# 目　录

| Dev_Team | 西邮 Linux 兴趣小组 嵌入式 Linux 组 | | | | |
|---|---|---|---|---|---|
| Change | 许振文 | | | | |
| Members | 许振文 | | | | |
| M_num | M_date | M_version | M_part | M_reason | M_author |
| 001 | 2008-06- | 0.01 | 第 1,2 章 | 创建文档 | 许振文 |
| 002 | 2008-07- | 0.02 | 第 5 章 | 创建第 5 章 | 许振文 |
| 003 | 2008-09 | 0.03 | 第 3 章 | 创建第 3 章 | 许振文 |
| 004 | 2008-10 | 0.04 | 1,2,3,5 章 | 完善所有已创建章节 | 许振文 |
| 005 | 2008-12 | 0.05 | 第 8 章 | 创建第八章 | 许振文 |
| 006 | 2008-02 | 0.06 | 6,7 章 | 创建 6,7 章 | 许振文 |
| 007 | 2008-04 | 0.07 | 所有 | 完善所有已创建章节 | 许振文 |
| 008 | 2008-09 | 0.08 | 所有 | 创建第 4 章，完善所有 | 许振文 |
| | | | | | |

注：

M_num：修改编号，从 001 开始，执行“ +1” 操作

M_dat：修改日期

M_v er si on：修改后版本

M_part：修改了那部分

M_r eas on：修改说明

M_author：修改者

# 第1章 Bootloader 的使用

## 1.1 BootLoader 简介

嵌入式系统中，BootLoader 的作用与 PC 机上的 BIOS 类似，通过 BootlLoader 可以完成对系统板上的主要部件如 CPU，SDRAM，Flash，串行口等进行初始化，也可以下载文件到系统板上，对 Flash 进行擦除与编程。当运行操作系统时，它会在操作系统内核运行之前运行，通过它可以分配内存空间的映射，从而将系统的软硬件环境带到一个合适的状态，以便为最终调用操作系统准备好正确的环境。现在在arm上使用的比较多的有u-boot、vivi 等。以 u-boot 使用得多一些。

## 1.2 U-Boot

最早，DENX 软件工程中心的 Wolfgang Denk 基于 8xxrom 的源码创建了 PPCBOOT 工程，并且不断添加处理器的支持。后来，Sysgo Gmbh 把ppcboot移植到 ARM 平台上，创建了 ARMboot 工程。然后以 ppcboot 工程和 armboot 工程为基础，创建了 u-boot 工程。

现在u-boot 已经能够支持 PowerPC、ARM、X86、MIPS 体系结构的上百种开发板，已经成为功能最多、灵活性最强并且开发最积极的开放源码 Bootloader。目前仍然由 DENX 的 Wolfgang Denk 维护。

u-boot 的源码包可以从sourceforge 网站下载，还可以订阅该网站活跃的u-boot Users 邮件论坛，这个邮件论坛对于 u-boot 的开发和使用都很有帮助。

u-boot 软件包下载网站： http://sourceforge.net/projects/u-boot

u-boot 邮件列表网站： http://lists.sourceforge.net/lists/listinfo/u-boot-users/

DENX 相关的网站： http://www.denx.de/re/DPLG.html

## 1.3 Vivi

VIVI 是韩国 MIZI 公司为其开发的 SMDK2410 开发板编写的一款引导程序。VIVI 是 CPU 加电后运行的第一段程序 ,其基本功能是初始化硬件设备,建立内存空间的映射图 ,从而为调用嵌入式 Linux 内核做好准备。VIVI 由 2 部分组成:一部分是依赖于 CPU 体系结构的代码 ,用汇编语言实现对硬件环境的初始化 ,并为第二部分代码的执行做好准备;另一部分是用 C 语言实现内存空间的映射,并将内存映像和根文件系统映像从 FLASH 中读取到 RAM 中，设置好启动参数后，启动内核。不过已经做好移植的只有 SA-1110, S3C2400, S3C2410, PXA250 Processors 下地址下载：http://www.mizi.com/developer/s3c2410x/download/vivi.html

然后需修改/vivi/Makefile 里的一些变量设置：

LINUX_INCLUDE_DIR = /kernel/include/ （LINUX_INCLUDE_DIR 为 kernel/include 的对应目录，我的是/home/chen/kerne-h2410eb /include/）

因此修改为：

LINUX_INCLUDE_DIR = /home/chenjun/ kerne-h2410eb/include/

CROSS_COMPILE = /usr/local/arm/2.95.3/bin/arm-linux- (CROSS_COMPILE 为 arm-linux 安装的相应目录，我的是/usr/local/arm/2.95.3/bin/arm-linux-)

因此修改为：

CROSS_COMPILE = /usr/local/arm/2.95.3/bin/arm-linux-

ARM_GCC_LIBS = /usr/local/arm/2.95.3/lib/gcc-lib/arm-linux/2.95.3 （需根据你 arm-linux 的安装目录修改，我的是/usr/local/arm/2.95.3/lib/gcc-lib/arm-linux/2.95.3）

进入/vivi 目录执行 make distclean。（目的是确保编译的有效性，在编译之前将 vivi 里所有的“*.o”和“*.o.flag”文件删掉）

进入/vivi 目录里，输入“make menuconfig”，开始选择配置。可以 Load 一个写好的配置文件也可以自己修改试试。注意 Exit 时一定要选“Yes”保存配置。

再输入“make”正式开始编译，一会儿就完了。如果不报错，在/vivi 里面就有你自己的“vivi”了。这个就是后面要烧写到 flash 中的 bootloader。修改 Malkefile,指定交叉编译：

ARCH = arm

CROSS_COMPILE=/opt/crosstool/gcc-3.3.6-glibc-2.3.2/arm-linux/bin/arm-linux-

(2)修改/arch/s3c2410/smdk..c 文件,按照自己所指定大小指定 NAND 分区。

(3)编译生成 VIVI

# 第2章搭建 linux－arm 运行环境

## 2.1 软件环境安装与设置

### 2.1.1 Linux 操作系统

Debian，ubuntu 或是其它 linux 环境都可以，操作系统的安装这里就不做介绍了。

### 2.1.2 交叉编译工具 arm-linux-的安装与使用

还有一种编译工具 arm-elf-，这种编译工具主要是针对于 uclinux 的编译，而 arm +linux 的编译一般使用 arm-linux-。另外 arm-linux-使用的 C 库是 glibc，而 arm-elf-用的 C 库是 newlibc,ulibc 等。arm-linux-下载网址：
http://www.handhelds.org/download/projects/toolchain/ 。或是从 ftp.arm.kernel.org.uk 网站上直接下载。下载 arm-linux-工具包到本地文件夹，并解压，如下图：

```
helight@helight-desktop:~$ ls arm-linux-gcc-3.4.1.tar.bz2
arm-linux-gcc-3.4.1.tar.bz2
helight@helight-desktop:~$ tar jxvf arm-linux-gcc-3.4.1.tar.bz2
```

图2.1

```
helight@helight-desktop:~$ cd usr/local/arm/3.4.1/
arm-linux/ include/   lib/       man/
bin/       info/      libexec/   tmp/
helight@helight-desktop:~$ cd usr/local/arm/3.4.1/
helight@helight-desktop:~/usr/local/arm/3.4.1$ pwd
/home/helight/usr/local/arm/3.4.1
helight@helight-desktop:~/usr/local/arm/3.4.1$
```

解压后的文件夹为 usr/local/arm/3.4.1/,如下图所示：

图2.2

只需把 arm 这个文件夹移到/usr/local/下就可以了。

```
$sudo mv usr/local/arm /usr/local/
```

设置 path 路径：使用 vim 打开/etc/profile 文件,在”export PATH”之前添加 “PATH =$PATH:/usr/local/arm/3.4.1/bin/”,然后重新启动计算机即可.在重新启动计算机后在终端键入“arm-l”然后按“tab”键看安装是否正确。如下图所示：

```
helight@helight-desktop:~$ arm-linux-
arm-linux-addr2line  arm-linux-gcc      arm-linux-objdump
arm-linux-ar         arm-linux-gcc-3.4.1 arm-linux-ranlib
arm-linux-as         arm-linux-gccbug   arm-linux-readelf
arm-linux-c++        arm-linux-gcov     arm-linux-size
arm-linux-c++filt    arm-linux-ld       arm-linux-strings
arm-linux-cpp        arm-linux-nm       arm-linux-strip
arm-linux-g++        arm-linux-objcopy
helight@helight-desktop:~$ arm-linux-
```

图2.3

使用 arm-linux-工具来编译调试 c 程序，程序如下:

```
/*main.c*/
#include <stdio.h>
int main()
{
    printf("hello world!\n");
    return 0;
}
$arm-linux-gcc -o main main.c
```

然后使用 file 命令查看编译后的目标文件的文件类型。

```
helight@helight-desktop:~/linux/linux-test/test$ arm-linux-gcc -o main main.c
helight@helight-desktop:~/linux/linux-test/test$ ls
main  main.c
helight@helight-desktop:~/linux/linux-test/test$ file main
main: ELF 32-bit LSB executable, ARM, version 1, for GNU/Linux 2.4.3, dynamically linked (uses shared libs),
not stripped
```

```
helight@helight-desktop:~/linux/linux-test/test$ arm-linux-gcc -o main main.c
helight@helight-desktop:~/linux/linux-test/test$ ls
main  main.c
helight@helight-desktop:~/linux/linux-test/test$ file main
main: ELF 32-bit LSB executable, ARM, version 1, for GNU/Linux 2.4.3, dynamicall
y linked (uses shared libs), not stripped
helight@helight-desktop:~/linux/linux-test/test$
```

如下图：

图 2.4

还可以使用 arm-linux-objdump 反汇编，不过这里的汇编代码是 arm 中的汇编指令，和平时我们学的汇编指令还是有点区别的。

```
helight@helight-desktop:~/linux/linux-test/test$ arm-linux-objdump -d main
main:     file format elf32-littlearm
Disassembly of section .init:
00008284 <_init>:
    8284: e52de004   str    lr, [sp, #-4]!
    8288: eb000021   bl     8314 <call_gmon_start>
    828c: eb000043   bl     83a0 <frame_dummy>
    8290: eb00007c   bl     8488 <__do_global_ctors_aux>
    8294: e49df004   ldr    pc, [sp], #4
Disassembly of section .plt:
```

## 2.2 linux 编译配置

### 2.2.1 基本配置

内核源码使用 2.6.14 ,首先进入到解压后的源码文件夹中编辑主 Makefile 文件。在主 Makefile 文件里主要修改这两行。

```
ARCH        ?= $(SUBARCH)
CROSS_COMPILE    ?=
```

修改后为：

```
ARCH            = arm
CROSS_COMPILE    =arm-linux-
```

在使用 make menuconfig 进入内核配置菜单，在 system type 中选择配置目标板上的 arm cpu 类型，并作对 arm cpu 的配置，当然还有其它的配置选项，建议在第一次配置是可使用系统的默认配置，默认配置文件在“arch/arm/configs/” 文件夹中。编译时只要把它 copy 到内核根目源码录下的”.config“即可。

```
helight@helight:linux-2.6.14$  ls arch/arm/configs/
assabet_defconfig          hackkit_defconfig          mainstone_defconfig
badge4_defconfig           integrator_defconfig       mx1ads_defconfig
bast_defconfig             iq31244_defconfig          neponset_defconfig
cerfcube_defconfig         iq80321_defconfig          netwinder_defconfig
clps7500_defconfig         iq80331_defconfig          omap_h2_1610_defconfig
collie_defconfig           iq80332_defconfig          pleb_defconfig
corgi_defconfig            ixdp2400_defconfig         poodle_defconfig
ebsa110_defconfig          ixdp2401_defconfig         pxa255-idp_defconfig
edb7211_defconfig          ixdp2800_defconfig         rpc_defconfig
enp2611_defconfig          ixdp2801_defconfig         s3c2410_defconfig
ep80219_defconfig          ixp4xx_defconfig           shannon_defconfig
epxa10db_defconfig         jornada720_defconfig       shark_defconfig
footbridge_defconfig       lart_defconfig             simpad_defconfig
fortunet_defconfig         lpd7a400_defconfig         smdk2410_defconfig
h3600_defconfig            lpd7a404_defconfig         spitz_defconfig
h7201_defconfig            lubbock_defconfig          versatile_defconfig
h7202_defconfig            lusl7200_defconfig
helight@helight:linux-2.6.14$ cp arch/arm/configs/s3c2410_defconfig  ./.config
```

在”make menuconfig“时需要 ncurses 头文件，需要安装 ncurses 开发包，本人使用的系统是 Debian，所以直接使用“sudo apt-get install libncurses5-dev”即可安装。

### 2.2.2 添加 devfs

为了我们的内核支持 devfs 以及在启动时并在/sbin/init 运行之前能自动挂载/dev 为 devfs 文件系统,修改 fs/Kconfig

文件找到 menu "Pseudo filesystems"添加如下语句:

```
helight@helight:linux-2.6.14$ vim fs/Kconfig
config DEVFS_FS
        bool "/dev file system support (OBSOLETE)"
        default y

config DEVFS_MOUNT
        bool "Automatically mount at boot"
        default y
        depends on DEVFS_FS
```

此时在“make menuconfig”的时候就会出现一下的选项：

```
File systems  --->
    Pseudo filesystems  --->
        [*] /dev file system support (OBSOLETE)
        [*]    Automatically mount at boot
```

### 2.2.3 添加 kernel 参数

1. 如果是命令参数问题，则作如下修改：

注释掉 arch/arm/kernel/setup.c 文件中的parse_tag_cmdline()函数中的strlcpy()函数，这样就可以使用默认的 CONFIG_CMDLINE 了，在.config 文件中它被定义为：

root=/dev/mtdblock2 ro init=/linuxrc console=ttySAC0 devfs=mount

上面内容也要视具体情况而定，在内核配置文件的 Boot options 中填入也可。

2. 出现如下错误：VFS:Cannot open root device "mtdblock/2" or unknown-block(0,0)

Please append a correct "root=" boot option.

