

目录

目录	1
前言	4
本手册对象	4
本手册架构	4
第一章 概述	5
1.1 什么是嵌入式系统	5
1.2 嵌入式处理器	5
1.3 嵌入式操作系统	7
1.3.1 嵌入式操作系统的种类	7
1.3.2 嵌入式操作系统的发展	7
1.3.3 使用实时操作系统的必要性	7
1.3.4 实时操作系统的优缺点	8
1.4 Linux 操作系统	8
1.5 嵌入式 Linux 以及选择的理由	9
1.6 背景知识	10
第二章 硬件平台设计	10
2.1 AT91RM9200 微处理器简介	11
2.1.1 AT91RM9200 微处理器基本功能模块	11
2.1.2 AT91RM9200 微处理器映射关系	12
2.2 基于 AT91RM9200 的嵌入式硬件平台介绍	16
2.2.1 初步认识实验开发板	16
2.2.2 核心板介绍	18
2.2.3 底板基本功能模块介绍	20
2.3 硬件平台使用说明	27
第三章 构建嵌入式 Linux 系统概述	11
3.1 Bootloader 概述	28
3.1.1 Bootloader 的概念	28
3.1.2 Bootloader 的存储介质	28
3.1.3 Bootloader 的操作模式	29
3.1.4 Bootloader 的典型架构	29
3.1.5 Bootloader 与 Linux 的参数传递	29
3.2 Linux 内核概述	33
3.2.1 嵌入式操作系统必要性	33
3.2.2 嵌入式操作系统移植浅析	33
3.2.3 Linux 操作系统的选择	34
3.2.4 Linux 内核作用	34
3.3 根文件系统概述	35
3.3.1 Linux 文件系统简介	35
3.3.2 根文件系统目录	35
第四章 嵌入式 BootLoader	37

4.1 BootLoader.....	37
4.1.1 BootLoader 介绍.....	37
4.1.2 BootLoader 的启动.....	37
4.1.3 BootLoader 的种类.....	40
4.2 U-Boot.....	41
4.2.1 U-Boot 介绍.....	41
4.2.2 U-Boot 主要目录结构.....	42
4.2.3 U-Boot 支持的主要功能.....	42
4.2.4 U-Boot 移植需要修改的文件.....	43
4.2.5 U-Boot 的编译.....	46
4.2.6 U-Boot 的烧写.....	48
4.2.7 U-Boot 的常用命令.....	55
4.2.8 U-Boot 的环境变量.....	59
4.2.9 U-Boot 启动过程.....	60
4.2.10 U-Boot 与内核的关系.....	66
第五章 Linux.....	75
5.1 Linux 原理.....	75
5.2 ARM Linux.....	81
5.3 编译嵌入式 Linux 内核映像.....	83
5.4 内核调试方法.....	85
5.5 分析 Linux 常用配置.....	86
第六章 根文件系统的制作.....	90
6.1 常用文件系统介绍.....	90
6.1.1 Romfs 文件系统.....	90
6.1.2 Ext2 on ramdisk 文件系统.....	90
6.1.3 Jffs2 文件系统.....	90
6.1.4 Yaffs2 文件系统.....	91
6.2 Ext2 on Ramdisk 文件系统制作流程.....	91
6.2.1 Ramdisk 介绍.....	91
6.2.2 制作 Ext2 on Ramdisk 根文件系统.....	91
6.3 Yaffs2 文件系统的制作流程.....	96
6.3.1 Yaffs2 文件系统作为普通文件系统的制作.....	96
6.3.2 Yaffs2 文件系统作为根文件系统的制作.....	97
第七章 linux 设备驱动.....	99
7.1 驱动开发概要.....	99
7.1.1 驱动概述.....	99
7.1.2 相关命令.....	100
7.1.3 主要接口.....	103
7.2 实验板驱动添加.....	110
7.2.1 网络驱动添加.....	110
7.2.2 U 盘驱动添加.....	110
7.2.3 SD 卡驱动添加.....	111
7.3 驱动开发实例.....	113
第八章 应用程序开发.....	115

8.1 miniGUI 概述	115
8.1.1 概述	115
8.1.2 使用必须注意点	115
8.2 编译安装 miniGUI	116
8.2.1 在 PC 上编译安装 miniGUI	116
8.2.2 交叉编译安装 miniGUI	118
8.3 快速开始 miniGUI 编程	122
8.3.1 基本编程概念	122
8.3.2 从简单例子迅速入门	122
8.3.3 编译、链接、运行	125
8.4 主窗口和对边框编程基础	129
8.4.1 主窗口编程基础	129
8.4.2 对话框编程基础	130
8.5 控件编程基础	133
8.5.1 控件和控件类	133
8.5.2 利用预定义控件创建控件实例	134
8.5.3 控件编程涉及的内容	136
8.5.4 控件专用的操作函数	140
8.6 图形设备接口	141
8.6.1 窗口绘制和刷新	141
8.6.1.1 何时进行绘制	141
8.6.1.2 MSG_PAINT 消息	141
8.6.1.3 有效区域和无效区域	142
8.6.2 图形设备上下文	142
8.6.2.1 图形设备的抽象	142
8.6.2.2 设备上下文句柄的获取和释放	144
8.6.2.3 系统内存中的设备上下文	146
8.6.2.4 屏幕设备上下文	146
8.6.3 基本的图形绘制	146
8.6.3.1 基本绘图属性	146
8.6.3.2 基本绘图函数	146
8.6.3.3 剪切域操作函数	147
8.6.4 文本的处理和显示	148
8.6.4.1 逻辑字体	148
8.6.4.2 文本输出	149
8.7 示波器例子	133
8.7.1 界面构建	151
8.7.2 后台处理	153
第九章 展望	155

前言

本手册是实验室部分同学在经过一段时间的摸索之后总结出来的一些经验。目的是结合嵌入式 Linux 系统的实验平台让实验室其他同学以及以后的同学能够方便、快捷地熟悉嵌入式 Linux 系统，并且能够独立处理一些常见的问题，从而能够更好地将嵌入式系统应用到今后的工作中去。本手册针对的是实验室的专用硬件平台和 Linux 操作系统，对于其它硬件平台和操作系统并不适用。

本手册对象

本手册并不适用所有的对象，因为本手册是建立在一定的基础之上的。必须对嵌入式系统有一定的了解，包括：嵌入式调试软件（ADS），ARM 体系结构，指令集，硬件平台的其他部件。同时还要具有一定的操作经验，做过一些基本的嵌入式系统实验，例如：本实验室的 ARM7 实验箱。同时还要对 Linux 的基础知识有一定了解。

本手册架构

本书总共分九章，具体地阐述了开发嵌入式系统所涉及的各个方面问题。

第一章 “概述”主要是对嵌入式系统有一个整体上的把握，简要地介绍了嵌入式系统所要涉及到的硬件和软件，还包括要了解的背景知识。编写者夏传凯。

第二章 “硬件平台设计”详细地介绍了本手册所用到的实验平台，编写者周庆松。

第三章 “构建嵌入式 Linux 系统概述”阐述了嵌入式系统的基本组成，开发嵌入式系统所需要的环境和具体的制作方法，编写者钱恒顺。

第四章 “嵌入式 BootLoader”具体阐述了 BootLoader 的功能，主要针对 U-Boot 进行具体的制作叙述，详细讲解了 U-Boot 的修改、编译与启动，并简单分析了 U-Boot 的执行代码，编写者周庆松。

第五章 “Linux”针对具体的硬件平台详细的介绍了制作内核映像的过程，并且简要地介绍内核原码中板级支持代码的结构，编写者夏传凯。

第六章 “根文件系统的制作”概括地介绍了根文件系统的组成以及制作方法，编写者钱恒顺。

第七章 “Linux 设备驱动”详细的介绍平台各个硬件接口的驱动开发过程，编写者周伯健。

第八章 “应用程序开发”主要介绍在 Linux 平台上怎样移植 MiniGUI 并且进行图形界面软件开发，编写者王敏敏，吴曜。

第九章 “展望”

第一章 概述

1.1 什么是嵌入式系统

嵌入式系统一般指非 PC 系统，有计算机功能但又不称之为计算机的设备或器材。它是以应用为中心，软硬件可裁减的，适应应用系统对功能、可靠性、成本、体积、功耗等综合性严格要求的专用计算机系统。简单地说，嵌入式系统集系统的应用软件与硬件于一体，类似于 PC 中 BIOS 的工作方式，具有软件代码小、高度自动化、响应速度快等特点，特别适合于要求实时和多任务的体系。

嵌入式系统主要由嵌入式处理器、相关支撑硬件、嵌入式操作系统及应用软件系统等组成，它是可独立工作的“器件”。

嵌入式系统几乎包括了生活中的所有电器设备，如掌上 PDA、移动计算设备、电视机顶盒、手机上网、数字电视、多媒体、汽车、微波炉、数字相机、家庭自动化系统、电梯、空调、安全系统、自动售货机、蜂窝式电话、消费电子设备、工业自动化仪表与医疗仪器等。

嵌入式系统的硬件部分，包括处理器 / 微处理器、存储器及外设器件和 I/O 端口、图形控制器等。嵌入式系统有别于一般的计算机处理系统，它不具备像硬盘那样大容量的存储介质，而大多使用 EPROM、EEPROM 或闪存 (Flash Memory) 作为存储介质。软件部分包括操作系统软件（要求实时和多任务操作）和应用程序编程。应用程序控制着系统的运作和行为；而操作系统控制着应用程序编程与硬件的交互作用。

1.2 嵌入式处理器

嵌入式系统的核心是嵌入式微处理器。嵌入式微处理器一般具备 4 个特点：(1) 对实时和多任务有很强的支持能力，能完成多任务并且有较短的中断响应时间，从而使内部的代码和实时操作系统的执行时间减少到最低限度；(2) 具有功能很强的存储区保护功能，这是由于嵌入式系统的软件结构已模块化，而为了避免在软件模块之间出现错误的交叉作用，需要设计强大的存储区保护功能，同时也有利于软件诊断；(3) 可扩展的处理器结构，以能迅速地扩展出满足应用的高性能的嵌入式微处理器；(4) 嵌入式微处理器的功耗必须很低，尤其是用于便携式的无线及移动的计算和通信设备中靠电池供电的嵌入式系统更是如此，功耗只能为 mW 甚至 μW 级。

据不完全统计，目前全世界嵌入式处理器的品种总量已经超过 1000 种，流行的体系结构有 30 多个系列。其中 8051 体系占多半，生产这种单片机的半导体厂家有 20 多个，共 350 多种衍生产品，仅 Philips 就有近 100 种。现在几乎每个半导体制造商都生产嵌入式处理器，越来越多的公司有自己的处理器设计部门。嵌入式处理器的寻址空间一般从 64kB 到 16MB，处理速度为 0.1~2000MIPS，常用封装 8~144 个引脚。

根据现状，嵌入式计算机可分成下面几类。

(1) 嵌入式微处理器 (Embedded Microprocessor Unit, EMPU)

嵌入式微处理器采用“增强型”通用微处理器。由于嵌入式系统通常应用于环境比较恶劣的环境中，因而嵌入式微处理器在工作温度、电磁兼容性以及可靠性方面的要求较通用的标准微处理器高。但是，嵌入式微处理器在功能方面与标准的微处理器基本上是一样的。根据实际嵌入式应用要求，将嵌入式微处理器装配在专门设计的主板上，只保留和嵌入式应用有关的主板功能，这样可以大幅度减小系统的体积和功耗。和工业控制计算机相比，嵌入式微处理器组成的系统具有体积小、重量轻、成本低、可靠性高的优点，但在其电路板上必须包括 ROM、RAM、总线接口、各种外设等器件，从而降低了系统的可靠性，技术保密性也较差。由嵌入式微处理器及其存储器、总线、外设等安装在一块电路主板上构成一个通常所说的单板机系统。嵌入式处理器目前主要有 Am186/88、386EX、SC-400、Power

PC、68000、MIPS、ARM 系列等。

(2) 嵌入式微控制器 (Microcontroller Unit, MCU)

嵌入式微控制器又称单片机，它将整个计算机系统集成到一块芯片中。嵌入式微控制器一般以某种微处理器内核为核心，根据某些典型的应用，在芯片内部集成了 ROM/EPROM、RAM、总线、总线逻辑、定时 / 计数器、看门狗、I/O、串行口、脉宽调制输出、A/D、D/A、Flash RAM、EEPROM 等各种必要功能部件和外设。为适应不同的应用需求，对功能的设置和外设的配置进行必要的修改和裁减定制，使得一个系列的单片机具有多种衍生产品，每种衍生产品的处理器内核都相同，不同的是存储器和外设的配置及功能的设置。这样可以使单片机最大限度地和应用需求相匹配，从而减少整个系统的功耗和成本。

和嵌入式微处理器相比，微控制器的单片化使应用系统的体积大大减小，从而使功耗和成本大幅度下降、可靠性提高。由于嵌入式微控制器目前在产品的品种和数量上是所有种类嵌入式处理器中最多的，而且上述诸多优点决定了微控制器是嵌入式系统应用的主流。微控制器的片上外设资源一般比较丰富，适合于控制，因此称为微控制器。通常，嵌入式微处理器可分为通用和半通用两类，比较有代表性的通用系列包括 8051、P51XA、MCS-251、MCS-96/196/296、C166/167、68300 等。而比较有代表性的半通用系列，如支持 USB 接口的 MCU 8XC930/931、C540、C541；支持 I2C、CAN 总线、LCD 等的众多专用 MCU 和兼容系列。目前 MCU 约占嵌入式系统市场份额的 70%。

(3) 嵌入式 DSP 处理器 (Embedded Digital Signal Processor, EDSP)

在数字信号处理应用中，各种数字信号处理算法相当复杂，一般结构的处理器无法实时的完成这些运算。由于 DSP 处理器对系统结构和指令进行了特殊设计，使其适合于实时地进行数字信号处理。在数字滤波、FFT、谱分析等方面，DSP 算法正大量进入嵌入式领域，DSP 应用正从在通用单片机中以普通指令实现 DSP 功能，过渡到采用嵌入式 DSP 处理器。嵌入式 DSP 处理器有两类：

(1) DSP 处理器经过单片化、EMC 改造、增加片上外设成为嵌入式 DSP 处理器，TI 的 TMS320C2000 / C5000 等属于此范畴；

(2) 在通用单片机或 SOC 中增加 DSP 协处理器，例如 Intel 的 MCS-296 和 Infineon(Siemens) 的 TriCore。

另外，在有关智能方面的应用中，也需要嵌入式 DSP 处理器，例如各种带有智能逻辑的消费类产品，生物信息识别终端，带有加解密算法的键盘，ADSL 接入、实时语音解压系统，虚拟现实显示等。这类智能化算法一般都是运算量较大，特别是向量运算、指针线性寻址等较多，而这些正是 DSP 处理器的优势所在。嵌入式 DSP 处理器比较有代表性的产品是 TI 的 TMS320 系列和 Motorola 的 DSP56000 系列。TMS320 系列处理器包括用于控制的 C2000 系列、移动通信的 C5000 系列，以及性能更高的 C6000 和 C8000 系列。DSP56000 目前已经发展成为 DSP56000、DSP56100、DSP56200 和 DSP56300 等几个不同系列的处理器。另外，Philips 公司最近也推出了基于可重置嵌入式 DSP 结构，采用低成本、低功耗技术制造的 R. E. A. L DSP 处理器，其特点是具备双 Harvard 结构和双乘 / 累加单元，应用目标是大批量消费类产品。

(4) 嵌入式片上系统 (System On Chip, SOC)

随着 EDI 的推广和 VLSI 设计的普及化，以及半导体工艺的迅速发展，可以在一块硅片上实现一个更为复杂的系统，这就产生了 SOC 技术。各种通用处理器内核将作为 SOC 设计公司的标准库，和其他许多嵌入式系统外设一样，成为 VLSI 设计中一种标准的器件，用标准的 VHDL、Verlog 等硬件语言描述，存储在器件库中。用户只需定义出其整个应用系统，仿真通过后就可以将设计图交给半导体工厂制作样品。这样除某些无法集成的器件以外，整个嵌入式系统大部分均可集成到一块或几块芯片中去，应用系统电路板将变得很简单，对于减小整个应用系统体积和功耗、提高可靠性非常有利。

SOC 可分为通用和专用两类，通用 SOC 如 Infineon(Siemens) 的 TriCore、Motorola 的 M-Core，以及某些 ARM 系列器件，如 Echelon 和 Motorola 联合研制的 Neuron 芯片等；专用 SOC 一般专用于某个或某类系统中，如 Philips 的 Smart XA，它将 XA 单片机内核和支持超过 2048 位复杂 RSA 算法的 CCU 单元制作在一块硅片上，形成一个可加载 Java 或 C 语言的专用 SOC，可用于互联网安全方面。

1.3 嵌入式操作系统

嵌入式操作系统是一种支持嵌入式系统应用的操作系统软件，它是嵌入式系统（包括硬、软件系统）极为重要的组成部分，通常包括与硬件相关的底层驱动软件、系统内核、设备驱动接口、通信协议、图形界面、标准化浏览器等 **Browser**。嵌入式操作系统具有通用操作系统的基本特点，如能够有效管理越来越复杂的系统资源；能够把硬件虚拟化，使得开发人员从繁忙的驱动程序移植和维护中解脱出来；能够提供库函数、驱动程序、工具集以及应用程序。与通用操作系统相比较，嵌入式操作系统在系统实时高效性、硬件的相关依赖性、软件固态化以及应用的专用性等方面具有较为突出的特点。

1.3.1 嵌入式操作系统的种类

一般情况下，嵌入式操作系统可以分为两类，一类是面向控制、通信等领域的实时操作系统，如 WindRiver 公司的 VxWorks、ISI 的 pSOS、QNX 系统软件公司的 QNX、ATI 的 Nucleus 等；另一类是面向消费电子产品的非实时操作系统，例如：Microsoft 的 Windows CE，Linux。这类产品包括个人数字助理 (PDA)、移动电话、机顶盒、电子书、WebPhone 等。

1.3.2 嵌入式操作系统的发展

嵌入式操作系统伴随着嵌入式系统的发展经历了 4 个比较明显的阶段。

第一阶段是无操作系统的嵌入算法阶段，是以单芯片为核心的可编程控制器形式的系统，同时具有与监测、伺服、指示设备相配合的功能。这种系统大部分应用于一些专业性极强的工业控制系统中，一般没有操作系统的支持，通过汇编语言编程对系统进行直接控制，运行结束后清除内存。这一阶段系统的主要特点是：系统结构和功能都相对单一，处理效率较低，存储容量较小，几乎没有用户接口。由于这种嵌入式系统使用简便、价格很低，以前在国内工业领域应用较为普遍，但是已经远远不能适应高效的、需要大容量存储介质的现代化工业控制和新兴的信息家电等领域的需求。

第二阶段是以嵌入式 CPU 为基础、以简单操作系统为核心的嵌入式系统。这一阶段系统的主要特点是：CPU 种类繁多，通用性比较差；系统开销小，效率高；一般配备系统仿真器，操作系统具有一定的兼容性和扩展性；应用软件较专业，用户界面不够友好；系统主要用来控制系统负载以及监控应用程序运行。

第三阶段是通用的嵌入式实时操作系统阶段，是以嵌入式操作系统为核心的嵌入式系统。这一阶段系统的主要特点是：嵌入式操作系统能运行于各种不同类型的微处理器上，兼容性好；操作系统内核精小、效率高，并且具有高度的模块化和扩展性；具备文件和目录管理、设备支持、多任务、网络支持、图形窗口以及用户界面等功能；具有大量的应用程序接口 (API)，开发应用程序简单；嵌入式应用软件丰富。第四阶段是以基于 Internet 为标志的嵌入式系统，这是一个正在迅速发展的阶段。目前大多数嵌入式系统还孤立于 Internet 之外，但随着 Internet 的发展以及 Internet 技术与信息家电、工业控制技术等结合日益密切，嵌入式设备与 Internet 的结合将代表着嵌入式技术的真正未来。

1.3.3 使用实时操作系统的必要性

嵌入式实时操作系统在目前的嵌入式应用中用得越来越广泛，尤其在功能复杂、系统庞大的应用中显得愈来愈重要。

首先，嵌入式实时操作系统提高了系统的可靠性。在控制系统中，出于安全方面的考虑，

要求系统起码不能崩溃，而且还要有自愈能力。不仅要求在硬件设计方面提高系统的可靠性和抗干扰性，而且也应在软件设计方面提高系统的抗干扰性，尽可能地减少安全漏洞和不可靠的隐患。长期以来的前后台系统软件设计在遇到强干扰时，使得运行的程序产生异常、出错、跑飞，甚至死循环，造成了系统的崩溃。而实时操作系统管理的系统，这种干扰可能只是引起若干进程中的一个被破坏，可以通过系统运行的系统监控进程对其进行修复。通常情况下，这个系统监视进程用来监视各进程运行状况，遇到异常情况时采取一些利于系统稳定可靠的措施，如把有问题的任务清除掉。

其次，提高了开发效率，缩短了开发周期。在嵌入式实时操作系统环境下，开发一个复杂的应用程序，通常可以按照软件工程中的解耦原则将整个程序分解为多个任务模块。每个任务模块的调试、修改几乎不影响其他模块。商业软件一般都提供了良好的多任务调试环境。再次，嵌入式实时操作系统充分发挥了 32 位 CPU 的多任务潜力。32 位 CPU 比 8、16 位 CPU 快，另外它本来是为运行多用户、多任务操作系统而设计的，特别适于运行多任务实时系统。32 位 CPU 采用利于提高系统可靠性和稳定性的设计，使其更容易做到不崩溃。例如，CPU 运行状态分为系统态和用户态。将系统堆栈和用户堆栈分开，以及实时地给出 CPU 的运行状态等，允许用户在系统设计中从硬件和软件两方面对实时内核的运行实施保护。如果还是采用以前的前后台方式，则无法发挥 32 位 CPU 的优势。

从某种意义上说，没有操作系统的计算机（裸机）是没有用的。在嵌入式应用中，只有把 CPU 嵌入到系统中，同时又把操作系统嵌入进去，才是真正的计算机嵌入式系统。

1.3.4 实时操作系统的优缺点

在嵌入式实时操作系统环境下开发实时应用程序使程序的设计和扩展变得容易，不需要大的改动就可以增加新的功能。通过将应用程序分割成若干独立的任务模块，使应用程序的设计过程大为简化；而且对实时性要求苛刻的事件都得到了快速、可靠的处理。通过有效的系统服务，嵌入式实时操作系统使得系统资源得到更好的利用。但是，使用嵌入式实时操作系统还需要额外的 ROM/RAM 开销，2~5% 的 CPU 额外负荷，以及内核的费用。

1.4 Linux 操作系统

详细参看嵌入式操作系统的PDF

Linux 操作系统核心最早是由芬兰的 Linus Torvalds 1991 年 8 月在芬兰赫尔辛基大学上学时发布的，后来经过众多世界顶尖的软件工程师的不断修改和完善，Linux 得以在全球普及开来，在服务器领域及个人桌面版得到越来越多的应用，在嵌入式开发方面更是具有其它操作系统无可比拟的优势，并以每年 100% 的用户递增数量显示 Linux 强大的力量。Linux 的是一套免费的 32 位多人多工的操作系统，运行方式同 UNIX 系统很像，但 Linux 系统的稳定性、多工能力与网络功能已是许多商业操作系统无法比拟的，Linux 还有一项最大的特色在于源代码完全公开，在符合 GNU GPL(General Public License)的原则下，任何人皆可自由取得、散布、甚至修改源代码。与其它操作系统相比，Linux 还具有以下特色：

- ① 采用阶层式目录结构，文件归类清楚、容易管理。
- ② 支持多种文件系统，如 Ext2FS、ISOFS 以及 Windows 的文件系统 FAT16、FAT32、NTFS 等。
- ③ 具有可移植性，系统核心只有小于 10% 的源代码采用汇编语言编写，其余均是采用 C 语言编写，因此具备高度移植性。
- ④ 可与其它的操作系统如 Windows98/2000/xp 等并存于同一台计算机上。

主流 Linux 操作系统发行版简介就 Linux 的本质来说，它只是操作系统的核心，负责控制硬件、管理文件系统、程序进程等。Linux Kernel(内核)并不负责提供用户强大的应用程序，没有编译器、系统管理工具、网络工具、Office 套件、多媒体、绘图软件等，这样的系统也

就无法发挥其强大功能，用户也无法利用这个系统工作，因此有人便提出以 Linux Kernel 为核心再集成搭配各式各样的系统程序或应用工具程序组成一套完整的操作系统，经过如此组合的 Linux 套件即称为 Linux 发行版。国外封装的 Linux 以 Red Hat(又称为“红帽 Linux”)、OpenLinux、SuSE、TurboLinux 等最为成功。Red Hat Linux: <http://www.redhat.com> Red Hat 是个商业气息颇为浓厚的公司，不仅展现开创 Linux 商业软件的企图心，也在 1999 年在美国科技股为主的那斯达克让公司股票成功上市，Red Hat 渐渐被拱为 Linux 商业界龙头。Red Hat 是目前销售量最高、安装最简便、最适合初学者的 Linux 发行版，也是目前世界上最流行的 Linux 发行套件，它的市场营销、包装及服务做的相当不错，自行开发了 RPM 套件管理程序及 X 桌面环境 Gnome 的众多软件并将其源代码回馈给 Open Source community (开源社区)。国内 Linux 发行版做的相对比较成功是红旗和中软两个版本，界面做得都非常的美观，安装也比较容易，新版本逐渐屏蔽了一些底层的操作，适合于新手使用。两个版本都是源于中国科学院软件研究所承担的国家 863 计划的 Linux 项目，但无论稳定性与兼容性与国外的版本相比都有一定的差距，操作界面与习惯与 Windows 越来越像，提供一定技术支持和售后服务，适宜于国内做低价的操作系统解决方案。

1.5 嵌入式 Linux 以及选择的理由

Linux 代表三个涵义：一个内核，一个系统，一个发行套件。严格来说，Linux 指的是 Linus Torvalds 维护的（及通过主要和镜像网站发布的）内核。Linux 的程序代码库只有内核，绝对不包含工具程序，并且内核中包含各种体系结构的代码，通过不同的编译工具就可以生成各种平台的映像（最终的二进制文件）。内核就是整个操作系统的核心。内核或许不是系统执行的第一个软件，因为引导加载程序可能会在它之前先执行。但内核执行之后，直到系统关机，都不会被置换出主存储器或被移除。实际上，内核会控制所有硬件，并对系统执行的其他软件提供较高级别的硬件抽象。因为内核不断更新，所以必须使用一种编号方案来区分特定的发行版本。这个编号机制用点号相隔的三个数字来区分各个发行版本（内核版本）。头两个数字用来指定版本编号（第一个数字是主版本号，第二个数字是次版本号），第三个数字用来指定发行编号。例如：Linux 2.6.19.2 的版本编号是 2.6，而发行编号是 19，发行次编号是 2。奇数的次版本号，如 2.5，用来指定的是开发版内核。偶数的次版本号，如 2.6，用来指定稳定版内核。通常应该为嵌入式系统使用最新发行的稳定版内核。

一个系统：嵌入式 Linux 系统，是一个基于 Linux 内核的嵌入式系统。一个发行套件：发行套件包括基于 PC 机的 Linux 发行套件和嵌入式 Linux 发行套件。前一个已经在上一节中提到过了，嵌入式 Linux 发行套件包括：用来开发嵌入式 Linux 系统的平台、各种为了在嵌入式系统中使用而裁剪过的应用软件的集合，例如：源代码浏览器、交叉编译器、调试器、项目管理软件、引导映像生成器等等。

让开发者自愿舍传统嵌入式操作系统而取 Linux，主要的原因有：

- 程序代码的质量与可靠度：在质量方面，Linux 提供模块化的设计方案，不同的功能放在不同的模块，复杂的功能被细分成适当数量的独立函数，同时源代码很容易修改和扩充，并且提供可视化的配置界面。在可靠度方面，Linux 可以提供从错误中恢复的能力和长期运行的能力。
- 程序代码的可用性：你可以没有限制地随意取用 Linux 的源码及所有生成工具。最重要的 Linux 组件，包括内核本身，都是在 GNU (GNU Not Unix) 通用公共许可证 (GNU General Public License, GPL) 的保护下发行的。因此取用这些组件的源码是每个人的权利。其他组件则是在类似的许可证下发行的。
- 对硬件的支持：广泛的硬件支持，意味着 Linux 支持各种类型的硬件平台与设备。

有没有详细的？

大量的驱动程序是由 Linux 社区自己维护, 所以将 Linux 移植到各种平台上非常方便。

- 通讯协议与软件标准: Linux 提供广泛的通信协议及软件标准支持, 所以可以轻易地将 Linux 系统整合进既有的架构, 以及将过去的软件移植到 Linux 上。也可以将 Linux 系统整合进既有的 Windows 网络。
- 可用工具: 目前 Linux 有各式各样的工具可用, 如果需要使用某个应用程序, 或许有人在你之前就需要这个程序, 也可能已经有人花时间将这个工具写了出来, 并且放在 Internet 上供人随意取用。访问 [Freshmeat\(http://www.freshmeat.net\)](http://www.freshmeat.net) 和 [SourceForge\(http://www.sourceforge.net\)](http://www.sourceforge.net) 等网站就可以找到很多有用工具了。
- 社群的支持: 受到社群的支持或许是 Linux 最大的优点。这就是自由软件及开源的精神所在。当需要使用某个应用程序时, 很可能早在你之前就有人遇到跟你类似的情况。通常, 这个人将会很乐意跟你分享他的解决方案, 为你回答问题。与开发和支持有关的邮件论坛, 是找到这类社群支持的最佳场所, 在这些地方找到的专家支持的级别通常胜过私有操作系统厂商昂贵的电话支持。
- 许可: Linux 的许可让程序设计者能够为 Linux 做他们无法为私有软件做的事。基本上, 可以使用、修改、并再次发行该软件, 但仅限于向该软件的用户提供相同的权利。这就是 Linux 用到的各种许可证(GPL、LGPL、BSD、MPL 等等)的主要精神。
- 不依赖特定厂商: 不依赖特定厂商, 意味着要获取或使用 Linux 不需要依赖单一厂商。此外, 只要不满意某个厂商的服务态度, 大可以更换厂商, 因为 Linux 发行时的许可证, 让你即使更换厂商也能获得相同的权利。
- 成本: 因为 Linux 是开放源码许可的, 所以比其他操作系统都有成本的优势。

1.6 背景知识

在正式开始下面的内容之前, 我们必须要了解一些必备的背景知识。主要包括: ARM 体系结构, 指令集, AT91RM9200 处理器, ADS 开发软件。因为我们的实验平台用的是基于 ARM 体系结构的 AT91RM9200 处理器。Linux 工作原理, GCC 工具链, 交叉编译。虚拟机软件, Red Hat Linux 使用。

自由软件网站: www.gnu.org。Linux 内核官方下载网址: www.kernel.org。ARM Linux 的官方网址: www.arm.linux.org.uk。

参考书籍:

1. 构建嵌入式 Linux 系统。
2. 嵌入式 Linux 系统开发技术详解----基于 ARM。

第二章 硬件平台设计

本章为大家介绍本手册所用到的硬件平台—ARM9 实验开发板, 内容包括开发板上核心芯片 AT91RM9200 和外围设备的简介、开发板支持的功能介绍, 以及开发板的使用说明。我们下面所涉及到的 U-BOOT、内核、根文件系统的实现都将在该开发板上进行; 因此, 了解该开发板资源, 熟悉开发板硬件结构是我们进行软件开发和调试的基础和前提。

2.1 AT91RM9200 微处理器简介

AT91RM9200 是完全围绕 ARM920T ARM Thumb 处理器构建的系统。它有丰富的系统与应用外设及标准的接口, 从而为低功耗、低成本、高性能的计算机宽范围应用提供一个单片解决方案。

AT91RM9200 包括一个高速片上 SRAM 工作区及一个低等待时间的外部总线接口 (EBI), 以完成应用所要求的片外存储器和内部存储器映射外设配置的无缝连接。EBI 有同步 DRAM (SDRAM)、Burst Flash 及静态存储器的控制器, 并设计了专用电路以方便与 SmartMedia、

CompactFlash 及 NAND Flash 连接。

高级中断控制器 (AIC) 通过多向量, 中断源优先级划分及缩短中断处理传输时间来提高 ARM920T 处理器的中断处理性能。

外设数据控制器 (PDC) 向所有的串行外设提供 DMA 通道, 使其与片内或片外存储器传输数据时不用经过处理器。这就减少了传输连续数据流时处理器的开销。包含双指针的 PDC 控制器极大的简化了 AT91RM9200 的缓冲器链接。

并行 I/O (PIO) 控制器与作为通用数据的 I/O 复用外设输入/ 输出口线, 以最大程度上适应器件的配置。每条口线上包含有一个输入变化中断、开漏能力及可编程上拉电阻。

电源管理控制器 (PMC) 通过软件控制有选择的使能 / 禁用处理器及各种外设来使系统的功耗保持最低。它用一个增强的时钟产生器来提供包括慢时钟(32 kHz) 在内的选定时钟信号, 以随时优化功耗与性能。

AT91RM9200 集成了许多标准接口, 包括 USB 2.0 全速主机和设备端口及与多数外设和在网络层广泛使用的 10/100 Base-T 以太网媒体访问控制器(MAC)。此外, 它还提供一系列符合工业标准的外设, 可在音频、电信、Flash 卡红外线及智能卡中使用。

为完善性能, AT91RM9200 集成了包括 JTAG-ICE、专门 UART 调试通道(DBGU)及嵌入式的实时追踪的一系列的调试功能。这些功能使得开发、调试所有的应用特别是受实时性限制的应用成为可能。

2.1.1 AT91RM9200 微处理器基本功能模块

ARM920T 微处理器芯片中集成的功能模块有:

- 1) ARM920T 处理器内核
- 2) 调试与测试单元
- 3) 引导程序
- 4) 嵌入式软件服务
- 5) 复位控制器
- 6) 存储控制器
- 7) 外部总线接口
- 8) 静态存储控制器

- 9) SDRAM 控制器
- 10) Burst Flash 控制器
- 11) 外设数据控制器
- 12) 增强的中断控制器
- 13) 电源管理控制器
- 14) 系统定时器
- 15) 实时时钟
- 16) 调试单元
- 17) PIO 控制器
- 18) USB 主机端口
- 19) USB 器件端口
- 20) 以太网 MAC
- 21) 串行外设接口
- 22) 两线接口
- 23) USART
- 24) 串行同步控制器
- 25) 定时/ 计数器
- 26) 多媒体卡接口

各个模块的主要特性请参考 at91rm9200 的 data sheet。

2.1.2 AT91RM9200 微处理器映射关系

微处理器通过译码将 4G 的地址空间分为 16 个 256M 字节的区域。区域 1 ~ 8 对应 EBI，和外部片选 NC0 ~ NCS7 相联系。

区域 0 为内部存储器地址，再通过第二级译码提供 1M 字节内部存储空间。

区域 15 为外设地址，且提供对高级外设总线(APB) 的访问。系统外设映射到地址空间的头 4K 字节中，地址范围为：0xFFFF F000 ~ 0xFFFF FFFF。每个外设 256 或 512 个字节。

其它区域未使用，使用它们进行访问时将向发出访问请求的主机发出异常中断。

图 2.1.1 是外部存储器映射示意图；图 2.1.2 是内部存储器映射示意图；图 2.1.2 是系统外设映射示意图。

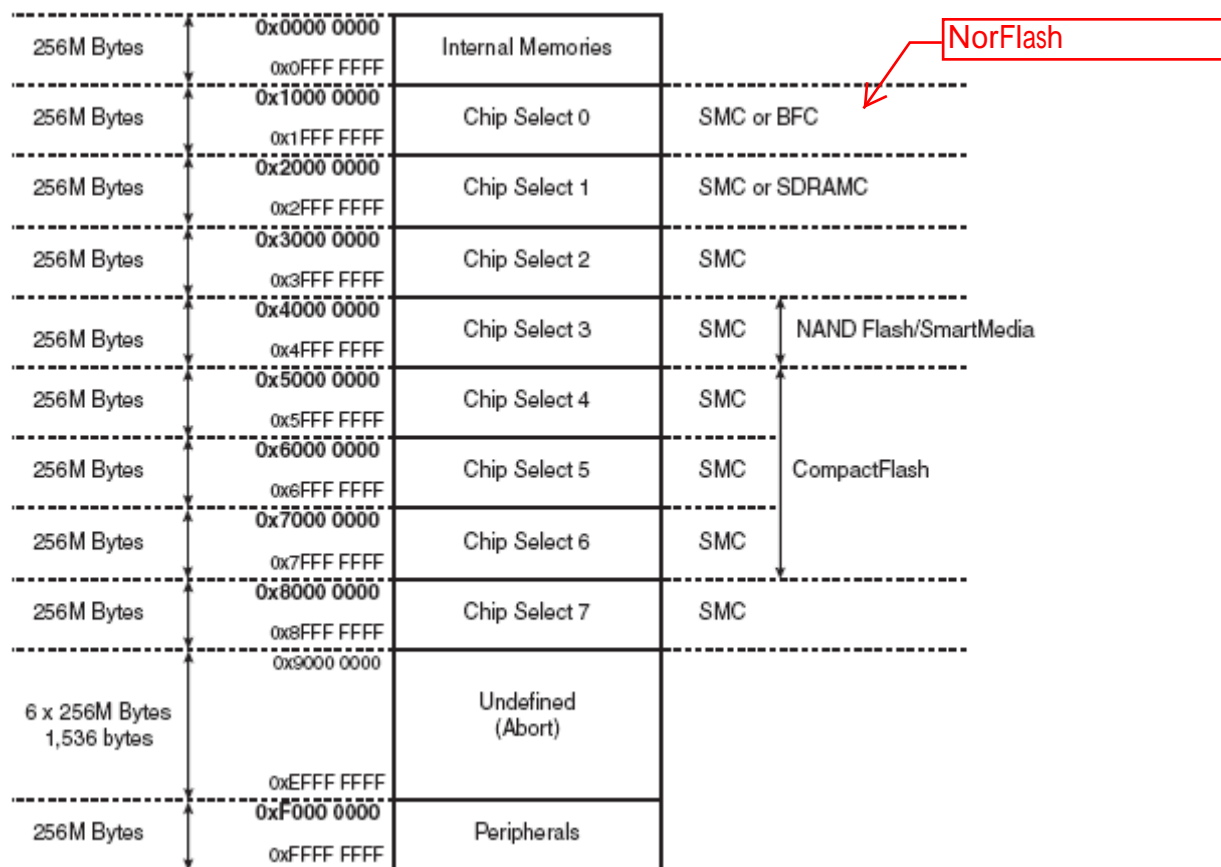


图 2.1.1 外部存储器映射

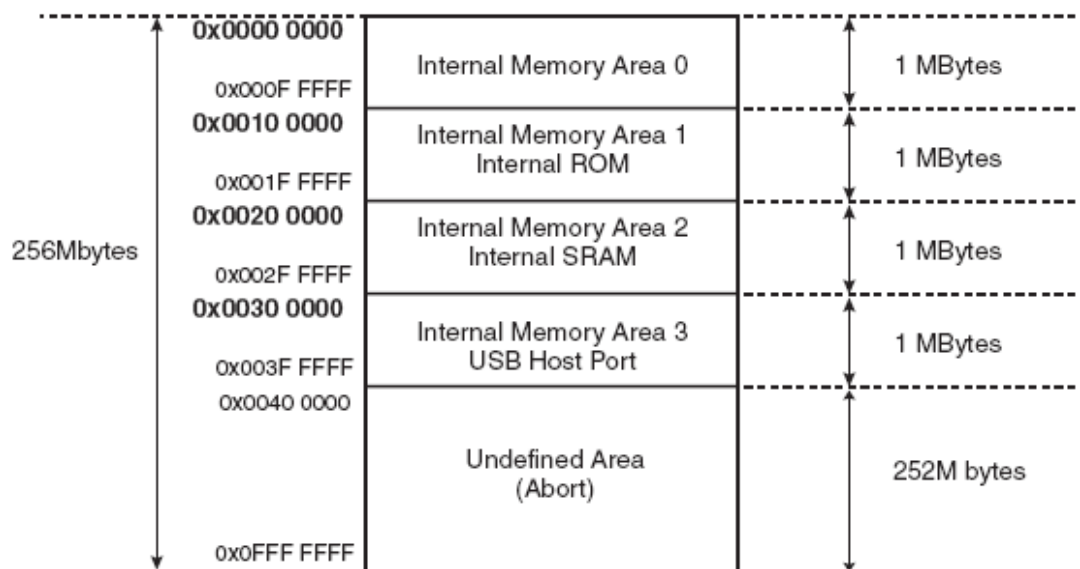


图 2.1.2 内部存储器映射

		Peripheral Name	Size
0xFFFF F000	AIC	Advanced Interrupt Controller	512 bytes/128 registers
0xFFFF F1FF 0xFFFF F200	DBGU	Debug Unit	512 bytes/128 registers
0xFFFF F3FF 0xFFFF F400	PIOA	PIO Controller A	512 bytes/128 registers
0xFFFF F5FF 0xFFFF F600	PIOB	PIO Controller B	512 bytes/128 registers
0xFFFF F7FF 0xFFFF F800	PIOC	PIO Controller C	512 bytes/128 registers
0xFFFF F9FF 0xFFFF FA00	PIOD	PIO Controller D	512 bytes/128 registers
0xFFFF FBFF 0xFFFF FC00	PMC	Power Management Controller	256 bytes/64 registers
0xFFFF FCFE 0xFFFF FD00	ST	System Timer	256 bytes/64 registers
0xFFFF FDFE 0xFFFF FE00	RTC	Real-time Clock	256 bytes/64 registers
0xFFFF FEFF 0xFFFF FF00	MC	Memory Controller	256 bytes/64 registers
0xFFFF FFFF			

图 2.1.3 系统外设映射示意图

用户外设映射到地址空间的前 256M 字节中，地址范围：0xFFFA 0000 ~ 0xFFFE 3FFF。每个外设地址空间为 16-K 字节。图 2.1.4 为用户外设映射示意图。

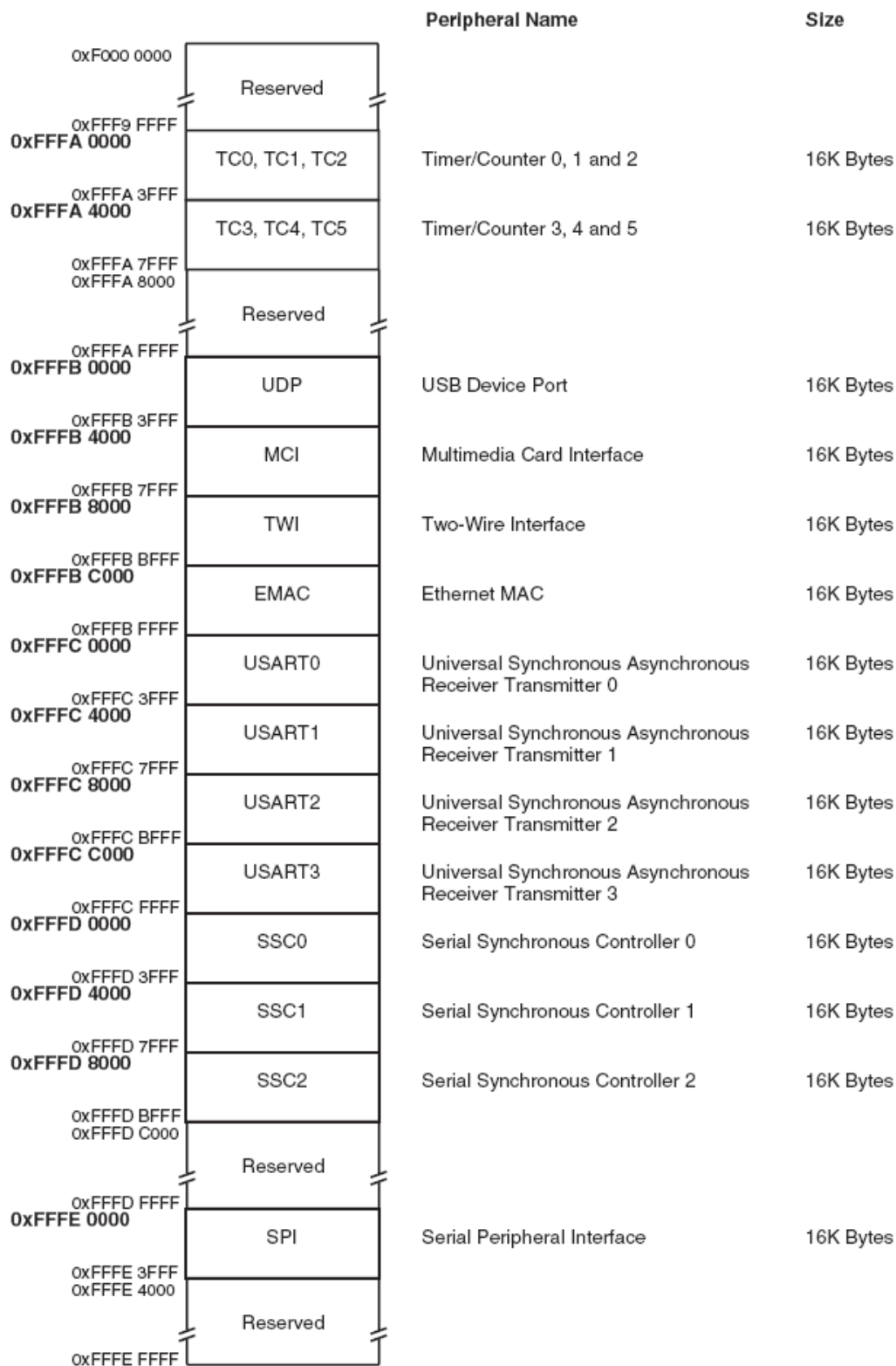


图 2.1.4 用户外设映射示意图

2.2 基于 AT91RM9200 的嵌入式硬件平台介绍

2.2.1 初步认识实验开发板

本实验板是基于 AT91RM9200 微处理器的实验开发系统。系统主要由 AT91RM9200 微处理器、存储系统、外围设备以及电源、接插件、指示灯和端口等部分组成。本实验板由核心板和底板两部分组成，核心板上主要由 AT91RM9200 微处理器，SDRAM、Flash、EEPROM 等存储单元、电源、跳线、JTAG 接口等几部分构成；底板上则主要是由外围设备构成，主要包括电源、键盘、USB 接口、网络接口、串行通讯接口、SD Card 接口、显示接口、总线扩展接口等。实验板(核心板和底板)的俯视图如下图 2.2.1 和图 2.2.2 所示。下面将分别对核心板和底板作相关介绍。

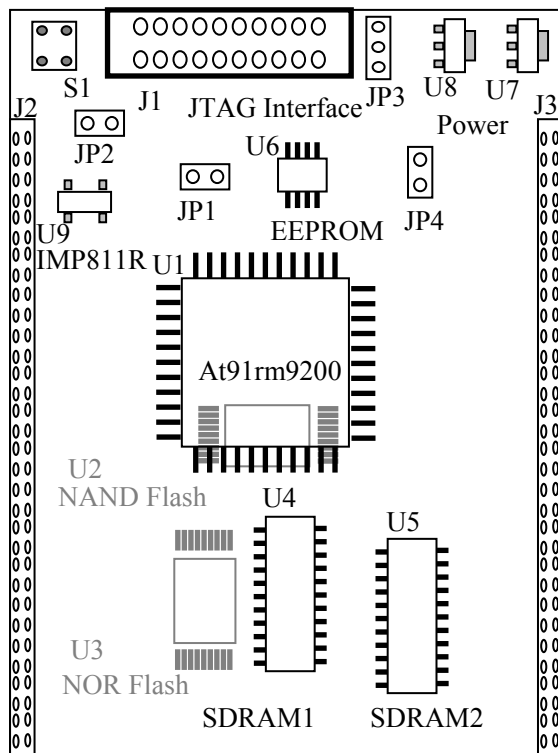


图 2.2.1 核心板俯视图

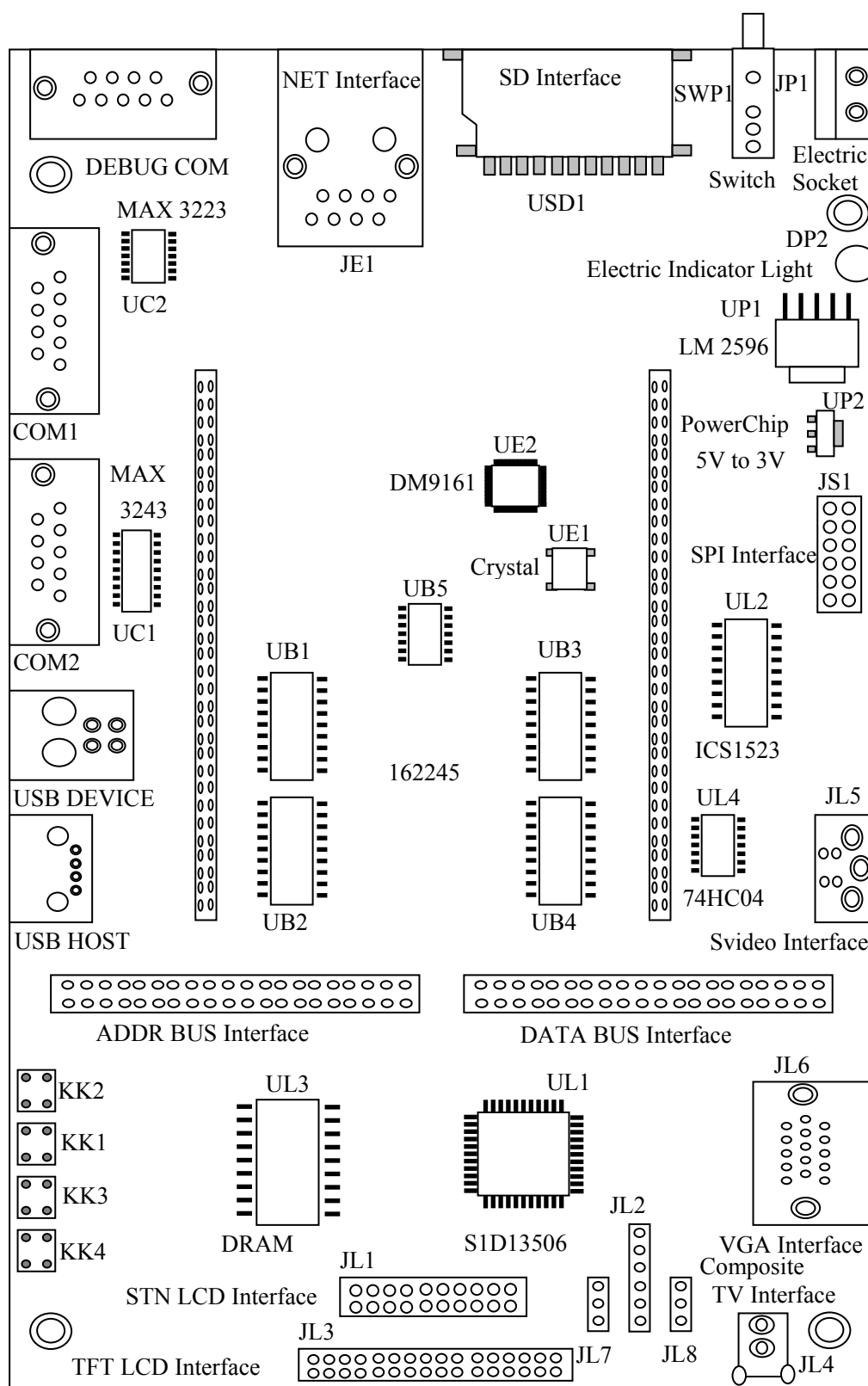


图 2.2.2 底板俯视图

2.2.2 核心板介绍

核心板主要由 AT91RM9200 微处理器和存储系统组成，微处理器已在前面介绍，这里主要介绍一下板子的存储系统。

1) 存储系统

板上存储系统主要包括一片 32Mbyte 的 NANDFlash (K9F5608U0D,在核心板的反面)、一片 8Mbyte 的 NORFlash(SST39VF6401(B),在核心板的反面)、和两片共 64Mbyte 的 SDRAM (HY57V561620CTP),另外还有一片 32Kbyte 的 EEPROM (AT24C256)。存储器芯片列表如表 2.2.1 所示, 图 2.2.3 为存储器与 AT91RM9200 的接口框图。

存储器类型	芯片型号	使用片数	寻址大小	片选对应微处理器管脚	说明
NAND Flash	K9F5608U0D	1	32Mbyte		
NOR Flash	SST39VF6401	1	8Mbyte	NCS0	另一个板子上为 SST39VF6401B
SDRAM	HY57V561620	2	64Mbyte	NCS1	
EEPROM	AT24C256	1	32Kbyte		

表 2.2.1: 板上使用的存储器芯片列表

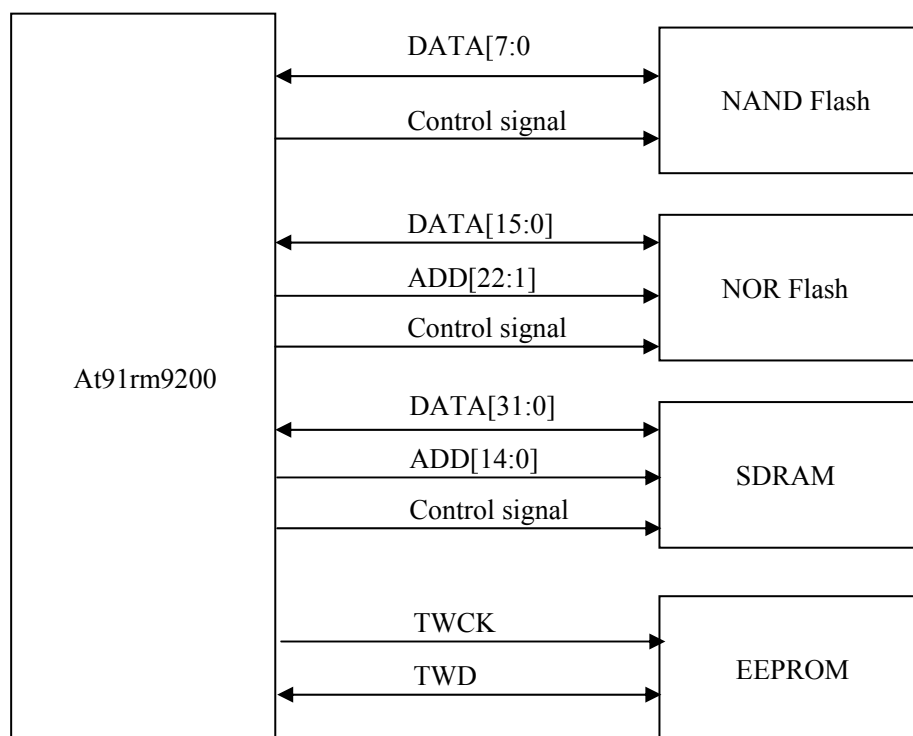


图 2.2.3 存储器接口框图

2) 其他器件说明

除了存储系统外，核心板上模块还包括电源、跳线及 JTAG 插座。

(1) 电源

U7、U8 为电源芯片，U7 的型号为 AMS1117-3.3，为 5V 转 3.3V 芯片；U8 的型号为 AMS1117-1.8，为 5V 转 1.8V 芯片。核心板上 5V 电源由底板提供。

