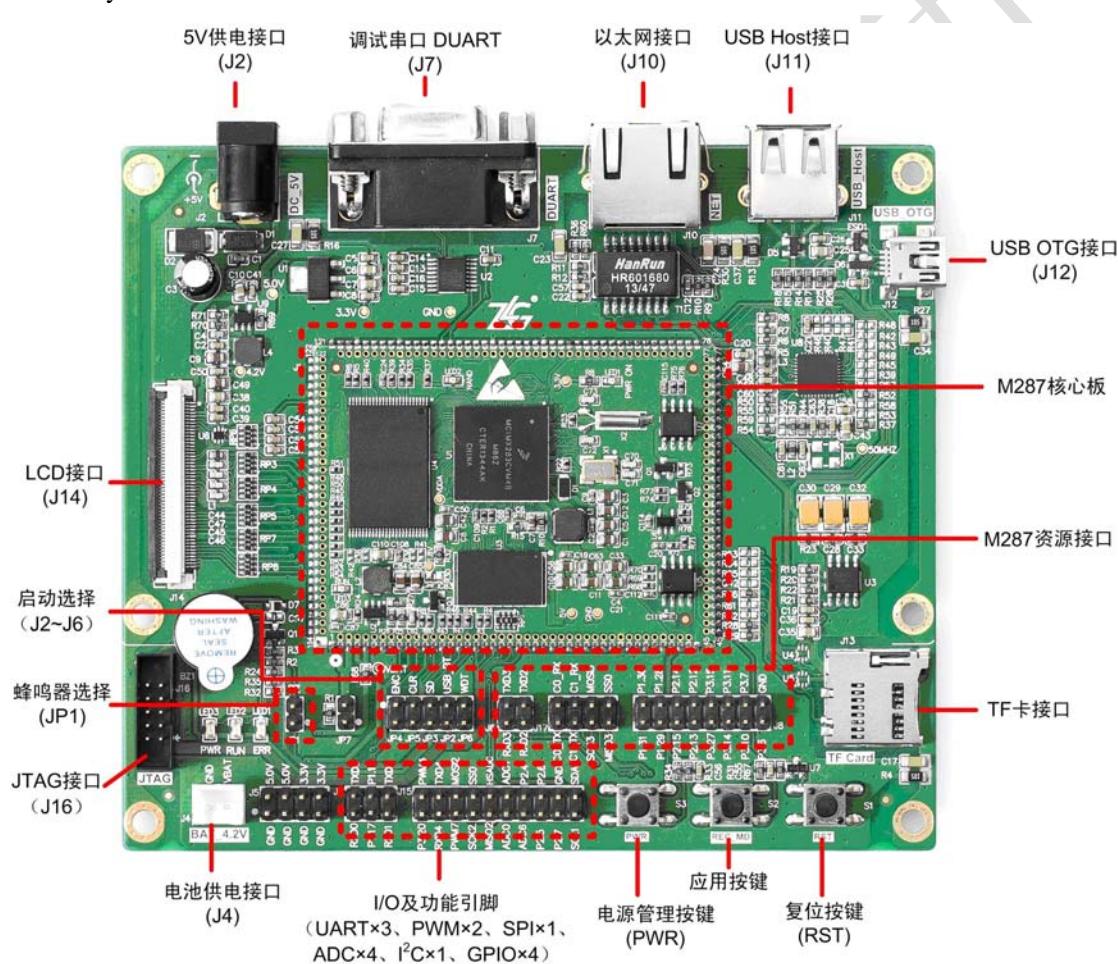


图10.1 EasyARM-iMX287 的基本接口分布及核心板位置

#### 10.1.1 核心板资源

EasyARM-iMX287 开发套件采用“M287 核心板+底板”的组合方式，M287 核心板资源如表1.1所示。



表10.1 M287 核心板资源

项目	参数	项目	参数
CPU	MCIMX287CVM4B	PWM	3 路
主频	454MHz	USB	1 路 HOST、1 路 OTG
			(USB2.0 高速, 480Mbps)
内存	64MB <sup>[1]</sup>	I <sup>2</sup> S	—
FLASH	128MB <sup>[1]</sup>	ADC	4 路
TFT 支持	最高支持 800*480 分辨率	TF 卡接口	1 路
触摸屏	四线电阻式	I <sup>2</sup> C	1 路
以太网	2 路 10/100M	看门狗	外置独立硬件看门狗
串口	6 路	硬件加密	支持 AES、SHA
SPI	3 路	RTC	支持内部 RTC 实时时钟模块
CAN	2 路		

[1]: EasyARM-iMX287 V1.00 版本的 NAND Flash 容量为 256MB, 内存为 128MB, V1.01 及之后版本 NAND Flash 容量为 128MB, 内存为 64MB。

### 10.1.2 底板资源

为方便用户灵活配置处理器相关功能复用 I/O, 评估处理器相关外设的项目应用, 除开发或学习所必需的外设外, 核心板其他资源均通过排针方式引出。EasyARM-iMX287 开发套件硬件底板资源如下:

- 6 路串口
  - ◆ 1 路调试用串口 DUART (DB9 座引出)
  - ◆ 5 路应用串口 UART (以排针方式引出, 分别是 UART0/1/2/3/4)
- 2 路 USB 2.0 接口
  - ◆ 1 路 Host 接口, 支持 U 盘、USB 鼠标和键盘等设备
  - ◆ 1 路 OTG 接口
- 1 路 TF 卡接口
- 1 路以太网接口
- 1 个蜂鸣器
- 3 个按键: 1 个复位键、1 个电源管理按键及 1 个应用按键
- 1 路 16 位液晶屏接口(含触摸屏接口, 支持 4 线电阻式触摸屏), 默认支持 480×272 TFT 液晶屏套件
- 以排针方式引出的其他接口
  - ◆ 1 路 I<sup>2</sup>C
  - ◆ 2 路 SPI (可复用为 UART2/3)
  - ◆ 3 个低速 ADC 通道, 1 个高速 ADC 模块
  - ◆ 19 个 GPIO
  - ◆ 2 路 PWM
  - ◆ 2 路 CAN

## 10.2 CAN接口的使用

### 10.2.1 内核开启CAN驱动支持

i.MX281/285/286/287 芯片带有 2 路 CAN, Linux 下的 CAN 驱动使用 socket 模式, 可像操作以太网一样操作 CAN。为了让内核支持 CAN 外设, 需要在内核配置中选中 Freescale



FlexCAN 模块，并重新编译内核，步骤如下：

在EasyARM-iMX283 的内核目录执行make menuconfig，进入Networking support子菜单，如图10.2所示：

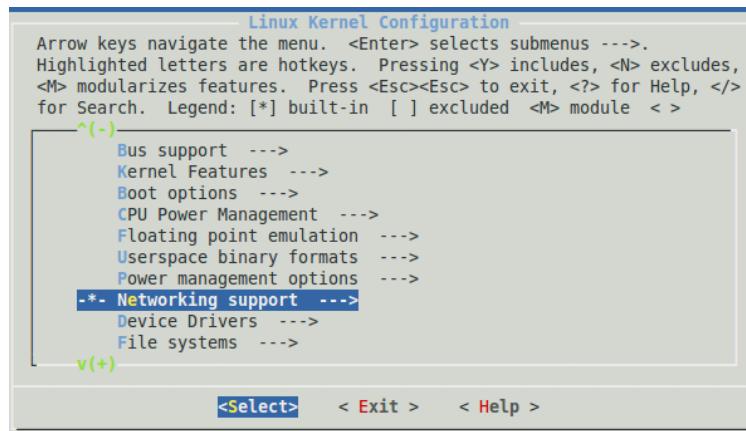


图10.2 进入 Networking support 子菜单

进入CAN bus subsystem support子菜单，如图10.3所示：

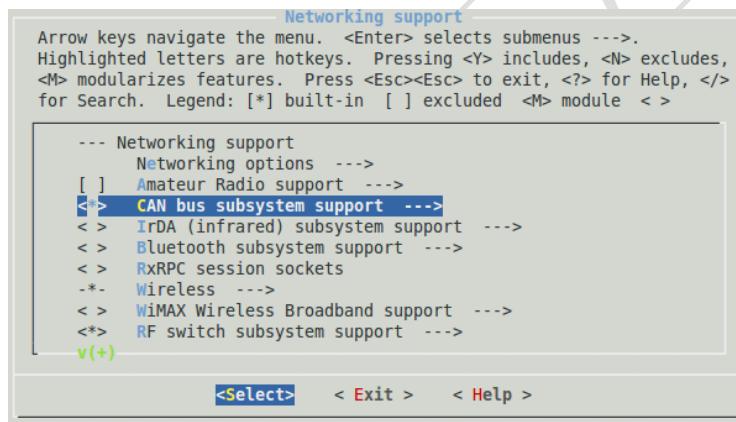


图10.3 进入 CAN bus subsystem support 子菜单

进入CAN Device Drivers子菜单，如图10.4所示：

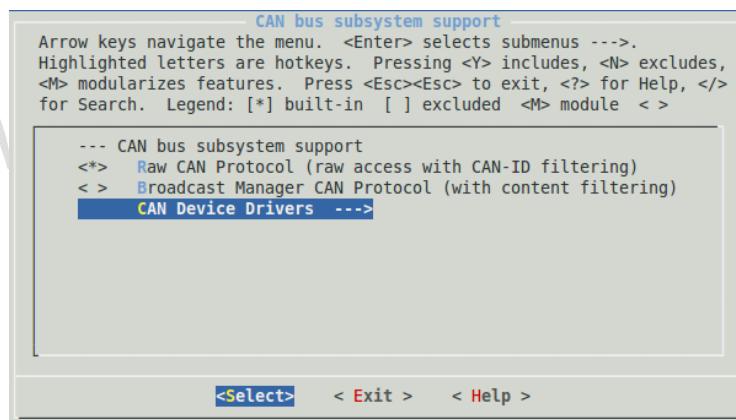


图10.4 进入 CAN Device Drivers 子菜单

按空格选中Freescale FlexCAN模块，如图10.5所示：

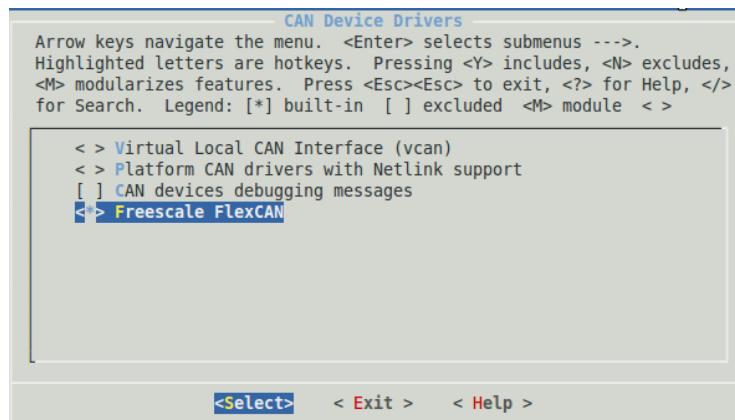


图10.5 选中 Freescale FlexCAN 设备

执行 make uImage 生成支持 CAN 驱动的内核镜像即可。

### 10.2.2 使用CAN设备

使用新的内核烧写系统后，进入串口终端，先关闭 CAN 设备：

```
ifconfig can0 down  
ifconfig can1 down
```

然后设置波特率，使用如下命令：

```
echo 1000000 > /sys/devices/platform/FlexCAN.0/bitrate  
echo 1000000 > /sys/devices/platform/FlexCAN.1/bitrate
```