Kernel panic not syncing,VFS Unable to mount root fs on unknown-block(0,0)

原因分析：出现这种情况一般有两种情况：没有文件系统或 boot options 参数不正确，如我的系统文件系统为 cramfs 格式，存放文件系统的分区为第三个分区，启动参数为：

```
noinitrd root=/dev/mtdblock2 ro init=/linuxrc console=ttySAC0,115200  rootfstype=cramfs  mem=64M
```

3. 关于文件系统问题：在有些资料上提到在配置内核时选上 devfs，如果不存在时则手动在文件里加上。但我在内核里面和 busybox-1.10.1 中都未加 devfs 支持，文件系统运行正常。在 linux-2.6.14.1 及以后版本中已去掉 devfs。

### 2.2.4 2410 的内核配置

修改 arch\arm\mach-s3c2410\devs.c 文件

增加头文件定义

```
/**** add by helight ********/
#include <linux/mtd/partitions.h>
#include <linux/mtd/nand.h>
#include <asm/arch/nand.h>
/*** end add by helight *****/
```

增加 nand flash 分区信息(分区信息和 bootloader 程序一致就可以)

```
/******** add by helight **********/
static struct mtd_partition partition_info[] = {
        [0] = {
                name: "boot",
                size: 0x20000,
                offset: 0x0
                },
        [1] = {
                name: "kernel",
                size: 0x300000,
                offset: 0x20000
                },
```

```
        [2] = {
                name: "rootfs",
                size: 0x500000,
                offset: 0x320000
                },
        [3] = {
                name: "etc",
                size: 0x100000,
                offset: 0x820000
                },
        [4] = {
                name: "user",
                size: 0xb00000,
                offset: 0x920000
                },
        [5] = {
                name: "qt",
                size: 0x2b00000,
                offset: 0x1420000
                },
        };
struct s3c2410_nand_set nandset = {
        nr_partitions: 6,
        partitions: partition_info,
};

struct s3c2410_platform_nand superlpplatform = {
        tacls:0,
        twrph0:30,
        twrph1:0,
        sets: &nandset,
        nr_sets: 1,
};
/********** end add **********/

struct platform_device s3c_device_nand = {
        .name                   = "s3c2410-nand",
        .id                     = -1,
        .num_resources          = ARRAY_SIZE(s3c_nand_resource),
        .resource               = s3c_nand_resource,
/******** add by helight **********/
        .dev = {
                .platform_data = &superlpplatform,
        }
```

```
/********** end add **********/
};
```

修改 arch\arm\mach-s3c2410\mach-smdk2410.c 文件

```
static struct map_desc smdk2410_iodesc[] __initdata = {
/*add by helight*/
        {vSMDK2410_ETH_IO, pSMDK2410_ETH_IO, SZ_1M, MT_DEVICE},
 /* end add by helight    */
};
static struct platform_device *smdk2410_devices[] __initdata= {
        &s3c_device_usb,
        &s3c_device_lcd,
        &s3c_device_wdt,
        &s3c_device_i2c,
        &s3c_device_iis,
        &s3c_device_nand, //add by helight
};
```

## 2.3 内核的编译

完成后通过输入命令：make mrproper 来清除冗余文件。make 即可在/arch/arm/boot 下生成所需要的内核文件 zImage。如图：

```
helight@helight:linux-2.6.14$ ls arch/arm/boot/ -l
total 5420
drwxr-xr-x 2 helight helight    4096 2005-10-28 08:02 bootp
drwxr-xr-x 2 helight helight    4096 2008-12-03 22:18 compressed
-rwxr-xr-x 1 helight helight 3708128 2008-12-03 22:18 Image
-rw-r--r-- 1 helight helight    1326 2005-10-28 08:02 install.sh
-rw-r--r-- 1 helight helight    2405 2005-10-28 08:02 Makefile
-rwxr-xr-x 1 helight helight 1818100 2008-12-03 22:18 zImage
helight@helight:linux-2.6.14$
```

image 是正常大小的映像文件，而 zImage 是经过压缩后的映像文件。

另外可加载 module 的的安装也须设置路径，因为这些模块是要在目标板上运行的内核要加载的。在内核配置时有 INSRALL_MOD_PATH=$TARGETDIR 的来设置。

## 2.4 Ramdisk 文件系统的制作

可以利用工具软件 BusyBox 制作 Ramdisk 文件系统.busybox 是一个集成了一百多个最常用 linux 命令和工具的软件,他甚至还集成了一个 http 服务器和一个 telnet 服务器,而所有这一切功能却只有区区 1M 左右的大小.完整的 BusyBox 源代码可以从 http://www.busybox.net 下载,压缩包大小为 1.3 MB 左右。下面是如何使用编译 BusyBox 的过程。

下载 busybox 到本地文件夹并解压。

```
#tar jxvf busybox-1.10.1.tar.bz2
# cd busybox-1.10.1/
```

修改 Makefile

```
ARCH            =arm
CROSS_COMPILE   =arm-linux-
```

```
$make menuconfig
```

Busybox Settings >

General Configuration >

[*] Support for devfs

Build Options >

[*] Build BusyBox as a static binary (no shared libs)

/* 将 busybox 编译为静态连接，少了启动时找动态库的麻烦 */

Installation Options

Don't use /usr

Init Utilities >

[*] init

[*] Support reading an inittab file

/* 支持 init 读取/etc/inittab 配置文件，一定要选上 */

其他选项都是一些 linux 基本命令选项,自己需要哪些命令就编译进去,一般用默认的就可以了.由于库的问题可能有些命令编译不过去，现在的办法只是在配置的时候不选这些命令就可以了，具体需要编译时看，在我编译的时候出现了 taskset 和 insmod_main 编译错误的问题。我分别取消了 taskset 命令和对 2.2 和 2.4 模块 insmod 支持就可以了。taskset 在 Miscellaneous Utilities 下，对 2.2 和 2.4 模块 insmod 在 Linux Module Utilities 下。其它类似的错误可以安同样的方法处理，目前还没有找到错误的缘由。

建立根文件系统结构

```
#mkdir rootfs
#cd rootfs
#mkdir bin dev etc lib proc sbin tmp usr var
#chmod 777 tmp
#mkdir usr/bin usr/lib usr/sbin
#mkdir var/lib var/lock var/log var/run var/tmp
#chmod 1777 var/tmp
```

准备所需的设备文件，可以直接拷贝宿主机上的，或者自建几个就是。

```
#cd rootfs/dev
#mknod -m 660 console c 5 1
```

创建 linuxrc 文件 内容如下:

```
$ vim rootfs/linuxrc
#!/bin/sh
echo "Hello linux--helight"
echo "mount /proc as proc"
/bin/mount -n -t proc none /proc
echo "mount /sys as sysfs"
```

```
/bin/mount -n -t sysfs none /sys

/bin/mkdir -p /var/tmp
/bin/mkdir -p /var/run
/bin/mkdir -p /var/log
/bin/mkdir -p /var/lock
/bin/mkdir -p /var/empty

exec /sbin/init
```

为 mdev 的运行准备环境

mdev 需要改写/dev 和/sys 两个目录。所以必须保证这两个目录是可写的(一般会用到 sysfs,tmpfs。所以要重新编译内核)。 然后在你的启动脚本文件中加入

/bin/mdev -s

然后修改权限：chmod 775 linuxrc

当然，lib 里面还要拷入一些库文件，为了方便，我将交叉编译的库全放进去。

```
#cp -rfd /usr/local/arm/3.4.1/arm-linux/lib/*./lib  (注意-d，保持库文件的链接关系)
#mkdir initrd
#dd if=/dev/zero of=initrd.img bs=1k count=8192
#/sbin/mke2fs -F -v -m0 initrd.img
#mount -o loop initrd.img initrd
#cp -avd rootfs/* initrd
#umount
#gzip -9 initrd.img
```

## 2.5 Cramfs 压缩的文件系统

Cramfs 被设计为简单的较小的可压缩的文件系统，它主要用于较小 ROM 的嵌入式系统，它是一个压缩的文件系统，它并不需要一次性的将文件系统中的所有内容都解压到内存中，而只是在系统需要访问某个位置的数据时，马上计算出该数据在 Cramfs 中的位置，将其实时地解压到内存中，然后通过对内存的访问来获取文件系统中需要读取的数据。

在本地建立根文件系统 myroot,然后再其目录下建立所需要的子目录，如 bin, dev,etc,lib,mnt,proc,sbin,usr 等。建立好了目录之后就要给各相应的目录复制相应的文件或库，通过 BusyBox 来实现。BusyBox 提供一个公平,完整的 POSIX 环境用于许多小系统,是一个可配置的工具。通过 make menuconfig 配置完成 BusyBox 后,改 BusyBox 的 Makefile 内容,使用交叉编译环境。然后通过 make 命令进行编译,make install 进行安装，再将生成的目录下的相应文件复制到我们所构建的文件系统的相应目录下。

以上这些步骤和前面的几乎都一样，在制作时可以参考上面的制作步骤。完成这些步骤后，利用 MKCRAMFS 工具来制作我们所需要的文件系统，通过命令 mkcramfs myroot myroot.fs 就可以把 myroot 制作成只读的压缩的 cramfs 文件系统。

这里主要介绍一下 mkcarmfs 工具的安装和使用。carmfs 工具包的下载地址是：http://sourceforge.net/projects/cramfs/

解压之后直接在解压后的文件夹中用 make 命令进行编译即可，如下图

```
helight@helight-desktop:~/linux/arm-2008/cramfs-1.1$ ls -l
total 156
-rw-r--r-- 1 helight ftp 18009 2002-02-16 08:11 COPYING
-rwxr-xr-x 1 helight ftp 31628 2008-05-24 21:10 cramfsck
-rw-r--r-- 1 helight ftp 18135 2002-02-23 08:00 cramfsck.c
-rw-r--r-- 1 helight ftp   156 2002-02-23 08:52 GNUmakefile
drwxr-xr-x 2 helight ftp  4096 2002-02-23 08:52 linux
-rwxr-xr-x 1 helight ftp 33274 2008-05-24 21:10 mkcramfs
-rw-r--r-- 1 helight ftp 25418 2002-02-20 16:03 mkcramfs.c
-rw-r--r-- 1 helight ftp  6322 2002-02-23 08:00 NOTES
-rw-r--r-- 1 helight ftp  2599 2002-02-16 09:06 README
helight@helight-desktop:~/linux/arm-2008/cramfs-1.1$
```

切换到要制作文件系统的文件夹中执行”mkcramfs rootfs my.fs”，这里的 roofs 是我的要制作文件系统的文件夹，my.fs 是要制作成的文件系统。

```
helight@helight-desktop:~/linux/arm-2008$ cd rootfs/
helight@helight-desktop:~/linux/arm-2008/rootfs$ ls
bin  dev  etc  lib  linuxrc  proc  sbin  [redacted]  usr  var
helight@helight-desktop:~/linux/arm-2008/rootfs$ cd ../
helight@helight-desktop:~/linux/arm-2008$ cr
cramfsck    cron        crontab     cryptsetup
helight@helight-desktop:~/linux/arm-2008$ mkcramfs rootfs/ myrootfs.fs
Directory data: 6460 bytes
Everything: 1376 kilobytes
Super block: 76 bytes
CRC: 33653f8b
warning: gids truncated to 8 bits (this may be a security concern)
```

制作好之后还可以挂载到一个文件夹上去看文件系统中的内容。如下图：

```
helight@helight-desktop:~/linux/arm-2008$ ls initrd/
helight@helight-desktop:~/linux/arm-2008$ mount myrootfs.fs initrd/ -o loop
mount: only root can do that
helight@helight-desktop:~/linux/arm-2008$ sudo mount myrootfs.fs initrd/ -o loop
helight@helight-desktop:~/linux/arm-2008$ cd initrd/
helight@helight-desktop:~/linux/arm-2008/initrd$ ls
bin  dev  etc  lib  linuxrc  proc  sbin  [redacted]  usr  var
```

关于这一部分还在进行中...

