Linux/uClinux + MiniGUI：嵌入式系统开发原理、工具及过程

魏永明 钟书毅 潘为国 编著

北京飞漫软件技术有限公司

2005 年 4 月

内容简介

本书是北京飞漫软件技术有限公司发布的“MiniGUI 学习版（MiniGUI-STR）”产品的配套教材。本书以 Linux/uClinux+MiniGUI 的应用为例，讲述嵌入式系统开发中的基本原理、常用工具及一般过程。本教材的主要内容包括：

- 嵌入式 Linux/uClinux 及 MiniGUI 的介绍。将简单介绍嵌入式 Linux/uClinux 的特点及其与其它传统嵌入式操作系统之间的区别。还将介绍嵌入式开发中的常用术语及概念。
- 运行嵌入式 Linux/uClinux。以 SkyEye 及 Xcopilot 模拟器为例，讲述如何针对目标系统编译并运行嵌入式 Linux 及 uClinux 操作系统；主要阐述内核配置和编译、文件系统的建立、交叉编译，GNU 工具链的使用等。
- 编写并运行嵌入式应用程序。将讲述如何针对嵌入式 Linux/uClinux 移植简单的应用程序；还将阐述字节序、对齐、浮点/定点等和嵌入式开发相关的概念以及相关示例。
- 在 Linux/uClinux 上运行 MiniGUI。这部分将讲述 MiniGUI-STR 的配置选项、编译、安装以及在 PC、SkyEye 及 Xcopilot 模拟器上的运行。
- MiniGUI 和常见嵌入式开发板。介绍如何在常见嵌入式开发板上运行 MiniGUI-STR。这些嵌入式开发板有：博创 ARM3000、华恒 HH2410R3 等。
- 附录。MiniGUI 配置选项详解、MiniGUI 授权策略等。

“MiniGUI 学习版”产品光盘中还包括其它一些文档，可供读者参考：

- 《MiniGUI 技术白皮书》V1.6。该文档描述了 MiniGUI V1.6 的技术特性。

- 《MiniGUI 编程指南》V1.6。该指南详细讲述了利用 MiniGUI 开发嵌入式应用软件的基础知识、技术资料和开发技巧，内容涉及到 MiniGUI 编程的各个方面，包括消息循环和窗口过程、对话框和控件、图形接口等。还讲述了如何编写定制的 MiniGUI-Lite 服务器程序以及定制的 IAL 引擎。
- 《MiniGUI API Reference Manual》 V1.6。对 MiniGUI API 接口的详细描述。

需要注意的是，MiniGUI-STR 是 MiniGUI 的精简版本，只支持 Linux 和 uClinux 操作系统，此外还缺少许多高级特性。上述 MiniGUI 学习版产品光盘中的文档，描述的是完整的 MiniGUI 特性。

本书作为北京飞漫软件技术有限公司 MiniGUI 学习版产品的配套教材发布，亦可作为大专院校及各类专业培训班的教材。

版权声明

《Linux/uClinux + MiniGUI：嵌入式系统开发原理、工具及过程》版本 1.1。

目　录

前言

随着高端消费类电子产品（PDA、手机、便携式移动多媒体终端等）的广泛应用，原先仅在军工、工业控制等领域中使用的实时嵌入式操作系统，受到越来越多软硬件开发人员的关注。因为嵌入式产品本身是一种高度定制化的软硬件集成产品，单个操作系统无法满足各类嵌入式产品的多样化需求，因此，业界有许多各具特色的实时嵌入式操作系统产品可供选择。但开放源码的 Linux 以及其变种 uClinux 因为其开源、免授权费等因素而得到了许多嵌入式产品开发者的青睐。本书将主要介绍基于 Linux/uClinux 开发嵌入式产品的基本原理、开发过程以及所涉及的工具。

在嵌入式产品开发中，除操作系统之外，开发人员关注最多的另外一个系统软件组件就是图形支持系统。只要是面向人机交互的嵌入式产品，就涉及到文字或者图形的输出问题。以手机为例，彩信、WAP 以及即将来临的 3G 应用，都需要功能完备的图形用户界面（GUI）的支持，以便运行多媒体应用程序、J2ME 应用以及 3D 应用软件等等。因此，嵌入式产品的开发，除了硬件选择、操作系统选择及设备驱动程序的开发等工作外，工作量最大的开发任务应属于应用软件，尤其是图形相关应用软件的开发。

北京飞漫软件技术有限公司（*http://www.fmsoft.cn*）的核心产品 MiniGUI，是国内最成熟、应用面最广、用户最多的嵌入式系统软件产品。MiniGUI 目前已成为跨操作系统的嵌入式图形用户界面支持系统，广泛使用在手持终端、机顶盒、工业控制及仪表等行业和领域。华为、UT 斯达康、大唐、TCL 等国内知名企业正在使用 MiniGUI 开发其嵌入式产品，MiniGUI 产品也得到了嵌入式操作系统开发商和关键应用软件开发商的支持。MiniGUI 成为嵌入式图形领域的事实工业标准已为时不远。

通过在嵌入式 Linux/uClinux 上移植并运行 MiniGUI，我们可以了解

到嵌入式系统中应用软件开发的基本过程和原理，比如：如何完成交叉编译，如何在目标板上运行操作系统和应用程序，应用程序如何和硬件交互等等。因此，本书将以 MiniGUI 的移植和运行为线索，讲述基于嵌入式 Linux/uClinux 的嵌入式系统开发过程及原理。

然而，本书不打算介绍内核移植、驱动开发等方面的内容，如果读者对这方面的内容感兴趣，可参阅如下书籍：

- 《Linux 设备驱动程序》第二版，魏永明等译，中国电力出版社
- 《深入理解 Linux 内核》第二版，陈莉君等译，中国电力出版社

本书作者

本书由北京飞漫软件技术有限公司的高级研发工程师编写。第 1 章、第 3 章、第 4 章及附录由魏永明（飞漫软件首席架构师）编写；第 2 章由钟书毅（飞漫软件技术总监）编写；第 5 章由潘为国（飞漫软件项目经理）编写。全书由魏永明统稿及审校。本书著者具有多年的系统软件开发经验和丰富的技术著作编撰及培训经验。作者将把自己多年的开发和培训心得融入本书内容，相信能够帮助初次接触嵌入式 Linux/uClinux 的开发人员快速入门。

本书的读者对象

本书内容将以 MiniGUI 在嵌入式 Linux/uClinux 上的移植、简单应用程序的开发为线索，讲述嵌入式 Linux/uClinux 应用开发中的基本概念及相关工具，适合初次接触嵌入式开发的读者阅读。您只要具备在 Linux 平台上的 C 语言开发经验即可顺利阅读。

本书的内容组织

本书共分如下几个章节：

- 第 1 章：综述。将介绍常用嵌入式操作系统，嵌入式 Linux/uClinux、MiniGUI 的特点，以及嵌入式开发中的常用术语及概念。
- 第 2 章：运行嵌入式 Linux/uClinux。以 SkyEye 及 Xcopilot 模拟器为例，讲述如何针对目标系统编译并运行嵌入式 Linux 及 uClinux 操作系统；主要阐述内核配置和编译、文件系统的建立、交叉编译和 GNU 工具链的使用等。
- 第 3 章：编写并运行嵌入式应用程序。将讲述如何针对嵌入式 Linux/uClinux 移植简单的应用程序；还将阐述字节序、对齐、浮点/定点等和嵌入式开发相关的概念以及相关示例。
- 第 4 章：在 Linux/uClinux 上运行 MiniGUI。这部分将讲述 MiniGUI 的配置选项、编译、安装以及在 PC、SkyEye 及 Xcopilot 模拟器上的运行。通过本章学习，读者将了解到大型嵌入式应用软件的移植过程。
- 第 5 章：MiniGUI 和常见嵌入式 Linux/uClinux 开发板。介绍如何在常见嵌入式开发板上运行 MiniGUI。这些嵌入式开发板有：博创 ARM3000、华恒 HH2410R3 等。
- 附录。MiniGUI-STR 和 MiniGUI-VAR 的功能差异、MiniGUI-STR 配置选项详解、MiniGUI 授权策略等。

光盘内容

MiniGUI 学习版产品光盘中的内容组织为不同的目录，具体如下：

- *minigui/* 目录。包含 MiniGUI-STR 的四个软件包，分别为 *libminigui-str-1.6.2.tar.gz*、*minigui-res-str-1.6.tar.gz*、*mg-samples-str-1.6.2.tar.gz* 以及 *mde-str-1.6.2.tar.gz*，还包含 MiniGUI 的综合示例包 *mgdemo-1.6.0.tar.gz*、*qvfb-1.0.tar.gz*、MiniGUI 的 Linux PC 开发包及 Windows 开发包。
- *skyeye/* 目录。包含 SkyEye 模拟器 V0.8.6 的完整源代码包。
- *xcopilot/* 目录。包含 Xcopilot 模拟器源代码包。
- *armlinux/* 目录。包含 ARM Linux 的内核源代码包、SkyEye EP7312 模拟器相关资源，以及相应的交叉编译工具链。
- *uclinux/* 目录。包含 uClinux-dist 源代码包，以及针对 Xcopilot 模拟器的 m68k-elf 交叉编译工具链。

- *samples/* 目录。包含本书第 3 章中提及的 *cal* 示例程序包。
- *docs/* 目录。MiniGUI 相关的其他文档（PDF 格式）。

对运行 Linux PC 机的要求

为了正确安装和使用本书提及的工具、模拟器及程序，本书作者建议读者准备好一台运行 Linux 的 PC 机，具体要求如下：

- 奔腾 III 以上 CPU；
- 256MB 以上内存；
- 至少 15GB 空闲的硬盘空间。
- 使用 USB/PS2 接口的鼠标（PS2 鼠标协议）；
- VESA2 兼容的显示卡，确保能达到 1024x768 分辨率，16 位色。
- 选择 Red Hat Linux 9 发行版，安装时请选择所有的软件包（需为 */usr* 文件系统保留至少 5GB 的空间）。因为 RedHat Linux 的后续版本 Fedora 目前还不是非常稳定，而且可能会因为其中的开发工具版本太高而导致兼容性问题，因此，我们不建议读者安装 Fedora 发行版。
- 对硬盘合理分区，将 */usr*、*/usr/local*、*/home*、*/var*、*/opt* 等文件系统挂装在不同的分区上，确保为 */usr/local* 和 */opt* 文件系统划分至少各 3GB 的空间。

排版约定

下面是本书用到的一些字体和特殊段落的排版约定。

带阴影的等宽字体段落，用于命令行、源代码、程序输出等，比如：

```
#include <stdio.h>

int main (void)
{
    return 0;
}
```

有边框的楷体段落，用于提示、思考题等段落，比如：

【提示】在 Linux 终端仿真程序中，您可以用 <Ctrl+C> 组合键终止程序的运行。

黑体段落，用于要点段落，表示需引起读者注意的问题，比如：

【注意】北京飞漫软件技术有限公司遵循 GPL 条款发布 MiniGUI-STR 1.6.x 函数库；如果您要在商用产品中使用 MiniGUI-STR，则必须从飞漫软件购买商业授权。

另外，正文段落内的斜体（*italic*）通常用来表示文件、目录、程序和命令的名称、命令行选项、URL 等；斜黑体（***bold italic***），用于可变选项、关键词或者需要用户用实际值替换的文字。等宽字体（constant width），表示引用自界面等的文字。

本书所举例子，假定用户所使用的 Linux Shell 为 Bash Shell。有些命令需要以超级用户（root）身份执行，这时，用 root# 表示这种命令的提示符，普通用户命令用 user$ 提示符表示，比如：

```
user$ make
user$ su
root# make install
```

如果命令或者程序中的某些选项需要根据用户计算机的配置做适当的修改，则用 <> 表示，比如：

```
user$ cd <path to libminigui-str-1.6.2>
```

因为篇幅限制，如果一行的输出需要续行时，我们用续行符（\）表示：

```
CFLAGS=-m68000 -mid-shared-library -mshared-library-id=0 -Os -Wall -g -fomit-frame-pointer \
        -I$(UCLINUX_DIR)/lib/uClibc/include
```

建议和评论

本书的示例均已在 PC 机或者相应的模拟器、开发板上测试，然而仍然难免出现错误，而且本书的内容将处在一个持续更新的状态。我们将在如下网址提供本书的勘误、注解等内容：

http://www.minigui.com/product/cstr.shtml

我们还将在“嵌入式 Linux/MiniGUI 论坛”上专门开设“MiniGUI 学习版”产品论坛，为大家提供交流和学习的基地：

http://www.minigui.org/cgi-bin/lb5000/leoboard.cgi

如果您对本书及其后续版本有其他意见和建议，可联系北京飞漫软件技术有限公司市场部：

公司地址：北京海淀区苏州街 12 号西屋国际公寓 D 座 501 室
邮政编码：100080
电话：010-82873666/82873999
传真：010-82873666/82873999-88
电子邮件：*info@minigui.com*

1 综述

1.1 常用嵌入式操作系统

1.1.1 实时嵌入式操作系统的一般结构

为了达到硬实时特性，不同于一般的通用操作系统，实时嵌入式操作系统（real-time embedded operating system，RTOS），通常都运行在直接寻址模式下，该模式是相对于虚拟内存管理机制而言的。因为是直接寻址，实时嵌入式操作系统一般不提供类似 UNIX 操作系统那样具有独立地址空间的进程机制，也就是说，实时嵌入式操作系统中的所有任务都共享同一个地址空间。在此基础上，RTOS 的内核提供任务创建和调度、消息队列、信号量、互斥锁、事件标志等基本的任务同步及通讯机制。

仅仅有内核，我们还无法方便地开发嵌入式产品。通常，RTOS 产品提供 C 语言接口，内核也一般用 C 语言及汇编语言编写，加上符合 ISO C（ANSI C）标准的基本 C 函数库，我们就可以基于该 RTOS 开发自己的嵌入式产品了。

“内核接口 + ANSI C 库”的模式，是大多数嵌入式操作系统开发采用的编程模式。但是，这种模式还存在一些问题。如果我们要使用 ANIS C 库中的标准 I/O 接口，则需要操作系统提供文件系统及字符输出的支持；如果我们要使用 ANSI C 库中的内存管理函数（*malloc/free* 函数族），就需要提供针对具体硬件的堆管理方案及实现代码。因此，RTOS 通常都设计为模块化的软件系统，需要什么样的功能，可向 RTOS 产品厂商购买对应的模块来实现。比如 pSOS 操作系统，它由实时多任务核心 pSOS＋、

TCP/IP 协议堆栈 pNA＋、远程过程调用库 pRPC＋、文件系统管理 pHILE＋、ANSI C 标准库 pREPC＋、调试功能模块 pROBE＋、系统信息实时分析工具 pMONT＋ 等模块组成。

实际上，不同 RTOS 之间的区别，除了在任务管理上的核心区别之外，其他的主要区别就在外围模块上。图 1-1 给出了 RTOS 的一般软件结构。

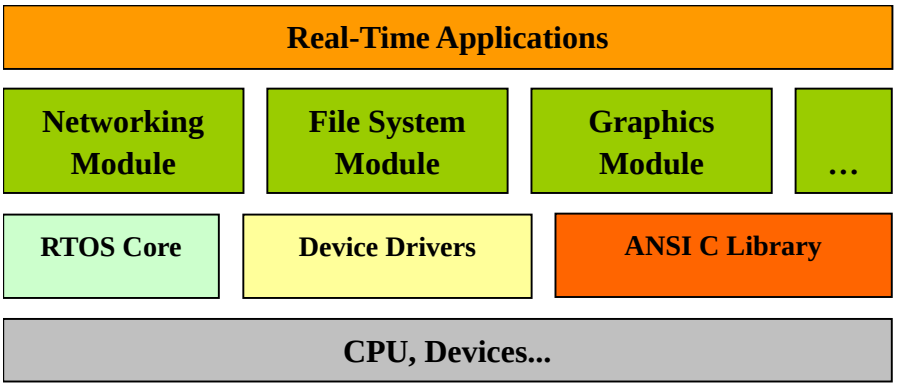

图 1-1　RTOS 的一般软件结构

1.1.2 常用（实时）嵌入式操作系统

得到业界更多关注的 RTOS 可划分为如下几个种类：

- 传统实时嵌入式操作系统。主要包括：VxWorks[1]、pSOS[2]、Nucleus[3]、WinCE[4] 等。这几个 RTOS 都已经拥有大量的用户群；VxWorks 和 pSOS 的用户主要集中在军工、工业控制及电信领域，Nucleus 和 WinCE 在消费类产品中应用较为广泛。
- 开放源码的嵌入式操作系统。典型代表有 Linux/uClinux[5]、eCos[6]。因为传统实时嵌入式操作系统价格比较昂贵，因此，许多用户开始使用免授权费的 Linux/uClinux 等操作系统开发自己的嵌入式产品。其中最为突出的代表是 Linux 操作系统。但是，因为 Linux 操作系统从本质上属于通用操作系统，缺少强实时支持，因此，嵌入式 Linux 在某些不需要强实时性的嵌入式产品中更加适用，典型的产品有智能手机（这类产品中的实时性主要通过专用硬件芯片保

[1] VxWorks 是美国风河（WindRiver）公司（http://www.windviver.com）产品。
[2] pSOS 原是美国加州集成系统公司（Integrated Systems Inc）产品，后来被风河收购。
[3] http://www.acceleratedtechnology.com/。
[4] 微软公司的嵌入式操作系统产品。
[5] Linux：http://www.kernel.org；uClinux：http://www.uclinux.org。
[6] http://sources.redhat.com/ecos/。

证）、查询终端等。这类产品的功能比较丰富，采用具有通用操作系统特征的操作系统能简化软件开发过程。uClinux 是 Linux 的一个变种，主要运行在没有内存管理单元（MMU）的 CPU 架构上。因为没有内存管理单元，uClinux 无法实现现代操作系统能够提供的进程地址空间保护等高级特性，但它最大程度地保留了 Linux 的系统调用功能，而且资源消耗低，因此在一些中低端的 32 位嵌入式产品中得到了应用。eCos 也是一种开源、免授权费的 RTOS 产品。和 Linux/uClinux 相比，它更类似传统的实时嵌入式操作系统，而且提供了丰富的外围模块，比如文件系统、TCP/IP 接口模块、POSIX 兼容接口模块等。根据笔者的实际应用经验，eCos 操作系统在一定程度上可以用来替代传统实时嵌入式操作系统。

- 新型实时嵌入式操作系统。主要包括 uC/OS-II[7]、ThreadX[8] 等操作系统。uC/OS-II 和 ThreadX 操作系统是近几年出现的实时嵌入式操作系统。它们的出现，填补了 RTOS 操作系统市场的中低端市场，给广大嵌入式产品开发者提供了性价比较高的选择。uC/OS-II 可以支持 16 位和 32 位 CPU，它最大的特点是内核非常小。虽然它的任务管理和调度部分比较简单，但仍然在工业控制等领域中得到了一些应用。另外，因为其代码简单易懂，因此也非常适合嵌入式操作系统的教学。ThreadX 操作系统类似传统的 RTOS，适合在需要实时处理的嵌入式产品中使用。另外，ThreadX 上还有面向 VxWorks 和 pSOS 的兼容软件包，从而可以方便地移植 VxWorks 和 pSOS 应用软件。
- 国产实时嵌入式操作系统。近几年，国内也有厂商开始提供自主研发的实时嵌入式操作系统，典型的有 Hopen 和 Delta 操作系统。这些产品已经在消费类电子产品和军工领域中得到了一些应用。

基于传统嵌入式操作系统的开发相对复杂一些，所使用的工具比较专业，而且往往需要相应的硬件才能验证和测试应用软件。使用通用操作系统 Linux 来开发嵌入式产品，大大降低了嵌入式产品开发的难度和门槛。我们不需要购买昂贵的开发工具和嵌入式操作系统，只要我们有相应的目标硬件电路板，即可轻松开始自己的嵌入式产品开发。甚至，如果仅仅想学习和掌握一些基本的工具和开发过程，使用软件模拟器就可以做到。本书的主要目标，就是要揭开嵌入式开发的神秘面纱，通过使用免费的软件

[7] http://www.ucos-ii.com/
[8] http://www.expresslogic.com/

模拟器、工具及中间件软件，我们不需要投入任何的硬件成本，即可了解嵌入式开发的基本概念，掌握基本的嵌入式开发工具和一般过程。

然而，我们需要清楚的是，嵌入式 Linux/uClinux 的方案并不是万能的。由于 Linux/uClinux 在实时性等方面的缺陷，使得它还不能用来开发对实时性、性能、成本等因素非常敏感的嵌入式产品。下面我们简单介绍一下嵌入式 Linux/uClinux 操作系统及其和传统嵌入式操作系统之间的区别。

1.2 嵌入式 Linux/uClinux

嵌入式 Linux 操作系统在本质上和普通的 Linux 操作系统是一样的。我们通常接触到的 Linux 操作系统运行在 PC 平台上，我们把它当成办公电脑、开发工作站，甚至服务器使用。运行在各种嵌入式硬件平台上的嵌入式 Linux，和运行在 PC 上的 Linux 操作系统，本质上一模一样，只是前者的运行环境没有后者好。比如 CPU 运算能力不强、存储空间不大、和用户交互的方式不同等等。因此，我们可以将嵌入式 Linux 操作系统看成一种定制后的 Linux 操作系统。

我们知道，Linux 操作系统是一种高级的 32 位操作系统，具有现代操作系统的许多高级特性，比如多进程支持、虚拟内存等等。这些特性的获得，通常需要得到硬件尤其是 CPU 的支持。比如虚拟内存，它能够让系统中运行的进程具有独立的、互不干扰的地址空间，而这需要内存管理单元（MMU）的帮助才能做到。但是，现今世界上所使用的大部分嵌入式 CPU，都没有内存管理单元。为了在基于这种 CPU 的嵌入式系统上运行 Linux，就需要对普通 Linux 做进一步的定制，这就是 uClinux 的来由。uClinux 这个名称的含义，就是针对“微控制器”的 Linux 操作系统，它主要运行在没有内存管理单元的 32 位 CPU（即“微控制器”）上。

现在，嵌入式 Linux/uClinux 已经能够支持各种各样 32 位的微处理器了，这得益于开放源码组织的大力支持和贡献。

【提示】嵌入式 Linux/uClinux 所支持的微处理器有：32 位 x86（386 或更高）、Compaq 的 Alpha AXP、Sun 的 SPARC 和 UltraSPARC、Motorola 的 68000、PowerPC、PowerPC64、ARM、Hitachi SuperH、IBM S/390、MIPS、HP PA-RISC、Intel IA-64、DEC VAX、AMD x86-64、AXIS CRIS 以及 Renesas M32R。

【提示】ARM 是各类嵌入式系统使用最多的架构。ARM 这种架构由英国 ARM 公司设计，然而 ARM 公司并不直接生产处理器，它的主要业务就是向各类处理器生产厂商提供 ARM 架构的授权，并提供其他开发工具给用户。市面上许多嵌入式处理器都采用的是 ARM 核心，著名的有三星公司的 S3C44B0、S3C2410、S3C4410，CirrusLogic 公司的 EP7312，Atmel 公司的 AT91 系列等。

因为嵌入式 Linux/uClinux 派生自通用操作系统，因此，基于嵌入式 Linux/uClinux 的开发和普通 PC 上的应用程序开发类似。而传统的嵌入式操作系统，如果没有特定工具的帮助，我们几乎无法运行这些操作系统以及其上的应用软件。另外，基于传统嵌入式操作系统的开发和基于嵌入式 Linux/uClinux 的开发还在如下一些方面有着重要的不同：

- 如 1.1 节所介绍的，传统实时嵌入式操作系统出于实时性考虑，通常运行在直接寻址模式下。要完成的应用程序工作通常由一个个的操作系统任务（task）来完成，这些任务和操作系统内核一起共享同一个地址空间。在开发自己的应用时，我们通常将自己的程序和操作系统内核、C 函数库编译并链接成一个大的程序，然后下载到目标硬件上运行。然而，基于 Linux/uClinux 的嵌入式开发更类似传统桌面上的应用软件开发过程。操作系统内核的开发和应用程序的开发可以明确分开；大部分嵌入式操作系统不提供文件系统的支持，而在嵌入式 Linux/uClinux 系统中，可通过文件系统组织数据和程序。

- 在传统嵌入式操作系统上的开发，内核通常表现为一个函数库，应用程序调用这些函数库中的函数，配合基本的 C 函数库提供的功能，来完成应用程序所能完成的各种工作。然而，传统嵌入式操作系统上的 C 函数库功能非常有限，甚至连基本的标准 I/O 机制都没有，这样，基于传统嵌入式操作系统的开发难度较大。相反的是，

嵌入式 Linux/uClinux 则拥有和 PC 一样或者基本一样的函数库可供应用程序开发人员使用。这样，我们可以将大量现成的应用程序移植到嵌入式 Linux/uClinux 上，有时，所涉及的工作仅仅是交叉编译而已。

- 由于嵌入式 Linux/uClinux 的运行环境和普通 Linux 基本相同，所以，大部分应用程序可在 PC 上完成开发和测试，然后再到具体的硬件开发板上进行最终测试。而基于传统嵌入式操作系统的开发，几乎所有应用程序的调试均要在硬件板上开展，加上底层设施的缺失，调试手段的单一，使得嵌入式应用的开发异常艰难。

当然，如果在硬件板上还没有运行起嵌入式 Linux/uClinux 操作系统的内核，则关于内核的开发（移植、驱动程序开发等）难度和传统嵌入式操作系统的开发难度基本上是一样的。

1.3 MiniGUI 简介

因为本书主要讲述基于嵌入式 Linux/uClinux 的应用软件开发，但如果我们所开发的应用软件只能输出一些简单的文字信息，则会显得非常枯燥乏味。我们在 1.1 中也提到，随着微处理器能力的提高，越来越多的嵌入式产品提供了良好的人机交互接口，这样，大部分嵌入式应用软件的开发都会涉及到图形输出问题。MiniGUI 就是一个专门面向嵌入式系统的图形用户界面支持系统，在软件体系上，它介于内核和应用程序之间，因此，亦可称为一种软件“中间件”产品。本书在讲解嵌入式 Linux/uClinux 的开发过程及工具时，将主要以 MiniGUI 为例来进行说明。

为了让读者对嵌入式图形技术及产品的现状有个大致了解，本节将为读者介绍 MiniGUI 一些基本情况。

1.3.1 嵌入式产品开发中常用的图形解决方案

在嵌入式产品的开发过程中，软件开发人员通常采取如下几种方法来解决产品的图形需求：

1) 编写针对特定图形输出设备的接口，自行开发图形相关的功能函数。一些使用单色 LCD 输出屏的低端嵌入式产品，比如电子词典，

因为图形功能简单，经常使用这种方案解决图形问题。然而，利用这种手段编写的程序，无法将显示逻辑和数据处理逻辑划分开来，从而导致程序结构不好，不便于调试，并导致大量的代码重复。这种方案的缺点很明显，即可移植性差，维护成本高。

2) 购买针对特定嵌入式操作系统的图形中间件软件包。一些嵌入式操作系统厂商，也为自己的操作系统专门开发了对应的图形用户界面（GUI）中间件产品。比如 uC/OS-II 上的 uC/GUI、Nucleus 上的 GRAFIX 包、VxWorks 上的 WindML 包等等。这种方案为嵌入式产品开发提供了直接可用的方案，并且能够和原有操作系统良好配合；但缺点是这类软件包的功能通常比较简单，且价格高昂。另外，基于这些软件包开发的 GUI 应用软件不具备跨操作系统的可移植性；也就是说，基于这些软件包开发的应用软件越复杂，将来替换操作系统的可能性就越小。

3) 采用开放源码的嵌入式 GUI 支持系统。随着嵌入式 Linux 操作系统的应用，开源社区也在不断为嵌入式系统提供不同的开放源码嵌入式图形解决方案，比如 MicroWindows、OpenGUI，以及新近出现的 picoGUI 等。这些开放源码的嵌入式 GUI 软件，为我们提供免授权费的解决方案。然而，因为缺少商业公司的支持，这些软件一般存在较多的软件缺陷，加上缺乏有担保的技术支持，因此，存在着很大的开发风险。

4) 使用由独立软件开发商提供的嵌入式 GUI 产品。这类产品有挪威 TrollTech 公司的 Qt/Embedded 以及由北京飞漫软件技术有限公司开发的 MiniGUI 等。这两种产品都是开源（遵循 GNU 的 GPL 条款发布）的嵌入式 GUI 软件产品，但均采用双授权模式，即针对商业使用收取软件许可费用。Qt/Embedded 属于高端产品，只支持嵌入式 Linux 操作系统（不包括 uClinux 操作系统），需要 16MB 以上的静态存储空间及 64MB 以上的动态存储空间。MiniGUI 属于中低端产品，其跨操作系统特性，以及适合嵌入式产品的小巧、高效的特点，使它受到了更多嵌入式产品开发商的青睐。

由北京飞漫软件技术有限公司[9]开发的 MiniGUI[10]，是国内为数不多的几大国际知名自由软件之一。MiniGUI 是面向实时嵌入式系统的轻量级图形用户界面支持系统，1999 年初遵循 GPL 条款发布第一个版本以来，

[9] 北京飞漫软件技术有限公司：http://www.fmsoft.cn
[10] http://www.minigui.com

已广泛应用于手持信息终端、机顶盒、工业控制系统及工业仪表、便携式多媒体播放机、查询终端等产品和领域。目前，MiniGUI 已成为跨操作系统的图形用户界面支持系统，可在 Linux/uClinux、eCos、uC/OS-II、VxWorks、pSOS、ThreadX 等操作系统以及 Win32 平台上运行；已验证的硬件平台包括 Intel x86、ARM（ARM7/AMR9/StrongARM/xScale）、PowerPC、MIPS、M68K（DragonBall/ColdFire）等等。

1.3.2 MiniGUI 的起源和发展

MiniGUI 的开发起始于 1998 年底，到现在已历经六年多的时间。任何人都没有想到 MiniGUI 会发展为一个跨操作系统的嵌入式图形支持系统，因为最初的 MiniGUI 仅仅是为了在 Linux 上有一个能显示中文的东西而开发。所幸的是，MiniGUI 从诞生开始，就不断得到实际项目的应用，同时新的项目也提出了更多技术需求，于是 MiniGUI 就逐步发展壮大，成为了一个跨操作系统的嵌入式图形支持系统。

1998 年 12 月，飞漫软件创始人魏永明开始开发 MiniGUI 并在一个数控系统中得到应用。2000 年 3 月，联想 HappyLinux 1.0 发行版采用 MiniGUI 开发了其安装程序。这时，MiniGUI 已形成了一个较为完整的嵌入式图形用户界面支持系统。2000 年 4 月到 2002 年 9 月，MiniGUI 作为中国为数不多的几个自由软件项目之一，继续以开源社区的形式进行开发。

2002 年 9 月，MiniGUI 的主要开发者成立了北京飞漫软件技术有限公司，尝试自由软件的商业化运作模式，并于 2003 年 5 月发布了 MiniGUI 1.2.6 版本；于 2003 年 9 月发布了 MiniGUI 1.3.0 版本。

2003 年 10 月，我们将 MiniGUI 移植到了 uClinux 和 eCos 操作系统。至此，MiniGUI 成为一个跨平台的嵌入式图形用户界面支持系统。

目前，MiniGUI 已发展到 1.6.2 版本，支持 Linux/uClinux、eCos、uC/OS-II、VxWorks、pSOS、ThreadX 等操作系统，也可以在 Win32 平台上运行。飞漫软件基于自由软件的商业化模式也获得了初步的成功。

1.3.3 基于 MiniGUI 的嵌入式系统软件结构

为什么 MiniGUI 能够在如此众多的嵌入式操作系统上运行？这是因为

MiniGUI 具有良好的软件架构，通过抽象层将 MiniGUI 上层和底层操作系统隔离开来。如图 1-2 所示，基于 MiniGUI 的应用程序一般通过 ANSI C 库以及 MiniGUI 自身提供的 API 来实现自己的功能；MiniGUI 中的“可移植层”可将特定操作系统及底层硬件的细节隐藏起来，而上层应用程序则无需关心底层的硬件平台输出和输入设备。

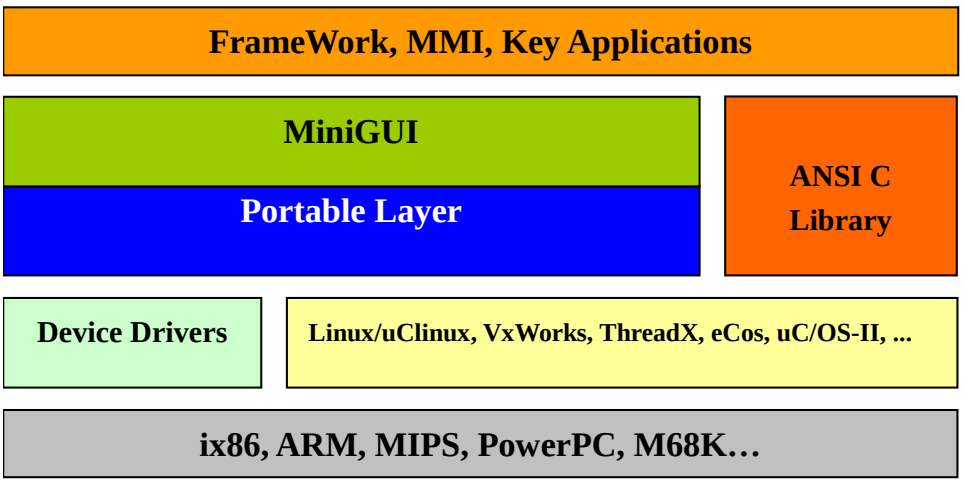

图 1-2 MiniGUI 和嵌入式操作系统的关系

另外，MiniGUI 特有的运行模式概念，也为跨操作系统的支持提供了便利。

1.3.4 MiniGUI 运行模式

和 Linux 这样的类 UNIX 操作系统相比，一般意义上的传统嵌入式操作系统具有一些特殊性。举例而言，诸如 uClinux、uC/OS-II、eCos、VxWorks 等操作系统，通常运行在没有 MMU（内存管理单元，用于提供虚拟内存支持）的 CPU 上；这时，往往就没有进程的概念，而只有线程或者任务的概念，这样，GUI 系统的运行环境也就大相径庭。因此，为了适合不同的操作系统环境，我们可将 MiniGUI 配置成三种运行模式：

- MiniGUI-Threads。运行在 MiniGUI-Threads 上的程序可以在不同的线程中建立多个窗口，但所有的窗口在一个进程或者地址空间中运行。这种运行模式非常适合于大多数传统意义上的嵌入式操作系统，比如 uC/OS-II、eCos、VxWorks、pSOS 等等。当然，在 Linux 和 uClinux 上，MiniGUI 也能以 MiniGUI-Threads 的模式运行。

- MiniGUI-Lite。和 MiniGUI-Threads 相反，MiniGUI-Lite 上的每

个程序是单独的进程，每个进程也可以建立多个窗口。MiniGUI-Lite 适合于具有完整 UNIX 特性的嵌入式操作系统，比如嵌入式 Linux。

- MiniGUI-Standalone。这种运行模式下，MiniGUI 可以以独立进程的方式运行，既不需要多线程也不需要多进程的支持，这种运行模式适合功能单一的应用场合。比如在一些使用 uClinux 的嵌入式产品中，因为各种原因而缺少线程库支持，或者线程库存在缺陷，这时，就可以使用 MiniGUI-Standalone 来开发应用软件。

一般而言，MiniGUI-Standalone 模式的适应面最广，可以支持几乎所有的操作系统，甚至包括类似 DOS 这样的操作系统；MiniGUI-Threads 模式的适用面次之，可运行在支持多任务的实时嵌入式操作系统，或者具备完整 UNIX 特性的普通操作系统；MiniGUI-Lite 模式的适用面较小，它仅适合于具备完整 UNIX 特性的通用操作系统。

但不论采用哪种运行模式，MiniGUI 为上层应用软件提供了最大程度上的一致性；只有少数几个涉及初始化的接口在不同运行模式上有所不同。

1.4 基于 Linux/uClinux 的嵌入式开发过程及相关概念

1.4.1 一般开发过程

基于 Linux/uClinux 的嵌入式产品的开发，在确定好产品需求之后，通常遵循如下的开发过程：

第1步. 确定硬件设备。包括处理器、存储设备、显示屏、触摸屏以及其他外设，比如网卡、声卡等。通常，我们可以从一些方案提供商那里购买得到比较符合自己需求的硬件开发板（或者硬件参考设计板），有了这类硬件板，我们就可以根据自己的需求进一步定制，从而缩短开发周期。

第2步. 移植操作系统并开发设备驱动程序。通常，针对一款新的硬件开发板移植 Linux/uClinux 操作系统是技术难度较高的工作。如果我们购买由方案供应商提供的硬件开发板，则内核和大部分设备驱

动程序是现成的，我们只需要开发定制设备的驱动程序即可。

第3步. 编写自己的应用软件。利用 Linux/uClinux 开发嵌入式产品有个最大的好处，就是我们可以在 PC 环境上完成绝大多数的应用软件开发和调试工作。比如，如果您的产品需要从某个特定的网站下载一幅图片然后显示在屏幕上，则我们可以在 PC 上实现该软件，之后移植到硬件板上即可。因为该程序涉及的主要接口：网络、显示卡等，不管在 PC 上，还是在硬件开发板上，均保持一致的接口。于是，只要在 PC 上运行正确，该程序就能够在嵌入式硬件开发板上正确运行。

第4步. 将应用软件移植到硬件板子上并进行测试及调试。这个过程主要就是将应用软件以及应用软件所使用的函数库等，通过交叉编译器编译成目标硬件板上的程序，然后和共享库、常用工具程序等一起，形成一个完整的文件系统映像，之后下载到硬件板上，并在硬件板上进行应用软件的测试和调试。需要注意的是，因为嵌入式系统上的资源毕竟有限，比如内存的可获得性、存储空间的可用性等均会影响程序的正常运行，因此，我们需要在实际的硬件板上运行应用程序以便测试整个系统。

第5步. 通常，上述步骤符合一种迭代关系。进行到第 4 步时，也许我们会发现应用程序本身的一些问题，也许会发现驱动程序存在问题。这时，我们就要回到第 2 步、第 3 步修正错误并开展第二次迭代。当然，也有可能会发现我们最初选择的硬件性能和能力有缺陷，从而会导致从第 1 步重新来过。

第6步. 经过严格测试之后，整个硬件和软件系统就可以交给产品设计部门设计外观和模具并最终到生产线上生产了。

以上就是基于 Linux/uClinux 的嵌入式产品开发的一般过程。接下来我们重点介绍上述过程中的一些重要概念。

1.4.2 交叉编译

交叉编译是嵌入式开发中最常见的概念。交叉编译是相对于通常桌面上的开发而言的。在我们开发 Windows 程序时，我们通常在 Windows 平台上运行一个集成开发环境，编写代码，然后利用集成开发环境所带的编

译器将代码编译并连接成 Windows 平台上的程序，之后还可以在集成开发环境中运行并调试该程序。这种开发方式对 PC 而言是非常方便的，因为 PC 平台的软硬件配置已经足以完成编译程序这类计算量非常大的工作。然而，对嵌入式系统来讲，由于其硬件能力所限，我们不可能在嵌入式系统上安装编译器和连接器，然后在嵌入式系统上编写代码、编译并连接程序。于是，就出现了交叉编译器（cross-compiler）。交叉编译器通常运行在某台功能足够强大的宿主机（host）上，可用来编译某个源程序，然后生成针对特定目标平台（target）的代码。

通常来讲，C 语言的交叉编译环境主要由如下几个部分组成：

- 交叉编译器及相关二进制工具（连接器、归档工具、符号剥离器等）。
- 针对目标平台的 C 头文件。不同的目标平台具有自身特有的函数库及对应的头文件，这样，在宿主机上，我们就需要保留一份针对该目标平台的头文件，以便用来交叉编译 C 程序。
- 针对目标平台的 C 函数库。和头文件一样，在宿主机上，我们也要保留一份针对特定目标平台的函数库，以便连接生成最终的目标程序。

构成交叉编译环境的以上各部分，通常又称为交叉“工具链（toolchain）”。我们在开发嵌入式系统之前，通常要做的第一件事情就是安装交叉工具链。

我们举例说明普通编译和交叉编译的不同。同样是下面的 helloworld 程序：

```
#include <stdio.h>

int main (void)
{
    printf (“Hello, world\n”);
    return 0;
}
```

当我们在一台 Linux PC 机上进行正常编译时，我们使用下面的命令：

```
user$ gcc -o helloworld helloworld.c
```

其中，gcc 就是普通的本机 C 编译器。该编译器在遇到 *#include*

<stdio.h> 这一预编译指令时，将在本机的 /usr/include 或者 /usr/local/include 目录下寻找 stdio.h 这个文件。当 gcc 正确编译上述代码，要连接生成 helloworld 程序时，gcc 就会调用本机连接器（ld），并连接本机 /lib 目录下的默认 C 函数库 libc.so。

当我们在一台 Linux PC 上针对一款 ARM 硬件平台交叉编译该程序时，我们使用下面的命令：

```
user$ arm-linux-gcc –o helloworld helloworld.c
```

其中，arm-linux-gcc 是针对 ARM 硬件平台的交叉编译器，该交叉编译器是可运行在 PC 平台上的本机程序。当它遇到 #include <stdio.h> 这一预编译指令时，将在交叉编译环境中寻找 stdio.h 这个头文件，而在连接生成目标平台的可执行程序时，也会在交叉编译环境中寻找 libc.so 这个函数库。这里的交叉编译环境指本机上的一个目录树结构，这个目录树通常和本机的开发环境类似，但位于不同的文件系统位置上，以避免和本机开发环境相混淆。以常用的 arm 的开发工具链为例，交叉编译器及二进制工具会被安装到 /usr/local/arm-linux/bin/ 目录下，头文件会被安装到 /usr/local/arm-linux/arm-linux/include/ 目录下，而函数库会被安装到 /usr/local/arm-linux/arm-linux/lib/ 目录下。在安装了交叉工具链之后，为了正确运行上述命令，我们通常还要如下设置 PATH 环境变量：

```
user$ export PATH=:/usr/local/arm-linux/bin:$PATH
```

这样，Shell 命令解释器就能够找到 arm-linux-gcc 的正确路径并执行上述命令。一般情况下，我们不需要特别指出头文件和函数库所在位置，交叉编译器会在默认的位置寻找头文件及库函数。然而，在某些情况下，尤其是在进行 uClinux 开发时，我们可能会希望使用不同的库函数（比如 uClibc），这时，就需要通过 –I、-L 等选项特别指定头文件和库函数的位置。这将在后面的章节中详细讨论。

1.4.3 内核移植

我们通常所讲的内核移植，在嵌入式 Linux/uClinux 开发中指的是在特定硬件平台上运行 Linux 的内核。内核的移植涉及到如下工作：

- 编写针对特定处理器的代码。内核中关于任务调度、中断处理等的代码要根据不同的处理器类型重新编写。

- 编写针对特定硬件平台的引导和初始化代码。基于同样的处理器，由于硬件系统设计上的不同，或者操作系统装载器的不同，需要实现有针对性的引导和初始化代码。

- 编写针对特定外设的设备驱动程序代码。

上述工作，技术难度从高到低排列（前两份工作要求开发者熟悉处理器的架构及汇编语言），工作量从低到高排列。所幸的是，基于嵌入式 Linux 和 uClinux 的内核移植，通常会由自由软件社区完成前两项工作，留给我们的工作通常只有第三份工作。这也是开放源码带给我们的好处。

1.4.4 驱动开发

如上所述，在嵌入式 Linux/uClinux 内核开发中，驱动程序的开发工作工作量最大，但所幸大多数驱动程序都可以参照已有的驱动程序来设计，并且绝大多数情况下只需用 C 语言开发，而不需要使用汇编语言。驱动程序的开发要求开发者熟悉 UNIX 系统的结构和内核的驱动接口，为上层提供良好的编程接口。而且驱动程序的开发不像应用程序那样，可以在 PC 上完成大部分的编码及测试，相反，设备驱动程序的开发始终离不开对应的设备，加上驱动程序运行在内核中，而目前又缺少有效的内核调试方法，因此，驱动程序的开发相对复杂且繁复。

1.4.5 应用软件开发及调试

前面已经提到，嵌入式 Linux/uClinux 上的应用软件开发调试相对传统嵌入式操作系统的应用软件开发要容易得多，只要内核、设备驱动程序以及函数库没有重大缺陷，我们就可以按照如下步骤开发嵌入式应用程序：

第1步. 在 Linux PC 环境中，确保安装有完整的开发工具包及交叉编译工具链。

第2步. 如果应用程序使用到的设备及其访问接口在目标板和 PC 上是一致的，则可以直接在 PC 上编写并调试。

第3步. 如果应用程序使用到的设备接口只能在目标板上验证，则可以

在 PC 上编写一个假的驱动程序或者接口来模拟实际的设备，以便完成应用软件其他部分的开发，然后等目标板上的设备驱动程序开发完毕之后，到目标板上运行并调试。

根据我们的经验，基于 Linux/uClinux 的嵌入式系统开发中，大部分的应用软件可在 PC 上开发。然而，因为 uClinux 操作系统的特殊性，我们在开发针对 uClinux 操作系统的应用软件时，需要注意如下问题：

- uClinux 内核不提供 *fork* 系统调用。因此，许多基于 *fork* 系统调用特性的功能，比如匿名管道，无法在 uClinux 上实现。也就是说，在 uClinux 系统上，无法运行下面这种命令：

```
user$ ls | more
```

- uClinux 系统可使用 uC-libc 和 uClibc 两种函数库，后者比前者完善，但两者均不是非常完善，某些函数接口存在缺陷。
- uClinux 在某些处理器架构上，信号的实现存在重大缺陷，会导致程序随机意外退出，因此，应在应用程序中避免使用 *setitimer* 等函数（该函数用来设置间隔定时器，当定时器到期时，会产生 *SIGALARM* 信号）。
- 因为信号处理上的问题，uClibc 中实现的 POSIX Threads 线程接口也存在问题，当系统负荷较重时，会发生互斥锁失效的问题。

在本书第 3 章“编写并运行嵌入式应用程序”中，我们将详细讲述嵌入式 Linux/uClinux 应用软件开发中需要注意的问题及相关工具的使用。

1.5 小结

本章概要介绍了嵌入式操作系统的种类，嵌入式 Linux/uClinux 和传统嵌入式操作系统之间的区别，还介绍了 MiniGUI 的起源和基本特色。作为重点，通过本章的学习，读者应该了解嵌入式系统开发的基本过程，理解基于嵌入式 Linux/uClinux 的开发过程涉及的一些基本概念。

本书第 2 章将以 SkyEye 及 Xcopilot 模拟器为例，讲述如何针对目标

系统编译并运行嵌入式 Linux 及 uClinux 操作系统；主要阐述 GNU 交叉编译工具链的安装及使用、内核的编译及基本文件系统的准备等。

2 运行嵌入式 Linux/uClinux

在前一章中我们概要地介绍了嵌入式 Linux/uClinux 开发的特点和过程，本章将结合理论和实际，深入地为读者剖析嵌入式 Linux/uClinux 开发中涉及到的重要概念、工具和过程。主要内容包括嵌入式 Linux 体系结构、交叉编译环境、根文件系统的创建等，最后用 SkyEye EP7312 模拟器和 Xcopilot 模拟器为例，说明了建立一个嵌入式 Linux/uClinux 系统的具体过程。

2.1 嵌入式 Linux 的体系结构

Linux 最初是面向桌面应用领域而设计的通用型操作系统，具有和其它 UNIX 类操作系统相似的设计，包括采用虚拟内存管理机制、支持多进程和多用户管理等。当 Linux 在桌面和服务器领域取得了成功之后，大家开始在嵌入式领域采用 Linux 来构建系统和开发产品。普通嵌入式 Linux 系统的架构与桌面 Linux 系统的架构在本质上并没有太大的差异，只是在系统的大小、组成与使用方式上有所不同。

嵌入式 Linux 系统是指基于 Linux 内核、相关工具和应用程序构建的嵌入式系统。这里所说的系统是指运行时系统，而不是开发系统。开源和公开的嵌入式 Linux 开发系统通常可以由互联网上获取，一般包括 Linux 内核、交叉编译工具链、相关的函数库和文件系统。商业 Linux 公司提供各种成熟的嵌入式 Linux 发行版。一个发行版通常包括：用于嵌入式 Linux 系统开发的开发工具集和各种适用该嵌入式系统的应用程序。目前比较有名的嵌入式 Linux 发行版是美国 MontaVista 公司提供的 HardHat Linux。

与一般实时嵌入式操作系统的运行模式不同，嵌入式 Linux 中的用户

应用程序是作为独立于内核的单独进程来运行的。因此，从应用开发的角度来看，嵌入式 Linux 开发的方式更接近普通的桌面应用开发，而不是典型的嵌入式系统应用开发。我们可以在桌面 Linux 操作系统上开发、编译和运行我们的应用程序，在基本确保正确无误之后就可以交叉编译后下载到设备上的嵌入式 Linux 系统中运行。所以，整个开发流程非常方便。

2.1.1 Linux 系统的构成

基于传统 RTOS 的嵌入式系统一般由操作系统内核、中间件模块和应用软件等部分构成，这些程序都是结合在一起作为一个单一的系统映像（image）烧写到系统的存储介质如 Flash 之上的；某些嵌入式系统还包括了文件系统支持，这时系统所使用的图片和数据等资源文件一般存放在文件系统中。

这种基于传统 RTOS 的嵌入式系统内核与应用之间关系非常紧密，整个系统的更新和升级非常困难，通常需要更换整个系统。

基于嵌入式 Linux 的系统在系统构成上与传统 RTOS 有较大的差别。和桌面 Linux 系统类似，嵌入式 Linux 系统通常由内核、函数库、工具程序和应用程序组成，而函数库、工具程序、配置文件和应用程序等通常存放在根文件系统之中。

图 2-1 描述了一个普通嵌入式 Linux 系统的组成，中间的部分为内核层。可以看到，嵌入式 Linux 系统的组成与桌面 Linux 并无本质上的不同。

图 2-1 嵌入式 Linux 系统的组成

2.1.2 内核

Linux这个名称严格意义上指的是以Linus Torvalds为核心的开发团队开发的操作系统内核。通常情况下，Linux 这个词也可以用来表示一个Linux 操作系统，或者一个 Linux 操作系统发行版。

我们需要注意的是，并没有官方（由 *www.kernel.org* 发布）的所谓“ 嵌入式Linux内核 ”。一般情况下我们可以使用官方发布的某个主流内核版本为基础来构建自己的嵌入式Linux系统。

有些情况下我们可能需要使用某个由第三方修改和发布的内核，这种内核通常是针对某款硬件平台而定制，或者是为了支持某种特殊的应用类型。不同的嵌入式发行版所提供的内核通常提供了主流内核中所没有的优化和针对内核调试工具的补丁。

总的来说，嵌入式 Linux 系统中所使用的内核和桌面及服务器系统中所使用的内核的主要区别在于它的配置选项。

作为一个操作系统内核而言，Linux 并不是一个可抢占式的内核，它基于优先级和时间片轮转的调度方法；也就是说，普通的 Linux 内核并不是一个实时操作系统。

Linux 内核由几个重要的部分组成：进程管理、内存管理、硬件设备驱动程序、文件系统驱动程序、网络管理以及各种其它部分，见图 2-2。

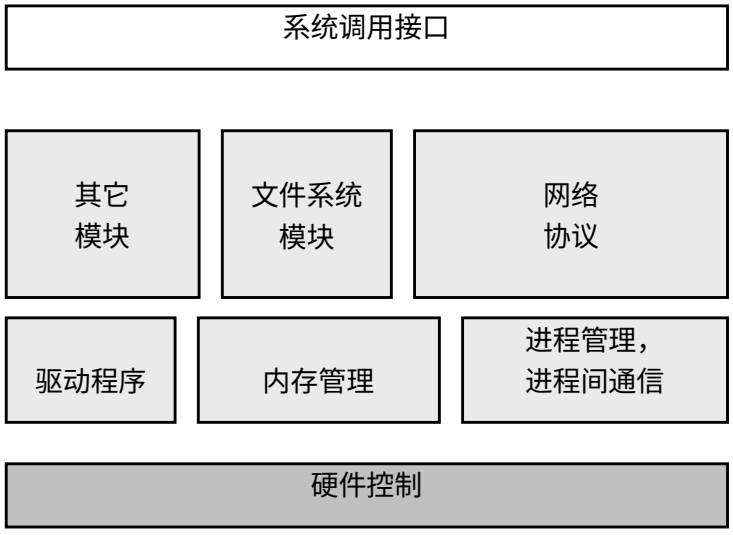

图 2-2 Linux 内核的组成

内核中最重要的部分是内存管理以及进程管理。进程管理用于创建进程，以及实现任务切换和多任务管理的功能。内存管理用来给进程分配内存区域以及高速缓冲分配和管理。

在最底层，内核包含各种硬件的设备驱动程序，而硬件设备驱动程序的数量是庞大的。内核采用虚拟文件系统（VFS）层将其文件系统操作从其实现中抽象出来，每一种文件系统类型提供了各自文件系统操作的实现。

和传统 RTOS 类似，Linux 内核提供了 OS 的各种系统功能服务，称为系统调用。用户可以通过系统调用在自己的应用程序中调用系统提供的功能。从应用程序的角度来看，系统调用和普通的函数调用非常相似。区别仅仅在于，系统调用由操作系统核心提供，运行于核心态；而普通的函数调用由应用函数库或应用程序提供，运行于用户态。

随 Linux 核心还提供了一些 C 语言函数库，这些库对系统调用进行了一些包装和扩展，因为这些库函数与系统调用的关系非常紧密，所以习惯上把这些函数也称为系统调用。实际上，很多我们习以为常的 C 语言标准函数在 Linux 平台上的实现都是靠系统调用完成的。

在 Linux 中，进程是不能访问内核的。它不能访问内核的内存地址空间，也不能调用内核函数。系统调用是这些规则的一个例外。所有进程同内核打交道的根本方式是系统调用。当一个进程需要内核提供某项服务时（像打开一个文件，生成一个新进程，或要求更多的内存），就会发生系统调用。

我们可以看到，和传统 RTOS 的应用程序直接调用内核服务，运行在内核空间不同，Linux 中应用程序是运行在用户空间的一个独立进程中的。这是应用模式上的最大区别。

2.1.3 根文件系统

内核启动之后的运行需要根文件系统（root filesystem）的支持。文件以及文件系统的概念是 UNIX 类操作系统的核心概念之一，内核的许多服务和设施都是以文件接口的方式来提供，例如对设备的访问就是通过操作设备文件的方式来进行的。此外，内核需要从文件系统装载和运行内核模块以及应用程序，而运行每一个进程时，内核需要赋予其一个当前工作目录的属性。内核启动之后的首要工作之一就是从根文件系统中装载和运行

init 程序，该程序是第一个运行的应用程序。

“内核＋库＋应用程序”是 Linux 系统运作的基本方式，这个方式严重依赖于文件系统，库和应用程序都需要存放在一个文件系统当中，核心的系统库和系统程序一般存放在根文件系统当中。

根文件系统可以存放在一个真实的硬件存储设备上，也可以存放在 RAM 里面进行操作，甚至可以通过网络文件系统来访问。

2.1.4 库和应用程序

除了内核之外，构成嵌入式 Linux 系统的主要组成部分就是库和应用程序了。与基于传统 RTOS 的应用开发和运行模式不同，Linux 操作系统提供的内核服务通常是不能由应用程序直接访问的。应用程序依赖于库来提供系统调用 API 和应用 API。Linux 系统的核心库是 GNU C 库——glibc，该库封装了 Linux 内核的系统调用，应用程序通过该库提供的接口来访问内核提供的功能和服务。在嵌入式 Linux 特别是 uClinux 系统中也可以选择使用 glibc 的替代品，如 uClibc 等，可以提高效率和减小库和程序的体积。除了 C 库之外，Linux 系统中通常还包括其它的应用级函数库，这可以根据应用需求来选择。

嵌入式 Linux 运行系统中提供的库均为动态库，应用程序在运行时与其所使用的库动态链接，动态库由内核装载和运行。也就是说，这些库并不是应用程序二进制代码的一部分，而是在应用程序启动时由内核装载到应用程序的地址空间的。动态装载机制使得使用某个库的多个应用程序可以同时使用该库的同一个代码实例，从而节省了运行时 RAM 空间和代码文件的存储空间，而且便于系统的升级。动态装载是 Linux 优于传统 RTOS 的地方，一般的 RTOS 都不提供动态装载的功能。

当然，嵌入式 Linux 开发环境中提供的库也可以是静态库，这时编译出来的应用程序将包括库代码的拷贝，程序体积较大，但程序运行时不再需要库的存在。在嵌入式系统的某些应用场合，静态链接是比较合适的链接方式。例如某个库只被一个到两个程序使用，或者只有库的一小部分被使用，使用静态链接就比较方便，而且能够节省存储空间。

2.2 开发流程、方法和开发环境

基于嵌入式 Linux 的嵌入式产品的开发流程和传统嵌入式系统的开发流程是类似的，概念上有许多共同之处。不过由于 Linux 内核与传统 RTOS 的差别，以及 Linux 作为开源软件的特点，基于嵌入式 Linux 的开发在开发工具及开发环境的搭建、资源的获取和运行调试方面都有自己独特的地方。

2.2.1 基本的开发流程和方法

嵌入式 Linux 开发一般包括如下的步骤：

- 安装和设置开发工具，建立交叉编译环境
- 在目标机上安装引导装载器（bootloader）
- 配置和编译内核
- 通过引导装载器安装和运行内核
- 驱动程序开发
- 准备目标根文件系统的内容
- 安装目标根文件系统
- 开发和运行应用程序
- 系统集成

这个流程基本上是一个由底而上、逐步推进的过程。当然，某些工作还是可以并行开展的，如应用程序的开发就可以和其它的过程同时进行。例如，我们进行基于 MiniGUI 图形系统的产品开发时，在进行内核移植及驱动程序开发的同时，就可以在 PC 平台上搭建 MiniGUI 开发环境，并进行 MiniGUI 应用程序的开发；当内核及根文件系统运行正确之后就可以把应用程序放到目标系统中运行和测试。

2.2.2 建立开发环境

和传统嵌入式开发一样，嵌入式 Linux 系统的开发通常是在桌面 PC 环境下进行的，而所开发出来的程序的运行目标环境是特定的嵌入式开发板或设备。

1) 宿主系统和目标系统

因为桌面系统提供了用于开发和编译的场所，该系统环境被称为宿主

系统。由于嵌入式系统开发板是应用程序最终的存储和运行场所，因此它又被称为目标系统。

用作嵌入式 Linux 开发的宿主系统类型通常有以下几种：

- Linux 工作站
- UNIX 工作站
- Windows 工作站

Linux 工作站是进行嵌入式 Linux 开发时最常用的一种宿主机系统，也是我们所推荐的开发环境。

首先，桌面 Linux 的运行机理和嵌入式 Linux 是类似的，在嵌入式系统系统中使用到的许多系统程序在桌面 Linux 中同样要用到。因此，以 Linux 工作站为开发宿主机系统有助于开发者熟悉 Linux 的运作原理、环境和常见工具的用法，这对嵌入式 Linux 开发是很有好处的。其次，互联网上免费可得的交叉编译工具链和嵌入式 Linux 厂商发行的开发套件通常都是运行在 Linux PC 之上的版本。此外，一般的桌面 Linux 发行版都提供了许多可用于嵌入式 Linux 开发的有用工具，非常方便。