可查看波特率：

```
cat /sys/devices/platform/FlexCAN.0/bitrate  
cat /sys/devices/platform/FlexCAN.1/bitrate
```

开启两个 CAN 设备：

```
ifconfig can0 up  
ifconfig can1 up
```

使用开发板上的 CAN0 接口进行测试，C0\_TX、C0\_RX 通过 CAN 收发器连接到 ZLG 的 CANScope 或 CANalyst 工具上，编译运行下面的程序：

程序清单10.1 CAN 测试程序

```
#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_BUSERROR) {
    errout("CAN_ERR_BUSERROR");
}

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);
        printf("=====\\n");
        printf("----- \\n");
        ret = read(fd, &frdup, sizeof(frdup));
        if (ret < sizeof(frdup)) {
            myerr("read failed");
            return -1;
        }
        if (frdup.can_id & CAN_ERR_FLAG) { /* 出错设备错误 */
            handle_err_frame(&frdup);
            myerr("CAN device error");
            continue;
        }
        print_frame(&frdup);
        ret = write(fd, &frdup, sizeof(frdup));
    }
}
```



```
    if (ret < 0) {
        myerr("write failed");
        return -1;
    }
}

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);
    if (s < 0) {
        perror("socket PF_CAN failed");
        return 1;
    }
    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;
    }
    if (1) {
        struct can_filter filter[2];
        filter[0].can_id = 0x200 | CAN_EFF_FLAG;
        filter[0].can_mask = 0xFFFF;

        filter[1].can_id = 0x20F | CAN_EFF_FLAG;
        filter[1].can_mask = 0xFFFF;
        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);
close(s);
return 0;
}
```

在CANTest软件中，设置帧ID为 0x20F或 0x200，数据为图10.6所示：



图10.6 CANTest 软件发送数据设置

注： CANScope或CANalyst等工具为广州致远电子开发的通用CAN分析仪，可通过致远电子销售购买，光盘中不提供对应的测试软件及使用方法。如无此工具，用户可自行使用其他方式测试CAN通信，CAN收发器推荐使用CTM8251AT模块，接法如图10.7所示：

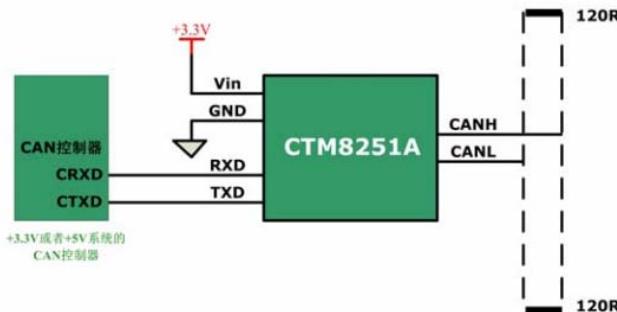


图10.7 CAN 隔离收发器 CTM8251AT

在 CANTest 软件里点击发送两次，运行结果如所下示：

```
root@EasyARM-iMX283 /mnt/nfs# ./cantest
=====
-----
0000020f
dlc = 8
data = 00 01 02 03 04 05 06 07
=====
-----
0000020f
dlc = 8
data = 00 01 02 03 04 05 06 07
```



=====

-----

### 10.2.3 socket CAN编程指南

#### 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()` 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。可以以下所示的方式来打开一个套接字：

```
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 `bind()` 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()`/`write()`，也可以使用 `send()`/`sendmsg()` 和对应的 `recv*` 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 10.2 所示。

程序清单 10.2 `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()`，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 10.3 所示。

程序清单 10.3 `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10.4所示。

程序清单10.4 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。recvfrom()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()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10.5所示。

程序清单10.5 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()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()获取数据源接口信息，程序清单10.6所示。

程序清单10.6 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10.7。

程序清单10.7 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

在上面的阶段中，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10.8所示。

程序清单10.8 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10.9所示。

#### 程序清单10.9 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。错误掩码的使用示例如下：

