User Manual

Linux 开发指南

EasyARM-iMX283 开发平台

UM13112601 V1.00 Date: 2013/11/26 产品用户手册

类别	内容
关键词	EasyARM-iMX283 嵌入式 Linux
摘 要	介绍在 EasyARm-iMX283 平台下的 Linux 软件开发

广州周立功单片机科技有限公司

修订历史

版本	日期	原因
V1.00	2013-11-26	创建文档

目　录

1. 嵌入式开发环境构建.....1
1.1 应用程序开发环境构建.....1
1.1.1 嵌入式Linux开发一般方法.....1
1.1.2 安装操作系统.....2
1.1.3 网卡配置.....2
1.1.4 ssh服务器配置.....2
1.1.5 NFS服务器配置.....2
1.2 TFTP服务器.....3
1.2.1 安装tftp软件.....3
1.2.2 配置tftp服务器.....3
1.2.3 启动tftp服务.....4
1.2.4 测试tftp服务器.....4
1.3 构建交叉开发环境.....4
1.4 Hello程序.....6
2. EasyARM-iMX283 资源简介.....8
2.1 硬件资源.....8
2.1.1 i.MX28 处理器特性.....8
2.1.2 EasyARM-iMX283 套件特性.....8
2.2 Linux平台软件开发资源.....8
3. 在EasyARM-iMX283 安装Linux系统.....10
3.1.1 nandflash存储器分区信息.....10
3.1.2 u-boot烧写前的准备.....10
3.1.3 烧写u-boot.....12
3.1.4 烧写内核与文件系统.....15
3.2 系统操作和基本设置.....17
3.2.1 系统启动跳线设置.....17
3.2.2 系统登陆.....17
3.2.3 网络设置.....18
3.2.4 RTC时间设置.....18
3.2.5 SD卡使用.....18
3.2.6 U盘使用.....18
3.2.7 usb device使用.....19
3.2.8 LED使用.....19
3.2.9 蜂鸣器使用.....19
3.2.10 LCD背光控制.....19
3.2.11 开机启动设置.....20
4. 功能部件编程.....21
4.1 GPIO应用编程.....21
4.1.1 导出GPIO.....21
4.1.2 GPIO方向设置.....21
4.1.3 输入电平读取.....21

4.1.4 GPIO输出电平控制 22
4.2 ADC接口 22
4.2.1 ADC驱动模块的加载 22
4.2.2 操作接口 22
4.2.3 计算公式 23
4.2.4 操作示例 23
4.3 串口编程 24
4.3.1 访问串口设备 24
4.3.2 配置串口接口属性 25
4.4 I^2C接口 34
4.4.1 open调用 34
4.4.2 ioctl调用 34
4.4.3 write调用 35
4.4.4 read调用 35
4.4.5 close调用 36
4.5 PWM接口 36
4.5.1 PWM占空比设置与输出 36
4.5.2 系统命令操作示例 36
4.5.3 应用程序操作示例 37
4.6 SPI接口 37
4.6.1 open调用 37
4.6.2 ioctl调用 37
4.6.3 示例代码 40
5. EasyARM-iMX283 的bootloader 46
5.1 U-Boot简介 46
5.1.1 U-Boot源代码目录结构 46
5.2 编译u-boot 47
5.3 U-Boot基本命令 47
5.3.1 预设的组合命令 49
5.3.2 通过网络启动内核 49
5.4 U-Boot Tools 50
6. Linux内核编译和驱动要点 51
6.1 编译内核 51
6.2 配置内核 51
6.3 内核GPIO使用方法 53
6.4 设置LCD的时序 54
7. 嵌入式Linux根文件系统 56
7.1 Linux根文件系统 56
7.2 FHS标准 56
7.2.1 顶层目录 56
7.2.2 /usr目录 57
7.3 BusyBox 57
7.4 NFS根文件系统 57
7.5 生成文件系统映像 58

8. 嵌入式Linux Qt编程指南 60
8.1 背景知识 60
8.2 Qt介绍 60
8.2.1 Qt简介 60
8.2.2 Qt/E简介 60
8.3 编译环境的搭建 61
8.4 Hello world程序开发 61
8.4.1 编译hello程序 61
8.4.2 在目标板上运行hello程序 63
8.5 qmake与pro文件 65
8.5.1 pro文件例程 65
8.5.2 pro文件常见配置 66
8.6 Qt编程简单入门 67
8.6.1 例程讲解 67
8.6.2 信号和槽机制 68
8.7 桌面版本的Qt SDK使用简介 69
8.7.1 桌面版本Qt SDK简介 69
8.7.2 桌面版本Qt SDK的安装 70
8.7.3 Qt Creator配置 70
8.7.4 Qt Creator使用例程 71
8.7.5 移植hello world 74
8.8 zylauncher图形框架 74

1. 嵌入式开发环境构建

1.1 应用程序开发环境构建

1.1.1 嵌入式Linux开发一般方法

嵌入式Linux系统由于系统资源的匮乏，无法安装本地编译器进行本地开发，通常需要在借助一台主机进行交叉开发。通常，主机运行Linux操作系统，在主机安装相应的交叉编译器，将在主机编辑好的程序代码交叉编译后，通过一定方式如以太网或者串口将程序下载到目标系统运行，或者进行调试，一般的交叉开发流程如图 1.1所示。

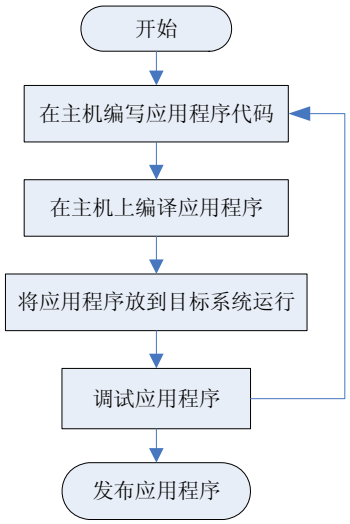

图 1.1 嵌入式 Linux 交叉开发一般流程

嵌入式Linux开发的一般模型如图 1.2所示。通常需要一台PC主机，在其中安装好各种进行交叉编译所需要的软件，通过串口和以太网和目标板相连。在主机上进行程序代码编辑和编译，得到的可执行文件通过串口或者以太网下载到目标板中运行或者进行调试。

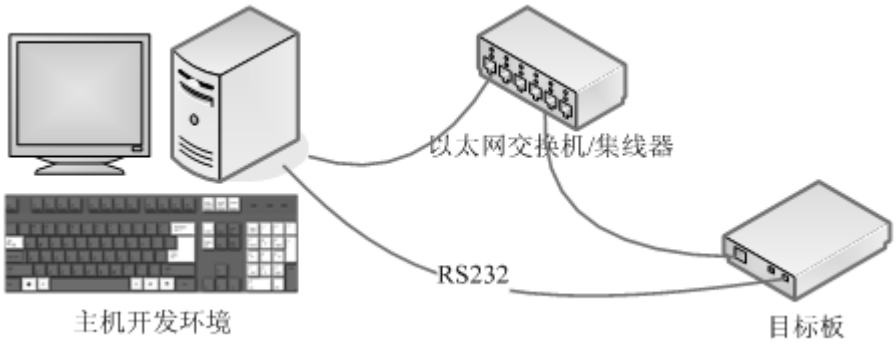

图 1.2 嵌入式 Linux 开发模型

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

对主机的要求，硬件方面，要求有串口和网口；软件方面，操作系统强烈推荐使用 ubuntu-12.04 以上版本的 **64 位**发行版，同时还需要其它的软件如交叉编译器等。

1.1.2 安装操作系统

可以直接在 PC 上安装全新的 ubuntu 操作系统，也可以在 Windows 操作系统中安装虚拟机，然后在虚拟机中安装 ubuntu 操作系统，如果采用虚拟机方式，请关闭 Windows 的防火墙，另外虚拟机网卡连接方式请选用 Bridged。

本节的主机设置以 ubuntu 发行版为例进行介绍，如果使用其它发行版，请参考对应发行版的教材或者其它资料。

另外，为了安全起见，都是以普通用户登录系统进行操作，在需要使用 root 权限的时候，使用 sudo 命令进行操作。

需要注意的是：安装的 ubuntu 必须能上外网，以便随时升级。虚拟机需要上外网时网卡连接方式请选用 Net 方式，与 Windows 主机共享文件时使用 Bridged 方式

1.1.3 网卡配置

首先需为主机设置一个静态 IP，如 192.168.1.138。

1.1.4 ssh服务器配置

主机安装ssh服务器是用于在windows下使用SSH Secure Shell Client软件与虚拟机linux系统进行文件共享和远程登录，需要注意的是SSH Secure Shell Client登录linux系统或共享文件必须使以太网连接方式为bridged，否则无法连接。在windows系统下的SSH Secure Shell Client软件在光盘“1.Windows工具软件”目录下有提供，也可以在网上搜索下载得到，参考下载网址http://www.onlinedown.net/softdown/20089_2.htm。

使用如下命令安装 ssh 服务器：

```
~$sudo apt-get install openssh-server
```

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

1. 安装NFS软件包

在 ubuntu 上请输入下面命令：

```
[chenxibing@localhost ~]$ sudo apt-get install nfs-kernel-server
[chenxibing@localhost ~]$ sudo apt-get install nfs-common
```

2. 添加NFS目录

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

定访问主机的 IP 以及访问权限。

```
[chenxibing@localhost ~]$ sudo vi /etc/exports
[sudo] password for chenxibing:
```

如增加/home/proton/EasyARM-iMX283 目录，并允许 IP 为 192.168.1.*的任何系统进行 NFS 访问，增加内容如下：

```
/home/proton/EasyARM-iMX283          192.168.1.*(rw,sync,no_root_squash)
```

3. 启动NFS服务

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

```
[chenxibing@localhost ~]$ sudo /etc/init.d/nfs-kernel-server start
[sudo] password for chenxibing:
*Exporting directories for NFS kernel daemon...                    [ OK ]
* Starting NFS kernel daemon                                       [ OK ]
```

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

4. 测试NFS服务器

首先可以在主机上进行自测，将已经设定好的 NFS 共享目录 mount 到另外一个目录下，看能否成功。假定主机 IP 为 192.168.1.138，NFS 共享目录为 /home/proton/EasyARM-iMX283，可使用如下命令进行测试：

```
# mount  -t  nfs  192.168.1.138: /home/proton/EasyARM-iMX283    /mnt –o nolock
```

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/proton/EasyARM-iMX283 目录下的内容。

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

```
[root@zlg /]# ifconfig eth0 192.168.1.136
[root@zlg /]# ping 192.168.1.138
[root@zlg /]# mount -t nfs 192.168.1.138: /home/proton/EasyARM-iMX283    /mnt -o nolock
```

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

1.2 TFTP服务器

1.2.1 安装tftp软件

安装 tftp 服务器需要安装 tftpd-hpa 软件。如果主机可以上网，直接通过工具即可进行简单安装：

```
[proton@localhost ~]$ sudo apt-get install tftpd-hpa  tftp-hpa
```

1.2.2 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/tftp-hpa：

```
# /etc/default/tftpd-hpa
```

```
TFTP_USERNAME="tftp"
TFTP_DIRECTORY="/tftpboot"                    #请改成自己的目录
TFTP_ADDRESS="0.0.0.0:69"
TFTP_OPTIONS="-l -c -s"
```

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，也可以更改为其它合适的目录。

```
[proton@localhost ~]$ sudo mkdir /tftpboot
[proton@localhost ~]$ sudo chmod –R    777    /tftpboot
[proton@localhost ~]$ sudo chown –R nobody    /tftpboot
```

1.2.3 启动tftp服务

tftp服务器安装配置完成后，请重启电脑。

然后启动tftp服务。

```
[proton@localhost ~]$ sudo service tftpd-hpa start
```

1.2.4 测试tftp服务器

复制一个文件到 tftp 服务器目录，然后在主机启动 tftp 软件，进行简单测试。

```
[proton@localhost ~]$ tftp 192.168.7.231
tftp> get file
tftp> q
```

如果没有出现错误提示，则表示 tftp 服务器已经配置成功。

1.3 构建交叉开发环境

1. 工具链介绍

交叉编译器可以自行制作，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。光盘提供了已经编译好的交叉工具链，在3.Linux\2. 工具软件\2.Linux 工具软件目录下，以压缩包形式提供：gcc-4.4.4-glibc-2.11.1-multilib-1.0_EasyARM-iMX283.tar.bz2。光盘资料可以使用 u 盘方式拷贝到 linux 主机中，也可使用 SSHSecureShellClient 通过共享的方式把光盘资料拷贝到 linux 主机！

2. 安装工具链

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

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

主机没有安装 32 位兼容库，在使用交叉编译工具的时候可能会出现错误：

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

安装 libncurses5-dev，使用如下命令进行安装：

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

如果没有安装此库，在使用 make menucofig 时会如所示的错误：

```
*** Unable to find the ncurses libraries or the
```