使用 UNIX 工作站（如 Solaris）为宿主机系统也是可行的选择，因为 Linux 和其它 UNIX 系统类似，在 Linux 运行的工具大部分都可以在其它 UNIX 系统上运行。不过 Linux 和 Solaris 等系统毕竟还是有不少的差别，互联网上可供免费下载的 Solaris 版本的开发工具较少。所以，如果使用 Solaris 的话，开发者可能需要花更多的时间在开发环境的配置上。

嵌入式 Linux 开发也可以在 Windows 平台上来进行，不过通常需要一个 UNIX 的模拟环境——Cygwin。Cygwin 是一个非常强大的 UNIX 模拟和运行环境，提供了几乎所有的 Linux 系统上常用的工具程序和库，而且使用方法和在真正的 Linux 系统上一样。针对 Cygwin 环境的交叉编译工具链目前也很多，包括 arm-linux 工具链和 arm-elf 工具链等都有比较成熟的版本可供使用。不过 Cygwin 环境的一个缺点就是速度太慢，开发效率较低。

2) 宿主/目标的开发环境设置

宿主机/目标机开发环境的典型配置有三种：

- 宿主/目标系统通过某种方式连接，比如串口、以太网等

- 以可移动存储为媒介
- 完全在目标系统上开发

第一种基于串口连接的方式是最常用的方式，也是我们的介绍重点。宿主系统通常通过串口线连接到目标系统，宿主机上运行终端模拟程序 *minicom*（Linux）或 *HyperTerminal*（Windows），作为目标机的控制台（console），见图 2-3。

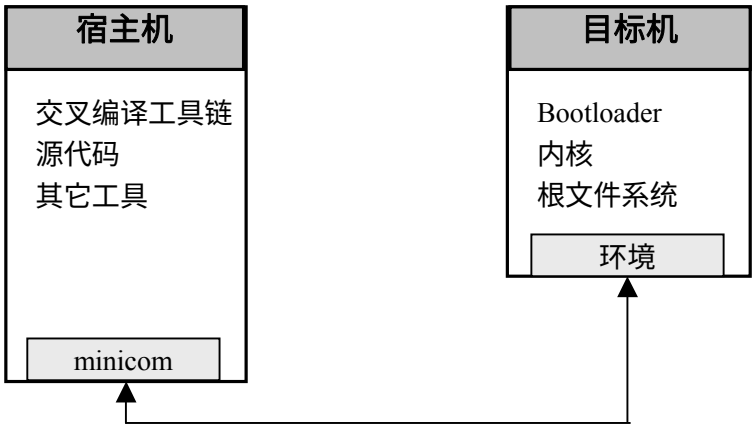

图 2-3 基于连接方式的开发环境配置

在这种方式下，包括引导装载器、内核和根文件系统等所有目标机运行所需的程序和数据都存放在目标机的存储介质之上，目标机通过串口或者以太网与宿主机通信。

不过，如果在开发阶段采用 NFS 网络文件系统的话，目标机的根文件系统就可以放在宿主机上，然后通过 NFS 来挂装和运行。内核同样也可以放在宿主机上，然后由引导装载器使用 TFTP（Trivial File Transfer Protocol）协议通过以太网来获取。

如果目标机同时具有串口和以太网连接的话，一般把以太网连接用于下载内核和根文件系统，而把串口用于调试和控制台，因为以太网连接的传输速度远比串行连接快。

第二种是基于可移动存储介质的方式，它适用于目标机使用 DOC、DOM、CompactFlash 等电子盘（可插拔存储器）情况。使用这种方式时，我们在宿主机上编译好内核和准备好根文件系统之后，把这些东西拷贝到电子盘上，然后再把电子盘从宿主机上拆下来，装到目标机上。引导装载器存储在目标机的 ROM 上，引导装载器启动之后再装载和运行位于电子盘上面的内核及文件系统；或者连引导装载器一起都可以放在电子盘上，

见图 2-4。

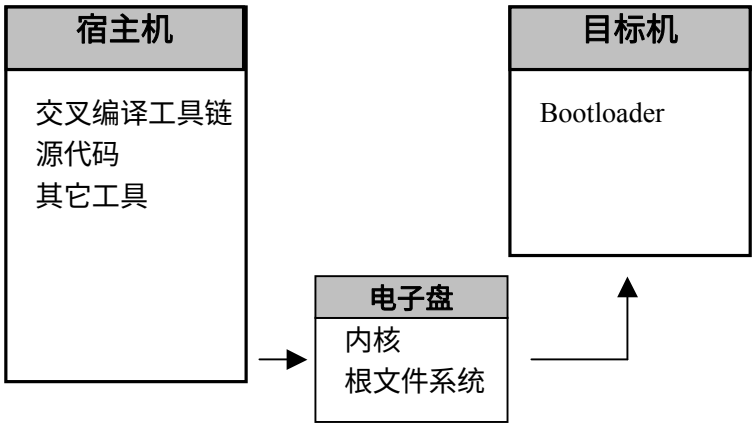

图 2-4 基于可移动存储介质的开发环境配置

使用可移动存储介质的方式需要宿主机在硬件和软件上都支持这种类型的存储器。例如，以 M-System 公司的 DOC 为存储介质时，硬件方面，主机需要有 ISA 插槽，这样把 DOC 放到一个 ISA 接口的转换器中，然后把该转换器插到宿主机的 ISA 插槽中就可以使用 DOC 了。软件方面，如果主机使用的是 Linux 系统，就需要安装 DOC 的相关驱动才能访问该设备；此外，M-System 公司提供的 DOC 相关工具很多都是 DOS/Windows 平台上的，使用这些工具还需要切换到 Windows 平台。

经常插拔存储器是比较麻烦和耗时的，而且容易损坏存储器，所以建议尽量避免采用这种方式。例如我们可以在系统开发的初始阶段采用该方式，然后在适当时候转向第一种基于连接的方式。

第三种方式和前两种方式有很大的不同，包括引导装载器、内核和根文件系统都放置在目标机上，而且开发工具也在目标机上。也就是说，采用的是本机编译工具链（native toolchain）和本机编译方式，见图 2-5。

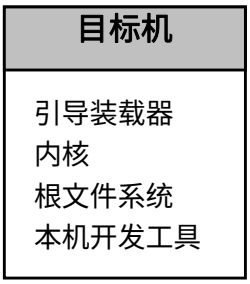

图 2-5 本地开发方式

使用本机编译方式的优点是开发过程非常简单，无需交叉编译、下载

到 Flash 或者拷贝到 DOC 等耗时的操作，但是需要目标机具有较强的计算能力以及较大的存储容量。这种方式常见于 x86 结构的系统，例如以硬盘作为存储介质的工控系统和使用电子盘的某些机顶盒系统。

使用这种方式的嵌入式 Linux 系统无论是在结构上还是在功能上都非常像一个桌面 PC，因此，许多开发者在这些系统中直接使用一个较老、体积较小的桌面 Linux 发行版，如 RedHat 5.0，然后在此基础上进行裁减和修改。

2.2.3 GNU 交叉开发工具链

嵌入式 Linux 的开发工具集一般包括源代码浏览器、交叉编译器、调试器、项目管理软件、引导映像创建工具，以及简化目标根文件系统生成的工具等。这些都是要安装到宿主开发机器上的，而其中的核心工具是 GNU 交叉开发工具链。

GNU 工具链一般包括以下几个重要组成部分：

- GCC 编译器
- Glibc 库
- Binutils 二进制工具
- GDB

1) GCC 编译器

GCC 是 GNU Compiler Collection 的缩写，是一个非常优秀的跨平台编译器，支持 x86、ARM、MIPS 和 PowerPC 等多种目标平台，支持 C、C++、Java、ADA、Fortran 和 Pascal 等多种高级语言。

GCC 的基本使用语法是：

```
gcc [ option | filenames ]...
g++ [ option | filenames ]...
```

其中 *gcc* 为 C 语言编译器，*g++* 为 C++语言编译器，***option*** 为 GCC 的命令行选项，而 ***filenames*** 为 GCC 要处理的文件。

编译一个程序通常需要经过四个阶段：预处理、编译、汇编和链接，而 GCC 编译器都能将不同阶段的输入文件做相应的处理。例如在预编译阶段 GCC 调用 *cpp* 工具，而在链接阶段 GCC 调用 *ld* 工具。

GCC 处理的文件类型包括：

- *.c*：C 源程序
- *.C*：C++源程序
- *.cc*：C++源程序
- *.cxx*：C++源程序
- *.h*：头文件
- *.s .S*：汇编源程序
- *.o*：目标文件
- *.a*：归档文件

2) 选择 C 函数库

C 函数库对于嵌入式 Linux 开发来说是必不可少的。通常我们可以根据目标系统是使用普通的 Linux 还是 uClinux，以及存储容量的大小来选择合适的 C 函数库。

以下是可供选择的 C 函数库版本：

- Glibc [11]：桌面Linux系统中使用的C函数库，功能强大，不过体积过于庞大，不太适用于小型的嵌入式系统；
- uClibc [12]：来自于uClinux项目，体积小，提供了大部分常用的C库函数，建议在小型嵌入式Linux系统中使用；
- newlib [13]：与uClibc的定位类似，由RedHat发布。

3) Binutils

Binutils 是一组二进制工具程序集，包括如下内容：

- *as*：GNU 汇编器（assembler），主要用于把汇编代码转换为二进制代码，并存放到目标文件中；
- *ld*：GNU 链接器（linker），主要用于确定相对地址，把多个目标文件、起始代码段、库等链接起来，并最终形成一个可执行文件；
- *ar*：创建归档文件（archive）、修改/替换库中的目标文件，向库中添加和提取目标文件；
- *nm*：列出目标文件中的符号；
- *objcopy*：拷贝和转换目标文件；

[11] http://directory.fsf.org/libs/c/glibc.html
[12] http://www.uclibc.org
[13] http://sources.redhat.com/newlib/ 或 http://sourceware.org/newlib/

- *objdump*：用于显示对象文件的信息；
- *ranlib*：根据归档文件中的内容建立索引；
- *readelf*：显示 ELF 格式执行文件中的信息；
- *size*：显示目标文件和可执行文件中各段（section）的大小和总大小；
- *strip*：去掉执行文件中多余的信息（符号和调试信息），可显著减小可执行文件。

一般来说，我们可以从网上下载一个适合目标平台的 GNU 工具链来使用。如果没有的话，就需要我们从源代码入手，编译和制作一个可用的工具链。在本书第 3 章，我们还会具体介绍这些二进制工具在开发中的使用用途。

2.2.4 终端模拟程序

在进行嵌入式 Linux 开发的时候，我们一般都需要运行一个终端模拟程序来和目标机器进行交互，显示来自目标机器的输出信息，或者输入命令对其进行控制。

目标系统的输出信息一般通过串行线传送到宿主机系统，终端模拟程序通过串口与目标机进行通信，好像宿主机是目标机的终端一样。

Linux 上流行的终端模拟程序是 *minicom*。在使用 minicom 之前首先需要进行适当的配置。运行 *minicom –s*，在“`Configuration`”菜单下选择“`Serial port setup`”，出现图 2-6 所示的配置界面。

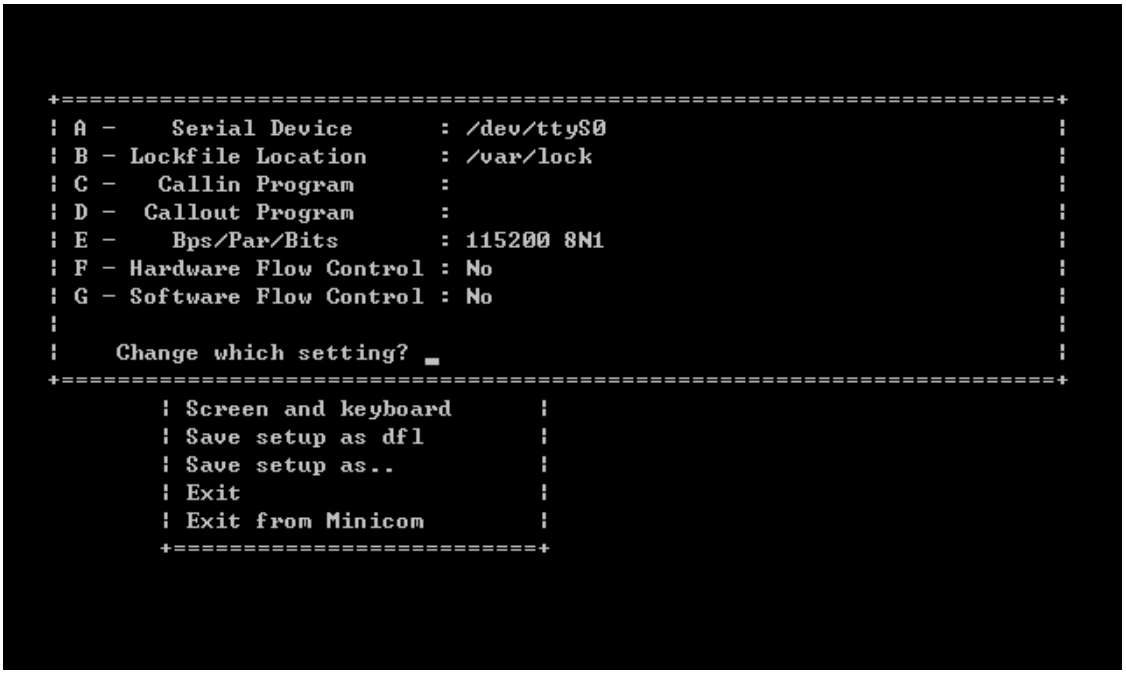

图 2-6 minicom 的配置

修改的方法是在“Change which setting?”提示符下输入想要修改的序号（不区分大小写），这时光标将跳到相应的项上；修改完该项后按回车键确认。全部修改完之后按回车回到主菜单，然后选择“Save setup as dfl”保存为缺省设置。

各配置项的含义是：

- A-Serial Device：串口设备名，如果是第一个串口的话名为*/dev/ttyS0*，第二个串口为*/dev/ttyS1*，依此类推。注意：用户必须对此设备具有读写等访问权限。
- E-Bps/Par/Bits：波特率、数据位、奇偶校验和停止位等端口参数的设置。常见的设置是：115200 8N1，就是 115200 的波特率、8 个数据位、无校验和 1 个停止位。该设置必须和目标机的串口设置一致，否则无法进行正常的通信。
- F-Hardware Flow Control：硬件流控制，一般设置为 No。
- G-Software Flow Control：软件流控制，一般设置为 No。

设置完端口之后，在 Shell 提示符下输入 *minicom* 就可以运行该终端模拟程序，如果这时串口有信息的话，就会显示在 *minicom* 屏幕上。*minicom* 本身的命令用“CTRL-A **<键>**”的方式输入，常用的命令有：

- CTRL-A Q：退出 *minicom*

- CTRL-A S：发送文件到串口（目标机）
- CTRL-A R：从串口（目标机）接收文件
- CTRL-A C：清屏
- CTRL-A Z：显示帮助信息

当目标机已经运行，但在 minicom 屏幕上却看不到任何输出信息，这时，按一下回车键就可以了。如果输出乱码，则应该检查串口设置是否正确。

2.3 系统引导

建立好开发环境之后，我们所做的第一个工作就是要把引导装载器在目标机上运行起来。

2.3.1 嵌入式 Linux 的启动过程

嵌入式 Linux 系统的启动和系统中的三个软件部分相关：

- 引导装载器（bootloader）
- 内核（kernel）
- init 进程

引导装载器通常是硬件系统启动时运行的第一个程序，它的主要工作是进行低级的硬件初始化，然后把 CPU 的控制权交给内核的启动代码。

最初执行的内核启动代码随目标系统体系架构的不同而不同，它的任务是进行体系架构相关的初始化，设置 C 语言代码的适当运行环境，然后调用内核的入口函数。比如在 PC 机上，LILO 或者 GRUB 就是我们这里所说的引导装载器。

之后，内核初始化高级功能，比如内存管理模块、进程管理模块和各种硬件设备等，之后挂装根文件系统，最后启动 *init* 进程。

余下的系统启动工作就完全由 *init* 进程在用户空间来完成了。*init* 进程通常来自于根文件系统中的 *init* 程序。

目标机从开机到 Linux 系统启动起来的典型过程如下：

- 处理器重新启动之后，执行 ROM 启动代码。
- ROM 启动代码初始化 CPU、内存控制器以及片上设备，然后配置存储器映射（memory map）。ROM 启动代码随后执行引导装载器。
- 引导装载器把 Linux 内核从闪存或者 TFTP 服务器解压到 RAM 中，然后跳转到内核的第一条指令处执行。
- 内核首先配置微处理器的寄存器和内存，然后调用 *start_kernel* 函数，它是内核初始化处理器无关部分的开始点。
- 内核初始化它的高速缓存和各种各样的硬件设备。
- 内核从存储介质或者通过 NFS 挂装根文件系统。
- 内核执行根文件系统中的 *init* 程序。
- *init* 进程装载运行时共享库。
- *init* 读取它的配置文件*/etc/inittab* 并执行其中定义的操作。一般而言，*init* 会执行一个系统启动脚本，如 */etc/rc.d/rcS*，该脚本配置和启动网络以及其它的系统服务。
- *init* 进入一个运行级别，在该级别下可以执行系统任务，开始登录进程，最后进入用户会话阶段。

2.3.2 引导装载器

引导装载器是和目标系统的硬件架构密切相关的。如果我们没有一个针对目标机的立即可用的引导装载器，则必须做一些移植的工作。一般情况下我们会选择一个合适的引导装载器，然后把它移植到目标机上。

下面是一些常用的引导装载器：

- LILO [14]、GRUB [15]：主要用于x86 架构，在嵌入式系统中使用不多。
- Blob [16]：LART工程开发的引导装载器。Blob支持众多的ARM架构开发板，它的缺点是功能稍显简单，不支持使用TFTP从网络装载内核。
- U-Boot [17]：一个功能强大的引导装载器，可以支持PowerPC和ARM等许多体系架构和开发板。
- RedBoot [18]：RedHat开发的一个非常强大的引导装载器，最初用于eCos操作系统。RedBoot支持目前流行的大部分嵌入式处理器架构

[14] http://sourceware.org/newlib/
[15] http://www.gnu.org/software/grub/
[16] http://www.lart.tudelft.nl/lartware/blob/
[17] http://sourceforge.net/projects/u-boot
[18] http://sources.redhat.com/redboot/

和很多开发板，而且它的文档很丰富，使用起来比较方便。

2.3.3 系统引导方式

根据所使用的存储介质和开发阶段的不同，我们可以采用不同的系统引导方式：

- 使用 Flash 存储设备
- 使用磁盘设备
- 通过网络引导

第一种引导方式通常用于最后的产品阶段。这时包括引导装载器及其配置参数、内核和根文件系统在内的所有代码和数据都存放在 Flash 上面，见图 2-7。

图 2-7 Flash 引导方式的布局

第二种引导方式适用于 x86 架构的系统。这时内核和根文件系统位于一个磁盘设备上，引导装载器从磁盘上装载第二个引导装载器（或者引导装载器的第二阶段），或者直接载入内核，磁盘某分区的文件系统将被用作根文件系统。

第三种引导方式又可以分为两种情况：通过网络挂装根文件系统，或者内核和根文件系统均通过网络来获取和安装。第 1 种情况下，内核位于 Flash 设备或者磁盘设备上，根文件系统通过 NFS 来挂装；第 2 种情况下，目标机上的存储介质只包含引导装载器，由引导装载器通过 TFTP 协议从网络下载内核，内核运行时通过 NFS 挂装根文件系统。

2.4 内核的选择、编译与安装

内核是嵌入式 Linux 系统的核心部分，将引导装载器在目标机上运行起来之后，下一步的工作就是选择一个合适的内核版本，进行适当的配置，编译之后下载到目标机上运行起来。如果没有直接可用的内核版本的话，也许您还需要做一些内核移植和驱动程序开发方面的工作，使内核能够支

持目标机的处理器和其它硬件设备。

本节将重点阐述嵌入式 Linux 系统上内核的选择、配置、编译和安装，而不对内核移植和驱动开发等方面的内容做深入介绍，这方面请参考《深入理解 Linux 内核》和《Linux 设备驱动程序》等相关书籍。

2.4.1 选择内核

我们前面说过，只有一个官方的Linux内核，由Linus Torvalds等人维护，可从 *http://www.kernel.org* 下载。但是，从该网站下载的Linux版本通常是不能直接用在您的目标机上的，因为该发布版本主要针对的是x86架构，而您的嵌入式系统一般用的是ARM、MIPS等架构的处理器。针对这些流行的嵌入式处理器的内核版本是由不同的组织来维护的，您需要从他们的网站来下载对应的内核版本或补丁：

- ARM：*http://www.arm.linux.org.uk/*
- MIPS：*http://www.linux-mips.org/*
- PowerPC：*http://penguinppc.org/*
- M68k：*http://www.linux-m68k.org/*

2.4.2 配置和编译内核

准备好内核的源代码之后，下一步就是配置和编译内核了。

1) 配置方法

在内核源代码目录下输入“ *make *config* ”等命令之一就可以对内核进行配置：

- “ *make config* ”提供了一个命令行界面，然后对每一个内核选项依次询问用户的选择。这种配置方法比较原始和麻烦，不建议使用。
- “ *make menuconfig* ”提供了一个基于终端 curses 的图形界面配置菜单，见图 2-8，使用起来非常方便，建议使用该方法。
- “ *make xconfig* ”提供了一个基于 TK 的 X Window 图形界面配置菜单，使用起来也不是很方便，不建议使用。
- “ *make oldconfig* ”和“ *make config* ”类似，不过它只提示用户设置之前没有配置过的选项。

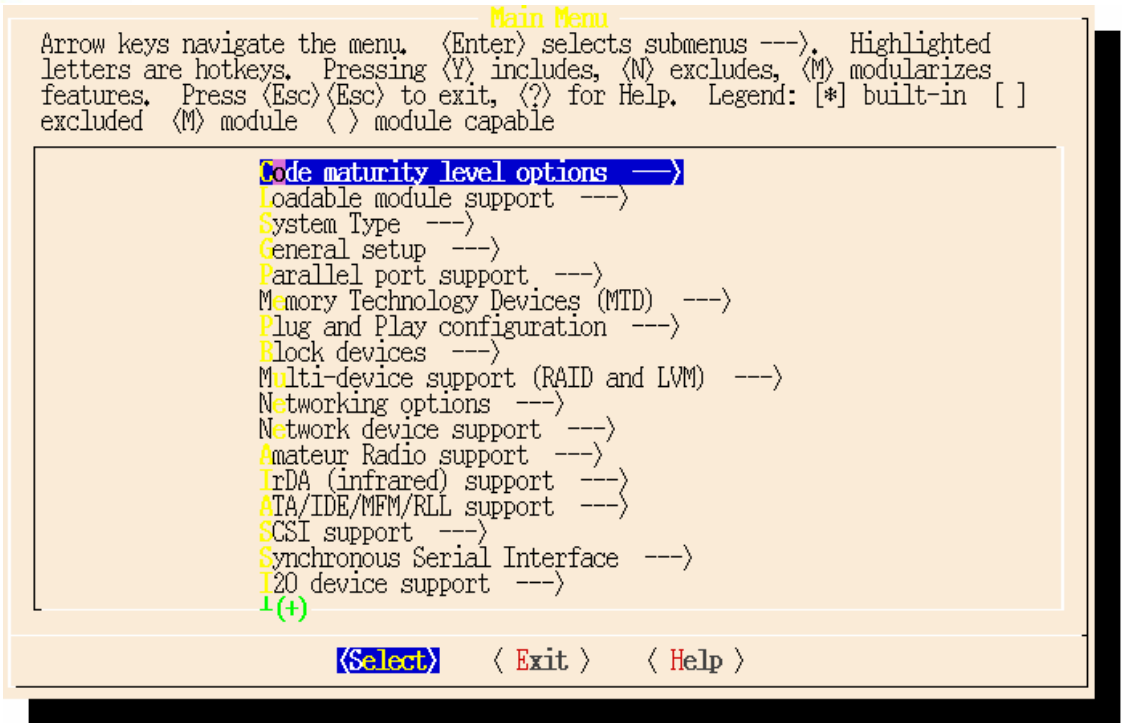

图 2-8 使用“ *make menuconfig* ”方式的内核配置界面

当配置完成之后，“ *make menuconfig* ”将在内核源代码根目录下生成一个 *.config* 配置文件，该文件包括了当前配置所有选项的设置信息。如果 *.config* 文件已经存在，然后运行“ *make menuconfig* ”命令的话，该命令将以 *.config* 为蓝本设置各选项的当前值。也就是说，我们要在上一次的基础上修改配置的话直接运行“ *make menuconfig* ”就可以了。

内核配置菜单还提供了保存配置文件和载入配置文件的功能，我们可以把设置好的配置保存到一个指定的文件中，也可以从一个已有的配置文件中装载先前的配置，然后以此为基础进行设置。

某些时候我们需要给“ *make menuconfig* ”命令加上合适的参数，例如编译 ARM Linux 内核时可能会使用如下的命令：

```
user$ make ARCH=arm CROSS_COMPILE=arm-linux- menuconfig
```

ARCH 变量指示目标体系架构的名字，*CROSS_COMPILE* 变量告诉 Makefile 编译内核时要使用的交叉编译器。这两个变量对于交叉编译内核非常重要，因此必须设置。不过我们可以直接修改内核源代码根目录下的 Makefile 文件中相应变量的定义，这样就不用每次都输那么长的命令了：

```
ARCH := arm
```

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

【提示】Makefile 会自动把 CROSS_COMPILE 变量的值和'gcc'组合起来构成交叉编译器的名字。例如我们要指定使用 arm-linux-gcc 编译器的话，只需把 CROSS_COMPILE 设为 arm-linux- 就可以了。

2) 内核配置选项

内核的配置选项很多，需要我们根据目标机的实际情况仔细选择。不过，并不是所有的选项都需要设置，例如某些选项对嵌入式 Linux 系统没有太大的意义，某些选项是不包括在我们的硬件和软件方案中的。

比较重要的内核选项有：

- Code maturity level options
- Loadable module support
- General setup
- Memory technology devices
- Block devices
- Network device support
- Character devices
- Filesystems
- Console drivers

我们可以参考配置界面中的说明来确定每个选项的含义。不需要的功能都可以去掉，这样能有效减小内核的尺寸。

3) 编译内核

编译内核通常分为几个步骤。首先要建立内核依赖关系：

```
user$ make dep
```

内核中的 C 源代码文件和头文件之间存在有一定的依赖关系，内核的 Makefile 必须知道这些依赖关系才能确定需要编译哪些源代码文件。但是使用“ *make menuconfig* ”命令配置完内核之后，该命令会自动生成一些编译所需的头文件。因此，当我们改变内核的配置之后，在编译之前应该重新建立正确的依赖关系。

【提示】'make dep'的步骤在 2.6 版本的内核中不再需要了。

执行完“ make dep ”命令之后，我们就可以执行“ *make zImage* ”、“ *make bzImage* ” 或“ *make vmlinux* ”来编译内核映像了。比如：

```
user$ make zImage
```

zImage 目标的含义是编译一个使用 gzip 算法压缩的内核映像 *zImage*；*bzImage* 的含义是“ big zImage ”，该映像也使用 gzip 算法，但确保生成的较大压缩内核可以工作；*vmlinux* 目标的含义是只编译一个非压缩的内核映像 *vmlinux*。注意：“ *make zImage* ”既生成 *zImage*，也生成 *vmlinux*。

在 x86 架构上，*zImage* 和 *bzImage* 的区别是 *zImage* 大小不能超过 512KB，而 *bzImage* 没有这个限制；为了克服 x86 架构上实模式的限制，*bzImage* 使用了不同的布局和装载方法，因此体积较大。

对于其它处理器架构来说，一般只有 *zImage* 和 *vmlinux* 映像，没有 *bzImage* 映像；Makefile 里面的 *bzImage* 目标等同于 *zImage*。

如果出现编译失败，则有可能是因为我们的配置不够完整，选择的内核功能不足以支持当前的某些特性；或者我们选择的目标架构不支持某些特性。为了避免出现这种情况，我们要仔细考虑各模块之间的依赖关系，以及目标机的实际需要。

如果编译成功，所生成的内核映像文件将放置在 *arch/**${ARCH}**/boot* 目录下，对于 ARM 架构来说就是 *arch/arm/boot* 目录。

如果我们在配置内核时选择了内核模块支持的话，还需要使用如下的命令单独地编译内核模块：

```
user$ make modules
```

【提示】可以使用'make clean'或'make distclean'来清除之前的（配置和）编译结果，然后重新（配置和）编译。

2.5 准备根文件系统

根文件系统一直以来是所有 UNIX 类操作系统的一个核心组成部分，在内核启动的最后阶段，所执行的操作之一就是挂装根文件系统。

根文件系统是 Linux 有别于传统 RTOS 的主要特征之一，它赋予了 Linux 系统强大和灵活的功能，同时也带来一定的复杂性。为了构建一个成功的嵌入式 Linux 系统，我们需要对根文件系统的基本结构有清楚的认识和理解，包括如何安装系统库、内核模块、内核映像、设备节点、系统应用程序和用户应用程序。我们还需要根据项目的实际情况来确定如何配置系统的各种初始化脚本和配置文件。在构造好根文件系统之后，我们还要把它放置到实际存储设备的适当位置才能够起作用。

2.5.1 根文件系统的基本结构

Linux 的根文件系统包含内核所需的文件和可执行文件，还有用于系统管理的可执行文件。

Linux 操作系统的根文件系统以树的结构方式组织，最上层的目录名为“/”，其下面的顶层目录多为有不同意义和用途的系统目录。典型的根文件系统目录结构如下：

- *bin*：包含基本的用户命令工具程序
- *sbin*：包含基本的系统管理程序
- *boot*：包含内核映像及启动相关文件
- *etc*：包含系统配置文件和脚本
- *lib*：包含系统库和内核模块
- *usr*：用户程序及库目录
- *home*：用户主目录
- *root*：root 用户主目录
- *dev*：设备文件目录
- *opt*：额外软件包所在目录
- *mnt*：文件系统临时挂装目录
- *var*：包含运行时改变的文件，例如 lock 和 log 文件

- *proc*：内核创建和使用的虚拟文件系统，存放运行时系统信息
- *tmp*：临时文件目录

一个嵌入式 Linux 系统的根文件系统没有这么复杂，因为它通常不支持多用户登录，功能和用途相对简单。一般来讲，只有*/bin*、*/sbin*、*/etc*、*/lib*、*/dev*、*/usr*、*/var* 和*/proc* 目录及其下的内容是需要的。而且，嵌入式 Linux 根文件系统的组成比较灵活，开发者可以根据项目的实际情况来决定采用什么样的结构来组织文件系统的内容。

例如，一个最简单的根文件系统也许只有*/bin*、*/etc*、*/lib*、*/dev* 和*/var* 这几个基本的目录，这种情况通常适用于使用 uClinux 的微型嵌入式系统中。

2.5.2 函数库

Linux 中的应用程序以以下两种方式之一链接到外部函数 :要么在构建时与静态库（*lib*.a*）静态地链接，并且将库代码包含在该应用程序的可执行文件里；要么在运行时与共享库（*lib*.so*）动态地链接。通过动态链接装入器，将动态库映射进应用程序的可执行内存中。在启动应用程序之前，动态链接装入器将所需的共享目标库映射到应用程序的地址空间，或者使用系统共享的目标并为应用程序解析所需的外部引用。

Linux 系统对动态库的命令和链接有一定的规则，我们在开发时和构建根文件系统时都要注意。在库目录下面的某个动态库的相关文件通常包括一个实际的动态库文件和指向该库文件的符号链接，文件按库的版本号命名。例如 glibc 的实际使用库文件是 *libc-2.3.3.so*，对应的库符号链接文件是 *libc.so.6*。后者称为动态库的共享对象名称，这个名称将被保存在链接该函数库的可执行程序映像中，这样，只要在目标系统上存在这个名称的动态库就可以了。

在系统库方面，只有 C 函数库和动态链接器 *ld-linux.so* 是必需的，其它的库根据项目的实际需要选择；有些库在开发的时候需要，在实际产品的运行时系统中不需要。每个 Linux 系统都需要一个 C 库。C 库提供了标准 C 应该包括的常用的文件操作（打开、读和写），内存管理操作（*malloc* 和 *free*），以及许多其它 Linux 系统调用函数。

Linux 上传统的 C 库是 GNU C 库——Glibc，大多数桌面 Linux 系统

和较大规模的嵌入式 Linux 系统使用 Glibc。Glibc 是成熟的、经过良好测试的标准函数库。遗憾的是，它的体积较大，使用的内存数量相对也较多，对于许多微型嵌入式 Linux 系统来说是不可接受的。

Glibc 包括如下的主要组成部分：

- *ld-linux*：动态链接器，必需
- *libc*：标准 C 函数库，必需
- *libm*：数学库，一般需要
- *libdl*：用于动态装载共享库，较少用到
- *libcrypt*：加密附加库，需要认证的程序用到，较少使用
- *libpthread*：POSIX 线程库，一般需要

如果需要某个函数库，我们可以将这些库和对应的符号链接从开发系统拷贝到目标根文件系统的 */lib* 目录下。简单起见，应该使用 *–d* 选项或者 *–a* 选项调用 *cp* 命令，这样可以保留完整的符号链接信息：

```
user$ cp -a lib*.so* ${ROOTFS}/lib/
```

为了减小运行时库的大小，我们应该使用交叉编译版本的 *strip* 工具来处理根文件系统中的库文件，把二进制文件中包含的符号表和调试信息删除掉。例如 ARM 版本的 *strip* 工具通常如下使用：

```
user$ arm-linux-strip ${ROOTFS}/lib/*.so
```

uClibc 包含的库比 Glibc 少，不过核心部分和 Glibc 是类似的，包括 *libc*、*ld-linux*、*libdl*、*libm*、*libcrypt* 和 *libpthread* 等。这些库的用法和 Glibc 是一样的。

2.5.3 内核映像和内核模块

内核放置在存储设备或根文件系统中的位置取决于引导装载器的装载方式。如果引导装载器被设置为从根文件系统中装载指定的内核映像，就应该把映像文件拷贝到根文件系统中，通常是 */boot* 目录下。

内核模块应该拷贝到 ***${ROOTFS}****/lib/modules* 目录下。此外，我们还应该在 ***${ROOTFS}****/etc* 目录下添加一个 *modules.conf* 的配置文件，该文件指定系统启动时自动装载的内核模块。

2.5.4 设备文件

在以文件抽象为核心概念的 Linux 操作系统中，每个设备都有一个对应的设备文件，这些设备文件（节点）位于根文件系统的 */dev* 目录下面。桌面 Linux 系统中 */dev* 目录下的设备节点是非常多的，而在嵌入式 Linux 系统中我们只需根据实际情况创建和设置系统运行必需的设备节点即可。

下面列出的是一些基本的设备节点：

- *mem*：物理内存设备
- *null*：空设备（向该设备写入的数据将完全被丢弃）
- *zero*：零字节源设备（从该设备上读出的数据全部为零）
- *random*：随机数生成设备
- *tty0*：当前的虚拟控制台
- *ttyS0*：第一个串口设备
- *tty*：当前的 tty 设备
- *console*：系统控制台
- *fb0*：Framebuffer 设备

在不同的系统中，设备节点的名字和设备号可能会有所不同；根据实际需要，可能会用到其它的设备节点。

有如下几种在目标根文件系统中创建设备文件的方法：

- 在主机上为目标系统准备的根文件系统中，手工创建 */dev* 目录中的设备节点。
- 在目标文件系统的启动脚本（如 *rcS*）中使用 *mknod* 命令动态创建。
- 从互联网下载基本可用的文件系统包，在此基础上修改。
- 使用相关的文件系统工具自动创建。

如果设备节点较少的话，可以在目标根文件系统中使用 *mknod* 命令手工创建：

```
user$ cd ${ROOTFS}/dev
user$ su
Password:
root# mknod -m 666 null c 1 3
…
```

mknod 命令用于创建设备文件节点，需要以超级用户的身份使用。上面 *mknod* 命令的*-m* 参数指定节点的访问许可属性为 666，null 为设备文

件名，c 指定设备类型为字符设备，1 和 3 为该设备的主、次设备号。可以参考内核源代码文档 *Documentation/devices.txt* 文件来确定每个设备的主设备号和次设备号。

或者我们也可以把主机系统 */dev* 目录下的设备文件直接拷贝到 ***${ROOTFS}****/dev* 目录下来使用。注意在调用 *cp* 命令时要使用*-a* 选项，否则 *cp* 命令会试图打开要复制的设备。

```
user$ cp -a /dev/zero ${ROOFS}/dev/
```

如果根文件系统内容较多、比较复杂的话，我们可以从互联网上选择和下载一个基本可用的根文件系统包，在此基础上做出符合我们需要的根文件系统。

此外，对于需要灵活处理的情况，我们可以在目标文件系统的启动脚本，如 *rcS*，中使用 *mknod* 命令来动态创建所需的个别设备节点。

如果您觉得手工处理根文件系统实在很麻烦，JFFS2 和 CRAMFS 文件系统相关的系统工具可以帮您在这些类型的根文件系统中自动创建设备文件节点，减少一些工作量。

2.5.5 安装系统程序

UNIX 类操作系统以丰富的命令著称，Linux 也不例外，桌面系统中通常提供了上千个工具程序。嵌入式 Linux 系统不需要太多的命令和工具，而且一个个地交叉编译所有的工具程序太麻烦了。

面向嵌入式 Linux 系统的一个很好的系统程序解决方案是 BusyBox（*http://www.busybox.net/*）。该工具体积小、功能强，有“瑞士军刀”的美誉，已经成为嵌入式 Linux 系统开发的必备工具，得到了广泛的应用。

BusyBox 在一个体积很小的程序中集成和实现了 Linux 下的许多常用工具命令。我们只需编译和安装 BusyBox，就可以在目标系统中使用很多经常在桌面上使用的命令。例如：*ls*、*cp*、*mkdir*、*init*、*ifconfig*、*cat*、*rm*、*mount* 等。

需要注意的是，BusyBox 提供的命令一般都只实现了基本的功能，许多不常用的或者在嵌入式系统中用处不大的命令选项是不支持的。不过这

些基本的命令已经足够好用了，而且我们在编译 BusyBox 时还可以进行配置，选择我们需要的命令。

BusyBox 既可以和 C 库动态链接，也可以静态链接；它支持 glibc 和 uClibc 两个流行的 C 库。

2.5.6 系统初始化

当 Linux 内核完成根文件系统的挂装、内核初始化结束时，它执行第一个用户进程 *init* 并把系统的控制权交给它。*init* 进程执行用户级的系统初始化工作，并生成所有其它的进程；当系统关闭时，*init* 负责结束所有用户进程并卸载所有挂装的文件系统。也就是说，*init* 进程是内核启动之后的系统总负责者。

可以通过内核启动参数“ *init*= ”来指定内核启动后所执行的 init 程序。内核首先寻找 */sbin/init*，接着是 */etc/init*， 然后是 */bin/init*，最后它会尝试 */bin/sh*。

init 的概念来自于 UNIX System V，传统的 *init* 程序完成许多工作；不过在嵌入式 Linux 系统中，*init* 没有那么复杂。

可以由以下几种方式来创建嵌入式 Linux 中的 *init* 程序：

- 使用类似 System V init 的 *init* 程序
- 使用 BusyBox 的 *init* 程序
- 自己编写一个简单的 *init*

1) System V init

大多数桌面Linux发行版中提供的*init*程序是由Miquel van Soorenburg编写的System V init。该*init*程序功能强大，不过体积较大，而且需要其它命令工具程序的配合使用，包括*halt*、*runlevel*、*shutdown*、*mesg*等。可以从 *ftp://ftp.cistron.nl/pub/people/miquels/sysvinit/* 获取该init程序的源代码。

交叉编译 System V init 程序很容易，然后就可以把 init 复制到目标根文件系统中。我们还需要为 init 准备一个适当的 */etc/inittab* 文件，并在 */etc/rc.d/* 目录下添加适当的启动脚本文件。一般情况下，*/etc/inittab* 定义了系统的运行级别，*/etc/rc.d/* 目录下的脚本文件定义了每一个运行级别上

运行的服务。表 2-1 描述了 init 的 7 个运行级别和一般用途。

表 2-1 System V init 的运行级别

运行级别	意义
0	关闭系统
1	单用户运行模式
2	NFS 多用户运行模式，命令行登录
3	完整多用户运行模式，命令行登录
4	不使用
5	X11，图形用户界面登录
6	重启系统

每个运行级别对应于一个特定的服务。例如，当进入运行级别 5 时，init 将启动 X11 和图形登录界面。当在用户级别之间切换时，前一个运行级别启动的服务被关闭，新的运行级别的服务被启动。运行级别 0 和 6 有特殊的含义，它们被用于安全地关闭系统；涉及的操作可能包括卸载除根文件系统之外的所有文件系统，然后把根文件系统重新挂装为只读模式以防文件系统损坏的情况发生。

在大多数桌面 Linux 系统中，系统启动时的缺省运行级别是 5。在嵌入式系统中，缺省运行级别可以设为 1，因为嵌入式系统中一般不需要多用户支持和访问控制。

2) BusyBox init

BusyBox 的 *init* 命令提供了类似传统 *init* 程序的系统启动和初始化功能。和 BusyBox 的其它工具命令一样，该 *init* 命令非常适用于嵌入式 Linux 系统，因为它只提供了嵌入式系统所需的 init 大部分功能，不包括 System V init 中那些用处不大的特性。此外，因为 BusyBox *init* 是一个独立的软件包，运行时不依赖于其它软件工具，因此开发和维护起来非常方便。不过，美中不足的是 BusyBox *init* 不支持运行级别（runlevel），这对于某些功能要求较强的嵌入式 Linux 系统来讲是不够的。

在以 BusyBox 为命令工具包构建的根文件系统中，*init* 程序（如 */sbin/init*）只是一个指向 *busybox* 程序（*/bin/busybox*）的符号链接。也就是说，BusyBox 是系统第一个执行的应用程序。BusyBox 由文件名识别出要执行的命令是 *init*，然后跳转到 *init* 例程去运行。

BusyBox *init* 依次执行如下的任务：

- 设置 *init* 的信号处理函数

- 初始化控制台
- 分析 */etc/inittab* 配置文件
- 执行系统初始化脚本，如 */etc/init.d/rcS*
- 执行 *inittab* 中 action 类型为 wait 的命令
- 执行 *inittab* 中 action 类型为 once 的命令
- 循环执行 *inittab* 中 action 类型为 respawn 的命令
- 循环执行 *inittab* 中 action 类型为 askfirst 的命令

完成控制台的初始化之后，BusyBox 检查 */etc/inittab* 文件是否存在。如果该文件不存在，BusyBox 将使用一个缺省的 *inittab* 配置。通常，我们在 *inittab* 中设置系统重启、系统关闭和 init 重启的缺省动作（action）；此外，还设置 action 以启动从 */dev/tty1* 到 */dev/tty4* 的四个虚拟控制台上的命令解释器。

如果 BusyBox *init* 找到 */etc/inittab* 文件，它将分析该文件，把其中包含的命令记录到内部结构当中，然后在适当的时刻执行这些动作。BusyBox *inittab* 文件的格式和 System V *init* 程序使用的 *inittab* 格式有所不同，具体请参考 BusyBox 软件包的相关文档和示例 *inittab* 文件。

inittab 文件中的每一行都有如下的格式：

```
id:runlevel:action:process
```

这个格式和传统 System V init 的格式是类似的，不过要注意：id 项有不同的含义。id 一般用来指定所要启动的进程（process 项指定）的控制终端。如果所要启动的进程不是一个交互式的 shell 的话，该项可以为空。Shell，例如 BusyBox 的 *sh*，总是需要一个控制终端的。BusyBox 的 *init* 忽略 *runlevel* 项，因为它不支持运行级别，所以该项可以为空。process 项指定所要运行的程序名和启动的命令行参数。action 项是应用到 process 的八个动作之一，见表 2-2。

表 2-2 BusyBox inittab 的动作类型

Action	效果
sysinit	指定初始化脚本的路径
respawn	当某进程结束时重启该程序
askfirst	类似 respawn，在启动程序前提示用户按回车键
wait	等待所启动的进程结束
once	只运行进程一次，不等待进程结束
ctrlaltdel	按 Ctrl+Alt+Del 组合键时执行的进程

shutdown	系统关闭时运行的进程
restart	init 重启时运行的进程，通常是 init 本身

下面是一个简单的 *inittab* 例子：

```
::sysinit:/etc/init.d/rcS
::respawn:/sbin/getty 115200 ttyS0
::restart:/sbin/init
::shutdown:/bin/umount -a -r
```

该 *inittab* 告诉 *init* 执行如下的任务：

a) 设置*/etc/init.d/rcS* 为系统初始化脚本
b) 在第一个串口以 115200 的波特率启动登录对话
c) 设置*/sbin/init* 为 init 重启时执行的程序
d) 告诉 *init* 在系统关闭时卸载所有的文件系统

这里 id 项为空，这不会影响命令的正常执行。

所有这些任务都是在 *init* 执行完指定的系统初始化脚本之后进行的。该初始化脚本一般用来进行系统的基本设置和初始化系统的各个组件，包括：

- 重新挂装根文件系统为读写模式
- 挂装另外的文件系统
- 初始化和启动网络接口
- 启动系统的各个守护进程

下面是一个系统初始化脚本的简单例子：

```
#!/bin/sh

# Remount the root filesystem in read-write
mount -n -o remount,rw /

# Mount /proc filesystem
mount /proc

# Start the network interface
/sbin/ifconfig eth0 192.168.1.200
```

上面的脚本还需要 */etc/fstab* 文件的存在。下面是一个简单的 *fstab* 例子：

```
# /etc/fstab
# device      directory   type    options
#
/dev/nfs      /           nfs     defaults
none          /proc       proc    defaults
```

这里指定从 NFS 挂装根文件系统以方便开发。

3) 自己编写 init

如果您的系统很简单，您完全可以自己参考 System V init 或者 BusyBox init 写一个简单的 *init* 程序，满足您特定的需求即可。比如，我们可以简单编写一个 *init* 程序，该程序在启动后完成一些初始化工作，如挂装文件系统、创建设备文件、初始化网络参数等等。之后，*init* 可 *fork* 一个子进程，然后在子进程中执行某个预先定义好的程序，而 *init* 进程本身则等待子进程退出；当子进程终止时，*init* 进程可以根据子进程的终止状态完成相应的工作，如重新启动程序或者安全关机等。

2.6 选择和安装文件系统

在准备好目标根文件系统的内容之后，我们需要为它选择一个合适的文件系统类型，这个过程通常需要考虑到目标系统的用途和文件系统的特性。一般而言，选择一个文件系统类型时需要考虑如下的因素：是否可写、持久存储能力、压缩特性和断电保护能力等。

此外，我们也可以在一个嵌入式 Linux 系统中使用多个不同类型的文件系统。例如，一个简单的系统有可能只需要对临时文件有写能力，其它的执行文件、库和配置文件等都是只读的；这种情况下，我们可以对根文件系统使用 CRAMFS 类型的文件系统，而对其临时文件部分使用 TMPFS 文件系统或者 RAM disk 上的文件系统，通常挂装到 */var/tmp* 目录，而 */tmp* 目录只是指向 */var/tmp* 的一个符号链接。

2.6.1 存储设备和文件系统

在嵌入式系统中使用的存储设备可以分为内部存储器和外部存储器两类，内部存储设备如 RAM（随机访问存储器）一般用于运行时代码和数据的保存，外部存储器用于保存程序映像和数据。我们这里所说的存储设备是指外部存储器。

一般而言，使用嵌入式 Linux 构建的嵌入式系统体积较大，所以要求的存储容量也相对要多一些。我们可以根据嵌入式 Linux 系统的运算能力

和存储容量（尺寸）把系统大致分为三大类：

- 小系统：较低性能的 CPU（无 MMU）和 4MB 左右的 ROM 和 RAM
- 中等系统：稍高性能的 CPU 和 8-16MB 左右的存储容量
- 大系统：高端 CPU 和 32MB 以上存储容量

使用 uClinux 的系统一般属于第一类；使用普通 Linux 的系统一般属于第二、三类。

嵌入式系统中使用的存储设备和桌面系统中使用的有很大的不同。传统的嵌入式系统中一般使用 ROM 和 EEPROM 等非易失性存储器来保存引导代码和操作系统映像，嵌入式 Linux 系统中一般使用基于 Flash（闪存）的存储芯片和电子盘。这些存储设备在由内核使用之前必须经过恰当的设置，而且使用这些存储设备也需要特别的软件工具。

1) Flash 的类型与特点

Flash 是一种非易失性存储器，可以对存储器以单元块为单位进行擦写和再编程，而且不需要特殊的编程器。Flash 实际上是对 EEPROM 的改进品种，任何 Flash 元件进行写入操作前必须先执行擦除操作。

Flash 主要分为 NOR 和 NAND 两类，分别具有不同的特点和用途。

Intel 于 1988 年首先开发出 NOR Flash 技术，它的特点是芯片内执行（XIP），这样应用程序可以直接在 Flash 上运行，不必再把代码读到系统 RAM 中。使用 NOR 就象和使用 RAM 一样，非常方便和直接。也就是说，NOR Flash 是在 CPU 的内存物理地址空间寻址范围之内的。

NOR 的传输效率很高，在 1-4MB 的小容量时具有很高的成本效益。但 NOR 要求在进行擦除前，先要将目标块内所有的位都写为 0，因此，擦除速度很慢。

NAND 器件使用独立的总线来存取数据，各个产品或厂商的方法可能各不相同。因此，NAND 要比 NOR 复杂得多，向 NAND 器件写入信息需要较高的技术和技巧。NAND 的读和写操作采用 512 字节的块，类似硬盘等块设备。

NAND Flash 的生产过程更为简单，可以在给定的模具尺寸内提供更高的容量，价格也相对较低。目前，NOR Flash 的容量从几 KB～64MB 不

等，NAND Flash 存储芯片的容量从 8MB～128MB，而作为一种特殊 NAND Flash 的 DiskOnChip 可以达到 1024MB。

NAND Flash 每个块的最大擦写次数是 100 万次，而 NOR Flash 的擦写次数是 10 万次。不过在可靠性方面，NAND 不如 NOR，容易出现位反转等故障。

下面是 NOR 和 NAND 特点的总结：

- NOR 的读取速度比 NAND 稍快一些
- NAND 的写入速度和擦除速度比 NOR 快很多
- NAND 的容量一般较大，耐用性好
- NOR 支持 XIP

根据 NOR 和 NAND Flash 的特点，经常把 NOR 用作启动 ROM，而把 NAND 用作大容量的文件系统存储介质。

2) MTD 设备

在 Linux 的术语中，MTD（memory technology device）包括所有的内存设备，例如传统的 ROM、RAM、Flash 和 M-System 公司的 DOC（DiskOnChip）。每一种内存设备都有不同的功能、特性和限制。在传统的嵌入式系统开发中，开发者必须针对不同的设备使用特定的硬件、软件工具和开发方法。

为了尽量避免对不同的设备使用不同的工具和技术，以及在不同的设备之上提供通用的功能和接口，Linux 内核开发者开发出了 MTD 子系统这样一个内存设备抽象模块。MTD 提供了一个统一的软件架构，把底层的 MTD 设备芯片驱动和高层的 MTD 用户模块无缝地结合起来。MTD 用户模块是指 MTD 子系统中建立在 MTD 设备芯片基础之上的内核软件模块，它向上层的内核其它模块提供了一致的抽象模型和接口。MTD 子系统的架构如图 2-9 所示。

图 2-9 内核的 MTD 子系统

MTD 芯片驱动通过一个 *add_mtd_device()* 函数和 *mtd_info* 结构把自己注册到 MTD 子系统中。*mtd_info* 结构定义了一套通用的设备操作接口，包括读、写、擦除和同步，高层的 MTD 模块将调用这些预定义的回调函数来进行相应的操作。

下面是常见的一些 Linux MTD 芯片驱动：

- DOC
- 非 DOC 封装的 NAND Flash
- Common Flash Interface (CFI)
- RAM 和 ROM

DOC 是一种特殊封装的 NAND Flash，由 M-System 公司提供。目前 Linux 支持 DOC1000、DOC2000 和 DOC Millennium 等系列的 DOC。

MTD 子系统提供了访问传统 RAM 和 ROM 的芯片驱动，这类驱动可以把映射到系统物理地址空间的内存当作 MTD 设备来使用。

此外，为了方便 MTD 用户模块的测试，MTD 子系统中提供了两类模拟真实 MTD 硬件设备的驱动：一个使用系统虚拟内存来模拟 MTD 设备，另外一个使用普通块设备来模拟 MTD 设备。

MTD 子系统允许把内存设备划分为多个分区。和硬盘分区类似，每个 MTD 分区可以当作一个独立的 MTD 设备来使用，而且每个分区的数据格式可以是不相同的。

2.6.2 各种类型文件系统的特性

表 2-3 列出了嵌入式 Linux 系统中常见的文件系统类型及其特性。

表 2-3 常见文件系统类型的特性

文件系统	可写	持久存储	断电保护	压缩	存储介质
CRAMFS	No	No	No	Yes	Flash
JFFS2	Yes	Yes	Yes	Yes	Flash
Ext2 over NFTL	Yes	Yes	No	No	Flash
Ext3 over NFTL	Yes	Yes	Yes	No	Flash
Ext2 over RAM disk	Yes	No	No	No	RAM

“可写”是指文件系统的内容是可以在运行时被改变的；“持久存储”

是指修改后的文件系统内容可以被“持久”保存，系统重启后不会丢失；“断电保护”是指文件系统可以从系统断电中恢复到正确状态，修改的内容不会丢失，文件系统不会出现混乱和崩溃；“压缩”是指文件系统的内容是被压缩的；“存储介质”是文件系统运行时所存放的物理介质，包括硬盘、Flash 和 RAM 等。

如果我们在运行时需要更新文件系统中的内容，才需要“可写”的能力；如果需要持久保存更新后的内容，才需要“持久存储”的能力。显然，一个只读的文件系统是不需要持久存储能力和断电保护的，因为它的内容不会被改变。同样，没有持久存储能力的文件系统也是不需要断电保护的。

一个压缩的文件系统可以大大地提高存储容量从而减小了存储介质的成本，不过在运行时需要额外的 CPU 运算以进行内容的压缩和解压缩。

大部分文件系统都是从它们的存储介质中直接挂装，RAM disk (initrd) 除外。RAM disk 上的文件系统必须首先从它们的存储介质上（如 Flash）解压缩到 RAM 中，然后才能够挂装到系统中。一般情况下，RAM disk 文件系统中的内容被制作成一个压缩格式的文件系统映像文件存放在存储介质上。

2.6.3 使用 NFS

Linux 内核可以从一个远程 NFS（网络文件系统）服务器上挂装根文件系统，这是非常重要的一个特性，它大大地加快了我们的开发速度，使我们避免了不断地更新目标机器中的文件系统内容和不断地重新启动目标机器。

通过 NFS 服务共享出去的文件系统目录可以被网络中的其它机器以 NFS 客户端的身份挂装到本地目录，然后当作本地文件系统一样来访问和使用。

我们把根文件系统内容所在的目录通过 NFS 服务共享出去，而目标机器把该共享目录通过 NFS 挂装到本地目录。当我们开发时在主机上更新了文件系统的内容之后，目标机器中的内容也相应地更新了，因为它们本来就是同一个东西，这样开发和调试就非常方便了。

1) 配置 NFS 服务器

一般来说，不需要专门一台机器用作 NFS 服务器，为我们使用的 Linux 主机加上 NFS 服务就可以了。如果您的 Linux 主机没有安装 NFS 服务器软件及相关工具的话，请从互联网上下载合适的版本并安装。

首先需要打开 NFS 服务。以 RedHat 9.0 为例子，运行 *setup* 命令，选择“System Services”，找到“nfs”项并选中。也可以运行如下命令来启动 NFS 服务：

```
root#/etc/init.d/portmap start
root#/etc/init.d/nfs start
```

然后我们需要修改 NFS 服务的配置文件 */etc/exports*，添加一个导出条目，每个条目包括一个共享目录、一个 IP 地址和访问许可选项：

```
/tftpboot/rootfs        *(rw,no_root_squash)
```

这里 */tftpboot/rootfs* 为开发主机上需要导出的根文件系统目录。*指示允许访问的 IP 地址不限，可以改为具体的 IP 地址；rw 指示允许客户端机器以读和写的许可来挂装文件系统；no_root_squash 选项允许客户端机器以主机上的 root 用户身份挂装文件系统。

如果对 */etc/exports* 文件做了修改，需要运行如下命令来重新启动 NFS 服务：

```
root#/etc/init.d/nfs restart
```

现在主机上的 NFS 服务就可以使用了。

2) 通过 NFS 挂装根文件系统

如果我们的目标机器需要通过 NFS 来挂装整个根文件系统的话，必须对内核进行适当的配置使之支持该功能。配置内核时我们必须选上“NFS file system support”和“Root file system on NFS”。注意，“Root file system on NFS”不能作为模块装载。

一般的引导装载器在启动内核时都可以传递相关的命令行参数给内核，使之根据这些信息能够正确地定位根文件系统的所在。

```
root=/dev/nfs
```

该参数告诉内核通过 NFS 来挂装根文件系统，而不是从某个存储设备上获取。*/dev/nfs* 并不是一个真正的设备，只是用来代表“Root file system on NFS”的一个同义词。

```
nfsroot=[<server-ip>:]<root-dir>[,<nfs-options>]
```

该参数指定根文件系统的位置和相关选项。如果没有指定 *nfsroot* 参数，将使用缺省的“*/tftpboot/%s*”。其中，

- **<server-ip>**：指定 NFS 服务器的 IP 地址。如果该参数没有给出的话，将使用“ip”变量的缺省值。通常情况下可以省略该项。
- **<root-dir>**：NFS 服务器上需要挂装为根文件系统的目录。如果该字符串中有“%s”符号，这个符号将被替换为目标机器的 IP 地址。
- **<nfs-options>**：标准的 NFS 选项，使用逗号分隔，一般情况下可以省略。

```
ip=<client-ip>:<server-ip>:<gw-ip>:<netmask>:<hostname>:<device>:<autoconf>
```

该参数告诉内核如何配置本机的 IP 地址和如何设置 IP 路由表。如果 ip 参数没有在命令行中给出的话，一般情况下内核将使用 RARP 和 BOOTP 协议来进行网络配置。其中，

- **<client-ip>**:目标机器的 IP 地址。如果没有指定的话将使用 RARP 和 BOOTP 协议来确定。
- **<server-ip>**：NFS 服务器的地址。一般情况下可以省略，通过 BOOTP 来确定。
- **<gw-ip>**:如果 NFS 服务器和目标机器不在同一个子网内的话，用来指定网关地址；如果该项为空则假定服务器在本地局域网内，或者通过 BOOTP 来确定。
- **<netmask>**：本网络的掩码。
- **<hostname>**：目标机器的主机名。如果为空可以通过 BOOTP 确定。
- **<device>**：所使用的网络设备名。一般情况下可以忽略。
- **<autconf>**：自动配置网络使用的方法。一般情况下可以忽略。