## 2.6 系统烧写

开发板针对于 S3c2410。S3C2410 由 ARM920T 内核（16-/32-bit RISC CPU）、独立的 16KB 指令和 16KB 数据 cache、MMU 虚拟内存管理单元、LCD 控制器（支持 STN 和 TFT）、NAND Flash bootloader、系统管理单元（SDRAM 控制器等）、3 通道 UART、4 通道 DMA、4 通道具备 PWM 功能的定时器、IO 口、RTC（实时时钟）、8 通道 10bit 精度 ADC 和触摸屏控制器、IIC 总线接口、IIS 数字音频总线接口、USB 主机、USB 设备、SD/MMC 卡控制器、2 通道 SPI 和 PLL 数字锁相环组成。

具体的开发板参数如下：

1. CPU：采用 SAMSUNG S3C2410AL 的 ARM920t CPU,
2. 存储器：

1. 64M SDRAM
2. 64M Nand Flash 用于存放应用程序
3. SD 卡

3. 一个 TFT 输出接口（可以选购配套的 TFT 真彩色 LCD）。
4. CS8900 以太网控制器
5. 一个 USB 主机接口
6. 一个 USB 设备接口（可以切换成第二主机，需要更改驱动）
7. 一个总线扩展接口。
8. 2 个串口输出
9. 一个 MIC 输入接口
10. 一个 LINE 音频输入接口。
11. 一个耳机输出接口。
12. 2 个 CPU 可控 LED 等。
13. AD 转换输入接口
14. SPI 和 IIC 接口
15. JTAG 调试接口
16. 5 个按键

## 2.7 调试环境－NFS

建立的目的：可以直接在开发板上挂载开发主机上的文件系统，减少程序的烧写次数，提高程序开发速度。

NFS(Network File System, 网络文件系统)可以通过 NFS 把远程主机的目录挂载到本机, 使得访问远程主机的目录就像访问本地目录一样方便快捷。

NFS 一般是实现 linux 系统之间实现共享.当然和 unix 之间也应该可以使用它来实现共享。但如果需要在 linux 和 windows 系统之间共享, 就得使用 samba 了!

NFS 是一个 RPC 服务程序,所以在使用它之前, 先要映射好端口——通过 portmap 设定. 比如: 某个 NFS client 发起 NFS 服务请求时, 它需要先得到一个端口(port).所以它先通过 portmap 得到 port number.所以在启动 NFS 之前, 需要启动 portmap 服务!

Debian 上默认是没有安装 NFS 服务器的，首先要安装 NFS 服务程序：

sudo apt-get install nfs-kernel-server

(安装 nfs-kernel-server 时，apt 会自动安装 nfs-common 和 portmap）

这样，宿主机就相当于 NFS Server。

与 NFS 相关的几个文件, 命令

1、/etc/exports 对共享目录的管理都是在这个文件中实现的

2、/sbin/exportfs 维护 NFS 的资源共享.通过它可以使修改后的/etc/exports 中的的共享目录生效关于这个命令的使用方法如下：

exportfs [-aruv]

-a ： 全部 mount 或者 unmount /etc/exports 中的内容

-r ： 重新 mount /etc/exports 中分享出来的目录-u ： umount 目录

-v ：在 export 的?r 候，将详细的信息输出到屏幕上。

3、/usr/sbin/showmount 用在 NFS Server 端。主要用查看 RPC 共享的连接

4、/var/lib/nfs/xtab NFS 的记录文档:通过它可以查看有哪些Client 连接到NFS 主机的记录.

下面这几个文件并不直接负责 NFS, 实际上它们负责所有的 RPC

5、/etc/default/portmap 实际上, portmap 负责映射所有的 RPC 服务端口

6、/etc/hosts.deny 设定拒绝 portmap 服务的主机

7、/etc/hosts.allow 设定允许 portmap 服务的主机

1.修改/etc/exports。/etc/exports 是 nfs 服务器的核心配置文件。在/etc/exports 中添加一个共享目录。

/var/nfs/ *(rw,sync)

/var/nfs/是要共享的文件夹，*是表示所有用户都可以挂载这个共享文件夹。这里也可以替换成 ip 地址，网段（192.168.1.0 /24）或是主机名。(rw,sync)表示以读写方式挂载，并且远程主机同步，sync 是 NFS 的默认选项。关于括号内的参数还有以下几种：rw：可读写的权限；ro：只读的权限；no_root_squash：登入到 NFS 主机的用户如果是 ROOT 用户，他就拥有 ROOT 的权限，此参数很不安全，建议不要使用。root_squash：all_squash：不管登陆 NFS 主机的用户是什么都会被重新设定为 nobody。anonuid：将登入 NFS 主机的用户都设定成指定的 user id,此 ID 必须存在于/etc/passwd 中。 anongid：同 anonuid ，但是?成 group ID 就是了！ sync：资料同步写入存储器中。async：资料会先暂时存放在内存中，不会直接写入硬盘。insecure 允许从这台机器过来的非授权访问。

2 使用命令 sudo exportfs -r 更新

3.重新启动 portmap 服务和 nfs-kernel-server 服务

命令分别为：

/etc/init.d/portmap start

/etc/init.d/nfs-kernel-server restart

4.linux 开发板端的配置

在 linux 开发板端在还需作一些配置才可以使用 mount 来挂载远程主机的 NFS 共享目录.配置修改如下：

1)配置内核

File systems –> Network File Systems –> NFS file system support

Provide NFSv3 client support

5．测试 NFS 启动客户端 uclinux 输入命令：

mount -t nfs 192.168.1.242:/var/nfs /mnt -o nolock

可以使用 ls /mnt 查看挂载过来的文件。在 linux 下挂载远程主机的共享文件主要是为了实现远程调试。在远程主机上进行交叉编译之后，在 linux 下直接运行编译好的程序。

# 第3章 Linux 启动设置与驱动移植

## 3.1启动画面设置

主要修改文件在 drivers/video/logo/下，主要操作是替换这个文件夹下的 logo_linux_clut224.ppm，同时还要删除 logo_linux_clut224.c logo_linux_clut224.o 文件，这样才能在编译内核的时候使用 logo_linux_clut224.ppm 重新编译图片。

ppm 图片的生成：

这里要生成 ppm 的图片还需要一些工具要辅助生成，因为内核要使用的图片是 224 色的，而一般使用的图片都是 256 色以上的。所以这里就要安装这些辅助工具。

```
#apt-get install netpbm
helight@Zhwen:elinux$ pn
pngtopnm        pnmdepth        pnminterp-gen   pnmpsnr         pnmtile         pnmtorast
pnmalias        pnmenlarge      pnminvert       pnmquant        pnmtoddif       pnmtorle
pnmarith        pnmfile         pnmmargin       pnmremap        pnmtofiasco     pnmtosgi
pnmcat          pnmflip         pnmmontage      pnmrotate       pnmtofits       pnmtosir
pnmcolormap     pnmgamma        pnmnlfilt       pnmscale        pnmtojpeg       pnmtotiff
pnmcomp         pnmhisteq       pnmnoraw        pnmscalefixed   pnmtopalm       pnmtotiffcmyk
pnmconvol       pnmhistmap      pnmnorm         pnmshear        pnmtoplainpnm   pnmtoxwd
pnmcrop         pnmindex        pnmpad          pnmsmooth       pnmtopng
pnmcut          pnminterp       pnmpaste        pnmsplit        pnmtops
helight@Zhwen:elinux$ pn

# pngtopnm logo_linux_clut224.png > logo_linux_clut224.pnm
# pnmquant 224 logo_linux_clut224.pnm > logo_linux_clut224.pnm
# pnmtoplainpnm logo_linux_clut224.pnm > logo_linux_clut224.ppm
```

然后重新编译内核，烧写之后就可以看到你自己设置的图片了！

注意：设置土图片的大小不应该超过显示屏的像素的大小。

## 3.2网卡驱动移植

针对于 S3c2410＋2.6.14＋cs8900a 的网卡驱动移植：

下载 cs8900.c 和 cs8900.h 到 driver/net 下并且修改 net 目录下的 Makefile 文件，添加如下内容：

```
obj-$(CONFIG_ARM_CS8900)      += cs8900.o
```

再修改 deriver/net/kconfig 文件。在 config DM9000 后面增加以下内容:

```
config ARM_CS8900
    tristate "CS8900 support"
    depends on NET_ETHERNET && ARM && ARCH_SMDK2410
    help
      Support for CS8900A chipset based Ethernet cards. If you have a network (Ethernet)card of this type, say Y
and read the Ethernet-HOWTO, available from<http://www.tldp.org/docs.html#howto> as well as
<file:Documentation/networking/cs89x0.txt>.    To compile this driver as a module, choose M here and read
```

```
<file:Documentation/networking/net-modules.txt>. The module will be called cs8900.o.
```

在此增加 cs8900 的配置选项，使我们在配置 ARCH_SMDK2410 的时候，可以使用 CS89X0 的配置选项。这个配置选项位于：

```
Device Drivers --
    Network device support --
        Ethernet(10 or 100Mbit) --
        <*>CS8900 support
```

接下来需要增加网卡的初始化代码。

一．首先新建一个头文件/linux-2.6.14/include/asm/arch-s3c2410/smdk2410.h

```
#ifndef _INCLUDE_SMDK2410_H_
#define _INCLUDE_SMDK2410_H_
#include <linux/config.h>
#define pSMDK2410_ETH_IO            0x19000000
#define vSMDK2410_ETH_IO            0xE0000000
#define SMDK2410_ETH_IRQ            IRQ_EINT9
#endif // _INCLUDE_SMDK2410_H_
```

二．打开文件/linux-2.6.14/arch/arm/mach-s3c2410/mach-smdk2410.c 需要修改如下

```
#include <asm/arch-s3c2410/smdk2410.h>
Static struct map_desc smdk2410_iodesc[] __initdata = {
    {vSMDK2410_ETH_IO, pSMDK2410_ETH_IO, SZ_1M, MT_DEVICE},
}
```

## 3.3 添加 yaffs2 文件系统支持

YAFFS2 是 YAFFS 的升级版,能更好的支持 NAND FLASH,我们采用最新的 YAFFS2 文件系统。

1．我们首先下载 YAFFS 文件系统。

http://www.aleph1.co.uk/cgi-bin/viewcvs.cgi/yaffs2.tar.gz?view=tar

下载后，在主机上用命令解压。

```
tar xzvf yaffs2.tar.gz
```

2．运行 yaffs2 目录下的 patch-ker.sh 脚本即可完成安装。

```
./patch-ker.sh c /home/helight/arm/nms/linux-2.6.14
```

3．如果没有这个文件，则需要手动修改配置文件，并需要将 yaffs2 拷贝到 fs 文件夹下。一下是要修改的文件。

4．修改 fs/Kconfig,vi Kconfig ，添加

```
source "fs/yaffs2/Kconfig"
```

5．修改 Makefile 文件，在其中添加如下语句

```
obj-$(CONFIG_YAFFS_FS)+= yaffs2/
```

6．在”meke menuconfig”中按如下选择。但是手动添加的话这就和你添加的地方有关了。

在使用 yaffs2 文件夹中的脚本后添加 yaffs2 文件阿系统后是如下的选择。

```
Filesystems  ―›
      <*> YAFFS2 file system support
```

如果是命令参数问题，则作如下修改：注释掉 arch/arm/kernel/setup.c 文件中的 parse_tag_cmdline()函数中的 strlcpy()函数，这样就可以使用默认的 CONFIG_CMDLINE 了，在.config 文件中它被定义为"root=/dev/mtdblock2 ro init=/linuxrc console=ttySAC0 devfs =mount"(视具体情况而定)，在内核配置文件的 Bootoptions 中填入也可。

7. 最后修改 yaffs 中的 Makefile 文件。内容如下：

## 3.4 添加 spi 驱动

由于 2.6.14 中没有 spi 的驱动，所以要移植高版本的 spi 驱动，目前拷贝的是 2.6.18 的。将 2.6.18 内核中 spi 文件夹直接拷贝到 2.6.14 的 driver 文件夹下。然后修改 arch/arm/Kconfig 和 driever/Makefile 文件。

修改 arch/arm/Kconfig 文件，添加如下代码：（红色部分为添加部分）

```
source "drivers/i2c/Kconfig"


source "drivers/spi/Kconfig"
```

修改 driver/spi/Makefile 文件，添加如下代码：（红色部分为添加部分）

```
obj-$(CONFIG_MTD)                 += mtd/
obj-$(CONFIG_SPI)                 += spi/
```

# 第4章 Linux 驱动开发

## 4.1 内核模块

Linux 就是通常所说的单内核（monolithic kernel），即操作系统的大部分功能都被称为内核，并在特权模式下运行。它与微型内核 不同，后者只把基本的功能（进程间通信[IPC]、调度、基本的输入/输出 [I/O] 和内存管理）当作内核运行，而把其他功能（驱动程序、网络堆栈和文件系统）排除在特权空间之外。因此，您可能认为 Linux 是一个完全静态的内核，但事实恰恰相反。通过 Linux 内核模块（LKM）可以在运行时动态地更改 Linux。

可动态更改 是指可以将新的功能加载到内核、从内核去除某个功能，甚至添加使用其他 LKM 的新 LKM。LKM 的优点是可以最小化内核的内存占用，只加载需要的元素（这是嵌入式系统的重要特性）。

简单 LKM 的源代码：

```
#include <linux/init.h>
#include <linux/module.h>

static __init int hello_init(void)
{
        printk("hello world\n");
        return 0;
}

static __exit void hello_exit(void)
{
        printk("bey world\n");
}

module_init(hello_init);
module_exit(hello_exit);

MODULE_LICENSE("Dual BSD/GPL");
MODULE_AUTHOR("http://zhwen.org");
```