(2) 跳线

核心板上有 4 个跳线，分别为 JP1、JP2、JP3 和 JP4，它们的功能如表 2.2.2 所示：

跳线编号	功能说明	默认状态
JP1	BOOT 模式选择(连接/不连接)	连接，选择从 NOR Flash 启动
JP2	JTAG 处 nTRST 与 nSRST 之间的连接(连接/不连接)	不连接
JP3	JTAG 模式选择(接 VCC/接 GND)	接地选择 ICE 模式
JP4	EEPROM 器件 ID 选择(连接 VCC/不连接)	连接,EEPROM 器件 ID 为 0b10

表 2.2.2 核心板跳线功能说明

(3) JTAG 插座

JTAG 插座对应于核心板上的插座 J1，它在板上连接到 AT91RM9200 芯片的 JTAG 接口，插座的信号配置要和 ARM Multi-ICE 接口一致。在实际的调试过程中，使用 ARM 的 EmbeddedICE 调试结构，图 2.2.4 和表 2.2.3 分别是其引脚排列和对应定义。

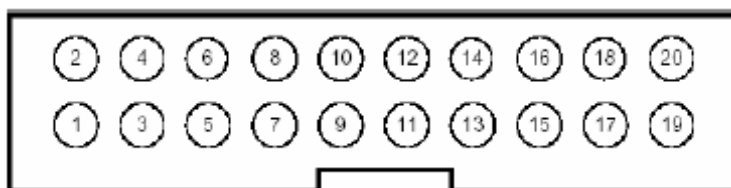


图 2.2.4 20PIN JTAG 插座示意图

PIN 号	名称	功能
1, 2	VDD	工作电压
3	TRST	测试复位，低电平有效（接上拉电阻）
5	TDI	测试数据输入（接上拉电阻）
7	TMS	测试模式选择（接上拉电阻）
9	TCK	测试时钟（接上拉电阻）
11	RTCK	接地（注：一般是与 9 脚连接，这里是接地）
13	TD0	测试数据输出
15	RESET	通过一个 10K 电阻
17, 19	NC	开路
4, 6, 8, 10, 12, 14, 16, 18, 20	GND	系统地

表 2.2.3 20PIN JTAG 插座管脚定义

2.2.3 底板基本功能模块介绍

底板上主要是由外围设备构成，外围设备主要包括键盘、USB 接口、网络接口、串行通讯接口、SPI 接口、SD Card 接口、LCD 控制器、显示接口等。

(1) 键盘

本实验板提供 2×2 键盘，其与 AT91RM9200 的连接电路图如图 2.2.5 所示，PB7~PB10 作为键盘的 4 根信号线，按键定义可以用软件调配，通过两组信号（PB7、PB8 和 PB9、PB10）的组合来判断哪个按键按下。

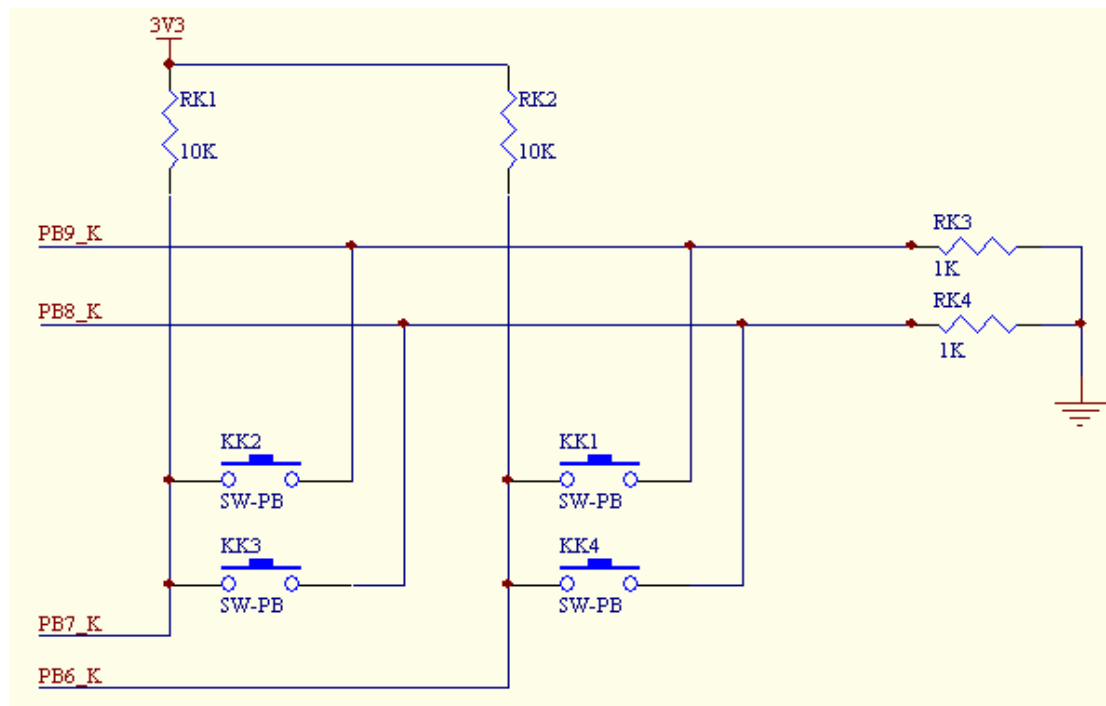


图 2.2.5 键盘与 AT91RM9200 的连接电路图

(2) USB 接口

AT91RM9200 微处理器带有 USB 主机端口和 USB 器件端口。

USB 主机端口(UHP)在主机应用中与 USB 连接，其主要特性如下：

- 与开 HCI Rev 1.0 规范兼容
- 与 USB V2.0 全速及低速规范兼容
- 支持低速 1.5 Mbps 与全速 12 Mbps 的 USB 设备
- 有两个下游 USB 端口的根集线器
- 内置 USB 收发器(收发器数目由产品决定)
- 支持电源管理
- 在 ASB 总线上作为主机工作

USB 在板子上的主机端口接口电路图如下图 2.2.6 所示：

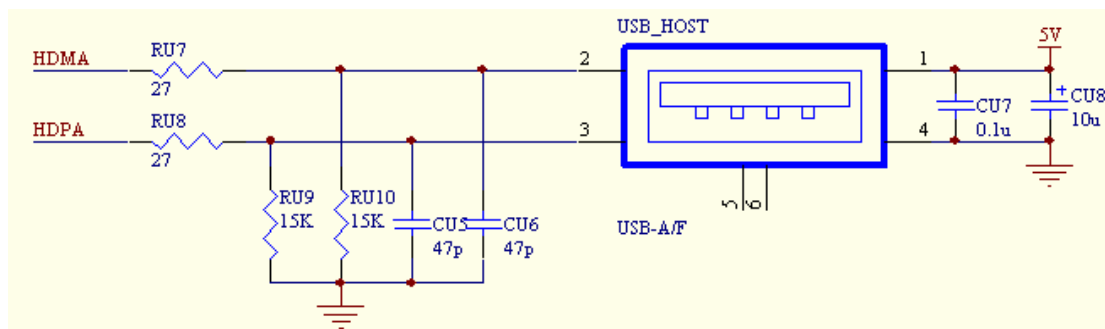


图 2.2.6 USB HOST 接口电路图

USB 器件端口(UDP)适用于通用串行总线(USB) V2.0 全速器件规范。它为 Atmel 的与 ARM7TDMI 与 ARM9TDMI 内核连接的内置 USB 收发器设计。

USB 器件自动检测挂起与恢复，通过出现中断来停止处理器。某些产品中可利用外部信号唤醒 USB 主机控制器发送。

UDP 主要特性如下：

- 与 USB V2.0 完全兼容，12 Mb/s
- 内置 USB V2.0 全速收发器
- RTL 中端点数目与大小全参数化
- 端点内置双端 RAM
- 挂起/恢复逻辑
- 用于同步与批端点的 Ping-pong 模式(2 存储器组)

USB 在板子上的器件端口接口电路图如下图 2.2.7 所示：

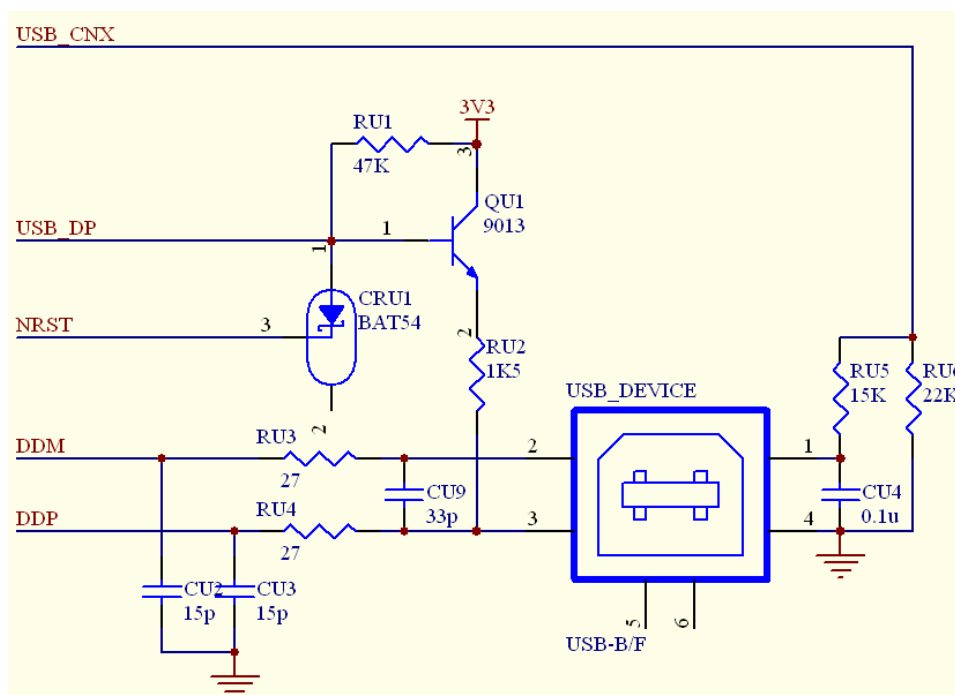


图 2.2.7 USB DEVICE 接口电路图

(3) 网络接口

以太网 MAC 是 OSI 参考模型物理层(PHY)与逻辑链路层(LLC)间 MAC 子层的硬件工具。它使用以太网 IEEE 802.3u 数据帧格式控制在主机与 PHY 层间的数据交换。以太网

MAC 包括所需逻辑与 DMA 管理的发送与接收 FIFO。此外，它通过与 MDIO/MDC 引脚连接来对 PHY 层进行管理。

以太网 MAC 根据引脚输出配置不同，可使用独立媒体接口(MII) 或简化独立媒体接口(RMII) 来传输数据。本系统设计中采用的是简化独立媒体接口(RMII)，

RMII 的目的是用缩减的引脚数来代替 IEEE 802.3u MII。它使用 2 位进行发送(ETX0 与 ETX1)，2 位进行接收(ERX0 与 ERX1)。有一个发送使能(ETXEN)，一个接收错误(ERXER)，一个采样敏感(ECRS_DV)，及一个对于 100Mb/s 数据率的 50 MHz 参考时钟(ETXCK_REFCK)。

在本系统设计中采用 DM9161 作为以太网的物理层接口。DM9161 是一款低功耗、高性能的 CMOS 芯片，支持 10M 和 100M 的以太网传输，它起编码、译码输入和输出数据的作用。它与 AT91RM9200 的引脚连接如下图 2.2.8 所示：

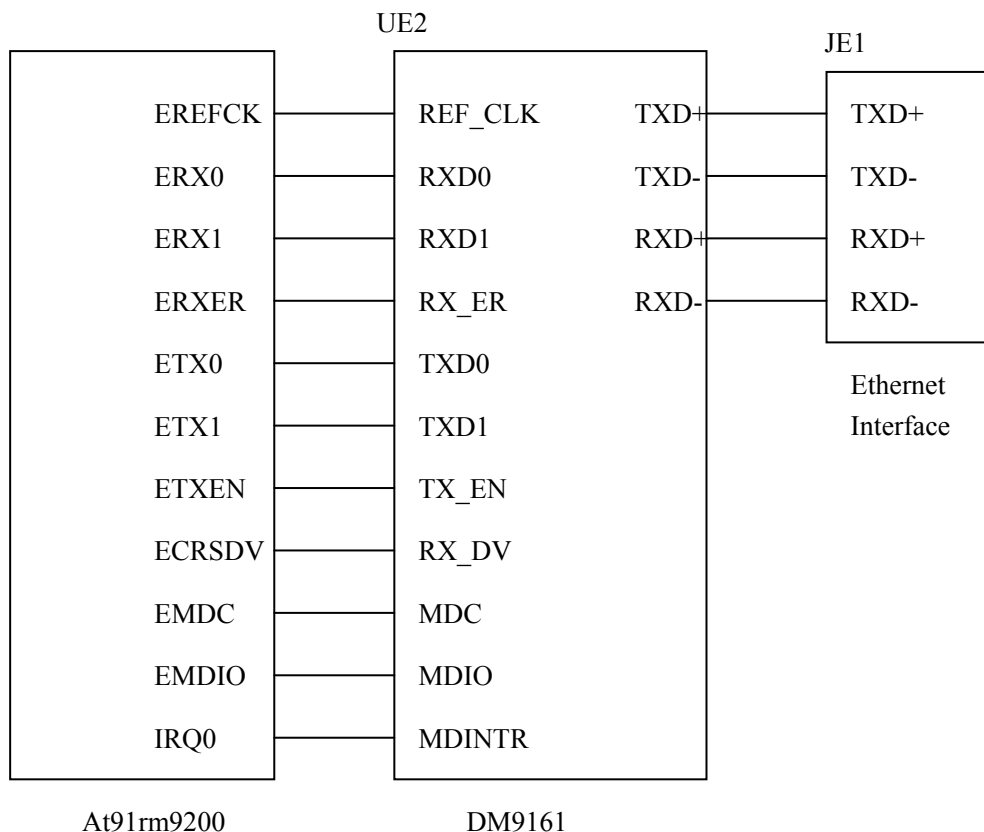


图 2.2.8 网络芯片 DM9161 与 AT91RM9200 的引脚连接电路图

(4) 串口 USART

通用同步异步收发器(USART) 提供一个全双工通用同步异步串行连接。AT91RM9200 芯片支持 4 个通用同步/异步接收/发送器(USART)，其中 USART1 为全调制解调控制线。本实验板只引出了 USART0 和 USART1 两路接口，使用 MAX3223 芯片实现 USART0 通信，使用 MAX3243 芯片实现 USART1 通信。

此外，AT91RM9200 芯片还支持专门的 UART 调试通道(DBGU)，其与 USART0 共用 MAX3223 芯片。

串口 USART0 和 DBGU UART 的接口电路图如下图 2.2.9 所示：

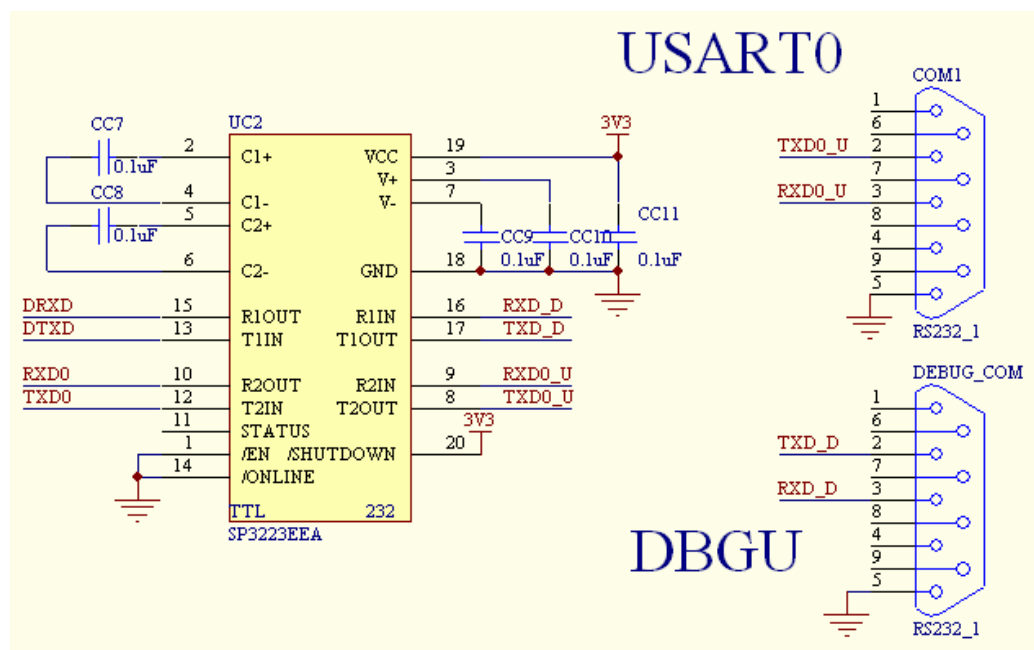


图 2.2.9 串口 USART0 和 DBGU UART 的接口电路

串口 USART1 的接口电路图如下图 2.2.10 所示：

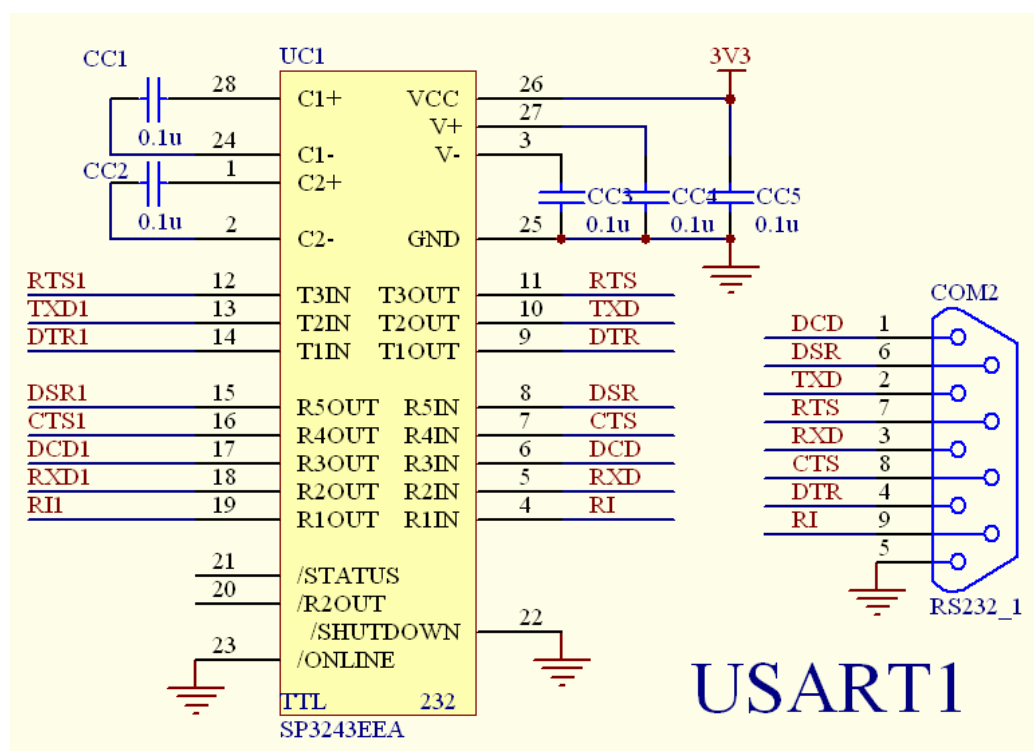


图 2.2.10 串口 USART1 的接口电路

(5) SPI 接口

AT91RM9200 提供主机/从机串行外设接口 (SPI)，支持与串行外设通讯，其相应的引脚名称和说明如下表 2.2.4 所示：板子将 SPI 相应引脚用排针 JS1 引出，如图 2.2.11 所示：

引脚名称	引脚说明	类型	
		主机	从机
MISO	主入从出	输入	输出
MOSI	主出从入	输出	输入
SPCK	串行时钟	输出	输入
NPCS1-NPCS3	外设片选	输出	未用
NPCS0/NSS	外设片选/从机选	输出	输入

表2.2.4 SPI接口引脚名称说明：

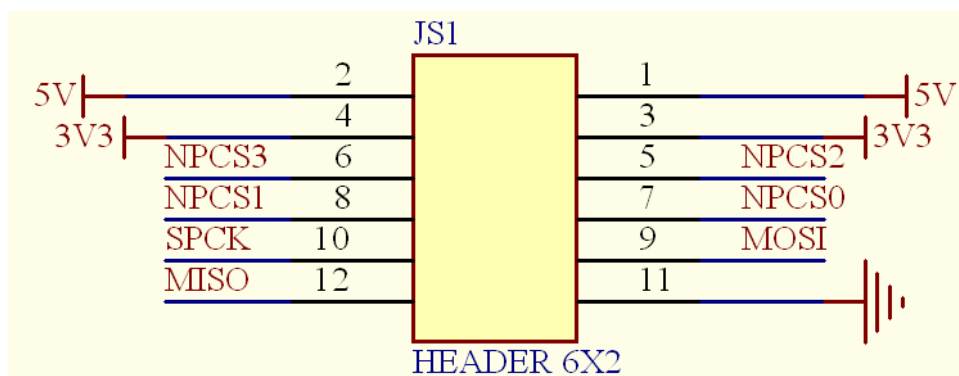


图 2.2.11 SPI 引脚接口电路

(6) SD Card 接口

AT91RM9200 芯片配备多媒体卡口，支持 SD 存储卡规范 V1.0。SD 存储卡通信基于一个 9 脚接口(时钟、命令、四数据与三电源线)，其接口电路图如下图 2.2.12 所示；表 2.2.5 为 SD Card 管脚定义情况。

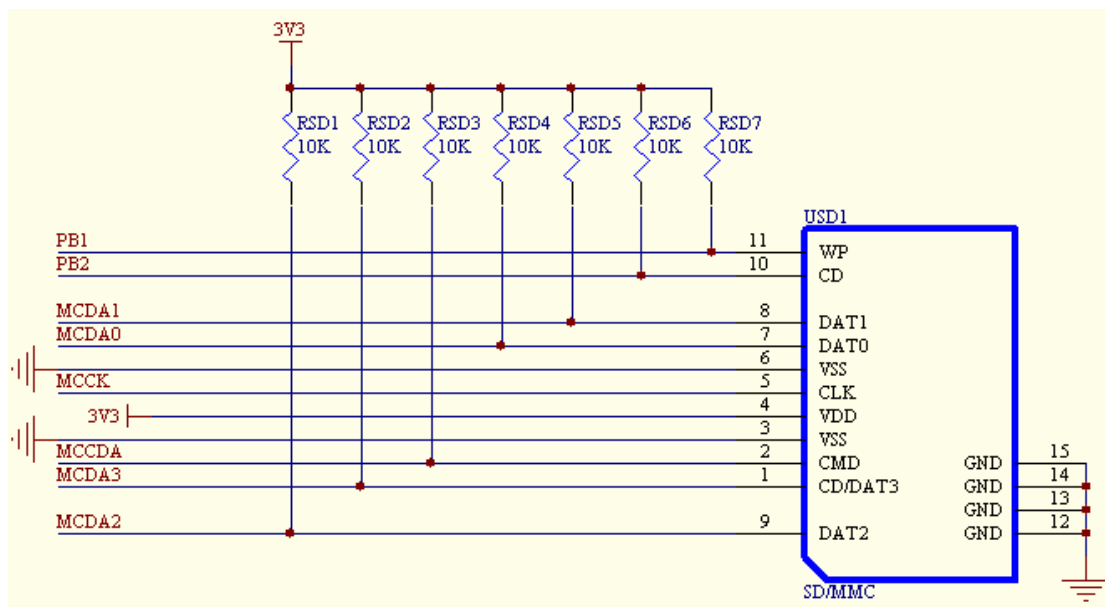


图 2.2.12 SD Card 接口电路图

管脚	描述
WP	写保护
CD	卡检测
DAT1	数据线[bit1]
DAT0	数据线[bit0]
CLK	时钟
CMD	命令/响应
CD/DAT3	数据线[bit3]
DAT2	数据线[bit2]

表 2.2.5 SD Card 管脚定义

(7) 显示接口

AT91RM9200 处理器内部没有集成 LCD 控制器，因而需要配备专用的显示控制器，才能实现 LCD 显示。本实验板我们选用了 EPSON 公司的显示控制芯片 S1D13506 用于控制 AT91RM9200 嵌入式系统中的 LCD、CRT 和 TV 的图像数据显示。

S1D13506 是 EPSON 大规模显示控制器家族中较新的一款，是以 S1D13504 为基础控制器发展起来的更多功能的 LCD 显示控制器（S1D13504 只支持 LCD 接口，S1D13504 不支持 TV 接口）。它内置 RAMDAC 硬件 X-Y 轴转置 2 维加速器，共有 114 个寄存器，可以灵活地对各种不同的显示方式进行设置，功能非常强大，可以和目前市场上流行的多种 CPU 总线兼容。

S1D13506 支持最高为 16 位数据宽度的 LCD 接口，使用 FRM 和调谐（dithering）技术，可以在 TFT LCD、CRT 和 TV 上最高显示 64K 颜色。它配置一个 16 位内存接口，可以支持最高 2MB 的 EDO-DRAM。支持 6 色，256 色和真彩色的颜色结构。对于 TV 电路，它支持 TV 的 NTSC 制式和 PAL 制式显示。在 NTSC 制式中它支持从 400×396 到 720×484 多种分辨率。

ICS1523 是一款高性能、低功耗的可编程行同步信号发生器，它带有一个 I2C 串行总线接口，可以方便地对内部寄存器进行配置，能产生用户需要的同步信号。

在这里，ICS1523 电路为 S1D13506 型显示控制器提供视频同步信号。这些信号是 S1D13506 连接 LCD 时所需要的。ICS1523 输入时钟是 50 MHz(引脚 12)、输出 CLK(25 MHz)和 CLK/2(12.5 Hz)，分别接 S1D13506 CLKI 和 CLKI2。

AT91RM9200 与 S1D13506 的电路连接原理图如下图 2.2.13 所示：

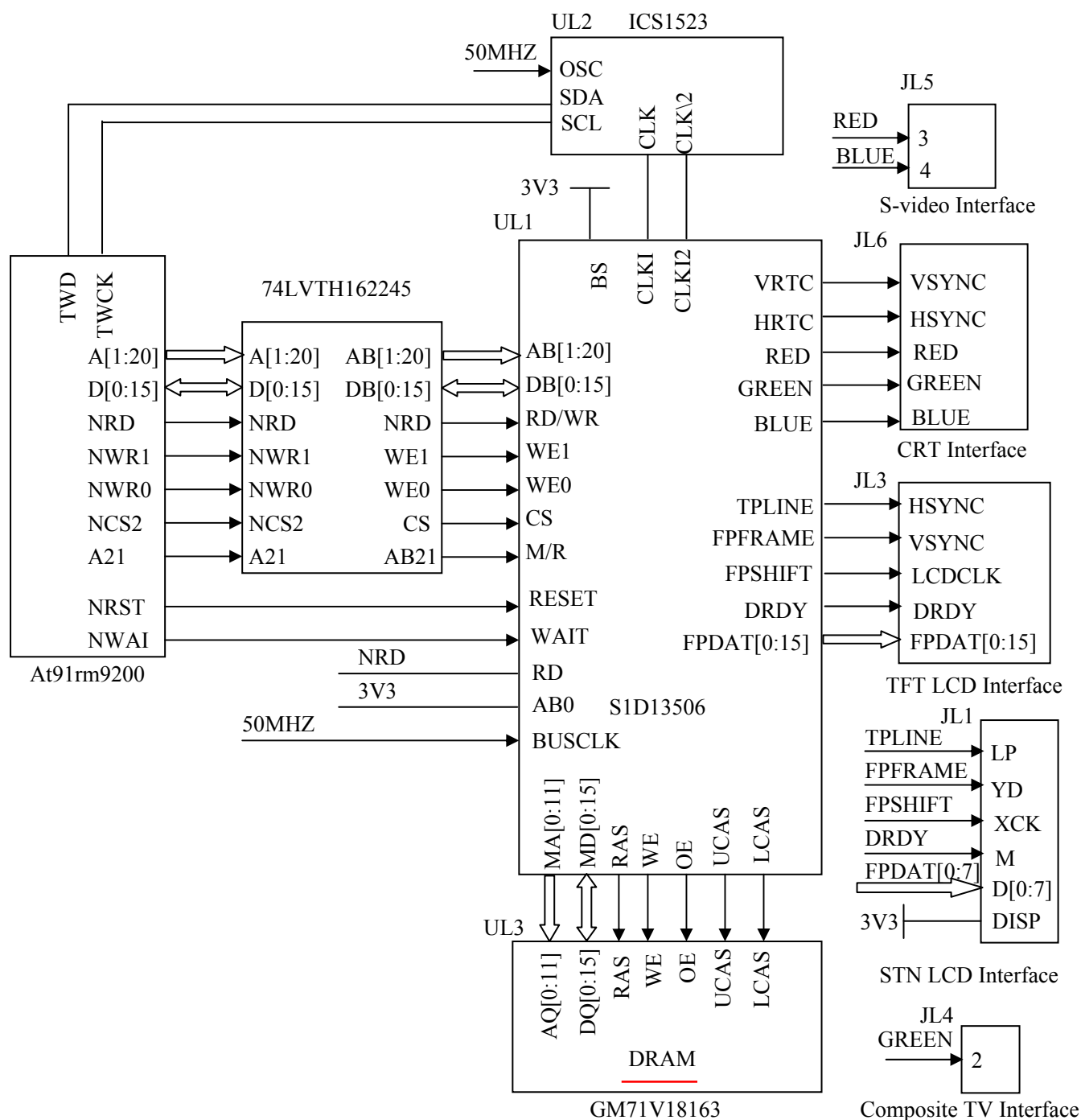


图 2.2.13 AT91RM9200 与 S1D13506 的电路连接原理图

(8) 总线扩展

本实验板中将数据总线、地址总线及一些控制信号通过 74LVTH162245 后由插针扩展出来，以方便与外部电路进行数据通讯。电路连接图如下图 2.2.14 所示：

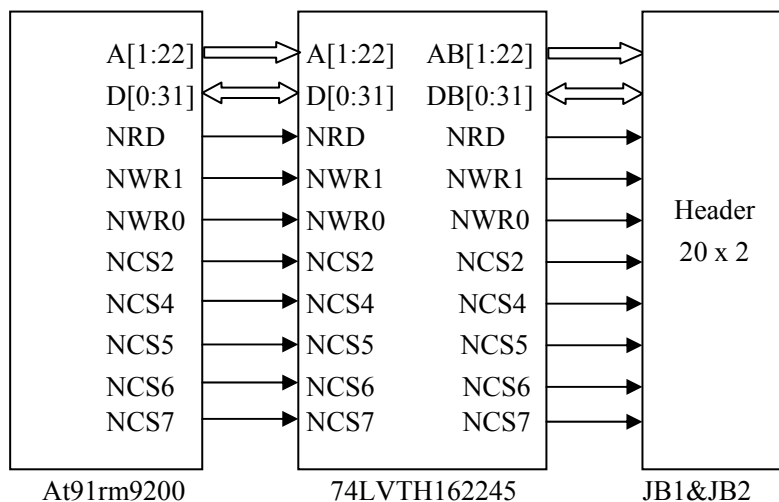


图 2.2.14 AT91RM920 总线扩展电路图

(9) 电源

UP1、UP2 为电源芯片，UP1 型号为 LM2596-5，为 12V 转 5V 芯片；UP2 的型号为 AMS1117-3.3，为 5V 转 3.3V 芯片。SWP1 为电源开关，DP2 为 12V 电源指示灯。JP1 为电源接口，外部需要为底板提供 12V 的电源。

(10) 跳线

底板上只在显示部分有两个跳线，分别为 JL7、JL8，JL7 用来选择接 5V 还是 3.3V 电源，JL8 用来选择接 5V 还是 12V 电源，在本开发平台中，JL7 接 3.3V 电源，JL8 接 12V 电源。

2.3 硬件平台使用说明

实验平台建议按如下方式配置：本实验开发板一套，一台 PC 机，JTAG 调试器一个，串行连接线一根，网线一根。使用开发板时，按下面建议进行，以保证系统的正常工作并延长其使用寿命。

(1) 上电顺序

上电时，先确认板上电源开关处于断开状态，再打开外部可调电源，确认电源电压无误后（DC12V），再打开板上电源开关；关电时，先关掉板上电源开关，再关掉外部可调电源。

(2) 接口使用说明

只有关电时才可以插拔 JTAG，UART 以及所有跳线。USB、SD 卡、网络可带电插拔。用示波器测量时注意不要造成短路。

参考资料：

1. At91rm9200 data sheet
2. 开发板原理图

第三章 构建嵌入式 Linux 系统概述

通常来说，一个可用的小型嵌入式 Linux 系统只需要具备下面几个基本元素就可以适用于一般的嵌入式环境了：

- 引导程序；
- Linux 内核，由内存管理、进程管理等构成；
- 初始化进程，与系统的硬件环境有相关。

以上三项是最为基本的部分，但只有这些是不能做任何事情的，它只是操作系统的基本框架，只是代表操作系统能够运行而已。如果要使系统能够真正有用，还必须加上：

- 设备驱动程序；
- 提供所需功能的一个或多个应用程序，用来完成系统特定应用领域的功能；
- 一般来说还需要文件系统，通常放在 Flash 中；
- 对于一个可以通过网络连接的系统而言，还需要加入 TCP/IP 网络协议栈。

具备了这些模块，整个嵌入式系统基本上就比较完整了。

3.1 Bootloader 概述

3.1.1 Bootloader 的概念

引导加载程序（Bootloader）是系统加电后运行的第一段软件代码。在我们所熟悉的 PC 结构体系中，引导加载程序一般由 BIOS 和位于硬盘 MBR 中的 OS Bootloader 一起组成。BIOS 在完成硬件检测和资源分配后，将硬盘 MBR 中的 Bootloader 读入到系统的 RAM 中，然后将控制权交给 OS Bootloader。而 Bootloader 的主要任务就是将内核映像从硬盘上读到 RAM 中，然后跳转到内核的入口点去运行，也即开始启动操作系统。

而在嵌入式系统中，也具有和 PC 结构体系中类似的引导加载程序，通过这段程序，我们可以初始化硬件设备、建立内存空间的映射图，从而将系统的软硬环境带到一个合适的状态，以便为最终调用操作系统内核准备好正确的环境。

Bootloader 的实现不仅依赖于 CPU 的体系结构，而且依赖于嵌入式系统板级设备的配置。这也就是说，每种不同的 CPU 体系结构都有不同的 Bootloader，并且即使基于同一种 CPU 而构建的两块不同的嵌入式板而言，要想让运行在一块板子上的 Bootloader 程序也能运行在另一块板子上，通常都需要修改 Bootloader 源程序。因此大多数的 Bootloader 都是为系统定制的，一般不存在通用的 Bootloader。

3.1.2 Bootloader 的存储介质

系统加电或复位后，所有的 CPU 通常都从由 CPU 制造商预先安排的地址上取指令。比如，X86 的复位向量在高地址端，ARM 处理器在复位时通常都从地址 0x00000000 取它的第一条指令。而基于 CPU 构建的嵌入式系统通常都有某种类型的固态存储介质（如：EEPROM、ROM、FLASH 等）等被映射到这个预先安排的地址上。因此 Bootloader 通常被存储在这些固态存储介质的起始地址处，以便系统加电或复位后，CPU 首先执行 Bootloader 程序。

3.1.3 Bootloader 的操作模式

大多数 Bootloader 都包含两种不同的操作模式：“启动加载”模式和“下载”模式，这种区别只有对于开发人员才有意义。但从最终用户的角度看，Bootloader 的作用永远就是用来加载操作系统，而并不存在所谓的启动加载模式与下载工作模式的区别。

启动加载模式：这种模式也称为“自主”模式，即 Bootloader 从目标机上的某个固体存储介质上将操作系统加载到 RAM 中运行，整个过程没有用户的介入。这种模式是 Bootloader 的正常工作模式，当以嵌入式产品发布的时候，Bootloader 必须工作在这种模式下。

下载模式：在这种模式下，目标机上的 Bootloader 一般通过串口或者网络连接从主机上下载文件，比如：下载内核映像或者根文件系统映像等。从主机下载的文件通常首先被 Bootloader 保存到目标机的 RAM 中；然后被 Bootloader 写到目标机上的 Flash 类固态存储介质中。Bootloader 的这种模式通常在调试内核与根文件系统时使用；此外，以后的系统更新也会使用 Bootloader 的这种工作模式。工作于这种模式下的 Bootloader 通常都会向它的终端用户提供一个简单的命令行接口。

3.1.4 Bootloader 的典型架构

虽然大多数的 Bootloader 都是为系统定制的，一般不存在通用的 Bootloader。但是从结构上分析，大多数的 Bootloader 都可以分为 Stage1 和 Stage2 两部分。Stage1 中通常包括依赖于 CPU 体系结构的代码，比如设备初始化代码等，而且一般用汇编语言来实现，以达到短小精悍的目的。而 Stage2 则通常用 C 语言来实现，这样可以实现更复杂的功能，而且代码会具有更好的可读性和可移植性。

Bootloader 的 Stage1 通常包括以下步骤：

- (1) 硬件设备初始化。
- (2) 为加载 Bootloader 的 Stage2 准备 RAM 空间。
- (3) 拷贝 Bootloader 的 Stage2 到 RAM 空间中。
- (4) 设置好堆栈。
- (5) 跳转到 Stage2 的 C 入口点。

PMC和外部存储初始化

Bootloader 的 Stage2 通常包括以下步骤：

- (1) 初始化本阶段要使用到的硬件设备。
- (2) 检测系统内存映射（memory map）。
- (3) 将 kernel 映像和根文件系统映像从 flash 上读到 RAM 空间中。
- (4) 为内核设置启动参数。
- (5) 调用内核。

3.1.5 Bootloader 与 Linux 的参数传递

内核参数链表

BootLoader 可以通过两种方法传递参数给内核，一种是旧的参数结构方式（parameter_struct），主要是 2.6 之前的内核使用的方式。另外一种就是现在的 2.6 内核在用的参数链表（tagged list）方式。这些参数主要包括，系统的根设备标志，页面大小，内存的起始地址和大小，RAMDISK 的起始地址和大小，压缩的 RAMDISK 根文件系统的起始地址和大小，内核命令参数等。

内核参数链表的格式和说明可以从内核源代码目录树中的 include/asm-arm/setup.h 中找

到，参数链表必须以 ATAG_CORE 开始，以 ATAG_NONE 结束。这里的 ATAG_CORE，ATAG_NONE 是各个参数的标记，本身是一个 32 位值，例如：ATAG_CORE=0x54410001。其它的参数标记还包括：ATAG_MEM32，ATAG_INITRD，ATAG_RAMDISK，ATAG_CMDLINE 等。每个参数标记就代表一个参数结构体，由各个参数结构体构成了参数链表。参数结构体的定义如下：

```
struct tag {
    struct tag_header hdr;
    union {
        struct tag_core      core;
        struct tag_mem32     mem;
        struct tag_videotext videotext;
        struct tag_ramdisk   ramdisk;
        struct tag_initrd    initrd;
        struct tag_serialnr   serialnr;
        struct tag_revision   revision;
        struct tag_videolfb   videolfb;
        struct tag_cmdline    cmdline;
        struct tag_acorn      acorn;
        struct tag_memclk     memclk;
    } u;
};
```

说的不明白

参数结构体包括两个部分，一个是 tag_header 结构体，一个是 u 联合体。

tag_header 结构体的定义如下：

```
struct tag_header {
    u32 size;
    u32 tag;
};
```

其中 size：表示整个 tag 结构体的大小(用字的个数来表示，而不是字节的个数)，等于 tag_header 的大小加上 u 联合体的大小，例如，参数结构体 ATAG_CORE 的 $size=(sizeof(tag->tag_header)+sizeof(tag->u.core))>>2$ ，一般通过函数 tag_size(struct * tag_xxx) 来获得每个参数结构体的 size。其中 tag：表示整个 tag 结构体的标记，如：ATAG_CORE 等。

联合体 u 包括了所有可选择的内核参数类型，包括：tag_core, tag_mem32, tag_ramdisk 等。参数结构体之间的遍历是通过函数 tag_next(struct * tag) 来实现的。本系统参数链表包括的结构体有：ATAG_CORE，ATAG_MEM，ATAG_RAMDISK，ATAG_INITRD32，ATAG_CMDLINE，ATAG_END。在整个参数链表中除了参数结构体 ATAG_CORE 和 ATAG_END 的位置固定以外，其他参数结构体的顺序是任意的。本 BootLoader 所传递的参数链表如下：第一个内核参数结构体，标记为 ATAG_CORE，参数类型为 tag_core。每个参数类型的定义请参考源代码文件。

tag_array 初始化为指向参数链表的第一个结构体的指针。

```
tag_array->hdr.tag=ATAG_CORE;
tag_array->hdr.size=tag_size(tag_core);
tag_array->u.core.flags=1;
tag_array->u.core.pagesize=4096;
```

自己看书

```

tag_array->u.core.rootdev=0x00100000;
tag_array=tag_next(tag_array);

tag_array->hdr.tag=ATAG_MEM;
tag_array->hdr.size=tag_size(tag_mem32);
tag_array->u.mem.size=0x04000000;
tag_array->u.mem.start=0x20000000;
tag_array=tag_next(tag_array);
.....
tag_array->hdr.tag=ATAG_NONE;
tag_array->hdr.size=0;
tag_array=tag_next(tag_array);

```

SDRAM

最后将内核参数链表复制到内核默认的物理地址 0x20000100 处。这样参数链表就建好了。

内核接收参数

下面从基于 ARM 体系结构的 zImage 映像启动来分析 Linux 内核是怎样接收 BootLoader 传递过来的内核参数，zImage 启动过程如下图所示。

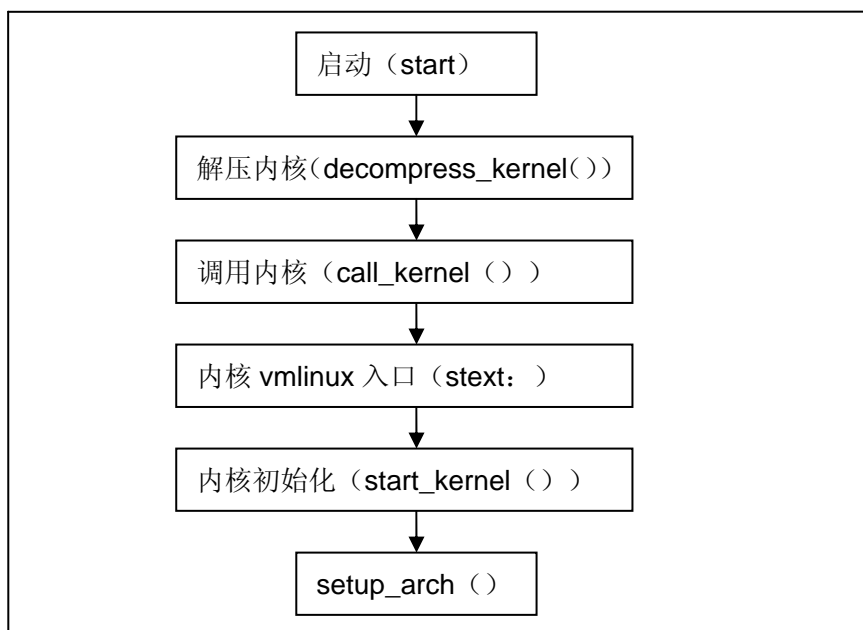


图 2 参数传递的路径[5]

在文件 arch/arm/boot/compressed/head.S 中 start 为 zImage 的起始点，部分代码如下：
start:

```

mov r7, r1
mov r8, r2
.....
mov r0, r4
mov r3, r7
bl decompress_kernel

```

```

        b    call_kernel
call_kernel:

```

```

        .....
        mov r0, #0
        mov r1, r7
        mov r2, r8
        mov pc, r4

```

首先将 BootLoader 传递过来的 r1（机器编号）、r2（参数链表的物理地址）的值保存到 r7、r8 中，再将 r7 作为参数传递给解压函数 `decompress_kernel()`。在解压函数中，再将 r7 传递给全局变量 `__machine_arch_type`。在跳到内核（vmlinux）入口之前再将 r7、r8 还原到 r1、r2 中。

在文件 `arch/arm/kernel/head.S[2]` 中，内核（vmlinux）入口的部分代码如下：

stext:

```

        mrc p15, 0, r9, c0, c0
        bl __lookup_processor_type
        .....
        bl __lookup_machine_type

```

首先从处理器内部特殊寄存器（CP15）中获得 ARM 内核的类型，从处理器内核描述符（`proc_info_list`）表（`__proc_info_begin—__proc_info_end`）中查询有无此 ARM 内核的类型，若无就出错退出。处理器内核描述符定义在 `include/asm-arm/procinfo.h` 中，具体的函数实现在 `arch/arm/mm/proc-xxx.S` 中，在编译连接过程中将各种处理器内核描述符组合成表。接着从机器描述符（`machine_desc`）表（`__mach_info_begin—__mach_info_end`）中查询有无 r1 寄存器指定的机器编号，如果没有就出错退出。机器编号 `mach_type_xxx` 在 `arch/arm/tools/mach-types` 文件中说明，每个机器描述符中包括一个唯一的机器编号，机器描述符的定义在 `include/asm-arm/mach/arch.h` 中，具体实现在 `arch/arm/mach-xxxx` 文件夹中，在编译连接过程中将基于同一种处理器的不同机器描述符组合成表。例如，基于 AT91RM9200 处理器的各种机器描述符可以参考 `arch/arm/mach-at91rm9200/board-xxx.c`，机器编号为 262 的机器描述符如下所示：

```

        MACHINE_START(AT91RM9200DK, "Atmel AT91RM9200-DK")
        /* Maintainer: SAN People/Atmel */
        .phys_io = AT91_BASE_SYS,
        .io_pg_offst = (AT91_VA_BASE_SYS >> 18) & 0xfffc,
        .boot_params = AT91_SDRAM_BASE + 0x100,
        .timer = &at91rm9200_timer,
        .map_io = dk_map_io,
        .init_irq = dk_init_irq,
        .init_machine = dk_board_init,
        MACHINE_END

```

最后就是打开 MMU，并跳转到 `init/main.c` 的 `start_kernel()` 初始化系统。

在 `init/main.c` 中，函数 `start_kernel()` 的部分代码如下：

```

{
    .....
    setup_arch();
    .....
}

```

在 `arch/arm/kernel/setup.c` 中，函数 `setup_arch()` 的部分代码如下：


```

    { .....
setup_processor();
mdesc=setup_machine(machine_arch_type);
.....
parse_tags(tags);
.....
}

```

setup_processor（）函数从处理器内核描述符表中找到匹配的描述符，并初始化一些处理器变量。setup_machine（）用机器编号（在解压函数 decompress_kernel 中被赋值）作为参数返回机器描述符。从机器描述符中获得内核参数的物理地址，赋值给 tags 变量。然后调用 parse_tags（）函数分析内核参数链表，把各个参数值传递给全局变量。这样内核就收到了 BootLoader 传递的参数。

参数传递的验证和测试

参数传递的结果可以通过内核启动的打印信息来验证。

Machine: Atmel AT91RM9200-DK

```

.....
Kernel command line: console=ttyS0,115200 root=/dev/ram rw init=/linuxrc
.....
Memory: 64MB = 64MB total
.....
checking if image is initramfs...it isn't (no cpio magic); looks like an initrd
Freeing initrd memory: 1024K
.....
RAMDISK: Compressed image found at block 0

```

3.2 Linux 内核概述

3.2.1 嵌入式操作系统必要性

以前嵌入式系统应用比较简单，不需要使用操作系统，只有循环控制，对于一些简单设备这是足够的。随着嵌入式系统硬件设计越来越复杂，提供的功能越来越强大，这时候再不引入操作系统，就很难开发应用软件。嵌入式操作系统主要用来管理系统硬件资源，同时为上层应用软件提供一个支撑平台，从而减少复杂软件的开发难度。

目前很多嵌入式产品都使用了嵌入式操作系统。嵌入式操作系统除了具有普通操作系统的功能和特征外，还有自己独特的一面。一般来说具有较小的系统内核、高效精简，以适合在有限的资源上运行；较强的实时性能，满足用户对响应时间的要求；具有可裁减性，针对具体应用能去除不必要的模块；还有很好的稳定性等。

3.2.2 嵌入式操作系统移植浅析

嵌入式操作系统的一大特点就是针对不同的硬件平台，系统不可以直接使用。它不像

Windows 那样有安装程序，将其安装到通用计算机上，便可以直接运行。嵌入式操作系统需要经过移植后才可以运行在不同的硬件平台上。使某一个平台上的代码运行在其他平台上的过程就叫移植。

在同一个硬件平台上可以嵌入不同的嵌入式操作系统，就好比 PC 既可以安装 Windows 又可以安装 Linux 一样。如果一个系统可以在不同的硬件平台上运行，那么我们就说这个系统就是可移植的。嵌入式操作系统一般需要经过针对专门平台进行改写、剪裁后，编译生成可执行映像，才能在硬件上正常工作。

3.2.3 Linux 操作系统的选择

Linux 应用于嵌入式系统的众多优点：

- **开放的源码，丰富的软件资源**

Linux 是开放源代码的自由的操作系统，它为用户提供了最大的自由度，由于嵌入式系统千差万别，往往需要针对具体的应用进行修改和优化，因而获取源代码就变得至关重要了。Linux 上的软件资源十分丰富，几乎每一种通用程序在 Linux 上都可以找到。

- **功能强大的内核，性能高效、稳定**

Linux 的内核稳定，它的高效和稳定性已经在各个领域，尤其在网络服务器领域，得到了事实的验证。

- **支持多种体系结构**

目前，Linux 能够支持 x86、ARM、MIPS、PowerPC 等多种体系结构，已经被成功移植到数十种硬件平台上，几乎支持所有流行的 CPU。

- **完善的网络系统和文件管理机制**

Linux 自诞生之日起就与网络密不可分，支持所有标准的 Internet 网络协议，并且很容易移植到嵌入式系统中。另外 Linux 还支持 ext2、fat32、Jffs2 等多种文件系统，这些都为嵌入式系统应用打下了很好的基础。

- **Linux 有着异常丰富的驱动程序资源**

Linux 源代码里包含丰富的设备驱动，支持大量的外围硬件设备，网上也有丰富的关于 Linux 的设备驱动程序，所以我们开发驱动可以有許多 DEMO 参考，不必从头开始，有些甚至可以直接使用。从而大大的缩短了开发周期。

- **可以根据需要灵活配置内核**

嵌入式系统要尽量减少体积，因为一般来说，可供嵌入式操作系统使用的存储容量都十分有限，不能像普通计算机那样采用海量存储器来进行数据存储，通常采用软件固化的方法，将程序和操作系统嵌入到整个产品里面。Linux 操作系统模块化的内核，使用户可以自己裁减内核，用户完全可以根据不同的任务来选择内核模块，而将不需要的部分去掉，减少体积，这使得 Linux 很适合嵌入系统的应用。

- **良好的开发环境，不断发展的开发工具集**

开发嵌入式系统的关键是需要有一套完善的开发和调试工具。Linux 有着非常优秀的完整开发工具链，有十几种集成开发环境，能够很方便地实现从操作系统到应用软件各个级别的调试。

3.2.4 Linux 内核作用

内核是操作系统内部核心程序，它负责管理系统的进程、内存、设备驱动程序、文件和

网络系统，决定着系统的性能和稳定性，整个系统的能力完全受内核本身能力的限制。从程序员的角度来讲，操作系统的内核提供了一个虚拟的机器接口。它抽象了许多硬件细节，程序可以以某种统一的方式来进行数据处理，而内核将所有的硬件抽象成统一的虚拟接口。倘若使用的内核无法支持目标板上的某个硬件组件，当在目标板运行此内核时，该硬件组件将会变得毫无用处。

Linux 内核必须完成下面的一些任务：

- (1) 管理存储器，为程序分配内存，并且管理虚拟内存。
- (2) 管理程序的运行，为程序分配资源，并且处理程序之间的通讯。
- (3) 对文件系统的读写进行管理，把对文件系统的操作映射为对磁盘或其他块设备的操作。
- (4) 管理输入输出，将设备映射成设备文件。
- (5) 管理网络。

Linux 内核由 5 个主要的子系统组成：内存管理程序、进程调度程序、虚拟文件系统、进程间通信、网络接口。

3.3 根文件系统概述

3.3.1 Linux 文件系统简介

文件系统是指在一个物理设备上的任何文件组织和目录，它构成了 Linux 系统上所有数据的基础，Linux 程序、库、系统文件和用户文件都驻留其中，因此，它是系统中庞大复杂且最为基本和重要的资源。Linux 系统中的文件不仅包括普通的文件和目录，每个和设备相关的实际实体也都被映射为一个设备文件，这样对用户来说，设备与普通文件并无区别，对设备的操作如同操作文件一般。文件系统的主要功能是对数据的物理存储进行管理，并向用户提供对数据的访问接口。Linux 支持多种不同的文件系统，操作系统通过虚拟文件系统 VFS 为用户、应用程序和系统的其它管理模块提供一个统一的接口，通过这些接口，它们可以访问存在于不同存储设备上的不同类型的具体的文件系统。

每个文件系统都有它自己的根目录。如果某文件系统的根目录是系统目录树的根，那么该文件系统称为根文件系统。一旦完成对根文件系统的初始化，就可以安装其他的文件系统，而其他文件系统安装到系统的目录树上。

根文件系统是 Linux 系统不可或缺的组件，在嵌入式 Linux 中，内核在系统启动期间进行的最后操作之一就是安装根文件系统，当根文件系统安装完毕后即开始初始化进程。所谓初始化进程，是一个由内核启动的用户级进程，也是系统上运行的所有其他进程的父进程，它会观察其子进程，并在需要的时候启动、停止、重新启动它们，主要用来完成系统的各项配置。

3.3.2 根文件系统目录

Linux 的文件系统是一个整体，所有的文件系统结合成一个完整的统一体，组织一个树形目录结构之中，目录是树的枝干，这些目录可能会包含其他目录，或是其他目录的“父目录”，目录树的顶端是一个单独的根目录，用/表示。

Linux 的根文件系统应该包括支持 Linux 系统正常运行的基本内容，包含着系统使用的

软件和库，以及所有用来为用户提供支持架构和用户使用的应用软件。Linux 嵌入式系统根文件系统的常用目录如下所示：

目录名	作用	是否可选
/bin	二进制可执行命令	必选
/boot	启动 Linux 时使用的一些核心文件，包括一些链接文件以及映像文件	可选
/dev	该目录下存放的是 Linux 的外部设备文件	必选
/etc	用来存放所有的系统管理所需要的配置文件和子目录	必选
/home	用户的主目录，在 Linux 中，每个用户都有一个自己的目录，一般该目录名以用户的账号命名的	可选
/lib	系统最基本的动态链接共享库，其作用类似于 Windows 里的 DLL 文件，几乎所有的应用程序都需要用到这些共享库	必选
/mnt	用户临时挂载别的文件系统	可选
/proc	一个虚拟的目录，它是系统内在的映射，我们可以通过直接访问这个目录来获取系统信息。这个目录的内容不在硬盘上而是在内存里	必选
/root	该目录为系统管理员，也称作超级权限者的用户主目录	
/tmp	这个目录是用来存放一些临时文件的	可选
/var	这个目录中存放着不断扩充着的东西，我们习惯将那些经常被除修改的目录放在这个目录下。包括各种日志文件。	可选
/usr	用户程序	可选
/opt	其它安装的软件包	可选
/sbin	系统管理员使用的系统管理程序	必选
/lost+found	一般情况下是空的，当系统非法关机后，这里就存放了一些文件	可选