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

安装交叉编译工具链需要 root 权限。在终端执行命令：

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

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

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

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

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

```
[chenxibing@localhost ~]$ arm-none-linux-gnueabi-
arm2hpdl                        arm-linux-addr2line             arm-none-linux-gnueabi-ar
arm-fsl-linux-gnueabi-addr2line arm-linux-ar                    arm-none-linux-gnueabi-as
arm-fsl-linux-gnueabi-ar        arm-linux-as                    arm-none-linux-gnueabi-c++
arm-fsl-linux-gnueabi-as        arm-linux-c++                   arm-none-linux-gnueabi-cc
arm-fsl-linux-gnueabi-c++       arm-linux-cc                    arm-none-linux-gnueabi-c++filt
arm-fsl-linux-gnueabi-cc        arm-linux-c++filt               arm-none-linux-gnueabi-cpp
arm-fsl-linux-gnueabi-c++filt   arm-linux-cpp                   arm-none-linux-gnueabi-ct-ng.config
arm-fsl-linux-gnueabi-cpp       arm-linux-ct-ng.config          arm-none-linux-gnueabi-g++
arm-fsl-linux-gnueabi-ct-ng.config arm-linux-g++                arm-none-linux-gnueabi-gcc
arm-fsl-linux-gnueabi-g++       arm-linux-gcc                   arm-none-linux-gnueabi-gcc-4.4.4
arm-fsl-linux-gnueabi-gcc       arm-linux-gcc-4.4.4             arm-none-linux-gnueabi-gccbug
arm-fsl-linux-gnueabi-gcc-4.4.4 arm-linux-gccbug                arm-none-linux-gnueabi-gcov
arm-fsl-linux-gnueabi-gccbug    arm-linux-gcov                  arm-none-linux-gnueabi-gdb
arm-fsl-linux-gnueabi-gcov      arm-linux-gdb                   arm-none-linux-gnueabi-gprof
arm-fsl-linux-gnueabi-gdb       arm-linux-gprof                 arm-none-linux-gnueabi-ld
arm-fsl-linux-gnueabi-gprof     arm-linux-ld                    arm-none-linux-gnueabi-ldd
arm-fsl-linux-gnueabi-ld        arm-linux-ldd                   arm-none-linux-gnueabi-nm
arm-fsl-linux-gnueabi-ldd       arm-linux-nm                    arm-none-linux-gnueabi-objcopy
arm-fsl-linux-gnueabi-nm        arm-linux-objcopy               arm-none-linux-gnueabi-objdump
arm-fsl-linux-gnueabi-objcopy   arm-linux-objdump               arm-none-linux-gnueabi-populate
arm-fsl-linux-gnueabi-objdump   arm-linux-populate              arm-none-linux-gnueabi-ranlib
arm-fsl-linux-gnueabi-populate  arm-linux-ranlib                arm-none-linux-gnueabi-readelf
```

arm-fsl-linux-gnueabi-ranlib	arm-linux-readelf	arm-none-linux-gnueabi-run
arm-fsl-linux-gnueabi-readelf	arm-linux-run	arm-none-linux-gnueabi-size
arm-fsl-linux-gnueabi-run	arm-linux-size	arm-none-linux-gnueabi-strings
arm-fsl-linux-gnueabi-size	arm-linux-strings	arm-none-linux-gnueabi-strip
arm-fsl-linux-gnueabi-strings	arm-linux-strip	
arm-fsl-linux-gnueabi-strip	arm-none-linux-gnueabi-addr2line	

3. 测试工具链

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

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

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

1.4 Hello程序

使用熟悉的文本编辑器，编写一个简单的程序，往终端打印“Hello”字符串，示例程序如程序清单 1.1所示。

程序清单 1.1 Hello 程序清单

```
#include <stdio.h>

int main(void)
{
    int i;

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

    return 0;
}
```

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

```
[chenxibing@localhost hello]$ arm-none-linux-gnueabi-gcc hello.c -o hello
```

编译完毕，将得到 hello 文件。

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

```
root@freescale /$ mount -t nfs 192.168.1.138: /home/proton/EasyARM-iMX283    /mnt -o nolock
root@freescale /$ cd /mnt/app/hello
root@freescale /$ ./hello
Hello 0!
Hello 1!
```

```
Hello 2!
Hello 3!
Hello 4!
```

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

```
root@freescale /$ cp hello /root/
```

这是一个非常简单的程序，并且只有一个文件，所以可以采用直接输入命令进行交叉编译，如果工程较大，文件较多，这种方式就不可取了，通常需要编写Makefile文件，通过make程序来进行工程管理。程序清单 1.2所示是一个简单的Makefile文件。

程序清单 1.2　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
```

有了合适的 Makefile 文件，只需在终端输入 make 命令即可编译程序。

2. EasyARM-iMX283 资源简介

本章导读

本章介绍 EasyARM-iMX283 套件硬件资源特性和 Linux 平台下的软件资源和开发资源。

2.1 硬件资源

2.1.1 i.MX28 处理器特性

- CPU：ARM926EJ-S，主频 454MHz，16Kbytes I-Cache，32K Kbytes D-Cache
- 128 Kbytes SRAM，128-bits 片上可一次编程 ROM
- 支持 16-bits DDR、DDR2、LV-DDR2，高达 205M DDR 时钟
- 支持 NANDFLASH 和 20-bits BCH ECC
- 10/100-Mbps 以太网、1 路集成 PHY 的 USB host 和、1 路集成 PHY 的 USB OTG
- 5 路带硬件流控功能应用串口，通信速率达 3.25Mbps，1 路调试串口，通信速率达 115Kbps
- 支持 4 SSP 接口，SSP0/SSP1 支持 1/4/8-bits 模式，SSP2/SSP3 支持 1/4-bits 模式
- 2 路I^2C主机或从机接口，通信速率高达 400kbps
- 支持内部 RTC，4 路 32-bits Timers ，1 个 Rotary Decoder，8 个 PWM 通道
- 所有 GPIO 具有中断能力
- 集成电源管理单元（PMU）
- 支持 24bits 液晶显示控制器，支持 4 线/5 线式电阻触摸屏
- 安全特性：唯一只读的 DRM ID、128-bits AES 硬件解密单元、SHA-1 和 SHA256 加密
- 289-pin BGA 封装

关于处理器更详细的特性请参考具体型号处理器的数据手册。

2.1.2 EasyARM-iMX283 套件特性

EasyARM-iMX283 开发套件以核心板加底板的方式组成，由于底板资源有限，底板仅将处理器常用的接口资源通过排针或相应的接口电路的方式引出，以方便方案评估与实验，其它未支持的驱动需要。EasyARM-iMX283 套件硬件资源特性：

- 核心板： 128M Bytes DDR2、256M Bytes Nand Flash
- 1 路调试用的 UART，4 路应用 UART（以排针方式引出）
- 2 路 USB 接口 ：一路 Host 接口，一路 OTG 接口
- 1 路 SD 卡接口
- 1 路以太网接口
- 1 路液晶屏接口，支持 4 线式电阻触摸屏，配套 480×272 TFT 液晶屏套件
- 以排针方式引出的接口：1 路 I2C、1 路 SPI、3 个通道 ADC、GPIO
- Linux平台提供的外设驱动：Nandflash、DDR2、MMc、SPI、I^2C、DUART、AUART0/1/4（其中AUART0/1 分别对应输入输出方向控制DIR1/DIR2 可扩展为RS485 接口）、ACD0/1/6、PWM3/4/7、GPIO2_4/5/6

2.2 Linux平台软件开发资源

EasyARM-iMX283 开发套件Linux平台提供的资源文件如表 2.1所示。

表 2.1　Linux 平台开发资源文件

文件		描述
MfgTool		用于烧写 u-boot 的工具，imx28_ivt_uboot.sb 需放在“MfgTool\Profiles\MX28 Linux Update\OS Firmware\files”目录下
SSHSecureShellClient-3.2.9.exe		SSH 登录 EasyARM-iMX283 或与主机进行文件共享
TeraTerm.rar		串口终端通信工具
tftpd32.exe		tftp 服务器软件，其所在的目录为 tftp 服务器的根目录
gcc-4.4.4-glibc-2.11.1-multilib-1.0_EasyARM-iMX283.tar.bz2		Linux 交叉编译工具，集成 QT 库
imx28_ivt_uboot.sb		可用于烧写并启动的 u-boot 文件，需要放置在“MfgTool\Profiles\MX28 Linux Update\OS Firmware\files”目录下进行烧写
rootfs.ubifs		根文件系统，通过u-boot命令烧写，需要放置在tftp服务器根目录下，参考本文第3章节
uImage		Linux内核文件，通过u-boot命令烧写，需要放置在tftp服务器根目录下，参考本文第3章节
bootloader_EasyARM-iMX283.tar.bz2		u-boot 源码文件
linux-2.6.35.3.tar.bz2		Linux 内核源码文件
开发示例	hello	输出 hello word 演示例程
	long-test	uart0 和 uart1 互相对发演示例程
	test_serial	uart0 自发自收演示例程
	adc_driver	adc 驱动例程
	adc_app	adc 演示例程，需要先加载 adc 驱动
	i2c_test	i2c 通信演示程序，读取 CAT24C04 读写演示，需要在 SCL1/SDA1 接 CAT24C04 器件
	spi	spi 通信演示程序，读取 MX25L1635E（spi flash）器件的 ID，需要接 MX25L1635E（spi flash）器件
	fb_test	液晶显示颜色例程，显示边框和矩形色块
	usb_device	usb 模拟 U 盘演示程序
	tcp	tcp 通信演示例程
	udp	udp 通信演示例程
	qt_hellow	QT label 演示功能，label 显示“hellow qt”
	qt_hellow_zh	QT label 显示“you should see some chinese words: 你好，世界”
	mkfs.ubifs/ubinize	制作根文件系统的工具，需要复制到主机/usr/sbin 目录下
	build_rootfs	制作根文件系统的脚本
	rootfs.tar.bz2	根文件系统压缩包

3. 在EasyARM-iMX283 安装Linux系统

本章导读

本章描述了在 EasyARM-iMX283 快速烧写固件的方法。

3.1.1 nandflash存储器分区信息

EasyARM-iMX283 板载 256MB的NAND Flash，芯片为K9F2G08U，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 3.1所列。

表 3.1 NAND Flash 分区信息

分　区	地址范围	大小	用　途
Bootloader	0x00000000-0x00200000	2048KB	u-boot 及其环境变量参数
kernel	0x00200000-0x00e00000	12MB	内核
rootfs	0x00000e00000-0x00010000000	242MB	根文件系统

3.1.2 u-boot烧写前的准备

如果 nandflash 在此之前写入过其它数据，如运行过 wince 系统，nandflash 的坏块标记信息可能被修改而不符合 linux 的坏块管理机制，就有可能在烧写 u-boot、内核和文件系统的时候烧写不成功，因此在烧写 u-boot 之前需要通过 u-boot 命令对 nandflash 进行整片擦除。此时需要先将 u-boot 写入 DDR2 中启动，u-boot 启动后使用’nand scrub’命令进行 nandflash 整片擦除。如果之前已经成功烧写并运行过 u-boot 和 linux 系统，则此步可以跳过。具体操作步骤：

1. 使用短路器短接 EasyARM-iMX283 底板的 JP6(WDT，短接禁能看门狗输出)和 JP2(USB_BT，设置为 USB 方式启动)；使用 mini USB 通信电缆接到 J12(OTG)接口，J7（DUART）接上串口，并打开串口终端通信软件，设置通信参数 115200bps，数据位 8-bits，停止位 1-bit，无校验（linux 系统中使用到的 DUART 的通信参数都为此参数，除非特殊说明）；在 J2 接上 5V 电源；
2. 在 windows XP 系统下解压 Mfgtools.rar，双击运行“MfgTool\Profiles\MX28 Linux Update\OS Firmware\files”目录下的 ubootloader.bat 脚本文件，
3. u-boot 在串口终端输出 u-boot 的基本信息，然后马上在串口终端界面一直按“空格”键，直到进入 u-boot 的命令行提示符，如下所示；

```
U-Boot 2009.08-dirty ( 1 鏈?08 2014 - 14:41:46)

Freescale i.MX28 family
CPU:   454 MHz
BUS:   151 MHz
EMI:   205 MHz
GPMI:   24 MHz
DRAM:  128 MB
NAND:  mfr_id: ec
Manufacturer          : Samsung (0xec)
Device Code           : 0xda
```

```
Cell Technology     : SLC
Chip Size           : 256 MiB
Pages per Block     : 64
Page Geometry       : 2048+64
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
tRHOH               : Unknown
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
Hit any key to stop autoboot:  0
MX28 U-Boot >
```

4. 输入‘nand scrub‘ 命令，并根据提示输入‘y’并回车，对整片 nandflash 进行擦除，如所示

```
MX28 U-Boot > nand scrub

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 if you
         are sure of what you are doing!

