摘　要

随着科学技术的进步和国民经济的发展，人们对于家庭安全，居住舒适，生活便捷等方面的要求越来越高。本论文正是在顺应了人们这些需求之下，以嵌入式 linux2.6 为操作系统，以 ARM9 CPU 为硬件基础，设计并实现了一个由中央处理器集中控制和管理的家庭网络系统。该系统向互联网提供服务器功能，不仅可以通过互联网终端上的 IE 浏览器网络控制和访问连接在家居网络中所有的电器设备（如照明，家电，温湿度监控，空气监测，视频监控等），还实现了对家居温湿度环境和室内空气质量情况的实时监测。

本文是在分析了各种家居网络监控产品设计方案的优缺点，并对各种实现技术进行了深入研究之后，研究并开发了一个基本可以满足用户需求的嵌入式智能家居网络监控系统，主要研究内容如下：

1. 给出了一个完整的 ARM-嵌入式 Linux 智能家居网络监控系统设计方案，并从硬件和软件两个方面对整个系统的方案选型（包括嵌入式处理器的选型，硬件平台的搭建，嵌入式操作系统的选型，整个系统的软件架构设计）和确定都做了详细的论证。

2. 在 U-Boot 源代码的基础上，根据具体的硬件配置重新设计了与 S3C2440 CPU 体系结构紧密相关的代码，实现了 NANDFLASH 和 NORFLASH 双启动方式。添加了 USB 下载功能，修改了 DM9000 网卡驱动。添加了 NANDFLASH 驱动，实现了烧写任意字节大小的的内核和 YAFFS2 文件系统的功能。实现了直接引导 zImage 格式的内核镜像的功能。

3. 对各个硬件模块的控制原理都进行了详细的分析，开发了电灯驱动程序，直流电机驱动程序，步进电机驱动程序，温湿度监控驱动程序以及空气监测驱动程序，并对驱动程序开发所涉及到的如内核定时器，中断处理，阻塞型输入输出，并发控制等技术及设计技巧都进行了研究。最后对各个模块驱动的设计都做了测试和验证。

4. 在嵌入式 WEB 服务器的基础上，使用 CGI+HTML 技术为整个系统开发了一整套应用程序，并在实验室环境下对该系统进行了整体测试。测试结果表明，该系统完成了预期功能，具有良好的稳定性和可靠性，易于使用。相比于同类嵌入式家居网络监控系统方案，在软硬件成本和开发周期方面有一定的优势。

关键词：智能家居，网络监控，嵌入式 Linux，BootLoader，设备驱动程序

ABSTRACT

With the progress of science and technology and the development of national economy, the demand of people for home security, comfortable living, convenient living and other aspects is higher and higher. In this dissertation, the designed intelligent home monitoring system that came into being is under conforming to the demand of people. It is embedded linux2.6 operating systems, embedded ARM9 CPU processor hardware, designed and implemented a home network system which centralized control and managed by a central processing unit. The system server functionality to the Internet, not only could through Internet terminals on the IE browser network control and access to all electrical equipment which is connected to the home network ((such as lighting, household appliances, temperature and humidity monitoring, air monitoring, video surveillance, etc.), but also achieved real-time monitoring of the home temperature and humidity and indoor air quality.

This dissertation analyzes the advantages and disadvantages of the various smart home network monitoring product design, and have made a lot of technology-depth research, research and development to meet the basic user needs embedded intelligent home network monitoring system, the main contents as follows:

1. Presented a complete ARM-embedded Linux smart home network monitoring system design, have made a detailed argument for the whole system’s selection (including embedded processor selection, hardware platforms, structures, selection of embedded operating systems, software architecture design of the entire system)and determination in terms of both hardware and software.

2. On the basis of the U-Boot source code, depending on the hardware configuration, Re-designed the first phase of the code which is closely related to the S3C2440 CPU architecture to achieve a NANDFLASH and NORFLASH dual boot. Add a USB download function, and modified and optimized the DM9000 network card driver to have been solved a network download incomplete and re-transfer phenomenon. Modify the NANDFLASH driver to programming any size in bytes of the kernel and the YAFFS2 file system, and solve the programming speed is too slow. Finally ,realized

the function of Boot zImage format kernel image.

3. Carried out a detailed analysis of the control principle of the various hardware modules , developed lamp driver , DC motor driver, stepper motor driver, temperature and humidity monitor driver, as well as air monitoring driver, have been made a lot of in-depth research which driver development process involved, such as kernel timers, interrupt handling, blocking input and output, and concurrency control theory and technology, and design skills. Finally, make a lot of rigorous testing and validation for the results of each module driven design

4. On the basis of the embedded WEB server, using the CGI + HTML to develop a set of application for the entire system, then made an overall test for this system in the lab environment. The test results show that the system have achieved the desired function, and with good stability and reliability, and ease of use. Compared to similar embedded home network monitoring system, there are certain advantages in hardware and software costs and development cycle.

Keywords: smart home, network monitoring, embedded Linux, BootLoader, device drivers.

目　录

图目录

表目录

第一章 绪论

1.1 研究背景及意义

近年来，随着人民生活水平的提高和科学技术的进步，特别是互联网技术、嵌入式技术、电子通信技术、自动控制技术的日新月异和迅猛发展，大大促进了社会信息化网络化的加速。在信息化网络化的浪潮中，人们的生活习惯与工作方式也在悄然发生着变化。住宅自动化，网络化，智能化、可随时随地远程控制化也逐渐成为了社会的迫切需求，智能家居网络监控系统正是在适应了人们的这些需求之下应运而生。

当前，以 ARM[1]为代表的 32 位嵌入式处理器正经历了前所未有的发展，处理速度和控制能力都在朝着 X86 SOC 的方向发展，甚至已经越来越接近桌面电脑；而功耗却越来越低，价格也越来越便宜。未来智能家居监控装置是否可以长期有效和不间断地稳定运行，并且在性价比方面能否吸引更多的客户，使用超低功耗的嵌入式处理器设计实现的智能家居网络监控产品发展空间很大；又因为互联网的介入彻底改变了传统智能家居产品的设计和应用。因此采用面向特定应用，面向互联网，在功能、可靠性、成本、体积、功耗都占有绝对优势的 S3C2440 处理器设计的智能家居网络监控系统将大有可为。

同时在软件方面，以嵌入式 Linux[2]为代表的嵌入式操作系统技术正迅速崛起，主要是由于人们对自由软件的渴望与嵌入式系统应用的特制性，要求提供系统源码层次上的支持，而嵌入式 Linux 正适应了这一需求，相比其它几种商用操作系统(如 WindowsCE，VxWorks 等)，它不仅具有免费开源、系统内核小、效率高、内核网络结构完整等特点，而且在开发工具成熟，可以自由裁剪，更容易得到技术支持等方面的优势也非常明显。因此使用嵌入式 Linux 操作系统是降低产品成本和缩短开发周期最重要的因素。毫无疑问，嵌入式 Linux 是最适于开发性价比更高的智能家居网络监控设备的操作系统。

总而言之，使用 S3C2440 ARM 主控制器和嵌入式 Linux 操作系统开发的嵌入式智能家居网络监控系统是新世纪新时代人们的迫切需求，是家居监控产品设计技术未来发展的必然趋势。加上当前互联网的大力普及和应用，尤其是国家大推动的物联网建设和三网融合战略[3]，更为家居智能化网络化的建设提供了坚强的技

术手段和大的发展环境，家居智能化网络化的建设也会因此变得更加容易和方便。相信使用嵌入式 ARM-Linux 技术开发的家居监控产品必将凭借自身个性化的独特魅力，赢得更加广阔的市场前景，在不久的将来，使用该技术设计的智能化网络化的家居监控产品必将普及每个家庭。

1.2 国内外研究现状

近几年以来，面对智能家居行业的广阔市场，一些欧美发达国家和日本都纷纷研制出了自己的智能化网络化的家居监控产品，其中比较有代表性的如美国霍尼韦尔公司最新推出的智能网络家居监控系统[4]，该系统采用了嵌入式硬件平台和嵌入式 Linux 操作系统，可以保证系统长期稳定运行，不会死机，不会受到网络病毒的攻击，所以整套系统的可靠性极高，网络故障的概率几乎为零。但它是通过在家庭中设置的主控制面板对各种设备进行集中控制。并且该产品只是在一定的范围内实现远程监控，所以广泛使用了专用的控制线路和信号传输协议，虽然突破了一定的距离限制，但是随着距离的增加，硬件成本也就大幅度上升，因此没有实现真正的网络监控。

中国自主研发的智能家居监控系统中比较有影响力的主要有：海尔公司的智能家电监控系统[5]，它通过手机向家庭设备发送短信就可以实现对家中的灯光，窗帘，电视机等设备进行远程控制。此外，还实现了对房屋周围的安全布防，一旦家中有意外入侵事件发生的时候，监控系统就会自动拨打就近的安保人员的电话报警，并在第一时间将情况通知给家庭成员。但因为家庭中各种独立的设备还没有集成在一个由中央处理器集中控制和协调的家庭网络系统中，没有向互联网提供服务器功能，也必然不能通过以太网远程控制。

论文《基于嵌入式的智能家居系统研究》[6]提出并实现一套完整的智能家居网络监控系统设计方案，并使用嵌入式处理器和 WindowsCE 操作系统为整个智能家居系统设计了网络控制终端，并把控制终端作为整个智能家居网络指挥中心，接收来自各种家居设备采集过来的电信号，然后对接收到信号综合分析之后再向有关设备发出控制命令，从而实现对家居设备的网络控制。使用 WindowsCE 设计的产品自然免不了要向软件巨头 MICROSOFT 公司交付授权费用，更大的问题是使用这种方案设计的产品由于控制终端由作者专门设计，这样对于用户想要随时随地监控自家设备来说就必须随身携带配套的控制终端登录到控制软件界面才能控制和访问家居设备，一方面操作过程相当繁杂，违背了智能家居系统简单易用的

本性。另一方面相对于使用通用的 IE 浏览器作为控制终端来说无谓的增加了软硬件成本。

论文《智能家居网络监控系统的研究与实现》[7]使用了嵌入式 ARM 和嵌入式 Linux 作为家庭网络的控制中心，并且向互联网提供服务器功能，它接收并接些来自互联网网络控制终端的命令，然后通过 ZigBee 技术发出指令到相应设备(每个设备都被看成是一个 ZigBee 子节点)做出智能化动作。这种方案不仅解决了以往各设备厂商都按照不同的接口标准与协议生产的设备之间相互连接相互通讯困难的问题。也为解决过去各厂商生产的设备没有一个标准的，统一的通讯协议，而使得用户在选择配套产品时不可能如同组装电脑那样有多样化的可选配件的问题指明了道路。不过方案必须在每一个子节点上都连接一个 ZigBee 模块，也就是说它是以提高硬件成本为代价来解决问题的。

因此以 ARM 为中央处理器，以嵌入式 Linux 软件平台，采用客服端+服务端方式开发家居网络监控系统，使用现成的互联网作为通信线路，将通用的浏览器作为网络控制终端的操作界面是比较好的解决方案。这样任何人只要随身携带一个上网设备就可以轻松的监控到家庭中的设备。同时对于产品研发来说，采用 ARM-Linux 技术，可以得到 Linux 社区众多爱好者的技术支持，这样更利于攻克技术上的难题，从而降低技术门槛并缩短产品的研发周期，自然也就降低了产品的成本。

1.3 本课题研究内容

本文通过详细的系统设计过程（包括 BootLoader 设计、嵌入式 Linux 内核定制和移植、各硬件接口设备驱动程序开发、系统应用程序设计），系统研究了基于嵌入式 Linux 操作系统和 ARM9 S3C2440 嵌入式开发平台为运行环境的智能家居网络监控机的设计与实现。具体章节安排如下：

第一章详细介绍了本课题的研究意义和国内外智能家居网络监控系统的研究现状。

第二章分析并给出了一个完整的 ARM-嵌入式 Linux 智能家居网络监控系统解决方案，从硬件和软件两个方面对整个系统方案的选型（包括嵌入式处理器的选型，硬件平台的搭建，嵌入式操作系统的选型，整个系统的软件架构设计）和确定都做了详细的论证。

第三章深入研究了适应本系统 BootLoader 移植过程中所要解决的关键问题及

所使用的关键技术，同时给出了关键代码的设计步骤和理论依据。最后对BootLoader 程序的设计结果进行了严格的测试和验证。

第四章详细介绍了嵌入式 Linux 在系统主板上的移植步骤和根文件系统制作过程。

第五章对各个硬件模块的硬件设计原理进行了详细分析，深刻剖析了嵌入式Linux 设备驱动程序开发中所涉及的复杂理论，以深入浅出的方式介绍了各个设备模块的驱动程序开发过程中关键代码的设计原理和步骤。同时给出了每个模块驱动的测试结果。

第六章在前面移植的 WEB 服务器基础上，使用了 CGI+HTML 技术开发了一整套的应用程序，并对整个智能家居网络监控系统的性能进行了测试。

第七章结论与展望，对论文所做工作进行了总结，客观分析了本次系统设计存在的不足，并指出了下一步尚待开展的研发工作。

第二章 系统总体设计及方案确定

2.1 引言

采用客户端+服务端方式开发的智能家居网络监控系统，只要随身拥有一个上网设备（如手机，上网本，笔记本电脑）就可以轻松的使用 IE 浏览器登陆到系统的主页面上控制和管理连在服务器上的任何设备。真正实现了在任何时候任何地点都可以监控家中发生的一切。另外使用现成的互联网作为网络控制终端和家居网络监控系统的通信线路，不但避免了很多技术攻关和产品稳定性难题，也可以降低软硬件的研发成本。

2.2 系统设计目标

图 2-1 是本论文研究的整个家居网络监控系统的结构示意图。分为客户端和服务端两部分，客户端为可以连接到互联网并且可以使用 IE 浏览器浏览网络信息的任何设备。服务端是由一个中央处理器集中控制和管理的家庭网络系统。它接收来自远程控制终端的信息，并对接收到信息进行综合分析之后，进而发出指令到相应模块做出智能化的动作，最后将结果返回给控制终端

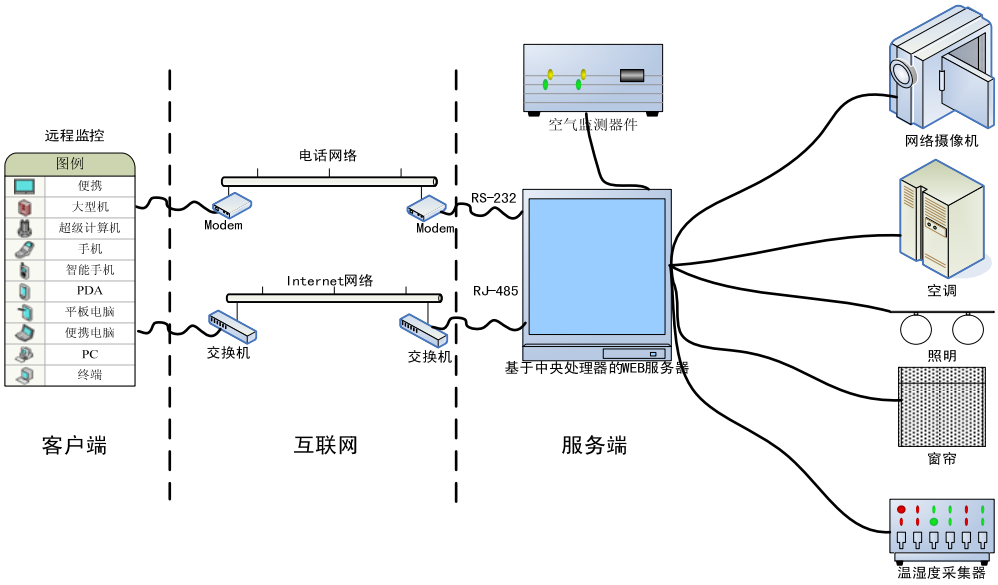

图 2-1 系统整体结构示意图

具体要实现的功能主要有：

（1）网络视频监控。

智能家居网络监控设备的一大卖点就是为客户提供安全的居住环境，其所集成的视频监控系统能够综合监控家里的一切，给客户提供了极大的便利。本次设计的视频监控模块是根据用户需要，可以在门外，阳台等区域安装网络摄像机，这样用户可以随时随地通过网络控制终端查询相关区域的图像，家中无人时也可以通过网络查询家中的状态。

（2）网络家电控制。

通过网络网络控制家庭中的电器设备可以让用户可以更方便的来管理和控制家庭设备，比如在上班的时候可以通过手机、笔记本、上网本打开家里的电器设备的空调，调节好室内的温湿度环境；并可以实现对每个房间的的灯光的亮度调节，开关控制等，还可以实现对窗帘的自动控制。真正实现在任何时候任何地点都可以监控家中发生的一切。

（3）温湿度环境监控，室内空气质量检测。

温湿度监控和空气质量监测对实时调控家居环境有着非常重要的意义。比如当控制终端显示当前室内温湿度环境超过主人舒适范围的时候，可以在回家之前将家庭温湿度环境调节到一个合理的范围。又如在出差或旅游途中，可以在能上 Internet 的任何地方，了解到家庭中的空气情况从而可以打开吸收器吸除掉空气中的有害的物质。

系统可以达到的性能指标主要以下几点：

（1）有较快的处理速度和较强数据处理能力。由于系统需要向互联网提供服务器功能，因此必须可以运行操作系统，能够分析计算复杂的数据，当客户端向服务端发出控制请求的时候，系统必须可以迅速执行其功能，因此对 CPU 的处理能力有很高的要求。

（2）有良好的稳定性和可靠性。监控系统必须能够长期有效和不间断地稳定运行。

（3）功耗低。功耗越低，意味着在同等供电能力的情况下，系统可以运行的时间就越长。由于嵌入式系统不可能使用大功率的电源，故功耗是必须要参考的性能指标之一。

（4）成本低，开发周期短，系统操作界面友好，易于使用，容易扩展，方便升级。

2.3 系统硬件方案选型

2.3.1 主控制器选型

在嵌入式系统中所有任务和功能的真正执行者是嵌入式微处理器[8]，它的性能是衡量整个嵌入式系统性能好坏的重要指标。在众多的嵌入式处理器中， 基于 ARM 体系架构处理器是现在应用最为广泛的一种微处理器，是嵌入式应用领域公认的处于领先地位的 32 位 RISC 处理器。 他在高性能和低功耗方面具有明显的优势。由 SAMSUNG 公司生产 S3C2440A 控制器[9]是 ARM 系列处理器中的佼佼者，该处理器基于 ARM920 内核，主频高达 405MHz，指令执行速度达到 1.1MIPS/MHz，带 MMU(内存管理单元)，性价比极高，片上资源丰富：集成了 NANDFLASH 控制器，UART，DSP，USB 等功能部件；同时支持大/小端模式存储字数据，8 个 BANK，每个 BANK 为 128 字节，因此其寻址空间达到 1G 字节，对于外部 IO 设备的数据宽度可以达到 8/16/32 位；支持 NORFLAH/NANDFLASH，EEPROM 等多种 ROM 启动方式。同时在 Linux 操作系统中有较完善的驱动支持。用此选择 S3C2440A 做为本系统的主控制器。

2.3.2 系统硬件平台选型

为了加快开发进度，减少硬件开发成本，本次系统设计是以在实验室已有的 2440 实验板上开发所有的软件，调试稳定后，再设计和生产系统所需要的硬件。

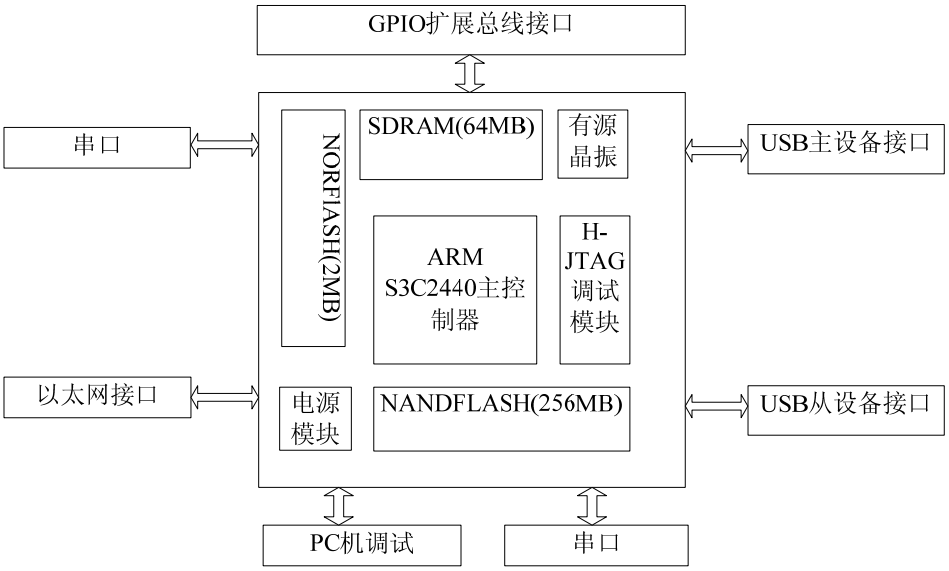

图 2-2 系统主板的硬件配置图

如图 2-2 是该开发板的硬件配置图。是一款功能强大，功耗极低的 ARM9 嵌入式开发板。其实质上是在一个最小系统板外扩了几个项目所需要的子模块，它包含最基本的电源电路(5V 供电)、复位电路、标准 JTAG 调试口、用户调试指示灯、以及核心的 CPU 和存储单元等。其中 FALSH 存储单元包 NAND FALSH 和 NORFALSH 两种类型，通过跳线可以选择从 NAND 或 NOR 启动系统。一般 NOR FALSH 里面放置的是不经常更改的 BootLoader， NAND FALSH 里面则烧写完整的系统程序(BootLoader、内核、文件系统等)。

具体硬件指标如下：

（1）主控制器为 S3C2440A，CPU 主频可以通过软件设置，最高可达到 533MHz。

（2）内存采用外挂 SDRAM。64M 大小，32bit 总线宽度。由两块两片 16 位的 K4S564632C-TC75 芯片并联组成。时钟频率高达 100MHz 。

（3）FALSH 存储器。板载 256M 的 NAND FALSH， 芯片型号是 K9VF2G08，掉电非易失；板载 2M NOR FALSH，芯片型号为 EN29LV160AB-70TCP，掉电非易失，用于安装 BootLoader 程序。

（4）在板复位电路。采用专用的低功耗复位芯片 MAX811，当复位键按下时间持续 140ms 以上时，就会输出一个脉宽为 200ms 的低电平复位信号使系统复位。

（5）在板 10Pin 2.0mm 间距 JTAG 调试接口。用于下载第一个程序（也就是 BootLoader 程序）到 FALSH 存储器中，以后都是在 BootLoader 的引导下下载内核和文件系统。

（6）系统时钟源。主板使用 12MHz 的外部晶振为系统提供时钟，通过设置 PLLCON 和 CLKDIVN 寄存器就可以确定 CPU 的工作时钟和 FCLK，HCK，PCLK 的比例关系。

（7）+5V，2A 的直流电源供电 。

（8）接口和资源。2 个 USB 接口，其中一个充当从设备的接口，用于在 BootLoader 的引导下从宿主机下载内核和文件系统，还有个充当主设备接口，用于连接 USB 外部设备（如 USB 摄像头）；1 个 TTL 电平的 9 孔串口通过电平转换之后用于在研发阶段和宿主机上的控制台通信；一个以太网接口在研发阶段用于从宿主机上下载内核和文件系统；系统运行之后，还需要通过以太网接口和互联网联系起来。

（8）扩展板接口。是 1 个 40 Pin 2.0mm 间距 GPIO 接口，将大部分常用的 GPIO 接口单独引出，在智能家居网络监控系统用于和扩展板连接。

2.3.3 系统模块扩展

为了模拟智能化的家庭居住环境，系统研发阶段通过扩展板总线接口连接了 5 个大的模块：两个发光二极管用于模拟房间客厅的电灯控制，一个直流电动机用于模拟网络家居中的空调控制，一个步进电机用于模拟网络家居中的窗帘控制，一个温湿度传感器用于实时采集和传输室内温湿度数据，一个气体传感器用于实时检测室内空气情况。并将所有的模块通过 PCB 引线连接到一个扩展板总线接口上集中控制，然后用 40PIN 排线将扩展板连接到主板上的 GPIO 口，扩展板总线接口和主板的电路连接如图 2-3 所示。未用到的引脚用于继续扩展其他模块。各个模块的电路图在第五章设备驱动程序开发中给出。

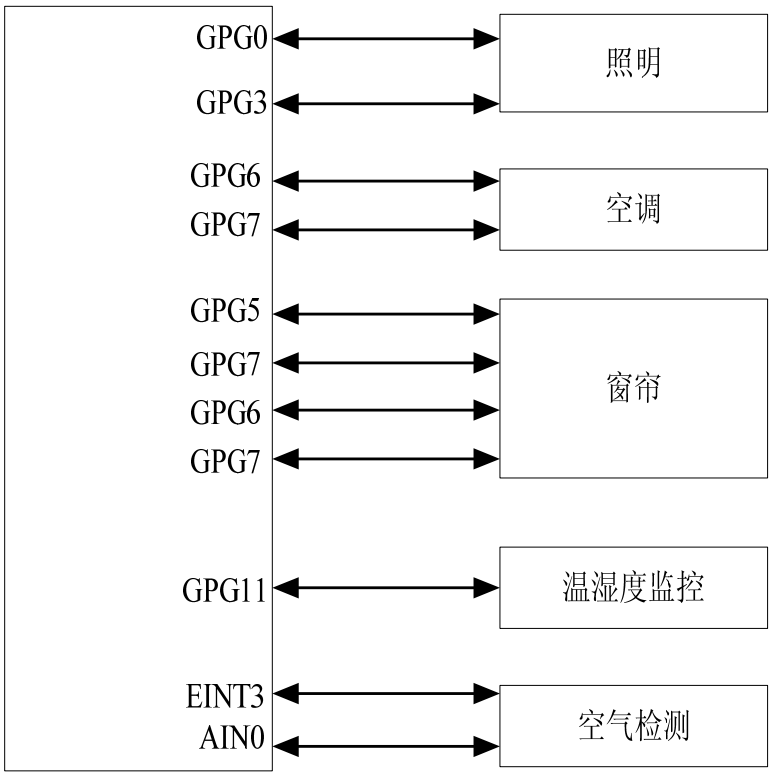

图 2-3 扩展板总线接口图

2.3.4 系统硬件方案确定

系统硬件框架由主板和扩展板组成，主板主要包括一个 S3C2440ARM 主控制器，是整个系统的运行中枢，是系统命令的真正执行者是一个 ARM 嵌入式系统运行所需要的最基本条件。扩展板主要包括一些外围电路，用于模拟智能化的家庭居住环境。总体硬件框架如图 2-4 所示。