从上面的表中可以看出：一般情况下 bin, dev, proc, sbin, etc, lib 目录在嵌入系统中是必选的，其它的目录都是可以根据具体的实际使用可选的。在必选的目录中，并不意味着该目录下的文件都要在嵌入式系统中保留，实际中对目录下的具体的文件也要进行选择删除处理。

第四章 嵌入式 BootLoader

本章为大家介绍 BootLoader 的功能和应用,并具体针对本实验平台所使用的 BootLoader—U-Boot,详细讲解了 U-Boot 的修改、编译与启动,并简单分析了 U-Boot 的执行代码。

平台如下:上位机系统:Redhat9.0,交叉编译环境:cross-2.95.3,U-Boot 版本:U-Boot-1.1.2

4.1 BootLoader

什么是嵌入式 BootLoader?在我们了解这个概念之前,先回忆一下 PC 的体系结构:对于计算机系统来说,从开机上电到操作系统启动需要一个引导加载程序,这个引导加载程序由 BIOS(其本质就是一段固件程序)和位于硬盘 MBR 中的引导程序一起组成。BIOS 在完成硬件检测和资源分配后,将硬盘 MBR 中的引导程序读到系统的 RAM 中,然后将控制权交给引导程序。引导程序的主要运行任务就是将内核映像从硬盘上读到 RAM 中,然后跳转到内核的入口点去运行,也即开始启动操作系统。嵌入式 Linux 系统同样离不开引导程序,这个引导程序就叫做 BootLoader。在嵌入式系统中,通常并没有像 BIOS 那样的固件程序(有的嵌入式系统也会内嵌一段短小的启动程序),因此整个系统的加载启动任务就完全由 BootLoader 来完成。

4.1.1 BootLoader 介绍

BootLoader 的功能是引导和加载内核镜像,它是芯片复位后进入操作系统之前执行的一段代码,主要用于完成由硬件启动到操作系统启动的过渡。BootLoader 首先完成系统硬件的初始化,包括时钟的设置、存储区映射、设置堆栈指针,以及完成处理器和周边电路设备正常运行所要的初始化工作,创建内核需要的信息,从而建立适当的系统软硬件条件,为最终调用操作系统内核做好准备。

对于嵌入式系统,BootLoader 是基于特定硬件平台来实现的。因此,不可能为所有的嵌入式系统建立一个通用的 BootLoader,不同的处理器架构都有不同的 BootLoader。BootLoader 不但依赖于 CPU 的体系结构,而且依赖于嵌入式系统板级设备的配置,对于两块不同的嵌入式板而言,即使它们使用同一种处理器,要想让运行在一块板子上的 BootLoader 程序也能运行在另一块板子上,一般也都需要修改 BootLoader 的源程序。

反过来,大部分 BootLoader 仍然具有很多共性,某些 BootLoader 也能够支持多种体系结构的嵌入式系统。例如,U-Boot 就同时支持 PowerPC、ARM、MIPS 和 X86 等体系结构,支持的板子有上百种。通常,它们都能够自动从存储介质上启动,都能够引导操作系统启动,并且大部分都可以支持串口和以太网接口。

4.1.2 BootLoader 的启动

Linux 系统是通过 BootLoader 引导启动的。一上电,就要执行 BootLoader 来初始化系统。系统加电或复位后,所有 CPU 都会从某个地址开始执行,这是由处理器设计决定的。比如,X86 的复位向量在高地址端,ARM 处理器在复位时从地址 0x00000000 取第一条指令。嵌入式系统的开发板都要把板上 ROM 或 Flash 映射到这个地址。因此,必须把 BootLoader 程序存储在相应的 Flash 位置,系统加电后,CPU 将首先执行它。

主机和目标机之间一般有串口可以连接,BootLoader 软件通常会通过串口来输入输出。例如,输出出错或执行结果信息到串口终端,从串口终端读取用户控制命令等。

BootLoader 的启动过程可以是单阶段的，也可以是多阶段的。通常多阶段的 BootLoader 能提供更为复杂的功能，以及更好的可移植性。从固态存储设备上（如 Flash）启动的 BootLoader 大多数是二阶段的启动过程，也即启动过程可以分为 stage 1 和 stage 2 两部分。依赖于 CPU 体系结构的代码，比如设备初始化、开关中断、初始化时钟等，通常都放在 stage 1 中，而且都用汇编来实现，以达到短小精悍的目的。而 stage 2 通常用 C 语言来实现，这样可以实现较复杂的功能，而且代码有较好的可读性和可移植性。

Boot Loader 的 stage1 通常包括以下步骤：

- (1) 初始化各种模式的堆栈和寄存器；
- (2) 初始化系统时钟、系统总线速率及一些系统常量；
- (3) 初始化存储控制器，如 FLASH 和 SDRAM 的大小、类型、数据宽度、地址范围等；
- (4) 初始化各种 I/O 口和各种控制器；
- (5) 为 stage2 做准备。

Boot Loader 的 stage2 通常包括以下步骤：

- (1) 初始化本阶段要使用到的硬件设备；
- (2) 检测系统内存映射（memory map）；
- (3) 将内核映像和根文件系统映像从 flash 拷贝 SDRAM 中；
- (4) 为内核设置启动参数；
- (5) 调用内核。

综合起来，整个 BootLoader 的实现流程如图 4.1.1 所示：

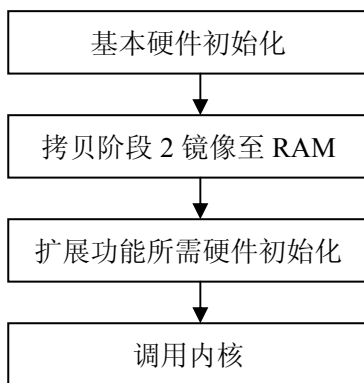


图 4.1.1 BootLoader 的实现流程图

大多数 BootLoader 都包含 2 种不同的操作模式：远程下载模式和本地加载模式。

远程下载模式下，目标机上的 BootLoader 通常通过串口或者网络连接从主机下载文件，比如：下载压缩的内核映像或者根文件系统映像等。它需要用户从控制台输入相关的命令来加载所需映像和跳转到压缩的内核映像入口。BootLoader 的这种模式通常在第一次安装内核与根文件系统时使用；此外，以后的系统更新也会使用 BootLoader 的这种工作模式。网络启动方式就是采用的远程下载模式。

本地加载模式下，BootLoader 从目标机上的某个固体存储介质上将操作系统加载到 RAM 中运行，整个过程没有用户的介入。这种模式是 BootLoader 的正常工作模式，因此当以嵌入式产品发布的时候，BootLoader 必须工作在这种模式下。本地加载模式的代表是 Flash 启动方式。

U-Boot 允许用户在这两种工作模式间进行切换。通常目标板启动时，会延时等待一段时间，如果在设定的延时时间范围内，用户没有按键，U-Boot 就进入本地加载模式。

这两种操作模式的区分仅对于开发人员才有意义，也就是不同启动方式的使用。从最终用户的角度看，BootLoader 的作用就是用来加载操作系统，而并不存在所谓的远程下载模式和本地加载模式的区分。下面详细介绍一下网络启动方式和 Flash 启动方式下的特点。

(1) 网络启动方式

这种方式开发板不需要配置较大的存储介质，但是使用这种启动方式之前，需要把

BootLoader 安装到目标板上的 EPROM 或 Flash 中。BootLoader 通过以太网接口远程下载 Linux 内核映像或者文件系统。

使用这种方式也有前提条件，就是目标板有串口、以太网接口或者其他连接方式。串口一般可以作为控制台，同时可以用来下载内核映像和 RAMDISK 文件系统。但串口通信传输速度过低，不适合用来挂接 NFS 文件系统。所以以太网接口成为通用的互连设备，一般的开发板可以配置 10M 以太网接口。

另外，还要在服务器上配置启动相关网络服务。BootLoader 下载文件一般都使用 TFTP 网络协议，还可以通过 DHCP 的方式动态配置 IP 地址。

DHCP/BOOTP 服务为 BootLoader 分配 IP 地址，配置网络参数，然后才能够支持网络传输功能。如果 BootLoader 可以直接设置网络参数，就可以不使用 DHCP。

TFTP 服务为 BootLoader 客户端提供文件下载功能，把内核映像和其他文件放在/tftpboot 目录下，这样 BootLoader 可以通过简单的 TFTP 协议远程下载内核映像到内存。如图 4.1.2 所示。

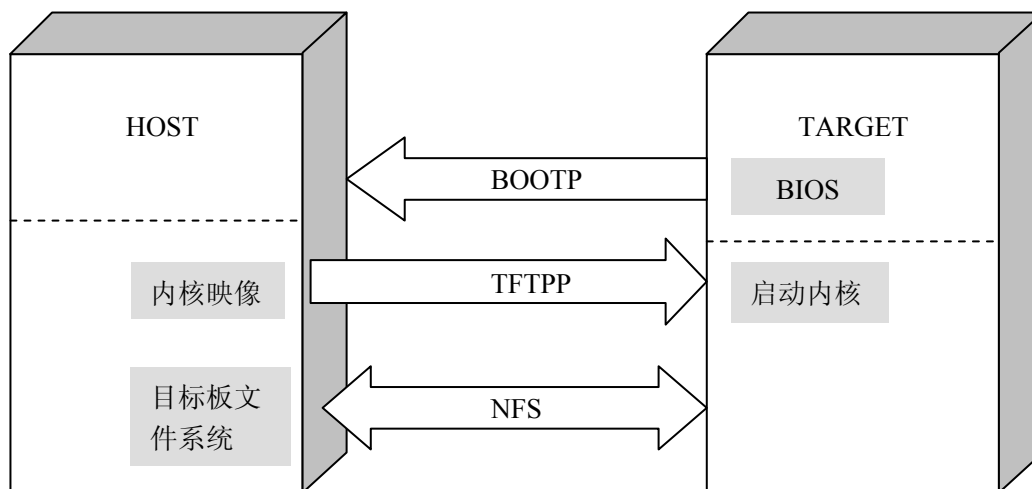


图 4.1.2 网络启动示意图

大部分引导程序都能够支持网络启动方式。U-Boot 也支持网络启动功能。

(2) Flash 启动方式

大部分嵌入式系统上都使用 Flash 存储介质。Flash 有很多类型，包括 NOR Flash、NAND Flash 和其他半导体盘。其中，NOR Flash 使用最为普遍。

NOR Flash 可以支持随机访问，所以代码是可以直接在 Flash 上执行的。BootLoader 一般是存储在 Flash 芯片上的。另外，Linux 内核映像和 RAMDISK 也可以存储在 Flash 上，通常需要把 Flash 分区使用，每个区的大小应该是 Flash 擦除块大小的整数倍。图 4.1.3 是 BootLoader 和内核映像以及文件系统的分区表。

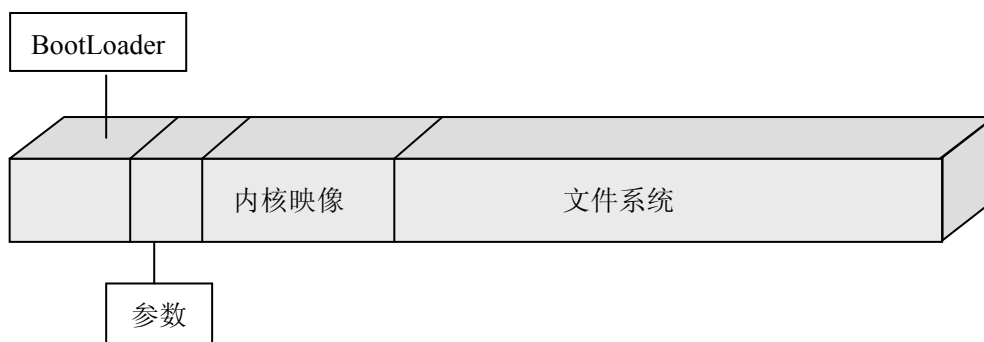


图 4.1.3 Flash 存储示意图

BootLoader 一般放在 Flash 的底端或者顶端，这要根据处理器的复位向量设置。要使 BootLoader 的入口位于处理器上电执行第一条指令的位置。

接下来分配参数区，这里可以作为 BootLoader 的参数保存区域。

再下来内核映像区，BootLoader 引导 Linux 内核，就是要从这个地方把内核映像解压到 RAM 中去，然后跳转到内核映像入口执行。

然后是文件系统区。如果使用 Ramdisk 文件系统，则需要 BootLoader 把它解压到 RAM 中。如果使用 JFFS2 文件系统，将直接挂载为根文件系统。

最后还可以分出一些数据区，这要根据实际需要和 Flash 大小来考虑了。

这些分区是开发者定义的，BootLoader 一般直接读写对应的偏移地址。到了 Linux 内核空间，可以配置成 MTD 设备来访问 Flash 分区。但是，有的 BootLoader 也支持分区的功能，例如，Redboot 可以创建 Flash 分区表，并且内核 MTD 驱动可以解析出 redboot 的分区表。

除了 NOR Flash，还有 NAND Flash、Compact Flash、DiskonChip 等。这些 Flash 具有芯片价格低，存储容量大的特点。但是这些芯片一般通过专用控制器的 I/O 方式来访问，不能随机访问，因此引导方式跟 NOR Flash 也不同。在这些芯片上，需要配置专用的引导程序。通常，这种引导程序起始的一段代码就把整个引导程序复制到 RAM 中运行，从而实现自举启动。

4.1.3 BootLoader 的种类

嵌入式世界已经有各种各样的 BootLoader，种类划分也有多种方式。除了按照处理器体系结构不同划分以外，还有功能复杂程度的不同。

首先区分一下“BootLoader”和“Monitor”的概念。严格来说，“BootLoader”只是引导设备并且执行主程序的固件；而“Monitor”还提供了更多的命令行接口，可以进行调试、读写内存、烧写 Flash、配置环境变量等。“Monitor”在嵌入式系统开发过程中可以提供很好的调试功能，开发完成以后，就完全设置成了一个“BootLoader”。所以，习惯上大家把它们统称为 BootLoader。

表 4.1.1 列出了 Linux 的开放源码引导程序及其支持的体系结构。表中给出了 X86、ARM、PowerPC 体系结构的常用引导程序，并且注明了每一种引导程序是不是“Monitor”。

Bootloader	Monitor	描述	X86	ARM	PowerPC
LILO	否	Linux 磁盘引导程序	是	否	否
GRUB	否	GNU 的 LILO 替代程序	是	否	否
Loadlin	否	从 DOS 引导 Linux	是	否	否
ROLO	否	从 ROM 引导 Linux 而不需要 BIOS	是	否	否
Etherboot	否	通过以太网卡启动 Linux 系统的固件	是	否	否
LinuxBIOS	否	完全替代 BIOS 的 Linux 引导程序	是	否	否
BLOB	否	LART 等硬件平台的引导程序	否	是	否
U-boot	是	通用引导程序	是	是	是
RedBoot	是	基于 eCOS 的引导程序	是	是	是

表 4.1.1 开放源码的 Linux 引导程序

对于每种体系结构，都有一系列开放源码 BootLoader 可以选用。这里主要了解一下 ARM 体系结构的 BootLoader。

ARM 处理器的芯片商很多，所以每种芯片的开发板都有自己的 BootLoader。结果 ARM BootLoader 也变得多种多样。最早有为 ARM720 处理器的开发板的固件，又有了 armboot，StrongARM 平台的 blob，还有 S3C2410 处理器开发板上的 vivi 等。现在 armboot 已经并入了 U-boot，所以 U-boot 也支持 ARM/XSCALE 平台，U-boot 已经成为 ARM 平台事实上的

标准 BootLoader。

4.2 U-Boot

通过本节的学习，我们希望能让大家了解如下一些事情：

- 1、什么是 U-Boot；
- 2、当我们下载一个 U-Boot 后，我们看到的是一个什么东西；
- 3、U-Boot 支持的主要功能有哪些；
- 4、U-Boot 与我们开发板所需要的具体的 BootLoader 相比，有哪些地方需要修改，而我们又该改动哪几个文件，哪几个函数；
- 5、修改后如何对 U-Boot 做编译，以产生我们所需要的 BIN 文档；
- 6、如何将 U-Boot 烧写到开发板的 Flash 中，运行后我们可以通过串口终端看到哪些信息；
- 7、U-Boot 的一些常用命令介绍。通过这些命令，我们可以对开发板进行调试。
- 8、U-Boot 源码中主要程序分析；
- 9、U-Boot 与内核有什么关系，它如何调用内核

4.2.1 U-Boot 介绍

U-Boot，全称 Universal Boot Loader，是遵循 GPL 条款的开放源码项目。从 FADSROM、8xxROM、PPCBOOT 逐步发展演化而来。其源码目录、编译形式与 Linux 内核很相似，事实上，不少 U-Boot 源码就是相应的 Linux 内核源程序的简化，尤其是一些设备的驱动程序，这从 U-Boot 源码的注释中能体现这一点。但是 U-Boot 不仅仅支持嵌入式 Linux 系统的引导，当前，它还支持 NetBSD, VxWorks, QNX, RTEMS, ARTOS, LynxOS 嵌入式操作系统。其目前要支持的目标操作系统是 OpenBSD, NetBSD, FreeBSD, 4.4BSD, Linux, SVR4, Esix, Solaris, Irix, SCO, Dell, NCR, VxWorks, LynxOS, pSOS, QNX, RTEMS, ARTOS。这是 U-Boot 中 Universal 的一层含义，另外一层含义则是 U-Boot 除了支持 PowerPC 系列的处理器外，还能支持 MIPS、x86、ARM、NIOS、XScale 等诸多常用系列的处理器。这两个特点正是 U-Boot 项目的开发目标，即支持尽可能多的嵌入式处理器和嵌入式操作系统。就目前来看，U-Boot 对 PowerPC 系列处理器支持最为丰富，对 Linux 的支持最完善。其它系列的处理器和操作系统基本是在 2002 年 11 月 PPCBOOT 改名为 U-Boot 后逐步扩充的。从 PPCBOOT 向 U-Boot 的顺利过渡，很大程度上归功于 U-Boot 的维护人德国 DENX 软件工程中心 Wolfgang Denk[以下简称 W.D]本人精湛专业水平和持着不懈的努力。当前，U-Boot 项目正在他的领军之下，众多有志于开放源码 BOOT LOADER 移植工作的嵌入式开发人员正如火如荼地将各个不同系列嵌入式处理器的移植工作不断展开和深入，以支持更多的嵌入式操作系统的装载与引导。

选择 U-Boot 的理由：

- ① 开放源码；
- ② 支持多种嵌入式操作系统内核，如 Linux、NetBSD, VxWorks, QNX, RTEMS, ARTOS, LynxOS；
- ③ 支持多个处理器系列，如 PowerPC、ARM、x86、MIPS、XScale；
- ④ 较高的可靠性和稳定性；
- ⑤ 高度灵活的功能设置，适合 U-Boot 调试、操作系统不同引导要求、产品发布等；
- ⑥ 丰富的设备驱动源码，如串口、以太网、SDRAM、FLASH、LCD、NVRAM、EEPROM、RTC、键盘等；
- ⑦ 较为丰富的开发调试文档与强大的网络技术支持；

U-Boot 软件包下载网站：<http://sourceforge.net/project/u-boot> .

4.2.2 U-Boot 主要目录结构

首先我们需要从网站上下载得到 U-Boot 源码包，例如：U-Boot-1.1.2.tar.bz2，

通过解压命令：tar jxvf U-Boot-1.1.2.tar.bz2 解压后就可以得到全部 U-Boot 源程序。我们可以看到在顶层目录下有 23 个子目录，分别存放和管理不同的源程序。这些目录中所要存放的文件有其规则，可以分为 3 类。

第一类目录与处理器体系结构或者开发板硬件直接相关；

第二类目录与一些通用的函数或者驱动程序；

第三类目录是 U-Boot 的应用程序、工具或者文档。

下表 4.2.1 列出了 U-Boot 顶层目录下主要目录及特性说明。

目录	特性	解释说明
board	平台依赖	存放电路板相关的目录文件
cpu	平台依赖	存放 CPU 相关的目录文件
Lib_arm	平台依赖	存放对 ARM 体系结构通用的文件，主要用于实现 ARM 平台通用的函数
Lib_i386	平台依赖	存放对 X86 体系通用的文件，主要用于实现 X86 平台通用的函数
Lib_ppc	平台依赖	存放对 PowerPC 体系通用的文件，主要用于实现 PowerPC 平台通用的函数
include	通用	头文件和开发板配置文件，所有开发板的配置文件都在 config 目录下
Lib_generic	通用	通用库函数的实现
Net	通用	存放网络的程序
Fs	通用	存放文件系统的程序
Post	通用	存放上电自检程序
drivers	通用	通用的设备驱动程序，主要有以太网接口的驱动
Disk	通用	硬盘接口程序
Rtc	通用	RTC 的驱动程序
Dtt	通用	数字温度测量器或者传感器的驱动
examples	应用例程	一些独立运行的应用程序的例子，例如 helloworld
tools	工具	存放制作 S-Record 或者 U-Boot 格式的映像等工具，例如 mkimage
Doc	文档	开发使用文档

表 4.2.1 U-Boot 顶层目录下主要目录及特性说明

U-Boot 的源代码包含对几十种处理器、数百种开发板的支持。可是对于特定的开发板，配置编译过程只需要其中部分程序。这里具体以 AT91RM9200 处理器为例，具体分析 AT91RM9200 处理器和开发板所依赖的程序，以及 U-Boot 的通用函数和工具。

4.2.3 U-Boot 支持的主要功能

U-Boot 所支持的主要功能如下表 4.2.2 所示：

系统引导	支持 NFS 挂载、RAMDISK(压缩或非压缩)形式的根文件系统
	支持 NFS 挂载, 从 Flash 中引导压缩或非压缩系统内核
基本辅助功能	强大的操作系统接口功能, 可灵活设置、传递多个关键参数给操作系统, 适合系统在不同开发阶段的调试要求与产品发布, 尤其对 Linux 支持最为强劲
	支持目标板环境参数的多种存储方式, 如 Flash、NVRAM、EEPROM
	CRC32 校验, 可校验 Flash 中内核、RAMDISK 镜像文件是否完好
设备驱动	串口、SDRAM、Flash、以太网、LCD、NVRAM、EEPROM、键盘、USB、PCMCIA、PCI、RTC 等驱动支持
上电自检功能	SDRAM、Flash 大小自动检测; SDRAM 故障检测; CPU 型号
特殊功能	XIP 内核引导

表 4.2.2 U-Boot 支持的主要功能

4.2.4 U-Boot 移植需要修改的文件

虽然 U-Boot 十分强大, 它支持几十种处理器及数百种开发板, 但要想让 U-Boot 适用于我们的开发板, 还必须对 U-Boot 做出相应的修改。原因是不同的开发板上其存储芯片如 SDRAM、Nor Flash、Nand Flash 等芯片类型和容量大小可能是不一样的; 虽然 U-Boot 上提供了一些常用的芯片类型供选择, 但显然它无法包括所有的芯片类型。此外, 一些宏定义和串口打印信息也需要作相应的修改。

令人高兴的是, 这样的修改并没有太多的地方, 而且 U-Boot 提供了许多开发板模板供参考, 一般, 我们只要选取与自己的开发板硬件设置最为接近的型号模板, 在此基础上作相应的修改即可。在 u-boot-1.1.2 中, 已经支持 AT91RM9200, 所以可以选取 AT91RM9200dk 作为模板进行修改。为发便起见, 我们下面直接就在该模板下修改; 因此, 请在修改前最好保存源码备份。

从移植 U-Boot 最小要求—U-Boot 能正常启动的角度出发, 主要考虑修改如下文件:

① <目标板>.h 头文件, 如 include/configs/RPXLite.h。可以是 U-Boot 源码中已有的目标板头文件, 也可以是新命名的配置头文件; 大多数的寄存器参数都是在这—文件中设置完成的;

② <目标板>.c 文件, 如 board/RPXLite/RPXLite.c。它是 SDRAM 的驱动程序, 主要完成 SDRAM 的 UPM 表设置, 上电初始化。

③ FLASH 的驱动程序, 如 board/RPXLite/flash.c, 或 common/cfi_flash.c。可在参考已有 FLASH 驱动的基础上, 结合目标板 FLASH 数据手册, 进行适当修改;

④ 串口驱动, 如修改 cpu/mpc8xx/serial.c 串口收发器芯片使能部分。

具体到本实验室的 AT91RM9200 开发板, 主要修改以下文件:

① include/configs/AT91RM9200dk.h, 包括了 SDRAM 的一些设置和定义;

② board/AT91RM9200dk/flash.c, 该程序完成的功能包括 Flash 初始化、打印 Flash 信息、Flash 擦除和 Flash 写入等操作。可在参考已有 FLASH 驱动的基础上, 结合目标板 FLASH 数据手册, 进行适当修改;

③ include/flash.h, FLASH 程序头文件。结合目标板修改 FLASH 型号和 ID 定义。

串口设置无需修改。

在具体修改 U-Boot 程序之前, 我们先来了解一下我们的开发板的一些相关配置。

CPU: AT91RM9200

Nor Flash: SST 39VF6401B(8M BYTE) (NOTE: 实验室目前有两块核心板, 这两块核心板上 Nor Flash 的型号不一样, 另一块为 SST 39VF6401, 两者 Device ID 不一样。这里以 SST 39VF6401B 为例讲解, 因此将修改过的 U-Boot 烧写到另一块板子上, 在自检时输出信息会有问题, 但不影响 Flash 的使用。)

Nand Flash:K9F5608U0D-PCB0 (32M BYTE)(Nand Flash 暂时没有使用)

SDRAM:HY57V561620(64M BYTE)

网络芯片: DM9161E

下面我们来对 U-Boot 程序作具体的修改:

1: 打开 include/configs/AT91RM9200dk.h:

(1) 将 #undef CONFIG_BOOTBINFUNC 改为

#define CONFIG_BOOTBINFUNC

从 1.1.2 开始, u-boot 有初始化 SDRAM 并拷贝自己到 SDRAM 运行的代码, 而之前的版本就没有这个功能, 定义了 CONFIG_BOOTBINFUNC 则使能这部分代码。

CONFIG_BOOTBINFUNC 主要在 U-Boot 的第一个执行文件 cpu/ AT91RM9200/start.s 中有使用; 在本文件中定义 U-Boot、环境变量在 FLASH 中的存储地址时也有涉及。

(2) 在 #define CFG_MALLOC_LEN (CFG_ENV_SIZE + 128*1024) 前面一行添加

#define CFG_MONITOR_LEN (256*1024) /* Reserve 256 kB for Monitor */

(3) 将 #define PHYS_SDRAM_SIZE 0x2000000 /* 32 megs */ 改为

#define PHYS_SDRAM_SIZE 0x4000000 /* 64 megs */

本目标板中 SDRAM 的大小为 64M BYTE;

(4) 将 #define PHYS_FLASH_SIZE 0x200000 /* 2 megs main flash */ 改为

#define PHYS_FLASH_SIZE 0x800000 /* 8 megs main flash */

本目标板中 Nor Flash 的大小为 8M BYTE;

(5) 将 #define CFG_MAX_FLASH_SECT 256 改为

#define CFG_MAX_FLASH_SECT 2048

SST39VF6401B 的扇区数为 2048 个;

(6) 将 #define CFG_PROMPT "U-Boot> " 改为

#define CFG_PROMPT "yourname_U-Boot> "

这里不改也没有什么关系, 更改主要为了识别这是你修改后的 U-Boot。

(7) 添加

#define CFG_LONGHELP 1

该处定义的作用是在 U-Boot 下输入 help 命令时, 可得到详细的命令解释。

include/configs/AT91RM9200dk.h 修改完毕, 保存退出。

2: 打开 include/flash.h:

(1) 在 #define SST_ID_xF6401 0x236B236B /* 39xF6401 ID (64M = 4M x 16) */

下添加如下行:

#define SST_ID_xF6401B 0x236D236D /* 39xF6401B ID (64M = 4M x 16) */

(如果 FLASH 型号为 SST39VF6401, 则无需添加这一句)

3: 打开 board/AT91RM9200dk/flash.c:

(1) 在 OrgDef OrgAT49BV6416[] =

```
{
    { 8, 8*1024 }, /* 8 * 8 kBytes sectors */
    { 127, 64*1024 }, /* 127 * 64 kBytes sectors */
};
```

下面添加如下几行:

OrgDef OrgSST39VF6401B[] =

```
{
    {2048,4*1024}, /* 2048 * 4 kBytes sectors */
};
```

此处表示 SST39VF6401B 的扇区数为 2048 个, 每个扇区的大小为 4K BYTE;该处数字来源见 SST39VF6401B 的 Datasheet.

(2) 将 void flash_identification (flash_info_t * info)函数里的如下两行 用 /* */屏蔽掉,

info->flash_id = ATM_MANUFACT & FLASH_VENDMASK;

printf ("Atmel: ");

在下面添加如下两行:

info->flash_id = SST_MANUFACT & FLASH_VENDMASK;

- ```

printf ("SST: ");
SST_MANUFACT 和 FLASH_VENDMASK 的宏定义在 include/flash.h 中:
#define SST_MANUFACT 0x00BF00BF ;该数值来源见 SST39VF6401B 的 Datasheet.
#define FLASH_VENDMASK 0xFFFF0000

```
- (3) 在 void flash\_identification (flash\_info\_t \* info)函数的最后, 即  

```

printf ("AT49BV6416 (64Mbit)\n");
 } 的后面添加如下几行:
 else if ((device_code & FLASH_TYPEMASK) == (SST_ID_xF6401B &
FLASH_TYPEMASK)) {
 info->flash_id |= SST_ID_xF6401B & FLASH_TYPEMASK;
 printf ("SST39VF6401B (64Mbit)\n");
 }

```
- (4) 在 ulong flash\_init (void)函数中  

```

else {
 flash_nb_blocks = 0;
 pOrgDef = OrgAT49BV16x4;
} 的前面添加如下几行:
else if ((flash_info[i].flash_id & FLASH_TYPEMASK) ==
(SST_ID_xF6401B & FLASH_TYPEMASK)) { /* SST39VF6401B Flash */
 pOrgDef = OrgSST39VF6401B;
 flash_nb_blocks = sizeof (OrgSST39VF6401B) / sizeof (OrgDef);
}

```
- (5) 在 void flash\_print\_info (flash\_info\_t \* info)函数中  

```

default:
printf ("Unknown Vendor ");
break;

```

这几句的前面添加如下几句:  

```

case (SST_MANUFACT & FLASH_VENDMASK):
 printf ("SST: ");
 break;

```
- (7) 在 void flash\_print\_info (flash\_info\_t \* info)函数中  

```

default:
 printf ("Unknown Chip Type\n");
 goto Done;
 break;

```

这几句的前面添加如下几句:  

```

case (SST_ID_xF6401B & FLASH_TYPEMASK):
 printf ("SST39VF6401B (64Mbit)\n");
 break;

```
- (8) 在 int flash\_erase (flash\_info\_t \* info, int s\_first, int s\_last)函数中, 将以下几句用/\* \*/屏蔽掉:  

```

if ((info->flash_id & FLASH_VENDMASK) !=
(ATM_MANUFACT & FLASH_VENDMASK)) {
 return ERR_UNKNOWN_FLASH_VENDOR;
}

```

并在下面添加如下几行:  

```

if ((info->flash_id & FLASH_VENDMASK) !=
(SST_MANUFACT & FLASH_VENDMASK)) {
 return ERR_UNKNOWN_FLASH_VENDOR;
}

```

至此, UBOOT 修改完毕。
- 如果核心板上的 FLASH 型号为 SST 39VF6401, 只需将上面修改部分中的 6401B 换成 6401 即可。比如 OrgSST39VF6401B、SST39VF6401B、SST\_ID\_xF6401B, 只要将后面的 B 去掉即可。

## 4.2.5 U-Boot 的编译

什么系统下？  
Makefile 知识？

U-BOOT 的源码是通过 GCC 和 Makefile 组织编译的。顶层目录下的 Makefile 首先可以设置开发板的定义，然后递归地调用各级子目录下的 Makefile，最后把编译过的程序链接成 U-BOOT 映像。

### (1) 顶层目录下的 Makefile

它负责 U-BOOT 整体配置编译。按照配置的顺序阅读其中关键的几行。

每一种开发板在 Makefile 都需要有板子配置的定义。例如 AT91RM9200DK 开发板的定义如下：

```
at91rm9200dk_config : unconfig
 @./mkconfig $(@:_config=) arm at91rm9200 at91rm9200dk
```

注意在 @./mkconfig \$(@:\_config=) arm at91rm9200 at91rm9200dk 前面是 Tab 来的，万万不能用空格代替，因为它是靠这个来识别命令的！

执行配置 U-BOOT 的命令 `make at91rm9200dk_config`，用过 `./mkconfig` 脚本生成 `include/config.mk` 的配置文件。文件的内容正是根据 Makefile 对开发板的配置生成的。

```
ARCH = arm
CPU = at91rm9200
BOARD = at91rm9200dk
```

上面的 `include/config.mk` 文件定义了 ARCH、CPU、BOARD 这些变量（VENDOR、SOC 这两个变量未作定义）。这样硬件平台依赖的目录文件可以根据这些定义来确定。At91rm9200dk 平台相关目录如下：

```
Board/at91rm9200dk/
Cpu/at91rm9200/
Lib_arm/
Include/asm-asm/
Include/configs/at91rm9200dk.h
```

再回到顶层目录的 Makefile 文件开始的部分，其中下列几行包含了这些变量的定义。

```
load ARCH, BOARD, and CPU configuration
include include/config.mk
export ARCH CPU BOARD VENDOR SOC
```

Makefile 的编译选项和规则在顶层目录的 `config.mk` 文件中定义。各种体系结构通用的规则直接在这个文件中定义。通过 ARCH、CPU、BOARD、SOC 等变量为不同硬件平台定义不同选项。不同体系结构的规则分别包含在 `ppc_config.mk`、`arm_config.mk`、`mips_config.mk` 等文件中。

顶层目录的 Makefile 中还要定义交叉编译器，以及编译 U-BOOT 所依赖的目标文件。

```
ifeq ($(ARCH),arm)
CROSS_COMPILE = arm-linux- //交叉编译器的前缀
endif
export CROSS_COMPILE
.....
U-Boot objects....order is important (i.e. start must be first)
OBJS = cpu/${CPU}/start.o //处理器相关的目标文件
.....
LIBS = lib_generic/libgeneric.a //定义依赖的目标，每个目录下先把目标文件连
```

//接成\*.a 文件

```
LIBS += board/$(BOARDDIR)/lib$(BOARD).a
LIBS += cpu/$(CPU)/lib$(CPU).a
LIBS += lib_$(ARCH)/lib$(ARCH).a
LIBS += fs/cramfs/libcramfs.a fs/fat/libfat.a fs/fdos/libfdos.a fs/jffs2/libjffs2.a \
 fs/reiserfs/libreiserfs.a fs/ext2/libext2fs.a
LIBS += net/libnet.a
LIBS += disk/libdisk.a
LIBS += rtc/librtc.a
LIBS += dtt/libdtt.a
LIBS += drivers/libdrivers.a
LIBS += drivers/sk98lin/libsk98lin.a
LIBS += post/libpost.a post/cpu/libcpu.a
LIBS += common/libcommon.a
```

.....

然后还有 U-BOOT 映像编译的依赖关系。

```
ALL = u-boot.srec u-boot.bin System.map
all: $(ALL)
u-boot.srec: u-boot
 $(OBJCOPY) ${OBJCFLAGS} -O srec $< $@
u-boot.bin: u-boot
 $(OBJCOPY) ${OBJCFLAGS} -O binary $< $@
u-boot.img: u-boot.bin
 ./tools/mkimage -A $(ARCH) -T firmware -C none \
 -a $(TEXT_BASE) -e 0 \
 -n $(shell sed -n -e 's/.*U_BOOT_VERSION//p' include/version.h | \
 sed -e 's/"[]*$$/ for $(BOARD) board"/') \
 -d $< $@
u-boot.dis: u-boot
 $(OBJDUMP) -d $< > $@
u-boot: depend $(SUBDIRS) $(OBJS) $(LIBS) $(LDSCRIPT)
 UNDEF_SYM=`$(OBJDUMP) -x $(LIBS) | sed -n -e
's/.*(u_boot_cmd_*)/u\1/p'|sort|uniq`; \
 $(LD) $(LDFLAGS) $$UNDEF_SYM $(OBJS) \
 --start-group $(LIBS) $(PLATFORM_LIBS) --end-group \
 -Map u-boot.map -o u-boot
```

Makefile 缺省的编译目标为 all，包括 u-boot.srec、u-boot.bin、System.map。u-boot.srec 和 u-boot.bin 又依赖于 U-Boot。U-Boot 就是通过 ld 命令按照 u-boot.map 地址表把目标文件组装成 u-boot。

其他 Makefile 内容就不再详细分析，上述代码分析应该可以为阅读代码提供了一个线索。

关于 Makefile 的介绍请参考文档 makefile.pdf。

## (2) 开发板配置头文件

除了编译过程 makefile 以外，还要在程序中为开发板定义配置选项或者参数。这个头文件是 include/configs/at91rm9200dk.h。

这个头文件中主要定义了两类变量。

一类是选项，前缀是 CONFIG\_，用来选择处理器、设备接口、命令、属性等。例如：

```
#define CONFIG_AT91RM9200DK 1
#define CONFIG_CMDLINE_TAG 1
```

另一类是参数，前缀是 CFG\_，用来定义总线频率、串口波特率、Flash 地址等参数。例如：

```
#define CFG_FLASH_BASE PHYS_FLASH_1
#define CFG_PROMPT "U-Boot> "
```

### (3) 编译结果

根据对 Makefile 的分析，编译分为 2 步，第一步配置，例如：

```
make at91rm9200dk_config
```

第二步编译，执行

```
make
```

就可以了。

注意可能 U-BOOT 文件中有一些中间文件会阻碍编译的运行，尤其是我们对编译过的 U-BOOT 修改再编译时，所以在编译前最好来个彻底清除，在 u-boot.1.1.2 目录下运行命令：

```
make distclean
```

make distclean 的作用是清除之前所编译的可执行文件、目标文件(Object Files, \*.o)以及由执行 ./configure 所产生的 Makefile 文件。

所以，编译 U-BOOT 一共就三个命令：

```
make distclean
make at91rm9200dk_config
make
```

编译完成后，可以得到 U-BOOT 各种格式的映像文件和符号表，如表 4.2.3 所示：

| 文件名称        | 说明                     |
|-------------|------------------------|
| system.map  | U-Boot 映像的符号表          |
| u-boot      | U-Boot 映像的 ELF 格式      |
| u-boot.bin  | U-Boot 映像原始的二进制格式      |
| u-boot.srec | U-Boot 映像的 S-Record 格式 |

表 4.2.3 U-Boot 编译生成的映像文件

U-Boot 的 3 种映像格式都可以烧写到 Flash 中，但需要看加载器是否识别这些格式。一般 u-boot.bin 最为常用，直接按照二进制格式下载，并且按照绝对地址烧写到 Flash 中就可以了。U-boot 和 u-boot.srec 格式映像都自带定位信息。可利用 AXD 软件将 U-boot 下载到芯片中单步执行调试。

## 4.2.6 U-Boot 的烧写

初始化后才能正常使用SDRAM

新开发的电路板没有任何程序可以执行，也就不能启动，需要先将 U-Boot 烧写到 Flash 中。

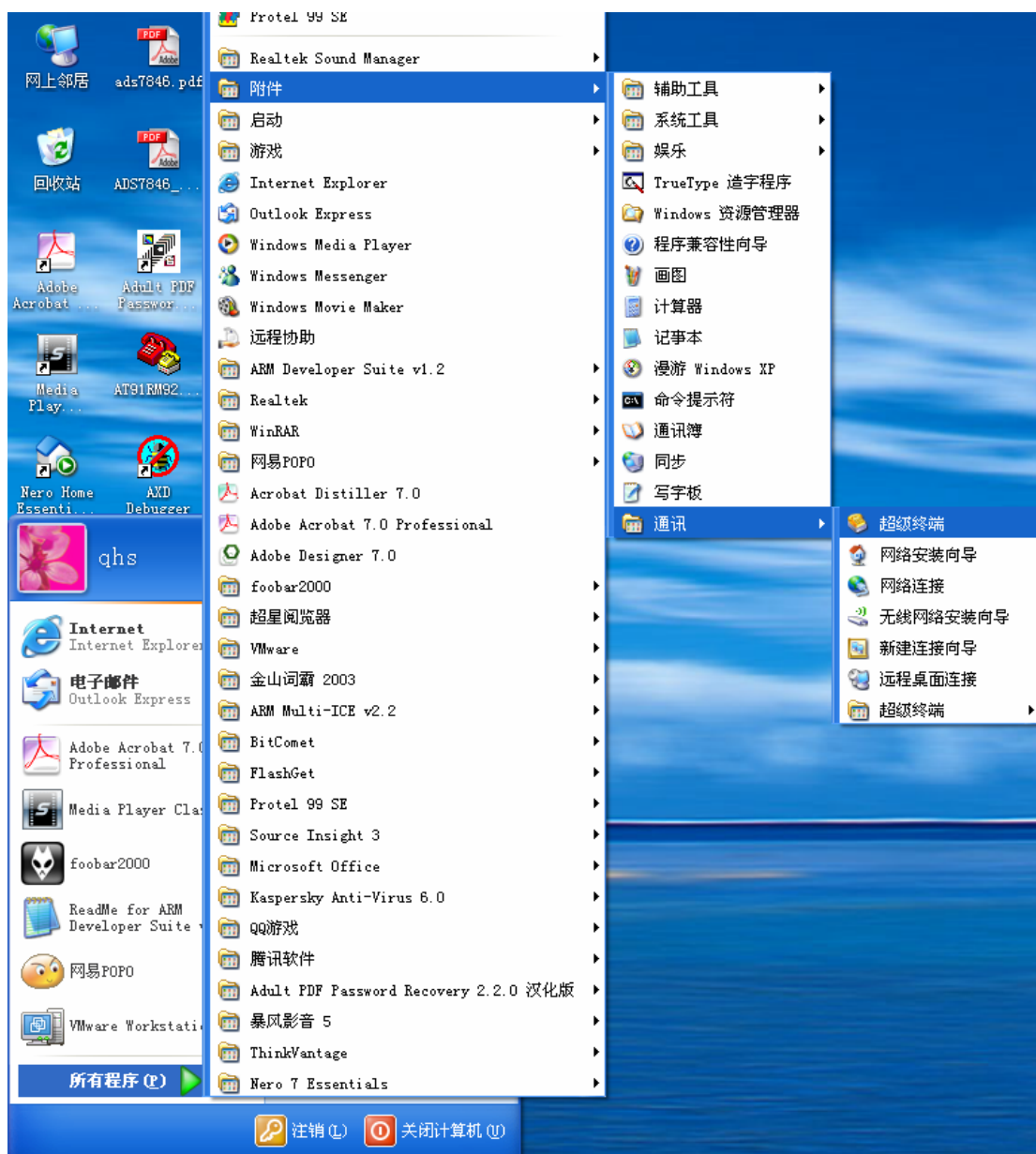
这里，我们使用 JTAG 作为调试接口，并提供烧写程序 FlashDown。我们首先运行 FlashDown 程序，对开发板进行初始化后，停止程序运行，然后将 u-boot.bin 下载到 SDRAM 中，再运行程序，将 SDRAM 中存储的 u-boot.bin 代码烧录到 Flash 中。

按照下面的步骤操作，就可将 u-boot.bin 烧写到板子上的 Flash 中，然后上电直接启动 U-BOOT。

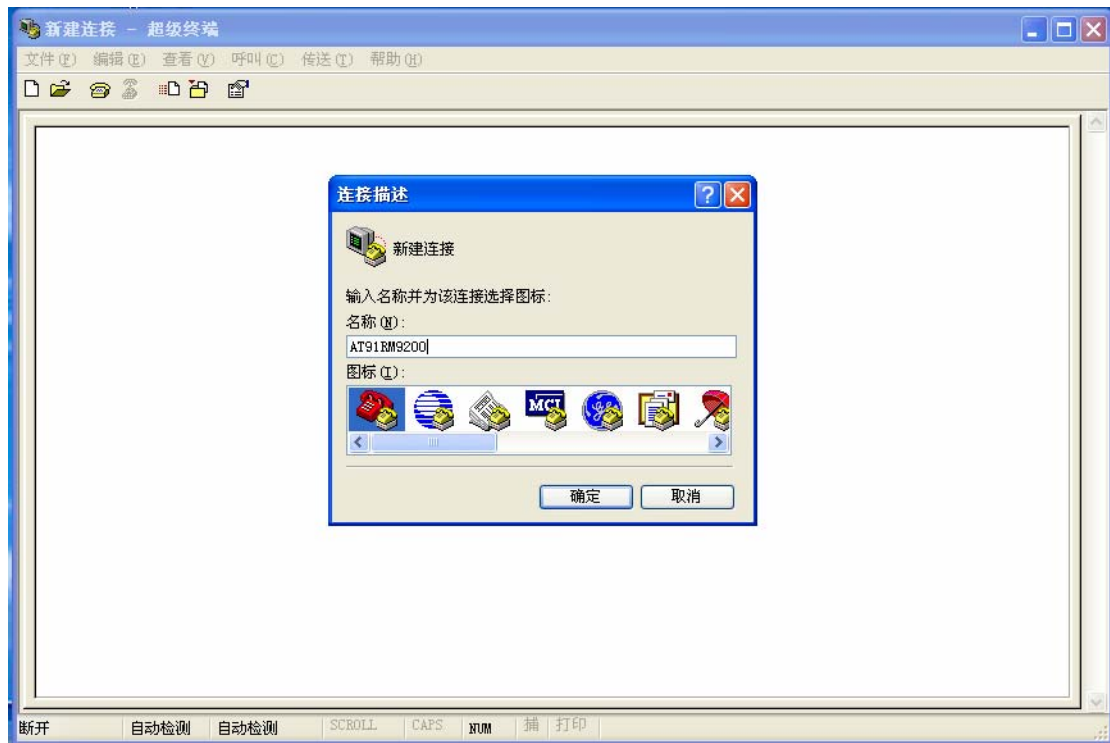
### 一：超级终端设置

首先：打开超级终端，

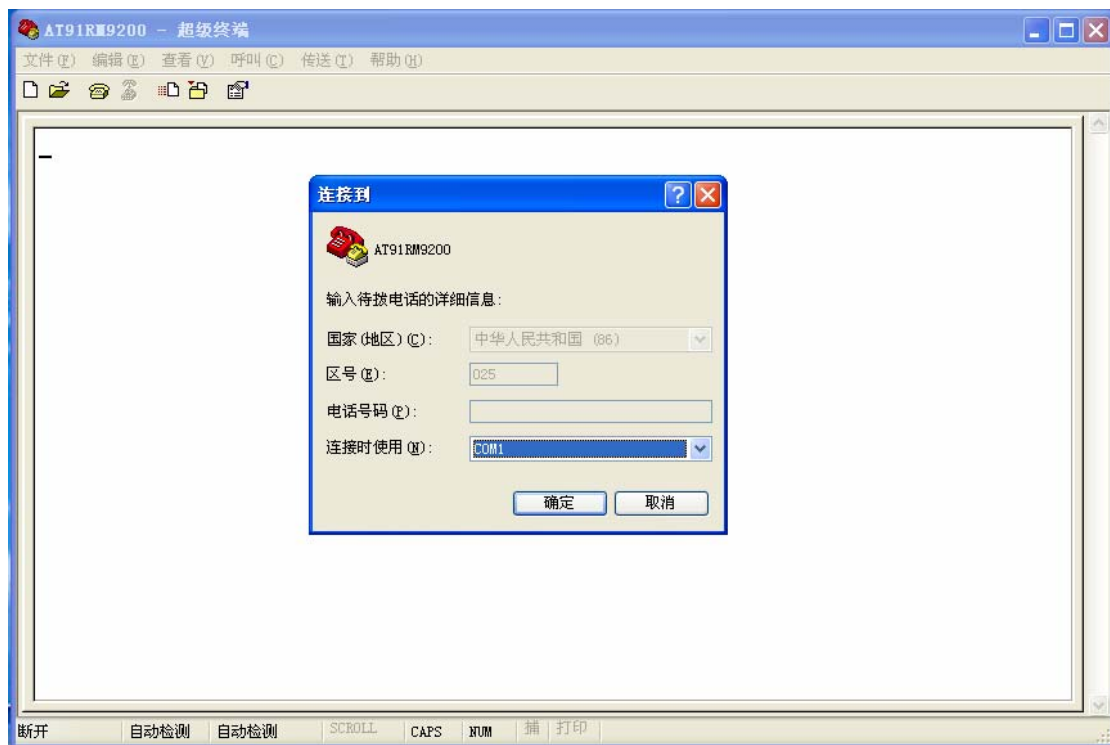




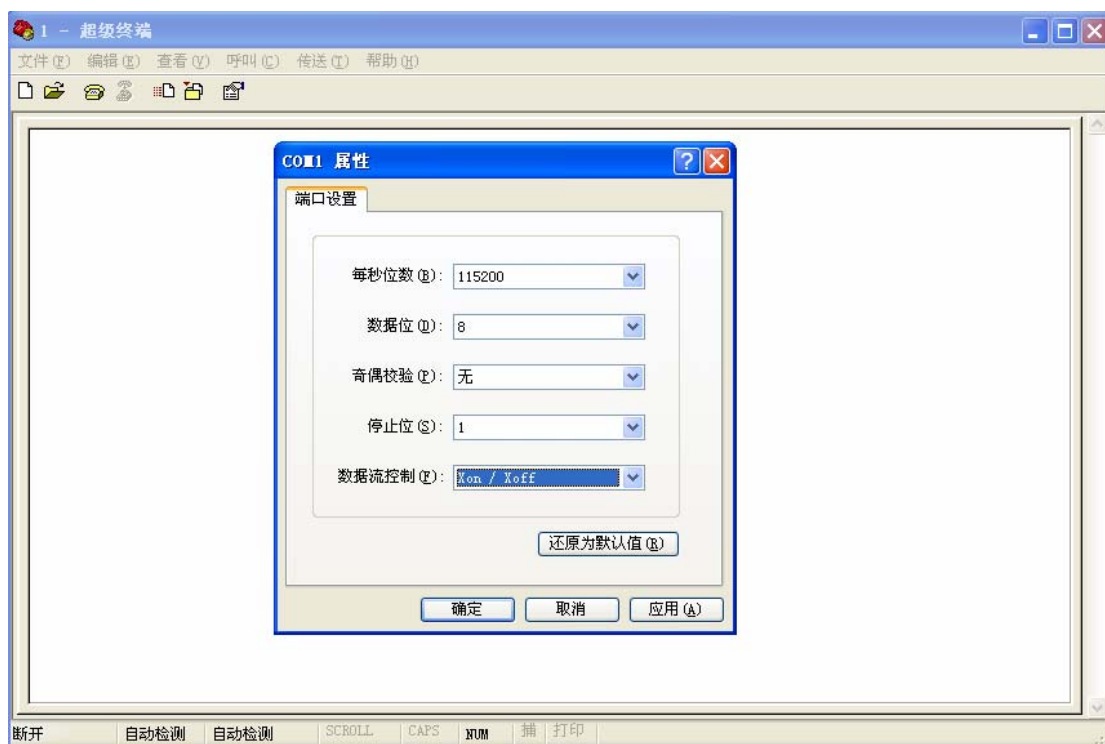
第二步：为新建超级终端输入名称，如：AT91RM9200



第三步：选择串口 COM1



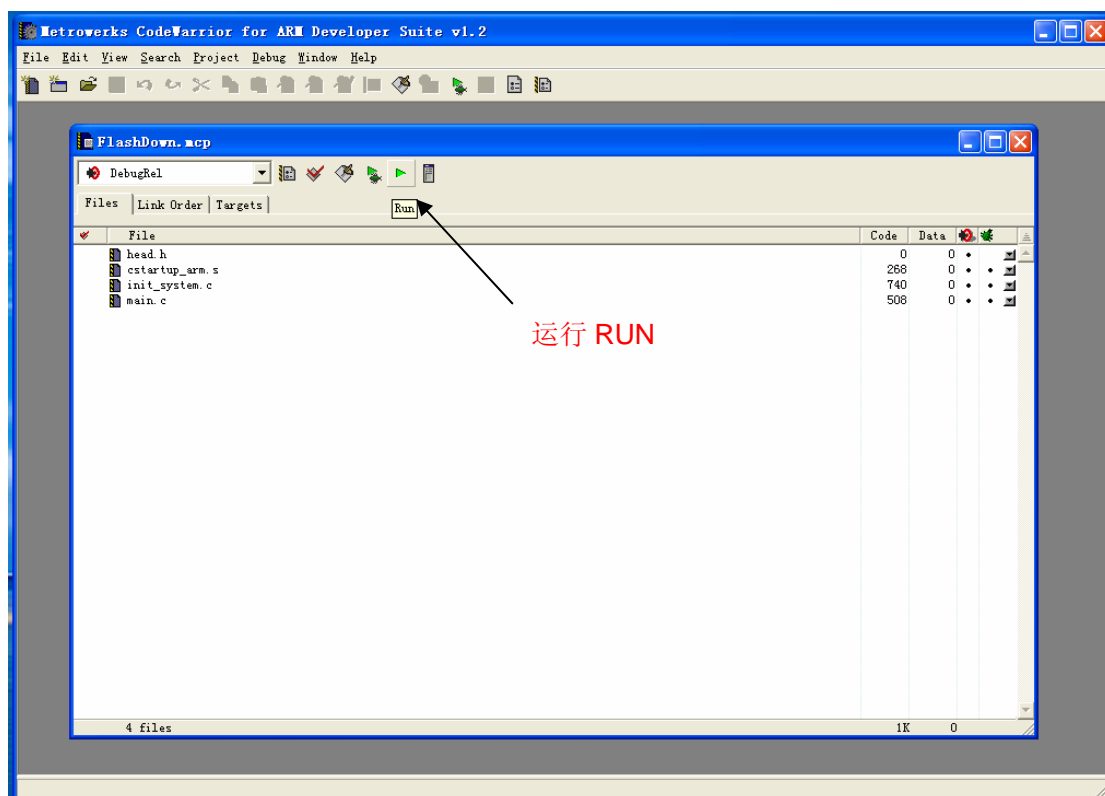
第四步：设置串口为波特率：115200，数据位：8，奇偶校验位：无，停止位：1，硬件流检测：~~Xon/Xoff~~，然后点击确定完成设置。



