分类号 _____ 密级 _____

U D C _____

学 位 论 文

基于 μC/OS-Ⅱ的嵌入式 TCP/IP 协议栈的研究与实现

作者姓名： 钟芳伟

指导教师： 傅仲逑 教授

东北大学秦皇岛分校计算机工程系

申请学位级别： 硕 士 学科类别： 工 学

学科专业名称： 计算机软件与理论

论文提交日期： 2006 年 12 月 10 日 论文答辩日期： 2007 年 1 月 13 日

学位授予日期： 答辩委员会主席： 才书训 教授

评阅人： 王翠荣 教授 张付志 教授

东 北 大 学

2006 年 12 月

A Dissertation in Computer Software and Theory

Research and Implement of Embedded TCP/IP Protocol Stack Based on μC/OS-Ⅱ

by Zhong Fangwei

Supervisor: Professor Fu Zhongqiu

Northeastern University

December 2006

独创声明

本人声明所呈交的学位论文是在导师的指导下完成的。论文中取得的研究成果除加以标注和致谢的地方外，不包含其他人已经发表或撰写过的研究成果，也不包括本人为获得其他学位而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示诚挚的谢意。

学位论文作者签名：

日　　期：

学位论文版权使用授权书

本学位论文作者和指导教师完全了解东北大学有关保留、使用学位论文的规定：即学校有权保留并向国家有关部门或机构送交论文的复印件和磁盘，允许论文被查阅和借阅。本人同意东北大学可以将学位论文的全部或部分内容编入有关数据库进行检索、交流。

（如作者和导师不同意网上交流，请在下方签名；否则视为同意）

学位论文作者签名：　　导师签名：

签字日期：　　签字日期：

基于 μC/OS-Ⅱ的嵌入式 TCP/IP 协议栈的研究与实现

摘 要

随着 Internet 的发展和应用，越来越多的设备需要接入网络，以实现远程控制或资源共享。嵌入式系统由于具有体积小、价格低、专用性能高的优势，被广泛应用于各种电子设备中。基于嵌入式系统的网络接入技术即嵌入式 Internet 技术是未来应用的主要趋势，该技术的出现是 Internet 发展过程中一个新的里程碑。

受嵌入式系统存储资源和计算资源的限制，嵌入式 Internet 主要有两种系统结构：基于网关的嵌入式 Internet 系统结构和直连嵌入式 Internet 系统结构。比较而言，直连嵌入式 Internet 系统结构是一种新的系统结构，在系统成本、开放性、移动性方面具有较大优势，因而更有应用前景。

针对嵌入式系统连入 Internet 的要求，本文以 32 位微控制器为出发点，首先介绍了目前嵌入式 Internet 技术的发展情况并确定本课题的实现方案，其次分析了 μC/OS-Ⅱ在 ARM 上的移植，然后重点研究了如何对标准 TCP/IP 协议的简化，以实现一个性能稳定的轻量级嵌入式 TCP/IP 协议栈的方法。课题中设计并实现的轻量级嵌入式 TCP/IP 协议栈是以标准 TCP/IP 协议栈为基础，采用 Lwip 中的缓冲区管理机制，使用基于中断驱动的模式，改造 API 层适应多个网络程序同时运行，该协议栈还具有可配置性强和移植性好的特点。

论文最后使用该协议栈提供的接口函数实现了一个简单的嵌入式 Web Server 测试用例，并在 S3C44B0X 开发板上进行测试，测试结果表明该协议栈具有可配置性、占用系统资源少、可移植性、接口简单易用等特点。

关键词：嵌入式系统；嵌入式 Internet；轻量级嵌入式 TCP/IP 协议栈

Research and Implement of Embedded TCP/IP Protocol Stack Based on μC/OS-Ⅱ

Abstract

With the development and application of the Internet, more and more equipments are needed to connect with network for achieving long-distance control and sharing resource. Embedded systems are used more widely, for small profile, low price and high performance. The technology of embedded network systems or embedded Internet is the main trend of future application. It has become a new milestone in the process of Internet development.

For the limitation of computation and memory resources in embedded system, embedded Internet can be classified into two architectures: Gateway-based architecture and Direct-connection architecture. Compared to Gateway-based architecture, Direct-c onnection architecture is a new architecture and has more applied prospect for the reason of its advantage in low system cost, open and portable characters.

Towards the requirement of connecting the embedded system to Internet, This paper based on 32-bit micro-controller. First, we introduce the development situation of the embedded Internet at present and determine the realization plan of this topic. Second, we analyze the porting of μC/OS-Ⅱ at ARM board, then, we study the methods of simplifying full TCP/IP protocol stack to implement a light-weight TCP/IP protocol stack of stable performance. The light-weight TCP/IP protocol stack of this paper is based on the standard TCP/IP protocol stack. This protocol stack uses the buffer management mechanism of Lwip and an interruption based programming model, we also improve the API layer of this protocol stack to adapt many programs to run at the same time. Moreover, this protocol stack has characters of configurable and portable.

At the end, a simple embedded web server as a testing case using network interface afforded by this stack is implemented. Also, this simple embedded web server is tested on an embedded system platform of S3C44B0X, According the testing result, we draw conclusion that this stack have configurable, portable characters and also have characters of limited system resources Consumption, simple and easy interface specification and so on.

Key words：embedded system; embedded Internet ; Light-weight embedded TCP/IP protocol stack

目 录

独创声明 I
摘 要 II
Abstract III
第一章 绪论 1
1.1 课题的背景、现状及意义 1
1.2 课题的提出 2
1.3 论文的组织 3
第二章 嵌入式 Internet 系统结构 5
2.1 嵌入式系统 5
2.1.1 嵌入式系统的基本概念 5
2.1.2 嵌入式系统的基本特征 6
2.2 嵌入式 Internet 技术 6
2.2.1 嵌入式 Internet 原理和特点 6
2.2.2 嵌入式 Internet 系统结构 7
2.3 嵌入式 TCP/IP 协议 10
2.3.1 TCP/IP 协议介绍 10
2.3.2 TCP/IP 协议进程模型 11
2.3.3 嵌入式 TCP/IP 协议栈的实现要素 12
第三章 μC/OS-Ⅱ在 ARM 上的移植 15
3.1 嵌入式实时操作系统 μC/OS-Ⅱ概述 15
3.2 μC/OS-II 的内核原理 15
3.3 μC/OS-II 的任务管理和调度 16
3.3.1 任务管理 16
3.3.2 任务的调度 18
3.4 μC/OS-II 的中断及时钟节拍 20
3.4.1 中断 20
3.4.2 μC/OS-II 的时钟节拍 20
3.5 μC/OS-II 的通信机制 21
3.5.1 事件控制块 21
3.5.2 通信机制的实现 22

3.6 μC/OS-II 在 S3C44B0X 上的移植.....24
3.6.1 移植条件.....24
3.6.2 任务模式的取舍.....24
3.6.3 移植实现.....24
3.7 移植测试.....28
第四章 嵌入式 TCP/IP 协议栈设计与实现.....31
4.1 协议栈实现框架.....31
4.2 嵌入式 TCP/IP 协议栈设计.....31
4.2.1 嵌入式 TCP/IP 协议栈需求分析.....31
4.2.2 缓冲区管理.....33
4.2.3 “一次拷贝”技术.....34
4.3 网卡驱动程序设计.....35
4.3.1 数据帧的组成.....35
4.3.2 初始化操作.....36
4.3.3 数据接收操作.....36
4.3.4 数据发送操作.....37
4.4 网络层协议实现.....37
4.4.1 IP 协议的实现.....37
4.4.2 ICMP 协议的实现.....40
4.4.3 ARP 协议的实现.....41
4.5 传输层协议的实现.....44
4.5.1 UDP 协议的实现.....44
4.5.2 TCP 协议的实现.....47
4.6 SOCKET API 接口实现.....54
4.6.1 API 层定义的主要数据结构.....54
4.6.2 API 主要功能函数的实现.....55
4.7 嵌入式 TCP/IP 协议通用接口层的实现.....55
第五章 嵌入式 TCP/IP 协议栈测试.....59
5.1 测试环境.....59
5.2 嵌入式协议的功能测试.....59
5.2.1 IP 层及其以下模块的测试.....59
5.2.2 IP 层以上模块的测试.....60
5.3 嵌入式协议的性能测试.....61
5.3.1 TCP 建立连接的响应时间.....61
5.3.2 TCP 关闭连接的响应时间.....61

5.3.3 TCP 协议的数据传输率 61
5.4 测试小结 62
第六章 结论与展望 63
6.1 结论 63
6.2 展望 63
参考文献 65
致 谢 69
攻读硕士期间发表的论文 71

第一章 绪论

1.1 课题的背景、现状及意义

近年来，随着嵌入式计算技术的不断发展，嵌入式计算已深入到人类社会的各个领域。从军用高技术装备到信息家电，从通信设备到医疗器械，从工业控制到智能仪器仪表，处处都需要嵌入式计算技术[1]。有人把这种嵌入式计算广泛应用的阶段叫“后 PC 时代”，所谓的后 PC 时代，是英文 pervasive computing 的中文意译，它以嵌入式计算机技术为依托正逐步渗透到人类社会的方方面面。随着 Internet 技术的快速发展和不断成熟，使得 Internet 的应用范围和领域不断扩大。除了传统的信息检索、电子邮件、远程登录、文件传输等业务外，各种新应用(如信息家电、远程医疗、嵌入式 Web 传感器、远程数据采集、工业控制等)越来越受到人们的大量关注[2]。

目前大多数嵌入式系统仍然处于单独应用的阶段，其系统架构一般以 MCU 为核心，与一些监测、指示设备进行连接以实现一定的应用功能[3]。在一些工业控制和汽车应用领域中，为了实现多个嵌入式系统之间的信息交流，一般利用 CAN、RS-232、RS-485 等总线将多个嵌入式系统组网，但这种网络的有效半径有限，有关的通信协议也比较少，同时又孤立于 Internet 之外。Internet 现已成为社会重要的基础设施之一，是信息流通的重要渠道，把嵌入式系统连接到 Internet，通过 Internet 实现各种嵌入式设备网络化应用是一种趋势，也是一种必然，在这种发展需求下，嵌入式 Internet (EI)技术应运而生[4]。

嵌入式 Internet 技术的出现是 Internet 发展历史上的一个里程碑，它依托于 Internet、Web 和嵌入式三大技术。嵌入式 Internet 技术是一种设备接入技术和异种网络互连技术，主要解决的问题是通过 Web 和嵌入式技术实现从不同子网、不同的物理区域对接入到 Internet 的设备和异类子网进行监控、诊断、测试、管理、及维护等功能，从而使接入到 Internet 的各种设备或其它类型的子网具有远程监控、诊断和管理的功能[5]。嵌入式 Internet 技术是传统的嵌入式技术与新兴的 Internet 技术相结合的产物，它的出现使得众多的工业仪器、设备和家用电器连入 Internet 成为可能。嵌入式 Internet 具有的主要优点在于它可以从设备的角度来看 Internet，把 Internet 的功能嵌入到设备中，称为 Embedded Internet Device (EID)，通过这种方式来方便设备操作，简化远程控制。

随着嵌入式 Internet 的迅速发展，针对不同的软硬件环境以及应用场合，国内

外也先后提出了很多不同的嵌入式系统的 Internet 解决方案。嵌入式 Internet 主要有两种系统结构：基于网关的嵌入式 Internet 系统结构和直连嵌入式 Internet 系统结构[6]。基于网关的嵌入式 Internet 系统结构的主要实现思想是：采用 PC 或高性能嵌入式计算设备等做网关，支持标准 TCP/IP 协议栈并运行基于 Internet 的服务程序(通常是 Web Server)。网关和嵌入式系统之间则通过一些标准的串行通讯协议(如：RS232, RS485 等)或者私有通讯协议(由嵌入式 Internet 解决方案提供商提供)进行通讯。这种系统结构相对比较成熟，应用也很广泛。如：基于嵌入式 Web Server 的网络视频监控系统，水电站综合自动化系统等。

直连嵌入式 Internet 系统结构则是一种新的嵌入式 Internet，其主要实现思想是：针对嵌入式系统的软硬件环境以及具体的嵌入式 Internet 应用对标准的 TCP/IP 协议栈进行简化。让系统资源有限的嵌入式系统支持一种轻量级 TCP/IP 协议栈而直接连入 Internet。直连嵌入式 Internt 系统结构还处于不断的研究和完善之中，目前应用不是很广泛。比较而言，直连嵌入式 Internt 系统结构作为一种新的嵌入式 Internet 系统结构，在系统成本、开放性、移动性方面具有较大优势，因而更有应用前景。

