华清远见 FARSIGHT 始于2004 | 嵌入式培训专家

10年口碑积累，成功培养50000多名研发工程师，铸就专业品牌形象

华清远见的企业理念是不仅要做良心教育、做专业教育，更要做受人尊敬的职业教育。

《ARM 嵌入式系统开发典型模块》

作者：华清远见

专业始于专注 卓识源于远见

第 1 章 基于 ARM 的最小系统模块

专业始于专注 卓识源于远见

1.1 嵌入式系统简介

1.1.1 嵌入式系统的概念

嵌入式系统是不同于常见计算机系统的一种计算机系统，它不以独立设备的物理形态出现，即它没有一个统一的外观，它的部件根据主体设备以及应用的需要嵌入在设备的内部，发挥着运算、处理、存储以及控制的作用。从体系结构上看，嵌入式系统主要由嵌入式处理器、支撑硬件和嵌入式软件组成。其中嵌入式处理器通常是单片机或微控制器；支撑硬件主要包括存储介质、通信部件和显示部件等；嵌入式软件则包括支撑硬件的驱动程序、操作系统、支撑软件以及应用中间件等。

1.1.2 嵌入式系统的结构

下一般说来，嵌入式系统由图 1.1 所示的 3 个部分组成。

嵌入式系统硬件平台	嵌入式操作系统	嵌入式系统应用

图 1.1 嵌入式系统组成示意图

如图 1.1 所示，嵌入式系统一般由 3 个部分组成：嵌入式系统硬件平台、嵌入式操作系统和嵌入式系统应用。其中嵌入式系统硬件平台为各种嵌入式器件、设备（如 ARM、51 单片机等），嵌入式操作系统是指在嵌入式硬件平台上运行的操作系统，目前比较主流的嵌入式操作系统有嵌入式 Linux、μCLinux、μC/OS-II 等。具体应用哪种嵌入式操作系统应视具体情况而定。 嵌入式 Linux 提供了完善的网络技术支持； μCLinux 是专门为没有 MMU 的 ARM 芯片开发的；μC/OS-II 操作系统也成为实时操作系统或 RTOS，使用它作为开发工具将使得实时应用程序变得相对容易（这一部分内容将在 μC/OS-II 操作系统的移植一章中详细讲述）。这 3 种操作系统的特点这里只是简单的介绍，具体参见相关技术手册。

1．嵌入式系统硬件平台

嵌入式系统硬件平台是整个嵌入式操作系统和应用程序运行的硬件平台，不同的应用通常有不同的硬件环境。在嵌入式系统中硬件平台具有多样性的特点。嵌入式系统的核心部件是各种类型的嵌入式处理器，目前据不完全统计，全世界嵌入式处理器的品种总量已经超过 1 000 种，流行体系结构有三十几个系列，数据总线宽度从 8～32 位，处理速度从 0.1～2000MIPS。按功能和内部结构等因素可以分成下面几类。

（1）嵌入式 RISC 微处理器

RISC（Reduced Instruction Set Computer）是精简指令集计算机，RISC 把着眼点放在如何使计算机的结构更加简单和如何使计算机的处理速度更加快速上。RISC 选取了使用频率最高的简单指令，抛弃复杂指令，固定指令长度，减少指令格式和寻址方式，不用或少用微码控制。这些特点使得 RISC 非常适合嵌入式处理器。嵌入式微控制器将整个计算机系统或者一部分集成到一块芯片中。嵌入式微控制器一般以某一种微处理器内核为核心，比如以 MIPS 或 ARM 核为核心，在芯片内部集成 ROM、RAM、内部总线、定时/计数器、WatchDog、I/O 端口、串行口等各种必要的功能和外设。和嵌入式微处理器相比，嵌入式微控制器的最大特点是单片化，实现同样功能时系统的体积大大减小。嵌入式微控制器的品种和数量较多，比较有代表性的通用系列包括 Atmel 公司 AT91 系列、三星公司 S3C 系列，Intel 公司 PXA25x 系列等。

（2）嵌入式 CISC 处理器

嵌入式微处理器的基础是通用计算机中的 CPU 在不同应用中将微处理器装配在专门设计的电路板上，只保留和嵌入式应用有关的功能，这样可以大幅度减小系统体积和功耗。嵌入式微处理器目前主要有 Intel 公司 x86 系列、Motorola 公司 68000 系列等。

2．嵌入式操作系统

嵌入式操作系统完成系统初始化以及嵌入式应用的任务调度和控制等核心功能。具有内核较精简、可配置、与高层应用紧密关联等特点。嵌入式操作系统具有相对不变性。嵌入式操作系统具有以下特点。

（1）体积小

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，而大多使用闪存（Flash Memory）作为存储介质。这就要求嵌入式操作系统只能运行在有限的内存中，不能使用虚拟内存，中断的使用也受到限制。因此，嵌入式操作系统必须结构紧凑，体积微小。

（2）实时性

大多数嵌入式系统都是实时系统，而且多是强实时多任务系统，要求相应的嵌入式操作系统也必须是实时操作系统（RTOS）。实时操作系统作为操作系统的一个重要分支已成为研究的一个热点，主要探讨实时多任务调度算法和可调度性、死锁解除等问题。

（3）特殊的开发调试环境

提供完整的集成开发环境是每一个嵌入式系统开发人员所期待的。一个完整的嵌入式系统的集成开发环境一般需要提供的工具是编译/连接器、内核调试/跟踪器和集成图形界面开发平台。其中的集成图形界面开发平台包括编辑器、调试器、软件仿真器和监视器等。

3．嵌入式系统应用

嵌入式系统应用是以嵌入式系统硬件平台的搭建、嵌入式操作系统的成功移植和运行为前提的，这一部分运行于嵌入式操作系统之上，完成特定的功能或利用操作系统提供的机制完成特定功能的嵌入式应用。不同的系统需要设计不同的嵌入式应用程序。

如何简洁有效地使嵌入式系统能够应用于各种不同的应用环境，是嵌入式系统发展中所必须解决的关键问题。经过不断地发展，嵌入式系统原有的 3 层结构逐步演化成为一种 4 层结构。这个新增加的中间层次叫硬件抽象层，有时也叫板级支持包，是一个介于硬件与软件之间的中间层次。硬件抽象层通过特定的上层接口与操作系统进行交互，向操作系统硬件的直接操作。硬件抽象层的引入大大推动了嵌入式操作系统的通用化。

1.1.3 嵌入式系统的特点

1．嵌入式系统工业的特点和要求

从某种意义上来说，通用计算机行业的技术是垄断的，但嵌入式系统则不同。嵌入式系统工业是不可垄断的高度分散的工业，充满了竞争、机遇与创新，没有哪一个系列的处理器和操作系统能够垄断全部市场，即便在体系结构上存在着主流，但各不相同的应用领域决定了不可能由少数公司、少数产品垄断全部市场。因此嵌入式系统领域的产品和技术，必然是高度分散的，留给各个行业高新技术公司的创新余地很大。另外，社会上的各个应用领域是不断向前发展的，要求其中的嵌入式处理器核心也同步发展，这也构成了推动嵌入式工业发展的强大动力。嵌入式系统工业的基础是以应用为中心的“芯片”设计和面向应用的软件产品开发。

2．嵌入式系统的产品特征

嵌入式系统是面向用户、面向产品、面向应用的，不能独立于应用自行发展，否则便会失去市场。嵌入式系统的核心部件，嵌入式微处理器的功耗、体积、成本、处理能力和电磁兼容性等方面均受到应用要求的制约，这些也是各个半导体厂商之间竞争的热点。嵌入式系统的硬件和软件设计都必须精心考虑，力争在同样的硅片面积上实现更高的性能，只有这样，才能在具体应用时对处理器的选择更具有竞争力。嵌入式处理器要针对用户的具体需求，对芯片配置进行裁剪和添加才能达到理想的性能。由于嵌入式系统和具体应用有机地结合在一起，具有较长的生命周期。

3．嵌入式处理器软件的特征

嵌入式处理器的应用软件是实现嵌入式系统功能的关键，对嵌入式处理器系统软件和应用软件的要求也和通用计算机有所不同，主要有以下几点。

（1）软件要求固态化存储

为了提高执行速度和系统可靠性，嵌入式系统中的软件一般都固化在存储器芯片或嵌入式微控制器本身中，而不是存贮于磁盘等载体中。在基于 ARM 的嵌入式系统开发当中，调试成功的应用程序通常被烧写到在 Flash 中，并在系统上电复位后首先执行 Flash 中的程序，完成系统的各项初始化工作。

（2）软件代码要求高质量、高可靠性

尽管半导体技术的发展使处理器速度不断提高、片上存储器容量不断增加，但在大多数应用中，存储空间仍然是宝贵的，还存在实时性的要求。为此要求程序编写和编译工具的质量要高，以减小程序代码长度、提高执行速度。

（3）要求系统软件具有较高的实时性

在多任务嵌入式系统中，对重要性各不相同的任务进行统筹兼顾的合理调度是保证每个任务及时执行的关键，单纯通过提高处理器速度是无法完成和没有效率的，这种任务调度只能由优化编写的系统软件来完成，因此系统软件的高实时性是基本要求。

（4）嵌入式系统开发需要开发工具和环境

嵌入式系统本身不具备开发能力，即使设计完成以后用户通常也是不能对其中的程序功能进行修改的，必须有一套开发工具和环境才能进行开发，这些工具和环境一般是基于通用计算机上的软硬件设备以及逻辑分析仪、示波器等。

（5）嵌入式系统软件需要实时多任务操作系统开发平台

通用计算机具有完善的操作系统和应用程序接口，是计算机基本组成不可缺少的部分，应用程序的开发以及完成后的软件都在系统软件平台上面运行，但一般不是实时的。嵌入式系统则不同，应用程序可以没有操作系统直接在芯片上运行；但是为了合理地调度多任务、利用系统资源，用户必须自行选配实时多任务系统开发平台，这样才能保证程序执行的实时性、可靠性，并减少开发时间，保障软件质量。

（6）采用 C 语言是嵌入式系统开发的是最佳和最终选择

由于汇编语言是一种非结构化的语言，对于大型的结构化程序设计已经不能完全胜任了。这就要求我们采用更高级的 C 语言去完成这一工作。

1.1.4 嵌入式系统的发展趋势

（1）提供强大的网络服务

为适应嵌入式分布处理结构和应用上网需求，面向 21 世纪的嵌入式系统要求配备标准的一种或多种网络通信接口。针对外部联网要求，嵌入设备必需配有通信接口，相应需要 TCP/IP 协议簇软件支持；由于家用电器相互关联（如防盗报警、灯光能源控制、影视设备和信息终端交换信息）及实验现场仪器的协调工作等要求，新一代嵌入式设备还需具备 IEEE1394、USB、CAN、Bluetooth 或 IrDA 通信接口，同时也

需要提供相应的通信组网协议软件和物理层驱动软件。为了支持应用软件的特定编程模式，如 Web 或无线 Web 编程模式，还需要相应的浏览器，如 HTML、WML 等。

（2）小型化、低成本、低功耗

为满足这种特性，要求嵌入式产品设计者相应降低处理器的性能，限制内存容量和复用接口芯片。这就相应提高了对嵌入式软件设计技术要求。如，选用最佳的编程模型和不断改进算法，采用 Java 编程模式，优化编译器性能。因此，既要软件人员有丰富经验，更需要发展先进嵌入式软件技术，如 Java、Web 和 WAP 等。

（3）人性化的人机界面

嵌入式设备之所以为亿万用户乐于接受，重要因素之一是它们与使用者之间的亲和力，自然的人机交互界面，如司机操纵高度自动化的汽车主要还是通过习惯的方向盘、脚踏板和操纵杆。人们与信息终端交互要求以 GUI 屏幕为中心的多媒体界面。手写文字输入、语音拨号上网、收发电子邮件以及彩色图形、图像已取得初步成效。目前一些先进的 PDA 在显示屏幕上已实现汉字写入、短消息语音发布，但离掌式语言同声翻译还有很大距离。

（4）完善的开发平台

随着因特网技术的成熟、带宽的提高，ICP 和 ASP 在网上提供的信息内容日趋丰富、应用项目多种多样，像电话手机、电话座机及电冰箱、微波炉等嵌入式电子设备的功能不再单一，电气结构也更为复杂。为了满足应用功能的升级，设计师们一方面采用更强大的嵌入式处理器如 32 位、64 位 RISC 芯片或信号处理器 DSP 增强处理能力；同时还采用实时多任务编程技术和交叉开发工具技术来控制功能复杂性，简化应用程序设计、保障软件质量和缩短开发周期。

1.2 最小系统结构及框图

1．什么是最小系统

嵌入式芯片自己是不能独立工作的，需要必要的外围芯片给它提供基本的工作条件。一个嵌入式芯片必须有供电系统为其供电；必须有时钟系统提供时钟信号；必须有复位系统。这些“侍服”系统有些是很简单廉价的芯片。嵌入式芯片还需要有存储系统。如果芯片内部没有存储器或存储器容量不足以满足需求，则需要外扩存储芯片。调试接口也是嵌入式系统不可缺少的一部分，这些嵌入式处理器运行的必要条件的电路或芯片与嵌入式处理器一起构成了嵌入式处理器的最小系统。

2．最小系统结构框图

最小系统结构框图如图 1.2 所示。

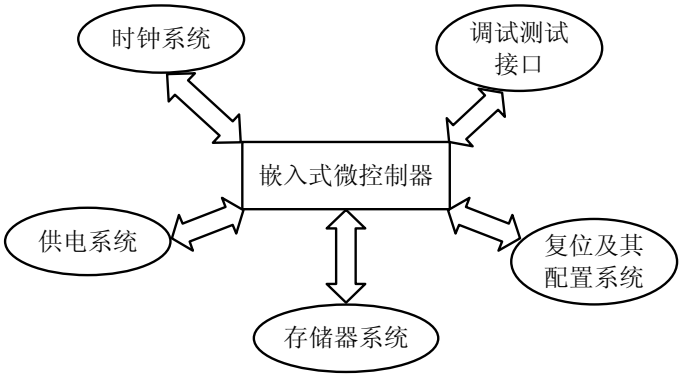

图 1.2 最小系统结构框图

1.3 最小系统的电源设计

电源系统的地位十分重要，但往往设计者对电源系统的重视程度不够。一个合理、稳定的电源系统可以大大减少系统故障的发生率。设计电源时应考虑以下因素。

- 电源系统输出的电压、电流、功率等因素。
- 电源系统输入的电压、电流。
- 电源的稳定性因素。
- 电源系统的输出波纹。
- 电源系统的兼容性。
- 电磁干扰因素。
- 电源系统的体积限制。
- 电源系统功耗限制。
- 电源系统的成本因素。

可以看出，设计一个好的电源系统需要考虑很多因素。ARM CPU 通常需要 2 种或 2 种以上的电源。一种是核电压 V_{CORE}，另一种是 I/O 电压（V_{DD}）。另外，不同厂商生产的 ARM 芯片对电源的要求也不尽相同。下面就以三星公司生产的一款 ARM7 芯片 S3C44B0X 和飞利浦公司生产的 ARM7 内核的 LPC2000 系列芯片的电源系统的设计为例说明电源系统的设计方法。

S3C44B0XARM 芯片的电源设计如下。

S3C44B0X 的 V_{DD} 的值为 3.3V，V_{CORE} 的值为 2.5V。系统的输入电压大多数时候不是 3.3V，这就要求实现电源的变换。考虑到电源效率和稳定性等因素的要求，采用 DC/DC 实现高电压到 3.3V 的转换。当系统功能不是很复杂时，比如系统不需要实现 AD 转换功能时，S3C44B0X 可采用 MICROCHIP 公司 TC120333 的 DC/DC 应用电路。TC120333 的应用电路如图 1.3 所示。

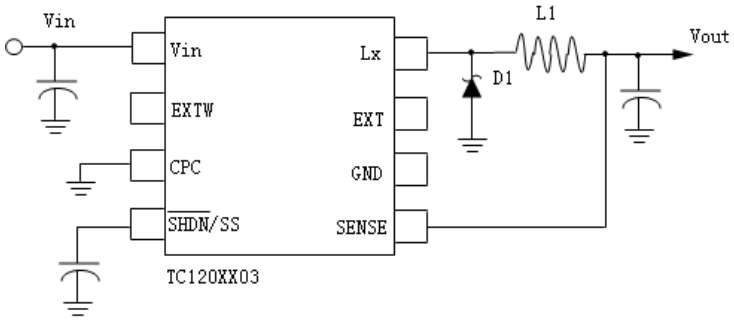

图 1.3　TC120333 应用电路

核电压消耗的电流相对较小，如果考虑到系统开发成本和体积的因素，可以考虑采用 LDO 型电源。图 1.4 所示的 LD1117 是一款常用的 LDO，能将 3.3V 电压转化成为 2.5V 的核电压。也有的设计者用简单的二极管实现 3.3V 电压到 2.5V 电压的转换，但考虑到电源电路对于系统的稳定运行起着至关重要的作用，笔者还是建议采用 LDO 或其他电源芯片，以提高系统的稳定性。