Really scrub this NAND flash? <y/N>
Erasing at 0xbae000000000000 --    0% complete.
nand0: MTD Erase failure: -5
Erasing at 0xffe000000000000 --    0% complete.
OK
MX28 U-Boot >
```

5. 对整片 nandflash 擦除过后就可以进行 u-boot、内核和文件系统的烧写。

3.1.3 烧写u-boot

1. 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 mini USB 通信电缆接到 J12 接口，在 J2 接上 5V 电源；
2. 在 windows XP 系统下解压 Mfgtools.rar，运行 MfgTool.exe；
3. 点击MfgTool菜单中的“Options”选择“Configuration…”，在“Profiles”标签的操作列表中鼠标右键点击“UTP_UPDATE”，在弹出的菜单中选择“编辑”，如图 3.1所示；

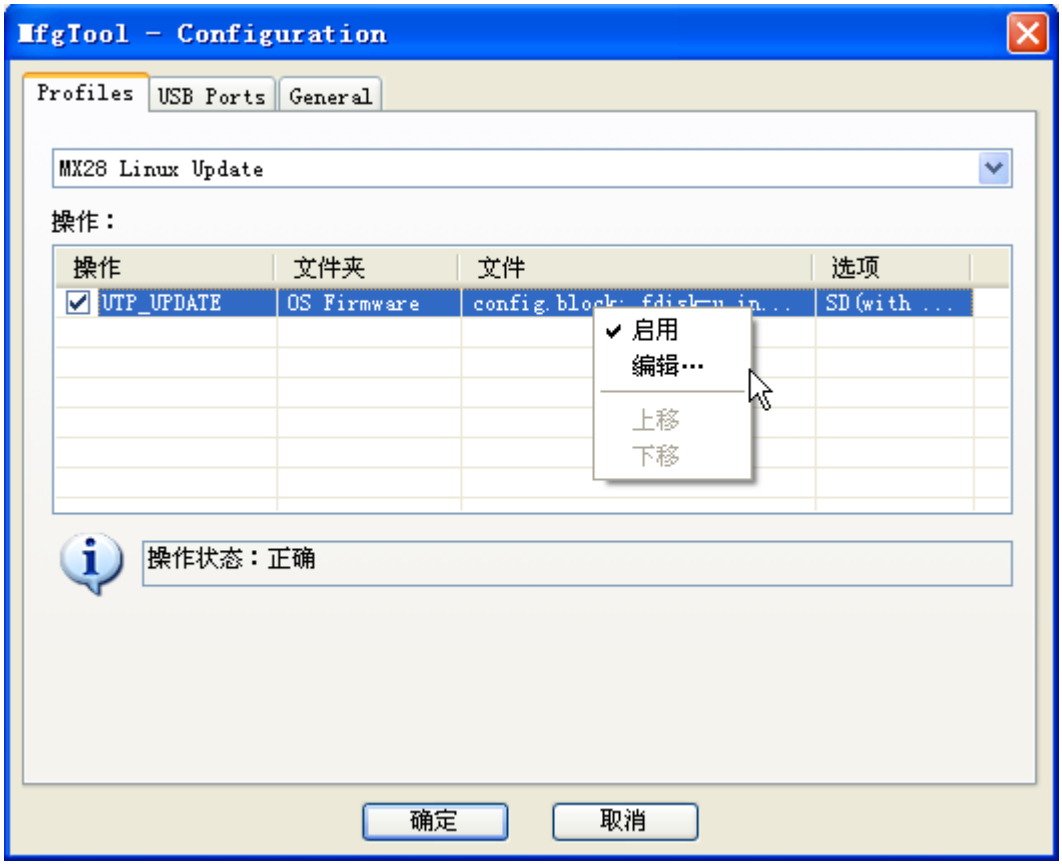

图 3.1 Profiles 标签

4. 在弹出的菜单中选择“Singlechip NAND”，然后点击“OK”，如图 3.2所示；

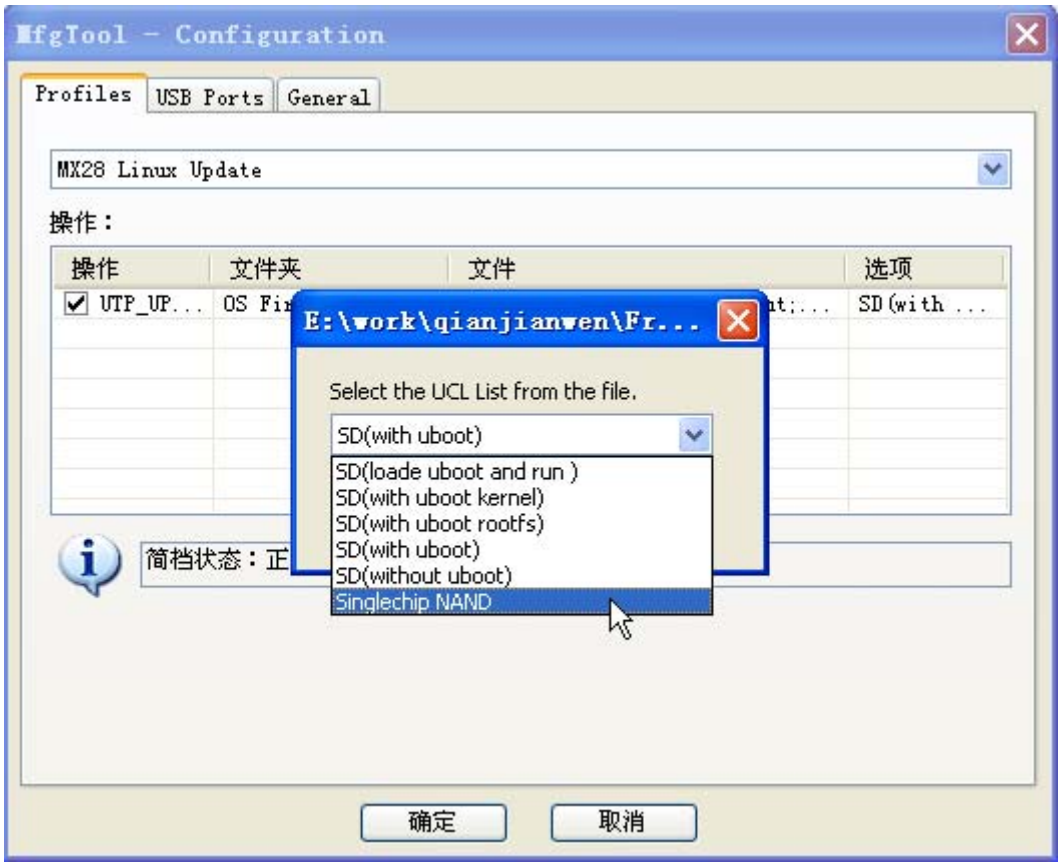

图 3.2　选择 Singlechip NAND

5. 切换到“USB Ports”标签，勾选已经连接上的的“HID-compliant device”（即EasyARM-iMX283 设备）；然后点击“确定”，如图 3.3所示；

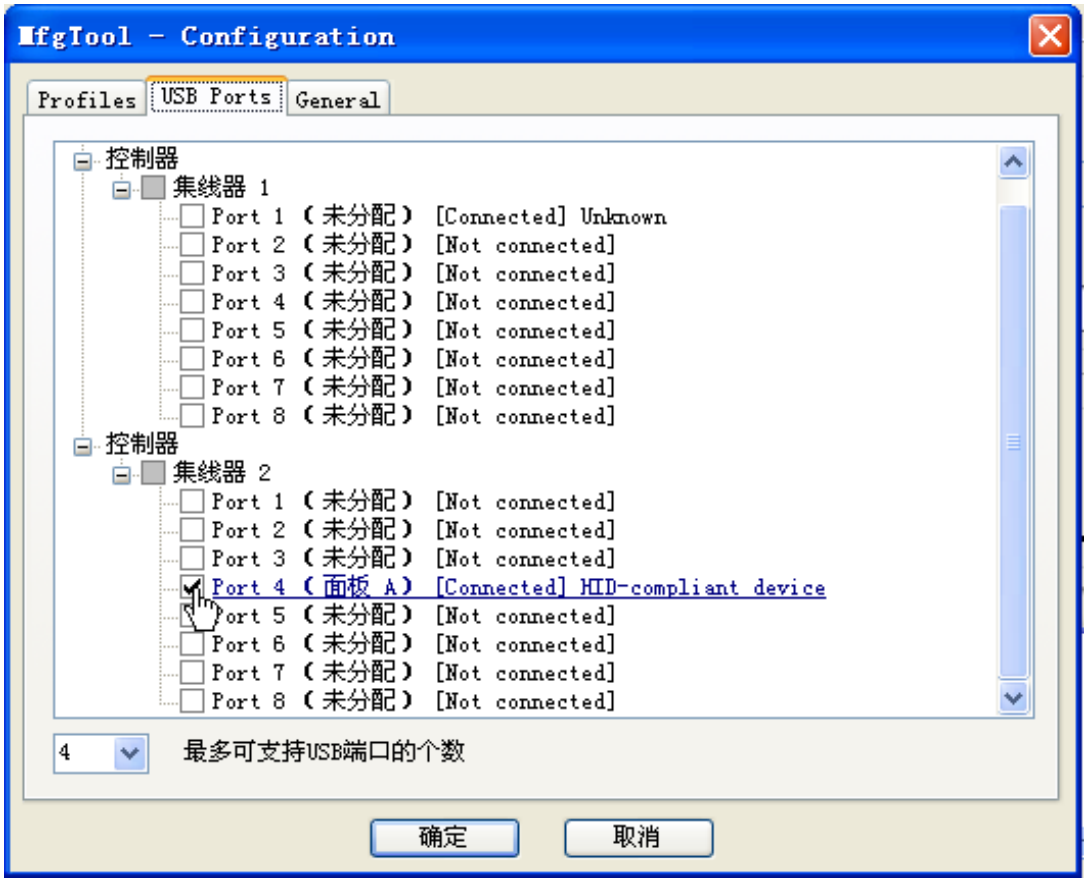

图 3.3　勾选连接的 HID-compliant device

6. 返回到MfgTool主界面后显示软件正在监视HID-compliant device，如图 3.4所示。此时点击“开始”进行u-boot的烧写，如图 3.5所示。直到烧写完成后点击“停止”，如图 3.6所示；

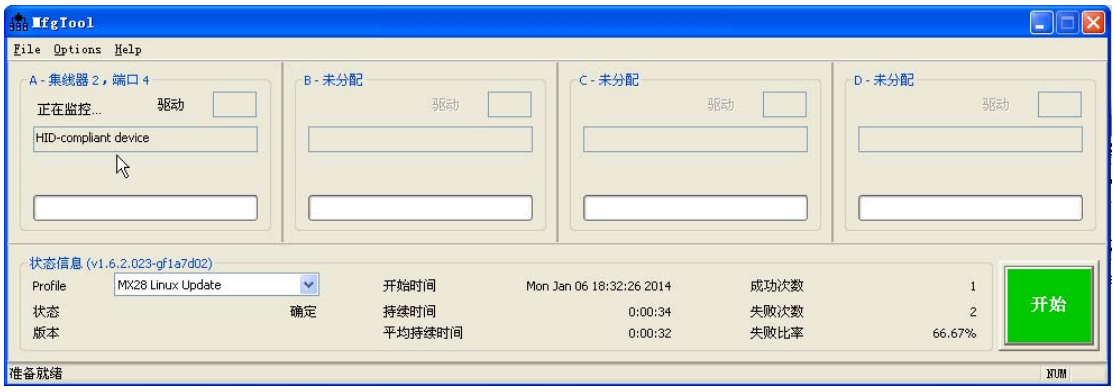

图 3.4　MfgTool 监视 HID-compiant device

图 3.5　u-boot 烧写

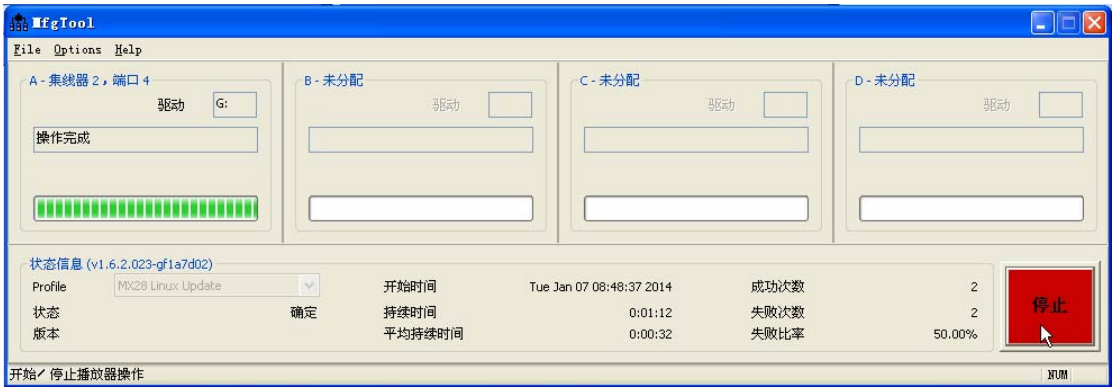

图 3.6　烧写完成

3.1.4 烧写内核与文件系统

1. 断开 JP2（USB_BT）短路器的连接， JP3 保持断开状态（SD：该跳线短接从 SD 卡启动，断开从 nandflash 启动），J7（DUART）接上串口，J10(NET)通过网线与主机连接；

2. 按 RST 键复位或重新上电，系统即从 nandflash 启动，先运行 u-boot，u-boot 在串口终端输出系统的基本信息，然后马上在串口终端界面一直按“空格”键，直到进入 u-boot 的命令行提示符，如下所示；

```
MX28 U-Boot >
```

3. 运行 MfgTool 所在的“Profiles\MX28 Linux Update\OS Firmware\files”目录下的 tftpd32.exe 做 tftp 服务器；

4. 修改 EasyARM-iMX283 本地 ip 和 tftp 服务器 ip（即主机的 ip），并保存（注意本地 ip 与服务器 ip 要在同一个网段）。本示例中主机的 ip 为 192.168.12.46，EasyARM-iMX283 本地的 ip 设置为 192.168.12.62；

```
MX28 U-Boot > setenv ipaddr 192.168.12.62
MX28 U-Boot > setenv serverip 192.168.12.46
MX28 U-Boot > saveenv
Saving Environment to NAND...
Erasing Nand...
Erasing at 0x140000000020000 --      0% complete.
Writing to Nand... done
MX28 U-Boot >
```

5. 执行 run upsystem 下载内核与根文件系统，此时等待内核与文件下载完成，此过程需要几分钟，请耐心等待；

```
MX28 U-Boot > run upsystem
Using FEC0 device
TFTP from server 192.168.12.46; our IP address is 192.168.12.62
Filename 'uImage'.
Load address: 0x42000000
Loading: T #################################################################
done
Bytes transferred = 2567308 (272c8c hex)

NAND erase: device 0 offset 0x200000, size 0x300000
Erasing at 0x4e000001800000 --      0% complete.
OK

NAND write: device 0 offset 0x200000, size 0x300000
  3145728 bytes written: OK

NAND erase: device 0 offset 0xe00000, size 0xf200000
Skipping bad block at    0x0
Erasing at 0xffe000000000000 --      0% complete.
OK
Creating 1 MTD partitions on "nand0":
0xe0000041057d20-0x1000000000000000 : "<NULL>"
UBI: attaching mtd1 to ubi0
UBI: physical eraseblock size:      131072 bytes (128 KiB)
UBI: logical eraseblock size:       126976 bytes
UBI: smallest flash I/O unit:       2048
UBI: VID header offset:             2048 (aligned 2048)
UBI: data offset:                   4096
UBI: empty MTD device detected
UBI: create volume table (copy #1)
UBI: create volume table (copy #2)
UBI: attached mtd1 to ubi0
UBI: MTD device name:                "mtd=3"
UBI: MTD device size:                1041530044416 MiB
UBI: number of good PEBs:            1935
UBI: number of bad PEBs:             1
UBI: max. allowed volumes:           128
UBI: wear-leveling threshold:        4096
UBI: number of internal volumes: 1
UBI: number of user volumes:         0
UBI: available PEBs:                 1912
UBI: total number of reserved PEBs: 23
```

```
UBI: number of PEBs reserved for bad PEB handling: 19
UBI: max/mean erase counter: 0/0
Creating dynamic volume rootfs of size 242778112
Using FEC0 device
TFTP from server 192.168.12.46; our IP address is 192.168.12.62
Filename 'rootfs.ubifs'.
Load address: 0x42000000
Loading:  #################################################################


done
Bytes transferred = 49647616 (2f59000 hex)
Volume "rootfs" found at volume id 0
resetting ...
```

6. 下载完成后将自动写入 nandflash，之后系统自动启动，先运行 u-boot，然后自动启动内核。系统启动完成后，在串口终端输入用户名“root”即可进入系统。

至于如何安装 Linux 下的 tftpd 服务器请参考第一章的“TFTP 服务器”小节。

3.2 系统操作和基本设置

3.2.1 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，具体描述与默认设置如所示表 3.2。

表 3.2 核心板配置信号功能描述

跳线器	标号	功能说明		默认
		短接	断开	
JP2	USB_BT	通过 USB/UART 下载程序	通过 SD 卡/NANDFLASH 启动	断开
JP3	SD	SD 卡启动	NANDFLASH 启动	断开
JP4	ENC（WinCE 系统有效）	厂家保留使用	正常工作模式	断开
JP5	CLR_REG	清除注册表	正常工作模式	断开
JP6[1]	WDO_EN	禁能看门狗输出	使能看门狗输出	短接

[1]出厂提供的系统未提供看门狗喂狗功能，所以需要短接 JP3 禁能看门狗输出，否则系统不断复位重启

3.2.2 系统登陆

EasyARM-iMX283 上电完成启动 Linux 系统后，在串口终端会提示用户输入帐号和密码。。这里帐号和密码都是 root。

同时 EasyARM-iMX283 也提供 SSH 网络登陆，帐号和密码都是 root。在 windows 可以

使用 putty 登陆。在 Linux 主机可以通过 ssh 命令登陆：