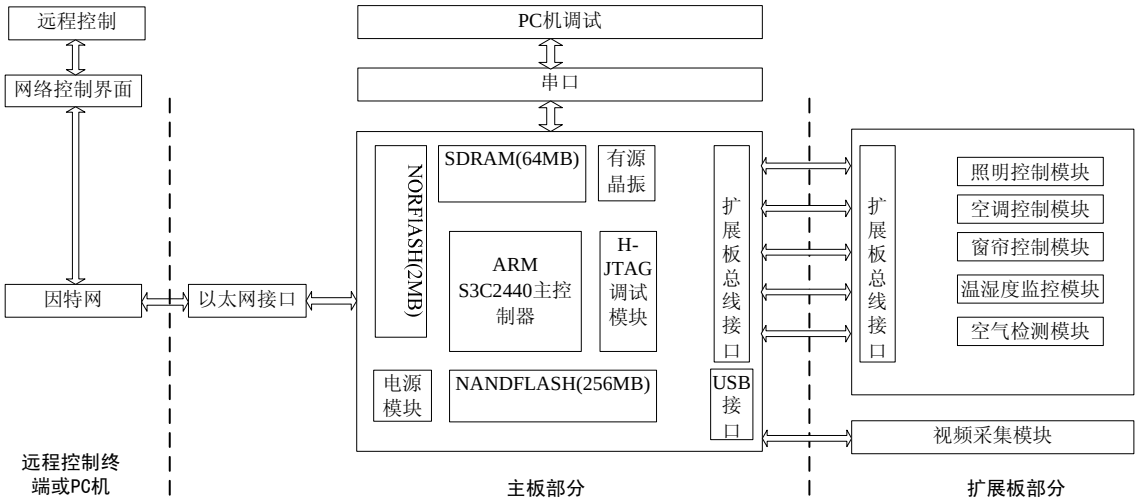

图 2-4 系统硬件结构框图

2.4 系统软件方案选型

2.4.1 嵌入式操作系统选型

嵌入式操作系统有多种，但大体上分为两种：商用型和免费型。目前商用型操作系统主要有 Vxworks，WindowsCE，pSOS，Palm OS[10]等，它们的优点是功能稳定，可靠，有完善的技术支持和售后服务，而且提供了如图形用户界面和网络支持等高端嵌入式系统要求的许多高级功能；缺点是价格昂贵且源码具有封闭性，这对开发者来说是个很大的阻力。

免费型操作系统主要有嵌入式 Linux，它不但在可靠性和稳定性方面毫不逊色于商用操作系统，而且在价格方面有很大的优势，正以价格低廉，功能强大，易于移植和开发，且源码完全公开等优点成为嵌入式操作系统中的新兴力量。归纳起来，使用嵌入式 Linux 作为本次系统设计的软件开发平台，主要有以下几点优势：

（1）完全免费开源，但软件资源丰富。有效降低产品成本对成本敏感的嵌入式系统来说至关重要，Linux 恰好具有这一特性。但 Linux 上的软件资源十分丰富，所需每一种通用程序在 Linux 上都可以找到。

（2）功能强大、性能高效、稳定，多任务，但大小、功能都可定制。Linux 的内核非常稳定，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。但 Linux 内核小巧灵活，非常容易裁减，很适合嵌入式系统的应用。

（3）对 ARM 体系架构的 CPU 有很好的支持，稍作修改就可以移植监控系统

的硬件平台上来，而且支持大量外部设备，驱动非常丰富。

（4）完善的网络通讯、图形、文件管理机制。Linux 自产生之日起就与网络密不可分，网络是 Linux 的强项。另外，Linux 还支持多种文件系统和图形系统。

（5）良好的开发环境，不断发展的开发工具集。 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,其中很多是免费的，大大降低了开发费用。

（6）件开发者的广泛支持。Linux 的自由精神吸引了成千上万的程序员投入到 Linux 的开发和测试中来，这使得 Linux 开发变得非常更加容易和方便。

由此，我们有理由相信：嵌入式 Linux 是最适于开发更高性价比的智能家居网络监控设备的操作系统。

2.4.2 系统软件方案确定

在嵌入式系统发展的早期阶段，嵌入式开发并没有使用操作系统的概念，所有的控制程序都是根据特定的体系结构用特定的汇编语言来实现。因此程序的兼容性、通用性和可移植性都很差，这样就会造成软件的开发周期难以预测的。引入嵌入式 Linux 操作系统之后，整个嵌入式系统就分成了硬件层，驱动层和用户层，从而使得软件开发有了层次化的设计思想，一般分为：BootLoader 移植，Linux 内核移植，应用程序开发。

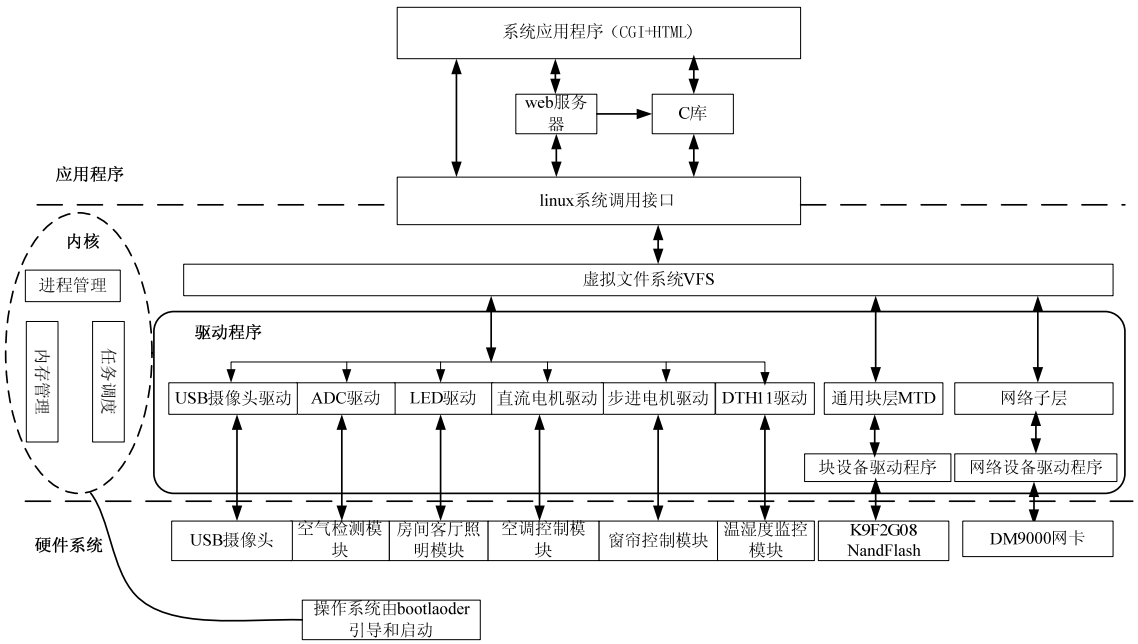

图 2-5 系统软件结构图

整个系统的软件架构如图 2-5 所示。其中 BootLoader，Linux 内核， Linux 设备驱动属于嵌入式底层软件的范畴，是和硬件紧密联系在一起的，正因为有了底层软件的存在，屏蔽了底层硬件的差异，使得上层应用程序的开发完全不必理会底层硬件的具体实现，使得不同体系架构的应用程序，只要所运行的操作系统相同，并且有相关的硬件支持，基本上不需要任何修改就可以运行在另外一种体系架构的硬件平台上，因此使用嵌入式操作系统来构建嵌入式软件平台之后更容易开发出功能复杂的系统，从而加快嵌入式产品的开发时间并降低研发成本。

2.5 系统整体方案确定

根据以上分析，最终选择了以三星公司的 S3C2440A 为主控制器的嵌入式 S3C2440 开发板做为主要的硬件平台，并在此基础上扩展了照明控制模块，空调控制模块，窗帘升降模块，温湿度监控模块，气体检测模块和视频采集等硬件模块。软件方面，选择嵌入式 Linux 操作系统作为软件开发平台，移植和裁剪适应监控系统的 Linux 内核，并在基础上设计各个硬件模块的设备驱动程序和 CGI 应用程序。

2.6 小结

本章给出了以 S3C2440 ARM 为主控制器，以嵌入式 Linux2.6 为操作系统的嵌入式智能家居网络监控系统的总设计方案，并给出硬件和软件的完整设计框架和总体开发流程。其中重点研究了所选方案中嵌入式处理器的选型，硬件平台的搭建，以及整个系统的软件架构设计中所牵涉到的理论依据和设计原理。

第三章 系统 BootLoader 移植

3.1 引言

在 ARM 体系结构的嵌入式系统中，通常需要一小段引导代码，在系统上电复位之后先对相关的硬件设备进行初始化，然后建立起内存空间映射，为系统运行准备好软硬件环境；再将操作系统从 FALSH 拷贝到内存中，并将启动参数传递给内核，最后跳转到操作系统的起始地址处启动操作系统。这一小段引导程序通常被称作为 BootLoader，也即引导加载程序。BootLoader 的基本功能[11]如图 3-1 所示。

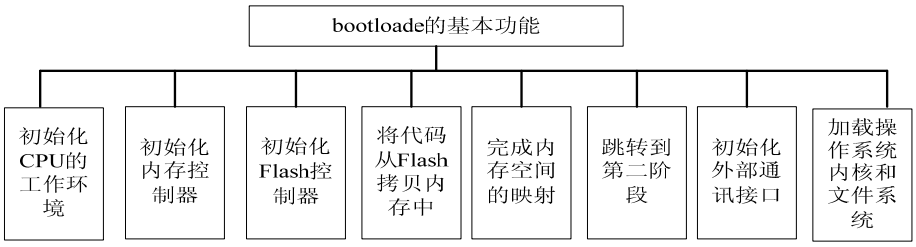

图 3-1 BootLoader 的基本功能

另外，在嵌入式产品开发阶段，BootLoader 除了要引导操作系统之外，还必须在 BootLoader 的引导下才能将 Linux 内核和文件系统烧写到 FALSH 中永久保存；在下一次启动的时候又必须先由 BootLoader 将 Linux 内核读到 RAM 中去执行。总而言之，BootLoader 的主要功能是设置并初始化硬件设备，终极目的是引导并加载操作系统；没有 BootLoader，不但无法将 Linux 操作系统和文件系统烧写到嵌入式硬件设备中去，嵌入式操作系统也永远不可能在嵌入式硬件设备上成功运行。

BootLoader 的开发一个是一项非常复杂的工程，本次系统设计是在免费开源的 U-Boot 的基础上，根据自己的硬件平台做相应的修改和优化，裁剪不要的代码并添加几个自己的驱动程序，从而得到一个适应本系统的专用 BootLoader 程序。

3.2 U-Boot 在网络监控系统上的移植

系统上电复位之后，CPU 会找到复位异常向量[12]，然后执行复位异常向量处

的跳转指令，跳转到复位异常处理程序中开始执行下一条指令，至此 U-Boot 进入第一阶段的启动任务。第一阶段的代码做了很多的工作，目的是为了完成 U-Boot 代码从 FALSH 器到 SDRAM 的搬移，并且为第二阶段准备好 C 运行环境，然后跳转到 SDRAM 中的 C 入口处继续执行第二阶段的代码。

当 U-Boot 的 start.S 运行到"_start_armboot:.word start_armboot"时，就会调用 lib_arm/board.c 中的 start_armboot 函数，至此 U-Boot 进入第二阶段的启动任务。第二阶段的的代码主要是完成外围设备接口的初始化，并且设置好启动参数，然后跳转到内核代码起始位置启动操作系统；另外嵌入式产品在研发阶段，第二阶段的代码还有一个重要的功能就是是建立起和宿主机的通信，并从宿主机上将下载 Linux 内核和文件系统到目标机。两个阶段启动流程可以用图 3-2 表示。

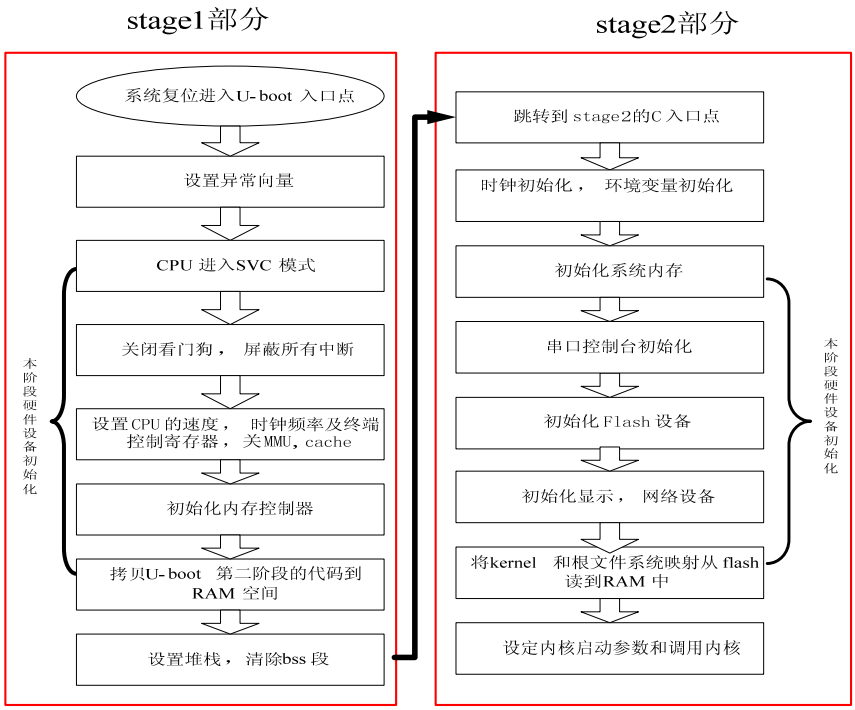

图 3-2 U-Boot 的启动流程

移植也分两个阶段进行，移植第一阶段代码的主要根据主板上 CPU 的体系结构，SDRAM 和 FALSH 存储芯片的型号不同来设计的，属于芯片级移植，主要是修改 cpu/arm920t/ start.S 文件中的大部分代码，因为此时还没有准备好 C 程序的运行环境，所以这一阶段的代码主要用汇编语言编写。第二阶段属于板级移植，主板所接的外部设备不同，代码就不同，主要是自行实现文件系统的烧写功能和修改添加一些外设接口的驱动程序，因为第一阶段已经准备好 C 程序的运行环境，

所以全部用 C 语言编写。本次移植的 BootLoader 需要实现的功能主要有：

（1）支持 TFTP 协议，NFS 协议从网口下载内核和根文件系统。

（2）支持 USB 协议下载内核和根文件系统。

（3）能烧写任意页对齐和非页对齐字节数的内核和根文件系统到 NANDFLASH。

（4）自动识别从 NANDFLASH 启动还是 NORFALSH 启动。

（5）能直接引导 zImage 格式的内核镜像。

3.2.1 U-Boot 第一阶段关键技术研究

（1）U-Boot 支持很多不同的硬件平台，因此必须在 U-Boot 根目录下修改 Makefile 文件将其编译成 ARM 平台如下：

CROSS_COMPILE ?=arm-Linux-//指定交叉编译器为 ARM 平台

MY2440_config。

@$(MKCONFIG)$(@:_config=) arm arm920t MY2440 samsung s3c24x0//添加编译选项，建立自己的开发板。

（2）ARM 体系结构的 CPU 有两种工作状态，七种工作模式[13]，但从 U-Boot 方面考虑，CPU 执行的是 ARM 指令集，所以系统上电复位的时候首先要将 CPU 的工作状态设置为 ARM 状态。另外，U-Boot 要做的事情主要是初始化系统相关的硬件资源，需要获取尽量多的权限，以方便初始化操作初始化硬件，所以工作模式设置为 svc 模式。CPSR 寄存器的格式如图 3-3 所示，设置 T=0，M4M3M2M1M0=10011。由于在这一阶段 CPU 不能响应任何外部中断，所以需要在前面将 CPSR 中的 IRQ 和 FIQ 位置 1，并设置 INTMSK 和 INTSUBMSK 寄存器，屏蔽掉所有的外部中断请求。

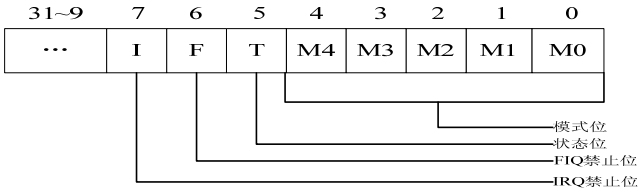

图 3-3 CPSR 寄存器的格式

（3）看门狗是一个硬件模块，其作用是：在嵌入式系统的很多应用场合，系统长期运行且无人值守，难免出现程序跑飞造成系统崩溃的现象，这时就需要看门狗发出复位信号重启系统。只有在整个系统运行起来之后才需要看门狗机制，U-Boot 只是初始化相关的硬件资源而已，完全用不到该机制，为了防止 U-Boot

的启动过程中 CPU 不断重启，需要关闭看门狗。

（4）CPU 上电之后，晶振输出稳定，CPU 时钟 FCLK=Fin(晶振频率)，CPU 开始执行指令，但实际上，可以通过软件设置 MPLLCON 寄存器来调节 FCLK 与 Fin 的倍数，可以将 CPU 工作频率提高到 405MHz。UPLLCON 用来设置 USB 的工作时钟，当系统主频设置为 405MHz，USB 时钟频率设置为 48MHz 的时候可以稳定运行，设置 MPLLCON，UPLLCON 如下：

```
MPLLCON=(0x7f<<12)|(0x02<<4)|(0x01)
UPLLCON=(0x38<<12)|(0x02<<4)|(0x02)
```

CLKDIVN 用于设置 FCLK，HCLK，PCLK 三者的分频比。寄存器的推荐值可参考 S3C2440 的数据手册[9]。设置 CLKDIVN 的值如下：CLKDIVN=0x05；即 HDIVN=0b10， PDIVN=1，所以分频比设为 FCLK:HCLK:PCLK=1:4:8。

（5）关闭 MMU，关闭 CPU 中的指令 Cache 和数据 Cache。在 ARM920T 的协处理器 CP15[14]中，其中 c7 是 Cache 寄存器，c8 是 TLB 控制寄存器，往 c7，c8 写 0，使 Cache，TLB 内容无效。由于 U-Boot 启动的时候访问的都是物理地址，不需要进行物理地址到虚拟地址的映射，所以设置 CP15 中的 c1 寄存器关闭 MMU，c1 寄存器各个位的意义如下：

表 3-1 协处理器中 c1 寄存器的格式

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
L4	RR	V	I	Z	F	R	S	B	L	D	P	W	C	A	M

V：表示异常向量表所在的位置： 0:异常向量在 0x00000000; 1: 异 常 向 量 在 0xFFFF0000。

I： 0:关闭 ICaches 1:开启 ICaches。

R 和 S：用来与页表描述符以其确定内存的访问权限

B：0:CPU 为小段字节序 1:CPU 为大端字节序

C：0:关闭 DCaches 1:开启 DCaches

A：0:数据访问时不能进行地址对齐检查 1:数据访问时进行地址对齐检查

M：0:关闭 MMU 1:开启 MMU

设置：V=0，R=0，S=0，B=0，C=0，A=0，M=0，A=1，I=1。表示中断向量表从 0 地址开始，关闭指令 Caches，关闭数据 Caches，数据访问时进行地址检查，小端字节序，关闭 MMU 即不需要进行物理地址到虚拟地址的映射。

```
mrc  p15, 0, r0, c1, c0, 0
```

```
    bic   r0, r0, #0x00002300      @ clear bits 13, 9:8 (--V- --RS)
    bic   r0, r0, #0x00000087      @ clear bits 7, 2:0 (B--- -CAM)
    orr   r0, r0, #0x00000002      @ set bit 2 (A) Align
    orr   r0, r0, #0x00001000      @ set bit 12 (I) I-Cache
    mcr  p15, 0, r0, c1, c0, 0
```

（6）SDRAM 控制器初始化。S3C2440 为 32 位 CPU，也就是说其数据总线和地址总线的宽度都是 32 位，那么内存数据的输入输出也保证是 32 位总线，所以采用两片 16 位宽总线的内存芯片并联构成 32 位总线的内存，其中一个芯片连接到 CPU 数据总线的低 16 位，另外一个芯片连接到数据总线的高 16 位。其中 lowlevel_init.S 文件是完成 SDRAM 控制器初始化代码，内存初始化完全依赖开发板上的具体的芯片型号，这里采用的是两片 16 位的 K4S564632C-TC75 芯片[15]，所以在 board/samsung/MY2440 中添加 lowlevel_init.S 文件，根据 K4S564632C-TC75 芯片手册和 S3C2440 中 13 个寄存器的配置说明编写 SDRAM 初始化代码。

```
#define BWSCON    0x48000000       /* 13 个存储控制器的开始地址  */
lowlevel_init:
    ldr         r0, =SMRDATA
    ldr   r1, _TEXT_BASE
    sub   r0, r0, r1
    ldr   r1, =BWSCON       /* Bus Width Status Controller */
    add         r2, r0, #13*4
0:
    ldr         r3, [r0], #4 /*将 13 个寄存器的值逐一赋给对应的寄存器*/
    str         r3, [r1], #4
    cmp          r2, r0
    bne          0b

    /* everything is fine now */
    mov pc, lr
.ltorg
SMRDATA:   /*下面是 13 个寄存器的值*/
……
```

（7）自动识别启动方式，初始化 FALSH 控制器，并将所有代码搬移到 SDRAM

中运行。拷贝代码之前先要判断系统是以什么方式启动，当选择开关设置外部引脚 OM[1:0]=01,10 时候，从 NORFALSH 启动； 当设置 OM[1:0]=00 的时候，从 NANDFLASH 启动。

```
/*****************CHECK_BOOT_FALSH**************************/
    #define BWSCON 0x48000000
    ldr r0,=BWSCON    /*BWSCON[2:1]=OM[1:0]=00 的时候，从 NANDFLASH 启动*/
        ldr r0,[r0]
        and r0,r0,#6
        cmp r0,#0
        bne relocate    /*BWSCON[2:1]=OM[1:0]！=00 的时候，从 NORFALSH 启动*/
```

对于 NORFALSH 启动的情况，因为 NORFALSH 的开始地址即为 0，则 CPU 可以直接从 NORFALSH 的开始地址 0x00000000 处取指令执行，并将 U-Boot 代码从 NORFALSH 中拷贝到 SDRAM 中。

```
    /**************NOR FALSH 的自拷贝循环*******************/
    /*r0 的初始值为当前 U-Boot 代码在 NORFALSH 中的存储位置
      r1 的初始值为 SDRAM 的起始地址，r2 为 U-Boot 的结束地址=r0+U-Boot 的代码
      长度      */
    copy_loop:
        ldmia     r0!, {r3-r10}   /*从地址为 r[0]的 NORFALSH 中读入 8 个字节的数据到
r3-r10*/
        stmia     r1!, {r3-r10}   /* 将 r3-r10 中 8 个字节的数据复制到地址为[r1] 的内存中*/
        cmp r0, r2                /*判断是否拷贝完毕 */
        ble   copy_loop
```

从 NANDFLASH 启动要比从 NORFALSH 启动机制复杂，系统上电之后，CPU 会自动通过内部的硬件机制将 NANDFLASH 前 4KB 的内容复制到 stepping stone 中，然后从 stepping stone 中开始取指执行，所以必须在前 4KB 的代码中完成对 NANDFLASH 控制器的初始化，然后通过 NANDFLASH 控制器将剩下的 BootLoader 代码从 NANDFLASH 拷贝到 SDRAM 中去。最后跳转到 SDRAM 中继续执行第二阶段的代码。

系统采用 NANDFLASH 芯片型号是 K9VF2G08[16]，U-Boot 第一阶段主要是从 NANDFLASH 拷贝代码，不需要往 NANDFLASH 写数据，所以只要实现对 k9f2g08 的初始化和读时序就可以；S3C2440 提供了专门控制 NANDFLASH 的控

制器接口，各个寄存器的配置，在数据手册中有详细的说明。nand_read_ll 函数的作用就是完成 U-Boot 数据从 NANDFLASH 到 SDRAM 中的拷贝，该函数调用了 s3c2440_write_addr_lp()函数完成一页数据的读取，整个 nand_read_ll 函数的读操作流程如 3-4 所示。