2.6.4 CRAMFS

CRAMFS 最初是 Linus Torvalds 编写的一个文件系统，具有简单、压缩和只读等特点。因为它非常简单，所以使用上有如下的限制：

- 文件的最大尺寸为 16MB，文件系统的最大尺寸是 256MB
- 没有当前目录节点（.）和父目录节点（..）
- 16 位的文件 UID 域和 8 位的 GID 域
- 所有的时间戳固定为 UNIX 纪元（00:00:00 GMT, January 1, 1970）
- 要求内核使用 4096 字节的内存页面大小（*PAGE_CACHE_SIZE* 的值为 4096），写文件系统的机器和读文件系统的机器必须具有同样的大小端（endian）。
- 所有文件的链接计数均固定为 1。

对于嵌入式系统来说，上面列举的限制都不会造成太大的影响。CRAMFS 文件系统是用于保存只读的根文件系统内容的一个很好的方案。

CRAMFS 主要的优点是将文件数据以压缩形式存储，在需要运行的时候进行解压缩。由于它存储的文件形式是压缩的格式，所以文件系统不能直接在 Flash 上运行。虽然这样可以节约很多 Flash 存储空间，但是文件系统运行需要将大量的数据拷贝进 RAM 中，造成一定的浪费。

创建 CRAMFS 根文件系统映像需要相关的文件系统工具 *mkcramfs* 和 *cramfsck*。*mkcramfs* 在一般的桌面 Linux 系统中都可以找到，如果没有的话我们可以从内核源代码编译和安装这些工具。这两个工具的源代码位于内核源代码的 *scripts/cramfs* 目录下，可以直接编译：

```
user$ cd ${KERNELDIR}/scripts/cramfs
user$ make
```

然后把生成的 *mkcramfs* 和 *cramfsck* 拷贝到系统执行路径中就可以使用了。

可以使用如下命令来制作一个 CRAMFS 映像，***${ROOTFS}***为目标根文件系统所在目录：

```
user$ mkcramfs ${ROOTFS}/ cramfs.img
```

得到的映像文件通常有 50%左右的压缩比率。

我们可以使用回环设备来查看一个已有 CRAMFS 映像的内容：

```
root# mount -o loop cramfs.img /mnt/cramfs
```

然后进入 */mnt/cramfs* 目录就可以和普通文件系统一样浏览内容了。注意：CRAMFS 是只读的文件系统，所以我们不能在 */mnt/cramfs* 目录下进行删除文件、修改文件和添加文件及目录等操作。上述命令要求主机系统的内核支持回环设备和 CRAMFS 文件系统。

【提示】回环设备可以用于查看包括 CRAMFS 和 Ext2 在内的各种类型的文件系统。

创建完 CRAMFS 映像之后，可以使用相关的工具把它写到 Flash 设备上，然后内核启动之后就可以使用其中的内容作为根文件系统了。

2.6.5 JFFS2

JFFS 文件系统是瑞典 Axis 通信公司开发的一种基于 Flash 的日志文件系统，它在设计时充分考虑了 Flash 的读写特性和电池供电的嵌入式系统的特点，系统可靠性高。RedHat 的 David WoodHouse 对 JFFS 进行改进后形成了 JFFS2。JFFS2 主要改善了存取策略以提高 Flash 的抗疲劳性，同时也优化了碎片整理性能，增加了数据压缩功能。需要注意的是，当文件系统已满或接近满时，JFFS2 会明显地放慢运行速度。

JFFS2 建立在 MTD 芯片驱动的基础之上，通过 MTD 的接口实现了文件系统的通用接口和功能。

2.6.6 ROMFS

ROMFS（*http://romfs.sourceforge.net/*）是 uClinux 操作系统使用最多的文件系统类型，它是一种简单、紧凑、存储空间效率高且只读的文件系统。ROMFS 顺序存储文件数据，并可以在 uClinux 支持的 Flash 存储设备上直接运行程序（XIP 运行方式），这样可以在系统运行时节省许多 RAM 空间。

ROMFS 最初是为 Linux 而设计的，现在也用到 eCos 等其它项目之中。目前的较新的 Linux 版本都支持 ROMFS。

类似 CRAMFS，ROMFS 文件系统对文件系统的许多特性进行了精简，使得内核中的 ROMFS 模块非常小，文件系统空间也得到了节省。

为了创建一个 ROMFS 类型的文件系统映像，我们需要一个名为 *genromfs*（*http://freshmeat.net/projects/genromfs/*）的工具程序。

2.6.7 使用 RAM disk

RAM disk （内存盘）是 Linux 内核支持的一种特殊的存储设备，它位于 RAM 中，但是可以当作一个磁盘设备分区来使用。显然，和真实磁盘设备不一样，RAM disk 中的内容在系统重启之后就会丢失。因为 RAM disk 模拟了一个块设备，所以任何用于磁盘设备的文件系统类型都可以用在 RAM disk 上面。内核可以同时支持多个 RAM disk，每个 RAM disk 都有同样的固定大小。

和 RAM disk 相关的内核配置选项有：*CONFIG_BLK_DEV_RAM*、*CONFIG_BLK_DEV_RAM_SIZE* 和 *CONFIG_BLK_DEV_INITRD*，分别指定 RAM disk 支持、RAM disk 的大小和 initrd（Initial RAM disk）支持。如果使用 ext2 文件系统的话，还要加上 *CONFIG_EXT2_FS*。

1) RAM disk 的用途

RAM disk 有什么用呢？在嵌入式 Linux 运行系统中，RAM disk 的主要用途是：

- 用作 initrd
- 存储系统的运行时数据和临时数据

RAM disk 的一个常见用途是用作内核启动时使用的 initrd（Initial RAM disk），也就是存放初始根文件系统内容的 RAM disk 映像压缩文件。initrd 机制是 Linux 内核的一个重要特性，它使得内核可以很容易地得到第一个可运行的根文件系统。

Linux 内核在启动时检查启动参数是否指定了 initrd 映像，如果是的话就从指定的存储介质上和位置找到该映像，如果是压缩格式的话把它解压缩，然后把内容拷贝到一个 RAM disk 中，再把该 RAM disk 挂装为根文件系统。

桌面 Linux 系统和 Linux 启动软盘中经常使用 initrd 的方式来启动系

统；内核使用 initrd 提供的文件系统内容先运行起来，然后再把某个磁盘设备分区上的正式文件系统挂装为根文件系统。

在嵌入式 Linux 系统中使用 initrd 方式的情况不多见，主要是一些无盘系统中用到。不过，在开发阶段 initrd 也是一个很方便的工具。例如，我们可以把文件系统内容制作成一个 initrd，把它下载到内存的指定位置，然后通过内核启动参数让内核在启动时从该映像中装载根文件系统。这样就可以在内存中方便地运行和调试系统，免去反复烧写 Flash 的痛苦。

除了用作 initrd 之外，RAM disk 是一个用来存储运行时数据特别是临时数据的理想场所。如果内存足够的话，我们可以把经常更新的运行时数据和临时文件存放到 RAM disk 中，这样能较大地提高系统的性能。如果我们使用的是一个只读类型的根文件系统，那么把运行时数据放到一个 RAM disk 文件系统中是一个很好的辅助方案。此外，在开发阶段我们也可以把经常需要更新的文件系统内容放到 RAM disk 中来使用，以避免经常性地更新 Flash。

2) 制作一个 RAM disk 映像

在 Linux 开发系统上制作一个 RAM disk 映像是非常简单的，步骤如下：

- 在一个目录中准备好文件系统的内容
- 把 RAM disk 文件的内容清空
- 在 RAM disk 中创建一个文件系统
- 把 RAM disk 挂装到一个指定目录
- 把准备好的文件系统内容拷贝到 RAM disk 挂装的目录下
- 卸载 RAM disk
- 制作压缩格式的文件系统映像文件

下面是一个简单的例子。在该例子中我们制作一个大小为 2MB 的 RAM disk 映像，文件系统类型为 ext2。

```
user$ su -m
password:

root# dd if=/dev/zero of=/dev/ram bs=1k count=2048
2048+0 records in
2048+0 records out

root# mke2fs -vm0 /dev/ram 2048
mke2fs 1.27 (8-Mar-2002)
Filesystem label=
OS type: Linux
```

```
Block size=1024 (log=0)
Fragment size=1024 (log=0)
256 inodes, 2048 blocks
0 blocks (0.00%) reserved for the super user
First data block=1
1 block group
8192 blocks per group, 8192 fragments per group
256 inodes per group

Writing inode tables: done
Writing superblocks and filesystem accounting information: done

This filesystem will be automatically checked every 26 mounts or
180 days, whichever comes first.  Use tune2fs -c or -i to override.

root# mount -t ext2 /dev/ram /mnt/ramdisk

root# cp -a ${yourfilesdir}/* /mnt/ramdisk

root# umount /mnt/ramdisk

root# dd if=/dev/ram of=ramdisk bs=1k count=2048
2048+0 records in
2048+0 records out

root# gzip ramdisk
```

第 1 条命令使用 Linux 的 *dd* 系统工具把 RAM disk 设备的前 2MB 内容清零。*dd* 是一个非常有用的文件拷贝和转换工具，可以操作普通文件和设备文件。*if=/dev/zero* 指定读入文件为系统设备文件*/dev/zero*，从该设备中读出的所有数据均为零；*of=/dev/ram* 指定输出文件为*/dev/ram*，*/dev/ram* 通常是一个符号链接，指向系统中的第一个 RAM disk 设备文件。这里 *bs=1k* 指定拷贝数据块的大小为 1k，*count=2048* 指定块的数量为 2048，总的大小就是 2MB。

第 2 条命令使用 *mke2fs* 系统工具在 RAM disk 中创建了一个大小为 2MB 的 ext2 类型的文件系统，*-m0* 选项指示 *mke2fs* 不为超级用户保留额外空间（缺省是 5%）。因为嵌入式 Linux 系统通常是单用户模式的，这样可以稍微地节省文件系统的空间。

第 3 条命令使用 *mount* 工具把 RAM disk 挂装到指定的 */mnt/ramdisk* 目录，*-t ext2* 选项指示 */dev/ram* 设备上的文件系统类型为 ext2。

第 4 条命令把准备好的文件系统内容拷贝到 RAM disk 中，这时 RAM disk 设备上的文件系统中就包含了所有的文件系统内容。

第 5 条命令把 RAM disk 设备卸载。

第 6 和第 7 条命令再次使用强大的 *dd* 工具，由 RAM disk 中的内容制作出一个压缩格式的映像文件。因为源内容来自于 RAM disk，所以这里用 *if=/dev/ram* 指定读入的文件为 RAM disk 设备文件。输出的映像文件为

ramdisk，由 *gzip* 处理成一个压缩文件 *ramdisk.gz*。

【提示】上述制作 RAM disk 映像的方法需要主机 Linux 开发系统的内核支持 RAM disk，并且所支持的 RAM disk 尺寸符合需求。mount、umount 和 mke2fs 等命令的执行需要超级用户身份。

下面是制作 RAM disk 映像的另外一种方法，该方法使用 Linux 的回环设备（loop back device）而不是 RAM disk 设备。

```
user$ su -m
password:
root# dd if=/dev/zero of=ramdisk bs=1k count=2048
root# mke2fs -F -vm0 ramdisk
root# mount -o loop ramdisk /mnt/ramdisk
root# cp -av ${yourfilesdir}/* /mnt/ramdisk
root# umount /mnt/ramdisk
root# gzip ramdisk
```

第 1 条命令使用 *dd* 工具制作一个指定大小的内容为零的普通文件，该文件将是放置文件系统内容的映像文件。

第 2 条命令使用 *mke2fs* 在 *ramdisk* 文件中创建一个 ext2 类型的文件系统。注意，*mke2fs* 工具既可以对设备文件进行操作，也可以对普通文件进行操作。*-F* 选项指定操作对象为普通文件，这样 *mke2fs* 就不会警告说该文件不是一个块设备并提示你确认是否继续。

第 3 条命令使用 *mount* 命令和*-o loop* 选项把 *ramdisk* 文件当作一个块设备挂装到 */mnt/ramdisk* 目录，对该目录下的文件系统的操作就是对 *ramdisk* 文件内容的操作。

第 4 条命令把准备好的文件系统内容拷贝到 */mnt/ramdisk* 目录下，因为挂装到该目录的是普通文件 *ramdisk*，因此，拷贝到这里的文件系统内容都写到了 *ramdisk* 文件中。也就是说，这时 *ramdisk* 文件中就包含了所有的文件系统内容。

第 5 和第 6 条命令的作用和前一种方法相同。

【提示】上述制作 RAM disk 映像的方法需要主机 Linux 开发系统的内核支持回环设备（loop back device）；流行的 Linux 发行版上默认安装的内核都支持该功能。

在制作完 RAM disk 映像之后，我们就可以把它放置到文件系统或者存储介质的适当位置。如果该映像是一个 initrd 的话，我们可能还需要对引导装载器进行适当的设置，使之能找到该映像。

3) 查看和更新 RAM disk 映像的内容

如果我们不是自己从头制作一个 RAM disk 映像，而是从别处（如互联网）得到一个可用的版本，如何查看该映像的内容并根据我们的需要进行修改和更新呢？

其实方法和在前面我们已经提到的一样，可以使用 RAM disk 设备或者回环设备把它挂装到一个目录下查看和修改。需要注意的是，开发系统的内核必须支持该映像上的文件系统类型。

相比而言使用回环设备是比较简单和直接的方法。首先我们使用 *gunzip* 命令把压缩格式的映像文件解压缩，然后使用回环方式把它直接挂装到一个指定目录下就可以查看和更新了。

```
root# gunzip ramdisk.gz
root# mount -o loop ramdisk /mnt/ramdisk
```

我们可以进入 */mnt/ramdisk* 目录查看该映像中文件系统的内容，还可以使用 *cp* 命令往该文件系统中添加新的目录和文件。不过要注意该映像上剩余空间的大小，如果新添内容的尺寸超过了剩余空间的大小，操作就会失败。

可以使用 *df* 命令来查看文件系统的剩余空间等使用情况，下面是查看一个刚创建的文件系统时显示的信息，其中 Available 项以 K 为单位指示有多少剩余空间：

```
root# df /mnt/ramdisk
Filesystem           1k-blocks      Used Available Use% Mounted on
/home/snig/ramdisk        2011        13      1998   1% /mnt/ramdisk
```

查看和更新 RAM disk 映像的另一种方法是使用 RAM disk 设备。首先需要使用 *dd* 命令把 *ramdisk* 映像的内容拷贝到*/dev/ram* 设备中，然后把 */dev/ram* 设备挂装到*/mnt/ramdisk* 目录，之后就可以查看其中的文件系统内容了。和前一种方法不同的是：*/dev/ram* 中的内容是 *ramdisk* 映像的拷贝，所以修改*/mnt/ramdisk* 目录下的文件系统内容不会影响原来的 *ramdisk*

文件；如果做了更新，我们还需要使用 *dd* 命令重新制作一个新的映像文件。

```
root# gunzip ramdisk.gz
root# dd if=ramdisk of=/dev/ram bs=1k count=2048
root# mount /dev/ram /mnt/ramdisk
```

同样，这里我们也要注意 RAM disk 的容量和剩余空间的大小。内核支持的 RAM disk 大小默认是 4MB。

4) 修改 RAM disk 的大小

我们经常会遇到这样一种情况：在刚拿到手的新开发板上面 Linux 内核已经可以正常启动，而且随开发板的软件包还提供了一个 initrd 映像；但是该 RAM disk 的容量只有 2MB，而我们需要往文件系统里面添加函数库和应用程序等许多内容，2MB 远远不够，怎么办呢？

假如一个 8MB 大小的文件系统才能够满足我们的需求（假设内存足够），那么根据之前讨论的内容我们可以很清楚地知道，需要如下的步骤和措施来解决这个问题：

- 首先确认开发板上面运行的内核支持的 RAM disk 容量（一般是 4MB），如果不够的话，需要重新配置、编译和安装新的内核，使之支持 8MB 大小的 RAM disk。
- 如果使用 RAM disk 设备的方法来制作 RAM disk 映像的话，需要确认开发系统上的内核支持 8MB 或以上大小的 RAM disk，否则要在主机上重新配置、编译和安装新的内核。
- 使用前面描述的两种方法之一重新制作一个 8MB 大小的 RAM disk 映像，新的映像包含了原来的 2MB 映像的文件系统内容。
- 把需要增加的新的目录和文件添加到新的映像中。

可以使用 *dmesg* 命令来查看内核支持的默认 RAM disk 大小：

```
user$ dmesg | grep RAMDISK
RAMDISK driver initialized: 16 RAM disks of 4096K size 1024 blocksize
RAMDISK: Compressed image found at block 0
```

因为内核可以通过启动时传递给它的启动参数来控制 RAM disk 的默认大小，所以，如果主机 Linux 系统使用的引导装载器是 GRUB 的话，我们可以使用在其配置文件 */etc/grub.conf* 的 *kernel* 段加上

ramdisk_size=8192 来指定默认 RAM disk 的大小为 8MB：

```
title Red Hat Linux (2.4.18-3)
        root (hd0,0)
        kernel /boot/vmlinuz-2.4.18-3 ro root=/dev/hda1 ramdisk_size=8192
        initrd /boot/initrd-2.4.18-3.img
```

修改完之后需要重新启动系统以使设置生效。

2.7 在 SkyEye 上运行 ARM Linux

SkyEye（*http://www.skyeye.org*）是一个开源软件项目，由清华大学的陈渝博士发起和组织，目前发展得很快。SkyEye 可以在 Linux 和 Windows 平台上模拟嵌入式硬件开发板，它基于 GNU gdb 对 ARM 系列的 CPU 进行了指令级的模拟，并且还提供了网络设备和 LCD 设备等外设的模拟，可以在 SkyEye 上面运行 Linux、uClinux 和 uCOS-II 等操作系统，并可对它们进行源码级的分析和测试。

SkyEye 是一个用来学习嵌入式开发的非常有用的工具，下面我们以 SkyEye 对 EP7312 的模拟为例子，具体讲述如何构建和运行嵌入式 Linux 系统。

首先我们要建立一个工作目录“*/opt/armlinux*”，所有的源代码和目标根文件系统都放在该目录，它是我们开展工作的主要场所：

```
user$ mkdir /opt/armlinux
```

注意，普通用户要具有对 */opt/armlinux* 目录的读写等访问权限。

2.7.1 安装 SkyEye

我们用的 SkyEye 版本是目前最新的 0.8.6，该版本支持 EP7312 上的 LCD 仿真，功能比较完善。

源代码包 *skyeye-0.8.6.tar.bz2* 在产品光盘的 *skyeye/* 目录下，也可从下面的网址下载：

http://gro.clinux.org/frs/?group_id=327&release_id=456

安装步骤如下：

```
user$ cd /opt/armlinux
user$ tar jxvf skyeye-0.8.6.tar.bz2
user$ cd skyeye
user$ ./configure --target=arm-elf --prefix=/usr/local
user$ make
user$ su -c 'make install'
```

可执行文件 *skyeye* 将被安装到 */usr/local/bin* 目录下，如果 */usr/local/bin* 在 *PATH* 环境变量中的话，直接在命令行下输入 *skyeye* 就可以启动 SkyEye 模拟器。

```
user$ skyeye
************************************************************
****                                                    ****
****   SkyEye  Simulator Ver 0.8.6 with  GDB 5.3 Interface ****
****                                                    ****
************************************************************
GNU gdb 5.3
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This SkyEye was configured as "--host=i686-pc-linux-gnu --target=arm-elf".
(SkyEye)
```

在 (SkyEye) 提示符下输入命令就可以装载和运行程序。

2.7.2 安装交叉编译工具链

我们所使用的 ARM 交叉编译工具链为 *arm-linux-cross-2.95.3.tar.bz2*（在产品光盘的 *armlinux/* 目录下）。安装方法如下：

```
root# cd /usr/local
root# mkdir arm
root# cd arm
root# tar jxvf <path to product cdrom>/armlinux/arm-linux-cross-2.95.3.tar.bz2
```

解开之后将在 */usr/local/arm* 目录下面生成一个名为 *2.95.3* 的目录，所包含的 ARM Linux 交叉编译器为 *2.95.3/bin* 目录下的 *arm-linux-gcc*。因此，我们需要在 PATH 环境变量中加上 */usr/local/arm/2.95.3/bin* 一项，使得 *arm-linux-gcc* 能够在命令行下直接执行。

该工具链也可以从下面网址下载：

http://www.lart.tudelft.nl/lartware/compile-tools/cross-2.95.3.tar.bz2

2.7.3 配置、编译和运行内核

我们使用的内核源代码包为 *linux-2.4.13-patched-for-EP7312.tar.bz2*（在产品光盘的 *armlinux/* 目录下），该版本是针对 ARM EP7312 的 Linux 内核版本。

在 */opt/armlinux* 目录下解压缩该文件，将生成一个 *linux-2.4.13* 目录：

```
user$ tar jxvf linux-2.4.13-patched-for-EP7312.tar.bz2
```

首先需要进行内核配置，在命令行下输入“ *make menuconfig* ”，进入内核的配置界面：

```
user$ cd linux-2.4.13
user$ make menuconfig
```

【提示】linux-2.4.13 目录下的 Makefile 文件中已经定义了 ARCH 和 CROSS_COMPILE 的值，分别为 *ARM* 和*/usr/local/arm/2.95.3/bin/arm-linux-*。

然后进入“ System Type ”子菜单，选择合适的处理器类型，如图 2-10 所示。

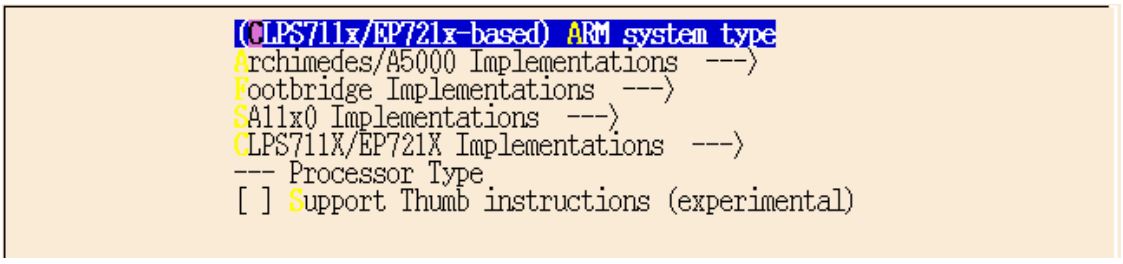

图 2-10 选择处理器类型

进入“ CLPS711X/EP721X Implementations ”菜单，选择“ EDB7312 ”，如图 2-11 所示。

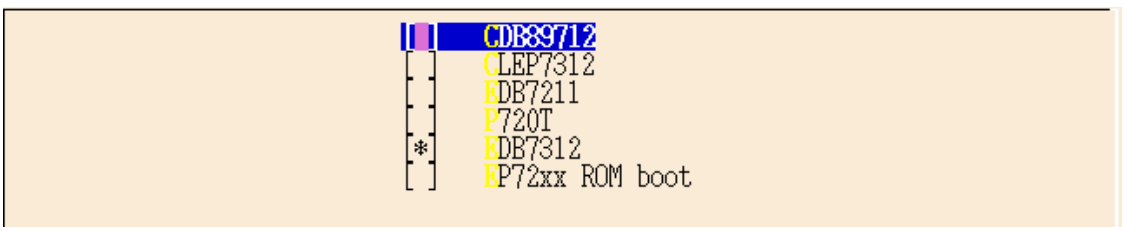

图 2-11 选择特定的处理器

进入“Block devices”子菜单，选择“RAM disk support”和“Intial RAM disk (initrd) support”，如图 2-12 所示。

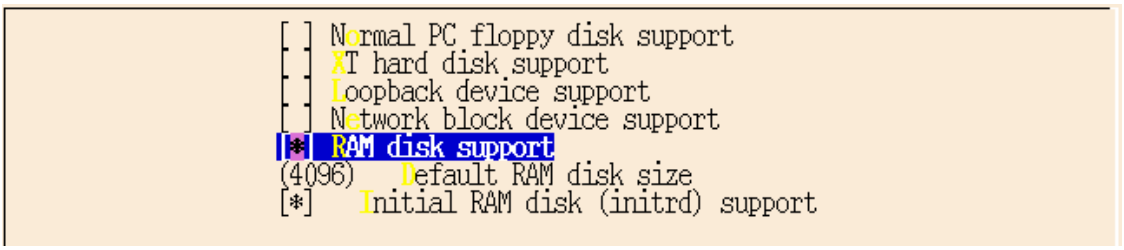

图 2-12 RAM disk 支持选项

我们将要使用 ROMFS 文件系统类型，所以，进入“Filesystems”子菜单，选择“ROM file system support”，如图 2-13 所示。

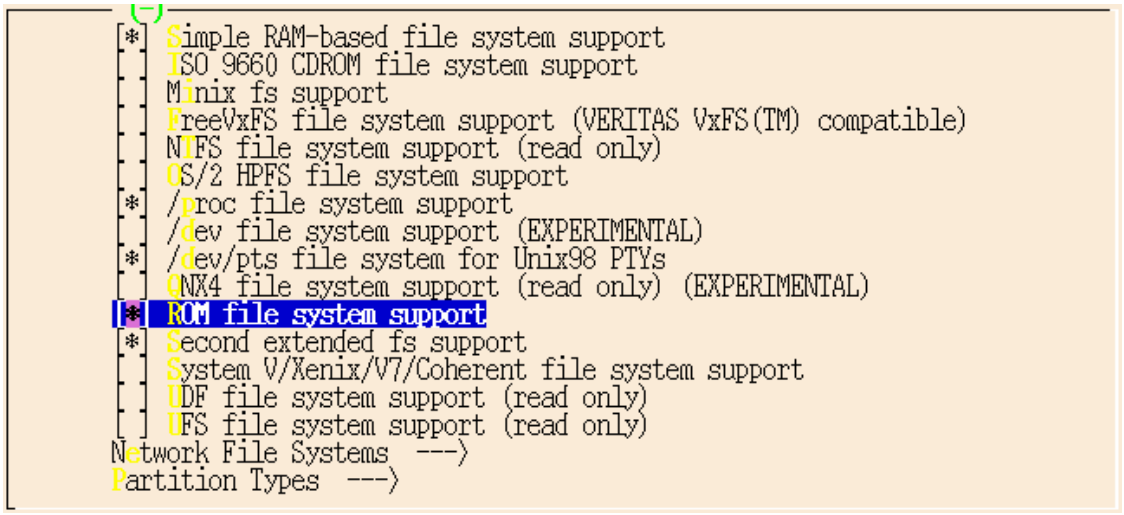

图 2-13 文件系统选项

配置完内核之后保存配置并退出配置界面，运行如下命令编译内核映像：

```
user$ make dep
user$ make zImage
```

生成的压缩格式映像为 *arch/arm/boot/compressed/vmlinux* 文件，未压缩的映像为内核根目录下的 *vmlinux* 文件。

现在我们可以在 SkyEye 上运行该内核了。

首先在*/opt/armlinux* 目录下建立一个用于保存运行时程序和数据的工作目录 *ep7312*：

```
user$ cd /opt/armlinux
user$ mkdir ep7312
user$ cd ep7312
```

然后把内核文件拷贝到该目录下：

```
user$ cp /opt/armlinux/linux-2.4.13/vmlinux . -f
```

在运行 SkyEye 之前我们还需要一个针对 EP7312 的 SkyEye 配置文件 *skyeye.conf*。在 */opt/armlinux/ep7312* 下面新建一个 *skyeye.conf* 文件，添加如下的内容：

```
#skyeye config file for ep7312
cpu: arm720t
mach: ep7312
mem_bank: map=I, type=RW, addr=0x80000000, size=0x00010000
mem_bank: map=M, type=RW, addr=0xc0000000, size=0x00200000
mem_bank: map=M, type=R,  addr=0xC0000,    size=0x340000
mem_bank: map=M, type=RW, addr=0xc0600000, size=0x00c00000
```

skyeye.conf 文件中各配置选项的含义如下：

- cpu：基本 CPU 核配置选项，arm720t 表示 SkyEye 所模拟的处理器类型是 ARM720T
- mach：具体的开发板(包括 CPU 扩展)配置选项，ep7312 表示特定处理器是 EP7312
- mem_bank：内存组配置选项

各配置选项的详细介绍请参考 IBM DeveloperWorks 上的 SkyEye 文章：

http://www.ibm.com/developerworks/cn/linux/l-skyeye/part2/index.shtml

然后我们运行“ *skyeye vmlinux* ”命令来装载内核：

```
user$ skyeye vmlinux
************************************************************
****                                                    ****
****   SkyEye  Simulator Ver 0.8.6 with  GDB 5.3 Interface ****
****                                                    ****
************************************************************
GNU gdb 5.3
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
```

```
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This SkyEye was configured as "--host=i686-pc-linux-gnu --target=arm-elf".
(SkyEye)
```

在 (skyeye) 提示符下输入如下命令来运行内核：

```
(SkyEye) target sim
cpu info: armv4, arm720t, 41807200, ffffff00, 1
mach info: name ep7312, mach_init addr 0x8141610
SKYEYE: use arm7100 mmu ops
Connected to the simulator.

(SkyEye) load
Loading section .init, size 0xc000 vma 0xc0028000
Loading section .text, size 0xd869c vma 0xc0034000
Loading section __ex_table, size 0x7b8 vma 0xc010c6a0
Loading section .data, size 0x985b vma 0xc010e000
Start address 0xc0028000
Transfer rate: 7812472 bits in <1 sec.

(SkyEye) run
Starting program: /opt/armlinux/ep7312/vmlinux
Linux version 2.4.13-ac4-rmk1 (snig@snig.lan.minigui.com) (gcc version 2.95.3 20010315 (relea
se)) #18 19:29:45 CST 2005
Processor: ARM ARM720T revision 0
Architecture: Cirrus Logic EDB7312 (EP7312 evaluation board)
Warning: bad configuration page, trying to continue
On node 0 totalpages: 4096
zone(0): 4096 pages.
zone(1): 0 pages.
zone(2): 0 pages.
Kernel command line: root=/dev/ram0 rw initrd=0xc0400000,0x00200000
Calibrating delay loop... 26.00 BogoMIPS
Memory: 16MB = 16MB total
Memory: 14820KB available (865K code, 209K data, 48K init)
Dentry-cache hash table entries: 2048 (order: 2, 16384 bytes)
Inode-cache hash table entries: 1024 (order: 1, 8192 bytes)
Mount-cache hash table entries: 512 (order: 0, 4096 bytes)
Buffer-cache hash table entries: 1024 (order: 0, 4096 bytes)
Page-cache hash table entries: 4096 (order: 2, 16384 bytes)
POSIX conformance testing by UNIFIX
Linux NET4.0 for Linux 2.4
Based upon Swansea University Computer Society NET3.039
Starting kswapd v1.8
pty: 256 Unix98 ptys configured
block: queued sectors max/low 9757kB/3252kB, 64 slots per queue
RAMDISK driver initialized: 16 RAM disks of 4096K size 1024 blocksize
Blkmem copyright 1998,1999 D. Jeff Dionne
Blkmem copyright 1998 Kenneth Albanowski
Blkmem 1 disk images:
0: FB0C0000-FB0BFFFF [VIRTUAL FB0C0000-FB0BFFFF] (RO)
NET4: Linux TCP/IP 1.0 for NET4.0
IP Protocols: ICMP, UDP, TCP
IP: routing cache hash table of 512 buckets, 4Kbytes
TCP: Hash tables configured (established 1024 bind 1024)
NET4: Unix domain sockets 1.0/SMP for Linux NET4.0.
NetWinder Floating Point Emulator V0.95 (c) 1998-1999 Rebel.com
Blkmem: bad access: block=2, count=2 (pos=800, len=0)
end_request: I/O error, dev 1f:00 (Blkmem), sector 2
EXT2-fs: unable to read superblock
Blkmem: bad access: block=0, count=2 (pos=400, len=0)
end_request: I/O error, dev 1f:00 (Blkmem), sector 0
romfs: unable to read superblock
Kernel panic: VFS: Unable to mount root fs on 1f:00
```

从上面的运行信息可以看出，内核已经成功地运行起来。不过由于我们还没有制作和指定 initrd，所以内核挂装根文件系统失败，显示如下的信息：

```
Kernel panic: VFS: Unable to mount root fs on 1f:00
```

当我们下一步把根文件系统准备好之后，这个问题就不成问题了。首先在工作目录下为根文件系统准备一个目录：

```
user$ mkdir /opt/armlinux/ep7312/rootfs
```

2.7.4 构建根文件系统

我们所使用的根文件系统系统程序方案是 BusyBox，所使用的版本是目前最新 *busybox-1.00.tar.bz2*。该软件包也可以从下面的网址下载：

http://www.busybox.net

在 */opt/armlinux* 目录下解压缩该软件包：

```
user$ tar jxvf busybox-1.00.tar.bz2
user$ cd busybox-1.00
```

然后执行“ *make menuconfig* ”命令对 BusyBox 进行配置，见图 2-14。

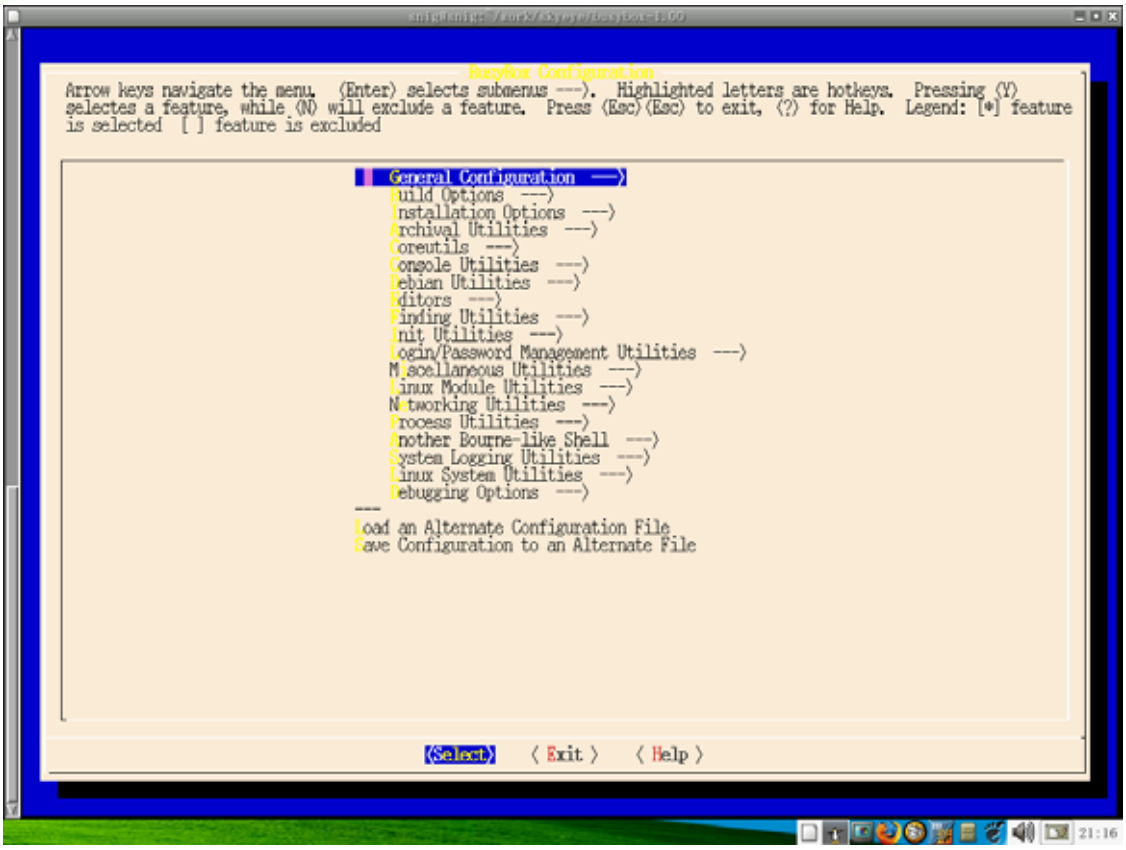

图 2-14 配置 busybox

进入“Build Options”子菜单，按图 2-15 配置。

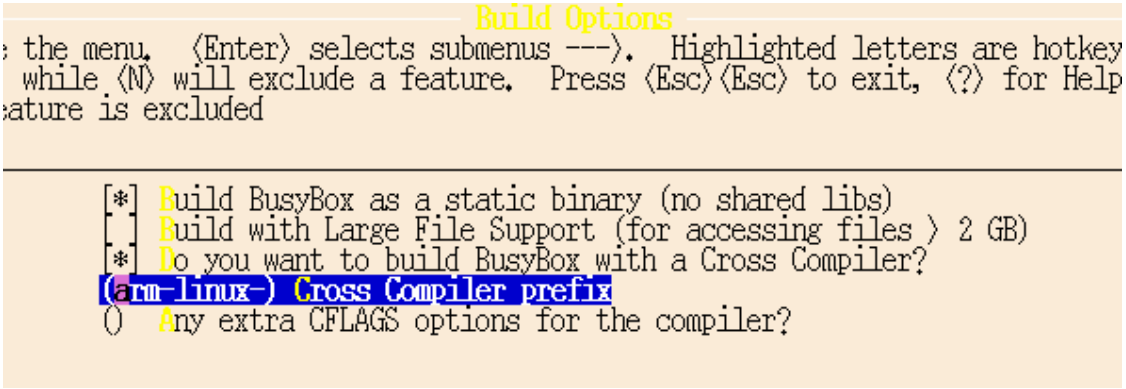

图 2-15 busybox 的构建配置

上述配置将 BusyBox 编译为一个静态可执行程序（不需要共享库），这样运行比较方便。

选择使用交叉编译器，把编译器的前缀设置为“arm-linux-”，相当于指定编译器为 *arm-linux-gcc*。

然后就是各工具程序的选择和配置。这些选项基本上可以保持不变，

不过为了减小 busybox 程序的体积，我们去掉了一些没有太大用处或者目前用不着的工具（注意需取消网络工具中的 *route* 工具），具体配置可参考产品光盘中 *armlinux/ep7312* 目录下的 *busybox-config* 文件。

配置完之后运行下面的命令来编译 BusyBox：

```
user$ make dep
user$ make
```

如果编译正确无误，将在当前目录下生成一个名为 *busybox* 的可执行文件。使用 *file* 命令查看一下，确定它是 ARM 版本的程序。

```
user$ file busybox
busybox: ELF 32-bit LSB executable, ARM, version 1 (ARM), for GNU/Linux 2.0.0, statically lin
ked, stripped
```

接着使用下面的命令就可将 BusyBox 的工具程序安装到指定的目标根文件系统目录下：

```
user$ make PREFIX=/opt/armlinux/ep7312/rootfs install
```

进入目标根文件系统目录看一下，应该已经有了 *bin*、*sbin*、*usr/bin* 和 *usr/sbin* 等系统目录，目录下有各种工具程序。

BusyBox 的默认安装目录是当前目录下的 *_install* 目录，我们也可以先把工具程序安装到该目录，然后拷贝到目标根文件系统中。

接下来我们要在目标根文件系统中创建常用的一些设备文件节点。首先创建 */dev* 目录，然后在该目录下使用 *mknod* 命令创建如下的设备：

```
crw-------    1 snig     snig       5,   1 1970-01-01  console
crw-------    1 snig     snig       1,   2 1970-01-01  kmem
crw-------    1 snig     snig       1,   1 1970-01-01  mem
crw-------    1 snig     snig       1,   3 1970-01-01  null
brw-------    1 snig     snig       1,   0 1970-01-01  ram0
crw-------    1 snig     snig       1,   8 1970-01-01  random
crw-------    1 snig     snig       5,   0 1970-01-01  tty
crw-------    1 snig     snig       4,   0 1970-01-01  tty0
crw-------    1 snig     snig       4,   1 1970-01-01  tty1
crw-------    1 snig     snig     204,  16 1970-01-01  ttyAM0
crw-------    1 snig     snig     204,  17 1970-01-01  ttyAM1
crw-------    1 snig     snig       1,   9 1970-01-01  urandom
crw-------    1 snig     snig       1,   5 1970-01-01  zero
```

创建完设备节点之后我们需要在 */etc* 目录下创建一些系统运行所需的

配置文件，包括 *inittab* 和 *fstab* 等。

```
user$ cd /opt/armlinux/ep7312/rootfs
user$ mkdir etc
```

因为我们的系统 *init* 程序使用的是 BusyBox 的 init，所以我们直接使用 BusyBox 提供的示例 *inittab* 文件即可。

示例 *inittab* 文件为 */opt/armlinux/busybox-1.00/examples/inittab*，其内容如下：

```
# /etc/inittab init(8) configuration for BusyBox
#
# Copyright (C) 1999-2004 by Erik Andersen <andersen@codepoet.org>
#
#
# Note, BusyBox init doesn't support runlevels.  The runlevels field is
# completely ignored by BusyBox init. If you want runlevels, use sysvinit.
#
#
# Format for each entry: <id>:<runlevels>:<action>:<process>
#
# <id>: WARNING: This field has a non-traditional meaning for BusyBox init!
#
#  The id field is used by BusyBox init to specify the controlling tty for
#  the specified process to run on.  The contents of this field are
#  appended to "/dev/" and used as-is.  There is no need for this field to
#  be unique, although if it isn't you may have strange results.  If this
#  field is left blank, it is completely ignored.  Also note that if
#  BusyBox detects that a serial console is in use, then all entries
#  containing non-empty id fields will _not_ be run.  BusyBox init does
#  nothing with utmp.  We don't need no stinkin' utmp.
#
# <runlevels>: The runlevels field is completely ignored.
#
# <action>: Valid actions include: sysinit, respawn, askfirst, wait, once,
#                                  restart, ctrlaltdel, and shutdown.
#
#       Note: askfirst acts just like respawn, but before running the specified
#       process it displays the line "Please press Enter to activate this
#       console." and then waits for the user to press enter before starting
#       the specified process.
#
#       Note: unrecognised actions (like initdefault) will cause init to emit
#       an error message, and then go along with its business.
#
# <process>: Specifies the process to be executed and it's command line.
#
# Note: BusyBox init works just fine without an inittab. If no inittab is
# found, it has the following default behavior:
#         ::sysinit:/etc/init.d/rcS
#         ::askfirst:/bin/sh
#         ::ctrlaltdel:/sbin/reboot
#         ::shutdown:/sbin/swapoff -a
#         ::shutdown:/bin/umount -a -r
#         ::restart:/sbin/init
#
# if it detects that /dev/console is _not_ a serial console, it will
# also run:
#         tty2::askfirst:/bin/sh
#         tty3::askfirst:/bin/sh
#         tty4::askfirst:/bin/sh
#
# Boot-time system configuration/initialization script.
# This is run first except when booting in single-user mode.
#
::sysinit:/etc/init.d/rcS
```

```
# /bin/sh invocations on selected ttys
#
# Note below that we prefix the shell commands with a "-" to indicate to the
# shell that it is supposed to be a login shell.  Normally this is handled by
# login, but since we are bypassing login in this case, BusyBox lets you do
# this yourself...
#
# Start an "askfirst" shell on the console (whatever that may be)
::askfirst:-/bin/sh
# Start an "askfirst" shell on /dev/tty2-4
tty2::askfirst:-/bin/sh
tty3::askfirst:-/bin/sh
tty4::askfirst:-/bin/sh

# /sbin/getty invocations for selected ttys
tty4::respawn:/sbin/getty 38400 tty5
tty5::respawn:/sbin/getty 38400 tty6

# Example of how to put a getty on a serial line (for a terminal)
#::respawn:/sbin/getty -L ttyS0 9600 vt100
#::respawn:/sbin/getty -L ttyS1 9600 vt100
#
# Example how to put a getty on a modem line.
#::respawn:/sbin/getty 57600 ttyS2

# Stuff to do when restarting the init process
::restart:/sbin/init

# Stuff to do before rebooting
::ctrlaltdel:/sbin/reboot
::shutdown:/bin/umount -a -r
::shutdown:/sbin/swapoff -a
```

把该文件拷贝到目标根文件系统的 *etc* 目录下：

```
user$ cp /opt/armlinux/busybox-1.00/examples/inittab /opt/armlinux/ep7312/rootfs/etc/
```

从 *inittab* 文件可以看到，BusyBox 的 *init* 程序自动启动 */etc/init.d* 目录下的名为 *rcS* 的脚本。因此，我们要在 ***${ROOTFS}****/etc* 目录下创建一个 *init.d* 目录：

```
user$ cd /opt/armlinux/ep7312/rootfs/etc/
user$ mkdir init.rd
user$ cd init.rd
```

然后在 *init.d* 目录下创建一个系统启动脚本 *rcS*，并使用 *chmod +x* 命令使其具有可执行属性。内容如下：

```
#!/bin/sh
hostname ARMLinux
mount -t proc proc /proc

cat /etc/motd
```

该脚本所做的工作很简单，就是挂装 *proc* 文件系统，并显示*/etc/motd* 文件中的系统欢迎信息。

我们还需要在 ***${ROOFS}****/etc* 目录下创建一个文件系统的配置文件

fstab，内容如下：

```
# /etc/fstab: static file system information.
#
# <file system><mount point>   <type>  <options>           <dump>  <pass>
proc       /proc       proc    defaults            0   0
```

到目前为止，一个可以运行的根文件系统的内容就要准备好了，我们再在*${ROOTFS}*目录下加上 *lib* 和 *var* 等一些有用的系统目录：

```
user$ cd /opt/armlinux/ep7312/rootfs
user$ mkdir lib var
user$ ln -s /var/tmp tmp
```

我们得到的目标根文件系统的目录结构如下：

```
bin  dev  etc  lib  linuxrc  proc  sbin  tmp  usr  var
```

接下来使用 *genromfs* 工具制作 ROMFS 类型的目标根文件系统映像：

```
user$ cd /opt/armlinux/ep7312/
user$ genromfs -d roofs -f initrd.romfs
```

有了目标根文件系统映像之后，我们需要修改 *skyeye.conf* 文件中的相应选项使 SkyEye 装载指定的映像文件。修改后的 *skyeye.conf* 内容如下：

```
#skyeye config file for ep7312
cpu: arm720t
mach: ep7312
mem_bank: map=I, type=RW, addr=0x80000000, size=0x00010000
mem_bank: map=M, type=RW, addr=0xc0000000, size=0x00200000
mem_bank: map=M, type=R,  addr=0xC0000,    size=0x340000, file=./initrd.romfs
mem_bank: map=M, type=RW, addr=0xc0600000, size=0x00c00000
```

现在，让我们在 */opt/armlinux/ep7312* 目录下再次运行 SkyEye：

```
user$ skyeye vmlinux
**************************************************************
****                                                      ****
****   SkyEye  Simulator Ver 0.8.6 with  GDB 5.3 Interface ****
****                                                      ****
**************************************************************
GNU gdb 5.3
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This SkyEye was configured as "--host=i686-pc-linux-gnu --target=arm-elf"...

(SkyEye) target sim
```

```
cpu info: armv4, arm720t, 41807200, ffffff00, 1
mach info: name ep7312, mach_init addr 0x8141610
SKYEYE: use arm7100 mmu ops
Loaded ROM ./initrd.romfs
Connected to the simulator.

(SkyEye) load
Loading section .init, size 0xc000 vma 0xc0028000
Loading section .text, size 0xd869c vma 0xc0034000
Loading section __ex_table, size 0x7b8 vma 0xc010c6a0
Loading section .data, size 0x985b vma 0xc010e000
Start address 0xc0028000
Transfer rate: 7812472 bits in <1 sec.

(SkyEye) run
Starting program: /opt/armlinux/ep7312/vmlinux
Linux version 2.4.13-ac4-rmk1 (snig@snig.lan.minigui.com) (gcc version 2.95.3 20010315 (relea
se)) #18 12 19:29:45 CST 2005
Processor: ARM ARM720T revision 0
Architecture: Cirrus Logic EDB7312 (EP7312 evaluation board)
Warning: bad configuration page, trying to continue
On node 0 totalpages: 4096
zone(0): 4096 pages.
zone(1): 0 pages.
zone(2): 0 pages.
Kernel command line: root=/dev/ram0 rw initrd=0xc0400000,0x00200000
Calibrating delay loop... 26.00 BogoMIPS
Memory: 16MB = 16MB total
Memory: 14820KB available (865K code, 209K data, 48K init)
Dentry-cache hash table entries: 2048 (order: 2, 16384 bytes)
Inode-cache hash table entries: 1024 (order: 1, 8192 bytes)
Mount-cache hash table entries: 512 (order: 0, 4096 bytes)
Buffer-cache hash table entries: 1024 (order: 0, 4096 bytes)
Page-cache hash table entries: 4096 (order: 2, 16384 bytes)
POSIX conformance testing by UNIFIX
Linux NET4.0 for Linux 2.4
Based upon Swansea University Computer Society NET3.039
Starting kswapd v1.8
pty: 256 Unix98 ptys configured
block: queued sectors max/low 9757kB/3252kB, 64 slots per queue
RAMDISK driver initialized: 16 RAM disks of 4096K size 1024 blocksize
Blkmem copyright 1998,1999 D. Jeff Dionne
Blkmem copyright 1998 Kenneth Albanowski
Blkmem 1 disk images:
0: FB0C0000-FB171FFF [VIRTUAL FB0C0000-FB171FFF] (RO)
NET4: Linux TCP/IP 1.0 for NET4.0
IP Protocols: ICMP, UDP, TCP
IP: routing cache hash table of 512 buckets, 4Kbytes
TCP: Hash tables configured (established 1024 bind 1024)
NET4: Unix domain sockets 1.0/SMP for Linux NET4.0.
NetWinder Floating Point Emulator V0.95 (c) 1998-1999 Rebel.com
VFS: Mounted root (romfs filesystem).
Freeing init memory: 48K
Welcome to
 _     _____      __   __    _     _
/ \   /  __ \    /  \_/  \  | |   |_|
| _ | | |  | |  / /\   /\ \ | |    _ ____  _   _  _  _
| /_\ | |__| | / /  \_/  \ \| |   | |  _ \| | | |\ \/ /
|/   \|_____/ /_/       |_|| |___ | | |_| | |_| |/    \
                           |_||_____||_|_| |_|\____|\_/\_/

ARMLinux for Skyeye
For further information check:
http://hpclab.cs.tsinghua.edu.cn/~skyeye/


Please press Enter to activate this console.
```

可以看到根文件系统已经被正确地挂装，*init* 程序也已经执行，*rcS* 脚本显示的欢迎信息也显示在屏幕上。

我们按照提示按一下回车键进入 BusyBox 的 Shell 环境：

```
BusyBox v1.00 (2005.03.12-13:23+0000) Built-in shell (ash)
Enter 'help' for a list of built-in commands.
```

```
-sh: can't access tty; job control turned off
/ #
```

在 Shell 提示符下输入 *ls –l* 命令查看一下目标机上根文件系统的结构：

```
/ # ls -l
drwxr-xr-x    1 0        0              32 Jan  1 00:00 bin
drwxr-xr-x    1 0        0              32 Jan  1 00:00 dev
drwxr-xr-x    1 0        0              32 Jan  1 00:00 etc
drwxr-xr-x    1 0        0              32 Jan  1 00:00 lib
lrwxrwxrwx    1 0        0              11 Jan  1 00:00 linuxrc -> bin/busybox
drwxr-xr-x    1 0        0              32 Jan  1 00:00 proc
drwxr-xr-x    1 0        0              32 Jan  1 00:00 sbin
lrwxrwxrwx    1 0        0               8 Jan  1 00:00 tmp -> /var/tmp
drwxr-xr-x    1 0        0              32 Jan  1 00:00 usr
drwxr-xr-x    1 0        0              32 Jan  1 00:00 var
/ #
```

再到 */bin* 目录和 */sbin* 目录下看一下：

```
/ # cd /bin
/bin # ls -l
lrwxrwxrwx    1 0        0               7 Jan  1 00:00 ash -> busybox
-rwxr-xr-x    1 0        0          710232 Jan  1 00:00 busybox
lrwxrwxrwx    1 0        0               7 Jan  1 00:00 cat -> busybox
lrwxrwxrwx    1 0        0               7 Jan  1 00:00 chgrp -> busybox
lrwxrwxrwx    1 0        0               7 Jan  1 00:00 chmod -> busybox
lrwxrwxrwx    1 0        0               7 Jan  1 00:00 chown -> busybox
lrwxrwxrwx    1 0        0               7 Jan  1 00:00 cp -> busybox
lrwxrwxrwx    1 0        0               7 Jan  1 00:00 date -> busybox
lrwxrwxrwx    1 0        0               7 Jan  1 00:00 dd -> busybox
lrwxrwxrwx    1 0        0               7 Jan  1 00:00 df -> busybox
lrwxrwxrwx    1 0        0               7 Jan  1 00:00 dmesg -> busybox
lrwxrwxrwx    1 0        0               7 Jan  1 00:00 echo -> busybox
lrwxrwxrwx    1 0        0               7 Jan  1 00:00 false -> busybox
lrwxrwxrwx    1 0        0               7 Jan  1 00:00 hostname -> busybox
lrwxrwxrwx    1 0        0               7 Jan  1 00:00 kill -> busybox
lrwxrwxrwx    1 0        0               7 Jan  1 00:00 ln -> busybox
lrwxrwxrwx    1 0        0               7 Jan  1 00:00 ls -> busybox
lrwxrwxrwx    1 0        0               7 Jan  1 00:00 mkdir -> busybox
lrwxrwxrwx    1 0        0               7 Jan  1 00:00 mknod -> busybox
lrwxrwxrwx    1 0        0               7 Jan  1 00:00 more -> busybox
lrwxrwxrwx    1 0        0               7 Jan  1 00:00 mount -> busybox
lrwxrwxrwx    1 0        0               7 Jan  1 00:00 mv -> busybox
lrwxrwxrwx    1 0        0               7 Jan  1 00:00 pidof -> busybox
lrwxrwxrwx    1 0        0               7 Jan  1 00:00 ping -> busybox
lrwxrwxrwx    1 0        0               7 Jan  1 00:00 ps -> busybox
lrwxrwxrwx    1 0        0               7 Jan  1 00:00 pwd -> busybox
lrwxrwxrwx    1 0        0               7 Jan  1 00:00 rm -> busybox
lrwxrwxrwx    1 0        0               7 Jan  1 00:00 rmdir -> busybox
lrwxrwxrwx    1 0        0               7 Jan  1 00:00 sh -> busybox
lrwxrwxrwx    1 0        0               7 Jan  1 00:00 sleep -> busybox
lrwxrwxrwx    1 0        0               7 Jan  1 00:00 sync -> busybox
lrwxrwxrwx    1 0        0               7 Jan  1 00:00 touch -> busybox
lrwxrwxrwx    1 0        0               7 Jan  1 00:00 true -> busybox
lrwxrwxrwx    1 0        0               7 Jan  1 00:00 umount -> busybox
lrwxrwxrwx    1 0        0               7 Jan  1 00:00 uname -> busybox
lrwxrwxrwx    1 0        0               7 Jan  1 00:00 usleep -> busybox
lrwxrwxrwx    1 0        0               7 Jan  1 00:00 vi -> busybox

/ # cd /sbin/
/sbin # ls -l
lrwxrwxrwx    1 0        0              14 Jan  1 00:00 halt -> ../bin/busybox
lrwxrwxrwx    1 0        0              14 Jan  1 00:00 ifconfig -> ../bin/busybox
lrwxrwxrwx    1 0        0              14 Jan  1 00:00 init -> ../bin/busybox
lrwxrwxrwx    1 0        0              14 Jan  1 00:00 klogd -> ../bin/busybox
lrwxrwxrwx    1 0        0              14 Jan  1 00:00 pivot_root -> ../bin/busybox
lrwxrwxrwx    1 0        0              14 Jan  1 00:00 poweroff -> ../bin/busybox
lrwxrwxrwx    1 0        0              14 Jan  1 00:00 reboot -> ../bin/busybox
lrwxrwxrwx    1 0        0              14 Jan  1 00:00 swapoff -> ../bin/busybox
```

```
lrwxrwxrwx    1 0        0              14 Jan  1 00:00 swapon -> ../bin/busybox
lrwxrwxrwx    1 0        0              14 Jan  1 00:00 syslogd -> ../bin/busybox
/sbin #
```

可以看到 */bin* 和 */sbin* 目录下的程序都是指向 *busybox* 程序的符号链接，*/usr/bin/* 和 */usr/sbin* 下的程序也是如此。所有这些程序的功能都包含在一个 700KB 大小的 *busybox* 程序中了，真是无愧“瑞士军刀”的称号。

到目前为止我们已经构建了一个可以在 SkyEye EP7312 模拟器上正确运行的 ARM Linux 内核和 ROMFS 根文件系统。在第 3 章和第 4 章的应用程序开发中，我们还将会对内核和文件系统的内容做适当修改，以便运行我们自己的示例程序。

2.8 在 Xcopilot 上运行 uClinux

uClinux（*http://www.uclinux.org*）操作系统是Linux针对没有内存管理单元（MMU）的微处理器/微控制器的衍生版本，没有MMU支持是uClinux与Linux的主要差异。uClinux继承了Linux的绝大部分特性，Linux上的应用程序不需要做任何改变或者做很少的改变就可以移植到uClinux之上。

由于 uClinux 具有体积小、功能强和性能好等优点以及开放源码的特性，在嵌入式系统开发中得到了广泛的应用。当然，由于没有 MMU 支持，uClinux 也有一些限制，例如进程没有独立的地址空间，内存空间没有保护，对共享库的支持不够等等。

Xcopilot 是一个掌上设备模拟器，模拟的是 m68k 的 CPU。uClinux 发行包中包括了对该模拟器的支持。

首先我们要建立一个工作目录“*/opt/uclinux*”，所有的源代码和目标根文件系统都放在该目录，它是我们开展工作的主要场所：

```
user$ mkdir /opt/uclinux
```

注意，普通用户要具有 */opt/uclinux* 目录的读写等访问权限。

2.8.1 安装 Xcopilot

把 Xcopilot 模拟器源代码包解开：

```
user$ cd /opt/uclinux
user$ tar zxvf <path to cdrom>/xcopilot/xcopilot-0.6.6-uc0.tar.gz
```

编译 Xcopilot：

```
user$ cd xcopilot-0.6.6-uc0
user$ ./configure
user$ make
```

上述命令将在当前目录下生成一个 *xcopilot* 可执行文件，不过因为没有内核及文件系统映像，所以运行起来只是一个 PDA 的界面。

2.8.2 安装交叉编译工具链

我们所使用的 m68k 交叉编译工具链为 *m68k-elf-tools-20030314.sh*，保存在产品光盘的 *uclinux* 目录中。该工具链的安装方法如下：

```
root# sh <path to cdrom>/m68k-elf/m68k-elf-tools-20030314.sh
```