编译使用的 Makefile 文件：

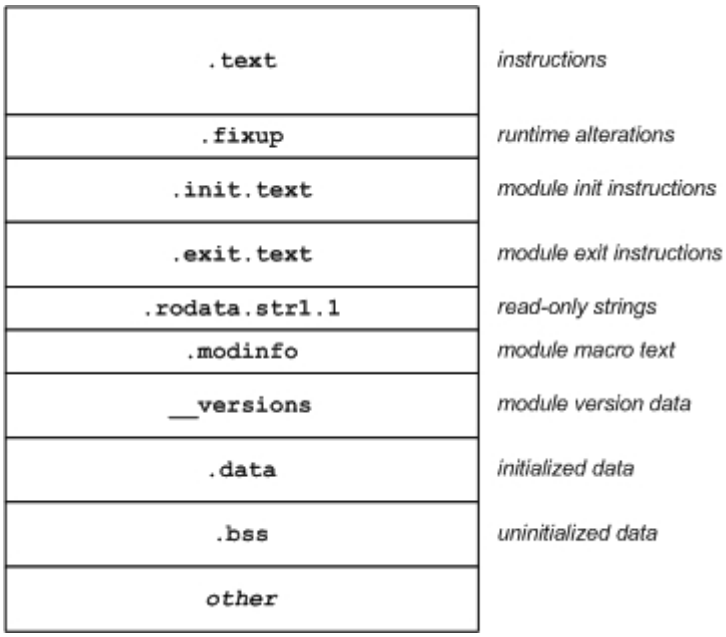
```
obj-m = hello.o

all:
        make -C /home/tao/arm/linux-2.7.8 M=$(shell pwd) modules
clean:
        make -C /home/tao/arm/linux-2.7.8 M=$(shell pwd) clean
        rm modules.order
```

LKM 只不过是一个特殊的可执行可链接格式（Executable and Linkable Format，ELF）对象文件。通常，必须链接对象文件才能在可执行文件中解析它们的符号和结果。

由于必须将 LKM 加载到内核后 LKM 才能解析符号，所以 LKM 仍然是一个 ELF 对象。您可以在 LKM 上使用标准对象工具（在 2.6 版本中，内核对象带有后缀 .ko,）。例如，如果在 LKM 上使用 objdump 实用工具，您将发现一些熟悉的区段（section），比如 .text（说明）、.data（已初始化数据）和 .bss（块开始符号或未初始化数据）。

您还可以在模块中找到其他支持动态特性的区段。.init.text 区段包含 module_init 代码，.exit.text 区段包含 module_exit 代码（参见下图）。.modinfo 区段包含各种表示模块许可证、作者和描述等的宏文本。

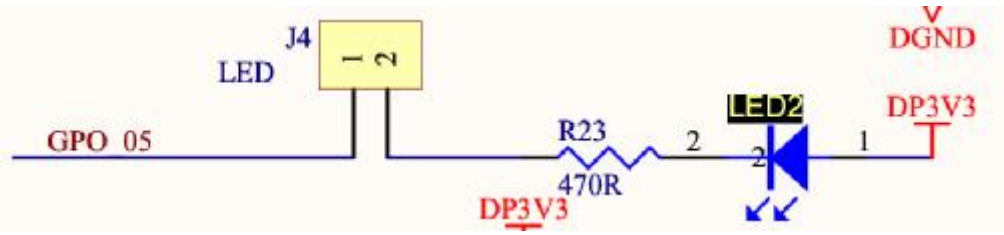
具有各种 ELF 区段的 LKM 的示例




## 4.2简单硬件驱动--LED 灯

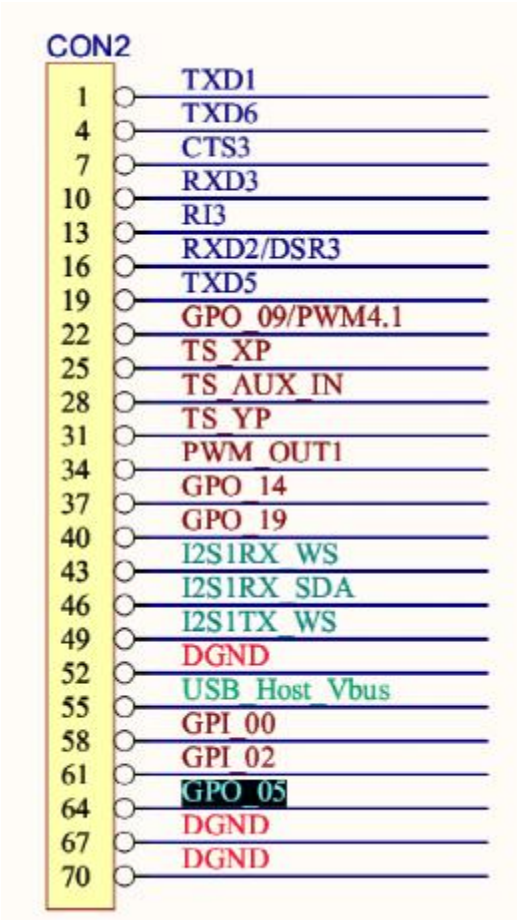
硬件的驱动可以使用简单的内核模块就可以完成。这里以 lpc3250 中的 led 灯为例来说明。开发板为 smartarm3250。

首先从硬件 PCB 原理图看起，从原理图上看的出如下的图：

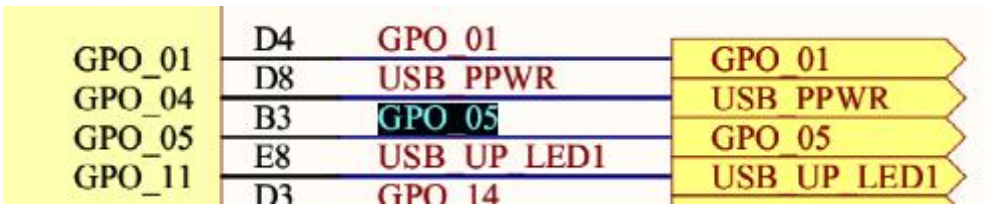



这里查到的是LED2灯的链接图，这里主要是查看到GPO_05。可以看出当GPO_05是地电位的时候 LED2 灯就会被点亮，而 GPO_05 为高电位的时候则 LED2 灯会熄灭。

再从原理图上继续查。从这个图上看到的是从底板到核心板的连接。是 CON2 中的第 64 脚。




接下来再看核心板的 PCB 原理图。




这里看到是从 MCU 连接出来的引脚，左边是引脚的名称。下一步就是在 MCU 芯片手册中查看这个引脚的控制寄存器了。

打开芯片的数据手册，以“GPO_05”这个关键字进行搜索。从这里可以看到这个引脚的寄存器和寄存器的物理地址，及其相应的控制位。如下图，设置寄存器（P3_OUTP_SET）是 0x40028004 ，GPO_05 的控制位是第五位。也就是在这个寄存器的第五位设置“1”的时候GPO_05输出为高电位。清除寄存器（P3_OUTP_CLR）的物理地址是0x40028008。也就是在这个寄存器的第五为设置”1”的时候 GPO_05 输出为地电位。

Table 636. P3 Output Pin Set Register (P3_OUTP_SET - 0x4002 8004)

| P3_OUTP_SET | Function | Description |
|---|---|---|
| 9 | GPO_09 / (LCDVD[9]) | Reflects the general purpose output pin GPO_9. |
| 8 | GPO_08 / (LCDVD[8]) | Reflects the general purpose output pin GPO_8. |
| 7 | GPO_07 / (LCDVD[2]) | Reflects the general purpose output pin GPO_7. |
| 6 | GPO_06 / (LCDVD[18]) | Reflects the general purpose output pin GPO_6. |
| 5 | GPO_05 | Reflects the general purpose output pin GPO_5. |
| 4 | GPO_04 | Reflects the general purpose output pin GPO_4. |
| 3 | GPO_03 / (LCDVD[1]) | Reflects the general purpose output pin GPO_3. |
| 2 | GPO_02 / T1_MAT.0 / (LCDVD[0]) | Reflects the general purpose output pin GPO_2. |
| 1 | GPO_01 | Reflects the general purpose output pin GPO_1. |
| 0 | GPO_00 / (TST_CLK1) | Reflects the general purpose output pin GPO_0. |

### 5.23 P3 Output Pin Clear Register (P3_OUTP_CLR - 0x4002 8008)

The P3_OUTP_CLR register is a write-only register that allows clearing one or more general purpose output GPO[32:0] and GPIO[5:0] configured as output pins.

Writing a one to a bit in P3_OUTP_CLR results in the corresponding output GPO[32:0] or GPIO[5:0] (if configured as an output) pin being driven low. Writing a zero to a bit in P3_OUTP_CLR has no effect.

Table 637. P3 Output Pin Clear Register (P3_OUTP_CLR - 0x4002 8008)

| P3_OUTP_CLR | Function | Description |
|---|---|---|
| 31:0 | | Same functions as P3_OUTP_SET. |

这些控制电路和控制寄存器都看明白之后再接下来的事情就是写程序去控制了。

这里很首先要知道控制 GPO_05 这是 MCU 的一种输出操作，所以要引用系统的 IO 库，在 Linux 内核中我们要使用的是针对于 arm 体系结构的 IO 库，这些库针对于不同的 MCU 或是 CPU 是不同的。在则会里引用<asm/io.h>这个头文件就可以了，这个头文件在内核源代码编译后会指向我们设定的 MCU 所使用的 IO 库。

接下来要只知道的就是，Linux 内核是不直接处理物理地址的，而是使用线性地址。所以在使用的时候还需要实现物理地址到线性地址的转换。这个转换针对于不同的体系结构是不一样的，在 x86 体系结构下的内核物理地址到线性地址的转换是差 0xc0000000,而在这里的转换就不是 0xc0000000 了。当然也是有相应的函数来进行地址转换的。这里是使用的是<mach/hardware.h>中的 io_p2v(x)函数进行转换的。它实现的就是物理地址到线性地址的转换。

最后一布就是开始写程序了，具体程序如下：

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/delay.h>
#include <asm/io.h>
#include <mach/hardware.h>

#define p3_out1         io_p2v(0x40028004)
#define p3_out0         io_p2v(0x40028008)

static __init int hello_init(void)
{
        int i = 0, tmp = 20;

        while (tmp) {
```

```
            __raw_writel((1 << 5), p3_outl);
            for (i = 100; i > 0; i--)
                udelay(1000);

            __raw_writel((1 << 5), p3_out0);
            for (i = 100; i > 0; i--)
                udelay(1000);
            tmp--;
        };

        printk("hello world\n");
        return 0;
}

static __exit void hello_exit(void)
{
        printk("bey world\n");
}

module_init(hello_init);
module_exit(hello_exit);

MODULE_LICENSE("Dual BSD/GPL");
MODULE_AUTHOR("http://zhwen.org");
```

这里只是实现了 LED2 灯的明灭控制。

总结：

这里所有的 PGO 端口都可以进行这样的操作，但是如果要对 GPIO 端口进行操作就会比较麻烦一点，还要设置其功能寄存器、掩码寄存器和方向寄存器等。

## 4.3简单硬件驱动—PWM

脉宽调制(PWM:(Pulse Width Modulation)是利用微处理器的数字输出来对模拟电路进行控制的一种非常有效的技术，广泛应用在从测量、通信到功率控制与变换的许多领域中。

简而言之，PWM 是一种对模拟信号电平进行数字编码的方法。通过高分辨率计数器的使用，方波的占空比被调制用来对一个具体模拟信号的电平进行编码。PWM 信号仍然是数字的，因为在给定的任何时刻，满幅值的直流供电要么完全有(ON)，要么完全无 (OFF)。电压或电流源是以一种通(ON)或断(OFF)的重复脉冲序列被加到模拟负载上去的。通的时候即是直流供电被加到负载上的时候，断的时候即是供电被断开的时候。只要带宽足够，任何模拟值都可以使用 PWM 进行编码。

PWM 的一个优点是从处理器到被控系统信号都是数字形式的，无需进行数模转换。让信号保持为数字形式可将噪声影响降到最小。噪声只有在强到足以将逻辑 1 改变为逻辑 0 或将逻辑 0 改变为逻辑 1 时，也才能对数字信号产生影响。

下面是针对于 lpc3250 的程序：

```
/*
```

```
 * Copyright (c) 2009-~ Helight.Xu
 *
 * This source code is released for free distribution under the terms of the
 * GNU General Public License
 *
 * Author:          Helight.Xu<Helight.Xu@gmail.com>
 * Created Time: Sun 06 Sep 2009 11:18:13 AM CST
 * File Name:       pwmdev.c
 *
 * Description:
 */

#include <linux/init.h>
#include <linux/module.h>
#include <linux/delay.h>
#include <asm/io.h>
#include <mach/hardware.h>
#include <mach/platform.h>
#include <mach/lpc32xx_gpio.h>

#define GPIO_IOBASE io_p2v(GPIO_BASE)

unsigned int *pwm_clk = NULL;
unsigned int *pwm_ctrl = NULL;

static __init int xpwm_init(void)
{
        unsigned long tmp = 6;

        pwm_clk = (unsigned int *)io_p2v(0x400040B8);
        pwm_ctrl = (unsigned int *)io_p2v(0x4005C000);

        tmp = (0x3 << 4) | 0x01;
        __raw_writel(tmp, pwm_clk);
        tmp = (0x01 << 31) | (0x01 << 30) | (0x0F << 8) | (0x01 << 7);
        __raw_writel(tmp, pwm_ctrl);

        printk("xpwm_init!!clk:%p ctrl:%p\n", (void *)pwm_clk, (void *)pwm_ctrl);
        return 0;
}
static __exit void xpwm_exit(void)
{
        printk("xpwm_exit!!\n");
        return;
```

```
}

module_init(xpwm_init);
module_exit(xpwm_exit);