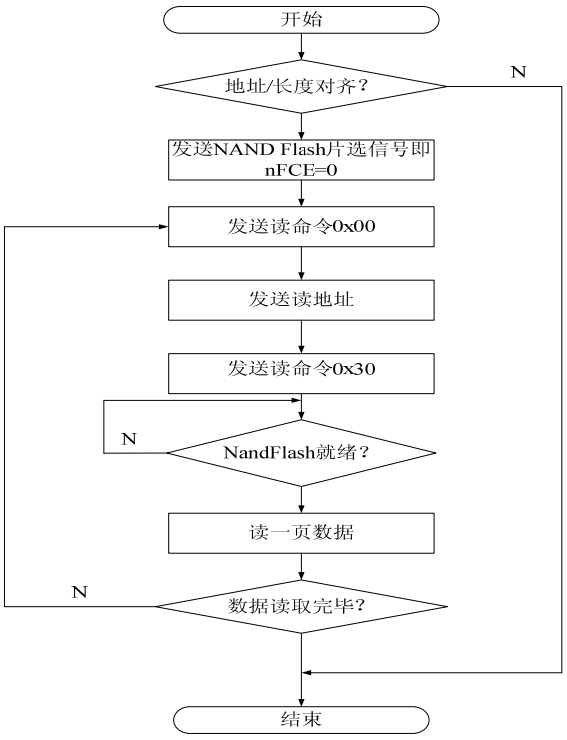

图 3-4 NANDFLASH 读操作流程图

（8）设置堆栈[17,30]。因为 U-Boot 还要运行 C 程序代码，这会导致出入栈操作，为了保证出入栈操作的正确，需要设置各模式下正确的的栈顶位置即 SP 指针的初始值，代码如下：

```
stack_setup:
    ldr   r0, _TEXT_BASE
    sub   r0, r0, #CONFIG_SYS_MALLOC_LEN /*计算全局数据的起始地址*/
    sub   r0, r0, #CONFIG_SYS_GBL_DATA_SIZE /*计算全局数据的结束地址*/
#ifdef CONFIG_USE_IRQ
    sub   r0, r0, #(CONFIG_STACKSIZE_IRQ+CONFIG_STACKSIZE_FIQ)
#endif
    sub   sp, r0, #12
```

根据上面的代码就可以知道 U-Boot 的内存使用情况，如图 3-5 所示。

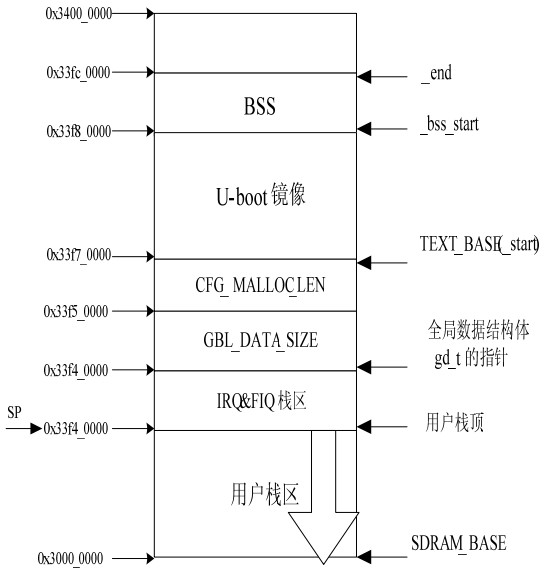

图 3-5 U-Boot 的内存分布图

（9）清除 bss 段。bss 段是未初始化的全局变量段（初始值为 0，无初始值的全局变量，静态变量将自动被放在 bss 段），C 语言标准规定，未初始化的全局变量必须为 0，否则这些变量的初始值是一个随机的值，若程序直接使用这些没有初始化的变量将引起未知的后果，所以要将 bss 段清 0，而不是置 1。

```
clear_bss:
    ldr   r0, _bss_start        /* bss 段的开始地址，在 U-Boot.lds 中指定 */
    ldr   r1, _bss_end          /* bss 段的结束地址，在 U-Boot.lds 中指定 */
    mov r2, #0x00000000
clbss_l:str     r2, [r0]        /*将 bss 段全部清 0*/
    add  r0, r0, #4
    cmp r0, r1
    ble   clbss_l
```

（10）跳转到 RAM 中的第二阶段代码的 C 入口处。在这里之所以通过修改 pc 指针的值来现程序执行点的跳转，是因为整个程序代码已经从 FALSH 搬移到了 SDRAM 中，内存起始地址已经从 0x00000000 变为 0x30000000。ldr pc，_start_armboot 语句执行完之后，pc 的值就是函数 start_armboot 在 SDRAM 内存中的首地址。而 b 指令的实现地址：将机器码的低 24 位作为偏移量，跳转到当前指令位置+偏移量，这样一来 b 指令跳转后，程序仍在 NORFALSH 中运行，并不跳转到 RAM 中，程序将无法执行。

```
ldr   pc, _start_armboot
```

```
_start_armboot:  .word start_armboot
```

3.2.2 U-Boot 第二阶段关键技术研究

第一阶段只是起到代码重定位的作用，真正的功能实现到了第二阶段才开始。因为系统的主板和 U-Boot 默认支持的 SMDK2410 开发板的有很大的变化，所以必须修改和添加几个底层的驱动代码，U-Boot 才能正常工作。原来的开发板是不支持 USB 下载的，NANDFLASH 也只支持 512B+16B 页大小的芯片。这里重点研究一下 NANDFLASH 驱动的设计，网卡驱动的修改，USB 驱动的移植，YAFFS2 文件系统烧写功能的实现和下载速度的优化，以及直接引导 zImage 格式的内核镜像的实现。

（1）添加 NANDFLASH 驱动

U-Boot 进入第二阶段之后，会调用 nand_init()函数初始化 NANDFLASH 控制器，nand_init()函数又调用了同文件下的 nand_init_chip()函数，nand_init_chip()函数接着又调用了同目录下 s3c2410_nand.c 文件中的 board_nand_init()函数，然后再调用 nand_base.c 中的 nand_scan()函数。

NANDFLASH 驱动中各层之间的函数调用关系如图 3-6 所示。其中 nand_scan()函数主要实现了 NANDFLASH 各种参数的设置(比如自动识别 NANDFLASH 的大小，每一页的大小，数据宽度等信息)，设置对 NANDFLASH 的片选，读写控制等，在整个 NANDFLASH 驱动中处于次底层的地位。 board_nand_init()函数主要实现 S3C2440 NANDFLASH 控制器相关的初始化，设置控制器的时序，设置寄存器的读写地址，在在整个 NANDFLASH 驱动中处于最底层，是真正与具体硬件密切相关的部分。至于向 NANDFLASH 中写数据和从 NANDFLASH 读数据的函数在 U-Boot 中都已经写好，与具体的芯片无关，所以只有 s3c2410_nand.c 下对 NANDFLASH 的操作函数是我们要移植的部分。

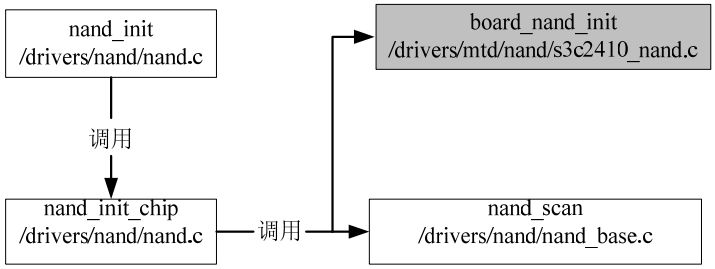

图 3-6 NANDFLASH 驱动中个层之间的函数调用关系

在 U-Boot 最新的 NANDFLASH 驱动中，对 NANDFLASH 的初始化，读写实

现采用了 Linux 内核中的 MTD 架构，对于 MTD 是将所有对 NANDFLASH 的操作方法封装 mtd_info 和 nand_chip 这两个结构体中，注册驱动时调用 nand_init_chip 最终注册的就是这两个结构体，对于移植也是在 s3c2410_nand.c 的 board_nand_init 函数中修改 mtd_info 和 nand_chip 结构中对新 NANDFLASH 的一些操作方法，主要修改的函数如图 3-7 所示。

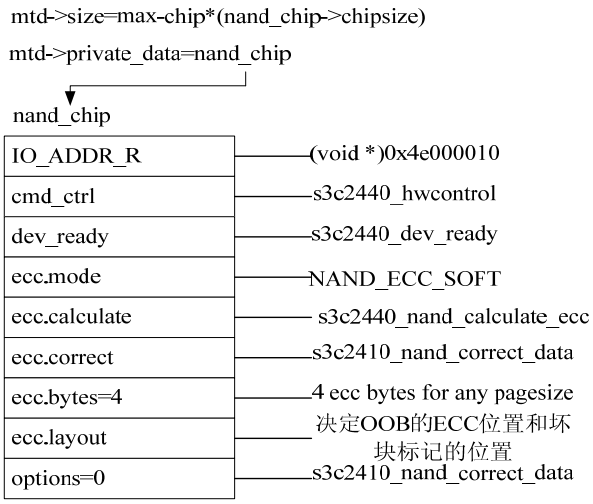

图 3-7 NANDFLASH 底层驱动中函数修改情况

（2）解决 YAFFS2 文件系统烧写速度和文件容量字节对齐问题

在 U-Boot 中添加对于 YAFFS2 文件系统的烧写支持其实就是在写入数据的同时，将文件系统镜像文件中的 oob 数据也写入 NANDFLASH 中的 spare 区。这和 YAFFS2 文件系统原理及 NANDFLASH 的结构有关。

K9VF2G8 存储设备由 4096 块组成，每一块由 64 页组成，一页大小为(2048+64)字节组成，2048 表示的 main 区用于正常的数据存储；64 表示的 spare 区的用于存储一些附加信息，如块好坏的标记，块的逻辑地址，页内数据的校验和等。由于 NANDFLASH 出现的位反转的概率很大，一般在读写时候需要用 ECC 进行错误校验和恢复。

YAFFS2 文件系统的设计充分考虑了 NANDFLASH 以页为存取单位的特点，将文件组织成固定大小的段，在 2KB 页大小的 NANDFLASH 中，YAFFS2 文件系统使用前面 2KB 的 main 区存储数据，后面 64B 的 spare 区存储数据的 ECC 和文件系统的组织信息(OOB 信息)。通过 OOB 不但能实现错误检测和坏块处理，同时还避免了加载时对整个存储介质的扫描，加快了文件系统的加载速度。U-Boot 中已经有对 cramfs，jffs2 文件系统的支持，但与带数据校验功能的 OOB 区的 YAFFS2 文件系统相比，它们是将所以文件数据以简单的线性表形式组织的。所以只要在

次基础上通过修改 U-Boot 的 NANDFLASH 读写命令，增加处理 OOB 数据的功能，即可以实现对 YAFFS2 文件系统的读写支持。实现对 YAFFS2 文件系统的读写支持关键是要解决烧写任意字节的内核和文件系统及 YAFFS2 文件系统烧写速度的问题。步骤如下：

（a）U-Boot 原来只能烧写大小为 2048B 整数倍大小的内核镜像和大小为 2048B+64B 整数倍的根文件系统，在 drivers/mtd/nand/nand_util.c 中的 NAND_write_skip_bad 函数中添加自动修正成文件系统和内核镜像的容量为 NANDFLASH 页对齐大小，这样可以将任意字节大小的内核和文件系统烧写进 NANDFLASH。代码实现如下：

```
    if (((*length)%(NAND->oobsize+NAND->writesize)) != 0)
        {
        printf ("Attempt to write error length data!\n");
        *length=(((*length)/(NAND->oobsize+NAND->writesize))+2)*(NAND->oobsize+NAND
->writesize);
        printf ("The length data is modified to %d bytes\n",*length);
        return -EINVAL;
        }
        datapages = *length/(datasize+oobsize);
        *length = datapages*datasize;
        left_to_write = *length;
```

（b）U-Boot 原来的代码中都是将文件系统数据的 DATA 区和 OOB 区分离，大页 NAND 的一页 DATA 区 2048 字节，OOB 区 64 字节，用上面的方法可以只用 buf 一块内存就可以完成分离，它是将整个 1 个 BLOCK 的数据进行整体搬移，取出前面 OOB 数据的放在 oobtemp 里，然后将后面的所有数据全部向前搬移至刚才 OOB 数据开始的地址，然后再将刚才 OOB 的数据（放在 oobtemp 里的）添加到全部数据后面。而且 memmove 函数比 memcpy 函数效率要低，由于 memcpy 不能覆盖目标区域和源区域有相同地址的部分，所以只能用 memmove，这样就造成了烧写速度非常的慢。

在分析了上述代码的弊端后，重新编写代码在后面的代码中用 data_buffer 和 oob_buffer 替换 buf， 虽然多用了 data_buffer 和 oob_buffer 两块内存，但是效率却大大提高了。看似很简单的修改，首先必须了解 NAND FALSH 的数据烧写原理，并且还要找到影响速度的代码位置。修改文件 drivers/mtd/NAND/NAND_base.c 中

的 NAND_write()函数如下：

```
    for(i = 0; i < (datapages); i++)
        {
            将：
            memcpy((void *)oobtemp, (void *)(buf + datasize * (i + 1)), oobsize);
            memmove((void *)(buf + datasize * (i + 1)), (void *)(buf + datasize * (i + 1) + oobsize), (datapages - (i + 1))* (datasize) + (datapages - 1) * oobsize);
            memcpy((void *)(buf+(datapages) * (datasize + oobsize) - oobsize), (void *)(oobtemp), oobsize);
            改为：
            memcpy((void*)(data_buffer+i*datasize),(void*)(buf+datasize*i+i*oobsize),datasize);
            memcpy((void*)(oob_buffer+i*oobsize),(void*)(buf+datasize*(i+1)+i*oobsize),oobsize);
        }
```

（3）解决网络协议下载文件不完整问题

U-Boot 源代码中已经对 CS8900，RTL8019 和 DM9000 等网卡有比较完善的代码支持，而本系统使用的是 DM9000 网卡芯片，所以在源代码中屏蔽掉对 CS8900 的支持，添加对 DM9000 的支持，在 include/configs/my2440.h 中添加对 DM9000 网卡的支持。

其中 CONFIG_DM9000_BASE 的定义尤为重要，不同的板子只要修改这个参数就可以，在系统主板上 DM9000 挂在 BANK4 上，即网卡基址为 0x23000000。并且在 board_init()函数中添加一个网卡初始化函数 board_eth_init()，重新编译后可以实现对 DM9000 网卡的支持，但是会出现 could not establish link 错误，而且在显示 MAC 地址很久后才会响应，跟踪源代码可以知道需要在/drivers/net/dm9000.c 文件中的 eth_init()函数中屏蔽掉 MII 接口所使用的语句。

对于 NFS，修改 NFS_TIMEOUT 为(CONFIG_SYS_HZ/1000*2000UL)，否则会出现*** ERROR：Cannot mount 的错误。

还有一个问题就是第一次总 ping 不通主机，经过尝试将上面屏蔽掉的语句修改为 if (i == 10000) {printf("could not establish link\n");return 0;}这样每一次都能 ping 通；但是每次 ping 结束之前会调用 dm9000_halt()函数把网卡禁用掉，这样网线的物理连接就自动断掉，所以需要屏蔽掉 dm9000_halt()函数中所以的内容改为空实现。

还有一点就是在使用 TFTP 协议下载数据的时候会出现传送不完整又重新传送，不断循环的问题，这主要是因为在网卡初始化的时候，设置 IMR 寄存器的 bit7 为 1，内存数据被设置为 0x0c00，也就是说读取这两个寄存器数据读取的地址指针复位，所以需要在读取状态位之前重新设置以下地址指针，代码如下：

```
u16 temp;
temp=DM9000_ior(DM9000_MRRH);
temp=DM9000_ior(DM9000_MRRL);
```

加上这三行代码之后，文件传输就正常了。

（4）添加 USB 下载功能

使用 USB 下载内核和文件系统具有传输速度快，即插即用，连接方便等特点，但 USB 协议非常复杂，开发基于 S3C2440 的 USB 设备驱动很困难，本次 USB 下载功能的实现是在 Linux 中自带 USB 驱动[18]的基础上，经过修改移植到 U-Boot 中来的。

将 Linux 中的 USB 驱动源代码拷贝到 U-Boot 的 drivers/usb/slave 目录下，然后在 lib_arm/board.c 的 init_sequence[]中添加 USB 设备的初始化函数 USB_init_slave()，这个函数定义在 USBinit.c 文件中，主要是设置 USB_slave 的引脚功能及相关的控制寄存器，然后设置 USB 中断程序的入口以及开启中断。

初始化完毕之后，主机就开始设备枚举的过程了，枚举过程中会调用 usbmain.c 中的 usbmain()函数，usbmain()函数又调用了 InitDescriptorTable()，ConfigUSBd()，PrepareEp1Fifo()三个函数。其中第一个函数定义在 usbsetup.c 中，主要是初始化一些 USB 描述符，如设备描述符，配置描述符等，并在 USB 控制传输的时候返回给从设备。

第一次配置时会调用 ConfigUSBd()函数，并开启中断。在重置 USB 的过程中再调用 ReconfigUSBd()函数，该函数定义在 usblib.c 中，首先是操作电源管理器关闭自动挂起功能，然后是针对每个端点（最主要的是端点 0 端点 3）来设置端点的最大包大小，端点类型，传输方向，以及是否支持 DMA，最后清除所有的中断状态寄存器标志。PrepareEp1Fifo()是配置端点 1 的，这里没用到。

USB 设备控制器最主要的部分是中断处理程序 Isrusbd()，定义在 usbmain.c 中，在 Isrusbd()中每个中断端点都有不同的处理程序，如端点 0 对应的中断处理程序为 Ep0Handler()，这个函数主要是针对中断的类型进行相应的处理：

```
writeb(EP0_INT, &USBdevregs->EP_INT_REG);
Ep0Handler();
```

EP0_CSR_IN_CSR1_REG 寄存器里面有相应端点的中断类型。其中有两种中断是不正常状态：EP0_SETUP_END 这个标志表示控制传输，SETUP 阶段在数据包到来前，传输就结束了，造成这个原因有可能是总线超时。如果主机在送出 SETUP 包以及数据包之后，USB 设备迟迟不发送应答包， EP0_SENT_STALL 这个标志会置位。

EP0_OUT_PKT_READY 这个标志是正常的状态，在 USB 设备接收到了一个正确的 Token 以及数据包时，这个标志就置位，从而程序可以从 FIFO 中读取数据，根据 setup 包中的请求类型来进行相应的处理，然后清除这个标志。接着 USB 设备控制器就会自动发送 ACK 包给主机，从而结束一个事务。

类似，用于下载的 OUT 批量传输端点 3 的中断处理程序 Ep3Handler 也是这样一个流程。但是不同的是这里还用到了 DMA 传输。中断状态 EPO_OUT_PKT_READY 标志置位就代表着第一个数据包的到来 ，由于端点 3 配置的大小为 32 个字节，所以一个包的数据就是 32 个字节。主机软件 DNW 在每个文件的头部加了 8 个字节的信息来标示下载地址与文件长度，所以这个数据就是处理下载地址与文件长度的信息的。下载地址保存在 downloadAddress 里，文件长度保存在 downloadFileSize 里。处理完第一个包后，首先禁 止 USBD 中断，主机发送的数据包都会被 USB 设备忽略，从而主机会重新发送。这个中断处理函数只处理一个包的数据就关了中断， USB 批量传输就进行不下去了。但是程序中还有一个函数是进行批量传输先运行的，这就是 usb_receive 。usb_receive 函数紧接着 usb_init_slave 运行，如果设备枚举成功，那么就会设置 downloadFileSize = 1，”USB host is connected. Waiting a download “就会输出到终端，从而运行 DMA 处理代码，否则就会等待 USB 设备配置完毕。头文件中是定义了 USBDMA 宏，所以#if USBDMA 后代码都会执行。在说这段代码之前首先说一下 Timer_InitEx 和 Timer_StartEx()，这两个函数和 if(wait)下面代码都是为了显示现在进度，以及下载平均耗时。 这里主要用了看门狗定时器作为硬件定时器， 如果定义了 wait 那么程序会将下载信息显示在终端，然后还统计每秒传输的字节数。

下面说明 DMA 的处理代码。这段代码的主要作用就是针对传输文件的大小来决定 DMA 的使用情况，如果使 用 DMA，则配置端点 3 为 DMA 模式，设置 DMA 传输的终点地址以及每次 DMA 传输的大小，开启 DMA 中断。当设置好 DMA 时并打开了 DMA 中断，每传完一个设置的大小，就会进入 DMA 中断。所以如果传输文件的大小小于 1023KB，那么只会进入一次 DMA 中断。DMA 中断处理程序在每次 DMA 传输完成后进入，根据还剩文件的字节数来决定是否继续传输，以

及怎样传输。 当传输完成后，禁止 DMA 中断，将端点配置成中断模式，好进行下一次的批量传输。经过上面的分析可知，USB 下载数据使用了 DMA 中断，因此需要第一阶段的启动代码中设置 DMA 中断栈如下：

```
        sub    lr，  lr，  #4                    @ the return address
        ldr     sp，  IRQ_STACK_START @ the stack for irq
        stmdb   sp!，      { r0-r12,lr }      @ save registers
        ldr      lr，  =int_return             @ set the return addr
        ldr      pc，  =IRQ_Handle             @ call the isr
int_return:
        ldmia   sp!，      { r0-r12,pc }^     @ return from interrupt
```

另外还需要在 common 目录下添加 usbslave 命令的实现文件，在/include/s3c24x0.h 中修改几个 USB 控制寄存器，接着重新编译 U-Boot 下载到 FALSH 就可以使用 USB 设备下载内核和文件系统了。

（5）直接引导 zImage 格式的内核镜像。在 do_bootm 函数中去掉对 mkimage 工具增加的头的分析，直接调用 do_bootm_linux 函数引导内核，但须在该函数中将内核的引导地址直接修改为内核的加载地址。

3.3 小结

至此 U-Boot 已经移植完毕，该 U-Boot 可以自动识别是从 NORFALSH 启动还是从 NANDFLASH 启动，同时支持 USB 下载和 TFTP，NFS 协议下载。图 3-8 是 U-Boot 的启动界面，快捷启动菜单是为了方便下载和安装操作系统自行添加的。选择 v 烧写 U-Boot 到 NANDFLASH，选择 k 烧写 Linux 到 NANDFlas，选择 y 烧写 YAFFS2 文件系统到 NANDFLASH，选择 b 启动 Linux 操作系统。

解决的关键问题主要有以下几点：

（1）本次 bootlaoder 设计自行添加了 NANDFLASH 的初始化代码和 NANDFLASH 驱动代码，以及从 FALSH 到 SDRAM 的搬移代码，实现了 U-Boot 自动识别启动方式：既支持从 NORFALSH 也支持从 NANDFLASH 双启动；实现了对 NANDFLASH 的读写操作。

（2）修改现成的网卡驱动和网络协议层的相关代码，解决了 TFTP 下载时总是出现传送不完整而又重新传送的现象。

（3）自行将 Linux 的 USB 驱动源码的基础上进行改进移植到 U-Boot 中。实

现了 USB 协议下载内核文件和根文件系统

（4）官方提供的 U-Boot，当烧写的文件超过一定大小时候就会出现停止传输的现象，所以需要修改源代码使 U-Boot 支持烧写任意字节大小的的内核和文件系统。

（5）实现了 YAFFS2 文件系统烧写功能并解决了原来烧写速度过慢的问题。

（6）实现了直接引导 zImage 格式的内核镜像文件。

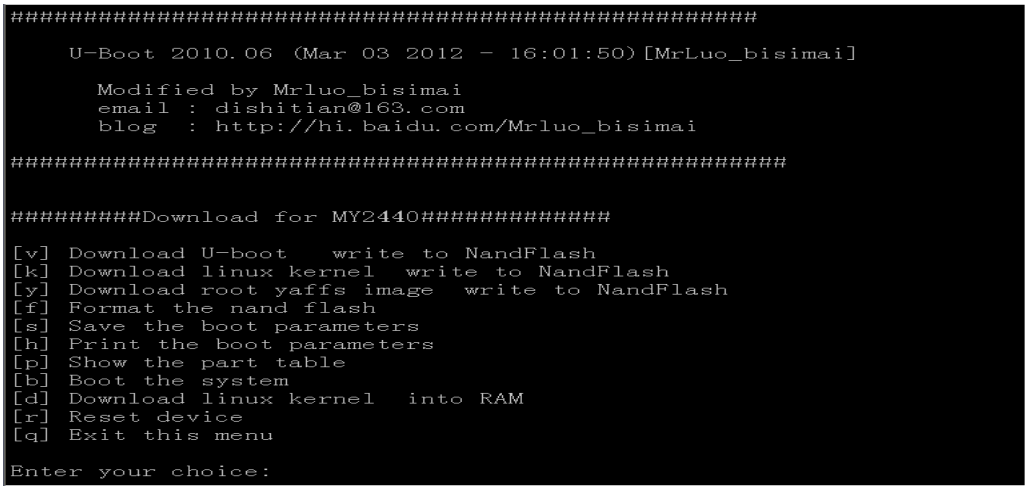

图 3-8 U-Boot 的启动界面

第四章 嵌入式 Linux 操作系统移植

4.1 引言

嵌入式 Linux 是在基于 x86 体系结构的标准 Linux 基础之上经过裁剪处理之后，所得容量只有几 K 字节或者几 M 字节的专用 Linux 操作系统；是适合存储在嵌入式存储芯片或者单片机 ROM 中的微小内核，是严格根据特定应用场合而设计的。但裁剪之后的嵌入式 Linux 同标准 Linux 一样，都具有免费开源，低成本，高性能，支持多种硬件平台和良好的网络协议支持等特点。

为了更好的适应嵌入式领域的后期软件开发，在很多嵌入式 Linux 开发过程中通常都是直接在 Linux 的基础上根据自己的硬件平台定制裁剪得到所需要的嵌入式 Linux 内核。裁剪之后只保留了 Linux 内核原来一些最基本的操作系统功能，如任务调度，内存管理，中断处理和文件系统支持及支持网络协议等功能，这样不但减小操作系统内核的体积，而且更利于嵌入式系统的维护和移植。

移植就是将 Linux 操作系统从 X86 平台迁移到基于 ARM 体系结构的 S3C2440 的嵌入式平台上运行，主要包括交叉编译工具链的安装，Linux 内核移植，文件系统制作，其次是所需函数库的移植，最后是是应用程序移植（如嵌入式 WEB 服务器的移植）。

