嵌入式与移动开发系列

NITE 国家信息技术紧缺人才培养工程
National Information Technology Education Project
国家信息技术紧缺人才培养工程系列丛书

众多专家、厂商联合推荐 • 业界权威培训机构的经验总结

嵌入式Linux应用程序开发标准教程（第2版）

华清远见嵌入式培训中心 编著

提供36小时嵌入式专家讲座视频和教学课件

Embeded Linux Application Development

人民邮电出版社
POSTS & TELECOM PRESS

第 5 章　嵌入式 Linux 开发环境的搭建

华清远见

本章目标

在了解了嵌入式开发的基本概念之后，本章主要学习如何搭建嵌入式 Linux 开发的环境，通过本章的学习，读者能够掌握以下内容。

- 掌握嵌入式交叉编译环境的搭建
- 掌握嵌入式主机通信环境的配置
- 学会使用交叉编译工具链
- 学会配置 Linux 下的 minicom 和 Windows 下的超级终端
- 学会在 Linux 下和 Windows 下配置 TFTP 服务
- 学会配置 NFS 服务
- 学会编译 Linux 内核
- 学会搭建 Linux 的根文件系统
- 熟悉嵌入式 Linux 的内核相关代码的分布情况
- 掌握 Bootloader 的原理
- 了解 U-Boot 的代码结构和移植

5.1 嵌入式开发环境的搭建

5.1.1 嵌入式交叉编译环境的搭建

交叉编译的概念在第 4 章中已经详细讲述过，搭建交叉编译环境是嵌入式开发的第一步，也是必备的一步。搭建交叉编译环境的方法很多，不同的体系结构、不同的操作内容甚至是不同版本的内核，都会用到不同的交叉编译器，而且，有些交叉编译器经常会有部分的 bug，这都会导致最后的代码无法正常地运行。因此，选择合适的交叉编译器对于嵌入式开发是非常重要的。

交叉编译器完整的安装一般涉及多个软件的安装（读者可以从 ftp://gcc.gnu.org/pub/下载），包括 binutils、gcc、glibc 等软件。其中，binutils 主要用于生成一些辅助工具，如 objdump、as、ld 等；gcc 是用来生成交叉编译器的，主要生成 arm-linux-gcc 交叉编译工具（应该说，生成此工具后已经搭建起了交叉编译环境，可以编译 Linux 内核了，但由于没有提供标准用户函数库，用户程序还无法编译）；glibc 主要是提供用户程序所使用的一些基本的函数库。这样，交叉编译环境就完全搭建起来了。

上面所述的搭建交叉编译环境比较复杂，很多步骤都涉及对硬件平台的选择。因此，现在嵌入式平台提供厂商一般会提供在该平台上测试通过的交叉编译器，而且很多公司把以上安装步骤全部写入脚本文件或者以发行包的形式提供，这样就大大方便了用户的使用。如优龙的 FS2410 开发光盘里就附带了 2.95.3 和 3.3.2 两个版本的交叉编译器，其中前一个版本是用于编译 Linux 2.4 内核的，而后一个版本是用于编译 Linux 2.6 版本内核的。由于这是厂商测试通过的编译器，因此可靠性会比较高，而且与开发板能够很好地吻合。所以推荐初学者直接使用厂商提供的编译器。当然，由于时间滞后的原因，这个编译器往往不是最新的版本，若需要更新时希望读者另外查找相关资料学习。本书就以优龙自带的 cross-3.3.2 为例进行讲解（具体的名称不同厂商可能会有区别）。

安装交叉编译器的具体步骤在第 2 章的实验二中已经进行了详细地讲解了，在此仅回忆关键步骤，对于细节请读者参见第 2 章的实验二。

在/usr/local/arm 下解压 cross-3.3.2.bar.bz2。

```
    [root@localhost arm]# tar –jxvf cross-3.3.2.bar.bz2
    [root@localhost arm]# ls
    3.3.2  cross-3.3.2.tar.bz2
    [root@localhost arm]# cd ./3.3.2
    [root@localhost arm]# ls
    arm-linux   bin   etc   include   info   lib   libexec   man   sbin   share
VERSIONS
    [root@localhost bin]# which arm-linux*
    /usr/local/arm/3.3.2/bin/arm-linux-addr2line
    /usr/local/arm/3.3.2/bin/arm-linux-ar
```

```
/usr/local/arm/3.3.2/bin/arm-linux-as
/usr/local/arm/3.3.2/bin/arm-linux-c++
/usr/local/arm/3.3.2/bin/arm-linux-c++filt
/usr/local/arm/3.3.2/bin/arm-linux-cpp
/usr/local/arm/3.3.2/bin/arm-linux-g++
/usr/local/arm/3.3.2/bin/arm-linux-gcc
/usr/local/arm/3.3.2/bin/arm-linux-gcc-3.3.2
/usr/local/arm/3.3.2/bin/arm-linux-gccbug
/usr/local/arm/3.3.2/bin/arm-linux-gcov
/usr/local/arm/3.3.2/bin/arm-linux-ld
/usr/local/arm/3.3.2/bin/arm-linux-nm
/usr/local/arm/3.3.2/bin/arm-linux-objcopy
/usr/local/arm/3.3.2/bin/arm-linux-objdump
/usr/local/arm/3.3.2/bin/arm-linux-ranlib
/usr/local/arm/3.3.2/bin/arm-linux-readelf
/usr/local/arm/3.3.2/bin/arm-linux-size
/usr/local/arm/3.3.2/bin/arm-linux-strings
/usr/local/arm/3.3.2/bin/arm-linux-strip
```

可以看到，在/usr/local/arm/3.3.2/bin/下已经安装了很多交叉编译工具。用户可以查看 arm 文件夹下的 VERSIONS 文件，显示如下：

```
Versions
  gcc-3.3.2
  glibc-2.3.2
  binutils-head
Tool chain binutils configuration:
../binutils-head/configure …
Tool chain glibc configuration:
../glibc-2.3.2/configure …
Tool chain gcc configuration
../gcc-3.3.2/configure …
```

可以看到，这个交叉编译工具确实集成了 binutils、gcc、glibc 这几个软件，而每个软件也都有比较复杂的配置信息，读者可以查看 VERSIONS 文件了解相关信息。

5.1.2 超级终端和 minicom 配置及使用

前文已知，嵌入式系统开发的程序只能在对应的嵌入式硬件平台上运行，那么如何把开发板上的信息显示给开发人员呢？最常用的就是通过串口线输出到宿主机的显示器上，这样，开发人员就可以看到系统的运行情况了。在 Windows 和 Linux 中都有不少串口通信软件，可以很方便地对串口进行配置，其中最主要的配置参数是波特率、数据位、停止位、奇偶校验位和数据流控制位等，但是它们一定要根据实际情况

进行相应配置。下面介绍 Windows 中典型的串口通信软件“超级终端”和在 Linux 下的“minicom”。

1．超级终端

首先，打开 Windows 下的“开始”→“附件”→“通讯”→“超级终端”，这时会出现如图 5.1 所示的新建超级终端界面，在“名称”处可随意输入该连接的名称。

图 5.1　新建超级终端界面

接下来，将“连接时使用”的方式改为“COM1”，即通过串口 1，如图 5.2 所示。

接下来就到了最关键的一步——设置串口连接参数。要注意，每块开发板的连接参数有可能会有差异，其中的具体数据在开发商提供的用户手册中会有说明。如优龙的这款 FS2410 采用的是波特率为 115200，数据位数为 8，无奇偶校验位，停止位数为 1，无硬件流控，其对应配置如图 5.3 所示。

图 5.2　选择连接时使用方式

图 5.3　配置串口相关参数

这样，就基本完成了配置，最后一步单击“确定”按钮就可以了。这时，读者可以把开发板的串口线和 PC 机相连，若配置正确，在开发板上电后，在超级终端的窗口里应能显示类似于图 5.4 的串口信息。

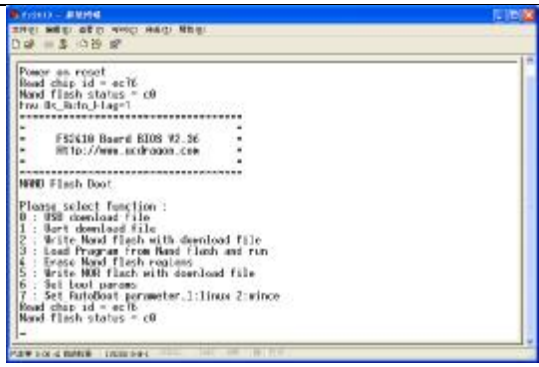

图 5.4　在超级终端上显示信息

注意　要分清开发板上的串口 1、串口 2，如在优龙的开发板上标有“UART1”、“UATR2”否则串口无法打印出信息。

2．minicom

minicom 是 Linux 下串口通信的软件，它的使用完全依靠键盘的操作，虽然没有“超级终端”那么易用，但是使用习惯之后读者将会体会到它的高效与便利。下面主要讲解如何对 minicom 进行串口参数的配置。

首先在命令行中键入“minicom”，这就启动了 minicom 软件。minicom 在启动时默认会进行初始化配置，如图 5.5 所示。可以通过“minicom -s”命令进行 minicom 的配置。

图 5.5　minicom 启动

注意　在 minicom 的使用中，经常会遇到 3 个键的操作，如“CTRL-A Z”，这表示先同时按下 CTRL 和“A”，然后松开这两个键再按下“Z”。

正如图 5.5 中的提示，接下来可键入 CTRL－A Z，来查看 minicom 的帮助，如图 5.6 所示。按照帮助所示，可键入“O”（代表 Configure minicom）来配置 minicom 的串口参数，当然也可以直接键入“CTRL-A O”来进行配置。如图 5.7 所示。