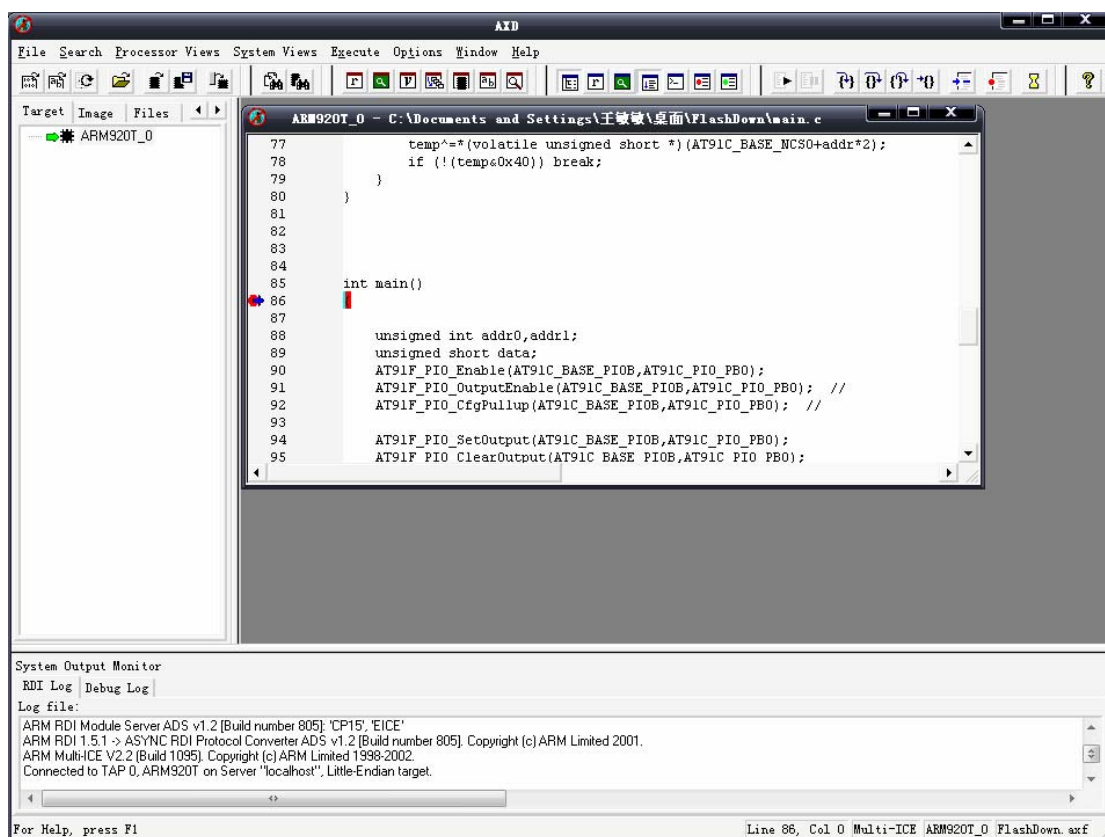
## 二：烧写 u-boot 映像文件

编译成功后得到的 u-boot.bin 二进制文件，即为需要的可执行映像文件。

第一步：在 ADS 中打开 FlashDown 程序：

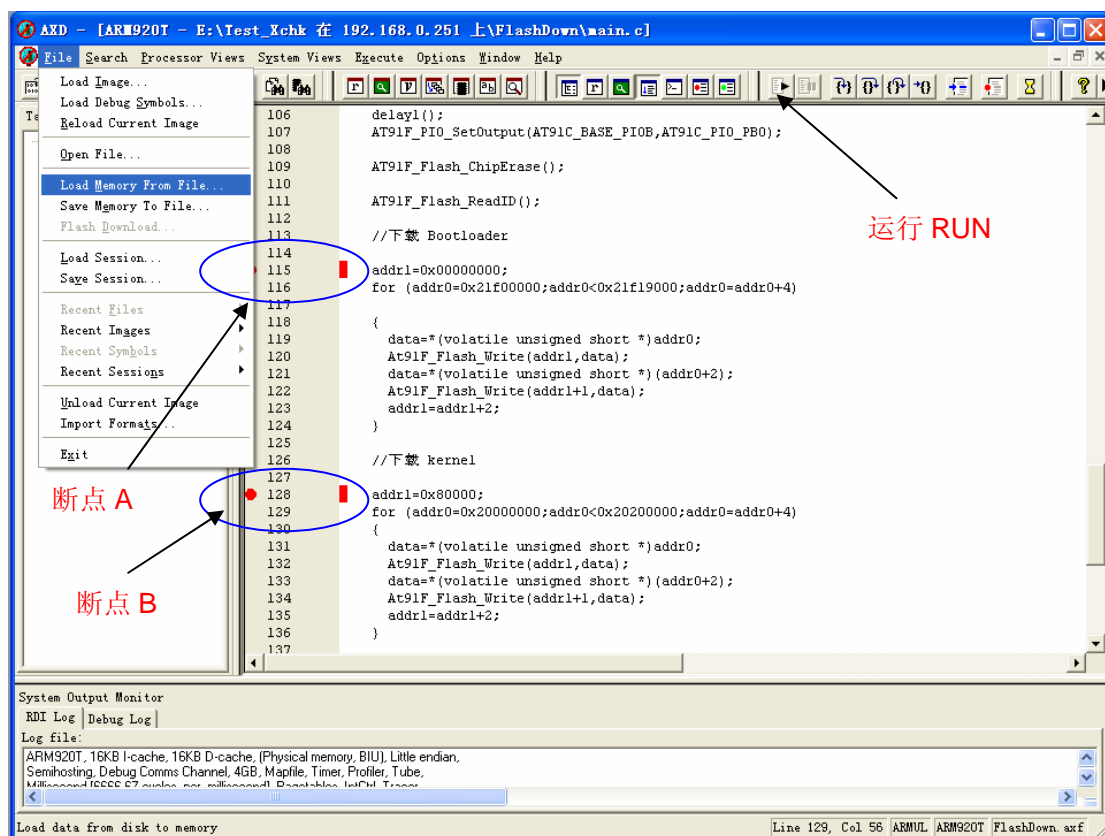


第二步：点击“RUN”按钮，进入 AXD 界面。

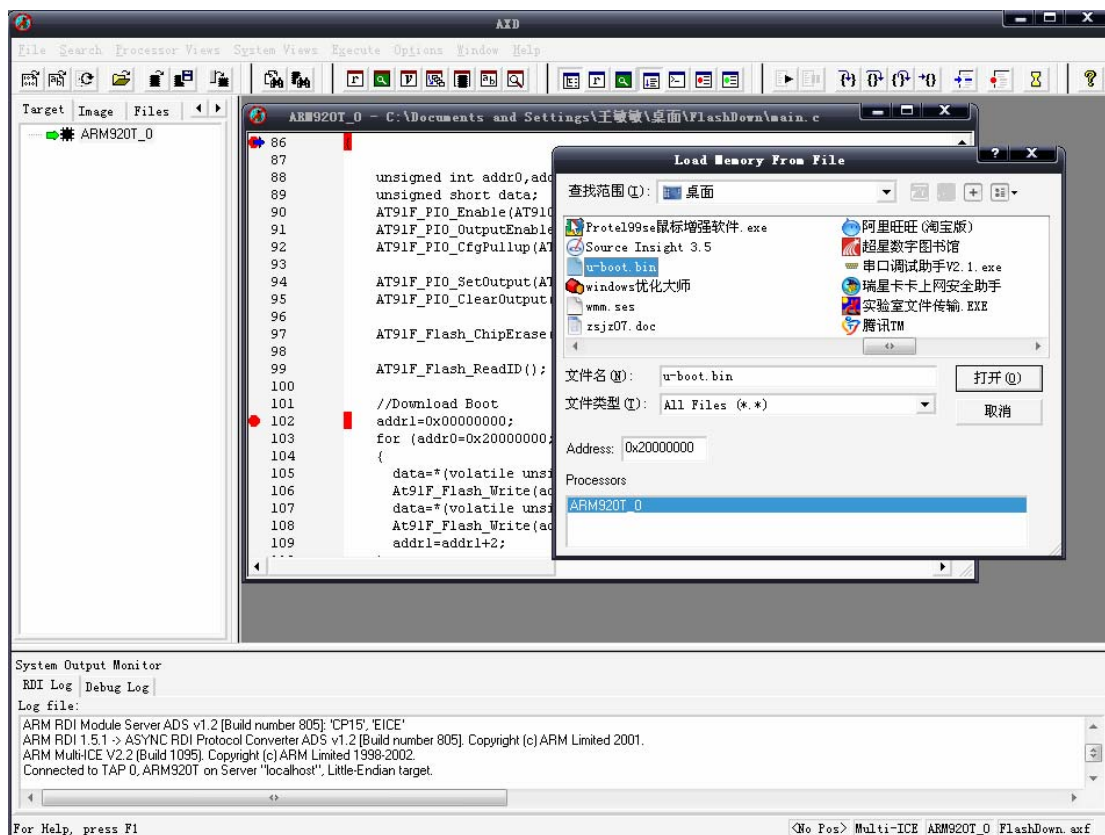


第三步：在下面两处设置断点（设置断点的方法：在行号 115、128 双击即可；若要取消，则在有断点处再双击即可），然后点击 RUN，光标将运行至断点 A 处，再点击 File 菜单选择：

“Load Memory From File...”



第四步：单击“Load Memory From File...”将出现如下对话框：

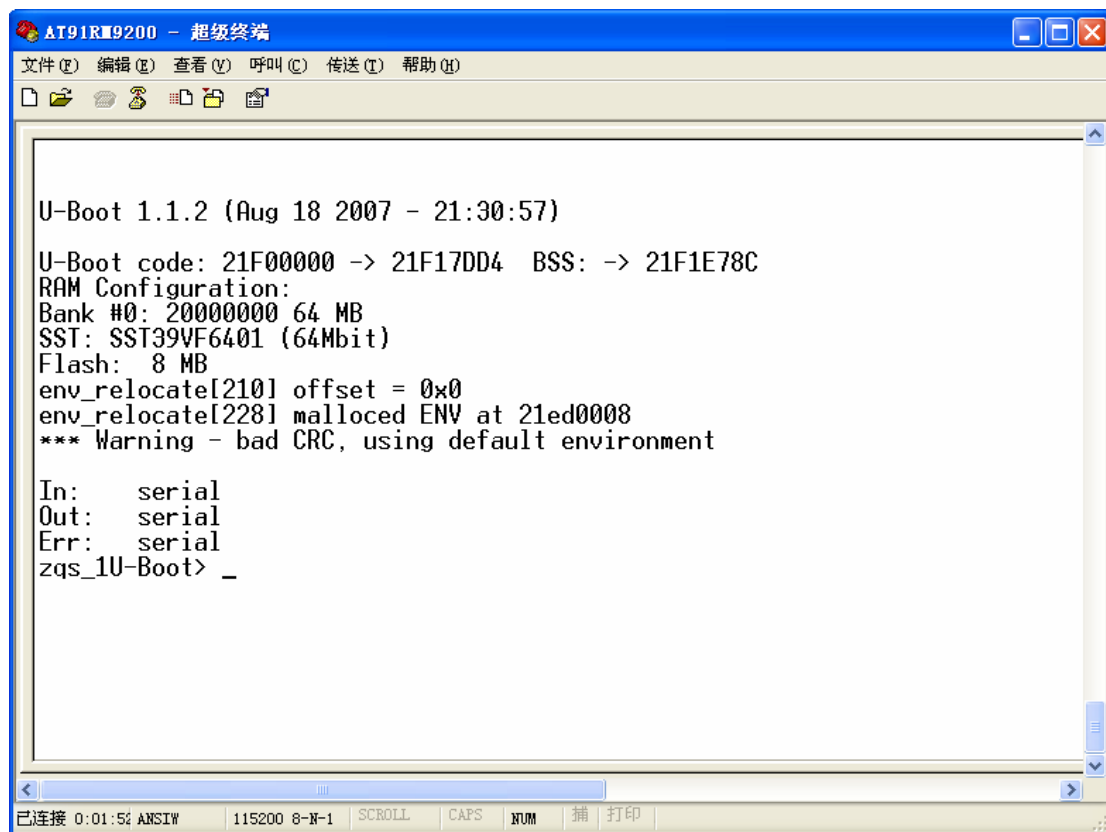


选择所要烧录的文件，即 u-boot.bin 文件（该文件存放地址因人而异，找到自己的该文

件存放目录)，然后在 Address 栏中填入所需的地址（本例中必须为 0x20000000），最后点击“打开”。

最后在调试状态中选中“运行”，等待片刻后，光标将运行至断点 B 处，此时已经将刚才 Load 的 u-boot.bin 文件从内存中烧录到 NOR FLASH 中了。

目标板重新上电复位后，U-Boot 启动成功，在超级终端里会显示如下信息，表明 U-Boot 成功启动。



```

AT91RM9200 - 超级终端
文件(F) 编辑(E) 查看(V) 呼叫(C) 传送(T) 帮助(H)

U-Boot 1.1.2 (Aug 18 2007 - 21:30:57)

U-Boot code: 21F00000 -> 21F17DD4 BSS: -> 21F1E78C
RAM Configuration:
Bank #0: 20000000 64 MB
SST: SST39VF6401 (64Mbit)
Flash: 8 MB
env_relocate[210] offset = 0x0
env_relocate[228] malloced ENV at 21ed0008
*** Warning - bad CRC, using default environment

In: serial
Out: serial
Err: serial
zqs_1U-Boot> _

```

不过此时显示信息上有一个警告，提示：“\*\*\* Warning - bad CRC, using default environment”，这是由于我们还未设置环境变量。

在提示符下输入如下命令：

**U-Boot> setenv ethaddr 12:34:56:78:90:aa** ; 设置 MAC 地址  
**U-Boot> setenv ipaddr 192.168.0.11** ; 设置 IP 地址  
**U-Boot> setenv serverip 192.168.0.xxx** ; 设置服务器地址。注：这里为你自己 PC 的 IP 地址，以后可将 PC 与开发板连接，用 tftf 协议传送数据)  
**U-Boot> setenv netmask 255.255.255.0** ; 设置以太网接口的掩码  
**U-Boot> setenv bootargs root=/dev/ram rw initrd=0x21200000, 6000000**  
**ramdisk\_size=15360 console=ttyS0,115200 mem=64M**  
 ; 设置传递给 Linux 内核的命令行参数  
**U-Boot> setenv bootcmd cp.b 10300000 21200000 300000\; bootm 10100000**  
 ; 自启动时执行的命令。cp.b COPY 根文件系统到 SDRAM 中，  
 ; bootm 用来引导内核，10100000 是内核存放在 Flash 中的地址

**U-Boot> saveenv** ; 保存环境变量

上面定义的环境变量有 ethaddr serverip ipaddr netmask bootargs bootcmd。环境变量 bootargs 中还使用了环境变量，bootargs 定义命令行参数，通过 bootm 命令传递给内核。这些设置的环境变量的用途将来 4.2.10 节解释。

此时，重启，将不会出现 Warning 提示；并且显示终端上多了一行延时等待信息：

**Hit any key to stop autoboot: 3**

在数字 3 变为 0 之前，我们按任意键可进入下载操作模式，等待我们从串口输入命令。如果在延时时间到达后，如果没有接收到相关命令，系统将自动进入本地装载模式，执行 Bootcmd 设置的命令。当然这里我们 0x10100000 里面还没有映像文件，因此是无法继续执行的，出现错误提示信息：

```
Booting image at 10100000 ...
Bad Magic Number
```

当我们将内核映像下载到 0x10100000 处时，自动进入装载模式下就会自动启动内核了，这里将在 4.2.10 节具体提到。

事实上，只要设置任意一条 setenv 命令就去掉 warning 提示，但只有设置 setenv bootcmd 命令后才会出现延时等待。我们这里设置这么多环境变量的目的是为以后使用方便。

## 4.2.7 U-Boot 的常用命令

U-Boot 是“Monitor”，除了 Bootloader 的系统引导功能，它还有用户命令接口，在命令行提示符下，可以输入 U-Boot 的命令并执行。U-Boot 可以支持几十个常用命令，通过这些命令，可以对开发板进行调试，可以引导 Linux 内核，还可以擦写 Flash 完成系统部署等功能。掌握这些命令的使用，才能够顺利地进行嵌入式系统的开发。

输入 help 命令，可以得到当前 U-Boot 的所有命令列表。每一条命令后面是简单的命令说明。

```
U-Boot> help
? - alias for 'help'
base - print or set address offset
boot - boot default, i.e., run 'bootcmd'
bootd - boot default, i.e., run 'bootcmd'
bootm - boot application image from memory
bootp - boot image via network using BootP/TFTP protocol
cmp - memory compare
coninfo - print console devices and information
cp - memory copy
crc32 - checksum calculat
dhcp - invoke DHCP client to obtain IP/boot params
echo - echo args to console
erase - erase FLASH memory
flinfo - print FLASH memory information
go - start application at address 'addr'
help - print online help
imls - list all images found in flash
itest - return true/false on integer compare
loadb - load binary file over serial line (kermit mode)
loop - infinite loop on address range
md - memory display
mm - memory modify (auto-incrementing)
mtest - simple RAM test
mw - memory write (fill)
nfs - boot image via network using NFS protocol
nm - memory modify (constant address)
printenv - print environment variables
protect - enable or disable FLASH write protection
rarpboot - boot image via network using RARP/TFTP protocol
```

```

reset - Perform RESET of the CPU
run - run commands in an environment variable
saveenv - save environment variables to persistent storage
setenv - set environment variables
tftpboot - boot image via network using TFTP protocol
version - print monitor version

```

事实上，U-Boot 支持的命令是不止这上面几十个的，上面为我们目前的 U-Boot 所支持的功能。支持命令的选项设置在 include\configs\t91rm9200dk.h 中：

```

#define CONFIG_COMMANDS \
 ((CONFIG_CMD_DFL | \
 CFG_CMD_DHCP) & \
 ~(CFG_CMD_BDI | \
 CFG_CMD_IMI | \
 CFG_CMD_AUTOSCRIP | \
 CFG_CMD_FPGA | \
 CFG_CMD_MISC | \
 CFG_CMD_LOADS))

```

而 CONFIG\_CMD\_DFL 等参数的定义在 include\cmd\_confdefs.h 中。我们可以更改 CONFIG\_COMMANDS 的选项来增加或减少我们的 U-Boot 所支持的命令的数量。

U-Boot 还提供了更加详细的命令帮助，通过 help 命令还可以查看每个命令的参数说明。由于开发过程的需要，有必要先把 U-Boot 命令的用法弄清楚。接下来，根据每一条命令的帮助信息，解释一下常用命令的功能和参数。

```

U-Boot> help bootm
bootm [addr [arg ...]]
 - boot application image stored in memory
 passing arguments 'arg ...'; when booting a Linux kernel,
 'arg' can be the address of an initrd image

```

bootm 命令可以引导启动存储在内存中的程序映像。这些内存包括 RAM 和可以永久保存的 Flash。

第 1 个参数 addr 是程序映像的地址，这个程序映像必须转换成 U-Boot 的格式。

第 2 个参数对于引导 Linux 内核有用，通常作为 U-Boot 格式的 RAMDISK 映像存储地址；也可以是传递给 Linux 内核的参数（缺省情况下传递 bootargs 环境变量给内核）。

```

U-Boot> help bootp
bootp [loadAddress] [bootfilename]

```

bootp 命令通过 bootp 请求，要求 DHCP 服务器分配 IP 地址，然后通过 TFTP 协议下载指定的文件到内存。

第 1 个参数是下载文件存放的内存地址。

第 2 个参数是要下载的文件名称，这个文件应该在开发主机上准备好。

```

U-Boot> help cp
cp [.b, .w, .l] source target count
 - copy memory

```

cp 命令可以在内存中复制数据块，包括对 Flash 的读写操作。

第 1 个参数 source 是要复制的数据块起始地址。

第 2 个参数 target 是数据块要复制到的地址。这个地址如果在 Flash 中，那么会直接调用写 Flash 的函数操作。所以 U-Boot 写 Flash 就使用这个命令，当然需要先把对应 Flash 区域擦干净。



第 3 个参数 count 是要复制的数目，根据 cp.b cp.w cp.l 分别以字节、字、长字为单位。

```
U-Boot> help echo
echo [args..]
 - echo args to console; \c suppresses newline
```

echo 命令回显参数。

```
U-Boot> help erase
erase start end
 - erase FLASH from addr 'start' to addr 'end'
erase N:SF[-SL]
 - erase sectors SF-SL in FLASH bank # N
erase bank N
 - erase FLASH bank # N
erase all
 - erase all FLASH banks
```

erase 命令可以擦 Flash。

参数必须指定 Flash 擦除的范围。

按照起始地址和结束地址，start 必须是擦除块的起始地址；end 必须是擦除末尾块的结束地址。这种方式最常用。举例说明：擦除 0x20000 – 0x3ffff 区域命令为 erase 20000 3ffff。

按照组和扇区，N 表示 Flash 的组号，SF 表示擦除起始扇区号，SL 表示擦除结束扇区号。另外，还可以擦除整个组，擦除组号为 N 的整个 Flash 组。擦除全部 Flash 只要给出一个 all 的参数即可。

```
U-Boot> help go
go addr [arg ...]
 - start application at address 'addr'
 passing 'arg' as arguments
```

go 命令可以执行应用程序。

第 1 个参数是要执行程序的入口地址。

第 2 个可选参数是传递给程序的参数，可以不用。

```
U-Boot> help mw
mw [.b, .w, .l] address value [count]
 - write memory
```

mw 命令可以按照字节、字、长字写内存，.b .w .l 的用法与 cp 命令相同。

第 1 个参数 address 是要写的内存地址。

第 2 个参数 value 是要写的值。

第 3 个可选参数 count 是要写单位值的数目。

```
U-Boot> help nm
nm [.b, .w, .l] address
 - memory modify, read and keep address
```

nm 命令可以修改内存，可以按照字节、字、长字操作。

参数 address 是要读出并且修改的内存地址。

```
U-Boot> help loadb
loadb [off] [baud]
 - load binary file over serial line with offset 'off' and baudrate 'baud'
```

loadb 命令可以通过串口线下载二进制格式文件。

**U-Boot> help loads**

**loads [ off ]**

- load S-Record file over serial line with offset 'off'

loads 命令可以通过串口线下载 S-Record 格式文件。

**U-Boot> help nfs**

**nfs [loadAddress] [host ip addr:bootfilename]**

nfs 命令可以使用 NFS 网络协议通过网络启动映像。

**U-Boot> help printenv**

**printenv**

- print values of all environment variables

**printenv name ...**

- print value of environment variable 'name'

printenv 命令打印环境变量。

可以打印全部环境变量，也可以只打印参数中列出的环境变量。

**U-Boot> help protect**

**protect on start end**

- protect Flash from addr 'start' to addr 'end'

**protect on N:SF[-SL]**

- protect sectors SF-SL in Flash bank # N

**protect on bank N**

- protect Flash bank # N

**protect on all**

- protect all Flash banks

**protect off start end**

- make Flash from addr 'start' to addr 'end' writable

**protect off N:SF[-SL]**

- make sectors SF-SL writable in Flash bank # N

**protect off bank N**

- make Flash bank # N writable

**protect off all**

- make all Flash banks writable

protect 命令是对 Flash 写保护的操作，可以使能和解除写保护。

第 1 个参数 on 代表使能写保护；off 代表解除写保护。

第 2、3 参数是指定 Flash 写保护操作范围，跟擦除的方式相同。

✗ tftpboot 命令可以使用 TFTP 协议通过网络下载文件。按照二进制文件格式下载。另外使用这个命令，必须配置好相关的环境变量。例如 serverip 和 ipaddr。

第 1 个参数 loadAddress 是下载到的内存地址。

第 2 个参数是要下载的文件名称，必须放在 TFTP 服务器相应的目录下。

**U-Boot>help run**

**run var [...]**

- run the commands in the environment variable(s) 'var'

run 命令可以执行环境变量中的命令，后面参数可以跟几个环境变量名。

**U-Boot> help setenv**

**setenv name value ...**

- set environment variable 'name' to 'value ...'  
**setenv name**  
 - delete environment variable 'name'

setenv 命令可以设置环境变量。

第 1 个参数是环境变量的名称。

第 2 个参数是要设置的值，如果没有第 2 个参数，表示删除这个环境变量。

前面 4.2.6 节的最后我们就是利用 setenv 命令设置了以太网接口的 MAC 地址、本地 IP 地址以及 TFTP 服务器端的 IP 地址等。U-Boot 的环境变量将在下一节介绍。

这些 U-Boot 命令为嵌入式系统提供了丰富的开发和调试功能。在 Linux 内核启动和调试过程中，都可以用到 U-Boot 的命令。但是一般情况下，不需要使用全部命令。比如已经支持以太网接口，可以通过 tftpboot 命令来下载文件，那么还有必要使用串口下载的 loadb 吗？反过来，如果开发板需要特殊的调试功能，也可以添加新的命令。

## 4.2.8 U-Boot 的环境变量

有点类似 Shell，U-Boot 也使用环境变量。可以通过 printenv 命令查看环境变量的设置。

```
U-Boot> printenv
bootdelay=3
baudrate=115200
ethaddr=12:34:56:78:90:aa
ipaddr=192.168.0.11
serverip=192.168.0.251
netmask=255.255.255.0
stdin=serial
stdout=serial
stderr=serial

Environment size: 160/65532 bytes
U-Boot>
```

表 4.2.4 是常用环境变量的含义解释。通过 printenv 命令可以打印出这些变量的值。

| 环 境 变 量   | 解 释 说 明              |
|-----------|----------------------|
| bootdelay | 定义执行自动启动的等候秒数        |
| baudrate  | 定义串口控制台的波特率          |
| netmask   | 定义以太网接口的掩码           |
| ethaddr   | 定义以太网接口的 MAC 地址      |
| bootfile  | 定义缺省的下载文件            |
| bootargs  | 定义传递给 Linux 内核的命令行参数 |
| bootcmd   | 定义自动启动时执行的几条命令       |
| serverip  | 定义 tftp 服务器端的 IP 地址  |
| ipaddr    | 定义本地的 IP 地址          |
| stdin     | 定义标准输入设备，一般是串口       |
| stdout    | 定义标准输出设备，一般是串口       |
| stderr    | 定义标准出错信息输出设备，一般是串口   |

表 4.2.4 U-Boot 环境变量的解释说明

环境变量的定义如下：

```
typedef struct environment_s {
 ulong crc; /* CRC32 over data bytes */
 uchar flags; /* active or obsolete */
 uchar *data;
} env_t;
```

环境变量是由一些以“0”结束的形如“name=value”的字符串所组成的序列，整个序列以两个“0”结束。环境变量存储于结构 env\_t 的 data 数组中。

有 3 处可以存放环境变量：

一是 Flash。**#define CFG\_ENV\_ADDR (PHYS\_FLASH\_1+0x60000)** 环境变量保存在 Flash 的 10060000 开始的地方；

二是 **default\_environment**。Default\_environment 是一个定义好的全局数组，定义在 common\env\_common.c 中，作用相当于 env\_t 中的 data。当没有对环境变量作设置或设置无效时，使用默认设置；

三是 SDRAM，**env\_ptr = (env\_t \*)malloc (CFG\_ENV\_SIZE)**。在 common\env\_common.c 中的 env\_relocate () 函数中，会根据 gd->env\_valid 的值将 default\_environment (gd->env\_valid = 0) 或 Flash (gd->env\_valid = 1) 中的环境变量 COPY 到 SDRAM 中，然后 gd->env\_addr = (ulong)&(env\_ptr->data)。

默认环境变量设置如下所示：

```
uchar default_environment[] = {

 #if defined(CONFIG_BOOTDELAY) && (CONFIG_BOOTDELAY >= 0) //
 "bootdelay=" MK_STR(CONFIG_BOOTDELAY) "\0"
 #endif
 #if defined(CONFIG_BAUDRATE) && (CONFIG_BAUDRATE >= 0) //
 "baudrate=" MK_STR(CONFIG_BAUDRATE) "\0"
 #endif

 "\0"
};
```

CONFIG\_BOOTDELAY 等变量的定义在 include\configs\at91rm9200dk.h 中，在我们当前的 U-Boot 中，只定义了 CONFIG\_BOOTDELAY 和 CONFIG\_BAUDRATE 这两个变量。在我们输入 setenv 命令之前，U-Boot 使用的就是默认环境变量配置。我们可以在 at91rm9200dk.h 中增加变量定义，从而添加默认环境变量配置。也可以在 U-Boot 运行结束后通过输入命令的方式来设置环境变量。

环境变量的设置命令为 setenv，在上一节有命令的解释。

## 4.2.9 U-Boot 启动过程

尽管有了调试跟踪手段，甚至也可以通过串口打印信息了，但是不一定能够判断出错原因。如果能够充分理解代码的启动流程，那么对准确地解决和分析问题很有帮助。

开发板上电后，执行 U-Boot 的第一条指令，然后顺序执行 U-Boot 启动函数。函数调用顺序如图 4.2.1 所示。

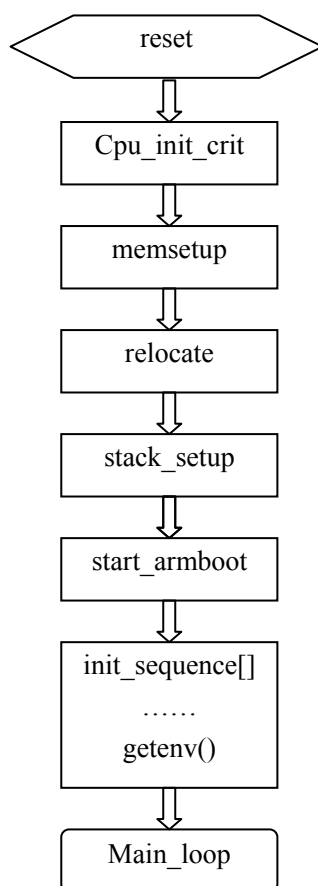


图 4.2.1 U-Boot 启动代码流程图

U-Boot 的启动过程可以分成 3 个阶段。首先在 Flash 中运行汇编程序，将 Flash 中的启动代码部分复制到 SDRAM 中，同时创造环境准备运行 C 程序；然后在 SDRAM 中执行，对硬件进行初始化；最后设置内核参数的标记列表，复制镜像文件，进入内核的入口函数。

在我们分析程序之前，先来看一下 board/at91rm920dk/u-boot.lds 这个链接脚本，通过这个脚本，我们可以知道目标程序的各部分链接顺序。Lds 文件说明如下：

**OUTPUT\_FORMAT("elf32littlearm","elf32littlearm","elf32littlearm")**

;指定输出可执行文件是 elf 格式,32 位 ARM 指令,小端

**OUTPUT\_ARCH(arm)**

;指定输出可执行文件的平台为 ARM

**ENTRY(\_start)**

;指定输出可执行文件的起始代码段为 \_start.

**SECTIONS**

{

. = 0x00000000 ; 从 0x0 位置开始

. = ALIGN(4) ; 代码以 4 字节对齐

.text : ;指定代码段

{

cpu/at91rm9200/start.o (.text) ; 代码的第一个代码部分

\*(.text) ;其它代码部分

}

. = ALIGN(4)

.rodata : { \*(.rodata) } ;指定只读数据段

. = ALIGN(4);

lds文件格式说明

```
.data : { *(.data) } ;指定读/写数据段
.= ALIGN(4);
.got : { *(.got) } ;指定 got 段, got 段式是 uboot 自定义的一个段, 非标准段
__u_boot_cmd_start = .;把__u_boot_cmd_start 赋值为当前位置, 即起始位置
.u_boot_cmd : { *(.u_boot_cmd) } ;指定 u boot cmd 段, uboot 把所有的 uboot 命令放在该段.
```

```
__u_boot_cmd_end = .;把__u_boot_cmd_end 赋值为当前位置,即结束位置
.= ALIGN(4);
__bss_start = .; 把__bss_start 赋值为当前位置,即 bss 段的开始位置
.bss : { *(.bss) } ; 指定 bss 段
__end = .; 把__end 赋值为当前位置,即 bss 段的结束位置
}
```

从 lds 文件中我们知道, 第一个要链接的是 cpu/at91rm9200/start.o, 那么 U-Boot 的入口指令一定位于这个程序中。下面详细分析一下程序跳转和函数的调用关系以及函数实现。

### 一: cpu/at91rm9200/start.S

#### (1) 定义入口

一个可执行的 Image 必须有一个入口点并且只能有一个唯一的全局入口, 通常这个入口放在 Rom(flash)的 0x0 地址。这个汇编程序就是 U-Boot 的入口程序。

```
.globl _start
_start:
```

#### (2) 设置异常向量(Exception Vector)

异常向量表, 也可称为中断向量表, 必须是从 0 地址开始, 连续的存放。如下面的就包括了复位(reset)、未定义处理(undef)、软件中断(SWI)、预取指令错误(Pabort)、数据错误(Dabort)、保留以及 IRQ、FIQ 等。

```
b reset //复位向量
 ldr pc, _undefined_instruction
 ldr pc, _software_interrupt
 ldr pc, _prefetch_abort
 ldr pc, _data_abort
 ldr pc, _not_used
 ldr pc, _irq //中断向量
 ldr pc, _fiq //中断向量
```

```
...
/* the actual reset code */
reset: //复位启动子程序
```

#### (3) 设置 CPU 为 SVC32 模式

```
mrs r0,cpsr
bic r0,r0,#0x1f
orr r0,r0,#0xd3
msr cpsr,r0
```

#### (4) 处理器时钟初始化, 设置系统主频

```
#define AT91C_BASE_CKGR 0xFFFFFC20
#define CKGR_MOR 0
/* Get the CKGR Base Address */
ldr r1, =AT91C_BASE_CKGR

/* Main oscillator Enable register APMC_MOR : Enable main oscillator ,
OSCOUNT = 0xFF */
/* ldr r0, = AT91C_CKGR_MOSCEN:OR:AT91C_CKGR_OSCOUNT */
ldr r0, =0x0000FF01
str r0, [r1, #CKGR_MOR]
```

```

/* Add loop to compensate Main Oscillator startup time */
ldr r0, =0x00000010
LoopOsc:
 subs r0, r0, #1
 bhi LoopOsc

```

## (5) 内存控制器初始化设置

```

bl lowlevelinit /* in board\at91rm9200dk\memsetup.S,对 Flash、Clock 和 Sdram 的
 寄存器作初始化设置*/
bl icache_enable; /*in cpu\at91rm9200dk\cpu.c*/
/* 这些初始化代码在系统重起的时候执行,运行时热复位从 RAM 中启动不执行 */
#ifdef CONFIG_INIT_CRITICAL
bl cpu_init_crit /*in cpu\at91rm9200dk\start.s。 do nothing */
#endif
#ifdef CONFIG_BOOTBINFUNC
/*

```

## (6) 将 Flash 中的程序复制到 RAM 中

首先利用 PC 取得 bootloader 在 flash 的起始地址,再通过标号之差计算出这个程序代码的大小。这些标号,编译器会在连接(link)的时候生成正确的分布的值。取得正确信息后,通过寄存器(r3 到 r10)做为复制的中间媒介,将代码复制到 RAM 中。

```

/*
relocate: /* 把 U-Boot 重新定位到 RAM */
 adr r0, _start /* r0 是代码的当前位置 */
 ldr r1, TEXT_BASE /* 测试判断是从 Flash 启动,还是 RAM */
 cmp r0, r1 /* 比较 r0 和 r1,调试的时候不要执行重定位 */
 beq stack_setup /* 如果 r0 等于 r1,跳过重定位代码 */
 /* 准备重新定位代码 */
 ldr r2, armboot_start
 ldr r3, bss_start
 sub r2, r3, r2 /* r2 得到 armboot 的大小 */
 add r2, r0, r2 /* r2 得到要复制代码的末尾地址 */
copy_loop: /* 重新定位代码 */
 ldmia r0!, {r3-r10} /*从源地址[r0]复制 */
 stmia r1!, {r3-r10} /* 复制到目的地址[r1] */
 cmp r0, r2 /* 复制数据块直到源数据末尾地址[r2] */
 ble copy_loop

```

## (7) 初始化堆栈等

```

stack_setup:
 ldr r0, TEXT_BASE /* 上面是 128 KiB 重定位的 u-boot */
 sub r0, r0, #CFG_MALLOC_LEN /* 向下是内存分配空间 */
 sub r0, r0, #CFG_GBL_DATA_SIZE /* 然后是 binfo 结构体地址空间 */
#ifdef CONFIG_USE_IRQ
 sub r0, r0, #(CONFIG_STACKSIZE_IRQ+CONFIG_STACKSIZE_FIQ)
#endif
 sub sp, r0, #12 /* 为 abort-stack 预留 3 个字 */
clear_bss:
 ldr r0, bss_start /* 找到 bss 段起始地址 */
 ldr r1, bss_end /* bss 段末尾地址 */
 mov r2, #0x00000000 /* 清零 */
clbss_l:str r2, [r0] /* bss 段地址空间清零循环... */
 add r0, r0, #4
 cmp r0, r1

```

**bne clbss\_1**

(8) 转到 RAM 中执行

使用指令 `ldr pc, "RAM 中 C 函数地址"` 就可以转到 RAM 中去执行。

/\* 跳转到 `start_armboot` 函数入口, `_start_armboot` 字保存函数入口指针 \*/

**ldr pc, \_start\_armboot**

**\_start\_armboot: .word start\_armboot** //start\_armboot 函数在 `lib_arm/board.c` 中实现

## 二: lib\_arm/board.c

`start_armboot` 是 U-Boot 执行的第一个 C 语言函数, 完成系统初始化工作, 进入主循环, 处理用户输入的命令。

**void start\_armboot (void)**

{

**DECLARE\_GLOBAL\_DATA\_PTR;**

/\*该句在 `include\asm-asm\global-data.h` 中定义, #define

**DECLARE GLOBAL DATA PTR** register volatile gd\_t \*gd asm ("r8"), 意思是声明一个指向 `gd_t` 结构体的指针 `gd`, 并固定使用寄存器 `r8` 来存放该指针\*/

**ulong size;**

**init\_fnc\_t \*\*init\_fnc\_ptr;**

**char \*s;**

/\* Pointer is writable since we allocated a register for it \*/

**gd = (gd\_t\*)(\_armboot\_start - CFG\_MALLOC\_LEN - sizeof(gd\_t));**

//为 `gd` 分配存储空间。`gd_t` 结构体的定义在 `include\asm-asm\global_data.h` 中

/\* compiler optimization barrier needed for GCC >= 3.4 \*/

**\_\_asm\_\_ \_\_volatile\_\_ ("": : : "memory");**

✓ /\*这行代码是内存屏障。`__asm__` 用于指示编译器在此插入汇编语句; `__volatile__` 用于告诉编译器, 严禁将此处的汇编语句与其它语句重组优化。即: 原原本本按原来的样子处理这里的汇编。`memory` 强制 `gcc` 编译器假设 RAM 所有内存单元均被汇编指令修改, 这样 `cpu` 中的 `registers` 和 `cache` 中已缓存的内存单元中的数据将作废。`cpu` 将不得不在需要的时候重新读取内存中的数据。这就阻止了 `cpu` 又将 `registers`, `cache` 中的数据用于去优化指令, 而避免去访问内存。""::表示这是个空指令。不用在此插入一条串行化汇编指令。\*/

**memset ((void\*)gd, 0, sizeof (gd\_t));**

**gd->bd = (bd\_t\*)((char\*)gd - sizeof(bd\_t));**

**memset (gd->bd, 0, sizeof (bd\_t));**

**monitor\_flash\_len = \_bss\_start - \_armboot\_start;**

/\* 顺序执行 `init_sequence` 数组中的初始化函数, 这些函数将在下面作介绍 \*/

**for (init\_fnc\_ptr = init\_sequence; \*init\_fnc\_ptr; ++init\_fnc\_ptr) {**

**if ((\*init\_fnc\_ptr)() != 0) {**

**hang ();** /\*如果出错, 输出出错信息, 提示重启\*/

**}**

**}**

/\*配置可用的 Flash \*/

**size = flash\_init ();** /\*设置 ID 号、每个分页的起始地址等信息, 将信息送到相应的结构体中; \*/

**display\_flash\_config (size);** /\*输出 Flash 大小的显示信息\*/

/\* `_armboot_start` 在特定板子的链接脚本中定义, 这里 `_armboot_start=0x21f00000`

\*/

**mem\_malloc\_init (\_armboot\_start - CFG\_MALLOC\_LEN);**

★ /\* 配置环境变量, 重新定位. 首先, 根据 `gd->env_valid` 的值来决定是使用默认的环境变量还是使用保存在 FLASH 中 `FLASH_ENV_ADDR` 处的环境变量。然后将环境变量的地址送到全局变量结构体中(`gd->env_addr = (ulong)&(env_ptr->data)`); \*/

**env\_relocate ();**

/\* 从环境变量中获取 IP 地址 \*/

语言规范?



```

gd->bd->bi_ip_addr = getenv_IPaddr ("ipaddr");
/* 以太网接口 MAC 地址 */
i = getenv_r ("ethaddr", tmp, sizeof (tmp));
s = (i > 0) ? tmp : NULL;

for (reg = 0; reg < 6; ++reg) {
 gd->bd->bi_enetaddr[reg] = s ? simple_strtoul (s, &e, 16) : 0;
 if (s)
 s = (*e) ? e + 1 : e;
}
devices_init (); /* 获取列表中的设备 */
jumpable_init ();
console_init_r (); /* 完整地初始化控制台设备 */
enable_interrupts (); /* 使能例外处理 */
/* 通过环境变量初始化 */
if ((s = getenv ("loadaddr")) != NULL) {
 load_addr = simple_strtoul (s, NULL, 16);
}
/* main_loop()总是试图自动启动, 循环不断执行 */
for (;;) {
 main_loop ();
} /* 主循环函数处理执行用户命令-- common/main.c */
/* NOTREACHED - no way out of command loop except booting */
}

```

main\_loop()函数主要用于设置延时等待, 从而确定目标板是进入下载操作模式还是装载镜像文件启动内核。在设定的延时时间范围内, 目标板将在串口等待输入命令, 当目标板接到正确的命令后, 系统进入下载模式。在延时时间到达后, 如果没有接收到相关命令, 系统将自动进入装载模式, 执行 bootm 命令, 程序进入 do\_bootm\_linux()函数, 调用内核启动函数;

注意, 只有设定了 BOOTCMD 命令时, 才会出现延时等待。

### 三: init\_sequence[]

init\_sequence[]数组保存着基本的初始化函数指针。这些函数名称和实现的程序文件在下列注释中。

```

init_fnc_t *init_sequence[] = {
 cpu_init, /* 基本的处理器相关配置,为 irq 和 fiq 分配堆栈空间, 此处, 由于未定义 CONFIG_USE_IRQ,因此 do nothing
 -- cpu/at91rm9200/cpu.c */
 board_init, /* 基本的板级相关配置, 设置 gd->have_console=1; 设置 PA23_TXD 为输出状态; 设置处理器类型
 gd->bd->bi_arch_number = 251; 设置启动参数地址
 gd->bd->bi_boot_params = 0x20000100
 -- board/at91rm9200dk/ at91rm9200dk.c */
 interrupt_init, /* 初始化例外处理 -- cpu/ at91rm9200/interrupt.c */
 env_init, /* 初始化环境变量 -- common/env_flash.c */
 init_baudrate, /* 初始化波特率设置 -- lib_arm/board.c */
 serial_init, /* 串口通讯设置,使用 DEBUG COM 作为串口通讯端口
 -- cpu/at91rm9200/serial.c */
 console_init_f, /* 控制台初始化阶段 1,设置 gd->have_console=1
 -- common/console.c */
 display_banner, /* 打印 u-boot 信息 -- lib_arm/board.c */
 dram_init, /* 设置 SDRAM 的起始地址和大小

```

```

-- board/at91rm9200dk/ at91rm9200dk.c */
display_dram_config, /* 显示 SDRAM 的起始地址和大小 -- lib_arm/board.c */
NULL,
};

```

## 4.2.10 U-Boot 与内核的关系

bootm中

除了初始化系统外，U-Boot 的功能还包括（1）将内核映像和根文件系统映像从 flash 拷贝到 SDRAM 中；（2）为内核设置启动参数；（3）调用内核。

U-Boot 作为 Bootloader,具备多种引导内核启动的方式。常用的 go 和 bootm 命令可以直接引导内核映像启动。U-Boot 与内核的关系主要是内核启动过程中参数的传递。

这一节我们将主要讲述 U-Boot 如何加载引导内核，并且将控制权交给内核。

在 4.1.2 BootLoader 的启动一节中我们曾讲到，大多数 BootLoader 都包含 2 种不同的操作模式：远程下载模式和本地加载模式。

远程下载模式下，目标机上的 BootLoader 通常通过串口或者网络连接从主机下载文件，比如：下载压缩的内核映像或者根文件系统映像等。它需要用户从控制台输入相关的命令来加载所需映像和跳转到压缩的内核映像入口。调试时一般使用这种模式，通过网络下载映像文件。

本地加载模式下，BootLoader 从目标机上的某个固体存储介质上（比如 Flash）将操作系统加载到 RAM 中运行，整个过程没有用户的介入。这种模式是 BootLoader 的正常工作模式。我们首先来看看如何将开发板设置成在这种模式下工作。

我们首先要做的是将内核映像和根文件系统映像文件烧写到 FLASH 中。

假设我们现在已经有一个内核映像文件 zImage 和一个根文件系统映像文件 myramdisk.gz（如何生成该文件见第五章和第六章，这两个文件可先从附件中得到）。

首先在我们烧写之前，必须对内核映像文件进行加头处理，在内核的前头加上 64byte 的信息，供建立 tag 之用。加这个头的原因是在下面将要使用的引导 LINUX 映像的命令 bootm 需要这里的内核映像文件必须是 U-Boot 格式的，它会在引导内核前对内核进行 CRC 验证，如果正确，才会跳到内核执行。

这个头在 include/image.h 中定义：

```

typedef struct image_header
{
 uint32_t ih_magic; /* Image Header Magic Number */
 uint32_t ih_hcrc; /* Image Header CRC Checksum */
 uint32_t ih_time; /* Image Creation Timestamp */
 uint32_t ih_size; /* Image Data Size */
 uint32_t ih_load; /* Data Load Address */
 uint32_t ih_ep; /* Entry Point Address */
 uint32_t ih_dcrc; /* Image Data CRC Checksum */
 uint8_t ih_os; /* Operating System */
 uint8_t ih_arch; /* CPU architecture */
 uint8_t ih_type; /* Image Type */
 uint8_t ih_comp; /* Compression Type */
 uint8_t ih_name[IH_NMLEN]; /* Image Name */
} image_header_t;

```

下面将讲述如何给内核映像文件加头及如何下载到 Flash 中。

### 一：使用 mkimage 生成镜像文件

加载帧头的方法是使用 U-Boot 自带的 mkimage 命令。mkimage 是 tools 目录下的 U-Boot 工具之一，它的作用就是转换 U-Boot 格式映像，也就是给映像文件加头。

具体做法如下（在 LINUX 环境下做，需要先将 U-Boot\tools\mkimage.exe COPY 到 \bin 目录下）：

```
[root@localhost tftpboot]#mkimage -n 'linux-2.6.19' -A arm -O linux -T kernel -C none -a 0x20008000 -e 0x20008000 -d zImage zImage.img
```

出现如下信息：

```
Image Name: linux-2.6.19
Created: Tue Aug 21 30:53:12 2007
Image Type: ARM Linux Kernel Image (uncompressed)
Data Size: 1055480 Bytes = 1030.74 kB = 1.01 MB
Load Address: 0x20008000
Entry Point: 0x20008000
```

然后进入到zImage  
所在的目录下，输  
入一下命令

输入 `mkimage -help` 可查看该命令的使用。  
这里解释一下参数的意义：

```
-A ==> set architecture to 'arch'
-O ==> set operating system to 'os'
-T ==> set image type to 'type'
-C ==> set compression type 'comp'
-a ==> set load address to 'addr' (hex)
-e ==> set entry point to 'ep' (hex)
-n ==> set image name to 'name'
-d ==> use image data from 'datafile'
-x ==> set XIP (execute in place)
```

-a 参数后是内核在 SDRAM 中的存储地址，内核将从 Flash 中被 COPY 到这个地方；-e 参数后是内核的入口地址，U-Boot 结束后就会跳到这个地址执行。加完帧头的映像文件名是 zImage.img。

## 二：下载内核和根文件系统到 Flash 中

有两种方法可以将内核和根文件系统下载到 Flash 中。

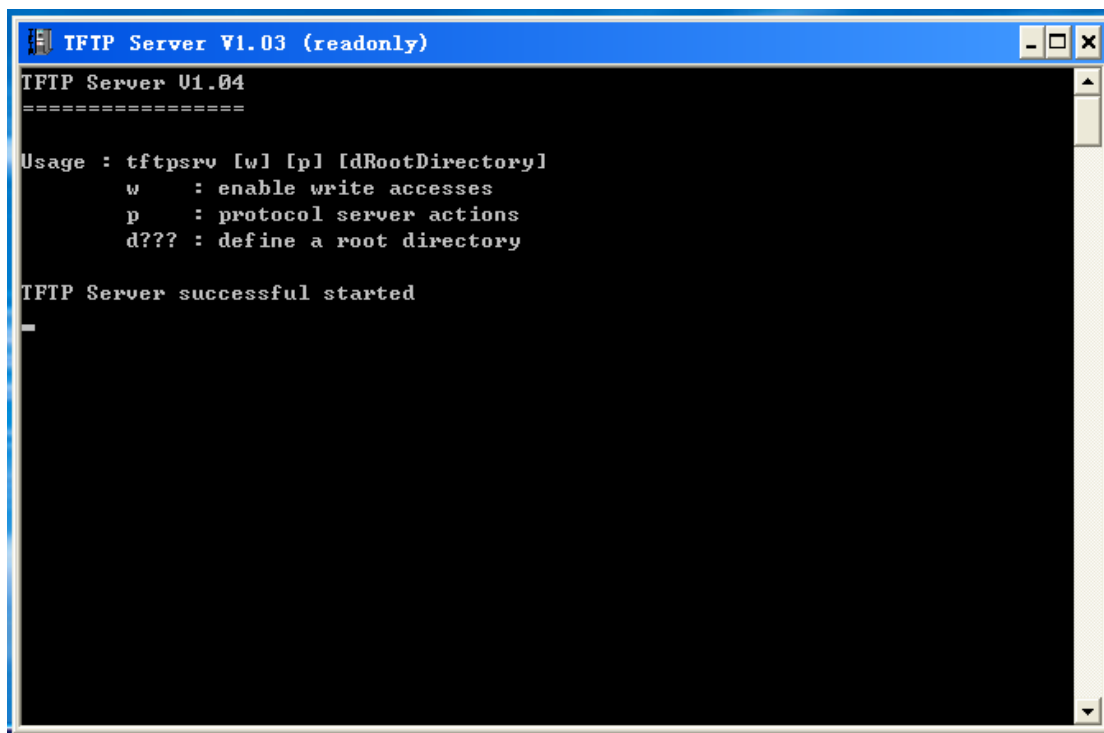
一是使用 AXD 工具，利用 FLASHDOWN 程序下载。

具体操作请参考 4.2.6 节 U-Boot 的烧写。

二是使用 U-Boot 命令，利用 TFTP 协议来下载文件。

下载步骤如下：

- (1) 连接网线，将目标机与宿主机通过交叉网线连接
- (2) COPY TFTPDRV.EXE 到你的电脑中，比如 E:\TFTFSERVER 文件夹下。  
TFTPDRV.EXE 见附件中。
- (3) 把内核映像（zImage.img）与根文件系统映像（myramdisk.gz）拷贝到 tftp 协议目录下，运行 TFTPDRV.EXE，启动 TFTP 协议，出现如下画面表示 tftp 已启动：



(4) 启动目标机，在 U-BOOT 中进行网络参数等环境变量设置

```

U-Boot> setenv ethaddr 12:34:56:78:90:aa //MAC 地址设置
U-Boot> setenv ipaddr 192.168.0.11 //目标板 IP 地址设置
U-Boot> setenv serverip 192.168.0.xxx //服务器 IP 地址设置, 设置成要和目标板
 //连接的电脑的 IP 地址, 一般是你自己 PC
 //的 IP 地址

U-Boot> setenv netmask 255.255.255.0 //设置以太网接口的掩码
U-Boot> setenv bootargs root=/dev/ram rw initrd=0x21200000, 6000000
ramdisk_size=15360 console=ttyS0,115200 mem=64M //设置启动参数
U-Boot> setenv bootcmd cp.b 10300000 21200000 300000\; bootm 10100000
 //自启动时执行的命令。cp.b COPY 根文件
 //系统到 SDRAM 中, bootm 用来引导内核,
 //10100000 是内核存放在 Flash 中的地址

U-Boot> saveenv //保存环境变量

```

如果你在 4.2.7 节已按照当时的操作要求设置过环境变量，那么此处可不必设置，因为这两处的环境变量设置是一样的。我们可通过 printenv 命令来查看一下我们的环境变量设置。如果有异常，请按照此处重新设置。

这里网络环境变量设置的目的是为了能够使用 TFTP 协议；

bootargs 用来定义传递给 Linux 内核的命令行参数。root=是定义从什么地方 mount 根文件系统，/dev/ram表示从 ramdisk mount 根文件系统；rw表示可读写；initrd=0x21200000制定根文件系统加载的起始地址；6000000代表根文件系统 myramdisk.gz 的大小，单位是 byte；ramdisk\_size=15360是 mount 后根文件系统的大小，定义请见 ramdisk 制作部分；console=ttyS0是定义的显示终端，这边选择 DEBUG COM 设备；115200是串口波特率；mem=64M是 SDRAM 的大小，这里的数值不能设置错误，否则会发生地址越界错误。

Bootcmd是定义自动启动时执行的命令；cp.b将 10300000 地址开始处的根文件系统 COPY 到 21200000 地址开始处的 SDRAM 中，大小为 0x300000；bootm 10100000表示从 10100000 处引导内核程序。

先拷贝再  
加载或挂  
载

(5) 下载内核与根文件系统映像到 SDRAM

U-Boot> tftp 20008000 zImage.img ; 下载内核到地址 20008000 处;  
U-Boot> tftp 21200000 myramdisk.gz ; 下载根文件系统到地址 21200000 处;

(6) 从 SDRAM 将映像文件 COPY 到 Flash

U-Boot> erase 10100000 105fffff ; 写之前将 Flash 擦除  
U-Boot> cp.b 20008000 10100000 200000 ; 下载内核到地址 10100000 处;  
U-Boot> cp.b 21200000 10300000 300000 ; 下载根文件系统到地址 10300000 处;

### 三：内核启动

至此，我们将 UBOOT 、内核、根文件系统都下载到 Flash 中了，并且设置了环境变量。重启目标板，等待延时时间结束自动进入本地装载模式，引导内核启动。

Linux 成功启动，将看到如下画面：



```

9200 - 超级终端
文件(F) 编辑(E) 查看(V) 呼叫(C) 传送(T) 帮助(H)

U-Boot 1.1.2 (Jul 24 2007 - 11:18:43)

U-Boot code: 21F00000 -> 21F15338 BSS: -> 21F19C0C
RAM Configuration:
Bank #0: 20000000 64 MB
SST: SST39VF6401B (64Mbit)
Flash: 8 MB
In: serial
Out: serial
Err: serial
Hit any key to stop autoboot: 0
Booting image at 10100000 ...
Image Name: linux-2.6.20.7
Image Type: ARM Linux Kernel Image (uncompressed)
Data Size: 1235928 Bytes = 1.2 MB
Load Address: 20008000
Entry Point: 20008000
Verifying Checksum ... OK

OK

Starting kernel ...

Uncompressing Linux.....