4.2 嵌入式 Linux2.6 操作系统移植

本次 Linux 操作系统移植是将原来基于 x86 体系结构的标准 Linux 内核编译成基于 ARM 体系结构的嵌入式 Linux，并根据硬件配置修改了系统的时钟频率，对 NANDFLASH 存储空间重新进行了分区，然后修改了网卡驱动程序，最后制作了基于 NANDFLASH 的 YAFFS2 根文件系统，从而得到一个能在主板上稳定运行的专用嵌入式 Linux 操作系统。其中内核移植主要包括两方面的工作，一是与体系结构相关的移植；二是系统所需硬件驱动的移植。具体步骤如下：

（1）修改 Linux 编译器，将 Linux 编译成 ARM 平台。在内核目录树下找到编译管理文件 Makefile，指定体系结构是 ARM，交叉编译工具是 arm-linux-gcc 如下：

```
ARCH ？=arm
CROSS_COMPILE ?= /usr/setup/MY2440/4.3.2/bin/arm-linux-。
```

（2）在 U-Boot 和 kernel 中都会有一个机器码，只有这两个机器码一致时才能引导内核，否则就会出现如下 machtype 不匹配的错误信息或者死机。在 U-Boot 的 include/asm-arm/mach-types.h 文件中定义开发板的机器码为 1999。所以在 Linux 内核源码的 arch/arm/tools/mach-types 文件中将系统开发板的 MACH_TYPE 码的值也定义为 1999，还需要在 arch/arm/mach-s3c2440/mach-smdk2440.c 中修改 kernel 的 MACH_TYPE 代码引用部分，即将 MACHINE_START(S3C2440,"SMDK2440 ")修改为 MACHINE_START(S3C2440，"MY2440")。并在 arch/arm/kernel/head.S 的 ENTRY 下添加如下代码：

```
ENTRY(stext)
mov r0,#0
mov r1,#0x7cf//MACH_TYPE_MY2440 值 1999 转换成十六进制就是 0x7cf。
ldr r2,=0x30000100//Linux 启动参数存放地址。
```

（3）设置 NANDFLASH 分区信息表。NANDFLASH 可以分成很多块，每块有不同的名字，大小和用途。在 arch/arm/plat-s3c24xx/common-smdk.c 中将 NANDFLASH 分区表设置成和 U-Boot 执行 mtdparts 命令所查看到的分区一致。这里只创建四个分区，四个分区空间分配结构如图 4-1 所示：

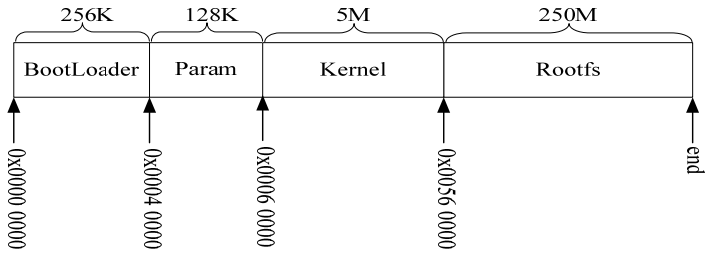

图 4-1 NANDFLASH 空间分配结果图

（4）修改系统时钟。S3C2440 支持两种晶振频率：12MHz 和 16MHz。系统主板所使用的是 12MHz 晶振，因此在 arch/arm/mach-s3c2440/mach-smdk2440.c 中的中将 s3c24xx_init_clocks(16934400)修改为 s3c24xx_init_clocks(12000000)。代码如下：

（5）在 drivers/mtd/NAND/s3c2410.c 中将 chip->ecc.mode=NAND_ECC_SOFT 改为 chip->ecc.mode=NAND_ECC_NONE 以修改 NAND FALSH 的校验方式，否则在系统上电启动时会提示 NANDFLASH 的 ECC 校验错误或者读取 I/O 的错误。

（6）在内核中打上支持 YAFFS2 文件系统的补丁，使内核支持挂载 YAFFS2 根文件文件系统。

目前还仅仅只支持外设 FALSH 和串口等一些基本的驱动，根据项目要求还需要继续移植 DM9000 网卡芯片驱动和 USB 摄像头驱动。

（7）修改网卡驱动。

（a）网卡芯片挂在系统的 bank4 上，当 cpu 访问 0x2000_0000~0x27ff_ffff 这个范围的地址时会激活片选使能信号 nGCS4，又 DM9000 默认的偏移的地址为 0x300，所以在 arch/arm/mach-s3c2440/mach-smdk.c 中创建 dm9000 设备 IO 资源到内核到内核虚拟地址的映射如下：

```
static struct map_desc smdk2440_iodesc[] __initdata = {
    {
        .virtual    = (u32)S3C2410_ADDR(0x02100300),//add by Mr luo
        .pfn        = __phys_to_pfn(0x20000300),
        .length         = SZ_1M,
        .type       = MT_DEVICE,
    }
};
```

(b)在设备初始化列表中添加对 DM9000 网卡设备的支持，并在内核中注册设备。

(c)修改 DM9000 驱动源代码。根据 S3C2440 到 dm9000 读写数据时候的时序图，在 drivers/net/dm9000.c 中的 dm9000_probe() 函数中修改 BWSCON 和 BANKCON4 寄存器如下：

```
*((volatile unsigned int *)S3C2410_BWSCON) =(oldval_bws con & ~(3<<16 | S3C2410_BWSCON_WS4 | S3C2410_BWSCON_ST4 )) | S3C2410_BWSCON_DW4
_16;
*((volatile unsigned int *)S3C2410_BANKCON4) = 0x3294; 。
    在 dm9000.c 文件中设置 GPGCON 寄存器使 GPG1 的功能设置为 EINT7 如下：
 s3c2410_gpio_cfgpin(s3c2410_GPG1,S3C2410_GPF3_EINT7);
```

（8）配置编译内核，添加对 DM9000 网卡设备和 ZC301 摄像头设备的支持如下：

(a)System Type 默认配置下保留 S3C2440 Machines 下的 SMDK2440 和 SMDK2440 with S3C2440 CPU module，其它类型的 Machines 全部去掉。

(b)配置 Boot options 下的启动选项，将第三个 FALSH 分区 mtdblock3，设为 rootfs 分区，控制台 console 设为 ttySAC0 使 kernel 启动期间的信息全部输出到串口 0 上，设置 Linuxrc 是系统启动之后首先运行的进程。

(c) 保留 Device Drivers--->Memory Technology Device (MTD) support--->NAND Device Surpport--->NAND FALSH surpport S3C2410/S3C2440 Soc--->以支持 NANDFLASH 驱动。

(d) 选中 File systems--->Miscellaneous filesystem--->YAFFS2 file system support 添加对 YAFFS2 文件系统的支持。

(e) 配置 Device Drivers--->Net device support--->[*]Ethernet(10 or 100Mbit) --->DM9000 support --->以支持 DM9000 驱动。

(f) 配置 Multimedia devices ---> (*)Video For Linux--->V4l USB devices--->USB ZC301[P] Image Processor and Control Chip support 支持 ZC301 摄像头驱动。

(g) Kernel Features--->Use The ARM EABI to compile the kernel 一定要选上，否则会出现内核恐慌即内核指针跑飞。主菜单中最后一项" Save an Alternate Configurations File"，按回车产生.config 内核配置文件，该文间在执行 make 命令的时候被调用。配置完成之后，执行 Make zImage 命令编译内核，会在 arch/arm/boot/ 目录下得到适应系统主板的 zImage 内核镜像。

4.3 YAFFS2 根文件系统制作

BusyBox[19]主要为各种小型嵌入式设备提供，简单的说就是 Linux 的一个巨大的工具集，它包括了 Linux 中大部分命令和工具 ,嵌入式根目录下的 bin，sbin 和 usr 目录中的命令和启动程序 Linuxrc 通常是通过 BusyBox 产生的，BusyBox 会根据配置的不同自动生成一些文件。

（1）编译 BusyBox。在 BusyBox 的根目录下中修改 MAKEFILE 文件如下：

```
CROSS_COMPILE=arm-Linux-gccc
ARCH=arm
```

配置编译 BusyBox，在 BusyBox 根目录下生成三个目录文件 bin,sbin,usr 和一个链接文件 Linuxrc。

（2）创建根文件系统必要的目录。将刚才生成的目录文件拷贝到根目录 root-2.6.30.4 目录下，并在当前目下继续创建 dev etc home lib mnt opt proc tmp var

www 目录。内核在引导时候设备节点 console、null 必须存在，所以需手动创建这连个设备节点，其他设备节点则会之后的初始化过程中由 mdev 创建。mdev 也是一个应用程序，必须调用 init 进程之后才能启动，而 init 进程的创建必须要用到/dev/console 和/dev/null 这两个设备文件，所以必须在制作根文件系统/dev 目录时时手动创建这两个设备节点。

（3）etc 目录制作。/etc 目录用来存放系统的配置文件，其中 inittab 是文件系统挂在之后，BusyBox 首先要解析的文件。在 inittab 文件中存放了系统所有的配置信息，BusyBox 需要这些信息来指明要启动的程序，有关 inittab 文件的配置条目如下：

::sysinit:/etc/init.d/rcS 表明启动系统初始化文件/etc/init.d/rcS，字段 sysinit 表明文件/etc/init.d/rcS 在系统启动之后最先执行，并且只执行一次，init 进程等待它结束后才继续执行其它动作。

console::respawn:-/bin/sh 表明在/dev/console 设备节点表示的控制台上启动 respawn 动作的 shell,字段 respawn 表明 init 进程监测发现子进程退出时重新启动它。

::ctrlaltdel:/sbin/reboot 表示当按下 Ctrl+Alt+Delete 组合键时,init 重启执行程序，字段 ctrlaltdel 表示当按下 Ctrl+Alt+Delete 组合键时会执行相应的进程。

::shutdown:/bin/umount –a –r 告诉 init 在关机的时候运行 umount 命令卸载所有的文件系统，如果卸载失败试图以只读方式重新挂载。字段 shutdown 表明关闭系统时候执行相应进程。

（4）当解析完 etc/inittab 后就将启动这些进程，首先要执行启动脚本/etc/init.d/rcS,将文件 etc/fstab 中指明的文件系统挂在到对应挂载点上。

（5）在启动脚本/etc/init.d/rcS 执行完之后将在控制台 console 启动一个 shell，shell 启动会根据文件/etc/profile 配置登陆环境，在用户登陆时候将在/etc 目录下寻找三个文件 passwd,shadow,group 匹配相关信息。

4.4 嵌入式 WEB 服务器移植

boa[20]是一个非常精巧的嵌入式 WEB 服务器，可执行代码只有几十 K，它是一个单任务的的 WEB 服务器，只能顺序完成客户端的请求，而不会 fork 出新的进程来处理并发连接请求，但是 boa 支持 CGI，能够为 CGI 创建一个进程来执行。移植 boa 服务器到只需要用 arm-Linux-gcc 重新编译 boa 源码，并且修改/etc/boa/目录下的 boa.conf 文件。有关 boa.conf 的配置如下：

（1）注释掉 bind 调用的 IP 地址绑定到 INADDR_ANY，通配于服务器所有的 IP 地址；

（2）将/www 目录设置为 HTML 文档的存放目录；

（3）设置/www/cgi_bin 为 CGI 脚本虚拟路径对应的实际路径，一般所有的 CGI 脚本都要放在实际路径。

（4）另外还需要创建日志文件所在的目录/var/boa/log，将 mime.types 文件拷贝到/etc 目录，mime.types 文件用来指明不同文件扩展名对应的 MIME 类型。将 PC 机上生产的可执行文件 boa 复制到嵌入式系统根目录下。执行命令./boa，在客户端打开浏览器输入 IP 地址可以看到服务器的网页则移植 boa 服务器成功。

4.5 小结

图 4-2 是 Linux 移植成功之后的启动界面，可以看出 Linux 可以挂在根文件系统，并且网卡启动成功， ping 命令测试结果表明可以通过网络和宿主机通信。运行 boa 程序，通过 PC 机上的浏览器就可以访问服务器上的测试网页。将 USB 摄像头连接到开发板的 USB 主设备接口，可以看到控制终端立即打印出 zc3xx: probe 2wr ov vga 0x0000，zc3xx: probe sensor -> 0011，zc3xx: Find Sensor HV7131R(c) gspca: probe ok 等信息，说明摄像头驱动加载成功，同时查看/dev 目录可以看到，内核已经自动创建了 video0 设备节点，主设备号为 81，从设备号为 0，USB 摄像头为一个字符设备。以后对/dev/ video0 的操作就等于对 USB 摄像头的操作。

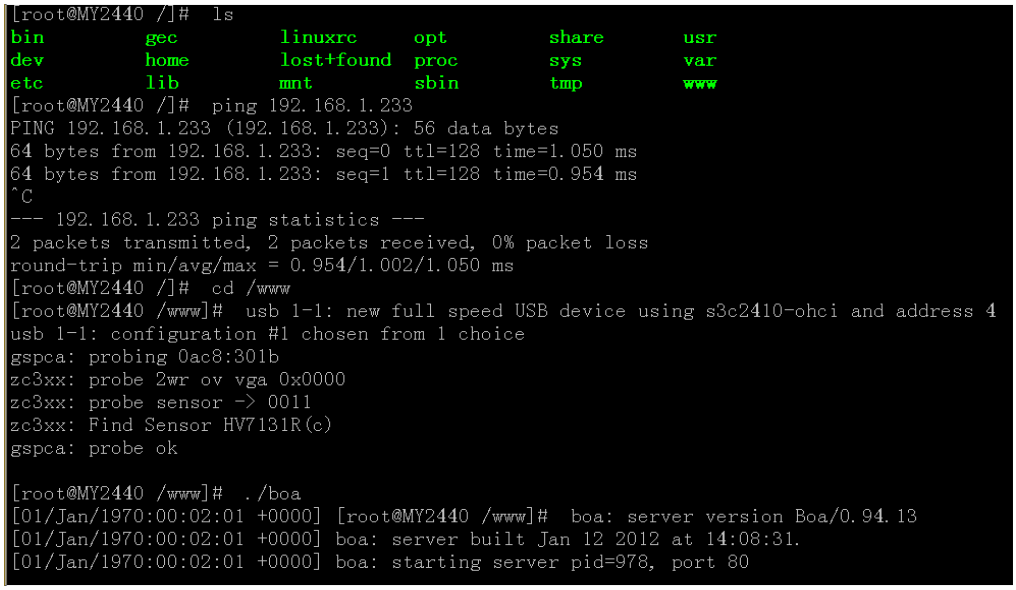

图 4-2 系统 Linux 的启动界面

第五章 系统硬件接口设备驱动程序开发

5.1 引言

在嵌入式系统中，当硬件平台，BootLoader（引导加载程序），内核和根文件系统都做好之后，那么一个集软硬件于一身的嵌入式平台就基本搭建起来了，但此时硬件还不能做任何工作，因为此时的应用程序还不能通过操作系统内核访问和控制底层硬件。图 5-1 是嵌入式系统中的底层硬件，设备驱动程序，嵌入式 Linux 操作系统，以及应用程序之间的关系图。可以看出，上层应用软件要访问硬件设备，只能通过标准 C 中的库函数，或直接通过 Linux 中的系统调用接口使系统从用户空间切换到内核空间，然后在 Linux 操作系统中调用相应设备的驱动程序[21]才能访问底层硬件。

相反，因为驱动程序的存在，屏蔽了底层硬件的差异，使得上层应用程序的开发完全不必理会底层硬件的具体实现，不同体系架构的应用程序，尽管具体的硬件实现完全不同，只要所运行的操作系统相同，不需要太多的修改就可以从一种体系结构的平台移植另外一种体系架构的平台上。由此可见，驱动程序是操作系统内核访问底层硬件的接口，是内核和硬件设备通信的桥梁。也就是说，没有驱动程序的嵌入式硬件就是一对废铁，应用程序和操作系统永远无法实现对底层硬件的控制。

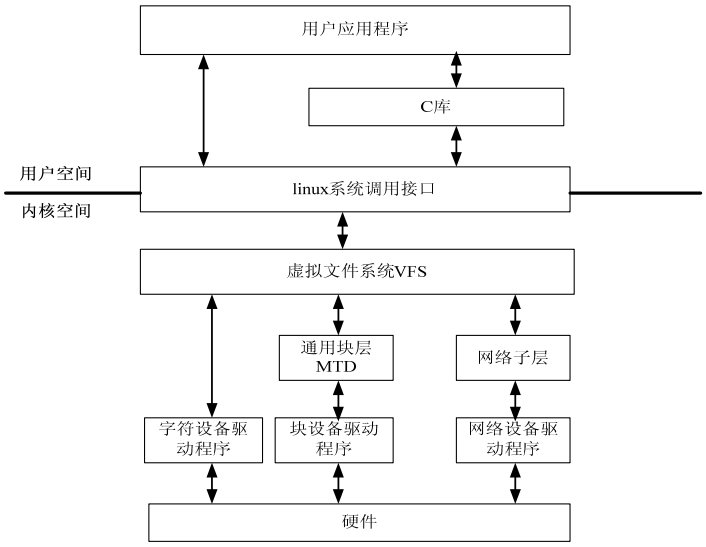

图 5-1 设备驱动整个系统中随处的位置

5.2 字符设备驱动程序的框架分析

5.2.1 字符设备驱动程序简介

嵌入式 Linux 系统从从各种设备的共性出发，将设备驱动程序分为三大类[22]：字符设备驱动，块设备驱动和网络设备驱动。相对于其它两种设备，字符设备驱动程序并不是由内核管理，而是由应用程序利用输入输出函数使用设备文件来访问，字符设备驱动通常都是以字符（即一个字节）为单位按先后顺序在应用程序和硬件设备之间传输数据的，由于没有使用内核中的缓存机制，写入的数据有可能被保存，也可能不保存，没有容量的概念，所以字符流的传输速率比较缓慢，常见的字符设备如：鼠标，键盘，打印机等都是使用字符设备驱动程序来控制。字符设备驱动程序利用应用程序中的文件输入输出函数在设备文件上写入读取数据，然后通过系统调用进入内核空间执行相应的驱动程序中定义的函数。如图 5-2 所示，为了从设备文件读取数据，在应用程序中调用了 read()函数，此时系统会通过系统调用切换到内核空间，调用对应字符设备驱动程序所定义的 xxx_read()函数，从硬件读取数据，并向应用程序返回所读取的数据。

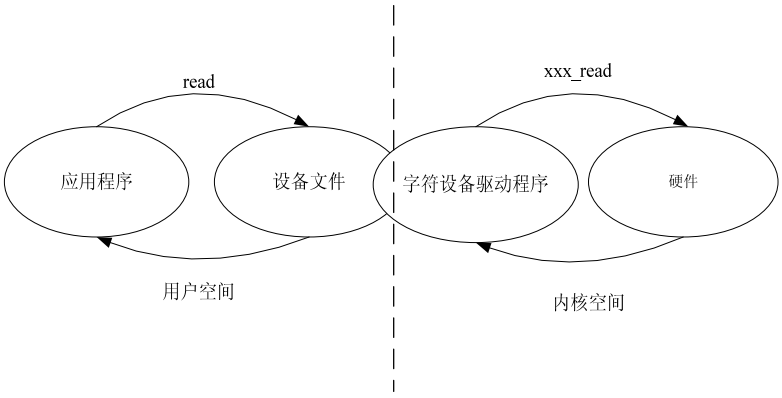

图 5-2 字符设备读取数据流程图

5.2.2 字符设备驱动程序的运作方式

内核利用设备文件记录设备类型和主设备号找到注册在内核上的设备驱动程序，当应用程序利用 open()函数打开设备文件后就获取设备文件上的类型信息和主设备号，再利用类型信息和主设备号得到注册在 chrdevs 变量上的设备驱动程序的索引，接着利用获取的索引值获得 file_operations 结构体[23]地址，file_operations 结构体定义所有操作字符设备的接口函数，也就是说利用设备文件信息和主设备

号在内核中找到对应的驱动程序。图 5-3 表示了应用程序调用字符设备驱动程序的方式，可以看出字符设备驱动程序的框架主要是围绕一个定义在 include/Linux/fs.h 中叫做 struct file_operations 数据结构来实现的，file_operations 结构体是与低级文件输入输出函数对应的函数指针域，相应域上代入函数地址，设备驱动程序赋上相应低级文件输入输出的功能，应用程序使用文件输入输出函数操作设备文件时，内核参考对应设备文件注册的字符设备驱动程序的 file_operations 结构体调用相应的函数。

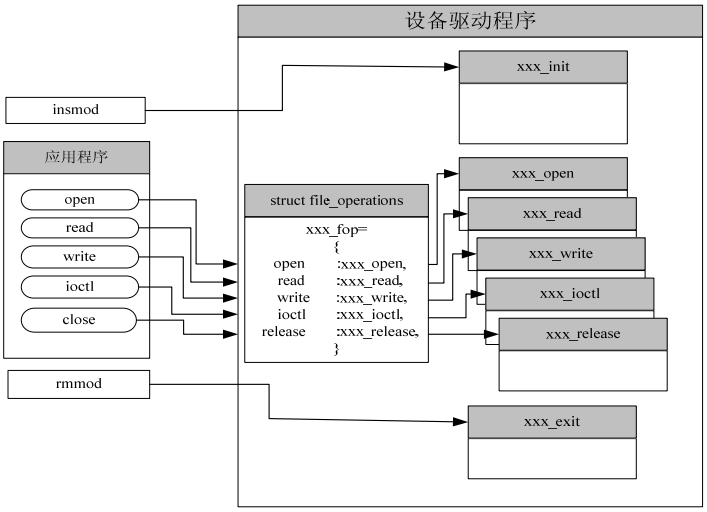

图 5-3 应用程序调用字符设备驱动程序的方式图

如图 5-4 所示，当应用程序需要访问设备文件的时候就会调用 open()函数打开设备节点，获取文件描述符，以后对设备的操作就变成了对设备文件的操作。应用程序调用 open 函数的时候，就会通过操作系统的内核机制执行设备驱动中的 xxx_open()函数，open()函数的的 pathname 和 flags 不能直接传送到 xxx_open 上，而是由 inode 和 filp 变量分配和传送，xxx_open()函数的返回值是文件描述符，传

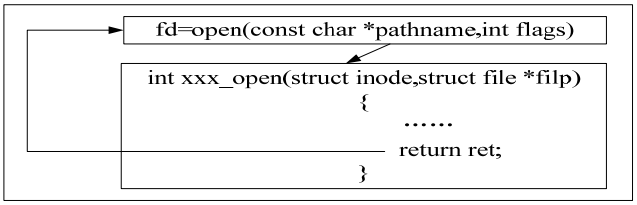

图 5-4 open 函数和 xxx_open 函数之间的调用关系

送到 open()函数的 fd 变量上，xxx_open()函数执行成功返回一个大于 0 的数，失败返回一个小于的 0 的错误码。

如图 5-5 所示， 当访问设备文件结束之后，就需要在应用程序使用 close()函

数释放设备文件，否则下一次启动进程的时候就无法再次访问该设备文件。close()函数会通过操作系统中的内核机制调用设备驱动程序中的 xxx_release()函数完成对底层硬件资源的释放操作，在 open()函数中使用了什么资源就需要在 close 函数中回收什么资源。与 open()函数不同的是，close()函数是由 fd 区分设备文件的，因此 close()函数使用的 fd 变量可以看成是管理 xxx_release()函数中 inode 和 filp 的编

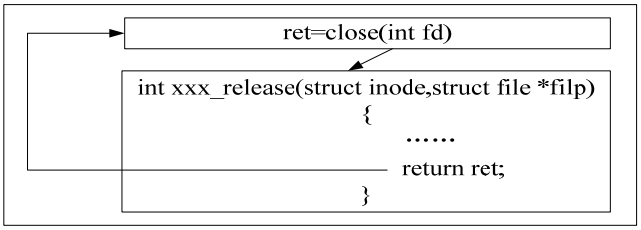

图 5-5 close 函数和 xxx_release 函数之间的调用关系

码，xxx_release()函数返回值被传送到 ret 变量上，只对错误码起作用，因为没有错误时，返回值为 0，返回整数的时候没有实际作用。

如图 5-6 所示，read()函数的 buf 是从设备驱动程序中读取数据的缓存地址，是位于进程空间中的地址，但是设备驱动程序不能直接访问该地址，由内核调用相应的机制将该地址将传送到 xxx_read()函数的 buf 上。xxx_read()函数从硬件中读取数据传递给应用程序的 buf 空间，同时返回读取的数据字节数给 ret 变量。

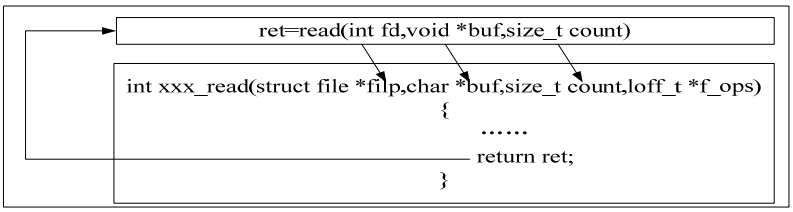

图 5-6 read 函数和 xxx_read 函数之间的调用关系

如图 5-7 所示，buf 是进程中的 write()函数向设备驱动要写入数据的缓存地址，设备驱动程序不能直接访问该地址，该地址被传送到 xxx_write()函数 buf 上，然后设备驱动程序将数据从 buf 拷贝到内核空间，然后写到硬件设备中去，xxx_write()函数和 xxx_read()函数一样，数据写入成功返回写入的字节数给 ret 变量。