MODULE_LICENSE("Dual BSD/GPL");
MODULE_AUTHOR("http://zhwen.org");
```

## 4.4内核驱动模型

## 4.5简单的字符驱动

# 第5章 Linux 应用编程

## 5.1 linux 串口编程

Linux 操作系统从一开始就对串行口提供了很好的支持，本文就 Linux 下的串行口通讯编程进行简单的介绍。

串行口是计算机一种常用的接口，具有连接线少，通讯简单，得到广泛的使用。常用的串口是 RS-232-C 接口（又称 EIARS-232-C）它是在 1970 年由美国电子工业协会（EIA）联合贝尔系统、调制解调器厂家及计算机终端生产厂家共同制定的用于串行通讯的标准。它的全名是"数据终端设备（DTE）和数据通讯设备（DCE）之间串行二进制数据交换接口技术标准"该标准规定采用一个 25 个脚的 DB25 连接器，对连接器的每个引脚的信号内容加以规定，还对各种信号的电平加以规定。传输距离在码元畸变小于 4% 的情况下，传输电缆长度应为 50 英尺。

Linux 操作系统从一开始就对串行口提供了很好的支持，这里就 Linux 下的串行口通讯编程进行简单的介绍，如果要非常深入了解，建议看看本文所参考的 《Serial Programming Guide for POSIX Operating Systems》

计算机串口的引脚说明（RS-232）：

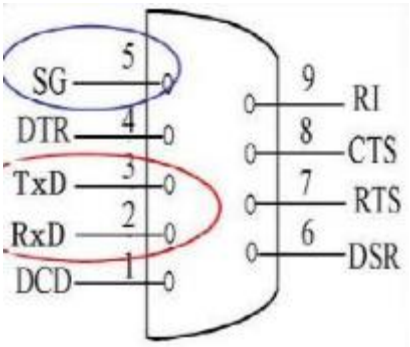



图 3.1 串口引脚序列

表 3.2 串口引脚功能

| Table 14 - RS-574 (IBM PC/AT) Signals | | | |
|---|---|---|---|
| Pin | Description | Pin | Description |
| 1 | Earth Ground | 6 | DSR - Data Set Ready |
| 2 | TXD - Transmitted Data | 7 | GND - Logic Ground |
| 3 | RXD - Received Data | 8 | DCD - Data Carrier Detect |
| 4 | RTS - Request To Send | 9 | DTR - Data Terminal Ready |
| 5 | CTS - Clear To Send | | |

最简单的双机互联图:

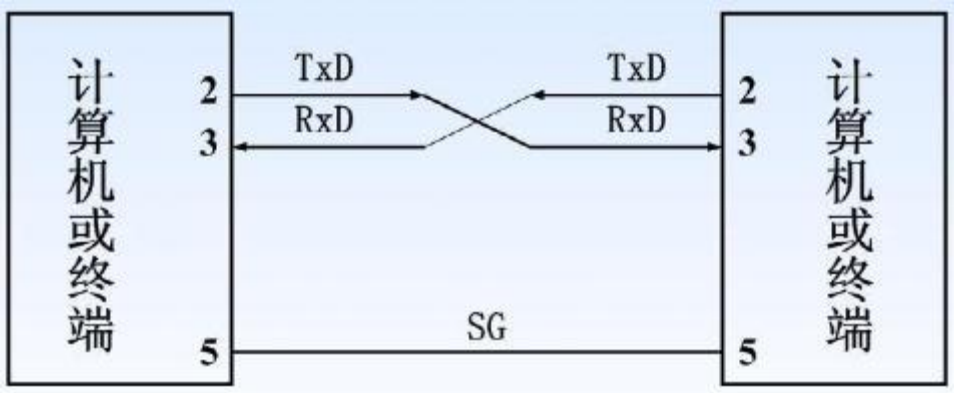



图 3.2 最简单的双机互联图

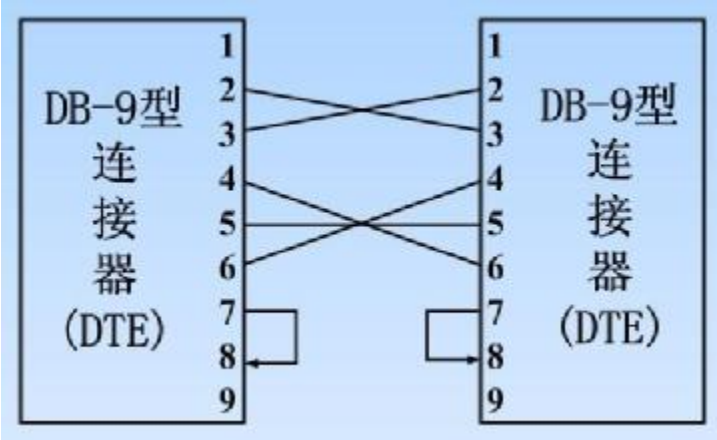



图 3.2 9 芯 DTE 与 9 芯 DTE 连接

## 5.2 linux 下串口的使用

在 Linux 下串口文件是位于 /dev 下的

串口 0 为 /dev/ttyS0

串口 1 为 /dev/ttyS1

打开串口是通过使用标准的文件打开函数操作：

```
#include <termios.h>      /*PPSIX 终端控制定义，串口操作需要的头文件*/
... ...
int main(){
    int fd;
    fd = open( "/dev/ttyS0", O_RDWR);   /*以读写方式打开串口*/
    if (fd<0){
        perror(" open");
    }
return 0;
}
```

串口设置最基本的设置串口包括波特率设置，效验位和停止位设置。

用命令查看串口的当前设置：stty -aF /dev/ttyS1

串口的设置主要是设置 struct termios 结构体的各成员值。

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

设置这个结构体很复杂，我这里就只说说常见的一些设置：

下面是修改波特率的代码：

```
struct   termios Opt;
tcgetattr(fd, &Opt);
cfsetispeed(&Opt,B19200);        /*设置为 19200Bps*/
cfsetospeed(&Opt,B19200);
tcsetattr(fd,TCANOW,&Opt);
```

设置波特率的例子函数：

```
/**
*      设置串口通信速率
*      fd        类型  int   打开串口的文件句柄
*      speed     类型  int   串口速度
*      返回值    void
*/
int speed_arr[] = { B38400, B19200, B9600, B4800, B2400, B1200, B300,
                                B38400, B19200, B9600, B4800, B2400, B1200, B300, };
int name_arr[] = {38400, 19200, 9600, 4800, 2400, 1200, 300, 38400, 19200, 9600, 4800, 2400, 1200, 300, };
void set_speed(int fd, int speed){
        int     i;
        int     status;
        struct termios     Opt;
        tcgetattr(fd, &Opt);
        for ( i= 0;   i < sizeof(speed_arr) / sizeof(int);   i++) {
                if   (speed == name_arr[i]) {
                        tcflush(fd, TCIOFLUSH);
                        cfsetispeed(&Opt, speed_arr[i]);
                        cfsetospeed(&Opt, speed_arr[i]);
                        status = tcsetattr(fd1, TCSANOW, &Opt);
                        if   (status != 0) {
                                perror("tcsetattr fd1");
                                return;
                        }
                        tcflush(fd,TCIOFLUSH);
```

```
            }
        }
}
```

效验位和停止位的设置：无效验　　8 位

```
Option.c_cflag &= ~PARENB;
Option.c_cflag &= ~CSTOPB;
Option.c_cflag &= ~CSIZE;
Option.c_cflag |= ~CS8;
奇效验(Odd)　7 位　Option.c_cflag |= ~PARENB;
Option.c_cflag &= ~PARODD;
Option.c_cflag &= ~CSTOPB;
Option.c_cflag &= ~CSIZE;
Option.c_cflag |= ~CS7;
偶效验(Even)　7 位　Option.c_cflag &= ~PARENB;
Option.c_cflag |= ~PARODD;
Option.c_cflag &= ~CSTOPB;
Option.c_cflag &= ~CSIZE;
Option.c_cflag |= ~CS7;
Space 效验 7 位　Option.c_cflag &= ~PARENB;
Option.c_cflag &= ~CSTOPB;
Option.c_cflag &= &~CSIZE;
Option.c_cflag |= CS8;
```

设置效验的函数：

```
/**
*     设置串口数据位，停止位和效验位
*     fd        类型    int   打开的串口文件句柄
*     databits 类型    int 数据位     取值 为 7 或者 8
*     stopbits 类型    int 停止位     取值为 1 或者 2
*     parity    类型    int   效验类型 取值为 N,E,O,,S
*/
int set_Parity(int fd,int databits,int stopbits,int parity)
{
        struct termios options;
        if   ( tcgetattr( fd,&options)   !=   0) {
                perror("SetupSerial 1");
                return(FALSE);
        }
        options.c_cflag &= ~CSIZE;
        switch (databits) /*设置数据位数*/
        {
        case 7:
                options.c_cflag |= CS7;
                break;
        case 8:
```

```
            options.c_cflag |= CS8;
            break;
        default:
            fprintf(stderr,"Unsupported data size\n"); return (FALSE);
        }
switch (parity)
{
        case 'n':
        case 'N':
            options.c_cflag &= ~PARENB;     /* Clear parity enable */
            options.c_iflag &= ~INPCK;      /* Enable parity checking */
            break;
        case 'o':
        case 'O':
            options.c_cflag |= (PARODD | PARENB); /* 设置为奇效验*/
            options.c_iflag |= INPCK;               /* Disnable parity checking */
            break;
        case 'e':
        case 'E':
            options.c_cflag |= PARENB;      /* Enable parity */
            options.c_cflag &= ~PARODD;     /* 转换为偶效验*/
            options.c_iflag |= INPCK;       /* Disnable parity checking */
            break;
        case 'S':
        case 's':  /*as no parity*/
            options.c_cflag &= ~PARENB;
            options.c_cflag &= ~CSTOPB;break;
        default:
            fprintf(stderr,"Unsupported parity\n");
            return (FALSE);
        }
/* 设置停止位*/
switch (stopbits)
{
        case 1:
            options.c_cflag &= ~CSTOPB;
            break;
        case 2:
            options.c_cflag |= CSTOPB;
           break;
        default:
             fprintf(stderr,"Unsupported stop bits\n");
             return (FALSE);
}
```

```
/* Set input parity option */
if (parity != 'n')
        options.c_iflag |= INPCK;
tcflush(fd,TCIFLUSH);
options.c_cc[VTIME] = 150; /*  设置超时 15 seconds*/
options.c_cc[VMIN] = 0; /* Update the options and do it NOW */
if (tcsetattr(fd,TCSANOW,&options) != 0)
{
        perror("SetupSerial 3");
        return (FALSE);
}
return (TRUE);
}
```

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

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

设置好串口之后，读写串口就很容易了，把串口当作文件读写就是。

发送数据

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

读取串口数据

使用文件操作 read 函数读取，如果设置为原始模式(Raw Mode)传输数据，那么 read 函数返回的字符数是实际串口收到的字符数。

可以使用操作文件的函数来实现异步读取，如 fcntl，或者 select 等来操作。

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

关闭串口就是关闭文件。

```
close(fd);
```

下面是一个简单的读取串口数据的例子，使用了上面定义的一些函数和头文件

```
/**********************************************************************
代码说明：使用串口二测试的，发送的数据是字符，
但是没有发送字符串结束符号，所以接收到后，后面加上了结束符号。
**********************************************************************/
#define FALSE  -1
#define TRUE   0
/**********************************************************************/
int OpenDev(char *Dev)
{
        int     fd= open( Dev, O_RDWR );
        if (-1 == fd)
        {
```