相关的工具和文件将安装到 */usr/local* 目录下，m68k 编译器的文件名为 *m68k-elf-gcc*。

2.8.3 配置、编译和运行 uClinux

首先，您需要有一个 uClinux 发行版，产品光盘的 *uclinux* 目录中包括了一个 uClinux 发行版 *uClinux-dist-20030522.tar.gz*，该发行版也可从 uClinux 网站下载。把这个软件包在工作目录 */opt/uclinux* 下解开：

```
user$ tar zxvf uClinux-dist-20030522.tar.gz
```

该命令将把 uClinux 发行版目录树解开到一个名为 *uClinux-dist* 的目录下。进入该源代码目录：

```
user$ cd uClinux-dist
```

然后进行 uClinux 内核和应用程序的配置：

```
user$ make menuconfig
```

运行该命令将进入 uClinux-dist 的图形配置界面，如图 2-16 所示。

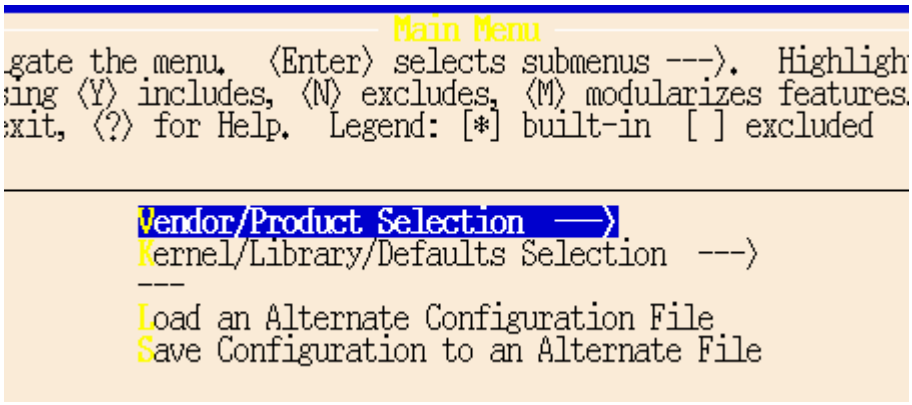

图 2-16 uClinux 配置界面

这里我们在“Vendor/Product”选项中选择目标平台为“3com/Xcopilot”。“Kernel”选为“Linux-2.4.x”，“libc”选“uClibc”。然后选中“Customize Kernel Settings”选项，以便在退出该配置后立即进行内核的定制配置。

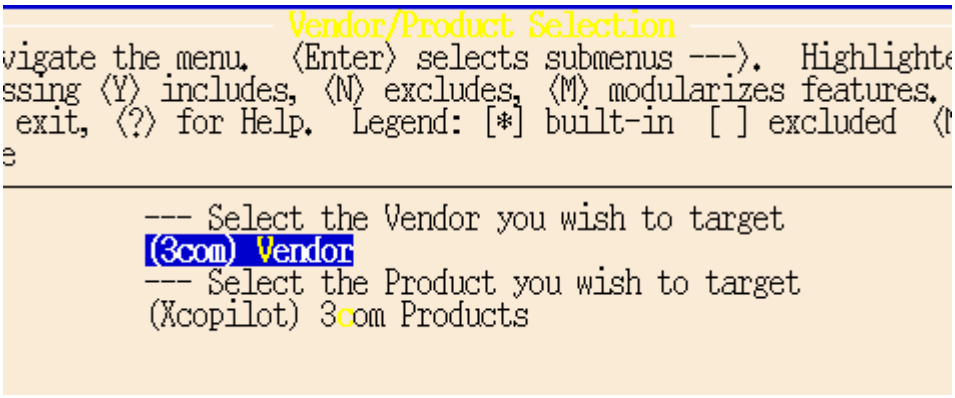

图 2-17 选择目标平台

然后我们需要确定内核支持的文件系统的类型，这里选择 ROMFS 和 RAM disk 上的 EXT2。ROMFS 文件系统用来保存只读的数据，可写的数据保存在 RAM disk 设备上的 EXT2 文件系统中。

进入“Block devices”菜单，选择“RAM disk support”和“ROM disk memory block device(blkmem)”，blkmem 是 Flash

上的 ROMFS 文件系统所需的块设备驱动。

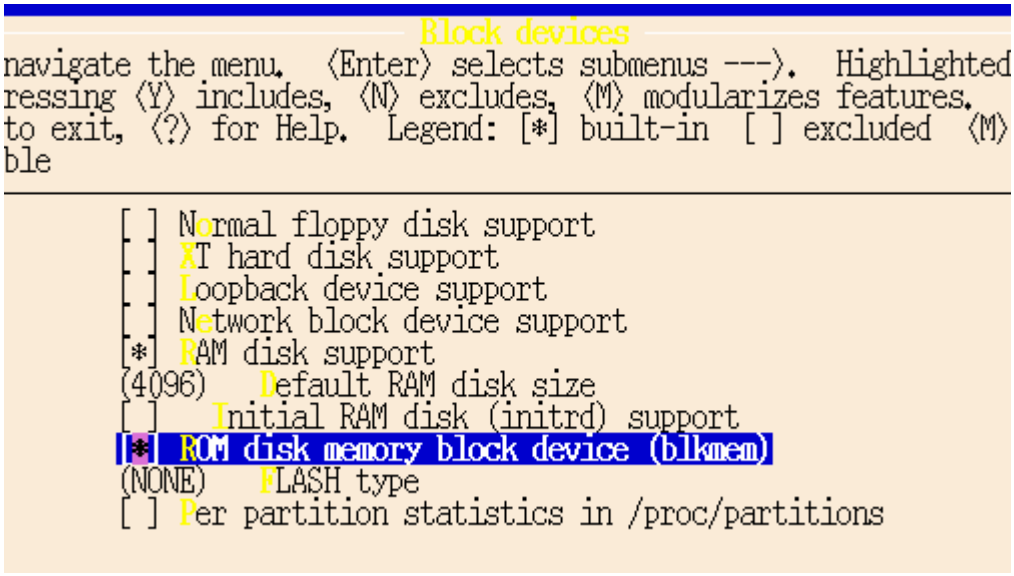

图 2-18 选择块设备支持

进入“File systems”菜单，选择 ROMFS、EXT2 和 proc 文件系统支持。如图 2-19 所示。

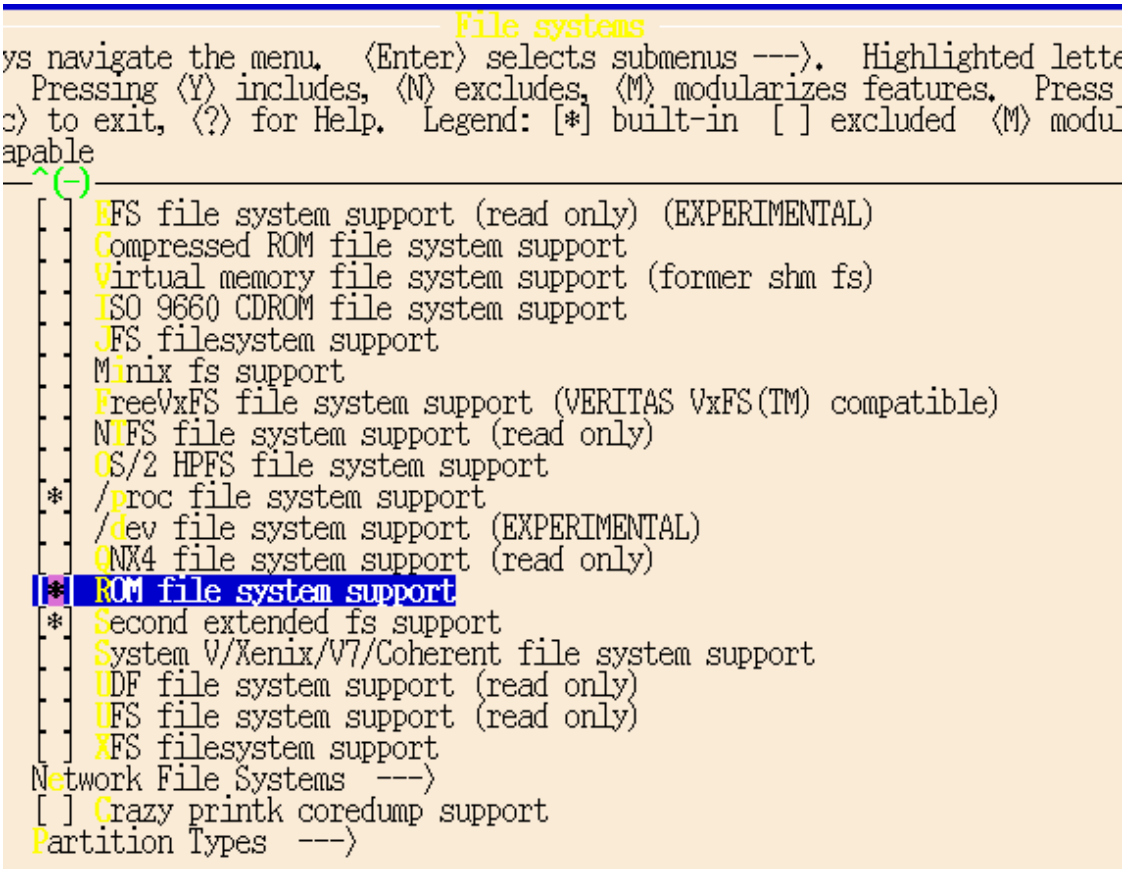

图 2-19 选择文件系统支持

配置完 uClinux 之后退出配置菜单，在 *uClinux-dist* 目录下执行如下命令编译 uClinux 的内核、C 函数库及应用程序：

```
user$ make dep
user$ make
```

如果编译成功的话将生成相应目标平台的映像文件，对于 Xcopilot 模拟器来说，就是 *images* 目录下的 *pilot.rom* 文件。

在用户主目录中建立一个符号链接 *pilot.rom*，并指向 uClinux-dist 中的映像文件，使 Xcopilot 启动时执行该映像：

```
user$ ln -s /opt/uclinux/uClinux-dist/images/pilot.rom ~/.xcopilot/pilot.rom
```

然后进入 Xcopilot 目录，执行 *xcopilot* 程序就可以运行 uClinux 系统了，见图 2-20。

```
user$ cd /opt/uclinux/xcopilot-0.6.6-uc0
user$ ./xcopilot
```

图 2-20　运行 uClinux 的 Xcopilot 模拟器

执行 *ls* 命令查看一下目录结构：

```
/> ls -l
drwxr-xr-x  1 0        0              32  Jan  1 1970  bin
drwxr-xr-x  1 0        0              32  Jan  1 1970  dev
drwxr-xr-x  1 0        0              32  Jan  1 1970  etc
drwxr-xr-x  1 0        0              32  Jan  1 1970  home
drwxr-xr-x  1 0        0              32  Jan  1 1970  lib
drwxr-xr-x  1 0        0              32  Jan  1 1970  mnt
dr-xr-xr-x  2 0        0               0  Nov 30 00:00 proc
lrwxrwxrwx  1 0        0               8  Jan  1 1970  tmp -> /var/tmp
drwxr-xr-x  1 0        0              32  Jan  1 1970  usr
drwxr-xr-x  7 0        0            1024  Nov 30 00:00 var
/>
```

目录结构和我们在 SkyEye 工程中的安排类似。不过进入 /bin 目录中可以看到，这里采用的是独立的系统程序方案，而不是 BusyBox：

```
/bin> ls -l
-rwxr-xr-x  1 0        0            1272  Jan  1 1970  expand
-rwxr-xr-x  1 0        0            9832  Jan  1 1970  init
-rwxr-xr-x  1 0        0             984  Jan  1 1970  reboot
-rwxr-xr-x  1 0        0           21840  Jan  1 1970  sh
-rwxr-xr-x  1 0        0            8764  Jan  1 1970  tip
/bin>
```

/bin 目录下只放了 *init* 和 *sh* 等比较重要的系统程序。

再看一下 */etc* 目录下的内容：

```
/etc> ls -l
-rw-r--r--  1 0        0               0  Jan  1 1970  inittab
-rw-r--r--  1 0        0             296  Jan  1 1970  motd
-rw-r--r--  1 0        0            2857  Jan  1 1970  ramfs.img
-rw-r--r--  1 0        0             200  Jan  1 1970  rc
/etc>
```

该目录下有一个很小的 RAM disk 映像和一个系统启动脚本 rc。用 cat 命令查看一下 rc 的内容：

```
/etc> cat rc
hostname CoPilot
/bin/expand /etc/ramfs.img /dev/ram0
mount -t proc proc /proc
mount -t ext2 /dev/ram0 /var
mkdir /var/tmp
mkdir /var/log
mkdir /var/run
mkdir /var/lock
mkdir /var/empty
cat /etc/motd
```

从 rc 脚本的内容可以看到，系统启动时把压缩的 RAM disk 映像 /etc/ramfs.img 解压和拷贝到 /dev/ram0 设备，然后把该设备挂装到 /var 目录，用来保存 /var 下的临时数据。

2.9 小结

在本章中，我们首先介绍了一些嵌入式 Linux 开发的基本概念和典型的开发过程，然后以 SkyEye 及 Xcopilot 模拟器为例，一步一步地针对特定的目标机器构建并运行一个嵌入式 Linux/uClinux 系统。

通过本章的理论学习和工程实践，读者应该对嵌入式 Linux/uClinux 开发有了整体上的了解和对过程细节的掌握。后面的章节将着重于阐述应用开发中的问题，特别是基于 MiniGUI 的嵌入式 Linux 应用开发的方法。

3 编写并运行嵌入式应用程序

通过第 2 章的学习，相信读者已经对基于 Linux/uClinux 的嵌入式系统开发有了一个大致的了解。如前所述，基于 Linux/uClinux 的嵌入式系统开发主要涉及三方面的内容：内核移植、驱动程序开发及应用软件开发，而其中应用软件的开发是和我们要开发的嵌入式系统最紧密相关的部分，也是直接实现产品需求、体现产品功能的部分。因此，本章将重点阐述基于 Linux/uClinux 开发嵌入式应用软件的原理及相关工具。我们首先会编写一个简单的应用程序，这个程序仅仅依赖于标准的 C 函数库，然后将该程序在 SkyEye 和 Xcopilot 模拟器上运行起来；最后我们将阐述一些和嵌入式开发紧密相关的 C 语言编程问题，这些问题常常在普通的桌面应用程序开发中被忽视，但在嵌入式系统开发中却非常重要，比如：字节序、对齐、浮点/定点运算等等。

在开始编写嵌入式应用程序之前，我们假定读者已经在自己的 PC 机上安装好了针对 SkyEye EP7312 模拟器和 Xcopilot 模拟器的交叉编译工具链，并编译好了默认的 Linux/uClinux 内核以及对应的基本根文件系统。

3.1 从“Hello, world”开始

标准的“Hello, world”程序代码如下面的清单所示：

```
#include <stdio.h>

int main (void)
{
    printf ("Hello, world\n");
    return 0;
}
```

假定该程序保存为当前目录下的 *helloworld.c* 文件，我们可在 Linux PC 上编译并运行该程序：

```
user$ gcc -Wall -O2 -o helloworld helloworld.c
user $ ./helloworld
Hello, world
```

如果我们要在 SkyEye 的 EP7312 模拟器平台上运行该程序，就要使用对应的交叉编译来完成该工作：

```
user$ arm-linux-gcc -Wall -O2 -o helloworld helloworld.c
```

当然，交叉编译生成的 *helloworld* 程序是无法在宿主机上运行的，但我们可以用 *file* 命令查看该文件的属性：

```
user$ ./helloworld
-bash: ./helloworld: cannot execute binary file

user $ file helloworld
helloworld: ELF 32-bit LSB executable, ARM, version 1 (ARM), for GNU/Linux 2.0.0, \
dynamically linked (uses shared libs), not stripped
```

注意上面 *file* 命令的输出信息。它告诉我们，*helloworld* 文件是 ELF 格式的 32 位可执行程序，指令集为 ARM，version 1 (ARM)，该可执行程序采用动态链接（即使用共享库），并且没有被剥离符号信息。

显然，我们要看到这个交叉编译之后的程序运行情况，必须在对应的目标平台上运行。但在把这个程序放在目标平台上运行之前，我们先看看程序其他方面的一些信息。

首先，我们在这个程序上运行 *arm-linux-strings* 命令：

```
user$ arm-linux-strings helloworld
/lib/ld-linux.so.2
libc.so.6
printf
abort
_IO_stdin_used
__libc_start_main
GLIBC_2.0
Hello, world
```

arm-linux-strings 是 *strings* 命令的交叉编译版本，和宿主机上的 *strings* 命令一样，它给出程序映像中的字符串常量。*helloworld.c* 程序中很明显定义了一个字符串常量，即“Hello, world\n”，这个字符串被

列在了最后；其他的字符串，如 printf、abort、_IO_stdio_used、__libc_start_main 等是这个程序要链接的函数名称，这些和程序要链接的函数库有关，这里我们略去不提。另外，上面的输出信息还告诉我们，这个程序要链接的动态函数库有两个：*/lib/ld-linux.so.2* 和 *libc.so.6*，这两个函数库名称作为字符串常量由连接器保存在程序映像中，以便在运行时查找对应的动态函数库。

上述这两个命令的输出信息，为我们提供了如下几个重要的信息：

- helloworld 是动态链接的，它的正确运行，需要在目标系统中存在相应的动态链接库。
- 要在目标平台上正确运行 helloworld 程序，我们需要准备好这个程序所使用的动态链接库：*/lib/ld-linux.so.2* 和 *libc.so.6*。前者是用来提供动态链接功能的函数库，是确保动态链接系统正常运行的必备函数库，每个动态链接的程序都要链接这个函数库，因此给出的是它的绝对路径：*/lib/ld-linux.so.2*；后者是标准的 C 函数库，即 glibc 2.0 函数库的共享对象名（shared object name）。程序在运行之前，内核的程序映像装载器将在指定的位置（一般是 */lib*、以及 *LD_LIBRARY_PATH* 环境变量指定的目录 [19]）搜寻具有 *libc.so.6* 这个名称的共享库。
- 目前的 helloworld 程序，是未剥离符号信息的程序，其中包含了用于调试程序的大量符号信息（比如某个指令对应的源代码路径及行号等），因此，它的体积比较大。

如果我们不打算在目标平台上调试这个程序，则应该使用 *arm-linux-strip* 命令剥离符号信息：

```
user$ ls -l helloworld
-rwxrwxr-x    1 weiym    weiym       11049  3月  6 02:41 helloworld
user$ arm-linux-strip helloworld
user$ ls -l helloworld
-rwxrwxr-x    1 weiym    weiym        2472  3月  6 03:01 helloworld
```

[19] *ld-linux.so.2* 这个共享库必须存放在绝对路径 */lib* 目录下，而其他共享库的默认搜索路径保存在这个共享库中。共享库系统在搜寻共享库时，首先要在该目录下寻找，然后在 *LD_LIBRARY_PATH* 环境变量定义的路径寻找。我们在该共享库上运行 arm-linux-strings 命令（$arm-linux-strings ld-linux.so.2 |grep ^/），可看到针对我们使用的 arm-linux 交叉编译环境，共享库默认路径为*/usr/local/arm/2.95.3/arm-linux/lib/*。

上面的命令给出了剥离符号前后，helloworld 程序的大小从 11,049 字节减小为 2,472 字节。

arm-linux-strip 命令和宿主机上的 *strip* 命令一样，用来将调试用的符号信息从可执行程序映像中剥离（删除），这个命令也可以用于动态和静态函数库。通常我们在将程序放到目标嵌入式系统上之前，都要使用 *strip* 命令将符号信息剥离掉。

接下来我们用静态链接方式编译一个 helloworld 程序，并观察它的文件属性、大小及剥离符号信息之后的情况（为了和动态链接版本对比，我们将静态链接版本命名为 *helloworld.static*）：

```
user$ arm-linux-gcc -static -Wall -O2 helloworld.static helloworld.c

user$ ls -l helloworld helloworld.static
-rwxrwxr-x    1 weiym    weiym       11049  3月  6 03:07 helloworld
-rwxrwxr-x    1 weiym    weiym      224624  3月  6 03:06 helloworld.static

user$ arm-linux-strip helloworld helloworld.static

user$ ls -l helloworld helloworld.static
-rwxrwxr-x    1 weiym    weiym        2472  3月  6 03:08 helloworld
-rwxrwxr-x    1 weiym    weiym      224624  3月  6 03:08 helloworld.static
```

我们可以看到，静态链接版本的程序映像要远远大于动态链接的程序映像，不论是否剥离了符号信息。在上面的命令中，我们没有在 *helloworld.static* 上运行 *strings* 命令，因为这个命令将把静态链接到程序映像中的大量信息输出在屏幕上（这些信息是从 C 函数库中链接到程序映像中的），其中主要是 glibc 函数库中的一些错误输出信息（字符串常量），读者可以运行一下看看。当然，静态链接生成的程序在某些情况下也有好处，因为它不需要额外的动态函数库支持。

strings 和 *strip* 命令都是 C 语言开发工具集合中的二进制工具，它们作用于二进制的程序或者函数库文件，通过运行这些命令，我们可以从多个角度来了解所生成的程序及函数库。交叉编译工具链中的二进制工具通常具有一个前缀名，比如在我们这个例子中，arm-linux 就是这个前缀名。表 3-1 给出了常见二进制工具的用途。

表 3-1 常见二进制工具及其用途

工具名称	用途
ar	创建、修改归档文件，也可从归档文件中析取单个文件。归档文

	件其实就是由多个文件有机组织成的一个大文件，其中包含了原始文件的内容、访问属性、所有者等信息。归档文件经常用于静态函数库，其中包含了公用函数及例程，因此，我们也可以将 *ar* 看成静态函数库的生成工具。
as	汇编程序。将汇编语言翻译成二进制指令的工具。
ld	链接器。用来链接各种目标文件生成可执行程序或动态链接库的程序。
nm	列出目标文件（程序映像、动态库及静态库）中的符号，作用于未被剥离符号的目标文件。
objdump	转储目标文件中的信息。可用来打印目标文件的头信息、调试信息，对目标文件进行反汇编等等。
objcopy	复制并翻译目标文件。可将一个目标文件中的内容转换为其他格式。
ranlib	生成归档文件的索引信息，该命令和使用 –s 选项的 ar 命令是等价的。索引信息可帮助链接器快速定位目标文件中的符号。
readelf	显示 ELF 文件的信息。ELF 是 UNIX 系统定义的一种可执行文件格式，其含义是可执行链接格式（Excutable and Linkable Format）。动态函数库及可执行文件通常使用这种格式；另一种老的可执行文件格式是 a.out 格式。
size	列出目标文件的段（正文段、数据段、BSS 段等）的大小及整个大小。
strings	列出目标文件中的可打印字符串。
strip	剥离目标文件中的符号信息。

接下来，我们就在 SkyEye 的 EP7312 模拟器上运行交叉编译后的程序。

针对 SkyEye 的 EP7312 模拟器，我们使用 initrd 技术，将根文件系统放在 RAMDISK 中，由内核在引导结束后挂装该文件系统。为此，我们只需将程序复制到已有的 initrd 映像中即可。为了简单起见，我们运行上面程序的静态链接版本（动态链接的例子可参阅本书 4.3 节中 MiniGUI 示例程序的运行）。

我们要使用的原始 *initrd.img* 映像文件保存在产品光盘的 */armlinux/ep7312/* 目录下。将 *helloworld.static* 文件添加到这个映像中，步骤如下：

第1步. 将已有的 *initrd.img* 文件挂装到 */mnt/tmp* 目录下：

```
root# cp <path to cdrom>/armlinux/ep7312/initrd.img .
root# file initrd.img
initrd.img: Linux rev 1.0 ext2 filesystem data

root# mkdir -p /mnt/tmp
root# mount -t ext2 -o loop initrd.img /mnt/tmp/
```

第2步. 将 *helloworld.static* 文件复制到 initrd 文件系统中：

```
root# cp helloworld.static /mnt/tmp/bin
```

```
root# umount /mnt/tmp
```

至此，包含 *helloworld.static* 示例程序的 initrd 文件系统映像就建立好了。在 *initrd.img* 所在的目录中，复制已编译好的 Linux 内核映像（可使用产品光盘中 */armlinux/ep7312/* 目录下的 *vmlinux.for.ep7312* 文件），并创建如下的 *skyeye.conf* 文件：

```
cpu: arm720t
mach: ep7312
mem_bank: map=I, type=RW, addr=0x80000000, size=0x00010000
mem_bank: map=M, type=RW, addr=0xc0000000, size=0x00200000
mem_bank: map=M, type=RW, addr=0xc0200000, size=0x00200000, file=./initrd.img
mem_bank: map=M, type=RW, addr=0xc0400000, size=0x00c00000
```

然后在 *skyeye.conf* 文件所在目录运行 SkyEye：

```
# skyeye vmlinux.for.ep7312
(SkyEye) target sim
(SkyEye) load
(SkyEye) run
```

内核启动之后，进入 *bin/* 目录运行 *helloworld.static* 程序：

```
Welcome to
     _      _____      __   __    _      _
    / \    /  __ \    /  \_/  \  | |    |_|
   / _ \   | |  | |  / /\   /\ \ | |      _ ____  _   _  _  _
  / /_\ \  | |__| | / /  \_/  \ \| |     | |  _ \| | | |\ \/ /
 / /___\ \ | |__\ \ | |       | || |___  | | |_| | |_| |/    \
/_/     \_\| |   \_\|_|       |_||_____||_|_| |_|\____|\_/\_/

ARMLinux for Skyeye
For further information check:
http://hpclab.cs.tsinghua.edu.cn/~skyeye/

Execution Finished, Exiting
Command: /bin/sh

Sash command shell (version 1.1.1)
/> cd bin
/bin> ls
helloworld.static
init
mount
sh
/bin> ./helloworld.static
Hello, world
/bin>
```

如果要使用针对 uClinux 系统的交叉编译工具编译“ Hello, world ”程序，其过程要复杂很多，这是因为 uClinux 操作系统有如下限制：

- uClinux 运行在没有 MMU 的处理器上，这种处理器缺乏对虚拟内

存特性的支持，因此，普通版本的 uClinux 操作系统不提供对共享库的支持。也就是说，我们只能编译生成静态链接的可执行程序。

- 针对 uClinux 操作系统的交叉编译工具链通常不包含针对特定平台的完整交叉编译环境所需要的头文件及函数库，相反，我们要自行选择 uC-libc 或者 uClibc 作为系统函数库及头文件。因此，在编译应用程序时，我们要通过 *-I* 及 *-L* 等选项显式指定要使用的系统头文件位置及函数库位置，以及要链接的函数库名称。
- 由于针对 uClinux 操作系统的交叉编译器通常可用来编译生成针对不同处理器类型的目标文件，因此我们必须在编译时显式指定目标处理器的类型。
- 由于没有 MMU 的支持，uClinux 不支持 ELF 可执行文件格式；它支持的可执行文件格式为 FLAT（或压缩的 FLAT 格式 ZFLAT）。因此，我们要将交叉编译器生成的 ELF 格式文件通过 *objcopy* 二进制工具转换为 FLAT 格式的可执行格式。

这样，我们在编译针对某个特定处理器的目标文件时，需要指定许多额外的编译参数。比如，针对 Xcopilot 模拟器，它模拟的是采用 DragonBall（Motorola M68000）处理器的 3Com Copilot PDA，我们要特别指定如下交叉编译选项：

- *-m68000* [20]。该选项指定交叉编译器生成针对 Motorola 68000 处理器的指令。同样采用 DragonBall 处理器，但不同的处理器型号在指令集上有差别，因此，必须根据处理器的型号确定这个选项。同样的情况也会发生在其他处理器上，比如基于 ARM 核的各种处理器。

- *-I* 选项。该选项显式指定头文件的搜索路径。

- *-L* 及 *–l* 选项。这两个选项分别指定函数库的位置及要链接的函数库名称。

- *-Wl,-elf2flt* 等选项。*-Wl,*打头的选项是通过编译器向链接器传递的附加参数。

[20] gcc 的 –m 选项通常是针对某类处理器类型的特殊编译选项，这里的 m 表示 machine。
针对 m68k 系列处理器的 GCC 3 特殊选项，可访问：
http://gcc.gnu.org/onlinedocs/gcc-3.4.3/gcc/M680x0-Options.html。
其他处理器相关的特殊选项，可访问：
http://gcc.gnu.org/onlinedocs/gcc-3.4.3/gcc/Submodel-Options.html#Submodel-Options

假定我们针对 Xcopilot 模拟器使用 uClibc 函数库，并已经按照第 2 章中的描述交叉编译好了 uClibc 函数库，且保存在 */opt/uclinux/uClinux-dist/uClibc* 目录下，则我们用来编译“ Hello, world ”程序的命令行为：

```
user$ m68k-elf-gcc \
        -m68000 -mid-shared-library -mshared-library-id=0 -Os -g -fomit-frame-pointer \
        -I/opt/uclinux/uClinux-dist/lib/uClibc/include \
        -Wl,-elf2flt -Wl,-move-rodata -Wl,-shared-lib-id,0 \
        -nostartfiles /opt/uclinux/uClinux-dist/lib/uClibc/lib/crt0.o \
        -L/opt/uclinux/uClinux-dist/lib/uClibc/lib \
        -o helloworld helloworld.c \
        -Wl,-R,/opt/uclinux/uClinux-dist/lib/uClibc/libc.gdb –lc
```

上面的命令行中，*-mid-shared-library -mshared-library-id=0* 选项告诉编译器通过使用库 ID 方法来支持共享库，并且指定共享库 ID 为 0[21]；*-Os* 选项是优化选项，即针对大小进行优化，使得生成的代码体积最小；*-g* 指定生成可调试的代码（即包含符号信息）。

-fomit-frame-pointer 是一种优化选项，可节省函数调用的开支；*-I/opt/uclinux/uClinux-dist/lib/uClibc/include* 指定头文件的搜索路径，这里是 uClibc 的头文件所在路径。

-Wl,-elf2flt 选项将由编译器传递到链接器，告诉链接器将 ELF 格式的可执行程序转换为 FLAT 格式；*-Wl,-move-rodata* 选项也将传递给链接器，告诉链接器将只读数据段（.rodata）移到正文段（.text）中；*-Wl,-shared-lib-id,0* 告诉链接器要链接的共享库 ID 为 0，*-nostartfiles /opt/uclinux/uClinux-dist/lib/uClibc/lib/crt0.o* 选项指定编译器不要链接默认的 C 程序启动代码，而要使用 uClibc 提供的启动代码（即 *crt0.o* 文件）；*-Wl,-R,/opt/uclinux/uClinux-dist/lib/uClibc/libc.gdb* 选项告诉链接器从 uClibc 的 *libc.gdb* 中读取符号及其地址信息；*–lc* 选项告诉链接器在生成可执行程序时，还要链接 uClibc 提供的 C 函数库（即 *libc.a*）文件。

【提示】我们可以在 uClinux-dist 的编译过程中获得针对某个处理器和目标平台的正确编译选

[21] 这两个选项使得程序可在没有虚拟内存管理（无共享库支持）的处理器上运行 in place 和共享库代码。使用该选项，将同时打开 –fPIC 选项（生成位置无关代码）。指定共享库 ID 为 0 时，将生成更为紧凑的代码。这两个选项是 m68k 处理器特有的选项。和这两个选项相关的选项还有 –msep-data，该选项使得数据段可放在正文段中的不同区域，从而可在没有虚拟内存管理的环境中执行 in place 代码。

项。如果您使用 Bash Shell，则可以在 uClinux-dist 的编译过程中按<Ctrl+S>组合键（相当于<ScrLck>）来仔细观察编译和链接选项；按<Ctrl+Q>组合键可取消滚动锁定。

上面的命令将生成两个文件，分别为 *helloworld.gdb* 和 *helloworld*。前者是 ELF 格式的可执行程序，后者是转换为 FLAT 格式的可执行程序：

```
user$ ls -l
-rwxr--r--    1 weiym    weiym         680  3月  7 06:39 helloworld
-rw-rw-r--    1 weiym    weiym          86  3月  5 23:21 helloworld.c
-rwxrwxr-x    1 weiym    weiym       88148  3月  7 06:39 helloworld.gdb

user$ file helloworld.gdb helloworld
helloworld.gdb: ELF 32-bit MSB executable, Motorola 68020, version 1 (SYSV), \
statically linked, not stripped
helloworld:     data
```

file 命令告诉我们，*helloworld.gdb* 是 ELF 的 32 位可执行程序，指令集为 Motorola 68020, version 1 (SYSV)，静态链接，未剥离符号信息。对 *helloworld*，*file* 命令告诉我们它是普通的数据文件，其实这就是 FLAT 格式的含义，因为它不包含任何的格式信息，而只是包含了可执行程序的代码和数据，因此，*file* 命令无法判断 *helloworld* 属于哪类文件。

接下来，我们把编译好的 *helloworld* 程序放到 Xcopilot 模拟器中运行。

首先，我们将编译好的 *helloworld* 程序复制到 uClinux-dist 的 *romfs* 中，然后运行 *make image* 命令：

```
root# cd /opt/uclinux/uClinux-dist
root# cp <path_to_helloworld>/helloworld romfs/bin
root# make image
```

该命令将把 *uClinux-dist/romfs* 下的文件系统重新打包生成映像文件[22]（保存为 *uClinux-dist/images/pilot.rom*文件）。确保您用户主目录下的 *pilot.rom* 文件通过符号链接指向 *uClinux-dist/images* 目录中的 *pilot.rom* 文件：

```
user$ ln –sf /opt/uclinux/uClinux-dist/images/pilot.rom ~/.xcopilot/pilot.rom
```

然后以普通用户身份在 X Window 的终端仿真程序中运行 *xcopilot* 命令。下面是运行 Xcopilot 模拟器之后的部分终端输出：

[22] 在这里，make 调用的是 Linux 的 genromfs 工具。

```
Command: mount -t proc proc /proc
Command: mount -t ext2 /dev/ram0 /var
Command: mkdir /var/tmp
Command: mkdir /var/log
Command: mkdir /var/run
Command: mkdir /var/lock
Command: cat /etc/motd
Welcome to
          ____ _  _
         /  __| ||_|
    _    _| |   | | _ ____  _    _  _  _
   | |  | | |   | || |  _ \| |  | |\ \/ /
   | |_| | |__| || | | | | | |_| |/    \
   |  ___\____|_||_|_| |_|\____|\_/\_/
   | |
   |_|

For further information check:
http://www.uclinux.org/

Execution Finished, Exiting

Sash command shell (version 1.1.1)
/> cd bin
/bin> ./helloworld
Hello, world
/bin>
```

其中，*cd bin* 及之后的命令是内核启动后我们键入的命令。

3.2 利用 Makefile 维护嵌入式应用工程

上面的小节描述了针对目标平台交叉编译一个简单应用程序的过程。然而，许多复杂的嵌入式应用程序不会像“Hello, world”程序一样简单到只需一个源文件。本节将通过一个由多个源文件组成的复杂嵌入式应用程序的交叉编译来阐述如何使用 make 和 makefile 有效完成交叉编译。

3.2.1 make 和 makefile 的简单回顾

make 是 Linux 下最常用的二进制程序、函数库的建立生成工具。make 运行时要根据当前目录下的 makefile 文件（一般是 Makefile），确定要生成什么样的二进制文件，以及对应的命令。我们还可以在 makefile 文件中建立要生成的目标与源代码之间的依赖关系，从而可以让 make 工具根据文件的最新改动时间自动判断是否需要通过中间过程而生成最终目标。

makefile 文件是许多编译器，包括 Win32 平台上的编译器维护编译信息的常用方法，只是在 Win32 的集成开发环境中，用户可以通过友好的人机界面修改 makefile 文件而已。

在 Linux 开发平台上，我们通常使用的 make 工具是 GNU make 工具。默认情况下，GNU make 工具在当前工作目录中按如下顺序搜索 makefile：

- GNUmakefile
- makefile
- Makefile

在 UNIX 系统中，习惯使用 *Makefile* 作为 makfile 文件。如果要使用其他文件作为 makefile，则可利用类似下面的 make 命令选项指定 makefile 文件：

```
$ make -f Makefile.debug
```

makefile 文件中一般包含如下内容：

- 需要由 *make* 工具创建的项目，通常是目标文件和可执行文件。我们使用“目标（target）”一词来表示要创建的项目。
- 要创建的项目依赖于哪些文件，即依赖关系（dependency）。
- 如何从被依赖的文件或子目标生成目标，即创建目标的规则（rule）。

例如，假设我们现在要编译 3.1 节中的 *helloworld.c* 程序，我们可以利用如下的 makefile 文件来定义 *helloworld* 程序的创建规则：

```
helloworld: helloworld.o
    gcc -o helloworld.o

helloworld.o: helloworld.c
    gcc -c -Wall -O2 -o helloworld.o helloworld.c
```

上面的 makefile 文件指定了两个目标，分别为 *helloworld* 程序以及中间目标文件 *helloworld.o*，前者依赖于 *helloworld.o* 目标，而后者依赖于 *helloworld.c* 文件；这两个目标的生成规则由目标下面的命令给出。

上面这个 makefile 文件定义了多个目标，利用 *make* **<target>** 命令可指定要编译的目标，如果不指定目标，则使用 *all* 目标或者第一个目标。通常，makefile 中定义有 *clean* 目标，可用来清除编译过程中的中间文件，例如：

```
clean:
    rm -f *.o
```

运行 *make clean* 时，将执行 *rm -f *.o* 命令，最终删除编译过程中产生的所有中间文件。一个最简单的 makefile 可以像上面这样非常直接，但是，这并不是我们的目的，因为我们还可以利用 makefile 的其他许多便利功能。

GNU 的 make 工具除提供有建立目标的基本功能之外，还有许多便于表达依赖性关系以及建立目标的功能和特色。其中之一就是变量或宏的定义能力。如果要以相同的编译选项同时编译十几个 C 源文件，而为每个目标的编译指定冗长的编译选项的话，将是非常乏味的。但利用简单的变量定义以及 makefile 的模式或者后缀名规则，可避免这种乏味的工作。例如，我们可以这样编写 helloworld 程序的 makefile：

```
CC= gcc
CFLAGS=-Wall -O2 -g

LD= gcc
LDFLAGS=

PROG=helloworld
OBJS=helloworld.o

# Rule for building an object file
.c.o:
        $(CC) $(CFLAGS) $(CPPFLAGS) -c -o $@ $<

$(PROG): $(OBJS)
        $(LD) $(LDFLAGS) -o $@ $<

all: $(PROG)

clean:
        rm $(OBJS) $(PROG) -f
```

在上面的例子中，*CC*、*CFLAGS*、*LD*、*LDFLAGS* 等就是 make 的预定义变量；*PROG* 和 *OBJS* 是用户自定义变量。在 makefile 中引用变量的值时，只需变量名之前添加 *$* 符号，如上面的 *$(CC)* 和 *$(CFLAGS)*。

另外，上面的 makefile 还使用了 GNU make 中的隐含规则，即目标 *.c.o*（这里使用的是后缀规则），我们也可以使用下面的模式规则：

```
%.o:%.c
        $(CC) $(CFLAGS) $(CPPFLAGS) -c -o $@ $<
```

如果读者对 make 和 makefile 的上述概念和用法还不是非常清楚，可阅读讲述 Linux 开发工具的相关书籍；INTERNET 上也有许多文章介绍 make 和 makefile 的用法。

3.2.2 编写针对交叉编译的 Makefile 文件

下面，我们假定要交叉编译一个复杂点的应用程序，这个应用程序就是 Linux 上的 *cal* 命令。*cal* 命令在终端上打印给定月份的月历，比如：

```
$ cal
     March 2005
Su Mo Tu We Th Fr Sa
       1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 31
```

这个程序的源代码被组织成了三个源文件 *cal.c*、*print.c* 和 *cal.h*。其中 *cal.c* 包含了 *main* 函数及 *usage* 函数的实现，*print.c* 包含了打印月历的函数，*cal.h* 包含了公用函数接口的声明。该示例程序的源代码可见产品光盘的 */samples/cal.tar.gz* 包。

如果我们针对嵌入式 Linux 操作系统交叉编译 *cal* 程序，则对应的 makefile 基本上和 PC 版本不会有大的差别。比如针对 SkyEye 的 EP7312 模拟器，我们可如下编写 makefile（*Makefile.arm-linux*）：

```
# Modify the following variables to meet your cross-compilation toolchain.
CROSS=arm-linux
PREFIX=/usr/local/arm-linux/arm-linux/

### do not modify the following variables and rules. ###
BINDIR=$(PREFIX)/bin

CC=$(CROSS)-gcc
CFLAGS=-Wall -O2 -g

LD=$(CROSS)-gcc
LDFLAGS=

STRIP=$(CROSS)-strip

RM=rm -f
INSTALL=install -c

PROG=$(CROSS)-cal
OBJS=cal.o print.o

# Rule for building an object file
%.o:%.c
        $(CC) $(CFLAGS) $(CPPFLAGS) -c -o $@ $<

$(PROG): $(OBJS)
        $(LD) $(LDFLAGS) -o $@ $^

all: $(PROG)

clean:
        $(RM) $(OBJS) $(PROG) .depend core

install: $(PROG)
        $(STRIP) $(PROG)
        $(INSTALL) -o root -g root -s -m 0755 $(PROG) $(BINDIR)

uninstall:
```

```
        rm -f $(BINDIR)/$(PROG)

depend .depend:
        $(CC) $(CFLAGS) -M *.c > .depend

ifeq (.depend, $(wildcard .depend))
include .depend
endif
```

上面这个 makefile 可作为嵌入式 Linux 应用程序的工程模板，它包含有通常 makefile 文件中比较完整的目标，比如 *clean*、*install*、*uninstall*、*depend* 等等。注意，这个 makefile 文件在生成的可执行程序名称上添加了 *arm-linux* 前缀。

在此 makefile 基础上，结合上一节我们提到的针对 uClinux 进行交叉编译所需的额外编译选项，我们可以非常方便地针对 uClinux 操作系统编写 cal 程序的交叉编译 makefile 文件（*Makefile.m68k-elf*）：

```
# Modify the following variables to meet your cross-compilation toolchain.
CROSS=m68k-elf
UCLINUX_DIR=/opt/uclinux/uClinux-dist
PREFIX=/usr/local/m68k-elf/

CFLAGS=-m68000 -mid-shared-library -mshared-library-id=0 -Os -Wall -g -fomit-frame-pointer \
        -I$(UCLINUX_DIR)/lib/uClibc/include
LDFLAGS=-Wl,-elf2flt -Wl,-move-rodata -Wl,-shared-lib-id,0 \
        -nostartfiles $(UCLINUX_DIR)/lib/uClibc/lib/crt0.o \
        -L$(UCLINUX_DIR)/lib/uClibc/lib \
        -Wl,-R,$(UCLINUX_DIR)/lib/uClibc/libc.gdb -lc

### do not modify the following variables and rules. ###

CC=$(CROSS)-gcc
LD=$(CROSS)-gcc
BINDIR=$(PREFIX)/bin

RM=rm -f
INSTALL=install -c

PROG=$(CROSS)-cal
OBJS=cal.o print.o

# Rule for building an object file
%.o:%.c
        $(CC) $(CFLAGS) $(CPPFLAGS) -c -o $@ $<

$(PROG): $(OBJS)
        $(LD) $(LDFLAGS) -o $@ $^

all: $(PROG)

clean:
        $(RM) $(OBJS) $(PROG) $(PROG).gdb .depend core

install: $(PROG)
        $(INSTALL) -o root -g root -s -m 0755 $(PROG) $(BINDIR)

uninstall:
        rm -f $(BINDIR)/$(PROG)

depend .depend:
        $(CC) $(CFLAGS) -M *.c > .depend

ifeq (.depend, $(wildcard .depend))
include .depend
endif
```

和前面用于 SkeEye EP7312 模拟器的 makefile 文件相比较，*Makefile.m68k-elf* 在如下几个方面有所不同：

- 根据 3.1 节中的描述，我们针对 *m68k-elf-gcc* 和 *uClibc* 分别设置了 *CFLAGS* 和 *LDFLAGS* 变量。
- 不再使用 *strip* 命令。因为链接器将 ELF 格式转换为 FLAT 格式之后，将自动剥离符号信息。
- *PREFIX* 变量取了不同的值。
- *clean* 目标中删除了 *$(PROG).gdb* 文件。

除上述差别之外，*Makefile.m68k-elf* 和 *Makefile.arm-linux* 文件没有任何差别。读者可以分别运行 *make* 命令、*make –f Makefile.arm-linux* 及 *make –f Makefile.m68k-elf* 命令编译针对 Linux PC、ARM Linux 和 Xcopilot 的 *cal* 程序。注意在进行新的平台的交叉编译时，一定要运行 *make clean* 命令清除中间文件，否则会出现编译错误。

3.2.3 将应用程序放到 uClinux-dist 中编译

从上面的示例中我们可以看到，针对嵌入式 Linux 的交叉编译，因为交叉编译工具链比较成熟，所以应用程序的交叉编译过程相对简单；相反，针对嵌入式 uClinux 的应用程序交叉编译就相对复杂。如果我们的应用程序不是非常复杂，则我们可以将自己的应用程序放到 *uClinux-dist/* 目录树中和 uClinux 内核、uClibc 函数库一起编译。这样做的优点主要是：

- 我们可以借助 uClinux-dist 的 makefile 结构以及预先设定的变量及规则，非常方便地编译自己的应用程序，而不需要关注那些特殊的编译选项。

然而，这样做也带来一个坏处，即：每次编译时，总是要从内核、uClibc 开始编译，从而会浪费一些时间。但不论如何，这种方法给我们提供了一种偷懒的途径，还是有必要掌握的。其具体步骤如下（以 3.2.2 中的 cal 为例）：

第1步. 在 *uClinux-dist/user* 目录下为新的程序建立目录，并将 *cal* 程序的三个源文件复制到该目录下：

```
root# cd /opt/uclinux/uClinux-dist/user
root# mkdir mycal
root# cd mycal
root# cp <path to cal>/cal/*.[ch] .
```

第2步. 从 *user* 目录下的 *tip* 目录中复制 *Makefile* 文件：

```
root# cp ../tip/Makefile .
```

第3步. 对复制后的 *Makefile* 文件做相应改动（只需修改前两行）：

```
EXEC = mycal
OBJS = cal.o print.o

all: $(EXEC)

$(EXEC): $(OBJS)
    $(CC) $(LDFLAGS) -o $@ $(OBJS) $(LDLIBS$(LDLIBS_$@))

romfs:
    $(ROMFSINST) /bin/$(EXEC)

clean:
    -rm -f $(EXEC) *.elf *.gdb *.o
```

第4步. 修改 *user/* 目录下的 *Makefile* 文件，在 *dir_y += games* 一行的后面加上一行：

```
dir_y += games
dir_y += mycal
```

这样在 uClinux-dist 目录下输入“ *make* ”进行编译时就会把该程序包括在编译范围之内，编译后将在 *romfs/bin* 目录下生成 *mycal* 程序文件。

3.3 嵌入式应用开发涉及的特殊 C 语言问题

3.3.1 嵌入式开发和桌面开发在 C 语言上的主要不同

大多数 C 语言程序员的主要开发经验来自于桌面，比如开发 Windows 程序，或者 Linux/UNIX 程序，加上这些通用操作系统的高级功能（比如独立的高达 4GB 的进程地址空间）以及硬件的超强性能，使得我们在日常的程序开发中，可以随心所欲地编写自己的应用程序。举例来说，在我们目前使用的桌面操作系统下,编写下面的函数基本上不会出现任何问题：

```
void foo (int bar)
{
    char temp [1024];
```

```
    // do your work
    if (bar > 1) {
        bar --;
        foo (bar);
    }

    return;
}
```

上述程序设立了一个局部变量，是个 char 型的数组，大小为 1024。但是，如果这个函数恰好是一个递归函数，则会出现一个重要的问题，即每递归调用一次这个函数，程序所使用的栈将增加至少 1024 字节。这种情况在嵌入式系统上是必须要避免的，因为嵌入式系统的资源有限，上述这样的递归函数将最终耗尽系统的栈空间，从而导致不可预期的程序运行效果。

在传统嵌入式操作系统的开发中，每个任务均会事先分配一个足够的栈空间大小，超过这个栈空间大小，程序的运行就将超出预期，从而导致整个系统崩溃。使用嵌入式 Linux （不包括 uClinux）的情况稍好，因为嵌入式 Linux 将通用操作系统的虚拟内存技术带到了嵌入式系统，从而可以在一定程度上解放嵌入式程序开发人员。但是，这种解放仍然是有限的，我们在嵌入式开发中仍然要时刻注意资源的可获得性。

另外，我们在桌面开发中经常使用的一些技巧，在嵌入式开发中也可能不再有效。这要求嵌入式开发人员在编写 C 程序时，必须对 C 语言许多不太常用的关键词及语言特性，以及编译器的编译选项有正确和深入的了解。

简单归纳，针对嵌入式系统进行 C 程序开发时，我们需要注意如下一些方面的问题：

- 在嵌入式系统开发中，要时刻注意节省使用内存资源，尤其是堆栈资源。
- 在嵌入式系统开发中，要知道不同的处理器架构在存储多字节数据对象时，可能采用不同的字节顺序（字节序）。
- 要知道某些在桌面上运行起来不会有任何问题的语句，可能在嵌入式系统中无法正常执行。本节后面将重点提到因为数据操作的对齐问题而可能导致的程序缺陷。
- 许多嵌入式处理器不具备浮点协处理器，浮点运算通常是通过软件实现的，因此，在嵌入式系统开发中，不能随意使用浮点运算，否则将对程序性能产生非常大的影响。

本节下面将就上面这些主题具体阐述其原理及解决办法。

3.3.2 字节序及其处理

字节序（byte order）指处理器在处理多字节的数据对象（比如 *short*、*int*、*double* 等）时，在寄存器及内存中以什么样的顺序保存字节。

我们知道，桌面系统通常运行在 Intel 的 x86 处理器上，而这种处理器按照低地址存放低位数据的原则保存多字节数据对象。假定整数 0x1234 是一个 16 位整数，则在 Intel x86 平台上，处理器将按图 3-3 所示的方式存储这个整数。即低地址 A 中保存 0x34 这个字节，而在高地址 A+1 中，保存 0x12 这个字节。这种字节序称为“小头（little-endian）”。

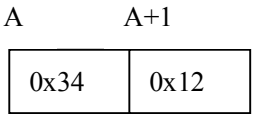

图 3-3 小头（little-endian）存储

当我们通过下面的语句将这个整数保存到文件中时，也将保留上述顺序：

```
unsigned short temp = 0x1234;
write (fd, &temp, sizeof (unsigned short));
```

当我们再次从这个文件中读取这个整数时，我们可以用下面的语句：

```
unsigned short temp;
read (fd, &temp, sizeof (unsigned short));
// temp = 0x1234
```

如果上述代码仍然在 Intel 的 x86 处理器上运行，则调用 read 函数之后，temp 中的值仍然为 0x1234。然而，当我们在使用大头（big-endian）字节序的处理器上执行上述读取代码时（这里假定文件由小头系统写入），我们会发现 temp 的值变成了 0x3412。

大头系统存储多字节数据对象时，采用和小头系统完全相反的方式，见图 3-4。

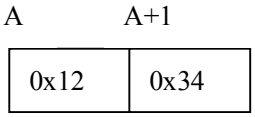

图 3-4 大头（big-endian）存储

许多 RISC 处理器都使用大头字节序，比如 PowerPC、M68k 等（MIPS 处理器可灵活设置采用哪种字节序）。Xcopilot 模拟的就是 M68k 处理器，因此，我们可以通过编写简单的程序来验证上述结论。

首先，我们在 PC 上编写下面的程序，将 0x1234 写入文件 *endina.dat*：

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

int main (void)
{
        unsigned char *p;
        unsigned short temp = 0x1234;

        int fd = open ("endian.dat", O_CREAT | O_TRUNC | O_RDWR, 0666);

        if (fd < 0) {
                perror ("Open file error");
                return 1;
        }

        // write the short data to the file
        write (fd, &temp, sizeof (unsigned short));

        // read back the data.
        lseek (fd, 0, SEEK_SET);
        read (fd, &temp, sizeof (unsigned short));
        close (fd);

        // print the bytes
        p = (unsigned char*) &temp;
        printf ("the first byte is 0x%x\n", *p);
        printf ("the second byte is 0x%x\n", *(p + 1));

        return 0;
}
```

编译并执行，我们将得到下面的结果：

```
user$ gcc -o endian endian.c
user$ ./endian
the first byte is 0x34
the second byte is 0x12
```

而如果在 Xcopilot 模拟器上运行下面读取 *endian.dat* 的程序，我们将得到相反的结果：

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

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

int main (void)
{
        unsigned char *p;
        unsigned short temp;

        int fd = open ("endian.dat", O_RDONLY);

        if (fd < 0) {
                perror ("Open file error");
                return 1;
        }

        // read back the data.
        read (fd, &temp, sizeof (unsigned short));
        close (fd);

        // print the bytes
        p = (unsigned char*) &temp;
        printf ("the first byte is 0x%x\n", *p);
        printf ("the second byte is 0x%x\n", *(p + 1));

        return 0;
}

--------
the first byte is 0x12
the second byte is 0x34
```

那么，我们怎么判断程序运行在大头系统还是小头系统上呢？在程序运行时，我们可以通过下面的代码段来判断：

```
#include <stdio.h>

inline int is_big_endian (void)
{
        union {
                unsigned short i;
                unsigned char  c[2];
        } data;

        data.i = 0x1234;

        if (data.c[0] == 0x12)
                return 1;

        return 0;
}

int main (void)
{
        if (is_big_endian ())
                printf ("This is a big-endian system\n");
        else
                printf ("This is a little-endian system\n");

        return 0;
}
```

注意，上述代码段的 *is_big_endian* 函数使用了 C 语言的联合类型，这种类型在处理这种形式的问题时具有非常好的便利性。

【思考题】下面的程序段，在大头系统和小头系统上的输出是一样的吗？

```
unsigned short temp = 0x1234;

unsigned char byte = (unsigned char)(temp >> 8);

printf (“the low byte is 0x%x\n”, byte);
```

在编译时，我们只能通过编译器的预定义宏来判断，比如在 MiniGUI 中，使用了下面的条件编译：

```
#define MGUI_LIL_ENDIAN  1234
#define MGUI_BIG_ENDIAN  4321

#if  defined(__i386__) || defined(__ia64__) || \
    (defined(__alpha__) || defined(__alpha)) || \
     defined(__arm__) || \
    (defined(__CC_ARM) && !defined(__BIG_ENDIAN)) || \
    (defined(__mips__) && defined(__MIPSEL__)) || \
     defined(__LITTLE_ENDIAN__) || \
    defined(WIN32)
#define MGUI_BYTEORDER   MGUI_LIL_ENDIAN
#else
#define MGUI_BYTEORDER   MGUI_BIG_ENDIAN
#endif
```

MiniGUI 通过（交叉）编译器预先定义的宏来判断目标平台是大头系统还是小头系统，然后将 *MGUI_BYTEORDER* 定义为相应的宏。在程序代码中，可采用下面的方法来处理字节序问题：

```
#if MGUI_BYTEORDER == MGUI_BIG_ENDIAN
    // for the big-endian system
#else
    // for the little-endian system
#endif
```

比如，我们为了在 Xcopilot 上获得读入的正确值，就可以用下面的方法编写 MiniGUI 应用程序：

```
        unsigned short temp;

        // read back the data.
        read (fd, &temp, sizeof (unsigned short));
        close (fd);

#if MGUI_BYTEORDER == MGUI_BIG_ENDIAN
        temp = ((temp << 8)|(temp >> 8));
#endif
```

上面的 `temp = ((temp << 8)|(temp >> 8));` 语句将 *temp* 的高八位和低八位互换然后重新组合成了新的整数。为了编程方便，MiniGUI 中还提供了一些函数，用来处理字节序问题，如果读者感兴趣，可参阅《MiniGUI 编程指南》12.5.1 节“理解并使用 MiniGUI 的 Endian 读写函数”。

除了多字节的整数类型（*short*、*int*、*long* 等）外，*float*、*double* 等浮点数类型在不同的架构上也具有不同的存储方式。尤其是 ARM 架构比较特殊，它的整数数据采用小头字节序，而浮点数（8 位的 double 数据）却使用大头字节序。

3.3.3 对齐

对齐问题是许多接触 C 语言不多的开发人员常常忽略的问题。为了对对齐有个感性认识，我们首先看下面的代码：

```
#include <stdio.h>

void align_1 (void)
{
        struct {
                char c;
                int i;
        } my_struct;

        printf ("The size of my_struct is %d\n", sizeof (my_struct));
}

int main (void)
{
        align_1 ();

        return 0;
}
```

在编译并运行这段代码之前，读者可以首先推断一下该程序的正确输出。然后，我们在 PC 上运行这个程序看看它的实际结果：

```
user$ gcc -o align align.c
user$ ./align
The size of my_struct is 8
```

许多人会对此结果有疑问，然而，如果我们不作任何的特殊设置，这个结构的大小的确是 8。为什么会这样呢？这是因为，在 32 位处理器上，在存放多字节操作数时，编译器会根据操作数的大小确保在内存空间中该操作数对齐于地址边界。比如在上面这个结构中，`char c` 成员是个单字节的成员，这个成员可被保存在任意的地址；而 `int i` 成员的大小是 4 字节，编译器就会确保将 `i` 保存在地址为 4 的倍数的位置上。因此，上述结构在内存中存储时，它的实际内存布局不是我们想象的 `c` 占用一个字节，然后立即是 4 个字节的 `i`，而是像图 3-5 那样。

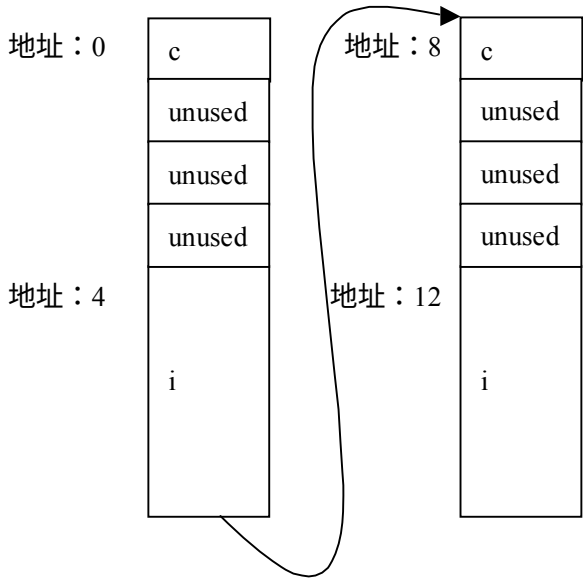

图 3-5 my_struct 结构数组的内存布局

编译器这样做的目的有两个：

- 根据操作数的大小和地址对齐存放操作数，可确保对该操作数的二进制运算速度最快，这是硬件设计决定的。
- 某些处理器指令在处理不对齐的操作数时，会出现处理器异常，从而导致总线错误[23]。

依此类推，我们可以看到下面这个结构的大小也是 8：

```
struct {
        char c;
        short s;
        int i;
} my_struct;
```

【思考题】下面这个结构的大小为多少字节？

```
struct {
        int i;
        short s;
        char c;
} my_struct;
```