嵌入式设备与 Internet 的结合代表着嵌入式系统和网络技术的真正未来，它具有巨大的市场潜力，目前，包括 Siemens、Philips 和 Motorola 在内的数十家公司联合成立的“嵌入式 Internet 联盟(ETI)”、中国单片机公共实验室、全国嵌入式系统学术交流会和国内外其他一些研究组织共同推动该技术的发展。国内外著名的公司有 emWare，北京英贝多嵌入式网络技术有限公司和沈阳东大新业信息技术股份有限公司，他们都使嵌入式 Internet 技术运用到了实用产品中。完善嵌入式 TCP/IP 协议栈，推动嵌入式 Internet 广泛应用将使我们这个世界变得更加自动化、智能化和人性化[7]。

1.2 课题的提出

嵌入式 Internet 技术产生的基础是嵌入式技术和 Internet 技术，实现嵌入式 Internet 技术的关键是如何在嵌入式微处理器上实现 TCP/IP 协议栈。本文采用的是直连嵌入式 Internet 系统结构，而实现直连嵌入式 Internet 系统结构的关键之处是：如何结合系统资源有限的嵌入式系统软硬件环境以及具体的嵌入式 Internet 应用对标准的 TCP/IP 协议栈进行简化，从而实现一种轻量级嵌入式 TCP/IP 协议栈。TCP/IP 是 Internet 的基本协议，以其实用性、高效性已经成为事实上的工业标准。嵌入式设备要与 Internet 网络直接交换信息，就必须支持 TCP/IP 协议。

在嵌入式 TCP/IP 协议栈的研究方面，国内外做了很多研究。像 Jeremy Bentham 的 PICmicro 协议栈，Texas Instument 的 MSP430 TCP/IP 协议栈以及 TinyTCP code

出现的比较早，但是由于这些协议栈的实现和应用紧密联系，没有实现协议栈与应用的分离，没有接口的概念，也没有做成函数库的形式，所以这些协议栈基本退出了历史舞台。近年来 Adam Dunkels 的 uIP 和 Adam Dunkels 等开发的 Lwip 在嵌入式 Internet 领域用的比较广泛，uIP 侧重于减小代码量（选择 AVR 为目标器件时，代码为 5K 左右）和减小 RAM 使用量（100 字节左右）。但是 uIP 采用了不保存需要应答的数据包的 RAM 使用方案，没有和 BSD 的套接字接口兼容，应用层接口较复杂；而 Lwip 的功能虽然很全面，但是相对来说代码较大，编程复杂。

基于上述嵌入式 TCP/IP 协议栈的不足之处，本课题主要研究如何结合嵌入式 Internet 应用以及嵌入式系统的软硬件系统环境，以标准 TCP/IP 协议为基础，结合 Lwip 中的缓冲区管理机制，采用基于中断驱动的模式，接口通过 API 形式以适应多个网络程序同时运行，设计并实现一种与应用分离，同时具有占用系统资源少、可配置、易于移植、接口简单易用等特点的轻量级嵌入式 TCP/IP 协议栈。使用这种轻量级嵌入式 TCP/IP 协议栈，可以让嵌入式 Internet 应用开发人员省去 TCP/IP 通讯协议部分的设计与实现，把主要的工作放在应用程序(协议)的分析和设计上，同时为了适应 μC/OS-Ⅱ这样的实时操作系统，协议栈整体作为任务的形式实现[8]。

最后，使用本课题实现的这个嵌入式 TCP/IP 协议栈提供的接口函数实现一个简单的嵌入式 Web server 测试用例，并在 32 位的 S3C44B0X ARM7 平台上，以 μC/OS-Ⅱ实时操作系统为软件基础进行功能和性能测试，用来证明本文提出的嵌入式 TCP/IP 协议栈设计方法的有效性和合理性。

1.3 论文的组织

本文共分为六章，以下是各章的简要介绍：

第一章是绪论，主要介绍了本课题的背景、目的、意义以及课题的提出。

第二章是嵌入式 Internet 系统结构，主要介绍了嵌入式 Internet 两种典型的系统结构，并在此基础上分析了嵌入式 Internet 系统结构的发展趋势。

第三章是 μC/OS-Ⅱ在 ARM 上的移植，主要介绍了开源的实时操作系统 μC/OS-Ⅱ在 S3C44B0X 上的移植。

第四章是嵌入式 TCP/IP 协议栈设计与实现，主要对嵌入式协议栈和 μC/OS-Ⅱ的结合方式，对各部分协议需要实现的功能进行说明，对各部分的难点、关键技术和相关数据结构进行分析与实现，并实现了网络接口的驱动程序。

第五章是嵌入式 TCP/IP 协议测试，主要是检测本文设计的协议栈及网络接口驱动的设计是否达到了预期的目的。

第六章是结论与展望，是全文工作的结论和未来工作的展望。

第二章 嵌入式 Internet 系统结构

2.1 嵌入式系统

近年来，随着软硬件资源的成熟与完善，嵌入式系统的应用得到迅猛的发展，其领域涉及通信、自动化、信息家电、军事、医疗、汽车电子等各个方面。IDG 发布的统计数字表明，未来的四五年内，信息家电市场会成长 5 倍-10 倍。信息家电作为家庭信息终端，之所以变得火爆，一个很重要的原因就是嵌入式系统的引入。嵌入式系统以其得天独厚的优势，广泛地应用于各种信息产品，其数量之大、种类之多，标志着嵌入式系统的应用发展浪潮已经来临[9]。

2.1.1 嵌入式系统的基本概念

嵌入式系统一般指非 PC 系统，它以应用为中心、以计算机技术为基础，软硬件可裁剪，适应应用系统对功能、可靠性、成本、体积、功耗严格要求的专用计算机系统[10]。它包括硬件和软件两部分。硬件包括处理器/微处理器、存储器及外设器件和 I/O 端口、图形控制器等。软件部分包括操作系统软件(要求实时和多任务操作)和应用程序软件。这种系统具有软件代码小、高度智能化、响应速度快等特点，特别适合于要求实时的和多任务的体系。有时设计人员把这两种软件组合在一起。应用程序控制着系统的运作和行为；而操作系统控制着应用程序软件与硬件的交互作用。嵌入式系统基本要素如图 2.1 所示。

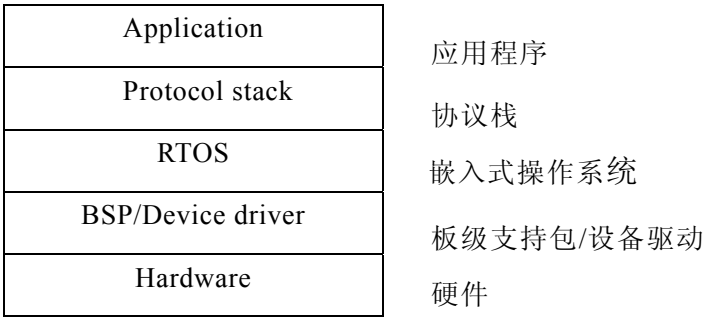

图 2.1 嵌入式系统的基本要素
Fig. 2.1 Elements of embedded system

嵌入式计算机系统，最早出现在 60 年代武器控制中，后来用于军事指挥控制和通信系统，现在广泛用于民用机电一体化产品中。嵌入式计算机在应用数量上远远超过了各种通用计算机，一台通用计算机的外部设备中就包含了 5-10 个嵌入式微处理器，键盘、鼠标、软驱、硬盘、显示卡、显示器、网卡、Modem、声卡、打印机、扫描仪、数字相机、USB 集线器等均是由嵌入式处理器控制的。在制造工业、过程

控制、医疗、通讯、仪器、仪表、汽车、船舶、航空、航天、军事装备、消费类产品等方面均是嵌入式计算机的应用领域[11]。

2.1.2 嵌入式系统的基本特征

(1) 嵌入式系统通常使用而向特定应用的嵌入式 CPU。与通用型 CPU 的最大不同就是嵌入式 CPU 大多工作在为特定用户群设计的系统中，它通常都具有低功耗、体积小、集成度高等特点，能够把通用 CPU 中许多由板卡完成的任务集成在芯片内部，从而有利于嵌入式系统设计趋于小型化，移动能力大大增强，跟网络的耦合也越来越紧密。

(2) 嵌入式系统是将先进的计算机技术、半导体技术和电子技术与各个行业的具体应用相结合后的产物。这一点就决定了它必然是一个技术密集、资金密集、高度分散、不断创新的知识集成系统。

(3) 嵌入式系统的硬件和软件都必须高效率地设计，量体裁衣、去除冗余，力争在同样的硅片面积上实现更高的性能，这样才能在具体应用中对处理器的选择更具有竞争力。

(4) 嵌入式系统和具体应用有机地结合在一起，它的升级换代也是和具体产品同步进行，因此嵌入式系统产品一旦进入市场，具有较长的生命周期。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单片机本身中，而不是存贮于磁盘等载体中。

2.2 嵌入式 Internet 技术

嵌入式 Internet 是指在嵌入式系统应用领域，以 Internet 技术为基础，使嵌入式系统与 Internet 相互连接，实现嵌入式系统与 Internet 之间的资源共享、信息通信和状态控制等功能[12]。它主要解决的问题是通过 Web 和嵌入式技术实现从不同子网、不同的物理区域对接入到 Internet 的设备和异类子网进行监控、诊断、测试、管理、及维护等操作，从而使用户对接入到 Internet 上的各种设备或其它类型的子网具有远程监控、诊断和管理的能力。

2.2.1 嵌入式 Internet 原理和特点

嵌入式 Internet 是嵌入式技术与 Internet 技术相结合的产物。它既保留了嵌入式设备的小巧、智能、可编程的特点，又借助于 Internet 这个全球最大的计算机网络来把对现场设备的控制延伸到地球上几乎任何一个角落。利用嵌入式 Internet 技术可以真正实现设备的远程管理和控制，并且可以利用 Internet 来对设备进行远程维护，甚至可以允许重新下载智能设备的运行程序[13]。而这一切的控制操作只需通过

标准的 Internet 浏览器就能实现，并不需要使用专用的客户端软件，所完成的功能与专有客户端是一样的。

在嵌入式 Internet 环境下，设备通常是现场总线上的多台设备或者是孤立的一台传统设备。要实现这些设备与 Internet 联网，需要有嵌入式 Internet 服务器(或称为嵌入式网关)为传统的孤立设备提供网络接口，或者为现场总线和 Internet 之间的通信提供协议转换的功能。Internet 上的用户只需使用标准的浏览器或专用的客户端软件就可以与嵌入式 Internet 服务器建立 TCP 或 HTTP 连接，由嵌入式 Internet 服务器来把用户的指令转换成设备能识别的代码或者把设备的信息打成 IP 包后再发给客户端应用程序。用户在客户端可以选择两种应用程序，如果通过在标准的浏览器运行 Java Applets，则能够做到客户端与操作系统平台无关；使用专用的客户端软件，则能够根据实际情况灵活地设计应用程序，从而避免对浏览器的依赖[14]，图 2.2 所示的是嵌入式 Internet 的基本原理图。

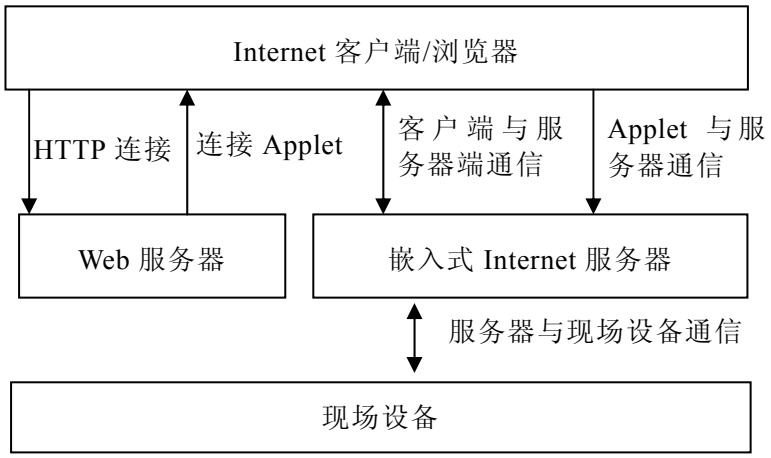

图 2.2 嵌入式 Internet 基本原理
Fig. 2.2 The basic theory of Embedded Internet

2.2.2 嵌入式 Internet 系统结构

嵌入式 Internet 技术正在不断的发展，新技术层出不穷，新产品不断产生，从底层硬件技术到上层软件所提供的解决方案都在不断的完善。目前国内外有关嵌入式 Internet 系统结构可以分为基于网关的嵌入式 Internet 系统结构和直连的嵌入式 Internet 系统结构。

2.2.2.1 基于网关的嵌入式 Internet 系统结构

让嵌入式系统支持 TCP/IP 协议栈的目的就是要使嵌入式系统能够与 Internet 上其它的网络计算设备进行通讯，既然让系统资源非常有限的嵌入式系统支持标准的 TCP/IP 协议栈显得不太合适，那么是否可以让这些嵌入式系统通过一个运行标准 TCP/IP 协议栈的代理而间接地提供对 TCP/IP 协议栈的支持。于是就出现了一种基