```

上面讲的是将内核和根文件系统固化到 Flash 中的情况，即使用本地加载模式，当以嵌入式产品发布的时候，BootLoader 必须工作在这种模式下。但我们实际在初期调试的时候，并不需要每次将这些映像文件下载到 Flash 中，一方面，Flash 的写入次数有限，另一方面，Flash 的读写速度相对较慢。我们可以在下载操作模式下直接用 TFTP 命令将这些映像文件下载到 SDRAM 中，然后用 bootm 命令从 SDRAM 中引导。进入下载操作模式方法是在延时等待结束前按任意键。

这里要注意一个问题：

注意 bootm xxxx 指定的地址 xxxx 是否与 mkImage 命令处的 -a 指定的加载地址相同。  
一：如果不同的话，mkImage 命令的写法和上面介绍的一致。操作步骤如下：

(1) 映像文件加头

[root@localhost tftpboot]#mkimage -n 'linux-2.6.19' -A arm -O linux -T kernel -C none -a 0x20008000 -e 0x20008000 -d zImage zImage.img

(注意，这条命令是在 Linux 环境下完成的，不是在超级终端下)

(2) 把内核映像 (zImage.img) 与根文件系统映像 (myramdisk.gz) 拷贝到 tftp 协议目

这里指直接在内存中引导的情况？

录下，运行 TFTP SRV.EXE，启动 TFTP 协议

(3) 下载内核与根文件系统映像到 SDRAM

**U-Boot> tftp 21000000 zImage.img**

；下载内核到地址 21000000 处；

**U-Boot> tftp 21200000 myramdisk.gz**

；下载根文件系统到地址 21200000

处；

(4) 启动

**U-Boot> bootm 21000000**

在下面有解释

二：如果相同的话，在使用 mkImage 命令时，**-e** 指定的入口地址应推后 64byte，以跳过这 64byte 的头部。因为地址相同，所以在 **do\_bootm** 函数执行时中并没有去下载代码，此时帧头并没有去掉，因此，**-e** 参数后的入口地址要比 **-a** 参数后的存储地址推后 64byte。操作步骤如下：

(1) 映像文件加头

**[root@localhost tftpboot]#mkimage -n 'linux-2.6.19' -A arm -O linux -T kernel -C none -a 0x20008000 -e 0x20008040 -d zImage zImage.img**

(2) 把内核映像（zImage.img）与根文件系统映像（myramdisk.gz）拷贝到 tftp 协议目录下，运行 TFTP SRV.EXE，启动 TFTP 协议

(3) 下载内核与根文件系统映像到 SDRAM

**U-Boot> tftp 20008000 zImage.img**

；下载内核到地址 0x20008000 处；

**U-Boot> tftp 21200000 myramdisk.gz**

；下载根文件系统到地址 21200000

处；

(4) 启动

**U-Boot> bootm 0x20008000**

从前面我们可以知道，bootm 命令可以直接引导内核映像启动，事实上它也是比较常用的命令。下面介绍一下 bootm 命令的实现。

### 一： bootm 命令的实现

bootm 命令调用 do\_bootm 函数。这个函数专门用来引导各种操作系统映像，可以支持引导 Linux、vxWorks、QNX 等操作系统。引导 Linux 的时候，调用 do\_bootm\_linux() 函数。

```
/* common/cmd_bootm.c */
int do_bootm (cmd_tbl_t *cmdtp, int flag, int argc, char *argv[])
{
 ulong iflag;
 ulong addr;
 ulong data, len, checksum;
 ulong *len_ptr;
 uint unc_len = 0x400000;
 int i, verify;
 char *name, *s;
 int (*appl)(int, char *[]);
 image_header_t *hdr = &header;

 s = getenv ("verify");
 verify = (s && (*s == 'n')) ? 0 : 1;
 if (argc < 2) {
 addr = load_addr; //默认的引导地址，load_addr=CFG_LOAD_ADDR=
 //0x21000000,在 include/configs/at91rm9200dk.h 中定义;
 } else {
 addr = simple_strtoul(argv[1], NULL, 16); //在我们的例子中 addr=0x10100000
 }
}
```



```

}
SHOW_BOOT_PROGRESS (1);
printf ("## Booting image at %08lx ...\n", addr);
/* Copy header so we can blank CRC field for re-calculation */
memcpy (&header, (char *)addr, sizeof(image_header_t));
//将内核的头移到 header 结构体中，以便下面分析
if (ntohl(hdr->ih_magic) != IH_MAGIC)
{
 puts ("Bad Magic Number\n");
 SHOW_BOOT_PROGRESS (-1);
 return 1;
}
SHOW_BOOT_PROGRESS (2);
data = (ulong)&header;
len = sizeof(image_header_t);

checksum = ntohl(hdr->ih_hcrc);
hdr->ih_hcrc = 0;

if(crc32 (0, (char *)data, len) != checksum) { //对帧头做 CRC32 验证
 puts ("Bad Header Checksum\n");
 SHOW_BOOT_PROGRESS (-2);
 return 1;
}
SHOW_BOOT_PROGRESS (3);
/* for multi-file images we need the data part, too */
print_image_hdr ((image_header_t *)addr);
data = addr + sizeof(image_header_t); //此时 data 指向的是真正内核开头的地址
len = ntohl(hdr->ih_size);
if(verify) {
 puts (" Verifying Checksum ... ");
 if(crc32 (0, (char *)data, len) != ntohl(hdr->ih_dcrc)) { //对内核作 CRC32 验证
 printf ("Bad Data CRC\n");
 SHOW_BOOT_PROGRESS (-3);
 return 1;
 }
 puts ("OK\n");
}
SHOW_BOOT_PROGRESS (4);
len_ptr = (ulong *)data;
switch (hdr->ih_comp) {
 case IH_COMP_NONE:
 /*判断 bootm xxxx 这个指定的地址 xxxx 是否与-a 指定的加载地址相同。
 (1)如果不同的话会从这个地址开始提取出这个 64byte 的头部，对其进行分析，然后把去掉
 头部的内核复制到-a 指定的 load 地址中去运行之；
 (2)如果相同的话那就让其原封不同的放在那，但使用 mkImage 命令时，-e 指定的入口地址
 应推后 64byte，以跳过这 64byte 的头部。
 */
 if(ntohl(hdr->ih_load) == addr) {
 printf (" XIP %s ... ", name);
 } else {
 /* !(CONFIG_HW_WATCHDOG || CONFIG_WATCHDOG) */
 memcpy ((void *) ntohl(hdr->ih_load), (uchar *)data, len);
 }
 break;
}
.....

```

XXX为实际内存中的内核地址

```

switch (hdr->ih_os) {
default: /* handled by (original) Linux case */
case IH_OS_LINUX:
 do_bootm_linux (cmdtp, flag, argc, argv, //跳转到 do_bootm_linux 执行
 addr, len_ptr, verify);
 break;

}

```

## 二：do\_bootm\_linux 函数的实现

do\_bootm\_linux()函数是专门引导 Linux 映像的函数，它还可以处理 ramdisk 文件系统的映像。这里引导的内核映像和 ramdisk 映像，必须是 U-Boot 格式的。我们这里只用它来引导内核映像，因此只对内核映像使用 mkImage 工具来转化格式。

do\_bootm\_linux()函数具体做如下事情：

(1) 检查是否有根文件系统映像文件；这里我们并没有用 bootm 命令去引导根文件系统，而是用 cp.b 命令把根文件系统 COPY 到 SDRAM 中。

如果我们用 bootm 命令去引导根文件系统，则需要将根文件系统的映像文件也加帧头，bootm 命令改为 bootm 10100000 10300000。

默认的地址

(2) 将要传递给 Linux 的参数存放到标记列表中。参数传递请参考 3.1.5 节；U-Boot 中参数链表的定义在 include/asm-arm/setup.h 中，参数设定好后，链表的首地址将指向 0x20000100，内核将会从此处接收参数，完成参数传递。

(3) 调用 Linux 内核。U-BOOT 调用 Linux 内核的方法是直接跳转到内核的第一条指令处。在跳转时，要满足下列条件：

(1) CPU 寄存器的设置：R0=0；R1=机器类型 ID（本系统的机器类型 ID=251）；R2=启动参数标记列表在 RAM 中的起始基地址；

(2) CPU 模式：必须禁止中断(IRQs 和 FIQs)；CPU 必须工作在 SVC 模式；

(3) Cache 和 MMU 的设置：MMU 必须关闭；指令 Cache 可以打开也可以关闭；数据 Cache 必须关闭。

系统采用下列代码来进入内核函数：

```

theKernel = (void (*)(int, int))ntohl(hdr->ih_ep);
theKernel(0, bd->bi_arch_number);

```

其中，hdr 是 image\_header\_t 类型的结构体，hdr->ih\_ep 指向内核的第一条指令地址，即 Linux 操作系统下的/kernel/arch/arm/boot/compressed/head.S 汇编程序。第一条指令将 theKernel 函数指向内核首地址处；第二条指令调用 theKernel()函数，也就等于跳进了内核入口地址处，并且传递了 r0、r1、r2 参数，该函数调用应该不会返回，如果该调用返回，则说明出错。

```

/* lib_arm/armlinux.c */
void do_bootm_linux (cmd_tbl_t *cmdtp, int flag, int argc, char *argv[],
 ulong addr, ulong *len_ptr, int verify)
{
 DECLARE_GLOBAL_DATA_PTR;
 ulong len = 0, checksum;
 ulong initrd_start, initrd_end;
 ulong data;
 void (*theKernel)(int zero, int arch, uint params);
 image_header_t *hdr = &header;
 bd_t *bd = gd->bd;
#ifdef CONFIG_CMDLINE_TAG
 char *commandline = getenv ("bootargs");
#endif
 theKernel = (void (*)(int, int, uint))ntohl(hdr->ih_ep); //将 theKernel 函数指向
 //内核首地址处
}

```



```

/* Check if there is an initrd image */
if(argc >= 3) { //检查是否有根文件系统映像，我们这里引导的内核映像中是没
 //有根文件系统的，根文件系统映像文件是另外下载的。

}
/* Now check if we have a multifile image */
} else if ((hdr->ih_type == IH_TYPE_MULTI) && (len_ptr[1])) {

} else {
 /* no initrd image */
 SHOW_BOOT_PROGRESS (14);
 len = data = 0;
}
if (data) {
 initrd_start = data;
 initrd_end = initrd_start + len;
} else {
 initrd_start = 0;
 initrd_end = 0;
}
SHOW_BOOT_PROGRESS (15);
debug ("## Transferring control to Linux (at address %08lx) ...\n",
 (ulong) theKernel);

/*下面开始建立内核参数传递列表*/
#ifdef CONFIG_SETUP_MEMORY_TAGS || \
 defined (CONFIG_CMDLINE_TAG) || \
 defined (CONFIG_INITRD_TAG) || \
 defined (CONFIG_SERIAL_TAG) || \
 defined (CONFIG_REVISION_TAG) || \
 defined (CONFIG_LCD) || \
 defined (CONFIG_VFD)
 setup_start_tag (bd);
#endif
#ifdef CONFIG_SERIAL_TAG
 setup_serial_tag (¶ms);
#endif
#ifdef CONFIG_REVISION_TAG
 setup_revision_tag (¶ms);
#endif
#ifdef CONFIG_SETUP_MEMORY_TAGS
 setup_memory_tags (bd);
#endif
#ifdef CONFIG_CMDLINE_TAG
 setup_commandline_tag (bd, commandline);
#endif
#ifdef CONFIG_INITRD_TAG
 if (initrd_start && initrd_end)
 setup_initrd_tag (bd, initrd_start, initrd_end);
#endif
 setup_end_tag (bd);
#endif
/* we assume that the kernel is in place */
printf ("\nStarting kernel ...\n\n");
cleanup_before_linux ();
/*跳转到内核程序*/
theKernel (0, bd->bi_arch_number, bd->bi_boot_params);

```

}

至此，程序分析结束，U-Boot 将控制权交给内核以后，就不再发挥作用。

**参考资料：**

1. <<嵌入式 Linux 系统开发技术详解—基于 ARM>>
2. << 构建嵌入式 Linux 系统>>
3. U-Boot-1.1.2 源代码

## 第五章 Linux

### 5.1 Linux 原理

操作系统的基本概念：任何计算机系统都包含一个名为操作系统的基本程序集合。在这个集合里，最重要的程序称为内核(kernel)。当操作系统启动时，内核被装入到 RAM 中，内核中包含了系统运行所必不可少的很多过程。内核也为系统中所有事情提供了主要功能，并决定高层软件的很多特性。操作系统必须完成两个主要目标：

1. 与硬件部分交互，为包含在硬件平台上的所有低层可编程部件提供服务。
2. 为运行在计算机系统上的应用程序提供执行环境。

前面已经对 Linux 操作系统进行了简单的介绍，这里将系统地阐述 Linux 的原理。叙述的顺序是：内存管理，中断管理，进程管理，设备管理，文件系统。

#### 内核体系结构：

Linux 是单一结构的操作系统，每个内核层都被集成到整个内核中，并代表当前进程在内核态下运行。单一结构不同于多层次的微内核结构，它只需要内核有一个很小的函数集，通常包括几个同步原语、一个简单的调度程序和进程间的通信机制，运行在微内核之上的几个系统进程实现从前操作系统级实现的功能，如内存分配、设备驱动程序、系统调用处理程序等等。尽管关于操作系统的学术研究都是面向微内核的，但这样的操作系统一般比单一结构内核的效率低，因为操作系统不同层次间显示的消息传递要花费一定的代价。微内核操作系统迫使系统程序员采用模块化的方法，可以很容易地移植到其他的体系结构上。为了达到微内核优点而不影响性能，Linux 内核提供了模块(module)。模块是一个目标文件，其代码可以在运行时链接到内核或从内核解除链接。这种目标代码通常由一组函数组成，用来实现文件系统、驱动程序或其他内核上层功能。与微内核操作系统的外层不同，模块不是作为特殊的进程执行的。相反，与其他静态链接的内核函数一样，它在内核态代表当前进程执行。

使用模块化的主要优点包括：

模块化方法：

模块化方法使系统程序员必须提出明确定义的软件接口以访问由模块处理的数据结构。这使得开发新模块变得容易。

平台无关性：

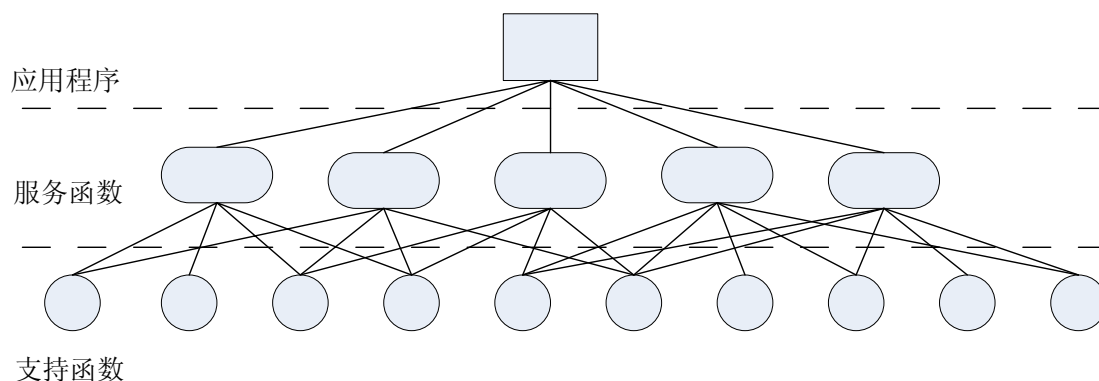
即使模块依赖于某些特殊的硬件特点，但它不依赖于某个固定的硬件平台。

节省内存使用：

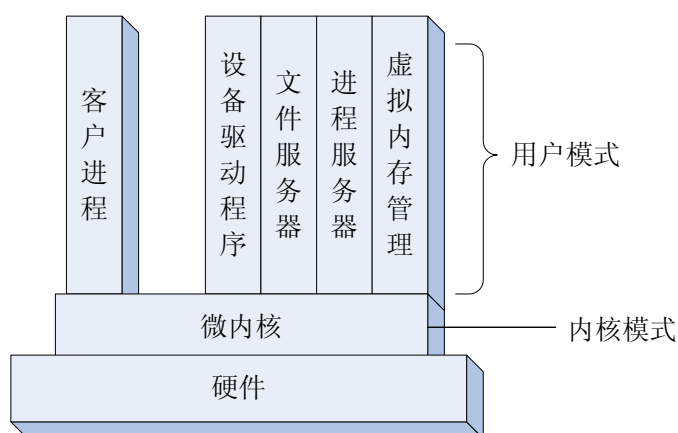
当需要模块功能时，把它链接到正在运行的内核中，否则，将该模块解除链接。这种机制也可以对用户透明，因为链接和解除链接可以由内核自动完成。

无性能损失：

模块的目标代码一旦被链接到内核，其作用与静态链接的内核目标代码完全等价。因此，当模块的函数被调用时，无需显式的消息传递。



单内核模式的简单结构模型

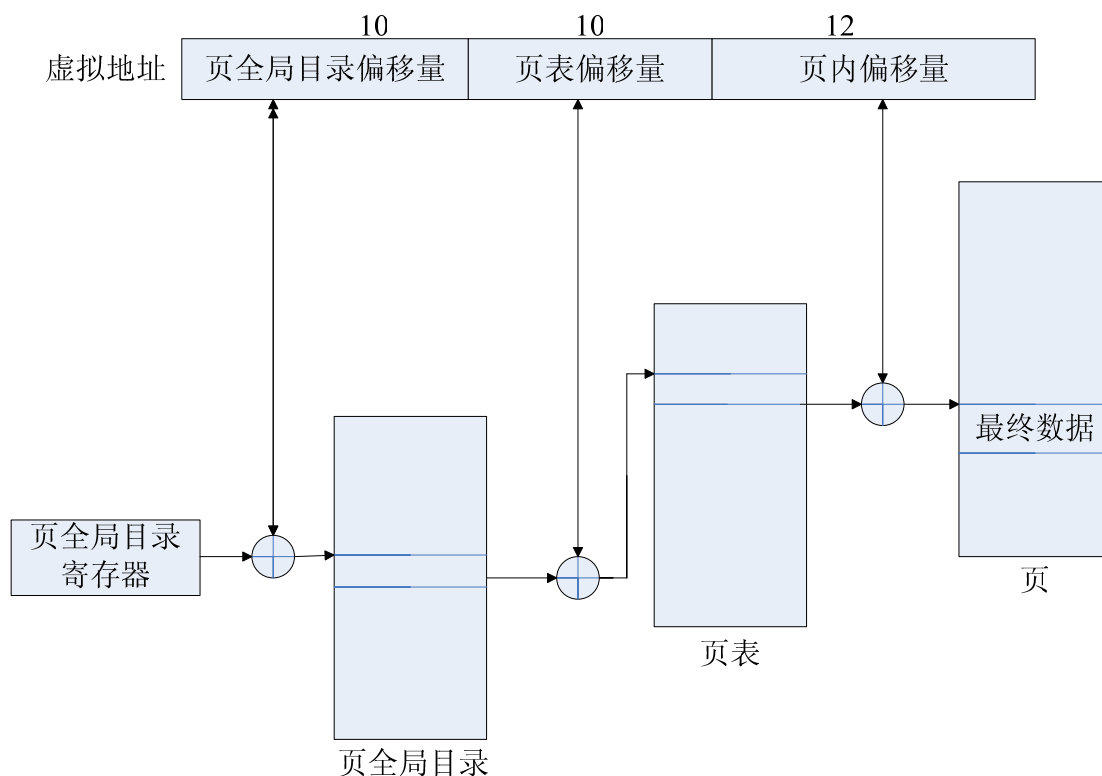


微内核模式的简单结构模型

### 存储器管理:

Linux 支持不同的体系结构，而不同的体系结构在对硬件的分段和分页支持上也是不一样的。例如：x86 的体系结构即支持分段又支持分页，而 ARM 体系结构只支持分页结构，所以这两种体系结构的板级支持(BSP)代码是有区别的。Linux 中定义了三个地址，分别是逻辑地址，物理地址，虚拟地址即线性地址。逻辑地址是指偏移量。物理地址是指处理器访问物理内存时的总线地址。虚拟地址是体系结构支持的可以将地址空间扩展为超过实际物理空间的地址，有些处理器并不支持虚拟地址，例如没有 MMU 单元的处理器，那么在选择 Linux 内核的时候就必须选择不支持 MMU 的 Linux 内核，如 ucLinux，ucLinux 在原有 Linux 的基础上去掉 Linux 对 MMU 支持的代码，并添加了实时性控制。

在 ARM Linux 中每个进程的地址空间从 0x0000 0000---PAGE\_OFFSET (0xC0000000)，进程的地址都是虚拟地址。从虚拟地址转变到物理地址的步骤是：从当前的页全局目录寄存器中找到全局目录的物理地址，根据虚拟地址高 10 位作为在页全局目录里的偏移量找到页表的物理地址，再以虚拟地址的中间 10 位作为偏移量在页表中查找页的物理地址，最后的页物理地址加虚拟地址的后 12 位就是虚拟地址对应的物理地址。如下图所示：



### 虚拟地址到物理地址的变换

这里所述的只是两级分页，线性地址（虚拟地址）只有 32 位，Linux 的最新版本支持 64 位地址总线，三级分页即在以前的版本上再添加页中间目录。

从 `PAGE_OFFSET—0xFFFF FFFF` 的地址空间是属于内核地址空间，只有内核态下的程序可以访问，任何用户态进程都不能访问。这个地址空间除了能够映射全部的物理地址以外，还可以映射其它的非连续的物理内存，例如嵌入式处理器中的中断向量表，I/O 控制器等等。

内核在初始化的时候就将物理内存每 4KB 标记为一页并且每一页都用描述符 `struct page` 来标记，以后所有的内存分配都会用到这个标记。内核在正常运行过程中的内存分配都通过一定的算法来进行，例如伙伴算法(buddy)，这是最低层的内存分配算法，在它之上还有 slab 算法，高速缓存算法等等。当然内核对物理页的管理是建立在许许多多的数据结构之上的，包括页描述符，伙伴算法的数据结构，进程线性区描述符等等。

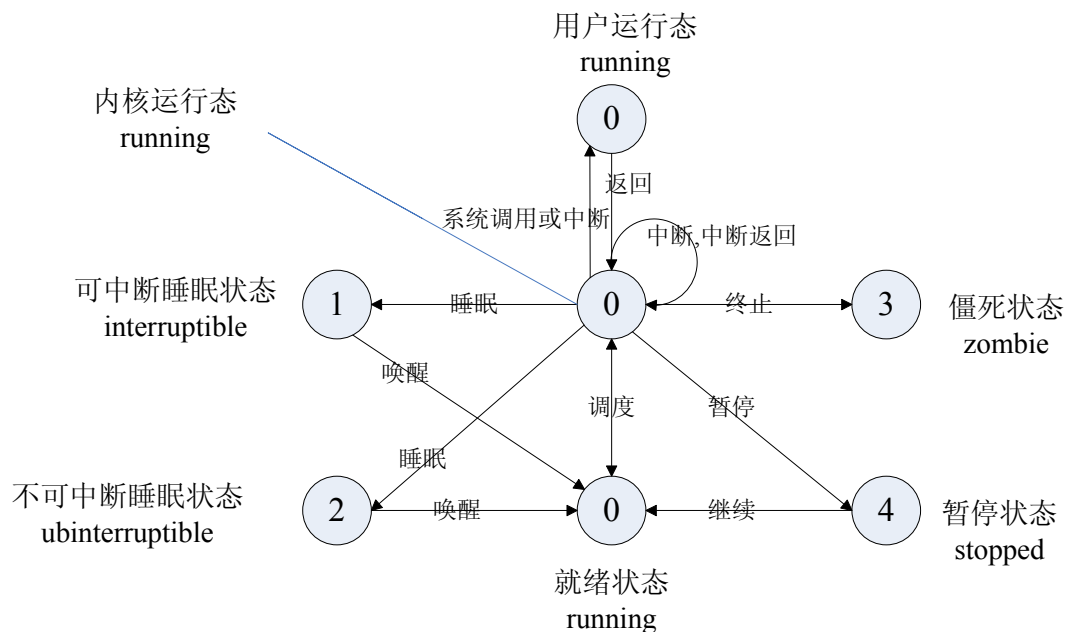
### 进程管理：

进程是任何多道程序设计的操作系统中的基本概念，通常把进程定义为程序执行的一个实例。你可以把它看作充分描述程序已经执行到何种程度的数据结构的汇集。从内核的观点看，进程的目的就是担当分配系统资源的实体。在 Linux 中一个新的进程被创建的时候，它与创建它的进程即父进程享有同样的地址空间，一旦新进程要写内存时，内核才为它申请内存空间，并且将父进程的数据拷贝到新的内存空间，这个技术叫写时复制。

Linux 内核为了管理进程定义了进程描述符(task\_struct)，在这个描述符里记录了与进程相关的所有信息，包括：进程的状态，进程的标记，进程的相关链表，内存分配等等，所以相当复杂。最新的 Linux 支持进程，轻量级进程和线程，它们的区别就在于被创建时要从其父进程的描述符中拷贝哪些信息。

进程共有五种状态，分别是：可运行状态，可中断的等待状态，不可中断的等待状态，暂停状态，僵死状态。可运行状态表示进程要么在 CPU 上执行，要么准备执行。可中断的等待状态表示进程被挂起，但可以内核唤醒。不可中断的等待状态和前一个类似，只是信号不能唤醒该进程。暂停状态表示进程收到一些特殊的暂停信号就挂起，接收相应的唤醒信号才醒来。僵死状态表示进程执行被终止。为了能够快速地遍历或者找到相关进程，内核建立

了很多的链表，包括：进程描述符链表，空闲进程描述符链表，可运行状态描述符链表等等。



## 进程状态及转换关系

为了控制进程的执行，内核必须有能力挂起正在 CPU 上运行的进程，并恢复执行以前挂起的某个进程。这种行为被称为进程切换、任务切换或文境切换。有些体系结构支持硬件直接切换，例如 X86。Linux 根据不同的体系结构支持不同进程切换，有些是由硬件本身实现，有些是软件实现。

进程之间的调度是根据一定的算法来进行的，Linux 调度算法把 CPU 的时间划分为时期，在一个单独的时期内，每个进程有一个指定的时间片，其持续时间从时期的开始计算。一般情况下不同的进程有不同的时间片大小。时间片的值是在那个时期内分配给进程的最大 CPU 时间部分。当一个进程用完它的时间片时，这个进程被抢占，并用另一个可运进程代替它。当然，在同一时期内，一个进程可以几次被调度程序选中，当所有的可能够进程都用完它们的时间片时，一个时期才结束；在这种情况下，调度程序的算法重新计算所有进程的时间片持续时间，然后，一个新的时期又开始。Linux 支持优先级调度，体现在优先级高的进程可以得到更多的时间片，因此允许用户开发实时应用。不过 Linux 不支持对实时要求很高的实时应用，因为 Linux 的内核是非抢占式的。

Linux 进程之间的数据同步和交换是通过管道、FIFO、信号量、消息、共享内存区、套接字实现的。

**管道和 FIFO：**最适合在进程之间实现生产者/消费者的交互，有些进程往管道中写入数据，而另外一些进程则从管道中读出数据。

**信号量：**是一种计数器，如果受保护的资源是可用的，那么信号量的值就是正数，否则信号量的值就是负数或 0。要访问资源的进程试图把信号量值减 1，如果信号量的值是负数或 0 内核阻塞这个进程直到在这个信号量上的操作产生一个正值。当进程释放受保护的资源时就把信号量的值增加 1，在这样处理的过程中，其他所有正在等待这个信号量的进程都必须被唤醒。

**消息：**允许进程在预定的消息队列中读和写消息来交换信息（小块数据）。

**共享内存区：**允许进程通过共享内存块来交换信息。在必须共享大量数据的应用中，这可能是最高效的进程通信形式。

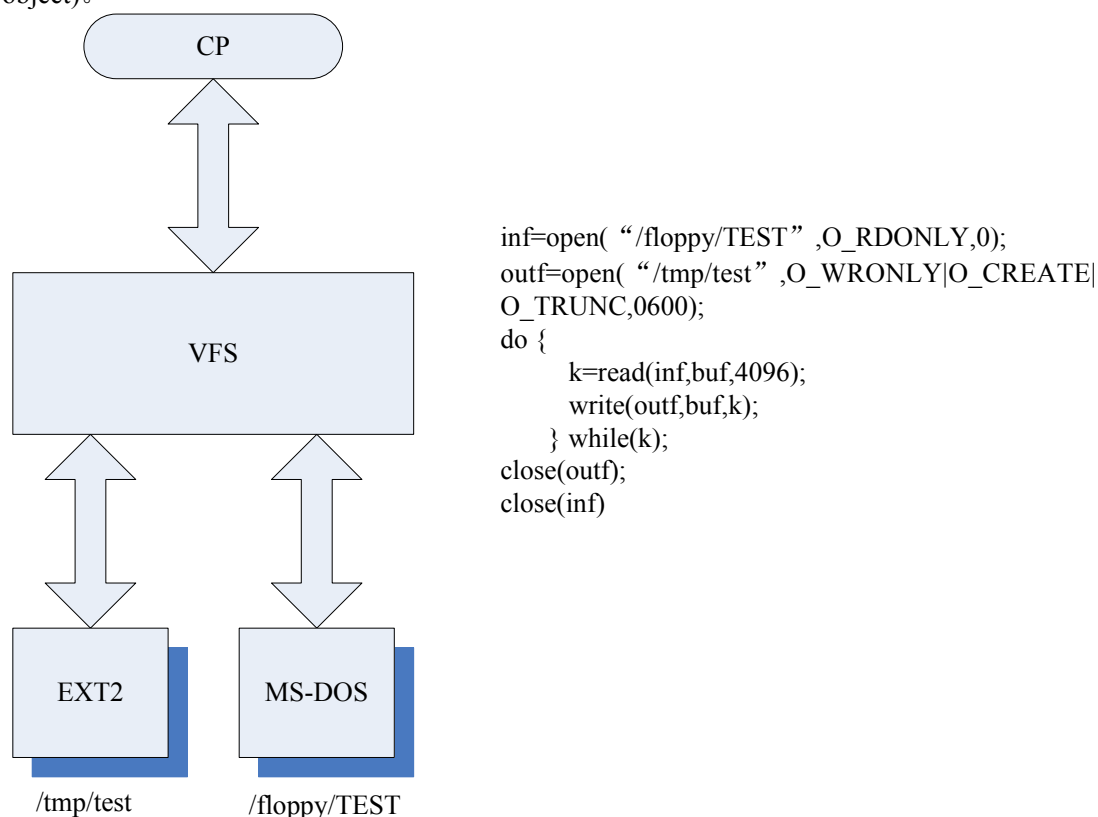
**套接字：**允许不同计算机上的进程通过网络交换数据，套接字还可以用作相同主机上的进程之间的通信工具。

**文件系统：**

Linux 成功的关键因素之一是具有与其他操作系统和谐共存的能力。通过虚拟文件系统 (VFS) Linux 使用与其他 Unix 变体相同的方式设法支持多种磁盘类型。虚拟文件系统所隐含的思想是把表示很多不同种类文件系统的广泛信息放入内核；其中有一个字段或函数来支持 Linux 支持的任何实际文件系统所提供的任何操作。对所调用的每个读、写或其他函数，内核都能把它们替换成支持本地 Linux 文件系统、NT 文件系统，或者文件所在的任何其他文件系统的实际函数。

VFS 所隐含的主要思想在于引入了一个通用的文件模型，这个模块能够表示所有支持的文件系统。该模型严格反映传统 Unix 文件系统提供的文件模型。要实现每个具体的文件系统，必须将其物理组织结构转换为虚拟文件系统的通用文件模型。从本质上说，Linux 的内核不能对一个特定的函数进行硬编码来执行如 `read()` 和 `write()` 这样的操作，而是每个操作都必须使用一个指针，指向要访问的具体文件系统的适当函数。

通用文件模型可以理解为面对对象的，在这里，对象是一个软件结构，其中既定义了数据结构也定义了其上的操作方法，通用文件模型由下列对象类型组成：超级块对象 (superblock object)，索引节点对象 (inode object)，文件对象 (file object)，目录项对象 (dentry object)。



### VFS在一个简单的文件复制操作中的作用

超级块对象，存放已安装文件系统的有关信息。对基于磁盘的文件系统，这类对象通常对应于存放在磁盘上的文件系统控制块，数据结构是 `struct super_block`，该结构的最后一个字段联合体 `u` 包含属于具体文件系统的超级块信息。

索引节点对象，存放关于具体文件的一般信息。对基于磁盘的文件系统，这类对象通常对应于存放在磁盘上的文件控制块。每个索引节点对象都有一个索引节点号，这个号唯一地标识文件系统的文件。数据结构 `struct inode`，该结构的最后一个字段联合体 `u` 包含属于具体文件系统的索引节点信息。

文件对象，存放打开文件与进程之间进行交互的有关信息。这类信息仅在进程访问文件期间存放于内核内存中。数据结构 `struct file`，存放在文件对象中的主要信息是文件指针，即文件中当前的位置，下一个操作将在该位置发生。



目录对象，存放目录项与对应文件进行交互的有关信息。每个基于磁盘的文件系统都以自己特有的方式将该类信息存放在磁盘上。数据结构是 `struct dentry`，目录项对象在磁盘上并没有对应的映像，因此在 `dentry` 结构中不包含指出该对象已被修改的字段。

### 设备管理：

Linux 操作系统是基于文件概念的，文件是以字符序列而构成的信息载体。所以 I/O 设备也是当作文件来处理的。因此，与磁盘上的普通文件进行交互所用的同一系统调用可直接用于 I/O 设备。例如，用同一 `write()` 系统调用既可以向普通文件中写入数据，也可以通过向 `/dev/lp0` 设备文件中写入数据从而把数据发往打印机。

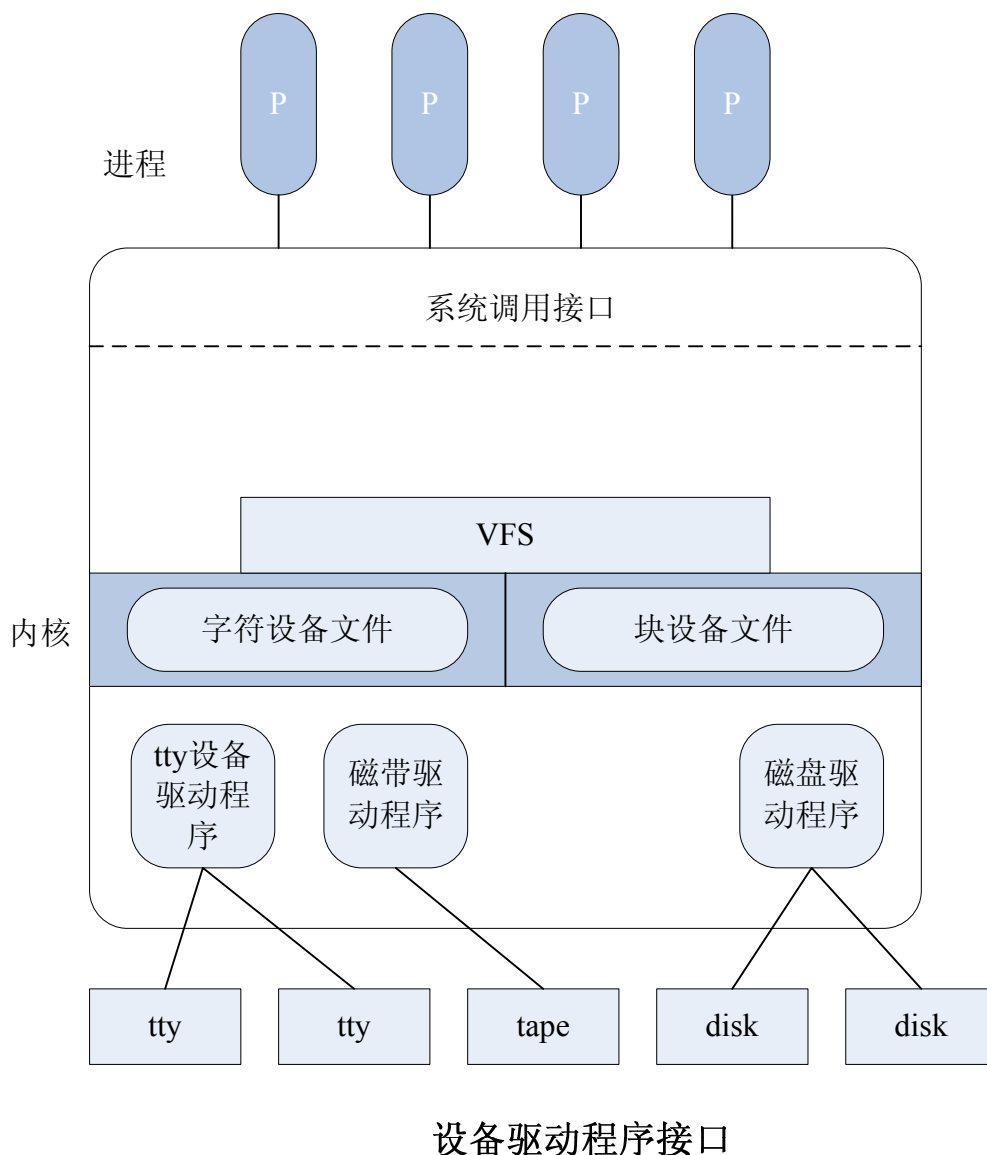
根据设备驱动程序的基本特性，Linux 将设备文件分为两种：块和字符。这两种硬件设备之间的差异并不容易划分。块设备的数据可以被随机的访问，而且从用户的观点看，传送任何数据块所需的时间都是较少且大致相同的。块设备的典型例子是：硬盘，软盘，CD-ROM 及 DVD 播放器。字符设备的数据或者不可以被随机访问，或者可以被随机访问，但是访问随机数据所需的时间很大程度上依赖于数据在设备内的位置。

根据设备文件的结构不同，Linux 有两种设备文件：老式的设备文件和 `devfs` 设备文件，前者是系统目录树中真实的文件，而后者是虚拟文件，如 `/proc` 文件系统中的文件。老式设备文件是存放在文件系统在实际文件。然而它的索引节点并不对磁盘上的数据块编址，而是包含硬件设备的一个标识。每个设备文件除了名字和类型外，还有两个主要属性：主设备号(`major number`)：从 1 到 254 之间标识设备的类型的一个数。通常，具有相同主设备号和相同类型的所有设备文件共享相同的文件操作集合，因为它们是由同一设备驱动程序来处理的。次设备号(`minor number`)：在一组主设备号相同的设备之间惟一标识特定设备的一个数。因为老式的设备文件存在缺陷，所以引入了 `devfs` 设备文件，devfs 虚文件系统允许设备驱动程序通过名字而不是主设备号和次设备号注册设备。内核提供了一个缺省的模式，给模式使得搜索特定设备变得容易，但老式的设备文件对于 Linux 系统又是必不可少的，因此目前 `devfs` 层让内核为每个设备驱动程序象老式设备文件一样定义主设备号和次设备号，几乎所有的设备驱动程序把 `devfs` 设备文件与相应老式设备文件的同一主设备号和次设备号关联起来。

对设备文件的访问不同于对普通文件的访问，当进程访问普通文件时，它会通过文件系统访问磁盘分区中的一些数据块；而在进程访问设备文件时，它只要驱动硬件设备就可以了。VFS 的责任是为应用程序隐藏设备文件与普通文件之间的差异。为了做到这点，VFS 在设备文件打开是时改变其缺省文件操作，因此可以把设备文件的每个系统调用都转换成与设备相关的函数的调用，而不是对主文件系统相应函数的调用。与设备相关的函数对硬件设备进行操作以完成进程所请求的操作。

设备驱动程序是一个软件层，该软件层使硬件设备响应预定义好的编程接口，这些接口由一组控制设备的 VFS 函数(`open`, `read`, `lseek`, `ioctl` 等等)组成，这些函数实际实现由设备驱动程序全权负责。由于每个设备都有一个惟一的 I/O 控制器，因此就有惟一的命令和惟一的状态信息，所以大部分的 I/O 设备都有自己的驱动程序。





## 5.2 ARM Linux

Linux 试图在硬件无关的源码与硬件相关的源码之间维持一个清晰的划分。为了做到这点，在源码的 `arch` 和 `include` 目录下都包含了很多子目录，也就是对应了各种所支持的硬件平台，这些平台包括：alpha, arm, cris, i386, ia64, m68k, mips, ppc, s390, sh, sparc 等等。本节将简要地介绍 ARM 公司为了能在 ARM 体系结构的处理器上实现 Linux 的功能所编写的板级支持代码。

首先介绍 Linux-2.6.19.2 源码的结构。

`/arch`: 里面定义了 Linux 所支持的各种硬件平台，每个目录里包括了各种硬件平台的板级支持代码。

`/block`: 这是一个比较新的目录，这里面定义了块设备驱动的高层函数。

`/crypto`: 收录了各种算法的源代码文件，例如：CRC 算法。

`/documentation`: 这个目录里存放了内核组件的一般解释及提示的文本文件。

`/drivers`: 各种设备驱动程序目录。

`/fs`: 文件系统目录。

**/include:** 头文件目录。这个目录里又分为两个部分，一个是以 `asm-xxx` 为名称的平台相关的头文件集合，一个是平台无关的头文件集合。

**/init:** 内核的初始化代码。入口函数是 `main.c` 中的 `start_kernel()` 函数。

**/ipc:** 进程之间通信的代码。

**/kernel:** Linux 与平台无关的内核核心代码。

**/lib:** 各种内核常用函数库。

**/mm:** 内存管理代码。

**/net:** 网络支持代码，主要是网络协议。

**/scripts:** 内部或者外部使用的脚本。

**/security:** 有关 Linux 安全模块的代码。

**/sound:** 声音驱动支持代码。

**/user:** 用户代码。

跟平台相关的代码主要分布在 `arch` 目录和 `include` 目录以及 `drivers` 目录里，下面基于我们的 AT91RM9200 实验板介绍 ARM Linux 的板级支持代码。

## arch 目录

在 `arch` 目录里有各种支持的体系结构的子目录，`arm` 也在其中。`/arch/arm` 目录里的代码主要分为两个部分，一部分是与处理器相关的代码，另一部分是与处理器无关代码，例如：`/arch/arm/mach-at91rm9200` 就是与具体 ARM 结构的处理器相关，而 `/arch/arm/mm` 就是与具体处理器无关的代码，主要是基于 ARM 体系结构的内存管理的低层代码。

**/arch/arm/boot:** ARM 体系结构的启动代码，内核就从这里启动，内核编译完成后的映像文件也在这个目录里。

**/arch/arm/common:** ARM 体系结构的杂项代码。

**/arch/arm/configs:** 各种平台的默认配置文件，这些配置文件都是针对具体 ARM 核处理器的。例如，基于 AT91RM9200 处理器的配置有：`at91rm9200dk_defconfig`、`at91rm9200ek_defconfig` 等等。这些只是默认的配置，具体的应用还是需要在这个基础上进行适当的修改的。

**/arch/arm/mach-xxxxx:** 这些目录是针对具体处理器的。例如，`/arch/arm/mach-at91rm9200` 目录，定义了基于 `at91rm9200` 的代码，文件如下：

- `at91rm9200.c`: 定义了 `at91rm9200` 的 I/O 口描述符，用来做地址映射。时钟结构体，以及初始化函数。
- `at91rm9200_time.c`: 定义 `at91rm9200` 的定时器(ST)的各种操作函数。
- `board-xxx.c`: 定义了基于 `at91rm9200` 的不同实验板、开发板、评估板区别的代码，包括机器描述符。例如，`at91rm9200` 的两种封装：PQFP 和 BGA。这两种封装在 I/O 的接口上存在一些区别。并且各种板上所支持的硬件也不完全一样。这里的各种板跟前面 `configs` 目录下的 `at91rm9200` 的各种默认配置是一一对应的。
- `clock.c`: 定义了对 `at91rm9200` 的晶振和时钟操作的函数。
- `devices.c`: 定义了 `at91rm9200` 的各种外设(I/O 控制器)资源的配置和初始化函数。
- `gpio.c`: 定义了对 `at91rm9200` 的通用 I/O 口进行配置和操作的函数。
- `irq.c`: 定义了对 `at91rm9200` 的中断控制器(AIC)操作的函数。
- `led.c`: 对由 GPIO 口控制的 LED 灯的控制函数。
- `pm.c`: 定义了对 `at91rm9200` 的电源管理模块(PMC)的控制函数。

**/arch/arm/mm:** 定义了基于 ARM 的各种版本体系结构内存管理模块的代码，包括：MMU 单元，硬件高速缓存器，I/O 端口映射等等。

**/arch/arm/nwfpe:** 关于浮点模拟器的代码（NetWinder Floating Point Emulator）。

**/arch/arm/oprofile:** Linux 评测工具 `oprofile` 的代码。

**/arch/arm/plat-xxx:** 特殊平台相关的代码。

**/arch/arm/tools:** 说明文档，其中 `mach-types` 文件定义了各种 ARM 平台所对应的配置文件以及它们的机器编号。例如：开发板名称为 `at91rm9200dk` 机器编号为 262。

**/arch/vfp:** 关于 ARM 体系结构的浮点处理器代码。

#### **/include 目录**

这个目录收录了绝大多数的头文件，分为两个部分，一个是与体系结构有关的头文件，例如 asm-arm，这个目录是有关 ARM 体系结构的头文件，同样它也分为两个部分，与处理器有关的目录，如 arch-at91rm9200 目录，与处理器无关的文件。另一个是与体系结构无关的头文件，例如 linux 目录。

#### **/drivers 目录**

这个目录里定义了所有设备的驱动程序，根据不同设备的名称来定义子目录。例如 at91rm9200 的串口驱动，可以在目录/drivers/serial 中找到文件 atmel\_serial.c。又如 at91rm9200 的 usb 主机端驱动，可以在目录/drivers/usb/host 中找到文件 ohci-at91.c。例子中的这些都是系统最底层的驱动文件，高一层的驱动文件也同样在相应目录下。驱动程序中很多底层函数都是建立在象/arch/arm/mach-xxxx 目录里函数的基础之上的。

有关 ARM 体系结构的板级支持代码主要就分布在以上的三个目录里，当然其它体系结构的板级支持代码也同样分布在这三个目录里。

## 5.3 编译嵌入式 Linux 内核映像

编译内核之前的准备工作。在 PC 机的 Red Hat Linux 环境下编译嵌入式内核之间的准备工作有：

1. 交叉编译器的建立，例如目标平台是 ARM 体系结构的就必须要用 arm-linux-gcc 编译器以及其它基于 ARM 的工具。
2. 根据自己的处理器或者体系结构找到相应的默认配置，如果没有默认配置只能重新配置编译选项（很可能编译不成功，因为有些选项之间具有相关性），有了默认配置，只要在默认配置的基础上添加自己想要的功能即可。
3. 对 Make 工具以及 Makefile 有些了解。
4. 选择一个合适的内核版本。
5. 熟悉虚拟机中 Red Hat Linux 中各种常用命令。

因为 Linux 支持不同的体系结构，所以在编译内核以前必须要选择符合自己平台的那部分代码来编译。Linux 是通过 Makefile 和 Kconfig 以及 Kbuild 来实现 Linux 的各种配置方法的。目前常用的配置方法有：make config, make menuconfig, make xconfig 三种。而我们常用的是 make menuconfig 菜单式的配置方法。

#### **编译内核**

1. 修改 linux-2.6.19.2 顶层目录下的 Makefile 文件中(Line 182,183)的：

```
ARCH ?=
CROSS_COMPILE ?=
```

根据实验平台的体系结构，我们修改的结果是：

```
ARCH ?=arm
CROSS_COMPILE ?=/home/xchk/compile/3.4.1/bin/arm-linux-
```

“arm”：表示我们的平台是基于 ARM 体系结构的，所以这里使用的交叉编译器是 arm-linux-gcc，由于 Makefile 脚本中使用的编译器是 CROSS\_COMPILE+gcc，所以我们在这里的交叉编译器只写到 arm-linux-，最后真正的编译器是 arm-linux-gcc。前面的这些目录是我们把交叉编译工具(cross tools)解压后所放的具体目录。这个交叉编译工具的版本是 3.4.1。所放的目录是/home/xchk/compile 目录。

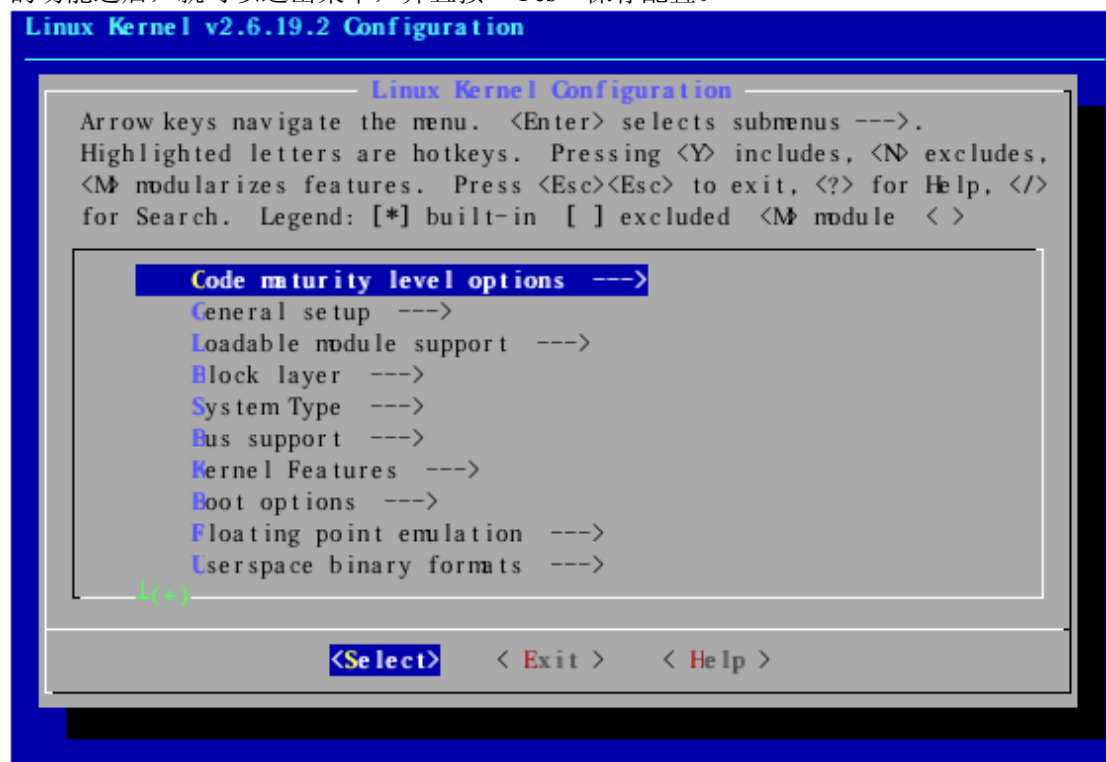
2. 根据自己平台处理器型号寻找适合的默认配置文件，例如：我们实验平台的处理器是 at91rm9200，那么就可以在/arch/arm/configs 目录里寻找到基于 at91rm9200 的默认配置，当然这里有很多基于这个处理器的默认配置，选择哪一种都可以。不过你可以到相应的 /arch/arm/mach-at91rm9200/board-xxx.c 中去看看这些平台之间的差别。at91rm9200 的封装有

两种 PQFP 和 BGA, 并且它们的资源也有些差别。在/arch/arm/mach-at91rm9200 目录里有三个平台用的是 PQFP 封装的, 分别是: board-larm, board-kafa, board-kb9202。其它的都是基于 BGA 封装的。但是我们在开始编译的时候用的是 at91rm9200dk\_defconfig 配置, 对应的平台代码是 board-dk.c 即这里定义的 BGA 封装的处理器, 结果也可以顺利的编译通过, 只不过在内核启动的时候会出现一些提示信息, 不影响系统其它的功能, 后来通过修改 board-dk.c 中的代码 (主要是不同封装之间区别的代码, 具体修改见第 7 章第 2 节) 就去掉了内核启动时的提示信息, 并且可以正常使用。其他的配置应该也没有问题。

3. 接下来就是将选择的默认配置 make 一下, 即在 Linux 提示符下输入: make at91rm9200dk\_defconfig, 经过简单的运行以后, 这个默认的配置文件就被输出为前台的.config。如果我们直接用默认配置就可以跳到第 5 步。此处我们采用默认配置即可。

先将at91rm拷贝到  
顶级目录下, 然后  
make at91rm

4. 除了默认配置以外, 我们可能还需要在内核中添加一些自己的模块或功能 (具体参考第 7 章第 2 节), 那么我们直接可以在提示符下输入: make menuconfig。此时出现了如下图所示的可视化配置菜单。这个菜单中打上 “\*” 号的已选项目都是来自于上一步的 make at91rm9200dk\_defconfig 命令, 也就是这个命令将文字性的默认配置转换成了可视化的配置。我们可以在菜单下选择自己想要添加的功能。菜单的操作指令是: “y” 键表示将该选项编译进内核, “n” 表示不对该选项进行编译, “m” 表示将该选项编译成模块, 而不是直接编译进内核。菜单的进入和退出分别使用 “select” 和 “exit” 按钮, 帮助按钮是 “help”, 通过帮助文件我们可以知道相应的选项对应的名称。例如: <>USB Mass Storage support 对应的名称是 CONFIG\_USB\_STORAGE, 这个名称也是代码中定义的宏变量。添加完你想要的功能之后, 就可以退出菜单, 并且按 “Yes” 保存配置。



菜单配置(make menuconfig)

5. 直接开始编译内核, 在提示符下输入: make 就可以了。如果没有问题的话在 /arch/arm/boot 的目录里就会生成 zImage 的内核映像文件了, zImage 是可来自启动的压缩的内核映像文件。并且在源代码的根目录下生成了另外两个文件: system.map 和 vmlinux。System.map 文件是一个特定内核的内核符号表, 它包含内核全局变量和函数的地址信息。Vmlinux 是内核在虚拟地址空间运行时代码的真实反映, 编译的过程就是按照特定顺序链接目标代码生成的, 因为 Linux 运行在虚拟地址空间所以名字附加 “vm”。Vmlinux 不具备引导的能力, 需要借助 bootloader 引导启动。所以 zImage 可以归结为两个部分构成: 压缩的 vmlinux 和自引导程序。zImage 通过自引导程序初始化系统, 并且解压缩启动 vmlinux, zImage



采用 gzip 压缩格式，并且自身包含 gzip 的解压缩函数。

接下来用 uboot 的  
mkimage 给映像加头

## 5.4 内核调试方法

内核调试方法有多种，很多书籍上都可以查阅到，在这里只介绍我们自己所用的方法，这个方法对于初级的调试还是很起作用的。

首先是在虚拟机的 Red Hat Linux 环境下建立 Samba 服务器(配置请参考《Red Hat Linux 宝典》一书)。将 Linux 的源代码文件进行共享。在 Windows 环境下用代码工具(source insight)打开共享的源代码，并且可以直接修改。这样我们就可以在 Windows 环境下直接修改源代码了。我们所用的初级调试方法就是在源代码中将需要的信息打印出来就是通过内核的 `printk()` 函数，这个函数可以在很多的内核文件中使用。`Printk()`的语法是：`Printk(打印级别 打印信息 输出值)`，例如：

```
Printk(KERN_DEBUG "This is test_%d\n",x).
```