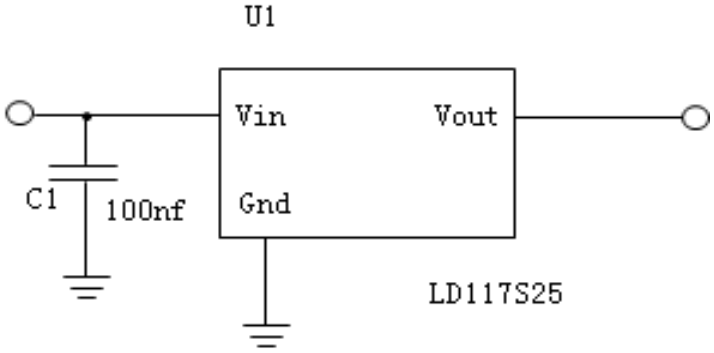

图 1.4　LDO 核电压应用电路

电源芯片的选择通常情况下要考虑系统连接外围电路的复杂度和功耗等因素，一个较为复杂的系统对电源质量的要求也较高。在实际应用当中，ARM 处理器通常要连接各种部件实现特定的功能，如在音频设备中连接视频电路芯片，实现视频信号的压缩、转换、解码等操作，连接数字信号编码解码芯片完成声音信号处理等。随着 ARM 技术的不断推广和成熟，ARM 的应用领域也不断拓宽。在特殊的应用领域里，如对数字信号处理的能力要求很高的信号处理领域，ARM 芯片可以和 DSP 芯片“强强联合”。关于 ARM+DSP 技术的详细内容将在本书第 14 章 DSP 芯片扩展一章中详细介绍。有一些嵌入式应用领域，要求多个功能相同的嵌入式设备协同工作，比如完成一座跨江大桥桥身负重数据采集的数据采集系统，可能会需要在桥身的不同地点安装多个数据采集设备，这就要求各个分布的设备系统工作时钟具有一致性，这就需要在 ARM 芯片上连接 GPS 接收芯片，完成系统的时间同步功能。在如上所述的相对复杂的电路里，对电源芯片的要求也相对较高。关于 ARM 芯片与 GPS 芯片连接的技术在本书中也有介绍。下面就介绍一款在实现视频、音频处理功能的 ARM 嵌入式系统中应用的电源芯片 R1224。R1224 具有电路简单，转换效率高，输出波纹小，电源质量好等特点。R1224 的电流流经 MOS 管，而不是芯片本身，所以对整个系统的干扰较小。3.3V 的电源电路图如图 1.5 所示。

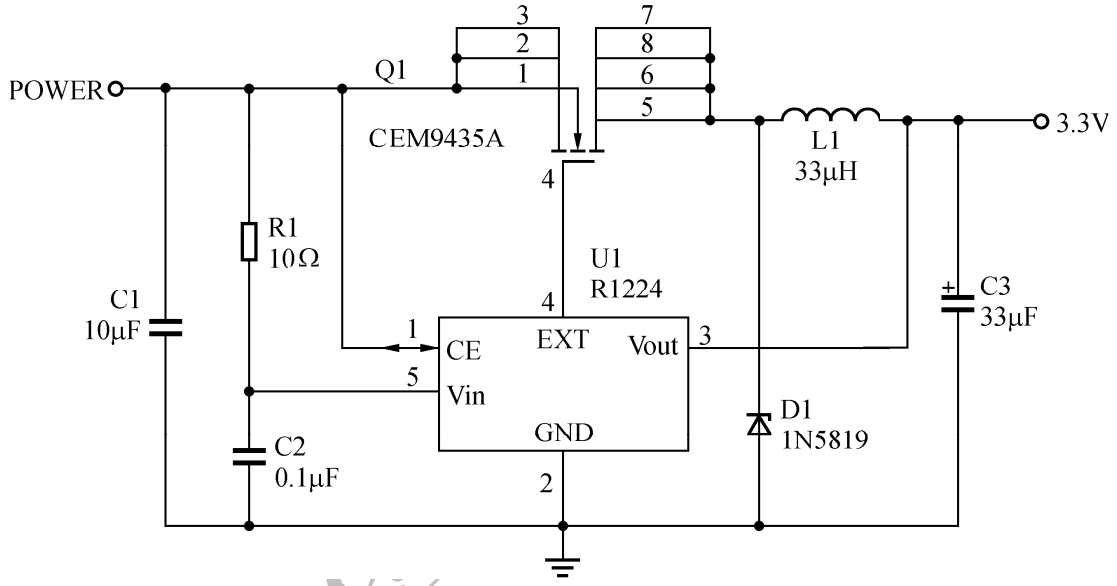

图 1.5　R1224 电源电路

如果系统中需要安装投影仪，那么对电源的输出质量要求也会相应提高，应尽量减小波纹，提高抗干扰能力，同时可以考虑在 AC-DC 输入处加保险丝和采取适当的滤波措施，如图 1.6 所示的电路就可以实现这样的功能。

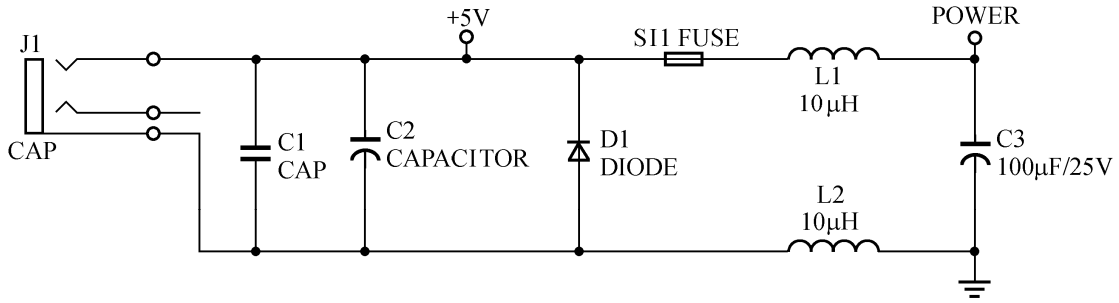

图 1.6　滤波功能电源电路

C3C44B0X 的核心工作电压是 2.5V，多功能 I/O 口及部分外设的工作电压是 3.3V。在一些系统中，部分外设的工作电压为 5V。所以在嵌入式系统的设计中，常常需要设计多种电源电路。嵌入式系统的一个重要的应用领域是便携式设备，在便携式设备中需要用到电池提供电能，一般情况下采用 3V 电池供电，这就需要将 3V 电压进行变换。这里在介绍 2 款实现电压转换的芯片，他们分别是 MAXIM 公司的 MAX1703 和 MAX860，TI 公司的 TPS73HD325。

MAX1703 是一款高功率、低噪声的升压 DC-DC 变换器，输入电压范围为 0.7～5.5V，输出电压范围有 2 种可选方案，分别是固定 5V 和 2.5～5.5V。MAX1703 的最大输出电流是 1.5A。MAX1703 具有如下特点：高达 95%的转换率，只有 300μW 的低功耗，较低的输入电压范围，高达 300kHz 的开关频率，具有欠压检测功能。下面的电路设计将实现 3V 电压到固定 5V 电压的转换，MAX1703 的引脚俯视图如图 1.7 所示。

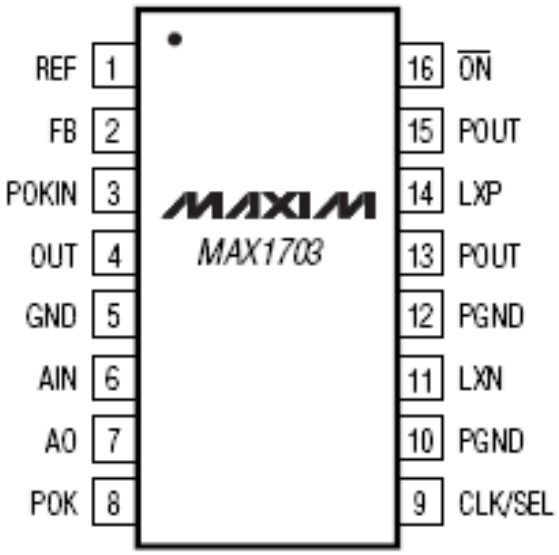

图 1.7　MAX1703 引脚俯视图

MAX1703 连接适当的外围电路实现从 3V 的电池供电电源转换为固定 5V 的供电电源的电路连接如图 1.8 所示。

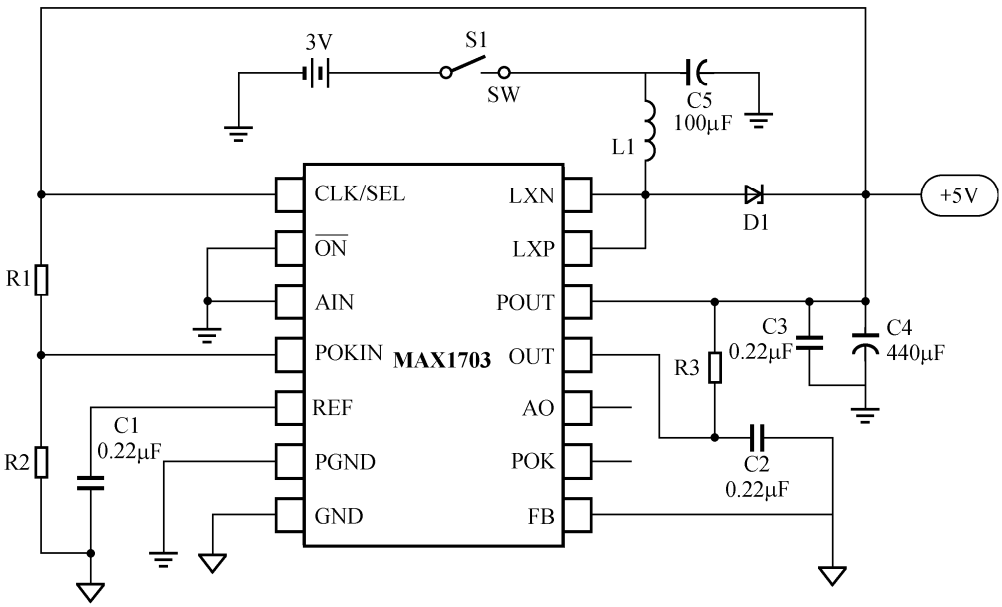

图 1.8　3～5V 变换电路

基于三星 S3C44B0X 的系统设计常见电源设计方案大致介绍以上几种。电源设计是系统稳定运行的基础，要充分考虑各种因素的影响。电源电路的复杂程度和低功耗常常是一对相互矛盾的因素，如何协调好二者之间的关系，找到最佳的契合点，是一名嵌入式系统设计者需要做好的工作之一。嵌入式系统通常使用在便携式设备中，具有移动的特点，这对电源系统更是提出了较高的要求，即在保证系统正常运行和实现既定功能的前提下减小系统功耗，延长电源系统不间断供电时间。下面介绍一些常用的芯片和设计方法供嵌入式设计人员和嵌入式技术爱好者参考。与三星一样，飞利浦也是较大的 ARM 生产商之一，下面介绍飞利浦的 LPC2000 系列微控制器的电源系统设计。

- LPC2000 系列 ARM 的电源设计。

鉴于现有的 ARM 技术教材和书籍中大多以三星公司 ARM 芯片作为示例芯片进行讲解，提供的技术资料也相对较多，而飞利浦生产的 ARM 芯片资料相对较少，这里对飞利浦生产的 LPC2000 系列 ARM 芯片作一介绍。

LPC2000 系列微控制器基于 ARM7TDMI-S CPU 内核，支持 ARM 和 Thumb 指令集，芯片内集成丰富外设，而且具有非常低的功率消耗，使该系列微控制器特别适用于工业控制、医疗系统、访问控制和 POS 机等场合。LPC2000 系列微控制器对 LCD 显示提供了完备的技术支持，是开发 LED 设备的理想选择。LPC2000 系列器件信息表如表 1.1 所示。

表 1.1　　LPC2000 系列器件信息表

器件型号	引脚数	片内 RAM	片内 Flash	10 位 AD 通道数	CAN 控制器	备注
LPC2114	64	16KB	128KB	4	—	
LPC2124	64	16KB	256KB	4	—	
LPC2210	144	16KB	—	8	—	带外部存储器接口
LPC2212	144	16KB	128KB	8	—	
LPC2214	144	16KB	256KB	8	—	
LPC2119	64	16KB	128KB	4	2	
LPC2129	64	16KB	256KB	4	2	
LPC2194	64	16KB	256KB	4	4	
LPC2290	144	16KB	—	8	2	带外部储器接口
LPC2292	144	16KB	256KB	8	2	
LPC2294	144	16KB	256KB	8	4	
LPC2131	64	8KB	32KB	8	—	
LPC2132	64	16KB	64KB	8	—	带 1 路 DAC
LPC2134	64	16KB	128KB	双 8 路	—	
LPC2136	64	16KB	256KB	双 8 路	—	
LPC2138	64	32KB	512KB	双 8 路	—	

由表 1.1 可以看出， LPC2000 系列芯片较显著的区别在于片内是否有 Flash 存储器及 CAN 控制器。LPC2210、LPC2290 芯片内部没有 Flash，需要外接 Flash 存储芯片，此 2 种芯片都有外部存储器接口，为芯片的存储器扩展做好了准备。关于片外扩展 Flash 的接线及编程技术将在本书第 2 章中作较为详细的介绍。

LPC2000 系列微控制器就功能而言包含 4 大部分。

① ARM7TDMI-S CPU。

② ARM7 局部总线及相关部件。

③ AHB 高性能总线及相关部件。

④ VLCI 外设总线及相关部件。

LPC2000 系列微控制器将 ARM7TDMI-S 配置为小端模式（Little-endian）。AHB 外设分配了 2MB 的地址范围，它位于 4GB ARM 寻址空间的最顶端。每个 AHB 外设都分配了 16KB 的地址空间。LPC2000 系列微控制器的外设功能（除中断控制器）都连接到 VPB 总线。AHB 到 VPB 的桥将 VPB 总线与 AHB 总线相连。VPB 外设也分配了 2MB 的地址范围，从 3.5GB 地址点开始。每个 VPB 外设都分配了 16KB 的地址空间。LPC2000 系列微控制器的功能结构框图如图 1.9 所示。

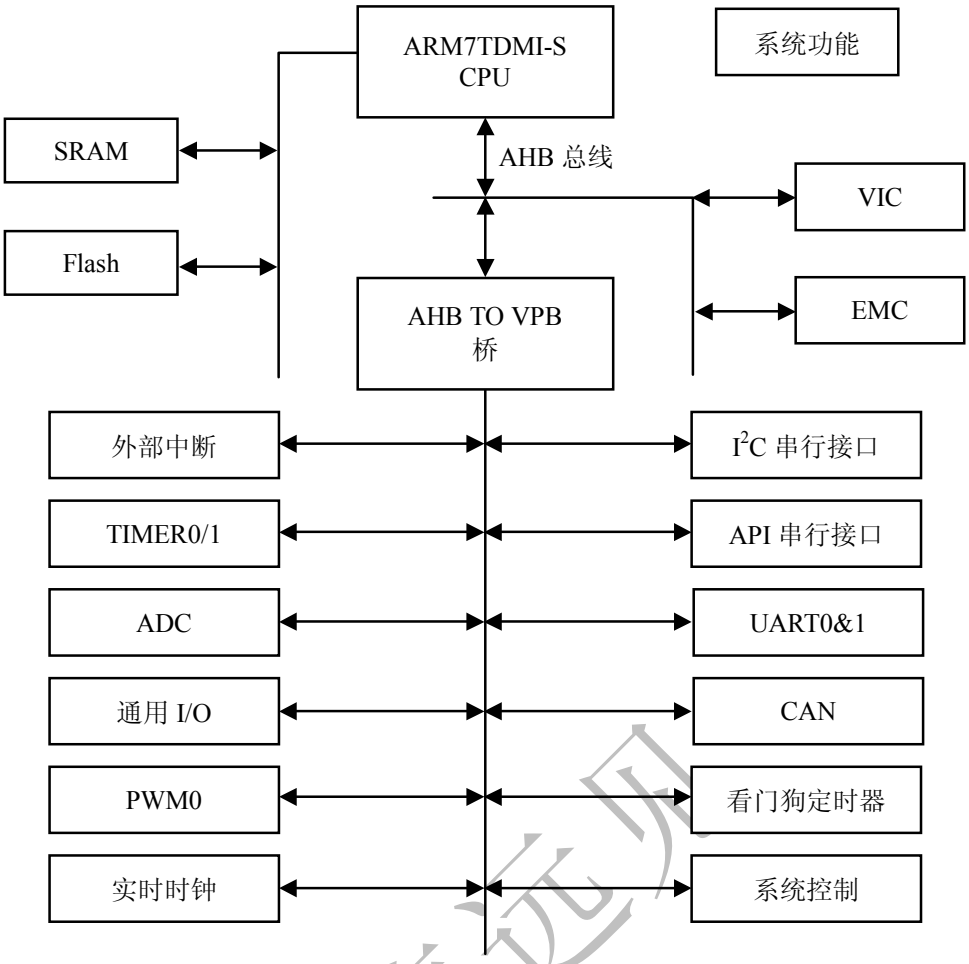

图 1.9　LPC2000 系列微控制器功能结构框图

以上简要介绍了 LPC2000 系列微控制器，详细技术资料请读者自行查阅相关技术书籍和飞利浦网站关于此系列芯片的介绍，在此不再详细叙述。

- LPC2000 系列微控制器电源设计。

LPC2000 系列微控制器提供 A/D 功能，是否使用 A/D 功能和对 A/D 功能要求的高低程度都会影响到电源系统的设计。若未使用 A/D 功能或对该功能要求较低，则模拟电源和数字电源不必分开供电，反之则模拟电源和数字电源需要分开供电。对于 LPC2200 系列微控制器有 4 组电源输入，理想情况下需要提供 4 组独立电源供电。如前所述，若没有使用 A/D 功能或对 A/D 功能要求不高，则模拟和数字电源可以不必分开供电，这样，LPC210X 和 LPC2200 都只需要 2 组电源。