图 5.6　minicom 帮助

图 5.7　minicom 配置界面

在这个配置框中选择“Serial port setup”子项，进入如图 5.8 所示的配置界面。

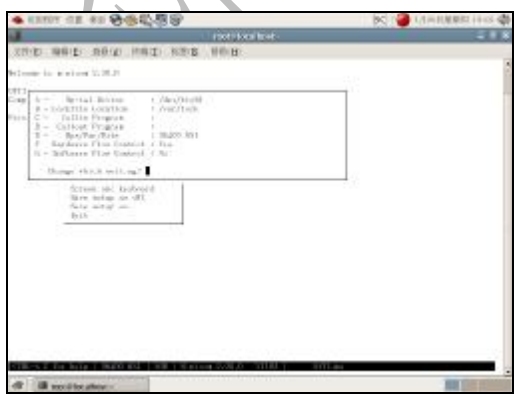

图 5.8　minicom 串口属性配置界面

上面列出的配置是 minicom 启动时的默认配置，用户可以通过键入每一项前的大写字母，分别对每一项进行更改。图 5.9 所示为在“Change which setting” 中键入了“A”，此时光标转移到第 A 项的对应处。

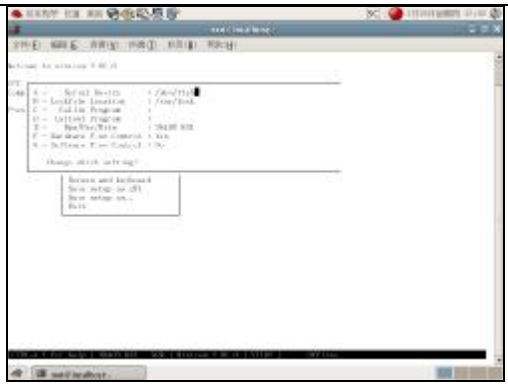

图 5.9　minicom 串口号配置

注意　在 minicom 中“ttyS0”对应“COM1”，“ttyS1”对应“COM2”。

接下来，要对波特率、数据位和停止位进行配置，键入“E”，进入如图 5.10 所示的配置界面。

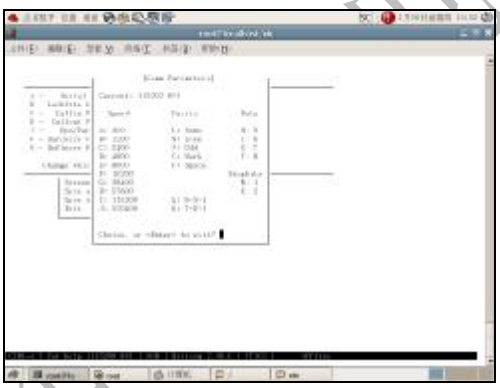

图 5.10　minicom 波特率等配置界面

在该配置界面中，可以键入相应波特率、停止位等对应的字母，即可实现配置，配置完成后按回车键就退出了该配置界面，在上层界面中显示如图 5.11 所示配置信息，要注意与图 5.8 进行对比，确定相应参数是否已被重新配置。

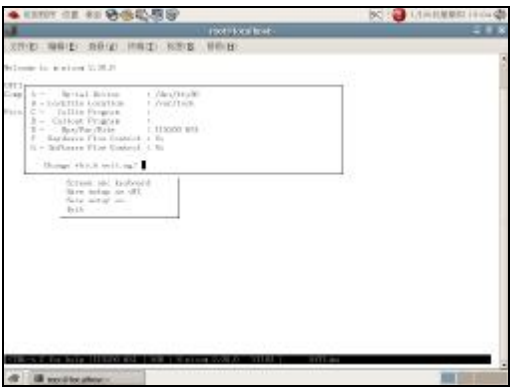

图 5.11　minicom 配置完成后界面

在确认配置正确后，可键入回车返回上级配置界面，并将其保存为默认配置，如图 5.12 所示。之后，可重新启动 minicom 使刚才配置生效，在用串口线将宿主机和开发板连接之后，就可在 minicom 中打印出正确的串口信息，如图 5.13 所示。

图 5.12　minicom 保存配置信息

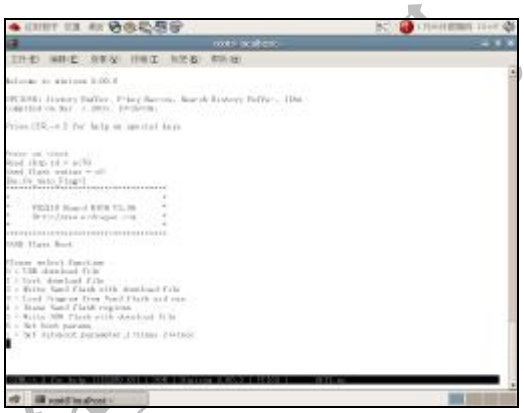

图 5.13　minicom 显示串口信息

到此为止，读者已经能将开发板的系统情况通过串口打印到宿主机上了，这样，就能很好地了解硬件的运行状况。

小知识　通过串口打印信息是一个很常见的手段，很多其他情况如路由器等也是通过配置串口的波特率这些参数来显示对应信息的。

5.1.3　下载映像到开发板

正如第 4 章中所述，嵌入式开发的运行环境是目标板，而开发环境是宿主机。因此，需要把宿主机中经过编译之后的可执行文件下载到目标板上。要注意的是，这里所说的下载是下载到目标机中的 SDRAM。然后，用户可以选择直接从 SDRAM 中运行或写入到 Flash 中再运行。运行常见的下载方式有网络下载（如 tftp、ftp 等方式）、串口下载、USB 下载等，本书主要讲解网络下载中的 tftp 方式和串口下载方式。

1．tftp

tftp 是简单文件传输协议，它可以看作是一个 FTP 协议的简化版本，与 FTP 协议相比，它的最大区别在于没有用户管理的功能。它的传输速度快，可以通过防火墙，

使用方便快捷，因此在嵌入式的文件传输中广泛使用。

同 FTP 一样，tftp 分为客户端和服务器端两种。通常，首先在宿主机上开启 tftp 服务器端服务，设置好 tftp 的根目录内容（也就是供客户端访问的根目录），接着，在目标板上开启 tftp 的客户端程序（现在很多 Bootloader 几乎都提供该服务）。这样，把目标板和宿主机用直连线相连之后，就可以通过 tftp 协议传输可执行文件了。

下面分别讲述在 Linux 下和 Windows 下的配置方法。

（1）Linux 下 tftp 服务配置。

Linux 下 tftp 的服务器服务是由 xinetd 所设定的，默认情况下是处于关闭状态。

首先，要修改 tftp 的配置文件，开启 tftp 服务，如下所示：

```
[root@localhost tftpboot]# vim /etc/xinetd.d/tftp
# default: off
# description: The tftp server serves files using the trivial file transfer\
#       protocol.  The tftp protocol is often used to boot diskless
\
#          workstations, download configuration files to network-aware
printers,\
#       and to start the installation process for some operating systems.
service tftp
{
        socket_type             = dgram  /* 使用数据报套接字*/
        protocol                = udp    /* 使用 UDP 协议 */
        wait                    = yes    /* 允许等待 */
        user                    = root   /* 用户 */
        server                  = /usr/sbin/in.tftpd /* 服务程序
*/
        server_args             = -s /tftpboot /* 服务器端的根目录
*/
        disable                 = no           /* 使能 */
        per_source              = 11
        cps                     = 100 2
        flags                   = IPv4
}
```

在这里，主要要将“disable=yes”改为“no”，另外，从“server_args”可以看出，tftp 服务器端的默认根目录为“/tftpboot”，用户如果需要则可以更改为其他目录。

接下来，重启 xinetd 服务，使刚才的更改生效，如下所示：

```
[root@localhost tftpboot]# service xinetd restart
（或者使用/etc/init.d/xinetd restart，而且因发行版的不同具体路径会有所不同）
关闭 xinetd:                                               [  确定  ]
启动 xinetd:                                               [  确定  ]
```

接着，使用命令“netstat -au”以确认 tftp 服务是否已经开启，如下所示：

```
[root@localhost tftpboot]# netstat –au | grep tftp
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address                Foreign Address
State
udp        0      0 *:tftp                      *:*
```

这时，用户就可以把所需要的传输文件放到“/tftpboot”目录下，这样，主机上的 tftp 服务就可以建立起来了（注意：需要在服务端关闭防火墙）。

接下来，用直连线把目标板和宿主机连起来，并且将其配置成一个网段的地址（例如两个 IP 都可以设置为 192.168.1.XXX 格式），再在目标板上启动 tftp 客户端程序（注意：不同的 Bootloader 所使用的命令可能会不同，例如：在 RedBoot 中使用 load 命令下载文件是基于 tftp 协议的。读者可以查看帮助来获得确切的命令名及格式），如下所示：

```
=>tftpboot 0x30200000 zImage
TFTP from server 192.168.1.1; our IP address is 192.168.1.100
Filename 'zImage'.
Load address: 0x30200000
Loading:
#################################################################
             #################################################################
             ##############################################
done
Bytes transferred = 881988 (d7544 hex)
```

可以看到，此处目标板使用的 IP 为“192.168.1.100”，宿主机使用的 IP 为“192.168.1.1”，下载到目标板的地址为 0x30200000，文件名为“zImage”。

（2）Windows 下 tftp 服务配置。

在 Windows 下配置 tftp 服务器端需要下载 tftp 服务器软件，常见的为 tftpd32。

首先，单击 tftpd32 下方的设置按钮，进入设置界面，如图 5.14 所示，在这里，主要配置 tftp 服务器端地址，也就是宿主机的地址。

接下来，重新启动 tftpd32 软件使刚才的配置生效，这样服务器端的配置就完成了，这时，就可以用直连线连接目标机和宿主机，且在目标机上开启 tftp 服务进行文件传输，这时，tftp 服务器端如图 5.15 和图 5.16 所示。

图 5.14　tftp 文件传输情况

图 5.15　tftpd32 配置界面

图 5.16　tftp 服务器端显示

小知识

tftp 是一个很好的文件传输协议，它的简单易用吸引了广大用户。但它同时也存在着较大的安全隐患。由于 tftp 不需要用户的身份认证，因此给了黑客的可乘之机。2003 年 8 月 12 日爆发的全球冲击波（Worm.Blaster）病毒就是模拟一个 tftp 服务器，并启动一个攻击传播线程，不断地随机生成攻击地址进行入侵。因此在使用 tftp 时一定要设置一个单独的目录作为 tftp 服务的根目录，如上文所述的“/tftpboot”等。