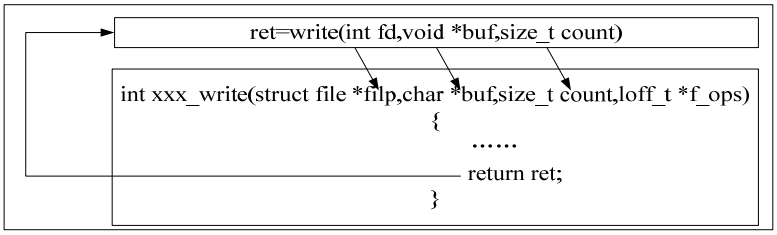

图 5-7 write 函数和 xxx_write 函数之间的调用关系

如图 5-8 所示，ioctl()函数是一个可变变量型函数，其中 request 是一个命令参数，传递到 xxx_ioctl 的 cmd 参数上，当 ioctl 定义三个参数时，xxx_ioctl()的 arg 标志被赋予第三个变量值。

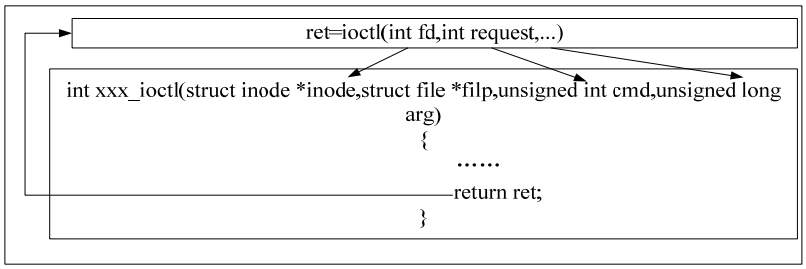

图 5-8 ioctl 函数和 xxx_ioctl 函数之间的调用关系

5.3 电灯驱动程序

5.3.1 电灯控制原理

图 5-9 所示：通过对 D6，D7 这两个发光二极管的开启和关闭来模拟对房间和客厅中的电灯控制， D6 为房间电灯，D7 为客厅电灯，虽然这里的发光二极管和实际电灯的功率驱动电路会有所不同，但是和现实中的电灯控制逻辑是一样的。

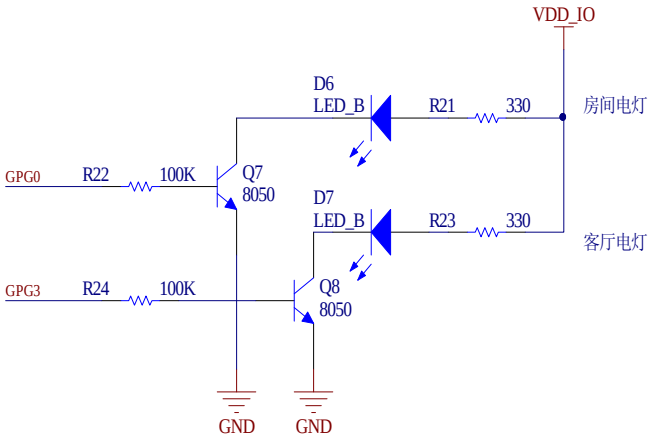

图 5-9 电灯控制电路

其中三极管 Q7，Q8 在这里起着导通开关的作用，当 GPG0 输出高电平的时候，三极管 Q7 导通，从而发光二极管 D6 被点亮；当 GPG0 输出低电平时，三极管 Q7 截止，D6 熄灭。同理当 GPG3 输出高电平的时候，三极管 Q8 导通，发光二极管 D7 被点亮；当 GPG3 输出低电平时，三极管 Q8 截止，D7 熄灭。于是可以得到灯控制的真值表如表 5-1 所示。

表 5-1 电灯控制真值表

GPG0	GPG3	房间电灯（D6）	客厅电灯（D7）
0	0	熄灭	熄灭
0	1	熄灭	点亮
1	0	点亮	熄灭
1	1	点亮	点亮

5.3.2 电灯驱动程序设计

通过前面电灯控制电路的分析，D6 由 GPG0 控制，D7 由 GPG3 控制，由于 S3C2440 控制器不能按位操作，所以需要定义整个 GPG 端口，又在 Linux 内核中访问的都是虚拟地址，因此需要将 GPG 端口寄存器的物理地址空间映射到虚拟地址空间[24]，定义 GPG 端口各控制寄存器如下：

```
static unsigned long gpio_va;//GPIO 从物理地址映射到虚拟地址空间之后的虚拟基地址
#define GPIO_OFT(x) ((x) - 0x56000060)//计算寄存器到基地址的偏移
#define GPGCON (*(volatile unsigned long *)(gpio_va + GPIO_OFT(0x56000060)))
#define GPGDAT (*(volatile unsigned long *)(gpio_va + GPIO_OFT(0x56000064)))
#define GPGDAT (*(volatile unsigned long *)(gpio_va + GPIO_OFT(0x56000068)))
```

同时在模块初始化函数 led_init()中将 GPG(0x56000000)开始的 12 字节的物理地址空间映射到虚拟地址空间并返回虚拟地址空间的起始地址，此后就可以在虚拟地址空间中像操作物理寄存器一样对各寄存器进行设置， 在电灯控制电路中，必须将 GPG0，GPG3 设置为 out 功能，并禁止上拉功能，因此将 GPGCON 寄存器[7:6]设置为 01，[1:0]设置为 01，同时需要将 GPGUP 寄存器的第 0 位和第 3 位设置为 1，并且还需要注册字符设备驱动程序，模块初始化函数功能如下：

```
    static __init led_init(void)
    {
    ……
    gpio_va = (unsigned long)ioremap(0x56000060, 12);//将 0x56000060 开始的 12 字节的物理空间映射到虚拟地址空间，返回虚拟地址空间的起始地址
    GPGCON=GPGCON&(~((0x3<<0)|(0x3<<6)))|((0x1<<0)|(0x1<<6));//GPG0,GPG3 配置为 out 功能
    GPGUP=GPGUP|(1<<0)|(1<<3);//禁止上拉
```

```
    GPGDAT=GPGDAT&(~(1<<0))&(~(1<<3));//所有的灯初始化为熄灭状态分配设备号；
    注册字符设备驱动；
    创建设备节点；
    ……
    }
```

当驱动程序调用完毕从内核空间卸载时需要调用模块卸载函数，在模块卸载函数中需要释放前面映射的虚拟地址空间，并调用 unregister_chrdev()函数卸载字符设备驱动，并且销毁前面创建的设备节点。电灯驱动卸载函数如下所示：

```
    static __exit led_exit(void)
    {
    ……
    gpio_va = iounremap(0x56000060, 0x0c)；//释放前面映射的虚拟地址
    销毁设备节点；
    卸载字符设备驱动；
    }
```

一个驱动程序要使硬件正常工作，需要实现一些 read，write，ioctl 系统调用方法，当应用程序操作设备文件的时候，需要调用驱动程序中这些方法，以实现具体的功能，这里只实现了 open，release，ioctl 方法，首先定义电灯驱动程序的 file_operations 结构体如下：

```
static struct file_operations led_fops=
{
    .owner    =THIS_MODULE,
    .open     = led_open,
    .release  = led_release,
    .ioctl    = led_ioctl,
}
```

在 S3C2440 主控制器上，GPG0，GPG3 通过 GPGCON[0:1]，GPGCON[6:7]寄存器设置为输出口之后，往 GPGDAT[0]，GPGDAT[3]写入 1 则 GPG0，GPG3 输出高电平，往 GPGDAT[0]，GPGDAT[3]写入 0 则 GPG0，GPG3 输出低电平，led_ioctl 功能实现代码如下：

```
static int led_ioctl(struct inode *inode, struct file *file ,unsigned int cmd ,unsigned long arg)
    {
```

```
.....
    switch( cmd )
    {
    caseLED_room://控制房间的电灯
        printk("cmd = %d, arg = %ld\n",cmd,arg);
        if(arg == on)//往 GPGDAT[0]写 1，房间电灯开启
            GPGDAT=GPGDAT|(0x1);
        if(arg == off)//往 GPGDAT[0]写 0，房间电灯关闭
            GPGDAT=GPGDAT&(~(0x1));
    break;
    case LED_living://控制客厅的电灯
        printk("cmd = %d, arg = %ld\n",cmd,arg);
        if(arg == on)//往 GPGDAT[3]写 1,客厅电灯开启
            GPGDAT=GPGDAT|(0x1<<3);
        if(arg == off)//往 GPGDAT[3]写 0,客厅电灯关闭
            GPGDAT=GPGDAT&(~(0x1<<3))
        break;
    default:
        break;
    }
    return 0;
}
```

一个完整的驱动程序还需实现 led_open()函数和 led_release()函数，当应用程序需要打开和关闭设备文件获取设备文件信息的时候，就会通过系统调用进入内核空间调用这里的 led_open()和 led_release 函数，函数执行成功返回 0，失败返回相应的错误码，但在这里设备打开函数和设备关闭函数仅仅是打印了一些调试信息，代码如下：

```
static int led_open(struct inode *inode,struct file *file)
{
    打印一些内核调试信息；
    成功返回 0；
```

```
}
static int led_release(struct inode *inode,struct file *file)
{
    打印一些内核调试信息；
    成功返回 0；

}
```

5.3.3 测试结果

编译内核下载到开发板上，查看已加载的设备，可以看到 LED 的主设备号，说明驱动加载成功。执行电灯测试程序可以看到开发板每一步操作如图 5-10 所示，都会对应 D6，D7 的亮灭。

```
[root@MY2440 led-ioremap]#  ls
Makefile         led.c           led.mod.c       modules.order
Module.symvers   led.c~          led.mod.o       test
led              led.ko          led.o           test.c
[root@MY2440 led-ioremap]#  insmod led.ko
led initialized
[root@MY2440 led-ioremap]#  ls -l /dev/led
crw-rw----    1 root     0        252,    0 Jan  1 00:05 /dev/led
[root@MY2440 led-ioremap]#  ./test
open major=252, minor=0
11 on LED_room
10 off LED_room
21 on LED_living
20 off LED_living
11
a=11
cmd = 1, arg = 1
10
a=10
cmd = 1, arg = 0
21
a=21
cmd = 2, arg = 1
20
a=20
cmd = 2, arg = 0
```

图 5-10 电灯驱动程序测试

5.4 空调驱动程序

在主板上连接了一个直流电动机[25]带动的小风扇用来模拟对空调的操作，当电风扇顺时针匀速旋转的时候，模拟空调的保暖操作，当电风扇顺时针加速旋转的时候，调高空调的保暖温度，当电风扇顺时针减速旋转的时候，降低空调的保暖温度。当电风扇逆时针匀速旋转的时候模拟空调的保冷操作，当电风扇逆时针加速旋转的时候，降低空调的保冷温度，当电风扇逆时针加速旋转的时候，调高空调的保冷温度。当电风扇空转的时候，空调的正常关闭；当电风扇制动的时候，空调快速关闭。直流电机的功能描述如表 5-2 所示。

表 5-2 直流电动机的功能描述

直流电机的旋转状态	模拟空调的功能
正转（顺时针）	保暖
增加顺时针旋转速度	调高保暖温度
降低顺时针旋转速度	降低保暖温度
反转（逆时针）	保冷
增加逆时针旋转速度	降低保冷温度
降低逆时针旋转速度	调高保冷温度
空转	正常关闭
制动	快速关闭

5.4.1 直流电机控制原理

在如图 5-11 所示，当 GPG6，GPG7 全部输出低电平的时候，Q1~Q6 六个三极管全部截止，直流电动机的两端就好像悬空一样，电动机会因为惯性自由旋转，此时的状态为空转状态，不久由于摩擦阻力的存在，电动机逐渐停止下来。

当 GPG6，GPG7 全部输出高电平的时候，Q1~Q7 六个三极管全部导通，直流电动机两端就好像短路一样，如果电动机因为惯性还在转动，则两端产生的电动势在电动机线圈中产生电流，该电流在磁场作用下产生一个与转动方向相反的阻力，从而阻止电动机的转动，起到了制动的作用。

当 GPG6 输出高电平，GPG7 输出低电平的时候，三极管 Q1，Q3，Q6 导通，三极管 Q2，Q4，Q5 截止，这样直流电动机的正极通过三极管 Q1 的集电极和发射极，电阻 R1 连接到电源 VDD5V，负极通过三极管 Q6 发射极和集电极接地，于

是电流经过电源 VDD5V，电阻 R1，三极管 Q1 的发射极和集电极，从直流电动机的正极流向直流电动机的负极，最后经过三极管 Q6 的发射极和集电极（如电路中的红色箭头所示），电动机正转。

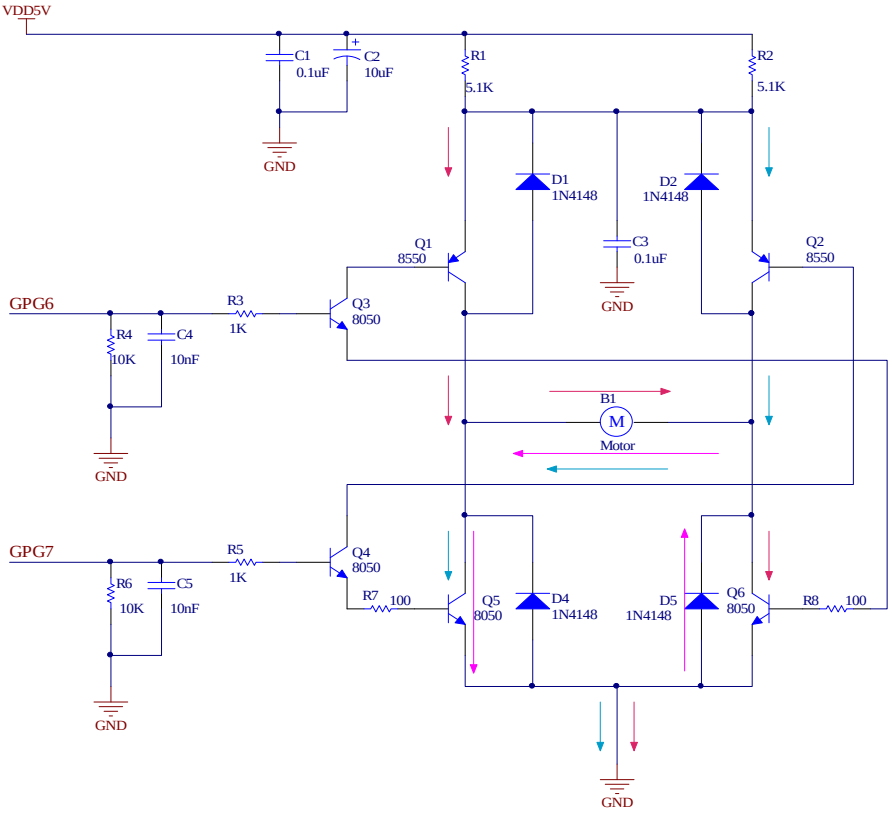

图 5-11 空调控制电路

当 GPG6 输出低电平，GPG7 输出高电平的时候，三极管 Q1,Q3,Q6 截止，三极管 Q2,Q4,Q5 导通，这样直流电动机的负极通过三极管 Q2 的集电极和发射极，电阻 R2 连接到电源 VDD5V，负极通过三极管 Q5 的发射极和集电极接地，于是电流经过电源 VDD5V，电阻 R2，三极管 Q2 的发射极和集电极，从直流电动机的负极流向直流电动机的正极，最后经过三极管 Q5 的发射极和集电极（如电路中的绿色箭头所示），电动机反转。于是可以得到控制直流电动机真值表如表 5-3 所示。

表 5-3 直流电动机的控制电路真值表

GPG6	GPG7	直流电动机的运转状态
1	0	正转
0	1	反转
0	0	空转
1	1	制动

当直流电动机正转的时候，只需要在在 GPG6 上施加一个脉宽可以调节的 PWM 信号，就可以控制直流电动机正转时候的转速，在直流电动机反转的时候，只要在GPG7上施加一个脉宽可以调节的PWM信号就可以控制直流电动机反转时候的转速。

PWM 信号是一种常用的以数字信号控制模拟信号平均电压的方法，通过使用高分辨率的计数器，对矩形的占空比进行脉宽调制，从而形成 PWM 信号。PWM 信号仍然是一个数字信号，在某一时刻，直流电平要么出现，要么不出现。电源以一系列脉冲形式向负载供电，在带宽足够宽的时候，任何模拟信号平均电压都可以由 PWM 信号产生。因此如果想让电动机的转速可控，就可以通过产生不同占空比的 PWM 信号来获得不同平均电压给直流电动机供电，如果需要降低电动机的转速则减少 PWM 信号的占空比，需要增加直流电动的转速则增大 PWM 信号的占空比。

5.4.2 直流电机驱动程序设计

根据直流电机控制电路，GPG6，GPG7 是 S3C2440 控制器上的普通 IO 口，所以需要用模拟的方法来产生 PWM 信号，在驱动程序中可以用内核定时器实现延时来控制 GPG6，GPG7 输出高电平的持续时间来调节 PWM 信号的占空比，从而达到控制直流电动机转速的目的。

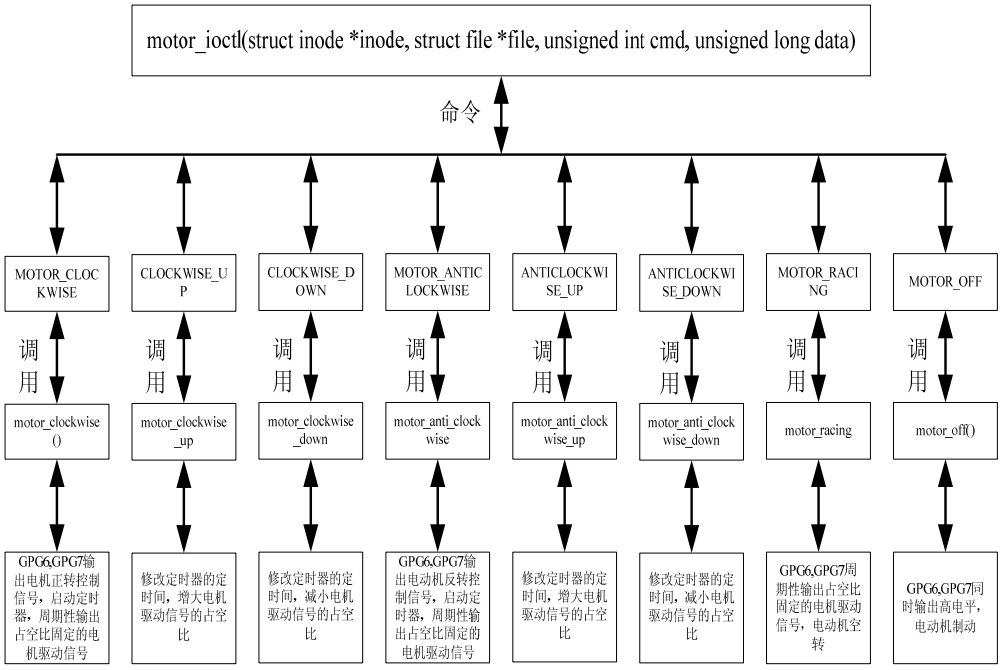

图 5-12 motor_ioctl()的函数调用关系图

在直流电动机设备驱动程序中，重点是实现 ioctl()方法，驱动程序中的motor_ioctl()函数接收来自应用程序的命令，然后根据所接收到的命令调用相应的函数触发电动机工作。一共定义了 8 个命令：MOTOR_CLOCKWISE (顺时针匀速旋转)，CLOCKWISE_UP (顺时针加速旋转)，CLOCKWISE_DOWN (顺时针减速旋转)，MOTOR_ANTICLOCKWISE (逆时针匀速旋转)， ANTICLOCKWISE_UP (逆时针加速旋转)，ANTICLOCKWISE_DOWN (逆时针减速旋转)，MOTOR_RACING (空转)，MOTOR_OFF (制动)。motor_ioctl()中的函数调用关系如图 5-12 所示。

由前面的电路分析可知，要让电动机正转，只需要在 GPG7 输出高电平，GPG6 周期性的输出低电平实现一个 PWM 信号就可以了，直流电机驱动程序中使用内核定时器输出占空比可调的 PWM 信号达到控制直流电机转速的目的，所以需要先在 motor_open()函数中定义并初始化一个内核定时器用于 PWM 信号延时。

根据内核定时器[26]运行过程，使用内核定时器时，先利用 init_timer()函数初始化定义为 struct timer_list 结构体类型的变量，设置结束时间，运行函数，数据后，再利用 add_timer()函数注册到内核上，这样注册的函数形式与 void(*function)(unsigned long)相同，内核中的_run_timer()函数管理利用 add_timer()函数注册的内核定时器，而该函数的运行依赖于以 1/HZ 间隔发生的定时器周期间断，该函数比比较内核定时器的结束时间和当前时间 jiffies 的值，大于或者等于该值的时候运行定时器函数，这样调用的内核定时器函数利用 kerneltimer_handler()函数的“arg”接受由 add_timer()函数注册的内核定时器结构体变量，调用内核定时器函数运行完毕之后，从内核定时器目录中自动删除被注册的内核定时器。在 motor_open()函数中定义并初始化内核定时器如下：

```
static int motor_open(…, …)
{          ……
            mytime.function    = time_tick；//定时器处理函数，需要在定时器处理函数中唤醒睡眠中的进程
            mytime.data = 'delay';//传给定时器处理函数的参数，delay 必须定义为一个全局变量
            init_timer(&mytime);//初始化内核定时器
            //初始化内核定时器的延时为 0.1s 注册内核定时器
            ……
}
```

当 moto_open 函数定义了一个定时器对象，并且通过 init_timer 函数及相应代码对该定时器中的成员 expires，data 和 function 等成员初始化之后，就需要在 motor_

clockwise ()函数和 motor_ anticlockwise ()等函数中启动内核定时器， 当定时器中断到达后，定时器中定义的中断处理函数就会执行。在定时器处理函数中改变电机驱动信号的电平状态，并且根据 motor_ioctl 函数处理的状态修改定和定时器的定时时间，然后重新启动内核定时器，这样就可以达到修改电机驱动信号的占空比来调节电动机转速的目的。图 5-14 是该内核定时器中断处理函数的处理流程。motor_anti_clockwise()函数和 motor_clockwise()函数的实现相同，只要让 GPG6 输出低电平，GPG7 输出高电平；clockwise_down()函数和 clockwise_up()函数的实现相同，只需要降低内核定时器的延时时间。

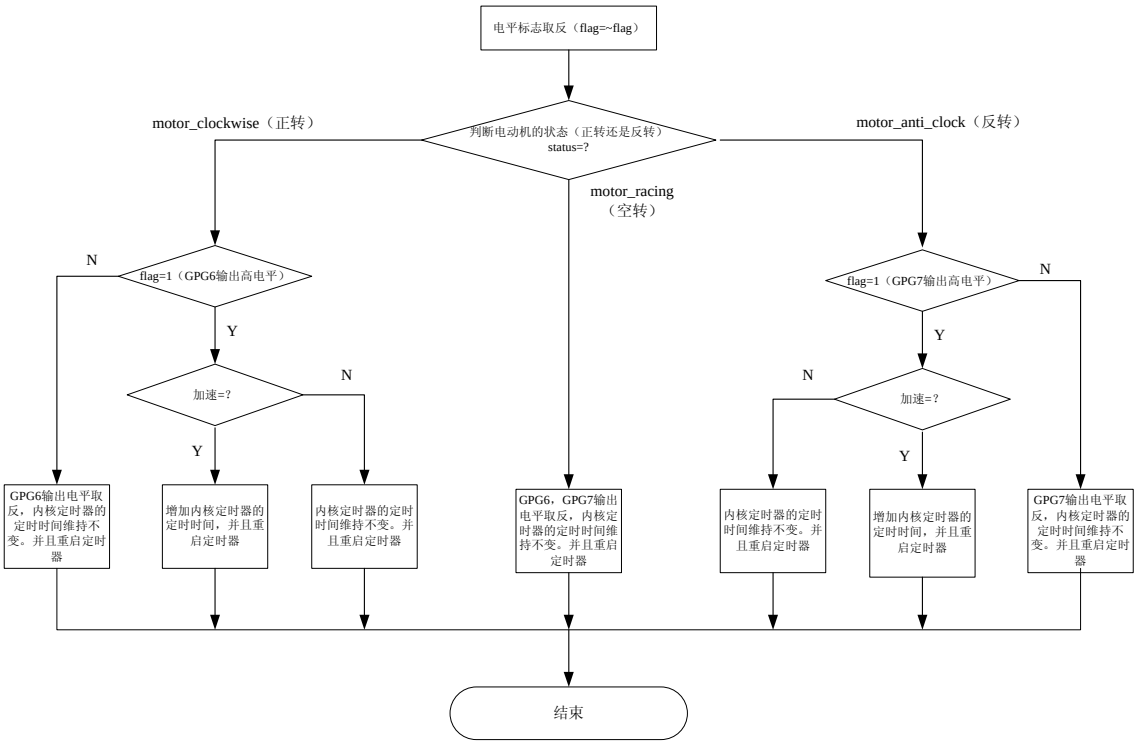

图 5-13 内核定时器中断处理函数

5.4.3 结果分析

加载直流电机驱动程序，运行空调驱动的测试程序如图 5-14 所示，选择 1 可以看到开发板上直流电动机开始顺时针旋转，选择 2 可以看到步进电动机开始逆时针旋转，选择 3 可以看到开发板上的直流电动机开始自由旋转，选择 4，可以看到直流电机立即停止下来。

内核定时器是嵌入式 Linux 设备驱动程序中经常用到的一个重要的内核设施，设备驱动程序中对内核定时器的一个典型使用场景是用它来实现轮询机制，因为

定时器函数自身可以重新启用它所在的定时器，所以在一个定时器到期后，定时器函数被调用，在函数内部因为又重新启用了该定时器，这样便形成了一个不断循环的定时器函数被系统调用的模式，这样就可以用内核定时器来控制引脚高低电平周期变化来模拟 PWM 信号

```
[root@MY2440 motor-timer-modify-updown]#  ls
Makefile           airconcgi.c~       motor.ko           motor_test.c
Makefile~          aircondition.html  motor.mod.c        motor_test.c~
Module.symvers     modules.order      motor.mod.o
airconcgi          motor.c            motor.o
airconcgi.c        motor.c~           motor_test
[root@MY2440 motor-timer-modify-updown]#  insmod motor.ko
MOTOR: chardev register success!
[root@MY2440 motor-timer-modify-updown]#  ls -l /dev/motor
crw-rw----    1 root     0        250,   0 Jan  1 00:04 /dev/motor
[root@MY2440 motor-timer-modify-updown]#  ./motor_test
insmod: cannot insert 'motor.ko': File exists
motor open
please select number to run program
1:MOTOR_CLOCKWISE
2:MOTOR_ANTICLOCKWISE
3:MOTOR_RACING
4:MOTOR_STOP
5:CLOCKWISE_UP
6:ANTICLOCKWISE_UP
7:CLOCKWISE_DOWN
8:ANTICLOCKWISE_DOWN
```