```
$ ssh root@xxx.xxx.xxx.xxx     # xxx.xxx.xxx.xxx为EasyARM-iMX283 的IP地址
```

3.2.3 网络设置

使用“ifconfig”命令可以配置 EasyARM-iMX283 的网络 IP 地址。如需要把 IP 地址设置为“192.168.12.61”：

```
# ifconfig eth0 192.168.12.61
```

注意使用该命令设置了 IP 地址后，在系统重启后将不作保持。若把 EasyARM-iMX283 固定某一 IP 地址，请使用 vi 命令把“ifconfig eth0 xxx.xxx.xxx.xxx”添加到/etc/rc.d/init.d/start_userapp 文件中，如下所示：

```
#!/bin/sh
#you can add your app start_command three
ifconfig eth0 192.168.12.62

#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 &
```

3.2.4 RTC时间设置

在 EasyARM-iMX283 并没有使用外部的 RTC 芯片，而是使用了 CPU 中内部 RTC，所以在把时间设置到 RTC 后，当系统断电后，时间将不会保持。若需要保持 RTC 时间在断电后不会丢失，请在 J4 位置装上 3.1~4.2V 的电池。

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 2009-06-24，10:30:10，则可用如下命令：

```
# date 2009.06.24-10:30:10          #设置系统时间
Wed Jun 24 10:30:10 UTC 2009
# hwclock –w                        #将时间写入 RTC
```

设置系统时间后，一定要使用 hwclock –w 将时间写入 RTC。

3.2.5 SD卡使用

把 MiniSD 卡插入 EasyARM-iMX283 的 SD 插槽后，EasyARM-iMX283 将会自动检测到 SD 卡，并自动挂载到/media/ sd-mmcblk**目录下。

注意 SD 卡不具备热插拔功能，需要在系统启动时先插上 SD 卡！

3.2.6 U盘使用

把 U 盘插入到 EasyARM-iMX283 的 USB HOST（J2）上后，EasyARM-iMX283 将会检测到 U 盘，并自动挂载到/media/ usb-sda*目录下。

3.2.7 usb device使用

EasyARM-iMX283 提供的 usb device 接口可以让 EasyARM-iMX283 虚拟成一个 U 盘。请把光盘“3.Linux\4.开发示例\2、功能部件\3、USB_device 接口”目录中的 g_file_storage.ko 文件通过 NFS 方式或其它方式（如 U 盘复制）方式复制到 EasyARM-iMX283 中。然后创建一个 loop 文件，该 loop 文件可以放置在任何目录：

```
# dd if=/dev/zero of=/dev/shm/disk bs=1024 count=10240
```

该命令表示在/dev/shm/目录创一个大小为 10M 的 loop 文件。然后：

```
# insmod g_file_storage.ko stall=0 file=/dev/shm/disk removable=1
```

使用 A-MiniUSB 线连接 EasyARM-iMX283 J12 和 PC 机。这时在 PC 机中“我的电脑”里即可看见多了一个 U 盘的驱动器。这是 EasyARM-iMX283 模拟出来的 U 盘。我们把这个驱动器进行格式化。格式化之后，即可对这个驱动器进行读写。若需要在 EasyARM-iMX283 下看这个虚拟 U 盘写入有什么文件，则请在电脑卸载这个虚拟 U 盘；然后在串口终端把这个 loop 文件挂载到一个指定目录，如/mnt：

```
# mount /dev/shm/disk /mnt
```

这时即可在/mnt/下看到在电脑写入的文件。

3.2.8 LED使用

在 EasyARM-iMX283 上有 POWER、RUN、ERR 三个 LED。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，其操作接口在/sys/class/leds/user-err 文件。在 shell 下可以直接控制 ERR LED 的点亮和熄灭：

```
echo 1 >/sys/class/leds/user-err/brightness          #控制 LED 点亮
echo 0 >/sys/class/leds/user-err/brightness          #控制 LED 熄灭
```

若在 C 语言代码中，也可以使用 system 系统调用直接使用这些命令：

```
system(“echo 1 >/sys/class/leds/user-err/brightness”);
```

或

```
system(“echo 0 >/sys/class/leds/user-err/brightness”);
```

3.2.9 蜂鸣器使用

在 EasyARM-iMX283 上的蜂鸣器控制操作文件是/sys/class/leds/beep/brightness。我们在可以在 shell 中直接控制蜂鸣器：

```
# echo 1 >/sys/class/leds/beep/brightness          #控制蜂鸣器鸣叫
# echo 0 >/sys/class/leds/beep/brightness          #控制蜂鸣器停止鸣叫
```

当然我们也在可以在 C 语言代码中调用 system 系统调用直接使用这些命令：

```
system(“echo 1 >/sys/class/leds/beep/brightness”);
```

或

```
system(“echo 0 >/sys/class/leds/beep/brightness”);
```

3.2.10 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。设置方法是：

```
$ echo 255 > /sys/class/backlight/mxs-bl /brightness
```

这里的默认值为 80。

3.2.11 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命令把启动应用程序的指令或要执行的指令添加到/etc/rc.d/init.d/start_userapp文件中。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是如程序清单 3.1所示。

程序清单 3.1 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 &
```

4. 功能部件编程

本章导读

本章主要介绍在 C 语言环境下对 EasyARM-iMX283 的 ADC、GPIO、串口接口（包括 485 接口）、CAN 编程。因此需要读者有一定的 Linux C 语言应用程序编程基础。

4.1 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 时，需要先将这些 GPIO 通过命令导出。

4.1.1 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 IO。iMX283 部分 BANK 的 GPIO 功能不支持，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，其序号计算方法为：BANK*32+N（BANK 为 GPIO 引脚所在的 BANK，N 为引脚所在的 BANK 的序号），如 P2.4 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@freescale /$ cd /sys/class/gpio/
root@freescale /sys/class/gpio$ ls
export          gpiochip128   gpiochip64    unexport
gpiochip0       gpiochip32    gpiochip96
root@freescale /sys/class/gpio$ echo 68 >export
root@freescale /sys/class/gpio$ ls
export          gpiochip0     gpiochip32    gpiochip96
gpio68          gpiochip128   gpiochip64    unexport root
@freescale /sys/class/gpio$ cd gpio68/
root@freescale /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。

4.1.2 GPIO方向设置

GPIO 导出后默认未输入功能，direction 文件记录 GPIO 引脚的方向，方向查看和设置通过一下命令进行设置：

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

4.1.3 输入电平读取

当 GPIO 为输入功能时，value 文件记录 GPIO 引脚的输入电平状态，通过参考 value 文件读取 GPIO 的输入电平

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

4.1.4 GPIO输出电平控制

当 GPIO 为输入功能时，通过向 value 文件写入 0 或 1 设置输入电平的状态：

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

4.2 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，其操作涉及 DMA 操作，本节不做讲解。ADC0、ADC1、ADC6 三路通道内部含有一个除 2 模拟电路，在未开启除 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，驱动提供了一个读取电池电压的接口，此通道内部有除 4 电路。

4.2.1 ADC驱动模块的加载

ADC 驱动以动态加载模块的形式提供，因此在进行 ADC 测试之前需要先安装驱动模块。先在主机中编译 ADC 驱动模块，生成 lradc.ko 文件，通过 nfs 或 u 盘挂载到开发板后通过 insmod lradc.ko 命令进行驱动的加载，驱动加载后就可以进行 ADC 的测试。

如 lradc 驱动已经挂载到开发板上/mnt/nfs 目录下，驱动加载步骤如下所示：

```
root@freescale /mnt/nfs/EasyARM-iM283/drivers/lradc$ ls
Makefile          built-in.o        lradc.ko          lradc.mod.o       modules.order
Module.symvers    lradc.c           lradc.mod.c       lradc.o
root@freescale /mnt/nfs/EasyARM-iM283/drivers/lradc$ insmod lradc.ko

adc module init!
root@freescale /mnt/nfs/EasyARM-iM283/drivers/lradc$
```

4.2.2 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，读取电压操作如下：

程序清单 4.1 ADC 操作接口

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

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

10：读取 ADC0 电压值

11：读取 ADC1 电压值

16：读取 ADC6 电压值

17：读取电池电压值

21：开启硬件除 2 电路，读取 ADC0 电压值

22：开启硬件除 2 电路，读取 ADC1 电压值

26：开启硬件除 2 电路，读取 ADC6 电压值

4.2.3 计算公式

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

对于命令 17，计算公式为：**V = 4 * 1.85 * (Val / 4096)**;

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

4.2.4 操作示例

程序清单 4.2 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 time = 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(time--){
                sleep(1);

                ioctl(fd, 20, &iRes);                   /*  开启除 2      CH0     */
                val = (iRes * 3.7) / 4096.0;
                printf("CH0:%.2f\t", val);

                ioctl(fd, 11, &iRes);                   /*  不开除 2      CH1     */
                val = (iRes * 1.85) / 4096.0;
                printf("CH1:%.2f\t", val);

                ioctl(fd, 26, &iRes);                   /*  开启除 2      CH6     */
                val = (iRes * 3.7) / 4096.0;
                printf("CH6:%.2f\t", val);
```

```
                ioctl(fd, 17, &iRes);                    /* 电池电压默认除 4      */
                val = (iRes * 7.4) / 4096.0;
                printf("Vbat:%.2f\t", val);

                printf("\n");
        }
        close(fd);
}
```

4.3 串口编程

EasyARM-iMX283 提供 3 路TTL电平的串口接口：UART0、UART1、UART4。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。硬件接口和设备文件节点的对应关系如表 4.1所示，需要注意的是iMX283 处理器没有UART2 和UART3 功能，引出其对应的/dev/ttySP2 和/dev/ttySP3 设备文件节点是不可实现串口通信的。

表 4.1 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

当 UART0 和 UART1 作为 RS485 接口使用时，串口驱动将在收发数据时自动控制 DR 线电平输出，无需用户的预干。

4.3.1 访问串口设备

1. 打开串口设备文件

在使用串口设备之前，我们需要先打开串口设备文件，获得串口设备文件描述符fd。在Linux系统中，串口设备文件名通常是“/dev/ttySPn”（其中n = 0、1、2、3……）。 我们可以使用标准的open函数打开串口设备文件，如程序清单 4.3所示。

程序清单 4.3 打开串口设备

```
fd = open(“/dev/ttySP0”, O_RDWR | O_NOCTTY | O_NDELAY);
if(iFd < 0) {
        perror(cSerialName);
        exit(0);
}
```

2. open选项

当打开串口设备文件时，我们除了用到 O_RDWR 选项标志外，还使用到 O_NOCTTY 和 O_NDELAY 选项标志：

```
fd = open("/dev/ttySP0", O_RDWR | O_NOCTTY | O_NDELAY);
```

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

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

3. 向串口设备写入数据

向串口设备写入数据是相当容易的，仅是使用标准的write系统调用，如程序清单 4.4所示。

程序清单 4.4 向串口设备写入数据

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

4. 从串口设备读取数据

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

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

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

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

这些操作通常在完成 open（带 O_NDELAY 选项）串口设备后执行。

5. 关闭串口设备

关闭串口设备仅需 close 系统调用：

```
close(fd);
```

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

4.3.2 配置串口接口属性

串口设备的波特率、数据位、校验方式等属性，是通过终端接口配置实现的。

1. 终端接口

终端属性用termios结构描述，如程序清单 4.5所示。

程序清单 4.5 termios 结构

```
struct termios {
    tcflag_t    c_cflag                 /*  控制标志          */
     tcflag_t    c_iflag;               /*  输入标志          */
    tcflag_t    c_oflag;                /*  输出标志          */
```

```
    tcflag_t    c_lflag;                    /* 本地标志                 */
    tcflag_t    c_cc[NCCS];                 /* 控制字符                 */
};
```

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

2. 获得和设置属性

使用函数tcgetattr和tcsetattr可以获得或设置termios结构，如程序清单 4.6所示。

程序清单 4.6 设置和获得 termios 结构函数

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

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

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

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

3. 控制标志

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

表 4.2 c_cflag 部分可用选项

标志	说明
CBAUD	波特率位屏蔽
B0	0 位/秒（挂起）
B110	100 位/秒
B134	134 位/秒
B1200	1200 位/秒
B2400	2400 位/秒
B4800	4800 位/秒
B9600	9600 位/秒
B19200	19200 位/秒
B57600	57600 位/秒
B115200	115200 位/秒
B460800	460800 位/秒

续上表

标志	说明
CSIZE	数据位屏蔽
CS5	5 位数据位
CS6	6 位数据位
CS7	7 位数据位
CS8	8 位数据位
CSTOPB	2 位停止位，否则为 1 位
CREAD	启动接收
PARENB	进行奇偶校验
PARODD	奇校验，否则为偶校验
HUPCL	最后关闭时断开
CLOCAL	忽略调制调解器状态行

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

- 标志选项

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

程序清单 4.7例示了怎样使用屏蔽标志或设置一个值。

程序清单 4.7　tcgetattr 和 tcsetattr 实例

```
#include <termios.h>

int main(void)
{
    struct termios term;
    int fd;

    fd = open("/dev/ttymxc1", O_RDWR | O_NOCTTY);
    if(iFd < 0) {
        perror(cSerialName);
        exit(0);
    }

    if (tcgetattr(fd, &term) < 0) {
        printf("tcgetattr error");
        exit(0);
    }

    switch(term.c_cflag & CSIZE) {
    case CS5:
```

```
            printf("5 bits/byte \n");
            break;
        case CS6:
            printf("6 bits/byte \n");
            break;
        case CS7:
            printf("7 bits/byte \n");
            break;
        case CS8:
            printf("8 bits/byte \n");
            break;
        default:
            printf("unknown bits/byte \n");
        }

        term.c_cflag &= ~CSIZE;
        term.c_cflag |= CS8;

        if (tcsetattr(fd, TCSANOW, &term) < 0) {
            printf("tcsetattr error");
            exit(0);
        }

        return 0;
}
```

- 设置波特率

cfsetispeed和cfsetospeed分别用于设置串口的输入和输出波特率，如程序清单 4.8所示。

程序清单 4.8　设置串口输入/输出波特率函数