打印级别通常采用宏来表示，在头文件 `/include/linux/kernel.h` 中定义了八种可用的日志级别字符串，以严重程度降序依次是：

**KERN\_EMERG**：用于紧急事件消息，它们一般是系统崩溃之前提示的消息。

**KERN\_ALERT**：用于需要立即采取行动的情况。

**KERN\_CRIT**：临界状态，通常涉及严重的硬件或软件操作失败。

**KERN\_ERR**：用于报告错误状态。设备驱动程序会经常使用这个来报告来自硬件的问题。

**KERN\_WARNING**：对可能出现问题的情况进行警告，但这类情况通常不会对系统造成严重问题。

**KERN\_NOTICE**：有必要进行提示的正常情形。许多与安全相关的状况用这个级别进行汇报。

**KERN\_INFO**：提示信息，很多驱动程序在启动的时候以这个级别来打印出它们找到的硬件信息。

**KERN\_DEBUG**：用于调试信息。

每个宏变量表示的是一个整数值，范围是 0—7，数值越小级别越高。

未指定打印级别的 `printk` 语句采用的默认级别 (`KERN_WARNING`)，根据打印级别，内核可能会把消息打印到当前控制台上，这个控制台可以是一个字符模式的终端、一个串口打印机或是一个并口打印机。当打印级别小于 `console_loglevel` 这个变量的值时消息才能显示出来，而且每次输出一行。如果系统同时运行了 `klogd` 和 `syslogd` (两个处理消息的进程)，则无论 `console_loglevel` 为何值，内核消息都将追加到用户空间中。变量 `console_loglevel` 的初始值为 `DEFAULT_CONSOLE_LOGLEVEL`，而且可以通过 `sys_syslog` 系统调用进行修改。

`Printk` 函数的工作机理是：将消息写到一个固定长度的缓冲区中，然后唤醒任何等待消息的进程，即那些调用消息打印的睡眠进程 (例如上面提到的 `klogd` 和 `syslogd`)，这些进程将把消息从缓冲去取走并输出到相应的输出终端上。如果缓冲区填满了，`printk` 就绕回缓冲区开始处填写新的数据，这将覆盖最陈旧的数据，于是这部分数据就会丢失。但与使用循环缓冲区所带来的好处相比，这个问题可以忽略不计。**Linux** 消息处理方法的另一个特点是，可以在源代码的任何地方调用 `printk`，甚至在中断处理函数里也可以调用，而且对数据量的大小没有限制。

有时候因为添加了 `printk` 打印语句而出现了上千条的消息，从而让信息充满了控制台，更可能使缓冲区产生溢出。如果使用某个慢速控制台设备 (如串口)，过高的消息输出速度会导致系统变慢，甚至使系统无法正常响应。当控制台被无休止的数据填充时，其实很难发现系统到底出现了什么问题。因此，我们应该非常小心地管理自己的打印信息。

**注意：**由于我们在对 **ARM** 学习中习惯了用 **ADS** 的 **AXD** 调试器，所以很容易想用 **AXD** 来单步调试 **Linux** 内核。这个办法根本行不通，因为在 **Linux** 内核开启 **MMU** 之后，**AXD** 所显示的存储器值就失去意义，并且 **AXD** 会提示出错。

除了上面所述的调试方法以外，我们还可以结合其它的方法来更好的定位错误，例如：通过 `/proc` 目录里提供的数据。`/proc` 文件系统是一种特殊的、由内核创建的文件系统，内核使用它向外界导出信息。`/proc` 下面的每个文件都绑定于一个内核函数，用户读取其中的文件时，该函数动态地生成文件的内容。在 **Linux** 系统中对 `/proc` 的使用很频繁，现在 **Linux**

发行版中很多工具都是通过 `/proc` 来获取它们需要的内核信息。`/proc` 文件不仅可以用于读出数据，也可以用于写入数据。最新的 Linux 内核又产生了一个 `/sys` 目录，它采用的是 `sysfs` 文件系统，用来记录设备驱动的信息。

## 5.5 分析 Linux 常用配置

本节将会根据 `at91rm9200dk_defconfig` 默认配置简单地介绍 Linux 的内核配置中的常用选项。Linux 中的选项有很多很多，要想完整地叙述是不现实的，我们在应用的过程中对添加一项功能需要增加哪些选项也是不断摸索的。这个完全靠不断地积累而成，最有用的帮助文件是源代码中 `/documentation` 目录里的文件。

“#”表示注释，“y”表示选中，未选中的都被注释掉。

```
体系结构的配置
CONFIG_ARM=y
CONFIG_MMU=y
ARM 体系结构并且带有 MMU 单元。
```

```
代码成熟度配置
CONFIG_EXPERIMENTAL=y
```

因为源代码中有些部分是未经过严格测试的，如果想体验一下这些未成熟的功能可以选择 y。

```
模块支持
Loadable module support
#
CONFIG_MODULES=y
CONFIG_MODULE_UNLOAD=y
设置内核可以加载模块也可以卸载模块。
```

```
处理器类型以及平台选择
System Type
#
CONFIG_ARCH_AT91=y
CONFIG_ARCH_AT91RM9200=y
CONFIG_ARCH_AT91RM9200DK=y
设置处理器的类型，以及所使用的硬件平台的名称。
```

```
CONFIG_AT91_PROGRAMMABLE_CLOCKS=y
CONFIG_CPU_32=y
CONFIG_CPU_ARM920T=y
CONFIG_CPU_32v4=y
CONFIG_CPU_ABRT_EV4T=y
CONFIG_CPU_CACHE_V4WT=y
CONFIG_CPU_CACHE_VIVT=y
CONFIG_CPU_COPY_V4WB=y
CONFIG_CPU_TLB_V4WBI=y
```

设置 AT91RM9200 的各项内部功能，包括：可编程的时钟，32 位指令集，ARM920T 核，以及 V4 架构的高速环存(CACHE)和转换后备缓冲区（TLB）等。

```
启动项配置
Boot options
```

```
#
CONFIG_CMDLINE="mem=32M console=ttyS0,115200 initrd=0x20410000,3145728
root=/dev/ram0 rw"
```

这里可以配置内核的启动参数，具体的启动参数格式可以在 `/documentation/kernel-parameters` 文件中找到。这里的 `CONFIG_CMDLINE` 告诉内核的信息是：系统内存 32MB，控制台是 `ttyS0`（由代码中决定 `ttyS0` 到底指向哪个串口），传输速率为 115200bit/s，根文件系统在内存中的物理地址是：0x20410000，压缩后的根文件系统的大小是 3M，根文件系统的类型是可读写的 `ramdisk`。

```
#用户可执行程序的格式
Userspace binary formats
#
```

```
CONFIG_BINFMT_ELF=y
```

用户可执行程序的格式是 ELF，也可以根据需要进行选择 AOUT 格式（老的格式）。

```
CONFIG_NET=y
```

选择网络功能支持。

```
CONFIG_MTD=y
```

选择 MTD（Memory Technology Device，存储技术设备）支持，如果要开发 ROM 和 Flash 的驱动就必须要有这项功能支持。MTD 子系统的配置相当的多，具体的可以根据自己的需要来选择。

```
CONFIG_BLK_DEV_RAM=y
CONFIG_BLK_DEV_RAM_COUNT=16
CONFIG_BLK_DEV_RAM_SIZE=8192
CONFIG_BLK_DEV_INITRD=y
```

块设备支持选项，这里表示系统支持 RAM 块设备，主要是为了支持 `ramdisk` 的根文件系统，意思是：把 RAM 中的一部分当作一个块设备，并且把它作为最终保存根文件系统的介质。这里的 `ramdisk` 的数量是 16 个，大小是 8M，且支持压缩的 `ramdisk` 根文件系统。

```
CONFIG_NETDEVICES=y
```

选择网络器件支持，这里定义了很多网络驱动程序，你可以根据需要进行选择。

```
CONFIG_NET_ETHERNET=y
CONFIG_MII=y
CONFIG_ARM_AT91_ETHER=y
```

选择网络硬件以及驱动程序。

每个8M还是一共8M？

```
CONFIG_INPUT=y
CONFIG_INPUT_MOUSEDEV=y
CONFIG_INPUT_MOUSEDEV_SCREEN_X=1024
CONFIG_INPUT_MOUSEDEV_SCREEN_Y=768
```

支持输入设备（鼠标）。

```
CONFIG_VT=y
CONFIG_VT_CONSOLE=y
CONFIG_HW_CONSOLE=y
```

支持虚拟终端，就是可以将显示器或者键盘作为终端使用，所谓虚拟终端就是将一个物理终端虚拟成几个终端，这里可以将一个虚拟终端作为控制台使用，内核的各种打印信息就可以输出到这个控制台上。

```
CONFIG_SERIAL_ATMEL=y
CONFIG_SERIAL_ATMEL_CONSOLE=y
CONFIG_SERIAL_CORE=y
```

```
CONFIG_SERIAL_CORE_CONSOLE=y
```

串口驱动选择，这里表示选择 ATMEL 的串口驱动，并且系统支持 TTY 串口协议。

```
CONFIG_WATCHDOG=y
```

```
CONFIG_WATCHDOG_NOWAYOUT=y
```

```
CONFIG_AT91RM9200_WATCHDOG=y
```

选择 Watchdog 功能支持。

```
CONFIG_USB_ARCH_HAS_HCD=y
```

```
CONFIG_USB_ARCH_HAS_OHCI=y
```

```
CONFIG_USB=y
```

```
CONFIG_USB_DEBUG=y
```

```
CONFIG_USB_OHCI_HCD=y
```

USB 选项的支持，具体的配置可以参考 USB Mass Storage 驱动。

```
CONFIG_MMC=y
```

```
CONFIG_MMC_BLOCK=y
```

```
CONFIG_MMC_AT91RM9200=y
```

MMC (Multi Media Card, 多媒体卡) 支持，具体配置可以参考 SD 卡驱动。

```
CONFIG_EXT2_FS=y
```

```
CONFIG_EXT2_FS_XATTR is not set
```

```
CONFIG_EXT2_FS_XIP is not set
```

```
CONFIG_EXT3_FS is not set
```

```
CONFIG_JBD is not set
```

```
CONFIG_REISERFS_FS is not set
```

```
CONFIG_JFS_FS is not set
```

```
CONFIG_FS_POSIX_ACL is not set
```

```
CONFIG_XFS_FS is not set
```

```
CONFIG_MINIX_FS is not set
```

```
CONFIG_ROMFS_FS is not set
```

```
CONFIG_INOTIFY=y
```

```
CONFIG_QUOTA is not set
```

```
CONFIG_DNOTIFY=y
```

```
CONFIG_AUTOFS_FS is not set
```

```
CONFIG_AUTOFS4_FS is not set
```

```
CONFIG_FUSE_FS is not set
```

```
CONFIG_ISO9660_FS is not set
```

```
CONFIG_UDF_FS is not set
```

```
CONFIG_MSDOS_FS is not set
```

```
CONFIG_VFAT_FS is not set
```

```
CONFIG_NTFS_FS is not set
```

```
CONFIG_PROC_FS=y
```

```
CONFIG_SYSFS=y
```

```
CONFIG_TMPFS=y
```

```
CONFIG_HUGETLB_PAGE is not set
```

```
CONFIG_RAMFS=y
```

```
CONFIG_RELAYFS_FS is not set
```

```
CONFIG_CRAMFS=y
```

```
CONFIG_NFS_FS is not set
```

这里是 Linux 支持的各项文件系统，常用的配置只有 EXT2，如果要读取光盘上的数据就要选择 CONFIG\_ISO9660\_FS 选项。如果需要打开 Windows 的文件，就要选择 CONFIG\_MSDOS\_FS, CONFIG\_VFAT\_FS, CONFIG\_NTFS\_FS 选项。为了能够方便调试，或者用户程序能够从 /proc、/sys 中获得系统信息，还必须选择 CONFIG\_PROC\_FS, CONFIG\_SYSFS。

以上这些选项只是我们比较常用的选项，有很多的选项都没有写出来，具体可以参考



at91rm9200dk\_defconfig，有关每个选项的意义可以参考配置时的“help”选项。

**参考资料：**

1. << 构建嵌入式 Linux 系统>>。
2. <<Linux 内核完全注释>>。
3. <<深入理解 Linux 内核>>。
4. Linux-2.6.19.2 源代码。

## 第六章 根文件系统的制作

什么是根文件系统，有什么作用？  
组织结构如何？

建立嵌入式系统的根文件系统是构建目标板 Linux 系统的重要步骤，嵌入式 Linux 系统的根文件系统与运行 Linux 的 PC 机的根文件系统类似，不过嵌入式 Linux 系统只包含运行系统必需的应用程序、链接库和相关文件的最小集合。

建立目标根文件系统的时候，采用反复尝试的方式，陆续加入需要的工具程序和链接库。使用这种“探索”根文件系统建立的方法，能够为目标板系统建立完整的根文件系统。

由于嵌入式 Linux 系统一般内存和存储器容量都有限，文件一般要求小而简单。通常很少采用 Linux 的标准文件系统 Ext2 或者 Ext3 等一些复杂文件系统。其次，系统的使用场合不同，对于文件系统要求的数据保存能力和掉电安全性要求也不一样。

### 6.1 常用文件系统介绍

#### 6.1.1 Romfs 文件系统

Romfs 是使用在 Linux 上的一种很小而高效的只读文件系统，它是基于块设备的文件系统，用在磁盘、光盘或 Rom 仿真块驱动上。但不支持 Linux 的 MTD（Memory Technology Device 存储管理技术）接口，所以不能直接应用在嵌入式 Rom 设备上。要想应用在嵌入式 Rom 上，就要开发基于 Rom 的块设备驱动程序。由于 Romfs 文件系统的尺寸小而简单，而且文件数据在 Romfs 上顺序存储，允许程序在基于 Romfs 文件的设备上直接运行。所以经常被修改后用于嵌入设备上，如 uCLinux，就采用 Romfs 文件系统作为它的根文件系统格式。

#### 6.1.2 Ext2 on ramdisk 文件系统

Ext2 第二代扩展文件系统是在 Linux 上设计最成功的文件系统，其目标是为 Linux 提供一个强大的可扩展文件系统。Ext2 文件系统的稳定性、可靠性和健壮性，几乎在所有基于 Linux 的系统上都使用。但在嵌入式设备中使用时，该文件系统复杂庞大、没有掉电保护功能，在异常断电下，可能造成数据的丢失。

Ext2 on ramdisk 是在内存中的 Ramdisk 上创建的 Ext2 文件系统，这种方法既有内存文件系统的快速访问性，同时具有 Ext2 文件系统的优点，而且可以压缩后存放在 Flash 上，可以节省 Flash 空间，缺点就是系统体积大，数据在掉电后，无法保存。

#### 6.1.3 Jffs2 文件系统

Jffs2 是在瑞典 Axis 公司开发的 Jffs 基础上对其进行改进，设计的一种专门用于嵌入式系统中的 Flash 设备的文件系统。

Jffs2 的写文件方式：在文件的某个数据区中的数据进行了修改后，不是直接把修改后的数据写到原来的地址，而是在 Flash 上的日志尾部申请一块新的可用区域来存放修改后的数据，并把原来的数据区表示为“脏”区。这样就能够把文件的变化快速记录下来，实现了

系统的防掉电功能。同时它有效地结合了 Flash 的读写特性，如果要把修改后的结果直接写回原处，就首先要进行擦除原来的数据，然后再写数据，而现在只写到日志尾部，只是到必要的时候才进行擦除回写，这样就能够提高系统的效率。当 Flash 上的“脏”数据块达到一定的数量，或者系统中的可用空间小于一定数量，系统会自动执行一个清理程序，擦除“脏”区，修改后的数据写回，释放日志占用的空间。

- Jffs2 上执行 Flash 擦除/写/读操作要比 Ext2 文件系统快。
- Jffs2 采用附加文件而不是重写整个扇区，具有掉电安全保护性。
- Jffs2 是专门为 Flash 芯片那样的嵌入设备设计，它提供了更好的 Flash 管理能力。

### 6.1.4 Yaffs2 文件系统

Yaffs (yet another flash file system) 是目前唯一专门为 NAND flash 设计的应用于嵌入式系统的文件系统，具有很好的可移植性。JFFS2 文件系统能够很好的用在 NOR Flash 设备中，而 YAFFS 则是专门为 NAND 设备设计的，它是一种类似于 JFFS2 的日志型文件系统。

YAFFS 充分考虑了 NAND 闪存的特点，根据 NAND 闪存以页面为单位存取的特点，将文件组织成固定大小的数据段。YAFFS2 是 YAFFS 的改进版本，在速度、内存使用上，对 NAND 设备的支持都有改善。YAFFS2 还支持大页面的 NAND 设备，并且对大页面的 NAND 设备作了优化。

YAFFS 相对于 JFFS2，减少了一些功能，因此速度更快、占用内存更少，并且对 NAND Flash 具有很好的保护机制，能够延长 NAND Flash 的使用寿命。因此，YAFFS 文件系统的出现，使得价格低廉的 NAND Flash 芯片具有高效性和健壮性。目前，大容量的 NAND 闪存器件和基于它的 YAFFS/YAFFS2 文件系统正越来越广泛地运用在嵌入式系统中。

## 6.2 Ext2 on Ramdisk 文件系统制作流程

### 6.2.1 Ramdisk 介绍

Ramdisk 就是把指定的内存区域模拟成磁盘设备，它属于块设备驱动程序，它不是一个文件系统。在 Linux 中，可以把系统内存中的一部分仿真成一个硬盘块，在它上面可以实现各种基于内存的文件系统，如 Ext2 on Ramdisk。Ramdisk 的最大特点是运行速度快，因为文件内容全部保存在内存中，所以可以用这个方法提高文件访问速度。如要创建一个快速访问的 web server，就可以考虑把 web 服务器的 web 根目录直接采用 Ramdisk 技术创建在内存中。

### 6.2.2 制作 Ext2 on Ramdisk 根文件系统

可以加上这里所用的 Linux 的常用指令

#### 1 构建 Linux 必需目录

为了建立根文件系统，首先在 home 目录下建立 rootfs 目录，作为放置根文件系统的目录。

```
cd /home/
mkdir rootfs
cd rootfs
```

指PC的Linux系统下  
进行操作

现在我们可以在此目录下建立根文件系统所必需的顶层目录：

```
mkdir bin dev etc lib proc sbin mnt
```

这里 mnt 目录不是必选的，之所以建立是为了下面挂载 YAFFS2 文件系统用的。另外，bin、sbin 目录下的内容是通过 busybox 工具生成的，所以将在下面介绍这里先略过。

### (1) /dev 目录设计

/dev 目录中包含了所有 Linux 系统中使用的外部设备。但是并不是放外部设备的驱动程序。它实际上是一个访问这些外部设备的端口。在 Linux 中它被看作一种特殊的文件，只能用 mknod 命令去创建。

在使用 mknod 命令手工创建这些设备文件时，需要知道设备的主设备号和从设备号，否则设备就不能正确使用。设备的主、次设备号可以查阅内核源码树的

Documentation/devices.txt。创建 mem、console、tty0、tty、ram 设备文件命令如下：

```
mknod -m 600 mem c 1 1
mknod -m 600 console c 5 1
mknod -m 600 tty0 c 4 0
mknod -m 666 tty c 5 0
mknod -m 666 ram b 1 0
mknod -m 600 ttyS0 c 4 64
```

(命令的第三列数字代表权限，c 代表创建的是字符设备，第 6 列数字代表主设备号，最后一列数字代表从设备号)

在嵌入式文件系统中，/dev 目录下必须要有 mem、console、tty0、tty、ram 等设备文件。

### (2) /etc 目录的设计

该目录包含了重要的系统配置文件。Busybox 源代码 example/bootfloppy/etc 目录中的文件算是一个简单的例子，可以把其中的文件拷贝过来作为基础。在嵌入式应用中，下面三个文件一般是必备的，其它的文件都是要根据自己的系统需求来添加。

#### A. init.d/rcS 脚本：系统启动与改变 runlevel 的 scripts

Linux 系统中许多后台服务都从这个脚本启动，在这个脚本中可以添加自己的任务。一般都是现存的系统脚本，然后删减无关的服务进程。

将该脚本文件内容修改成如下行：

```
#脚本头
#!/bin/sh
#以读写模式重新安装根文件系统（需要/etc/fstab）
/bin/mount -n -o remount,rw /
#安装/proc 文件系统
/bin/mount /proc
#设置网卡 MAC 地址
/sbin/ifconfig eth0 hw ether 12:34:56:78:90:aa
#设置网卡 IP 地址
/sbin/ifconfig eth0 192.168.0.11
```

这就是个简单的启动脚本，上面这个初始化命令行要能够正常执行，目标板的根文件系统中必须存在/etc/fstab 文件。

#### B. fstab 文件：要安装的文件系统清单

fstab 主要给出系统在启动时，要安装的文件系统清单，其中默认必须有 proc 文件系统。将/etc/fstab 文件内容修改成如下行：

```
/etc/fstab
```

chmod -x

```
设备 目录 类型 选项
#
/dev/ram / ext2 defaults 0 1
none /proc proc defaults 0 0
```

此例中可以看出，通过 ramdisk 安装目标板的根文件系统。

介绍Busybox

### C. inittab 文件：提供初始化进程的参数

init 是 Linux 系统上运行的所有其它进程的父进程。除了缺省支持的命令，busybox 还提供与 init 类似的能力。如同原始的主流 init，busybox 也可以处理系统的启动工作。Busybox 尤其适合在嵌入式系统中使用，因为它可以为嵌入式系统提供所需要的大部分 init 功能。

BusyBox 的 init 进程会依次进行以下工作：

- 为 init 设置信号处理进程
- 初始化控制台
- 剖析 inittab 文件、/etc/inittab 文件
- 执行系统初始化命令行。Busybox 在缺省情况下会使用/etc/init.d/rcS
- 执行所有会导致 init 暂停的 inittab 命令
- 执行所有仅执行一次的 inittab 命令

一旦完成以上工作，init 进程便会循环执行以下工作：

- 执行所有终止时必须重新启动的 inittab 命令（动作类型：respawn）。
- 执行所有终止时必须重新启动但启动前必须先询问用户的 inittab 命令（动作类型：askfirst）。

Inittab 文件中每一行的格式如下所示：

Id: runlevel: action: process

对 busybox 而言，id 用来指定所启动进程的控制 tty；busybox 会完全忽略 runlevel 字段，所以可以将它空着不填。Process 字段用来指定所执行程序的路径，包括命令行选项。Action 字段用来指定下表所列八个可应用到 process 的动作之一。

| 动作         | 结果                                                                                                                      |
|------------|-------------------------------------------------------------------------------------------------------------------------|
| sysinit    | 为 init 提供初始化命令行的路径                                                                                                      |
| respawn    | 每当相应的进程终止执行便重新启动                                                                                                        |
| askfirst   | 类似于 respawn，主要用途是减少系统上执行的终端应用程序的数量，会在控制台上显示 “Please press Enter to active this console” 信息，并在系统重新启动进程之前等待用户按下 “Enter” 键 |
| Wait       | Wait 动作会通知 init 必须等到相应的进程执行完之后才能继续执行其他的动作                                                                               |
| Once       | 进程只执行一次，而且不会等待它完成                                                                                                       |
| Ctrlaltdel | 当按下 Ctrl-Alt-Delete 组合键时运行的进程                                                                                           |
| Shutdown   | 当系统关机时运行的进程                                                                                                             |
| restart    | 当 init 进程重新启动的时候执行的进程，事实上就是 init 本身                                                                                     |

系统中使用的 inittab 文件如下：

```
::sysinit:/etc/init.d/rcS
::askfirst:/bin/sh
::restart:/sbin/init
::ctrlaltdel:/sbin/reboot
::shutdown:/sbin/swapoff -a
::shutdown:/bin/umount -a -r
```

其中第一行指定了系统启动时执行的第一个脚本文件；第二行指定在控制台启动一个“askfirst”shell；第三行指定等待重新启动 init 进程；第四行指定了当按下 ctrl+alt+del 组合键时的执行命令；最后两行指定了关机时执行的操作。

### (3) /lib 目录的设计

在/lib 目录中，存放着必须的共享库与装载程序。如果程序无法在/lib 目录中找到对应的库，程序将无法运行。

## 2 busybox 制作

### (1) busybox 工程介绍

Busybox 是一个很成功的开源软件，它是许多嵌入式 Linux 系统的基石。它用极小型的应用程序来提供上述位于/bin 和/sbin 中的整个命令集的功能，它还支持动态和静态方式链接 glibc，允许根据需要修改缺省配置，选中或移除所包含的命令。Busybox 提供的配置界面与 Linux 菜单配置界面非常相似。使用 busybox 可以大大简化制作嵌入式系统根文件系统的过程，所以 busybox 工具在嵌入式开发中得到了广泛的应用。

### (2) 配置编译 busybox

A. 下载 busybox。最新版本的 busybox 可以在官方网站 [www.busybox.net](http://www.busybox.net) 下载，此处以 busybox-1.01.tar.gz 为例来说明。

B. 拷贝 busybox 源码压缩文件到指定目录，并解压

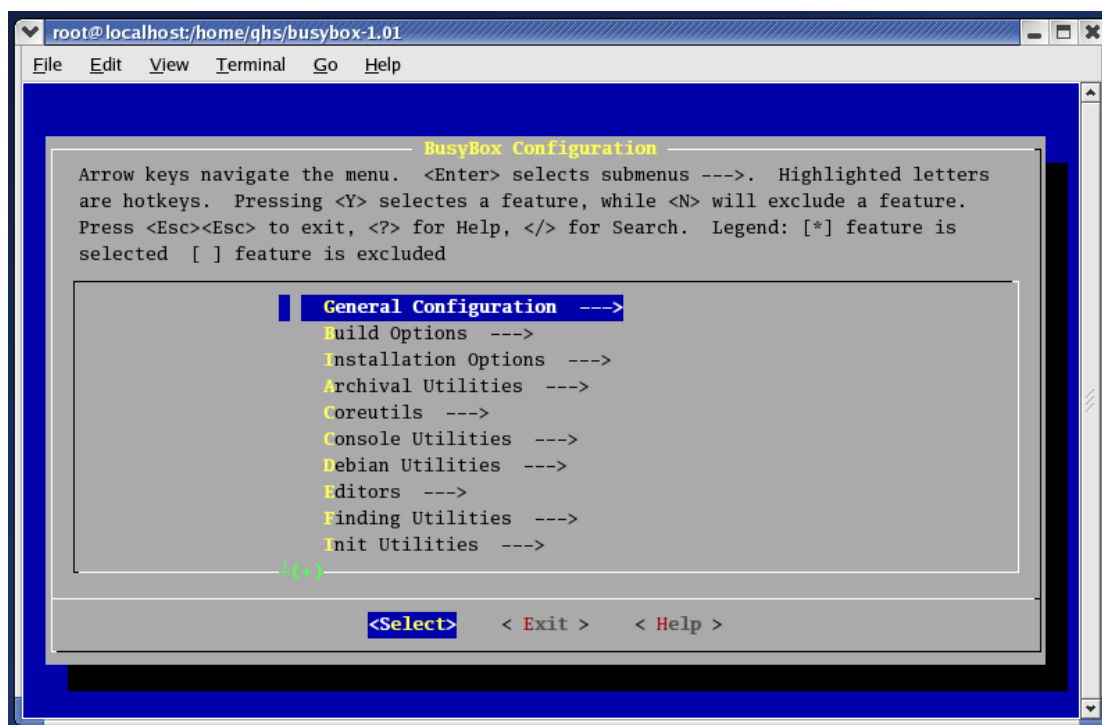
```
tar zxvf busybox-1.01.tar.gz
```

```
cd busybox-1.01
```

C. 对 busybox 进行配置，运行 make menuconfig 命令

```
make menuconfig
```

配置界面如下：



具体配置如下：

#### Build Option ---→

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

[\*] Build with Large File Support (for accessing files > 2 GB)



[\*] Do you want to build BusyBox with a Cross Compiler?  
 (/usr/local/arm/3.4.1/bin/arm-linux-) Cross Compiler prefix

### Installation Options ---→

[\*] Don't use /usr

(./\_install) BusyBox installation prefix

Build Option 中第一个选项指定把 busybox 编译成静态链接的可执行文件，运行时独立于其他函数库，否则必需要其他共享库才能运行，此外采用静态编译也可以大大减少磁盘使用空间。第三个选项指定 busybox 进行交叉编译，这里默认的交叉编译路径不是我们所合适的，需要我们点击进去更改交叉编译器路径如上所示。Installation Options 中第一项推荐选上，因为不选的话 busybox 将默认安装到系统的/usr 目录下面，这将覆盖掉系统原有的命令。

D. 编译、安装

# make

# make install

当安装命令执行完成后会在 busybox 目录下生成一个\_install 的目录，里面有 bin 目录、sbin 目录和指向 busybox 可执行文件的链接 linuxrc，将这些文件一起移至先前建立的/home/rootfs/目录下。这样根文件系统所必需目录的内容就制作完成了。

3 建立 Ramdisk

A. 在宿主机上建立 loop 设备的临时挂接点；

# mkdir /mnt/loop

B. 创建空的文件系统映像

dd if=/dev/zero of=/home/myramdisk bs=1k count=8192

此命令在目录/home 下建立了 8M 的文件系统映像 myramdisk，从设备文件/dev/zero 中读取数据填充文件 myramdisk，以便后面得到更高的压缩率；

C. 将此 8M 的 ramdisk 空间格式化为 ext2 类型的文件系统；

# mke2fs -F -v -m0 /home/myramdisk

D. 将已格式化 ext2 类型文件系统的 ramdisk 挂载至目录/mnt/loop 上；

# mount -o loop /home/myramdisk /mnt/loop

这样，/mnt/loop 目录就对应 myramdisk 存储设备了。

E. 将先前制作好的/home/rootfs 目录下根文件系统的内容拷贝至/mnt/loop 目录下（这就相当于拷贝至文件 myramdisk 中了）；

# cp -av /home/rootfs/\* /mnt/loop

F. 卸载掉/mnt/loop；

# umount /mnt/loop

G. 压缩映像

# gzip -v9 /home/myramdisk

这样一个压缩的 ramdisk 映像就制作完成了，最后将生成的 myramdisk.gz 下载到 Flash 相应的分区中。使用 ramdisk 还需要配置内核命令行参数。

这里的根文件系统是最基本的，容量大概在 700K~1M 字节左右。在以后应用程序开发的过程中我们肯定需要大量的动态、静态链接库。把这些库放入相应的目录以后，根文件系统将会变大。最基本的动态、静态链接库在相应的交叉编译器的目录下（例如 3.4.1 交叉编译器的库在 ./3.4.1/arm-linux/lib 目录下），其它的库在制作过程中会要求你放到相应的根文件系统目录里的。

## 6.3 Yaffs2 文件系统的制作流程

文件系统需要内核  
重新增加选项编译  
以支持

### 6.3.1 Yaffs2 文件系统作为普通文件系统的制作

1. 由于 Linux 内核不支持 YAFFS2 文件系统，首先通过下面的网址下载 YAFFS2 源码，  
<http://www.aleph1.co.uk/cgi-bin/viewcvs.cgi/yaffs2.tar.gz?view=tar>

2. 拷贝 YAFFS2 源码压缩文件到指定目录，解压并加入内核

```
tar zxvf yaffs2.tar.gz
```

```
cd yaffs2
```

```
./patch-ker.sh /home/linux-2.6.20.7
```

这里假定内核 linux-2.6.20.7 在目录/home/ linux-2.6.20.7 下，通过上面的命令，YAFFS2 已经加入到了内核，在内核的 fs 目录下会发现多了个 yaffs2 目录，在这个目录下是 yaffs2 的相关内容。

3. 针对硬件平台，修改内核源码

(1) 修改/home/linux-2.6.20.7/arch/arm/mach-at91rm9200/board-dk.c

代码修改如下：

```
static struct at91_nand_data __initdata dk_nand_data = {
 .ale = 22,
 .cle = 21,
 //.det_pin = AT91_PIN_PB1,
 .rdy_pin = AT91_PIN_PC14,
 .enable_pin = AT91_PIN_PC15,
 .partition_info = nand_partitions,
};
```

这里将整个 NAND Flash 作为一个分区，挂载在根文件系统的一个目录下，用于保存需要动态存储的数据，而没有对其进行分区设置。

(2) 修改/home/linux-2.6.20.7/arch/arm/mach-at91rm9200/at91rm9200\_devices.c

代码修改如下：

```
static struct resource nand_resources[] = {
 {
 .start= NAND_BASE,
 .end = NAND_BASE + SZ_32M - 1,
 .flags = IORESOURCE_MEM,
 }
};
```

4. 配置内核增加 MTD 和 YAFFS2 文件系统的支持

Device Drivers --->

Memory Technology Devices (MTD) --->

<\*> Memory Technology Device (MTD) support

[\*] MTD partitioning support

--- User Modules And Translation Layers

<\*> Direct char device access to MTD devices



<\*> Caching block device access to MTD devices

NAND Flash Device Drivers --->

<\*> NAND Device Support

[\*] Support for NAND Flash / SmartMedia on AT91

File systems --->

Miscellaneous filesystems --->

<\*> YAFFS2 file system support

修改根文件系统

## 5. 添加 MTD 设备

由于将 NAND Flash 从整体上作为一个 MTD 分区，用于保存用户数据。所以在根文件系统的/dev 目录下添加 mtd0、mtdblock0 设备文件。这里 mtd0、mtdblock0 对应整个 NAND Flash 空间。Mtd0 是字符设备，用于接收二进制映像文件到 Flash 设备上；mtdblock0 是安装文件系统所需要的块设备。字符设备与块设备必须成对添加，需要说明的是：这里 MTD 字符设备的主设备号为 90，次设备号为 0、2...，如果次设备号为奇数，则表明这是只读设备，只有次设备号是偶数时才表示设备可读写。

在/dev 目录下，执行以下命令，添加设备文件，

```
mknod -m 666 mtd0 c 90 0
```

```
#mknod -m 666 mtdblock0 b 31 0
```

## 6. YAFFS2 文件系统的挂接与测试

将u-boot、重新编译的内核映像、Ramdisk根文件系统映像分别写入到Flash中的指定区域后，并在u-boot中设置相应的自启动参数，重新启动目标板内核，可以看到与YAFFS2相关的启动信息如下：

```
NAND device: Manufacturer ID: 0xec, Chip ID: 0x75 (Samsung NAND 32MiB 3,3V 8-bit)
```

```
Scanning device for bad blocks
```

```
Creating 1 MTD partitions on "NAND 32MiB 3,3V 8-bit":
```

```
0x00000000-0x02000000 : "NAND Partition 1"
```

从启动信息中可知，系统已经检测到了 NAND Flash，另外还可以在进入系统后通过下面的命令查看分区信息：

```
cat /proc/mtd
```

如果出现上述内容，则说明 YAFFS2 文件系统已经配置成功了。

在目标板的 Linux 系统启动后，执行 mount -t yaffs2 /dev/mtdblock0 /mnt 命令后，就建立起了 YAFFS2 文件系统，可以在挂接的/mnt 目录下建立删除目录、文件等操作。如果想在系统启动时，自动挂载建好的 yaffs2 文件系统，可以在启动脚本 rcS 文件里加入 mount -t yaffs2 /dev/mtdblock0 /mnt。当系统启动后，我们就可以直接通过访问/mnt 目录来实现对 NAND Flash 的读写了。

## 6.3.2 Yaffs2 文件系统作为根文件系统的制作

### 1. 拷贝根文件系统内容至 NAND Flash

当 YAFFS2 文件系统作为普通文件系统挂载成功时，通过下面命令将根文件系统内容拷贝到 NAND Flash

```
cp -av / bin dev etc lib proc sbin /mnt
```

这样根文件系统的内容就拷贝至 NAND Flash 中。

### 2. 修改 fstab 文件，将 yaffs2 文件系统安装为根文件系统

/etc/fstab 文件如下：

```
/dev/mtdblk0 / yaffs defaults 0 1
none /proc proc defaults 0 0
```

3. 设置启动参数，使系统从 NAND Flash 中安装根文件系统。

在 U-BOOT 中设置启动参数命令如下：

```
Setenv bootargs noinitrd root=/dev/mtdblock0 console=ttyS0
```

## 第七章 linux 设备驱动

本手册不是一个教科书类型的材料，而是针对实验室 arm9 平台的使用说明，所以，以下对于一些原理性和基础性的内容只作简单的介绍，但会注明可以参考的书目。本章节所有代码均通过实际测试，平台如下：上位机系统 Redhat9.0，交叉编译环境 3.4.1，下位机版本 linux2.6.19.2

### 7.1 驱动开发概要

#### 7.1.1 驱动概述

我们在学习驱动代码之前，先了解一下驱动的作用，讲到驱动，就不得不说到驱动的使用者，这在初学者是必须首先知道的内容。

驱动只是一个接口，它被添加进内核，但是这个接口本身并不会(通常情况下)按照用户的意愿完成一系列的操作，例如，一个 USB 摄像头的驱动，它提供的是一个从 CPU 向 USB 接口获取图像数据的路径，但驱动本身并不会自动完成打开摄像头，捕捉图像，修改图像分辨率等一系列的动作。那么这些动作需要怎样产生呢，这就是应用程序要做的事情，应用程序通过接口函数调用（这些接口函数将在下节介绍）控制硬件动作。还是以摄像头为例，我们在系统下运行摄像头应用程序，应用程序将提供一个交互界面，比如我们选择“捕捉”选项，应用程序将调用设备驱动，完成摄像头的打开和数据传输。

所以,设备驱动程序是操作系统内核和机器硬件之间的接口。设备驱动程序为应用程序屏蔽了硬件的细节，这样在应用程序看来，硬件设备只是一个设备文件，应用程序可以象操作普通文件一样对硬件设备进行操作。

设备驱动程序是内核的一部分（有时候可以动态地加载），它完成以下的功能：

- 1.对设备初始化和释放。
- 2.把数据从内核传送到硬件和从硬件读取数据。
- 3.读取应用程序传送给设备文件的数据和回送应用程序请求的数据。
- 4.检测和处理设备出现的错误。

Linux 内核把驱动程序划分为 3 种类型：

- 字符设备
- 块设备
- 网络设备

其中，字符设备和块设备可以像文件一样被访问。对于大多数字符设备来说，它提供给用户（应用程序）一个类似流控制的设备节点，即通常的字符设备都是顺序写入和读取的，它为操作提供优化；块设备一般作为文件系统的宿主，它为存储提供优化。

两者的区别是，在对字符设备发出读/写请求时，实际的硬件 I/O 一般就紧接着发生了，块设备则不然，它利用一块系统内存作缓冲区，当用户进程对设备请求能满足用户的要求，就返回请求的数据，如果不能，就调用请求函数来进行实际的 I/O 操作。块设备是主要针对磁盘等慢速设备设计的，以免耗费过多的 CPU 时间来等待。有些设备在 linux 系统中，既有对应的字符设备驱动程序，也有对应的块设备驱动程序。比如 MTD（Memory Technology

Device 存储管理技术) 设备和文件系统。

网络设备在 linux 中是一类比较特殊的设备, 它不像字符设备或块设备那样通过对应的设备文件节点去访问, 也不通过常用的接口调用访问(下节介绍调用接口)。关于网络设备的详细流程和描述, 请阅读参考文献 1。

由于字符设备和块设备的架构及接口类似, 我们在这一章的介绍将针对字符设备展开, 块设备的细节差异请阅读参考文献 1 的相关部分。

具备这些基础知识, 我们可以进行下一节的学习。

## 7.1.2 相关命令

操作系统通过各种驱动程序来驾驭硬件设备, 它为用户屏蔽了各种各样的设备, 驱动硬件是操作系统最基本的功能, 并且提供统一的操作方式。正如我们查看屏幕上的文档时, 不用去管到底使用 nVIDIA 芯片, 还是 ATI 芯片的显示卡, 只需知道输入命令后, 需要的文字就显示在屏幕上。硬件驱动程序是操作系统最基本的组成部分, 在 Linux 内核源程序中占有较高的比例。

Linux 内核中采用可加载的模块化设计(LKMs, Loadable Kernel Modules), 一般情况下编译的 Linux 内核是支持可插入式模块的, 也就是将最基本的核心代码编译在内核中, 其它的代码可以选择是在内核中, 或者编译为内核的模块文件。

如果需要某种功能, 比如需要访问一个温度传感器, 就加载相应的传感器模块。这种设计可以使内核文件不至于太大, 但是又可以支持很多的功能, 必要时动态地加载。这是一种跟微内核设计不太一样, 但却是切实可行的内核设计方案。

我们常见的驱动程序很多都是作为内核模块动态加载的, 而 Linux 最基础的驱动, 如 CPU、PCI 总线、TCP/IP 协议、APM(高级电源管理)、VFS 等驱动程序则编译在内核文件中。有时也把内核模块就叫做驱动程序, 只不过驱动的内容不一定是硬件罢了, 比如 ext3 文件系统的驱动。

理解这一点很重要。因此, 加载驱动就是加载内核模块。下面来看一下有关模块的命令, 在加载驱动程序要用到它们: lsmod、insmod、rmmod、modinfo。

### lsmod

lsmod 列出当前系统中加载的模块, 例如

```
[root@localhost root]# lsmod
Module Size Used by Tainted: PF
parport_pc 19076 1 (autoclean)
lp 8996 0 (autoclean)
parport 37056 1 (autoclean) [parport_pc lp]
nfsd 80176 8 (autoclean)
lockd 58704 1 (autoclean) [nfsd]
sunrpc 81564 1 (autoclean) [nfsd lockd]
autofs 13268 0 (autoclean) (unused)
ide-cd 35708 0 (autoclean)
cdrom 33728 0 (autoclean) [ide-cd]
vnhgfs 38380 4
vmxnet 8236 1
keybdev 2944 0 (unused)
mousedev 5492 1
hid 22148 0 (unused)
input 5856 0 [keybdev mousedev hid]
usb-uhci 26348 0 (unused)
usbcore 78784 1 [hid usb-uhci]
ext3 70784 2
jbd 51892 2 [ext3]
BusLogic 100796 3
sd_mod 13452 6
scsi_mod 107128 2 [BusLogic sd_mod]
```

图 7.1.1 lsmod 命令运行结果

上图 7.1.1 为在 RedHat9.0（以下简称 RH9）下的运行结果。上面显示了当前系统中加载的模块，左边数第一列是模块名，第二列是该模块大小，第三列则是该模块使用的数量。如果后面为 `unused`，则表示该模块当前没在使用。如果后面有 `autoclean`，则该模块可以被 `rmmod -a` 命令自动清洗。`rmmod -a` 命令会将目前有 `autoclean` 的模块卸载，如果这时候某个模块未被使用，则将该模块标记为 `autoclean`。如果在行尾的 `[]` 括号内有模块名称，则括号内的模块就依赖于该模块。例如

```
Input 5856 0 [keybdev mousedev hid]
```

其中 `keybdev mousedev hid` 模块就依赖于 `input` 模块。

### insmod

`insmod` 命令用来向系统加载所需要的模块，在后续的章节中我们将看到生成的模块以 `.o` 文件及 `.ko` 文件的形式存在，`insmod` 可以将这些模块文件添加进当前的系统中，比如，假如我们现在有一个模块文件叫 `hello.o`，在终端中键入：

```
Insmod hello.o
```

此时 我们再次键入 `lsmod`，效果如下：

```
[root@localhost root]# lsmod
Module Size Used by Tainted: PF
hello 748 0 (unused)
parport_pc 19076 1 (autoclean)
lp 8996 0 (autoclean)
parport 37056 1 (autoclean) [parport_pc lp]
nfsd 80176 8 (autoclean)
lockd 58704 1 (autoclean) [nfsd]
sunrpc 81564 1 (autoclean) [nfsd lockd]
autofs 13268 0 (autoclean) (unused)
ide-cd 35708 0 (autoclean)
cdrom 33728 0 (autoclean) [ide-cd]
vnhgfs 38380 4
vmxnet 8236 1
keybdev 2944 0 (unused)
mousedev 5492 1
hid 22148 0 (unused)
input 5856 0 [keybdev mousedev hid]
usb-uhci 26348 0 (unused)
usbcore 78784 1 [hid usb-uhci]
ext3 70784 2
jbd 51892 2 [ext3]
BusLogic 100796 3
sd_mod 13452 6
scsi_mod 107128 2 [BusLogic sd_mod]
```

图 7.1.2 加载模块后 Lsmod 命令效果

可见，我们已经向系统添加了名为 `hello` 的模块。

### rmmod

`rmmod` 命令用来从系统中移除相应的模块，但是，为了系统的安全性，该命令只能删除没有使用的模块，从图 7.2 中看到，`hello` 模块没有被使用（`unused`），此时，我们键入：

```
rmmod hello
```

此时再次 `lsmod`，得到效果如下：

```

[root@localhost root]# rmmod hello
[root@localhost root]# lsmod
Module Size Used by Tainted: PF
parport_pc 19076 1 (autoclean)
lp 8996 0 (autoclean)
parport 37056 1 (autoclean) [parport_pc lp]
nfsd 80176 8 (autoclean)
lockd 58704 1 (autoclean) [nfsd]
sunrpc 81564 1 (autoclean) [nfsd lockd]
autofs 13268 0 (autoclean) (unused)
ide-cd 35708 0 (autoclean)
cdrom 33728 0 (autoclean) [ide-cd]
vnhgfs 38380 4
vmxnet 8236 1
keybdev 2944 0 (unused)
mousedev 5492 1
hid 22148 0 (unused)
input 5856 0 [keybdev mousedev hid]
usb-uhci 26348 0 (unused)
usbcore 78784 1 [hid usb-uhci]
ext3 70784 2
jbd 51892 2 [ext3]
BusLogic 100796 3
sd_mod 13452 6
scsi_mod 107128 2 [BusLogic sd_mod]
[root@localhost root]#

```

图 7.1.3 rmmod 命令效果

Hello 模块已经被成功移除了。

### modinfo

Modinfo 命令用来查看模块的相关信息，例如：

```

[root@localhost root]# modinfo sd_mod
filename: /lib/modules/2.4.20-8/kernel/drivers/scsi/sd_mod.o
description: <none>
author: <none>
license: "GPL"

```

图 7.4 modinfo 命令效果

以上，我们了解了与驱动模块相关的几个常用命令，虽然只是简介，但凭着这些，我们就可以继续下面的学习内容。

## 7.1.3 主要接口

在之前的小节中，我们了解了 linux 下设备驱动的一些大体知识，在这一节中，我们将正式进入到驱动的代码学习，在整个过程中，我们会结合一个简单的设备驱动程序范例（Test 模块），对各个接口函数进行简单讲解，最后将完成设备驱动向操作系统的加载。同时编写对应的应用程序，从实践中理解 linux 下驱动程序的工作过程。

需要再次强调的是，在这里我们可能只涉及了设备驱动的某些方面，没有将所有的内容都一一介绍，但可以肯定的是，这里一定是最主要的部分，有了这些基础将会给后续的学习带来极大的方便。

一个简单的字符设备驱动主要可以看成 2 大块，分别是驱动的注册和卸载，以及文件操

作。其中，注册和卸载是驱动必须提供的接口，而文件操作则是驱动真正发挥作用的地方，各个不同的驱动，其根本的区别就是文件操作的不同。下面我们将对这两块进行介绍，在这之前，先来做一些必要的准备工作。

### 获得设备号

我想许多人肯定和我一样会有这样的疑问，linux 操作系统中有各种各样的设备驱动，我们在编写应用程序的时候怎样来确定调用的是那个驱动程序呢。是不是有一个特定的标识来唯一确定一个设备驱动呢。你想对了，在 linux 操作系统中，每个设备驱动都有唯一名字，同时对应唯一的号码，我们称为设备号，在后面的内容中可以看到，应用程序正是通过驱动名称（同时相当于通过设备号）来确定相应的驱动函数。

刚才说到，对字符设备的访问是通过文件系统内的设备名称进行。这些名称是设备文件的节点，它们通常位于/dev 目录。

我们在/dev 目录下执行 ls -l 命令，结果中包含如下项目：

|    |            |        |      |      |    |            |         |
|----|------------|--------|------|------|----|------------|---------|
| 1  | crw-rw-rw- | 1 root | tty  | 3,   | 87 | 2003-01-30 | ttyu7   |
| 2  | crw-rw---- | 1 root | uucp | 188, | 0  | 2003-01-30 | ttyUSB0 |
| 3  | crw-rw---- | 1 root | uucp | 188, | 1  | 2003-01-30 | ttyUSB1 |
| 4  | crw--w---- | 1 vcsa | tty  | 7,   | 0  | 2003-01-30 | vcs     |
| 5  | crw--w---- | 1 vcsa | tty  | 7,   | 1  | 2003-01-30 | vcs1    |
| 6  | crw--w---- | 1 vcsa | tty  | 7,   | 10 | 2003-01-30 | vcs10   |
| 7  | crw-r--r-- | 1 root | root | 1,   | 10 | 2003-01-30 | vsys    |
| 8  | brw-rw---- | 1 root | disk | 13,  | 0  | 2003-01-30 | xda     |
| 9  | brw-rw---- | 1 root | disk | 13,  | 1  | 2003-01-30 | xda1    |
| 10 | brw-rw---- | 1 root | disk | 13,  | 10 | 2003-01-30 | xda10   |
| 11 | brw-rw---- | 1 root | disk | 13,  | 11 | 2003-01-30 | xda11   |

图 7.1.4 /dev 目录下驱动条目

该目录下包含字符设备和块设备，字符设备可以在第一列开头的‘c’来识别，而块设备头一个字母为‘b’（图中后 4 行条目）。这里我们主要关注字符设备。

上图中的第 6、7 两列的数字就是设备号，在 linux 中，设备号由两部分组成，分别叫做主设备号和次设备号，上图中第 6 列为主设备号，第 7 列为次设备号。通常，主设备号标识设备对应的驱动程序。一个设备号对应一个驱动程序。次设备号由内核使用，用于正确确定设备文件所指的设备。

内核中，设备编号为 dev\_t 类型，它是一个 32 位的数，包括主设备号和次设备号，它和主次设备号之间可以下述接口转换：

```
MAJOR(dev_t dev); //由 dev_t 获得主设备号
MINOR(dev_t dev); //由 dev_t 获得次设备号
MKDEV(int major, int minor); //由主次设备号获得 dev_t
```

在建立一个字符设备之前，我们的驱动程序首先要做的是获得一个或多个设备编号。完成该工作的接口函数如下：

```
int register_chrdev_region(dev_t first, unsigned int count, char *name);
```

其中，first 是要分配的设备编号范围的起始值，可由 MKDEV 函数产生，次设备号经常被置为 0。count 为所请求的设备编号的个数，该值不宜太大，name 参数是和该编号范围相关联的设备名称。

该函数在分配成功时返回 0，错误时将返回一个负的错误码。

有时候，我们并不知道设备将使用哪些设备号，这种情况下，我们可以采用动态分配机制，使用如下的接口，让内核为我们分配所需要的主设备号：



```
int alloc_chrdev_region(dev_t *dev, unsigned int firstminor, unsigned int count,
char *name);
```

在上面的函数中, dev 用于输出分配的第一个设备号, firstminor 是请求分配的第一个次设备号, count 和 name 的定义与 register\_chrdev\_region 一致。

对于一个新的驱动程序,特别是在我们进行测试时,通常采用动态分配机制来获取设备号,从而避免给内核带来隐患。所以,我们在 Test 模块中采用了动态获取设备号的方式。(参见下节的代码部分)

当然,在驱动从内核卸载时,我们需要释放设备号,该工作由下面的函数完成:

```
void unregister_chrdev_region(dev_t first, unsigned int count);
```

参数定义与先前一致。

### 字符驱动的注册和卸载

注册和卸载字符驱动需要使用下面的接口函数,这里以 Test 模块为例:

```
module_init(test_init);
```

```
module_exit(test_exit);
```

module\_init()和 module\_exit()是 linux 内核提供的两个宏,其定义的入口函数就是驱动的注册和卸载函数,(该例中就是 test\_init 和 test\_exit)

在内核中,字符设备用 cdev 结构表示,在驱动的注册时,需要分配一个或多个上述结构。分配接口如下:

```
struct cdev *my_cdev = cdev_alloc(); //分配结构体
```

```
my_cdev->ops = &my_fops; //注册文件操作接口
```

此时,我们需要使用下面的函数来初始化已分配的结构

```
void cdev_init(struct cdev *cdev, struct file_operations *fops);
```

并设置其所有者字段(见后面的代码)。

在结构设置好后,最后一个步骤就是通过下面的调用告诉内核该结构的消息

```
int cdev_add(struct cdev *dev, dev_t num, unsigned int count);
```

dev 就是 cdev 结构, num 是该设备对应的第一个设备编号, count 是和该设备关联的设备编号的数量。该函数调用成功则返回 0, 错误返回一个负值。

同前面讲到的设备号一样,当设备从系统卸载时需要调用下面的函数移除字符设备:

```
void cdev_del(struct cdev *dev);
```

对应于我们的 test 模块,它的注册函数代码如下:

```
static int __init test_init(void)
{
 int result, err;
 dev_t dev = 0;

 result = alloc_chrdev_region(&dev, 0, 1, module_name); //获得设备号
 test_major = MAJOR(dev); //保存主设备号
 if (result < 0) {
 printk(KERN_WARNING "Test: can't get major %d\n", test_major);
 return result;
 }

 test_cdev = cdev_alloc(); //初始化结构体
 test_cdev->ops = &test_fops;
```

```

cdev_init(&test_cdev, &test_fops);
test_cdev->owner = THIS_MODULE;
err = cdev_add(&test_cdev, dev, 1); //向系统添加设备

if (err)
 printk(KERN_NOTICE "Error %d adding test driver", err);

printk("<1>Test driver register OK! major=%d\n", test_major);
return 0;
}

```

对于设备卸载部分的工作，前面的讲解中都已经做了说明，具体代码见所附 test 模块源文件。

## 文件操作

到现在为止，我们已经获得了设备号，也已经向系统注册了设备模块，但还没有任何驱动程序操作连接到这个驱动，这就需要有一个重要的结构——file\_operations，在驱动调用中，每个打开的设备文件对应一个 file\_operations 结构，该结构包含一组函数的指针，而这一组函数帮助驱动完成所有的功能，编写驱动就是向这些函数体中填充所需要的代码。

### 1) file\_operations 结构简介

下面我们介绍一下 file\_operations 结构的一些指针，但不包含全部的内容，更详细的信息请参考代码和文献。

```
struct module *owner
```

该字段不是一个操作，而是指向“拥有”该结构的模块的指针，一般为 THIS\_MODULE。

```
ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
```

该函数用来从设备中读取数据，函数返回非负值表示成功读取的字节数。

```
ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
```

向设备发送数据，返回值非负表示成功写入的字节数。关于 read 和 write 函数，后面将继续讨论。

```
int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
```

ioctl 为设备执行特定命令的方法，内核预先定义了一部分命令，驱动编写者可以自己定义 ioctl 命令。具体使用见 test 源码。

```
int (*open) (struct inode *, struct file *);
```

这是打开字符设备时执行的第一个操作，但不一定要声明，如果该函数为 null，设备的打开永远成功，但系统不会通知驱动程序。

```
int (*release) (struct inode *, struct file *);
```

当字符设备被释放时，将调用这个操作，与 open 相似，该函数也可以设置成 null。

在 Test 模块中，我们就把 open 和 release 函数简单返回 0，表示调用成功。

以上介绍的是 file\_operation 的一些基本操作，还有像 llseek、mmap 等接口，这里不作介绍。

在 test 模块中，上述结构声明代码如下：

```

static struct file_operations test_fops=
{
 owner : THIS_MODULE,
 read : test_read,

```

```

write : test_write,
ioctl : test_ioctl,
open : test_open,
release: test_release,
};

```

## 2) read 和 write

read 和 write 方法完成的任务是拷贝数据到应用程序和从应用程序拷贝数据，现在将函数原型写下：

```

ssize_t read(struct file *filp, char __user *buff,
 size_t count, loff_t *offp);
ssize_t write(struct file *filp, const char __user *buff,
 size_t count, loff_t *offp);

```

其中，filp 是文件指针，该指针代表一个打开的文件，它由内核在 open 时创建，并传递给在该文件上进行操作的所有函数。在文件的所有实例都被关闭后，内核会释放这个数据结构。buff 参数是用户空间的缓冲区指针。count 是请求传输的数据长度，最后的 offp 是一个偏移量的指针，指明用户在文件中进行存取操作的位置。

需要注意的是，上面的参数 buff 是用户空间的指针，内核空间不能直接引用其内容。因为内核模式运行时，用户空间的指针可能是无效的，该地址根本不能被映射到内核空间，而且，由于分页机制的存在，对用户空间的直接应用可能导致页错误。同时，也是为了确保内核的稳定性，在访问用户空间的数据时，需要特殊的接口函数，这也是 read 和 write 函数的核心。

```

unsigned long copy_to_user(void __user *to,
 const void *from,
 unsigned long count);

```

```

unsigned long copy_from_user(void *to,
 const void __user *from,
 unsigned long count);

```

这两个函数首先检查用户空间的指针是否有效。如果指针无效就不进行拷贝，同时，如果拷贝过程中发生错误，则仅仅复制部分数据，返回值是还需要拷贝的内存数量。返回 0 表示数据拷贝完成。

Test 模块就是直接利用了这两个函数进行用户空间和内核空间的数据拷贝，相关代码如下：

```

static ssize_t test_read(struct file *filp, char *buf, size_t count,
 loff_t *f_pos)
{
 if (copy_to_user(buf, temp, count))
 {
 printk("Test driver Read error! \n");//出错
 return 1;
 }
 printk("Test driver Read done! \n");
 return 0;
}

```

```

static ssize_t test_write(struct file *filp, const char __user *buf, size_t count,
 loff_t *f_pos)
{
 if(count<11)
 {
 if(copy_from_user(temp, buf, count))
 {
 printk("Test driver Write error! \n"); //出错
 return 1;
 }
 printk("Test driver Write done! \n");
 return 0;
 }
 else//如果数目超出 只拷贝部分数据

 {
 if(copy_from_user(temp, buf, 10))
 {
 printk("Test driver Write error! \n");
 return 1;
 }
 printk("Test driver Write done! \n");
 return 0;
 }

}

```

### 测试 test 模块

以上，我们就完成了一个简单的数据拷贝的驱动程序，配合应用程序（见附件）就可以进行测试，下面是模块在开发板上的测试步骤和结果。

1. 将 test.ko 文件添加进文件系统（操作见文件系统部分）
2. 将 test App 文件夹下的 Test 文件添加进文件系统
3. 进入操作系统的相关目录，向内核加载模块（`insmod test.ko`）
4. 可以从打印的信息中看到主设备号（测试时一般为 `major=253`），添加设备节点，  
`mknod /dev/test c 253 0`
5. 运行应用程序（`./Test`）

```
/mnt # insmod test.ko
Using test.ko
Test driver register OK! major=253
/mnt # ./Test
This is Command 1!
Test driver Write done!
Test driver Read done!
Current temp is:0 1 2 3 4
Test OK!
/mnt # lsmod
Module Size Used by Not tainted
test 3476 0 - Live 0xbf000000
/mnt # rmmod test
Test driver:GOOD-bye! !
/mnt # lsmod
Module Size Used by Not tainted
/mnt #
```

图 7.1.5:测试 test 模块

## 7.2 实验板驱动添加

以下的添加都是建立在 Linux-2.6.19.2 内核的默认配置 at91rm9200dk\_defconfig 的基础之上的。

### 7.2.1 网络驱动添加

#### 1) 修改代码:

将 arch/arm/mach-at91rm9200/board-dk.c 下面的代码

```
static struct at91_eth_data __initdata dk_eth_data = {
 .phy_irq_pin = AT91_PIN_PC4,
 .is_rmii = 1,
};
```

修改为

```
static struct at91_eth_data __initdata dk_eth_data = {
 .phy_irq_pin = AT91_PIN_PB29,
 .is_rmii = 1,
};
```

因为我们的网卡中断源是 PB29 端口。

#### 2) 配置内核:

由于内核的默认配置已经带了网卡驱动, 所以不用修改

#### 3) 测试驱动:

进入操作系统后可以用 ping 命令测试网络连接。

### 7.2.2 U 盘驱动添加

#### 1) 配置内核

Device Drivers --->

SCSI device support-->

<\*> SCSI device support

.....