图 5-14 空调驱动程序测试结果

5.5 窗帘驱动程序

系统使用一个步进电动机来模拟窗帘的收起和放下操作，步进电机顺时针旋转，窗帘收起，步进电机逆时针旋转，窗帘放下。步进电机顺时针加速，窗帘加速收起，逆时针加速，窗帘加速放下。同理步进电机减速旋转，窗帘减速收起和放下。步进电机的功能描述如表 5-4 示。

表 5-4 步进电动机的功能描述

步进电机的旋转状态	**窗帘操作**
正转（顺时针匀速）	窗帘卷起
顺时针加速	加速卷起
顺时针减速	减速卷起
反转（逆时针匀速）	窗帘展开
逆时针加速	加速展开
逆时针减速	减速展开

5.5.1 步进电机控制原理

如图 5-16 是窗帘控制模块的电路接线图，U1 是驱动芯片 ULN2003，输入端 1B~7B（1~7 管脚）的信号经过反相后由 1C~7C 输出（10~16 管脚），这个从 ULN2003 输出的信号的驱动能力已经有了很大的提升，足够驱动一般步进电机的励磁线圈。要想让步进电机运转起来，需要依次给励磁线圈 1、2、3、4 通电。也就是依次在控制线 GPB5、GPB6、GPB7、GPB8 上出现高电平，如表 5-3 所示：如果反顺序给励磁线圈通电，则步进电机反转。按表 5-3 所示的操作顺序使步进电机正转，每一时刻只有一个励磁线圈通电，消耗电能较小，但是转矩小且抖动比较大。除了以上这种励磁方式外，还可以让励磁线圈两两通电，比如相邻两个线圈 GPB5，GPB6 同时通电，接着励磁线圈 A 断电，而励磁线圈 B 和 C 通电，然后是励磁线圈 C 和 D，依此类推。这种方法称为 2 相励磁，表 5-4 描述了 2 相励磁的过程，这种励磁方式转矩较大，抖动小，所以应用的比较多。

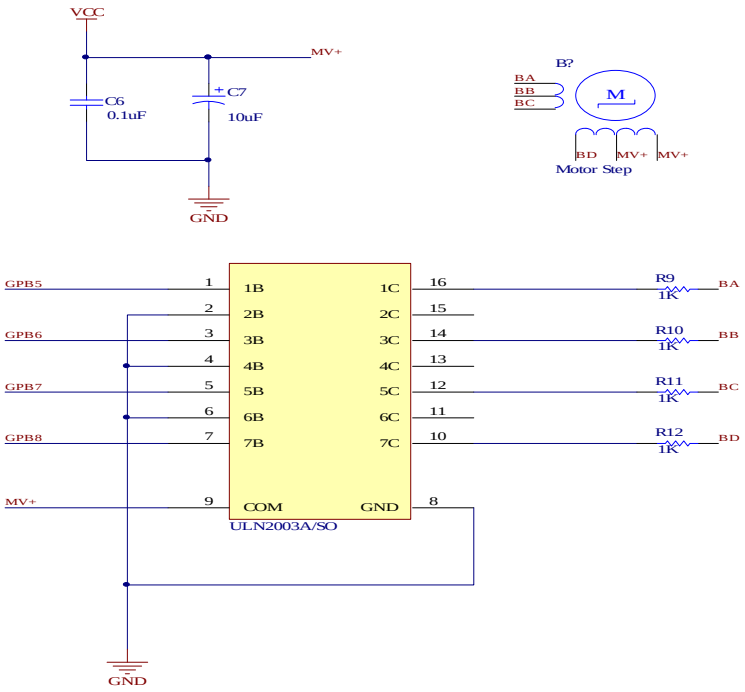

图 5-15 窗帘控制电路

表 5-5 单极步进电动机正转操作顺序表（1 相励磁法）

顺序	GPB5	GPB6	GPB7	GPB8
1	1	0	0	0
2	0	1	0	0

3	0	0	1	0
4	0	0	0	1

表 5-6 单极步进电动机正转操作顺序表（2 相励磁法）

顺序	GPB5	GPB6	GPB7	GPB8
1	1	1	0	0
2	0	1	1	0
3	0	0	1	1
4	1	0	0	1

为了获得更大的分辨率，还可以将以上两种励磁方式结合起来，形成 1-2 相励磁法。表 5-5 描述了 1-2 相励磁法，先让线圈 A 通电，接着励磁线圈 B 加进来，然后励磁线圈 B 加进来，然后励磁线圈 A 断电，随后励磁线圈 C 加进来，依次类推，1-2 相励磁法使步进电机运转平滑，步进小，故被广泛使用。

表 5-7 单极步进电动机正转操作顺序表（1-2 相励磁法）

顺序	GPB5	GPB6	GPB7	GPB8
1	1	0	0	0
2	1	1	0	0
3	0	1	0	0
4	0	1	1	0
5	0	0	1	0
6	0	0	1	1
7	0	0	0	1
8	1	0	0	1

由于步进电机的负载转矩与速度成反比，速度越快负载转矩越小。但速度过快而超过极限后，步进电机将不再转动。

5.5.2 步进电机驱动程序设计

步进电机的转动速度与励磁线圈通电切换的频率有关，所以在每走一步后，程序必须延时一段时间。在窗帘驱动驱动程序中使用 S3C2440 中的定时器 0 中断实现延时以调节 GPG5，GPG6，GPG7，GPG8 输出的控制信号频率，从而达到控制步进电动机转速的目的。

为了处理 IRQ 中断服务[27]，使用 request_irq()函数注册相应的 IRQ 序号和中断服务函数，发生中断后，内核对每一个体系结构都使用固定的 IRQ 检查程序，获取发生中断的 IRQ 序号，并调用 do_IRQ()函数。调用 do_IRQ()函数从 irq_desc 全局变量查找 IRQ 序号对应的已注册的中断服务函数，并调用已注册的函数，此时，使用 request_irq()函数注册的过程中参考的数据地址或参数值会传送到中断服务函数的变量 dev_id 上，此类中断服务函数等待特定的硬件发生中断，硬件发生中断时，在 Linux 内核中运行自身定义的中断处理内容，不做中断处理，利用 free_irq()函数注销注册在 irq_desc 全局变量上的中断服务函数，运行正常时，使用 free_irq()函数之前，注册了中断服务函数的设备驱动程序会处理中断，防止硬件再次发生中断。

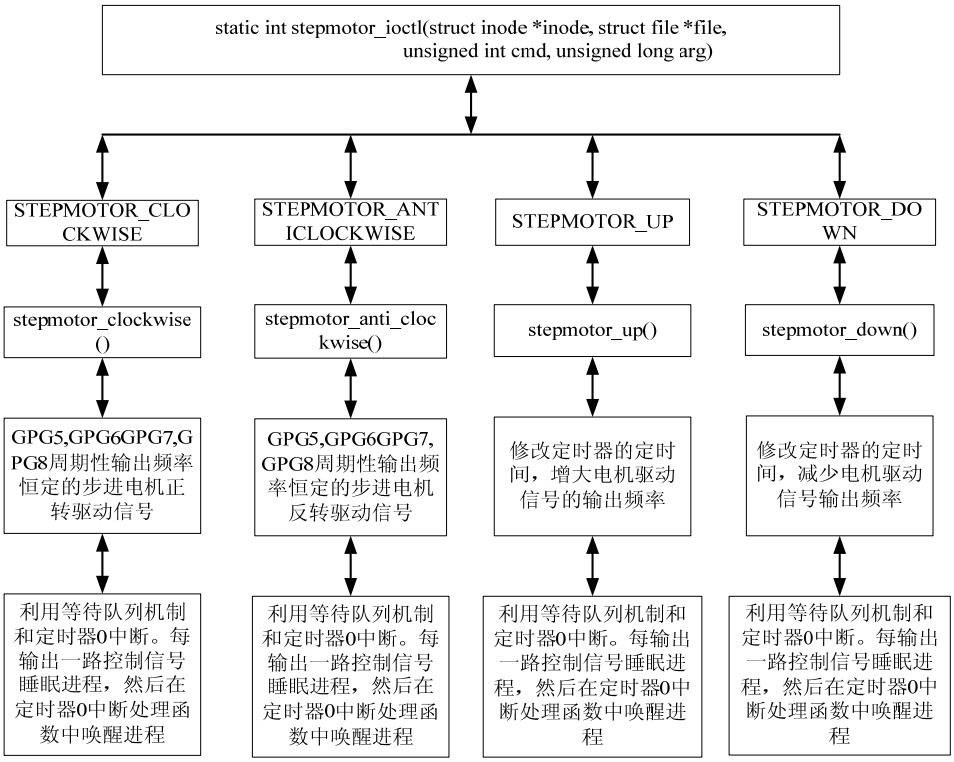

图 5-16 stepmotor_ioctl 函数的调用关系图

步进电机驱动程序主要是向步进电机发送控制信号，所以没有实现 write，read 这两个向设备文件读写数据的函数，这里只实现了 open，close，ioctl 方法。定义 file_operations 结构体如下。

```
static struct file_operations stepmotor_fops = {
        owne     =THIS_MODULE,
        open     =step_motor_open,
```

```
    ioctl     =step_motor_ioctl,
    release =step_motor_release,
};
```

step_motor_ioctl 是整个驱动程序的关键，它接收来自应用程序的控制命令，然后向步进电动机控制芯片发送控制信号，step_motor_ioctl 函数的实现功能如图 5-16 所示。motor_clockwise()是输出驱动步进电机旋转控制信号的具体实现，这里采用 1-2 相励磁法来控制电动机的转向和转速，由真值表 5-7 可知，电动机 1-2 相励磁法正转的时序为 AB-B-BC-C-CD-D-DA-A，依次在 GPG5，GPG6，GPG7，GPG8 输出各路控制信号即可以达到驱动电动机顺时针旋转的目的。每输出一路控制信号，步进电机就向前推进一个步进角，如果在驱动程序中每输出一路控制信号，就休眠进程，直到定时器中断到来时再唤醒进程，再输出下一路控制信号，这样就可以完全控制步进电机驱动信号的频率而达到调节步进电机旋转速度的目的。motor_clockwise()函数的实现如下。

```
static void stepmotor_clockwise(void)
{
    int i;
        for(i=0; i<40; i++) {
            ……
        在 GPG5，GPG6，GPG7，GPG8 上输出控制信号 AB；
            wait_event_interruptible(timer_wait,ready);//休眠进程，直到定时器 0 中断到来唤醒进程，接着输出下一路控制信号
        ……
        在 GPG5，GPG6，GPG7，GPG8 上输出控制信号 B；
            wait_event_interruptible(timer_wait,ready); //休眠进程，直到时器 0 中断到来唤醒进程，接着输出下一路控制信号


        ……
        在 GPG5，GPG6，GPG7，GPG8 上输出控制信号 BC；
            wait_event_interruptible(timer_wait,ready); //休眠进程，直到时器 0 中断到来唤醒进程，接着输出下一路控制信号
        ……
        在 GPG5，GPG6，GPG7，GPG8 上输出控制信号 C；
```

```
                wait_event_interruptible(timer_wait,ready); //休眠进程，直到时器 0 中断到来
唤醒进程，接着输出下一路控制信号
            ……
              在 GPG5，GPG6，GPG7，GPG8 上输出控制信号 CD；
                wait_event_interruptible(timer_wait,ready); //休眠进程，直到时器 0 中断到来
唤醒进程，接着输出下一路控制信号
           ……
              在 GPG5，GPG6，GPG7，GPG8 上输出控制信号 D；
                wait_event_interruptible(timer_wait,ready); 在 GPG5，GPG6，GPG7，GPG8
上输出控制信号
              ……
              在 GPG5，GPG6，GPG7，GPG8 上输出控制信号 DA；
                wait_event_interruptible(timer_wait,ready); //休眠进程，直到时器 0 中断到来
唤醒进程，接着输出下一路控制信号
    ……
              在 GPG5，GPG6，GPG7，GPG8 上输出控制信号 A；
                wait_event_interruptible(timer_wait,ready); //休眠进程，直到时器 0 中断到来
唤醒进程，接着输出下一路控制信号
            ……
             }
    }
```

进程休眠之后，必须要在中断处理函数需要重新唤醒进程，同时中断处理函数在这里还需要有根据应用程序的需要(如需要调节步进电机转速的时候)重新修改定时器的定时时间的本领，代码如下所示。

```
    static irqreturn_t timer0_irq_handle(int irq, void *devid)
    {
        ready=1;
        唤醒等待队列中的进程，在 stepmotor_clockwise 函数中休眠的进程在这里每唤醒

        if(flag !=0) {
                    当收到加速/减速命令时，重新修改定时器的定时时间即将新的计数值写
入 TCNT0,并重启定时器 0
```

```
                flag = 0;
            }
        return IRQ_HANDLED;
    }
```

timer0_irq_handle 函数具体执行流程如图 5-17 所示

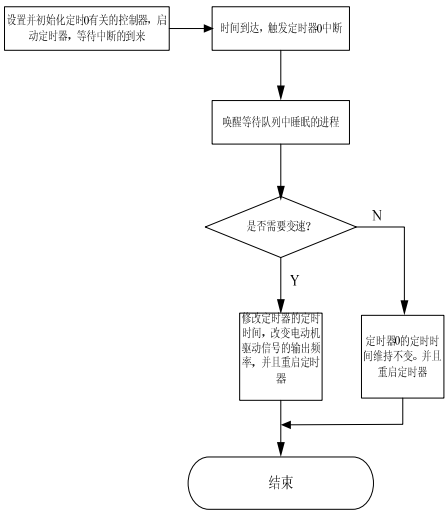

图 5-17 timer0_irq_handle 函数的执行流程

由于这里用到定时器 0 中断，所以需要在 step_motor_open 函数中注册中断，并初始化定时器 0。step_motor_open 函数的实现如下：

```
    static int stepmotor_open(struct inode *inode, struct file *filp)
    {
        配置 GPG5，GPG6，GPG7，GPG8 引脚功能;
        ret = request_irq(IRQ_TIMER0, timer0_irq_handle, IRQF_DISABLED,
DEVICE_NAME, NULL);//向内核注册定时器 0 中断
        ……
        new_value=600;//设置定时器的初始值
        ……
        timer0_init();        //初始化定时器 0

        return 0;
    }
```

在 stepmotor_open 函数中注册中断就必须在 step_motor_release 函数中释放前

面注册中断，代码如下：

```
static int step_motor_release(struct inode *inode, struct file *filp)
{
    ……
    free_irq(IRQ_TIMER0, NULL);//驱动程序调用完毕，释放前面注册的中断
    return 0;
}
```

另外还有加速函数 stepmotor_speedup()和减速函数 stepmotor_speedown()实现起来非常简单，只需要修改定时 0 的定时器时间，也即向定时器 0 中的 TCNT0 寄存器写入新的计数值，当加速时增大 TCNT0 的值，减速时减少 TCNT0 的值，就可以改变步进电机控制信号的输出频率，从而改变步进电机的转动速度。

5.5.3 结果分析

处理器的速度远快于外设的速度，因此不希望一直等待外设，嵌入式 Linux 中提供了中断处理机制，当进程不需要占用 CPU 资源的时候，可以将 CPU 礼让给其它进程，只有在外设需要处理器的时候才通知处理器在中断到来的时候重新唤起进程。这样避免了用户进程向获取资源只能不停的查询，从而无谓的浪费 CPU 资源。

```
[root@MY2440 driver]#  insmod stepmotor_driver.ko
s3c2410  motor_driver init
[root@MY2440 driver]#  cd ..
[root@MY2440 stepmoto]#  ls
driver  test
[root@MY2440 stepmoto]#  cd test/
[root@MY2440 test]#  ls
Makefile              curtain1.c              stepmotor_driver.mod.o
Makefile~             driver                  stepmotor_driver.o
Module.symvers        modules.order           stepmotor_test
curtain.c             stepmotor_driver.c      stepmotor_test.c~
curtain.cgi           stepmotor_driver.ko
curtain.html          stepmotor_driver.mod.c
[root@MY2440 test]#  ls -l /dev/stepmotor
crw-rw----    1 root     0      250,   0 Jan  1 00:26 /dev/stepmotor
[root@MY2440 test]#  ./stepmotor_test
insmod: cannot insert 'stepmotor_driver.ko': File exists
stepmotor: opened.
Please select how to operate the step motor...
1 : clockwise
2 : anti_clockwise
3 : speedup,then select 1 or 2 again!
4 : speeddown, then select 1 or 2 again!
5 : exit!
1
2
^Cstepmotor: released.
```

图 5-18 步进电机驱动测试结果

如图 5-18 所示，选择 1 可以看到开发板上步进电动机开始顺时针旋转，选择 3 可以看到步进电动机开始加速，选择 2 可以看到开发板上的电动机开始逆时针旋转，选择 4，可以看到步进电机旋转速度减慢。

5.6 温湿度驱动程序

系统使用一个 DTH11 数字温湿度传感器来模拟对室内的温湿度监控操作，它对实时监测和调控家居环境有着非常重要的意义，比当用户在回家的路上，通过移动电话打查看到家里的温湿度超过主任的舒适范围时，就可以打开空调将家庭内的温湿度调节到一个合理的程度，使自己一回到家中就有一个温馨舒适的环境。

5.6.1 温湿度传感器的数据传输

如图 5-19 为 DHT11 温度传感器和 S3C2440 的电路接线图，它的连接方法极为简单。第一脚接电源正，第四脚接电源地，电源引脚（VDD,GND）之间用一个 100nF 左右的耦合电容用于去藕滤波。第三脚为空脚，此管脚悬空不用。数据端为第二脚，直接接主机 GPG11 端口口。为提高稳定性，在数据端和电源正之间接一只约 4.7K 的上拉电阻。

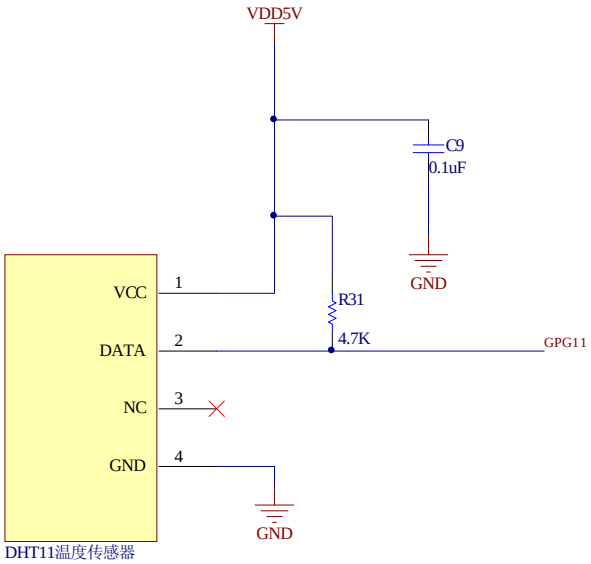

图 5- 19 DTH11 电路接线图

总线空闲时处于高电平状态，当主控制器发送开始信号时可以随时通过拉低总线电平来实现，DHT11 传感器的供电电压为 3.3V,上电后 1 秒左右，传感器将处于稳定工作状态，在此之前不接收任何主控制器发送的指令，电源引脚（VDD,GND）之间必须用一个 100nF 左右的耦合电容用于去藕滤波。DATA 信号线是主控制器器与 DHT11 之间数据和命令的传输通路，是一线总线接口，一次通讯的时间大概为 0.4s，一次数据传输为 5 个字节，高位先出，分为小数部分和整数

部分，数据格式为：1 个字节的湿度整数数据+1 个字节的湿度小数数据+1 个字节的温度整数数据+1 个字节的温度小数数据，最后一个字节是温湿度数据的校验和。其中正确传送时校验和数据等于“8 位湿度数据、8 位湿度小数数据、8 位温度整数数据、8 位温度小数数据算术和”的低 8 位。主机发出 start 信号后，DHT11 自动从低功耗模式切换到 speed 模式，主机开始信号发送完毕之后，DHT11 响应主机之后紧接着往总线上送出 5 个字节的数据并触发下一次数据的采集。用户可以从总线上接收部分数据或者全部数据，从模式下，如果 DHT11 没有接收到主机发出的启动信号，DHT11 不会主动触发温度采集，采集数据后就自动转换到 slow 模式。通讯过程如下：

(1)主机复位信号和 DHT11 响应信号

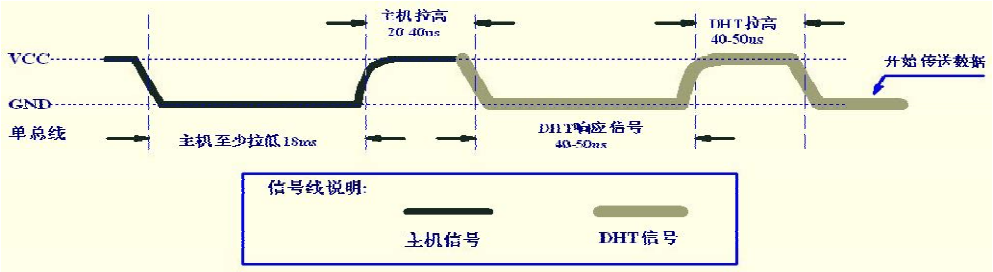

图 5-20 DHT11 复位时序

如图 5-20 所示，用户主机发送一次开始信号（低电平），DHT11 从低速模式转换到高速模式，等待主机开始结束（拉高）后，DHT11 发送响应信号，送出 40bit 的数据，并触发一次信号采集，用户可选择读取部分或全部数据。注意：总线空闲状态为高电平，主机把总线拉低等待 DHT11 响应，主机把总线拉低必须大于 18 毫秒，保证 DHT11 能检测到起始信号。DHT11 接收到主机的开始信号后，等待主机开始信号结束，然后发送低电平响应主机。主机发送开始信号结束后，延时等待 20-40us 后，读取 DHT11 的响应信号，主机发送开始信号后，可以切换到输入模式，或者输出高电平均可，总线由上拉电阻拉高。

(2)DHT11 开始发送数据流程

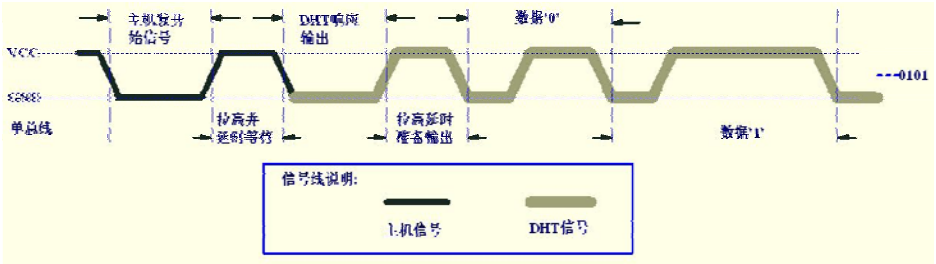

图 5-21 读 DHT11 数据流程

如图 5-21 所示，主机发送开始信号后, 延时等待 20us-40us 后读取 DH11T 的回应信号，读取总线为低电平, 说明 DHT11 发送响应信号，DHT11 发送响应信号后，再把总线拉高, 准备发送数据, 每一 bit 数据都以低电平开始。如果读取响应信号为高电平, 则 DHT11 没有响应, 请检查线路是否连接正常。

(3)数字‘0’信号表示方法

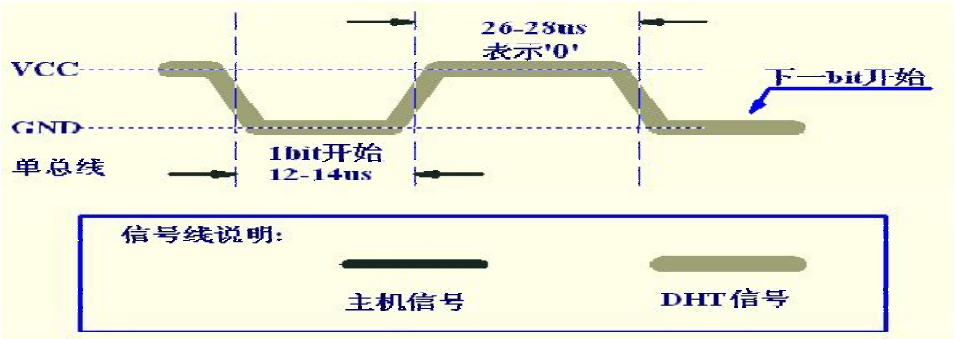

图 5-22 信号‘0’时序图

如图 5-22 所示，数字‘0’表示方法为，首先 DHT11 把总线拉低 12-14us 然后拉高，高电平保存时间在 26-28us 这个范围内。则此比特为‘0’电平。