```
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));
```

### 10.3 SPI3 的使用

i.MX287 相对于 i.MX283 多了 1 路 SPI3，共 4 路 SPI，SPI0、SPI1 作为 MMC，SPI2 与 SPI3 为普通 SPI 设备。由于在 i.MX283 上管脚稀缺，默认 SPI1 是没有管脚分配的，需要从切换其他管脚功能才能使用 SPI1，因此在默认的 EasyARM-iMX283 开发包中，只支持 MMC 与 SPI2 两路 SPI 驱动。

而在 EasyARM-iMX287 开发板上，SPI1 与 SPI3 都引出了，此处仅拿增加 SPI3 的驱动作为示例，描述使用 SPI3 的方法。

开发板上关于 4 路 SPI 的引脚如图10.8所示：

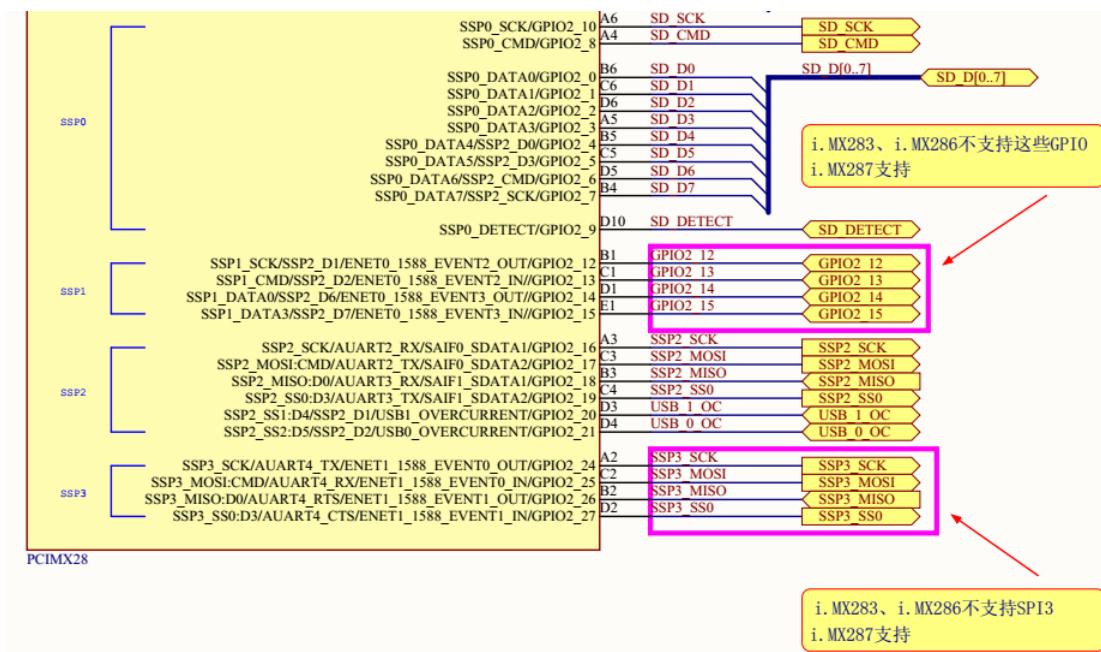


图10.8 4路 SPI 引脚原理图

为了让开发包兼容 EasyARM-iMX283 与 EasyARM-iMX287 两款开发板，默认的内核没有开启 SPI3 功能，对应的 SPI3 开启驱动的方法如下：

- 1) 在 V1.04 光盘资料“3.Linux\6.源代码”目录下找到 spi3\_for\_v1\_03.patch 文件，将其拷贝到内核目录下：  
vmuser@ Linux-host:~/old\_linux/linux-2.6.35.3\$ cp ..//spi3\_for\_v1\_03.patch .
- 2) 给内核打补丁：  
vmuser@ Linux-host:~/old\_linux/linux-2.6.35.3\$ patch -p1 < spi3\_for\_v1\_03.patch
- 3) 编译新内核：  
vmuser@ Linux-host:~/old\_linux/linux-2.6.35.3\$ make uImage
- 4) 烧写新内核，启动后，可以看到 SPI3 的设备节点为 spidev2.0：  
root@EasyARM-iMX283 ~# ls /dev/spi\*  
/dev/spidev1.0 /dev/spidev2.0

在运行程序清单5.17对应的SPI测试程序时只需要将设备文件名作为运行参数传递给测试程序即可，具体方法为在串口终端中输入如下测试指令，连接对应的SPI Flash芯片后，测试结果如图10.9所示：

```
root@EasyARM-iMX283 ~# ./spidev_test -D /dev/spidev2.0
```

```
spi mode: 0
bits per word: 8
max speed: 50000 Hz (50 KHz)

C2 25 15
```

图10.9 SPI3 测试结果

## 10.4 双网口的使用

i.MX287 处理器内部集成了一个 3 口交换机，因此可以引出两个以太网接口。双网口硬件设计可参考“2.硬件设计”目录下的设计文档。当需要应用 i.MX287 处理器的双网口功能时，需要在内核中开启双网口支持，具体操作步骤如下：

打开终端，通过命令浏览至内核源码所在目录，然后输入make menuconfig命令进入内核配置界面，打开内核配置界面后进入Device Drivers子菜单，如图10.10所示：