于网关的嵌入式 Internet 系统结构解决方案[15,16]，这种基于网关的嵌入式 Internet 系统结构蕴含的核心思想是:采用 PC 或高性能嵌入式计算设备等做网关，支持标准 TCP/IP 协议栈并运行基于 Internet 的服务程序(通常是 Web Server)。网关和嵌入式系统之间则通过一些标准的串行通讯协议(如:RS232, RS485 等)或者私有通讯协议(由嵌入式 Internet 解决方案提供商提供)进行通讯。在这种基于网关的嵌入式 Internet 系统结构下，Internet 上的用户可以运行客户端程序(如:浏览器)通过网关上运行的网络服务程序(如：Web Server)实现对嵌入式系统的远程访问和控制。从根本上来看，这个支持标准 TCP/IP 协议栈的网关起了一个代理的作用，嵌入式系统通过这个标准 TCP/IP 协议栈的代理提供对 TCP/IP 协议栈的支持，进而连入 Internet。

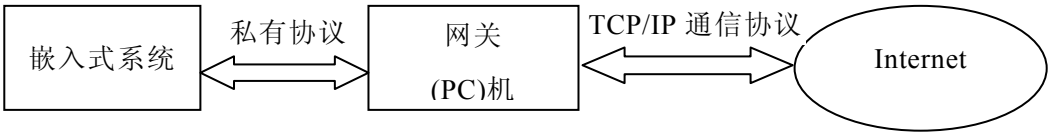

图 2.3 基于网关的嵌入式 Internet 系统结构
Fig. 2.3 The Embedded Internet system structure based on gateway

emWare 公司开发的嵌入式微 Internet 网络互连技术(EMIT：Embedded Micro Internetworking Technology)就是这种基于网关的嵌入式 Internet 系统结构的典型应用[17]。EMIT 由 emNet 和 emGateway 两部分组成，具体如图 2.4 所示。emNet 是 emWare 公司自己定义的一种私有通讯协议，是为嵌入式系统和其他网络(如 RS485、IR、RF 和电力线等)进行联接的网络协议，它运行在 MCU 的内部，emNet 通讯协议使得集成 emMicro 的嵌入式系统能够和嵌入式微控制器网关 emGateway 进行有效的通信。EMIT 采用桌面计算机或高性能嵌入式处理器作为网关 emGateway，支持 TCP/IP 协议并运行 Internet 服务程序，形成一个用户可通过网络浏览器进行远程访问的服务器，emGateway 通过 RS232、RS485、CAN、红外、射频等总线将多个嵌入式设备联系起来，每个嵌入式设备的应用程序中包含一个独立的通信任务，称为 emMicro，监测嵌入式设备中预先定义的各个变量，并将结果反馈到 emGateway 中；同时 emMicro 还可以解释 emGate-way 的命令，修改设备中的变量，或进行某种控制。嵌入式微控制器网关(即 emGateway)运行在计算机、TV 机顶盒或专用的家用电器服务器中，它是嵌入式设备和 Internet 之间连接的桥梁。嵌入式系统通过 emNet 通讯协议以及 emGateway 连入 Internet。

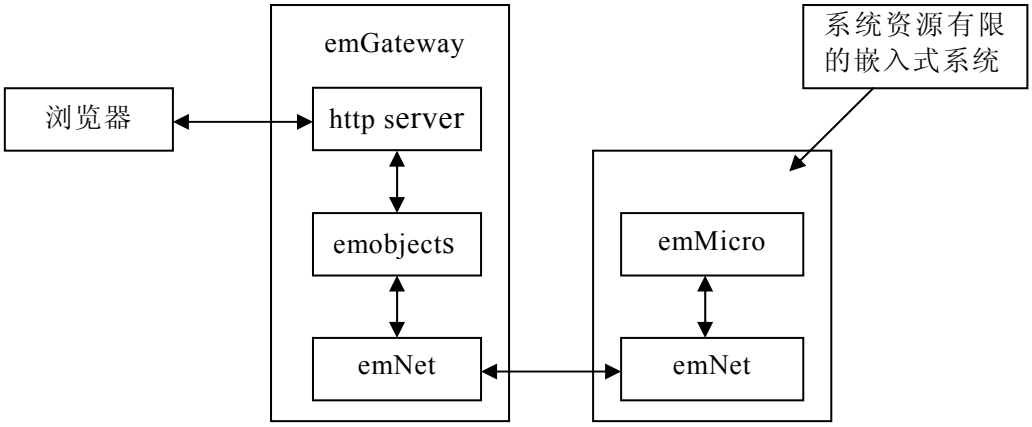

图 2.4 EMIT 系统结构

Fig. 2.4 The system structure of EMIT

这种方式要求设计工程师必须熟悉 emNet 协议和相关的接口，并且软硬件设计的工作量仍然较大。应用系统的 MCU 处理 emNet 协议要占用一定的系统资源，对 MCU 的要求也较高，同时需要微机做网关。优点是网关中的一个 IP 地址可以联接多个嵌入式应用系统。

还有另一种基于网关的嵌入式 Internet 系统结构的典型应用实例是 Webchip[18]。Webchip 是独立于各种微控制器的专用网络接口芯片，它通过标准的输入、输出接口与各种嵌入式系统中的 MCU 相连。Webchip 专用芯片与网关之间则采用一些标准的串行通讯协议(如 RS232，RS485)。MCU 通过 Webchip 与网关连接即可接收并执行通过 Internet 远程传来的命令或将应用数据交给 Webchip，然后由 Webchip 把这些应用数据发送到网关，再由网关发送到 Internet。基于 Webchip 的嵌入式 Internet 系统结构最终还是要通过一个网关才能连入 Internet，它只不过是用硬件代替软件实现的方式屏蔽了嵌入式系统与网关的一些通讯细节和过程。可以说是 EMIT 系统结构的一个变种。

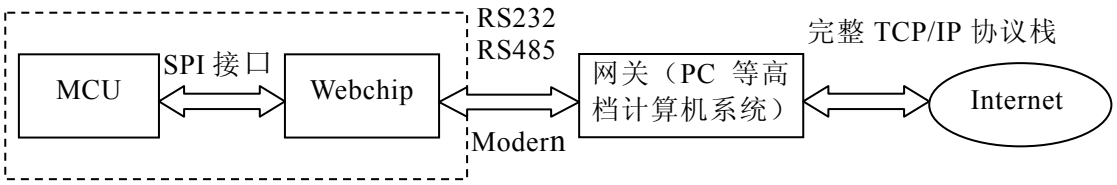

图 2.5 基于专用芯片 Webchip 的嵌入式 Internet 系统结构

Fig. 2.5 The Embedded Internet system structure based on Special-purpose chip Webchip

基于网关的嵌入式 Internet 系统结构最大优点就是网关中的一个 IP 地址可以提供多个嵌入式系统同时连入 Internet 的能力。但它需要 PC 或高性能嵌入式系统等作为网关，增加了整个系统的运行成本。同时有些系统结构在嵌入式系统与网关之间采用私有协议，影响了这种系统结构的开放性可扩展性。

2.2.2.2 直连嵌入式 Internet 系统结构

让嵌入式系统连入 Internet 最直接的方法就是让嵌入式系统本身支持标准的

TCP/IP 协议栈。MCU 处理机像 PC 机一样直接处理 TCP/IP 协议，一般需要高档的处理机，如 32 位的 ARM、SH3、MIPS 等 MCU 和一些单周期指令速度较高的 8 位 MCU，如 AVR、SX 等。

对 TCP/IP 协议的具体处理又有 2 种方法。一种方法是采用实时操作系统 RTOS，用软件方式直接处理 TCP/IP 协议。这种方法实现起来很灵活，功能也很强大，技术要求高；另一种是采用固化了 TCP/IP 协议的硬件芯片，如 Seiko Instruments 公司的 57600A 等，它支持 HTTP、SMTP、POP3、 MIME 等多种协议，通过外部硬件电路处理 TCP/IP 协议。该方式与前一种相比更方便，开发难度有所降低，但还是需要熟悉 TCP/IP 协议和相关接口。基于 Webit 的直连嵌入式 Internet 系统结构就是一个典型的应用实例，这种结构采用的是前一种 TCP/IP 协议处理方法。

Webit 是沈阳东大新业信息技术股份有限公司研制开发的用于嵌入式系统接入 Internet 的一个实用产品，它将 MCU 和以太网控制器集成到一块小板卡上，将它内置到嵌入式系统中就可以完成嵌入式系统与 Internet 的连接。Webit 有自己的 IP 地址，有很高的集成度，它将通讯协议处理部分独立出来，这样开发人员就省去了网络通讯协议部分的设计，可将主要精力放在应用系统本身[19]。

Webit 实际上是个嵌入式瘦 Web 服务器，其中运行的 TCP/IP 通讯协议是经过简化了的。直连的嵌入式 Internet 系统结构无需网关，实现比较灵活，开发和运行的成本较低，但需要独立的 IP 地址。

图 2.6 基于 Webit 的嵌入式 Internet 系统结构
Fig. 2.6 The Embedded Internet system structure based on Webit

2.3 嵌入式 TCP/IP 协议

把嵌入式设备连接到 Internet，实现数据信息交换、远程访问和设备控制，必须给它们嵌入 TCP/IP 协议栈或者给它们构建一个嵌入式 Web 服务器，TCP/IP 协议栈是其核心技术。嵌入式 Internet 与 MCU 技术密切相关。利用嵌入式系统实现嵌入式互联网方案的技术难点是：如何利用嵌入式系统有限的资源对信息进行 TCP/IP 协议处理，使之变成可以在互联网上传输的 IP 数据包。我们的宗旨就是要最大限度地利用嵌入式系统资源，根据 TCP/IP 协议对网络数据信息进行最高效的处理。

2.3.1 TCP/IP 协议介绍

Internet 通信协议对计算机系统的 CPU 速度、存储器容量等的要求比较高，用

于 PC 系统不存在任何困难，但是用于自身资源有限的嵌入式系统就必须根据需要有所取舍，所以应合理选择通信协议的实现和处理方案。

TCP/IP 协议通常被认为是一个四层协议系统，每一层分别负责不用的通信功能。分层的原理是将每一层协议的实现细节对相邻协议层加以屏蔽，提供服务访问接口进行层间数据传递，各层数据独立封装，发送和接收端的各层协议存在逻辑上点对点(peer-to-peer)连接。局域网 Internet 协议层次模型[20]，如图 2.7 所示。

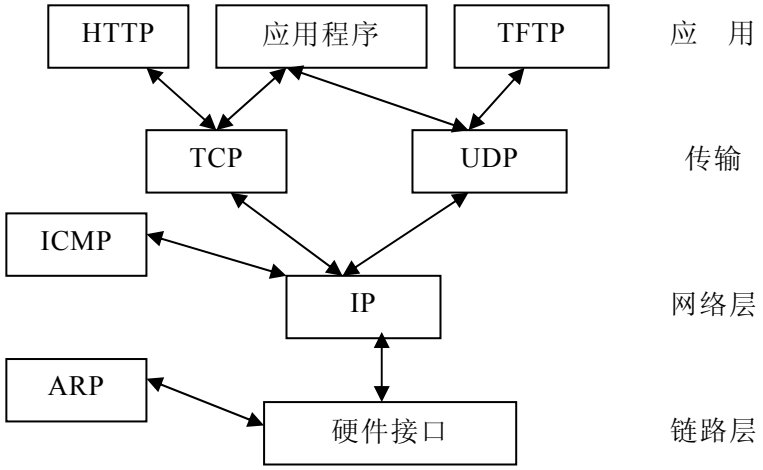

图 2.7 局域网 Internet 协议层次模型
Fig. 2.7 Model of LAN's Internet protocol level

数据封装是体现协议层次模型的重要特征。每层协议按照自己的方式进行数据成帧。数据发送时，各层在收到的上一层数据前面添加对应的头部信息，进行数据封装，然后传递到下一层；数据接收时，各层对数据进行解包，剥离出头部信息，进行适当的保存，然后将数据传递到上一层处理。

2.3.2 TCP/IP 协议进程模型

协议的进程模型是指如何在操作系统中把协议软件划分为不同的进程。通常 TCP/IP 的实现有三种进程模型：

(1) TCP/IP 协议的每一层是一个单独进程。链路层是一个进程，IP 层是一个进程，传输层是一个进程。这样的好处是网络协议的每一层都非常清晰，代码的调试和理解都非常容易。但是最大的坏处是数据跨层传递时会引起上下文切换(context switch)。对于接收一个 TCP segment 要引起 3 次 context switch(从网卡驱动程序到链路层进程，从链路层进程到 IP 层进程，从 IP 层进程到 TCP 进程)。通常对于操作系统来说，任务切换是要浪费时间的，过频的 context switch 是不可取的。