另外，编译器通常提供某种方法，以便取消定义结构时的成员对齐。在 gcc 中，我们可以用*__attribute__ ((packed))*修饰词取消结构内部的成

[23] 在支持虚拟内存的操作系统，比如 Linux 中，因为内核会捕捉这种处理器异常并作适当的处理，从而在嵌入式 Linux 系统上，对齐问题表现不是非常突出，而在 uClinux 上却很明显。

员对齐：

```
struct __attribute__ ((packed)) {
                int i;
                short s;
                char c;
} my_struct;
```

为了避免因为对齐给我们带来麻烦，有的情况下我们需要仔细编码以便确保程序的正确性。比如，下面的程序试图将内存中的一个 32 位整数值取出来：

```
unsigned int read_uint_from_mem (const unsigned char* data)
{
    unsigned int u;
    memcpy (&u, data, sizeof (unsigned int));
    return u;
}
```

这段程序看起来没有什么问题，我们甚至可以这样编码：

```
unsigned int read_uint_from_mem (const unsigned char* data)
{
    unsigned int *u;
    u = (unsigned int*)data;
    return *u;
}
```

然而，上面的两个程序段，当传入的 data 地址不在 4 字节边界上对齐的时候，就会出现问题。读者可以在 SkyEye 和 Xcopilot 模拟器上测试上面的程序段，用下面的方式调用：

```
        unsigned char data [1024];
        unsigned char u;

       u = read_uint_from_mem (data + 1);
```

我们会发现，在 uClinux（即没有虚拟内存支持时）系统，上述代码始终会给出总线错误，从而导致程序异常退出。正确的 *read_uint_from_mem* 函数应该如下设计，才能确保避免因为对齐造成程序错误：

```
unsigned int read_uint_from_mem (const unsigned char* data)
{
    Uint32 q1, q2, q3, q4;

    q1 = data[0];
    q2 = data[1];
    q3 = data[2]
    q4 = data[3]
    return ((q1<<24)|(q2<<16)|(q3<<8)|(q4));
}
```

【思考题】上述代码段对小头系统和大头系统均适用吗？

3.3.4 浮点数运算

前面已经说过，在许多嵌入式处理器上因为缺少浮点数协处理器，程序中的大量浮点运算会造成大的程序性能瓶颈。为此，只要可能，我们就应该用整数运算替代浮点数运算。当然，用整数运算替代浮点数运算，肯定会造成精度丧失，然而，许多情况下我们可以通过一些数学技巧确保精度在可接受范围内。

作为示例，我们举例解决嵌入式系统中经常会遇到的问题：触摸屏的线性校正。

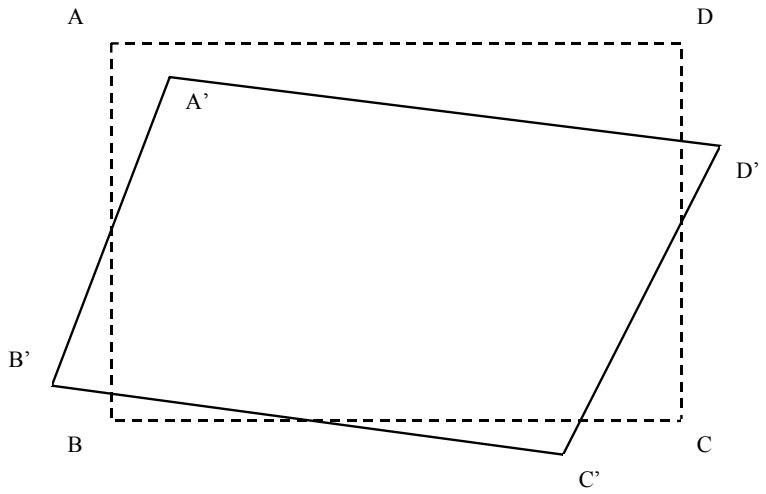

图 3-6 触摸屏的线性校正

首先，我们了解一下触摸屏线性校正的数学模型。如图 3-6 所示，ABCD 是触摸屏的安装位置，但由于触摸屏本身的物理缺陷，或者由于触摸屏安装的原因，当用户在触摸屏上点击时，获得的数据却可能是 A’B’C’D’空间范围内的数据。举例来说，当用户在 A 点点击时，他应该获得 （0,0）点的坐标，然而，触摸屏驱动程序返回的却是 A’的坐标（10,10）。这种情况下，用户无法通过触摸屏正确操作显示屏上的用户界面。为解决这个问题，我们可以在理论空间和实际空间之间建立一种线性的校正算法，使得当我们从触摸屏驱动程序那里获得 A’的坐标时，通过线性校正计算，返回 A 点的坐标。为此，通常的做法如下：

- 假定理论空间和实际空间之间的误差是线性的。
- 为了便于计算，我们将触摸屏坐标空间中取五个点（矩形的四个角和中点），并划分为四个三角形，如图 3-7 所示。其中给出了理论三角形（虚线）和实际三角形（实线）。
- 根据线性假设，理论三角形和实际三角形之间的线性关系可能为旋转和偏移，从而可以建立这两个三角形之间的线性变换矩阵。当我们求得这个变换矩阵之后，就可以从实际得到的数据中推算出理论数据。
- 对上述四对三角形，我们可以求得四个变换矩阵。已知一个实际点的坐标，我们可以分别用这四个变换矩阵求得四个理论值，然后对这四个理论值取平均，就可以获得唯一的理论值。

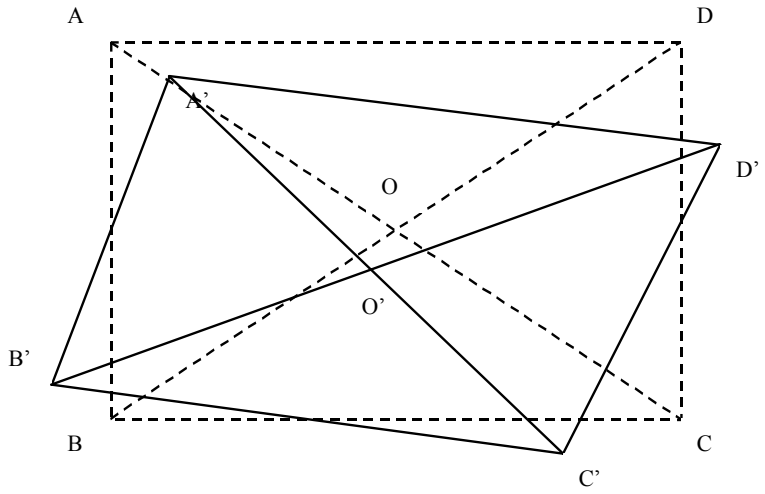

图 3-7 触摸屏的线性校正算法模型

上面就是这触摸屏线性校正算法的数学基础。从一个实际三角形（$x_1,y_1;x_2,y_2;x_3,y_3$）到一个理论三角形（$x'_1,y'_1;x'_2,y'_2;x'_3,y'_3$），我们可以建立如下的线性关系（旋转和偏移）：

$$\begin{cases} x'_1 = ax_1 + by_1 + c \\ y'_1 = dx_1 + ey_1 + f \\ x'_2 = ax_2 + by_2 + c \\ y'_2 = dx_2 + ey_2 + f \\ x'_3 = ax_3 + by_3 + c \\ y'_3 = dx_3 + ey_3 + f \end{cases}$$

采用高斯消元法，可求解上述方程，得到下面的公式：

$$
\begin{cases}
a = \dfrac{(x_1' - x_2')(y_2 - y_3) - (x_2' - x_3')(y_1 - y_2)}{(x_1 - x_2)(y_2 - y_3) - (x_2 - x_3)(y_1 - y_2)} \\
b = \dfrac{(x_1' - x_2')(x_2 - x_3) - (x_2' - x_3')(x_1 - x_2)}{(y_1 - y_2)(x_2 - x_3) - (y_2 - y_3)(x_1 - x_2)} \\
c = x_1' - ax_1 - by_1 \\
d = \dfrac{(y_1' - y_2')(y_2 - y_3) - (y_2' - y_3')(y_1 - y_2)}{(x_1 - x_2)(y_2 - y_3) - (x_2 - x_3)(y_1 - y_2)} \\
e = \dfrac{(y_1' - y_2')(x_2 - x_3) - (y_2' - y_3')(x_1 - x_2)}{(y_1 - y_2)(x_2 - x_3) - (y_2 - y_3)(x_1 - x_2)} \\
f = y_1' - dx_1 - ey_1
\end{cases}
$$

有了上面的数学结论，我们就可以按照上面解释过的触摸屏线性校正算法编写用于触摸屏校正的 C 程序代码。首先，我们编写下面的函数用来求解上述方程中的 *a*、*b*、*c*、*d*、*e*、*f*：

```
static BOOL doGaussianElimination_f (double* x, const POINT* src_pts, const POINT* dst_pts)
{
    double x12, x23, y12, y23, nx12, nx23, ny12, ny23;
    double numerator, denominator1, denominator2;

    x12 = (double)(src_pts [0].x - src_pts [1].x);
    x23 = (double)(src_pts [1].x - src_pts [2].x);
    y12 = (double)(src_pts [0].y - src_pts [1].y);
    y23 = (double)(src_pts [1].y - src_pts [2].y);

    nx12 = (double)(dst_pts [0].x - dst_pts [1].x);
    nx23 = (double)(dst_pts [1].x - dst_pts [2].x);
    ny12 = (double)(dst_pts [0].y - dst_pts [1].y);
    ny23 = (double)(dst_pts [1].y - dst_pts [2].y);

    denominator1 = x12*y23 - x23*y12;
    if (fabs (denominator1) < 0.0E-10)
        return FALSE;

    denominator2 = y12*x23 - y23*x12;
    if (fabs (denominator2) < 0.0E-10)
        return FALSE;

    numerator = nx12*y23 - nx23*y12;
    x [0] = numerator / denominator1;
    numerator = nx12*x23 - nx23*x12;
    x [1] = numerator / denominator2;
    x [2] = dst_pts [0].x - x [0] * src_pts [0].x - x [1] * src_pts [0].y;

    numerator = ny12*y23 - ny23*y12;
    x [3] = numerator / denominator1;
    numerator = ny12*x23 - ny23*x12;
    x [4] = numerator / denominator2;
    x [5] = dst_pts [0].y - x [3] * src_pts [0].x - x [4] * src_pts [0].y;

    return TRUE;
}
```

上面的函数根据传入的三对实际点（*src_pts*）和理论点（*dst_pts*）计算 *a*、*b*、*c*、*d*、*e*、*f* 六个值，并保存在 *x* 指向的数组中。

接下来我们编写求解校正参数以及完成校正的函数：

```
DO_MOUSE_CALIBRATE_PROC __mg_do_mouse_calibrate;

#define NR_EQUATIONS    6

static double vars1 [NR_EQUATIONS], vars2 [NR_EQUATIONS], vars3 [NR_EQUATIONS], vars4 [NR_EQU
ATIONS];

static void do_mouse_calibrate (int* x, int* y)
{
    double x1, y1, x2, y2, x3, y3, x4, y4;

    x1 = vars1 [0] * (*x) + vars1 [1] * (*y) + vars1 [2];
    y1 = vars1 [3] * (*x) + vars1 [4] * (*y) + vars1 [5];

    x2 = vars2 [0] * (*x) + vars2 [1] * (*y) + vars2 [2];
    y2 = vars2 [3] * (*x) + vars2 [4] * (*y) + vars2 [5];

    x3 = vars3 [0] * (*x) + vars3 [1] * (*y) + vars3 [2];
    y3 = vars3 [3] * (*x) + vars3 [4] * (*y) + vars3 [5];

    x4 = vars4 [0] * (*x) + vars4 [1] * (*y) + vars4 [2];
    y4 = vars4 [3] * (*x) + vars4 [4] * (*y) + vars4 [5];

    *x = (int)((x1 + x2 + x3 + x4)/4.0 + 0.5);
    *y = (int)((y1 + y2 + y3 + y4)/4.0 + 0.5);
}

BOOL SetMouseCalibrationParameters (const POINT* src_pts, const POINT* dst_pts)
{
    POINT my_src_pts [3];
    POINT my_dst_pts [3];

    my_src_pts [0] = src_pts [0];
    my_src_pts [1] = src_pts [1];
    my_src_pts [2] = src_pts [4];
    my_dst_pts [0] = dst_pts [0];
    my_dst_pts [1] = dst_pts [1];
    my_dst_pts [2] = dst_pts [4];
    if (!doGaussianElimination_f (vars1, my_src_pts, my_dst_pts))
        return FALSE;

    my_src_pts [0] = src_pts [1];
    my_src_pts [1] = src_pts [2];
    my_src_pts [2] = src_pts [4];
    my_dst_pts [0] = dst_pts [1];
    my_dst_pts [1] = dst_pts [2];
    my_dst_pts [2] = dst_pts [4];
    if (!doGaussianElimination_f (vars2, my_src_pts, my_dst_pts))
        return FALSE;

    my_src_pts [0] = src_pts [2];
    my_src_pts [1] = src_pts [3];
    my_src_pts [2] = src_pts [4];
    my_dst_pts [0] = dst_pts [2];
    my_dst_pts [1] = dst_pts [3];
    my_dst_pts [2] = dst_pts [4];
    if (!doGaussianElimination_f (vars3, my_src_pts, my_dst_pts))
        return FALSE;

    my_src_pts [0] = src_pts [0];
    my_src_pts [1] = src_pts [3];
    my_src_pts [2] = src_pts [4];
    my_dst_pts [0] = dst_pts [0];
    my_dst_pts [1] = dst_pts [3];
    my_dst_pts [2] = dst_pts [4];
    if (!doGaussianElimination_f (vars4, my_src_pts, my_dst_pts))
        return FALSE;

    __mg_do_mouse_calibrate = do_mouse_calibrate;
    return TRUE;
}
```

其中，*do_mouse_calibrate* 函数根据传入的实际坐标点求解理论坐标点，所使用的 *vars1* 到 *vars4* 分别是四对三角形线性关系的 *a*～*f* 参数。*vars1* 到 *vars4* 由 *SetMouseCalibrationParameters* 函数通过高斯消元法计

算得到。该函数将传入的五个点（矩形的四个角及中点）组成了四个三角形，然后调用 *doGaussianElimination_f* 函数四次，从而求得 *vars1* 到 *vars4* 四个数组。注意，这里的 *src_pts* 应该是实际坐标值，即从触摸屏驱动程序那里获得的值，*dst_pts* 是理论坐标值，即校正后的坐标值。

在实际的系统中，我们可在系统启动时，显示一个简单的界面提示用户点击屏幕上的五个点，画这五个点的位置就是理论坐标值，而当用户点击时从触摸屏驱动程序那里获得的值就是实际坐标值。有了这五个点的数据，我们调用 *SetMouseCalibrationParameters* 函数，如果传入的参数不溢出，则可进行线性校正，并将*__mg_do_mouse_calibrate* 函数指针的值设置为 *do_mouse_calibrate* 这个函数（在没有设置线性校正之前，这个函数指针的值为 NULL）。之后，当我们再从触摸屏驱动程序那里获得坐标值时，即可以调用 *__mg_do_mouse_calibrate* 函数将实际值转换为理论值。

MiniGUI 根据上面的数学原理提供了触摸屏的线性校正函数接口。但是，上述函数的实现中使用了大量的浮点数运算，这将导致一定程度上的性能损失。如果我们分析这个触摸屏校正算法，我们会发现：

- 我们传入的是整数参数，最后获得的也是整数的结果。
- 我们要获得的校正结果并不需要非常精确。对用户在触摸屏上的点击操作，得到(10,10) 和得到 (9,9)这样的点击坐标，并不会导致严重的问题。

基于上述考虑，我们在实现触摸屏的校正算法时，根本没有必要带来额外开销的浮点数运算。在 MiniGUI 中，我们完全用整数运算实现了上述函数：

```
DO_MOUSE_CALIBRATE_PROC __mg_do_mouse_calibrate;

#define NR_EQUATIONS    6
#define LSHIFT(x)       ((x)<<10)
#define RSHIFT(x)       ((x)>>12)

static int vars1 [NR_EQUATIONS], vars2 [NR_EQUATIONS], vars3 [NR_EQUATIONS], vars4 [NR_EQUATI
ONS];

static void do_mouse_calibrate (int* x, int* y)
{
    int x1, y1, x2, y2, x3, y3, x4, y4;

    x1 = vars1 [0] * (*x) + vars1 [1] * (*y) + vars1 [2];
    y1 = vars1 [3] * (*x) + vars1 [4] * (*y) + vars1 [5];

    x2 = vars2 [0] * (*x) + vars2 [1] * (*y) + vars2 [2];
    y2 = vars2 [3] * (*x) + vars2 [4] * (*y) + vars2 [5];

    x3 = vars3 [0] * (*x) + vars3 [1] * (*y) + vars3 [2];
    y3 = vars3 [3] * (*x) + vars3 [4] * (*y) + vars3 [5];

    x4 = vars4 [0] * (*x) + vars4 [1] * (*y) + vars4 [2];
```

```
    y4 = vars4 [3] * (*x) + vars4 [4] * (*y) + vars4 [5];

    *x = RSHIFT (x1 + x2 + x3 + x4);
    *y = RSHIFT (y1 + y2 + y3 + y4);
}

static BOOL doGaussianElimination (int* x, const POINT* src_pts, const POINT* dst_pts)
{
    int x12, x23, y12, y23, nx12, nx23, ny12, ny23;
    int numerator, denominator1, denominator2;

    x12 = (src_pts [0].x - src_pts [1].x);
    x23 = (src_pts [1].x - src_pts [2].x);
    y12 = (src_pts [0].y - src_pts [1].y);
    y23 = (src_pts [1].y - src_pts [2].y);

    nx12 = (dst_pts [0].x - dst_pts [1].x);
    nx23 = (dst_pts [1].x - dst_pts [2].x);
    ny12 = (dst_pts [0].y - dst_pts [1].y);
    ny23 = (dst_pts [1].y - dst_pts [2].y);

    denominator1 = x12*y23 - x23*y12;
    if (denominator1 == 0)
        return FALSE;

    denominator2 = y12*x23 - y23*x12;
    if (denominator2 == 0)
        return FALSE;

    numerator = nx12*y23 - nx23*y12;
    x [0] = LSHIFT (numerator) / denominator1;
    numerator = nx12*x23 - nx23*x12;
    x [1] = LSHIFT (numerator) / denominator2;
    x [2] = LSHIFT (dst_pts [0].x) - x [0] * src_pts [0].x - x [1] * src_pts [0].y;

    numerator = ny12*y23 - ny23*y12;
    x [3] = LSHIFT (numerator) / denominator1;
    numerator = ny12*x23 - ny23*x12;
    x [4] = LSHIFT (numerator) / denominator2;
    x [5] = LSHIFT (dst_pts [0].y) - x [3] * src_pts [0].x - x [4] * src_pts [0].y;

    return TRUE;
}

BOOL SetMouseCalibrationParameters (const POINT* src_pts, const POINT* dst_pts)
{
    POINT my_src_pts [3];
    POINT my_dst_pts [3];

    my_src_pts [0] = src_pts [0];
    my_src_pts [1] = src_pts [1];
    my_src_pts [2] = src_pts [4];
    my_dst_pts [0] = dst_pts [0];
    my_dst_pts [1] = dst_pts [1];
    my_dst_pts [2] = dst_pts [4];
    if (!doGaussianElimination (vars1, my_src_pts, my_dst_pts))
        return FALSE;

    my_src_pts [0] = src_pts [1];
    my_src_pts [1] = src_pts [2];
    my_src_pts [2] = src_pts [4];
    my_dst_pts [0] = dst_pts [1];
    my_dst_pts [1] = dst_pts [2];
    my_dst_pts [2] = dst_pts [4];
    if (!doGaussianElimination (vars2, my_src_pts, my_dst_pts))
        return FALSE;

    my_src_pts [0] = src_pts [2];
    my_src_pts [1] = src_pts [3];
    my_src_pts [2] = src_pts [4];
    my_dst_pts [0] = dst_pts [2];
    my_dst_pts [1] = dst_pts [3];
    my_dst_pts [2] = dst_pts [4];
    if (!doGaussianElimination (vars3, my_src_pts, my_dst_pts))
        return FALSE;

    my_src_pts [0] = src_pts [0];
    my_src_pts [1] = src_pts [3];
    my_src_pts [2] = src_pts [4];
    my_dst_pts [0] = dst_pts [0];
    my_dst_pts [1] = dst_pts [3];
```

```
    my_dst_pts [2] = dst_pts [4];
    if (!doGaussianElimination (vars4, my_src_pts, my_dst_pts))
        return FALSE;

    __mg_do_mouse_calibrate = do_mouse_calibrate;
    return TRUE;
}
```

阅读上述代码段，我们可以看到如下变化：

- *doGaussianElimination* 函数在整数基础上进行运算，得到的是整数的 *vars1* 到 *vars4* 参数。
- 为了减小精度损失，我们将 *vars1* 到 *vars4* 的值在运算时首先通过左移 10 次而增加到了 2^{10} 倍。这通过 *LSHIFT* 宏实现。
- 在调用 *do_mouse_calibrate* 函数计算校正后的理论值时，我们也在整数基础上进行了运算，但相反，最终结果要通过右移缩小相应的倍数，加上我们还要取均值，因此我们右移了 12 次。

上述方法不仅得到了好的运算性能，同时也得到了其他两个好处：

- *vars1* 到 *vars4* 数组占用的空间缩小为原来的一半（double 型数据占用 8 字节，而 int 型数据占用 4 个字节）。
- 在判断高斯消元法是否有解时，浮点数版本只能对比分母是否小于某个足够小的实数（`fabs (denominator2) < 0.0E-10`），而整数版本只需判断分母是否为零（`denominator1 == 0`）。

上面这个例子在许多嵌入式系统中都可能会遇到，通过分析具体的数学问题，我们可以在很大程度上获得一个可以接受的结果，并且使得程序代码的性能提高。读者可以在自己的实践中参照上述方法解决实际问题。

更为一般的解决办法是采用定点数运算来替代浮点数运算。定点数实质上是整数，它始终用整数中给定不变的位数来表示一个实数的整数部分，然后用其余的位数来表示实数的小数部分；而浮点数的整数部分和小数部分所占的位数会根据实数的值发生变化。这样，如果用定点数来表示实数，则四则运算可用整数的四则运算来表示，其他的非线性运算（比如平方根、立方根、三角运算等）则可以用查表法获得。在 MiniGUI 中，专门定义了定点数类型及定点数的四则运算及常用非线性运算，其规则可简单描述如下。

首先，定点数用有符号的 32 位整数来表示，可用来表示从 -32767.0 到 32767.0 范围内的实数，它用高 16 位表示符号及实数的整数部分，用低 16 位表示小数部分。这样，浮点数和定点数之间的相互转换的运算如下：

```
static inline fixed ftofix (double x)
{
   if (x > 32767.0) {
      errno = ERANGE;
      return 0x7FFFFFFF;
   }

   if (x < -32767.0) {
      errno = ERANGE;
      return -0x7FFFFFFF;
   }

   return (long)(x * 65536.0 + (x < 0 ? -0.5 : 0.5));
}

static inline double fixtof (fixed x)
{
   return (double)x / 65536.0;
}
```

整数和定点数之间的转换运算如下：

```
static inline fixed itofix (int x)
{
   return x << 16;
}

static inline int fixtoi (fixed x)
{
   return (x >> 16) + ((x & 0x8000) >> 15);
}
```

这样，定点数的加减运算就变成了整数的加减运算。

其次，三角运算等非线性运算采用查表法，这是需要采用角度值而非弧度值（因为弧度值有效范围在 0~2π 时，用上述定点数规则表示的精度太低）。

如果读者对上述定点数的运算感兴趣，可参阅 MiniGUI 源代码及《MiniGUI 编程指南》12.6“定点数运算”一节。

3.4 小结

本章我们为读者介绍了如何在嵌入式 Linux/uClinux 上运行简单的应用程序，并介绍了嵌入式应用程序开发中需要重点注意的一些问题，比如字节序、对齐、浮点数运算等。通过本章的学习，读者应该掌握应用程序的交叉编译方法，应该学会使用 make 和 makefile 来组织和维护嵌入式应用工程。

接下来，我们将通过在嵌入式 Linux/uClinux 上运行 MiniGUI 来阐述将大型嵌入式应用软件移植到嵌入式 Linux/uClinux 上所面临的问题及解决办法。

4 在 Linux/uClinux 上运行 MiniGUI

通过前面章节的阐述，读者应该已经对嵌入式 Linux/uClinux 上的应用软件开发有了一定的了解，也掌握了基本的原理及工具的使用。

本章起，我们将主要讲述如何在 Linux/uClinux 上运行 MiniGUI。通过讲述 MiniGUI 及其应用程序的开发，并最终在 SkyEye 及 Xcopilot 模拟器上运行 MiniGUI 的应用程序，读者将进一步掌握嵌入式开发工具的使用，也将了解 MiniGUI 应用程序的开发方法。

4.1 了解 MiniGUI

为了读者能够对 MiniGUI 有一个感性认识，我们首先运行一下 MiniGUI 在 Windows 平台上的演示程序。

4.1.1 在 Windows 平台上运行 MiniGUI

访问飞漫软件网站，从如下地址下载 MiniGUI 在 Windows 平台上的演示程序包[24]（*mgdemo-win32-1.6.rar*）：

http://www.minigui.com/download/cindex.shtml[25]

将上述演示程序包解开到 Windows 平台上的任意目录下，然后根据 README.txt 文件中的描述首先运行 *wvfb.exe* 程序，然后运行 *mgdemo_win32.exe* 程序。我们将在 wvfb 程序窗口中看到 MiniGUI 演示

[24] 该演示程序包的完整源代码保存在产品光盘的 */minigui/mgdemo-1.6.0.tar.gz* 中。该源代码包可在 Win32 平台和运行 Linux 的 PC 上编译，编译时，请使用上述目录中的两个开发工具包。
[25] 本章提到的示例程序包，保存在产品光盘的 */minigui* 目录中。

程序的输出效果，如图 4-1 所示。

图 4-1MiniGUI 在 Windows 平台上的运行

初看起来，MiniGUI 在 Windows 平台上的运行效果和普通的 Windows 程序没有什么区别。但本质就在这里，MiniGUI 其实就是为嵌入式操作系统提供类似 Windows 这样的图形支撑的系统，它使得在嵌入式系统上开发类似 Windows 的图形用户界面应用程序成为可能。

这里，wvfb 模拟了嵌入式系统上的 LCD 屏幕，鼠标就相当于触摸笔。MiniGUI 的应用程序不需要了解它到底运行在 Windows 上的 wvfb 窗口中，还是运行在实际的 LCD 屏幕上，也不需要知道用户到底是通过鼠标操作程序还是通过触摸屏操作程序，它只要调用 MiniGUI 的接口接收用户输入、完成绘图和文字输出即可。为了更进一步了解 MiniGUI 给上层应用程序提供的接口，我们接下来在 Windows 上开发一个简单的 MiniGUI 应用程序。

4.1.2 在 Windows 平台上开发 MiniGUI 应用程序

为了在 Windows 平台上开发 MiniGUI 应用程序，您需要在 Windows 上安装 MS Visual Studio 98 集成开发工具。

访问飞漫软件网站，从如下地址下载 MiniGUI 在 Windows 平台上的开发包（*minigui-dev-1.6-win32.rar*）：

http://www.minigui.com/download/cindex.shtml

将上述开发包解开到 Windows 平台上的任意目录下，然后根据 *README* 文件中的描述打开 helloworld 工程，如图 4-2 所示。

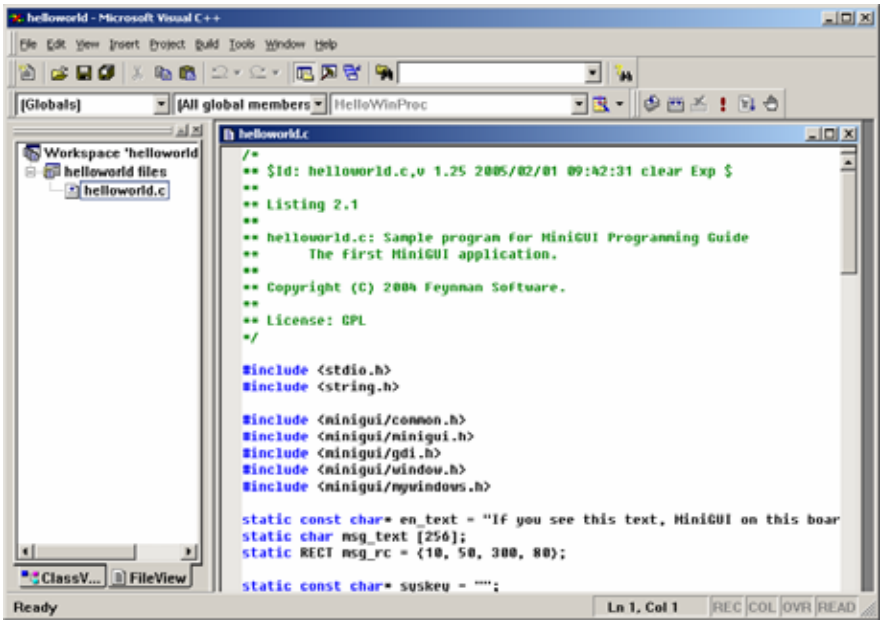

图 4-2 打开 MiniGUI 的 helloworld 工程

编译后，需首先运行 wvfb 程序，然后运行 hellowolrd 程序（注意需要将 *helloworld.exe* 文件复制到 *dll* 目录中，以便程序能够在当前目录下找到 *minigui.dll* 等 DLL 文件），将在 wvfb 中看到运行效果，如图 4-3 所示。

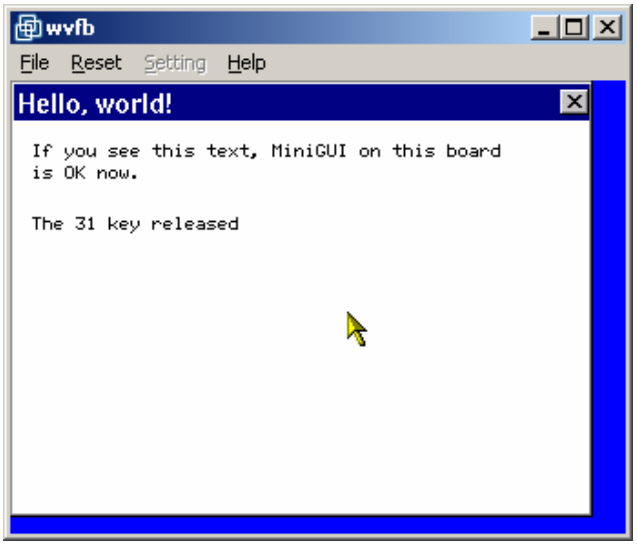

图 4-3 在 Windows 平台上编译并运行 MiniGUI 应用程序

如果读者仔细阅读 MiniGUI 的 *helloworld.c* 源代码的话，就会发现，MiniGUI 应用程序的代码结构和 Win32 程序非常相似：比如消息循环、设备上下文等概念。实际上，MiniGUI 的最初就是参考 Windows 的接口设计的，因此，有许多概念非常类似。如果读者想了解 MiniGUI 编程的细节信息，请从飞漫软件网站下载最新的《MiniGUI 编程指南》，其中详细讲述了 MiniGUI 应用程序开发所涉及的概念及主要接口。

在嵌入式系统中，MiniGUI 位于操作系统之上，应用程序之下，它为图形应用程序提供窗口、控件、事件管理、图形设备接口等相关接口。我们可以依照上述方法在 Windows 环境中开发和调试 MiniGUI 应用程序，然后将该应用程序移植到目标平台上。

当然，针对嵌入式 Linux/uClinux 的 MiniGUI 应用软件开发，我们推荐的方法是：

第1步. 在运行 Linux 的 PC 机上安装 MiniGUI，建立 MiniGUI 的运行环境。

第2步. 在 PC 上开发并调试 MiniGUI 应用程序。

第3步. 为目标系统编写 MiniGUI 的输入引擎，确保在目标系统上能够运行 MiniGUI。

第4步. 在目标系统上运行并测试 MiniGUI 应用软件。

接下来，我们将详细描述这些步骤。

4.2 在运行 Linux 的 PC 机上安装并运行 MiniGUI-STR

MiniGUI-STR 是 MiniGUI 学习版产品随附的 MiniGUI 源代码包。它是 MiniGUI 增值版（MiniGUI-VAR）产品的简化版本，只支持 Linux 和 uClinux 操作系统，并且取消了许多不常用的配置选项，更加适合初学者学习和研究。关于 MiniGUI-STR 和 MiniGUI-VAR 之间的功能区别，读者可参阅本书附录 A“ MiniGUI-STR 和 MiniGUI-VAR 的功能差异 ”。

本节将讲述如何在运行 Linux 的 PC 机上配置、安装及运行

MiniGUI-STR。之后，我们将把 MiniGUI 移植到运行 Linux 操作系统、模拟 EP7312 开发板的 SkyEye 模拟器上，最后我们还将讲述如何在运行 uClinux 的 Xcopilot 模拟器上运行 MiniGUI-STR 示例程序。

4.2.1 MiniGUI-STR 的组成

MiniGUI-STR 由源代码包、资源包、演示程序包三个部分组成：

1) *libminigui-str-1.6.2.tar.gz*。MiniGUI-STR V1.6.2 的核心源代码包，编译后可生成 MiniGUI 函数库。
2) *minigui-res-str-1.6.tar.gz*。这是针对 MiniGUI-STR 的基本资源包，其中包含了运行 MiniGUI-STR 1.6.2 需要的基本字体（ISO8859-1）、鼠标光标、图标和位图等文件，还包括 16 点阵中文简体（GB2312）一款。
3) *mg-samples-str-1.6.2.tar.gz*。这是针对 MiniGUI-STR 的示例程序包，其中包含了《MiniGUI 编程指南》所使用示例程序。因为 MiniGUI-STR 中缺少一些 MiniGUI 的高级特性，所以通常的 mg-samples 包无法在 MiniGUI-STR 上运行。
4) *mde-str-1.6.2.tar.gz*。这是针对 MiniGUI-STR 的演示程序包，其中包含了一些高级的 MiniGUI 演示程序。因为 MiniGUI-STR 中缺少一些 MiniGUI 的高级特性，所以通常的 mde 包无法在 MiniGUI-STR 上编译运行。

4.2.2 获得 MiniGUI-STR

您可以从北京飞漫软件技术有限公司网站的“下载”区（*http://www.minigui.com/download/cindex.shtml*）下载得到 MiniGUI-STR V1.6.2 版本以及对应的资源和示例程序包。

购买 MiniGUI 学习版产品的用户，也可从产品光盘的 *minigui/* 目录中获得上述四个软件包。产品光盘中还包含 *mgdemo-1.6.0.tar.gz* 包，这个包不能在 MiniGUI-STR 下编译运行，但可配合 MiniGUI 开发包（*minigui-dev-1.6-linux.tar.gz* 和 *minigui-dev-1.6-win32.rar*）编译和运行。

【注意】北京飞漫软件技术有限公司遵循 GPL 条款发布 MiniGUI-STR 1.6.x 函数库；遵循 GPL 条款发布 mg-samples-str 示例程序；minigui-res-str 等资源包中的文件，仅限于配合 MiniGUI 使用，不

得和 MiniGUI 分离并用于其它场合。如果您要在商用产品中使用 MiniGUI-STR，则必须从飞漫软件购买商业授权。

4.2.3 建立 MiniGUI 的 PC 运行环境

在运行 Linux 的 PC 上，MiniGUI 应用程序可以两种方式运行：

- 在 X Window 系统上，运行在虚拟 FrameBuffer 程序 qvfb 中；
- 在 Linux 的字符控制台上，运行在 Linux 内核提供的 FrameBuffer 驱动程序上。

不管是 qvfb 还是控制台下的 FrameBuffer 驱动程序，这两者的本质是一样的，即它们都为 MiniGUI 提供了一种可以用来绘图的底层设施，就像一块画布一样。下面我们分别描述如何准备这两种运行环境。

1) QVFB

QVFB 是 Qt 提供的一个虚拟 FrameBuffer 工具。这个程序基于 Qt 开发（Qt 是 Linux 窗口管理器 KDE 使用的底层函数库），运行在 X Window 上。您可以在 Qt 2 或者 Qt 3 源代码的 *src/tools* 目录中找到这个程序。因为读者很少有机会需要独立编译 Qt 函数库，因此，您可以从飞漫软件的“免费下载”区下载由飞漫打包的独立 QVFB 包（*qvfb-1.0.tar.gz*）：

http://www.minigui.com/download/c3rdparty.shtml

您可以下载这个软件包，按下面的命令单独编译 qvfb 程序：

```
user$ tar zxf qvfb-1.0.tar.gz
user$ cd qvfb-1.0
user$ ./configure
user$ make
```

上述命令将编译 qvfb 程序，为了使用方便，您可以将该程序安装到系统搜索路径中：

```
user$ su -c 'make install'
```

上述命令的含义是，以超级用户身份运行命令“*make install*”。该命令将提示用户输入超级用户密码，如果正确则会执行 *make install* 命令。*make install* 命令将把 qvfb 安装到默认的 */usr/local/bin* 目录下。因为

/usr/local/bin 目录存在于默认的搜索路径设置中，因此，我们可以在命令提示符处直接键入 *qvfb* 来运行 qvfb 程序。

在 X Window 环境中，我们可以打开一个终端仿真程序，然后执行 *qvfb* & 命令：

```
user$ qvfb &
```

上述命令将启动 qvfb 程序，我们会在屏幕上看到 qvfb 程序会建立一个黑色的空白窗口，该窗口就代表了一个可以用来在上面显示图像的显示屏。用户在该窗口中的鼠标点击，将被看成是在这个模拟的显示屏上的点击。我们还可以通过程序菜单打开 qvfb 的配置界面，以便按自己的需要设置不同的显示模式，从而可以让 qvfb 模拟具有不同显示能力（分辨率及可显示的颜色数等）的显示屏，如图 4-4 所示。

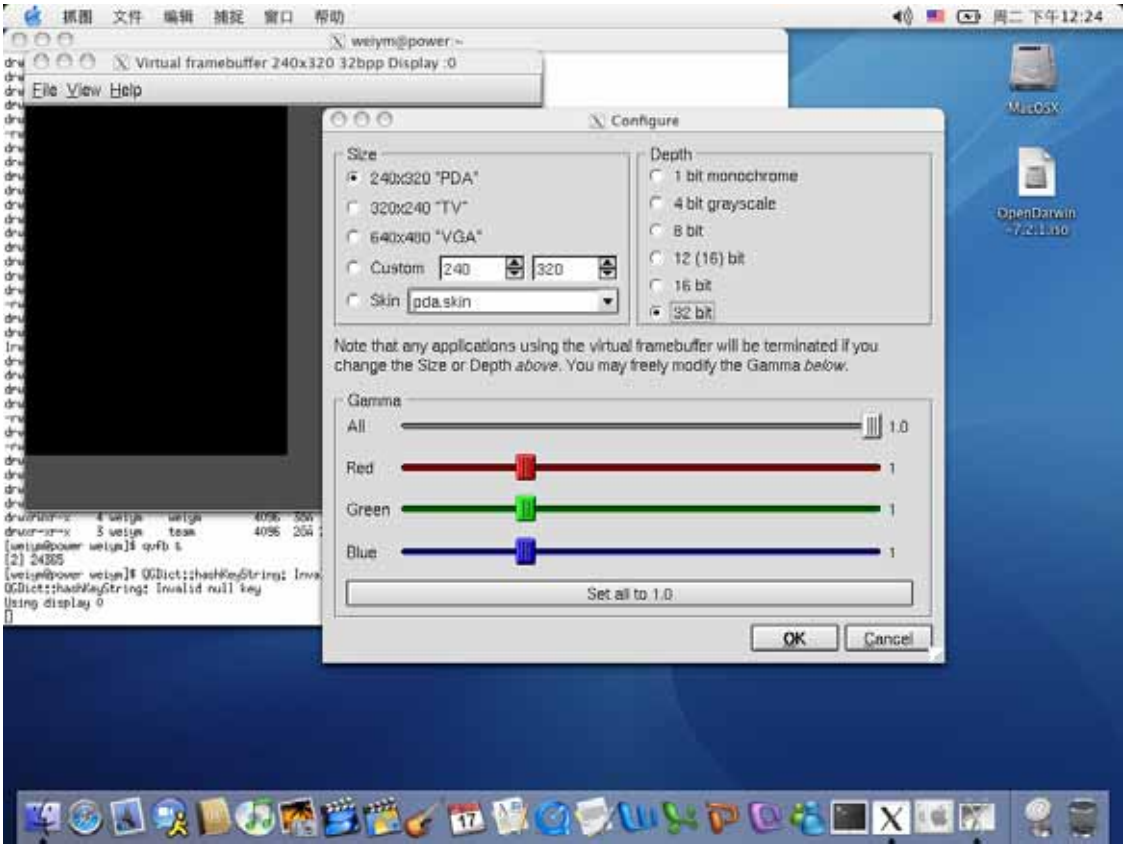

图 4-4 qvfb 程序及其配置界面

qvfb 给我们提供了一种软件方法，通过这种方法，我们可以在 PC 上看到自己的图形程序运行起来的效果，因为它能模拟不同的分辨率及颜色模式（单色、4 色、256 色、16 位色及 32 位色等），因此，它可以用来替

代实际的嵌入式硬件显示屏，从而大大方便应用程序的开发和调试。

2) 配置 Linux 内核的 FrameBuffer 驱动程序

FrameBuffer 是出现在 Linux 2.2.xx 内核当中的一种驱动程序接口。这种接口将显示设备抽象为帧缓冲区。用户可以将它看成是显示内存的一个映像，将其映射到进程地址空间之后，就可以直接进行读写操作，而写操作可以立即反应在屏幕上。该驱动程序的设备文件一般是 */dev/fb0*、*/dev/fb1* 等等。比如，假设现在的显示模式是 1024x768-8 位色，则可以通过如下的命令清空屏幕：

```
user$ dd if=/dev/zero of=/dev/fb0 bs=1024 count=768
```

在应用程序中，一般是通过将 FrameBuffer 设备映射到进程地址空间的方式来使用 FrameBuffer 的。比如下面的程序打开了 */dev/fb0* 设备，并通过 *mmap* 系统调用进行地址映射，随后用 *memset* 将屏幕清空（这里假设显示模式是 1024x768-8 位色模式，线性内存模式）：

```
int fb;
unsigned char* fb_mem;

fb = open ("/dev/fb0", O_RDWR);
fb_mem = mmap (NULL, 1024*768, PROT_READ|PROT_WRITE,MAP_SHARED,fb,0);
memset (fb_mem, 0, 1024*768);
```

这种简单的编程模型使我们可以在应用程序中通过打开 FrameBuffer 设备、然后向这个设备写入数据的方式来在实际的显示屏上画出图形。

然而，为了使用 Linux 的 FrameBuffer 驱动程序，我们首先要激活 FrameBuffer 驱动程序。在 PC 机上，这一工作相对容易和简单，因为大多数的 PC 显示卡是 VESA 兼容的。VESA 是一种标准，它为 PC 的显示卡制定了一个统一的接口，通过该接口，不论显示卡采用的是什么芯片组，只要它是 VESA 兼容的，我们就可以用一样的方法设置显示模式、获得显示内存地址等等。因为 VESA 兼容显卡使用得如此普遍，因此，Linux 内核中包含有针对 VESA 兼容显卡的 FrameBuffer 驱动程序。如果您的显卡是 VESA 兼容的，则需要完成如下两个工作来激活 VESA FrameBuffer 驱动程序：

- 确认自己的内核已经包含了 VESA FrameBuffer 驱动程序（2.2 以上的版本）。

- 配置 LILO 或者 GRUB 的启动选项，使内核启动时能切换到指定的显示模式。

目前流行的 Linux 发行版所带的内核，均会默认包含 VESA FrameBuffer 驱动程序。我们还可以运行 *fbset* 命令确认内核是否支持 FrameBuffer。若系统中没有安装 *fbset* 命令，可通过下面的方法确认内核中是否已包含了活动的 FrameBuffer 驱动程序：

- 查看 */proc/devices*，观察是否有 fb 设备；
- 运行 *cat /dev/fb0 > /dev/null* 命令，观察是否能够正常打开 */dev/fb0* 设备。

如果您使用的内核没有包含 VESA FrameBuffer 驱动程序，则需要重新检查 Linux 内核的配置，将 VESA FrameBuffer 驱动程序包含进去，然后重新编译内核。

当我们确认自己的内核包含有 VESA FrameBuffer 驱动程序后，可通过修改引导管理器（LILO或者GRUB）的引导参数，强制 Linux 内核在引导时激活 VESA FrameBuffer 驱动程序并进入指定的显示模式。注意，VESA FrameBuffer 驱动程序必须在内核引导时激活，在 Linux 内核的正常运行情况下，无法激活 VESA FrameBuffer 驱动程序或者动态改变其显示模式 [26]。

如果您使用 LILO 作为引导装载器，则需要修改 */etc/lilo.conf* 配置文件。下面是典型的 */etc/lilo.conf* 文件：

```
boot = /dev/hda2
timeout = 50
prompt
read-only

image = /boot/vmlinuz-2.4.16-22
    label = linux
    root = /dev/hda2
    other = /dev/hda1
```

假如您支持 VESA FrameBuffer 的内核映像名称为 */boot/vmlinuz-2.4.16-fb*，则可以在上述文件中增加如下描述（括号中的内容是注释，不用键入实际的配置文件）：

```

```

[26] 这是因为 VESA 接口必须在 386 处理器的实模式下访问，而 Linux 内核正常启动后，处理器工作在保护模式下。

```
image = /boot/vmlinuz-2.4.16-fb      # 新编译的支持 VESA FrameBuffer 内核
    label = linuxfb                  # 启动标号，可自定
    root = /dev/hda2                 # 您的根文件系统，具体会有不同
    vga = 0x317                      # VESA 显示模式号，参照下表
    other = /dev/hda1
```

保存所作的修改，然后以超级用户身份运行下面的命令：

```
root# lilo
```

该命令将使上述新的引导选项生效。

注意上面添加的 *vga=0x0317* 选项。该选项指定 VESA FrameBuffer 驱动程序在启动时将显示模式设置为 1024x768 16 位色，这里的 0x0317 是 VESA 标准定义的模式号。如果您的显示芯片或者显示器无法达到这个显示模式，则可以参照下表设置一个合理的模式号：

```
Colours   640x400 640x480 800x600 1024x768 1280x1024 1600x1200
  --------+-------------------------------------------------
   4 bits |    ?       ?     0x302      ?         ?         ?
   8 bits |  0x300   0x301   0x303    0x305     0x307     0x31C
  15 bits |    ?     0x310   0x313    0x316     0x319     0x31D
  16 bits |    ?     0x311   0x314    0x317     0x31A     0x31E
  24 bits |    ?     0x312   0x315    0x318     0x31B     0x31F
  32 bits |    ?       ?       ?        ?         ?         ?
```

如果您的系统使用 GRUB 内核装载器（引导器），则应该编辑 */boot/grub/menu.lst* 文件，复制某个已有的内核启动选项，并在 *kernel* 打头的一行末尾添加 *vga=0x0317* 选项，如下所示：

```
# grub.conf generated by anaconda
#
# Note that you do not have to rerun grub after making changes to this file
# NOTICE:  You do not have a /boot partition.  This means that
#          all kernel and initrd paths are relative to /, eg.
#          root (hd0,0)
#          kernel /boot/vmlinuz-version ro root=/dev/hda1
#          initrd /boot/initrd-version.img
#boot=/dev/hda
default=0
timeout=10
splashimage=(hd0,0)/boot/grub/splash.xpm.gz
title Red Hat Linux (2.4.18-3, FrameBuffer)
        root (hd0,0)
        kernel /boot/vmlinuz-2.4.18-3 ro root=/dev/hda1 vga=0x0317
        initrd /boot/initrd-2.4.18-3.img

title Red Hat Linux (2.4.18-3)
        root (hd0,0)
        kernel /boot/vmlinuz-2.4.18-3 ro root=/dev/hda1
        initrd /boot/initrd-2.4.18-3.img
```

使用 GRUB，无须像 LILO 那样在重新启动之前运行某个程序使选项

生效。

在设置好引导装载器之后，我们就可以重新启动 PC 机。这时，将出现 `lilo:` 提示符或者图形的启动画面，这时，可以选择支持 FrameBuffer 的选项。如果一切正常，则在 Linux 的引导过程中，会在屏幕的左上方显示一个可爱的小企鹅或者 Linux 发行版厂商的 LOGO 图片。

除了最通用的 VESA FrameBuffer 驱动程序之外，Linux 内核中还包含有一些常见显示芯片的专有 FrameBuffer 驱动程序。如果您的显示芯片恰好是 Linux 内核的 FrameBuffer 驱动程序所支持的，比如，ATI Mach64、ATI Radeon、NeoMagic、3Dfx Banshee/Voodoo3 等（详细列表可参见 Linux 内核的配置文件），则可以将对应的 FrameBuffer 驱动程序编译到内核，或者编译成模块，然后使用 fbset 程序，就可以灵活改变显示芯片所支持的显示模式。而且因为使用了显示芯片专有的 FrameBuffer 驱动程序，Linux 控制台上的字符刷新速度会明显好于 VESA FrameBuffer。其他关于 Linux FrameBuffer 的信息，可参阅 Framebuffer-HOWTO 文档 [27]。

至此，我们设置好了运行 MiniGUI 所需的两种显示设施。我们建议大多数用户使用 qvfb 程序，因为 qvfb 运行环境容易安装和使用，而控制台上的 FrameBuffer 驱动程序使用起来相对复杂。

我们还需要知道的是，大多数运行 Linux/uClinux 操作系统的嵌入式产品，均提供有 FrameBuffer 驱动程序，这使得我们在 PC 机和嵌入式系统上可以使用一样的程序代码来访问底层的显示设备，MiniGUI 在嵌入式 Linux/uClinux 上的运行也的确通过对 FrameBuffer 设备（*/dev/fb0* 等）的访问来建立一致的底层图形设施。

4.2.4 编译和安装 MiniGUI-STR

1) MiniGUI-STR 的图形引擎和输入引擎

在 MiniGUI 内部实现中，有两个重要的概念：图形和输入抽象层（Graphics and Input Abstract Layer，GAL 和 IAL）。抽象层的概念类似 Linux 内核虚拟文件系统的概念。它定义了一组不依赖于任何特殊硬件的抽象接口，所有顶层的图形操作和输入处理都建立在抽象接口之上，而用于实现这一抽象接口的底层代码称为“图形引擎”或“输入引擎”，类似操

[27] 关于 Linux 的绝大多数文档，可访问 http://www.tldp.org/ 获得。

作系统中的驱动程序。这实际是一种面向对象的程序结构。利用 GAL 和 IAL，MiniGUI 可以在许多已有的图形设施上运行，比如上面提到的 QVFB 和 Linux 内核的 FrameBuffer 驱动程序。

相比图形来讲，将 MiniGUI 的底层输入与上层相隔显得更为重要。在基于 Linux/uClinux 的嵌入式系统中，图形引擎可以通过 FrameBuffer 而获得，而输入设备的处理却没有统一的接口。在 PC 上，我们通常使用键盘和鼠标，而在嵌入式系统上，可能只有触摸屏和为数不多的几个按键。在这种情况下，提供一个抽象的输入层，就显得格外重要。

我们知道 MiniGUI 目前已经是一个跨操作系统的嵌入式图形用户界面支持系统，其中，图形和输入抽象层的设计，为实现 MiniGUI 跨操作系统、跨硬件平台提供了非常大的便利。因为大多数嵌入式 Linux/uClinux 平台均提供了统一的 FrameBuffer 接口，因此，在使用 Linux/uClinux 的嵌入式平台上，要运行 MiniGUI，通常我们只需要编写针对这种平台的特定输入引擎就可以了。在本章后面的小节，我们将讲述如何为特定的嵌入式平台编写 MiniGUI 的输入引擎。

在 MiniGUI-STR V1.6.2 中，包含有若干图形引擎和输入引擎，这些图形引擎和输入引擎的作用，可见表 4-1 和表 4-2。

表 4-1 MiniGUI-STR V1.6.2 中包含的图形引擎

图形引擎名称	用途	对应的配置参数	默认设置
qvfb	用来在 qvfb 上运行 MiniGUI 应用程序。	galqvfb	包含
fbcon	用来在 Linux/uClinux 的 FrameBuffer 设备之上运行 MiniGUI 应用程序。	galfbcon	包含

表 4-2 MiniGUI-STR V1.6.2 中包含的输入引擎

输入引擎名称	用途	对应的配置参数	默认配置
qvfb	将用户在 qvfb 窗口中输入的按键和鼠标信息翻译成 MiniGUI 能够识别的信息；用来在 qvfb 上运行 MiniGUI 应用程序。	qvfbial	包含
console	在 Linux 控制台的 FrameBuffer 设备之上运行 MiniGUI 应用程序时，使用该引擎可从鼠标和键盘中获得用户输入。	nativeial	包含
dummy	“哑”输入引擎。该引擎不依赖任何硬件设备，不产生任何输入。通常在移植 MiniGUI 时，输入引擎还没有准备好的情况下使用。	dummyial	包含
auto	“自动”输入引擎。该引擎不依赖任何硬件设备，和“哑”输入引擎一样，也属于一种软件输入引擎。但该引擎能够自动循环产生一些输入数据，可用于 MiniGUI 应用软件的自动测试。	autoial	不包含

ipaq	针对 Compaq iPAQ H3600/H3800 手持终端的输入引擎	ipaqial	不包含
arm3000	针对博创 ARM3000 开发板的输入引擎	arm3000ial	不包含
fft7202	针对傅立叶 7202 开发板的输入引擎	fft7202ial	不包含
hh2410r3	针对华恒 2410R3 开发板的输入引擎	hh2410r3ial	不包含
embest2410	针对英蓓特 2410 开发板的输入引擎	embest2410ial	不包含

2) MiniGUI-STR 的依赖库

MiniGUI 的某些特性需要一些第三方函数库的支持，比如对 JPEG、PNG 图片格式的支持，需要系统中存在 libjpeg 和 libpng 库。

在 MiniGUI 的配置过程中，如果配置脚本发现系统中缺少上述这些函数库，就会自动禁止依赖这些函数库的功能特性。因此，运行 MiniGUI 并不一定非要安装这些依赖库。MiniGUI-STR 的依赖函数库见表 4-3。

表 4-3 MiniGUI-STR 的依赖库

依赖库名称	用途	配置参数	默认设置
libjpeg	用来读取 JPEG 图片	jpgsupport	包含
libpng	用来读取 PNG 图片	pngsupport	包含

在常见的 Linux 发行版中，均已包含有上述这些函数库。

3) MiniGUI 的工程组织方式

在第 3 章，我们给大家介绍了通过 makefile 维护嵌入式应用程序工程的基本方法。尽管通过 makefile 文件可以组织一个应用软件工程项目，但往往手工编写一个 makefile 文件并不是一件轻松的事情，并且在需要维护大型的源代码目录树时，makefile 文件的维护工作就会大大增加。为此，GNU 又开发了 Autoconf/Automake 工具，可以用来自动生成 makefile 文件，并且能够检查系统的配置信息，从而帮助提供应用软件的可移植性。MiniGUI 工程（MiniGUI 库本身及示例程序包）就是通过 GNU 的 Automake/Autoconf 脚本组织的。

GNU 的 Autoconf 及 Automake 这两个软件实际是由若干 Shell 脚本组成的，它可以帮助程序员轻松产生 makefile 文件。现在的各种自由软件，如 Apache、MySQL 等都是利用 Autoconf/Automake 实现自动配置和编译的。MiniGUI 也采用了 Autoconf/Automake 接口。用户只要在源代码目录下使用 *./configure*、*make*、*make install* 这三条命令就可以把程序或函数库编译并安装到系统中。

利用 configure 脚本所产生的 makefile 文件有几个预先设定的目标可

供使用，这里只对其中几个简述如下：

- *make all* 产生设定的目标。只敲入 *make* 命令也可以，此时会开始编译源代码，然后连接并产生可执行文件或者函数库。
- *make clean* 清除之前所编译的目标文件（*.o）。
- *make distclean* 除了清除目标文件以外，也把 configure 所产生的 Makefile 清除掉。通常在发布软件前执行该命令。
- *make install* 将程序安装到系统中，若源码编译成功，且执行结果正确，便可以把程序安装到系统预先设定的执行文件存放路径中，若用 *bin_PROGRAMS* 宏的话，程序会被安装到 */usr/local/bin* 下。
- *make dist* 将程序和相关的文档包装为一个压缩包以供发布。执行该命令后，目录下会产生一个以 ***PACKAGE-VERSION***.*tar.gz* 为名称的文件。*PACKAGE* 和 *VERSION* 这两个参数是根据 *configure.in* 文件中 *AM_INIT_AUTOMAKE (**PACKAGE**, **VERSION**)* 宏定义的。
- *make distcheck* 和 *make dist* 类似，但是加入检查打包以后的压缩文件是否正常，这个目标除了把程序和相关文档打包成 ***PACKAGE-VERSION***.tar.gz 文件外，还会自动把这个压缩文件解开，执行 *configure* 和 *make all*，确认编译无错误以后，方显示这个 ***PACKAGE-VERSION***.*tar.gz* 文件已经准备好并可以发布了。

利用 Autoconf/Automake 所产生出来的软件包完全可以在没有安装 Autoconf/Automake 的环境中使用，这是因为 *configure* 是一个 Shell 脚本，它已被设计为可以在一般的 Linux/UNIX Shell 下执行。

我们还可以利用 MiniGUI 及其示例程序包的 configure 脚本生成可用于交叉编译的 makefile 文件，本章后面的小节将介绍如何利用 configure 脚本生成针对 SkyEye 和 Xcopilot 模拟器的交叉编译 makefile 文件。

4) 编译并安装 MiniGUI-STR

接下来我们编译并安装 MiniGUI-STR。首先，将 4.2.1 节所描述的软件包复制到自己的 PC 机上：

- *libminigui-str-1.6.2.tar.gz*
- *minigui-res-str-1.6.tar.gz*
- *mg-samples-str-1.6.2.tar.gz*

【注意】上述文件保存在学习版产品光盘的 /minigui/ 目录下。

用如下命令解开 *libminigui-str-1.6.2.tar.gz* 软件包：

```
user$ tar zxf <path to cdrom>/minigui/libminigui-str-1.6.2.tar.gz
```

该命令将在当前目录建立 *libminigui-str-1.6.2* 目录。进入该目录，并运行 *./configure* 命令：

```
user$ cd libminigui-str-1.6.2
user$ ./configure
```

不带任何参数执行 *./configure* 脚本将按照默认选项生成 Makefile。运行 *./configure --help* 可以看到各种可配置的选项，关于可配置选项的具体含义，请参阅本书附录 B“ MiniGUI-STR 配置选项详解 ”。简单说来，运行 *./configure --enable-**xxx*** 可以配置 MiniGUI 函数库打开某项功能，而运行 *./configure --disable-**xxx*** 则可以禁止某项功能。

如果运行 *./configure* 脚本的时候没有出现问题，就可以继续运行 *make* 和 *make install* 命令编译并安装 *libminigui*，注意要有 root 权限才能向系统中安装函数库：

```
user$ make
user$ su -c ‘make install’
```