(4) 数字‘1’信号表示方法

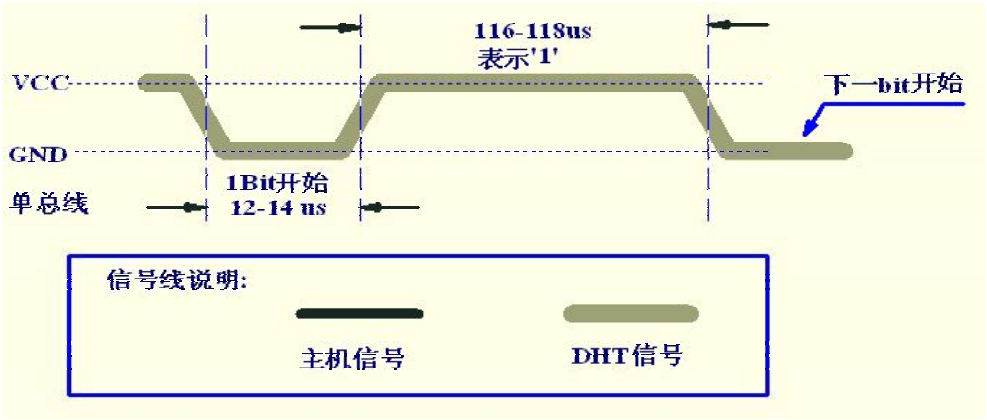

图 5-23 信号‘1’时序图

如图 5-23 所示，数字‘1’表示方法为，首先 DHT11 把总线拉低 12-14us 然后拉高，高电平保存时间在 116-118us 这个范围内。则此比特为‘1’电平。

5.6.2 温湿度驱动程序设计

由于不需要向设备发送命令和数据，所以在 dht11 驱动程序中只需要实现 dht11_read()函数，dht11_open()函数，dht11_release()几个函数。dht11 驱动程序的框架如图 5-24 所示。

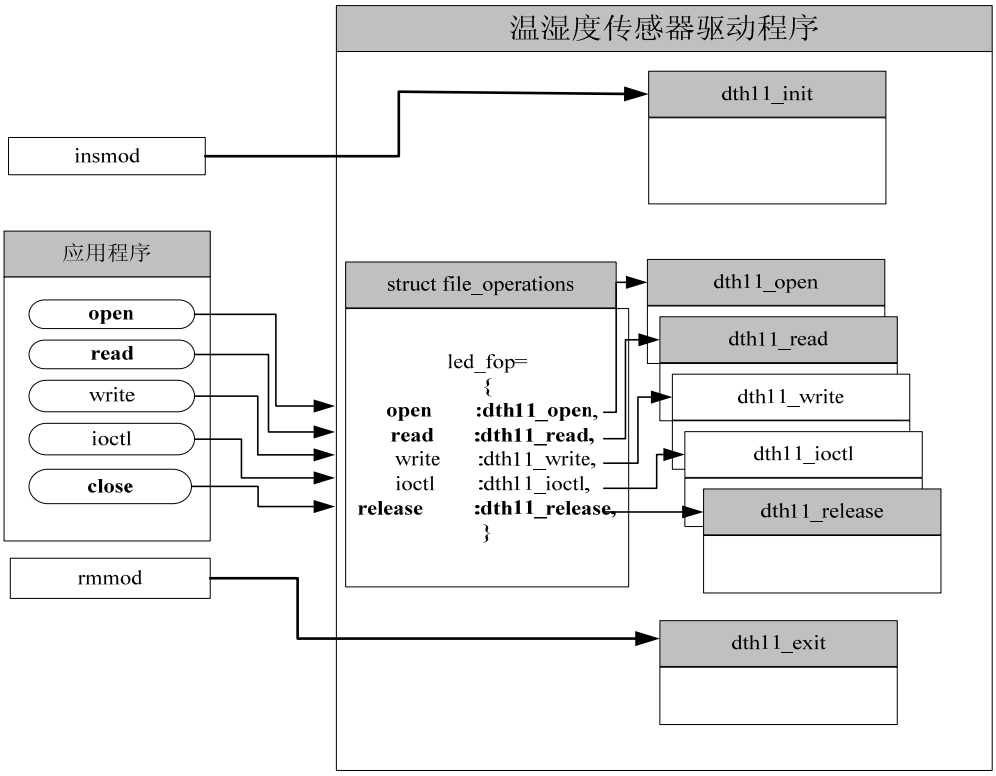

图 5-24 dht11 驱动程序框架

所以定义 dht11 驱动程序的 file_operation 结构体如下：

```
static struct file_operations dht11_fops = {
    .owner = THIS_MODULE,
   .open = dht11_open,
   .read = dht11_read,
    .release = dht11_release,
    };
```

接下来把 DHT11 定义成一个杂项设备，可以省去内核分配主设备号的麻烦

```
static struct miscdevice dht11_misdev = {
    .miNOR = MISC_DYNAMIC_MINOR,
    .name = DEVICE_NAME,
    .fops = & dht11_fops,
    };
```

在模块初始化函数中注册杂项设备驱动

```
static int __init dht11_init ( void )
{
```

```
        ……
        misc_register(&dht11_misdev);//使用杂项设备注册 DTH11 设备驱动;
        ……
        return 0;
    }
```

同样还需要在模块卸载函数中卸载杂项设备驱动

```
    static void __exit dht11_exit ( void )
    {
        ……
        misc_deregister(&dht11_misdev);//卸载 DTH11 设备驱动;
        ……
    }
```

在本驱动中关键是实现主机从 DTH11 设备读取数据的函数，根据 DHT11 发送数据的时序图，在 DHT11_read()函数中将实现一个完整的取数据的过程。

```
static ssize_t dth11_read ( struct file* filp, char __user* buf, size_t count, loff_t* f_pos )
{
        主机将总线拉低；
        延时 20ms
        主机将总线拉高；
        延时 20~40us；
        等待从机响应；
        从机响应错误{返回错误码}；
        从机响应正确
        {
            调用字节获取函数，读取 5 个字节的数据保存到内核缓冲区；
            返回读取的字节数
        }
        主机将总线拉高，使总线重新处于空闲状态

}
```

整个 dht11_read ()函数的程序流程图如下

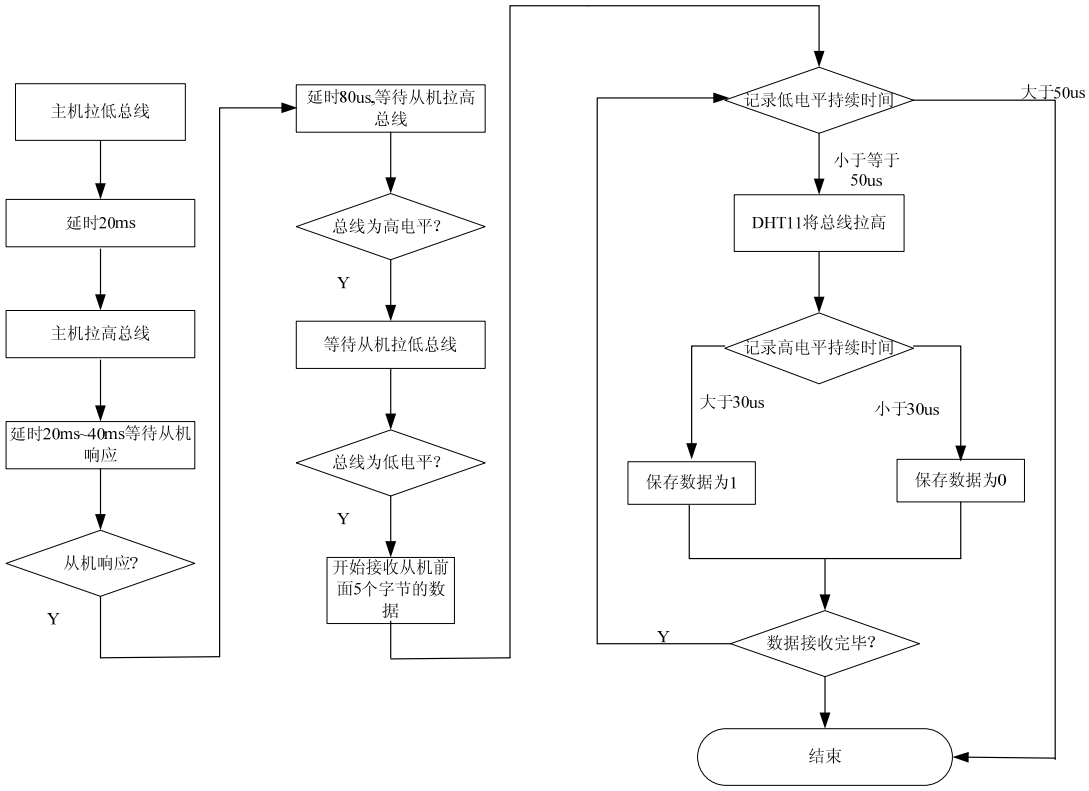

图 5-25 dht11_read ()函数的程序流程

5.6.3 结果分析

温湿度环境监控模块驱动程序的测试结果 5-26 所示，加载 dht11 驱动程序，运行测试程序，可以看到控制终端上正实时打印出从温湿度传感器中采集到的温湿度数据。

```
[root@MY2440 humidity-dht11]#  ls
Makefile          humidity.c        humidity.o        humtest
Makefile~         humidity.ko       humidity.pro      modules.order
Module.symvers    humidity.mod.c    humidity_test     readme.txt
form              humidity.mod.o    humidity_test.c
[root@MY2440 humidity-dht11]#  insmod humidity.ko
dht11    initialized
[root@MY2440 humidity-dht11]#  ls -l /dev/dht11
crw-rw----    1 root     0          10,   60 Jan   1 00:42 /dev/dht11
[root@MY2440 humidity-dht11]#  ./humidity_test
open in kernel
/dev/humidiy open suc
humidity = 42.0
temp = 21.0
humidity = 44.0
temp = 22.0
humidity = 44.0
temp = 22.0
humidity = 44.0
temp = 22.0
humidity = 44.0
temp = 22.0
```

图 5-26 温湿度驱动程序测试结果

实践证明：DHT11 的温湿度采集模块具有电路简单，方便适用的，采集的数据实时准确等优点，将此方案应用在基于嵌入式 Linux 操作系统和 ARM 平台的温湿度环境监测，模块运行正常，方案可行有效，在环境测量准确度和系统实时性方面得到令人满意的效果。

5.7 空气检测驱动程序

当用户在出差或旅游途中，可以在世界任何能上 Internet 的地方，了解家庭中的空气指标情况，比如用户想知道家庭鱼缸中的空气指标超过小金鱼的生存范围时，就可以打开烟雾吸收器吸除掉空气中对鱼类有害的物质，让用户在任何时候对家都有一个安心的心情。

5.7.1 空气检测模块的工作原理

如图 5-27 是空气检测模块的电路接线图，其中 MQ-2 是一种体电阻控制型的气敏器件，当室内空气的浓度发生变化（如烟雾增多）时，引脚 2 与引脚 5 之间的阻值随气体浓度在一定范围内呈线性变化。引脚 6 和引脚 4 连接到 S3C2440 的 AIN0 引脚上，通过 S3C2440 内部集成的 AD 转换模块将 MQ-2 的阻值变化情况实时转换成数字信号，已达到将被测气体的浓度信号转变成电信号的目的。

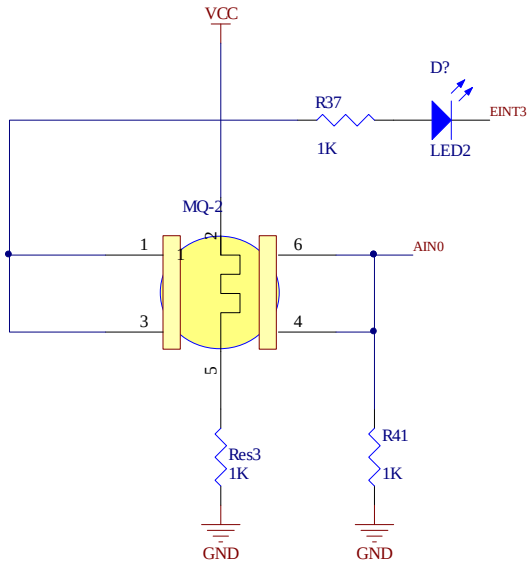

图 5-27 空气检测模块电路

5.7.2 空气检测驱动程序设计

S3C2440 内部集成的 AD 转换模块有两种读取数据的方式，一种是查询方式，另一种是中断方式，由于监控系统设计的方式是只有接收到用户的空气检测命令时，才会通知空气检测模块对室内的空气质量进行检测，所以驱动程序选用中断方式来读取 AD 转换的数据，这里重点是实现数据读取 adc_read()及相关的函数，如图 5-28 是 adc_read()函数的执行流程。

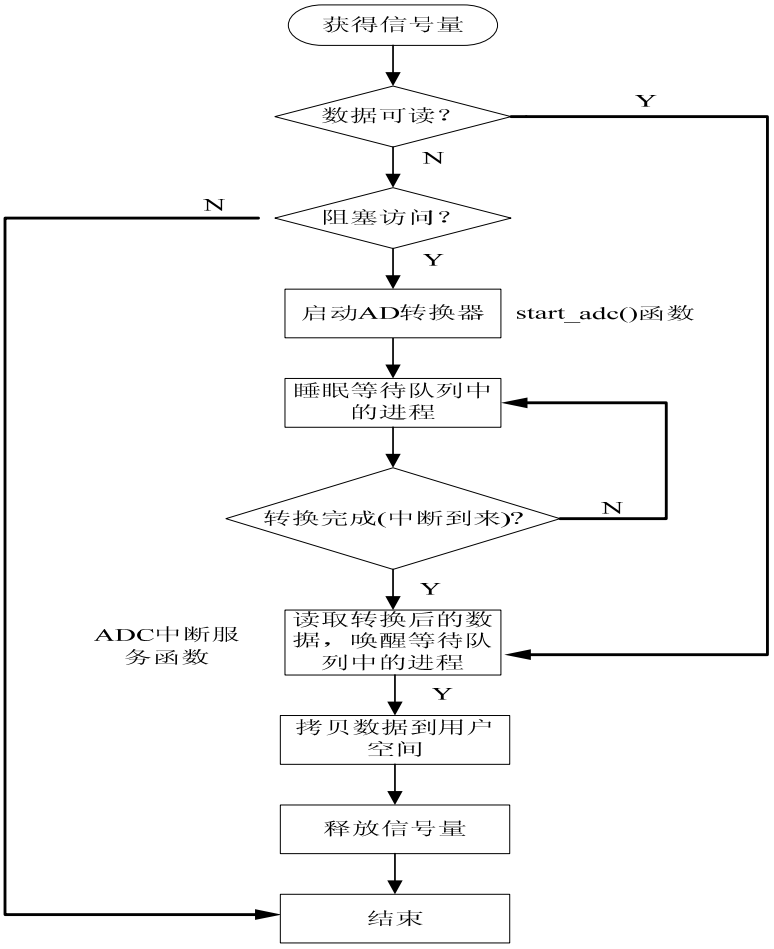

图 5-28 AD 转换驱动程序的执行流程

AD 转换器从 APB 总线分频之后获得工作频率，PCLK=50MHz， AD 转换器的频率 AD_{freq}=PCLK/PRSCVL+1， PRSCVL 取最大值 prescaler =255，AD 转换通道设为 ch=0，根据 S3C2440 的芯片手册在 start_adc ()函数中设置 ADCCON 如下：

```
ADCCON=(1<<14)|(prescaler<<6)|(ch<<3);
//配置 ADC 为：允许预分频，分频值为 prescaler,AD 转换通道为 ch
```

当多个进程对共享资源的并发访问时，会导致竞争，Linux 提供了多种解决竞

争问题的方法，这里采用信号量的方式用于保护 AD 转换后得到数据，只有拿到信号量的进程才能访问和读取 AD 数据寄存器中的数据。另外当运行中的进程要获得某一资源而不可得时，进程有时候需要等待，此时可以在驱动程序中使进程进入睡眠状态，内核为此生成一个新的等待队列节点将睡眠中的进程挂载到等待队列中。在驱动程序中分别定义并初始化信号量和等待队列如下：

```
DECLARE_MUTEX(ADC_LOCK);
static DECLARE_WAIT_QUEUE_HEAD(adc_waitq);
```

另外因为这里用到了 ADC 中断，所以需要在 adc_init()函数中注册 ADC 中断，由于内核中访问的都是虚拟地址，同样需要将 ADC 控制器中用到的寄存器物理地址映射成虚拟地址。adc_init()函数所做的初始化工作如下：

```
static int __init adc_init(void)
{
        ……
        adc_clk=clk_get(NULL,"adc");//从平台时钟队列中获取 ADC 的时钟，因为 ADC 的转换频率和系统时钟有关
       …….
        clk_enable(adc_clk);//使能 ADC 转换时钟
       ……
        adc_base=ioremap(0x58000000, 64);//物理地址到虚拟地址的映射
       ……
        ret=request_irq(IRQ_ADC,adc_irq,IRQF_DISABLED,DEVICE_NAME,1);
        //向内核注册 ADC 中断
       错误处理；
       成功返回 0；
}
```

5.7.3 结果分析

如图 5-31 是空气检测驱动程序的测试结果，查看/dev 目录可以看到，内核已经自动创建了 adc1104070 设备节点，主设备号为 10，从设备号为 60，表明驱动程序加载成功。运行测试程序以后可以看到控制终端上正实时打印出从气体传感器中采集到的空气数据。

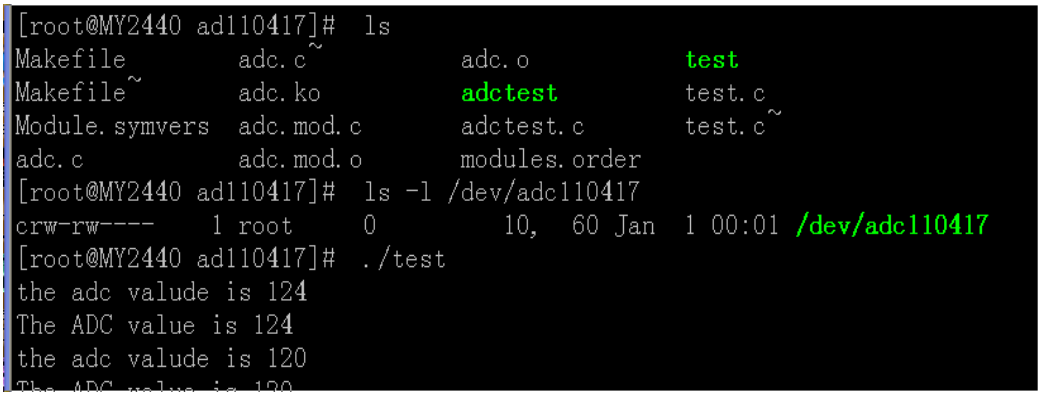

图 5- 29 空气检测驱动测试结果

第六章 系统应用程序开发及性能测试

6.1 引言

CGI 是通用网关接口（Common Gateway Interface）的简称，是 WEB 服务器和其他应用程序信息交互的标准接口。WEB 服务器获取客户端提交的控制请求，然后转交给 CGI 程序，CGI 程序从 WEB 服务器得到一组命令 URL 编码，解码之后再调用其它其他程序处理，最后将结果返回给 WEB 服务器，由服务器将最终处理结果返回给客户端。

CGI 通信系统通常都是两部分组成：一部分是 html 页面，就是在用户端浏览器上显示的页面；另一部分则是运行在服务器上的 CGI 程序。它们之间的通讯方式如图 6-1 所示，服务器和客户端之间的通信实际上是客户端的浏览器和服务器端的 http 服务器之间的 HTTP 通信，CGI 程序只需要知道浏览器请求执行服务器上哪个程序就可以了，其他不必深究细节，因为这些过程不需要在 CGI 中操作。服务器和 CGI 程序之间的通讯才是我们关注的。一般情况下，服务器和 CGI 程序之间是通过标准输入输出来进行数据传递的，而这个过程需要环境变量的协作方可实现。

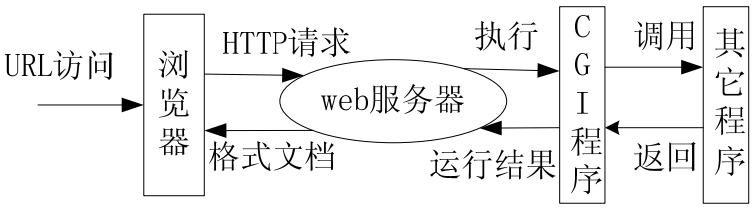

图 6-1 CGI 工作流程示意图

6.2 系统应用程序设计

根据 CGI 程序的工作原理，系统应用程序主要分两部分组成：静态表单页面设计和动态 WEB 页面设计，静态页面的设计用 HTML 来实现，动态页面设计用 CGI 技术实现。本系统应用程序的框架结构如图 6-2 所示。

网络客户端访问服务器时，访问的第一个文件是 index.html，当用户进入该页面时，将弹出一个对话框要求用户进行身份验证，由 login.cgi 对用户的登陆信息

进行验证判断，只有通过认证之后才允许进入系统的主页面 home.html，在 home.html 页面里为用户提供三种功能：

(1)网络家电控制

(2)家庭温湿度环境，空气质量查看

(3)网络视频监控

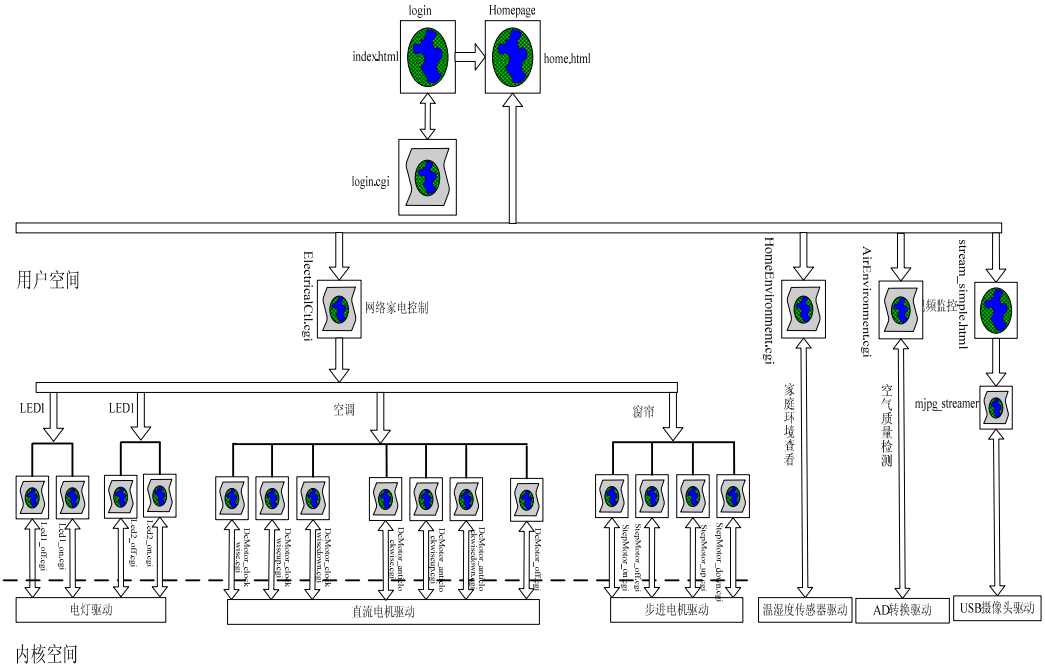

图 6-2 系统应用程序结构框图

其中家庭温湿度环境查看工作由 HomeEnvironment.cgi 程序响应，网络视频监控由 mjpeg_streamer 响应，网络家电控制由 ElectriclCtl.cgi 程序响应，对客厅电灯，房间电灯，空调、窗帘的控制分别由 led1_on.cgi 等程序响应。服务器主要页面文件及应用程序功能如表 6-1 所示。用户在 index.html 页面发出的查询和控制请求最终会通过 WEB 服务器转发给各 CGI 程序，CGI 程序解码后将命令发给底层驱动程序，驱动程序再将硬件处理结果返回给 CGI 程序，CGI 程序再将信息交给 WEB 服务器从而传给网络控制终端。

表 6-1 系统页面文件功能说明

序号	程序名	功能描述
1	index.html	用户登陆界面
2	login.cgi	登陆信息判断程序
3	home.html	监控系统主页

4	led1_on.cgi	房间电灯开启程序
5	led1_off.cgi	房间电灯关闭程序
6	Led2_on.cgi	客厅电灯开启程序
7	Led2_off.cgi	客厅电灯关闭程序
8	DcMotor_clockwise.cgi	直流电机顺时针旋转(空调保暖)
9	DcMotor_ clockwiseup.cgi	增大顺时针旋转速度(增加保暖温度)
10	DcMotor_ clockwisedown.cgi	降低顺时针旋转速度(降低保暖温度)
11	DcMotor_anticlockwise.cgi	直流电机逆时针旋转(空调保冷)
12	DcMotor_ anticlockwiseup.cgi	增大顺时针旋转速度(降低保冷温度)
13	DcMotor_ anticlockwisedown.cgi	降低顺时针旋转速度(增加保冷温度)
14	DcMotor_off.cgi	直流电机关闭程序(空调关闭)
15	StepMotor_on.cgi	步进电机开启程序(打开窗帘)
16	StepMotor_off.cgi	步进电机关闭程序(收起窗帘)
17	StepMotor_up.cgi	步进电机加速程序(窗帘加速)
18	StepMotor_down.cgi	步进电机减速程序(窗帘减速)
19	AirEnvironment.cgi	家庭温空气环境检测程序
20	HomeEnvironment.cgi	家庭温湿度环境监控程序
21	mjpeg_streamer	视频采集程序

6.2.1 系统主页面设计

用户登录是保障系统安全性的一个重要方面，本嵌入式 WEB 服务器采用 get 方式传输信息，这样用户能够通过 WEB 页面提交用户名和密码，WEB 服务器对提交的信息进行验证，用 HTML 语言编写 WEB 页面及表单是实现动态 WEB 页面的第一步，在编写 WEB 页面时要通过 ACTION 属性来指明对应 CGI 程序的存放位置，如 ACTION=cgi-bin/login.cgi，用过 METHOD 属性来指明提交数据的方法，

如 METHOD=“get”,本系统的登陆界面如图 6-3 所示。

图 6-3 系统的登陆界面

要实现动态 WEB 页面，还需要用 C 编写 CGI 程序，将编写好的 CGI 程序编译成二进制文件存放在 cgi-bin 目录下，login.cgi 程序对表单数据行处理，进而判断输入的用户名和密码是否正确。若正确，启动监控系统的主页。本系统中登陆界面的 CGI 程序的执行过程如图 6-4 所示。