(2) 另外一种方式是在操作系统内核中实现 TCP/IP 协议栈。应用程序通过操作系统的系统调用(system call)和协议栈来进行通讯。这样的 TCP/IP 协议栈

就限定于特定的操作系统内核了，如 Windows 就是这种方式。

(3) 所有 TCP/IP 协议栈都在一个进程中实现，这样 TCP/IP 协议栈就和操作系统内核分开了。而应用层程序既可以是单独的进程也可以驻留在 TCP/IP 进程中。如果应用程序是单独的进程可以通过操作系统的邮箱，消息队列等进程间通信机制和 TCP/IP 进程进行通信[21]。

本课题就采用第三种进程模型来实现嵌入式 TCP/IP 协议栈。

2.3.3 嵌入式 TCP/IP 协议栈的实现要素

随着嵌入式工业的持续增长，BSD4.4 (Berkeley Software Distribution Version4.4) TCP/IP 源代码被直接或者经过修改以后移植到 Internet。大多数 RTOS 厂商和协议厂商将 BSD TCP/IP 作为实现嵌入式协议栈的基础，但是修改和利用其源代码、并移植到嵌入式实时环境实现，必须对以下的因素加以考虑：

(1) 缓存管理

网络固有的异步特性要求系统创建并管理缓冲区保持网络数据。缓冲区管理是实时系统设计的重要方面。缓冲区分配方案可分为：静态分配和动态分配。方案的选择由嵌入式系统的实际需要决定。

(2) 最小化数据拷贝

现在大多数 NIC 采用 DMA 方式将接收到的数据直接存放到缓冲区，系统通过指针操作，而不是数据拷贝方式，将缓冲区中的数据沿协议栈向上传递，减少由于数据拷贝引入的系统开销。

(3) 最小延迟时间

实时操作系统 RTOS 的存在不应增加额外的延迟。处理中断的接口必须快并且具有确定性。在处理物理帧的发送和接收请求的中断过程中，RTOS 不应增加任何延迟。处理分组要求大量的上下文转换和 CPU 处理，这样就增加了使用操作系统的重要性，即具有最小线程任务切换时间的操作系统。

(4) 并发处理

最早的 TCP/IP 协议实现是基于 UNIX 系统，依靠操作硬件中断级和软件中断级的临时改变来消除资源竞争问题。但是在由任务构造的系统中，共享资源的访问需要额外的保护方法。

(5) 定时器管理

为了减少 CPU 资源的使用，避免并发问题的发生，协议中用于连接管理的定时器，要由 RTOS 统一负责超时和重设定时[21]。

(6) 可移植和可裁减

由于嵌入式应用的要求千差万别，各种嵌入式应用系统的要求不尽相同，并且在嵌入式应用中对产品的成本、价格比较敏感，存储器的容量往往都是比较有限的，因此必须根据嵌入式网络产品的具体功能，对完整的 TCP/IP 协议栈功能进行裁剪，特别是对应用协议提供可裁剪性，以满足用户的需求。嵌入式应用的多样性决定了嵌入式应用平台也是变化多端的。因此，在我们开发网络协议栈软件的过程中，保证软件的可移植性是非常重要的。这样，在对嵌入式产品进行软、硬件升级的过程中除了与硬件直接相关的部分代码需要重新编写以外，不必再对上层协议进行大的修改。

第三章 μC/OS-Ⅱ在 ARM 上的移植

3.1 嵌入式实时操作系统 μC/OS-Ⅱ概述

μC/OS-II 是由 Labrosse 先生编写的一个公开源代码的实时操作系统内核，它提供了实时系统所需的基本功能，内核简单高效，实时性好。μC/OS-II 主要是针对嵌入式应用，用户可以对它进行移植、裁剪和固化，核心部分代码只有 8. 3KB，短小精悍。其源码公开，注解详尽，组织结构协调，便于理解和掌握，已经被成功移植到 51、ARM 等几十种 8 位、16 位和 32 位微处理器中，在嵌入式开发领域中得到了广泛的应用。

μC/OS-II 是一个多任务实时操作系统，任务是系统执行和调度的基本单位。这里的任务与其它操作系统中描述的进程基本类似。系统可以管理 56 个用户任务，每个任务都有自己的栈空间，用户任务越多，需要到 RAM 就越大。在 μC/OS-II 系统中，应用程序作为 μC/OS-II 内核的一个子任务，并与内核一起编译组成一个完整的可执行程序，也正是这种特殊的调度机制，保证了系统的实时性[22]。

本课题采用 μC/OS-II 主要基于以下的考虑：首先，它的内核是完全免费的，用户不需要支付任何费用，采用它有利于降低系统开发成本；其次，它的源代码是公开的，并且仍在不断的升级，增加新功能。源代码的开放可以使得用户根据实际要求对源代码进行取舍，去掉不必要的变量和不使用的函数，提高系统性能。另外，由于对系统内核有源代码级的了解，用户可以添加自己的模块，与原有系统内核兼容，使得系统具有可扩展性；再次，系统内核实用性强、可靠性高；最后，该操作系统内核对处理器以及 ROM，RAM 资源的要求不高，有利于移植。

3.2 μC/OS-II 的内核原理

多任务系统中，内核负责管理各个任务，或者说为每个任务分配 CPU 时间，并且负责任务之间的通讯。内核提供的基本服务是任务切换，之所以使用实时内核可以大大简化应用系统的设计，是因为实时内核允许将应用分成若几个任务，由实时内核来管理它们。内核本身也增加了应用程序的额外负荷，代码空间增加 ROM 的用量，内核本身的数据结构增加了 RAM 的用量。但更主要的是，每个任务要有自己的栈空间，一般来说内核本身对 CPU 的占用时间一般在 2 到 5 个百分点之间，通过提供必不可缺少的系统服务，诸如信号量管理，邮箱、消息队列、延时等，实时内核使得 CPU 的利用更为有效。μC/OS-II 的内核结构如图 3.1 。

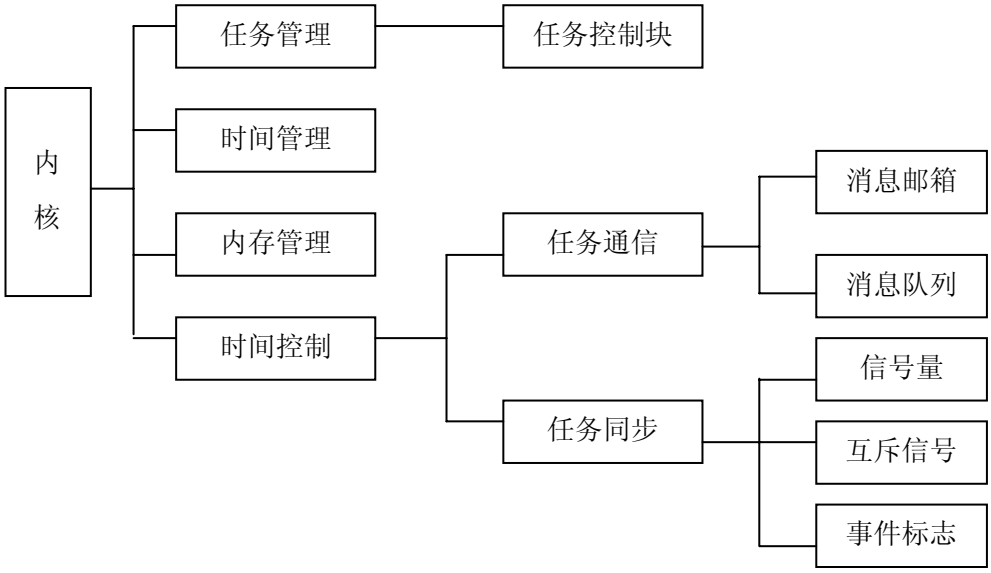

图 3.1 μC/OS-II 内核结构
Fig. 3.1 The kernel structure of μC/OS-II

μC/OS-II 的工作核心原理是：尽可能地使最高优先级的就绪任务处于运行状态。首先，初始化 MCU，再进行操作系统初始化，主要完成任务控制块 TCB 初始化，TCB 优先级表初始化，TCB 链表初始化，事件控制块(ECB)链表初始化，任务的创建等等；然后就可以开始创建新任务，并可在新创建的任务中再创建其它的新任务；最后调用 OSSTART( )函数启动多任务调度。在多任务调度开始后，启动时钟节拍开始计时，此节拍给系统提供周期性的时钟中断信号，实现延时和超时确认。

3.3 μC/OS-II 的任务管理和调度

3.3.1 任务管理

一个任务，也称作一个线程，是一个简单的程序，该程序可以认为 CPU 完全只属该程序自己。实时应用程序的设计过程，包括如何把问题分割成多个任务，每个任务都是整个应用的某一部分，每个任务被赋予一定的优先级，有它自己的一套 CPU 寄存器和自己的栈空间。μC/OS-II 可以管理 64 级任务，但系统保留了 4 个最高优先级和 4 个最低优先级，因此用户可以使用 56 个应用任务，但是必须给每个不同的任务赋予不同的优先级，μC/OS-II 总是运行进入就绪状态优先级最高的任务。

每个任务都是一个无限的循环。每个任务都处在以下 5 种状态之一的状态下，这 5 种状态是睡眠态，就绪态、运行态、等待态和中断服务态。这 5 种状态之间的转化见图 3.2。

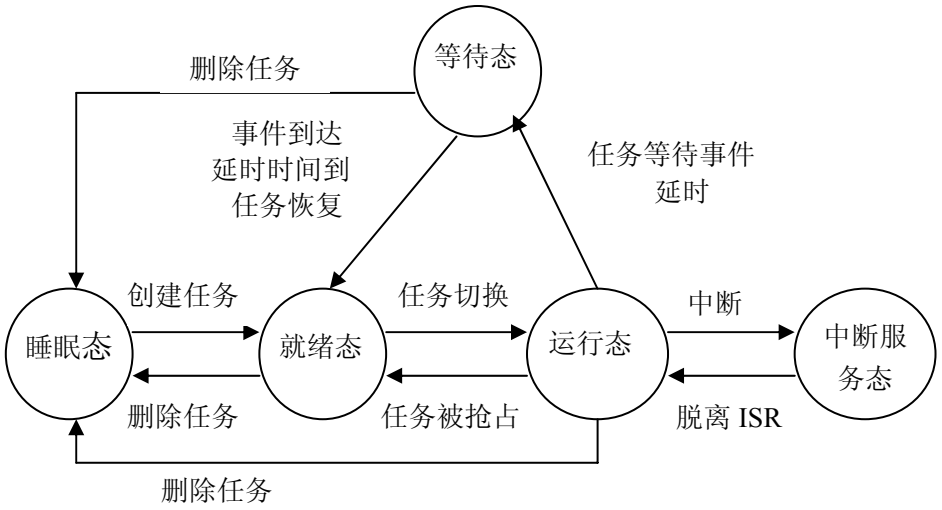

图 3.2 任务的状态转化
Fig. 3.2 condition transformation of task

睡眠态是指任务驻留在程序空间(ROM 或 RAM)，还没有交给 μC/OS-II 来管理。把任务交给μC/OS-II是通过调用下述2个函数之一：OSTaskCreate( )或OSTASKCreateExt( )来实现的。

就绪态意味着任务已经准备好，可以运行了，但是由于该任务的优先级比正在运行的任务的优先级低，暂时还不能运行。任务一旦建立，这个任务就进入了就绪态，准备运行。一个任务可以通过调用 OSTaskDel( )返回到睡眠态，或通过调用该函数让另一个任务进入睡眠态。

运行态指该任务已经得到 CPU 的使用权，正在运行中。调用 OSStart( )可以启动多任务，而且只能在启动时调用一次，该函数运行用户初始化代码中已经建立的、进入就绪态的优先级最高的任务。任何时刻只能有一个任务处于运行态，就绪的任务只有当所有优先级高于这个任务的任务都转为等待状态，或者是被删除了，才能进入运行态。

等待态是指该任务在等待某一事件的发生。正在运行的任务可以通过调用以下 2 个函数之一，将自身延迟一段时间。这两个函数是 OSTimeDly( )或 OSTimeDlyHMSM( )。这个任务于是进入等待状态，一直到函数中定义的延迟时间到。这两个函数会立即强制执行任务切换，让下一个优先级最高的、并进入了就绪态的任务运行。正在运行的任务可能需要等待某一事件的发生，可以通过调用以下函数之一实现：OSFlagPend( )、OSSemPend( )、OSMutexPend( )、OSMboxPend( )或 OSQPend( )。如果某事件并未发生，调用上述函数的任务就进入了等待状态，直到等待的事件发生了。

中断服务态是指如果中断没有被关闭，发生中断时，CPU 提供相应的中断服务，原来正在运行的任务暂不能运行，被中断的任务于是就进入了中断服务状态。响应中断时，正在执行的任务被挂起，中断服务子程序控制了 CPU 的使用权，中断返回后 μC/OS-II 会从所有就绪态中选择优先级最高的任务进入运行态。当所有的任务都处于等待时间发