2．串口下载

使用串口下载需要配合特定的下载软件，如优龙公司提供的 DNW 软件等，一般在 Windows 下进行操作。虽然串口下载的速度没有网络下载快，但由于它很方便，不需要额外的连线和设置 IP 等操作，因此也广受用户的青睐。下面就以 DNW 软件为例，介绍串口下载的方式。

与其他串口通信的软件一样，在 DNW 中也要设置“波特率”、“端口号”等。打开“Configuration”下的“Options”界面，如图 5.17 所示。

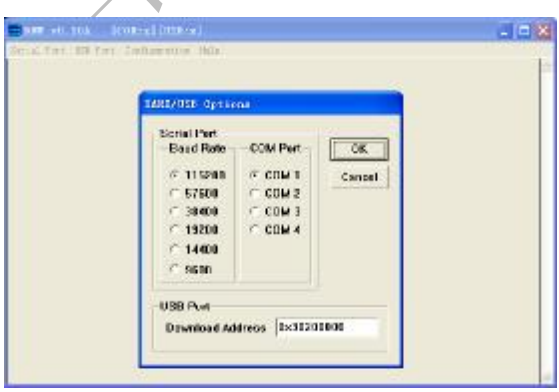

图 5.17　DNW 配置界面

在配置完之后，单击“Serial Port”下的“Connect”，再将开发板上电，选择“串口下载”，接着再在“Serial Port”下选择“Transmit”，这时，就可以进行文件传输了，如图 5.18 和图 5.19 所示。这里 DNW 默认串口下载的地址为 0x30200000。

图 5.18　DNW 串口下载图

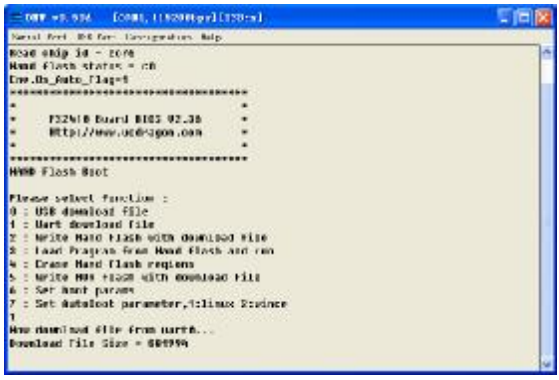

图 5.19　DNW 串口下载情形图

5.1.4　编译嵌入式 Linux 内核

在做完了前期的准备工作之后，在这一步，读者就可以编译嵌入式 Linux 的内核了。在这里，本书主要介绍嵌入式 Linux 内核的编译过程，在下一节会进一步介绍嵌入式 Linux 中体系结构相关的内核代码，读者在此之后就可以尝试嵌入式 Linux 操作系统的移植。

编译嵌入式 Linux 内核都是通过 make 的不同命令来实现的，它的执行配置文件就是在第 3 章中讲述的 makefile。Linux 内核中不同的目录结构里都有相应的 makefile，而不同的 makefile 又通过彼此之间的依赖关系构成统一的整体，共同完成建立依赖关系、建立内核等功能。

内核的编译根据不同的情况会有不同的步骤，但其中最主要分别为 3 个步骤：内核配置、建立依赖关系、创建内核映像，除此之外还有一些辅助功能，如清除文件和依赖关系等。读者在实际编译时若出现错误等情况，可以考虑采用其他辅助功能。下面分别讲述这 3 步主要的步骤。

（1）内核配置。

第一步内核配置中的选项主要是用户用来为目标板选择处理器架构的选项，不同的处理器架构会有不同的处理器选项，比如 ARM 就有其专用的选项如“Multimedia capabilities port drivers”等。因此，在此之前，必须确保在根目录中 makefile 里“ARCH”的值已设定了目标板的类型，如：

```
ARCH        := arm
```

接下来就可以进行内核配置了，内核支持 4 种不同的配置方法，这几种方法只是与用户交互的界面不同，其实现的功能是一样的。每种方法都会通过读入一个默认的配置文件——根目录下“.config”隐藏文件（用户也可以手动修改该文件，但不推荐使用）。当然，用户也可以自己加载其他配置文件，也可以将当前的配置保存为其他名字的配置文件。这 4 种方式如下。

- make config：基于文本的最为传统的配置界面，不推荐使用。

- make menuconfig：基于文本选单的配置界面，字符终端下推荐使用。
- make xconfig：基于图形窗口模式的配置界面，Xwindow 下推荐使用。
- make oldconfig：自动读入“.config”配置文件，并且只要求用户设定前次没有设定过的选项。

在这 4 种模式中，make menuconfig 使用最为广泛，下面就以 make menuconfig 为例进行讲解，如图 5.20 所示。

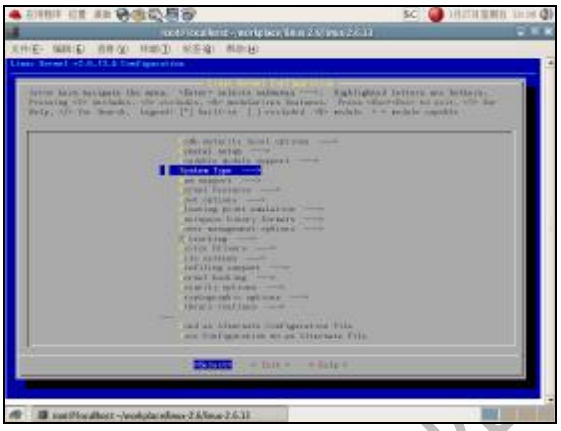

图 5.20　make menuconfig 配置界面

从该图中可以看出，Linux 内核允许用户对其各类功能逐项配置，一共有 18 类配置选项，这里就不对这 18 类配置选项进行一一讲解了，需要的时候读者可以参见相关选项的 help。在 menuconfig 的配置界面中是纯键盘的操作，用户可使用上下键和“Tab”键移动光标以进入相关子项，图 5.21 所示为进入了“System Type”子项的界面，该子项是一个重要的选项，主要用来选择处理器的类型。

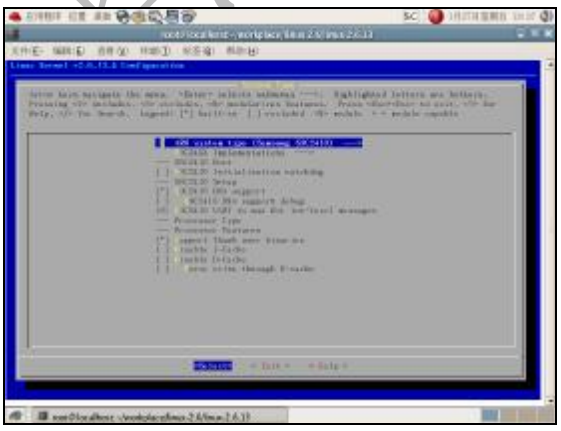

图 5.21　System Type 子项

可以看到，每个选项前都有个括号，可以通过按空格键或“Y”键表示包含该选项，按“N”表示不包含该选项。

另外，读者可以注意到，这里的括号有 3 种，即中括号、尖括号或圆括号。读者可以用空格键选择相应的选项时可以发现中括号里要么是空，要么是“*”；尖括号里可以是空，“*”和“M”，分别表示包含选项、不包含选项和编译成模块；圆括号的

内容是要求用户在所提供的几个选项中选择一项。

此外，要注意 2.4 和 2.6 内核在串口命名上的一个重要区别，在 2.4 内核中“COM1”对应的是“ttyS0”，而在 2.6 内核中“COM1”对应“ttySAC0”，因此在启动参数的子项要格外注意，如图 5.22 所示，否则串口打印不出信息。

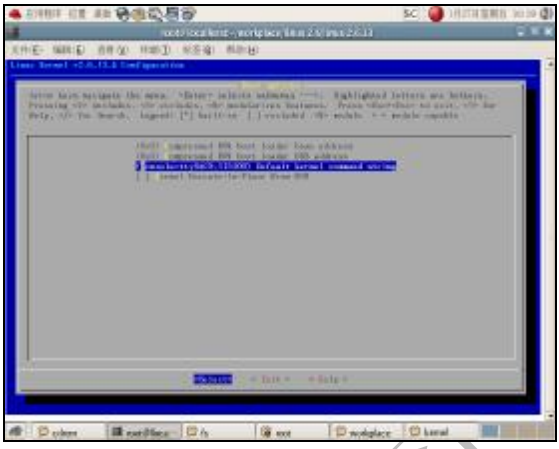

图 5.22　启动参数配置子项

一般情况下，使用厂商提供的默认配置文件都能正常运行，所以用户初次使用时可以不用对其进行额外的配置，在以后需要使用其他功能时再另行添加，这样可以大大减少出错的几率，有利于错误定位。在完成配置之后，就可以保存退出，如图 5.23 所示。

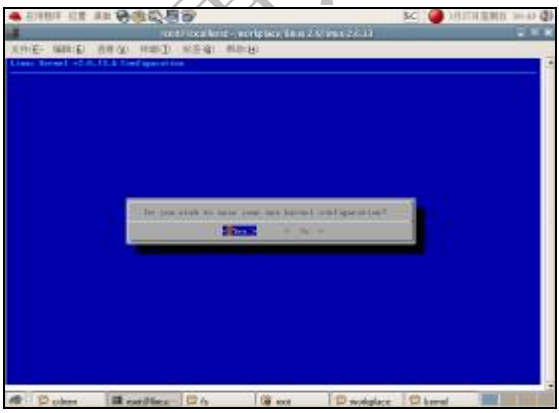

图 5.23　保存退出

（2）建立依赖关系。

由于内核源码树中的大多数文件都与一些头文件有依赖关系，因此要顺利建立内核，内核源码树中的每个 Makefile 都必须知道这些依赖关系。建立依赖关系通常在第一次编译内核的时候（或者源码目录树的结构发生变化的时候）进行，它会在内核源码树中每个子目录产生一个“.depend”文件。运行“make dep”即可。在编译 2.6 版本的内核通常不需要这个过程，直接输入“make”即可。