如果在运行 *./configure* 命令时出现函数库检查错误，通常是 MiniGUI 的依赖库没有正确安装，而用户却配置 MiniGUI 使用此依赖库，这时，可安装对应的依赖库，或者使用 *--disable* 参数禁止该选项。比如，如果系统中缺少 libjpeg 库，则可以使用如下命令取消对 JPEG 图片的支持：

```
user$ ./configure --disable-jpgsupport
```

在一切正常之后，确保已经将 */usr/local/lib* 目录添加到 */etc/ld.so.conf* 文件中 [28]，然后运行 *ldconfig* 命令刷新系统的共享库搜索缓存：

```
user$ su -c /sbin/ldconfig
```

[28] 在 Linux PC 上，共享库系统可在默认路径之外搜索共享库。通常，我们在 */etc/ld.so.conf* 文件中维护额外的共享库搜索路径。在新增共享库搜索路径，或者安装了新的共享库，或者安装了共享库的新版本时，都要运行 *ldconfig* 命令，以便更新 */etc/ld.so.cache* 文件（该文件是加速共享库搜索而使用的缓存文件）。

因为默认的配置脚本将把 MiniGUI 的配置文件、函数库和头文件安装到以 */usr/local/* 为前缀的目录树中，具体说来：

- 运行时配置文件将被安装到 */usr/local/etc/* 目录下；
- 库文件将被安装到 */usr/local/lib/* 目录下。
- 头文件将被安装到 */usr/local/include/minigui/* 目录下；

下面的清单给出的是安装 MiniGUI 后系统中新增的头文件和库文件：

```
user$ ls -l -R /usr/local/include/minigui/
/usr/local/include/minigui/:
-rw-r--r--    1 root     root        38066  2月 18 13:37 common.h
-rw-r--r--    1 root     root        19085  2月 18 13:37 config.h
-rw-r--r--    1 root     root         3410  2月 18 13:37 control.h
drwxr-xr-x    2 root     root         4096  2月 18 13:37 ctrl
-rw-r--r--    1 root     root         2250  2月 18 13:37 dti.c
-rw-r--r--    1 root     root        23773  2月 18 13:37 endianrw.h
-rw-r--r--    1 root     root        17703  2月 18 13:37 fixedmath.h
-rw-r--r--    1 root     root       220372  2月 18 13:37 gdi.h
-rw-r--r--    1 root     root         3536  2月 18 13:37 mgext.h
-rw-r--r--    1 root     root        70073  2月 18 13:37 minigui.h
-rw-r--r--    1 root     root        24753  2月 18 13:37 own_malloc.h
-rw-r--r--    1 root     root         3278  2月 18 13:37 own_stdio.h
-rw-r--r--    1 root     root       202242  2月 18 13:37 window.h

/usr/local/include/minigui/ctrl:
-rw-r--r--    1 root     root        14761  2月 18 13:37 button.h
-rw-r--r--    1 root     root        24041  2月 18 13:37 combobox.h
-rw-r--r--    1 root     root        13360  2月 18 13:37 ctrlhelper.h
-rw-r--r--    1 root     root        21606  2月 18 13:37 edit.h
-rw-r--r--    1 root     root        27000  2月 18 13:37 listbox.h
-rw-r--r--    1 root     root         9615  2月 18 13:37 menubutton.h
-rw-r--r--    1 root     root        10294  2月 18 13:37 newtoolbar.h
-rw-r--r--    1 root     root         5653  2月 18 13:37 progressbar.h
-rw-r--r--    1 root     root         9601  2月 18 13:37 propsheet.h
-rw-r--r--    1 root     root         2875  2月 18 13:37 scrollbar.h
-rw-r--r--    1 root     root        24187  2月 18 13:37 scrollview.h
-rw-r--r--    1 root     root         7190  2月 18 13:37 static.h
-rw-r--r--    1 root     root         2966  2月 18 13:37 textedit.h
-rw-r--r--    1 root     root         4107  2月 18 13:37 toolbar.h
-rw-r--r--    1 root     root        10576  2月 18 13:37 trackbar.h

$ ls -l /usr/local/lib
lrwxrwxrwx    1 root     root           23  2月 18 14:07 libminigui-1.6.so.2 -> \
libminigui-1.6.so.2.0.0
-rwxr-xr-x    1 root     root      2417885  2月 18 14:07 libminigui-1.6.so.2.0.0
-rw-r--r--    1 root     root      8855152  2月 18 14:07 libminigui.a
-rwxr-xr-x    1 root     root          738  2月 18 14:07 libminigui.la
lrwxrwxrwx    1 root     root           23  2月 18 14:07 libminigui.so -> \
libminigui-1.6.so.2.0.0
```

在进行交叉编译时，为了将 MiniGUI 的头文件和库文件安装到交叉编译环境中，有时我们需要使用 *--prefix* 选项指定不同的安装路径之前缀。详情请参阅本章后面的章节。

MiniGUI-STR 资源包的安装比较简单，只需解开软件包并以 root 身份

运行 *make install* 命令，如下所示：

```
$ tar zxf <path to cdrom>/minigui/minigui-res-str-1.6.tar.gz
$ cd minigui-res-str-1.6
$ su -c make install
```

默认的安装脚本会把 MiniGUI 资源文件安装到 */usr/local/lib/minigui/res/* 目录下。

在安装完 MiniGUI 函数库及资源包之后，我们就可以尝试编译并运行 MiniGUI 的示例程序包了。

5) 运行 MiniGUI-STR 的示例程序

首先，我们编译 mg-samples-str 包中的程序。按如下命令操作：

```
user$ tar zxf <path to cdrom>/minigui/mg-samples-str-1.6.2.tar.gz
user$ cd mg-samples-str-1.6.2
user$ ./configure
user$ make
```

如果不出任何问题，说明 MiniGUI-STR 安装正确，我们可以在 *src/* 目录下看到若干编译好的示例程序。进入 *src/* 目录，即可在当前目录下运行这些示例程序——但是，在运行这些示例程序之前，我们要先确认是否已经准备好了运行 MiniGUI 的环境。

根据 4.2.3 小节的描述，我们可以让 MiniGUI 程序在 qvfb 中运行，也可以在 Linux 控制台的 FrameBuffer 设备上运行。如果您打算在 qvfb 中运行 MiniGUI 应用程序，则应该按如下步骤操作：

1) 在 X Window 环境中，启动 qvfb 程序。
2) 点击 qvfb 的菜单：`file->configure`，设置成 320×240 16 位色的显示模式。
3) 修改 MiniGUI 的运行时配置文件：*/usr/local/etc/MiniGUI.cfg*，指定要使用的图形引擎和输入引擎名称及其相关参数，如下所示：

```
[system]
gal_engine=qvfb
ial_engine=qvfb

[qvfb]
defaultmode=320x240-16bpp
display=0
```

之后，我们可以运行已经编译好的 MiniGUI 示例程序。您会发现，MiniGUI 的窗口将显示在 qvfb 程序创建的窗口中，如图 4-5 所示。

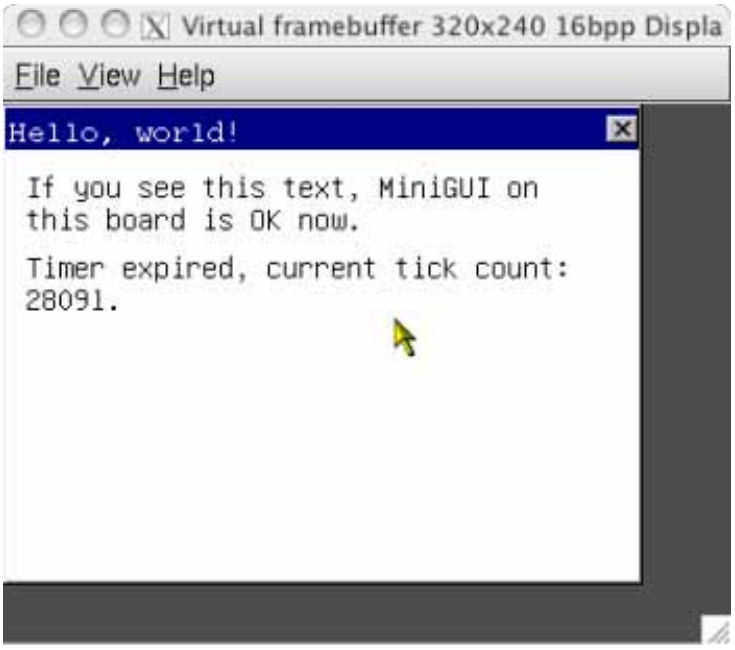

图 4-5 在 qvfb 中运行 MiniGUI 应用程序

如果您打算在 Linux 控制台的 FrameBuffer 设备之上运行 MiniGUI 应用程序，则应该按如下步骤操作：

1) 切换到 Linux 字符控制台下（如果当前在 X Window 环境中，则可以按 <Ctrl+Alt+F1>组合键）。
2) 修改 MiniGUI 的运行时配置文件：*/usr/local/etc/MiniGUI.cfg*，指定要使用的图形引擎和输入引擎名称及其相关参数，如下所示：

```
[system]
gal_engine=fbcon
ial_engine=console
mdev=/dev/mouse
mtype=PS2

[fbcon]
defaultmode=1024x768-16bpp
```

这里假定您使用的鼠标采用 PS2 协议，连接在 */dev/mouse* 设备上，FrameBuffer 的分辨率为 1024x768，颜色深度为 16 位色。如果您的系统没有安装鼠标，或者鼠标不正常，您可以通过指定 *mtype=none* 来禁止鼠标。

在 Linux 字符控制台上运行 MiniGUI 应用程序时，您可以使用如下快

捷键：

- <*Ctrl+Alt+BackSpace*>：强制退出 MiniGUI 应用程序。
- <*Ctrl+Esc*>：激活系统菜单，您可以选择退出 MiniGUI 会话。

图 4-6 给出了在 Linux 控制台上运行 MiniGUI 应用程序（该程序源代码在 *mde-str-1.6.2.tar.gz* 中）的效果图。

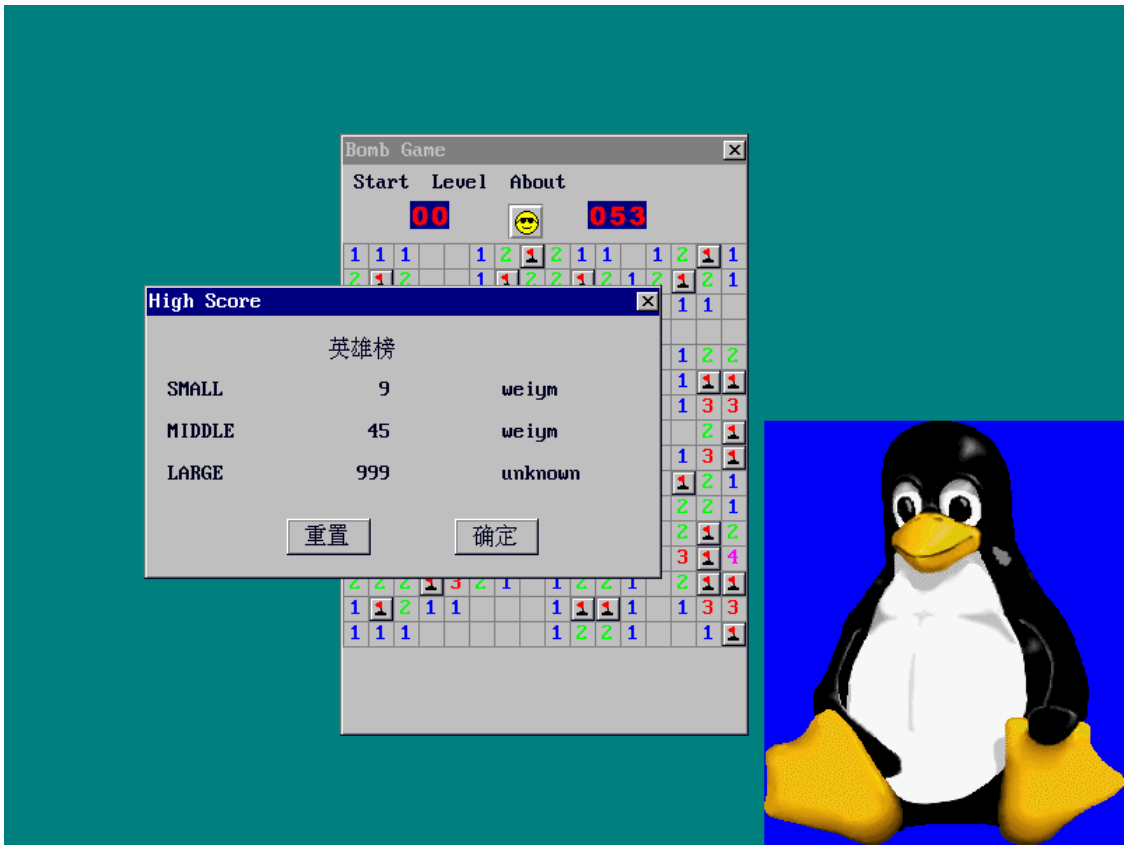

图 4-6 在 Linux FrameBuffer 设备上运行 MiniGUI 应用程序

4.3 在 SkyEye 的 EP7312 模拟器上运行 MiniGUI

SkyEye 项目组于 2004 年针对 EP7312 模拟器完成了 LCD 和触摸屏的仿真 [29]，本节内容基于 SkyEye 项目组的这些工作举例描述如何在嵌入式 Linux 上运行 MiniGUI。

在开始之前，希望读者通过第 2 章和第 3 章的学习，已经掌握了如何

[29] SkyEye EP7312 上的 LCD 和触摸屏仿真工作由 SkyEye 项目组成员尹文超完成，他的电子邮件地址为： ywc02@mails.tsinghua.edu.cn。

针对 SkyEye 的 EP7312 模拟器编译内核，且已准备好了基本的文件系统。我们假定本节所需的源代码、工具等的安装布局如下：

- armlinux 内核：*/opt/armlinux/linux-2.4.13*
- armlinux 交叉编译工具：*/usr/local/arm/2.95.3*

和本节内容相关的其他代码包，保存在产品光盘的 */armlinux/ep7312/* 目录下。

4.3.1 确认内核配置

首先，我们要将针对 SkyEye EP7312 模拟器的 LCD 驱动程序和触摸屏驱动程序添加到默认的 armlinux 内核中。将产品光盘中的 */armlinux/ep7312/armlinux4skyeye-ep7312.tar.gz* 文件解开到主机的 */opt/armlinux/ep7312* 目录下：

```
root# cd /opt/armlinux/
root# mkdir /opt/armlinux/ep7312
root# cd ep7312
root# cp <path to cdrom>/armlinux/ep7312/armlinux4skyeye-ep7312.tar.gz .
root# tar jxf armlinux4skyeye-ep7312.tar.gz
```

然后，将 LCD FrameBuffer 驱动程序和触摸屏驱动程序源文件复制到 armlinux 内核中：

```
root# cd /opt/armlinux/
root# cp ep7312/armlinux4skyeye-ep7312/skyeye_ts_drv.[ch] linux-2.4.13/drivers/char/
root# cp ep7312/armlinux4skyeye-ep7312/ep7312_sys.h linux-2.4.13/drivers/char/
root# cp ep7312/armlinux4skyeye-ep7312/ep7312.h linux-2.4.13/drivers/video/
root# cp ep7312/armlinux4skyeye-ep7312/clps711xfb.c linux-2.4.13/drivers/video/ -f
```

注意，上面最后一条命令将覆盖内核中原有的 *clps711xfb.c* 文件。也就是说，SkyEye EP7312 模拟器的 LCD FrameBuffer 是在 CLPS711x FrameBuffer 驱动的基础上修改而成的。实际的 EP7312 板子支持 12 位色的显示模式，但模拟器中的实现仅支持 8 位色。

接下来，我们修改 Linux 内核的配置文件，将触摸屏驱动程序添加到 Linux 内核的配置选项中。修改 *linux-2.4.13/drivers/char/Config.in* 文件，在 33 行处添加如下一行：

```
bool 'Touch scren driver support for SkyEye EP7312 simulation' CONFIG_TS_SKYEYE_EP7312
```

修改 *linux-2.4.13/drivers/char/Makefile* 文件，在 263 行处添加如下一

行：

```
obj-$(CONFIG_TS_SKYEYE_EP7312) += skyeye_ts_drv.o
```

上述修改将把针对 SkyEye EP7312 的触摸屏驱动程序作为配置选项添加到了 Linux 内核中，这样，我们就可以通过 *make menuconfig* 命令在配置 Linux 内核时选择是否编译该触摸屏驱动程序。

在完成上述修改之后，我们就可以在 *linux-2.4.13/* 目录中运行 *make menuconfig* 命令配置 Linux 内核了。保持其他配置不变，在“System Type”的“CLPS711X/EP721X Implementations”选中“EDB7312”；在“Character devices”选项中，我们按图 4-7 那样，确保选中“Virtual terminal”以及刚才我们添加的针对 SkyEye EP7312 模拟器的触摸屏驱动程序（“Touch scren driver support for SkyEye EP7312 simulation”）选项。

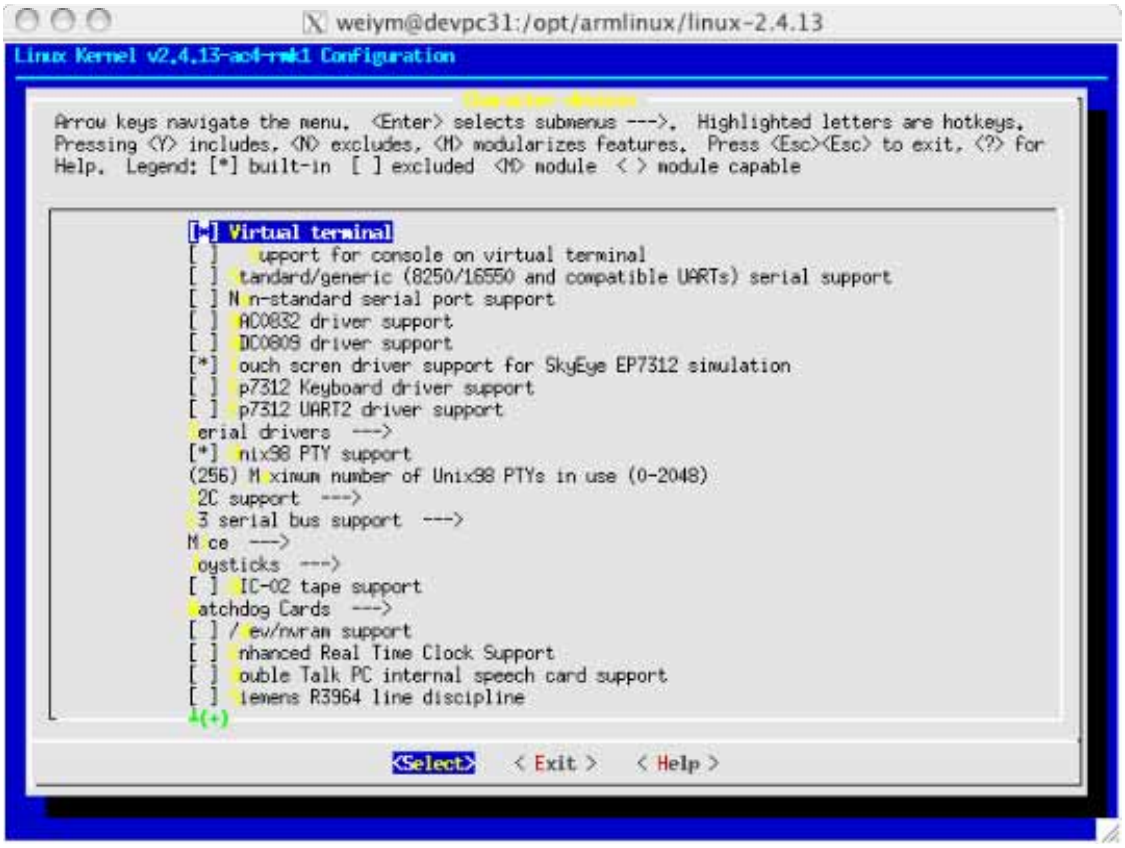

图 4-7 配置 Linux 内核激活触摸屏驱动程序

在“Console drivers”选项组中，取消“VGA tex console”选项，并在“Frame-buffer Support”中，按图 4-8 所示那样选择各

选项。

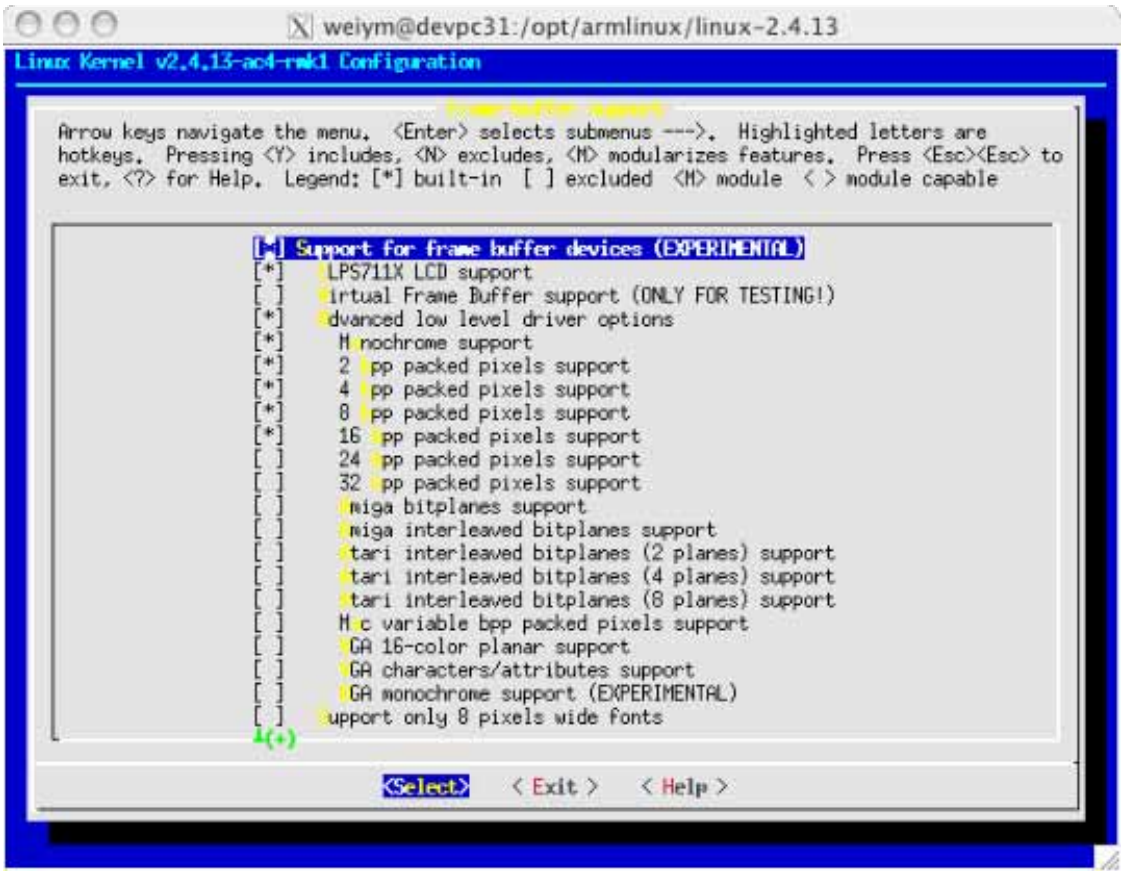

图 4-8 配置 Linux 内核激活 FrameBuffer 驱动程序

完成上述配置后，保留其他配置不变，即可保存并退出 *make menuconfig* 命令。检查 *linux-2.4.13/* 目录下的 *.config* 文件，可看到如下选项：

```
CONFIG_VT=y

CONFIG_TS_SKYEYE_EP7312_TS=y


CONFIG_FB=y
CONFIG_DUMMY_CONSOLE=y
CONFIG_FB_CLPS711X=y
CONFIG_FBCON_ADVANCED=y
CONFIG_FBCON_MFB=y
CONFIG_FBCON_CFB2=y
CONFIG_FBCON_CFB4=y
CONFIG_FBCON_CFB8=y
CONFIG_FBCON_CFB16=y
```

接下来我们就可以针对 SkyEye 的 EP7312 模拟器编译包含 LCD 仿真和触摸屏仿真的 Linux 内核了：

```
root# make dep; make bzImage
```

4.3.2 SkyEye EP7312 模拟器的 MiniGUI 输入引擎

MiniGUI-STR 中已经包含了针对 SkyEye EP7312 模拟器的输入引擎，完整的源代码可见 MiniGUI 源代码包中的 *src/ial/skyeye-ep7312.h* 和 *src/ial/skyeye-ep7312.c* 文件。下面我们简单介绍一下这个输入引擎。

该输入引擎的名称为“ SkyEyeEP7312 ”，在 MiniGUI 源代码包的 *src/ial/ial.c* 文件中定义了该输入引擎的入口项：

```
...

#ifdef _SKYEYE_EP7312_IAL
    #include "skyeye-ep7312.h"
#endif


...

static INPUT inputs [] =
{
...

#ifdef _SKYEYE_EP7312_IAL
    {"SkyEyeEP7312", InitSkyEyeEP7312Input, TermSkyEyeEP7312Input},
#endif

...

}
```

上述代码中的 inputs 结构数组中保存了所有 MiniGUI 的输入引擎入口项。在 MiniGUI 的初始化过程中，MiniGUI 将根据 MiniGUI.cfg 配置文件中设定的输入引擎名称在该数组中寻找匹配项，然后调用对应的初始化函数，并在终止时调用终止函数。对 SkyEye 的 EP7312 模拟器来讲，输入引擎的名称为“ SkyEyeEP7312 ”，初始化和终止函数分别为：*InitSkyEyeEP7312Input* 和 *TermSkyEyeEP7312Input*。这两个函数在 *skyeye-ep7312.h* 头文件中声明，在 *skyeye-ep7312.c* 文件中实现。因为 SkyEye EP7312 模拟器的输入引擎是参照 Xcopilot 的输入引擎编写的，因此，我们将在本章后面介绍 Xcpilot 的输入引擎时详细介绍相关源代码。

该输入引擎的编译配置选项为 *--enable-skyeyeep7312ial*，默认是关闭的。如果要在 MiniGUI 库中包含该输入引擎，则需要在配置 MiniGUI 时使用如下的配置选项：

```
--enable-skyeyeep7312ial
```

该配置选项将打开*_SKYEYE_EP7312_IAL* 宏的定义，从而确保在 MiniGUI 中增加该输入引擎的相关代码。

4.3.3 为 SkyEye 的 EP7312 模拟器交叉编译 MiniGUI-STR

我们可以直接使用 MiniGUI-STR 源代码包中包含的配置脚本来为 SkyEye EP7312 模拟器配置并交叉编译 MiniGUI-STR。我们首先看这个配置脚本的内容。进入 MiniGUI-STR 源代码目录：

```
user$ cd libminigui-str-1.6.2
```

打开 build/ 目录下的 *buildlib-linux-ep7312-skyeye* 文件。该文件的内容如下：

```
#!/bin/bash

rm config.cache config.status -f

CC=arm-linux-gcc \
CFLAGS=-I/opt/armlinux/linux-2.4.13/include \
./configure --prefix=/usr/local/arm/2.95.3/arm-linux/ \
    --build=i386-linux \
    --host=arm-unknown-linux \
    --target=arm-unknown-linux \
    --disable-static \
    --disable-cursor \
    --disable-micemoveable \
    --disable-galqvfb \
    --enable-nativegal \
    --enable-skyeyeep7312ial \
    --disable-qvfbial \
    --disable-nativeial \
    --disable-latin9support \
    --disable-big5support \
    --disable-imegb2312 \
    --disable-savebitmap \
    --disable-savescreen \
    --disable-aboutdlg \
    --disable-dblclk
```

接下来我们详细解释一下该配置脚本中的设置。

```
CC=arm-linux-gcc
```

该设置告诉 configure 脚本在生成 makefile 文件时，使用 *arm-linux-gcc* 来编译 C 程序。

```
CFLAGS=-I/opt/armlinux/linux-2.4.13/include
```

该设置告诉 configure 脚本在生成 makefile 文件时，为 C 编译器传递额外的编译选项。在这里，我们通过*-I/opt/armlinux/linux-2.4.13/include* 选项指定了 Linux 内核头文件的路径，主要是因为 MiniGUI 库中使用了 Linux 内核的头文件。

```
./configure --prefix=/usr/local/arm/2.95.3/arm-linux/ \
```

--prefix 选项设定了安装 MiniGUI 配置文件、函数库及头文件的目录前缀，在执行 *make install* 命令时，将把 MiniGUI 配置文件、库文件和头文件分别安装到如下位置：

- */usr/local/arm/2.95.3/arm-linux/etc/*
- */usr/local/arm/2.95.3/arm-linux/lib/*
- */usr/local/arm/2.95.3/arm-linux/include/*

```
    --build=i386-linux \
    --host=arm-unknown-linux \
    --target=arm-unknown-linux \
```

上述 configure 的命令行选项分别指定了交叉编译环境：在 i386 系统上运行交叉编译工具链（build），宿主机及目标机为 arm-linux（host、target）。

```
    --disable-cursor \
    --disable-micemoveable \
    --disable-galqvfb \
    --disable-nativeial \
    --disable-qvfbial \
    --enable-skyeyeep7312ial \
    --disable-latin9support \
    --disable-big5support \
    --disable-imegb2312 \
    --disable-savebitmap \
    --disable-savescreen \
    --disable-aboutdlg \
    --disable-dblclk
```

上面这些选项是 MiniGUI 本身的配置选项，和 SkyEye EP7312 模拟器特别相关的选项是*--enable-skyeyeep7312ial*。该选项打开了针对 SkyEye EP7312 模拟器的输入引擎。其他的选项可参阅本书附录 B“MiniGUI-STR 配置选项详解”。注意，本配置脚本将 MiniGUI 配置成了 MiniGUI-Threads 模式。

在 MiniGUI-STR 源代码目录下，可直接运行该脚本配置 MiniGUI，然后运行 *make* 和 *make install* 命令：

```
user$ cd libminigui-str-1.6.2
user$ ./build/buildlib-linux-ep7312-skyeye
user$ make clean; make
user$ su –c ‘make install’
```

在成功运行上述命令之后，我们可在通过 *--prefix* 选项指定的目录下看到 MiniGUI 的头文件及交叉编译后的函数库文件。

```
user$ cd /usr/local/arm/2.95.3/arm-linux
user$ ls etc/ include/minigui/ lib/libminigui* -l -R
-rwxr-xr-x    1 root     dip        923818 Mar 14 16:30 lib/libminigui.a
lrwxrwxrwx    1 root     dip            23 Mar 14 16:30 \
lib/libminigui-1.6.so.2 -> libminigui-1.6.so.2.0.0
-rwxr-xr-x    1 root     dip       1028020 Mar 14 16:30 lib/libminigui-1.6.so.2.0.0
-rwxr-xr-x    1 root     dip           748 Mar 14 16:30 lib/libminigui.la
lrwxrwxrwx    1 root     dip            23 Mar 14 16:30 \
lib/libminigui.so -> libminigui-1.6.so.2.0.0

etc/:
-rw-r--r--    1 root     dip          5335 Mar 14 16:30 MiniGUI.cfg

include/minigui/:
-rw-r--r--    1 root     dip         37838 Mar 14 16:30 common.h
-rw-r--r--    1 root     dip         12482 Mar 14 16:30 config.h
-rw-r--r--    1 root     dip          3410 Mar 14 16:30 control.h
drwxr-sr-x    2 root     dip          4096 Mar 14 16:30 ctrl
-rw-r--r--    1 root     dip          2250 Mar 14 16:30 dti.c
-rw-r--r--    1 root     dip         23704 Mar 14 16:30 endianrw.h
-rw-r--r--    1 root     dip         17634 Mar 14 16:30 fixedmath.h
-rw-r--r--    1 root     dip        220623 Mar 14 16:30 gdi.h
-rw-r--r--    1 root     dip         69897 Mar 14 16:30 minigui.h
-rw-r--r--    1 root     dip        202605 Mar 14 16:30 window.h

include/minigui/ctrl:
-rw-r--r--    1 root     dip         14761 Mar 14 16:30 button.h
-rw-r--r--    1 root     dip         24041 Mar 14 16:30 combobox.h
-rw-r--r--    1 root     dip         13360 Mar 14 16:30 ctrlhelper.h
-rw-r--r--    1 root     dip         21828 Mar 14 16:30 edit.h
-rw-r--r--    1 root     dip         27000 Mar 14 16:30 listbox.h
-rw-r--r--    1 root     dip          9615 Mar 14 16:30 menubutton.h
-rw-r--r--    1 root     dip         10294 Mar 14 16:30 newtoolbar.h
-rw-r--r--    1 root     dip          5653 Mar 14 16:30 progressbar.h
-rw-r--r--    1 root     dip          9601 Mar 14 16:30 propsheet.h
-rw-r--r--    1 root     dip          2875 Mar 14 16:30 scrollbar.h
-rw-r--r--    1 root     dip         24187 Mar 14 16:30 scrollview.h
-rw-r--r--    1 root     dip          7190 Mar 14 16:30 static.h
-rw-r--r--    1 root     dip          2966 Mar 14 16:30 textedit.h
-rw-r--r--    1 root     dip          4107 Mar 14 16:30 toolbar.h
-rw-r--r--    1 root     dip         10576 Mar 14 16:30 trackbar.h
```

至此，我们编译好了针对 SkyEye EP7312 模拟器的 MiniGUI-STR，并将配置文件、头文件和函数库安装到了指定的位置。将 MiniGUI 头文件和函数库安装到上述这个位置主要的好处是，在进行交叉编译时，我们无须显式指定 MiniGUI 相关头文件和库文件的搜索路径。接下来，我们编译并运行 MiniGUI 应用程序。

4.3.4 在 SkyEye 的 EP7312 模拟器上运行 MiniGUI 示例程序

1) 交叉编译 MiniGUI 示例程序

单独编译 MiniGUI 应用程序的办法可参照 3.1 节中所讲过程进行，当然，我们不能忘记在链接生成可执行程序时，通过 *–l* 选项指定要链接的 MiniGUI 函数库，比如：

```
user$ arm-linux-gcc -Wall -O2 -o helloworld helloworld.c -lminigui –lpthread
```

如果您使用 *mg-samples-str* 包，我们还可以直接使用预先准备的 *configure* 脚本生成可用于交叉编译的 makefile 文件：

```
user$ cd <path to mg-samples-str-1.6.2>
user$ ./build-linux-ep7312-skyeye
user$ make clean; make
```

运行上述命令之后，将交叉编译生成 mg-samples-str 包中针对 SkyEye EP7312 模拟器的所有示例程序。

2) 准备文件系统

一般而言，在嵌入式系统开发过程中，我们编译完 MiniGUI 和应用程序之后，需要把 MiniGUI 库、资源和应用程序拷贝到为目标系统准备的文件系统目录中，然后使用相关的工具生成目标映像，再下载到目标板上运行。在某些目标系统上，您也可以使用某些下载工具通过串口或以太网口单独下载文件到目标系统中。

针对 SkyEye 的 EP7312 模拟器，我们使用 ROMFS 技术，将根文件系统放在 ROMFS 中，由内核在引导结束后挂装该文件系统。为此，我们需在 *skyeye.conf* 文件中指定好 ROMFS 的映像文件名称及合适的大小。

我们在一个已有的 *initrd.img* 基础上制作包括 MiniGUI 函数库、配置文件、资源文件及示例程序在内的 ROMFS 映像文件。这个原始的 *initrd.img* 文件保存在产品光盘的 */armlinux/ep7312/* 目录下。制作新 ROMFS 映像文件的步骤如下：

第1步. 创建一个 *romfs/* 目录：

```
root# mkdir /opt/armlinux/ep7312/romfs -p
```

```
root# cd /opt/armlinux/ep7312
```

第2步. 将已有的 *initrd.img* 文件挂装到另一个目录下：

```
root# cp <path to cdrom>/armlinux/ep7312/initrd.img .
root# mount -t ext2 -o loop initrd.img /mnt/tmp/
```

第3步. 将老的文件系统内容全部复制到新的 romfs 目录：

```
root# cp /mnt/tmp/* romfs/ -a
root# umount /mnt/tmp
```

第4步. 将 minigui-res-str 包中的资源直接安装到目标文件系统中。进入 *minigui-res-str-1.6.2* 目录并编辑 *config.linux* 文件，将其中的 `prefix` 变量修改为：

```
prefix=/opt/armlinux/ep7312/romfs
```

然后执行 *make install* 即可将 MiniGUI-STR 所使用的资源文件复制到 */opt/armlinux/ep7312/romfs/usr/local/lib/minigui* 目录下。删除不需要的位图、图标等资源：

```
root# cd romfs/usr/local/lib/minigui/res/
root# rm bmp/*flat.bmp -f
root# rm bmp/*phone.bmp -f
root# rm icon/*flat.ico -f
root# rm imetab/ -rf
```

第5步. 因为 MiniGUI 应用程序使用动态链接，因此，我们需要将交叉编译环境中的 C 函数库、libpthread 函数库、MiniGUI 函数库复制到 ROMFS 文件系统的 */lib* 目录下：

```
root# cd /opt/armlinux/ep7312/romfs
root# cp /usr/local/arm/2.95.3/arm-linux/lib/libminigui-1.6.so.2 lib/
root# cp /usr/local/arm/2.95.3/arm-linux/lib/libpthread.so.0 lib/
root# cp /usr/local/arm/2.95.3/arm-linux/lib/libm.so.6 lib/
root# cp /usr/local/arm/2.95.3/arm-linux/lib/libc.so.6 lib/
root# cp /usr/local/arm/2.95.3/arm-linux/lib/ld-linux.so.2 lib/
root# arm-linux-strip lib/*
```

上述命令将 *ld-linux.so.2*、*libc.so.6*、*libm.so.6*、*libpthread.so.0* 等文件复制到 ROMFS 文件系统的 */lib* 目录。注意，我们最后调用 *arm-linux-strip* 命令剥离了共享库中的符号信息。

【提示】您可以利用 arm-linux-strings <excutable> | grep ^lib 命令来确定一个可执行程序使用的共享库名称。

【思考题】我们为什么没有复制 libminigui.so 及 libminigui-1.6.so.2.0.0 文件？

第6步. 在 ROMFS 中建立下面的符号链接：

```
root# mkdir -p usr/local/arm/2.95.3/arm-linux/
root# ln -s /lib usr/local/arm/2.95.3/arm-linux/lib
```

上面的命令确保动态链接系统能够找到正确的动态链接库。ROMFS 中的 */usr/local/arm/2.95.3/arm-linux/lib* 目录其实就是在主机上编译生成可执行文件时共享库的位置，我们要在目标系统上建立一样的目录，以便在执行程序时，动态链接系统能够找到这些共享库。因为我们将程序所使用的共享库全部复制到了 ROMFS 的 */lib* 目录下，因此，这里我们使用符号链接指向这个位置。

第7步. 复制 MiniGUI 的运行时配置文件：

```
root# mkdir usr/local/etc
root# cp /usr/local/arm/2.95.3/arm-linux/etc/MiniGUI.cfg usr/local/etc/
```

修改配置文件，指定正确的输入引擎名称：

```
[system]
# GAL engine
gal_engine=fbcon

# IAL engine
ial_engine=SkyEyeEP7312

mdev=/dev/ts
mtype=def
```

第8步. 将 MiniGUI 示例程序剥离符号信息并复制到 ROMFS 文件系统的 */bin* 目录下：

```
root# arm-linux-strip <path to mg-samples-str-1.6.2>/src/
root# cp <path to mg-samples-str-1.6.2>/src/helloworld bin/
```

第9步. 调用 *genromfs* 命令制作 ROMFS 映像文件：

```
root# cd /opt/armlinux/ep7312
root# genromfs -d romfs -f romfs.img
root# ls -l romfs.img
-rw-r--r--    1 root     root      4218880 Mar 15 17:03 romfs.img
```

至此，包含 MiniGUI 示例程序的 ROMFS 文件系统映像就建立好了。

3) 运行 MiniGUI 示例程序

在 /opt/armlinux/ep7312 目录中，创建如下的 *skyeye.conf* 文件：

```
#skyeye config for MiniGUI sample
cpu: arm720t
mach: ep7312
lcd:state=on
mem_bank: map=I, type=RW, addr=0x80000000, size=0x00010000
mem_bank: map=M, type=R,  addr=0x00000000, size=0x000C0000
mem_bank: map=M, type=R,  addr=0x000C0000, size=0x00800000, file=./romfs.img
mem_bank: map=M, type=RW, addr=0xc0000000, size=0x01000000
```

注意，上面的 *skyeye.conf* 文件在定义内存块时，根据 *romfs.img* 的大小（4218880 字节）将 `size` 设置成了 `0x00800000`。

将我们编译好的 Linux 内核复制到 */opt/armlinux/ep7312* 目录，然后在 X Window 终端仿真程序中运行 SkyEye：

```
root# cp /opt/armlinux/linux-2.4.13/vmlinux .
root# skyeye vmlinux
(SkyEye) target sim
(SkyEye) load
(SkyEye) run
```

内核启动之后，进入 */bin* 目录运行 *helloworld* 程序：

```
Welcome to
 _      _____      __   __    _       _
    / \    /  __ \    /  \_/  \  | |     |_|
   / _ \   | |  | |  / /\    /\ \ | |      _ ____  _   _  _ _
  / /_\ \  | |__| | / /  \_/  \ \| |     | |  _ \| | | |\ \/ /
 / /___\ \ | |__\ \ | |        | || |___  | | |_| | |_| |/    \
/_/     \_\| |   \_\|_|       |_||_____||_|_|  |_|\____|\_/\_/

ARMLinux for Skyeye
For further information check:
http://hpclab.cs.tsinghua.edu.cn/~skyeye/

Execution Finished, Exiting
Command: /bin/sh

Sash command shell (version 1.1.1)
/> cd bin
/bin> ./helloworld
```

则 MiniGUI 的 helloworld 程序将运行在 SkyEye 建立的用来模拟

LCD 的窗口中，如图 4-9 所示。

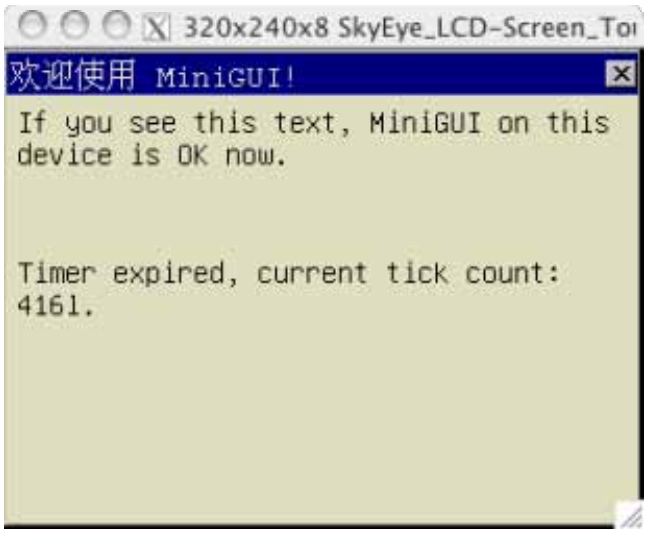

图 4-9 在 SkyEye 的 EP7312 模拟器上运行 MiniGUI helloworld 程序

我们还可以运行 mde-str 包中的 MiniGUI 示例程序：

```
user$ cd <path to mde-str-1.6.2>
user$ ./build-linux-ep7312-skyeye
user$ make clean; make
```

然后，我们将 same 程序及其资源放到 ROMFS 中：

```
root# cd /opt/armlinux/ep7312/romfs/bin
root# cp <path to mde-str-1.6.2>/same/same .
root# mkdir res; cd res
root# cp <path to mde-str-1.6.2>/same/res/*.gif .
```

重新生成 *romfs.img* 并执行 SkyEye，我们可在 SkyEye 的模拟 LCD 窗口中看到 MiniGUI same 游戏，见图 4-10 所示 [30]。

[30] 在 SkyEye 的 LCD 模拟窗口中运行 MiniGUI 示例程序时，您会发现屏幕输出比较慢，有时还会出现刷新不及时的情况。这主要是因为 SkyEye 模拟器的 LCD 模拟部分还需要进一步优化。读者可以关注 SkyEye 项目的开发进展。

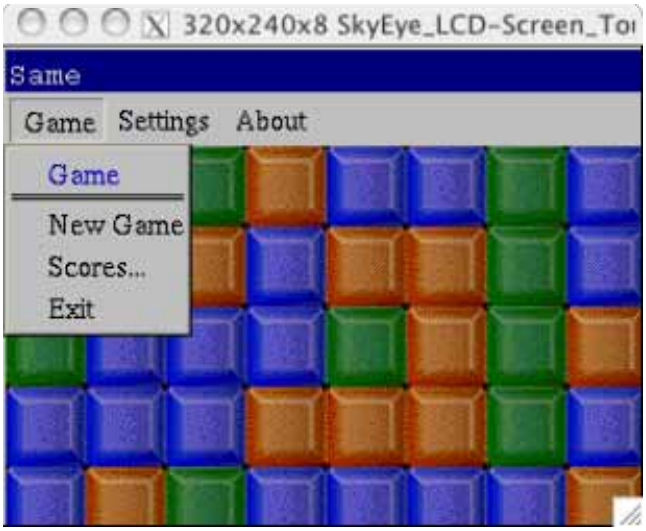

图 4-10 在 SkyEye 的 EP7312 模拟器上运行 MiniGUI same 程序

4.4 在 Xcopilot 模拟器上运行 MiniGUI

4.4.1 确认内核及 uClibc 配置

假定您已经将产品光盘所带的 uClinux-dist 安装到了系统的 */opt/uclinux* 目录下。

由于该 uClinux-dist 发行版的 Xcopilot 仿真部分有一些缺陷，所以我们需要打一个补丁，该补丁保存在产品光盘的 */uclinux/* 目录下，运行下面的命令以将该补丁作用于原始的 uClinux-dist：

```
root# cd /opt/uclinux/uClinux-dist
root# patch -p1 < <path to the patch>/uClinux-dist-20030522.patch
```

为了在 Xcopilot 模拟器上运行 MiniGUI，我们要打开内核中的 FrameBuffer 驱动程序和触摸屏驱动程序，在 */opt/uclinux/uClinux-dist* 中运行下面的命令：

```
root# make menuconfig
```

运行该命令将进入 uClinux-dist 的图形配置界面，如图 4-11 所示。

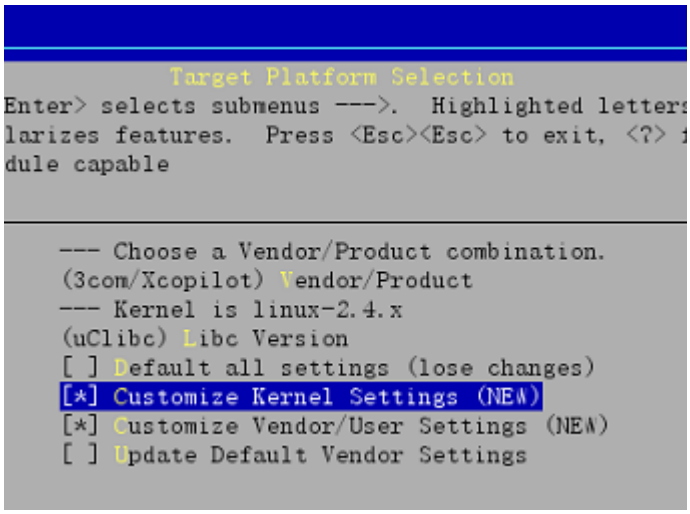

图 4-11 uClinux 配置界面

在“Vendor/Product”选项中确保选择目标平台为“3com/Xcopilot”。“Kernel”选为“linux-2.4.x”，“Libc Version”选“uClibc”。然后选中“Customize Kernel Settings”选项，进行内核的配置。

进入内核配置的“Character devices”菜单，如图 4-12 所示。选中“68328 digitizer support”和“Virtual terminal”。“68328 digitizer support”是 Xcopilot 的触摸屏驱动程序，“Virtual terminal”是虚拟终端支持，选中该项才能使“Frame-buffer support”选项出现。

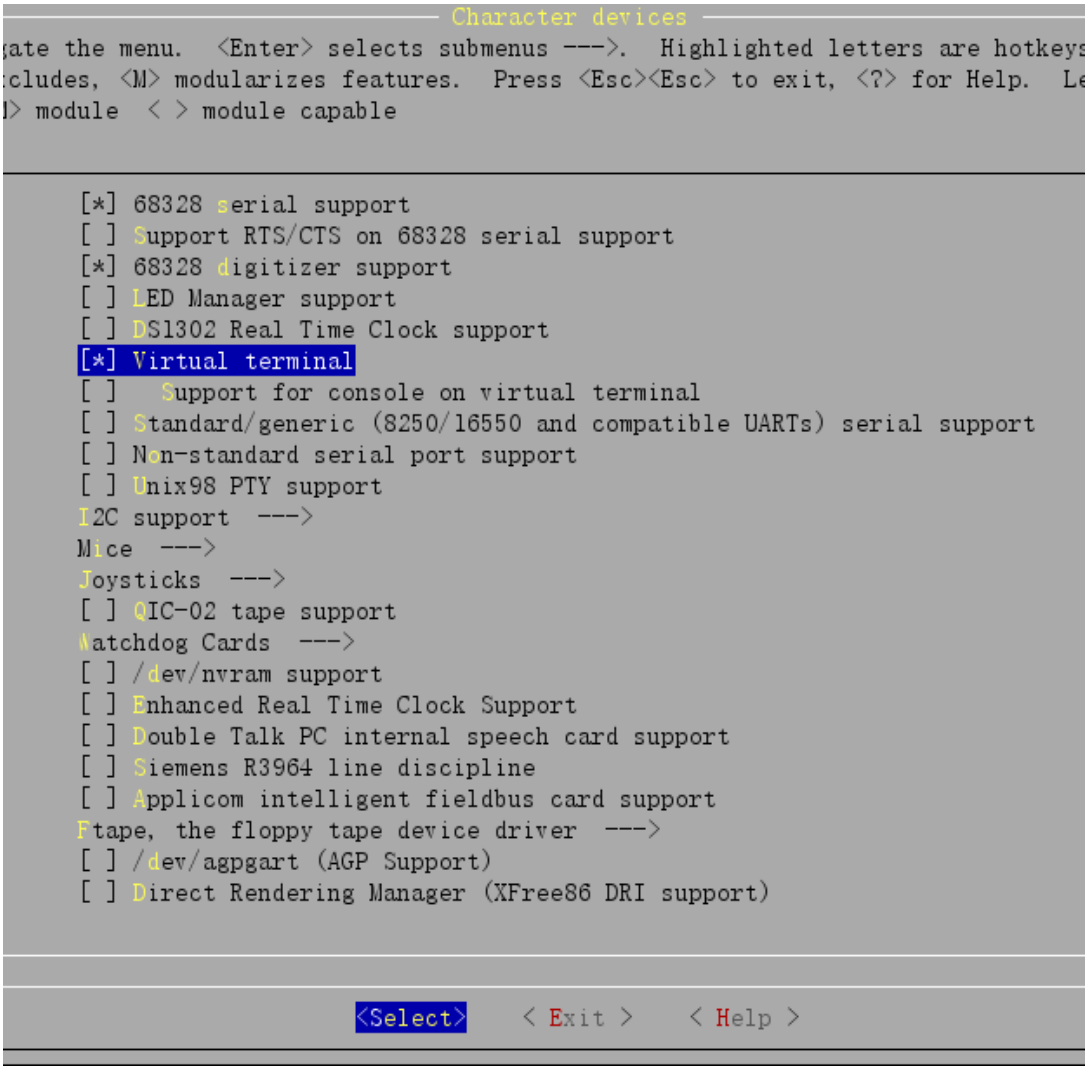

图 4-12 字符设备配置选项

进入“Frame-buffer support”子菜单，选中“Support for frame buffer devices”，然后选择“Dragonball frame buffer”以激活 Xcopilot 的 FrameBuffer 驱动程序；选中“Advanced low level driver options”，然后选择“Monochrome support”（因为 Xcopilot 是单色显示）。如图 4-13 所示。

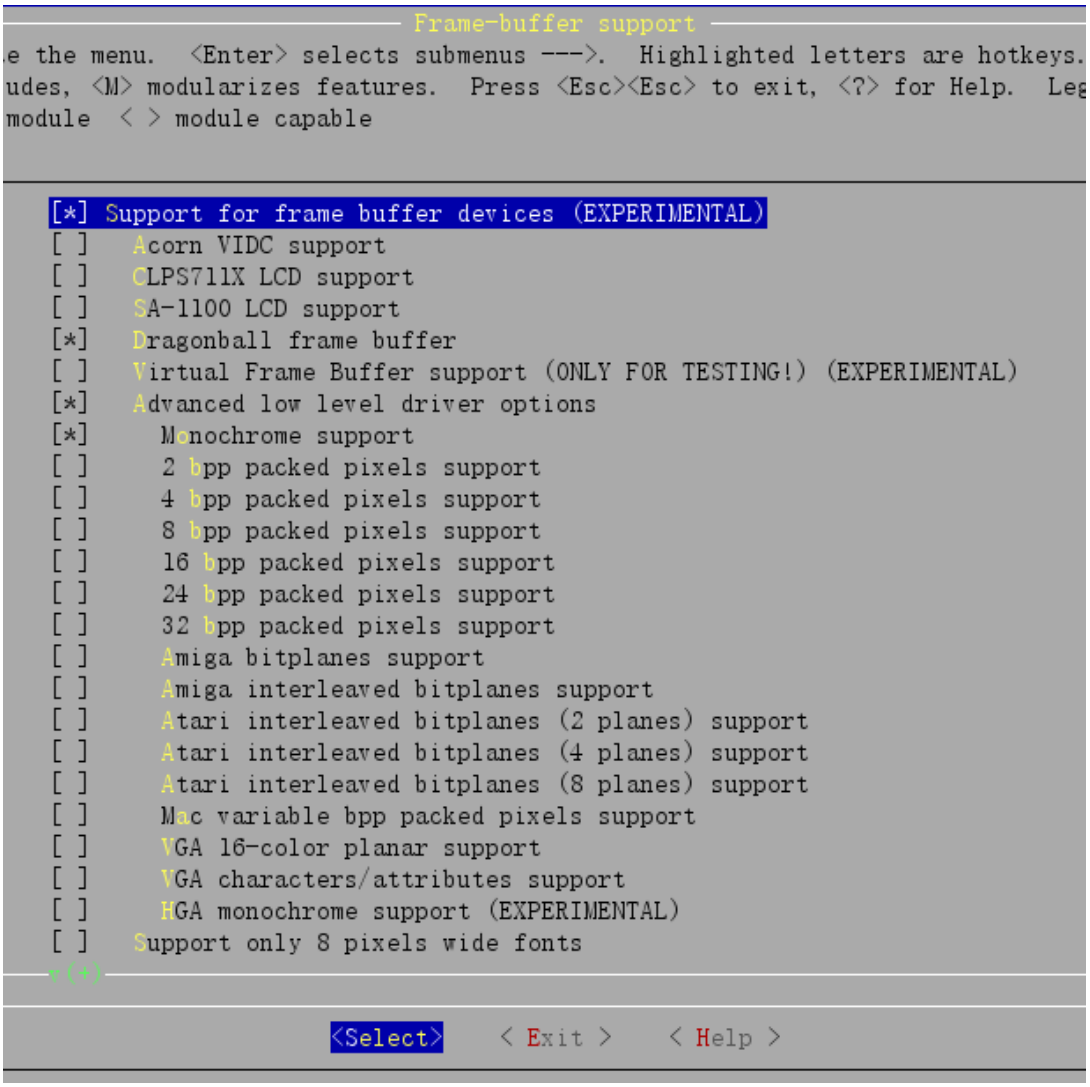

图 4-13 FrameBuffer 配置选项

以上是和 MiniGUI 相关的内核配置选项。其他选项可保持不动，然后保存 uClinux 配置并退出菜单配置程序。

接下来我们进入 *uClinux-dist/uClibc* 目录，运行下面的命令以确认 uClibc 中和 MiniGUI 相关的选项：

```
root# make TARGET_ARCH=m68k menuconfig
```

运行该命令将进入 uClibc 的图形配置界面，如图 4-14 所示。

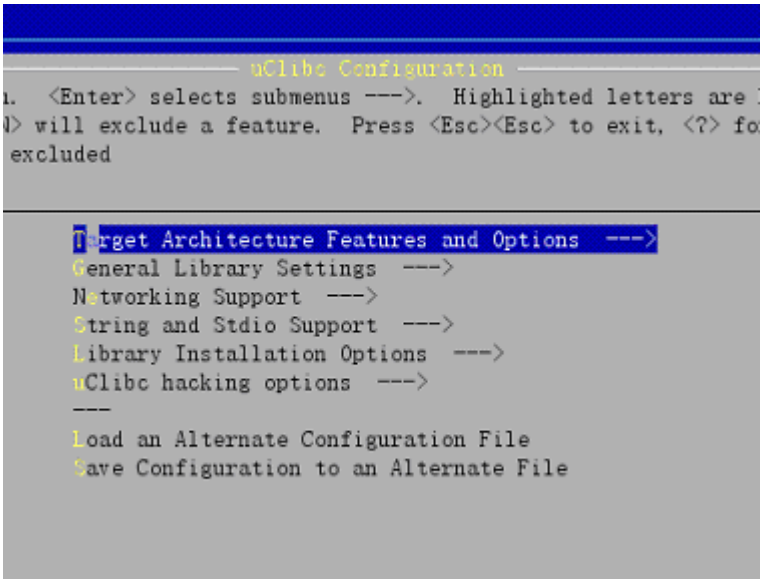

图 4-14 uClibc 配置界面

【提示】使用 make TARGET_ARCH=m68k menuconfig 可针对某个特定的平台配置 uClibc，如果是别的平台，把 m68k 改为相应的平台名称，比如针对 ARM，可使用 make TARGET_ARCH=arm menuconfig 命令。

进入“Target Architecture Features and Options”子菜单,选中“Enable floating point number support”和“Enable full C99 math library support”。如图 4-15 所示。这两个选项确保 uClibc 中包含完整的数学函数接口。

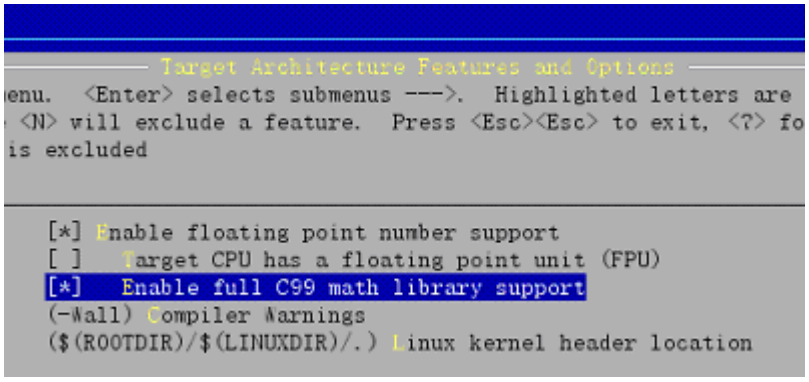

图 4-15 uClibc 的浮点和数学库支持

如果要在 Xcopilot 上运行 MiniGUI-Threads，则需要打开 uClibc 中的线程库支持。进入“General Library Settings”子菜单,选择“POSIX Threading Support”线程支持。如图 4-16 所示。