LPC2000 系列微控制器需要 2 种类型的电源，分别是 3.3V 和 1.8V，多数外接器件需要 5V 电源供电，这 3 个电压量为系统前级和后级电源的设计提供了重要的参数。LPC2000 系列 1.8V 级电压消耗电流的极限值是 70mA。考虑到系统的稳定性因素和为今后的系统扩展留下余量，LPC2000 系列电源系统 1.8V 能够提供的最大电流应大于 300mA。LPC2000 系列 3.3V 及电压消耗的电流与系统外围器件的选择有很大的关系，假设所选择的外围器件消耗大的最大电流为 300mA，这样 3.3V 电源能够提供的最大电流设计为 600mA 即可。后级电源自然是 3.3V 和 1.8V，关键是前级电源电压和供电芯片类型的选择。系统对电源质量的要求较高，而其功率消耗不是很大，所以不宜选择开关电源，而选择低压差模拟电源。

LPC2000 系列微控制器所需电源类型如表 1.2 所示。

表 1.2　　LPC2000 系列微控制器所需电源类型表

	3.3V 类型	1.8V 类型
LPC210X	V3.3	V1.8
LPC22XX	V3.3D、V3.3A	V1.8D、V1.8A
LPC213X	V3.3D、V3.3A	无

明确了器件选择的要求之后下面来选择合适的电源器件。如上所述，应选择模拟电源器件（LDO）。合乎要求的器件很多，Sipex 半导体 SPX1117 就是一个很好的选择。Sipx1117 的性价比较高，具有很好的可扩展性，可与许多产品直接连接。SPX1117 为一个低功耗正向电压调节器，可以用在一些高效率、小封装的低功耗设计中。这款器件非常适合便携式电脑及电池供电的应用。SPX1117 有很低的静态电流，在满负载时其低压差仅为 1.1V。当输出电流减少时，静态电流随负载变化，并提高效率。SPX1117 可调节，以选择 1.5V，1.8V，2.5V，2.85V，3.0V，3.3V 及 5V 的输出电压。SPX1117 提供多种 3 引脚封装：SOT-223，TO-252，TO-220 及 TO-263。一个 10μF 的输出电容可有效地保证稳定性，然而在大多数应用中，仅需一个更小的 2.2μF 电容。

SPX1117 具有以下特点。

- 0.8A 稳定输出电流。
- 峰值电流 1A。
- 3V 电压可调节。
- 低静态电流。
- 0.8A 时低压差为 1.1V。
- 0.1%线性调整率。
- 0.2%负载调整率。
- 过流及温度保护。
- 多种封装选择。

SPX1117 的模拟电源如图 1.10 所示。

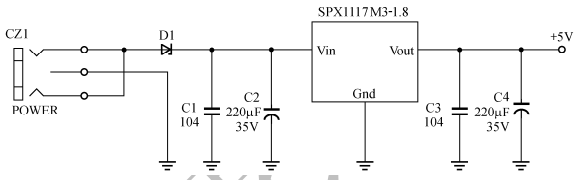

图 1.10　SPX1117 模拟电源

SPX1117 的前级输入电压最高可达 20V，但在嵌入式设备的设计当中，发热量也是一个不容忽视的问题。设备过高的发热量常常使得系统出现这样或那样的故障，无法正常工作，同时也给散热系统的设计带来诸多困难，这时需要将系统的前级电路做适当调整。在许多情况下系统可能使用多种电源，如交流电和电池等，这时也需要调整前级输入以适应末级输入。综上，电源系统的前级输入选择为 5V。5V 电压满足 SPX1117 的要求，加之目前许多设备的工作电压为 5V，所以选择 5V 可以兼顾前极和末级电压要求。

至此，最小系统的电源设计的介绍就告一段落，在后面的章节中，电源设计仍将作为嵌入式系统设计的内容之一进行介绍。

1.4　最小系统的时钟系统设计

在介绍时钟晶振电路之前，先来简单了解一下有源晶振和无源晶振的概念。在电子学上，通常将含有晶体管元件的电路称作“有源电路”（如有源音箱、有源滤波器等），而仅由阻容元件组成的电路称作“无源电路”。电脑中的晶体振荡器也分为无源晶振和有源晶振 2 种类型。无源晶振与有源晶振的英文名称不同，无源晶振为 crystal（晶体），而有源晶振则叫做 oscillator（振荡器）。无源晶振是有 2 个引脚的无极性元件，需要借助于时钟电路才能产生振荡信号，自身无法振荡起来，所以“无源晶振”这个说法并不准确；有源晶振有 4 只引脚，是一个完整的振荡器，其中除了石英晶体外，还有晶体管和阻容元件，因此体积较大。石英晶体振荡器的频率稳定度很高，例如 10MHz 的振荡器，频率在一日之内的变化一般不大于 0.1Hz。因此，完全可以将晶体振荡器视为恒定的基准频率源（石英表、电子表中都利用石英晶体来做计时的基准频率）。从 PC 诞生至现在，主板上一直都使用一颗 14.318MHz 的石英晶体振荡器作为基准频率源。至于始终沿用 14.318MHz 这个频率的原因，或许是保持兼容性的需要吧。但是，笔者在显卡、闪存盘和手机中也发现了 14.318MHz 的晶振，就不知道是什么原因了。

石英晶体振荡器是利用石英晶体（二氧化硅的结晶体）的压电效应制成的一种谐振器件，它的基本构成大致是：从一块石英晶体上按一定方位角切下薄片（简称为晶片，它可以是正方形、矩形或圆形等），在它的 2 个对应面上涂敷银层作为电极，在每个电极上各焊一根引线接到管脚上，再加上封装外壳就构成了石英晶体谐振器，简称为石英晶体或晶体、晶振。其产品一般用金属外壳封装，也有用玻璃壳、陶瓷或塑料封装的。

晶振电路用于向 CPU 及其他工作电路提供工作时钟。绝大多数的 CPU 可使用有源晶振和无源晶振。下面以三星公司另一款 ARM7TDMI 芯片 S3C4510B 为例来说明晶振电路的设计。

连接无源晶振是利用 CPU 内部的时钟振荡电路，在 CPU 的引脚 X1 和 X2 之间接一枚晶体就可以产生稳定的时钟信号。为了尽量避免高频辐射的干扰，时钟电路的走线应尽可能短。无源晶振的连接方法如图 1.11 所示。

系统采用有源晶振的情况下，电路连接方法与无源晶振时的接法有所不同。常用的有源晶振的接法如图 1.12 所示。

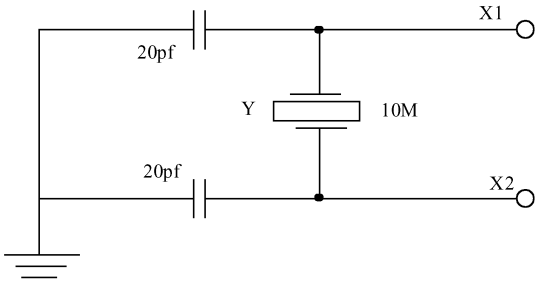

图 1.11　无源晶振连接示意图

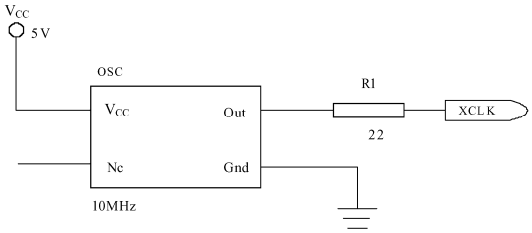

图 1.12　有源晶振连接示意图

从图 1.12 中可以看出，晶振频率为 10MHz。系统要想获得更高的工作频率，需要借助 PLL 电路，经过 PLL 电路倍频，系统最高可获得 50MHz 的工作频率。系统以较低的外部时钟获得较高的工作频率的优点在于大大降低了因告诉开关时钟而造成的高频噪音。下面结合图 1.13 介绍 PLL 电路的工作原理，以便读者清楚频率的变化关系，正确设计电路参数。

PLL 的主要原理是一种类似运算放大器般的负反馈电子电路结构，如图 1.13 所示，PLL 主要有 2 个输入端，分别是参考输入频率（Fref）和回馈输入频率（Fvco），共同连接到 PLL 内部的第一个组件——相位/频率检测器（Phase Frequency Detector，PFD）。相位/频率检测器会比较参考频率与回馈频率两者间的差别，检测出两者间的相位与频率的差异量，当参考频率高于回馈频率时，PFD Up 端会输出 Up 脉波；反之若是参考频率低于回馈频率时，PFD Dn 端会输出 Dn 脉波。相位/频率检测器产生的脉波信号随后经由电流控制器（Charge Pump）与环路滤波器（Loop Filter），转换成为最后一阶压控振荡器（Voltage Controled Oscillator，VCO）的控制电压，产生 Fvco 时脉讯号的输出。此时若是输出的时脉讯号直接连接的负回授频率输入端，就形成了所谓的“相位锁定回路”，输出端所送出的回馈输入频率（Fvco）的时脉讯号将会被用来锁定参考输入频率（Fref），永远与参考频率同步保持一致的相位与频率状态。当回馈输入频率（Fvco）与参考输入频率（Fref）的频率与相位一致时，也就是整个相位回路已经锁定了（Locked）。时脉产生器借着 PLL 的相位锁定特性，于 PLL 的 2 个输入端与输出端，若将参考输入频率（Fref）与反馈输入频率（Fvco）之后分别接上除频电路，设除频电路的除频系数为 P、Q、R。当 PLL 处于稳定锁定的状态时，PFD 的 2 个输入端频率与相位应为相等，故

$$Fref / Q = Fvco / P$$

所以

$$Fvco = Fref * P / Q$$

因为实际的输出端还有一个除频电路 R，所以时脉产生器的输出频率就会变成

$$Fou\ t= (Fref*P)/ (Q * R)。$$

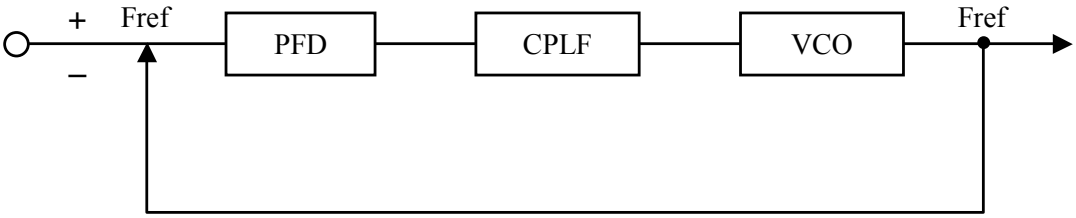

图 1.13　PLL 电路原理图

几乎所有的 ARM 芯片都提供时钟发生电路，需要开发者完成工作只是外接晶体，从而使用 ARM 芯片内部的振荡电路产生时钟，由 PLL 倍频产生所需的工作频率。在一些特殊的场合，如迫切需要减少功耗和要求系统工作时间严格同步等，可以考虑外接时钟信号，连接方法是外部时钟芯号连接 Xin 引脚，Xout 引脚悬空。

下面介绍一款常用的串行时钟芯片 HT1380 及其基本变程原理。HT1380 的管脚及功能如图 1.14 所示。

符号	管脚号	描述
NC	1	空脚
X1	2	振荡器输入
X2	3	振荡器输出
V_{SS}	4	地
/RST	5	复位引脚
I/O	6	数据输入/输出引脚
SCLK	7	串行时钟
V_{DD}	8	正电源

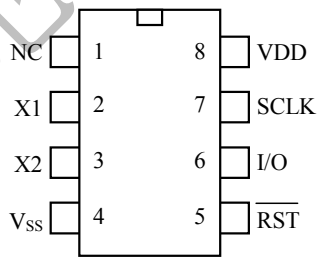

图 1.14　HT1380 管脚及功能图

HT1380 是 HOLTEK 公司推出的一款带秒分时日星期月年的串行时钟保持芯片，每个月多少天以及闰年能自动调节。HT1380 具有低功耗工作方式，并用若干寄存器存储对应信息。一个 32.768kHz 的晶振校准时钟，为了使用最小引脚，HT1380 使用一个 I/O 口与微信息处理机相连，仅使用 3 根引线，RST、SCLK 串行时钟、I/O 口，数据就可以传送 1 字节或 8 字节的字符组。

HT1380 与嵌入式微控制器的连接方法如图 1.15 所示。

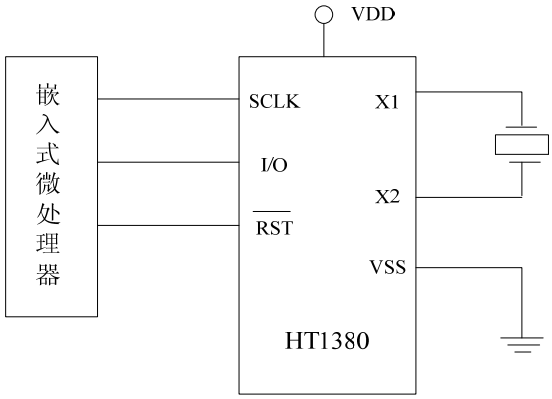

图 1.15　HT1380 与嵌入式微处理器连接图

下面给出 HT1380 的单字节、多字节传送的程序流程图。

多字节传送的程序流程图如图 1.16 所示。

单字节传送的程序流程图如图 1.17 所示，与多字节传送不同的是，单字节传送是对寄存器逐个地操作，但 2 种操作都是从寄存器的 0 位开始操作。

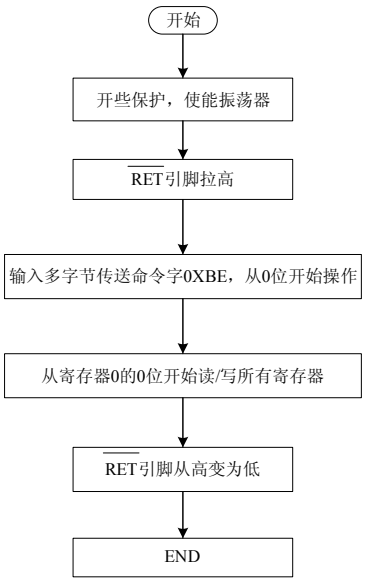

图 1.16　多字节传送程序流程图

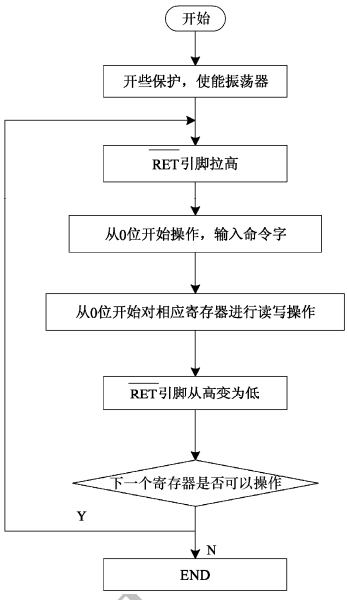

图 1.17　单字节传送程序流程图

1.5　最小系统的复位系统设计

复位电路的功能是完成系统的上电复位和系统运行时的按键复位功能。复位电路的选择可以是简单的 RC 电路，也可以是相对复杂的电路。复位系统的质量也关系到系统是否可以稳定地运行，所以建议采用相对复杂但功能更加完善的复位电路，如复位芯片。系统上电时，状态是不可知的，复位逻辑系统的功能就是负责将控制器初始化为某个确定的状态。这个复位逻辑系统需要一个复位信号才能正常工作，一些微控制器可以自己产生这个复位信号，但大多数的微控制器需要从外部引入这个信号。复位信号的可靠性和稳定性对微控制器是否可以正常工作有着重大的影响。

下面介绍一种常见的 RC 复位电路。RC 复位电路的基本功能是系统上电时提供复位信号直至系统电源稳定后撤销复位信号。为可靠起见，电源稳定后还要经一定的延时才撤销复位信号，以防电源开关或电源插头分-合过程中引起的抖动而影响复位。经使用证明，这种简单的 RC 复位电路的复位逻辑是可靠的，其电路图如图 1.18 所示。

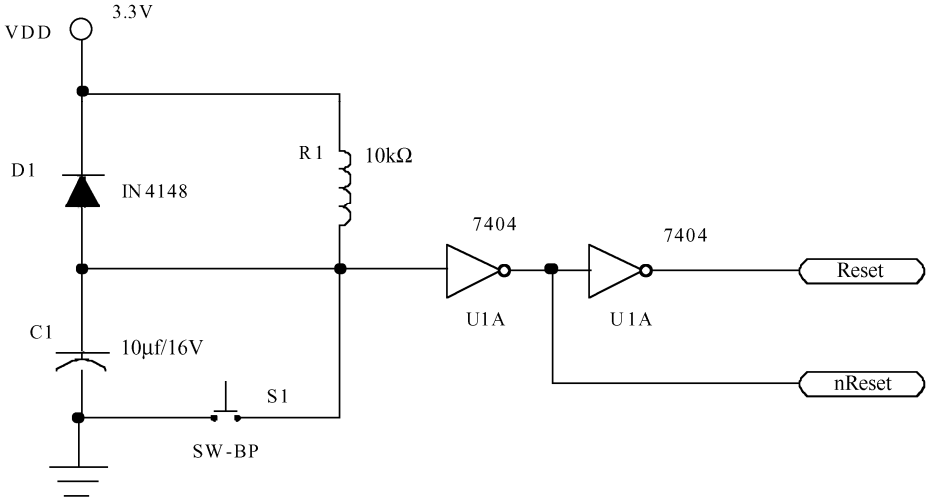

图 1.18　RC 复位电路

该 RC 复位电路的基本原理如下。系统上电时，通过电阻 R1 向电容 C1 充电，C1 电压逐渐上升。在 C1 电压达到高电平的门限电压时，Reset 输出为低电平，系统处于复位状态；C1 电压继续上升，当 C1 电压达到高电平的门限电压时，Reset 输出变为高电平，系统进入正常工作状态。当人为按下开关 S1 时，C1 两端电荷被释放掉，Reset 输出变为低电平，系统又进入复位状态，当 C1 再次被充电时，系统又进入正常工作状态。