（3）建立内核

建立内核可以使用“make”、“make zImage”或“make bzImage”，这里建立的为

压缩的内核映像。通常在 Linux 中，内核映像分为压缩的内核映像和未压缩的内核映像。其中，压缩的内核映像通常名为 zImage，位于“arch/$（ARCH）/boot”目录中。而未压缩的内核映像通常名为 vmlinux，位于源码树的根目录中。

到这一步就完成了内核源代码的编译，之后，读者可以使用上一小节所讲述的方法把内核压缩文件下载到开发板上运行。

小知识	在嵌入式 Linux 的源码树中通常有以下几个配置文件，“.config”、“autoconf.h”、“config.h”，其中“.config”文件是 make menuconfig 默认的配置文件，位于源码树的根目录中。“autoconf.h”和“config.h”是以宏的形式表示了内核的配置，当用户使用 make menuconfig 做了一定的更改之后，系统自动会在“autoconf.h”和“config.h”中做出相应的更改。它们位于源码树的“/include/linux/”下。

5.1.5 Linux 内核源码目录结构

Linux 内核源码的目录结构如图 5.24 所示。

图 5.24 Linux 内核目录结

- /include 子目录包含了建立内核代码时所需的大部分包含文件，这个模块利用其他模块重建内核。
- /init 子目录包含了内核的初始化代码，这里的代码是内核工作的起始入口。
- /arch 子目录包含了所有处理器体系结构特定的内核代码。如：arm、i386、alpha。
- /drivers 子目录包含了内核中所有的设备驱动程序，如块设备和 SCSI 设备。
- /fs 子目录包含了所有的文件系统的代码，如：ext2、vfat 等。
- /net 子目录包含了内核的网络相关代码。
- /mm 子目录包含了所有内存管理代码。
- /ipc 子目录包含了进程间通信代码。
- /kernel 子目录包含了内核核心代码。

5.1.6 制作文件系统

读者把上一节中所编译的内核压缩映像下载到开发板后会发现，系统在进行了一些初始化的工作之后，并不能正常启动，如图 5.25 所示。

可以看到，系统启动时发生了加载文件系统的错误。要记住，上一节所编译的仅仅是内核，文件系统和内核是完全独立的两个部分。读者可以回忆一下第 2 章讲解的 Linux 启动过程的分析（嵌入式 Linux 是 Linux 裁减后的版本，其精髓部分是一样的），其中在 head.S 中就加载了根文件系统。因此，加载根文件系统是 Linux 启动中不可缺少的一部分。本节将讲解嵌入式 Linux 中文件系统的制作方法。

华清远见

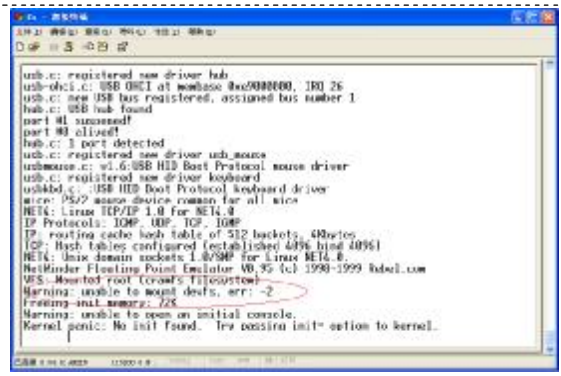

图 5.25　系统启动错误

制作文件系统的方法有很多，可以从零开始手工制作，也可以在现有的基础上添加部分内容并加载到目标板上去。由于完全手工制作工作量比较大，而且也很容易出错，因此，本节将主要介绍把现有的文件系统加载到目标板上的方法，主要包括制作文件系统映像和用 NFS 加载文件系统的方法。

1．制作文件系统映像

读者已经知道，Linux 支持多种文件系统，同样，嵌入式 Linux 也支持多种文件系统。虽然在嵌入式系统中，由于资源受限的原因，它的文件系统和 PC 机 Linux 的文件系统有较大的区别，但是，它们的总体架构是一样的，都是采用目录树的结构。在嵌入式系统中常见的文件系统有 cramfs、romfs、jffs、yaffs 等，这里就以制作 cramfs 文件系统为例进行讲解。cramfs 文件系统是一种经过压缩的、极为简单的只读文件系统，因此非常适合嵌入式系统。要注意的是，不同的文件系统都有相应的制作工具，但是其主要的原理和制作方法是类似的。

在嵌入式 Linux 中，busybox 是构造文件系统最常用的软件工具包，它被非常形象地称为嵌入式 Linux 系统中的“瑞士军刀”，因为它将许多常用的 Linux 命令和工具结合到了一个单独的可执行程序（busybox）中。虽然与相应的 GNU 工具比较起来，busybox 所提供的功能和参数略少，但在比较小的系统（例如启动盘）或者嵌入式系统中已经足够了。busybox 在设计上就充分考虑了硬件资源受限的特殊工作环境。它采用一种很巧妙的办法减少自己的体积：所有的命令都通过“插件”的方式集中到一个可执行文件中，在实际应用过程中通过不同的符号链接来确定到底要执行哪个操作。例如最终生成的可执行文件为 busybox，当为它建立一个符号链接 ls 的时候，就可以通过执行这个新命令实现列出目录的功能。采用单一执行文件的方式最大限度地共享了程序代码，甚至连文件头、内存中的程序控制块等其他系统资源都共享了，对于资源比较紧张的系统来说，真是最合适不过了。在 busybox 的编译过程中，可以非常方便地加减它的“插件”，最后的符号链接也可以由编译系统自动生成。

下面用 busybox 构建 FS2410 开发板的 cramfs 文件系统。

首先从 busybox 网站下载 busybox 源码（本实例采用的 busybox-1.0.0）并解压，接下来，根据实际需要进行 busybox 的配置。

```
[root@localhost fs2410]# tar jxvf busybox-1.00.tar.bz2
[root@localhost fs2410]# cd busybox-1.00
[root@localhost busybox-1.00]# make defconfig  /* 首先进行默认配置 */
```

```
[root@localhost busybox-1.00]# make menuconfig
```

此时需要设置平台相关的交叉编译选项，操作步骤为：先选中“Build Options”项的“Do you want to build Busybox with a Cross Complier？”选项，然后将“Cross Compiler prefix”设置为“/usr/local/arm/3.3.2/bin/arm-linux-”（这是在实验主机中的交叉编译器的安装路径）。

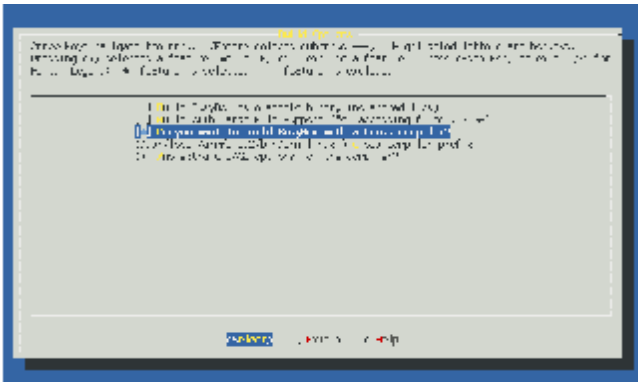

图 5.26　busybox 配置画面

下一步编译并安装 busybox。

```
[root@localhost busybox-1.00]# make
[root@localhost busybox-1.00]# make install
PREFIX=/home/david/fs2410/cramfs
```

其中，PREFIX 用于指定安装目录，如果不设置该选项，则默认在当前目录下创建_install 目录。创建的安装目录的内容如下所示：

```
[root@localhost cramfs]# ls
bin  linuxrc  sbin  usr
```

从此可知，使用 busybox 软件包所创建的文件系统还缺少很多东西。下面我们通过创建系统所需要的目录和文件来完善一下文件系统的内容。

```
[root@localhost cramfs]# mkdir mnt root var tmp proc boot etc lib
[root@localhost cramfs]# mkdir /var/{lock,log,mail,run,spool}
```

如果 busybox 是动态编译的（即在配置 busybox 时没选中静态编译），则把所需的交叉编译的动态链接库文件复制到 lib 目录中。

接下来，需要创建一些重要文件。首先要创建/etc/inittab 和/etc/fstab 文件。inittab 是 Linux 启动之后第一个被访问的脚本文件，而 fstab 文件是定义了文件系统的各个“挂接点”，需要与实际的系统相配合。接下来要创建用户和用户组文件。

以上用 busybox 构造了文件系统的内容，下面要创建 cramfs 文件系统映像文件。制作 cramfs 映像文件需要用到的工具是 mkcramfs。此时可以采用两种方法，一种方法是使用我们所构建的文件系统（在目录“/home/david/fs2410/cramfs”中），另一种方法是在已经做好的 cramfs 映像文件的基础上进行适当的改动。下面的示例使用第二种方法，因为这个方法包含了第一种方法的所有步骤（假设已经做好

的映像文件名为“fs2410.cramfs”)。

首先用 mount 命令将映像文件挂载到一个目录下，打开该目录并查看其内容。

```
[root@localhost fs2410]# mkdir cramfs
[root@localhost fs2410]# mount fs2410.cramgs cramfs –o loop
[root@localhost fs2410]# ls cramfs
bin  dev  etc  home  lib  linuxrc  proc  Qtopia  ramdisk  sbin
testshell  tmp  usr  var
```

因为 cramfs 文件系统是只读的，所以不能在这个挂载目录下直接进行修改，因此需要将文件系统中的内容复制到另一个目录中，具体操作如下所示：

```
[root@localhost fs2410]# mkdir backup_cramfs
[root@localhost fs2410]# tar cvf backup.cramfs.tar  cramfs/
[root@localhost fs2410]# mv backup.cramfs.tar backup_cramfs/
[root@localhost fs2410]# umount cramfs
[root@localhost fs2410]# cd backup_cramfs
[root@localhost backup_cramfs]# tar zvf backup.cramfs.tar
[root@localhost backup_cramfs]# rm backup.cramfs.tar
```