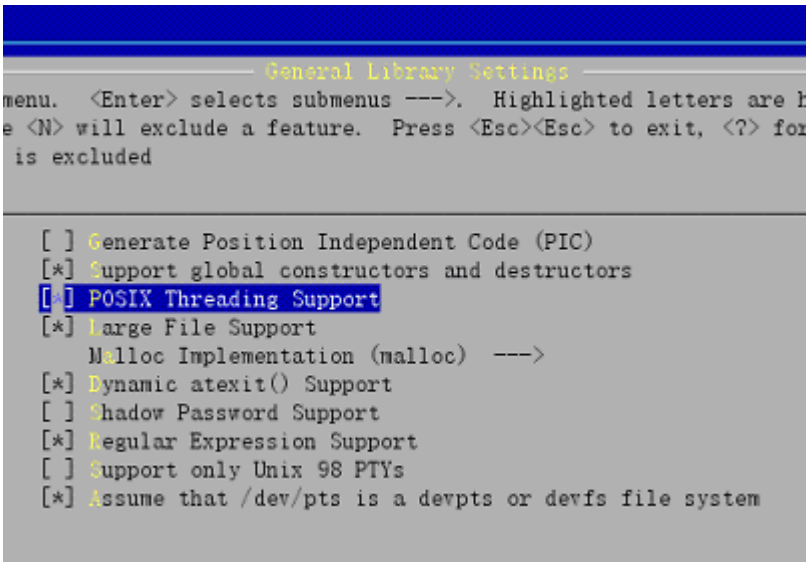

图 4-16 uClibc 的 POSIX 线程支持

【提示】uClibc 所带的线程库支持在许多平台上的实现非常不稳定，因此，建议在 uClinux 上运行 MiniGUI-Standalone 模式。本书所举 MiniGUI 示例，是以 MiniGUI-Standalone 模式运行的。

其他选项可保持默认值不变，保存配置并退出配置界面，然后我们即可在 uClinux-dist 目录运行下面的命令编译内核及 uClibc 库：

```
root# cd /opt/uclinux/uClinux-dist
root# make dep
root# make
```

上述命令还会进入 uClinux-dist 下的 *user/* 目录编译应用程序，并生成针对 Xcopilot 模拟器的 ROM 映像，即 *images/* 目录下的 *pilot.rom* 文件。

4.4.2 Xcopilot 模拟器的 MiniGUI 输入引擎

MiniGUI-STR 中已经包含了针对 Xcopilot 模拟器的输入引擎，完整的源代码可见 MiniGUI 源代码包中的 *src/ial/mc68x328.h* 和 *src/ial/mc68x328.c* 文件。下面我们简单介绍一下这个输入引擎。

该输入引擎的名称为“MC68X328”，在 MiniGUI 源代码包的 *src/ial/ial.c* 文件中定义了该输入引擎的入口项：

```
...
```

```
#ifdef _MC68X328_IAL
    #include "mc68x328.h"
#endif

...

static INPUT inputs [] =
{
...

#ifdef _MC68X328_IAL
    {"MC68X328", InitMC68X328Input, TermMC68X328Input},
#endif

...

}
```

上述代码中的 *inputs* 结构数组中保存了所有 MiniGUI 的输入引擎入口项。在 MiniGUI 的初始化过程中，MiniGUI 将根据 MiniGUI.cfg 配置文件中设定的输入引擎名称在该数组中寻找匹配项，然后调用对应的初始化函数，并在终止时调用终止函数。对 Xcopilot 模拟器来讲，输入引擎的名称为“ MC68X328 ”，初始化和终止函数分别为：*InitMC68X328Input* 和 *TermMC68X328Input*。这两个函数在 *mc68x328.h* 头文件中声明，在 *mc68x328.c* 文件中实现：

```
static int ts = -1;

...

BOOL InitMC68X328Input (INPUT* input, const char* mdev, const char* mtype)
{
    int err;
    struct ts_drv_params  drv_params;
    int mx1, mx2, my1, my2;
    int ux1, ux2, uy1, uy2;

     /*
     * open up the touch-panel device.
     * Return the fd if successful, or negative if unsuccessful.
     */

    ts = open("/dev/ts", O_NONBLOCK | O_RDWR);
    if (ts < 0) {
        fprintf (stderr, "Error %d opening touch panel\n", errno);
        return FALSE;
    }

    err = ioctl(ts, TS_PARAMS_GET, &drv_params);
    if (err == -1) {
        close (ts);
        return FALSE;
    }

    drv_params.version_req    = MC68328DIGI_VERSION;
    drv_params.event_queue_on = 1;
    drv_params.sample_ms      = 10;
#if LINUX_VERSION_CODE > KERNEL_VERSION(2, 4, 0)
    drv_params.deglitch_on    = 1;
#else
    drv_params.deglitch_ms    = 0;
#endif
    drv_params.follow_thrs    = 0;
    drv_params.mv_thrs        = 2;
    drv_params.y_max          = 159 + 66;  // to allow scribble area
    drv_params.y_min          = 0;
    drv_params.x_max          = 159;
    drv_params.x_min          = 0;
    drv_params.xy_swap        = 0;
```

```
    /*
     * according to mc68328digi.h 'How to calculate the parameters',
     * we have measured:
     */
    mx1 = 508; ux1 =   0;
    my1 = 508; uy1 =   0;
    mx2 = 188; ux2 = 159;
    my2 = 188; uy2 = 159;

    /*
     * now calculate the params:
     */
    drv_params.x_ratio_num    = ux1 - ux2;
    drv_params.x_ratio_den    = mx1 - mx2;
    drv_params.x_offset       =
    ux1 - mx1 * drv_params.x_ratio_num / drv_params.x_ratio_den;

    drv_params.y_ratio_num    = uy1 - uy2;
    drv_params.y_ratio_den    = my1 - my2;
    drv_params.y_offset       =
    uy1 - my1 * drv_params.y_ratio_num / drv_params.y_ratio_den;

    err = ioctl(ts, TS_PARAMS_SET, &drv_params);
    if (err == -1) {
        close (ts);
        return FALSE;
    }

    input->update_mouse = mouse_update;
    input->get_mouse_xy = mouse_getxy;
    input->set_mouse_xy = NULL;
    input->get_mouse_button = mouse_getbutton;
    input->set_mouse_range = NULL;

    input->update_keyboard = NULL;
    input->get_keyboard_state = NULL;
    input->set_leds = NULL;

    input->wait_event = wait_event;
    return TRUE;
}
```

观察 *InitMC68X328Input* 函数，我们可以看到，该函数打开了 */dev/ts* 设备文件，然后调用 *iotcl* 系统调用获得了一些参数，并根据 LCD 的显示分辨率等信息重新设置了参数。之后，该函数填充了 *INPUT* 结构的其他成员。

【提示】关于 INPUT 结构的描述，可参阅笔者文章“MiniGUI 体系结构之四: 图形抽象层和输入抽象层及 Native Engine 的实现”:

http://www-900.ibm.com/developerworks/cn/linux/embed/minigui/minigui-8/index.shtml

http://www-900.ibm.com/developerworks/cn/linux/embed/minigui/minigui-9/index.shtml

TermMC68X328Input 函数只完成了一件事情，即关闭 */dev/ts* 设备文件：

```
void TermMC68X328Input (void)
{
    if (ts >= 0)
        close (ts);
    ts = -1;
}
```

该文件中的其他函数用来实现输入引擎具体工作，比如监听设备文件、读取设备文件中出入的点击和移动操作等等。监听设备文件的工作在 *wait_event* 函数中实现：

```
#ifdef _LITE_VERSION
static int wait_event (int which, int maxfd, fd_set *in, fd_set *out, fd_set *except,
                struct timeval *timeout)
#else
static int wait_event (int which, fd_set *in, fd_set *out, fd_set *except,
                struct timeval *timeout)
#endif
{
    fd_set rfds;
    int    retvalue = 0;
    int    e;

    if (!in) {
        in = &rfds;
        FD_ZERO (in);
    }

    if ((which & IAL_MOUSEEVENT) && ts >= 0) {
        FD_SET (ts, in);
#ifdef _LITE_VERSION
        if (ts > maxfd) maxfd = ts;
#endif
    }

#ifdef _LITE_VERSION
    e = select (maxfd + 1, in, out, except, timeout) ;
#else
    e = select (FD_SETSIZE, in, out, except, timeout) ;
#endif

    if (e > 0) {
        if (ts >= 0 && FD_ISSET (ts, in)) {
            FD_CLR (ts, in);
            retvalue |= IAL_MOUSEEVENT;
        }
    }
    else if (e < 0) {
        return -1;
    }

    return retvalue;
}
```

该函数调用了 *select* 系统调用在 *ts* 文件描述符上监听可读数据，当有数据可读时，向 MiniGUI 上层返回 *IAL_MOUSEEVENT* 标志。MiniGUI 上层代码接着会调用 *mouse_update* 函数，请求输入引擎更新鼠标位置及按钮信息：

```
static int pen_down = 0; /* pen down */

static int mousex = 79, mousey = 79;

/*********************  Low Level Input Operations *********************/
/*
 * Mouse operations -- Event
 */
static int mouse_update(void)
{
    struct ts_pen_info pen_info;
    int bytes_read;
```

```
    if (ts < 0)
        return 0;

    bytes_read = read (ts, &pen_info, sizeof (pen_info));

    if (bytes_read != sizeof (pen_info)) {
        return 0;
    }

    switch(pen_info.event) {
    case EV_PEN_UP:
        pen_down = 0;
        mousex = pen_info.x;
        mousey = pen_info.y;
        break;

    case EV_PEN_DOWN:
        pen_down = IAL_MOUSE_LEFTBUTTON;
        mousex = pen_info.x;
        mousey = pen_info.y;
        break;

    case EV_PEN_MOVE:
        pen_down = IAL_MOUSE_LEFTBUTTON;
        mousex = pen_info.x;
        mousey = pen_info.y;
        break;
    }

    return 1;
}
```

该函数调用 *read* 系统调用从 *ts* 设备上读取数据，并根据设备的返回值更新位置和按钮状态。若 *mouse_update* 函数返回 1，则 MiniGUI 上层会调用 *mouse_getxy* 和 *mouse_getbutton* 函数获得鼠标新的位置和按钮的点击信息：

```
static void mouse_getxy (int *x, int* y)
{
    *x = mousex;
    *y = mousey;
}

static int mouse_getbutton (void)
{
    return pen_down;
}
```

需要注意的是，该输入引擎不支持鼠标位置的设置，也不处理 Xcopilot 模拟器的按键信息，因此，该输入引擎 *INPUT* 结构对应的相关函数被置成了 NULL：

```
    input->set_mouse_xy = NULL;
...
    input->set_mouse_range = NULL;

    input->update_keyboard = NULL;
    input->get_keyboard_state = NULL;
    input->set_leds = NULL;
```

另外，该输入引擎的编译配置选项为 *--enable-mc68x328ial*，默认是关

闭的。如果要在 MiniGUI 库中包含该输入引擎，则需要在配置 MiniGUI 时使用如下的配置选项：

```
--enable-mc68x328ial
```

该配置选项将打开*_MC68X328_IAL* 宏的定义，从而确保在 MiniGUI 中增加该输入引擎的相关代码。

4.4.3 为 Xcopilot 模拟器交叉编译 MiniGUI-STR

我们可以直接使用 MiniGUI-STR 源代码包中包含的配置脚本来为 Xcopilot 模拟器配置并交叉编译 MiniGUI-STR。我们首先看这个配置脚本的内容。进入 MiniGUI-STR 源代码目录：

```
user$ cd libminigui-str-1.6.2
```

打开 build/ 目录下的 *buildlib-uclinux-xcopilot* 文件。该文件的内容如下：

```
#!/bin/sh

#
# Please make sure that the following things are OK:
#
#   1. You have installed the m68k-elf-tools already.
#   2. Your uClinux distribution has been installed into
#      the directory of '/opt/uclinux/uClinux-dist'.
#   3. You configured the uClinux to use uClibc, and have
#      configured uClibc to support pthread.
#   4. You have made the uClinux distribution.
#   5. You have made a symbol link of "uClibc" to the correct uClibc directory.
#

rm config.cache config.status -f

CC=m68k-elf-gcc \
CFLAGS="-m68000 -Os -I/opt/uclinux/uClinux-dist/lib/uClibc/include \
-I/opt/uclinux/uClinux-dist/linux-2.4.x/include -fno-builtin \
-mid-shared-library -mshared-library-id=0 " \
LDFLAGS="-Wl,-elf2flt -Wl,-move-rodata -Wl,-shared-lib-id,0 -Wl,-elf2flt -Wl,-move-rodata \
-L/opt/uclinux/uClinux-dist/lib/uClibc/lib \
-Wl,-R,/opt/uclinux/uClinux-dist/lib/uClibc/libc.gdb -lc" \
./configure --prefix=/opt/uclinux/uClinux-dist/minigui/m68k-elf/ \
    --build=i386-linux \
    --host=m68k-elf-linux \
    --target=m68k-elf-linux \
    --disable-shared \
    --with-osname=uclinux \
    --with-style=flat \
    --enable-lite \
    --enable-standalone \
    --disable-micemoveable \
    --disable-cursor \
    --enable-fblin1l \
    --disable-fblin8 \
    --disable-fblin16 \
    --disable-fblin32 \
```

```
    --disable-textmode \
    --enable-dummyial \
    --enable-mc68x328ial \
    --disable-nativeial \
    --disable-qvfbial \
    --disable-latin9support \
    --disable-gbksupport \
    --disable-big5support \
    --disable-savebitmap \
    --disable-jpgsupport \
    --disable-pngsupport \
    --disable-imegb2312 \
    --disable-aboutdlg \
    --disable-savescreen \
    --enable-grayscreen \
    --enable-tinyscreen
```

接下来我们详细解释一下该配置脚本中的设置。

```
CC=m68k-elf-gcc
```

该设置告诉 configure 脚本在生成 makefile 文件时，使用 *m68k-elf-gcc* 来编译 C 程序。

```
CFLAGS="-m68000 -Os -I/opt/uclinux/uClinux-dist/lib/uClibc/include \
-I/opt/uclinux/uClinux-dist/linux-2.4.x/include -fno-builtin \
-mid-shared-library -mshared-library-id=0 "
```

该设置告诉 configure 脚本在生成 makefile 文件时，使用指定的 C 编译选项来编译生成 C 的目标文件。上面这些编译选项在本书第 3.1 节中解释过了。在这里，我们通过 *-I/opt/uclinux/uClinux-dist/linux-2.4.x/include* 选项指定了 Linux 内核头文件的路径，主要是因为 MiniGUI 库中使用了 Linux 内核的头文件。

```
LDFLAGS="-Wl,-elf2flt -Wl,-move-rodata -Wl,-shared-lib-id,0 -Wl,-elf2flt -Wl,-move-rodata \
-L/opt/uclinux/uClinux-dist/lib/uClibc/lib \
-Wl,-R,/opt/uclinux/uClinux-dist/lib/uClibc/libc.gdb -lc"
```

该设置告诉 configure 脚本，在生成 makefile 文件时，使用指定的链接选项来链接生成可执行文件。这些选项在本书 3.1 节中已经解释过了。其实编译 MiniGUI 函数库本身并不需要生成可执行文件，但在 configure 脚本的执行过程中，它要调用由 CC 指定的编译器来判断编译器是否能有效生成可执行程序，因此，我们要指定正确的 *LDFLAGS* 选项（不添加这些选项而直接调用 *m68k-elf-gcc* 命令，无法正确生成可执行程序）。

```
./configure --prefix=/opt/uclinux/uClinux-dist/minigui/m68k-elf/ \
```

--prefix 选项设定了安装 MiniGUI 配置文件、函数库及头文件的目录前缀，在执行 *make install* 命令时，将把 MiniGUI 配置文件、库文件和头文件分别安装到如下位置：

- */opt/uclinux/uClinux-dist/minigui/m68k-elf/etc/*
- */opt/uclinux/uClinux-dist/minigui/m68k-elf/lib/*
- */opt/uclinux/uClinux-dist/minigui/m68k-elf/include/*

```
--build=i386-linux \
--host=m68k-elf-linux \
--target=m68k-elf-linux \
```

上述 configure 的命令行选项分别指定了交叉编译环境：在 i386 系统上运行交叉编译工具链（build），宿主机及目标机为 m68k-elf（host、target）。

```
--disable-shared \
```

因为 uClinux 不支持共享库，因此，该选项告诉 configure 脚本，不要试图编译生成 MiniGUI 的共享库。

```
--with-osname=uclinux \
--with-style=flat \
--enable-lite \
--enable-standalone \
--disable-micemoveable \
--disable-cursor \
--enable-fblin1l \
--disable-fblin8 \
--disable-fblin16 \
--disable-fblin32 \
--disable-textmode \
--enable-dummyial \
--enable-mc68x328ial \
--disable-nativeial \
--disable-qvfbial \
--disable-latin9support \
--disable-gbksupport \
--disable-big5support \
--disable-savebitmap \
--disable-jpgsupport \
--disable-pngsupport \
--disable-imegb2312 \
--disable-aboutdlg \
--disable-savescreen \
--enable-grayscreen \
--enable-tinyscreen
```

上面这些选项是 MiniGUI 本身的配置选项，和 Xcopilot 模拟器特别相关的选项是 *--enable-fblin1l* 和 *--enable-mc68x328ial*。前者打开了 MiniGUI fbcon 引擎的单色、左边为高位（即每个字节表示八个象素，高

位表示靠左边的象素；有些 LCD 控制器的高位表示的是靠右边的象素）的 FrameBuffer 引擎；后者打开了针对 Xcopilot 模拟器的输入引擎。其他的选项可参阅本书附录 B “ MiniGUI-STR 配置选项详解 ”。注意，本配置脚本将 MiniGUI 配置成了 MiniGUI-Standalone 模式。

在 MiniGUI-STR 源代码目录下，可直接运行该脚本配置 MiniGUI，然后运行 *make* 和 *make install* 命令：

```
user$ cd libminigui-str-1.6.2
user$ ./build/buildlib-uclinux-xcopilot
user$ make clean; make
user$ su –c ‘make install’
```

在成功运行上述命令之后，我们可在通过 *--prefix* 选项指定的目录下看到 MiniGUI 的头文件及交叉编译后的函数库文件：

```
user$ cd /opt/uclinux/uClinux-dist/minigui/m68k-elf
user $ ls -l -R
.:
total 12
drwxr-xr-x    2 root     root         4096 Mar 12 00:48 etc
drwxrwxr-x    3 root     root         4096 Mar 12 00:48 include
drwxrwxr-x    2 root     root         4096 Mar 12 00:48 lib

./etc:
total 8
-rw-r--r--    1 root     root         5138 Mar 12 00:48 MiniGUI.cfg

./include:
total 4
drwxrwxr-x    3 root     root         4096 Mar 12 00:48 minigui

./include/minigui:
total 612
-rw-r--r--    1 root     root        37838 Mar 12 00:48 common.h
-rw-r--r--    1 root     root        12360 Mar 12 00:48 config.h
-rw-r--r--    1 root     root         3410 Mar 12 00:48 control.h
drwxrwxr-x    2 root     root         4096 Mar 12 00:48 ctrl
-rw-r--r--    1 root     root         2250 Mar 12 00:48 dti.c
-rw-r--r--    1 root     root        23704 Mar 12 00:48 endianrw.h
-rw-r--r--    1 root     root        17634 Mar 12 00:48 fixedmath.h
-rw-r--r--    1 root     root       220619 Mar 12 00:48 gdi.h
-rw-r--r--    1 root     root        69897 Mar 12 00:48 minigui.h
-rw-r--r--    1 root     root       202605 Mar 12 00:48 window.h

./include/minigui/ctrl:
total 212
-rw-r--r--    1 root     root        14761 Mar 12 00:48 button.h
-rw-r--r--    1 root     root        24041 Mar 12 00:48 combobox.h
-rw-r--r--    1 root     root        13360 Mar 12 00:48 ctrlhelper.h
-rw-r--r--    1 root     root        21828 Mar 12 00:48 edit.h
-rw-r--r--    1 root     root        27000 Mar 12 00:48 listbox.h
-rw-r--r--    1 root     root         9615 Mar 12 00:48 menubutton.h
-rw-r--r--    1 root     root        10294 Mar 12 00:48 newtoolbar.h
-rw-r--r--    1 root     root         5653 Mar 12 00:48 progressbar.h
-rw-r--r--    1 root     root         9601 Mar 12 00:48 propsheet.h
-rw-r--r--    1 root     root         2875 Mar 12 00:48 scrollbar.h
-rw-r--r--    1 root     root        24187 Mar 12 00:48 scrollview.h
-rw-r--r--    1 root     root         7190 Mar 12 00:48 static.h
-rw-r--r--    1 root     root         2966 Mar 12 00:48 textedit.h
-rw-r--r--    1 root     root         4107 Mar 12 00:48 toolbar.h
-rw-r--r--    1 root     root        10576 Mar 12 00:48 trackbar.h

./lib:
total 580
```

```
-rw-r--r--    1 root     root        585068 Mar 12 00:48 libminigui.a
-rwxr-xr-x    1 root     root           742 Mar 12 00:48 libminigui.la
```

至此，我们编译好了针对 Xcopilot 模拟器的 MiniGUI-STR，并将头文件和库函数安装到了指定的位置。接下来，我们编译并运行 MiniGUI 应用程序。

4.4.4 在 Xcopilot 模拟器上运行 MiniGUI 示例程序

1) 交叉编译 MiniGUI 示例程序

编译 MiniGUI 应用程序有两种办法：第一种办法是按照本书 3.2.3 节介绍的那样，把 MiniGUI 应用程序添加到 uClinux-dist 的 *user* 目录中，在 uClinux-dist 中进行编译和链接，直至生成最后的映像文件；另一种办法是先单独编译 MiniGUI 程序，生成可执行程序文件，然后放到 uClinux-dist 的 *romfs* 目录中，执行 *make image* 命令生成映像。

我们先介绍第一种方法。在 *uClinux-dist/user* 目录下建立一个 *mgdemo* 目录，然后从 *mg-samples-str* 包中复制 *helloworld.c* 文件：

```
root# cd /opt/uclinux/uClinux-dist/user
root# mkdir mgdemo
root# cd mgdemo
root# cp <path to mg-samples-str>/src/helloworld.c .
```

然后在该目录下建立 *Makefile* 文件，其内容如下：

```
EXEC = mghello
OBJS = helloworld.o

CFLAGS+=-I/opt/uclinux/uClinux-dist/minigui/m68k-elf/include
LDFLAGS+=-L/opt/uclinux/uClinux-dist/minigui/m68k-elf/lib

LDLIBS+=-lminigui

all: $(EXEC)

$(EXEC): $(OBJS)
    $(CC) $(LDFLAGS) -o $@ $(OBJS) $(LDLIBS$(LDLIBS_$@))

romfs:
    $(ROMFSINST) /bin/$(EXEC)

clean:
    -rm -f $(EXEC) *.elf *.gdb *.o
```

最后修改 *uClinux-dist/user/* 目录下的 *Makefile* 文件，加上一行：

```
dir_y += mgdemo
```

这样在 uClinux-dist 目录下输入“ *make* ”进行编译时就把该示例程序包括在编译范围之内，编译后将在 *romfs/bin* 目录下生成 *mghello* 程序。

上述针对 *mgdemo* 的 makefile 文件和本书 3.2.3 节中针对 *mycal* 程序的 makefile 文件在如下两个方面有所不同：

- 它使用 *CFLAGS* 变量指定了额外的头文件搜索路径，这里是 MiniGUI 头文件所在路径。
- 它使用了 *LDFLAGS* 变量指定了额外的库文件搜索路径，这里是 MiniGUI 库文件所在路径。
- 它使用了 *LDLIBS* 变量指定了生成可执行程序时额外要链接的库，这里是 *-lminigui*。

单独编译 MiniGUI 程序的方法可参照 3.2 所讲的方法进行，要注意的是需要通过 *–I*、*–L* 以及 *–l* 选项指定头文件、库文件搜索路径以及要链接的函数库名称。当然，如果您使用 *mg-samples-str* 包，我们还可以直接使用预先准备的 *configure* 脚本生成可用于交叉编译的 makefile 文件：

```
user$ cd <path to mg-samples-str-1.6.2>
user$ ./build-uclinux-xcopilot
user$ make
```

运行上述命令之后，将交叉编译生成 mg-samples-str 包中针对 Xcopilot 的所有示例程序。

2) 准备文件系统

到目前为止，我们已经编译好了运行 MiniGUI 的应用程序，应该可以在 Xcopilot 上运行了。然而，MiniGUI 所使用的配置文件，字体、位图等资源文件还没有集成到 ROM 文件系统的映像中，这时运行 MiniGUI 示例程序，我们将得到下面的错误信息：

```
Welcome to
          ____ _  _
         /  __| ||_|
    _    _| |   | |  _ ____  _    _  _  _
   | |  | | |   | || |  _ \| | | |\ \/ /
   | |_| | |__| || | | | | | |_| |/    \
   |  ___\____|_||_|_| |_|\____|\_/\_/
   | |
   |_|

For further information check:
http://www.uclinux.org/

Execution Finished, Exiting
```

```
Sash command shell (version 1.1.1)
/> cd bin
/bin> ./mghello
MISC: Can not locate your MiniGUI.cfg file or bad files!
DESKTOP: Initialization of misc things failure!
pid 10: failed 256
/bin>
```

因此，我们还需要将这些文件放到 uClinux-dist 的 *romfs/* 目录下，然后重新生成 ROM 映像文件。

首先我们在 *romfs* 中建立相应的目录，并将 MiniGUI 的运行时配置文件放到 *romfs* 中：

```
root# cd /opt/uclinux/uClinux-dist/
root# mkdir romfs/usr/local/etc -p
root# cp minigui/m68k-elf/etc/MiniGUI.cfg romfs/usr/local/etc/
```

接下来，我们要根据 MiniGUI.cfg 中的设置，将 MiniGUI 运行时所需要的位图、字体、图标和鼠标光标复制到对应的目录中。一个简单的办法是，修改 minigui-res-str 包中的配置信息，然后运行 *make install* 让 *make* 命令帮我们复制这些文件。打开 minigui-res-str 包中的 *config.linux* 文件，并修改 *TOPDIR*：

```
TOPDIR=/opt/uclinux/uClinux-dist/romfs
```

然后以超级用户身份运行 *make install* 命令，该命令将把 MiniGUI 所需资源文件复制到 TOPDIR 设定的目录树底下，这里就是 uClinux-dist 的 *romfs* 目录。

【提示】 minigui-res-str 包中包含了针对不同风格的位图信息，如果目标系统的储存资源紧张，则应该将 MiniGUI.cfg 中未指定的文件从文件系统中删除。

另外，我们还需要修改 *MiniGUI.cfg* 文件，以便为 Xcopilot 模拟器指定正确的输入引擎名称。打开 *uClinux-dist* 目录下的 *romfs/usr/local/etc/MiniGUI.cfg* 文件，然后在 *[system]* 段中做如下修改：

```
[system]
# GAL engine
gal_engine=fbcon

# IAL engine
ial_engine=MC68X328
```

```
mdev=/dev/ts
mtype=unknown
```

最后，我们在 uClinux-dist 目录下运行 *make* 命令，最终将生成 ROM 映像文件。

如果您在 mg-samples-str 包中单独编译 MiniGUI 应用程序，则可以将编译后的程序复制到 *romfs/* 中，最后运行 *make image* 命令生成映像文件。比如：

```
root# cp <path to mg-samples-str>
root# cp src/helloworld /opt/uclinux/uClinux-dist/romfs/bin/
root# cd /opt/uclinux/uClinux-dist
root# make image
```

接下来我们就可以在 Xcopilot 模拟器中运行 MiniGUI 示例程序了。

3) 运行 MiniGUI 示例程序

确保 Xcopilot 使用正确的 ROM 映像文件：

```
user$ cd ~/.xcopilot
user$ ln -s /opt/uclinux/uClinux-dist/images/pilot.rom
```

在 X Window 的终端仿真程序中启动 Xcopilot 模拟器 [31]：

```
user$ cd <path to xcopilot>
user$ ./xcopilot
```

在 Xcopilot 模拟器中运行 MiniGUI 示例程序：

```
Welcome to
          ____ _  _
         /  __| ||_|
   _   _| |  | | _ ____  _   _  _  _
  | | | | |  | || |  _ \| | | |\ \/ /
  | |_| | |__| || | | | | |_| |/    \
  |  ___\____|_||_|_| |_|\____|\_/\_/
  | |
  |_|

For further information check:
http://www.uclinux.org/

Execution Finished, Exiting

Sash command shell (version 1.1.1)
```

[31] 要在 Xcopilot 上运行 MiniGUI 应用程序，请使用产品光盘中的 /xcopilot/xcopilot-0.6.6-uc0-mg.tar.gz 包编译生成 xcopilot 可执行程序。该 Xcopilot 版本做了少量修改，以便有更多的内存来运行 MiniGUI 运行程序。

```
/> cd bin
/bin> ./mghello
```

MiniGUI “Hello, world”示例程序将在 Xcopilot 模拟器上显示，如图 4-17 所示。我们可以用鼠标单击 Xcopilot 模拟器的 LCD 屏幕（相当于点击实际设备的触摸屏），将看到该示例程序将打印信息表明接收到了鼠标点击事件。

图 4-17 Xcopilot 上的 MiniGUI 示例程序

4.5 小结

本章我们了解到 MiniGUI 在嵌入式系统中的作用，并在 PC 上运行了 MiniGUI 及其示例程序。通过本章的学习，读者应该学会如何在 PC 上运行并开发 MiniGUI 应用程序，掌握将 MiniGUI 及其应用程序移植到嵌入式 Linux/uClinux 上的方法。如果读者理解了移植 MiniGUI 应用程序的基本原理和相关工具，相信也能非常容易地移植其他大型的嵌入式应用软件。

我们在前面的章节中，主要通过两个模拟器来学习嵌入式 Linux/uClinux 开发中的基本原理及工具的使用，接下来我们将介绍两款实际的开发板，通过在这些开发板上运行 MiniGUI，让读者对实际的嵌入

式开发有个大致的了解。其实，在开发过程、工具的使用上，模拟器和实际开发板之间没有本质的区别。

5 MiniGUI 和常见嵌入式 Linux/uClinux 开发板

通过前面几章的描述，相信读者对嵌入式 Linux/uClinux 的开发原理和相关工具已经相对熟悉了。然而，我们的讲述大多集中在软件的模拟器上，为了让读者对实际的硬件开发板有一个了解，本章将介绍两种国内常见的开发板，讲述如何在这些开发板上运行 MiniGUI。

其实，实际硬件开发板和软件模拟器的最大区别在于，前者通常拥有自己独特的映像制作及烧写工具，而这些工具通常和引导装载器有关。硬件开发板随附的文档会详细描述如何使用这些工具。

在本章的示例中，我们并没有将 MiniGUI 及其示例程序烧写到开发板中，而采用了 NFS 文件系统的方式。这是一种非常常见的开发方式，通过这种方法，我们可以方便地在实际目标板上运行程序，而不需要每次将应用程序或者整个文件系统映像烧写到目标板上。这也是嵌入式 Linux/uClinux 带给我们的极大便利，而在传统的嵌入式系统开发中，通常无法使用这种手段。

5.1 博创 ARM3000 开发板

5.1.1 开发板基本配置

该开发板的硬件配置可见表 5-1，图 5-1 是该开发板的外观。

表 5-1 博创 ARM3000 开发板的配置

配置名称	型号	规格
CPU	S3C44B0X01	
以太网	RTL8019AS	
FLASH 盘	SAMSUNG 9F2808L10B	16MB
BIOS 盘	AM29LV160DB	2MB
LCD		320*240 256 色
内存	HY57V561620BT-H	32M/8M

图 5-1 博创 ARM3000 开发板

5.1.2 建立开发环境及运行环境

以下操作过程可参考开发板附带光盘中的文档《NET-ARM3000-uClinux 开发指南》。

1) 安装开发环境（包括交叉编译器及 uClinux 内核，uClibc）

进入 Linux 系统，将随开发板附带的光盘放入光驱中，一般情况下系统会自动挂装 CD-ROM 到 */mnt/cdrom* 目录。用 root 用户身份执行开发光盘中的 *install.sh* 脚本：

```
user$ su -
password:<输入root用户的口令>
root# cd /mnt/cdrom
root# ./install.sh
```

如果没有出错信息，表示安装已经完成，uClinux 内核及 uClibc 安装的位置为 */uclinux* 目录，交叉编译器的安装目录为 */usr/arm-linux-uclibc*。

2) 设置环境变量 PATH

修改环境变量 PATH （使用 BASH Shell 的用户可中修改用户主目录下的 *.bash_profile* 文件）：

```
export PATH=/usr/arm-linux-uclibc/bin:$PATH
```

可重新登陆系统使环境变量生效，也可在用户主目录下执行：

```
user$ . .bash_profile
```

使环境变量在当前会话中生效。

3) 编译 uClinux 内核

使用 root 用户编译内核：

```
root# cd /uclinux/uClinux-2.4.x
root# make dep
root# make
```

如编译过程没有出错，则表示 uClinux 内核编译成功。

4) 配置并运行 minicom

使用 root 用户运行 *minicom*，并进行配置：

```
root# minicom -s
```

配置串口参数，其基本配置为：

```
A——Serial Device: /dev/ttyS0 （端口号使用串口1，根据实际连接的串口来设置）
E——BPS/par/bits: /115200 8N1 （波特率）
F，E 硬件流，软件流都改为NO
```

设置好后，选择“Save setup as df1”保存退出。

重新运行 *minicom* 命令:

```
root# minicom
```

5) 启动开发板

准备好开发板后，使用串口线连接主机与开发板，开启开发板电源。开发板中已有的引导器将启动。在 *minicom* 中看到提示符后键入 *boot* ，就可启动目标板系统，启动后自动以 root 权限进入系统。

6) 配置主机的 NFS

在主机的 */etc/exports* 中添加 NFS 目录共享信息， 如：

```
/uclinux/   *(rw,sync)
```

保存后，用 root 用户重启 NFS 服务：

```
root# /etc/init.d/nfs restart
```

7) 配置开发板 IP 地址

使用普通网线将开发板连接到主机所在的局域网，或使用直连网线将开发板与主机连接起来，根据主机所在网段配置开发板的 IP 地址。在 *minicom* 中执行：

```
root# ifconfig eth0 192.168.1.199     # 请根据主机所在网段确定该地址
```

8) 将主机 NFS 的共享目录挂装到开发板上

在 minicom 中执行：

```
# mount -o nolock -t nfs 192.168.1.11:/uclinux   /home
```

其中 192.168.1.11 为导出 NFS 共享的主机地址，*/home* 为开发板上的目录。

9) 编译 MiniGUI-STR 库

```
user$ cd  <path to libminigui-str-1.6.2>
```

```
user$ ./build/buildlib-uclinux-arm3000
user$ make clean; make
user$ su -c 'make install'
```

其中 *buildlib-uclinux-arm3000* 是为此开发板编写好的 MiniGUI 编译配置脚本。运行 *make install* 命令后， MiniGUI-STR 将被安装到*/uclinux* 目录下的 *minigui/* 目录中。

10) 编译 MiniGUI-STR 示例程序

```
user$ cd <path to mg-samples-str-1.6.2>
user$ ./build-uclinux-arm3000
user$ make clean; make
```

编译完成后，在 *src* 目录下会有许多可执行程序产生，在*/uclinu*x 下有一个 *demos* 目录，可将这些可执行程序拷贝到其中：

```
root# cp src/helloworld /uclinux/demos
```

11) 准备 MiniGUI-STR 的资源文件

MiniGUI 程序运行时会根据配置文件读取 MiniGUI 的资源，如图片、字体文件等。将 MiniGUI-STR 的资源包在主机上安装后，将主机 */usr/local/lib/* 目录下的 *minigui/* 目录整个拷贝到 */uclinux/demos* 目录下。

12) 修改 MiniGUI 的配置文件 MiniGUI.cfg

要使编译好的程序能够在该板子上运行，还要针对该板子对 MiniGUI 的运行时配置文件作相应的修改。将*/uclinux/minigui/etc* 下的 *MiniGUI.cfg* 文件拷贝到*/uclinux/demos* 下，并作如下修改：

```
[system]
gal_engine=fbcon
ial_engine=arm3000

[fbcon]
defaultmode=320x240-8bpp
```

为了保证 MiniGUI 程序能找到资源文件的位置，需将 *MiniGUI.cfg* 中指定资源文件的路径由缺省的*/usr/local/lib* 改为*/home/demos*，其中 */home* 为目标开发板挂装 NFS 共享的目录。

MiniGUI 程序运行时，首先会在当前目录下寻找 MiniGUI.cfg，如果

没有找到，则会到当前用户的主目录下去查找 *.MiniGUI.cfg* 文件，如果还没有找到，则按顺序会在 */usr/local/etc*、*/etc* 以及当前目录下去查找 MiniGUI.cfg 文件。本示例中我们采用 NFS 方式来运行程序，所以将 MiniGUI.cfg 放在程序的当前工作目录下。

13) 运行 MiniGUI-STR 示例程序

在 minicom 中，执行下列命令

```
# cd /home/demos
# ./helloworld
```

此时可在 LCD 屏幕上看到 MiniGUI 程序的运行效果，用触摸笔点击 LCD 屏幕，可看到 helloworld 程序的动态输出。

5.1.3 配置脚本的说明

针对博创 ARM3000 开发板，专门编写了配置脚本 buildlib-uclinux-arm3000，位于 MiniGUI-STR 源码目录的 *build/* 子目录下。其中的编译参数设置及 MiniGUI 的配置选项说明如下：

编译参数的设置说明如下：

- `CC=arm-uclibc-gcc`：CC 变量一般用来指定编译器名称，不同的交叉编译工具的编译器名称一般不同，可以通过修改该名称来指定不同的编译器；
- `CFLAGS="-Wall -O2 -g -D__linux__ -I/uclinux/uClinux-2.4.x/include -I/uclinux/uClibc-0. 9.19/include -fno-builtin -nostartfiles "`：*CFLAGS* 变量用来设置编译器的一些参数；其中-*Wall* 表示显示编译过程中的所有警告；*-O2* 表示优化级别为 2；*-g* 表示为编译为调试版本，在程序中会包含调试信息；*-D__linux__* 定义 *__linux__* 这个宏；*-I/uclinux/uClinux-2.4.x/include* 指定系统的头文件路径；*-I/uclinux/uClibc-0.9.19/include* 指定 uClibc 的头文件路径；*-fno-builtin* 用来取消所有的内联函数；*-nostartfiles* 表示链接时不使用标准的启动文件；
- `LDFLAGS="-elf2flt -static`

`-L/uclinux/uClibc-0.9.19/lib -L/uclinux/uClinux-2.4.x/lib -lc"`：*LDFLAGS* 变量用于设置链接器的一些参数；其中，*-elf2flt* 表示指定将生成的 ELF 格式的可执行文件转换成 FLAT 格式的文件；*-static* 表示使用静态方式连接库；*-L/uclinux/uClibc-0.9.19/lib* 指定 uClibc 的库所在路径；*-L/uclinux/uClinux-2.4.x/lib* 指定 uClinux 系统的库所在的路径；*-lc* 指定要链接 libc 库；

MiniGUI 的配置参数说明如下：

- `--prefix=/uclinux/minigui`：指定编译 MiniGUI 后，执行 *make install* 时 MiniGUI 的安装目录，上述设定将把 MiniGUI 的头文件将安装到 */uclinux/minigui/include/* 目录下，MiniGUI 的库文件将安装到 */uclinux/minigui/lib* 目录下；
- `--build=i386-linux`：表示执行编译的环境为 i386-linux；
- `--host=arm-elf-linux`：表示主机类型 arm-elf-linux；
- `--target=arm-elf-linux`：表示目标平台类型为 arm-elf-linux，即编译器工具将生成的代码的系统类型
- `--disable-shared`：禁止产生共享库，只生成静态库，这样在编译应用程序时，MiniGUI 将自动以静态库方式链接到应用程序中，执行时只要一个可执行文件即可；
- `--with-osname=uclinux`：指定操作系统为 uclinux；
- `--enable-lite`：指定 MiniGUI 编译成进程版；
- `--enable-standalone`：和 –enable-lite 选项一同使用，则会将 MiniGUI 编译成 MiniGUI-Standalone 模式。
- `--disable-micemoveable`：禁止使用鼠标拖动窗口；
- `--disable-cursor`：禁止显示鼠标光标；
- `--enable-galfbcon`：包含针对 Linux FrameBuffer 的图形引擎；
- `--enable-fblin4l`：包含 4 位色图形子引擎。由于该板子是 8 位色的液晶屏，实际使用的是 fblin8 子引擎，MiniGUI-STR 中缺省情况下，包含了对 fblin8 的子引擎；
- `--enable-textmode`：打开 Linux 控制台文本模式的支持；
- `--enable-dummyial`：包含“哑”输入引擎，当没有针对特定板子的输入引擎时，可以将输入引擎配置成“哑”引擎，使 MiniGUI 程序也能正常运行；

- `--enable-autoial`：包含“自动”输入引擎，当没有针对特定板子的输入引擎时，也可以将输入引擎配置成“自动”输入引擎，使 MiniGUI 程序能正常运行，并自动模拟点击和按键等动作；
- `--enable-arm3000ial`：包括 arm3000 输入引擎，该输入引擎是专门针对博创 ARM3000 开发板编写的；
- `--disable-jpgsupport`：取消对 jpg 格式图片的支持，因为 uClibc 中没有 jpg 库；
- `--disable-pngsupport`：取消对 png 格式图片的支持，因为 uClibc 中没有 png 库；
- `--enable-mousecalibrate`：启用鼠标校正；通过启用鼠标校正可以调用 *SetMouseCalibrationParameters* 函数来设置校正参数，MiniGUI 会根据参数自动进行校正；函数 *SetMouseCalibrationParameters (const POINT* src_pts, const POINT* dst_pts)*有两个参数，*src_pts* 传递一组原始点的坐标点（一般为 5 个点），*dst_pts* 传递一组目标坐标点（一般为 5 个点），实际校正时，*dst_pst* 的坐标是固定的五个点，*src_pts* 的每个点的值是根据 *dst_pst* 的每个点的位置，执行点击动作而获得的值。

在编译 mg-samples-str 应用程序时，我们也使用了编辑好的配置脚本 *build-uclinux-arm300*，其中也有一些设置，简要说明如下：

- *CFLAGS* 中的参数*-I/uclinux/minigui/include*：指定 MiniGUI-STR 的头文件所在的位置，其中 */uclinux/minigui* 是在编译 MiniGUI-STR 时由*--prefix* 指定的路径；
- *LDFLAGS* 中的*/uclinux/uClibc-0.9.19/lib/crt0.o*：指明程序在链接时，要链接 *crt0.o* 文件，*crt0.o* 中包含了 C 程序的初始代码；
- *LDFLAGS* 中的*-L/uclinux/minigui/lib*：指定 MiniGUI-STR 的库文件所在的目录，其中*/uclinux/minigui* 是在编译 MiniGUI-STR 时由 *--prefix* 指定的路径；

5.1.4 MiniGUI 中与博创 ARM3000 开发板相关的源程序

MiniGUI 中与博创 ARM3000 开发板相关的源程序主要是输入引擎，对应文件为：

- *src/ial/arm3000.c*：针对该板子的输入引擎源码；
- *src/ial/arm3000.h*：针对该板子的输入引擎头文件。

5.1.5 注意事项和常见问题

应用程序不能运行，提示“GAL fbcon engine: Error when opening /dev/fb0”等类似信息，表示无法打开 */dev/fb0* 设备，则需要在 minicom 中执行：

```
#  ln -s /dev/fb0 /dev/fb/0
```

若提示“GAL fbcon engine: Can't open /dev/tty0"，表示无法打开 */dev/tty0* 设备，需要创建该设备，在 minicom 中执行：

```
# mknod /dev/tty0 c 4 0
```

若提示“GDI: Error in loading raw bitmap fonts.”，表示装载字体失败，有可能是 MiniGUI.cfg 中指定的资源路径不对，需检查资源文件的路径和 MiniGUI.cfg 中指定的路径是否一致。

5.2 华恒 HH2410R3 开发板

5.2.1 开发板基本配置

该开发板的硬件配置见表 5-2。

表 5-2 华恒 HH2410R3 开发板硬件配置

配置名称	型号	规格
CPU	S3C2410	32 位处理器，203MHz
FLASH 盘		16M 字节
LCD		（240x320） 256 色
内存		64M 字节 SDRAM

5.2.2 建立开发环境及运行环境

以下操作过程可参考随开发板附带光盘中的文档。

1) 安装开发环境（包括交叉编译器及 Linux 内核）

进入 Linux 系统（假定为 RedHat 9），将随开发板附带的光盘放入光驱中，一般情况下系统会自动挂装 CDROM 到 */mnt/cdrom* 目录下，用 root 用户执行开发光盘中的安装脚本：

```
user$ su -
root# cd /mnt/cdrom
root# ./arminstall
```

安装过程中会有一些确认信息，按 y 键后，按回车键即可完成整个开发环境的安装。

如果没有出错信息，表示安装已经完成。Linux 内核、应用程序源代码以及各个工具软件安装的位置为*/HHARM2410-R3* 目录，交叉编译工具安装路径为 */opt/host/armv4l*。

2) 设置环境变量 PATH

修改环境变量 PATH （使用 BASH Shell 的用户可修改用户主目录下的 *.bash_profile* 文件）：

```
export PATH=/opt/host/armv4l/bin:$PATH
```

可重新登录系统使环境变量生效，也可在用户主目录下执行下面的命令

```
user$ . .bash_profile
```

使环境变量在当前终端中生效。

3) 配置并运行 minicom

使用 root 用户运行 *minicom*， 并进行配置

```
root# minicom -s
```

配置串口参数，其基本配置为：

```
A——Serial Device ：/dev/ttyS0 （端口号使用串口1，根据实际连接的串口来设置）
E——BPS/par/bits ：/115200 8N1 （波特率）
F，E 硬件流，软件流都改为NO
```

设置好后，选择“ Save setup as dfl ”保存并退出。

重新运行 minicom：

```
root# minicom
```

4) 启动开发板

连接好开发板后，使用串口线连接主机与开发板，开启开发板电源。开发板中已有的操作系统将启动，在 minicom 中按回车键进入控制台。

5) 配置主机 NFS

在主机的 */etc/exports* 文件中添加 NFS 共享目录信息，如：

```
/demos   *(rw,sync)
```

保存后，用 root 用户重启 NFS 服务：

```
root# /etc/init.d/nfs restart
```

6) 配置开发板 IP 地址

使用普通网线将开发板连接到主机所在的局域网，或使用直连网线将开发板与主机连接起来，根据主机所在网段配置开发板的 IP 地址，在 minicom 中执行：

```
# ifconfig eth0 192.168.1.199     # 请根据主机所在网段确定该地址
```

7) 将主机 NFS 共享目录挂装到开发板上

在 minicom 中执行：

```
#  mount -t nfs 192.168.1.11:/demos   /home
```

其中 ***192.168.1.11*** 为主机地址。

8) 编译 MiniGUI-STR 库

```
user$ cd  <path to libminigui-str-1.6.2>
user$ ./build/buildlib-linux-hh2410r3
user$ make clean; make
user$ su -c 'make install'
```

此时会将 MiniGUI 编译并安装到如下目录：

/opt/host/armv4l/armv4l-unknown-linux/

9) 编译 MiniGUI-STR 示例程序

```
user$ cd <path to mg-samples-str-1.6.2>
user$ ./build-linux-hh2410r3
user$ make clean; make
```

编译完成后，在 *src/*目录下会有许多可执行程序产生，将可执行程序拷贝到 */demos* 中。

```
root# cp src/helloworld /demos
```

10) 准备 MiniGUI-STR 的资源文件

MiniGUI 程序运行时会根据配置文件读取 MiniGUI 的资源，如图片、字体文件等。将 MiniGUI 的资源包在主机上安装后，将主机 */usr/local/lib/* 目录下的 *minigui/*目录整个拷贝到 */demos* 目录下。

11) 修改 MiniGUI 的配置文件 MiniGUI.cfg

要使编译好的程序能够在该板子上运行，还要针对该板子，对配置文件作相应的修改。将 */opt/host/armv4l/armv4l-unknown-linux/etc* 下的 *MiniGUI.cfg* 文件拷贝到 */demos* 下，并作如下修改：

```
[system]
gal_engine=fbcon
ial_engine=hh2410r3
mdev=/dev/touchscreen/0raw

[fbcon]
defaultmode=240x320-8bpp
```

为了保证 MiniGUI 程序能找到资源文件的位置，需将 MiniGUI.cfg 中指定资源文件的路径由缺省的 */usr/local/lib* 修改为 */home*，其中 */home* 为开发板挂装 NFS 共享的目录。

MiniGUI 程序运行时，首先会在当前目录下寻找 MiniGUI.cfg，如果

没有找到，则会到当前用户的主目录下去查找 *.MiniGUI.cfg* 文件，如果还没有找到，则按顺序会在 */usr/local/etc*、*/etc* 以及当前目录下去查找 MiniGUI.cfg 文件。本示例中我们采用 NFS 方式来运行程序，所以将 MiniGUI.cfg 放在程序的当前工作目录下。

在该型号的板子上可能已有 MiniGUI V1.3 的程序，并且在 */etc* 中存在老版本的 MiniGUI.cfg 文件，这时运行 MiniGUI 程序，程序会读取 */etc/MiniGUI.cfg* 文件，而 1.3 版本的 MiniGUI.cfg 与 MiniGUI-STR 的配置文件有差别，所以程序会出错，因此，用户确保在当前目录中保存 MiniGUI-STR 对应的 MiniGUI.cfg 文件，然后再运行 MiniGUI 示例程序。

12) 运行 MiniGUI-STR 程序

在 minicom 中，执行下列命令

```
root# cd /home/
root# ./helloworld
```

此时可在 LCD 屏幕上看到 MiniGUI 程序的运行效果，使用触摸笔点击触摸屏，可看到程序的动态输出信息。

5.2.3 配置脚本的说明

针对华恒 HH2410R3 开发板，我们专门编写了配置脚本 *buildlib-linux-hh2410r3*，位于 MiniGUI 源码目录的 *build/* 子目录下。其中主要的配置选项说明如下：

- `--prefix=/opt/host/armv4l/armv4l-unknown-linux`：指定 MiniGUI 将安装到交叉编译工具中系统文件所在的目录，这样在编译 MiniGUI 的应用程序时不需要再使用*-I* 和*-L* 参数来指定 MiniGUI-STR 的头文件和库文件所在的目录；
- `--target=arm-unknown-linux`：指定目标系统为 arm-unknown-linux；
- `--disable-shared`：禁止产生共享库，只生成静态库，这样在编译应用程序时，MiniGUI 以静态库方式链接到应用程序中，执行时只要一个可执行文件即可；
- `--enable-hh2410r3ial`：使 MiniGUI 中包含针对 HH2410R3 开发板的输入引擎；

- `--with-style=phone`：指定 MiniGUI 在创建控件时，使用 phone 风格，缺省为 pc3d 风格；
- `--enable-fblin16`：包含 16 位线性图形子引擎；
- `--enable-tinyscreen`：表示目标板为小屏幕；

5.2.4 MiniGUI 中与华恒 HH2410R3 开发板相关的源程序

MiniGUI 中与华恒 HH2410R3 开发板相关的源程序主要是输入引擎，对应文件为：

- *src/ial/hh2410r3.c*：针对该板子的输入引擎源码；
- *src/ial/hh2410r3.h*：针对该板子的输入引擎头文件。

5.3 小结

本章我们为读者介绍了两款国内常见的嵌入式开发板，并描写了如何在这两款开发板上运行 MiniGUI。国内有许多提供此类开发板的厂商，产品形态也非常丰富。除了本章讲述的两款开发板之外，MiniGUI-STR 中还包含了对下面这些开发板（或者设备）的支持：

- iPAQ H3600/H3800
- Intel xScale PX255B
- 三星 SMDK2410
- 傅立叶 ARM7202
- 复旭 RM9200
- 英蓓特 ARM2410

有条件、有需求的读者可以联系这些厂商购买开发板。

附录 A MiniGUI-STR 和 MiniGUI-VAR 的功能差异

MiniGUI-STR 是 MiniGUI-VAR（MiniGUI 增值版）的简化版本。MiniGUI-STR 仅仅提供对 Linux 和 uClinux 操作系统的支持，但包含了针对许多硬件开发板的输入引擎支持，比如 iPAQ H3600/H3800、SMDK2410、Intel xScale PX255B、傅立叶 ARM7202、博创 ARM3000、复旭 RM9200、英蓓特 ARM2410 以及 华恒 HHARM2410-R3 等。

北京飞漫软件技术有限公司发行 MiniGUI-STR 的主要目的是为那些需要评估 MiniGUI 和/或学习嵌入式 Linux/uClinux 开发技能的人提供一个好的、易于上手的产品。如果您打算在商用产品中使用 MiniGUI，则应该选择 MiniGUI 增值版产品或 MiniGUI-AOR 产品，以便从飞漫软件获得有担保的技术支持服务。

A.1 MiniGUI-STR 中不包括的功能特性

和 MiniGUI 增值版产品相比，MiniGUI-STR 中缺少以下特性：

- 对 eCos、uC/OS-II、VxWorks 以及 ThreadX 等操作系统的支持。
- 内嵌资源功能的支持。
- 对 QPF、TrueType 等字体的支持。
- 对 GBK、GB18030、EUCKR、EUCJP 及 UNICODE 字符集的支持。
- 对 PCX、TGA、PBM 等图像格式的支持。

- NEWGAL 图形抽象接口及其图形引擎的支持。
- 基于 NEWGAL 的高级 GDI 接口及高级 2 维图形接口的支持。
- MiniGUI 虚拟控制台支持。
- MiniGUI 扩展库，具体包括：
  - 扩展控件，如 ListView、GridView、MonthCalendar、SpinBox、Animation 以及 IconView 等。
  - 皮肤界面的支持。
  - MyWindows 接口，包括文件打开对话框、颜色选取对话框等。

A.2 MiniGUI-STR 中已包含的功能特性

已包含在 MiniGUI-STR 中的功能特性有：

- 对 Linux/uClinux 操作系统的支持。
- 对三种运行模式的支持：MiniGUI-Threads、MiniGUI-Lite 和 MiniGUI-Standalone。
- 完整的窗口/消息 API 及基本的绘图 API。
- 用于鼠标、触摸屏校正的函数接口。
- 两种 GAL 图形引擎：FrameBuffer 和 qvfb。
- 针对多个开发板或设备的输入引擎：
  - iPAQ H3600/H3800
  - Intel xScale PX255B
  - 三星 SMDK2410
  - 傅立叶 ARM7202
  - 博创 ARM3000
  - 复旭 RM9200
  - 英蓓特 ARM2410
  - 华恒 HHARM2410-R3
- 对下述字符集的支持：
  - ISO8859-1 ~ ISO8859-16
  - GB2312
  - BIG5
- RBF 及 VBF 字体的支持，并在资源包中包含下述设备字体：
  - rbf-fixed-rrncnn-8-16-ISO8859-1

- rbf-fixed-rrncnn-16-16-GB2312.1980-0
- rbf-fixed-rrncnn-16-16-BIG5
- vbf-Courier-rrncnn-10-15-ISO8859-1
- vbf-Helvetica-rrncnn-15-16-ISO8859-1
- vbf-Times-rrncnn-13-15-ISO8859-1

■ 对下述图像文件格式的支持：
- Windows BMP
- GIF
- JPEG 和 PNG（分别通过 libjpeg 和 libpng）

■ 对下述窗口风格的支持：
- PC 3D
- Flat
- Phone

■ 对键盘布局的支持。

■ 对 GB2312 全拼输入法的支持。

■ 基本控件，具体包括：
- Static
- Button
- Simple edit box
- Single-line edit box
- Multiple-line edit box
- ListBox
- ComboBox
- ProgressBar
- NewToolbar
- MenuButton
- TrackBar
- PropertySheet
- ScrollView