```
#include <termios.h>
int cfsetispeed(struct termios *termptr, speed_t speed);
int cfsetospeed(struct termios *termptr, speed_t speed);
```

这两个函数若执行成功返回 0，若出错则返回-1。

使用这两个函数时，应当理解输入、输出波特率是存在串口设备termios结构中的。在调用任一cfset函数之前，先要用tcgetattr获得设备的termios结构。与此类似，在调用任一cfset函数后，波特率都被设置到termios结构中。为使用这种更改影响到设备，应当调用tcsetattr函数。操作方法如程序清单 4.9所示。

程序清单 4.9　设置波特率示例

```
if (tcgetattr(fd, &opt) < 0) {
        return ERROR;
}

cfsetispeed(&opt, B9600);
```

```
cfsetospeed(&opt, B9600);

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

- 设置数据位

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

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

- 设置奇偶校验

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

- 无奇偶校验（8N1）

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

- 7 位数据位奇偶校验（7E1）

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

- 奇校验（7O1）

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

4. 本地标志

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

表 4.3 c_lflag 标志

标志	说明
ISIG	启用终端产生的信号
ICANON	启用规范输入
XCASE	规范大/小写表示
ECHO	进行回送
ECHOE	可见擦除字符
ECHOK	回送 kill 符

续上表

标志	说明
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：

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

5. 输入标志

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

表 4.4 c_iflag 标志

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

- 设置输入校验

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

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

这里有个选项值得注意：IGNPAR。 IGNPAR 选项在一些场合的应用带有一定的危险性，它是指示串口驱动程序忽略串口接口接收到的字符奇偶校验出错。也是说，IGNPAR 使

即使奇偶校验出错的字符也通过输入。这在测试通信链路的质量时也许有用，但在通常的数据通信应用中不应使用。

- 设置软件流控制

使用软件流控制是启用 IXON、IXOFF 和 IXANY 选项：

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

相反，要禁用软件流控制是禁止上面的选项：

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

6. 输出标志

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

表 4.5 c_oflag 标志

标志	说明
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
ONOCR	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

启用输出处理是在 c_oflag 成员启用 OPOST 选项：

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

- 使用原始输出

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

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

当 OPOST 被禁止，c_oflag 其它选项也被忽略。

7. 控制字符组

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

表 4.6　c_cc 标志

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

- VMIN 和 VTIME

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

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

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

- 当 MIN > 0，TIME > 0 时

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

- 当 MIN > 0，TIME == 0 时

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

- 当 MIN == 0, TIME > 0 时

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

- 当 MIN == 0，TIME == 0 时

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

8.　实例

程序清单 4.10是原始模式下串口操作的实例。

程序清单 4.10 串口程序示例

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

```
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <errno.h>
#include <limits.h>
#define DEV_NAME                “/dev/mxc3”
int main(void)
{
        int iFd, i;
        int len;
        unsigned char ucBuf[1000];
        struct termios opt;

        iFd = open(DEV_NAME, O_RDWR | O_NOCTTY);
        if(iFd < 0) {
                perror(DEV_NAME);
                return -1;
        }

        tcgetattr(iFd, &opt);
        cfsetispeed(&opt, B115200);
        cfsetospeed(&opt, B115200);

        if (tcgetattr(iFd,    &opt)<0) {
                return    -1;
        }
        opt.c_lflag   &=    ~(ECHO    |    ICANON    |    IEXTEN    |    ISIG);
        opt.c_iflag   &=    ~(BRKINT    |    ICRNL    |    INPCK    |    ISTRIP    |    IXON);
        opt.c_oflag   &=    ~(OPOST);
        opt.c_cflag   &=    ~(CSIZE    |    PARENB);
        opt.c_cflag    |=    CS8;

        opt.c_cc[VMIN]    =    255;
        opt.c_cc[VTIME]   =    150;

        if (tcsetattr(iFd,    TCSANOW,    &opt)<0) {
                return    -1;
        }
        tcflush(iFd,TCIOFLUSH);
        for (i = 0; i < 1000; i++){
                ucBuf[i] = 0xff - i;
        }

        write(iFd, ucBuf, 0xff);
```

```
        len = read(iFd, ucBuf, 0xff);
        printf("get date: %d \n", len);
        for (i = 0; i < len; i++){
                printf(" %x", ucBuf[i]);
        }
        printf("\n");

        close(iFd);

        return 0;
}
```

测试上述代码时，需要把“/dev/mxc3 接口的 RXD 和 TXD 用杜邦线短接起来。当程序执行时，程序在串口把发出去的数据完封不动地读回来，并打印出来。

4.4 I²C接口

EasyARM-iMX283 底板一排针形式引出了I²C1 接口，在核心板上I²C0 接口与DS2460 加密芯片连接（目前提供的Linux系统未支持DS2460 加密功能）。Linux系统实现了I²C0 和I²C1 总线的驱动，用户通过应用程序即可进行I²C总线的通信。

4.4.1 open调用

在使用I²C驱动操作接口前，需要先调用open打开I²C驱动设备文件获得文件描述符。如程序清单 4.11所示。

程序清单 4.11　打开I²C设备文件

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

4.4.2 ioctl调用

当使用I²C驱动操作I²C从机器件时，首先需要把I²C从机器件的I²C从机地址和I²C从机地址长度设置到I²C驱动设备文件接口，使用ioctl调用可以设置此参数。ioctl调用使用到的命令在kernel/linux-2.6.35.3/include/linux/i2c-dev.h头文件中有定义，如程序清单 4.17所示。

程序清单 4.12　ioctl 命令定义

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

```
#define I2C_TIMEOUT         0x0702    /* set timeout in units of 10 ms */

/* NOTE: Slave address is 7 or 10 bits, but 10-bit addresses
 * are NOT supported! (due to code brokenness)
 */
#define I2C_SLAVE           0x0703     /* Use this slave address */
#define I2C_SLAVE_FORCE 0x0706     /* Use this slave address, even if it
                                      is already in use by a driver! */
#define I2C_TENBIT          0x0704    /* 0 for 7 bit addrs, != 0 for 10 bit */

#define I2C_FUNCS           0x0705    /* Get the adapter functionality mask */

#define I2C_RDWR            0x0707    /* Combined R/W transfer (one STOP only) */

#define I2C_PEC             0x0708    /* != 0 to use PEC with SMBus */
#define I2C_SMBUS           0x0720    /* SMBus transfer */
```

1. I2C_TENBIT命令

I2C_TENBIT 宏的定义为：

```
#define I2C_TENBIT  0x0704
```

调用ioctl使用I2C_TENBIT命令，可以设置I^2C从机地址的长度。参数可为 1 或 0：当为 1 时，表示I^2C从机地址长度为 10 位；当为 0 时，表示I^2C从机地址长度为 8 位。

当需要把I^2C从机地址长度设置为 8 位时：

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

2. I2C_SLAVE命令

I2C_SLAVE 宏定义为：

```
#define I2C_SLAVE    0x0703
```

调用ioctl使用I2C_SLAVE命令，可以设置I^2C从机地址，参数就是I^2C从机地址的值。当需要把I^2C从机地址设置为 0xA0 时，示例代码如下：

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

4.4.3 write调用

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

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

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

4.4.4 read调用

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

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

当read调用完成后，I^2C主机向I^2C从机器件发出总线始起信号；在读取数据之前会先发出通过ioctl设置的从机地址，从机器件在接收到从机地址后返回ACK信号，然后向I^2C主机

发送数据，驱动程序将接收到的数据存入buf中，最后发出I²C总线结束信号。

对于类似于eeprom之类具有子地址的I²C接口的器件，在发送或读取数据之前需要先发送I²C子地址。eeprom的读写操作例程请参考开发示例中的I²C接口部分的代码。

对于读取有子地址的器件的数据，在读取前需要先发送子地址：

```
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
```

4.4.5 close调用

当I²C驱动操作完成后，请使用close调用关闭之前打开的I²C驱动设备文件：

```
close(fd);
```

4.5 PWM接口

EasyARM-iMX283 底板引出了 PWM_3、PWM_4、PWM7 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。

4.5.1 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 PWM 的计数周期和设置占空比，也可以通过应用程序进行操作。

4.5.2 系统命令操作示例

命令清单 4.13 读取和设置 PWM 信息

```
root@freescale ~$ cd /sys/class/backlight/
root@freescale /sys/class/backlight$ ls
easy283-pwm.4    easy283-pwm.7    mxs-bl
root@freescale /sys/class/backlight$ cd easy283-pwm.4
root@freescale /sys/devices/platform/easy283-pwm.4/backlight/easy283-pwm.4$ cat max_brightness
400
root@freescale /sys/devices/platform/easy283-pwm.4/backlight/easy283-pwm.4$ cat brightness
200
root@freescale /sys/devices/platform/easy283-pwm.4/backlight/easy283-pwm.4$ echo 300 >brightness
period 4 = 400,intensity=300
root@freescale /sys/devices/platform/easy283-pwm.4/backlight/easy283-pwm.4$ cat brightness
300
root@freescale /sys/devices/platform/easy283-pwm.4/backlight/easy283-pwm.4$
```

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 300 >brightness 设置 PWM 的占空比；实际输出的 PWM 波形的占空比为 brightness / max_brightness ，本例子中为 300/400。

4.5.3 应用程序操作示例

程序清单 4.14 PWM 操作示例

```
int main(int args,char* arg[])
{
        char cmd_string[]="echo 10 >
/sys/devices/platform/easy283-pwm.4/backlight/easy283-pwm.4/brightness ";

        return system(cmd_string);

}
```

编译应用程序后直接运行就能设置 PWM_4 通道的占空比输出为 2.5%。

4.6 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，EasyARM-iM283 的 Linux SPI 驱动为 SS0 提供了 spidev1.0 设备文件接口。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。

4.6.1 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 4.15所示。

程序清单 4.15 打开 SPI 设备文件

```
fd = open(device, O_RDWR);
    if (fd < 0)
        pabort("can't open device");
```

4.6.2 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SPI总线的极性和相位

Linux SPI驱动提供了SPI_IOC_WR_MODE命令用于设置SPI总线的极性和相位，如表4.7所示。

表 4.7 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。
命令参数说明	命令参数可以选择：SPI_MODE_0、SPI_MODE_1、SPI_MODE_2、SPI_MODE_3。 至于 SPI 总线极性和相位的 4 种模式请参考 SPI 协议。

续上表

返回值说明	0：设置成功 1：设置不成功
errno 说明	-EINVAL：不允许设置
特别说明和注意点	无

2. 读取SPI总线的极性和相位

Linux 的SPI驱动提供了SPI_IOC_RD_MODE命令用于读取SPI总线的极性和相位设置模式，如表 4.7所示。

表 4.8 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
命令参数说明	参数的返回值为：SPI_MODE_0、SPI_MODE_1、SPI_MODE_2、SPI_MODE_3。 至于 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。
errno 说明	无
特别说明和注意点	无

3. 设置SPI总线上每字的数据位长度

Linux的SPI驱动提供了SPI_IOC_WR_BITS_PER_WORD命令用于设置SPI总线上每字的数据位长度，如表 4.9所示。

表 4.9 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	取值可在 1~N 之间
返回值说明	恒为 0：读取成功。

续上表

errno 说明	无
特别说明和注意点	无

4. 设置最大总线速率命令

Linux的SPI驱动提供了SPI_IOC_WR_MAX_SPEED_HZ命令用于设置SPI总线的最大速率，如表 4.10所示。

表 4.10 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	设置总线的最大速率
命令参数说明	取值可在 1~N 之间
返回值说明	恒为 0：读取成功。
errno 说明	无
特别说明和注意点	无

5. 数据接收/发送操作命令

Linux的SPI驱动提供了SPI_IOC_MESSAGE(1)命令用于实现在SPI总线接收/发送数据操作，如表 4.11所示。

表 4.11 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	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
errno 说明	无
特别说明和注意点	无

使用SPI_IOC_MESSAGE(1)命令进行接收/发送的数据都需要使用struct spi_ioc_transfer结构体来封装，该结构体的定义如程序清单 4.16所示。

程序清单 4.16 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;
}
```

在 struct spi_ioc_transfer 结构体的 len 是指 tx_buf 和 rx_buf 指向缓冲区中的据是同样的长度。在 speed_hz 成员设置的总线速率不能大于使用 SPI_IOC_WR_MAX_SPEED_HZ 命令设置的总线速率。当只需要发送数据时，需要把要发送的数据填充到 tx_buf 指向的缓冲区，rx_buf 指向的缓冲区只需填充全 0 即可；当只需要接收数据时，需要在 tx_buf 指向的缓冲区全填充 0，接收到的数据会返回在 rx_buf 指向的缓冲区；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 只能设置有个有效，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。

4.6.3 示例代码

程序清单 4.17所示的代码是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

程序清单 4.17 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]))

static void pabort(const char *s)
{
```

```
        perror(s);
        abort();
}

static const char *device = "/dev/spidev1.0";
static uint8_t mode = 0;
static uint8_t bits = 8;
static uint32_t speed = 50000;
static uint16_t delay;
#define WRITE    0
static void transfer(int fd)
{
        int ret;
        int i = 10000;
        uint8_t tx[] = {
                0x9f, 0x00, 0x00, 0x00, 0x01, 0x02,
        };
        uint8_t rx[ARRAY_SIZE(tx)] = {0, };
        struct spi_ioc_transfer tr_txrx[] = {
                {
                        .tx_buf = (unsigned long)tx,                    /* 发送数据缓存区                  */
                        .rx_buf = 0,                                    /* 半双工通信，接收缓存区置为 0    */
                        .len = 1,                                       /*  发送命令码，1 个字节           */

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

            ret = ioctl(fd, SPI_IOC_MESSAGE(2), &tr_txrx[0]);           /*  发送 2 条消息                  */
            if (ret == 1) {
                        pabort("can't revieve spi message");
        }

        for (ret = 0; ret < tr_txrx[1].len; ret++) {
                if (!(ret % 6))
```

```
                puts("");
            printf("%.2X ", rx[ret]);
        }
        puts("");
}

void print_usage(const char *prog)
{
        printf("Usage: %s [-DsbdlHOLC3]\n", prog);
        puts("    -D --device     device to use (default /dev/spidev1.0)\n"
             "    -s --speed      max speed (Hz)\n"
             "    -d --delay      delay (usec)\n"
             "    -b --bpw        bits per word \n"
             "    -l --loop       loopback\n"
             "    -H --cpha       clock phase\n"
             "    -O --cpol       clock polarity\n"
             "    -L --lsb        least significant bit first\n"
             "    -C --cs-high    chip select active high\n"
             "    -3 --3wire      SI/SO signals shared\n");
        exit(1);
}

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

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