此时我们就像用 busybox 所构建的文件系统一样，可以在 backup_cramfs 的 cramfs 子目录中任意进行修改。例如可以添加用户自己的程序：

```
[root@localhost fs2410]# cp ~/hello backup_cramfs/cramfs/
```

在用户的修改工作结束之后，用下面的命令可以创建 cramfs 映像文件：

```
[root@localhost fs2410]#  mkcramfs backup_cramfs/cramfs/ new.cramfs
```

接下来，就可以将新创建的 new.cramfs 映像文件烧入到开发板的相应位置了。

2．NFS 文件系统

NFS 为 Network File System 的简称，最早是由 Sun 公司提出发展起来的，其目的就是让不同的机器、不同的操作系统之间通过网络可以彼此共享文件。NFS 可以让不同的主机通过网络将远端的 NFS 服务器共享出来的文件安装到自己的系统中，从客户端看来，使用 NFS 的远端文件就像是使用本地文件一样。在嵌入式中使用 NFS 会使应用程序的开发变得十分方便，并且不用反复地烧写映像文件。

NFS 的使用分为服务端和客户端，其中服务端是提供要共享的文件，而客户端则通过挂载（“mount”）这一动作来实现对共享文件的访问操作。下面主要介绍 NFS 服务端的使用。在嵌入式开发中，通常 NFS 服务端在宿主机上运行，而客户端在目标板上运行。

NFS 服务端是通过读入它的配置文件“/etc/exports”来决定所共享的文件目录的。下面首先讲解这个配置文件的书写规范。

在这个配置文件中，每一行都代表一项要共享的文件目录以及所指定的客户端对它的操作权限。客户端可以根据相应的权限，对该目录下的所有目录文件进行访问。

配置文件中每一行的格式如下：

```
[共享的目录] [客户端主机名称或 IP] [参数 1，参数 2…]
```

在这里，主机名或 IP 是可供共享的客户端主机名或 IP，若对所有的 IP 都可以访问，则可用“*”表示。这里的参数有很多种组合方式，常见的参数如表 5.1 所示。

表 5.1　　常见参数

选　项	参 数 含 义
rw	可读写的权限
ro	只读的权限
no_root_squash	NFS 客户端分享目录使用者的权限，即如果客户端使用的是 root 用户，那么对于这个共享的目录而言，该客户端就具有 root 的权限
sync	资料同步写入到内存与硬盘当中
async	资料会先暂存于内存当中，而非直接写入硬盘

如在本例中，配置文件“/etc/exports”的代码如下：

```
[root@localhost fs]# cat /etc/exports
/root/workplace    192.168.1.*(rw,no_root_squash)
```

在设定完配置文件之后，需要启动 nfs 服务和 portmap 服务，这里的 portmap 服务是允许 NFS 客户端查看 NFS 服务在用的端口，在它被激活之后，就会出现一个端口号为 111 的 sun RPC（远端过程调用）的服务。这是 NFS 服务中必须实现的一项，因此，也必须把它开启。如下所示：

```
[root@localhost fs]# service portmap start
启动 portmap:                                [确定]
[root@localhost fs]# service nfs start
启动 NFS 服务:                               [确定]
关掉 NFS 配额:                               [确定]
启动 NFS 守护进程:                           [确定]
启动 NFS mountd:                             [确定]
```

可以看到，在启动 NFS 服务的时候启动了 mountd 进程。这是 NFS 挂载服务，用于处理 NFS 递交过来的客户端请求。另外还会激活至少两个以上的系统守护进程，然后就开始监听客户端的请求，用“cat /var/log/messages”命令可以查看操作是否成功。这样，就启动了 NFS 的服务，另外还有两个命令，可以便于使用 NFS。

其中一个是 exportfs，它可以重新扫描“/etc/exports”，使用户在修改了“/etc/exports”配置文件之后不需要每次重启 NFS 服务。其格式为：

exportfs [选项]

exportfs 的常见选项如表 5.2 所示。

表 5.2　　常见选项

选　项	参 数 含 义

-a	全部挂载（或卸载）/etc/exports 中的设定文件目录
-r	重新挂载/etc/exports 中的设定文件目录
-u	卸载某一目录
-v	在 export 的时候，将共享的目录显示到屏幕上

另外一个是 showmount 命令，它用于当前的挂载情况。其格式为：

```
showmount [选项] hostname
```

showmount 的常见选项如表 5.3 所示。

表 5.3　　常见选项

选　项	参 数 含 义
-a	在屏幕上显示目前主机与客户端所连上来的使用目录状态
-e	显示 hostname 中/etc/exports 里设定的共享目录

5.2　U-Boot 移植

5.2.1　Bootloader 介绍

1．概念

简单地说，Bootloader 就是在操作系统内核运行之前运行的一段程序，它类似于 PC 机中的 BIOS 程序。通过这段程序，可以完成硬件设备的初始化，并建立内存空间的映射关系，从而将系统的软硬件环境带到一个合适的状态，为最终加载系统内核做好准备。

通常，Bootloader 比较依赖于硬件平台，特别是在嵌入式系统中，更为如此。因此，在嵌入式世界里建立一个通用的 Bootloader 是一件比较困难的事情。尽管如此，仍然可以对 Bootloader 归纳出一些通用的概念来指导面向用户定制的 Bootloader 设计与实现。

（1）Bootloader 所支持的 CPU 和嵌入式开发板。

每种不同的 CPU 体系结构都有不同的 Bootloader。有些 Bootloader 也支持多种体系结构的 CPU，如后面要介绍的 U-Boot 支持 ARM、MIPS、PowerPC 等众多体系结构。除了依赖于 CPU 的体系结构外，Bootloader 实际上也依赖于具体的嵌入式板级设备的配置。

（2）Bootloader 的存储位置。

系统加电或复位后，所有的 CPU 通常都从某个由 CPU 制造商预先安排的地址上取指令。而基于 CPU 构建的嵌入式系统通常都有某种类型的固态存储设备（比如 ROM、EEPROM 或 Flash 等）被映射到这个预先安排的地址上。因此在系统加电后，CPU 将首先执行 Bootloader 程序。

华清远见

（3）Bootloader 的启动过程分为单阶段和多阶段两种。通常多阶段的 Bootloader 能提供更为复杂的功能，以及更好的可移植性。

（4）Bootloader 的操作模式。大多数 Bootloader 都包含两种不同的操作模式："启动加载"模式和"下载"模式，这种区别仅对于开发人员才有意义。

- 启动加载模式：这种模式也称为"自主"模式。也就是 Bootloader 从目标机上的某个固态存储设备上将操作系统加载到 RAM 中运行，整个过程并没有用户的介入。这种模式是嵌入式产品发布时的通用模式。
- 下载模式：在这种模式下，目标机上的 Bootloader 将通过串口连接或网络连接等通信手段从主机（Host）下载文件，比如：下载内核映像和根文件系统映像等。从主机下载的文件通常首先被 Bootloader 保存到目标机的 RAM 中，然后再被 Bootloader 写入到目标机上的 Flash 类固态存储设备中。Bootloader 的这种模式在系统更新时使用。工作于这种模式下的 Bootloader 通常都会向它的终端用户提供一个简单的命令行接口。

（5）Bootloader 与主机之间进行文件传输所用的通信设备及协议，最常见的情况就是，目标机上的 Bootloader 通过串口与主机之间进行文件传输，传输协议通常是 xmodem/ ymodem/zmodem 等。但是，串口传输的速度是有限的，因此通过以太网连接并借助 TFTP 等协议来下载文件是个更好的选择。

2．Bootloader 启动流程

Bootloader 的启动流程一般分为两个阶段：stage1 和 stage2，下面分别对这两个阶段进行讲解。

（1）Bootloader 的 stage1。

在 stage1 中 Bootloader 主要完成以下工作。

- 基本的硬件初始化，包括屏蔽所有的中断、设置 CPU 的速度和时钟频率、RAM 初始化、初始化外围设备、关闭 CPU 内部指令和数据 cache 等。
- 为加载 stage2 准备 RAM 空间，通常为了获得更快的执行速度，通常把 stage2 加载到 RAM 空间中来执行，因此必须为加载 Bootloader 的 stage2 准备好一段可用的 RAM 空间。
- 复制 stage2 到 RAM 中，在这里要确定两点：①stage2 的可执行映像在固态存储设备的存放起始地址和终止地址；②RAM 空间的起始地址。
- 设置堆栈指针 sp，这是为执行 stage2 的 C 语言代码做好准备。

（2）Bootloader 的 stage2。

在 stage2 中 Bootloader 主要完成以下工作。

- 用汇编语言跳转到 main 入口函数。

由于 stage2 的代码通常用 C 语言来实现，目的是实现更复杂的功能和取得更好的代码可读性和可移植性。但是与普通 C 语言应用程序不同的是，在编译和链接 Bootloader 这样的程序时，不能使用 glibc 库中的任何支持函数。

- 初始化本阶段要使用到的硬件设备，包括初始化串口、初始化计时器等。在初始化这些设备之前可以输出一些打印信息。

- 检测系统的内存映射，所谓内存映射就是指在整个 4GB 物理地址空间中指出哪些地址范围被分配用来寻址系统的内存。
- 加载内核映像和根文件系统映像，这里包括规划内存占用的布局和从 Flash 上复制数据。
- 设置内核的启动参数。

5.2.2 U-Boot 概述

1．U-Boot 简介

U-Boot（UniversalBootloader）是遵循 GPL 条款的开放源码项目。它是从 FADSROM、8xxROM、PPCBOOT 逐步发展演化而来。其源码目录、编译形式与 Linux 内核很相似，事实上，不少 U-Boot 源码就是相应的 Linux 内核源程序的简化，尤其是一些设备的驱动程序，这从 U-Boot 源码的注释中能体现这一点。但是 U-Boot 不仅仅支持嵌入式 Linux 系统的引导，而且还支持 NetBSD、VxWorks、QNX、RTEMS、ARTOS、LynxOS 等嵌入式操作系统。其目前要支持的目标操作系统是 OpenBSD、NetBSD、FreeBSD、4.4BSD、Linux、SVR4、Esix、Solaris、Irix、SCO、Dell、NCR、VxWorks、LynxOS、pSOS、QNX、RTEMS、ARTOS。这是 U-Boot 中 Universal 的一层含义，另外一层含义则是 U-Boot 除了支持 PowerPC 系列的处理器外，还能支持 MIPS、x86、ARM、NIOS、XScale 等诸多常用系列的处理器。这两个特点正是 U-Boot 项目的开发目标，即支持尽可能多的嵌入式处理器和嵌入式操作系统。就目前为止，U-Boot 对 PowerPC 系列处理器支持最为丰富，对 Linux 的支持最完善。