```
$ make menuconfig
```

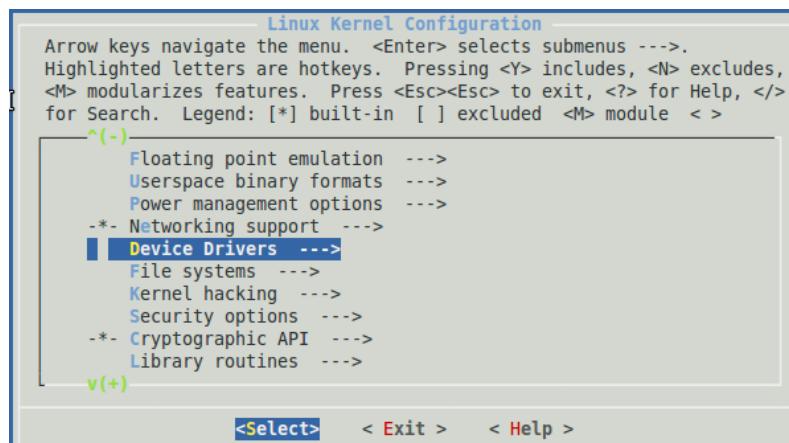


图10.10 进入 Device Drivers 子菜单

进入Network device support子菜单，如图10.11所示：

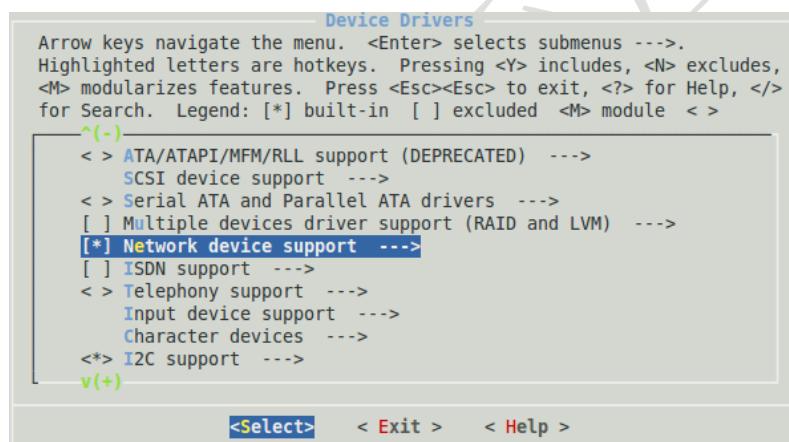


图10.11 进入 Network device support 子菜单

进入Ethernet子菜单，如图10.12所示：

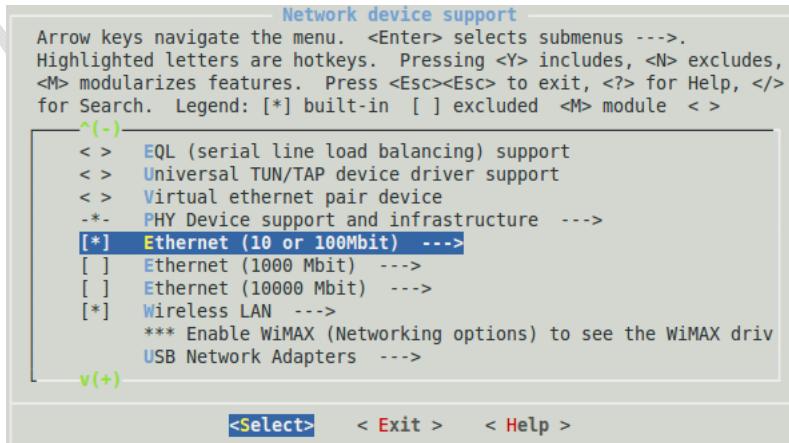


图10.12 进入 Ethernet 子菜单



取消选中FEC ethernet controller，如图10.13所示：

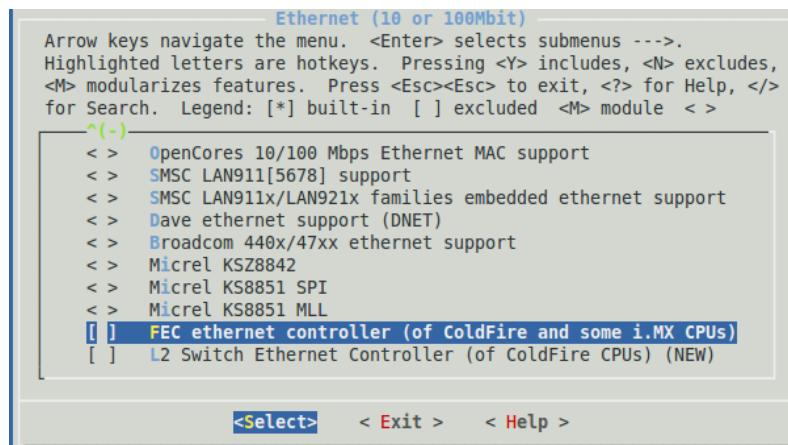


图10.13 取消选中 FEC ethernet controller

选中L2 Switch Ethernet Controller模块，如图10.14所示：

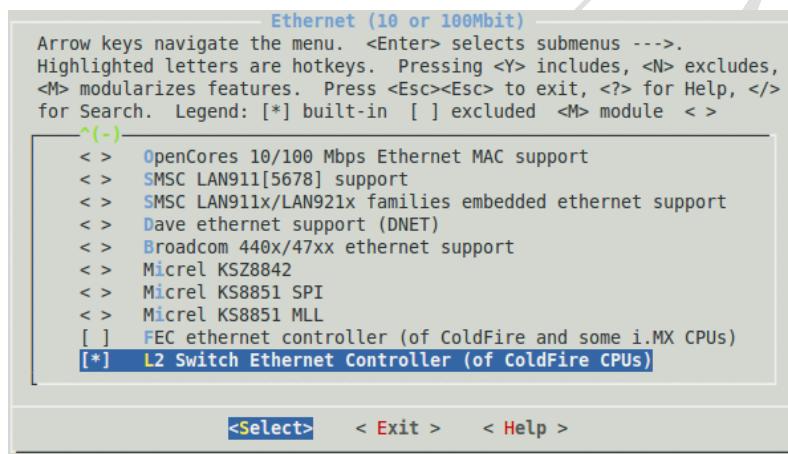


图10.14 选中 L2 Switch Ethernet Controller

保存配置后，重新编译内核，设置 eth0 的 IP，尝试将笔记本分别接在两个以太网接口上，测试能否 ping 通 eth0 的 IP。

**注意：如果开启了双网口功能而硬件上并没有两个以太网接口，会有启动问题，本章节仅针对使用 M287 核心板设计产品的用户群，不适用于 EasyARM-iMX287 开发板。**



## 图索引

图1.1 EasyARM-iMX283 产品正面 .....	5
图1.2 EasyARM-iMX283 的基本接口分布及核心板位置 .....	6
图2.1 熔丝配置专用启动卡制作完成 .....	10
图2.2 配置为从 TF 卡启动 .....	11
图2.3 使能未签名固件启动 .....	11
图2.4 跳线设置及接线示意图 .....	12
图2.5 脚本界面 .....	13
图2.6 完成 NAND Flash 格式化 .....	13
图2.7 NAND 格式化失败 .....	13
图2.8 正常的连接情况 .....	14
图2.9 NAND 格式化专用启动卡制作完成 .....	15
图2.10 “TF 卡烧写方案”的目录内容 .....	18
图2.11 添加的驱动器 .....	18
图2.12 提示用户输入读卡器的盘符 .....	19
图2.13 输入盘符 .....	19
图2.14 制作完成 .....	20
图2.15 TF 卡启动跳线设置 .....	20
图2.17 选择 SinglechipNAND .....	21
图2.18 勾选连接的 HID-compliantdevice .....	22
图2.19 MfgTool 监视 HID-compliantdevice .....	22
图2.20 U-Boot 烧写 .....	22
图2.21 烧写完成 .....	23
图2.22 U-Boot 启动信息。 .....	23
图2.23 烧写“内核+文件系统” .....	24
图2.24 网络烧写方案的跳线设置 .....	25
图2.25 进入 U-Boot 命令行模式 .....	25
图2.26 启动 Cisco TFTP Server 工具软件 .....	26
图2.27 设定 tftp 服务器根目录 .....	26
图2.28 配置并保存 U-Boot 参数 .....	27
图2.29 利用 tftp 服务器下载内核及根文件系统 .....	28
图2.30 tftp 服务器访问超时 .....	28
图2.31 tftp 服务器根目录下没有对应的文件 .....	29
图3.1 启动跳线设置 .....	30
图3.2 EasyARM-iMX283 模拟出来的 U 盘 .....	33
图3.3 在 Linux 系统中浏览 U 盘中的文件 .....	34



图3.4 LED 所在位置 .....	34
图3.5 蜂鸣器使能排针位置 .....	35
图4.1 嵌入式 Linux 开发环境模型 .....	36
图4.2 嵌入式 Linux 交叉开发流程 .....	36
图4.3 下载光盘镜像文件 .....	37
图4.4 解压下载文件得到安装镜像文件 .....	38
图4.5 下载 VMware Player .....	38
图4.6 安装 VMware Player .....	39
图4.7 接受许可协议 .....	39
图4.8 准备安装 .....	40
图4.9 完成安装 .....	40
图4.10 使能“Intel Virtualization Technology” .....	41
图4.11 创建新虚拟机 .....	41
图4.12 选择“稍后安装操作系统” .....	42
图4.13 选择客户机操作系统 .....	42
图4.14 设置虚拟机名称及存储位置 .....	43
图4.15 指定虚拟磁盘容量 .....	43
图4.16 点击“自定义硬件” .....	44
图4.17 将“CD/DVD”驱动器配置为待安装的 Ubuntu 映像文件 .....	44
图4.18 配置“网络适配器”为“桥接模式” .....	45
图4.19 完成虚拟机配置 .....	45
图4.20 启动虚拟机 .....	46