<\*> SCSI disk support

USB support --->

.....

<\*> USB Mass Storage support

U盘属于海量存储设备, Linux将它定义为SCSI块设备, 所以必须要配置以上选项。

File systems --->

DOS/FAT/NT filesystems --->

<\*> MSDOS fs support

<\*> VFAT (windows 95) fs support

```

<*> NTFS file system support
Native Language Support --->
<*> Base native language support
<*> Codepage 437 (Unite States, Canada)
.....
<*> NLS ISO 8859-1 (Latin 1; Western European Languages)

```

因为我们使用的U盘可能是Windows下的文件格式，所以这里必须要选上对Windows下文件格式和语言的支持。

## 2) 在/dev目录下新建sda设备节点

命令如下：

```

mknod -m 600 sda0 b 8 0
mknod -m 600 sda b 8 0
mknod -m 600 sda1 b 8 1
mknod -m 600 sda2 b 8 2

```

如果系统只有这一个SCSI设备就可以了，假如有很多的SCSI设备则需要建立更多的sda\*设备节点。

## 3) 安装U盘

将U盘插入后，首先我们可以通过fdisk -l 命令看到U盘的信息，然后通过命令 mount -t vfat /dev/sda\* /mnt/usb (前提是已经在/mnt目录下建立了usb目录，并且U盘的格式为Windows下的vfat)安装U盘到指定的目录下。这里的“\*”表示可能是sda0或者sda1或者其它，这取决于系统上有多少SCSI设备。

## 7.2.3 SD 卡驱动添加

### 1) 修改代码：

将 arch/arm/mach-at91rm9200/board-dk.c 下面的代码

```

static struct at91_mmc_data __initdata dk_mmc_data = {
 .is_b = 0,
 .wire4 = 1,
};

```

修改为

```

static struct at91_mmc_data __initdata dk_mmc_data = {
 .det_pin = AT91_PIN_PB2,
 .is_b = 0,
 .wire4 = 1,
 .wp_pin = AT91_PIN_PB1,
};

```

管脚定义详见硬件图

### 2)配置内核

选中以下选项

- a. code maturity lever option
  - prompt for development and/or incomplete code/drivers
- b. Device Drivers

为什么这么改？

MMC/SD card Support

MMC Support

MMC block device driver

AT91RM9200 SD/MMC Card Interface support

c. File System

DOS/FAT/NT filesystem

MSDOS fs support

VFAT fs support

Native Language support

Codepage 437

Simplified Chinese charset

Codepage NLS ISO 8859 -1

**3)测试 SD 卡驱动**

将 SD 卡插入插槽

进入系统，键入以下命令

```
mknod /dev mmcblk0 b 254 1
```

```
mount -t vfat -o sync /dev/mmcblk0 /mnt
```

在/mnt 目录下就能看到 SD 卡中的内容。



## 7.3 驱动开发实例

### 7.3.1 DS18B20 驱动开发

#### 1) ds18b20 简介

Ds18b20 是 DALLAS 开发的一线温度传感器，只需要一根口线就能够传输数字温度信号，由于每个传感器对应唯一的内部 ID 号，可以支持并行连接，也就是一根口线可以控制多个温度传感器，由于其使用方便，已经被广泛地应用在工业测温的场合。

我们考虑到 DS18B20 口线简单，作为驱动开发入门恰到好处。这里，提供驱动源码和编译文件给读者参考。

#### 2) 驱动源码分析

在 ds18b20 的驱动中，很多是涉及到 ds18b20 的内部传输协议的代码，我们将不做分析，读者可以对照 datasheet 中的讲解阅读。

协议部分函数如下：

|                         |                 |
|-------------------------|-----------------|
| ds18b20_read_bit        | 从 18b20 读取一位数据  |
| ds18b20_write_bit       | 向 18b20 写入一位数据  |
| ds18b20_read_byte       | 从 18b20 读取一字节数据 |
| ds18b20_write_byte      | 向 18b20 写入一字节数据 |
| ds18b20_reset           | 将 18b20 复位      |
| ds18b20_get_temperature | 读温度             |

其中，很多函数都用到了控制 IO 的操作函数：

|                      |              |
|----------------------|--------------|
| at91_set_gpio_output | 设置 IO 脚为输出状态 |
| at91_set_gpio_input  | 设置 IO 脚为输入状态 |
| at91_set_gpio_value  | 设置 IO 脚的输出电平 |
| at91_get_gpio_value  | 读取 IO 脚电平状态  |

#### a) 驱动的注册和卸载

和前面讲解的 test 模块不同，我们这次采用静态获得设备号的方式。

```
dev=MKDEV(module_major, 0);
result=register_chrdev_region(dev, 0, module_name);

if (result < 0) {
 printk(KERN_WARNING "DS18b20: can't get major %d\n", module_major);
 return result;
}
```

初始化及卸载部分代码和 test 模块类似，这里不作介绍。

#### b) 文件操作

由于 ds18b20 只需要向应用程序提供温度，并不用从用户读取数据，所以我们没有实现 write 函数，fops 结构如下：

```
static struct file_operations ds18b20_fops=
{
 read : ds18b20_read,
 open : ds18b20_open,
 release: ds18b20_release,
```

```
};
```

Open 和 release 只是简单的返回 0，主要函数就是 read，在 read 函数中，我们首先调用 ds18b20\_get\_temperature 函数，获得温度数据，然后将温度数据存储到和应用程序交互的数组中，最后利用 copy\_to\_user 函数（见第一节）向应用程序传递温度数据。

### 3) 测试 ds18b20 驱动

将编译后的文件（源码中的 ds18b20.ko 和 ds18b20\_test）拷贝到文件系统，这里假设在 /bin 目录下，进入操作系统后，键入以下命令

```
mknod /dev/ds18b20 c 253 0
```

到 bin 目录下

```
Insmod ds18b20.ko
```

```
./ds18b20_test
```



```
/bin # insmod ds18b20.ko
Using ds18b20.ko
/bin # ./ds18b20_test
Current temperature is:29. 3125
/bin # ./ds18b20_test
Current temperature is:31. 8125
/bin # rmmod ds18b20
DS18b20 driver:GOOD-bye! !
/bin # _
```

图 7.3.1 测试 ds18b20 驱动

图中第一次测得的温度是室温，第二次的温度是将手捏住传感器约十秒后测得的温度。

#### 参考文献

1. Linux 设备驱动程序 第 3 版 中国电力出版设

## 第八章 应用程序开发

### 8.1 miniGUI 概述

#### 8.1.1 概述

MiniGUI (<http://www.minigui.com>) 是根据嵌入式系统应用特点量身定做的图形支持系统。它源自一个由魏永明主持和开发的自由软件项目, 现由北京飞漫软件技术有限公司维护并开展后续开发。

MiniGUI 项目的最初目标是为基于 Linux 的实时嵌入式系统提供一个轻量级的图形用户界面。MiniGUI 能够支持包含 Linux 在内的多种操作系统, 例如 uClinux、VxWorks、eCos、uC/OS-II、pSOS、ThreadX、Nucleus、OSE 等, 也可以在 Win32 平台上运行。MiniGUI 接口清晰, 设计严谨, 利用 MiniGUI 进行开发和调试也十分简单、高效。

MiniGUI 为应用程序定义了一组轻量级的窗口和图形设备接口。利用这些接口, 每个应用程序可以建立多个窗口并在这些窗口中创建各种控件。MiniGUI 还提供了丰富的图形功能, 帮助用户显示各种格式的位图并在窗口中输出各种文本或绘制复杂图形。

MiniGUI 增值版是飞漫软件为专有或商用产品开发商提供的 MiniGUI 商业产品。除 MiniGUI 增值版之外, 飞漫软件还遵循 GPL 发布 MiniGUI 的两个版本: MiniGUI V1.3.3 和 MiniGUI-STR V1.6.2。您可以在如下网页下载得到这两个 GPL 版本以及相关的示例代码和文档:

<http://www.minigui.com/download/cmgothor.shtml>

MiniGUI 增值版 (MiniGUI-VAR) 产品目前有两个版本: MiniGUI-VAR V1.6.x 及 MiniGUI-VAR V2.0.x。MiniGUI-VAR V1.6.x 版本主要用来支持基于线程或者任务的传统实时嵌入式操作系统, 如 VxWorks、ThreadX、Nucleus、OSE、eCos、uC/OS-II 等, 主要提供对 MiniGUI-Threads 运行模式的支持; MiniGUI-VAR V2.0.x 主要用于支持具有多进程特性的操作系统, 如 Linux 和 VxWorks 6, 提供对 MiniGUI-Processes 运行模式的支持。

#### 8.1.2 使用必须注意点

1. 目前遵循 GPL 发布的版本为: MiniGUI V1.3.3 和 MiniGUI-STR V1.6.2。

MiniGUI V1.3.3 是飞漫软件于 2003 年底开发完成的版本, 是当年的增值版, 在那时是需要收费的, 现在是遵循 GPL 发布的; MiniGUI-STR V1.6.2 则是 MiniGUI-VAR V1.6.2 的精简版本, 相比 MiniGUI V1.3.3, 增强了某些功能, 但是作为精简版本, 缺乏某些高级功能, 比如配置 miniGUI 时不能使用 `make menuconfig`, 并且库中没有字体库。

2. 必备手册 (注意和使用的源码对应版本号):

**用户手册**和**编程指南**。到官网下载:

<http://www.minigui.com/download/cindex.shtml>

**API 手册**。网址同上，但是，是在线阅读的。

3. 如果使用 linux 自带的 gedit 编辑源代码，在保存的时候必须注意：要修改首选项，将保存方式改为：“如果可用的话适用当前 locale 编码”，否则中文不能正常显示。
4. 如果使用 linux 的 Kdevelop 开发 miniGUI 应用程序，注意：RedHat 9 自带的 Kdevelop 是不能输入中文的。因为 Red Hat 9 自带 Kdevelop 是 2.1.5 版本的，而 Kdevelop 是从 3.0 往后才开始支持中文的。若在 RedHat 9 下用 Kdevelop 开发，最好先装 Kdevelop3.0 以上的版本。

## 8.2 编译安装 miniGUI

我们这里暂时使用 MiniGUI V1.3.3 版本。

首先，你需要下载 MiniGUI 的包：

<http://www.minigui.com/download/libminigui-1.3.3.tar.gz> （miniGUI 库）

<http://www.minigui.com/download/minigui-res-1.3.3.tar.gz> （miniGUI 资源包）

注意：上面这两个包的下载需要在 [www.minigui.com](http://www.minigui.com) 上注册过才可以下载，免费注册的。

<http://www.minigui.com/downloads/minigui13/mde-1.3.0.tar.gz> （综合例子程序）

<http://www.minigui.com/downloads/minigui13/mg-samples-1.3.0.tar.gz> （控件例子程序）

<http://www.minigui.com/downloads/dep-libs/qvfb-1.0.tar.gz> （qvfb,用于开发使用的，模拟 LCD 屏）

下载下来的源码包假设放在 /home/temp 目录中。

### 8.2.1 在 PC 上编译安装 miniGUI

**第一步：解压缩源码包到指定目录。**

这里，使用 /home/source/minigui/minigui1.3.x/ 作为指定目录，解压出来的代码都放到这个目录下。

1. 创建 /home/source/minigui/minigui1.3.x 目录。
 

```
cd /home/temp
mkdir -p /home/source/minigui/minigui1.3.x
```
2. 解压缩下载下来的四个包
 

```
tar -zxvf libminigui-1.3.3.tar.gz -C /home/source/minigui/minigui1.3.x
tar -zxvf minigui-res-1.3.3.tar.gz -C /home/source/minigui/minigui1.3.x
tar -zxvf mde-1.3.0.tar.gz -C /home/source/minigui/minigui1.3.x
tar -zxvf mg-samples-1.3.0.tar.gz -C /home/source/minigui/minigui1.3.x
```

**第二步：编译安装 miniGUI 库 libminigui-1.3.3。**

```
cd /home/source/minigui/minigui1.3.x/libminigui-1.3.3
./configure // 使用默认编译方式, 缺省的是 Thread 模式而不是 Lite
// 模式, 因为这种方式在 PC 机上使用最方便, 后面移
// 到板子上再改成 Lite 模式。缺省安装目录为
// /usr/local/lib

make
su // 切换到 root 模式, 下面 make install 需要 root 权限, 如
// 果已经是 root 权限, 此步省去, 以下用到 make install
// 的地方同此处。

make install
```

### 第三步: 编译安装 miniGUI 资源包 minigui-res-1.3.3

```
cd /home/source/minigui/minigui1.3.x/minigui-res-1.3.3
make install
默认安装到/usr/local/lib/minigui/res 目录。
```

### 第三步: 编译控件例子 mg-samples-1.3.0

```
cd /home/source/minigui/minigui1.3.x/ mg-samples-1.3.0
./configure // 默认配置
make
```

### 第四步: 编译综合例子 mde-1.3.0

```
cd /home/source/minigui/minigui1.3.x/mde-1.3.0
./configure // 默认配置
make
```

### 第五步: 编译安装 qvfb

qvfb 是用于在 PC 上开发 miniGUI 应用程序时模拟 LCD 屏, 可以实时看到自己编写的界面效果, 以便及时改进。装 qvfb 时注意, 之前必须装有 qt 库, 若在装 linux 系统时候安装了 KDE 桌面, 那么 qt 库已经装上了。

```
cd /home/source/minigui/minigui1.3.x/mde-1.3.0/qvfb-1.0
./configure // 使用默认配置, 安装目录为/usr/local
make
make install
```

至此, PC 上开发 miniGUI 应用程序所需的東西已全部安装完毕。

### 第六步: 在正式开发 miniGUI 应用程序之前, 需做如下事情:

1. 必须修改/etc/ld.so.conf 文件, 将/usr/local/lib 添加到此文件中。因为我们将 miniGUI 安装了这个地方, 要将 miniGUI 的库添加进来, 不然 miniGUI 应用程序会找不到 miniGUI 的库函数。修改完 ld.so.conf 文件后, 需运行如下命令:  

```
ldconfig ld.so.conf // 刷新可共享的动态链接库
```
2. 修改/usr/local/etc/MiniGui.cfg 文件。修改如下:

```
[system]
GAL engine
gal_engine=qvfb //这里修改，开发时使用 qvfb 模拟

IAL engine
ial_engine=qvfb //这里修改，同上

[qvfb]
defaultmode=800x600-16bpp // 改成所需要模拟屏幕分辨率的大小，这里我们使用的是 800x600，16 位深度
```

下面即可在 PC 上进入 miniGUI 应用程序开发了，开发完的程序经过交叉编译就可移到板子上运行了。

## 8.2.2 交叉编译安装 miniGUI

基本步骤 PC 上编译安装 miniGUI 一致，但是具体过程要稍显复杂点。和 PC 上编译安装 miniGUI 的主要不同点如下：

1. 使用交叉编译器(`arm-linux-gcc`)，而 PC 上使用的是 linux 系统自带的 `gcc` 编译器。
2. 在编译安装 miniGUI 之前，需要安装 `zlib` 库（安装 `png` 库的时候需要）、`png` 库、`jpeg` 库、`freetype` 库。这些库视实际具体情况需要，选择安装或者不安装。不安装的时候，在 miniGUI 配置的时候把这些相应的选项去掉即可。
3. `qvfb` 不用进行编译安装。

具体安装如下（包括上面 2 中提到的这些库）：

首先，下载如下源码包：

<http://www.zlib.net/zlib-1.2.3.tar.gz> (`zlib` 库)

<http://prdownloads.sourceforge.net/libpng/libpng-1.2.18.tar.bz2?download> (`png` 库)

<http://download.enet.com.cn/html/242232006010901.html> (`jpeg` 库)

并解压缩出来，这里我们解压缩到 `/home/source/minigui`

**前提：**电脑上安装有 `arm-linux-gcc` 交叉编译器，若没有安装交叉编译器，请先安装交叉编译器。这里我们使用的是 `arm-linux-gcc3.3.2` 版本（在目录 `/usr/local/arm/3.3.2/` 中）。

### 第一步：安装 `zlib` 库

**注意：**由于 `zlib` 库的 `configure` 脚本不支持交叉编译选项，只好手动临时把 `gcc` 修改成指向我们的交叉编译器 `arm-linux-gcc`。

```
cd /usr/bin
mv gcc gcc_back // 将原来的 gcc 备份成 gcc_back，过后还原为 gcc
ln -s /usr/local/arm/3.3.2/bin/arm-linux-gcc ./gcc
mv ld ld_back
ln -s /usr/local/arm/3.3.2/bin/arm-linux-ld ./ld
```

修改完成后回到 `/home/source/minigui/zlib-1.2.3` 目录下,进行配置安装：

```
cd /home/source/minigui/zlib-1.2.3
```

```
./configure --prefix=/usr/local/arm/3.3.2/arm-linux/ --shared
```

注意：这里配置指向 /usr/local/arm/3.3.2/arm-linux/ 目录，会自动安装在 /usr/local/arm/3.3.2/arm-linux/ [include,lib] 目录下，千万不要装错目录了，不然后面会找不到这个库的。

```
make
```

```
make install
```

安装完了，查看下 /usr/local/arm/3.3.2/arm-linux/ 目录下的 lib 文件夹里是否有 libz.so, libz.so.1, libz.so.1.2.3 和 include 文件中有 libz.h。

将刚刚备份的 gcc 还原为原来系统的 gcc:

ld 忘记了

```
cd /usr/bin
```

```
mv gcc_back gcc
```

## 第二步：安装 png 库

```
cd /home/source/minigui/libpng-1.2.18
```

```
./configure CC=arm-linux-gcc --prefix=/usr/local/arm/3.3.2/arm-linux --host=arm-linux
```

注意这边的配置：①使用交叉编译器②安装目录③—host 指定软将运行平台，不然终端也会提示说叫你使用—host 参数的。

```
make
```

```
make install
```

安装完了，查看下 /usr/local/arm/3.3.2/arm-linux/ 目录下的 lib 文件夹里是否有 libpng.a, libpng.so 等文件和 include 文件夹里是否有 png.h, pngconf.h 文件以及 libpng12 文件夹。

## 第三步：安装 jpeg 库

```
cd /home/source/minigui/jpeg-6b
```

```
./configure CC=arm-linux-gcc --prefix=/usr/local/arm/3.3.2/arm-linux --enable-shared
```

在 make 之前注意：从前面 libpng-1.2.18 的源码目录中把 libtool 拷贝过来，放在 /home/source/minigui/jpeg-6b 目录下，否则会出现 “make: ./libtool: command not found” 错误。另外要注意的是：不能使用系统的 libtool，因为这里要进行的是交叉编译。

```
make
```

下面创建一个目录，否则 make install 的时候会报错。

```
mkdir -p /usr/local/arm/3.3.2/arm-linux/man/man1
```

```
make install
```

安装完了，查看下 /usr/local/arm/3.3.2/arm-linux/ 目录下的 lib 文件夹里是否有 libjpeg.so, libjpeg.so.62.0.0 等文件和 include 文件夹里是否有 jpeglib.h 文件。

## 第四步：配置编译安装 miniGUI

MiniGUI 1.3.3 版本在链接 libjpeg libpng 时老是要链接 /usr/lib 下的库，没办法，只好用前面的方法，使用备份和软连接的方法。

```
cd /usr/lib
```

```
mv libjpeg.so libjpeg.so_back
```

```
mv libpng.so libpng.so_back
```

```
ln -s /usr/local/arm/3.3.2/arm-linux-gcc/lib/libpng.so libpng.so
```

```
ln -s /usr/local/arm/3.3.2/arm-linux-gcc/lib/libjpeg.so libjpeg.so
```

下面进行配置安装：

```
cd /home/source/minigui/minigui-1.3.x/libminigui-1.3.3
```

```
make menuconfig
```

make menuconfig 不行的话可以 ./configure

在 GAL engine Options 里

(NEWGAL) GAL and its engines

- [\*] NEWGAL engine on Linux FrameBuffer console
- [ ] NEWGAL engine on Qt Virtual FrameBuffer
- [ ] NEWGAL engine on eCos LCD interface
- [\*] Dummy NEWGAL engine
- [\*] Have console on Linux FrameBuffer

在 Font Option 里

- [\*] Raw bitmap font
- [ ] Var bitmap font // 这个选项去掉，否则，编译时老出错
- [\*] Incore font sansserif
- [\*] Incore font courier
- [\*] Incore font symbol
- [\*] Incore font vgas
- [\*] Qt Prerendered Font
- [ ] TrueType font // 如果需要对这个字体支持，需要下载编译安装 freetype 库
- [ ] Adobe Type1 font // 如果需要对这个字体支持，需要下载编译安装 freetype 库

NOTE: 上面两个需要编译安装的字体库，在 miniGUI 官网上有的下。

在 Image Options 里

- [\*] Includes SaveBitmap-related functions
- [ ] PCX file support
- [ ] LBM/PBM file support
- [ ] TGA file support
- [\*] GIF file support
- [\*] JPG file support // 前面我们已经编译安装了 JPG 库，所以这里我们可以选择此项
- [\*] PNG file support // 前面我们已经编译安装了 PNG 库，所以这里我们可以选择此项

在 Development Environment Option 里

(Linux) Platform

(arm-linux-gcc) Compiler

(glibc) Libc

--- Installation options

Path prefix: "/usr/local/arm/3.3.2/arm-linux" // 这里，我们把库装到交叉编译器的目录下

--- Additonal Compiler Flags

CFLAGS: ""

LDFLAGS: ""

其他的选项暂时不做改变，使用默认的配置即可。

接下来：

若使用的 arm-linux-gcc 为 2.95.3 版本，则直接进行 make；若使用的 arm-linux-gcc 为 3.3.2 版本，在 make 之前修改 libminigui-1.3.3/src/newgal/stretch.c 文件，从此文件 265 行开始：



将

```
__asm__ __volatile__ ("
 call _copy_row
"
: "=&D" (u1), "=&S" (u2)
: "0" (dstp), "1" (srcp)
: "memory");
```

改为:

```
__asm__ __volatile__ ("call _copy_row"
: "=&D" (u1), "=&S" (u2)
: "0" (dstp), "1" (srcp)
: "memory");
```

否则 make 的时候会出现如下错误:

```
stretch.c:265:47: missing terminating " character
stretch.c:267:25: missing terminating " character
make[4]: *** [stretch.lo] Error 1
make[4]: Leaving directory `/home/source/minigui/minigui1.3.x/libminigui-1.3.3/src/newgal'
make[3]: *** [all-recursive] Error 1
make[3]: Leaving directory `/home/source/minigui/minigui1.3.x/libminigui-1.3.3/src/newgal'
make[2]: *** [all-recursive] Error 1
make[2]: Leaving directory `/home/source/minigui/minigui1.3.x/libminigui-1.3.3/src'
make[1]: *** [all-recursive] Error 1
make[1]: Leaving directory `/home/source/minigui/minigui1.3.x/libminigui-1.3.3'
make: *** [all] Error 2
```

修改之后, 进行编译安装:

```
make // 需要一点时间, 耐心等待
make install
```

安装完了, 查看下 /usr/local/arm/3.3.2/arm-linux/ 目录下的 lib 文件夹里是否有 libminigui.so, libmgext.so, libvcongui.so 等文件和 include 文件夹里是否有 minigui 文件夹(内有 minigui 相关的头文件)。

好了, 现在不要忘记把前面刚刚备份的改回来:

```
cd /usr/lib
mv libjpeg.so_back libjpeg.so
mv libpng.so_back libpng.so
```

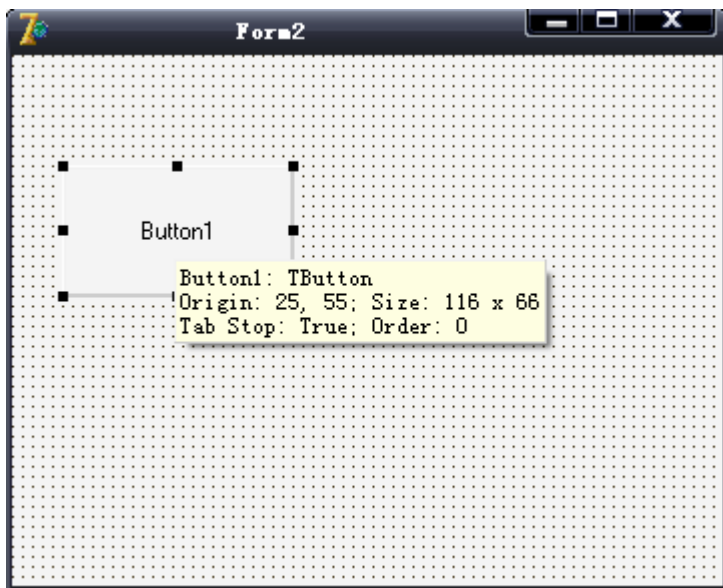
## 8.3 快速开始 miniGUI 编程

### 8.3.1 基本编程概念

MiniGUI 的编程重点在于两点:

1、界面坐标的规划。

这个，可以利用 Delphi 先画出界面，再提取出各个控件的坐标，放到 minigui 界面中的控件坐标中，如下图所示：



把鼠标停留在控件上面，可以看到该控件的坐标以及大小信息。把该信息添加到 minigui 控件中即可。

2、正确写好窗口和控件的回调函数。

miniGUI 是一种消息传递机制，由窗口和控件接收鼠标和键盘的信息，然后转化为消息投递到应用程序中，由过程回调函数处理各种消息。所以，正确写好回调函数，处理好各种消息是关键。

### 8.3.2 从简单例子迅速入门

开始之前，几点需要注意：

- 1、数据类型、minigui 的函数具体含义可参见 minigui 的官方 API 手册：  
[http://www.minigui.com/api\\_ref/1.3.x/index.html](http://www.minigui.com/api_ref/1.3.x/index.html)，本文中不再具体说明。
- 2、具体控件的使用，不再赘述，参见 minigui 的官方编程指南。

讲述一个简单例子，快速了解 miniGUI 的程序结构。

/\*\*\*\*\*\* 头文件部分 \*\*\*\*\*

<minigui/common.h>、 <minigui/minigui.h>、<minigui/gdi.h> 、<minigui/window.h> 是所

有的 MiniGUI 应用程序都**必须包括**的头文件：

common.h: miniGUI 宏以及数据类型的定义

minigui.h: minigui 全局通用接口函数以及其他杂项函数定义

gdi.h: minigui 的绘图函数的定义

window.h: minigui 的与窗口有关的相关定义

另外**注意**：还有个头文件，这边没加上去：

control.h：包含了 libminigui 中所有内建控件的接口定义。

一般都要加上这个头文件，我们编程过程中大部分情况下都会用到控件的。

\*\*\*\*\*/

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

```
#include <minigui/common.h>
```

```
#include <minigui/minigui.h>
```

```
#include <minigui/gdi.h>
```

```
#include <minigui/window.h>
```

\*\*\*\*\*/ 回调函数 \*\*\*\*\*/

第一个参数，是接收消息的窗口的句柄，该值标识了哪个窗口收到消息；

第二个参数，标识了窗口所收到消息类型，说明收到的是什么消息；

最后两个参数都是 32 位的消息参数，和具体按消息相关，它提供和消息相关的特定信息。

\*\*\*\*\*/

```
static int HelloWinProc(HWND hWnd, int message, WPARAM wParam,
LPARAM lParam)
```

```
{
```

```
 HDC hdc;
```

```
 switch (message) {
```

```
 case MSG_PAINT:
```

```
 hdc = BeginPaint (hWnd);
```

```
 TextOut (hdc, 100, 100, "Hello world!");
```

```
 EndPaint (hWnd, hdc);
```

```
 return 0;
```

```
 case MSG_CLOSE:
```

```
 DestroyMainWindow (hWnd);
```

```
 PostQuitMessage (hWnd);
```

```
 return 0;
```

```
 }
```

```
 return DefaultMainWinProc(hWnd, message, wParam, lParam);
```

```
}
```

\*\*\*\*\*/ miniGUI函数主函数 \*\*\*\*\*/

MiniGUI函数是minigui程序的入口函数，相当与C语言中的main（）函数。

主要目的就是创建主窗口或者对话框，进入消息循环。

```

*****/
int MiniGUIMain (int argc, const char* argv[])
{
 MSG Msg;
 HWND hMainWnd;
 MAINWINCREATE CreateInfo;

#ifdef _LITE_VERSION
 SetDesktopRect(0, 0, 800, 600);
#endif

 // 下面是定义主窗口的属性，具体含义见 8.4.1主窗口编程基础
 CreateInfo.dwStyle = WS_VISIBLE | WS_BORDER | WS_CAPTION;
 CreateInfo.dwExStyle = WS_EX_NONE;
 CreateInfo.spCaption = "HelloWorld";
 CreateInfo.hMenu = 0;
 CreateInfo.hCursor = GetSystemCursor(0);
 CreateInfo.hIcon = 0;
 CreateInfo.MainWindowProc = HelloWinProc;
 CreateInfo.lx = 0; // 这边定义了主窗口四角坐标
 CreateInfo.ty = 0;
 CreateInfo.rx = 320;
 CreateInfo.by = 240;
 CreateInfo.iBkColor = COLOR_lightwhite;
 CreateInfo.dwAddData = 0;
 CreateInfo.hHosting = HWND_DESKTOP; // 主窗口的托管窗口是桌面窗口

 // 创建主窗口
 hMainWnd = CreateMainWindow (&CreateInfo);

 if (hMainWnd == HWND_INVALID)
 return -1;

 ShowWindow(hMainWnd, SW_SHOWNORMAL);

 // 下面是消息循环，不停的取消息、解释消息、投递消息，在回调函数中进行对消息
 的处理
 while (GetMessage(&Msg, hMainWnd))
 {
 TranslateMessage(&Msg);
 DispatchMessage(&Msg);
 }

 MainWindowThreadCleanup (hMainWnd);

```

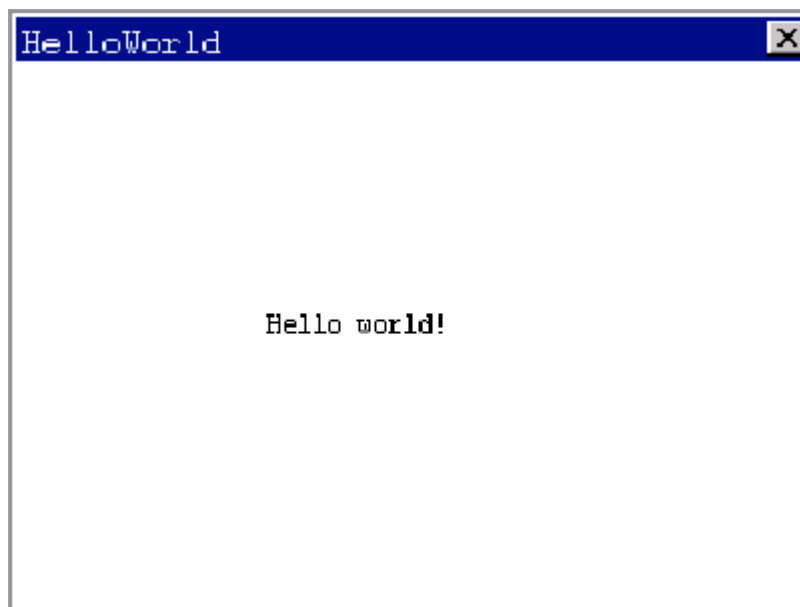
```

 return 0;
}

#ifdef _LITE_VERSION
#include <minigui/dti.c>
#endif

```

该程序在屏幕上创建一个大小为 320x240 像素的应用程序窗口，并在窗口客户区的中部显示"Hello world!"，如下图所示：



### 8.3.3 编译、链接、运行

1、用 gcc 进行编译。

```
gcc -o helloworld helloworld.c -lminigui
```

-lminigui 是必须加的选项。

如果你 MiniGUI 配置为 MiniGUI-Threads，则需要使用-lpthread 编译选项：

```
gcc -o helloworld helloworld.c -lpthread -lminigui
```

libpthread: 是提供 POSIX 兼容线程支持的函数库，编译 MiniGUI-Threads 程序时必须连接这个函数库。

另外，还有其他两个编译选项：

-lmgext: 若使用了 minigui 中扩展库中的内容，需要加上此编译选项；

-lvcongui: 若使用了虚拟控制台窗口，需要加上此编译选项。

2、用 Autoconf、Automake 工具进行编译。

Autoconf/Automake 是 UNIX 系统下维护一个软件项目的最佳工具。我们可以不用重复

敲打命令。随 MiniGUI 一同发布的 MDE 就是一个 Autoconf/Automake 脚本组织起来的软件项目。

这里不再详细介绍这两个工具的使用，只是利用 MDE 中的 Autoconf/Automake 脚本来改写成 HELLOWORLD 的 Autoconf/Automake 脚本，进行项目维护。具体 Autoconf/Automake 的使用可以参照以下网址：

<http://www.linuxforum.net/books/autoconf.html>

我们在系统适当的目录下建 samples 目录作为项目的根目录，并为项目取名为 samples。比如：

```
mkdir -p /home/samples
```

然后 sample 下建立 src 目录，用来存放 helloworld 程序的源代码。将 helloworld.c 保存在 samples/src/目录下，然后从 mde-1.3.x 中复制 configure.in 文件。

下面，我们就在 MDE 的管理脚本基础上针对 samples 项目进行修改。

### 第一步，我们修改 configure.in 文件。

大部分我们都不用去考虑，只需考虑需要修改的那些，修改后的文件如下所示（注意我们所做的中文注释，我们只修改了通过中文注释注解的那些宏）：

```
dnl Process this file with autoconf to produce a configure script.
```

```
AC_PREREQ(2.13)
```

```
dnl 在下面的宏中指定一个项目源文件
```

```
AC_INIT(src/helloworld.c)
```

```
dnl
```

```
==
```

```
dnl needed for cross-compiling
```

```
C_CANONICAL_SYSTEM
```

```
dnl
```

```
==
```

```
dnl Checks for programs.
```

```
AC_PROG_MAKE_SET
```

```
AC_PROG_CC
```

```
dnl 在下面的宏中指定项目名称（samples）和项目版本号（1.3.1）
```

```
AM_INIT_AUTOMAKE(samples,1.3.1)
```

```
dnl
```

```
==
```

```
dnl Checks for typedefs, structures, and compiler characteristics.
```

```
AC_C_CONST
```

```
dnl
```

```
=====
==
dnl Checks for header files.
```

```
AC_HEADER_STDC
```

```
AC_HEADER_SYS_WAIT
```

```
AC_HEADER_TIME
```

```
AC_CHECK_HEADERS(sys/time.h unistd.h)
```

```
dnl
```

```
=====
==
dnl check for libminigui
```

```
have_libminigui="no"
```

```
AC_CHECK_HEADERS(minigui/common.h, have_libminigui=yes, foo=bar)
```

```
dnl
```

```
=====
==
dnl check for lite or threads version of MiniGUI
```

```
lite_version="no"
```

```
AC_CHECK_DECLS(_LITE_VERSION, lite_version="yes", foo=bar, [#include
<minigui/common.h>])
```

```
dnl
```

```
=====
==
dnl check for newgal or oldgal interface.
```

```
use_newgal="no"
```

```
AC_CHECK_DECLS(_USE_NEWGAL, use_newgal="yes", foo=bar, [#include
<minigui/common.h>])
```

```
dnl
```

```
=====
==
dnl Write Output
```

```
if test "$ac_cv_prog_gcc" = "yes"; then
```

```
 CFLAGS="$CFLAGS -Wall -Wstrict-prototypes -pipe"
```

```
fi
```

```
if test "x$lite_version" = "xyes"; then
```

```
 LIBS="$LIBS -lminigui"
```

```
)
```

```

if test "x$have_libminigui" != "xyes"; then
 AC_MSG_WARN([
 MiniGUI is not properly installed on the system. You need MiniGUI
Ver 1.2.6
 or later for building this package. Please configure and
 install MiniGUI Ver 1.2.6 first.
])
fi
else
 CFLAGS="$CFLAGS -D_REENTRANT"
 LIBS="$LIBS -lpthread -lminigui"
fi

```

dnl 先注释如下两个宏，将在以后打开这两个宏

```

dnl AM_CONDITIONAL(LITE_VERSION, test "x$lite_version" = "xyes")
dnl AM_CONDITIONAL(USE_NEWGAL, test "x$use_newgal" = "xyes")

```

dnl 在下面的宏中列出要生成的 Makefile 文件

```

AC_OUTPUT(
Makefile
src/Makefile

```

利用这个 configure.in 生成的 configure 脚本和 Makefile 文件将帮助我们完成如下工作：

生成适于进行交叉编译的 configure 脚本

检查系统中是否安装了 MiniGUI

检查系统中已安装的 MiniGUI 被配置成 MiniGUI-Lite 还是 MiniGUI-Threads，并适当设置程序要连接的函数库。

生成项目根目录下的 Makefile 文件以及 src/ 子目录中的 Makefile 文件。

接下来，我们建立项目根目录下的 Makefile.am 文件。该文件内容如下：

```
SUBDIRS = src
```

上述文件内容告诉 Automake 系统进入 src/ 目录继续处理。

**第二步，我们建立 src/ 子目录下的 Makefile.am 文件。**

该文件内容如下：

```
noinst_PROGRAMS=helloworld
```

```
helloworld_SOURCES=helloworld.c
```

上述文件内容告诉 Automake 生成一个用来从 helloworld.c 建立 helloworld 程序的 Makefile 文件。

**第三步，在项目根目录（注意是项目根目录，不是根目录）下建立一个 autogen.sh 文件，内容如下：**

```
#!/bin/sh
```

```
aclocal
```

```
automake --add-missing
```

```
autoconf
```

该文件是一个 shell 脚本，依次调用了 aclocal、automake 和 autoconf 命令。



请注意：

在建立该文件之后，要运行 `chmod` 命令使之变成可执行文件：

```
chmod +x autogen.sh
```

第四步，运行如下命令生成项目所需的 **Makefile** 文件：

```
./autogen.sh
```

```
./configure
```

【注意】每次修改 `configure.in` 文件之后，应执行 `./autogen.sh` 命令更新 `configure` 脚本以及 `makefile` 文件。

运行完上述命令之后，你会发现项目根目录下多了许多自动生成的文件。我们无需关注这些文件的用途，忽略这些文件，然后执行 `make` 命令。

假如你的 `helloworld` 程序规模非常庞大，因此将代码分开放在不同的源文件中了，这时，你只需修改 `src/` 下的 `Makefile.am`，在 `helloworld_SOURCES` 后面添加这些源文件的名称，然后在项目根目录下重新执行 `make` 命令即可。例如：

```
noinst_PROGRAMS=helloworld
```

```
helloworld_SOURCES=helloworld.c helloworld.h module1.c module2.c
```

## 8.4 主窗口和对边框编程基础

MiniGUI 中有三种窗口类型：主窗口、对话框和控件窗口（子窗口）。

### 8.4.1 主窗口编程基础

1、创建主窗口。

MiniGUI 中，应通过初始化一个 `MAINWINCREATE` 结构（描述主窗口的各个属性，如风格，大小等等），然后调用 `CreateMainWindow` 函数来创建一个主窗口。`MAINWINCREATE` 结构的成员解释如下：

|                                        |                        |
|----------------------------------------|------------------------|
| <code>CreateInfo.dwStyle</code>        | 窗口风格                   |
| <code>CreateInfo.spCaption</code>      | 窗口的标题                  |
| <code>CreateInfo.dwExStyle</code>      | 窗口的附加风格                |
| <code>CreateInfo.hMenu</code>          | 附加在窗口上的菜单句柄            |
| <code>CreateInfo.hCursor</code>        | 在窗口中所使用的鼠标光标句柄         |
| <code>CreateInfo.hIcon</code>          | 程序的图标                  |
| <code>CreateInfo.MainWindowProc</code> | 该窗口的消息处理函数指针           |
| <code>CreateInfo.lx</code>             | 窗口左上角相对屏幕的绝对横坐标，以像素点表示 |
| <code>CreateInfo.ty</code>             | 窗口左上角相对屏幕的绝对纵坐标，以像素点表示 |
| <code>CreateInfo.rx</code>             | 窗口右下角相对屏幕的绝对横坐标，以像素点表示 |
| <code>CreateInfo.by</code>             | 窗口右下角相对屏幕的绝对纵坐标，以像素点表示 |
| <code>CreateInfo.iBkColor</code>       | 窗口背景颜色                 |
| <code>CreateInfo.dwAddData</code>      | 附带给窗口的一个 32 位值         |
| <code>CreateInfo.hHosting</code>       | 窗口消息队列的托管窗口            |

窗口风格用来控制窗口的一些外观及行为方式，比如窗口的边框类型、窗口是否可见、窗口是否可用等等。这个不具体介绍了，可详细参见《MINIGUI-PROG-GUIDE-V1.3-C》3.2.3 节。

2、销毁主窗口。

要销毁一个主窗口，可以利用 `DestroyMainWindow (hWnd)` 函数。该函数将向窗口

过程发送 `MSG_DESTROY` 消息，并在该消息返回非零值时终止销毁过程。

应用程序一般在主窗口过程中接收到 `MSG_CLOSE` 消息时调用这个函数销毁主窗口，然后调用 `PostQuitMessage` 消息终止消息循环。如下所示：

```
case MSG_CLOSE:
```

```
// 销毁主窗口
```

```
DestroyMainWindow (hWnd);
```

```
// 发送 MSG_QUIT 消息
```

```
PostQuitMessage(hWnd);
```

```
return 0;
```

`DestroyMainWindow` 销毁一个主窗口，但不会销毁主窗口所使用的消息队列以及窗口对象本身。因此，应用程序要在线程或进程的最后使用 `MainWindowCleanup` 最终清除主窗口所使用的消息队列以及窗口对象本身。

主窗口的例子可以参照前面的8.3.2节的例子。

## 8.4.2 对话框编程基础

上面介绍了主窗口的创建与销毁，在主窗口中创建控件时，每创建一个控件，就得使用一句 `CreateWindow` 函数，这样，使得程序看起来很复杂。使用对话框，可使得程序看起来更清楚，也便于修改。

1、对话框的创建，是使用模板创建的。在 `MiniGUI` 中，用两个结构来表示对话框模板（`minigui/window.h`）：

`CTRLDATA`：规定了对话框的属性，如对话框大小，坐标等等。

`DLGTEMPLATE`：规定了对话框里面所有控件的属性，如各控件大小，坐标，风格等等。

对话框使用例子如下：

```
/****** 头文件部分 *****/
```

同 8.3.2

```
*****/
```

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

```
#include <minigui/common.h>
```

```
#include <minigui/minigui.h>
```

```
#include <minigui/gdi.h>
```

```
#include <minigui/window.h>
```

```
#include <minigui/control.h>
```

```
/****** 模板部分 *****/
```

`static DLGTEMPLATE DlgInitProgress`: 定义了对话框的各个属性

`static CTRLDATA CtrlInitProgress []`: 定义了对话框中的各个控件的属性

```
*****/
```

```
static DLGTEMPLATE DlgInitProgress =
```

```

{
 WS_BORDER | WS_CAPTION, // 风格
 WS_EX_NONE, // 扩展风格
 120, 150, 400, 130, // 坐标
 "VAM-CNC 正在初始化", // 标题
 0, 0, // 图标和菜单
 3, NULL, // 控件数和指向控件数组的指针
 0 // 扩展数据，必须为0
};

```

```
#define IDC_PROMPTINFO 100
```

```
#define IDC_PROGRESS 110
```

```
static CTRLDATA CtrlInitProgress [] =
```

```

{
 {
 "static", // 控件类
 WS_VISIBLE | SS_SIMPLE, // 风格
 10, 10, 380, 16, // 坐标
 IDC_PROMPTINFO, // ID
 "正在...", // 标题
 0 // 扩展数据
 },
 {
 "progressbar",
 WS_VISIBLE,
 10, 40, 380, 20,
 IDC_PROGRESS,
 NULL,
 0
 },
 {
 "button",
 WS_TABSTOP | WS_VISIBLE | BS_DEFPUSHBUTTON,
 170, 70, 60, 25,
 IDOK,
 "确定",
 0
 }
};

```

/\*\*\*\*\*\* 回调函数 \*\*\*\*\*

参数同 8.3.2 中例子类似。

```

*****/
static int InitDialogBoxProc (HWND hDlg, int message, WPARAM wParam, LPARAM
lParam)
{
 switch (message) {
 case MSG_INITDIALOG:
 return 1;

 case MSG_COMMAND:
 switch (wParam) {
 case IDOK:
 case IDCANCEL:
 EndDialog (hDlg, wParam);
 break;
 }
 break;
 }

 return DefaultDialogProc (hDlg, message, wParam, lParam);
}

```

/\*\*\*\*\*\* 主函数部分 \*\*\*\*\*/

对话框的创建用两句话实现。

- 1、对话框控件数组指针的传递；
- 2、对话框的创建。需要传递的 4 个参数依次如下：

对话框模板、托管窗口、回调函数、要传递到对话框过程的参数

\*\*\*\*\*/

```

int MiniGUIMain (int argc, const char* argv[])
{
 #ifdef _LITE_VERSION
 SetDesktopRect(0, 0, 1024, 768);
 #endif

 DlgInitProgress.controls = CtrlInitProgress; // 控件数组指针传递
 DialogBoxIndirectParam(&DlgInitProgress, HWND_DESKTOP, InitDialogBoxProc,
0L); // 建立对话框
 return 0;
}

#ifdef _LITE_VERSION
#include <minigui/dti.c>
#endif

```

## 8.5 控件编程基础

较为复杂的GUI系统中，都带有预定义的控件集合，它们是人机交互的主要元素。本节将说明什么是控件、控件类，并简单介绍MiniGUI中的预定义控件类。

### 8.5.1 控件和控件类

许多人对控件（或者部件）的概念已经相当熟悉了。控件可以理解为主窗口中的子窗口。这些子窗口的行为和主窗口一样，既能够接收键盘和鼠标等外部输入，也可以在自己的区域内进行输出——只是它们的所有活动被限制在主窗口中。MiniGUI也支持子窗口，并且可以在子窗口中嵌套建立子窗口。我们将MiniGUI中的所有子窗口均称为控件。

在Windows或X Window中，系统会预先定义一些控件类，当利用某个控件类创建控件之后，所有属于这个控件类的控件均会具有相同的行为和外观。利用这些技术，可以确保一致的人机操作界面，而对程序员来讲，可以像搭积木一样地组建图形用户界面。MiniGUI使用了控件类和控件的概念，并且可以方便地对已有控件进行重载，使得它有一些特殊效果。比如，需要建立一个只允许输入数字的编辑框时，就可以通过重载已有编辑框而实现，而不需要重新编写一个新的控件类。

如果读者曾经编写过Windows应用程序的话，应该记得在建立一个窗口之前，必须确保系统中存在新窗口所对应的窗口类。在Windows中，程序所建立的每个窗口，都对应着某种窗口类。这一概念和面向对象编程中的类、对象的关系类似。借用面向对象的术语，Windows中的每个窗口实际都是某个窗口类的一个实例。在X Window编程中，也有类似的概念，比如我们建立的每一个Widget，实际都是某个Widget类的实例。

这样，如果程序需要建立一个窗口，就首先要确保选择正确的窗口类，因为每个窗口类决定了对应窗口实例的表象和行为。这里的表象指窗口的外观，比如窗口边框宽度，是否有标题栏等等，行为指窗口对用户输入的响应。每一个GUI系统都会预定义一些窗口类，常见的有按钮、列表框、滚动条、编辑框等等。如果程序要建立的窗口很特殊，就需要首先注册一个窗口类，然后建立这个窗口类的一个实例。这样就大大提高了代码的可重用性。

在MiniGUI中，我们认为主窗口通常是一种比较特殊的窗口。因为主窗口代码的可重用性一般很低，如果按照通常的方式为每个主窗口注册一个窗口类的话，则会导致额外不必要的存储空间，所以我们并没有在主窗口提供窗口类支持。但主窗口中的所有子窗口，即控件，均支持窗口类（控件类）的概念。MiniGUI提供了常用的预定义控件类，包括按钮（包括单选钮、复选钮）、静态框、列表框、进度条、滑块、编辑框等等。程序也可以定制自己的控件类，注册后再创建对应的实例。表8.5.1给出了MiniGUI预先定义的控件类和相应类名称定义。

表 8.5.1 MiniGUI 预定义的控件类和对应类名称

| 控件类   | 类名称      | 宏定义         | 备注                                        |
|-------|----------|-------------|-------------------------------------------|
| 静态框   | "static" | CTRL_STATIC |                                           |
| 按钮    | "button" | CTRL_BUTTON |                                           |
| 简单编辑框 | "edit"   | CTRL_EDIT   | 使用等宽系统字符来显示输入文字，只能处理ISO8859 和 GB2312 字符集。 |
| 单行编辑框 | "sledit" | CTRL_SLEDIT | 可处理变宽字符，支持任                               |

|       |                 |                    |                             |
|-------|-----------------|--------------------|-----------------------------|
| 多行编辑框 | "mledit"        | CTRL_MLEDIT        | 意字符集                        |
| 列表框   | "listbox"       | CTRL_LISTBOX       |                             |
| 进度条   | "progressbar"   | CTRL_PRORESSBAR    |                             |
| 滑块    | "trackbar"      | CTRL_TRACKBAR      |                             |
| 工具条   | "toolbar"       | CTRL_TOOLBAR       | 已废弃，不推荐使用                   |
| 新工具条  | "newtoolbar"    | CTRL_NEWTOOLBAR    |                             |
| 菜单按钮  | "menubutton"    | CTRL_MENUBUTTON    |                             |
| 属性页   | "propsheet"     | CTRL_PROPSHEET     |                             |
| 树型控件  | "treeview"      | CTRL_TREEVIEW      | 包含在 mgext 库，即 MiniGUI 扩展库中。 |
| 列表型控件 | "listview"      | CTRL_LISTVIEW      |                             |
| 月历    | "MonthCalendar" | CTRL_MONTHCALENDAR |                             |
| 旋钮控件  | "SpinBox"       | CTRL_SPINBOX       |                             |
| 酷工具栏  | "CoolBar"       | CTRL_COOLBARA      |                             |

## 8.5.2 利用预定义控件创建控件实例

在MiniGUI中，通过调用CreateWindow函数（CreateWindow其实是CreateWindowEx函数的宏），可以建立某个控件类的一个实例。控件类既可以是表 8.5.1中预定义MiniGUI控件类，也可以是用户自定义的控件类。下面是与CreateWindow函数相关的几个函数的原型（<minigui/window.h>）：

```

904 HWND GUIAPI CreateWindowEx (const char* spClassName, const char* spCaption,
905 DWORD dwStyle, DWORD dwExStyle, int id,
906 int x, int y, int w, int h, HWND hParentWnd, DWORD dwAddData);
907 BOOL GUIAPI DestroyWindow (HWND hWnd);
908
909 #define CreateWindow(class_name, caption, style, id, x, y, w, h, parent, add_data) \
910 CreateWindowEx(class_name, caption, style, 0, id, x, y, w, h, parent, add_data)