2．U-Boot 特点

U-Boot 的特点如下。

- 开放源码；
- 支持多种嵌入式操作系统内核，如 Linux、NetBSD、VxWorks、QNX、RTEMS、ARTOS、LynxOS；
- 支持多个处理器系列，如 PowerPC、ARM、x86、MIPS、XScale；
- 较高的可靠性和稳定性；
- 高度灵活的功能设置，适合 U-Boot 调试、操作系统不同引导要求和产品发布等；
- 丰富的设备驱动源码，如串口、以太网、SDRAM、Flash、LCD、NVRAM、EEPROM、RTC、键盘等；
- 较为丰富的开发调试文档与强大的网络技术支持。

3．U-Boot 主要功能

U-Boot 可支持的主要功能列表。

- 系统引导：支持 NFS 挂载、RAMDISK（压缩或非压缩）形式的根文件系统。支持 NFS 挂载，并从 Flash 中引导压缩或非压缩系统内核。
- 基本辅助功能：强大的操作系统接口功能；可灵活设置、传递多个关键参数

华清远见

给操作系统，适合系统在不同开发阶段的调试要求与产品发布，尤其对 Linux 支持最为强劲；支持目标板环境参数多种存储方式，如 Flash、NVRAM、EEPROM；CRC32 校验，可校验 Flash 中内核、RAMDISK 映像文件是否完好。

- 设备驱动：串口、SDRAM、Flash、以太网、LCD、NVRAM、EEPROM、键盘、USB、PCMCIA、PCI、RTC 等驱动支持。
- 上电自检功能：SDRAM、Flash 大小自动检测；SDRAM 故障检测；CPU 型号。
- 特殊功能：XIP 内核引导。

5.2.3 U-Boot 源码导读

1．U-Boot 源码结构

U-Boot 源码结构如图 5.27 所示。

图 5.27 U-Boot 源码

- board：和一些已有开发板有关的代码，比如 makefile 和 U-Boot.lds 等都和具体开发板的硬件和地址分配有关。
- common：与体系结构无关的代码，用来实现各种命令的 C 程序。
- cpu：包含 CPU 相关代码，其中的子目录都是以 U-BOOT 所支持的 CPU 为名，比如有子目录 arm926ejs、mips、mpc8260 和 nios 等，每个特定的子目录中都包括 cpu.c 和 interrupt.c，start.S 等。其中 cpu.c 初始化 CPU、设置指令 Cache 和数据 Cache 等；interrupt.c 设置系统的各种中断和异常，比如快速中断、开关中断、时钟中断、软件中断、预取中止和未定义指令等；汇编代码文件 start.S 是 U-BOOT 启动时执行的第一个文件，它主要是设置系统堆栈和工作方式，为进入 C 程序奠定基础。
- disk：disk 驱动的分区相关代码。
- doc：文档。
- drivers：通用设备驱动程序，比如各种网卡、支持 CFI 的 Flash、串口和 USB 总线等。
- fs：支持文件系统的文件，U-BOOT 现在支持 cramfs、fat、fdos、jffs2 和 registerfs 等。
- include：头文件，还有对各种硬件平台支持的汇编文件，系统的配置文件和对文件系统支持的文件。
- net：与网络有关的代码，BOOTP 协议、TFTP 协议、RARP 协议和 NFS 文件系统的实现。
- lib_arm：与 ARM 体系结构相关的代码。
- tools：创建 S-Record 格式文件和 U-BOOT images 的工具。

2．U-Boot 重要代码

（1）cpu/arm920t/start.S

这是 U-Boot 的起始位置。在这个文件中设置了处理器的状态、初始化中断向量和内存时序等，从 Flash 中跳转到定位好的内存位置执行。

```
.globl_start （起始位置：中断向量设置）
_start:    b       reset
     ldr   pc, _undefined_instruction
     ldr   pc, _software_interrupt
     ldr   pc, _prefetch_abort
     ldr   pc, _data_abort
     ldr   pc, _not_used
     ldr   pc, _irq
     ldr   pc, _fiq

_undefined_instruction:   .word undefined_instruction
_software_interrupt:   .word software_interrupt
_prefetch_abort:   .word prefetch_abort
_data_abort:      .word data_abort
_not_used:     .word not_used
_irq:          .word irq
_fiq:          .word fiq

   _TEXT_BASE:（代码段起始位置）
.word   TEXT_BASE

.globl _armboot_start
_armboot_start:
   .word _start

/*
 * These are defined in the board-specific linker script.
 */
.globl _bss_start （BSS 段起始位置）
_bss_start:
     .word __bss_start

.globl _bss_end
_bss_end:
    .word _end
```

```
    reset: （执行入口）
          /*
          * set the cpu to SVC32 mode;使处理器进入特权模式
          */
          mrs    r0,cpsr
          bic    r0,r0,#0x1f
          orr    r0,r0,#0xd3
          msr    cpsr,r0
    relocate:    （代码的重置）          /* relocate U-Boot to RAM    */
         adr    r0, _start     /* r0 <- current position of code   */
         ldr    r1, _TEXT_BASE     /* test if we run from flash or RAM */
         cmp    r0, r1               /* don't reloc during debug       */
         beq    stack_setup

         ldr    r2, _armboot_start
         ldr    r3, _bss_start
         sub    r2, r3, r2    /* r2 <- size of armboot          */
         add    r2, r0, r2    /* r2 <- source end address       */

    copy_loop: （拷贝过程）
         ldmia r0!, {r3-r10}     /* copy from source address [r0]    */
         stmia r1!, {r3-r10}     /* copy to   target address [r1]    */
         cmp   r0, r2          /* until source end addreee [r2]    */
         ble   copy_loop

         /* Set up the stack; 设置堆栈  */
    stack_setup:
         ldr   r0, _TEXT_BASE     /* upper 128 KiB: relocated uboot   */
         sub       r0,   r0,   #CFG_MALLOC_LEN         /*   malloc   area
*/
         sub   r0, r0, #CFG_GBL_DATA_SIZE /* bdinfo                    */

    clear_bss: （清空 BSS 段）
         ldr    r0, _bss_start    /* find start of bss segment       */
         ldr    r1, _bss_end       /* stop here                    */
         mov    r2, #0x00000000    /* clear                         */

    clbss_l:str    r2, [r0]       /* clear loop...                 */
         add   r0, r0, #4
         cmp   r0, r1
```

```
    bne   clbss_l
    ldr   pc, _start_armboot

_start_armboot:    .word start_armboot
```

（2）interrupts.c

这个文件是处理中断的，如打开和关闭中断等。

```
#ifdef CONFIG_USE_IRQ
/* enable IRQ interrupts; 中断使能函数 */
void enable_interrupts (void)
{
    unsigned long temp;
    __asm__ __volatile__("mrs %0, cpsr\n"
                         "bic %0, %0, #0x80\n"
                         "msr cpsr_c, %0"
                         : "=r" (temp)
                         :
                         : "memory");
}

/*
 * disable IRQ/FIQ interrupts; 中断屏蔽函数
 * returns true if interrupts had been enabled before we disabled them
 */
int disable_interrupts (void)
{
    unsigned long old,temp;
    __asm__ __volatile__("mrs %0, cpsr\n"
                         "orr %1, %0, #0xc0\n"
                         "msr cpsr_c, %1"
                         : "=r" (old), "=r" (temp)
                         :
                         : "memory");
    return (old & 0x80) == 0;
}
#endif
void show_regs (struct pt_regs *regs)
{
    unsigned long flags;
    const char *processor_modes[] = {
```

```
        "USER_26", "FIQ_26",  "IRQ_26",  "SVC_26",
        "UK4_26",  "UK5_26",  "UK6_26",  "UK7_26",
        "UK8_26",  "UK9_26",  "UK10_26", "UK11_26",
        "UK12_26", "UK13_26", "UK14_26", "UK15_26",
        "USER_32", "FIQ_32",  "IRQ_32",  "SVC_32",
        "UK4_32",  "UK5_32",  "UK6_32",  "ABT_32",
        "UK8_32",  "UK9_32",  "UK10_32", "UND_32",
        "UK12_32", "UK13_32", "UK14_32", "SYS_32",
        };
    …
    }
    /* 在 U-Boot 启动模式下，在原则上要禁止中断处理，所以如果发生中断，当作出错处理 */
    void do_fiq (struct pt_regs *pt_regs)
    {
        printf ("fast interrupt request\n");
        show_regs (pt_regs);
        bad_mode ();
    }

    void do_irq (struct pt_regs *pt_regs)
    {
        printf ("interrupt request\n");
        show_regs (pt_regs);
        bad_mode ();
    }
```