2 个反相器的作用是实现按钮去抖和波形调整，从图中可以看出，nReset 的输出和 Reset 的输出相反，可用于高电平的复位器件。适当调整 R、C 的值，可得到需要的阻容参数，调整复位时间。

下面 2 种 RC 复位电路的原理与图 1.18 介绍的 RC 复位电路原理相似，不再详细介绍，电路原理图如图 1.19 所示。

为了增加复位电路的可靠性，在 RC 复位电路中增加了二极管，其作用是在电源电压下降时使电容迅速放电，即使电源带有毛刺，也可使系统可靠复位，带有二极管的电路原理图如图 1.20 所示。

复位电路亦可通过复位芯片来实现，与简单的 RC 复位电路相比，复位芯片电路更加复杂，但也更加稳定和可靠，加之多数复位芯片的价钱不高，所以复位芯片是复位系统的较理想的选择。

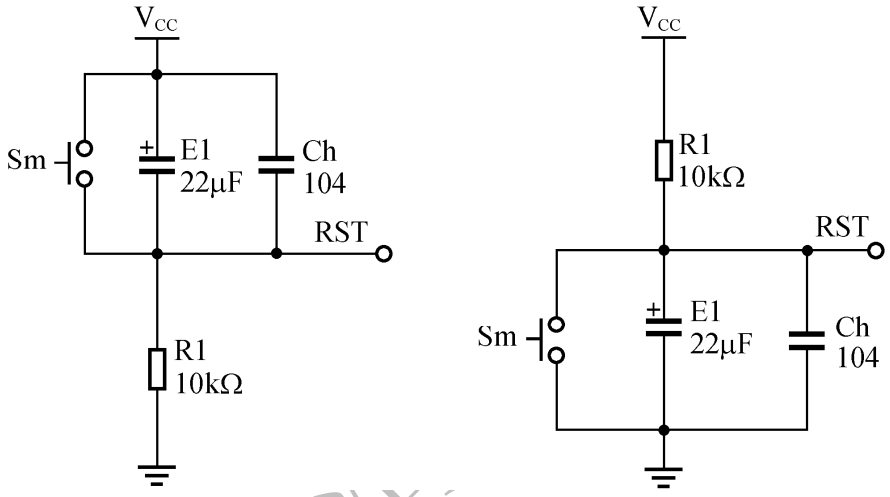

图 1.19　RC 复位电路图

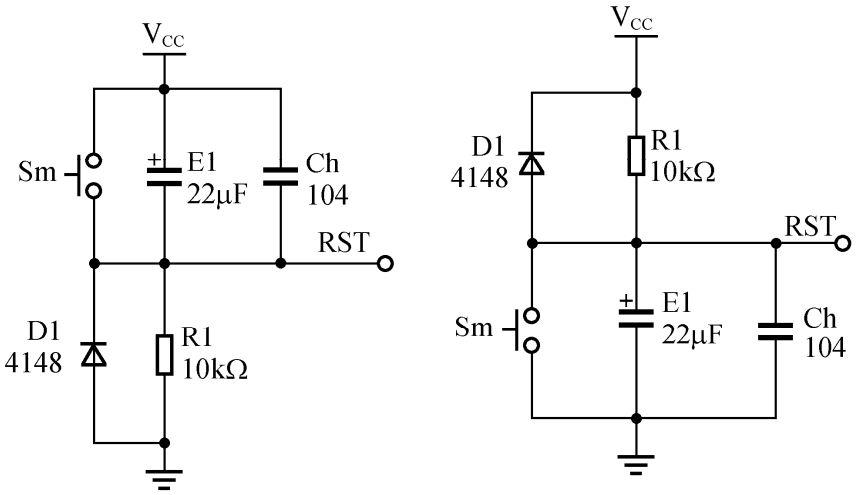

图 1.20　带有二极管的 RC 复位电路

MAX8xx 是一种系列的复位芯片，该芯片有多种型号，可根据系统实际需要任意挑选。下面介绍该系列复位芯片的一种——MAX810。MAX810 是一种单一功能的微处理器复位芯片，用于监控微控制器和其他逻辑系统的电源电压。它可以在上电掉电和节电情况下向微控制器提供复位信号。当电源电压低于预设的门槛电压时器件会发出复位信号，直到在一段时间内电源电压又恢复到高于门槛电压为止。MAX810 有高电平有效的复位输出，典型值是 17μA 的低电源电流使 MAX810 能理想地用于便携式电池供电的设备。它们使用 3 管脚的 SOT23 封装。

MAX810 具有以下特征。

- 监控 5.0V、3.3V、3V 电源。
- 复位延长时间最短为 140ms。
- 复位输出为高电平有效。
- 电压低至 1.1V 时仍可产生复位信号。
- 小型三管脚 SOT-23 封装。
- 无需外部配件。
- 适用温度范围为−40～+150℃。

MAX810 的适用范围如下。

- 嵌入式微控制器。
- 电池供电系统。
- PDA 和各种手持设备。

MAX810 的管脚配置如图 1.21 所示。

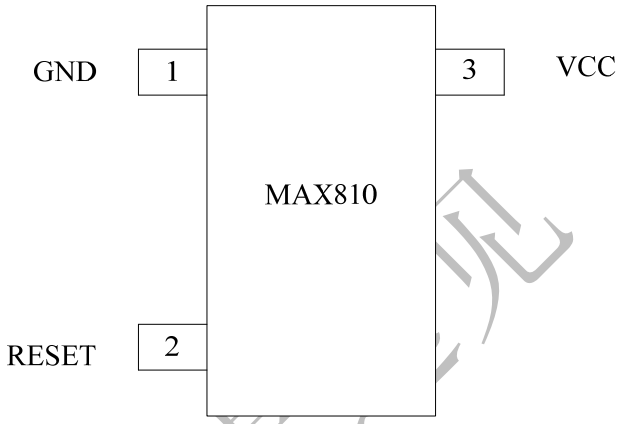

图 1.21　MAX810 管脚配置图

MAX810 的管脚说明如表 1.3 所示。

表 1.3　　MAX810 管脚配置表

管脚	符号	说明
1	GND	器件地
2	RESET	复位信号高电平有效
3	VCC	电源输入电压

MAX810 与微控制器的连接也很简单，VCC 外接电压，GND 管脚接地，RESET 管脚接入嵌入式微控制器的 RESET 输入端口即可。MAX810 与微控制器的连接图如图 1.22 所示。

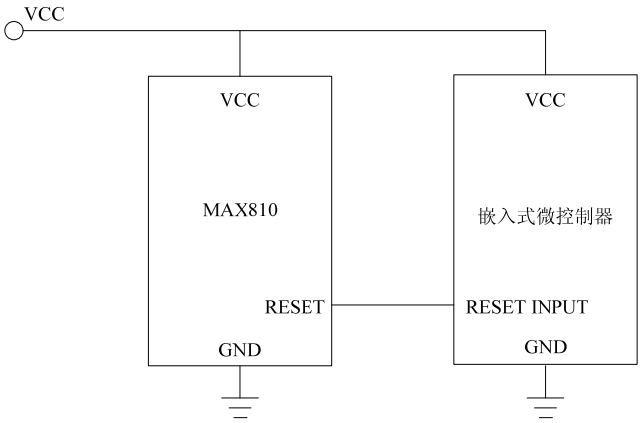

图 1.22　MAX810 与嵌入式微控制器连接图

1.6 最小系统的存储器系统设计

本部分内容是针对不具备片内程序存储器或数据存储器的微控制器而言的。大部分微控制器具备片内程序存储器和数据存储器。一般的设计规则是，如果微控制片内的存储器能够满足系统设计目标的需要，则采用片内存储器系统，这样可以有效地降低开发成本，降低系统开发难度，缩短开发时间。若片内部存储器的存储容量过小，或者系统今后有很大的扩展必要和潜力，可使用外接的 Flash 存储器和 SDRAM 存储器。

嵌入式系统中使用的Flash主要分为NOR和NAND 2种类型，这里我们以NOR型Flash为例进行介绍。NOR型Flash主要特点如下。

- 体积小、容量大，目前可以达到十几 MB。
- 掉电数据不丢失，数据可以保存 10～100 年。
- 有独立的地址和数据总线，可以快速地通过总线读取数据。因此它具有和静态 RAM 相同的读取速度，既可以作为数据存储器使用也可以作为程序存储器使用。
- 写入操作必须通过指令序列来完成。以字节（Byte）或字（Word）为单位，每写入一个 Byte 或 Word 需十几微秒。
- 擦除也通过指令序列完成，以块（Block）为单位，通常块的大小为 64KB，每擦除一个块需要十几毫秒。

由于 Flash 有一定的使用寿命，一般为 10 万～100 万次，所以随着使用次数的增加，会有一些单元逐渐变得不稳定或失效，因此必须能够对其状态加以识别。

由 Flash 特点可以看出，操作 Flash 需要注意以下几点。

- 必须以几 KB 到几十 KB 块为单位进行数据的操作。
- 擦除操作耗时较多，应减少擦除操作。
- 尽量避免频繁地对同一地址操作，以免造成局部单元提前损坏。
- 另外，大部分嵌入式操作系统所挂接的文件系统是建立在以扇区（Sector）为单位的磁盘操作基础上（通常为 512 字节/扇区），因此也需要一段特殊的 Flash 存储管理程序来解决。
- 以扇区为单位的文件系统接口和以块为单位的 Flash 物理特性之间的矛盾，同时，完成各块之间的擦写次数均衡和坏块管理等工作。

关于最小系统的存储器系统设计的详细内容，将在本书第 2 章 Flash 存储器模块和第 3 章 SDRAM 存储器模块中详细介绍，这里只简单介绍 Flash 存储器芯片和 SDRAM 存储器芯片与嵌入式微控制器的连接方法。

下面简要介绍一款 Flash 芯片 SST39LF160 和 ARM 连接电路。

SST39LF160 是一个 1M×16 的 CMOS 多功能 FlashMPF 器件，由 SST 特有的高性能 SuperFlash 技术制造而成（SuperFlash 技术的分裂闸 split-gate 单元结构和厚氧化物制成的）。

SST39LF160 具有高性能的字编程功能，字编程时间为 14μs。器件通过触发位或数据查询位来指示编程操作的完成。为了防止意外写的发生，器件还提供了硬件和软件数据保护机制。SST39LF160 的 10 000 个周期的耐用性和大于 100 年的数据保持时间使其可广泛用于设计制造和测试等应用中。

SST39LF160 具有高性能的字编程功能，字编程时间为 14μs。器件通过触发位或数据查询位来指示编程操作的完成。为了防止意外写的发生，器件还提供了硬件和软件数据保护机制。SST39LF160 的 10 000 个周期的耐用性和大于 100 年的数据保持时间使其可广泛用于设计制造和测试等应用中。SST39LF160 尤其适用于要求程序配置或数据存储器，可方便和低成本地更新的应用。

对于所有的系统，SST39LF160 的使用可显著增强系统的性能和可靠性，降低功耗，它们比其他技术制造的 Flash 器件在擦除和编程操作中消耗更少的能量。所有能量的消耗与应用的电压电流和操作时间有关。由于对于任何的电压范围，SuperFlash 技术都消耗很小的电流，使用很短的擦除时间，因此 SST39LF160 在擦除或编程操作中消耗的能量小于其他 Flash 技术制造而成的器件。SST39LF160 也增强了程序数据和配置存储器的低成本应用的灵活性特性。

SST39LF160 和三星 ARM 芯片 S3C4510 的连接电路如图 1.23 所示。

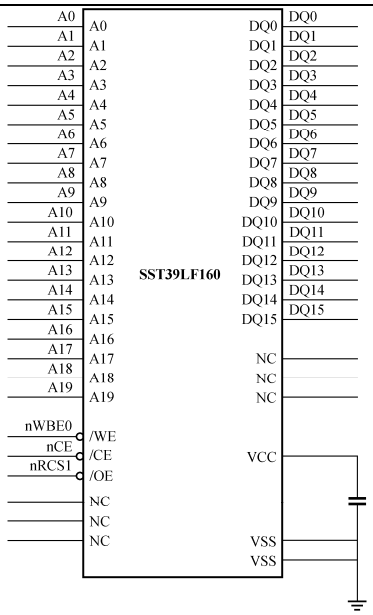

图 1.23　SST39LF160 和 ARM 芯片连接电路图

关于 Flash 的编程将在后面章节中介绍。下面介绍另一类存储芯片 SDRAM 及其与 ARM 的连接方法。

SDRAM（Synchronous DRAM，同步动态随机存取储器）是目前应用最为广泛的一种内存类型，其工作速度与系统总线速度是同步的，现在使用的 SDRAM 按照总路线速度分为 PC-100 或 PC-133 两种。

SDRAM 的特点是大容量和高速度。其单片容量可达 256MB 或更高，工作速度可达 100～200MHz 以上，但是其控制方式比 EDO/FP DRAM 复杂得多。目前，许多嵌入式设备的大容量存储器都采用 SDRAM 来实现。在设计中采用 SDRAM 存储器时，大多都是用专用芯片完成其控制电路。SDRAM 与 S3C4510 的连接电路如图 1.24 所示。

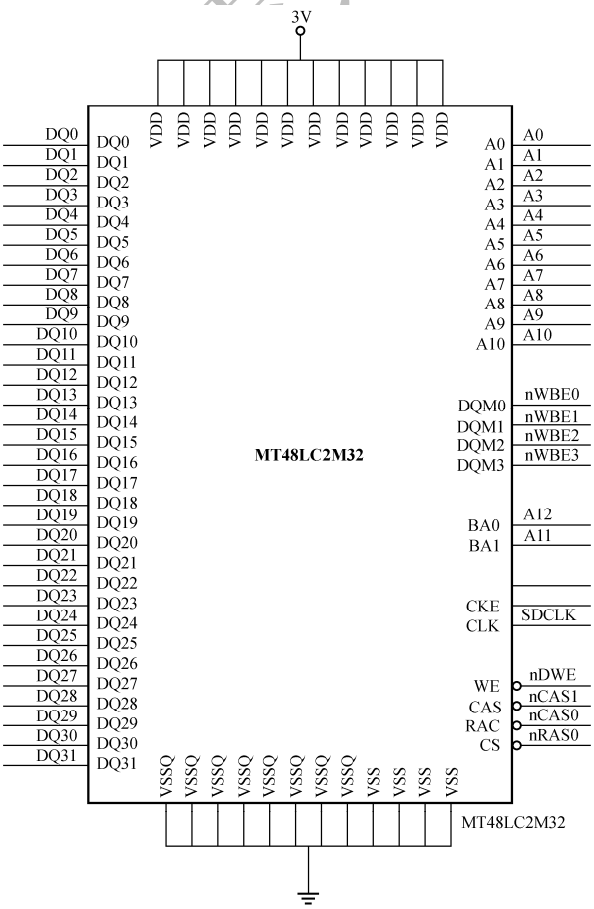

图 1.24　MT48LC2M32 和 ARM 芯片连接电路图

SDRAM 具有多种工作模式，内部操作是一个复杂的状态机。SDRAM 器件的管脚分为以下几类。

- 控制信号，包括片选，时钟，时钟使能，行列地址选择，读写选择，数据有效。
- 地址，时分复用管脚，根据行列地址选择管脚，控制输入的地址为行地址或列地址。
- 数据，双向管脚。

SDRAM 的所有操作都同步于时钟。根据时钟上升沿时控制管脚和地址输入的状态，可以产生多种输入命令。

- 模式寄存器设置命令。
- 激活命令。
- 预充命令。
- 读命令。
- 写命令。
- 带预充的读命令。
- 带预充的写命令。
- 自动刷新命令。
- 自我刷新命令。
- 突发停命令。

本章只是简要介绍 Flash、SDRAM 芯片的用途、特点及与微控制器的连接方法。这两种芯片目前电子市场上可选择的品牌很多，在本书第 2 章中将详细地介绍其他常用的 Flash 和 SDRAM 芯片及编程方法。

1.7 最小系统的软件设计

1.7.1 ARM 嵌入式操作系统简介及选择

在进行基于 ARM 技术的嵌入式系统开发时，通常会遇到操作系统的选择问题。操作系统选择是否恰当直接关系到系统的运行质量。从 8 位/16 位单片机发展到以 ARM CPU 核为代表的 32 位嵌入式处理器，嵌入式操作系统将替代传统的由手工编制的监控程序或调度程序，成为重要的基础组件。更重要的是嵌入式操作系统对应用程序可以起到屏蔽的作用，使应用程序员面向操作系统级开发应用软件，并易于在不同的 ARM 核的嵌入式处理器上移植。嵌入式操作系统都具有一定的实时性，易于裁剪和伸缩，可以适合于从 ARM7 到 Xscale 各种 ARM CPU 和各种档次的应用，嵌入式操作系统可以使用广泛流行的 ARM 开发工具，如 ARM 公司的 SDT/ADS 和 RealView 等，也可以使用开发软件，如 GCC/GDB、KDE 或 Eclipe 开发环境，市场上还有专用的开发工具，如 Tornado、mC/View、mC/KA、CODE/Lab、Metroworks 等。本节就目前国内在 ARM CPU 上广泛采用的 3 种嵌入式操作系统（μC/OS-II、μCLinux 和嵌入式 Linux）进行分析。