```

CreateWindowEx函数指定控件的扩展风格（dwExStyle）CreateWindow函数建立一个子窗口，即控件。它指定了控件类（class\_name）、控件标题（caption）、控件风格（style）、控件的标识符（id）、以及窗口的初始位置和大小（x, y, w, h）。该函数同时指定子窗口的父窗口（parent）。参数add\_data用来向控件传递其特有数据的指针，该指针所指向的数据结构随控件类的不同而不同。

CreateWindowEx函数的功能和CreateWindow函数一致，不过，可以通过CreateWindowEx函数指定控件的扩展风格（dwExStyle）。

DestroyWindow 函数用来销毁用上述两个函数建立的控件或者子窗口。

清单 8.5.1 中的程序利用预定义控件类创建了几种控件：静态框、按钮和单行编辑框。其中 hStaticWnd1 是建立在主窗口 hWnd 中的静态框；hButton1、hButton2、hEdit1、hStaticWnd2 则是建立在 hStaticWnd1 内部的几个控件，并作为 hStaticWnd1 的子控件而存在；而 hEdit2 是 hStaticWnd2 的子控件，是 hStaticWnd1 的子子控件。

清单 8.5.1 利用预定义控件类创建控件

```
#define IDC_STATIC1 100
#define IDC_STATIC2 150
#define IDC_BUTTON1 110
#define IDC_BUTTON2 120
#define IDC_EDIT1 130
#define IDC_EDIT2 140

/* 创建一个静态框 */
hStaticWnd1 = CreateWindow (CTRL_STATIC,
 "This is a static control",
 WS_CHILD | SS_NOTIFY | SS_SIMPLE | WS_VISIBLE | WS_BORDER,
 IDC_STATIC1,
 10, 10, 180, 300, hWnd, 0);

/* 在 hStaticWnd1 中创建两个按钮控件 */
hButton1 = CreateWindow (CTRL_BUTTON,
 "Button1",
 WS_CHILD | BS_PUSHBUTTON | WS_VISIBLE,
 IDC_BUTTON1,
 20, 20, 80, 20, hStaticWnd1, 0);

hButton2 = CreateWindow (CTRL_BUTTON,
 "Button2",
 WS_CHILD | BS_PUSHBUTTON | WS_VISIBLE,
 IDC_BUTTON2,
 20, 50, 80, 20, hStaticWnd1, 0);

/* 在 hStaticWnd1 中创建一个编辑框控件 */
hEdit1 = CreateWindow (CTRL_EDIT,
 "Edit Box 1",
 WS_CHILD | WS_VISIBLE | WS_BORDER,
 IDC_EDIT1,
 20, 80, 100, 24, hStaticWnd1, 0);

/* 在 hStaticWnd1 中创建一个静态框 hStaticWnd2 */
hStaticWnd2 = CreateWindow (CTRL_STATIC,
 "This is child static control",
 WS_CHILD | SS_NOTIFY | SS_SIMPLE | WS_VISIBLE | WS_BORDER,
 IDC_STATIC1,
 20, 110, 100, 50, hStaticWnd1, 0);

/* 在 hStaticWnd2 中创建一个编辑框 hEdit2, 这时, hEdit2 是 hStaticWnd1 的
孙窗口 */
hEdit2 = CreateWindow (CTRL_EDIT,
 "Edit Box 2",
 WS_CHILD | WS_VISIBLE | WS_BORDER,
 IDC_EDIT2,
 0, 20, 100, 24, hStaticWnd2, 0);
```

### 8.5.3 控件编程涉及的内容

在控件编程中，所涉及到的内容除了控件的创建和销毁之外，一般还涉及到如下主题：

- 控件具有自己的窗口风格定义，需要在创建控件时指定需要的风格，不同的风格将使得控件具有不同的表象和行为。
- 获取或设置控件的状态、内容等。一般可通过向控件发送一些通用或者特有的消息来完成。另外，针对窗口的通用函数一般都适用于控件，例如：ShowWindow、MoveWindow、EnableWindow、SetWindowFont 等等。
- 控件内部发生某种事件时，会通过 **通知消息** 通知其父窗口。通知消息一般通过 MSG\_COMMAND 消息发送，该消息的 wParam 参数由子窗口标识符和通知码组成，lParam 参数含有发出通知消息的控件句柄。例如，当用户修改编辑框中的内容时，编辑框会向父窗口发出 EN\_CHANGED 通知消息。如果父窗口的窗口过程需要了解这一变化，则应该在父窗口的窗口过程中如下处理该通知消息：

```
switch (message) {
 case MSG_COMMAND:
 {
 int id = LOWORD(wParam);
 int nc = HIWORD(wParam);
 if (id == ID_MYEDIT && nc == EN_CHANGE) {
 /* 用户修改了子窗口 ID_MYEDIT 编辑框的内容，现在做进一步处理...
*/
 }
 }
 break;
}
```

- MiniGUI 1.2.6 中针对控件的通知消息处理引入了 SetNotificationCallback 函数，该函数可以为控件设置一个通知消息的回调函数。当控件有通知消息时，将调用该函数，而不是发送通知消息到父窗口。新的应用程序应尽量使用这个函数来处理控件的通知消息，以便获得良好的程序结构。本指南示例程序全部使用这一接口来处理控件的通知消息。

清单 8.5.2 中的函数使用预定义控件类建立了一个简单的对话框。当用户在编辑框中输入以毫米为单位的数据时，系统将在编辑框之下的静态框中显示对应的以英寸为单位的数据，并在用户选择“确定”按钮时将用户输入的数据返回到调用该对话框的程序。

清单 8.5.2 使用预定义控件实现简单输入对话框

```
/*
** $Id: input.c,v 1.1 2003/06/04 01:38:14 weym Exp $
**
** Listing 5.2
**
** input.c: Sample program for MiniGUI Programming Guide
** Use dialog box to interact with user.
**
** Copyright (C) 2003 Feynman Software.
**
```



```

** License: GPL
*/
#include <stdio.h>
#include <stdlib.h>
#include <minigui/common.h>
#include <minigui/minigui.h>
#include <minigui/gdi.h>
#include <minigui/window.h>
#include <minigui/control.h>
/* 定义对话框模板 */
static DLGTEMPLATE DlgBoxInputLen
{
 WS_BORDER | WS_CAPTION,
 WS_EX_NONE,
 120, 150, 400, 160,
 "请输入长度",
 0, 0,
 4, NULL,
 0
};
#define IDC_SIZE_MM 100
#define IDC_SIZE_INCH 110
/*
 * 该对话框一共含有 4 个控件，分别用于显示提示信息、
 * 用户输入框、显示转换后的长度值以及关闭程序用的“确定”按钮。
 */
static CTRLDATA CtrlInputLen []
{
 {
 CTRL_STATIC,
 WS_VISIBLE | SS_SIMPLE,
 10, 10, 380, 18,
 IDC_STATIC,
 "请输入长度（单位：毫米）",
 0
 },
 {
 CTRL_EDIT,
 WS_VISIBLE | WS_TABSTOP | WS_BORDER,
 10, 40, 380, 24,
 IDC_SIZE_MM,
 NULL,
 0
 },
 {
 CTRL_BUTTON,
 WS_VISIBLE | WS_TABSTOP | WS_BORDER,
 10, 150, 380, 24,
 IDC_SIZE_INCH,
 "确定",
 0
 },
 {
 CTRL_BUTTON,
 WS_VISIBLE | WS_TABSTOP | WS_BORDER,
 10, 175, 380, 24,
 IDC_SIZE_MM,
 "取消",
 0
 }
};

```

```

{
 CTRL_STATIC,
 WS_VISIBLE | SS_SIMPLE,
 10, 70, 380, 18,
 IDC_SIZE_INCH,
 "相当于 0.00 英寸",
 0
},
{
 CTRL_BUTTON,
 WS_TABSTOP | WS_VISIBLE | BS_DEFPUSHBUTTON,
 170, 100, 60, 25,
 IDOK,
 "确定",
 0
}
};
/* 这是输入框的通知回调函数。*/
static void my_notif_proc (HWND hwnd, int id, int nc, DWORD add_data)
{
 /* 当输入框中的值发生变化时，取出值并转换成英寸显示在英寸框中。
 */
 if (id == IDC_SIZE_MM && nc == EN_CHANGE) {
 char buff [60];
 double len;
 GetWindowText (hwnd, buff, 32);
 len = atof (buff);
 len = len / 25.4;
 sprintf (buff, "相当于 %.5f 英寸", len);
 SetDlgItemText (GetParent (hwnd), IDC_SIZE_INCH, buff);
 }
}
/* 该对话框的窗口过程 */
static int InputLenDialogBoxProc (HWND hDlg, int message, WPARAM wParam, LPARAM lParam)
{
 switch (message) {
 case MSG_INITDIALOG:
 /* 将通过 DialogBoxIndirectParam 的最后一个参数传递进入的指针
 * 以窗口附加数据的形式保存下来，以便在以后使用。
 */
 SetWindowAdditionalData (hDlg, lParam);
 /* 设置控件的通知回调函数。
 */

```

```

 SetNotificationCallback (GetDlgItem (hDlg, IDC_SIZE_MM), my_notif_proc);
 }
 return 1;
}

case MSG_COMMAND:
 switch (wParam) {
 case IDOK:
 {
 char buff [40];
 /* 从输入框中获得数据，并保存在传入的指针中。
 */
 double* length = (double*) GetWindowAdditionalData (hDlg);
 GetWindowText (GetDlgItem (hDlg, IDC_SIZE_MM), buff, 32);
 *length = atof (buff);
 }
 case IDCANCEL:
 EndDialog (hDlg, wParam);
 break;
 }
 break;
}

return DefaultDialogProc (hDlg, message, wParam, lParam);
}

static void InputLenDialogBox (HWND hWnd, double* length)
{
 DlgBoxInputLen.controls = CtrlInputLen;

 DialogBoxIndirectParam (&DlgBoxInputLen, hWnd, InputLenDialogBoxProc,
 (LPARAM)length);
}

int MiniGUIMain (int argc, const char* argv[])
{
#ifdef _LITE_VERSION
 SetDesktopRect(0, 0, 800, 600);
#endif

 double length;
 InputLenDialogBox (HWND_DESKTOP, &length);
 /* 把用户在对话框中输入的值打印在终端上。
 */
 printf ("The length is %.5f mm.\n", length);
 return 0;
}

#endif _LITE_VERSION

```

```
#include <minigui/dti.c>
#endif
```

清单 8.5.2 程序的运行效果见图 8.5.1 该程序的完整源代码请见本指南示例程序包 samples 中的 input.c 文件。



图 8.5.1 简单输入对话框

在 MiniGUI 编程指南中介绍的预定义控件的各个章节, 主要从三个方面介绍所有预定义控件: 控件的用途、控件风格、控件消息以及控件的通知消息, 并给出控件的编程实例。

8.5.4 控件专用的操作函数

MiniGUI 提供了一些控件专用的操作函数, 见表 8.5.2。

表 8.5.2 通用控件操作函数

| 函数名称                    | 用途            | 备注                    |
|-------------------------|---------------|-----------------------|
| GetNotificationCallback | 获取控件的通知消息回调函数 | 在 MiniGUI 1.2.6 版本中出现 |
| SetNotificationCallback | 设置控件的通知消息回调函数 |                       |
| NotifyParentEx          | 发送控件通知消息      |                       |

## 8.6 图形设备接口

图形设备接口（GDI: Graphics Device Interface）是 GUI 系统的一个重要组成部分。通过 GDI，GUI 程序就可以在计算机屏幕上，或者其他的显示设备上进行图形输出，包括基本绘图和文本输出。

### 8.6.1 窗口绘制和刷新

#### 8.6.1.1 何时进行绘制

应用程序使用窗口来作为主要的输出设备，也就是说，MiniGUI应用程序在它的窗口之内进行绘制。

MiniGUI对整个屏幕上的显示输出进行管理。如果窗口移动之类的动作引起窗口内容的改变，MiniGUI对窗口内应该被更新的区域打上标志，然后给相应的应用程序窗口发送一个 `MSG_PAINT` 消息，应用程序收到该消息后就进行必要的绘制，以刷新窗口的显示。如果窗口内容的改变是由应用程序自己引起的，应用程序可以把受影响而需更新的窗口区域打上标志，并产生一个 `MSG_PAINT` 消息。

如果需要在窗口内绘制，应用程序首先要获得该窗口的设备上下文句柄。应用程序的大部分绘制操作是在处理MSG\_PAINT消息的过程中执行的，这时，应用程序应调用BeginPaint函数来获得设备上下文句柄。如果应用程序的某个操作要求立即的反馈，例如处理键盘和鼠标消息时，它可以立刻进行绘制而不用等待 MSG\_PAINT 消息。应用程序在其它时候绘制时可以调用GetDC或GetClientDC来获得设备上下文句柄。

#### 8.6.1.2 MSG\_PAINT 消息

通常应用程序在响应 `MSG_PAINT` 消息时执行窗口绘制。如果窗口的改变影响到客户区的内容，或者窗口的无效区域不为空，MiniGUI 就给相应的窗口过程函数发送 `MSG_PAINT` 消息。

接收到 `MSG_PAINT` 消息时，应用程序应调用 `BeginPaint` 函数来获得设备上下文句柄，并用它调用 GDI 函数来执行更新客户区所必需的绘制操作。绘制结束之后，应用程序应调用 `EndPaint` 函数释放设备上下文句柄。

`BeginPaint` 函数用来完成绘制窗口之前的准备工作。它首先通过 `GetClientDC` 函数获得窗口客户区的设备上下文，把设备上下文的剪切域设置为窗口的无效区域。窗口内只有那些改变了的区域才重新绘制，对剪切域以外的任何绘制尝试都被裁减掉，不会出现在屏幕上。为了不影响绘制操作，`BeginPaint` 函数隐藏了插入符。最后，`BeginPaint` 将窗口的无效区域清除，避免不断产生 `MSG_PAINT` 消息，然后返回所获取的设备上下文句柄。

`MSG_PAINT` 消息的 `lParam` 参数为窗口的无效区域指针，应用程序可以用窗口的无效区域信息来优化绘制。例如把绘制限制在窗口的无效区域之内。如果应用程序的输出很简单，

就可以忽略更新区域而在整个窗口内绘制，由 MiniGUI 来裁剪剪切域外的不必要的绘制，只有无效区域内的绘制才是可见的。

应用程序绘制完成之后应调用 EndPaint 函数终结整个绘制过程。EndPaint 函数的主要工作是调用 ReleaseDC 函数释放由 GetClientDC 函数获取的设备上下文，此外，它还要显示被 BeginPaint 函数隐藏的插入符。

### 8.6.1.3 有效区域和无效区域

更新区域（无效区域）指的是窗口内过时的或无效的需要重新绘制的区域。MiniGUI 根据需要更新的区域为应用程序产生 MSG\_PAINT 消息，应用程序也可以通过设置无效区域来产生 MSG\_PAINT 消息。

应用程序可以使用 InvalidateRect 函数来使窗口的某一区域无效。该函数原型如下：  
 BOOL WINAPI InvalidateRect (HWND hWnd, const RECT\* prc, BOOL bEraseBkgnd)

各参数含义如下：

hWnd            需要更新的窗口句柄  
 prc            指向无效矩形的指针  
 bEraseBkgnd   是否擦除窗口背景

InvalidateRect 函数把给定的矩形区域添加到指定窗口的更新区域中。该函数把给定的矩形和应用程序窗口先前的更新区域合并，然后投递一个 MSG\_PAINT 消息到该窗口的消息队列中。

如果 bEraseBkgnd 为 TRUE，应用程序窗口将收到一个 MSG\_ERASEBKGD 消息，窗口过程可以处理该消息，自行擦除窗口背景。如果应用程序不处理 MSG\_ERASEBKGD 消息而将它传给 DefaultMainWinProc，MiniGUI 对 MSG\_ERASEBKGD 消息的默认处理方式是以窗口的背景色为画刷擦除背景。

窗口背景是指绘制窗口之前用于填充客户区的颜色和风格。窗口背景可以覆盖屏幕上窗口客户区所在区域的原有内容，使得应用程序的输出显示不受屏幕已有内容的干扰。

MSG\_ERASEBKGD 消息的 lParam 参数包含了一个 RECT 结构指针，指明应该擦除的矩形区域，应用程序可以使用该参数来绘制窗口背景。绘制完成之后，应用程序可以直接返回零，无需调用 DefaultMainWinProc 进行缺省的消息处理。有关处理 MSG\_ERASEBKGD 消息的示例，可参阅 MINIGUI 编程指南第 3 章中的相关章节。

## 8.6.2 图形设备上下文

### 8.6.2.1 图形设备的抽象

应用程序一般在一个图形上下文（graphics context）上调用图形系统提供的绘制原语进行绘制。上下文是一个记录了绘制原语所使用的图形属性的对象。这些属性通常包括：

- 前景色（画笔），绘制时所使用的颜色值或图像。
- 背景色或填充位图（画刷），绘制原语在填充时所使用的颜色或图像。
- 绘制模式，描述前景色与已有的屏幕颜色如何组合。常见的选项是覆盖已有的屏幕内容或把绘制颜色和屏幕颜色进行“XOR”位逻辑运算。XOR 模式使得绘制对象可以通过重绘进行擦除。

■ 填充模式，描述背景色或图像与屏幕颜色如何组合。常见的选项是覆盖或透明，也就是忽略背景和已有的屏幕内容。

■ 颜色掩蔽，它是一个位图，用于决定绘制操作对屏幕像素影响的风格。

■ 线形，它的宽度、端型和角型。

■ 字体，字体通常是对应于某个字符集的一组位图。字体一般通过指定其大小、磅值、类别和字符集等属性进行选择。

■ 绘制区域，在概念上是一个大小和位置可以为任意值的、映射到窗口之上的视口。可以通过改变视口的原点来移动视口。有时候系统允许视口的缩放。

■ 剪切域，在该区域内绘制原语才有实效。剪切域之外的输出将不被绘制出来。剪切域主要用于重绘，由各个窗口的有变化的区域相交而成。应用程序可以调整剪切域，增加需要改变的区域。

■ 当前位置，例如可以通过 MoveTo 和 LineTo 等绘制原语来画线。

MiniGUI 采用了在 Windows 和 X Window 等 GUI 系统中普遍采用的图形设备上下文 (Device Context, DC, 也称作“图形设备环境”) 的概念。每个图形设备上下文定义了图形输出设备或内存中的一个矩形的显示输出区域，以及相关的图形属性。在调用图形输出函数时，均要求指定经初始化的图形设备上下文。也就是说，所有的绘制操作都必须在某个图形设备上下文之内起作用。

从程序员的角度看，一个经过初始化的图形设备上下文定义了一个图形设备环境，确定了之后在其上进行图形操作的一些基本属性，并一直保持这些属性，直到被改变为止。这些属性包括：输出的线条颜色、填充颜色、字体颜色、字体形状等等。而从 GUI 系统角度来讲，一个图形设备上下文所代表的含义就要复杂得多，它起码应该包含如下内容：

■ 该设备上下文本所在设备信息（显示模式、色彩深度、显存布局等等）；

■ 该设备上下文所代表的窗口以及该窗口被其他窗口剪切的信息（在 MiniGUI 中，称作“全局剪切域”）；

■ 该设备上下文的基本操作函数（点、直线、多边形、填充、块操作等），及其上下文信息；

■ 由程序设定的局部信息（绘图属性、映射关系和局部剪切域等）。

当你想在一个图形输出设备（如显示器屏幕）上绘图时，你首先必须获得一个设备上下文的句柄。然后在 GDI 函数中将该句柄作为一个参数，标识你在绘图时所要使用的图形设备上下文。

设备上下文中包含许多确定 GDI 函数如何在设备上工作的当前属性，这些属性使得传递给 GDI 函数的参数可以只包含起始坐标或者尺寸信息，而不必包含在设备上显示对象时需要的其它信息，因为这些信息是设备上下文的一部分。当你想改变这些属性之一时，你可以调用一个可以改变设备上下文属性的函数，以后针对该设备上下文的 GDI 函数调用将使用改变后的属性。

设备上下文实际上是 GDI 内部保存的数据结构。设备上下文与特定的显示设备相关。设备上下文中的有些值是图形化的属性，这些属性定义了一些 GDI 绘图函数工作情况的特殊内容。例如，对于 TextOut 函数，设备上下文的属性确定了文本的颜色、背景色、x 坐标和 y 坐标映射到窗口客户区的方式，以及显示文本时使用的字体。

当程序需要绘图时，它必须首先获取设备上下文句柄。设备上下文句柄是一个代表设备上下文的数值，程序使用该句柄。

### 8.6.2.2 设备上下文句柄的获取和释放

在MiniGUI中，所有绘图相关的函数均需要有一个设备上下文。当程序需要绘图时，它必须首先获取设备上下文句柄。程序绘制完毕之后，它必须释放设备上下文句柄。程序必须在处理单个消息期间获取和释放设备上下文句柄，也就是说，如果程序在处理一条消息时获取了设备上下文句柄，它必须在处理完该消息退出窗口过程函数之前释放该设备上下文句柄。

获取和释放设备上下文的常用方法之一是通过BeginPaint和EndPaint函数。这两个函数的原型如下（window.h）：

```
HDC GUIAPI BeginPaint(HWND hWnd);
```

```
void GUIAPI EndPaint(HWND hWnd, HDC hdc);
```

但需要注意的是，这两个函数只能在处理 MSG\_PAINT 的消息中调用。一般地，MSG\_PAINT 消息的处理通常有如下的形式：

```
MSG_PAINT:
```

```
 HDC hdc = BeginPaint (hWnd);
 /* 使用 GDI 函数进行图形绘制 */
 EndPaint (hWnd, hdc);
 return 0;
}
```

BeginPaint 以和窗口过程函数相对应的窗口句柄 hWnd 为参数，返回一个设备上下文句柄。然后 GDI 函数就可以使用该设备上下文句柄进行图形操作。

在典型的图形用户界面环境（包括 MiniGUI）中，应用程序一般只能在窗口的客户区绘制文本和图形，但是图形系统并不确保客户区的绘制内容会一直保留下去。如果该程序窗口的客户区被另一个窗口覆盖，图形系统不会保存程序窗口被覆盖区域的内容，图形系统把这个任务留给应用程序自己完成。在需要恢复程序窗口某部分的内容时，图形系统通常会通知程序刷新该部分客户区。MiniGUI 通过向应用程序发送 MSG\_PAINT 消息来通知应用程序进行窗口客户区的绘制操作。也就是说，由于 MiniGUI 是消息驱动的系统，MiniGUI 的应用程序只有在需要的情况下（一般是程序收到 MSG\_PAINT 消息）才进行绘制。如果程序觉得有必要更新客户区的内容，它可以主动产生一个 MSG\_PAINT 消息，从而使客户区重绘。

一般来说，在以下情况下，MiniGUI 程序的窗口过程会接收到一个 MSG\_PAINT 消息：

- 用户移动窗口或显示窗口时，MiniGUI 向先前被隐藏的窗口发送 MSG\_PAINT 消息；
- 程序使用 InvalidateRect 函数来更新窗口的无效区域，这将产生一个 MSG\_PAINT 消息；
- 程序调用 UpdateWindow 函数来重绘窗口；
- 覆盖程序窗口的对话框或消息框被消除；
- 下拉或弹出菜单被消除。

在某些情况下，MiniGUI 保存某些被覆盖掉的显示区域，然后在需要的时候恢复，例如鼠标光标的移动。

一般情况下，窗口过程函数只需要更新客户区的一部分。例如，显示对话框覆盖了客户区的一部分；在擦除了对话框之后，只需要重新绘制之前被对话框覆盖的客户区部分。这个区域称为“无效区域”。

MiniGUI 在 BeginPaint 函数中通过 GetClientDC 获取客户区设备上下文，然后将窗口当前的无效区域选择到窗口的剪切区域中；而 EndPaint 函数则清空窗口的无效区域，并释放设备上下文。



因为 BeginPaint 函数将窗口的无效区域选择到了设备上下文中，所以，可以通过一些必要的优化来提高 MSG\_PAINT 消息的处理效率。比如，某个程序要在窗口客户区中填充若干矩形，就可以在 MSG\_PAINT 函数中如下处理：

MSG\_PAINT:

```
{
 HDC hdc = BeginPaint (hWnd);

 for (j = 0; j < 10; j++) {
 if (RectVisible (hdc, rcs + j)) {
 FillBox (hdc, rcs[j].left, rcs[j].top, rcs[j].right, rcs[j].bottom);
 }
 }

 EndPaint (hWnd, hdc);
 return 0;
}
```

这样可以避免不必要的重绘操作，从而提高绘图效率。

设备上下文可通过 GetClientDC 和 ReleaseDC 获取和释放。由 GetDC 所获取的设备上下文是针对整个窗口的，而 GetClientDC 所获取的设备上下文是针对窗口客户区，也就是说，前一个函数获得的设备上下文，其坐标原点位于窗口左上角，输出被限定在窗口范围之内；后一个函数获得的设备上下文，其坐标原点位于窗口客户区左上角，输出被限定在窗口客户区范围之内。下面是这三个函数的原型说明（gdi.h）：

HDC GUIAPI GetDC (HWND hwnd);

HDC GUIAPI GetClientDC (HWND hwnd);

void GUIAPI ReleaseDC (HDC hdc);

GetDC 和 GetClientDC 是从系统预留的若干个 DC 当中获得一个目前尚未使用的设备上下文。所以，应该注意如下两点：

1. 在使用完成一个由 GetDC 或 GetClientDC 返回的设备上下文之后，应该尽快调用 ReleaseDC 释放。
2. 避免同时使用多个设备上下文，并避免在递归函数中调用 GetDC 和 GetClientDC。

为了方便程序编写，提高绘图效率，MiniGUI 还提供了建立私有设备上下文的函数，所建立的设备上下文在整个窗口生存期内有效，从而免除了获取和释放的过程。这些函数的原型如下：

HDC GUIAPI CreatePrivateDC (HWND hwnd);

HDC GUIAPI CreatePrivateClientDC (HWND hwnd);

HDC GUIAPI GetPrivateClientDC (HWND hwnd);

void GUIAPI DeletePrivateDC (HDC hdc);

在建立主窗口时，如果主窗口的扩展风格中指定了 WS\_EX\_USEPRIVATEDC 风格，则 CreateMainWindow 函数会自动为该窗口的客户区建立私有设备上下文。通过 GetPrivateClientDC 函数，可以获得该设备上下文。对控件而言，如果控件类具有 CS\_OWNDC 属性，则所有属于该控件类的控件将自动建立私有设备上下文。DeletePrivateDC 函数用来删除私有设备上下文。对上述两种情况，系统将在销毁窗口时自动调用 DeletePrivateDC 函数。

### 8.6.2.3 系统内存中的设备上下文

MiniGUI 也提供了内存设备上下文的创建和销毁函数。利用内存设备上下文，可以在系统内存中建立一个类似显示内存的区域，然后在该区域中进行绘图操作，结束后再复制到显示内存中。这种绘图方法有许多好处，比如速度很快，减少直接操作显存造成的闪烁现象等等。用来建立和销毁内存设备上下文的函数原型如下(gdi.h)：

```
401 HDC GUIAPI CreateCompatibleDC (HDC hdc);
402 void GUIAPI DeleteCompatibleDC (HDC hdc);
```

### 8.6.2.4 屏幕设备上下文

MiniGUI 在启动之后，就建立了一个全局的屏幕设备上下文。该 DC 是针对整个屏幕的，并且没有任何预先定义的剪切域。在某些应用程序中，可以直接使用该设备上下文进行绘图，将大大提高绘图效率。在 MiniGUI 中，屏幕设备上下文用 HDC\_SCREEN 标识，不需要进行任何获取和释放操作。

## 8.6.3 基本的图形绘制

### 8.6.3.1 基本绘图属性

在了解基本绘图函数之前，我们首先了解一下基本绘图属性。在 MiniGUI 的目前版本中，绘图属性比较少，大体包括线条颜色、填充颜色、文本背景模式、文本颜色、TAB 键宽度等等。表 26.1 给出了这些属性的操作函数。

表 8.6.1 基本绘图属性及其操作函数

| 绘图属性    | 操作函数                        | 受影响的 GDI 函数             |
|---------|-----------------------------|-------------------------|
| 线条颜色    | GetPenColor/SetPenColor     | LineTo、Circle、Rectangle |
| 填充颜色    | GetBrushColor/SetBrushColor | FillBox                 |
| 文本背景模式  | GetBkMode/SetBkMode         | TextOut、DrawText        |
| 文本颜色    | GetTextColor/SetTextColor   | 同上                      |
| TAB 键宽度 | GetTabStop/SetTabStop       | 同上                      |

MiniGUI 目前版本中还定义了刷子和笔的若干函数，这些函数是为将来兼容性而定义的，目前无用。

### 8.6.3.2 基本绘图函数

MiniGUI 中的基本绘图函数为点、线、圆、矩形、调色板操作等基本函数，原型定义如下：

```
void GUIAPI SetPixel (HDC hdc, int x, int y, gal_pixel c);
void GUIAPI SetPixelRGB (HDC hdc, int x, int y, int r, int g, int b);
gal_pixel GUIAPI GetPixel (HDC hdc, int x, int y);
```

```
void GUIAPI GetPixelRGB (HDC hdc, int x, int y, int* r, int* g, int* b);
gal_pixel GUIAPI RGB2Pixel (HDC hdc, int r, int g, int b);
```

```
void GUIAPI LineTo (HDC hdc, int x, int y);
void GUIAPI MoveTo (HDC hdc, int x, int y);
```

```
void GUIAPI Circle (HDC hdc, int x, int y, int r);
void GUIAPI Rectangle (HDC hdc, int x0, int y0, int x1, int y1);
```

这里有两个基本的概念需要明确区分，即像素值和 RGB 值。RGB 是计算机中通过三原色的不同比例表示某种颜色的方法。通常，RGB 中的红、绿、蓝可取 0~255 当中的任意值，从而可以表示 255x255x255 种不同的颜色。而在显示内存当中，要显示在屏幕上的颜色并不是用 RGB 这种方式表示的，显存当中保存的其实是所有像素的像素值。像素值的范围根据显示模式的不同而变化。在 16 色显示模式下，像素值范围为 [0, 15]；而在 256 色模式下，像素值范围为 [0, 255]；在 16 位色模式下，像素值范围为 [0, 2<sup>16</sup>-1]。通常我们所说显示模式是多少位色，就是指像素的位数。

在 MiniGUI 中，设置某个像素点的颜色，既可以直接使用像素值 (SetPixel)，也可以间接通过 RGB 值来设置 (SetPixelRGB)，并且通过 RGB2Pixel 函数，可以将 RGB 值转换为像素值。

### 8.6.3.3 剪切域操作函数

在利用设备上下文进行绘图时，还可以进行剪切处理。MiniGUI 提供了如下函数完成对指定设备上下文的剪切处理 (include/gdi.h)：

```
// Clipping support
void GUIAPI ExcludeClipRect (HDC hdc, int left, int top,
 int right, int bottom);
void GUIAPI IncludeClipRect (HDC hdc, int left, int top,
 int right, int bottom);
void GUIAPI ClipRectIntersect (HDC hdc, const RECT* prc);
void GUIAPI SelectClipRect (HDC hdc, const RECT* prc);
void GUIAPI SelectClipRegion (HDC hdc, const CLIPRGN* pRgn);
void GUIAPI GetBoundsRect (HDC hdc, RECT* pRect);
BOOL GUIAPI PtVisible (HDC hdc, const POINT* pPt);
BOOL GUIAPI RectVisible (HDC hdc, const RECT* pRect);
```

ExcludeClipRect 从设备上下文的当前可见区域中排除给定的矩形区域，设备上下文的可见区域将缩小；IncludeClipRect 向当前设备上下文的可见区域中添加一个矩形区域，设备上下文的可见区域将扩大；ClipRectIntersect 将设备上下文的可见区域设置为已有区域和给定矩形区域的交集；SelectClipRect 将设备上下文的可见区域重置为一个矩形区域；SelectClipRegion 将设备上下文的可见区域设置为一个指定的区域；GetBoundsRect 获取当前可见区域的外包最小矩形；PtVisible 和 RectVisible 用来判断给定的点或者矩形是否可见，即是否全部或部分落在可见区域当中。

## 8.6.4 文本的处理和显示

### 8.6.4.1 逻辑字体

MiniGUI 的逻辑字体功能强大，它包括了字符集、字体类型、风格等等丰富的信息，不仅仅可以用来输出文本，而且可以用来分析多语种文本的结构。这在许多文本排版应用中非常有用。在使用 MiniGUI 的逻辑字体之前，首先要创建逻辑字体，并且将其选择到要使这种逻辑字体进行文本输出的设备上下文当中。每个设备上下文的默认逻辑字体是 MiniGUI.cfg 中定义的系统默认字体。你可以调用 `CreateLogFont`、`CreateLogFontByName` 以及 `CreateLogFontIndirect` 三个函数来建立逻辑字体，并利用 `SelectFont` 函数将逻辑字体选择到指定的设备上下文中，在使用结束之后，用 `DestroyLogFont` 函数销毁逻辑字体。注意你不能销毁正被选中的逻辑字体。这几个函数的原型如下（include/gdi.h）：

```
PLOGFONT GUIAPI CreateLogFont (const char* type, const char* family,
 const char* charset, char weight, char slant, char set_width,
 char spacing, char underline, char struckout,
 int size, int rotation);
PLOGFONT GUIAPI CreateLogFontByName (const char* font_name);
PLOGFONT GUIAPI CreateLogFontIndirect (LOGFONT* logfont);
void GUIAPI DestroyLogFont (PLOGFONT log_font);

void GUIAPI GetLogFontInfo (HDC hdc, LOGFONT* log_font);

PLOGFONT GUIAPI GetSystemFont (int font_id);

PLOGFONT GUIAPI GetCurFont (HDC hdc);
PLOGFONT GUIAPI SelectFont (HDC hdc, PLOGFONT log_font);
```

下面的程序段建立了多个逻辑字体：

```
static LOGFONT *logfont, *logfontgb12, *logfontbig24;
logfont = CreateLogFont (NULL, "SansSerif", "IS08859-1",
 FONT_WEIGHT_REGULAR, FONT_SLANT_ITALIC, FONT_SETWIDTH_NORMAL,
 FONT_SPACING_CHARCELL, FONT_UNDERLINE_NONE, FONT_STRUCKOUT_LINE,
 16, 0);
logfontgb12 = CreateLogFont (NULL, "song", "GB2312",
 FONT_WEIGHT_REGULAR, FONT_SLANT_ROMAN, FONT_SETWIDTH_NORMAL,
 FONT_SPACING_CHARCELL, FONT_UNDERLINE_LINE, FONT_STRUCKOUT_LINE,
 12, 0);
logfontbig24 = CreateLogFont (NULL, "ming", "BIG5",
 FONT_WEIGHT_REGULAR, FONT_SLANT_ROMAN, FONT_SETWIDTH_NORMAL,
 FONT_SPACING_CHARCELL, FONT_UNDERLINE_LINE, FONT_STRUCKOUT_NONE,
 24, 0);
```

其中，第一个字体，即 `logfont` 是属于字符集 IS08859-1 的字体，并且选用 SansSerif 体，大小为 16 像素高；`logfontgb12` 是属于字符集 GB2312 的字体，并选用 song 体（宋体），

大小为 12 像素高；logfontbig24 是属于字符集 BIG5 的字体，并选用 ming 体（明体）。

我们还可以调用 GetSystemFont 函数返回指定的系统逻辑字体，其中 font\_id 参数可取如下值：

- SYSLOGFONT\_DEFAULT：系统默认字体，必须是单字节字符集逻辑字体，必须由 RBF 设备字体组成。
- SYSLOGFONT\_WCHAR\_DEF：系统默认多字节字符集字体，通常由 RBF 设备字体组成，并且多字节字体的宽度是 SYSLOGFONT\_DEFAULT 逻辑字体的两倍。
- SYSLOGFONT\_FIXED：固定宽度的系统字体。
- SYSLOGFONT\_CAPTION：用于显示标题栏文本的逻辑字体。
- SYSLOGFONT\_MENU：用于显示菜单文本的逻辑字体。
- SYSLOGFONT\_CONTROL：用于控件的默认逻辑字体。

GetCurFont 函数返回当前选中的逻辑字体，注意不要调用 DestroyLogFont 删除系统逻辑字体。

### 8.6.4.2 文本输出

以下函数可以用来计算逻辑字体的输出长度和高度信息(gdi.h)：

```
int GUIAPI GetTextExtentPoint (HDC hdc, const char* text, int len, int max_extent,
 int* fit_chars, int* pos_chars, int* dx_chars, SIZE* size);
```

```
// Text output support
int GUIAPI GetFontHeight (HDC hdc);
int GUIAPI GetMaxFontWidth (HDC hdc);
void GUIAPI GetTextExtent (HDC hdc, const char* spText, int len, SIZE* pSize);
void GUIAPI GetTabbedTextExtent (HDC hdc, const char* spText, int len, SIZE* pSize);
```

GetTextExtentPoint 函数计算在给定的输出宽度内输出多字节文本时（即输出的字符限制在一定的宽度当中），可输出的最大字符个数、每个字符所在的字节位置、每个字符的输出位置，以及实际的输出高度和宽度。GetTextExtentPoint 函数是个综合性的函数，它对编辑器类型的应用程序非常有用，比如 MiniGUI 的单行和多行编辑框控件中，就使用这个函数来计算插入符的位置信息。

GetFontHeight 和 GetMaxFontWidth 则返回逻辑字体的高度和最大字符宽度。GetTextExtent 计算文本的输出高度和宽度。GetTabbedTextExtent 函数返回格式化字符串的输出高度和宽度。

以下函数用来输出文本(include/gdi.h)：

```
int GUIAPI TextOutLen (HDC hdc, int x, int y, const char* spText, int len);
int GUIAPI TabbedTextOutLen (HDC hdc, int x, int y, const char* spText, int len);
int GUIAPI TabbedTextOutEx (HDC hdc, int x, int y, const char* spText, int nCount,
 int nTabPositions, int *pTabPositions, int nTabOrigin);
void GUIAPI GetLastTextOutPos (HDC hdc, POINT* pt);

// Compatibility definitions
#define TextOut(hdc, x, y, text) TextOutLen (hdc, x, y, text, -1)
```

```
#define TabbedTextOut(hdc, x, y, text) TabbedTextOutLen (hdc, x, y, text, -1)
...
```

```
int GUIAPI DrawTextEx (HDC hdc, const char* pText, int nCount,
 RECT* pRect, int nIndent, UINT nFormat);
```

TextOutLen 函数用来在给定位置输出指定长度的字符串，若长度为-1，则字符串必须以'\0'结尾的。TabbedTextOutLen 函数用来输出格式化字符串。TabbedTextOutEx 函数用来输出格式化字符串，但可以指定字符串中每个 TAB 键的位置。

DrawText 是功能最复杂的文本输出函数，可以以不同的对齐方式在指定的矩形内部输出文本。

清单 8.6.1 中的程序段，就根据要输出的字符串所描述的那样，调用 DrawText 函数进行对齐文本输出。该程序的完整源代码见 MDE 中的 fontdemo.c 程序。图 8.6.1 是该程序段的输出效果。

清单 8.6.1 DrawText 函数的使用

```
void OnModeDrawText (HDC hdc)
{
 RECT rc1, rc2, rc3, rc4;
 const char* szBuff1 = "This is a good day. \n"
 "这是利用 DrawText 绘制的文本，使用字体 GB2312 Song 12. "
 "文本垂直靠上，水平居中";
 const char* szBuff2 = "This is a good day. \n"
 "这是利用 DrawText 绘制的文本，使用字体 GB2312 Song 16. "
 "文本垂直靠上，水平靠右";
 const char* szBuff3 = "单行文本垂直居中，水平居中";
 const char* szBuff4 =
 "这是利用 DrawTextEx 绘制的文本，使用字体 GB2312 Song 16. "
 "首行缩进值为 32. 文本垂直靠上，水平靠左";
 rc1.left = 1; rc1.top = 1; rc1.right = 401; rc1.bottom = 101;
 rc2.left = 0; rc2.top = 110; rc2.right = 401; rc2.bottom = 351;
 rc3.left = 0; rc3.top = 361; rc3.right = 401; rc3.bottom = 451;
 rc4.left = 0; rc4.top = 461; rc4.right = 401; rc4.bottom = 551;
 SetBkColor (hdc, COLOR_lightwhite);
 Rectangle (hdc, rc1.left, rc1.top, rc1.right, rc1.bottom);
 Rectangle (hdc, rc2.left, rc2.top, rc2.right, rc2.bottom);
 Rectangle (hdc, rc3.left, rc3.top, rc3.right, rc3.bottom);
 Rectangle (hdc, rc4.left, rc4.top, rc4.right, rc4.bottom);
 InflateRect (&rc1, -1, -1);
 InflateRect (&rc2, -1, -1);
 InflateRect (&rc3, -1, -1);
 InflateRect (&rc4, -1, -1);
 SelectFont (hdc, logfontgb12);
 DrawText (hdc, szBuff1, -1, &rc1, DT_NOCLIP | DT_CENTER | DT_WORDBREAK);
 SelectFont (hdc, logfontgb16);
 DrawText (hdc, szBuff2, -1, &rc2, DT_NOCLIP | DT_RIGHT | DT_WORDBREAK);
```

```

SelectFont (hdc, logfontgb24);
DrawText (hdc, szBuff3, -1, &rc3, DT_NOCLIP | DT_SINGLELINE | DT_CENTER |
DT_VCENTER);
SelectFont (hdc, logfontgb16);
DrawTextEx (hdc, szBuff4, -1, &rc4, 32, DT_NOCLIP | DT_LEFT | DT_WORDBREAK);
}

```

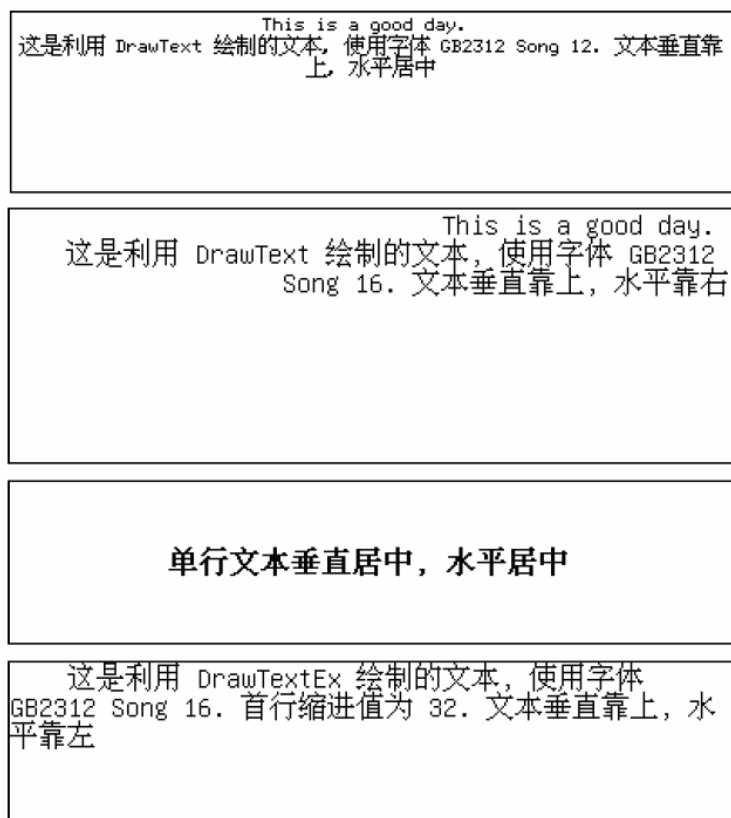


图 8.6.1 DrawText 函数的输出效果

## 8.7 示波器例子

要求：

1. 两种采样模式选择：实时和非实时；
2. 可以设置采样速度、采样开始时间；
- 2、非实时模式时。

有三种触发模式选择：无触发、电平触发、边沿触发；

电平触发时触发方式有：高电平、低电平触发；

边沿触发时触发方式有：上升沿、下降沿触发；

并且可以设置触发值。

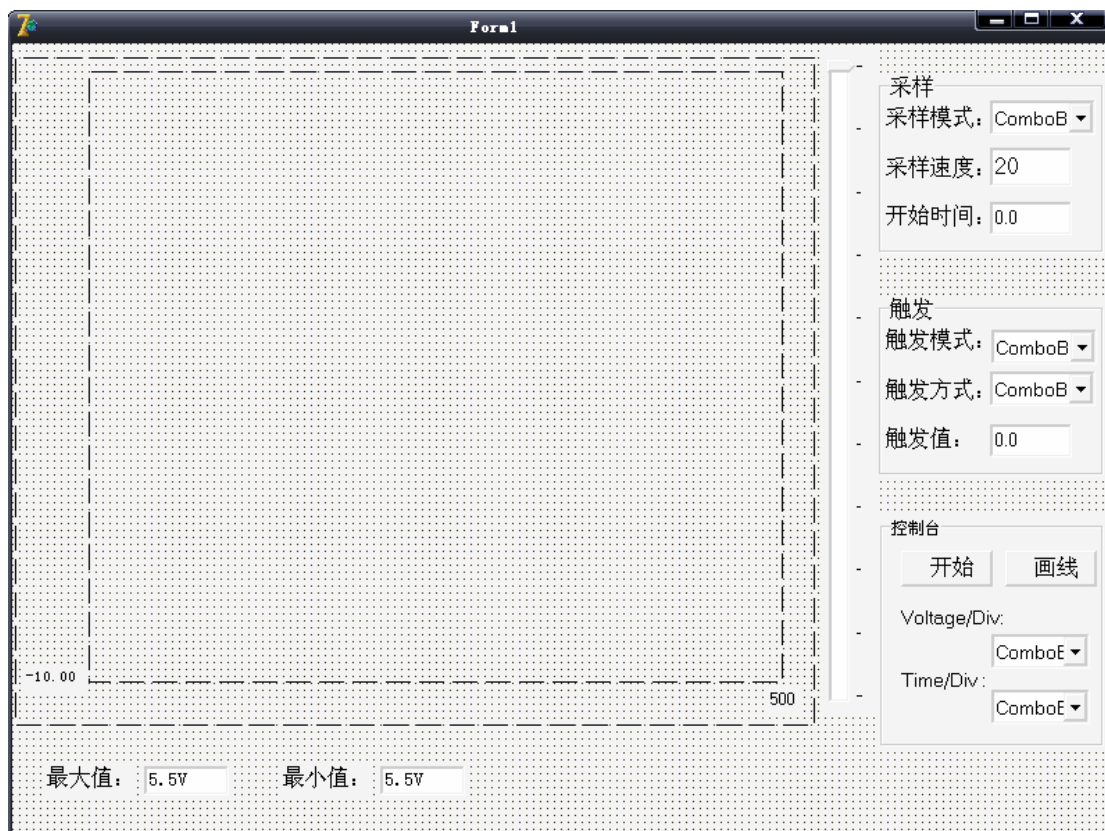
### 8.7.1 界面构建

**第一步，在delphi中画出想要的界面。**之所以先在delphi中先画，目的是取得各个控件

的坐标。

**声明：**当然，你可以用其他工具绘制界面取得坐标；你也可以不用其他工具先画界面，只要应用界面中的各个控件的坐标都能确切知道。

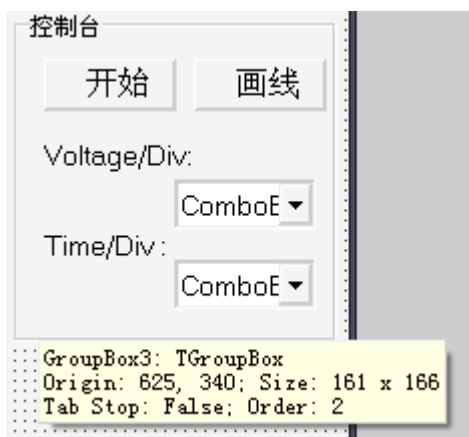
如下图所示：



## 第二步，绘制minigui中的界面。

从delphi中取得控件坐标，在minigui中放入坐标。

本例中，例如，从delphi中取得“控制台”这个GROUPBOX这个控件的坐标：（625，340），大小：161\*166。

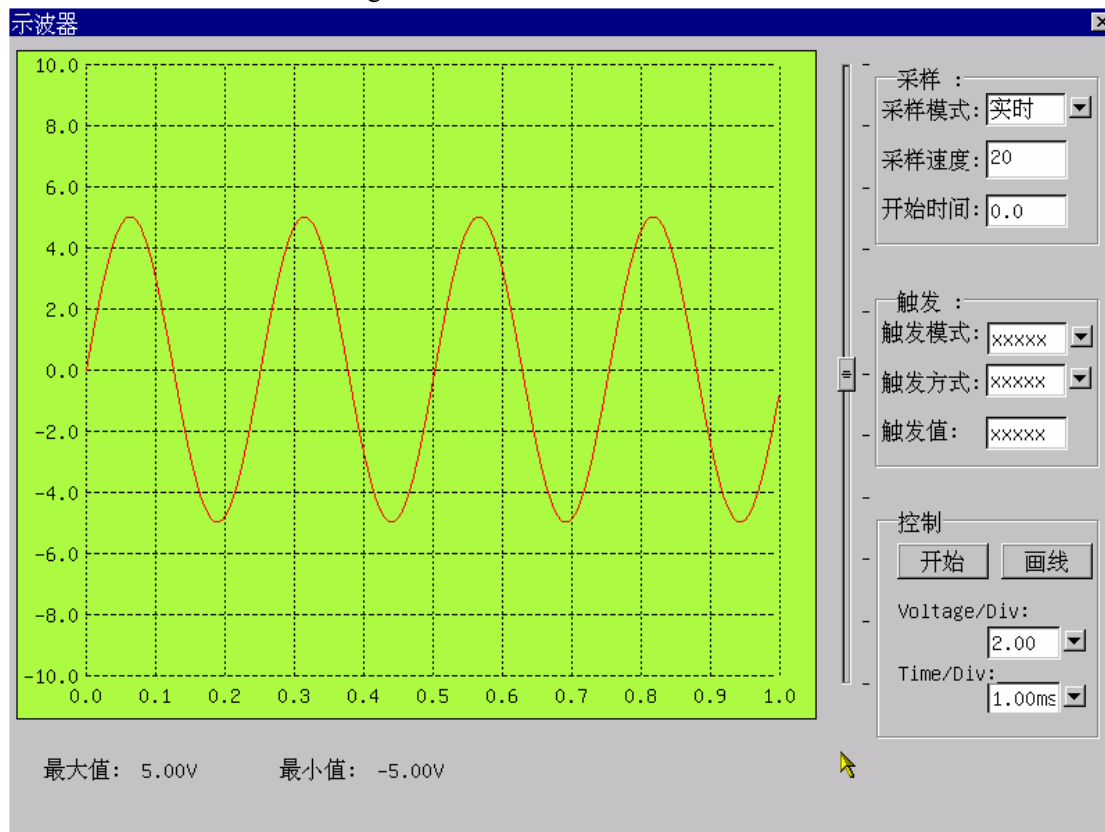


然后Minigui中创建“控制台”这个GROUPBOX这个控件如下（这里用的是主窗口创建，没有使用对话框）：



```
CreateWindow (CTRL_STATIC,
"控制",
WS_CHILD | SS_GROUPBOX | WS_VISIBLE,
IDC_CONSOLE_STATIC,
625, 340, 161, 166, hWnd, 0);
```

所有控件绘制完后的minigui界面如下：



## 8.7.2 后台处理

### 1. 坐标轴和波形的绘制

**虚线框：**minigui中没有画虚线的直接函数，需要自己手动绘制，这里可以考虑将一直线划分成N段小直线来画，用moveto、lineto语句来实现。

**坐标轴标签：**坐标轴上的标签存在数组里，显示出来时候，先用C库里的sprintf（）函数将数据转换为字符串，然后用minigui的DrawText（）函数来实现，DrawText（）函数比TextOut（）函数具有更多的参数，可以精确定位输出文本，具体参数可以参照API手册。

**波形：**本图中的波形为500点的实际数据，存放在一数组中，绘制波形时，先将500点实际数据转换为屏幕上的屏幕坐标，然后采用描点函数SetPixel（）将500点的数据描出来；如果相邻数据值差距较大，那么曲线就不是很明显，而是一点点的，这时候可以用LineTo（）函数，将这500点用线连接起来。

**注意：**当改变Voltage/Div的时候，波形要随之发生变化，因此，当Voltage/Div下拉框的选择发生变化时，要使用InvalidateRect (hWnd, &rc, 1)函数更新绘图区域（通过&rc指定所要更新区域的指针）。

## 2. 采样模式、触发模式、触发方式的选择

根据要求，实时模式下，没有触发这个概念；非实时模式下，要对触发模式、方式进行选择。因此，触发里面的选择项，要根据采样模式的选择而变化。

选择实时模式：触发里面没有任何选择；

选择非实时模式：触发模式下拉框有3个选项：无触发、电平触发、边沿触发；

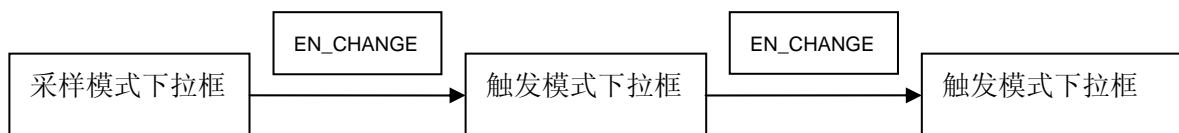
选择触发模式后，触发方式里的选择项，也要根据触发模式的选择而变化：

选择无触发：触发方式里没有任何选项；

选择电平触发：触发方式里有2选项：高电平、低电平；

选择边沿触发：触发方式里有2选项：上升沿、下降沿；

根据上述描述，在回调函数中，可以采用下拉组合框控件通知码里的EN\_SELCHANGE来实现，即采样模式中的选项发生变化，那么就发出通知码EN\_SELCHANGE通知下面触发模式的选项也发生变化；触发模式的选择发生改变后，利用通知码EN\_SELCHANGE通知触发方式里的选项也发生变化。即通知码过程如下：



取得通知码：

```
CODE=HIWORD(wParam); // 通知码在wParam参数的高字节（wParam的低字节为发出消息控件的ID）
```

判断通知码：

```
if (CODE != CBN_SELCHANGE) break; // 如果没有发生变化，退出；若发生变化，按照上述过程处理。
```

.....

**注意：**使用通知码的时候，在创建下拉框控件时，要在其风格里面添加 CBS\_NOTIFY 风格。不然，当你选择下拉框里的选项发生变化时，是不产生任何通知码的。

### 参考资料：

1. << MINIGUI-PROG-GUIDE-V1.3-C>>
2. MiniGUI 1.3.3 移植详解

## 第九章 展望

本手册介绍的嵌入式 Linux 系统平台目前只是为了实验目的，同时硬件自身也存在着一些缺陷。未来为了能够更好地将嵌入式 Linux 系统应用到具体的工作中去，还需要做一些设计改进。并且随着 Linux 内核的不断更新、以及对各种硬件的支持，嵌入式 Linux 系统应用开发将会更加方便快捷。