```
            perror("Can't Open Serial Port");
            return -1;
        }
        else
            return fd;
}
int main(int argc, char **argv){
        int fd;
        int nread;
        char buff[512];
        char *dev  = "/dev/ttyS1";                  //串口二
        fd = OpenDev(dev);
        set_speed(fd,19200);
        if (set_Parity(fd,8,1,'N') == FALSE)  {
            printf("Set Parity Error\n");
            exit (0);
        }
while (1)                                           //循环读取数据
{
        while((nread = read(fd, buff, 512))>0)
        {
            printf("\nLen %d\n",nread);
            buff[nread+1] = '\0';
            printf( "\n%s", buff);
        }
}
        close(fd);
        exit (0);
}
```

部分资料来自网络。我只做了整理、调试和加工。

# 第6章图形编程

## 6.1Framebuffer 图形编程

### 6.1.1Framebuffer 介绍

Framebuffer 在 Linux 中是作为设备来实现的，它是对图形硬件的一种抽象[1]，代表着显卡中的帧缓冲区（Framebuffer）。通过 Framebuffer 设备，上层软件可以通过一个良好定义的软件接口访问图形硬件，而不需要关心底层图形硬件是如何工作的，比如，上层软件不用关心应该如何读写显卡寄存器，也不需要知道显卡中的帧缓冲区从什么地址开始，所有这些工作都由 Framebuffer 去处理，上层软件只需要集中精力在自己要做的事情上就是了。

Framebuffer 的优点在于它是一种低级的通用设备，而且能够跨平台工作，比如 Framebuffer 既可以工作在 x86 平台上，也能工作在 PPC 平台上，甚至也能工作在 m68k 和 SPARC 等平台上，在很多嵌入式设备上 Framebuffer 也能正常工作。诸如 Minigui 之类的 GUI 软件包也倾向于采用 Framebuffer 作为硬件抽象层（HAL）。

从用户的角度来看，Framebuffer 设备与其它设备并没有什么不同。Framebuffer 设备位于/dev 下，通常设备名为 fb*，这里*的取值从 0 到 31。对于常见的计算机系统而言，32 个 Framebuffer 设备已经绰绰有余了（至少作者还没有看到过有 32 个监视器的计算机）。最常用到的 Framebuffer 设备是/dev/fb0。通常，使用 Framebuffer 的程序通过环境变量 FRAMEBUFFER 来取得要使用的 Framebuffer 设备，环境变量 FRAMEBUFFER 通常被设置为”/dev/fb0”。

从程序员的角度来看，Framebuffer 设备其实就是一个文件而已，可以像对待普通文件那样读写 Framebuffer 设备文件，可以通过 mmap() 将其映射到内存中，也可以通过 ioctl() 读取或者设置其参数，等等。最常见的用法是将 Framebuffer 设备通过 mmap()映射到内存中，这样可以大大提高 IO 效率。

要在 PC 平台上启用 Framebuffer，首先必须要内核支持，这通常需要重新编译内核。另外，还需要修改内核启动参数。在作者的系统上，为了启用 Framebuffer，需要将/boot/grub/menu.lst 中的下面这一行：

kernel /boot/vmlinuz-2.4.20-8 ro root=LABEL=/1

修改为

kernel /boot/vmlinuz-2.4.20-8 ro root=LABEL=/1 vga=0x0314

即增加了 vga=0x0314 这样一个内核启动参数。这个内核启动参数表示的意思是：Framebuffer 设备的大小是 800x600，颜色深度是 16bits/像素。

### 6.1.2让 pc 环境的 linux 进入 framebuffer 模式

在系统启动时向 kernel 传送 vga=mode-number 的参数

| 色彩 | 640x400 | 640x480 | 800x600 | 1024x768 | 1280x1024 | 1600x1200 |
|---|---|---|---|---|---|---|
| 4bits | ? | ? | 0x302 | ? | ? | ? |
| 8bits | 0x300 | 0x301 | 0x303 | 0x305 | 0x307 | 0x31C |

| | | | | | | |
|---|---|---|---|---|---|---|
| 15bits | ? | 0x310 | 0x313 | 0x316 | 0x319 | 0x31D |
| 16bits | ? | 0x311 | 0x314 | 0x317 | 0x31A | 0x31E |
| 24bits | ? | 0x312 | 0x315 | 0x318 | 0x31B | 0x31F |
| 32bits | ? | ? | ? | ? | ? | ? |

如在启动选项中添加 vga=0x318 进入 1024X768,24bits 的 framebuffer 模式，这样就可以直接在 pc 上调试我们板子上的 GUI 和应用程序了！

### 6.1.3DEMO：

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <linux/fb.h>
#include <sys/mman.h>

int main()
{
        int fbfd = 0;
        struct fb_var_screeninfo vinfo;
        struct fb_fix_screeninfo finfo;
        long int screensize = 0;
        char *fbp = 0;
        int i = 0;
        int x = 0, y = 0;
        long int location = 0;

        /* Open the file for reading and writing */
        fbfd = open("/dev/fb0", O_RDWR);
        if (!fbfd) {
                printf("Error: cannot open framebuffer device.\n");
                exit(1);
        }
        printf("The framebuffer device was opened successfully.\n");

        /* Get fixed screen information */
        if (ioctl(fbfd, FBIOGET_FSCREENINFO, &finfo)) {
                printf("Error reading fixed information.\n");
                exit(2);
        }

        /* Get variable screen information */
```

```
        if (ioctl(fbfd, FBIOGET_VSCREENINFO, &vinfo)) {
                printf("Error reading variable information.\n");
                exit(3);
        }

        /* Figure out the size of the screen in bytes */
        screensize = vinfo.xres * vinfo.yres * vinfo.bits_per_pixel / 8;
          printf("screensize: %ld \n", screensize);
        /* Map the device to memory */
        fbp = (char *)mmap(0, screensize, PROT_READ | PROT_WRITE, MAP_SHARED, fbfd, 0);
        if ((int)fbp == -1) {
          printf("Error: failed to map framebuffer device to memory.\n");
          exit(4);
        }
        printf("The framebuffer device was mapped to memory successfully.\n");

    for (i = 0; i < 100; i++) {
          x = i; y = i;          /* Where we are going to put the pixel */

          /* Figure out where in memory to put the pixel */
          location = (x+vinfo.xoffset) * (vinfo.bits_per_pixel/8) +
          (y+vinfo.yoffset) * finfo.line_length;

          *(fbp + location) = 100;       /* Some blue */
          *(fbp + location + 1) = 15;      /* A little green */
          *(fbp + location + 2) = 200;      /* A lot of red */
          *(fbp + location + 3) = 0;          /* No transparency */
    }

    munmap(fbp, screensize);

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

# 第7章 Net-snmp 移植编译

## 7.1 Net-snmp 介绍

Simple Network Management Protocol (SNMP) 是一个被广泛使用的协议，可以监控网络设备（比如路由器）、计算机设备甚至是 UPS。NET-SNMP 是一种开放源代码的简单网络管理协议（Simple Network Management Protocol）软件。Net-SNMP 是用于实施 SNMP v1，SNMP v2，SNMPv3 的应用程序套件，可以使用在 IPv4、IPv6 的环境中。这个套件包括：

命令行程序包括：

从支持 SNMP 的设备中检索信息的命令。用于执行单个请求（snmpget,snmpgetnext），或者执行多个请求（snmpwalk,snmptable,snmpdelta）。

可以用于手动设置信息的命令（snmpset）。

检索一套固定信息的命令（snmpdf，snmpnetstat，snmpstatus）。

可以把MIB oid 的信息在“数字”形式和“字符”形式之间进行转换的命令（snmptranslate），它还能显示 MIB 的内容和结构。

使用 Tk/perl 来提供一个图形化的 MIB 浏览器（tkmib）。

一个接收 SNMPtrap 信息的 daemon。经过选择的 snmp 通知信息可以被日志记录（记录在 syslog，或者 NT 的日志，或者文本文件），转发到另一个 SNMP 管理系统，或者传递到其它的程序。

一个可扩展的代理程序（snmpd），用于对管理系统提出的 SNMP 请求做出响应。这包括了内建的多种支持性：

支持广泛的 MIB 信息模块,可以使用动态加载的模块进行扩展,可以使用外部的脚本和命令进行扩展,对多路复用 SNMP（SMUX）和代理可扩展性协议（AgentX）的支持。

*包括一个库，用于支持对新的 SNMP 开发，支持 C 和 Perl API。

Net-SNMP 对于许多的 UNIX 和类 UNIX 操作系统都是支持的，也支持 windows。注意：对于不同的系统功能会有所变化。请阅读你所在平台的 README 文件。

Net-SNMP 是什么？

包含了以下多个支持 SNMP 协议的工具：

* 一个可扩展的代理
* 一个 SNMP 库
* 用于对 SNMP 代理进行查询操作和设置操作的工具
* 用于生成和处理 SNMP traps 的工具
* 一个支持 SNMP 的‘netstat'命令
* 一个基于 Perl/Tk/SNMP 的 MIB 信息浏览器

这个包最初基于卡耐基梅隆大学的 SNMP 开发工程（version 2.1.2.1）。

## 7.2 Net-snmp 移植编译

从 Net-snmp 官方网站（http://www.net-snmp.org/）下载 Net-snmp 的源代码包。本测试是下载的比较早期的 Net-snmp 版本（net-snmp-5.0.11.2）。

下载到本地后解压。

下载：

```
wget http://jaist.dl.sourceforge.net/sourceforge/net-snmp/net-snmp-5.0.11.2.tar.gz
```

解压：

helight@helight:nms$ tar zxf net-snmp-5.0.11.2.tar.gz

配置：

```
helight@helight:nms$ cd net-snmp-5.0.11.2/ helight@helight:net-snmp-5.0.11.2$ ./configure --host=arm-linux
target=arm    --with-cc=arm-linux-gcc --with-ar=arm-linux-ar --disable-shared    --with-endianness=little
```

编译：

```
helight@helight:net-snmp-5.0.11.2$ make
```

编译后：

```
helight@helight:net-snmp-5.0.11.2$ ls -X -n apps/
total 12948
-rwxr-xr-x 1 1000 1000   664381 2008-11-06 16:29 encode_keychange
-rw-r--r-- 1 1000 1000   111137 2008-11-06 16:28 Makefile
-rwxr-xr-x 1 1000 1000   671728 2008-11-06 16:29 snmpbulkget
-rwxr-xr-x 1 1000 1000   674729 2008-11-06 16:29 snmpbulkwalk
-rwxr-xr-x 1 1000 1000   675024 2008-11-06 16:29 snmpdf
-rwxr-xr-x 1 1000 1000   670961 2008-11-06 16:29 snmpget
-rwxr-xr-x 1 1000 1000   670848 2008-11-06 16:29 snmpgetnext
drwxr-xr-x 3 1000 1000     4096 2008-11-06 16:29 snmpnetstat
-rwxr-xr-x 1 1000 1000   674177 2008-11-06 16:29 snmpset
-rwxr-xr-x 1 1000 1000   676708 2008-11-06 16:29 snmpstatus
-rwxr-xr-x 1 1000 1000   682717 2008-11-06 16:29 snmptable
-rwxr-xr-x 1 1000 1000   683799 2008-11-06 16:29 snmptest
-rwxr-xr-x 1 1000 1000   666939 2008-11-06 16:29 snmptranslate
-rwxr-xr-x 1 1000 1000   675973 2008-11-06 16:29 snmptrap
-rwxr-xr-x 1 1000 1000 1181665 2008-11-06 16:29 snmptrapd
-rwxr-xr-x 1 1000 1000   679162 2008-11-06 16:29 snmpusm
-rwxr-xr-x 1 1000 1000   681936 2008-11-06 16:29 snmpvacm
-rwxr-xr-x 1 1000 1000   674319 2008-11-06 16:29 snmpwalk
。。。
helight@helight:net-snmp-5.0.11.2$ ls -X -n agent/
total 2760
drwxr-xr-x  2 1000 1000    4096 2008-06-19 10:23 dlmods
drwxr-xr-x  3 1000 1000    4096 2008-11-06 16:29 helpers
-rw-r--r--  1 1000 1000  214531 2008-11-06 16:28 Makefile
drwxr-xr-x 21 1000 1000    4096 2008-11-06 16:29 mibgroup
-rwxr-xr-x  1 1000 1000 1635427 2008-11-06 16:29 snmpd
```

。。。

使用 file 命令查看文件类型：