图4.21 Ubuntu 安装镜像正常启动 .....	46
图4.22 选择系统语言 .....	47
图4.23 点击“继续”按钮 .....	47
图4.24 清除整个虚拟磁盘 .....	48
图4.25 点击“现在安装”按钮 .....	48
图4.26 选择系统时间所在时区 .....	49
图4.27 确定键盘布局 .....	49
图4.28 设置用户名及密码均为“vmuser” .....	50
图4.29 系统安装完成后点击“现在重启”按钮 .....	50
图4.30 选择安装 VMware Tools .....	51
图4.31 VMware Tools 安装因没有网络连接而失败 .....	51
图4.32 Linux 启动后的桌面 .....	52
图4.33 按“Ctrl+Alt+T”组合键打开终端 .....	53
图4.34 成功安装 ssh 服务器 .....	55
图4.35 通过 vi 命令打开/etc/exports 文件 .....	56



图4.36 编辑“exports”文件 .....	56
图4.37 保存并退出 vi 编辑.....	57
图4.38 在主文件夹目录下创建“EasyARM-iMX283”目录.....	57
图4.39 创建测试目录.....	58
图4.40 启动 NFS 服务 .....	58
图4.41 挂在 NFS 目录测试 .....	59
图4.42 NFS 挂载成功 .....	59
图4.43 通过 DUART 配置目标板 Linux 系统.....	60
图4.44 NFS 挂载成功 .....	61
图4.45 安装 tftpd-hpa 及 tftp-hpa 软件.....	62
图4.46 通过 vi 打开“/etc/default/tftp-hpa”文件.....	62
图4.47 退出并保存“tftp-hpa”文件修改.....	63
图4.48 终端执行完 “/tftpboot” 目录权限设置 .....	63
图4.49 启动 tftp 服务 .....	63
图4.50 在 “/tftpboot” 目录下创建测试文档 tftpTestFile .....	64
图4.51 用文本编辑器打开测试文档 .....	64
图4.52 修改测试文档内容并保存 .....	64
图4.53 测试 tftp 服务器 .....	65
图4.54 更新环境变量 .....	67
图4.55 测试交叉编译器是否安装正确 .....	67
图4.56 编译“hello.c”文件 .....	68
图4.57 运行测试程序 .....	69
图4.58 创建 makefile 文件 .....	70
图4.59 利用 make 命令编译 .....	70
图5.1 所有功能部件例程 .....	71
图5.2 开发板引出的 283 可用 GPIO.....	72
图5.3 ADC 接口示意图 .....	74
图5.4 EasyARM-iMX283 可用的应用串口 .....	76
图5.5 I <sup>2</sup> C0 接口位置图 .....	85
图5.6 DS2460 连接原理图 .....	87
图5.7 测试 DS2460.....	89
图5.8 板上 3 路 PWM 输出位置图 .....	89
图5.9 开发板上的 SPI 接口位置图 .....	90
图5.10 SPI Flash 信号连接图 .....	93
图5.11 SPI 示例执行结果 .....	98
图7.1 内核配置主菜单 .....	106
图7.2 进入 Device Drivers 子菜单 .....	107



图7.3 选中 Block devices 支持 .....	107
图7.4 进入 MMC/SD 配置子菜单 .....	107
图7.5 TF 卡模块配置 .....	107
图7.6 进入 Device Drivers 子菜单 .....	108
图7.7 进入 USB support 子菜单 .....	108
图7.8 选中 Host 和 USB 电源管理两个模块 .....	108
图7.9 按图中选中各个 USB 子模块 .....	109
图7.10 选中 U 盘存储模块 .....	109
图7.11 进入 I2C support 子菜单 .....	109
图7.12 选中图中两个 I2C 子模块 .....	110
图7.13 进入 I2C Hardware Bus support 子菜单 .....	110
图7.14 选中 I2C 各个子模块 .....	110
图7.15 蜂鸣器在原理图 .....	112
图7.16 编译驱动 .....	116
图9.1 Qt 支持的平台 .....	125
图9.2 Qt/E 编程图示 .....	126
图9.3 hello 开发流程图 .....	127
图9.4 运行 hellow 程序 .....	128
图9.5 修改 start_userapp 文件 .....	130
图9.6 hellow 程序运行界面 .....	131
图9.7 Qt 例程界面 .....	135
图9.8 Qt Creator 主界面 .....	136
图9.9 更新 Linux 安装源 .....	137
图9.10 QT SDK 安装过程中可能出现的警告窗口 .....	137
图9.11 完成 QT SDK 的安装 .....	138
图9.12 显示 qtcreate 的主菜单 .....	138