生或等待延迟时间结束的状态时，μC/OS-II 执行空闲任务。

μC/OS-II 最多可以管理 64 个任务。为了简化设计，μC/OS-II 规定各个任务的优先级必须不同。任务的优先级号就是任务编号，任务的优先级号越低，任务的优先级越高。μC/OS-II 提供进行任务管理的各种函数，包括创建、删除任务，改变任务优先级，挂起和恢复任务等。系统初始化时会自动运行两个系统任务：一个是空闲任务 OSTaskIdle( )，它的优先级最低，该任务在没有其它任务进入就绪态时运行；另一个任务是统计任务 OSTaskStat( )，它的优先级次低，负责计算当前 CPU 的利用率。

μC/OS-II 为每个任务分配了一个任务控制块(OS_TCB)。与大多数操作系统的进程控制块类似，μC/OS-II 的 TCB 中包含了任务运行和管理的信息。当此任务处于挂起等待状态时，内核就把与该任务相关的状态信息保存到 TCB 中。当任务重新被调度执行时，该任务 TCB 中的信息就被调入 CPU 寄存器中，该任务得以继续运行。μC/OS-II 中所有的 TCB 都通过*OSTCBNext 和*OSTCBPrev 链接成一个用指针 OSTCBFreeList 指向的空闲 TCB 链表。当任务创建的时候，就从该空闲链表中取一个空闲 TCB 并初始化。

3.3.2 任务的调度

μC/OS-II 是可占先式内核，采用的是最高优先级调度算法，该算法的关键是如何在最短的时间内找到具有最高优先级的任务，以便进行任务切换。对于最高优先级任务的查找，最简单且易于实现的方法是顺序检索：将所有任务按优先级从高到低进行排序，查找时从队列头节点开始检索，遇到的第一个就绪的任务就是最高优先级的就绪任务。μC/OS-II 利用哈希表来定位最高优先级的就绪任务。

μC/OS-II 最多可管理 64 个任务，用一个 8 字节的数组 OSRdyTbl[8]来实现。每个任务对应于该数组中的一位，同时每一组的就绪状态又对应于一个 8 位变量 OSRdyGrp 中的一位。在数组 OSRdyTbl[ ]中，每个字节的 8 位表示对应的 8 个任务是否就绪，就绪就置 1。同时，只要每组中有任意一个任务就绪，OSRdyGrp 中的对应位也置 1。数组 OSRdyTbl[ ]和变量 OSRdyGrp 构成任务就绪表。任务就绪表的示意图如图 3.3 所示。

内核主要是通过操作 OSRdyTbl[ ] 和 OSRdyGrp 来实现任务的调度，例如一个任务在被建立的时候，这个任务就被赋予了特定的优先级，之后就通过如下算法使任务进入就绪态：

```
/*使 OSRdyGrp 的相应位置位*/
   OSRdyGrp        |=OSMapTbl[prio>>3];
/*使 OSRdyTbl[ ]的相应位置位*/
   OSRdyTbl[prio>>3]   |=OSMapTbl[prio&0x07];
```

OSMapTbl[ ]是在 ROM 中的屏蔽字，用于限制 OSRdyTbl[ ]数组的元素下标在 0 到 7

之间。

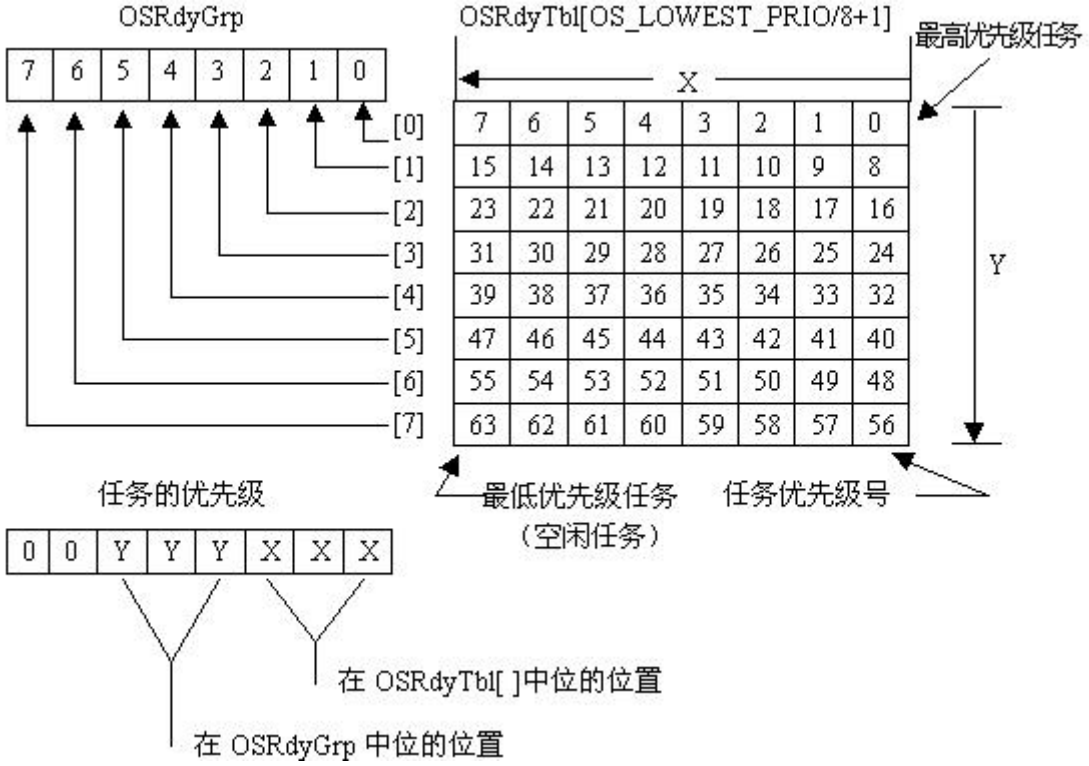

图 3.3 μC/OS-II 的任务就绪表

Fig. 3.3 The task ready list of μC/OS-II

当任务进入等待挂起状态或任务被删除进入休眠态时，需调用如下算法：

```
/*将 OSRdyTbl[ ]中相应位清 0*/
   if((OSRdyTbl [prio>>3] & = OSMapTbl[prio&0x07])= =0)
/*若该任务组没有任务就绪，则将 OSRdyGrp 相应位清 0*/
   OSRdyGrp & = OSMapTbl[prio>>3];
```

以上算法将就绪 OSRdyTbl[ ]中相应元素的相应位清零，而对于 OSRdyGrp，只有当被删除任务所在任务组中的全组任务一个都没有进入就绪态时，才将相应位清零。

当内核进行任务调度的时候，首先就是要找出进入就绪态的最高优先级任务，算法如下：

```
y=OSUnMapTbl[OSRdyGrp];   /*任务在 OSRdyTbl[ ]中的行位置*/
x=OSUnMapTbl[OSRdyTbl[y]];   /*任务在 OSRdyTbl [ ]中的列位置*/
OSPrioHighRdy=(INT8U) (y<<3) + x; /*算出任务的优先级*/
```

最后得出的 OSPrioHighRdy 就是处于就绪态的最高优先级任务的优先级号。为了找到那个进入就绪态的优先级最高的任务，并不需要从 OSRdyTbl[0]开始扫描整个就绪任务表，只需要查另外一张表，即优先级判定表 OSUnMapTbl[256]。

我们可以把任务的优先级看作是一种描述符，内核通过它来进行任务操作。但获得任务的优先级并不是最后目的，我们最终想得到的是任务的任务控制块(TCB)。μC/OS-II

提供了一个任务控制块优先级表 OSTCBPrioTbl[ ]，TCBPrioTbl[ ]是个指针数组，根据数组的下标号来存放指向具有对应的优先级号 TCB 的指针。例如，指向任务优先级为 6 的 TCB 的指针就存放在 OSTCBPrioTbl[6]里面。其最高优先级任务的找寻可以用下面的语句实现：

OSTCBHighRdy=OSTCBPrioTbl[OSPrioHighRdy];

最高优先级任务的找寻是通过建立就绪任务表来实现的。μC/OS-II 中的每一个任务都有独立的堆栈空间和任务控制块(TCB)，任务控制块的第一个成员变量就是保存的任务堆栈指针。任务调度模块首先用变量 OSTCBHighRdy 来记录当前最高优先级就绪任务的 TCB 地址，然后调用 OS_TASK_SW( )函数来进行任务切换。

如上面分析，μC/OS-II 的任务调度所花的时间是常数，与应用程序中建立的任务数无关。而且与 Linux 的调度相比，μC/OS-II 的任务调度相当简洁，效率也高出不少。

3.4 μC/OS-II 的中断及时钟节拍

3.4.1 中断

μC/OS-II 的中断服务子程序要用汇编程序来写。下面是用户中断服务子程序的一般处理过程的伪代码：

保存全部 CPU 寄存器；

调用 OSIntEnter 或 OSIntNesting 直接加 1；

中断服务；

调用 OSIntExit；

恢复所有 CPU 寄存器；

执行中断返回指令；

如上所示，μC/OS-II 的中断处理与 PC 的中断处理类似。当发生中断时，首先保护现场，CPU 寄存器入栈，再执行中断服务子程序，之后恢复现场，CPU 寄存器出栈，最后执行中断返回。但是 μC/OS-II 需要知道用户在做中断服务，因此用户应该调用 OSIntEnter( )，或者将全局变量 OSIntNesting 直接加 1。内核通过判断 OSIntNesting 的值来了解中断嵌套的情况。中断服务子程序退出时要调用脱离中断函数 OSIntExit( ), OSIntExit( )首先应该将 OSIntNesting 减 1，如果中断服务程序使更高优先级的任务就绪且没有中断嵌套，还必须进行中断级的任务调度，使高优先级的就绪任务运行。

3.4.2 μC/OS-II 的时钟节拍

μC/OS-II 需要用户提供周期性信号源，用于实现时间延时和确认超时。μC/OS-II 的时钟节拍(clock tick)是特定的周期性中断。中断之间的时间间隔取决于不同的应用，一

般在 l0ms 到 200ms 之间。时钟的节拍中断使得内核可以将任务延时若干个整数时钟节拍，以及当任务等待事件发生时，提供等待超时的依据。μC/OS-II 的时钟节拍中断类似于一个定时器中断，事实上在移植到特定的处理器上时，通常都用目标处理器的定时器做时钟节拍源。每个时钟节拍中断到来时调用时钟节拍中断服务将对任务延时做一次裁决。

OSTimtick( )是时钟节拍中断服务函数，函数中大量的工作是给每个用户任务控制块 OS_TCB 中的时间延时项 OSTCBDly 减 1(如果该项不为 0 的话)。OSTimTick( )从 OSTCBList 开始，沿着 OS_TCB 链表做，一直做到空闭任务。当某任务的任务控制块中的时间延时项 OSTCBDly 减到了零，这个任务就进入了就绪态，而被挂起的任务则不会进入就绪态。OSTimTick( )的执行时间直接与应用程序中建立了多少个任务成正比。

3.5 μC/OS-II 的通信机制

3.5.1 事件控制块

任务或者中断服务子程序可以通过发信号的方式来保护任务间的共享数据以及实现任务间的同步和通信。所有的信号都被看成是事件(event)。μC/OS-II 提供了几种基本的事件：信号量(Semaphore)，邮箱(Mailbox)，消息队列(Queue)等。μC/OS-II 为各种事件定义了一个通用的数据结构，叫做事件控制块(ECB)。事件控制块的数据结构成员如下所示，该数据结构用来维护一个事件控制块的所有信息。

```
void      OSEventPtr; /*指向消息或者消息队列的指针*/
INT8U     OSEventTbl[OS_EVENT_TBL _SIZE] ; /*等待任务列表*/
INT16U    OSEventCnt; /*计数器(当事件是 Semaphore 时)*/
INT8U     OSEventType; /*事件类型*/
INT8U     OSEventGrp; /*等待任务所在的组*/
```

OSEventTbl [ ] 和 OSEventGrp 很像前面的 OSRdyTbl[ ]和 OSRdyGrp，只不过前者包含的是等待某事件的任务，而后者包含的是系统中处于就绪态的任务。而且每个事件都有自己的事件控制块，而 OSRdyTbl[ ]在系统中是独一无二的，负责整个系统的任务调度。

每个事件控制块中的 OSEventTbl [ ] 和 OSEventGrp 构成该事件的等待任务列表。这个等待任务列表与前面讲述的任务就绪表非常相似。μC/OS-II 提供了一个空闲事件控制块链表(OSEventFreeList)，在创建一个事件的过程中需要从 OSEventFreeList 中取出一个空闲事件控制块并初始化。