```
helight@helight:net-snmp-5.0.11.2$ file agent/snmpd
```

```
agent/snmpd: ELF 32-bit LSB executable, ARM, version 1 (ARM), for GNU/Linux 2.4.3, dynamically linked
(uses shared libs), for GNU/Linux 2.4.3, not stripped
helight@helight:net-snmp-5.0.11.2$ file apps/snmpwalk
apps/snmpwalk: ELF 32-bit LSB executable, ARM, version 1 (ARM), for GNU/Linux 2.4.3, dynamically linked
(uses shared libs), for GNU/Linux 2.4.3, not stripped
helight@helight:net-snmp-5.0.11.2$
```

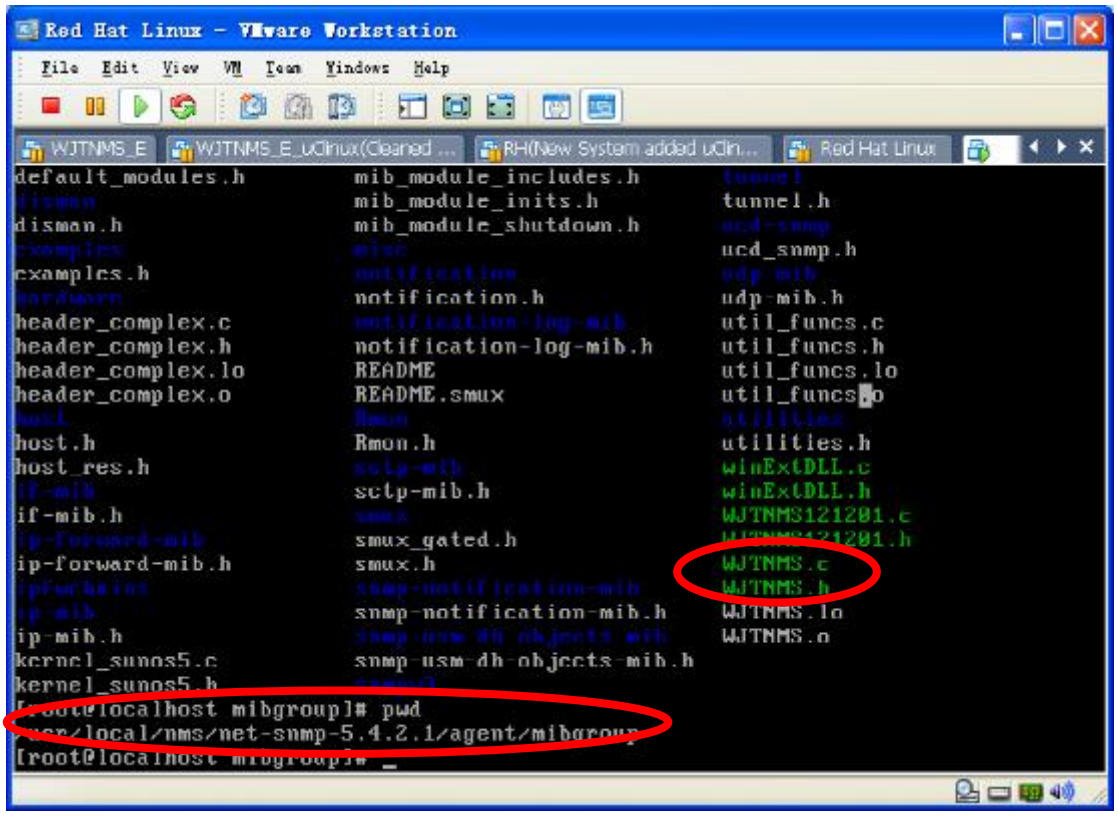
## 7.3 Net-snmp 扩展

这里所扩展是说添加自己定义的管理对象，在这里也就是写一些 C 文件，然后添加到 Net-snmp 中，再进行编译，使之具有自定义的管理对象库。具体扩展的 C 文件示例可以看 agent/mibgroup/examples 这个目录中的 example.c，example.h。下面是针对于一个项目中的扩展实例。

Net-snmp 扩展后，就可以把自定义的管理对象加入到原有的系统中，这里依据最新版本 net-snmp-5.4.2.1.tar.gz 说明，其移植编译过程与上一节介绍的 net-snmp-5.0.11.2 过程相同。

下面介绍扩展步骤：

1. 把编写好的代理程序（WJTNMS.c 和 WJTNMS.h）复制到目录/usr/local/nms/net-snmp-5.4.2.1/agent/mibgroup 中，如下图所示。




说明：WJTNMS.h 较之前版本没有改动，而 WJTNMS.c 较之前在 ucdsnmp 中编写的程序相比，需要改动：

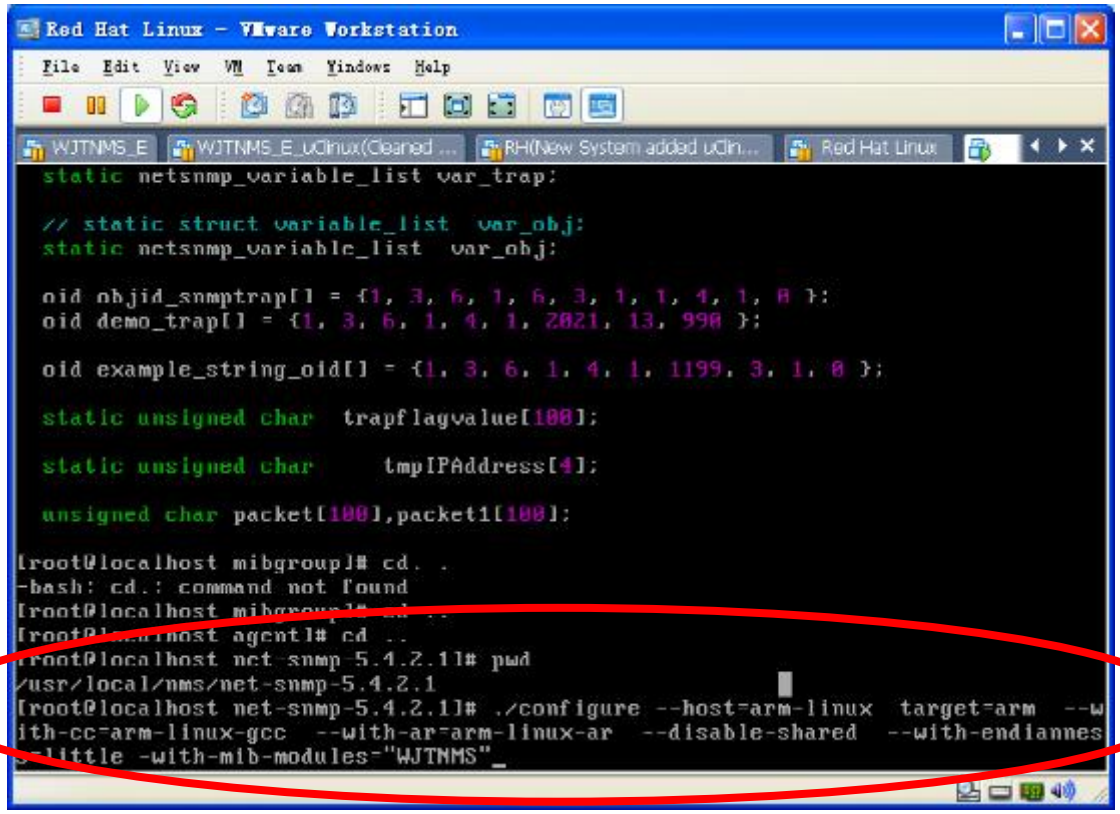
（1） 头文件结构；

（2） 如下图，注释部分为原有的代码，接着内容为 net-snmp 中的定义。

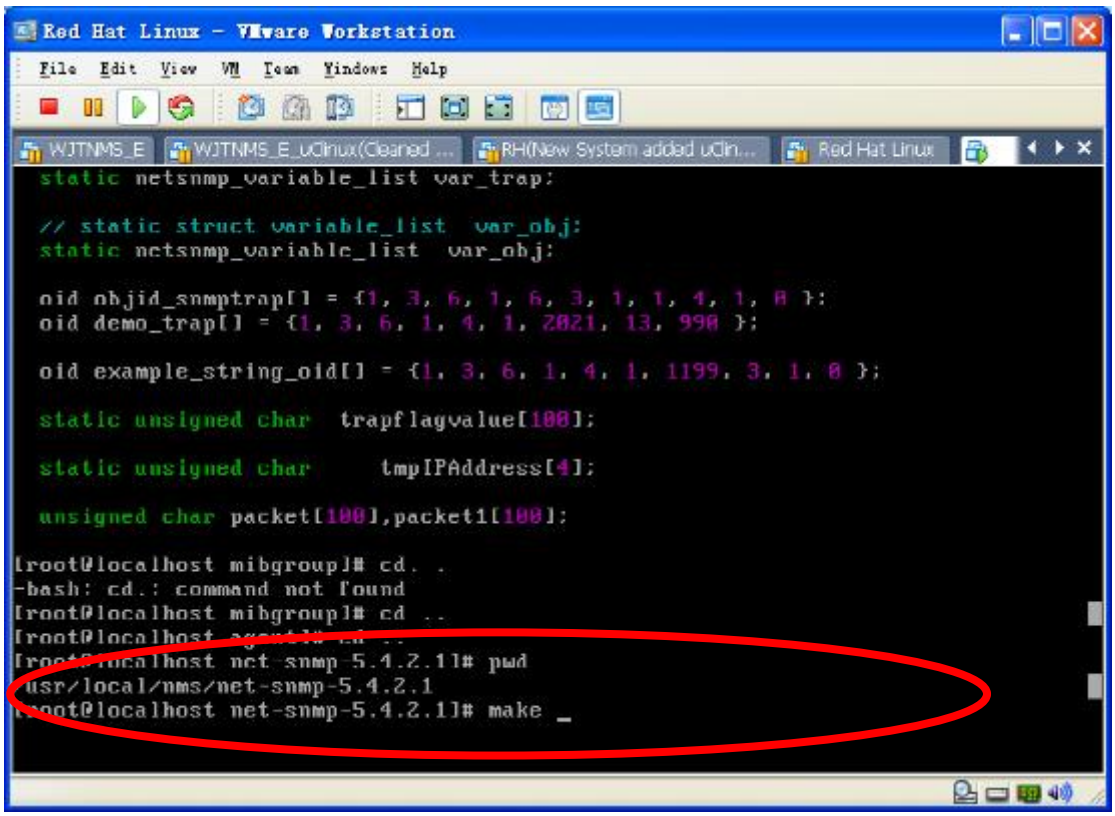
```
// static struct variable_list  var_trap;
static netsnmp_variable_list var_trap;

// static struct variable_list  var_obj;
static netsnmp_variable_list  var_obj;
```

详细请参看最新版本的 WJTNMS.c 代码。

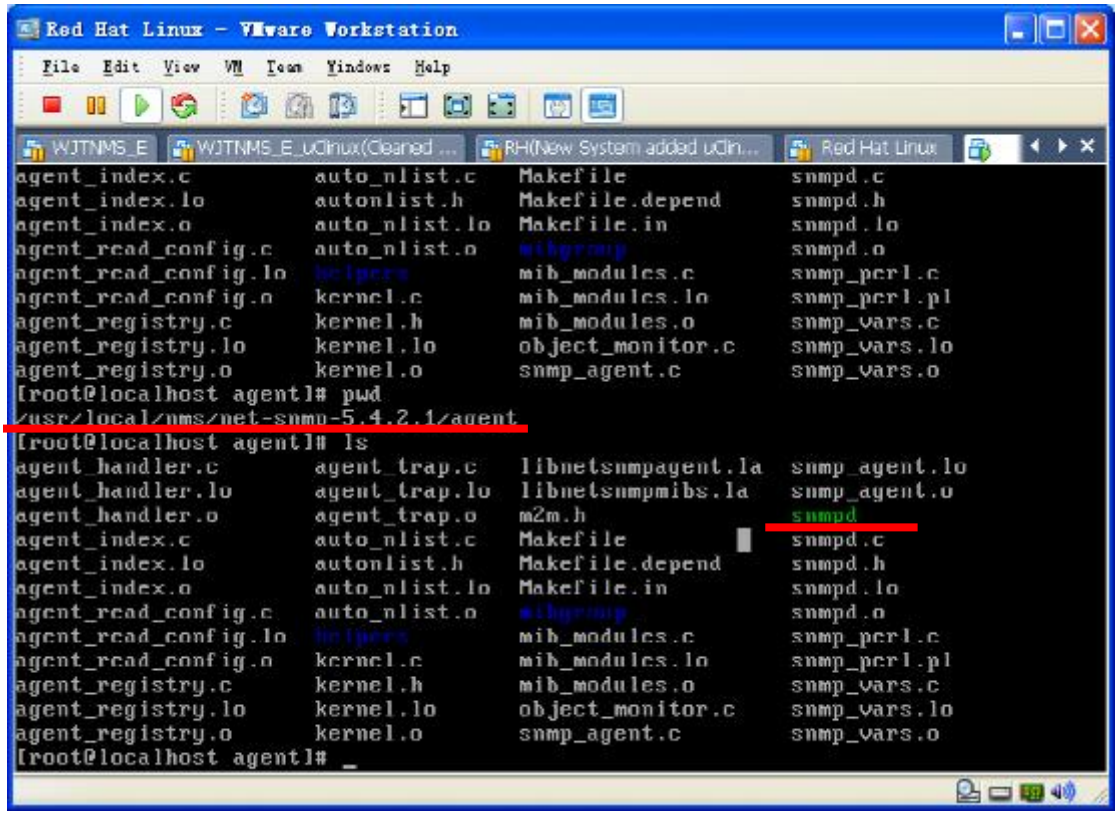
2．在 net-snmp-5.4.2.1 目录，进行配置：./configure --host=arm-linux target=arm --with-cc=arm-linux-gcc --with-ar=arm-linux-ar --disable-shared --with-endianness=little –with-mib-modules=”WJTNMS”(红色部分为扩展时必须的，这样才能加入自己的代理程序)




3．在 net-snmp-5.4.2.1 目录，进行编译：make




4．在 net-snmp-5.4.2.1/agent 目录中，即可看到生成的 snmpd 程序，如下图：




5．把 snmpd 程序通过 NFS 复制到开发板的“/usr/bin”目录中，并用如下命令启动：

/usr/bin/snmpd –V –c /usr/snmpd.conf

说明：（1）snmpd.conf 通过 NFS 复制到开发板的/usr 目录中，其内容与之前 ucdsnmp 内容相同。

（2）NFS 的安装使用方法请查看 2410 开发板随机资料的“用户手册”。

至此，net-snmp 扩展代理就实现了。

# 第8章 xHttp 的编写与移植

## 8.1xhttp 介绍

“基于 WEB 的嵌入式控制系统”软件是在传统意义下的远程监控的基础上，融合了 WEB 和嵌入式技术，从而提供比传统远程监控更为强大的功能。

该系统以嵌入式 Linux 为目标系统，以 B/S 软件体系结构架起客户端与目标系统之间的桥梁；以简单标准的 CGI 支持为基准，自行开发了轻量级的 Web 服务器。以开放式的开发框架为设计准则，为最终用户提供了扩充功能和二次开发的优势。

本系统的开发采用了业界流行的开源协作开发模式，使任何有兴趣的开发者都可以较容易地参与进来。本系统可以直接使用在一些产品中。比如一般的交换机，路由器，家电设备，甚至工业控制设备等的远程控制上。该系统具有实用性和可扩展性，值得推广。

### 8.1.1关键技术：

(1). Web 服务器开发（实现标准的 HTTP1.0 和 1.1 协议）

(2). 支持标准的 CGI 技术

(3). 与 SNMP 接口实现(在实现中。。。)

### 8.1.2特点：

(1). 实现轻量级 Web 服务器，更适合嵌入式环境

(2). Web 服务器支持更为适合嵌入式开发的简化 CGI，是应用开发更简易，快捷

(3). 采用开放框架设计，以支持企业，个人的二次开发

(4). 与 SNMP 相结合，以增强 Web 的管理能力和相对减少 Web 应用程序开发的工作量

(5). 本系统具有良好的可移植性，可以使用在交换机，路由器，家电设备及其工控设备上

(6). 采取先进的源码开放的开发模式

### 8.1.3系统整体结构

整个系统以一个 http 服务器为核心，在 http 服务器之下包含两个基本的系统：主控系统和系统监控子系统。

**http 服务器：**

即可以处理普通的静态页面，还可以处理处理普通的 CGI 程序，目前测试语言有 shell 语言和 C 语言写的 CGI 程序。

**系统主控系统：**

完成系统插件的查找和系统界面的生成。

**系统控制子系统：**

完成系统信息的收集，并且以比较美观的图形化方式显示到客户端浏览器。进行系统控制管理

**与 net-snmp 相结合：**

另外该系统还留有和 net-snmp 简单网络管理的接口。

## 8.2编译使用

项目目前情况：

已初步完成 web 服务器程序，但有待继续完善优化。

项目地址：

http://sites.google.com/site/xiyouhttpd/

目前还处于测试阶段，测试方法是，在程序目录中运行一下命令:

make

./xiyouhttpd

即可。

在浏览器中输入：http://ip:8080

**注意：在 PC 机或是在 ARM 上使用的时候在 make.conf 中做相应的编译器设置，**

目前端口可以由配置文件中配置。在系统控制的应用程序上目前比较缺少。只实现了文件系统浏览等少数操作。

另外：在该程序目录中有该项目的设计和实现文档。还有在 web 服务器下编写应用程序的相关规范。

http://sites.google.com/site/xiyouhttpd/ 这里有项目文档的在线部分。

# 第9章 SQlite 数据库移植与应用

嵌入式系统在基于网络控制系统的应用中，经常会遇到这样的要求，嵌入式系统负责采集和处理，并>把处理后的数据通过网络传输到远程计算机上并以 Web 页的方式显示，通常采集和处理的数据是大量随时间变化的动态数据，而对数据的存取通常可采取两种方式：一种是基于文件方式，另一种是基于数据库的方式[31]。对于文件方式，用户的应用程序以独占方式打开数据文件进行读写操作，I/O 开销较大，数据的共享性和应用程序的可重用性差，影响系统的整体性能；对于数据库方式，数据和应用程序相互独立，通过事务来进行调度和并发控制，有效地对数据进行了存取、查询等共享操作，确保系统具有较好的整体性能。因此，本系统中选用了嵌入式 SQLite 数据库存储从监控站点采集来的动态数据。

## 9.1 嵌入式数据库 SQLite 简介

SQLite 是 D·理查德·希普开发的一种的嵌入式关系数据库，具有体积小、数据容量大、处理速度快、占用内存少的特点。它包含在一个相对小的 C 库中，提供了对 ANSI SQL92 的大多数支持，实现了完备的、可嵌入的、零配置的 SQL 数据库引擎。SQLite 不是一个与程序通信的独立进程，而是连接到程序中成为它的一个主要部分，所以主要的通信协议是在编程语言内的直接 API 调用。SQLite 的特性有以下几点

(1)ACID 事务；

(2)零配置，无需安装和管理配置；

(3)储存在单一磁盘文件中的一个完整的数据库；

(4)数据库文件可以在不同字节顺序的机器间自由的共享；

(5)支持数据库大小至 2TB；

(6)足够小，大致 3 万行 C 代码，250K；

(7)比一些流行的数据库在大部分普通数据库操作要快；

(8)简单，轻松的 API；

(9)包含 TCL 绑定，同时通过 Wrapper 支持其他语言的绑定；

(10)良好注释的源代码，并且有着 90%以上的测试覆盖率；

(11)独立，没有额外依赖；

(12)Source 完全的 Open，你可以用于任何用途，包括出售它；

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

## 9.2 SQLite 在 ARM-Linux 下的移植

将 SQLite 应用到 S3C2410+Linux 环境中，就是将 SQLite 编译成 Linux 下的一个的普程序，然后通过它的 C 语言 API 接口函数在应用程序中操作它。

移植 SQLite 的主要步骤如下：

1. 在宿主机上建立基于 ARM9-Linux 的交叉编译环境，下载 sqlite-3.6.7.tar.gz。
2. 解压 sqlite-3.6.7.tar.gz。