附录 B
MiniGUI-STR 配置选项详解

从整体上讲，MiniGUI 的配置选项分两个阶段。第一阶段是在编译前确定的，第二阶段是在运行时确定的。本附录将详细解释这两个阶段的配置选项及其含义。

B.1 MiniGUI-STR 的编译配置选项

假定用户使用的是 RedHat Linux 9 系统，下面是在 MiniGUI 的源代码目录下运行 ./configure –-help 输出的结果（注意该命令在 Red Hat 低版本发行版上的输出可能有所不同）：

```
`configure' configures this package to adapt to many kinds of systems.

Usage: ./configure [OPTION]... [VAR=VALUE]...

To assign environment variables (e.g., CC, CFLAGS...), specify them as
VAR=VALUE.  See below for descriptions of some of the useful variables.

Defaults for the options are specified in brackets.

Configuration:
  -h, --help              display this help and exit
      --help=short        display options specific to this package
      --help=recursive    display the short help of all the included packages
  -V, --version           display version information and exit
  -q, --quiet, --silent   do not print `checking...' messages
      --cache-file=FILE   cache test results in FILE [disabled]
  -C, --config-cache      alias for `--cache-file=config.cache'
  -n, --no-create         do not create output files
      --srcdir=DIR        find the sources in DIR [configure dir or `..']

Installation directories:
  --prefix=PREFIX         install architecture-independent files in PREFIX
                          [/usr/local]
  --exec-prefix=EPREFIX   install architecture-dependent files in EPREFIX
                          [PREFIX]

By default, `make install' will install all the files in
`/usr/local/bin', `/usr/local/lib' etc.  You can specify
```

```
an installation prefix other than `/usr/local' using `--prefix',
for instance `--prefix=$HOME'.

For better control, use the options below.

Fine tuning of the installation directories:
  --bindir=DIR           user executables [EPREFIX/bin]
  --sbindir=DIR          system admin executables [EPREFIX/sbin]
  --libexecdir=DIR       program executables [EPREFIX/libexec]
  --datadir=DIR          read-only architecture-independent data [PREFIX/share]
  --sysconfdir=DIR       read-only single-machine data [PREFIX/etc]
  --sharedstatedir=DIR   modifiable architecture-independent data [PREFIX/com]
  --localstatedir=DIR    modifiable single-machine data [PREFIX/var]
  --libdir=DIR           object code libraries [EPREFIX/lib]
  --includedir=DIR       C header files [PREFIX/include]
  --oldincludedir=DIR    C header files for non-gcc [/usr/include]
  --infodir=DIR          info documentation [PREFIX/info]
  --mandir=DIR           man documentation [PREFIX/man]

Program names:
  --program-prefix=PREFIX            prepend PREFIX to installed program names
  --program-suffix=SUFFIX            append SUFFIX to installed program names
  --program-transform-name=PROGRAM   run sed PROGRAM on installed program names

System types:
  --build=BUILD     configure for building on BUILD [guessed]
  --host=HOST       cross-compile to build programs to run on HOST [BUILD]
  --target=TARGET   configure for building compilers for TARGET [HOST]

Optional Features:
  --disable-FEATURE       do not include FEATURE (same as --enable-FEATURE=no)
  --enable-FEATURE[=ARG]  include FEATURE [ARG=yes]
  --enable-shared=PKGS  build shared libraries default=yes
  --enable-static=PKGS  build static libraries default=yes
  --enable-fast-install=PKGS  optimize for fast installation default=yes
  --disable-libtool-lock  avoid locking (might break parallel builds)
  --disable-dependency-tracking Speeds up one-time builds
  --enable-dependency-tracking  Do not reject slow dependency extractors
  --enable-lite            build MiniGUI-Lite version <default=no>
  --enable-standalone      build MiniGUI-Lite Stand-Alone version <default=no>
  --enable-miniguientry    use minigui_entry function in MiniGUI <default=no>
  --enable-fixedmath       include fixed math routines <default=yes>
  --enable-debug           build with debugging messages <default=no>
  --enable-tracemsg        trace messages of MiniGUI <default=no>
  --enable-msgstr          include symbol name of message <default=no>
  --enable-micemoveable    user can move window by using mouse <default=yes>
  --enable-dblclk          mouse button can do double click <default=yes>
  --enable-cursor          include cursor support (for MiniGUI-Lite) <default=yes>
  --enable-clipboard       include clipboard support <default=yes>
  --enable-galfbcon        support native FrameBuffer GAL engine on Linux FrameBuffer \
console <default=yes>
  --enable-galqvfb         support native FrameBuffer GAL engine on Qt Virtual \
FrameBuffer <default=yes>
  --enable-coortrans_cw    support clockwise rotation of screen in the native \
FB GAL engine <default=no>
  --enable-coortrans_ccw   support counterclockwise rotation of screen in the \
native FB GAL engine <default=no>
  --enable-fblin1r         build the 1BPP FB subdriver of native graphics engine \
(MSB is right) <default=no>
  --enable-fblin1l          build the 1BPP FB subdriver of native graphics engine \
(MSB is left) <default=no>
  --enable-fblin2r         build the 2BPP FB subdriver of native graphics engine \
(MSB is right) <default=no>
  --enable-fblin2l         build the 2BPP FB subdriver of native graphics engine \
(MSB is left) <default=no>
  --enable-fblin4r         build the 4BPP FB subdriver of native graphics engine \
(MSB is right) <default=no>
  --enable-fblin4l         build the 4BPP FB subdriver of native graphics engine \
(MSB is left) <default=no>
  --enable-fblin8          build the 8BPP FB subdriver of native graphics engine \
<default=yes>
  --enable-fblin16         build the 16BPP FB subdriver of native graphics engine \
<default=yes>
  --enable-fblin24         build the 24BPP FB subdriver of native graphics engine \
(incompleted) <default=no>
  --enable-fblin32         build the 32BPP FB subdriver of native graphics engine \
<default=no>
  --enable-ipaqial         build the input engine for iPAQ H3600 <default=no>
  --enable-l7200ial        build the input engine for L7200 <default=no>
  --enable-arm3000ial      build the input engine for ARM3000 <default=no>
  --enable-fxrm9200ial     build the IAL engine for FXRM9200 <default=no>
```

```
  --enable-embest2410ial   build the IAL engine for EMBEST ARM2410 <default=no>
  --enable-fft7202ial      build the input engine for FFT7202 <default=no>
  --enable-px255bial       build the input engine for px255b <default=no>
  --enable-mc68x328ial     build the input engine for uClinux touch screen palm/mc68ez328\
 <default=no>
  --enable-skyeyeep7312ial build the input engine for touch screen of SkyEye EP7312\
 simulation <default=no>
  --enable-smdk2410ial     build the input engine for SMDK2410 touch screen <default=no>
  --enable-hh2410r3ial     build the input engine for HHARM2410R3 touch screen <default=no>
  --enable-dummyial        build the Dummy IAL engine <default=yes>
  --enable-autoial         build the Automatic IAL engine <default=no>
  --enable-qvfbial         build the QVFB IAL engine <default=yes>
  --enable-nativeial       build the native (console) input engine <default=yes>
  --enable-nativeps2       build the native engine subdriver for PS2 mouse <default=yes>
  --enable-nativeimps2     build the native engine subdriver for \
IntelligentMouse (IMPS/2) mouse <default=yes>
  --enable-nativems        build the native engine subdirver for \
old MS serial mouse <default=yes>
  --enable-nativems3       build the native engine subdirver for \
MS3 mouse <default=yes>
  --enable-nativegpm       build the native engine subdirver for \
GPM daemon <default=yes>
  --enable-textmode        Linux system have console (text mode) \
on FrameBuffer <default=yes>
  --enable-vbfsupport      include var bitmap font support <default=yes>
  --enable-latin2support   include East European (Latin 2, ISO-8859-2) \
charset support <default=no>
  --enable-latin3support   include South European (Latin 3, ISO-8859-3) \
charset support <default=no>
  --enable-latin4support   include North European (Latin 4, ISO-8859-4) \
charset support <default=no>
  --enable-cyrillicsupport include Cyrillic (ISO-8859-5) charset support <default=no>
  --enable-arabicsupport   include Arabic (ISO-8859-6) charset support <default=no>
  --enable-greeksupport    include Greek (ISO-8859-7) charset support <default=no>
  --enable-hebrewsupport   include Hebrew (ISO-8859-8) charset support <default=no>
  --enable-latin5support   include Turkish (Latin 5, ISO-8859-9) charset \
support <default=no>
  --enable-latin6support   include Nordic, Latin 6, ISO-8859-10) charset \
support <default=no>
  --enable-thaisupport     include Thai (ISO-8859-11) charset support <default=no>
  --enable-latin7support   include Latin 7 (ISO-8859-13) charset support <default=no>
  --enable-latin8support   include Latin 8 (ISO-8859-14) charset support <default=no>
  --enable-latin9support   include Latin 9 (ISO-8859-15, West Extended) \
charset support <default=yes>
  --enable-latin10support  include Latin 10 (ISO-8859-16, Romanian) \
charset support <default=no>
  --enable-gbsupport       include EUC encoding of GB2312 charset support <default=yes>
  --enable-big5support     include BIG5 charset support <default=yes>
  --enable-kbdfrpc         include keyboard layout for French \
PC keyboard (non-US 102 keys) <default=no>
  --enable-kbdfr           include keyboard layout for French <default=no>
  --enable-kbdde           include keyboard layout for German <default=no>
  --enable-kbddelatin1     include keyboard layout for German Latin1 <default=no>
  --enable-kbdit           include keyboard layout for Italian <default=no>
  --enable-kbdes           include keyboard layout for Spanish <default=no>
  --enable-kbdescp850      include keyboard layout for Spanish CP850 <default=no>
  --enable-savebitmap      include SaveBitmap-related functions <default=yes>
  --enable-gifsupport      include GIF file support <default=yes>
  --enable-jpgsupport      include JPG file support <default=yes>
  --enable-pngsupport      include PNG file support <default=yes>
  --enable-imegb2312       include IME (GB2312) support <default=yes>
  --enable-mousecalibrate  include code doing mouse calibration <default=yes>
  --enable-aboutdlg        include About Dialog Box <default=yes>
  --enable-savescreen      include code for screenshots <default=yes>
  --enable-grayscreen      target is a gray screen <default=no>
  --enable-tinyscreen      target is a tiny-size screen <default=no>
  --enable-ctrlstatic      include STATIC control <default=yes>
  --enable-ctrlbutton      include BUTTON control <default=yes>
  --enable-ctrlsimedit     include Simple EDIT control <default=yes>
  --enable-ctrlsledit      include Single-Line EDIT control <default=yes>
  --enable-ctrllistbox     include LISTBOX control <default=yes>
  --enable-ctrlpgbar       include PROGRESSBAR control <default=yes>
  --enable-ctrltoolbar     include TOOLBAR control <default=yes>
  --enable-ctrlnewtoolbar  include NEWTOOLBAR control <default=yes>
  --enable-ctrlmenubtn     include MENUBUTTON control <default=yes>
  --enable-ctrltrackbar    include TRACKBAR control <default=yes>
  --enable-ctrlcombobox    include COMBOBOX control <default=yes>
  --enable-ctrlpropsheet   include PROPSHEET control <default=yes>
  --enable-ctrlscrollview  include SCROLLVIEW and SCROLLWINDOW controls <default=yes>
  --enable-ctrltextedit    include TEXTEDIT control which is based-on \
SCROLLVIEW control <default=yes>
```

```
Optional Packages:
  --with-PACKAGE[=ARG]    use PACKAGE [ARG=yes]
  --without-PACKAGE       do not use PACKAGE (same as --with-PACKAGE=no)
  --with-gnu-ld           assume the C compiler uses GNU ld default=no
  --with-pic              try to use only PIC/non-PIC objects default=use both
  --with-osname=linux/uclinux
  --with-targetname=unknown
  --with-style=pc3d/flat/phone

Some influential environment variables:
  CC          C compiler command
  CFLAGS      C compiler flags
  LDFLAGS     linker flags, e.g. -L<lib dir> if you have libraries in a
              nonstandard directory <lib dir>
  CPPFLAGS    C/C++ preprocessor flags, e.g. -I<include dir> if you have
              headers in a nonstandard directory <include dir>
  CPP         C preprocessor

Use these variables to override the choices made by `configure' or to help
it to find libraries and programs with nonstandard names/locations.
```

上面这些参数是我们在 configure 脚本中设置好的命令行参数，可以控制编译后的 MiniGUI 中包含支持哪些功能的代码。比如，运行：

```
$ ./configure --with-style=phone  --enable-lite --enable-standalone
```

就可以将 MiniGUI 配置成具有 phone 风格的外观，且具有 MiniGUI-Standalone 运行模式。相反，如果运行：

```
$ ./configure --with-style=flat  --enable-lite
```

将 MiniGUI 配置成 MiniGUI-Lite 模式，使用 flat 风格。不带任何参数执行 ./configure 命令将按照默认选项生成 Makefile。注意在每个选项的说明中都给出了默认设置：default=yes 或者 default=no。

B.1.1 通用配置选项

下面这些选项是 GNU 风格软件包使用的一些重要的通用选项：

- --prefix=PREFIX

该选项用于指定 MiniGUI 函数库的安装路径。默认的安装路径是 /usr/local。如果运行：

```
./configure --prefix=/home/test
```

那么在执行 make install 之后，函数库、头文件以及参考手册页将被安

装在 /home/test/lib、/home/test/include、/home/test/man 目录下。

该选项和 --build、--host 以及 --target 等选项对 MiniGUI 和应用程序的交叉编译非常重要，相关信息，请读者参阅本书第 4 章。

■ --enable-static 和 --enable-shared

这两个选项指定生成函数库的静态库和动态库版本。如果不需要生成静态库，则可以使用 --disable-static 选项，这样，将缩短编译函数库的时间。

B.1.2 MiniGUI-STR 配置选项

接下来一些具体的配置 MiniGUI 的选项基本上都是基于 --disable-FEATURE 和 --enable-FEATURE 实现的。--disable-FEATURE 选项禁止某项特性，也就是在函数库中将不对该特性进行支持。--enable-FEATURE 选项打开某项特性，也就是在函数库中将对该特性进行支持。

另外，MiniGUI 配置脚本还提供了 --with 选项，该选项可用来从多个可选项中指定其中某一个选项。MiniGUI 所支持的操作系统以及控件风格通过 with 选项指定：

■ --with-osname=linux/uclinux

用于指定 MiniGUI 所运行的操作系统，默认为 linux。如果要让 MiniGUI 在 uClinux 上运行，应使用 --with-osname=uclinux 选项。

■ --with-style=pc3d/flat/phone

用于指定 MiniGUI 控件的外观风格，默认为 pc3d，即类似 PC 的三维风格。其它两个选项分别是：flat：适合灰度屏的平板风格；phone：适合手持终端的华丽风格。

其它的 MiniGUI 配置选项，均是开关选项，见表 B-1。

表 B-1 MiniGUI-STR 的开关配置选项

选项名称	含义	默认值	备注
lite	是否生成 MiniGUI-Lite 模式	no	

standalone	是否生成 MiniGUI-Standalone 模式	no	仅仅在 lite 为 yes 时有效
miniguientry	是否使用 minigui_entry 函数	no	
fixedmath	是否包括定点数运算函数	yes	
debug	是否包含调试信息	no	
tracemsg	是否跟踪 MiniGUI 的消息传递	no	将生成大量的消息发送输出信息
msgstr	是否包含 MiniGUI 消息的字符串名称	no	和 tracemsg 配合，用来打印消息名称
micemoveable	关闭该选项将禁止用户使用鼠标来移动窗口。在很多的嵌入式系统上，不使用层叠式的多窗口用户界面，也不需要移动窗口，这时，就可以使用 --disable-micemoveable 配置选项。	yes	
dblclk	是否支持鼠标的双击操作	yes	
cursor	是否显示鼠标光标	yes	在某些用触摸笔操作的设备上，不需要显示鼠标光标来标识鼠标的位置
clipboard	是否支持剪切板	yes	
galfbcon	是否包含针对 Linux FrameBuffer 控制台的图形引擎代码	yes	
galqvfb	是否包含针对 Qvfb 的图形引擎代码	yes	
coortrans_cw	打开该选项将激活 GAL 接口中的坐标转换功能，这个功能将使屏幕的显示以顺时针旋转 90 度，此功能通常在 COMPAQ iPAQ H3600 系列的产品中使用。	no	
coortrans_ccw	打开该选项将激活 GAL 接口中的坐标转换功能，这个功能将使屏幕的显示以逆时针旋转 90 度，此功能通常在 COMPAQ iPAQ H3800 系列的产品中使用。	no	
fblinXX	fblin1、fblin2、fblin_2、fblin4、fblin8、fblin16、fblin24、fblin32 等选项用来指定 GAL 接口的图形引擎中包含对哪些颜色深度的支持。	默认包含 8 位和 16 位	
ipaqial	是否包含针对 iPAQ H3600/H3800 的输入引擎	no	
arm3000ial	是否包含针对博创 ARM3000 开发板的输入引擎	no	
fxrm9200ial	是否包含针对复旭 9200 开发板的输入引擎	no	
embest2410ial	是否包含针对英蓓特 2410 开发板的输入引擎	no	
fft7202ial	是否包含针对傅立叶 7202 开发板的输入引擎	no	
px255bial	是否包含针对 Intel xScale PX255B 开发板的输入引擎	no	
mc68x328ial	是否包含针对 Xcopilot 模拟器的输入引擎	no	
skyeyeep7312ial	是否包含针对 SkyEye EP7312 模拟器的输入引擎	no	
smdk2410ial	是否包含针对三星 2410 开发板的输入引擎	no	
hh2410r3ial	是否包含针对华恒 2410R3 开发板的输入引擎	no	
dummyial	是否包含“哑”输入引擎	yes	
autoial	是否包含“自动”输入引擎	no	
qvfbial	是否包含针对 qvfb 的输入引擎	yes	
nativeial	是否包含针对 Linux 标准控制台的输入引擎	yes	
nativeps2 等	该选项以及 nativeimps2、nativems、nativems3、nativegpm 等选项，用来指定是否在 MiniGUI 库中包含对特定鼠标协议的支持，分别是 PS2、IMPS2、MS（老式串口鼠标）、MS3（带中键的串口鼠标）以及 GPM。	yes	
textmode	关闭该选项可用于无字符模式的 Linux 系统（即没有字符控制台的 Linux 系统，对某些嵌入式设备有用）。	yes	
vbfsupport	关闭该选项可禁止 MiniGUI 对 Var Bitmap Font（VBF）的支持，同时将忽略 MiniGUI.cfg 文件中的 [varbitmapfonts] 段。	yes	

latin2support 等	atin2support 、 latin3support 、 cyrillicsupport 、 arabicsupport 、 greeksupport 、 hebrewsupport 、 latin5support 、 latin6support 、 thaisupport 、 latin7support 、 latin8support 、 latin9support 、 latin10support 等选项，用来控制对 ISO8859-2 到 ISO8859-16 字符集的支持。这些字符集均是单字节字符集。	yes	
gbsupport	用来控制对 GB2312 多字节字符集/编码系统的支持。	yes	
big5support	用来控制对 BIG5 多字节字符集/编码系统的支持。	yes	
kbdfrpc 等	kbdfrpc、kbdfr、kbdde、kbddelatin1、kbdit、kbdes、kbdescp850 等选项用来指定 MiniGUI 包含哪些键盘布局，一般无需设置。默认不包含上述键盘布局。	no	
savebitmap	控制是否提供将 BITMAP 结构保存为 Windows BMP 文件格式的支持。	yes	
gifsupport	控制是否提供对 GIF 文件格式的支持。	yes	
jpgsupport	控制是否提供对 JPEG 文件格式的支持。	yes	需要 libjpeg 库的支持
pngsupport	控制是否提供对 PNG 文件格式的支持。	yes	需要 libpng 库的支持
imegb2312	用来控制是否包含 GB2312 中文简体输入法。	yes	
mousecalibrate	用来控制是否包含鼠标线性校正接口。	yes	
aboutdlg	用来控制是否包含 About MiniGUI 对话框。	yes	
savescreen	用来控制是否响应 <PrntScrn> 键并保存屏幕到当前目录。在将 MiniGUI 移植到嵌入式系统中时，最好采用该选项禁止屏幕的保存功能。包含该功能时，用户按 <PrntScrn> 键将在当前目录保存运行时刻的屏幕图，按 <Ctrl+PrntScrn> 键将保存当前活动窗口的屏幕图。	yes	
grayscreen	该选项用来控制某些屏幕元素的绘制方式。如果您的目标平台采用灰度屏幕，则建议打开该选项。	no	
tinyscreen	该选项用来控制某些屏幕元素（比如 MessageBox）的绘制方式。如果您的目标平台具有很小的屏幕，比如小于 320x240，则建议打开该选项。	no	
ctrlstatic 等	ctrlstatic 、 ctrlbutton 、 ctrlsimedit 、 ctrlsledit 、 ctrloldmledit、ctrllistbox、ctrlpgbar、ctrltoolbar、 ctrlnewtoolbar 、 ctrlmenubtn 、 ctrltrackbar 、 ctrlcombobox 、 ctrlpropsheet 、 ctrlscrollview 、 ctrltextedit 等选项分别用来控制是否在库中包含静态框、按钮、简单编辑框（只能处理等宽字体和 GB2312 字符集）、单行编辑框、老的多行编辑框、列表框、进度条、工具栏、新工具栏、菜单按钮、跟踪条、组合框、属性页、滚动视图、增强文本编辑框等控件。	yes	

B.1.3 支持中文显示的最小 MiniGUI-STR 函数库配置选项

在 MiniGUI-STR 1.6.2 版本中包含了一个 buildlib-min 脚本（在 build/ 目录下）。该脚本的内容如下：

```
#!/bin/sh

./configure \
    --with-style=flat \
    --enable-grayscreen \
    --enable-tinyscreen \
    --disable-galqvfb \
```

```
    --disable-qvfbial \
    --disable-dummyial \
    --disable-micemoveable \
    --disable-cursor \
    --disable-vbfsupport \
    --disable-latin9support \
    --disable-big5support \
    --disable-savebitmap \
    --disable-jpgsupport \
    --disable-pngsupport \
    --disable-imegb2312 \
    --disable-aboutdlg \
    --disable-savescreen
```

利用这个脚本可将 MiniGUI 配置成支持中文显示的最小函数库，从而使生成的 MiniGUI 函数库较小。由这个脚本配置的 MiniGUI 函数库包含或不包含如下功能：

- 将 MiniGUI 编译为 MiniGUI-Threads。
- 只包含 fbcon 图形引擎，支持 8 位色 和 16 位色显示模式。
- 只包含 native/console 输入引擎。
- 不支持 VBF 字体。
- 不包含对 Latin2 等字符集的支持，只包含对 Latin1，即 ISO8859-1 字符集的支持。
- 不包含对 BIG5 字符集的支持，只包含对 GB2312 字符集的支持（对不需要中文支持的系统，甚至可以利用 --disable-gb2312support 将 GB2312 字符集的支持取消）。
- 不包含 GB2312 输入法支持。
- 不包含 BITMAP 保存支持。
- 不包含对 JPEG、PNG 图片格式的支持。
- 不包含“ AboutMiniGUI ”对话框。
- 不包含保存屏幕的功能。

在上述配置基础上，用户还可以根据需求取消一些功能。

B.2 MiniGUI-STR 的运行时配置文件：MiniGUI.cfg

下面是 MiniGUI-STR V1.6.2 版本默认安装的 MiniGUI 函数库的运行时配置文件的内容：

```
# MiniGUI Ver 1.6.x.
# This configuration file is for PC3D window style.
#
# Copyright (C) 2002~2005 Feynman Software
```

```
# Copyright (C) 1998~2002 Wei Yongming.
#
# Web:   http://www.minigui.com
# Web:   http://www.minigui.org
#
# This configuration file must be installed in /etc,
# /usr/local/etc or your home directory. When you install it in your
# home directory, it should be named ".MiniGUI.cfg".
#
# The priority of above configruation files is ~/.MiniGUI.cfg,
# /usr/local/etc/MiniGUI.cfg, and then /etc/MiniGUI.cfg.
#
# If you change the install path of MiniGUI resource, you should
# modify this file to meet your configuration.
#
# NOTE:
# The format of this configuration file has changed since the last release.
# Please DONT forget to provide the latest MiniGUI.cfg file for your MiniGUI.
#

[system]
# GAL engine
gal_engine=fbcon

# IAL engine
ial_engine=console

mdev=/dev/mouse
mtype=IMPS2

[fbcon]
defaultmode=1024x768-16bpp

[qvfb]
defaultmode=640x480-16bpp
display=0

# The first system font must be a logical font using RBF device font.
[systemfont]
font_number=5
font0=rbf-fixed-rrncnn-8-16-ISO8859-1
font1=*-fixed-rrncnn-*-16-GB2312
font2=*-Courier-rrncnn-*-16-GB2312
font3=*-Times-rrncnn-*-16-GB2312
font4=*-Helvetica-rrncnn-*-16-GB2312

default=0
wchar_def=1
fixed=1
caption=2
menu=3
control=4

[rawbitmapfonts]
font_number=2
name0=rbf-fixed-rrncnn-8-16-ISO8859-1
fontfile0=/usr/local/lib/minigui/res/font/8x16-iso8859-1.bin
name1=rbf-fixed-rrncnn-16-16-GB2312.1980-0
fontfile1=/usr/local/lib/minigui/res/font/song-16-gb2312.bin

[varbitmapfonts]
font_number=3
name0=vbf-Courier-rrncnn-10-15-ISO8859-1
fontfile0=/usr/local/lib/minigui/res/font/Courier-rr-10-15.vbf
name1=vbf-Helvetica-rrncnn-15-16-ISO8859-1
fontfile1=/usr/local/lib/minigui/res/font/Helvetica-rr-15-16.vbf
name2=vbf-Times-rrncnn-13-15-ISO8859-1
fontfile2=/usr/local/lib/minigui/res/font/Times-rr-13-15.vbf

[mouse]
dblclicktime=300

[event]
timeoutusec=300000
repeatusec=50000

[cursorinfo]
# Edit following line to specify cursor files path
cursorpath=/usr/local/lib/minigui/res/cursor/
cursornumber=4
cursor0=d_arrow.cur
```

```
cursor1=d_beam.cur
cursor2=d_pencil.cur
cursor3=d_cross.cur

[iconinfo]
# Edit following line to specify icon files path
iconpath=/usr/local/lib/minigui/res/icon/
# Note that max number defined in source code is 5.
iconnumber=5
icon0=form.ico
icon1=w95mbx01.ico
icon2=w95mbx02.ico
icon3=w95mbx03.ico
icon4=w95mbx04.ico

[bitmapinfo]
# Edit following line to specify bitmap files path
bitmappath=/usr/local/lib/minigui/res/bmp/
# Note that max number defined in source code is 7
bitmapnumber=2
bitmap0=capbtns.bmp
# bitmap1=arrows.bmp
# use large bitmap if your default font is 16 pixel height.
bitmap1=arrows16.bmp
bitmap2=none
bitmap3=none
bitmap4=none
bitmap5=none

# background picture, use your favirate photo
bitmap6=none

# bitmap used by BUTTON control
button=button.bmp
pushbutton=none
pushedbutton=none

# bitmap used by LISTBOX control
checkmark=checkmark.bmp

# bitmap used by COMBOBOX control
downarrow=downarrow.bmp
updownarrow=updownarrow.bmp
leftrightarrow=leftrightarrow.bmp

# bitmap used by IME window
IMEctrlbtn=shurufa.bmp

# bitmap used by About dialog box
logo=MiniGUI256.bmp
# logo=MiniGUI16.bmp

[bgpicture]
position=center
# position=upleft
# position=downleft
# position=upright
# position=downright
# position=upcenter
# position=downcenter
# position=vcenterleft
# position=vcenterright
# position=none

[mainwinmetrics]
minwidth=50
minheight=50
border=2
thickframe=2
thinframe=1
captiony=+4
iconx=16
icony=16
menubary=+0
menubaroffx=8
menubaroffy=5
menuitemy=+0
intermenuitemx=12
intermenuitemy=2
menuitemoffx=18
menutopmargin=4
```

```
menubottommargin=4
menuleftmargin=4
menurightmargin=4
menuitemminx=64
menuseparatory=4
menuseparatorx=4
sb_height=14
sb_width=16
sb_interx=2
cxvscroll=16
cyvscroll=16
cxhscroll=16
cyhscroll=16
minbarlen=9
defbarlen=18

[windowelementcolors]
bkc_caption_normal=0x00808080
fgc_caption_normal=0x00C0C0C0
bkc_caption_actived=0x00800000
fgc_caption_actived=0x00FFFFFF
bkc_caption_disabled=0x00808080
fgc_caption_disabled=0x00C0C0C0
wec_frame_normal=0x00000000
wec_frame_actived=0x00FF0000
wec_frame_disabled=0x00000000
bkc_menubar_normal=0x00C0C0C0
fgc_menubar_normal=0x00000000
bkc_menubar_hilite=0x00800000
fgc_menubar_hilite=0x00FFFFFF
fgc_menubar_disabled=0x00808080
bkc_menuitem_normal=0x00C0C0C0
fgc_menuitem_normal=0x00000000
bkc_menuitem_hilite=0x00800000
fgc_menuitem_hilite=0x00FFFFFF
fgc_menuitem_disabled=0x00808080
bkc_pppmenutitle=0x00C0C0C0
fgc_pppmenutitle=0x00FF0000
fgc_menuitem_frame=0x00C66931
wec_3dbox_normal=0x00C0C0C0
wec_3dbox_reverse=0x00000000
wec_3dbox_light=0x00FFFFFF
wec_3dbox_dark=0x00808080
wec_flat_border=0x00808080
bkc_control_def=0x00C0C0C0
fgc_control_normal=0x00000000
fgc_control_disabled=0x00C0C0C0
bkc_hilight_normal=0x00FF0000
bkc_hilight_lostfocus=0x00BDA69C
fgc_hilight_normal=0x00FFFFFF
fgc_hilight_disabled=0x00C0C0C0
bkc_desktop=0x00FF0000
bkc_dialog=0x00C0C0C0
bkc_tip=0x00C8FCF8

[imeinfo]
imetabpath=/usr/local/lib/minigui/res/imetab/
imenumber=1
ime0=pinyin

[appinfo]
apprespath=/usr/local/lib/shared/miniguiapps/
```

在使用默认配置安装 MiniGUI 之后，将把 MiniGUI 源代码树中的 etc/MiniGUI-pc3d.cfg 文件安装到系统 /usr/local/etc/ 目录，并更名为 MiniGUI.cfg。在 MiniGUI 应用程序启动时，MiniGUI 优先查找用户主目录下的 .MiniGUI.cfg 文件，其次是 /usr/local/etc/MiniGUI.cfg，最后是 /etc/MiniGUI.cfg 文件。如果用户没有在自己的主目录下建立 .MiniGUI.cfg 文件，则通常情况下，/usr/local/etc/MiniGUI.cfg 文件就是 MiniGUI 所使用的默认运行时配置文件。

该配置文件采用了非常简洁的格式，所以修改起来也很容易。其格式如下：

```
[section-name1]
key-name1=key-value1
key-name2=key-value2

[section-name2]
key-name3=key-value3
key-name4=key-value4
```

该配置文件中的参数以段（section）分组，然后用 key=value 的形式指定参数及其值。下面介绍一些重要的段。

B.2.1 system 段

system 段中指定了 MiniGUI 要使用的图形引擎、输入引擎以及鼠标设备和协议类型，分别由 gal_engine、ial_engine、mdev 和 mtype 键指定。因为 MiniGUI 库中可以同时包含多个图形引擎和多个输入引擎，可以分别通过 gal_engine、ial_engine 指定要使用哪个图形引擎。mdev 和 mtype 分别用来指定鼠标设备文件以及鼠标协议类型。

B.2.2 fbcon 和 qvfb 段

fbcon 段的 defaultmode 关键字定义使用 FBCON 图形引擎时默认的显示模式。

qvfb 段的 defaultmode 关键字定义使用 qvfb 引擎时默认的显示模式；display 关键字指定运行 qvfb 时使用了 X Window 的哪个 Display，一般取 0。

B.2.3 systemfont 段

systemfont 段定义了 MiniGUI 的系统字体，默认情况下不应作改动。font_number 指定了要创建的系统字体个数，然后用 name<NR> 表示编号为 <NR> 的系统字体名称。字体名的格式如下：

```
<type>-<facename>-<style>-<width>-<height>-<charset1>
```

需要说明的是，系统字体是 MiniGUI 装载了由 rawbitmapfonts、

varbitmapfonts、qpf、truetypefonts 、t1fonts 等段定义的设备字体之后，根据上述字体名称调用 CreateLogFontFromName 函数建立的逻辑字体。这些字体将用于 MiniGUI 的标题、菜单、控件的显示，同时还用来指定窗口的默认字体。

逻辑字体名称各部分的含义如下：

- type 是所选用的设备字体类型，如果不想指定，则取 *。
- facename 指字体样式名，比如 Courier、Times 等等。
- style 是由六个字母组成的字符串，用来指定逻辑字体风格，比如是否斜体、是否加粗、是否含有下划线或者删除线等等。
- width 指定要创建的逻辑字体的宽度。一般不用指定，取 *。
- height 指定要创建的逻辑字体的高度。
- charset 指定要创建的逻辑字体的字符集。

此外，在 systemfont 段中，还有下面的键需要定义，这些键的值就是上述逻辑字体的编号：

```
default=0
wchar_def=1
fixed=1
caption=2
menu=3
control=4
```

举例来说，上面的键定义了系统（单字节字符集的）默认字体为编号为 0 的系统字体；多字节字符集的默认字体是 1 号系统字体；等宽字体为 1 号系统字体；窗口标题使用 2 号系统字体；菜单使用 3 号系统字体；控件使用 4 号系统字体。

您可以修改要创建的系统字体个数，但至少要创建一种单字节字符集（比如 ISO8859-1）的字体。还可以修改系统字体的大小，比如，下面的 systemfont 就指定 MiniGUI 使用 16 点阵的系统字体，并只创建了用于显示 ISO8859-1 字符集和 GB2312-80 字符集的逻辑字体，然后用这两个逻辑字体显示标题、菜单和控件中的文字：

```
[systemfont]
font_number=2
name0=rbf-Fixed-rrncnn-8-16-ISO8859-1
name1=rbf-Fixed-rrncnn-16-16-GB2312.1980-0

default=0
wchar_def=1
fixed=1
caption=1
```

```
menu=1
control=1
```

需要进一步说明的是：

- MiniGUI 根据 default、wchar_def 系统字体来定义系统的默认字符集，这影响 GetSysCharset、GetSysCharWidth、GetSysCCharWidth 和 GetSysHeight 函数的返回值。一般来讲，default 和 wchar_default 必须是等宽点阵字体，即 RBF 字体，并且多字节字符集字体的宽度必须是单字节字符集宽带的两倍。
- MiniGUI 的许多窗口元素尺寸是根据系统字体的大小定义的，详情请参阅下面对 mainwinmetric 段的说明。

B.2.4 rawbitmapfonts 和 varbitmapfonts 段

这些段用来指定要装载的设备字体信息。段中的 font_number 键定义了要装载的设备字体个数；name<NR> 和 fontfile<NR> 键分别定义了编号为 <NR> 的设备字体名称及对应的字体文件。如果您不想装载某个类型的设备字体，则可以通过设置 font_number 为 0 值来跳过对这种设备字体的装载。MiniGUI 使用的设备字体名称格式如下：

```
<type>-<facename>-<style>-<width>-<height>-<charset1[,charset2,...]>
```

设备字体名称各部分的含义如下：

- type 是设备字体类型，对上述四种设备字体，分别取 rbf、vbf、qpf、ttf 和 t1f。
- facename 指设备字体的样式名，比如 Courier、Times 等等。
- style 是由六个字母组成的字符串，用来指定字体风格，比如是否斜体、是否加粗、是否含有下划线或者删除线等等。
- width 表示字体的宽度。对变宽字体来讲，用来指定最大宽度。对可缩放的矢量字体来讲，设置为 0。
- height 表示字体的高度。对可缩放的矢量字体来讲，设置为 0。
- charset1、charset2 等，表示该设备字体所支持的字符集。

A.2.5 mouse 和 event 段

这两个段用于系统的内部事件处理，一般无须作任何改动。

B.2.6 cursorinfo、iconinfo 和 bitmapinfo 段

指定 MiniGUI 要装载的鼠标光标、图标以及系统位图信息。这三个段的定义方法和 rawbitmapfonts 等段类似。首先分别用 cursorpath、iconpath 和 bitmappath 指定了存放鼠标光标、图标和位图的路径，然后用 cursornumber、iconnumber 和 bitmapnumber 分别指定了要装载的鼠标光标、图标和位图的个数，最后用 cursor<NR>、icon<NR>、bitmap<NR> 指定了编号为 <NR> 的鼠标光标、图标和位图文件。

需要注意的是，您可以修改 cursornumber、iconnumber、bitmapnumber 来减少 MiniGUI 装载的鼠标光标、图标以及位图文件，并同时删除对应的哪些资源文件，这样，可以减少 MiniGUI 的存储空间占用量。当然，在减少这些资源之前，您要确保自己的应用程序真的不需要这些资源。

如果您在配置 MiniGUI 时使用了 --disable-cursor 选项，则 MiniGUI 会忽略 cursorinfo 段。

bitmap6 用来指定 MiniGUI 的背景图片。如果您不需要背景图片，则可以修改 bitmap6 的值，将其指定为 none。bitmapinfo 中还有若干特殊的键，比如 IMEctrlbtn 和 logo；前者由输入法模块使用，后者由 About 对话框使用。如果您在配置 MiniGUI 时使用了 --disable-imegb2312 或者 --disable-aboutdlg 选项，则可以删除这两个键定义的位图。

B.2.7 bgpicture 段

这个段定义了背景图片在背景上显示时的位置，可取 center、upleft、downleft、upright、downright、upcenter、downcenter、vcenterleft、vcenterright、none 这几个值，分别表示正中、左上、左下、右上、右下、中上、中下、左中、右中、不显示等。

B.2.8 mainwinmetrics 和 windowelementcolors 段

mainwinmetrics 定义了默认的窗口元素的尺寸，一般不需要进行修改。我们可以使用系统字体的高度来设定窗口元素的尺寸。比如在指定标题栏的高度时，用 captiony=+4 这样的形式指定标题栏高度是系统字体高度加 4 个象素。windowelementcolors 定义了默认的窗口元素所使用的颜色，一般也不需要进行修改。

B.2.9 imeinfo 段

这个段指定了 GB2312 输入法要装载的输入法个数及对应的模块。键的名称规则类似 bitmapinfo 段。pinyin、wubi、shuangpin、ziranma 等分别表示全拼、五笔、双拼、自然码等输入法模块。如果在配置 MiniGU 时打开了 GB2312 输入法（--enable-imegb2312），则会根据 imenumber 的值装载指定的输入法模块；如果 imenumber 的值为 0，则输入法模块仅仅提供内码输入法。

B.2.10 只支持英文显示的最小配置文件

MiniGUI 中包含一个仅支持英文显示的最小配置文件（etc/MiniGUI-min.cfg），使用该配置文件，将使运行时的 MiniGUI 占用最少的内存。该配置文件主要修改了字体和鼠标光标配置段。该文件的内容如下：

```
# MiniGUI Ver 1.6.x
# This configuration file is for minimal configuration.
#
# Copyright (C) 2002~2004 Feynman Software.
# Copyright (C) 1998~2002 Wei Yongming.
#
# Web:   http://www.minigui.com
# Web:   http://www.minigui.org
#
# This configuration file must be installed in /etc,
# /usr/local/etc or your home directory. When you install it in your
# home directory, the name should be ".MiniGUI.cfg".
#
# The priority of above configruation files is ~/.MiniGUI.cfg,
# /usr/local/etc/MiniGUI.cfg, and then /etc/MiniGUI.cfg.
#
# If you change the install path of MiniGUI resource, you should
# modify this file to meet your configuration.
#
# NOTE:
# The format of this configuration file has changed since last release.
# Please DONT forget to provide the latest MiniGUI.cfg file for your MiniGUI.
#

[system]
# GAL engine
gal_engine=fbcon

# IAL engine
ial_engine=console

mdev=/dev/mouse
mtype=IMPS2

[fbcon]
defaultmode=1024x768-16bpp

[qvfb]
defaultmode=640x480-16bpp
display=0

# The system fonts must be raw bitmap fonts
[systemfont]
```

```
font_number=1
font0=rbf-fixed-rrncnn-8-16-ISO8859-1

default=0
wchar_def=0
fixed=0
caption=0
menu=0
control=0

[rawbitmapfonts]
font_number=1
name0=rbf-fixed-rrncnn-8-16-ISO8859-1
fontfile0=/usr/local/lib/minigui/res/font/8x16-iso8859-1.bin

[varbitmapfonts]
font_number=0

 [mouse]
dblclicktime=300

[event]
timeoutusec=300000
repeatusec=50000

[cursorinfo]
# Edit following line to specify cursor files path
cursorpath=/usr/local/lib/minigui/res/cursor/
cursornumber=2
cursor0=d_arrow.cur
cursor1=d_beam.cur
```

如果要使用该配置文件，可将其复制到 /usr/local/etc/ 目录下，并覆盖该目录下的 MiniGUI.cfg 文件；或者，您也可以将这个文件复制到自己的主目录下 ~/.MiniGUI.cfg，这样，就会优先使用该配置文件。

需要注意的是，上面给出的这个配置文件省略了部分未作修改的段，比如 [mainwinmetrics] 以及 [windowelementcolors] 等。

附录 C
MiniGUI 授权策略

MiniGUI、MiniGUI 综合演示程序 MDE 、MiniGUI 示例程序 mg-samples 和 MiniGUI 综合示例包 mgdemo 均遵循 GUN 通用公共许可证（简称 GPL）发布。飞漫软件针对 MiniGUI 的授权策略概括而言就是：MiniGUI 是 100% 按照 GPL 条款发布的自由软件，如果用户能 100% 遵守 GPL 许可证条款，则无需支付任何授权费用。在其他任何情况下，都需要获得飞漫软件技术有限公司的商业授权。

老版本 MiniGUI（1.3.0 版本之前）遵循 LGPL 条款发布。因为嵌入式系统无法保障用户获得修改 MiniGUI 函数库并进行调试的自由，从而无法确保 LGPL 在嵌入式产品中的实施，因此，如果要在嵌入式产品中使用老版本的 MiniGUI， 也必须首先购买商业授权。

为了确保对 GPL 条款 的正确理解，本附录第三小节给了从 GNU 网站获得的英文条款原文（因为任何经过翻译的文本均有可能无法精确表达该条款的原意）。用户，尤其是打算将自由软件用于商业产品中的用户，必须仔细阅读并理解这些自由软件条款。

C.1 授权详解

自 MiniGUI V1.3.0 起，飞漫软件技术有限公司遵循 GNU General Public License 许可证（简称 GPL）发布 MiniGUI。该条款的原文可见和 MiniGUI 源代码一同发布的 COPYING 文件。相比早期版本使用的 LGPL（GNU Lesser General Public License）许可证，GPL 所定义的条款要严

格的多，更能有效维护自由软件的权益，避免自由软件成为专有系统的一部分。

飞漫软件技术有限公司为无法或者不愿 100% 遵循 GPL 条款使用 MiniGUI 的用户提供商业授权。具体价格及购买方法，请访问飞漫软件网站：

http://www.minigui.com/product/cindex.shtml

1) 如果您 100% 遵循 GPL，无需获得商业授权

如果您使用 MiniGUI 的应用程序以 GPL 发布，则无需获得我们的商业授权。我们非常欢迎任何人在遵循 GPL 条款的基础上复制、修改和发布 MiniGUI。在这种情况下，您无需获得飞漫软件的任何形式的（包括口头或书面）使用授权，因为 GPL 条款本身就足够确保您的权益。但需要注意的是，飞漫软件不对这种形式下的使用提供任何形式的担保或技术支持。

2) 如果您从不复制、修改和发布 MiniGUI，无需获得商业授权

只要您从不复制、修改和发布 MiniGUI，则您可以在您的应用程序中使用 MiniGUI，而无需获得商业授权。举个例子，您在完成一篇学位论文，并因此在您的程序中使用了 MiniGUI，您的程序仅仅用来说明您论文中试验的可行性或者结果，而且您没有修改 MiniGUI，并且该程序不以任何方式被复制和发布，则您无需获得商业授权。飞漫软件也不对这种形式下的使用提供任何担保。

但需要特别指出：

- 修改——我们欢迎您对 MiniGUI 进行任意的修改。如果您发布该修改版本，则您对 MiniGUI 所做的任何修改、所有的接口代码以及直接和间接地与接口相关联的代码将遵循 GPL 许可证。
- 复制。我们允许您复制 MiniGUI 二进制代码和/或源代码，但一旦这么做了，所有的副本应遵循 GPL 许可证。

3) 其他情况均需获得商业授权

如果您使用 MiniGUI 的应用程序并不以 GPL 条款发布，却打算在内部或外部发布使用 MiniGUI 的应用程序或者函数库，则您必须首先获得飞漫软件的商业授权。

特别是：

- 如果您使用的嵌入式操作系统不采用 GPL 授权条款（比如 uC/OS-II、ThreadX、VxWorks 等操作系统），则必须购买 MiniGUI 商业授权。
- 如果您使用的是传统嵌入式操作系统（非 Linux/uClinux 的其他嵌入式操作系统），操作系统、函数库、MiniGUI 和您的应用程序需要编译链接成完整的可执行映像，则必须购买 MiniGUI 商业授权。
- 您的非 GPL 应用程序连接了 MiniGUI，不管静态还是动态连接，您需要为每一个 MiniGUI 函数库副本购买商业授权。
- 如果您在自己的单位使用 MiniGUI 函数库，但又不希望将其置于 GPL 许可证之下，则需要购买商业授权。
- 当然，更多的人购买 MiniGUI 的商业授权，其目的非常简单，他们希望获得来自飞漫软件的技术支持和软件质量担保。

4) 建议

对商业用户，我们建议购买 MiniGUI 的商业授权。这不仅仅能帮助您避免为满足 GPL 条款而付出太多的硬件和软件开发费用，从而保护自己专有系统的商业利益，也可以从飞漫软件获得质量担保——GPL 软件不含任何形式的、间接或直接的担保。

对自由软件社区的用户，或者经费不足的科研院校，我们建议您在开发基于 MiniGUI 的应用程序时，采用 GPL 或者其他的开放源码许可证条款。这样，能在最大程度上满足 GPL 许可证条款，您也不必购买 MiniGUI 的商业授权。

如果您不知道自己的产品能否 100% 满足 GPL 条款，则建议您选择商业授权，因为对飞漫软件来讲，我们会保护商业客户的利益，并通过优秀的技术支持和产品担保来确保您的产品能够顺利开发并良好运行。

5) MiniGUI 商业授权方式

飞漫软件提供以出货量为单位的 MiniGUI 商业授权办法，用户可以根据自己产品的出货量灵活选择：

- 用户需要为每个 MiniGUI 副本购买商业授权，每个授权的单价和使用 MiniGUI 的产品数量有关。

另外，商业授权费用与您使用的操作系统（Linux、uClinux、VxWorks

等）有关。具体的价格及购买方法，请访问飞漫软件网站：

http://www.minigui.com/product/cindex.shtml

C.2 老的版本

在版本 1.3.0 之前，MiniGUI 使用 LGPL 条款发布。这种条款为非自由软件使用自由函数库而提供了非常宽松的条件。但是，LGPL 条款仍然为复制、修改和发布 LGPL 软件定义了一些约束性的条款，以避免 LGPL 软件变成专有系统事实上的一部分，或者通过一些技术障碍来阻止用户获得 LGPL 条款定义的自由。这些条款包括以下几个方面：

- 对 MiniGUI 本身的复制、修改和发布行为，均应在确保 MiniGUI 仍然为函数库、仍然遵循 LGPL 或者 GPL 许可证的前提下进行。
- 使用 MiniGUI 必然要生成可执行文件。根据 LGPL 条款的定义，该可执行文件是 MiniGUI 的“衍生作品”，并且应该按照 LGPL 许可证之第 6 条发布该可执行文件。该条款的核心思想是，确保用户知悉在该“衍生作品”中使用了遵循 LGPL 条款发布的 MiniGUI 函数库，用户因此将得到修改 MiniGUI 的权利；在用户修改了 MiniGUI 函数库的情况下，只要修改后的版本和原先的版本在接口上是兼容的，则应确保用户仍能够生成“衍生作品”（静态连接的情况），或者“衍生作品”仍然能够正常 工作（动态连接的情况）。因此，“衍生作品”的发布必须以确保用户获得上述自由为前提和条件。

飞漫软件认为，MiniGUI 在大多数嵌入式系统中的应用，会因为某些技术上的障碍而不可避免地阻碍用户获得上述自由：

- 某些嵌入式系统根本不存在任何硬件上的条件或机制（比如程序上载接口），以帮助用户运行和调试修改后的 MiniGUI 函数库。
- 某些嵌入式系统采用了其他非开放源码的专有操作系统，用户根本无法免费获得用于编译、连接和调试修改后 MiniGUI 函数库的工具。
- 某些嵌入式系统，因为专利或技术保密等原因，禁止用户对程序采用反向工程和反汇编，从而禁止用户调试修改后的 MiniGUI 函数库。

如果您剥夺了用户获得修改 MiniGUI 的权利，即使您遵循了 LGPL 许可证的其他条款发布了修改后的 MiniGUI 源代码，则仍不属于 100% 遵循 LGPL 条款。这时，您需要获得飞漫的商业授权。那么，嵌入式系统需要完成哪些工作才算 100% 遵循 LGPL 条款呢？

- 确保按照 LGPL 条款复制、修改和发布 MiniGUI 函数库本身。
- 如果您采用静态连接方式生成使用 MiniGUI 函数库的可执行程序，则应确保提供用于生成最终可执行程序的全部目标代码和/或源代码。
- 如果您采用动态连接方式使用 MiniGUI 函数库，则应确保在达到接口兼容性的情况下，使用 MiniGUI 函数库的可执行程序仍然能够正常工作。
- 您必须允许用户对您的程序进行反向工程，以便用户调试修改后的 MiniGUI 函数库。
- 如果您采用了非开放源码的专有嵌入式操作系统，则请确保和您的产品一同提供用于编译、连接和调试程序的工具，或者您的用户可免费获得这些工具。
- 您的产品还应该在硬件和软件上提供替换原先 MiniGUI 函数库和/或使用 MiniGUI 的可执行程序的机制和条件，以便用户在您的嵌入式产品中调试和运行修改之后的 MiniGUI 函数库。
- 其他为确保用户具有修改 MiniGUI 函数库的自由而应该提供的法律许可及软硬件条件。

因此，我们认为在嵌入式产品中使用老版本 MiniGUI 也必须购买商业授权。

C.3 GNU 通用公共许可证

```
            GNU GENERAL PUBLIC LICENSE
               Version 2, June 1991

 Copyright (C) 1989, 1991 Free Software Foundation, Inc.
                       59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 Everyone is permitted to copy and distribute verbatim copies
 of this license document, but changing it is not allowed.

                Preamble

  The licenses for most software are designed to take away your
freedom to share and change it.  By contrast, the GNU General Public
License is intended to guarantee your freedom to share and change free
software--to make sure the software is free for all its users.  This
General Public License applies to most of the Free Software
Foundation's software and to any other program whose authors commit to
```

```
using it.  (Some other Free Software Foundation software is covered by
the GNU Library General Public License instead.)  You can apply it to
your programs, too.

  When we speak of free software, we are referring to freedom, not
price.  Our General Public Licenses are designed to make sure that you
have the freedom to distribute copies of free software (and charge for
this service if you wish), that you receive source code or can get it
if you want it, that you can change the software or use pieces of it
in new free programs; and that you know you can do these things.

  To protect your rights, we need to make restrictions that forbid
anyone to deny you these rights or to ask you to surrender the rights.
These restrictions translate to certain responsibilities for you if you
distribute copies of the software, or if you modify it.

  For example, if you distribute copies of such a program, whether
gratis or for a fee, you must give the recipients all the rights that
you have.  You must make sure that they, too, receive or can get the
source code.  And you must show them these terms so they know their
rights.

  We protect your rights with two steps: (1) copyright the software, and
(2) offer you this license which gives you legal permission to copy,
distribute and/or modify the software.

  Also, for each author's protection and ours, we want to make certain
that everyone understands that there is no warranty for this free
software.  If the software is modified by someone else and passed on, we
want its recipients to know that what they have is not the original, so
that any problems introduced by others will not reflect on the original
authors' reputations.

  Finally, any free program is threatened constantly by software
patents.  We wish to avoid the danger that redistributors of a free
program will individually obtain patent licenses, in effect making the
program proprietary.  To prevent this, we have made it clear that any
patent must be licensed for everyone's free use or not licensed at all.

  The precise terms and conditions for copying, distribution and
modification follow.

            GNU GENERAL PUBLIC LICENSE
   TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

  0. This License applies to any program or other work which contains
a notice placed by the copyright holder saying it may be distributed
under the terms of this General Public License.  The "Program", below,
refers to any such program or work, and a "work based on the Program"
means either the Program or any derivative work under copyright law:
that is to say, a work containing the Program or a portion of it,
either verbatim or with modifications and/or translated into another
language.  (Hereinafter, translation is included without limitation in
the term "modification".)  Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not
covered by this License; they are outside its scope.  The act of
running the Program is not restricted, and the output from the Program
is covered only if its contents constitute a work based on the
Program (independent of having been made by running the Program).
Whether that is true depends on what the Program does.

  1. You may copy and distribute verbatim copies of the Program's
source code as you receive it, in any medium, provided that you
conspicuously and appropriately publish on each copy an appropriate
copyright notice and disclaimer of warranty; keep intact all the
notices that refer to this License and to the absence of any warranty;
and give any other recipients of the Program a copy of this License
along with the Program.

You may charge a fee for the physical act of transferring a copy, and
you may at your option offer warranty protection in exchange for a fee.

  2. You may modify your copy or copies of the Program or any portion
of it, thus forming a work based on the Program, and copy and
distribute such modifications or work under the terms of Section 1
above, provided that you also meet all of these conditions:

    a) You must cause the modified files to carry prominent notices
    stating that you changed the files and the date of any change.
```

```
    b) You must cause any work that you distribute or publish, that in
    whole or in part contains or is derived from the Program or any
    part thereof, to be licensed as a whole at no charge to all third
    parties under the terms of this License.

    c) If the modified program normally reads commands interactively
    when run, you must cause it, when started running for such
    interactive use in the most ordinary way, to print or display an
    announcement including an appropriate copyright notice and a
    notice that there is no warranty (or else, saying that you provide
    a warranty) and that users may redistribute the program under
    these conditions, and telling the user how to view a copy of this
    License.  (Exception: if the Program itself is interactive but
    does not normally print such an announcement, your work based on
    the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole.  If
identifiable sections of that work are not derived from the Program,
and can be reasonably considered independent and separate works in
themselves, then this License, and its terms, do not apply to those
sections when you distribute them as separate works.  But when you
distribute the same sections as part of a whole which is a work based
on the Program, the distribution of the whole must be on the terms of
this License, whose permissions for other licensees extend to the
entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest
your rights to work written entirely by you; rather, the intent is to
exercise the right to control the distribution of derivative or
collective works based on the Program.

In addition, mere aggregation of another work not based on the Program
with the Program (or with a work based on the Program) on a volume of
a storage or distribution medium does not bring the other work under
the scope of this License.

  3. You may copy and distribute the Program (or a work based on it,
under Section 2) in object code or executable form under the terms of
Sections 1 and 2 above provided that you also do one of the following:

    a) Accompany it with the complete corresponding machine-readable
    source code, which must be distributed under the terms of Sections
    1 and 2 above on a medium customarily used for software interchange; or,

    b) Accompany it with a written offer, valid for at least three
    years, to give any third party, for a charge no more than your
    cost of physically performing source distribution, a complete
    machine-readable copy of the corresponding source code, to be
    distributed under the terms of Sections 1 and 2 above on a medium
    customarily used for software interchange; or,

    c) Accompany it with the information you received as to the offer
    to distribute corresponding source code.  (This alternative is
    allowed only for noncommercial distribution and only if you
    received the program in object code or executable form with such
    an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for
making modifications to it.  For an executable work, complete source
code means all the source code for all modules it contains, plus any
associated interface definition files, plus the scripts used to
control compilation and installation of the executable.  However, as a
special exception, the source code distributed need not include
anything that is normally distributed (in either source or binary
form) with the major components (compiler, kernel, and so on) of the
operating system on which the executable runs, unless that component
itself accompanies the executable.

If distribution of executable or object code is made by offering
access to copy from a designated place, then offering equivalent
access to copy the source code from the same place counts as
distribution of the source code, even though third parties are not
compelled to copy the source along with the object code.

  4. You may not copy, modify, sublicense, or distribute the Program
except as expressly provided under this License.  Any attempt
otherwise to copy, modify, sublicense or distribute the Program is
void, and will automatically terminate your rights under this License.
However, parties who have received copies, or rights, from you under
this License will not have their licenses terminated so long as such
parties remain in full compliance.
```

```
  5. You are not required to accept this License, since you have not
signed it.  However, nothing else grants you permission to modify or
distribute the Program or its derivative works.  These actions are
prohibited by law if you do not accept this License.  Therefore, by
modifying or distributing the Program (or any work based on the
Program), you indicate your acceptance of this License to do so, and
all its terms and conditions for copying, distributing or modifying
the Program or works based on it.

  6. Each time you redistribute the Program (or any work based on the
Program), the recipient automatically receives a license from the
original licensor to copy, distribute or modify the Program subject to
these terms and conditions.  You may not impose any further
restrictions on the recipients' exercise of the rights granted herein.
You are not responsible for enforcing compliance by third parties to
this License.

  7. If, as a consequence of a court judgment or allegation of patent
infringement or for any other reason (not limited to patent issues),
conditions are imposed on you (whether by court order, agreement or
otherwise) that contradict the conditions of this License, they do not
excuse you from the conditions of this License.  If you cannot
distribute so as to satisfy simultaneously your obligations under this
License and any other pertinent obligations, then as a consequence you
may not distribute the Program at all.  For example, if a patent
license would not permit royalty-free redistribution of the Program by
all those who receive copies directly or indirectly through you, then
the only way you could satisfy both it and this License would be to
refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under
any particular circumstance, the balance of the section is intended to
apply and the section as a whole is intended to apply in other
circumstances.

It is not the purpose of this section to induce you to infringe any
patents or other property right claims or to contest validity of any
such claims; this section has the sole purpose of protecting the
integrity of the free software distribution system, which is
implemented by public license practices.  Many people have made
generous contributions to the wide range of software distributed
through that system in reliance on consistent application of that
system; it is up to the author/donor to decide if he or she is willing
to distribute software through any other system and a licensee cannot
impose that choice.

This section is intended to make thoroughly clear what is believed to
be a consequence of the rest of this License.

  8. If the distribution and/or use of the Program is restricted in
certain countries either by patents or by copyrighted interfaces, the
original copyright holder who places the Program under this License
may add an explicit geographical distribution limitation excluding
those countries, so that distribution is permitted only in or among
countries not thus excluded.  In such case, this License incorporates
the limitation as if written in the body of this License.

  9. The Free Software Foundation may publish revised and/or new versions
of the General Public License from time to time.  Such new versions will
be similar in spirit to the present version, but may differ in detail to
address new problems or concerns.

Each version is given a distinguishing version number.  If the Program
specifies a version number of this License which applies to it and "any
later version", you have the option of following the terms and conditions
either of that version or of any later version published by the Free
Software Foundation.  If the Program does not specify a version number of
this License, you may choose any version ever published by the Free Software
Foundation.

  10. If you wish to incorporate parts of the Program into other free
programs whose distribution conditions are different, write to the author
to ask for permission.  For software which is copyrighted by the Free
Software Foundation, write to the Free Software Foundation; we sometimes
make exceptions for this.  Our decision will be guided by the two goals
of preserving the free status of all derivatives of our free software and
of promoting the sharing and reuse of software generally.

                NO WARRANTY
```

```
  11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY
FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW.  EXCEPT WHEN
OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES
PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED
OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.  THE ENTIRE RISK AS
TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU.  SHOULD THE
PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING,
REPAIR OR CORRECTION.

  12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING
WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR
REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES,
INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING
OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED
TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY
YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER
PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE
POSSIBILITY OF SUCH DAMAGES.

             END OF TERMS AND CONDITIONS

        How to Apply These Terms to Your New Programs

  If you develop a new program, and you want it to be of the greatest
possible use to the public, the best way to achieve this is to make it
free software which everyone can redistribute and change under these terms.

  To do so, attach the following notices to the program.  It is safest
to attach them to the start of each source file to most effectively
convey the exclusion of warranty; and each file should have at least
the "copyright" line and a pointer to where the full notice is found.

    <one line to give the program's name and a brief idea of what it does.>
    Copyright (C) <year>  <name of author>

    This program is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation; either version 2 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this program; if not, write to the Free Software
    Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA


Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this
when it starts in an interactive mode:

    Gnomovision version 69, Copyright (C) year name of author
    Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type `show w'.
    This is free software, and you are welcome to redistribute it
    under certain conditions; type `show c' for details.

The hypothetical commands `show w' and `show c' should show the appropriate
parts of the General Public License.  Of course, the commands you use may
be called something other than `show w' and `show c'; they could even be
mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your
school, if any, to sign a "copyright disclaimer" for the program, if
necessary.  Here is a sample; alter the names:

  Yoyodyne, Inc., hereby disclaims all copyright interest in the program
  `Gnomovision' (which makes passes at compilers) written by James Hacker.

  <signature of Ty Coon>, 1 April 1989
  Ty Coon, President of Vice

This General Public License does not permit incorporating your program into
proprietary programs.  If your program is a subroutine library, you may
consider it more useful to permit linking proprietary applications with the
library.  If this is what you want to do, use the GNU Library General
Public License instead of this License.
```