图9.13 选项界面 .....	139
图9.14 添加 Qt 版本 .....	139
图9.15 添加完 Qt 版本 .....	140
图9.16 添加一个 GCC 项 .....	140
图9.17 添加新的 GCC 项 .....	141
图9.18 添加交叉编译器路径 .....	141
图9.19 新建 Gui 应用项目 .....	142
图9.20 设置项目名称及创建路径 .....	142
图9.21 目标设置界面 .....	143
图9.22 类信息界面 .....	143
图9.23 项目管理界面 .....	144



图9.24 mainwindow.cpp 界面 .....	144
图9.25 可视化界面编辑器.....	145
图9.26 hello world 程序界面 .....	145
图9.27 选择 Qt 版本.....	146
图9.28 hello world 界面 .....	146
图9.29 选择交叉编进行构建译器.....	147
图9.30 交叉编译后的输出内容.....	147
图9.31 Hello World 程序在开发套件上运行 .....	148
图9.32 zy launcher 界面 .....	148
图9.33 zy launcher 框架结构 .....	149
图10.1 EasyARM-iMX287 的基本接口分布及核心板位置 .....	153
图10.2 进入 Networking support 子菜单 .....	155
图10.3 进入 CAN bus subsystem support 子菜单 .....	155
图10.4 进入 CAN Device Drivers 子菜单 .....	155
图10.5 选中 Freescale FlexCAN 设备 .....	156
图10.6 CANTest 软件发送数据设置.....	160
图10.7 CAN 隔离收发器 CTM8251AT .....	160
图10.8 4 路 SPI 引脚原理图 .....	165
图10.9 SPI3 测试结果 .....	165
图10.10 进入 Device Drivers 子菜单.....	166
图10.11 进入 Network device support 子菜单 .....	166
图10.12 进入 Ethernet 子菜单 .....	166
图10.13 取消选中 FEC ethernet controller .....	167
图10.14 选中 L2 Switch Ethernet Controller .....	167



## 表格索引

表1.1 M283 核心板资源 .....	6
表2.1 NAND Flash 分区信息.....	9
表3.1 核心板配置信号功能描述.....	30
表5.2 c_cflag 部分可用选项 .....	78
表5.3 c_iflag 标志.....	79
表5.4 c_oflag 标志.....	80
表5.5 c_lflag 标志.....	80
表5.6 c_cc 标志 .....	81
表5.7 SPI_IOC_WR_MODE 命令 .....	91
表5.8 SPI_IOC_RD_MODE 命令 .....	91
表5.9 SPI_IOC_WR_BITS_PER_WORD 命令 .....	92
表5.10 SPI_IOC_WR_MAX_SPEED_HZ .....	92
表5.11 SPI_IOC_MESSAGE(1)命令 .....	92
表6.1 U-Boot 重要目录说明.....	99
表8.1 FHS 顶层目录 .....	120
表8.2 /usr 目录.....	121
表10.1 M287 核心板资源 .....	154



## 程序清单索引

程序清单4.1 Hello 程序清单 .....	67
程序清单4.2 应用程序 Makefile 范例 .....	69
程序清单5.1 C 语言操作 GPIO 示例 .....	73
程序清单5.2 ADC 操作示例 .....	75
程序清单5.3 打开串口设备 .....	77
程序清单5.4 向串口设备写入数据 .....	77
程序清单5.5 读入串口数据 .....	77
程序清单5.6 termios 结构 .....	78
程序清单5.7 设置和获取 termios 结构属性 .....	82
程序清单5.8 设置串口输入/输出波特率函数 .....	82
程序清单5.9 设置波特率示例 .....	82
程序清单5.10 串口操作示例 .....	83
程序清单5.11 打开 I <sup>2</sup> C 设备文件 .....	85
程序清单5.12 ioctl 命令定义 .....	86
程序清单5.13 DS2460 测试程序 .....	87
程序清单5.14 打开 SPI 设备文件 .....	91
程序清单5.15 struct spi_ioc_transfer 结构体的定义 .....	92
程序清单5.16 SPI 测试代码 .....	93
程序清单7.1 引脚的功能定义 .....	111
程序清单7.2 驱动的初始化操作 .....	112
程序清单7.3 gpio_miscdev 的实现 .....	113
程序清单7.4 gpio_fops 的实现 .....	113
程序清单7.5 gpio_exit 函数的实现 .....	114
程序清单7.6 gpio_open 函数的实现 .....	114