（3）cpu.c

这个文件是对处理器进行操作，如下所示：

```
    int cpu_init (void)
    {
        /*
          * setup up stacks if necessary; 设置需要的堆栈
        */
    #ifdef CONFIG_USE_IRQ
        DECLARE_GLOBAL_DATA_PTR;

        IRQ_STACK_START=_armboot_start - CFG_MALLOC_LEN - CFG_GBL_DATA_SIZE -
4;
        FIQ_STACK_START = IRQ_STACK_START - CONFIG_STACKSIZE_IRQ;
    #endif
```

```
    return 0;
}
int cleanup_before_linux (void) /* 准备加载 linux */
{
    /*
     * this function is called just before we call linux
     * it prepares the processor for linux
     *
     * we turn off caches etc ...
     */
    unsigned long i;
    disable_interrupts ();

    /* turn off I/D-cache: 关闭 cache */
    asm ("mrc p15, 0, %0, c1, c0, 0":"=r" (i));
    i &= ~(C1_DC | C1_IC);
    asm ("mcr p15, 0, %0, c1, c0, 0": :"r" (i));

    /* flush I/D-cache */
    i = 0;
    asm ("mcr p15, 0, %0, c7, c7, 0": :"r" (i));
    return (0);
}

OUTPUT_ARCH(arm)
ENTRY(_start)
SECTIONS
{
    . = 0x00000000;
       . = ALIGN(4);
    .text      :
    {
      cpu/arm920t/start.o (.text)
      *(.text)
    }

    . = ALIGN(4);
    .rodata : { *(.rodata) }

    . = ALIGN(4);
```

```
    .data : { *(.data) }

    . = ALIGN(4);
    .got : { *(.got) }

    __u_boot_cmd_start = .;
    .u_boot_cmd : { *(.u_boot_cmd) }
    __u_boot_cmd_end = .;

    . = ALIGN(4);
    __bss_start = .;
    .bss : { *(.bss) }
    _end = .;
}
```

（4）memsetup.S

这个文件是用于配置开发板参数的，如下所示：

```
/* memsetup.c */
    /* memory control configuration */
    /* make r0 relative the current location so that it */
    /* reads SMRDATA out of FLASH rather than memory ! */
    ldr    r0, =SMRDATA
   ldr r1, _TEXT_BASE
   sub r0, r0, r1
   ldr r1, =BWSCON /* Bus Width Status Controller */
    add    r2, r0, #52
0:
    ldr    r3, [r0], #4
    str    r3, [r1], #4
    cmp    r2, r0
    bne    0b

    /* everything is fine now */
   mov pc, lr

    .ltorg
```

5.2.4 U-Boot 移植主要步骤

（1）建立自己的开发板类型。

阅读 makefile 文件，在 makefile 文件中添加两行，如下所示：

```
fs2410_config: unconfig
    @./mkconfig $(@:_config=) arm arm920t fs2410
```

其中“arm”为表示处理器体系结构的种类，“arm920t”表示处理器体系结构的名称，“fs2410”为主板名称。

在 board 目录中建立 fs2410 目录，并将 smdk2410 目录中的内容（cp –a smdk2410/* fs2410）复制到该目录中。

- 在 include/configs/目录下将 smdk2410.h 复制到（cp smdk2410.h fs2410.h）。
- 修改 ARM 编译器的目录名及前缀（都要改成以“fs2410”开头）。
- 完成之后，可以测试配置。

```
$ make fs2410_config;make
```

（2）修改程序链接地址。

在 board/s3c2410 中有一个 config.mk 文件，它是用于设置程序链接的起始地址，因为会在 U-Boot 中增加功能，所以留下 6MB 的空间，修改 33F80000 为 33A00000。

为了以后能用 U-Boot 的“go”命令执行修改过的用 loadb 或 tftp 下载的 U-Boot，需要在 board/ s3c2410 的 memsetup.S 中标记符”0:”上加入 5 句：

```
mov r3, pc
ldr r4, =0x3FFF0000
and r3, r3, r4 （以上 3 句得到实际代码启动的内存地址）
aad r0, r0, r3 （用 go 命令调试 u-boot 时，启动地址在 RAM）
add r2, r2, r3 （把初始化内存信息的地址，加上实际启动地址）
```

（3）将中断禁止的部分应该改为如下所示（/cpu/arm920t/start.S）：

```
# if defined(CONFIG_S3C2410)
    ldr    r1, =0x7ff
    ldr    r0, =INTSUBMSK
    str    r1, [r0]
# endif
```

（4）因为在 fs2410 开发板启动时是直接从 Nand Flash 加载代码，所以启动代码应该改成如下所示（/cpu/arm920t/start.S）：

```
#ifdef CONFIG_S3C2410_NAND_BOOT  @START
@ reset NAND
   mov r1, #NAND_CTL_BASE
   ldr   r2, =0xf830          @ initial value
   str   r2, [r1, #oNFCONF]
   ldr   r2, [r1, #oNFCONF]
   bic  r2, r2, #0x800            @ enable chip
   str   r2, [r1, #oNFCONF]
```

```
    mov r2, #0xff       @ RESET command
    strb r2, [r1, #oNFCMD]
    mov r3, #0              @ wait
    nand1:
    add  r3, r3, #0x1
    cmp r3, #0xa
    blt   nand1
nand2:
    ldr   r2, [r1, #oNFSTAT]     @ wait ready
    tst    r2, #0x1
    beq  nand2
    ldr   r2, [r1, #oNFCONF]
    orr  r2, r2, #0x800           @ disable chip
    str   r2, [r1, #oNFCONF]
    @ get read to call C functions (for nand_read())
    ldr   sp, DW_STACK_START      @ setup stack pointer
    mov fp, #0                @ no previous frame, so fp=0
@ copy U-Boot to RAM
    ldr   r0, =TEXT_BASE
    mov    r1, #0x0
    mov r2, #0x20000
    bl    nand_read_ll
    tst    r0, #0x0
    beq  ok_nand_read
bad_nand_read:
    loop2:    b    loop2        @ infinite loop
ok_nand_read:
@ verify
    mov r0, #0
    ldr   r1, =TEXT_BASE
    mov r2, #0x400    @ 4 bytes * 1024 = 4K-bytes
go_next:
    ldr   r3, [r0], #4
    ldr   r4, [r1], #4
    teq   r3, r4
    bne  notmatch
    subs r2, r2, #4
    beq  stack_setup
    bne  go_next
notmatch:
```

```
loop3:    b    loop3        @ infinite loop
#endif @ CONFIG_S3C2410_NAND_BOOT  @END
```

在“ _start_armboot: .word start_armboot ”后加入：

```
.align     2
DW_STACK_START:  .word  STACK_BASE+STACK_SIZE-4
```

（5）修改内存配置（board/fs2410/lowlevel_init.S）。

```
#define BWSCON    0x48000000
#define PLD_BASE   0x2C000000
#define SDRAM_REG  0x2C000106

/* BWSCON */
#define DW8            (0x0)
#define DW16           (0x1)
#define DW32           (0x2)
#define WAIT           (0x1<<2)
#define UBLB           (0x1<<3)

/* BANKSIZE */
#define BURST_EN       (0x1<<7)

#define B1_BWSCON     (DW16 + WAIT)
#define B2_BWSCON     (DW32)
#define B3_BWSCON     (DW32)
#define B4_BWSCON     (DW16 + WAIT + UBLB)
#define B5_BWSCON     (DW8 + UBLB)
#define B6_BWSCON     (DW32)
#define B7_BWSCON     (DW32)
/* BANK0CON */
#define B0_Tacs           0x0 /*  0clk */
#define B0_Tcos           0x1 /*  1clk */
#define B0_Tacc           0x7 /*  14clk */
#define B0_Tcoh           0x0 /*  0clk */
#define B0_Tah      0x0   /*  0clk */
#define B0_Tacp          0x0    /* page mode is not used */
#define B0_PMC        0x0 /* page mode disabled */

/* BANK1CON */
#define B1_Tacs           0x0 /*  0clk */
```

```
    #define B1_Tcos           0x1 /*  1clk */
    #define B1_Tacc           0x7 /*  14clk */
    #define B1_Tcoh           0x0 /*  0clk */
    #define B1_Tah         0x0 /*  0clk */
    #define B1_Tacp          0x0    /* page mode is not used */
    #define B1_PMC         0x0 /* page mode disabled */
    ……
    /* REFRESH parameter */
    #define REFEN          0x1 /* Refresh enable */
    #define TREFMD         0x0 /* CBR(CAS before RAS)/Auto refresh */
    #define Trp        0x0 /* 2clk */
    #define Trc        0x3 /* 7clk */
    #define Tchr          0x2 /* 3clk */
    #define REFCNT         1113 /*period=15.6us,HCLK=60Mhz, (2048+1-15.6*60)
*/
    ......
        .word ((B6_MT<<15)+(B6_Trcd<<2)+(B6_SCAN))
        .word ((B7_MT<<15)+(B7_Trcd<<2)+(B7_SCAN))
        .word
((REFEN<<23)+(TREFMD<<22)+(Trp<<20)+(Trc<<18)+(Tchr<<16)+REFCNT)
        .word 0x32
        .word 0x30
        .word 0x30
```

（6）加入 Nand Flash 读函数（创建 board/fs2410/nand_read.c 文件）。

```
    #include <config.h>
    #define __REGb(x) (*(volatile unsigned char *)(x))
    #define __REGi(x) (*(volatile unsigned int *)(x))
    #define NF_BASE  0x4e000000
    #define NFCONF  __REGi(NF_BASE + 0x0)
    #define NFCMD  __REGb(NF_BASE + 0x4)
    #define NFADDR  __REGb(NF_BASE + 0x8)
    #define NFDATA  __REGb(NF_BASE + 0xc)
    #define NFSTAT  __REGb(NF_BASE + 0x10)
    #define BUSY 1
    inline void wait_idle(void)
    {
        Int i;
        while(!(NFSTAT & BUSY))
        {
```

```
            for (i = 0; i < 10; i++);
        }
    }

    /* low level nand read function */
    int nand_read_ll(unsigned char *buf, unsigned long start_addr, int size)
    {
        int i, j;
        if ((start_addr & NAND_BLOCK_MASK) || (size & NAND_BLOCK_MASK))
        {
            return -1; /* invalid alignment */
        }
        /* chip Enable */
        NFCONF &= ~0x800;
        for (i = 0; i < 10; i++);
        for (i = start_addr; i < (start_addr + size);)
        {
            /* READ0 */
            NFCMD = 0;
            /* Write Address */
            NFADDR = i & 0xff;
            NFADDR = (i >> 9) & 0xff;
            NFADDR = (i >> 17) & 0xff;
            NFADDR = (i >> 25) & 0xff;
            wait_idle();
            for (j = 0; j < NAND_SECTOR_SIZE; j++, i++)
            {
                *buf = (NFDATA & 0xff);
                buf++;
            }
        }
        /* chip Disable */
        NFCONF |= 0x800; /* chip disable */
        return 0;
    }
```