```
            break;

        switch (c) {
        case 'D':
            device = optarg;
            break;
        case 's':
            speed = atoi(optarg);
            break;
        case 'd':
            delay = atoi(optarg);
            break;
        case 'b':
            bits = atoi(optarg);
            break;
        case 'l':
            mode |= SPI_LOOP;
            break;
        case 'H':
            mode |= SPI_CPHA;
            break;
        case 'O':
            mode |= SPI_CPOL;
            break;
        case 'L':
            mode |= SPI_LSB_FIRST;
            break;
        case 'C':
            mode |= SPI_CS_HIGH;
            break;
        case '3':
            mode |= SPI_3WIRE;
            break;
        case 'N':
            mode |= SPI_NO_CS;
            break;
        case 'R':
            mode |= SPI_READY;
            break;
        default:
            print_usage(argv[0]);
            break;
        }
    }
```

```
}


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

        parse_opts(argc, argv);

        fd = open(device, O_RDWR);
        if (fd < 0)
                pabort("can't open device");

        /*
         * spi mode
         */
        ret = ioctl(fd, SPI_IOC_WR_MODE, &mode);
        if (ret == -1)
                pabort("can't set wr spi mode");

        ret = ioctl(fd, SPI_IOC_RD_MODE, &mode);
        if (ret == -1)
                pabort("can't get spi mode");

        /*
         * bits per word
         */
        ret = ioctl(fd, SPI_IOC_WR_BITS_PER_WORD, &bits);
        if (ret == -1)
                pabort("can't set bits per word");

        ret = ioctl(fd, SPI_IOC_RD_BITS_PER_WORD, &bits);
        if (ret == -1)
                pabort("can't get bits per word");

        /*
         * max speed hz
         */
        ret = ioctl(fd, SPI_IOC_WR_MAX_SPEED_HZ, &speed);
```

```
    if (ret == -1)
        pabort("can't set max speed hz");

    ret = ioctl(fd, SPI_IOC_RD_MAX_SPEED_HZ, &speed);
    if (ret == -1)
        pabort("can't get max speed hz");

    printf("spi mode: %d\n", mode);
    printf("bits per word: %d\n", bits);
    printf("max speed: %d Hz (%d KHz)\n", speed, speed/1000);
    sleep (1);
    transfer(fd);

    close(fd);

    return ret;
}
```

5. EasyARM-iMX283 的bootloader

本章导读

本章主要介绍 EasyARM-iMX283 的 Boot。EasyARM-iMX283 是使用了 U-Boot 作为 bootloader。若 EasyARM-iMX283 设置了 nand flash 启动方式（断开 JP2（USB_BT）短路器的连接， JP3 保持断开状态（SD，短接从 SD 卡启动，断开从 nandflash 启动）），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。

5.1 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。

5.1.1 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）。几个重要的目录的说明如表 5.1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 5.1 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、nios2 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

5.2 编译u-boot

请把光盘中“3.Linux\6.源代码”目录的 bootloader_EasyARM-iMX283.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_EasyARM-iMX283.tar.bz2
```

然后得到一个 bootloader 目录。在 bootloader 目录下有 u-boot-2009.08 目录和 imx-bootlets-src-10.12.01 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 nandflash 的文件）。

1. 生成u-boot文件

进入 u-boot-2009.08 目录：

```
$ cd bootloader/ u-boot-2009.08
```

清除原有的编译文件：

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

配置平台：

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

执行编译：

```
$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能在 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有电源设置的 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
```

进入 imx-bootlets-src-10.12.01 目录，首先将目录下的 elftosb 复制到/usr/bin/，然后执行编译命令：

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

编译完成后imx-bootlets-src-10.12.01 目录下的imx28_ivt_uboot.sb文件就是我们可以烧写到NAND FLASH的固件文件。具体的烧写方法请参考3.1.2小节“烧写u-boot”的内容。

5.3 U-Boot基本命令

在 U-Boot 启动阶段，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 MiB
```

```
In:       serial
Out:      serial
Err:      serial
Hit any key to stop autoboot:  0
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。

```
U-Boot$?  或者
U-Boot$ help
?         - alias for 'help'
base      - print or set address offset
bdinfo    - print Board Info structure
bootm     - boot application image from memory
bootp     - boot image via network using BootP/TFTP protocol
cmp       - memory compare
cp        - memory copy
crc32     - checksum calculation
dcache    - enable or disable data cache
dhcp      - invoke DHCP client to obtain IP/boot params
echo      - echo args to console
go        - start application at address 'addr'
help      - print online help
icache    - enable or disable instruction cache
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
mm        - memory modify (auto-incrementing)
mtest     - simple RAM test
mw        - memory write (fill)
nand      - NAND sub-system
nboot     - boot from NAND device
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
saves     - save S-Record file over serial line
setenv    - set environment variables
tftpboot  - boot image via network using TFTP protocol
version   - print monitor version
```

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

```
U-Boot$ help nand
nand info                    - show available NAND devices
nand device [dev]            - show or set current device
nand read[.jffs2]            - addr off|partition size
nand write[.jffs2]           - addr off|partition size - read/write `size' bytes starting
                               at offset `off' to/from memory address `addr'
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 at offset (UNSAFE)
nand biterr off              - make a bit error at offset (UNSAFE)
nand lock [tight] [status]   - bring nand to lock state or display locked pages
nand unlock [offset] [size]  - unlock section
```

5.3.1 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，我们为 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 擦除并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：

```
upsafefs= 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)
```

5.3.2 通过网络启动内核

当我们把内核的固件文件 uImage 放在 PC 机的 tftp 服务器的根目录时，我们就可以通过网络来启动内核，方便内核调试。假设 PC 的 IP 为 192.168.12.61，EasyARM-iMX283 的 IP 可以设置为 192.168.12.62，那么在 u-boot 执行下面指令：

```
MX28 U-Boot > setenv ipaddr 192.168.12.62
MX28 U-Boot > setenv serverip 192.168.12.61
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 SDRAM，然后在 SDRAM 中启动内核。

5.4 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，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/
```

6. Linux内核编译和驱动要点

本章导读

阅读本章需要读者有一定的 Linux 内核移植和驱动开发经验。

6.1 编译内核

参考5.4“u-boot Tools”的内容准备好mkimage文件，并复制到/usr/bin/目录下。

请把光盘中的“3.Linux\6.源代码\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，进入该目录：

```
$ cd linux-2.6.35.3
```

输入“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。

当然输入“make menuconfig”命令可以进入内核配置界面。

6.2 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 6.1所示。

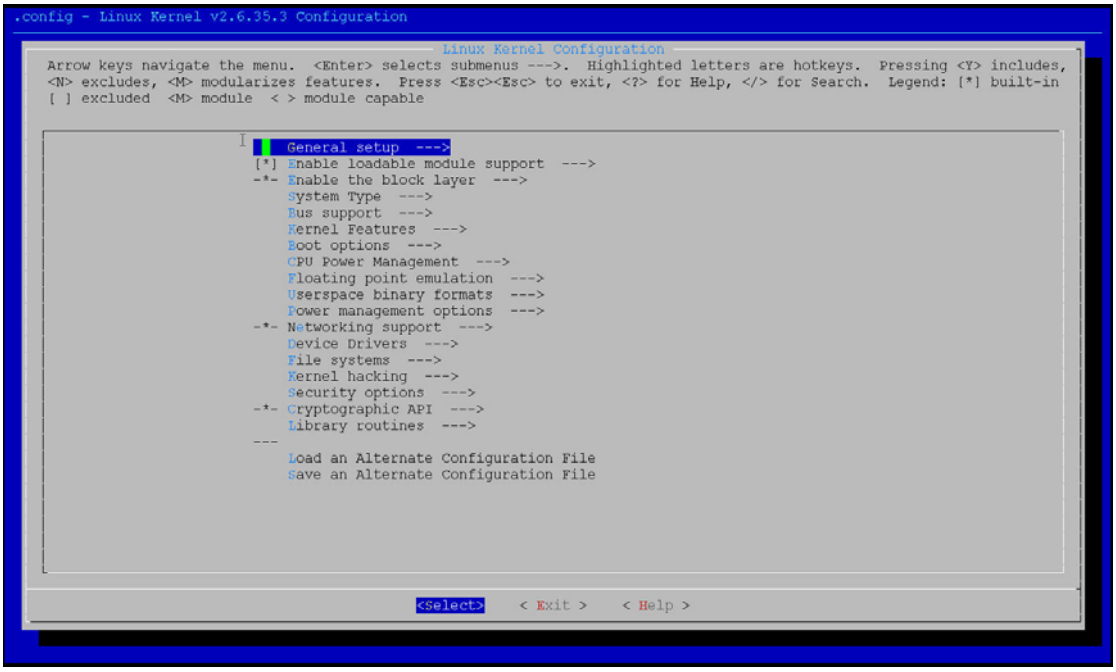

图 6.1 内核配置主菜单

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，如下图所示。

当在该项健入“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。

1. 启用SD卡驱动

请按以下方法配置内核模块：

```
Device Drivers  →
    [*] Block devices →                #启用块设备模块
    <*> MMC/SD/SDIO card support →
            [*]   Assume MMC/SD cards are non-removable (DANGEROUS)
            <*>   MMC block device driver
            [*]     Use bounce buffer for simple hosts
            <*>   MXS MMC support
```

2. 启用USB host

请按以下方法配置内核模块：

```
Device Drivers  →
  [*] USB support →
        <*> Support for HOST-side USB
      [*] USB runtime power management (suspend/resume and wakeup)
              <*>       EHCI HCD (USB 2.0) support
               [*]        Support for Freescale controller
               [*]          Support for Host1 port on Freescale controller
               [*]          Support for DR host port on Freescale controller
               [*]   Root Hub Transaction Translators
               <*>   USB Mass Storage support
```

3. 启用I^2C

请按以下方法配置内核模块：

```
Device Drivers  →
      <*> I2C support →
          [*] Enable compatibility bits for old usr-space
          <* > I2C device interface
           [*] Autoselect pertinent helper modules
           I2C Hardware BUS support →
        <*> MXS I2C Support
                [*]   Enable I2C0 module
                <*>     MXS I2C0 PIOQUEUE MODE Support
                [*]   Enable I2C1 module
                 <*>     MXS I2C1 PIOQUEUE MODE Support
```

6.3 内核GPIO使用方法

i.MX283 CPU 的大多功能引脚是实现了功能复用，当需要把某一引脚作 GPIO 功能使用时，我们需要先把该引脚配置成 GPIO 工作模式。

这里我们以 GPIO1_17 作例子，讲述如何把指定的引脚配置成 GPIO 工作模式，并操作 GPIO。

从 IMX28 芯片手册中可以得知 GPIO1_17 的引脚名字是 LCD_D17，内核代码的 /arch/arm/mach-mx23/mx23_pins.h 文件中查阅到对 PINID_LCD_D17 引脚的定义：

```
#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。

如把 LCD_D17 引脚配置成 GPIO 工作模式，则对 PINID_LCD_D17 的描述做如下的修改，则可设置成 GPIO 工作模式，并同时设置其属性（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）：

```
{
        .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 6.1所示。PIN_GPIO的值为 3，表示引脚的模式为工作模式 3。查阅IMX28RM芯片手册可以知道 LCD_D17 引脚的工作模式 3 是GPIO功能模式。

程序清单 6.1 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。

6.4 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 6.2所示。

程序清单 6.2 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_HB_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 6.3所示，用于设置屏幕的分辨率、像素时钟。

程序清单 6.3 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 6.2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 6.2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=10^12/(( 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
```

7. 嵌入式Linux根文件系统

7.1 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：

1）基本的目录结构：/bin、/sbin、/dev、/etc、/lib、/var、/proc、/sys、/tmp 等。

2）基本程序运行所需的库文件，如 glibc 等。

3）基本的系统配置文件，如 inittab、rc 等。

4）必要的设备文件，如/dev/ttyS0、/dev/console 等。

5）基本应用程序，如 sh、ls、cd、mv 等。

7.2 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。

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。

7.2.1 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 7.1所列。

表 7.1 FHS 顶层目录

目录	说明
bin	基本的的命令二进制文件（所有用户可用），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（可选）
sbin	基本的系统命令二进制文件（仅 root 用户可用）
srv	系统服务的一些数据
tmp	临时文件
usr	该目录有二级标准

续上表

目录	说明
var	可变数据

7.2.2 /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也有相应的规定，如表 7.2 所列。

表 7.2 /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（安装完毕后为空）
sbin	不是至关重要的系统命令（仅 root 用户可用）
share	独立数据
X11R6	X-Window 系统（可选）
games	游戏和教育相关的二进制文件
src	源代码（可选）

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。

7.3 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），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。

7.4 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 nfsroot=\$(serverip):\$(rootpath) ip=\$(ipaddr):\$(serverip):\$(gatewayip):\$(netmask)
```

```
:\$(hostname)::off
```

其中个参数的意义如下：

- 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。

下面是一个实际设置范例，NFS 服务器 IP 为 192.168.7.239，NFS 根文件系统路径为/root/nfsroot，目标板 IP 为 192.168.7.236：

```
"root=/dev/nfsroot rw console=ttyS0,115200 nfsroot=192.168.7.239:/root/nfsroot
ip=192.168.7.236:192.168.7.1:192.168.7.1:255.255.255.0:epc.zlgmcu.com:eth0:off"
```

光盘附带了一个根文件系统压缩包，将其解压并将根文件系统目录设置为 NFS 共享目录，然后在 EasyARM-iMX283 的 u-boot 中设置内核启动参数从 NFS 启动即可使用:

```
 MX28 U-Boot > setenv bootargs "root=/dev/nfsroot rw console=ttyS0,115200
nfsroot=192.168.7.239:/root/nfsroot ip=192.168.7.236:192.168.7.1:192.168.7.1:255.255.255.0:epc.zlgmcu.com:
eth0:off"
```