1．μC/OS-II 嵌入式实时内核

全世界数百种设备已经在使用 μC/OS-II，包括手机、路由器、不间断电源、飞行器、医疗设备和工业控制设备。μC/OS-II 已经有 ARM7TDMI、ARM9 和 Strong ARM 等各种 ARM CPU 的移植，支持包含 Atmel、Hynix、Intel、Motorola、Philips、Samsung、Sharp 等公司的 ARM 核的 CPU。μC/OS-II 的移植也相当容易，与 CPU 相关的代码包装在 3 个文件中，它们是 os_cpu.h、os_cpu_a.asm 和 os_cpu_c.c。μC/OS-II 有 60 多个系统调用，覆盖任务、定时器、信号量、事件标志、邮箱、队列和内存管理，包含了传统嵌入式操作系统内核（如 PSOS，VRTX）的功能，还支持互斥型信号量，这是 20 世纪 90 年代的嵌入式操作系统内核，如 VxWorks 和 VRTXsa 才有的技术。μC/OS-II 因为是可抢占的实时内核，所以 μC/OS-II 与商业嵌入式实时内核在性能上没有什么差异，μC/OS-II 没有用户态和内核态，任务（线程）或中断和任务切换的响应可

以很快，主要是和 ARM CPU 相关的。最新的 2.7x 版本还增加了算法以避免在移植中修改堆栈指针，这样可以保证μC/OS-II 在不同的 CPU 上运行更稳定，移植更方便。μC/OS-II 目前除了内核外还有商业化文件系统 μC/FS，图形系统 μC/GUI 以及任务调试工具 μC/KA 和 μC/View，但是 μC/OS-II 自己目前还没有 TCP/IP 协议系统。总的来说，μC/OS-II 是一个非常容易学习，结构简单，功能完备和实时性很强的嵌入式操作系统内核，适合于各种嵌入式应用以及大专院校教学和科研。

2．μCLinux 操作系统

μCLinux 是 Linux 小型化后，适合于没有 MMU（内存管理单元）的微处理器芯片而裁剪成的操作系统，如果 ARM CPU 系列中的 ARM7TDMI、ARM940T 等产品希望使用 Linux 操作系统，只能用μCLinux。当然，μCLinux 也支持 MotorolaDragonball、Coldfire 等其他中低端嵌入式处理器。μCLinux 保持了传统 Linux 操作系统的主要特性，包括稳定、强大的网络和文件系统的支持，μCLinux 裁剪了大量的 Linux 内核以缩小尺寸，适合像 512k/b RAM、1M/b Flash 这样小容量、低成本的嵌入式系统。μCLinux 系统小型化的另一简化是采用了 uCLib 库替代 Linux 的 Glib 库，使用μCLib 可以大大减少应用程序的代码尺寸，对于中小型嵌入式应用，uCLib 功能可以满足需要，所以目前即使是某些采用 Linux 2.4 内核的嵌入式 Linux 系统也采用 mCLib 库的做法。μCLinux 中，由于没有内存保护机制，应用代码一般采用静态连接的方式，而且在μCLinux 中采用 Flat 文件执行格式（Linux 是 Coff 或 Elf 格式），μCLinux 和应用代码都支持固化，存储在 Flash 存储介质中，不需像 Linux 需要经过一次 Flash 到 RAM 的拷贝。所有这些，使得μCLinux 更像传统意义上的嵌入式操作系统。 μCLinux 是由 Linux 2.0.38 内核开始移植的，目前已经有 2.4 Linux 支持的一些移植的版本，如 S3C2500、44B0 等 ARM 芯片，可以在μCLinux 的官方网站 www.μCLinux.org 上找到这些版本。μCLinux 近期主要是在发展各种 2.4.x 版本的移植，以期跟上 Linux 社会主流的发展趋势（因为今后 Linux2.6 将开始成为主流的内核）。μCLinux 主要是针对没有 MMU 的嵌入式处理器开发设计，那么它也失去了由 MMU 所带来的 Linux 操作系统的特色。比如，上面已经提到的静态加载（Linux 支持动态应用的加载），μCLinux 对内存操作是直接的物理内存，这样，任何程序的异常都可能导致内核崩溃。μCLinux 支持多线程，但需要父子线程协调同步。μCLinux 的文件系统相对比较陈旧，支持的 CPU 和参考设计还比较少，社区的发展和维护也相对缓慢。目前μCLinux 官方网站还很少看见像 IBM、Intel 这样的大型公司的身影，只有 2～3 家小型的硬件或方案提供商，这对于商业化的产品开发采用是有一定的风险。

3．嵌入式 Linux 操作系统

这里要讨论的是可以嵌入在 ARM CPU 上的，具有 MMU 功能的 Linux 操作系统，也可以称为嵌入式 Linux 操作系统。与微软公司的软件不一样，Linux 不是由一家公司所拥有、维护开发的，Linux 在市场有多种发行版本，所有发行版本都包含一样的 Linux 内核、基本工具和应用，不同的发行版本主要是在附加的工具链、应用、配置以及各种内核补丁上有所不同。嵌入式 Linux 主要是在实时性增强、内核精简和裁减、支持多种 CPU 结构（如 ARM CPU）等方面做了改进和提高。使用嵌入式 Linux 系统有 2 种途径，第一是用户自己装配（称为 DIY 内核），你可以在 www.kernel.org 找到全部 Linux 代码，或直接到 ARM CPU 的源代码树下 www.armlinux.org.uk 找到所需要的 Linux 版本的移植，或者某些半导体公司，如三星、Motorola 在自己的网站或在自己的 ARM 评估板含有一个最小 Linux 内核系统。如果这个最小内核没有包含 GCC/GDB 工具链，可能还要到 GNU 的网站下载全部的源代码，然后再编译生成所需要版本的 ARM 工具链和应用程序库，这个过程是相当耗时和困难的。还需要指出，这种 DIY 内核的配置，添加应用和驱动程序也是不标准的和复杂的，这是嵌入式系统的特殊性所在。第二是选择一个商业化的嵌入式 Linux 操作系统平台。商业化的嵌入式 Linux 版本是针对嵌入式处理器，如 ARM 所优化设计的，支持各种半导体厂家的评估板和主要的设备驱动，商业化的嵌入式 Linux 包含了文件系统、应用、实时性扩展和技术支持培训服务，现今国外著名的商业化嵌入式 Linux 产品有：MontaVista Linux、Bluecat Linux、Timesys Linux、Metrowork Linux、Vlinux 和 Redhat Linux 等，国内也有红旗、中软、新华嵌入式 Linux 等。

MontaVistaLinux 是 MontaVista 软件公司于 1999 年推出的，它是目前全球优秀的嵌入式 Linux 操作系统和工具供应商。MontaVista 在嵌入式 Linux 的实时性、交叉开发工具、高可用性、动态电源管理等 Linux 技术要点方面具有领先地位。MontaVista Linux 最版本是 3.1，采用 Linux 2.4.20，针对 8 种 CPU 系列（包含 ARM 和 Xscale）优化定制的商业化版本。选择像 MontaVista Linux 这样商业化嵌入式 Linux，可以让用户把时间和资金放在应用软件和特定的硬件接口和设备驱动程序方面。使用商业化嵌入式 Linux 可以得到一定时间（一般是 1 年）的技术支持、升级和培训（这很重要，因为 Linux 是每天都在变化的）。商业化嵌入式 Linux 目前除国内的产品外，价格还是很昂贵的，根据配置和服务时间，大约从几千到几万美元，多数国内用户从资金和心理上还很难承受，商业化嵌入式 Linux 开发工具相对于 Microsoft 和像 Tornado/VxWork 的开发工具，在易于使用和丰富性方面还有待于提高和改进。同样作为 Linux 操作系统，笔者推荐使用带有 MMU 的嵌入式 Linux，而不是μCLinux，因为绝大多数新的 ARM CPU 都是 AMR9 核，它们都带有 MMU 了，无论是开放源码的 Linux 社区还是商业化的嵌入式 Linux 公司的支持和维护都比μCLinux 要好要快并丰富得多。

归纳一下选择一个合适的 ARM CPU 的嵌入式操作系统有以下几个重要因素。

第一是应用。如果你想开发的嵌入式设备是一个和网络应用密切相关或者就是一个网络设备，那么你应该选择用嵌入式 Linux 或者 CLinux，而不是μC/OS-II。

第二是实时性。没有一个绝对的数字可以告诉你什么是硬实时，什么是软实时，它们之间的界限是十分模糊的，这与你选择的 ARM CPU 的主频、内存等参数有一定的关系。如果你使用加入实时补丁等技术的嵌入式 Linux，如 MontaVista Linux（2.4.17 版本），最坏的情况只有 436μs，而 99.9%的情况是 195μs。考虑到最新的 Linux 在实时性方面的改进，它可以适合于 90%～95%的嵌入式系统应用。当然，你如果希望更快地实时响应，如高速的 A/D 转换需要几个微秒以内的中断延时，可能采用μC/OS-II 是合适的。当然，采用像 Vxworks 这样传统的嵌入式操作系统也可以满足强实时性要求。

第三是开发工具。显然，目前μC/OS-II、μCLinux 和嵌入式 Linux 的开发工具与商业嵌入式操作系统工具还有一些差距，目前在 ARM CPU 上广泛流行和使用的是 ARM 公司 SDT/ADS 工具链，产品无论在功能、稳定性和众多的第三方厂商支持方面都很好，唯一不足的是缺少对嵌入式 Linux 操作系统的支持，SDT/ADS 的升级产品 RealView 计划支持 GCC 和嵌入式 Linux，但目前还没有看到，μC/OS-II 可以使用 ARM SDT/ADS，但没有操作系统调试功能。

第四是所选择的 ARM CPU 和参考板，像 ARM7TDMI 和 ARM940T（如 S3C2500/2510）核是不能使用嵌入式 Linux 的，如果想用 Linux，只能用μCLinux，如果想用 VxWorks，需要了解一下提供评估板的公司是否有 BSP（板支持包），VxWorks 自己只有少数 ARM 公司评估板的支持。

最后是价格和技术服务。在考虑购买商业嵌入式操作系统时，会遇到是买还是自己做的问题，这是很正常的，尤其是在采用开放源代码技术时，这个问题更加突出。

1.7.2 基于 μCLinux 操作系统的设计

下面以μCLinux 操作系统为例介绍嵌入式系统软件设计的基本方法。

基于μCLinux 的系统开发大致可分为以下几个步骤。

- 建立主机平台。
- 准备内核源代码和交叉编译工具。
- 配置并编译内核。
- 移植μClinux。
- 开发μCLinux 下的应用程序。

1．建立主机平台

在个人 PC 上建立 Linux 操作系统，这里选择热 ahat9.0。

2．准备内核源代码和交叉编译工具

μCLinux 的源代码可以到μCLinux 的官方网站上（http://www. μCLinux.org）下载，随着标准 Linux 内核的升级，该网站也不断推出新版本的μCLinux 内核版本，同时还有某些硬件体系的针对型版本，即不需要进行移植。这里以 S3C4510 的μCLinux 版本为例进行介绍，即μClinux-dist-20030522.tar.gz 压缩包，然后用以下命令解压内核源码包：tar xzf μCLinux-Samsung-20020522.tar.gz，这样就产生μCLinux-dist 子目录，所有的内核源码都在此目录下。

有了内核源代码，还要根据目标平台到上述网站下载交叉编译器。交叉编译器的作用是实现在主机上编译而在目标平台上运行的代码的生成。这里下载的交叉编译器是 arm-elf-tools-20030314.sh 压缩包，该压缩包包含了内核代码和应用程序编译，连接以及调试用的大部分工具，例如 C 编译器 arm-elf-gcc，连接器 arm-elf-ld，目标格式工具 genromfs 和 elf2flt 等。接下来需要安装交叉编译器，即用下列命令：sharm-elf-tools-20030314.sh，这样就会自动在/usr/local/bin 目录下建立整套的 ARM 的 ELF 交叉编译器工具。

3．配置和编译内核

有了源码和交叉编译器就可以开始配置和编译内核了。在解压源码时生成的μCLinux-dist 目录下用命令 make menuconfig 开始配置内核。

首先在弹出的一个对话框中单击 Target Platform Selection，进入顶级配置界面，具体如下。

[Target Platform Selection]

vendor/product:即厂商/产品。μCLinux 支持很多厂商的标准评估板，可以根据自己的实际情况进行选择，这里暂时选择三星 4510；Kernel Version：内核版本。有 2 个版本可以选择，linux-2.0.x 和 linux-2.4.x，这里选择后者；Libc Version:库函数的版本。有 uClibc 和 uC-libc 2 种库函数，前者内容更丰富，支持多线程，但选择后者编译生成的内核相对小了很多，这里暂时选择后者。nux-dist/vendors/下相应的微处理器的默认配置文件是用来设置各个选项的，因为选择该项后，以前的选择将丢失，故一般打上 N 表示不选。

[]Customize Kernel Settings：自定义内核，一般选择该项。

[]Customize Vendor/User Settings：自定义用户程序和库函数设置。

[]Update Default Vendor Settings：更新厂家默认设置，一般不选。

顶级配置完成后，选择 Save and exit，如果顶级设置中 Customize Kernel Settings 选择了 Y，则接下来进入 Kernel Configuration，否则跳过该配置，如果对内核配置不熟悉，那省事的办法就是不做改动，选择默认设置。

内核配置完成后选择 Save and Exit。如果顶级设置中 Customize Vendor/User Settings 选择了 Y，则接下来进入 Application Configuration，否则跳过该配置，如果对系统应用配置不熟悉，同样省事的办法就是不做改动，选择默认设置。然后选择 Save and Exit。配置完后，按顺序执行以下命令进行编译：make dep、make clean、make lib_only、make user_only、make romfs、make image 和 make。

在执行 make image 时会出现错误报告，可忽略继续执行下面的命令。这样当 make 命令执行完后，在μCLinux-dist/images 目录中生成 3 个新文件：romfs.img、image.ram 和 image.rom。romfs.img 是文件系统的二进制文件，iamge.ram 文件是未经压缩的μCLinux 系统文件，把它复制到内存中就可以直接从入口运行了，而 iamge.rom 文件是压缩的μCLinux 系统文件，把它烧录在作为启动的 Flash 芯片中，上电后 image.rom 就通过内置的引导程序启动μCLinux，需要注意的是，这里生成的μCLinux 是三星 4510 版本的。

4．移植μCLinux

所谓操作系统的移植是指使一个操作系统能够在某个微处理器平台上正常稳定运行的过程。操作系统的移植主要指 3 个级别的移植：结构体系级别的移植、处理器级别的移植和基于平台的板级移植。在

移植内核前，一般先进行 BootLoader 的移植，有了 BootLoader 就可以通过串口或网口甚至 USB 口把 μCLinux 内核倒到 RAM 中直接启动，无须烧写 flash（flash 的烧写次数有限，且操作麻烦），这样给应用程序的开发调试带了极大的方便！移植 bootloader 同样是先找个接近的版本，目前常见的 bootloader 有 ARMBOOT、REDBOOT、DBUG、BLOB，其中 BLOB 因其良好的可移植性和强大的功能而在网上被广大嵌入式爱好者和开发者进行广泛地讨论，在网站 http://www.start-web.net/tpu/上有文件名为 blob-mba44b0.tgz 压缩包，是针对 S344B0X 的开发板移植的 BLOB 版本，当然要应用在自己的系统上，还需作些修改。

移植步骤如下。

在 makefile 中添加如下代码。

```
ifeq ($(CONFIG_ARCH_44B0X),y)
TEXTADDR = 0x0c008000
MACHINE = 44b0x
Endif
```

在 config.in 中添加如下代码。

```
44B0X CONFIG_ARCH_ 44B0X \
if [ "$CONFIG_ARCH_44B0X " = "y" ]; then
define_bool CONFIG_NO_PGT_CACHE y
define_bool CONFIG_CPU_32 y
define_bool CONFIG_CPU_26 n
define_bool CONFIG_CPU_ARM710 y
define_bool CONFIG_CPU_WITH_CACHE y
define_bool CONFIG_CPU_WITH_MCR_INSTRUCTION n
define_bool CONFIG_SERIAL_44B0 y
define_bool CONFIG_VT y
define_hex DRAM_BASE 0x0C000000
define_hex DRAM_SIZE 0x00800000
define_hex FLASH_MEM_BASE 0x00000000
define_hex FLASH_SIZE 0x00200000
```

在/linux-2.4.x/arch/armnommu/下新建 mach-44b0x 文件夹，同时参照 MICETEK 版本相应目录下的文件，在该文件夹中分别新建 3 个文件：arch.c，irq.c，time.c。在/linux-2.4.x/arch/ armnommu/boot/makefile 中添加如下代码。

```
ifeq ($(CONFIG_ARCH_44B0X),y)
ZRELADDR
= 0x0C008000
ZTEXTADDR = 0x00000000
ZBSSADDR = 0x0C400000
Endif
```

在μCLinux-2.4.x/arch/armnommu/kernel/ entry-armv.S 中添加如下代码。

```
#elif defined(CONFIG_ARCH_44B0X)
.macro disable_fiq
.endm
.macro get_irqnr_and_base, irqnr, irqstat, base, tmp
ldr \base, =rI_ISPR
ldr \irqnr, [\base]
teq \irqnr, #0
.endm
```

```
.macro irq_prio_table
.endm
```

在μCLinux-2.4.x/arch/armnommu/kernel/head-armv.S 中添加如下代码。

```
#elif   defined(CONFIG_ARCH_44B0X)
mov r1, #MACH_TYPE_44B0X
#if     defined(CONFIG_ARCH_44B0X)
adr     r5, LC0
ldmia   r5, {r5, r6, r8, r9, sp}
@ Setup stack
/* Copy data sections to their new home. */
/* Clear BSS */
mov r4, #0
1:
cmp r5, r8
strccr4, [r5],#4
bcc     1b
/* Pretend we know what our processor code is (for arm_id)*/
ldr r2, EV44B0_PROCESSOR_TYPE
str     r2, [r6]
mov     r2, #MACH_TYPE_44B0X
str     r2, [r9]
mov fp,   #0
b         start_kernel
LC0:
.long     __bss_start
.long     processor_id
.long     _end
.long     __machine_arch_type
.long     init_task_union+8192
EV44B0_PROCESSOR_TYPE:
.long   0x34345036
#endif
```