如果有任务请求这个事件并且该事件可得，那么该任务就在得到该事件后继续运行；若请求失败，任务不可得，那么该任务就将被挂起并且被放到这个事件控制块的等

待任务列表中，直到超时或者有其它的任务或中断子程序发送了该事件，该任务才能从事件的等待任务列表中移除并继续运行；如果同时有多个任务请求一个事件，那么就与任务就绪表类似，每个等待事件的任务的优先级决定了谁可以获得等待的事件。对事件控制块的操作如表 3.1。

表 3.1 事件控制块的操作函数

Table 3.1 The operation function of Event Control Blocks

操作函数	主要功能
OS_EventWaitListInit( )	初始化一个空的等待任务列表
OS_EventTaskRdy( )	从等待任务列表中删除任务，并使该任务就绪，同时传递事件给该任务
OS_EventTaskWait( )	使任务进入等待状态并将该任务放到事件控制块的等待任务列表中
OS_EventTo( )	由于等待超时而将任务置为就绪态

3.5.2 通信机制的实现

μC /OS-II 的通信机制是建立在事件控制块的基础上的。由于信号量、邮箱、消息队列的操作方式类似，这里以邮箱为例来具体分析通信机制的实现。

邮箱是 μC/OS-II 的一种通信机制，邮箱包含的内容是一个指向一条消息的指针。它可以使一个任务或者 ISR 向另一个任务发送一个指针型的变量。该指针指向一个包含了特定“消息”的数据结构。

要使用一个邮箱，首先要调用 OSMboxCreate( )来创建一个邮箱，包括取得一个空闲的事件控制块，并初始化它的等待任务列表，还要指定指针的初始值。如果这个初始值是 NULL，那么使用邮箱的目的是为了通知一个事件的发生；如果这个初始值是非 NULL 的指针(如 void *1)，那么邮箱就被当作一个二值信号量使用，用来共享资源。

接收邮件和发送邮件分别由 OSMboxPend( )和 OSMboxPost( )来完成，实质上就是指针变量的传递，要理解整个过程首先必须明白下面几个域的作用：

(1) OSTCBEventPtr：任务控制块的成员，OS_EVENT 类型的指针，指向该任务所等待事件的事件控制块。如果该任务在等待某事件的发生，OSTCBEventPtr 就指向这个等待事件的事件控制块。如果该事件发生了(或者等待超时)，则使 OSTCBEventPtr 指向 NULL。

(2) OSTCBMsg：任务控制块的成员，指向传递给该任务的消息的指针。任务等待的消息收到后就用 OSTCBMsg 来指向收到的消息。

(3) OSEventPtr：事件控制块的成员，当事件是 Mbox、Queue 时才使用。事件是

Mbox 时指向一条消息；事件是 Queue 时指向一个容纳消息指针队列的数据结构。当消息被发送后，若没有任务在等待该消息，该消息就被 OSEventPtr 指向。若有任务等待该消息，则消息直接被传递给等待任务列表中的任务，用该任务的 OSTCBMsg 来指向收到的消息。

可用图 3.4 所示的状态机来表示以上各个域的条件变迁，任务 M、N 表示两个不同的任务，图中各个状态为事件发生后的瞬时状态，虚线箭头表示事件控制块中有消息，msg 表示等待或发送的消息。

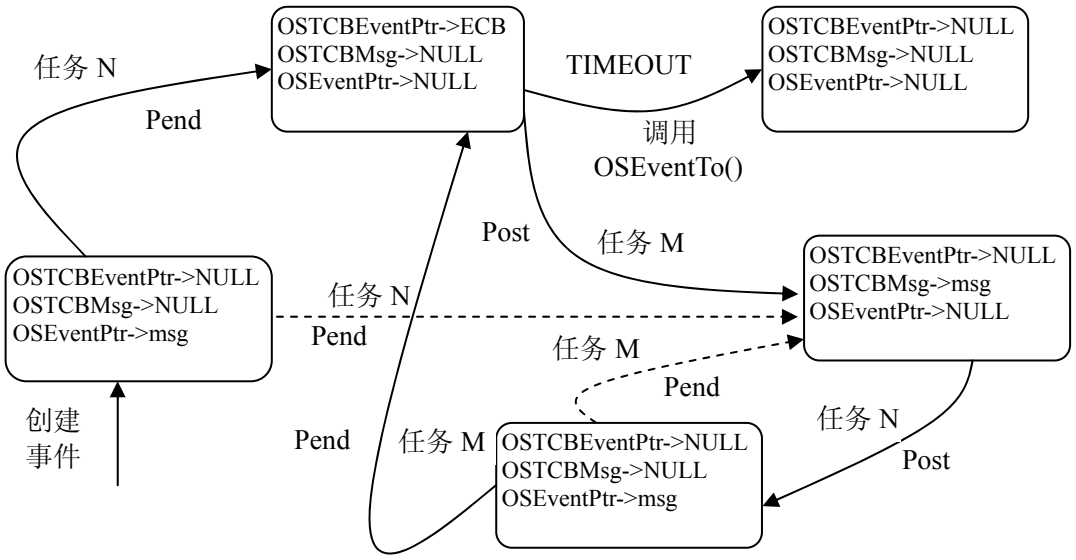

图 3.4 随事件操作而变化的结构域

Fig. 3.4 The change structure territory with event operation

消息队列很像是多个邮箱，实际上我们可以将消息队列看作是多个邮箱组成的数组，只是他们共用一个等待任务列表。在邮箱中，事件控制块的 OSEventPtr 指向一条消息。而队列是邮箱数组，还要对它进行维护，就需要增加数据结构。μC/OS-II 为消息队列提供了一个队列控制块(OS_Q)的数据结构，并通过 OS_EVENT 中的 OSEventPt：域链接到对应的事件控制块，还需要定义一个与含有消息队列最大消息相同个数的指针数组 MsgTb1[ ]，如图 3.5 所示。

信号量是一种约定机制，在多任务内核中信号量用于控制共享资源的访问以及任务间的同步。信号量要用到一个计数域 OSEventCnt，用邮箱也可以实现信号量的功能，且 μC/OS-II 中它们的实现相似。

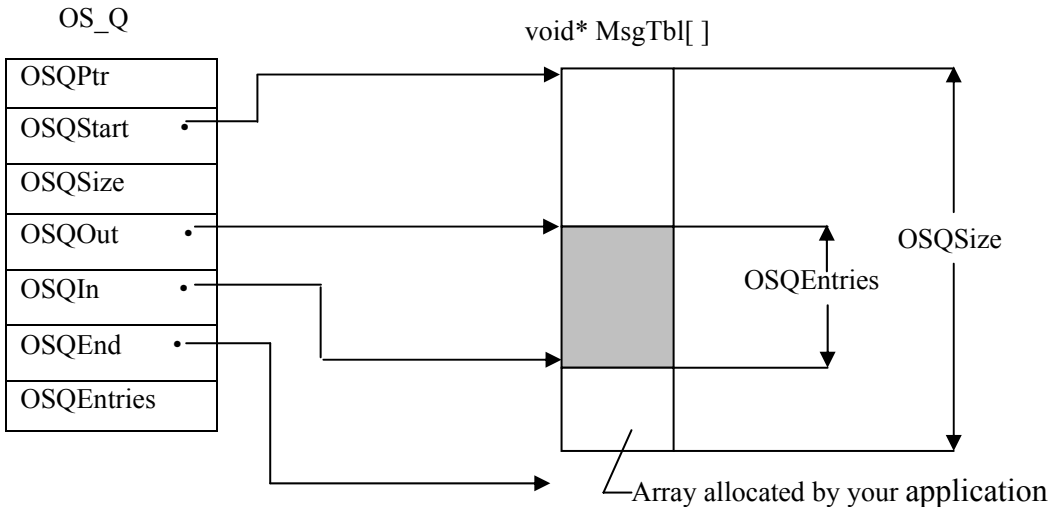

图 3.5 OS_Q 的数据结构
Fig. 3.5 The data structure of OS_Q

3.6 μC/OS-II 在 S3C44B0X 上的移植

3.6.1 移植条件

嵌入式实时操作系统 μC/OS-II 可以在多种平台上运行，有时候我们得到的操作系统版本并不是针对我们使用的 CPU 的版本，这时，就需要对操作系统进行移植，使它能够在我们的 CPU 上运行。μC/OS-II 的移植需要满足以下要求[22]，

(1) 处理器的 C 编译器可以产生可重入代码；

(2) 可以使用 C 调用进入和退出临界区代码；

(3) 处理器必须支持硬件中断，并且需要一个定时中断源；

(4) 处理器需要能够容纳一定数据的硬件堆栈；

(5) 处理器需要有能够在 CPU 寄存器与内核和堆栈交换数据的指令。

S3C44B0X 处理器完全满足上述要求。

3.6.2 任务模式的取舍

对于 ARM7TDMI 处理器来说，具有用户、系统、管理、中止、未定义、中断和快中断 7 种工作模式，其中除了用户模式外，其他的都是特权模式。管理、中止、未定义、中断和快中断都与相应的异常相联系，不适合作为任务的工作模式；而系统模式除了是特权模式外，其他的与用户模式一样，因而可选的任务模式只有用户模式和系统模式。为了尽量减少任务代码错误对整个程序的影响，缺省的任务模式都是用户模式，可选为系统模式。同时提供接口使任务可以在用户模式和系统模式这两种模式之间切换。

3.6.3 移植实现

μC/OS-II 在设计时已经充分考虑了可移植性，μC/OS-II 的移植关键就是要改写与处

理器硬件相关的几个函数，移植所需完成的工作包括以下内容：

(1) 用#define 设置一些常量的值(OS_CPU.H)。

(2) 声明 10 个数据类型(OS_CPU.H)。

(3) 用#define 声明 2 个宏(OS_CPU.H)。

(4) 用 C 语言编写 6 个简单的函数(OS_CPU_C.C)。

(5) 根据硬件编写了 4 个函数(OS_CPU_ A.ASM)。

图 3.6 说明 μC/OS-II 的结构及其与硬件的关系。

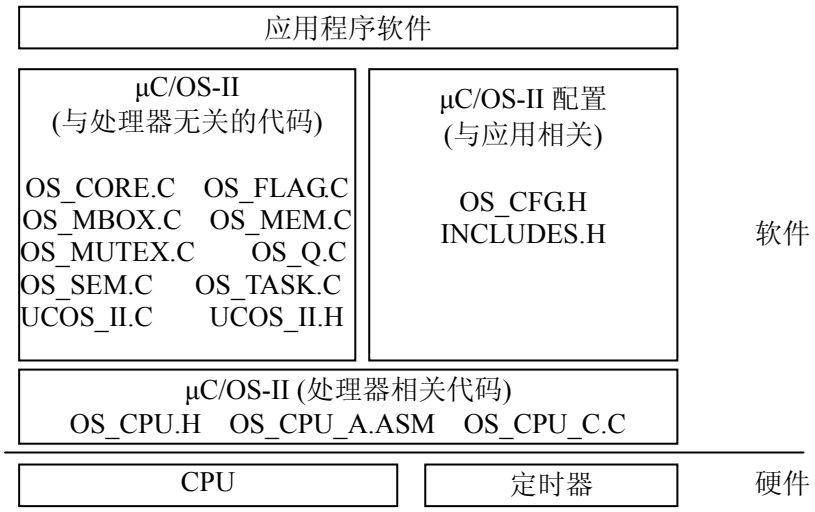

图 3.6 μC/OS-II 硬件/软件体系结构图

Fig. 3.6 The hardware/software system structure of μC/OS-II

3.6.3.1 OS_CPU.H 文件

(1) 数据类型的定义

μC/OS-II 不使用 C 语言中的 short, int, long 等数据类型的定义，因为它们与处理器类型有关，隐含着不可移植型。μC/OS-II 代之以移植性强的数据类型，这样，既直观又可移植。因此就需要移植这些定义。根据 ADS 编译器的特性，OS_CPU.H 中数据类型定义实现代码如下：

```
typedef unsigned char    BOOLEAN;
typedef unsigned char    INT8U;    /*Unsigned 8 bit quantity*/
typedef signed char    INT8S;    /* Signed 8 bit quantity*/
typedef unsigned short    INT16U;    /* Unsigned 16 bit quantity*/
typedef signed short    INT16S;    /* Signed 16 bit quantity*/
typedef unsigned int    INT32U;    /* Unsigned 32 bit quantity*/
typedef signed int    INT32S;    /* Signed 32 bit quantity*/
typedef float    FP32;    /* Single precision floating point */
```

(2) 堆栈增长方向

堆栈增长方向是和编译器有关的，当进行函数调用时，入口参数和返回地址一般都会保存在当前任务的堆栈中，编译器的编译选项和由此生成的堆栈指令就会决定堆栈的增长方向。这里 OS_STK_GROWTH 定义如下：

```
#define   OS_STK_GROWTH   1   /*堆栈由高地址向低地址增长*/
```