程序清单7.7 gpio_write 函数的实现 .....	114
程序清单7.8 gpio_ioctl 的调用 .....	115
程序清单7.9 gpio_release 函数的实现 .....	115
程序清单7.10 Makefile 文件的实现 .....	116
程序清单7.11 蜂鸣器测试程序代码 .....	116
程序清单7.12 LCD 时序列表 .....	118
程序清单7.13 fb_entry 的定义与初始化 .....	118
程序清单8.1 用户启动文件 .....	123
程序清单9.1 hello 程序代码 .....	127
程序清单9.2 启动脚本代码 .....	129
程序清单9.3 Qt 例程代码 .....	133



程序清单9.4 MainMenu.qml 文件 .....	149
程序清单10.1 CAN 测试程序 .....	156
程序清单10.2 can_frame 的定义 .....	161
程序清单10.3 struct sockaddr_can 结构体 .....	161
程序清单10.4 绑定接口 .....	162
程序清单10.5 接收 CAN 帧 .....	162
程序清单10.6 获取数据源接口信息 .....	163
程序清单10.7 指定输出接口的详细信息 .....	163
程序清单10.8 can_filter 的定义 .....	163
程序清单10.9 启用过滤器示例代码 .....	164



## 参考文献

- [1] Freescale. 《IMX28CEC.pdf》, <http://www.freescale.com/>, Rev.3, 2012
- [2] Freescale. 《IMX28RM.pdf》, <http://www.freescale.com/>, Rev.1, 2010

广州周立功



## 免责声明

广州周立功单片机科技有限公司所提供的所有服务内容旨在协助客户加速产品的研发进度，在服务过程中所提供的任何程序、文档、测试结果、方案、支持等资料和信息，都仅供参考，客户有权不使用或自行参考修改，本公司不提供任何的完整性、可靠性等保证，若在客户使用过程中因任何原因造成的特别的、偶然的或间接的损失，本公司不承担任何责任。

广州周立功



## 销售与服务网络

### 广州周立功单片机科技有限公司

地址：广州市天河北路 689 号光大银行大厦 12 楼 F4

邮编：510630

传真：(020)38730925

网址：[www.zlgmcu.com](http://www.zlgmcu.com)

电话：(020)38730916 38730917 38730972 38730976 38730977



<http://www.zlgmcu.com>

### 广州专卖店

地址：广州市天河区新赛格电子城 203-204 室

电话：(020)87578634 87569917

传真：(020)87578842

### 南京周立功

地址：南京市珠江路 280 号珠江大厦 1501 室

电话：(025)68123920 68123923 68123901

传真：(025)68123900

### 北京周立功

地址：北京市海淀区知春路 108 号豪景大厦 A 座 19 层

电话：(010)62536178 62536179 82628073

传真：(010)82614433

### 重庆周立功

地址：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 68796439

传真：(023)68796439

### 杭州周立功

地址：杭州市天目山路 217 号江南电子大厦 502 室

电话：(0571)89719480 89719481 89719482

89719483 89719484 89719485

传真：(0571)89719494

### 成都周立功

地址：成都市一环路南二段 1 号数码科技大厦 403 室

电话：(028)85439836 85437446

传真：(028)85437896

### 深圳周立功

地址：深圳市福田区深南中路 2072 号电子大厦 12 楼 1203

电话：(0755)83781788 (5 线) 83782922 83273683

传真：(0755)83793285

### 武汉周立功

地址：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 87168297 87168397

传真：(027)87163755

### 上海周立功

地址：上海市北京东路 668 号科技京城东座 12E 室

电话：(021)53083452 53083453 53083496

传真：(021)53083491

### 西安办事处

地址：西安市长安北路 54 号太平洋大厦 1201 室

电话：(029)87881296 83063000 87881295

传真：(029)87880865

### 厦门办事处

E-mail：[sales.xiamen@zlgmcu.com](mailto:sales.xiamen@zlgmcu.com)

### 沈阳办事处

E-mail：[sales.shenyang@zlgmcu.com](mailto:sales.shenyang@zlgmcu.com)