在μCLinux-2.4.x/arch/armnommu/kernel/irq.c 中添加如下代码。

```
asmlink void do_IRQ (int irq, struct pt_regs * regs)
{
struct irqdesc * desc;
struct irqaction * action;
int cpu;
#ifdef CONFIG_ARCH_SAMSUNG
CLEAR_PEND_INT(irq);
#endif
irq = fixup_irq(irq);
#ifdef CONFIG_ARCH_44B0X ////新增代码
irq = 44B0X _fix_44b0(irq);
CLEAR_PEND_INT(irq);
#endif
```

```
#ifdef CONFIG_ARCH_44B0X
int 44B0X _fix_44b0(int x)
{
int i= 0;
for (i = 0; i< 26; i++)
{
if( x = = 1)                break;
x = x >> 1;
}
return i;
}
#endif
```

在/linux-2.4.x/arch/armnommu/mm 目录下 proc-arm6,7.S 文件中添加如下代码。

```
cpu_ev44b0_manu_name:
        .asciz      "44B0X"
cpu_ev44b0_name:
        .asciz      "S3C44B0X"
```

在/linux-2.4.x/arch/armnommu//tools/mach-types 中添加如下代码。

```
44b0x     ARCH_44B0X     44B0X     178
```

在/linux-2.4.x/driver/char/config.in 文件中添加如下代码。

```
if [ "$CONFIG _ARCH_44B0X " = "y" ]; then
bool 'EV44B0 serial port support' CONFIG_SERIAL_44B0
if [ "$CONFIG_SERIAL_44B0" = "y" ]; then
bool'Support for console on Samsung serial port' CONFIG_SERIAL_SAMSUNG_CONSOLE
fi
if [ "$CONFIG_SERIAL_44B0" = "y" ]; the
bool ' Support for IRDA on Samsung serial port' CONFIG_SERIAL_SAMSUNG_IRDA
fi
fi
```

在/linux-2.4.x/driver/char/makefile 文件中添加如下代码。

```
obj-$(CONFIG_SERIAL_44B0X) += serial_44b0.o
```

在/linux-2.4.x/driver/char/目录下新建 serial_44b0.c。

µCLinux-2.4.x/include/asm-armnommu/下，新建 arch-44b0x 文件夹，拷贝相应 arch-micetek 下的所有文件。

在µCLinux-2.4.x/include/asm-armnommu/proc-armv/ system.h 中添加如下代码。

```
#ifdef __ARM_ARCH_4__(源码)
#define vectors_base() ((cr_alignment & CR_V) ? 0xffff0000 : 0)
#else
#ifdef   CONFIG_ARCH_44B0X(新增代码)
#define vectors_base() (0x0c000000)
#else
#define vectors_base() (0)
#endif
#endif
```

到这里移植过程完毕，经重新配置生成的µCLinux 版本就可以在 44B0 上运行了。

5．开发μCLinux 下的应用程序

接下来的工作就是开发者根据自己的需要开发应用程序。开发μCLinux 下的应用程序应注意以下内容。

我们知道，在标准 linux 平台上，已经有了非常丰富且源码开放的应用程序，使得开发者很容易获得参考。然而，值得注意的是，由于标准 linux 和 μCLinux 之间的差异，许多已经在标准 linux 环境中运行很好的程序并不能直接在 μCLinux 环境下工作。这其间的原因可简略归纳为以下 2 个方面，一是由于μCLinux 所使用的处理器和普通 PC 不同，指令集，CPU 结构上的差异导致 μCLinux 上的运行的程序需要专门为该类型处理器交叉编译产生，就是前面提到的交叉编译器；另一方面，μCLinux 是为了没有内存管理单元（MMU）的处理器或控制器设计的，在内存处理上作了较大的修改和精简，所以在标准 Linux 上可以使用的一些函数和系统调用在μCLinux 上就有可能行不通。当然在μCLinux 源码中也有许多开发好的应用例程，具体在/usr/目录下，例如常用的有以下几个，boa，适合于嵌入式应用的 WebServer；busybox，适合于嵌入式应用的工具软件集；flashw，Flash 写入程序；gdbserver，目标系统端远程调试程序，与主机上运行的 GDB 软件配合完成对目标系统上运行的程序进行远程调试的功能；jffs-tools，jffs 文件系统（适合 Flash 存储器的）工具软件；ping，网络测试程序 ping；tip，串口连接程序。/usr/目录下所有例子，有的可以直接利用，即在配置内核时把它选上即可使用，如 ping；而有的必须针对具体系统情况进行修改，如 flashw。在编写自己的应用程序时可以参考这些例程。

1.7.3 BootLoader

BootLoader 是学习嵌入式系统的一个非常重要的环节。它与硬件紧密结合，通过对它的结构、原理的学习并实际动手编写简单的 BootLoader 程序，将有助于更深入地理解嵌入式系统。所以同建立μCLinux 的根文件系统一样，编写 BootLoader 程序是一个非常重要的实验项目。做好这 2 个实验能为以后应用程序设计的实验打下良好的软硬件知识基础。这 2 个实验项目可以让学生熟悉该实验平台，掌握 ARM 的体系结构及其启动初始化过程，能熟练地应用 Thumb 指令及 C 编写初始化程序。同时还可以了解μCLinux 的文件系统，掌握内核的编译方法及根文件系统的制作方法。BootLoader 是在操作系统内核运行之前运行的一段小程序。通过这段小程序，初始化硬件设备、建立内存空间的映射图，从而将系统的软硬件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，即是我们说所的引导加载程序。在嵌入式系统中，通常没有像 PC 机 BIOS 那样的固件程序，因此整个系统的加载启动任务就完全由 BootLoader 来完成。对于我们的实验开发板，由于 flash 接的是微处理器 Bank0 的片选信号，映射的地址为 0x00000000。而实验系统在上电或复位时从地址 0x00000000 处开始执行，所以在这个地址处安排系统的 BootLoader 程序。由于 BootLoader 与硬件结合非常紧密，所以对于不同结构的 CPU 其 BootLoader 程序是不同的。正是由于这样，我们以 S3C44B0X 为例来介绍 BootLoader。

首先要对 S3C44B0X 的寄存器有所了解。S3C44B0X 处理器共有 37 个寄存器，分为若干个组（Bank），这些寄存器包括：31 个通用寄存器，包括程序计数器（PC 指针），均为 32 位的寄存器；6 个状态寄存器，用以标识 CPU 的工作状态及程序的运行状态，均为 32 位，目前仅使用了其中的一部分。

通用寄存器包括 R0～R15，可分为 3 类。

- 未分组寄存器 R0～R7。
- 分组寄存器 R8～R14。
- 程序计数器 R15。

同时，ARM 处理器又有 7 种不同的处理器模式。

- 用户模式（usr）ARM 处理器正常的程序执行状态。
- 快速中断模式（fig）用于高速数据传输或通道处理。
- 外部中断模式（irq）用于通用的中断处理。
- 管理模式（svc）操作系统使用的保护模式。
- 数据访问终止模式（abt）当数据或指令预取终止时进入该模式。可用于虚拟存储及存储保护。

- 系统模式（sys）运行具有特权的操作系统任务。
- 未定义指令终止模式（und）当未定义的指令执行时进入该模式，可用于支持硬件协处理器的软件仿真。

ARM 微处理器的运行模式可以通过软件改变，也可以通过外部中断或异常处理改变。在所有的运行模式下，未分组寄存器都指向同一物理寄存器，它们未被系统用作特殊的用途，因此，在中断或异常处理进行运行模式转换时，由于不同的处理器运行模式均使用相同的物理寄存器，可能会造成寄存器中数据的破坏。对于分组寄存器，它们每一次所访问的物理寄存器与处理器当前的运行模式有关。对于 R8～R12，每个寄存器对应 2 个不同的物理寄存器，当使用 fig 模式时，访问寄存器 R8_fig-R12_fig；当使用除 fig 模式以外的其他模式时，访问寄存器 R8_usr-R12_usr。对于 R13、R14，每个寄存器对应 6 个不同的物理寄存器，其中 1 个是用户模式和系统模式共用，另外 5 个物理寄存器对应其他 5 种不同的运行模式。寄存器 R13 在 ARM 指令中常用作堆栈指针，但也可以使用其他的寄存器作为堆栈指针。而在 Thumb 指令集中，某些指令强制性地要求使用 R13 作为堆栈指针。R14 也称作子程序连接寄存器（Subroutine Link Register）或连接寄存器 LR。当执行 BL 子程序调用指令时，R14 中得到 R15（程序计数器 PC）的备份。其他情况下，R14 用作通用寄存器。R15 用作程序计数器（PC），虽然也可以用作通用寄存器，但一般都不那么使用，因为对 R15 的使用有一些特殊的限制，当违反了这些限制时，程序的执行结果是未知的。由于 ARM 体系结构采用了多级流水线技术，对 ARM 指令集而言，PC 总是指向当前指令的下 2 条指令的地址，即 PC 的值为当前指令的地址值加 8 个字节。总体说来，在 ARM 状态下，任一时刻可以访问以上所讨论的 16 个通用寄存器和 1～2 个状态寄存器。在非用户模式（特权模式）下，则可以访问到特定模式分组寄存器。

在编写 BootLoader 程序之前，我们必须明白这段程序的任务是什么，要做些什么样的工作。而由于 BootLoader 的实现依赖于 CPU 的体系结构，针对不同的 CPU 需要编写不同的代码，如果全部用汇编来写那么代码将不能移植到其他结构不同的 CPU 上。因此我们将 Boot Loader 设计为 stage1 和 stage2 两大部分。依赖于 CPU 体系结构的代码，比如设备初始化代码等，都放在 stage1 中，并且都用汇编语言来实现，以达到短小精悍的目的。而 stage2 则用 C 语言来实现，这样在后续的实验开发中可以不断对功能进行扩展和加强，而且代码会具有更好的可读性和可移植性。BootLoader 的 stage1 完成的功能可多可少，但是必须包括以下步骤。

- 硬件设备初始化。
- 设置堆栈。
- 为加载 BootLoader 的 stage2 准备 RAM 空间。
- 拷贝 BootLoader 的 stage2 到 RAM 空间中。
- 跳转到 stage2 的 C 入口点。

BootLoader 的 stage2 必须包括以下基本的步骤。

- 初始化本阶段要使用到的硬件设备。
- 下载内核及根文件系统的映像文件到实验开发板。
- 将 kernel 映像和根文件系统映像从 flash 上读到 RAM 空间中。
- 加载内核映像和根文件系统映像。
- 调用内核。

1．BootLoader stage1 的功能

在涉及具体汇编代码前，有一些术语必须要说明一下。

- 映像文件（image）：指一个可执行文件，在执行的时候被加载到处理器中。它是 ELF Executable and Linking Format）格式的。
- 段（section）：描述映像文件的代码或数据块。
- RO：是 Read-Only 的简写形式。一般存放的代码。
- RW：是 Read-Write 的简写形式。一般存放初始化的数据。
- ZI：是 Zero-Initialized 的简写形式。一般是存放零初始化数据。

- 输入段（Input Section）：它包含代码、初始化数据或描述了在应用程序运行之前必须初始化为 0 的一段内存。
- 输出段（Output Section）：它包含了一系列具有相同 RO、RW、ZI 属性的输入段。
- 域（Regions）：在一个映像文件中，一个域包含了 1～3 个输出段。多个域组织在一起就构成了最终的映像文件。

2．硬件的初始化

任何一个系统启动的最初任务都是硬件初始化。我们自己的 BootLoader 也不例外，其主要目的是为 stage2 的执行以及随后的 kernel 的执行准备好一些基本的硬件环境。它包括以下步骤。

（1）屏蔽所有的中断。为中断提供服务是操作系统的设备驱动程序的责任，因此在 BootLoader 的执行全过程中不必响应任何中断。中断屏蔽可以通过写 ARM 的 CPSR 寄存器来完成。代码段如下。

```
ldr r0,=INTMSK；INTMSK 中断屏蔽寄存器，地址为 0x01e0000c
ldr r1,=0x07ffffff；禁止所有中断
str r1,[r0]
```

（2）设置 CPU 的速度和时钟频率。代码段如下。

```
ldr r0,=LOCKTIME；LOCKTIME 锁定时间计数值寄存器，地址为
0x01d8000c
ldr r1,=0xfff；初始值
str r1,[r0]
PLLSETSTART
ldr r0,=PLLCON；PLLCON 锁相环控制寄存器，地址为 0x01d80000
ldr r1,=((M_DIV<<12)+(P_DIV<<4)+S_DIV)
```

设定系统主时钟频率

```
str r1,[r0]
ldr r0,=CLKCON；CLKCON 时钟控制寄存器，地址为 0x01d80004
ldr r1,=0x7ff8
```

3．堆栈初始化

设置堆栈指针是为执行 C 语言代码做准备。把 SP 的值设置在 RAM 空间距最顶端 1.5KB 的地方（堆栈向上生长）。因为在不同的工作模式下所访问的堆栈指针寄存器（R13）不同，所以要初始化各种工作模式下的堆栈。

```
mrs r0,cpsr ;
bic r0,r0,#MODEMASK ;
orr r1,r0,#UNDEFMODE|NOINT;UNDEFMODE:0X1BNOINT:0XC0,设为未定义指令终止模式禁止 FIQ 和 IRQ 中断
msr cpsr_cxsf,r1
ldr sp,=UndefStack;0xc7ffb00
orr r1,r0,#ABORTMODE|NOINT
msr cpsr_cxsf,r1;数据访问终止模式
ldr sp,=AbortStack ;0xc7ffd00
orr r1,r0,#IRQMODE|NOINT
msr cpsr_cxsf,r1 ;外部中断模式
ldr sp,=IRQStack ;0xc7ffe00
orr r1,r0,#FIQMODE|NOINT
```

```
msr cpsr_cxsf,r1 ;快速中断模式
ldr sp,=FIQStack ;0xc7fff00
bic r0,r0,#MODEMASK|NOINT
orr r1,r0,#SVCMODE
msr cpsr_cxsf,r1 ;管理模式
ldr sp,=SVCStack ;0xc7ffb00
```

4．RAM 初始化

包括正确地设置系统内存控制器的功能寄存器以及各内存库控制寄存器等共 13 个需要初始设置的寄存器。设置代码段如下。

```
adr r0, ResetHandler ;取得 ResetHandler 的地址
ldr r1, =ResetHandler
sub r0, r1, r0 ;
ldr r1, =SMRDATA ;取得这 13 个寄存器的初始设置段地址
sub r0, r1, r0 ;r0 指向 13 个寄存器初始化段
ldmia r0, {r1-r13} ;将 13 个初始值推入到 r1-r13 中;
ldr r0, =0x01c80000 ;将 r0 指向 BWSCON, 地址为 0x01c80000.
stmia r0, {r1-r13} ;将初始值填入这 13 个寄存器
```

其中从 SMRDATA 标号开始的一段连续的地址中存放了这 13 个寄存器的初始化数据。

5．为 stage2 准备 RAM 空间

为了获得更快的执行速度，我们设计把 stage2 加载到 RAM 空间中来执行，因此必须为加载 BootLoader 的 stage2 准备好一段可用的 RAM 空间范围。但是这个空间范围到底需要多大呢。考虑到 stage2 是 C 语言执行代码，因此在考虑空间大小时，除了 stage2 可执行映像的大小外，还必须把堆栈空间也考虑进来。

另外 ARM 的寻址空间虽然为 4G，但实际映射的固态存储器的地址范围并没那么大，往往只有几十兆，对于我们 S3C44B0X 开实验板来说，在 Bank6 的片选信号上我们接了 8M 的 SDRAM，而 Bank6 映射的起始地址为 0x0c000000，进而 RAM 的空间范围为 0x0c000000～0x0c7fffff，这时只要保证在 ADS 开发环境下指定程序的 RAM 运行空间在该范围内就可以避免使用到无效的 RAM 空间。但就一般情况而言，可以对 RAM 空间是否有效进行测试。

```
Ldr r2, BaseOfBSS ;数据段基地址
Ldr r3, BaseOfZero ;未初始化的数据段基址 0
cmp r2, r3 ;比较 r2、r3 的值
ldrcc r1, [r0], #4 ;r2<r3,就将 r0 的值赋给 r1,r0 地址+4，r0 的值就是 RW 段的值
strcc r1, [r2], #4 ;r2<r3,就将 r1 的值赋给 r2 指向的地址,r2 地址+4
bcc%B0 ;跳回标号 0 处执行,直到 r2=r3
mov r0,#0 ;将 0 装入 r0
cmp r2,r3 ;比较 r2 和 r3,此时 r2 值为 BaseOfZero
strcc r0, [r2], #4 ;r2<r3,就将 r0 的值装入 r2 指向的地址,r2 地址+4
bcc%B1 ;跳回标号 1 处执行,直到 r2=r3
```