(3) 相关宏定义

OS_CPU.H 文件中还包括与硬件系统结构相关的定义，如开关中断的宏定义，以及进行任务切换的宏定义。

```
#define   OS_TASK_SW       OSCtxSw
#define OS_ENTER_CRITICAL( )   (cpu_sr = OSCPUSaveSR( ))
#define OS_EXIT_CRITICAL( )    (OSCPURestoreSR(cpu_sr))
```

与所有的实时内核一样，μC/OS-II 需要先禁止中断再访问代码的临界段，并且在访问完毕后重新允许中断。μC/OS-II 定义了两个宏来禁止和允许中断：OS_ENTER_CRITICAL( )和 OS_EXIT_CRITICAL( )。在我们的实现中考虑到提高代码的效率，采用汇编来实现禁止和允许中断，具体实现为 OSCPUSaveSR( )和 OSCPURestoreSR(cpu_sr)。具体实现在 OS_CPU_A.S 中。

在 μC/OS-II 中，OS_TASK_SW( )函数是用来实现任务切换。任何非运行状态的任务堆栈都设置成一次中断发生后的样子，OS_TASK_SW( )实际上是模拟发生了一次中断，在这个中断返回的时μC/OS-II 会调用 OSCtxSw( )函数实现任务切换。

3.6.3.2 OS_CPU_C.C 文件

OS_CPU_C.C 中共定义了 6 个函数。但是最重要的是任务堆栈的初始化函数 OSTaskStkInit( )，其他都是对系统内核的扩展时用的。

(1) 任务堆栈初始化

μC/OS-II 的任务，在没有执行的时候就象是刚刚被中断一样，堆栈则是任务上下文(context)的一部分，CreateTask( )调用 OSTaskStkInit( )来给任务做一个初始的任务上下文堆栈。在 ARM 体系结构下，任务堆栈空间由高至低依次将保存 pc，lr，r12，r1l， r10，...，r1，r0，CPSR，SPSR。每个任务都是一个函数，执行这个任务就是调用这个函数[23]。我们设想，这个任务的函数是 void Taskl(void* pdata)；执行这任务就是用 BL 指令来调用函数，刚执行完 BL 指令，中断发生了。OS_CPU_C.C 中的 OSTaskStkInit( )大体实现如下：

```
OS_STK *OSTaskStkInit(void (*task)(void *pd), void *pdata, OS_STK *ptos, INT16U opt){
    unsigned int *stk;
    opt = opt;    /* 'opt' is not used, prevent warning */
    stk = (unsigned int *)ptos;    /* Load stack pointer */
```

```
    /* build a context for the new task */
    *--stk = (unsigned int) task;           /* pc */
    *--stk = (unsigned int) task;           /* lr */
    *--stk = 0;                                 /* r12 */
    …                                           /* r11-r2*/
    *--stk = 0;                                 /* r1 */
    *--stk = (unsigned int) pdata;          /* r0 */
    *--stk = (SVC32MODE|0x0);       /* cpsr    IRQ, FIQ disable */
    *--stk = (SVC32MODE|0x0);       /* spsr    IRQ, FIQ disable */
    return ((void *)stk);
}
```

(2) 系统 Hook 函数

此外，在这个文件里面还需要实现几个操作系统规定的 hook 函数，如下：OSTaskCreateHook( )、OSTaskDelHook( )、OSTaskSwHook( )、OSTaskStatHook( )和 OSTimeTickHook( )，在一般情况下，则只需要简单地将它们都实现为空函数就可以了。

3.6.3.3 OS_CPU_A.ASM 文件

需要根据硬件写 4 个函数：

(1) OSStartHighRdy( )

OSStartHighRdy( )是在 OSStart( )多任务启动之后运行。OSStart( )找到最优先的任务，最后就调用 OSStartHighRdy( )来启动那个任务。在 OSStartHighRdy( )被调用之前，优先级最高的任务的 TCB 控制块的指针已经被 OSStart( )放在全局变量 STCBHighRdy 中，即 OSTCBHighRdy 已经指向了就绪态中的优先级最高的任务控制块中。OSStartHighRdy( )的任务归结如下：

进来了之后，先可以调用一个允许用户定义的 HOOK 函数；把全局变量 OSRuning 设置为 TRUE，标志多任务系统开始运行；从 OSHighRdy 指向的 TCB 中得到堆栈指针，放在 R13 里；从堆栈中恢复所有其他的相关寄存器，包括 CPSR，R0-R12，LR，PC。

这样，任务函数就像被 BL 指令调用了一样，从 Task 的第一条指令开始执行了。源代码如下：

```
OSStartHighRdy   /*运行优先级最高的就绪任务*/
LDR   r4,   addr OSTCBCur;   /*得到当前任务的 TCB 地址*/
LDR   r5,   addr OSTCBHighRdy;   /*得到高优先级任务的 TCB 地址*/
LDR   r5,   [r5];   /*得到堆栈指针*/
LDR   sp,   [r5];   /*切换到新的堆栈*/
```

```
STR   r5,  [r4];  /*设置新的当前任务的 TCB 地址*/
LDMFD   sp! ,  {r4};  /*从栈顶得到新的状态*/
MSR   CPSR_cxsf, r4;  /*设置处理器模式*/
LDMFD   sp!,  {r0-r12,lr,pc};  /*开始新的任务*/
```

(2) OSCtxSw( )

OSCtxSw 函数是任务级的上下文切换，它是当任务因为被阻塞而主动请求 CPU 调度时被执行，由于此时的任务切换都是在非异常模式下进行的，因此用于区别中断级别的任务切换。

该函数的主要工作是先将当前任务的 CPU 现场保存到该任务堆栈中，随后，获得最高优先级任务的堆栈指针，最后从该堆栈中恢复此任务的 CPU 现场，从而完成一次任务切换。

(3) OSIntCtxSw( )

OSIntCtxSw 函数中断级的任务切换，它是在时钟中断 ISR(中断服务例程)中发现有高优先级任务等待的时钟信号到来，则需要在中断退出后并不返回被中断任务，而是直接调度就绪的高优先级任务执行。它的原理基本上与任务级的切换相同，只需要对堆栈指针做相应的调整。

(4) OSTickISR( )

OSTickISR 函数是中断处理函数，主要任务是负责处理时钟中断，调用系统实现的 OSTimeTick 函数，如果有等待时钟信号的高优先级任务，则需要在 ISR 中执行。因 TickISR 也是 IRQ 中断的一种，所以只需要在中断调用 OSTimeTick 然后再开启 Timer 中断就可以了。其它相关的两个函数是 OSIntEnter( )和 OSIntExit( )，都需要在 ISR 中执行。

完成上述工作，再在系统启动文件和板子配置文件做相应的配置后 μC/OS-II 就可以在 S3C44B0X 硬件平台上运行。

3.7 移植测试

在开发板上移植 μC/OS-II 后，还须测试其移植是否成功。测试 μC/OS-II 这样多任务实时内核并不复杂，甚至可以在没有应用程序的情况下测试。这样做有两个好处：第一，避免使本来就复杂的事情更加复杂；第二，如果出现问题，可以知道问题出在内核代码上而不是应用程序。

本节编写了一个简单的测试程序来验证 μC/OS-II 内核的任务调度在 S3C44B0X 处理器系统上运行的正确性。本测试程序建立了三个任务：

Taskl：串口上输出“Task1 running”，然后延迟 1 秒，优先级设为 10；

Task2：串口上输出“Task2 running”，然后延迟 2 秒，优先级设为 11；

Task3：串口上输出“Task3 running”，然后延迟 3 秒，优先级设为 12。

测试结果如图 3.7 所示。

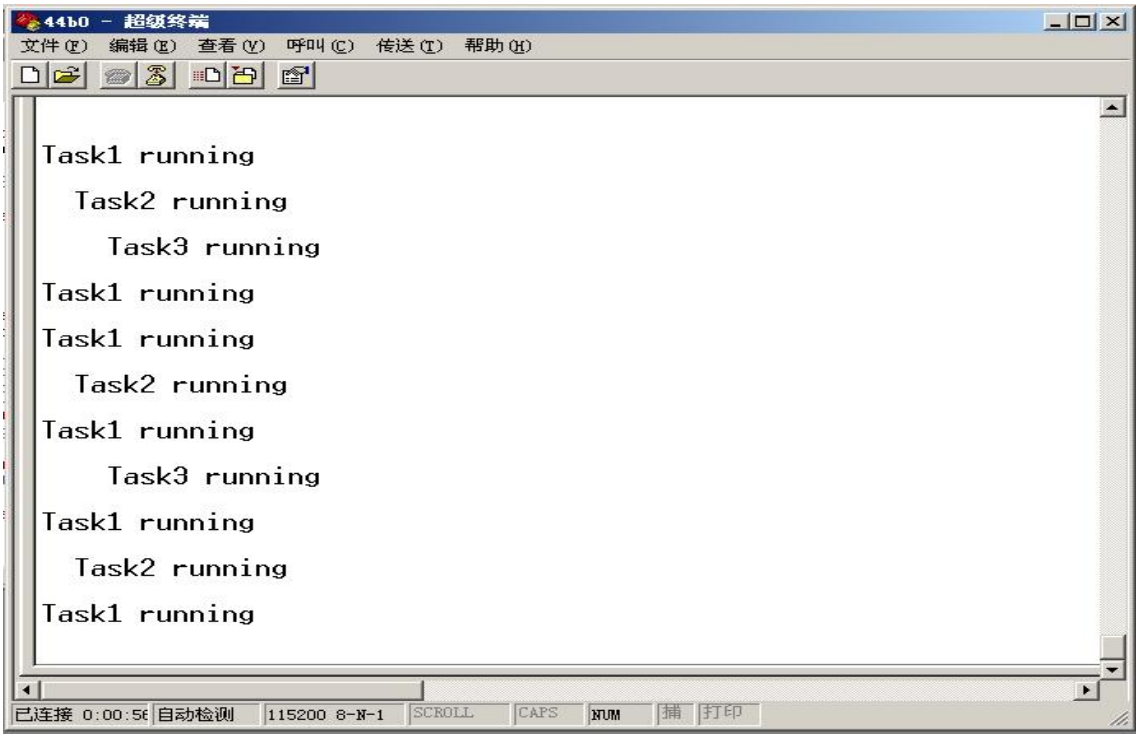

图 3.7 μC/OS-II 移植测试结果

Fig. 3.7 The result of porting and testing μC/OS-II

至此，经上述过程移植的操作系统内核已经可以正常在 S3C44B0X 芯片上运行。

第四章 嵌入式 TCP/IP 协议栈设计与实现

4.1 协议栈实现框架

本课题设计的协议栈采用了 Lwip 中的缓冲区管理机制[24]并在此基础上进行优化，同时，为了使协议栈适应 μC/OS-II 这样的 RTOS，协议采用基于中断驱动的模式，协议栈整体作为任务的形式实现，API 层可以适应多个网络程序同时运行。该协议实现框架中设计了两个任务和一个中断响应函数，此外还有一个接口良好的 API 层，以方便用户编写自己的网络应用程序。该协议栈的实现结构如图 4.1 所示。

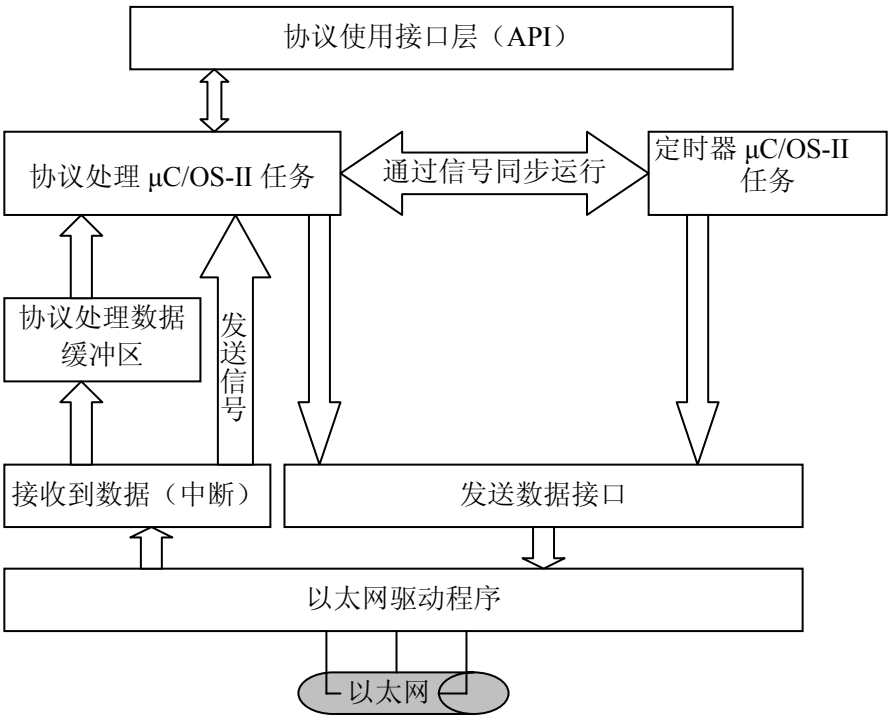

图 4.1 协议栈实现结构
Fig. 4.1 The realization structure of protocol stack

该协议最低层是基于中断的以太网驱动程序，它以中断的方式异步地把数据发送给协议栈任务（协议栈中 TCP/IP 协议处理 μC/OS-II 主任务），中间经过了协议处理数据的缓冲区；用户程序通过 API 层接收数据和发送数据；定时协议任务专门处理 TCP 相关数据；非 TCP 的数据则直接交给以太网络驱动程序发送。整个协议栈的数据传输都通过全局缓冲区管理模块来管理。