7.5 生成文件系统映像

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格式，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，所以 UBIFS 文件系统的稳定性值得我们相信。

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 240MiB；页大小为 2048 字节（2KiB）；擦除块大小为 128KiB。

（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\5、文件系统”目录下有 mkfs.ubifs、ubinize 程序。请把这两个程序复制到 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：

```
[proton@localhost ~]$sudo chmod 777 /usr/sbin/mkfs.ubifs
[proton@localhost ~]$ 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，也可以参考我们光盘上提供的根文件系统。如果使用我们提供的根文件系统，请把光盘上的 rootfs 目录下的 rootfs_qt.tar.bz2 文件复制到 Linux 主机中自己的工作目录下。然后解压 rootfs_qt.tar.bz2 文件：

```
[proton@localhost ~]$ tar –jxvf    rootfs_qt.tar.bz2
[proton@localhost ~]$ cd rootfs_qt
```

（3）生成 ubi 根文件系统

在 rootfs 和 ubinize.cfg 所在的目录下执行下面的指令：

```
[proton@localhost ~]$    mkfs.ubifs -r rootfs -m 2048 -e 129024 -c 1920 -o rootfs.ubifs
[proton@localhost ~]$    ubinize -o ubi.img -m 2048 -p 128KiB -s 512 ubinize.cfg
```

这时所得到的 rootfs.ubifs 文件就是我们所需的 ubi 根文件系统映像。

8. 嵌入式Linux Qt编程指南

本章主要目的是，介绍如何使用 EasyARM-iMX283 嵌入式 Linux 的 Qt 库进行图形程序的开发。

8.1 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。

1. C++ 基础知识，了解简单的类，继承，重载等面向对象概念。
2. linux 基础知识，了解基本的 shell 命令，懂得对 linux 进行简单的配置。
3. 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。
4. 交叉编译与动态库的基础知识。

8.2 Qt介绍

8.2.1 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）开发维护，现在被nokia公司收购。目前在nokia的推动下，Qt的发展非常快速，版本不断更新。目前最新的Qt主版本为 4.8，所支持的平台如图 8.1所示：

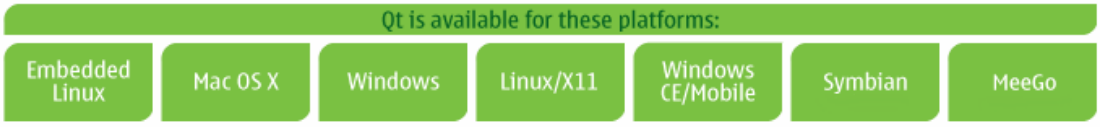

图 8.1 Qt 支持的平台

8.2.2 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 8.2所示：

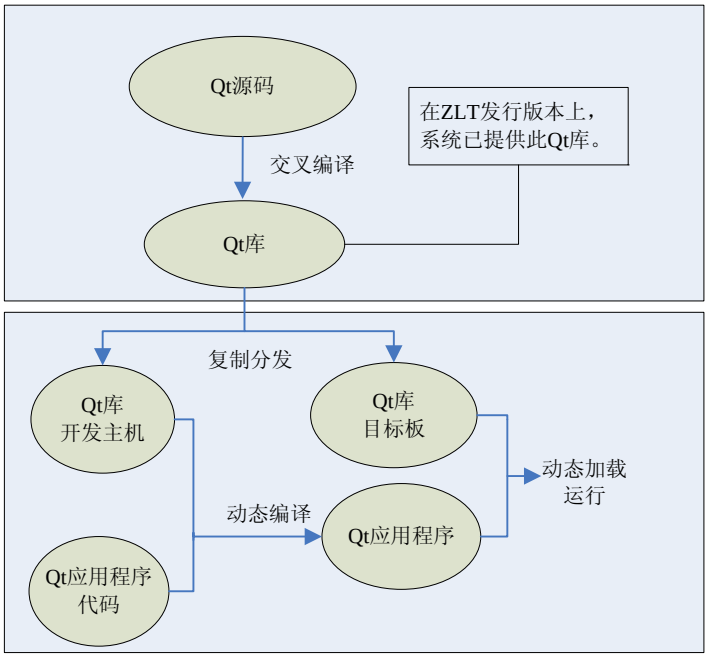

图 8.2　Qt/E 编程图示

使用 EasyARM-iMX283 提供的 Qt 平台，用户无需编译 Qt 库，系统已经将 Qt 源码交叉编译成库。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。

8.3　编译环境的搭建

光盘附带的交叉编译器中已经集成了 QT 库。用户只需按第 1 章描述的方法安装交叉编译器即可。

8.4　Hello　world程序开发

8.4.1 编译hello程序

下面介绍如何开发一个简单的hello world程序。其流程如图 8.3：

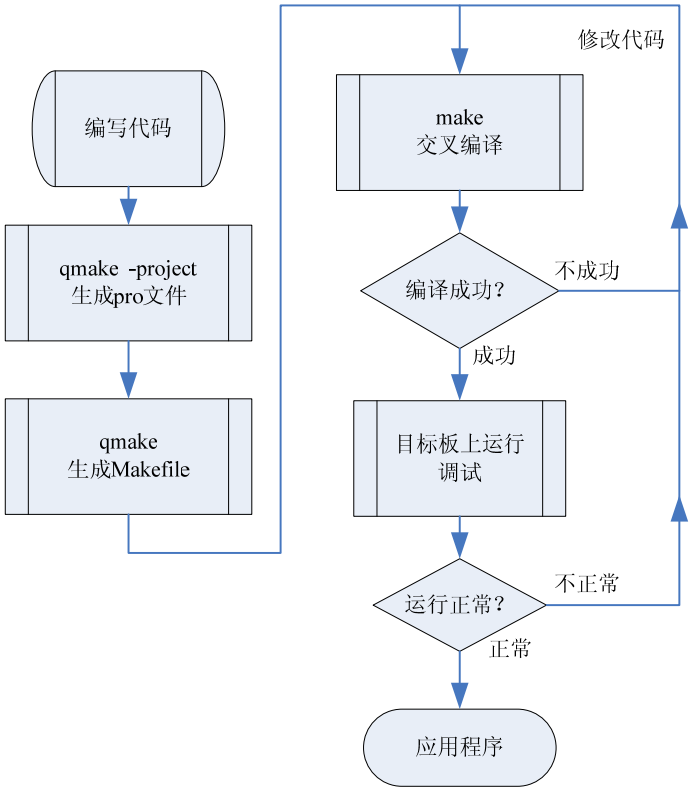

图 8.3　hello 开发流程图

hello.cpp程序代码如程序清单 8.1所示：

程序清单 8.1　hello 程序代码

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

将 hello.cpp 拷贝至 hello 目录下，运行如下命令：

```
qmake   -project
```

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

```
qmake
```

根据上一步的 pro 文件，生成 Makefile 文件。

```
make
```

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

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

执行程序。

8.4.2 在目标板上运行hello程序

下面介绍如何在目标板上运行 hello 程序。

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

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

```
export QWS_MOUSE_PROTO=Tslib:/dev/input/event0
```

上述命令中 Tslib 指定了触摸屏对应的设备文件，这里指定为/dev/input/event0。但其值并不固定，需要根据实际情况确定。正常情况下，所需的设备文件位于/dev/input 目录下。在命令行下输入如下命令：

```
cat /dev/input/event0 | hexdump
```

点击触摸屏，如果有数据输出，那么对应的设备文件可能就是所需的设备文件。

如果需要使用 usb 鼠标，可如下设置：

```
export QWS_MOUSE_PROTO=LinuxInput:/dev/input/event1
```

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

在成功设定鼠标设备后，可以如下启动 Qt 程序。

```
./hello   -qws
```

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

为了方便用户使用，我们提供一个启动脚本代码。该脚本在运行时会自动检测触摸屏、鼠标和键盘的设备文件，并自动设置环境变量。脚本代码如程序清单 8.2所示。

程序清单 8.2　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
        export TSLIB_PLUGINDIR=/usr/lib/ts/
        export TSLIB_CONFFILE=/etc/ts.conf
        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

# add your qt programme start command
./qt_hellow –qws       #请修改这里，改成自己的程序
```

在本例中，程序启动成功后，界面如图 8.4所示：

图 8.4　hello 程序运行界面

8.5　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。

8.5.1 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。

假设我们的项目中有如下源代码文件：

- hello.cpp
- hello.h

- main.cpp

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

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

对于每一个 cpp 文件，都需要如此指定。代码如下：

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

也可以通过反斜线形式指定：

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

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

```
HEADERS += hello.h
SOURCES += hello.cpp
SOURCES += main.cpp
```

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

```
TARGET = helloworld
```

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

```
CONFIG += qt
HEADERS += hello.h
SOURCES += hello.cpp
SOURCES += main.cpp
TARGET = helloworld
```

现在就可以利用此 pro 文件生成 Makefile，命令如下：

```
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。

8.5.2 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

CONFIG 变量用于控制编译过程中的各个方面。常用参数如下：

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

8.6　Qt编程简单入门

8.6.1 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。学习利用signal和slot（信号与槽）的方法使控件同步。

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；他们是QWidget的children。反过来，我们也可以称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 8.3：

程序清单 8.3　Qt 例程代码

```
1  #include <QApplication>
2  #include <QHBoxLayout>
3  #include <QSlider>
4  #include <QLCDNumber>
5  int main(int argc, char *argv[])
6  {
7          QApplication app(argc, argv);
8          QWidget *window = new QWidget;
9          window->setWindowTitle("Enter Your Age");
10         QLCDNumber * lcd = new QLCDNumber;
11         QSlider *slider = new QSlider(Qt::Horizontal);
12         slider->setRange(0, 99);
13         QObject::connect(slider, SIGNAL(valueChanged(int)),
14                                  lcd, SLOT(display(int)));
15        slider->setValue(35);
16         QHBoxLayout *layout = new QHBoxLayout;
17         layout->addWidget(lcd);
```

```
18        layout->addWidget(slider);
19        window->setLayout(layout);
20        window->show();
21        return app.exec();
22    }
```

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：

- QHBoxLayout：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。

现在，虽然我们还没有看见lcd和slider控件的大小和位置，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。

8.6.2 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，我们连接了信号和槽，发送了信号，触发槽函数的响应。现在来更深入了解这个机制。

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。SIGNALE()和SLOT()宏的作用是把他们转换成字符串。

在目前的例子中，我们已经连接了信号和槽。实际使用中还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 8.5所示：

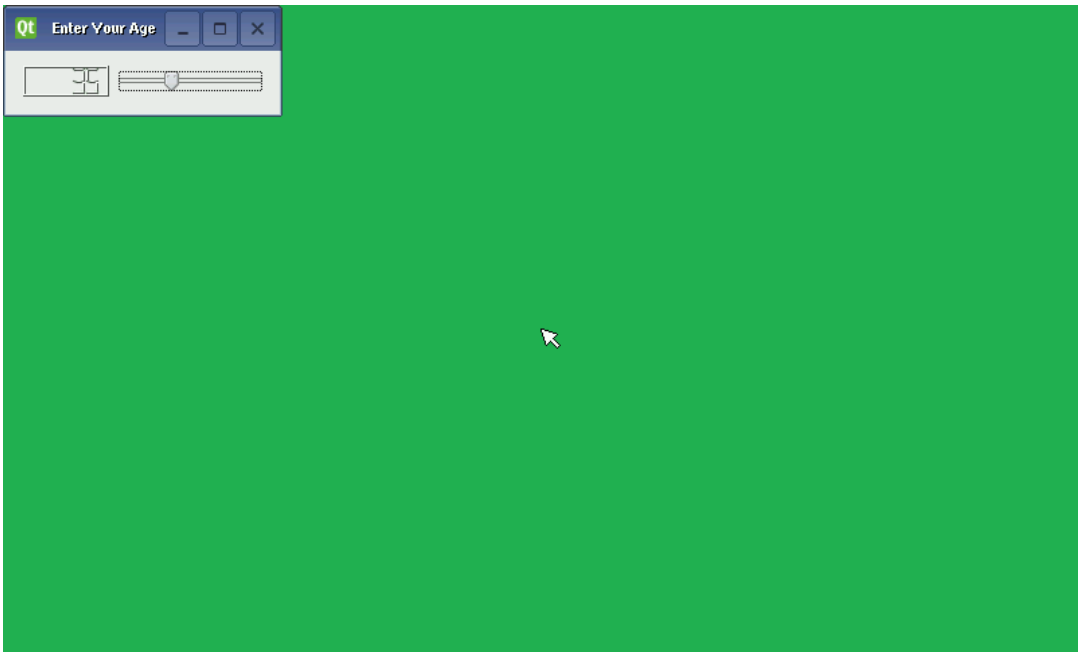

图 8.5 Qt 例程界面

8.7 桌面版本的Qt SDK使用简介

8.7.1 桌面版本Qt SDK简介

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

桌面版本的 Qt SDK 主要包括以下两个部分：

- 用于桌面版本的 Qt 库
- Qt Creator（集成开发环境）

Qt Creator是一个强大的跨平台IDE，集编辑，编译，运行，调试功能于一体。其代码编辑器支持关键字高亮，上下文信息提示，自动完成，智能重命名等高级功能。IDE中集成的可视化界面编辑器，可以让用户以所见即所得的方式进行图形程序的设计。其编译，运行无需敲入命令，直接点击按钮或使用快捷键即可完成。同时还支持图形化的调试方式，可以以插入断点，单步运行，追踪变量，查看函数堆栈等方式进行应用程序的调试开发。Qt Creator 主界面如图 8.6所示。

图 8.6　Qt Creator 主界面

8.7.2 桌面版本Qt SDK的安装

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