6．拷贝 stage2 到 RAM

这里我们专门在 BootLoader 中设计了一个拷贝程序，负责将 stage2 的代码搬运到 RAM 中，让它在 RAM 中运行，以获得更快的速度。其整个搬运过程的代码段如下。

```
ldr r2,
=CopyProcBeg ;拷贝程序的起始地址
sub r1, r2, r1 ;r1 为 RO 段基址，计算偏移量
add r0, r0, r1;r0=0,为 flash 的起始地址,算出了拷贝程序在 flash 中的地址,该值赋给 r0
ldr r3,=CopyProcEnd ;拷贝程序的结束地址;
//将拷贝程序复制到 RAM 中//
ldmia r0!, {r4-r7};将以 r0 所指的连续地址空间的内容写到 r4-r7 中,并将最后地址写回 r0
stmia r2!, {r4-r7} ;将 r4-r7 的内容写回到 r2 所指的联系地址空间中,并将最后地址写回到 r2
cmp r2, r3 ;较 r2 与 r3 的内容,看是否到拷贝终止地址
bcc %B0 ;跳转到标号 0 处执行,直到 r2=r3
;//开始用 flash 中的拷贝程序复本将所有剩下的代码复制到 ram 中//
ldr pc, =CopyProcBeg ; 将程序指针指向 RAM 中拷贝程序的起始地址;//本段将代码由实际烧入的地址拷贝到 ro-base 所指定的位置只拷贝 CopyProcEnd 以后的代码,即 stage2 的代码//
CopyProcBeg
0
ldmia r0!, {r4-r11}
stmia r2!, {r4-r11}
cmp
r2, r3
bcc
%B0
CopyProcEnd
```

上述 2 段代码所用到的 BaseOfBSS、EndOfBSS、BaseOfROM、TopOfROM 等的定义如下。

```
BaseOfROM DCD |Image$$RO$$Base|;为 BaseOfROM 分配一段空间并用 Image 中 RO 段基址的值初始化
TopOfROM DCD |Image$$RO$$Limit| ;RO 段的大小
BaseOfBSS DCD |Image$$RW$$Base|
;RW 段的基址
BaseOfZero DCD |Image$$ZI$$Base| ;ZI 段基址
EndOfBSS DCD |Image$$ZI$$Limit| ;ZI 段的大小
```

在上述一切都就绪后，就可以跳转到 BootLoader 的 stage2 去执行了。

在 ARM 系统中，通过修改 PC 寄存器为合适的地址来实现。stage2 的代码用 C 语言实现，以便于实现更复杂的功能和取得更好的代码可读性和可移植性。当 stage1 的工作完成，为 stage2 准备好运行环境后就可以跳转进 main()函数了。直接把 main()函数的起始地址作为整个 stage2 执行映像的入口点是最直接的想法。但是这样做将无法处理 main()返回的情况。为解决这个问题，可以写一段 ARM 的汇编程序，作为 main()函数的外部包裹。

```
Lable
BL      Main;从汇编进入 C 语言代码空间
B       Lable
```

这样当 main()函数返回后，我们又用一条跳转指令跳转到标号处继续执行程序，也就重新执行 main()函数。

7．硬件初始化

```
{
//PORT A GROUP
PCONA
```

位名称	BIT	描述
PA9	[9]	0= Output 1 = ADDR24
PA8	[8]	0 = Output 1 = ADDR23
PA7	[7]	0 = Output 1 = ADDR22
PA6	[6]	0 = Output 1 = ADDR21
PA5	[5]	0 = Output 1 = ADDR20
PA4	[4]	0 = Output 1 = ADDR19
PA3	[3]	0 = Output 1 = ADDR18
PA2	[2]	0 = Output 1 = ADDR17
PA1	[1]	0 = Output 1 = ADDR16
PA0	[0]	0 = Output 1 = ADDR0

//

rPCONA = 0x3ff;

//PORT B GROUP

rPCONB

位名称	BIT	描述
PB10	[10]	0 = Output 1 = nGCS5
PB9	[9]	0 = Output 1 = nGCS4
PB8	[8]	0 = Output 1 = nGCS3
PB7	[7]	0 = Output 1 = nGCS2
PB6	[6]	0 = Output 1 = nGCS1
PB5	[5]	0 = Output 1 = nWBE3/nBE3/DQM3
PB4	[4]	0 = Output 1 = nWBE2/nBE2/DQM2
PB3	[3]	0 = Output 1 = nSRAS/nCAS3
PB2	[2]	0 = Output 1 = nSCAS/nCAS2
PB1	[1]	0 = Output 1 = SCLK
PB0	[0]	0 = Output 1 = SCKE

//

rPDATB = 0x1cf;

rPCONB = 0x1cf;

//PORT C GROUP

//BUSWIDTH=16

PCONC

位名称	BIT	描述
PC15	[31:30]	00 = Input 01 = Output 10 = DATA31 11 = nCTS0
PC14	[29:28]	00 = Input 01 = Output 10 = DATA30 11 = nRTS0
PC13	[27:26]	00 = Input 01 = Output 10 = DATA29 11 = RxD1
PC12	[25:24]	00 = Input 01 = Output 10 = DATA28 11 = TxD1
PC11	[23:22]	00 = Input 01 = Output 10 = DATA27 11 = nCTS1
PC10	[21:20]	00 = Input 01 = Output 10 = DATA26 11 = nRTS1
PC9	[19:18]	00 = Input 01 = Output 10 = DATA25 11 = nXDREQ1
PC8	[17:16]	00 = Input 01 = Output 10 = DATA24 11 = nXDACK1
PC7	[15:14]	00 = Input 01 = Output 10 = DATA23 11 = VD4
PC6	[13:12]	00 = Input 01 = Output 10 = DATA22 11 = VD5
PC5	[11:10]	00 = Input 01 = Output 10 = DATA21 11 = VD6

PC4 [9:8] 00 = Input 01 = Output 10 = DATA20 11 = VD7

PC3 [7:6] 00 = Input 01 = Output 10 = DATA19 11 = IISCLK

PC2 [5:4] 00 = Input 01 = Output 10 = DATA18 11 = IISDI

PC1 [3:2] 00 = Input 01 = Output 10 = DATA17 11 = IISDO

PC0 [1:0] 00 = Input 01 = Output 10= DATA16 11 = IISLRCK

//

rPDATC = 0x0001;

rPCONC = 0x5f540554;

rPUPC = 0x0300; /*允许上拉电阻连接到对应脚*/

//PORT D GROUP

PCOND

位名称 BIT 描述

PD7 [15:14] 00 = Input 01 = Output 10 = VFRAME 11 = Reserved

PD6 [13:12] 00 = Input 01 = Output 10 = VM 11 = Reserved

PD5 [11:10] 00 = Input 01 = Output 10 = VLINE 11 = Reserved

PD4 [9:8] 00 = Input 01 = Output 10 = VCLK 11 = Reserved

PD3 [7:6] 00 = Input 01 = Output 10 = VD3 11 = Reserved

PD2 [5:4] 00 = Input 01 = Output 10 = VD2 11 = Reserved

PD1 [3:2] 00 = Input 01 = Output 10 = VD1 11 = Reserved

PD0 [1:0] 00 = Input 01 = Output

10= VD0 11 = Reserved

//

rPDATD= 0x55;

rPCOND= 0xaaaa;

rPUPD = 0x00;

//PORT E GROUP

PCONE

位名称 BIT 描述

PE8 [17:16] 00 = Reserved(ENDIAN) 01 = Output 10 = CODECLK 11 = Reserved

PE7 [15:14] 00 = Input 01 = Output 10 = TOUT4 11 = VD7

PE6 [13:12] 00 = Input 01 = Output 10 = TOUT3 11 = VD6

PE5 [11:10] 00 = Input 01 = Output 10 = TOUT2 11 = TCLK in

PE4 [9:8] 00 = Input 01 = Output 10 = TOUT1 11 = TCLK in

PE3 [7:6] 00 = Input 01 = Output 10 = TOUT0 11 = Reserved

PE2 [5:4] 00 = Input 01 = Output 10 = RxD0 11 = Reserved

PE1 [3:2] 00 = Input 01 = Output 10 = TxD0 11 = Reserved

PE0 [1:0] 00 = Input 01 = Output 10= Fpllo out 11 = Fout out

//

rPDATE = 0x357;

rPCONE = 0x556b;

rPUPE = 0x6;

//PORT F GROUP

PCONF

位名称 BIT 描述

PF8 [21:19] 000 = Input 001 = Output 010 = nCTS1

011 = SIOCLK 100 = IISCLK Others = Reserved

PF7 [18:16] 000 = Input 001 = Output 010 = RxD1
011 = SIORxD 100 = IISDI Others = Reserved
PF6 [15:13] 000 = Input 001 = Output 010 = TxD1
011 = SIORDY 100 = IISDO Others = Reserved
PF5 [12:10] 000 = Input 001 = Output 010 = nRTS1
011 = SIOTxD 100 = IISLRCK Others = Reserved
PF4 [9:8] 00 = Input 01 = Output 10 = nXBREQ 11 = nXDREQ0
PF3 [7:6] 00 = Input 01 = Output 10 = nXBACK 11 = nXDACK0
PF2 [5:4] 00 = Input 01 = Output 10 = nWAIT 11 = Reserved
PF1 [3:2] 00 = Input 01 = Output 10 = IICSDA 11 = Reserved
PF0 [1:0] 00 = Input 01 = Output
10= I^2CSCL 11 =Reserved
//
rPDATF = 0x0;
rPCONF = 0x22445a;
rPUPF = 0x1d3;
//PORT G GROUP
PCONG
位名称 BIT 描述
PG7 [15:14] 00 = Input 01 = Output 10 =IISLRCK 11 = EINT7
PG6 [13:12] 00 = Input 01 = Output 10 = IISDO 11 = EINT6
PG5 [11:10] 00 = Input 01 = Output 10 = IISDI 11 = EINT5
PG4 [9:8] 00 = Input 01 = Output 10 = IISCLK 11 = EINT4
PG3 [7:6] 00 = Input 01 = Output 10 = nRTS0 11 = EINT3
PG2 [5:4] 00 = Input 01 = Output 10 = nCTS0 11 = EINT2
PG1 [3:2] 00 = Input 01 = Output 10 = VD5 11 = EINT1
PG0 [1:0] 00 = Input 01 = Output 10 = VD4 11 = EINT0
//
rPDATG = 0xff;
rPCONG = 0x00ff;
rPUPG = 0x00;
rSPUCR=0x7;
rEXTINT=0x0; /*所有的外部硬件中断为低电平触发*/
}
void Uart_Init(int mclk,int baud) /*串口初始化程序*/
{
int i;
if(mclk= =0)
mclk=MCLK;
rUFCON0=0x0; /*禁止 FIFO*/
rUFCON1=0x0;
rUMCON0=0x0;
rUMCON1=0x0;
/初始化 UART0 相关寄存器/
rULCON0=0x3;
/*8 位数据，1 位停止，无奇偶校验*/

```
rUCON0=0x245;
/*接收为边缘触发，发送为电平触发。禁止超时中断，使能接收中断*/
rULCON0=0x3
;/*8 位数据，1 位停止，无奇偶校验*/
rUCON0=0x245;
/*接收为边缘触发，发送为电平触发。禁止超时中断，使能接收中断*/
rUBRDIV0=( (int)(mclk/16./baud + 0.5) −1 );/*波特率计算*/
/*初始化 UART1 相关寄存器*/
rULCON1=0x3;/*8 位数据，1 位停止，无奇偶校验*/
rUCON1=0x245; /*接收为边缘触发，发送为电平触发。禁止超时中断，使能接收中断*/
rUBRDIV1=( (int)(mclk/16./baud + 0.5) −1 );
}
设定系统主频。代码段如下：
void PllValue_Init(int mdiv,int pdiv,int sdiv)
{
int i = 1;
rPLLCON = (mdiv << 12) | (pdiv << 4) | sdiv; /*rpllcon 为 PLL 控制寄存器*/
while(sdiv--)
i *= 2;
MCLK = (10000000*(mdiv+8))/((pdiv+2)*i); /* 固定计算公式 ,
10000000 为外部
晶振频率*/
}
初始化网络设置等其代码如下:
IP_ADDRESS = IP4_ADDR(192,168,3,100);
MASK_ADDRESS = IP4_ADDR(255,255,255,0);
GATE_ADDRESS = IP4_ADDR(192,168,3,1);
IP4_ADDR()函数将采用 IPV4 协议地址转换为十六进制数并填入一个字中。
设备初始化完成后，输出一些打印信息。提示用户可以进行进一步的操作了。
```

8. 内核映像文件及根文件系统的下载

我们设计编写 BootLoader 的最终目的是要在实验板上运行μCLinux。这就需要我们将 μCLinux 的内核及根文件系统的映像文件下载到我们的实验板上。要完成这样的任务，根据实验板上的硬件接口配置，有 2 种实现途径。

- 利用网口，使用 TFTP 下载
- 利用串口下载

使用网络过程中，由于现阶段没有操作系统的支持，对从主机接收 TFTP 包的一些电子科技大学硕士学位论文处理需要自己编程实现，不但增加了编程的难度也增大了 BootLoader 的体积。而使用串口虽然速度上不及 TFTP 快，但毕竟要传输的文件不大，编程比较容易实现。而且目前许多运行在 PC 上的串口工具都有发送文件的功能。鉴于上述考虑，我们决定使用串口来完成传输内核和根文件系统映像文件的到 SDRAM 的任务，配合一段搬运程序，就可以将它们放到 Flash 中我们要求的位置。其代码段如下。

```
int ComLoad(int argc, char *argv[])
{
int i,size;
unsigned
```

```
char
*buf=(unsigned
char
*)DFT_DOWNLOAD_ADDR;/*DFT_DOWNLOAD_ADDR 为宏定义，默认的目的下载地址为 0x0c008000*/
unsigned char RxTmp[8];
if(argc<2)
/*采用默认的下载目的地址*/
download_addr = DFT_DOWNLOAD_ADDR;
else {
tmp = strtoul((unsigned char*)argv[1]);
download_addr = (tmp= =−1)?download_addr:tmp;
}
buf = (unsigned char *)download_addr;
printf("Now downloadfile from uart0 to 0x%x... \ n",download_addr);
i = 0;
while(i<4)
RxTmp[i++] = getch();/*发送的文件的头部，占 4 个字节*/
i = 0;size = *(unsigned long *)RxTmp - 4;/*发送的文件头部为发送文件的长度*/
while(i<size)
buf[i++] = getch();
download_len = size;
printf("Download File Size = %x \ n", download_len);
puts("Dwonload success \ n");
return 0;
}
```

上面的程序执行完以后，文件是放在以 0x0c008000 开始的 SDRAM 中的，下面的程序段则是负责将文件搬运到 Flash 中，如 SST39VF160。其有 20 根地址线，寻址空间为（2^{20}=1MB），数据位宽为 16 位。当执行编程操作时，会需要检验是否将数据写到了正确的位置。

```
int FlashProg(unsigned int ProgStart, unsigned short *DataPtr, unsigned
int WordCnt)
{
unsigned short ok, count;
unsigned short i, j;
ProgStart += ROM_BASE;/*ROM_BASE 为宏定义,为 0*/
ok = 1;
for( ; WordCnt && ok; ProgStart+=2, DataPtr++, WordCnt--)
{
j = *DataPtr;
CMD_ADDR0 = 0xaaaa;/*完成 flash 编程的必要步骤, CMD_ADDR0、CMD_ADDR1 为宏定义*/
CMD_ADDR1 = 0x5555;
CMD_ADDR0 = 0xa0a0;
*(volatile unsigned short *)ProgStart = j;
count = 10000;
while(count --)/*测试是否正确写入了要写的数据*/
{
i = *(volatile unsigned short *)ProgStart&0x40;
```

```
if(i!=*(volatile unsigned short *)ProgStart&0x40) /*比较数据第 6 位*/
continue;
if((*(volatile unsigned short *)ProgStart)= =j) /*是否为真
实的写入数据*/
break;
}
if(count = = 0) ok = 0;
}
return ok;
}
```

实现了以上两个函数以后，我们的 BootLoader 就具备了最基本的功能，就可以准备加载内核，开始运行操作系统了。

由于嵌入式系统的硬件资源非常有限，所以对内存的使用需要事先进行规划。要使用 µCLinux 操作系统需要两个映像文件：内核映像和根文件系统映像。在启动系统时它们都需要写入内存，所以在对内存进行规划时包括两个方面。

- 内核映像所占用的内存范围，基地址和映像的大小；
- 根文件系统所占用的内存范围，基地址和映像的大小。

说到这里，就需要说说µCLinux 映像文件的执行方式。一种方式是内核映像文件直接在 FLASH 中运行。而另一种方式是先将其解压到 SDRAM 中，再在 SDRAM 中运行。由于在 SDRAM 中的运行速度比 FLASH 快，所以我们将采用后一种方式。对于内核映像，因为嵌入式 Linux 的内核一般都不操过 1MB，所以我们将其拷贝到 SDRAM 地址 0x0c300000 处。这样当开始启动µCLinux 时，压缩的µCLinux 内核将自动解压到 0x0c008000 处开始运行。当然也可以修改为其他的地址范围，但这样做就需要对µCLinux 源代码包的 makefile 文件进行相应的修改，操作起来非常麻烦，所以我们不对它进行改动。

而对根文件系统映像，我们将其拷贝到 0x0c000000+0x00100000 开始的地方。像 ARM 这样的嵌入式微处理器都是在统一的内存地址空间中寻址 Flash 等固态存储设备的，所以从 Flash 上读取数据与从 RAM 单元中读取数据的操作并没有什么不同。完成从 Flash 设备上拷贝映像的工作代码段如下。

```
s = (unsigned char *)prog_s_addr;
/*内核映像在 FLASH 中的起始地址*/
r = (unsigned char *)prog_r_addr;/*搬运到 RAM 中的起始地址*/
size = __ROM_SIZE - prog_s_addr;/*计算映像文件大小，字节*/
for(i=0; i<size; i++)
r[i] = s[i];/*搬运*/
```

9．调用内核

BootLoader 调用 Linux 内核的方法是直接跳转到内核的第一条指令处，对 S3C44B0X 来说也即直接跳转到 0x0c300000 地址处。

当调转后µCLinux 就开始启动。我们的 BootLoader 使用 mrun 命令，设置启动参数,将内核映像从 Flash 拷贝到地址 0x0c300000 处，并开始运行，将控制权交给操作系统，进而启动操作系统。由于 mrun 命令使用的函数调用及宏定义比较多，这里就不给出源代码了。