```
helight@helight:elinux$tar-zxvf sqlite-3.6.7.tar.gz
helight@helight:elinux$ cd sqlite-3.6.7/
```

3. 在该目录下另外新建一个临时编译的目录，比如本实验在做时建立“xbuild”。

```
helight@helight:sqlite-3.6.7$ mkdir xbuild
helight@helight:sqlite-3.6.7$ cd xbuild/
```

4. 使用以下编译命令来编译：

```
helight@helight:xbuild$ ./../configure --host=arm-linux —prefix=/home/helight/elinux/sqlite-3.6.7/xbuild/
--disable-tcl
helight@helight:xbuild$ ls
config.h  config.log  config.status  libtool  Makefile  sqlite3.pc
helight@helight:xbuild$ make&&make install
```

在 configure 的参数中“—host=arm-linux”使用来表明是给 arm 芯片来编译的，“--prefix”是表示安装要安装到该参数所指定的文件夹下。”—disable-tcl”是不实用 tcl 脚本程序，嵌入式系统中一般不会支持太多的功能，所以 tcl 这种脚本程序一般不会被支持。

5. 编译之后可以发现在当前目录下已经生成目标文件，相关库文件和相关头文件。

```
helight@helight:xbuild$ ls
bin              lemon            mkkeywordhash  parse.out  sqlite3.o
config.h         lempar.c         opcodes.c      parse.y    sqlite3.pc
config.log       lib              opcodes.h      sqlite3    tsrc
config.status    libsqlite3.la    parse.c        sqlite3.c
include          libtool          parse.h        sqlite3.h
keywordhash.h    Makefile         parse.h.temp   sqlite3.lo
helight@helight:xbuild$ ls bin/
sqlite3
helight@helight:xbuild$ ls lib
libsqlite3.a   libsqlite3.so     libsqlite3.so.0.8.6
libsqlite3.la  libsqlite3.so.0   pkgconfig
helight@helight:xbuild$ ls include/
sqlite3ext.h  sqlite3.h
helight@helight:xbuild$
```

编译之后会在安装目录下生成下面三个文件夹：

bin，lib，include，在看一下这三个文件夹中的内容：

```
helight@helight:xbuild$ ls bin/
sqlite3
helight@helight:xbuild$ ls lib
libsqlite3.a   libsqlite3.so     libsqlite3.so.0.8.6
libsqlite3.la  libsqlite3.so.0   pkgconfig
helight@helight:xbuild$ ls include/
sqlite3ext.h  sqlite3.h
helight@helight:xbuild$
```

bin 目录下放的就是二进制主文件 sqlite3，到时候直接拷贝到目标文件系统的相应目录下即可。

lib 目录下存放的是相应的动态链接库和静态链接库。

include 目录下存放的是相应的头文件，在我们写程序使用数据库的是有就得引用该文件夹中的头文件。

6. 可以使命令“arm-linux-strip sqlite3”去掉其中的调试信息，进一步缩小其体积。
7. 拷贝 sqlite3 到目标板根文件系统/bin 目录下，重新制作目标板的根文件系统，烧写到 FLASH。在目标板的启动 Linux，在可读写目录的命令提取示符下输入 sqlite3，就可以创建数据库了。

## 9.3 SQLite 常用的 C 语言 API 函数

sqlite3 与 C 接口的 API 函数主要有以下几个函数(头文件 sqlite3.h):

```
int sqlite3_open(const char *filenam, sqlite3 **ppDb); //打开一个数据库
int sqlite3_close(sqlite3 *ppDb); //关闭
int sqlite3_exec(sqlite3 *ppDb, const char *sql, sqlite_callback, void*, char
**errmsg);//执行
int sqlite3_get_table(sqlite3 *ppDb, const char *sql,char***result, int *nrow,int
 ncolumn ,char **errmsg );
//result 中是以数组的形式存放所查询的数据，首先是表名，再是数据;
//nrow/ncolumn 分别为查询语句返回的结果集的行数/列数，没有查到结果时返回 0
sqlite3_errcode() 通常用来获取最近调用的 API 接口返回的错误代码.
sqlite3_errmsg() 则用来得到这些错误代码所对应的文字说明.
```

(1)打开或创建一个数据库

```
int sqlite3_open(const char *filename,sqlite3 **ppDb);
```

*filename 是数据库文件名，**ppDb 是数据库句柄。

(2)运行函数及回调函数

```
int sqlite3_exec(sqlite3*ppDb, const char*sql, sqlite_callback, void*, char **errmsg);
typedef  int(*sqlite3_callback)(void*,int,char**,char**); //sql 是要执行的 SQL 语句，可以是一条
```

或多条，sqlite_callback 是回调函数，void*是

传递给回调函数的参数，运行中每得到一个数据，callback 函数将被调用,用户能够根据具体需要自己编写 callback 函数获得操作数据库而得到的结果。**errmsg 用来保存错误的信息。

(3)查询数据

```
int sqlite3_get_table(sqlite3 *, const char*sql, char***resultp, int*nrow, int*ncolumn, char**errmsg);
```

***resultp 中是以数组的形式存放查询的数据，首先是字段名，再是数据。*nrow，*ncolumn 分别为查询语句返回的结果集的行数，列数，没有查到结果时返回 0。

exec 错误码

```
#define SQLITE_OK           0   /* Successful result */
#define SQLITE_ERROR        1   /* SQL error or missing database */
#define SQLITE_INTERNAL     2   /* An internal logic error in SQLite */
#define SQLITE_PERM         3   /* Access permission denied */
#define SQLITE_ABORT        4   /* Callback routine requested an abort */
#define SQLITE_BUSY         5   /* The database file is locked */
#define SQLITE_LOCKED       6   /* A table in the database is locked */
#define SQLITE_NOMEM        7   /* A malloc() failed */
#define SQLITE_READONLY     8   /* Attempt to write a readonly database */
```

```
#define SQLITE_INTERRUPT  9   /* Operation terminated by sqlite_interrupt() */
#define SQLITE_IOERR      10  /* Some kind of disk I/O error occurred */
#define SQLITE_CORRUPT    11  /* The database disk image is malformed */
#define SQLITE_NOTFOUND   12  /* (Internal Only) Table or record not found */
#define SQLITE_FULL       13  /* Insertion failed because database is full */
#define SQLITE_CANTOPEN   14  /* Unable to open the database file */
#define SQLITE_PROTOCOL   15  /* Database lock protocol error */
#define SQLITE_EMPTY      16  /* (Internal Only) Database table is empty */
#define SQLITE_SCHEMA     17  /* The database schema changed */
#define SQLITE_TOOBIG     18  /* Too much data for one row of a table */
#define SQLITE_CONSTRAINT 19  /* Abort due to contraint violation */
#define SQLITE_MISMATCH   20  /* Data type mismatch */
#define SQLITE_MISUSE     21  /* Library used incorrectly */
#define SQLITE_NOLFS      22  /* Uses OS features not supported on host */
#define SQLITE_AUTH       23  /* Authorization denied */
#define SQLITE_ROW        100 /* sqlite_step() has another row ready */
#define SQLITE_DONE       101 /* sqlite_step() has finished executing */
```

## 9.4 程序实例

程序：test.c

```
/*test.c*/
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>
#include <stdlib.h>
#include <string.h>
#include <sched.h>
#include "sqlite3.h"

int main(int argc, char **argv){
        int i,j;
        sqlite3 *db;
        char *zErr;
        char *eMsg;
        int status;
        int nrow=0, ncolumn=0;
        char **azResult; //存放结果
        char sql[]="SELECT * FROM tt";
  status = sqlite3_open("test.db", &db);
  if( status ){
```

```
    printf("Cannot initialize: \n");
    return 1;
  }
    printf("open\n");
  sqlite3_exec(db, "CREATE TABLE tt(a,b)", 0, 0, 0);    //该句在创建好表 tt 后就可以注释调了。
  sqlite3_exec(db, "insert into tt values(4,7)", 0, 0, 0);
  sqlite3_exec(db, "insert into tt values(3,3)", 0, 0, 0);
  sqlite3_exec(db, "insert into tt values(6,9)", 0, 0, 0);
  sqlite3_exec(db, "insert into tt values(1,5)", 0, 0, 0);
    printf("create table tt\n");

 sqlite3_get_table(db,sql,&azResult,&nrow,&ncolumn,&eMsg);
//其中 nrow 为行数，ncolum 为列数
 printf("\nThe result of querying is : \n");
 for(i=1;i<nrow+1;i++)
 {
      for(j=0;j<ncolumn;j++)
           printf("%s      ",azResult[i*ncolumn+j]);
       printf("\n");
 }

  sqlite3_close(db);
    printf("close db\n");
  return 0;
}
```

Makefile 文件：

```
CC=arm-linux-gcc
LIB=-L/home/helight/elinux/sqlite-3.6.7/xbuild/lib
INCLUD=-I/home/helight/elinux/sqlite-3.6.7/xbuild/include/
all:
        $(CC) test.c -o test -lsqlite3 $(LIB) $(INCLUD)
```

编译：

```
helight@helight:include$ make
arm-linux-gcc test.c -o test -lsqlite3 -L/home/helight/elinux/sqlite-3.6.7/xbuild/lib
-I/home/helight/elinux/sqlite-3.6.7/xbuild/include/
helight@helight:include$ ls
Makefile  sqlite3ext.h  sqlite3.h  test  test.c
helight@helight:include$ file test
test: ELF 32-bit LSB executable, ARM, version 1, dynamically linked (uses shared libs), for GNU/Linux 2.4.3, not
stripped
helight@helight:include$
```

在开发板上运行：

```
# ./test
open
create table tt

The result of querying is :
4       7
3       3
6       9
1       5
close db
# ls -l
drwxr-xr-x       1 0             0                 512 Jan    1 00:09 bin
drwxr-xr-x       1 0             0                 512 Jan    1 00:11 include
drw-rw-rw-       1 0             0                 512 Jan    1 00:00 lost+found
-rwxr-xr-x       1 0             0                9145 Jan    1 00:34 test
-rw-r--r--       1 0             0                2048 Jan    1 00:41 test.db
#
```

## 9.5数据表的设计与用户信息的本地化

## 9.6CGI 与 SQLite 交互程序的实现