修改 board/fs2410/makefile 文件，以增加 nand_read()函数。

```
OBJS := fs2410.o  flash.o  nand_read.o
```

（7）加入 Nand Flash 的初始化函数（board/fs2410/fs2410.c）。

```
#if (CONFIG_COMMANDS & CFG_CMD_NAND)
typedef enum
{
    NFCE_LOW,
    NFCE_HIGH
} NFCE_STATE;
static inline void NF_Conf(u16 conf)
{
    S3C2410_NAND * const nand = S3C2410_GetBase_NAND();
    nand->NFCONF = conf;
}
static inline void NF_Cmd(u8 cmd)
{
    S3C2410_NAND * const nand = S3C2410_GetBase_NAND();
    nand->NFCMD = cmd;
}
static inline void NF_CmdW(u8 cmd)
{
    NF_Cmd(cmd);
    udelay(1);
}
static inline void NF_Addr(u8 addr)
{
    S3C2410_NAND * const nand = S3C2410_GetBase_NAND();
    nand->NFADDR = addr;
}
static inline void NF_SetCE(NFCE_STATE s)
{
    S3C2410_NAND * const nand = S3C2410_GetBase_NAND();
    switch (s)
    {
        case NFCE_LOW:
           nand->NFCONF &= ~(1<<11);
           break;
        case NFCE_HIGH:
           nand->NFCONF |= (1<<11);
           break;
    }
}
static inline void NF_WaitRB(void)
```

```
{
    S3C2410_NAND * const nand = S3C2410_GetBase_NAND();
    while (!(nand->NFSTAT & (1<<0)));
}
static inline void NF_Write(u8 data)
{
    S3C2410_NAND * const nand = S3C2410_GetBase_NAND();
    nand->NFDATA = data;
}
static inline u8 NF_Read(void)
{
    S3C2410_NAND * const nand = S3C2410_GetBase_NAND();
    return(nand->NFDATA);
}
static inline void NF_Init_ECC(void)
{
    S3C2410_NAND * const nand = S3C2410_GetBase_NAND();
    nand->NFCONF |= (1<<12);
}
static inline u32 NF_Read_ECC(void)
{
    S3C2410_NAND * const nand = S3C2410_GetBase_NAND();
    return(nand->NFECC);
}
#endif
/*
* NAND flash initialization.
*/
#if (CONFIG_COMMANDS & CFG_CMD_NAND)
extern ulong nand_probe(ulong physadr);
static inline void NF_Reset(void)
{
    int i;
    NF_SetCE(NFCE_LOW);
    NF_Cmd(0xFF); /* reset command */
    for (i = 0; i < 10; i++); /* tWB = 100ns. */
    NF_WaitRB(); /* wait 200~500us; */
    NF_SetCE(NFCE_HIGH);
}
static inline void NF_Init(void)
```

```
{
    #define TACLS 0
    #define TWRPH0 4
    #define TWRPH1 2
    NF_Conf((1<<15)|(0<<14)|(0<<13)|(1<<12)
|(1<<11)|(TACLS<<8)|(TWRPH0<<4)|(TWRPH1<<0));
    /* 1 1 1 1, 1 xxx, r xxx, r xxx */
    /* En 512B 4step ECCR nFCE=H tACLS tWRPH0 tWRPH1 */
    NF_Reset();
}
void nand_init(void)
{
    S3C2410_NAND * const nand = S3C2410_GetBase_NAND();
    NF_Init();
    #ifdef DEBUG
       printf("NAND flash probing at 0x%.8lX\n", (ulong)nand);
    #endif
    printf ("%4lu MB\n", nand_probe((ulong)nand) >> 20);
}
#endif
```

（8）修改 GPIO 配置（board/fs2410/fs2410.c）。

```
/* set up the I/O ports */
gpio->GPACON = 0x007FFFFF;
gpio->GPBCON = 0x002AAAAA;
gpio->GPBUP = 0x000002BF;
gpio->GPCCON = 0xAAAAAAAA;
gpio->GPCUP = 0x0000FFFF;
gpio->GPDCON = 0xAAAAAAAA;
gpio->GPDUP = 0x0000FFFF;
gpio->GPECON = 0xAAAAAAAA;
gpio->GPEUP = 0x000037F7;
gpio->GPFCON = 0x00000000;
gpio->GPFUP = 0x00000000;
gpio->GPGCON = 0xFFEAFF5A;
gpio->GPGUP = 0x0000F0DC;
gpio->GPHCON = 0x0018AAAA;
gpio->GPHDAT = 0x000001FF;
gpio->GPHUP = 0x00000656
```

（9）提供 nand flash 相关宏定义（include/configs/fs2410.h），具体参考源码。

（10）加入 Nand Flash 设备（include/linux/mtd/nand_ids.h）

```
    static struct nand_flash_dev nand_flash_ids[] =
    {
        ......
       {"Samsung KM29N16000",NAND_MFR_SAMSUNG, 0x64, 21, 1, 2, 0x1000, 0},
       {"Samsung K9F1208U0M",  NAND_MFR_SAMSUNG, 0x76, 26, 0, 3, 0x4000,
0},
       {"Samsung unknown 4Mb", NAND_MFR_SAMSUNG, 0x6b, 22, 0, 2, 0x2000,
0},
        ......
        {NULL,}
    };
```

（11）设置 Nand Flash 环境（common/env_nand.c）

```
    int nand_legacy_rw (struct nand_chip* nand, int cmd,
           size_t start, size_t len,
           size_t * retlen, u_char * buf);
    extern struct nand_chip nand_dev_desc[CFG_MAX_NAND_DEVICE];
    extern int nand_legacy_erase(struct nand_chip *nand,
                                 size_t ofs, size_t len, int clean);

    /* info for NAND chips, defined in drivers/nand/nand.c */
    extern nand_info_t nand_info[CFG_MAX_NAND_DEVICE];
    ......
    #else /* ! CFG_ENV_OFFSET_REDUND */
    int saveenv(void)
    {
       ulong total;
       int ret = 0;
       puts ("Erasing Nand...");
        if (nand_legacy_erase(nand_dev_desc + 0,
               CFG_ENV_OFFSET, CFG_ENV_SIZE, 0))
        {
            return 1;
        }
        puts ("Writing to Nand... ");
        total = CFG_ENV_SIZE;
        ret = nand_legacy_rw(nand_dev_desc + 0, 0x00 | 0x02, CFG_ENV_OFFSET,
               CFG_ENV_SIZE, &total, (u_char*)env_ptr);
        if (ret || total != CFG_ENV_SIZE)
        {
```

```
            return 1;
        }
        puts ("done\n");
        return ret;
        ......
#else /* ! CFG_ENV_OFFSET_REDUND */
void env_relocate_spec (void)
{
#if !defined(ENV_IS_EMBEDDED)
        ulong total;
        int ret;
        total = CFG_ENV_SIZE;
        ret = nand_legacy_rw(nand_dev_desc + 0, 0x01 | 0x02, CFG_ENV_OFFSET,
              CFG_ENV_SIZE, &total, (u_char*)env_ptr);
        ......
```

5.3 实验内容——创建 Linux 内核和文件系统

1. 实验目的

通过移植 Linux 内核，熟悉嵌入式开发环境的搭建和 Linux 内核的编译配置。通过创建文件系统，熟练掌握使用 busybox 创建文件系统和如何创建文件系统映像文件。由于具体步骤在前面已经详细讲解过了，因此，相关部分请读者查阅本章前面内容。

2. 实验内容

首先在 Linux 环境下配置 minicom，使之能够正常显示串口的信息。然后再编译配置 Linux 2.6 内核，并下载到开发板。接下来，用 busybox 创建文件系统并完善所缺的内容。用 mkcramfs 创建 cramfs 映像文件并下载到开发板。在 Linux 内核和文件系统加载完了之后，在开发板上启动 Linux。

3. 实验步骤

（1）设置 minicom，按键“CTRL-A O”配置相应参数。
（2）连接开发板与主机，查看串口是否有正确输出。
（3）查看 Linux 内核顶层的 Makefile，确定相关参数是否正确。
（4）运行“make menuconfig”，进行相应配置。
（5）运行“make dep”。
（6）运行“make zImage”。
（7）将生成的内核映像通过 tftp 或串口下载到开发板中。
（8）用 busybox 创建文件系统。
（9）创建添加和修改所缺的目录和文件。
（10）在文件系统添加用户程序或者删除不需要的文件。

（11）用 mkcramfs 创建文件系统映像文件。

（12）将生成的文件系统映像通过 tftp 或串口下载到开发板中。

（13）在开发板上启动 Linux。

4．实验结果

开发板能够正确运行新生成的内核映像。

5.4 本章小结

本章详细讲解了嵌入式 Linux 开发环境的搭建，包括 minicom 和超级终端的配置，如何创建并下载映像文件到开发板，如何移植嵌入式 Linux 内核以及如何移植 U-Boot。这些都是操作性很强的内容，而且在嵌入式的开发中也是必不可少的一部分，因此希望读者切实掌握。

5.5 思考与练习

1．适当更改 Linux 内核配置，再进行编译下载查看结果。

2．配置 NFS 服务。

3. 深入研究一下 U-Boot 源码以及移植的具体步骤。

华清远见

推荐课程： 嵌入式学院-嵌入式 Linux 长期就业班

- 招生简章：http://www.embedu.org/courses/index.htm
- 课程内容：http://www.embedu.org/courses/course1.htm
- 项目实战：http://www.embedu.org/courses/project.htm
- 出版教材：http://www.embedu.org/courses/course3.htm
- 实验设备：http://www.embedu.org/courses/course5.htm

推荐课程： 华清远见-嵌入式 Linux 短期高端培训班

- 嵌入式 Linux 应用开发班：

http://www.farsight.com.cn/courses/TS-LinuxApp.htm

- 嵌入式 Linux 系统开发班：

  http://www.farsight.com.cn/courses/TS-LinuxEMB.htm

- 嵌入式 Linux 驱动开发班：

  http://www.farsight.com.cn/courses/TS-LinuxDriver.htm

华清远见