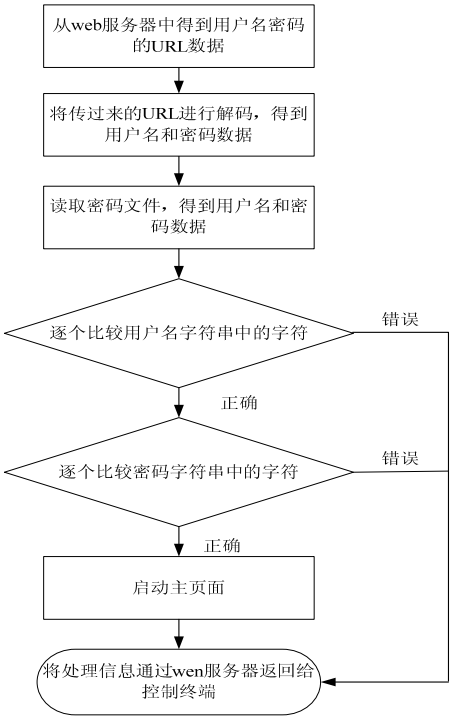

图 6-4 login.cgi 程序处理过程

由于本表单中只有两个输入框，如图所示，WEB 服务器收到的 URL 格式的信息将是 UserName=valuel&Password=value2。通过上面的 pass.cgi 程序就可以提取 valu 和 value2 的具体内容，也就是用户输入的用户名和密码的具体值，对其验证后启动监控系统的主页面，如图 6-5 所示。

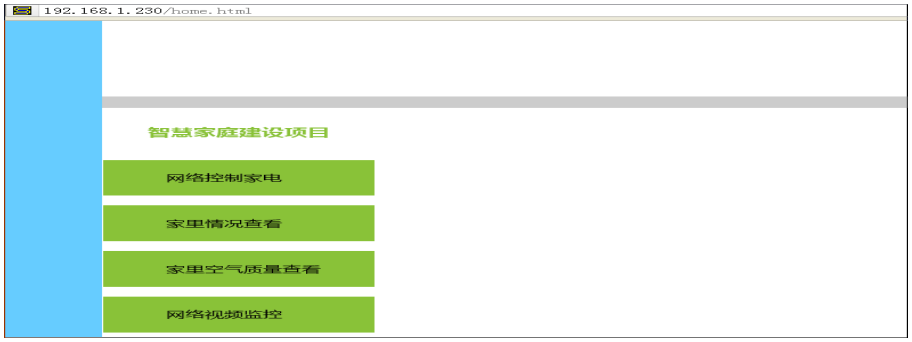

图 6-5 系统主页

监控主页面的 HTML 代码此处不再给出。通过主页面可以进入网络控制家电页面控制并查询家用电器的工作状态；进入家里情况查看页面可以实时查看到家庭温湿度环境，并可以通过网络视频监控页面实时查看到当前室内人员流动情况。

6.2.2 网络家电控制程序设计

网络家电控制程序的执行流程如图 6-6 所示，客户端提交了两个数据，一个操作。CGI 从服务器的环境变量中得到的是这两个数据的 url 编码，所以 CGI 程序需要从编码中将两个数据的变量名和变量值解码出来，并根据变量值向底层驱动传送控制命令执行电器的开启和关闭等操作，编译 CGI 程序下载到开发板的 /www/cgi-bin 目录下，运行 boa 程序，在客户端中打开网络控制家电页面如图 6-7 所示，可以看到每一步操作都会对应开发板上的电器的开启关闭等动作。为数据的序号代表要控制的电器，一个为数据的操作按钮名称代表对电器要执行的的动作。

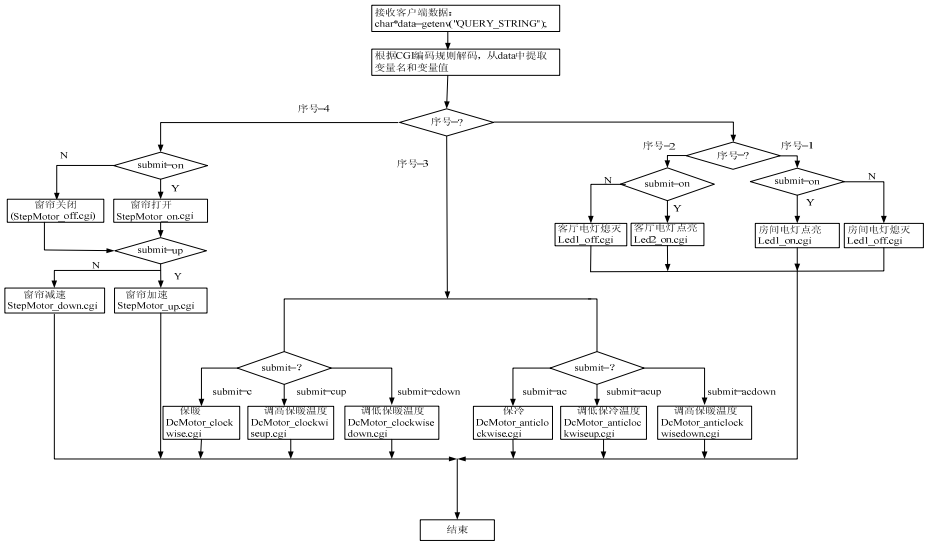

图 6-6 网络家电 CGI 程序结构框图

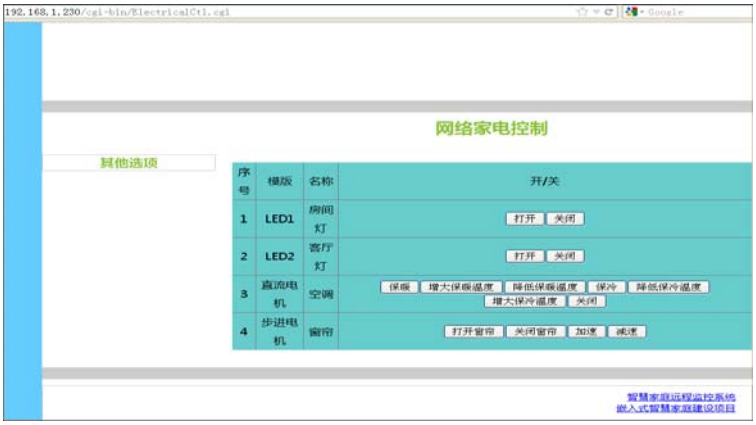

图 6-7 网络家电控制页面

6.2.3 温湿度环境监控程序设计

DHT11 一次完整的数据传输为 40bit，高位先出。数据格式：8bit 湿度整数数据+8bit 湿度小数数据 +8bit 温度整数数据+8bit 温度小数数据+8bit 校验和校验和数据为前四个字节相加。传感器数据输出的是未编码的二进制数据。如果，某次从传感器中读取如下 5Byte 数据，计算方法：humi (湿度)= byte4 . byte3=45.0 (% RH)，temp (温度)= byte2 . byte1=28.0(℃)，jiaoyan(校验)= byte4+ byte3+ byte2+ byte1=73(=humi+temp)(校验正确)，温湿度监控程序中最关键的是要将从设备文件中读取过来的温湿度数据解码得到温度和湿度的整数和小部分的值。解码过程如下，读到湿度的整数部分：humidityz = (temperature & 0xff000000)>>24; 读到湿度的小数部分：humidityx = (temperature& 0x00ff0000)>>16; 读到温度的整数部分：tempz = (temperature&0x0000ff00)>>8; 读到温度的小数部分：tempx = (temperature &0x000000ff);温湿度环境监控模块的控制界面如图 6-8 所示，点击刷新按钮可以实时更新从温湿度传感器中接收到的温湿度值。

图 6-8 温湿度监控页面

6.2.4 空气质量检测程序设计

在气体检测程序中，主要是调用了 read()函数从设备文件中拷贝数据，并将得到的二进制数据转换成空气质量的指标数据，空气检测程序的执行流程如图 6-9 所示。

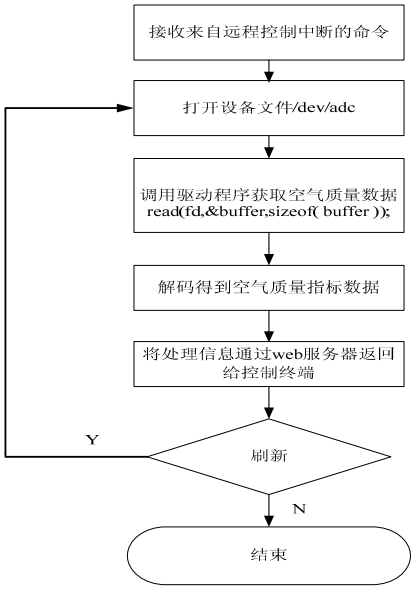

图 6-9 温湿度监控页面

室内空气质量监控模块的控制界面如图 6-10 所示，点击刷新按钮可以实时更新从器传感器中接收到的气体浓度值。

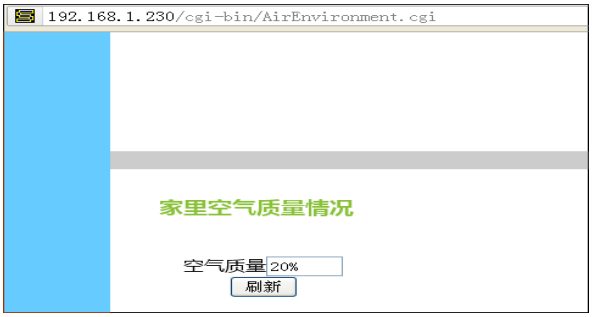

图 6-10 空气质量检测页面

6.2.5 网络视频监控程序设计

本系统设计的视频采集传输程序是基于 mjpg-streamer 开源项目基础之上，根据系统特定的工程需求，对源代码进行了修改之后得到的一个视频服务器。使用的是新一代 Linux 内核的 v4l2 的接口。视频服务器的源代码主要由三部分组成，

第一部分是 v4l2 接口，第二部分是 socket 编程，第三部分是多线程编程。

v4l2 接口这里涉及到我们如何从摄像头中把数据取出来，主要是封装了一个 struct vdIn 结构体用来描述摄像头的一些信息，比如采集图片的宽高，图片的格式，等等。接着是把这个结构体写入驱动中，用来初始化摄像头。这个操作通过 ioctl 完成，并通过 mmap 完成内存的映射。然后把获得的数据写入到文件里就是图片，通过网络传输出去连续的图片就是视频。

socket 编程在这个程序里使用的是 tcp 套接字，每有一个连接请求就创建一个线程单独和这个请求通信，这里涉及到的函数包括 socket、bind、listen、accept 和 write。还有第三部分是多线程编程，为了能同时响应多个客户端的请求，这里使用了多线程编程，为每一个请求建立一个连接，每个连接就是一个线程。这里涉及到的函数包括 pthread_create 、 pthread_detach 、 pthread_cond_init 、 pthread_cond_destroy、pthread_mutex_init、pthread_mutex_destroy。mjpg-streamer 工作流程如图 6-10 所示。

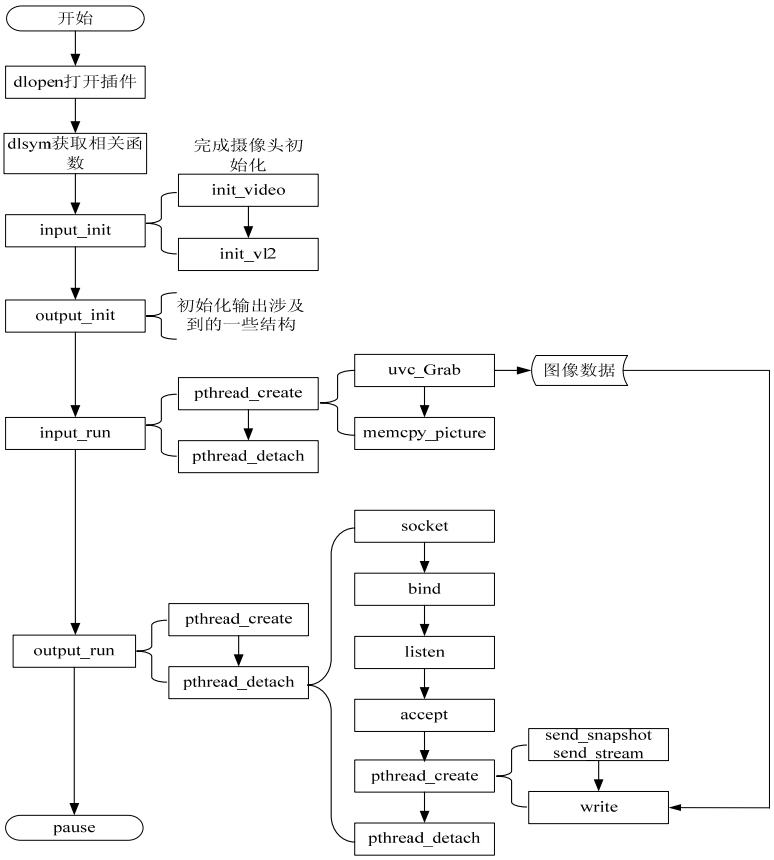

图 6-11 mjpg-streamer 工作流程

打开开发板运行 mjpg_streamer 进入系统主页面，点击网络视频监控页面就可以看到开发板采集得到的如图所示视频图像。

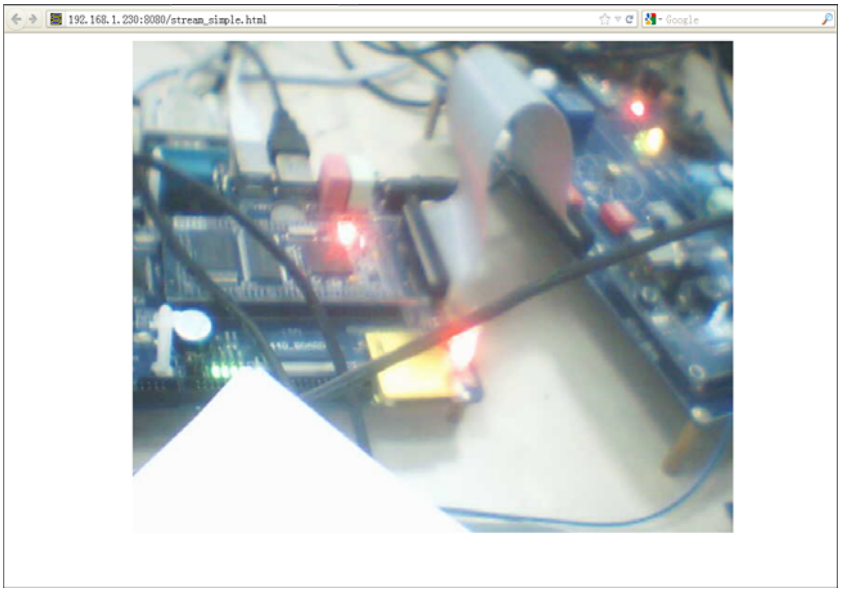

图 6-12 网络视频监控测试结果

6.3 系统性能测试

客户端为运行 WindowsXP 操作系统的 PC 机，将网络服务器通过以太网有线接口连接到 PC 机上，在实验环境对系统的性能进行了整体的测试。

当启动视频采集系统的时候可以看到从 USB 摄像头采集过来的视频数据经过服务器上的视频压缩程序处理之后，再通过以太网接口传送到 PC 终端依然可以看到非常流畅的视频图像。根据人眼的暂留时间可以推算，当每秒钟播放至少 24 帧图像的时候，可以看到流畅的播放画面，该视频监控每帧图像的分辨率为 320*240，像素为 RGB565 格式，可以得到该系统每秒钟至少可以传输 24*320*240*16/8/1024/1024=3.5MB 以上的数据，同时该系统还需要同时可以处理 20 多个由控制终端发送过来的命令请求，表明该系统不仅具备较强的数据处理能力，而且有较快的处理速度。

在毕业设计完成过程中，在实验室环境下每天都需要调试和测试系统各个模块的功能，系统平均每天工作都在 12 个小时以上，经过半年多不断的测试，除了有时因为硬件线路的松动出现过异常以外，大多时间内整个系统都很少出现不稳定和死机现象，总体来说达到了网络监控系统在稳定性和可靠性方面的要求。

在系统测试的时候，一直使用的是 5V-2A 的直流电源供电，当系统启动之后，用万用表测试电源模块输出的电流在 0.3A 左右，计算系统的功耗大约为 5V*0.3A=1.5W，如此复杂的系统功耗大约相当于两个彩色诺基亚手机的功耗，达到了监控系统在功耗方面的要求。

成本方面，整个系统的硬件成本大概在 500 元以内，软件上由于采用了开源的 Linux 操作系统作为软件开发平台，所有的开发和调试工具都是免费，Linux 操作系统更是一分钱的授权费用都没有收取，在现实中如果撇开具体各大家电的成本不算，用 ARM-Linux 做一个再复杂的家电网络控制系统成本也不过就是几百元。实践证明，使用该方案开发的智能家居网络监控产品在成本方面有一定的优势。

另外 ARM 体系结构的控制器接口资源都相当丰富，S3C2440 控制器更是达到了 289 个引脚资源，相信再增加几十个硬件模块，该系统依然可以胜任数据处理和命令收发工作。

第七章 结论与展望

7.1 总结

本系统以实验室现有 S3C2440 硬件平台为主板，在此基础上扩展了房间客厅的电灯照明，空调，窗帘控制，温度监控，气体检测，视频采集等硬件模块，并为整个系统移植了 BootLoader 程序和 Linux 内核以及根文件系统，然后又在 Linux 平台上开发了这些硬件模块的驱动程序和系统应用程序，成功实现了一个低成本、高效率、体积小、易于使用的嵌入智能家居网络监控系统设计方案。在整个开发过程中从软硬件开发平台的选型，到各个模块驱动的调试，以及整个系统的应用程序设计全部都由我一个人独立完成，前后经过了很多次反反复复的修改和验证，花费了半年多的时间，最终基本调试通过。总结这次毕业设计的经历主要取得了以下成果：

（1）针对目前市面上各种家居监控系统解决方案的优缺点，分析了下一代家居系统的新需求及新特点，给出了一个嵌入式 Linux 解决方案。

（2）软件开发采用了先模块后整体的设计思想，即按照嵌入式系统的最基本的开发流程分别完成了系统主板的 BootLoader 程序设计、嵌入式 Linux 内核移植，及各大扩展模块的驱动程序设计。最后将 BootLoader，嵌入式 Linux 操作系统和所有驱动模块都整合在一起，完成了一个基本可以投入使用的完整的嵌入式智能家居网络监控系统。

（3）本系统在功能设计上采用服务器+客户端的模式。所有的程序都运行在服务端。这样任何人都可以轻松方便的通过网页浏览器远程控制和管理连接在家居网络系统中的任何设备。

7.2 展望

由于时间和项目经验的限制，本文只是在实验室环境下通过模拟的手段实现了对网络智能家电系统的网络控制功能，另外几个比较大的功能像门禁系统，视频图像识别，GPRS 通信，无线控制等功能都还来不及实现，更重要的是各个功能模块之间还没有真正实现相互通讯相互调控，也就是说还没有做出一个将各个功

能模块完全整合一起形成一个完整的智能化系统，甚至说没有真正实现一个智能化网络化的家居产品。因本设计还存在相当多的不足。还需要从以下几个方面进一步研究和设计来完善和改进网络家居监控产品：

（1）本系统设计的重点是要把家庭内各个功能模块采集到的信息传给控制终端进行综合分析，进而由终端发出指令到相应模块做出智能化的动作，如何将家庭内各家居模块通过网络连接在一起协调工作还需进一步的研究。

（2）系统中的数据传输都是采用传统的以太网方式传输，要真正实现随时随地监控家庭住宅内的一切设备就必须完全采用移动通信的手段，进而保证数据传输的实时和可移动性。

（3）受条件所限，服务器和各电器模块采用了传统的有限控制线路，采用一个标准的，统一的无线通讯协议连接所有的设备有待研究。

参考文献

[1] 杜春雷.ARM 体系结构与编程.北京：清华大学出版社，2003：1-21

[2] 赵炯.Linux 内核完全剖析.北京：机械工业出版社，2010：1-8

[3] 杨炼.三网融合的关键技术及建设方案：人民邮电出版社，2011：1-10

[4] 霍尼韦尔. 霍尼韦尔智能家居解决方案. http://www.china-homewell.com/solutions/home_net.asp.

[5] 北京雅天. 物联网时代的智能家居调研报告.http://www.yadantecg.com/dowload/06214114f18583d0599964592d.html.

[6] 李程.基于嵌入式的智能家居系统研究:[硕士学位论文].成都：电子科技大学论文.2006.03.

[7] 马季.智能家居网络监控系统的研究与实现: [硕士学位论文].青岛：中国海洋大学论文.2009.06.

[8] 周立功等.ARM 嵌入式系统基础教程（第 2 版）.北京.北京航空航天大学出版社，2005:15-51

[9] Samsung.S3C2440 MIRCROCTROLLER USER'MANUAL. http://www.samsungsemi.com/.

[10] 王洪辉.嵌入式系统 Linux 内核开发实战指南(ARM 平台).北京.电子工业出版社，2009：142-176

[11] 杨铸.嵌入式底层软件开发.北京：北京航空航天大学出版社，2010：308-346

[12] 赵庆明.嵌入式 boot loader 分析与设计:[硕士学位论文].成都:电子科技大学硕士学位论文.2008.11

[13] 周立功.ARM 嵌入式系统实验教程（二）.北京：北京航空航天大学出版社，2005:20-60

[14] 周立功.ARM 嵌入式系统实验教程（三）.北京：北京航空航天大学出版社，2005:30-200

[15] Samsung. K4S561632F USER'MANUAL.http://www.samsung.com/Products/Semiconductor/Flash/TechnicalInfo/datasheets.htm

[16] Samsung. K9VF2G1608 USER'MANUAL. http://www.samsung.com/Products/Semiconductor/

[17] 肖德贵.ARM 结构与程序开发入门.北京：清华大学出版社，2009：92-111

[18] 孙戈.基于 S3C2440 嵌入式 Linux 开发实例.北京：清华大学出版社，2009：248-280

[19] 国嵌.嵌入式 Linux 工程师实验手册上册.成都：2010：74-107

[20] 李驹光.嵌入式 Linux 开发详解——基于 EP93XX 系列 ARM.北京.电子工业出版社.2006:241-278

[21] 冯国进.嵌入式 Linux 驱动程序设计从入门到精通.北京：人民邮电出版社，2010：1-10

[22] 宋宝华.Linux 设备驱动开发详解.北京：人民邮电出版社，2010：1-10

[23] 英倍特.S3C2410 嵌入式 Linux 开发实验. http://www.edukit.com.cn/list.asp?id=250.

[24] 郑强.Linux 驱动开发入门与实战.北京：清华大学出版社，2011.:152-166

[25] 王晓明.电动机的单片机控制.北京：北京航空航天大学出版社，2011:50-78

[26] 陈雪松.深入 Linux 设备驱动程序内核机制.北京：电子工业出版社，2012:274-293

[27] 徐成.嵌入式 Linux 实训教程.北京：人民邮电出版社，2010:147-158.

[28] 文全刚.汇编语言程序设计.北京：人民邮电出版社，2009：69-73

[29] 韦东山.嵌入式 Linux 应用开发完全手册.北京：人民邮电出版社，2008.

[30] 李异光.基于嵌入式实时 Linux 的视频监控系统的设计与实现:[硕士学位论文].成都:电子科技大学硕士学位论文. 2006.01

[31] 吴雪琴.基于 S3C2410 网络监控系统的设计与实现:[硕士学位论文].成都:电子科技大学大学硕士学位论文. 2008.06

[32] 张涵.基于 GPRS 网络的无线数据通信设备的设计与实现:[硕士学位论文].成都:电子科技大学硕士学位论文. 2009.05

[33] 王显涛.GPRS 网络监控系统:[硕士学位论文].成都:电子科技大学硕士学位论文. 2005.5

[34] 徐晟.嵌入式文件系统的研究与实现：[硕士学位论文]. 成都:电子科技大学. 2006.6

[35] 刘涛.一种嵌入式实时 Linux 的设计与实现:[硕士学位论文].成都:电子科技大学硕士学位论文. 2007.5

[36] 高仁才.基于 ARM 的网络视频监控机设计及实现:[硕士学位论文].长春:吉林大学硕士学位论文.2010.5

[37] 任飞.基于 H.264 的视频采集编码系统设计:[硕士学位论文].成都:电子科技大学硕士学位论文. 2006.1

[38] 方卫民. 基于 ARM 的嵌入式网络视频监控系统设计与实现:[硕士学位论文].北京:北京邮电大学硕士学位论文. 2008.3

[39] 柳亚东. 基于 S3C2410 的嵌入式网络视频监控系统:[硕士学位论文].上海:上海交通大学硕士学位论文. 2009.01

[40] 李保国. 基于嵌入式 ARM 的网络视频监控系统研究:[硕士学位论文].南京:南京理工大学硕士学位论文. 2009.06

[41] 余冰.基于 ZigBee 的智能家居网络控制系统:[硕士学位论文]北京.电子设计应用.2009.08

[42] 王小强.ARM 处理器裸机开发实战.北京：电子工业出版社，2012:110-140

[43] 刘波文.ARM 嵌入式项目开发.北京：北京航空航天大学出版社，2011：290-390

[44] 刘淼.嵌入式系统接口设计与 Linux 驱动程序开发.北京：北京航空航天大学出版社，2006：1-7

[45] 周立功.ARM 嵌入式系统软件开发范例（一）.北京：北京航空航天大学出版社，2004：100-248

[46] 周立功.ARM 嵌入式系统软件开发范例（二）.北京：北京航空航天大学出版社，2005：158-359