```
sudo apt-get install qt-sdk
```

8.7.3 Qt Creator配置

通过如下命令启动 Qt Creator。

```
qtcreator
```

将得到如图 8.6界面。如已安装交叉编译工具链，则此时系统中同时存在两个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。点击如图 8.6菜单栏上“工具”——>“选项”。得如类似图 8.7界面：

图 8.7　选项界面

在打开“选项”对话框后，点击“构建和运行”的选项，然后选择“Qt 版本”标签。然后添加一项“/opt/freescale/usr/local/gcc-4.1.2-glibc-2.5-nptl-3/arm-none-linux-gnueabi/bin/qmake”的 qmake 路径。

8.7.4 Qt Creator使用例程

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

单击界面中“创建项目”按钮，得到如图 8.8界面：

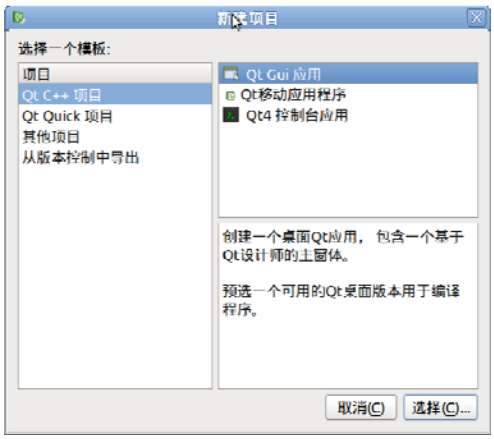

图 8.8　新建项目界面

如图 8.8所示进行项目模板的选择，使用默认的“Qt Gui应用”模板。点击选择，进入如图 8.9界面，设置项目名称与路径。接下来一路点击“下一步”直到如图 8.10界面。

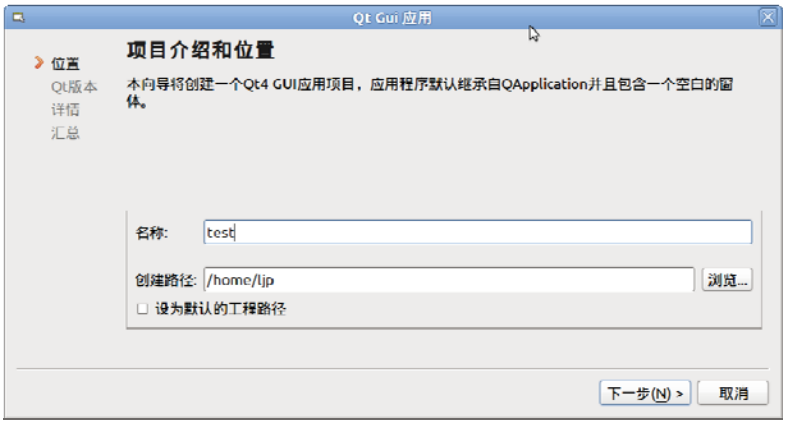

图 8.9　项目介绍和位置界面

图 8.10 项目管理界面

单击“完成”后，将进入如图 8.11界面。可以看到，Qt Creator已经自动生成一个最基本Qt程序所需的代码，用户可以在此基础上做进一步的开发。

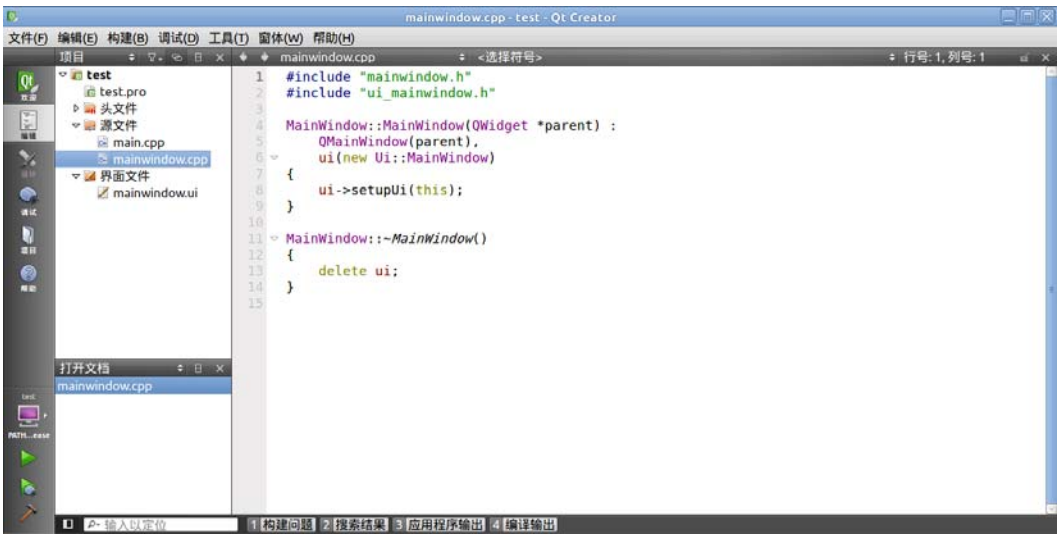

图 8.11　mainwindow.cpp 界面

单击项目侧边栏中mainwindow.ui可以启动可视化编辑器，如图 8.12所示。

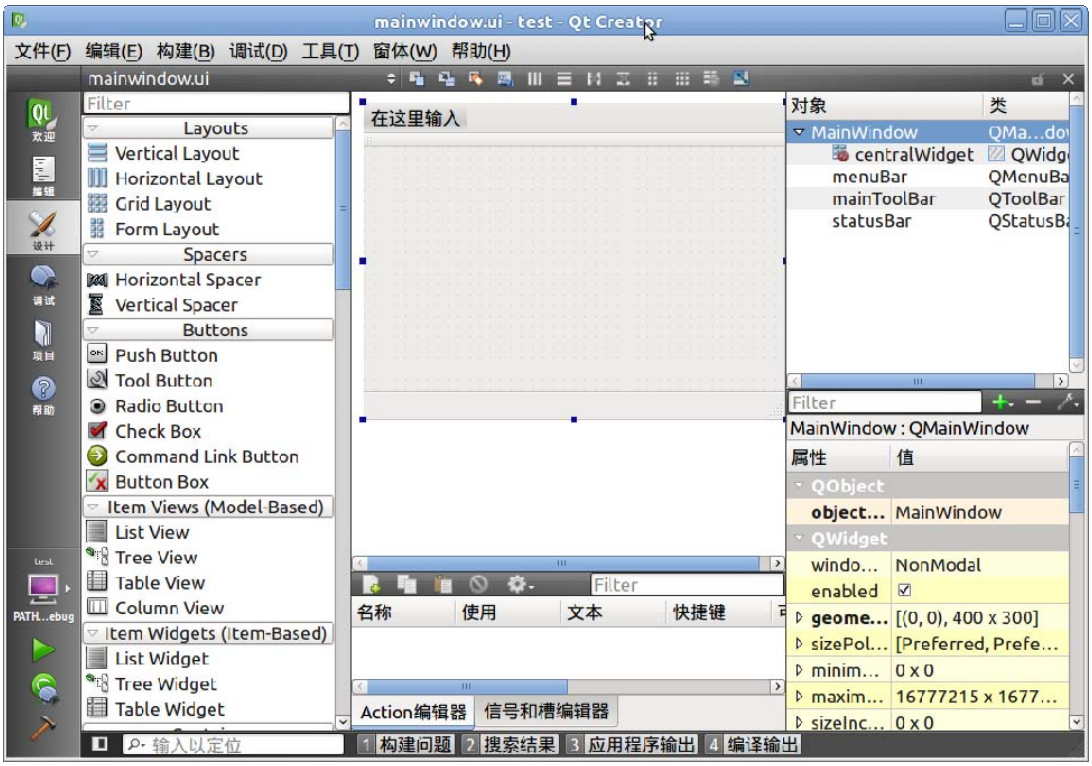

图 8.12　可视化界面编辑器

在图 8.12的界面中，通过拖动控件侧边栏中控件到程序主页面中，以所见即所得的方式设计程序界面。下面拖动一个QLabel控件到程序主页面上，并设置QLabel上文字为“Hello World”。如图 8.13：

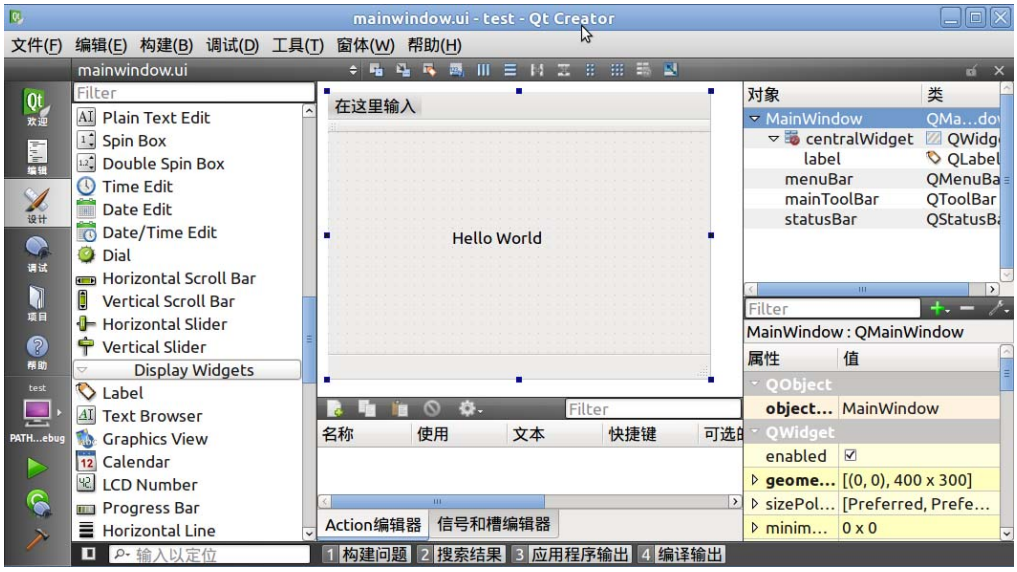

图 8.13　hello world 程序界面

接下来通过侧边栏上  选择前面所设置桌面版本的Qt。如图 8.14所示：

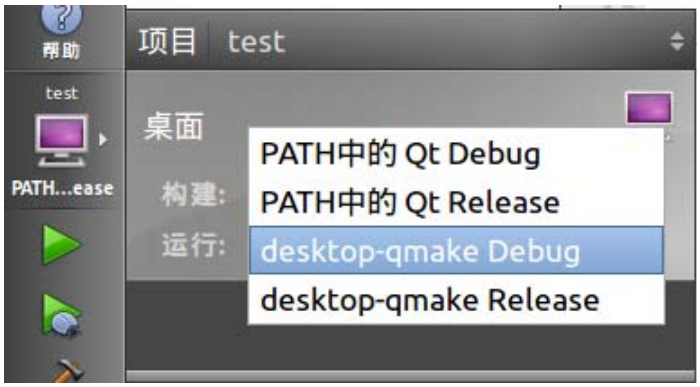

图 8.14　选择 Qt 版本

最后，点击  按钮对程序进行编译链接运行。如编译无误，将自动启动应用程序。界面如图 8.15所示：

图 8.15　hello world 界面

8.7.5 移植hello world

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

```
cd  /home/ljp/test
qmake -project
qmake
make
```

8.8 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 8.16。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。

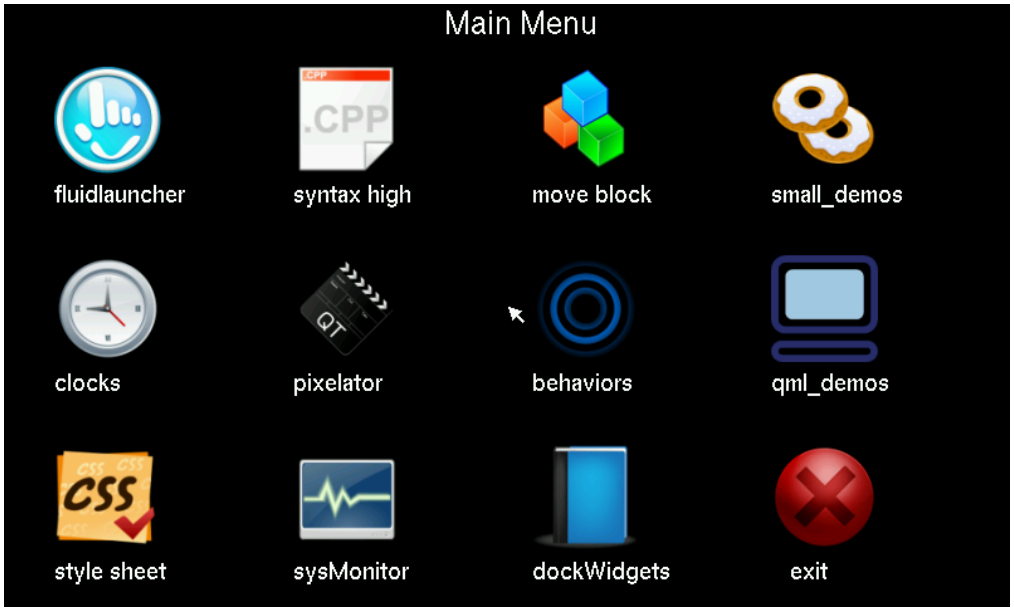

图 8.16　zylauncher 界面

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 8.17所示：

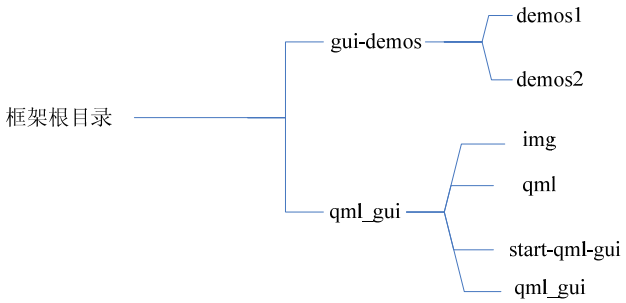

图 8.17　zylaucher 框架结构

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 8.16。MainMenu.qml代码如程序清单 8.4所示：

程序清单 8.4　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"
    }
}
```

下来关注代码：

```
    ProButton {
                x:Logic.indexToX(0);    y:Logic.indexToY(0);
                imgPath: "../img/switch.png";
```

```
        imgText : "fluidlauncher";
        execText : "../gui-demos/fluidlauncher/start-fluidlauncher"
    }
```

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

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

imgPath: 指定按钮的图标文件。

imgText: 指定按钮下方的描述文字。

execText: 指定按钮对应的演示程序可执行文件。

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

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

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

接着关注代码：

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

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

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

最后关注的代码：

```
    ExitButton{
        x:Logic.indexToX(3); y:Logic.indexToY(2);
        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。

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
电话：(020)38730916　38730917　38730972　38730976　38730977

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

沈阳办事处

E-mail：sales.shenyang@zlgmcu.com