10．µCLinux 文件系统的生成

首先在 PC 主机上解压 µCLinux-dist-20030522.tar.gz 到/work 目录下，命令如下。

cd /work

tar jxvf µCLinux-dist-20030522.tar.gz

之后会在/work 目录下生成µCLinux-dist 目录。进入该目录，加入 S3C44B0 的在 make config 时的厂商/产品选项。方法如下。

在µCLinux-dist\vendors\Samsung 下新建 S3C44B0 目录，将µCLinux-dist\vendors\ Samsung\4510B 下的内容全部复制到 S3C44B0 目录下。

这里面有以下几个文件需要修改。

“config.linux-2.4.x”是 linux 内核编译配置选项文件。现在针对 S3C44B0 要修改的是#System Type 到 # General setup 之间的内容。修改如下。

```
#
# System Type
#
# CONFIG_ARCH_DSC21 is not set
# CONFIG_ARCH_CNXT is not set
# CONFIG_ARCH_SWARM is not set
CONFIG_ARCH_S3C44B0=y
#指明是处理器类型的是 S3C44B0
# CONFIG_ARCH_ATMEL is not set
CONFIG_NO_PGT_CACHE=y
CONFIG_CPU_32=y
# CONFIG_CPU_26 is not set
CONFIG_CPU_ARM710=y
CONFIG_CPU_WITH_CACHE=y
# CONFIG_CPU_WITH_MCR_INSTRUCTION is not set
CONFIG_SERIAL_S3C44B0=y
#使用 S3C44B0 的串口
DRAM_BASE=0x0c000000
#SDRAM 起始是地址
DRAM_SIZE=0x00800000
#SDRAM 大小
FLASH_MEM_BASE=0x00000000
#FLASH 起始地址
FLASH_SIZE=0x00200000
#FLASH 大小
# General setup
```

以后的 make 命令都以 CONFIG_ARCH_S3C44B0=y 选项来解决是编译和 S3C44B0 相关的其他选项。

这样在 make menuconfig 后在 Vendor/Product 下就可以看到有 Samsung/S3C44B0 的选项了。跟着我们要编译源码得到内核映像文件，具体操作步骤如下。

（1）解压µCLinux-2.4.24.tar.bz2 到/work 目录下。在 Red Hat Linux9.0 桌面下启动终端，输入以下命令。

```
cd /work
tar jxvf µCLinux-2.4.24.tar.bz2
```

之后会在/work 目录下生成 linux-2.4.x 的目录。

（2）进入 linux-2.4.x 目录，执行 make menuconfig，出现配置菜单，移动到 load an alternate configuration file 选项并回车，在文件名输入框输入 kernel_44b0.cfg 后再回车返回主菜单，再选 exit 退出，退出时选 Y 确认保存设置。

（3）执行 make dep 和 make zImage，完成后会在 arch/armnommu/boot 目录下生成压缩内核 zImage。保存该压缩内核到/work 目录下备用，因为在后续的制作 ROMFS 文件系统的过程中，/boot 目录下的压缩内核会被使用。

接下来就是生成 ROMFS 文件系统了，其步骤如下。

（1）用/work 目录下的 linux-2.4.x 替换/work/µCLinux-dist/linux-2.4.x 文件夹。

（2）回到/work/µCLinux-dist 目录：cd /work/µCLinux-dist。输入命令：make menuconfig。选择 Linux-2.4.x 内核及 uclibc。这里我们用 0522dist 来编译生成 ROMFS 根文件系统。

（3）在 user application 配置上选上 boa、ping、console shell、sash、tftp、ppp、http、telnet、busybox 等应用程序。

（4）make dep。

（5）make lib_only。

（6）make user_only。

（7）make romfs。

执行完上述命令后就会在/work/µCLinux-dist 目录下出现 ROMFS 目录，我们需要的 RMOFS 根文件系统的印象文件 ROMFS.img 就保存在里面。

最后通过局域网将压缩内核印象文件及根文件系统印象文件传输到开发板的 SDRAM 中,再由 BootLoader 搬运到 Flash 中。这样就在开发板上建立了µCLinux 环境。

下面给出 S3C44B0X 较完整的 BootLoader 程序。

```
#include "..\inc\44b.h"
#include "..\inc\44blib.h"
#include "..\inc\def.h"
#include "..\inc\option.h"
#include "..\inc\drv\Serial.h"
#include <stdarg.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>
#define STACKSIZE       0xa00 //SVC 堆栈大小
#define HEAPEND         (_ISR_STARTADDRESS-STACKSIZE-0x500) // = 0xc7ff000//
//SVC 栈区域:0xc(e)7ff000-0xc(e)7ffaff//
#define SIO_START    0x08 //SIO 启动
extern char Image$$RW$$Limit[];
volatile unsigned char *downPt;
unsigned int fileSize;
void *mallocPt=Image$$RW$$Limit;
void (*restart)(void)=(void (*)(void))0x0;
void (*run)(void)=(void (*)(void))DOWNLOAD_ADDRESS;
//-------------------------------SYSTEM--------------------------------//
static int delayLoopCount=200;
void Delay(int time)
// time=0: 用看门狗计时器调整延迟函数
// time>0: 循环时间
// 100 微秒
{
    int i,adjust=0;
```

```
    if(time= =0)
    {
        time=200;
        adjust=1;
        delayLoopCount=400;
        rWTCON=((MCLK/1000000-1)<<8)|(2<<3);   //看门狗使能,nRESET,中断禁止 disable
        rWTDAT=0xffff;
        rWTCNT=0xffff;
        rWTCON=((MCLK/1000000-1)<<8)|(2<<3)|(1<<5); // 1M/64,看门狗使能,nRESET,中断禁止
    }
    for(;time>0;time--)
        for(i=0;i<delayLoopCount;i++);
    if(adjust= =1)
    {
        rWTCON=((MCLK/1000000-1)<<8)|(2<<3);
        i=0xffff-rWTCNT;
     delayLoopCount=8000000/(i*64); //400*100/(i*64/200)     //
    }
}
//------------------------PORTS-----------------------------//
void Port_Init(void)
{
    //CAUTION:Follow the configuration order for setting the ports.
    // 1) 设置初始值
     // 2) 设置控制寄存器
     // 3) 配置堆栈寄存器
    //16 位总线结构
    //端口 A
    /*位      9          8          7          6          4          3          2          1          0 */
    /*   ADDR24   ADDR23   ADDR22   ADDR21   ADDR20   ADDR19   ADDR18   ADDR17   ADDR16
    ADDR0*/
    /*       1,         1,         1,         1,         1,         1,         1,         1,         1,
     1*/
    rPCONA = 0x3ff;
    // 端口 B
    /*      位   10        9      8       6        5       4        3        2       1      0*/
    /*/CS5      /CS     /CS3   /CS2   /CS1     nWBE3   nWBE2   /SRAS
    /SCASSCLK   SCKE*/
    /*rtl8019    (Reserve) (Reserve)    FLASH        D12   Out     Out      Sdram     Sdram     Sdram     Sdram*/
    /*   1,          1,              1,          1,          1,    0,        0,          1,         1,
       1,,         1*/
    rPDATB = 0x7ff;
    rPCONB = 0x7cf;
    //端口 C
    //BUSWIDTH=16*/
    /*   PC15        14         13         12         11         10         9          8*/
```

```
    /*    0          0          RXD1      TXD1      0          0          0          0        0*/
    /*Nand-CE      UDA-CE   Uart1     Uart1     NandCLE   NandALE   L3DATA   L3CLK*/
    /*    01         01         11        11        01         01         01         01*/
    /* PC 7     6         5         4         3         2         1         0*/
    /*0         0         0         I         IISCLK    IISDI     IISDO     IISLRCK*/
    /*VD4       VD5       VD6       VD7       [for UDA1341]*/
    /*11        11        11        11        11        11        11        11*/
    rPDATC = 0x3fff;      //All IO is high
    rPCONC = 0x5f55ffff;
    rPUPC    = 0x3000;    //PULL UP RESISTOR should be enabled to I/O
    //端口 D，供 LCD 使用
    /*位 7      6         5         4         3         2         1         0*/
    /*VF        VM        VLINE     VCLK      VD3       VD2       VD1       VD0*/
    /*10        10        10        10        10        10        10        10*/
    rPDATD= 0xff;
    rPCOND= 0xaaaa;
    rPUPD = 0x0;
    //PORT E GROUP
    /*Bit          8         7         6         5         4         3         2     1         0*/
    /*CODECLK   TOUT4TOUT3   TOUT2     TOUT1     TOUT0     RXD0      TXD0      SMRB(I)   */
    /*10           10        10        10        10        10        10        10          00*/
    rPDATE = 0x1ff;
    rPCONE = 0x2aaa8;
    rPUPE    = 0x106;
    //PORT F GROUP
    /*位  8     7         6         5         4         3         2         1         0*/
    /*SIOCLK    SIORxD    7843CS    SIOTxD    [Input(DMA)]  Output    I2CSDA    I2CSCL*/
    /*011       011       001       011       00        00        01        10    10*/
    rPDATF = 0x1fb; //GPF2=0
    rPCONF = 0x1B2C1A; //0x9241A;
    rPUPF    = 0x3;
    //端口 G 设置
    /*位  7         6         5         4         3         2         1         0*/
    /*INT7      INT6      INT5      INT4      INT3      INT2      INT1      INT0*/
    /*11        11        11        11        11        11        11        11*/
    //BIOS 设置
    rPDATG = 0xff;
    rPCONG = 0xffff;
    rPUPG    = 0x0;   //should be enabled
    rSPUCR=0x7;   //D15-D0 pull-up disable
    /*定义非 Cache 区*/
    rNCACHBE0=((Non_Cache_End>>12)<<16)|(Non_Cache_Start>>12);
    /*所有的外部硬件中断为低电平触发*/
    rEXTINT=0x0;
}
/************************ UART ****************************/
```

```
extern serial_driver_t s3c44b0_serial0_driver, s3c44b0_serial1_driver;
serial_driver_t* serial_drv[]={&s3c44b0_serial0_driver, &s3c44b0_serial1_driver};
serial_loop_func_t Getch_loopfunc[]={(serial_loop_func_t)NULL,
                                     (serial_loop_func_t)NULL,
                                     (serial_loop_func_t)NULL,
                                     (serial_loop_func_t)NULL};
#define GETCH_LOOPFUNC_NUM    (sizeof(Getch_loopfunc)/sizeof(serial_loop_func_t))
int Uart_Init(int whichUart, int baud)
{
    if(whichUart>=sizeof(serial_drv)/sizeof(serial_driver_t*))
        return FALSE;
    return serial_drv[whichUart]->init(baud);
}
/*************************************************************
    设置等待串口数据时候的循环函数，
    成功返回函数的序号(删除的时候使用)，
    如果失败则返回-1，
*************************************************************/
int Set_UartLoopFunc(serial_loop_func_t func)
{
    int i;
    for(i=0;Getch_loopfunc[i];i++);
    if(i>=GETCH_LOOPFUNC_NUM)
        return -1;
    Getch_loopfunc[i]=func;
    return i;
}
/*************************************************************
    清除等待串口数据时候的循环函数，
    参数是函数的序号，
    成功返回 TURE，失败则返回 FALSE
*************************************************************/
int Clear_UartLoopFunc(int index)
{
    if(index>=GETCH_LOOPFUNC_NUM || index<0)
        return FALSE;
    Getch_loopfunc[index]=NULL;
    return TRUE;
}
char Uart_Getch(int whichUart)
{
    int i;
    if(whichUart>=sizeof(serial_drv)/sizeof(serial_driver_t*))
     return FALSE;
    while(!serial_drv[whichUart]->poll()){
        for(i=0;i<GETCH_LOOPFUNC_NUM;i++){
```

```
                if(Getch_loopfunc[i])
                        (*Getch_loopfunc[i])();
            }
        }
        return serial_drv[whichUart]->read();
}
//串口是否有数据输入
int Uart_Poll(int whichUart)
{
        if(whichUart>=sizeof(serial_drv)/sizeof(serial_driver_t*))
            return FALSE;
        return serial_drv[whichUart]->poll();
}
//发送缓冲区清空
void Uart_TxEmpty(int whichUart)
{
        if(whichUart<sizeof(serial_drv)/sizeof(serial_driver_t*))
            serial_drv[whichUart]->flush_output();
}
//接收缓冲区清空
void Uart_RxEmpty(int whichUart)
{
        if(whichUart<sizeof(serial_drv)/sizeof(serial_driver_t*))
            serial_drv[whichUart]->flush_input();
}
int Uart_SendByte(int whichUart,int data)
{
        if(whichUart>=sizeof(serial_drv)/sizeof(serial_driver_t*))
            return FALSE;
        return serial_drv[whichUart]->write(data);
}
void Uart_GetString(char *string)
{
        char *string2=string;
        char c;
        while((c=Uart_Getch(0))!='\r')
        {
            if(c= ='\b')
            {
                if((int)string2 < (int)string )
                {
                    Uart_Printf("\b \b");
                    string--;
                }
            }
            else
```

```
        {
            *string++=c;
            Uart_SendByte(0,c);
        }
    }
    *string='\0';
    Uart_SendByte(0,'\r');
    Uart_SendByte(0,'\n');
}
int Uart_GetIntNum(void)
{
    char str[30];
    char *string=str;
    int base=10;
    int minus=0;
    int lastIndex;
    int result=0;
    int i;
    Uart_GetString(string);
    if(string[0]= ='-')
    {
        minus=1;
        string++;
    }
    if(string[0]= ='0' && (string[1]= ='x' || string[1]= ='X'))
    {
        base=16;
        string+=2;
    }
    lastIndex=strlen(string)-1;
    if( string[lastIndex]= ='h' || string[lastIndex]= ='H' )
    {
        base=16;
        string[lastIndex]=0;
        lastIndex--;
    }
    if(base= =10)
    {
        result=atoi(string);
        result=minus ? (-1*result):result;
    }
    else
    {
        for(i=0;i<=lastIndex;i++)
        {
            if(isalpha(string[i]))
```

```
                {
                    if(isupper(string[i]))
                        result=(result<<4)+string[i]-'A'+10;
                    else
                        result=(result<<4)+string[i]-'a'+10;
                }
                else
                {
                        result=(result<<4)+string[i]-'0';
            }
        }
        result=minus ? (-1*result):result;
    }
    return result;
}
void Uart_SendString(char *pt)
{
    while(*pt){
    if(*pt= ='\n')
        Uart_SendByte(0,'\r');
    Uart_SendByte(0,*pt++);
    }
}
//如果不使用 vsprintf(),代码量将减少
void Uart_Printf(char *fmt,...)
{
    va_list ap;
    static char string[256];
    va_start(ap,fmt);
    vsprintf(string,fmt,ap);
    Uart_SendString(string);
    va_end(ap);
}
/************************* Timer ********************************/
void Timer_Start(int divider)    //0:16μs,1:32μs 2:64μs 3:128μs
{
    rWTCON=((MCLK/1000000-1)<<8)|(divider<<3);
    rWTDAT=0xffff;
    rWTCNT=0xffff;
    // 1/16/(65+1), nRESET & interrupt    disable
    rWTCON=((MCLK/1000000-1)<<8)|(divider<<3)|(1<<5);
}
int Timer_Stop(void)
{
    rWTCON=((MCLK/1000000-1)<<8);
    return (0xffff-rWTCNT);
```

```
}
/************************* PLL ********************************/
void ChangePllValue(int mdiv,int pdiv,int sdiv)
{
	rPLLCON=(mdiv<<12)|(pdiv<<4)|sdiv;
}
/************************* General Library *********************/
void Cache_Flush(void)
{
	int i,saveSyscfg;
	saveSyscfg=rSYSCFG;
	rSYSCFG=SYSCFG_0KB;
	for(i=0x10004000;i<0x10004800;i+=16)
	{
		*((int *)i)=0x0;
	}
	rSYSCFG=saveSyscfg;
}
void init_SIO()
{
	/*7		6	5	4	3		2	1	0 */
	/*Internal clock,MSB mode,Transmit/Receive mode,rising edge??,No action,Non hand-shaking mode,SIO interrupt mode*/
	/*0		0	1	0(not 1) 0		0	0	1*/
	rSIOCON=0x21;
	rSBRDR=15;//band rate = 60MHz/2/(15+1)=1.875MHz
	rIVTCNT=7;//Intervals=60MHz/4/(7+1)=1.875MHz
}
unsigned char ReadSIOData()
{
//	while(rSIOCON&SIO_START);//等待发送
	return rSIODAT;
}
void SendSIOData(unsigned char data)
{
//	while(rSIOCON&SIO_START);//等待发送
	rI_ISPC=BIT_SIO;
	rSIODAT=data;
	rSIOCON|=SIO_START;
	while(!(rINTPND&BIT_SIO));
	rI_ISPC=BIT_SIO;
}
/*****************************************************************
	SIOTXD 和 SIORXD 通过两个三态门连接，组成了 SDIO 的双向接口，
	通过 GPF2 来控制三态门的开启，从而控制 SDIO 的方向，
	对于处理器，GPF2=0 的时候为输出，GPF2=1 的时候为输入
*****************************************************************/
```

```
#define SDIO_CTRLIO                (0x4) //GPF2
#define SETSDIO_OUT()              (rPDATF&=(～SDIO_CTRLIO))
#define SETSDIO_IN()               (rPDATF|=SDIO_CTRLIO)
unsigned char ReadSDIO()
{
    SETSDIO_IN();
    SendSIOData(0);
    SETSDIO_OUT();
    return rSIODAT;
}
```

DS 的调试软件 AXD 调试应用程序。

联系方式