4.2 嵌入式 TCP/IP 协议栈设计

4.2.1 嵌入式 TCP/IP 协议栈需求分析

嵌入式 TCP/IP 协议和标准的 TCP/IP 协议相比，在协议组成、协议实现算法上做了

较大的简化，简化原则是：在满足基本通信要求的前提下，减少协议的数量；在保证协议基本功能的前提下，去除一些不相关的功能。但是嵌入式 TCP/IP 协议在设计和实现上和标准的 TCP/IP 协议一样采用的是四层结构[25]，其分层结构见图 4.2。

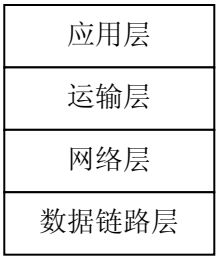

图 4.2 嵌入式 TCP/IP 协议分层结构
Fig. 4.2 The layer structure of Embedded TCP/IP protocol

本课题设计的这个轻量级的嵌入式 TCP/IP 协议栈是在标准 TCP/IP 协议栈的基础上进行简化和优化，具体而言，简化可以分为两个不同层次：

(1) 总体上，从 TCP/IP 协议中选取必须的协议，构成嵌入式 TCP/IP 协议栈。

(2) 在局部范围内，针对已选取的协议，进一步对它的功能进行选取和简化，使其在存储资源、计算能力、通信带宽等方面消耗最小。

使用嵌入式 TCP/IP 协议的目的是使资源受限的嵌入式设备与传统的计算机网络互连互通，因此只选取每层与数据传输直接相关的协议。本着实用的原则，本文对各层所需实现的协议进行了分析：

(1) 数据链路层

链路层主要负责把上层封装好的数据经网络接口设备发送出去并进行可靠的传输，把收到的有效数据存入接收数据缓冲区中并向网络层传递；对软件设计就是实现网络控制芯片的驱动程序。

(2) 网络层

IP 协议是网络层的主要协议，同时还包含 ARP, RARP, ICMP 和 IGMP 四个辅助协议。IP 协议实现了 IP 最基本的功能，包括发送、接收、转发。不支持 IP 分片功能。由于嵌入式系统生成的数据较小，能够直接通过网络传输，因此不需要将输出数据报进行分片;但有可能接收到分片的数据报,因此要对输入的数据报片重组，以保证与传统计算机网络的通信。

其中 RARP 逆地址解析协议是为无盘工作站或动态分配 IP 地址所设计，在嵌入式系统应用中各节点的 IP 地址通常都是预先指定的，因此这部分就不予实现。IGMP 因特网组报文协议用来实现组播通信，这在嵌入式应用系统中的优势并不突出。因为要实现组的管理，需要保存分组信息，而且在判断报文接收者地址时，还需遍历本地分组信息。即使确实需要进行分组通信也可以通过应用程序层予以实

现，所以本着节省资源的目的，IGMP 协议也不予实现。

ARP 地址解析协议实现了网卡 MAC 地址到 IP 地址的映射，ICMP 网间控制报文协议用于控制网络传输中出现的各种问题，这两个协议都是本文需要研究的。ARP 协议实现了 ARP 请求和 ARP 应答协议；ICMP 协议只实现了 ICMP 中类型号为 0，代码号为 0 的 Ping 应答协议。

(3) 传输层

传输层包括两个协议：TCP 和 UDP。这两个协议是本课题研究并实现的嵌入式 TCP/IP 协议的核心部分。对于 UDP 协议，因为 IP 层取消了 IGMP 协议的实现，而 IGMP 是 UDP 协议组播功能实现的重要条件，所以在实现 UDP 协议时就取消了组播功能的实现；在该层中需要重点实现的是 TCP 协议，由于标准的 TCP 协议太过于复杂和功能强大，而在资源有限的嵌入式系统中不可能全部去实现 TCP 协议的所有功能，所以只实现该协议的基本连接、传输功能，以保证协议数据的可靠传输。

(4) 应用层

应用层包括 SMTP, TELNET, FTP, HTTP, NFS 等协议，这些协议在嵌入式采集控制系统中的应用较少，且较占用资源，本课题为了实现嵌入式设备连入 Internet，仅仅实现了 HTTP 协议，对于其他的协议不予实现。

为了便于移植、配置和对传输数据有效的进行管理，本课题实现的轻量级嵌入式 TCP/IP 协议的实现还包括如下辅助模块：

(1) 通用接口层：该层把所有与硬件、OS、编译器相关的部分独立出来，保存在一个目录中，当需要移植到不同的系统时，只修改这个目录中与硬件、OS、编译器对应的文件即可。因此，实现的嵌入式 TCP/IP 协议可以方便的移植。其中实现的重点是对 OS 进行了封装，形成了对上层软件的统一接口。当 TCP/IP 需要系统调用时，并不直接调用 OS 提供的函数，而是使用该层的接口。

(2) 嵌入式 TCP/IP 协议的配置项：为适合不同的应用需求，通过一个可配置项来设置一些可配置信息。根据实际应用情况，选取合适的参数对于节约存储空间、加快程序运行具有很大的作用。

4.2.2 缓冲区管理

TCP/IP 协议栈中缓冲区方案的选择对 TCP/IP 协议栈的性能影响很大。而在嵌入式系统中存储空间有限，因此在这些系统中设计和实现 TCP/IP 协议栈，不但要考虑 TCP/IP 协议栈本身的大小，还要考虑缓冲区的设置。标准 TCP/IP 协议通常都采用动态内存管

理方式来管理缓冲区，但是这种方式会带来频繁的内存申请、释放和内存拷贝，这对于嵌入式系统是非常不利的，所以本课题设计的嵌入式 TCP/IP 协议栈采用全局缓冲管理模块来实现协议栈缓冲区的管理，采用静态内存管理方法，即在系统初始时，一次申请到足够的内存，这保证了网络系统不会用尽所有的可用内存，不会干扰其他程序的运行。整个协议栈都是以 packet 为单位进行数据处理，该管理模块主要负责数据的分配、释放、合并、拆分等工作。这些数据操作在通信协议中的实现是经常需要的，因此这部分实现思想可完全重用到其他类似数据处理的应用中去。全局缓冲管理模块用到的主要数据结构如下所示：

```
typedef struct zbuffer{
    struct zbuffer    *next; /*数据链标记下一个数据块*/
    void    *pdata; /*该块指向的数据区*/
    u16_t    tot_len; /*整个数据链的总体大小*/
    u16_t    len; /*这个块的自身大小*/
}zbuffer_t;
```

全局缓冲管理模块采用数据链方式来管理数据，通过数据链管理可以实现从中断发送，到协议处理、用户接收等整个过程中，数据只需要存在一份拷贝，减少了对数据空间需求。

4.2.3 “一次拷贝”技术

由于 TCP/IP 协议的层次特性，每个协议层都有自己的数据格式。发送数据时，各个协议层从自己的上层协议层接收数据，然后加上本层的控制信息再交给自己的下层协议层，这个过程叫打包。其中，该控制信息只有其他主机上的同层协议层才能正确解释；接收数据时，各个协议层从自己的下层协议接收数据，然后取出本层的控制信息再把剩下部分数据交给上层的协议层，这个过程叫拆包。

用户数据在从本地主机传输到远程主机的过程中，需要不断地拆包和打包。如果在拆包和打包时，各协议之间均采用数据拷贝进行数据传递，则将大大增加系统的存储和数据处理开销，从而降低系统实时性能。为了解决这一矛盾，有些协议产品采用“零拷贝”技术，是指 TCP/IP 协议栈没有用于各层间数据传递的缓冲区，协议栈各层间传递的都是数据指针，只有当数据最终被驱动程序发送出去或是被用户应用程序取走时，才进行真正的数据搬移。

“零拷贝”技术的优点是系统存储开销小，去掉了不必要的数据搬移，使得数据处理快。但是“零拷贝”技术也有一个致命的缺点。由于全部操作都是在用户程序空间进行，使得用户程序空间不能立即回收，实际上间接加大了用户程序空间负载，特别是当

应用程序的存储空间也很有限时，将给应用程序的存储空间管理带来一定的不便。

为此，本设计中，采用“一次拷贝”技术以缓解上述矛盾。所谓“一次拷贝”，即指数据在整个 TCP/IP 协议栈的上下流动过程中，进行了一次数据搬移。具体地说，在发送过程中，当应用程序把数据交给 TCP/IP 协议栈时，协议栈首先把数据装入协议栈缓冲区，并将用户数据缓冲区释放给应用程序，以便再次使用。然后，TCP/IP 协议栈在自己内部再采用“零拷贝”技术进行处理。这样，一方面使得用户数据空间不致长期被系统占用，另一方面使得系统的性能又得到了很好的保证。

4.3 网卡驱动程序设计

本系统使用 ARM 板子上自带的 RTL8019AS 芯片实现网络接口，RTL8019AS 是台湾 Realtek 公司生产的高集成度以太网控制其芯片，它集成了介质访问控制子层(MAC)和物理层的功能，可以方便地应用在基于 ISA 总线系统或单片机系统[26]。另外，它还具有与 NE2000 兼容、软件移植性好以及价格低廉等优点。本文采用中断的方式接收网络上的数据，RTL8019AS 的驱动程序主要分为初始化、发送、接收、中断处理这几部份。

4.3.1 数据帧的组成

以太网协议不止一种，本文用的是 802.3 帧格式。它的帧结构如图 4.3 所示。物理信道上的收发操作均使用这个帧格式。其中，前导序列、帧起始位、CRC 校验由硬件自动添加/删除，与上层软件无关[27]。

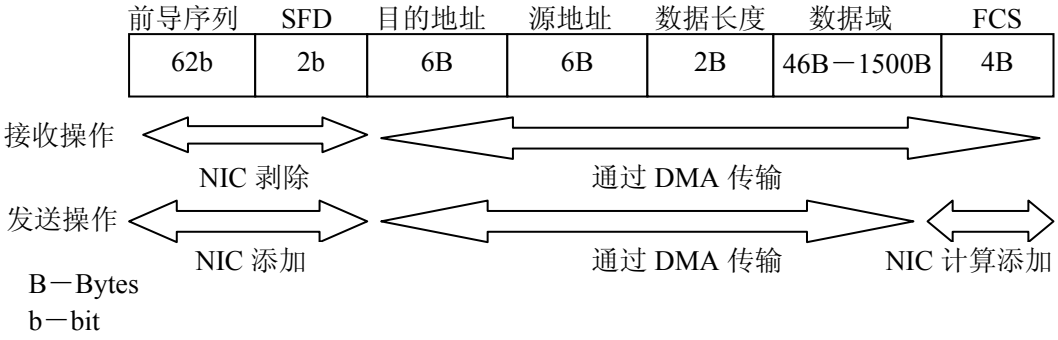

图 4.3 802.3 帧结构

Fig. 4.3 The frame structure of 802.3

值得注意的是，收到的数据包格式并不是以太网帧的真子集，而是如图 4.4 所示。8019 自动添加了“接收状态、下一页指针、以太网帧长度(以字节为单位)” 三个数据(共 4 字节)。这些数据的引入方便了驱动程序的设计，体现了软硬件互相配合协同工作的设计思路，当然，发送数据包的格式是 802.3 帧的真子集，如图 4.4 所示。

接收状态	下页指针	帧长度	目标地址	源地址	数据长度	数据域	FCS
1B	1B	2B	6B	6B	2B	46B－1500B	4B

图 4.4 RTL8019AS 接收帧结构
Fig. 4.4 The received frame structure of RTL8019AS

4.3.2 初始化操作

网卡的初始化主要包括：芯片复位、接收缓冲区和发送缓冲区的设置、工作模式设置，物理地址设置、中断处理程序安装等。网卡中断号和使用的端口范围与实际板卡设置有关，比如本文讨论的板卡网卡中断号为 3，占用端口范围为 0x300-0x31F[28]。具体操作函数原型为：u8_t low_r8019init (znetif_t *pnetif)。参数 pnetif 是协议代码与网络驱动之间可以交互的参数，使用这个参数使协议栈可以支持多个网络设备，通过 pnetif 可以区别操作指向的具体操作设备。初始化操作流程如图 4.5 所示。

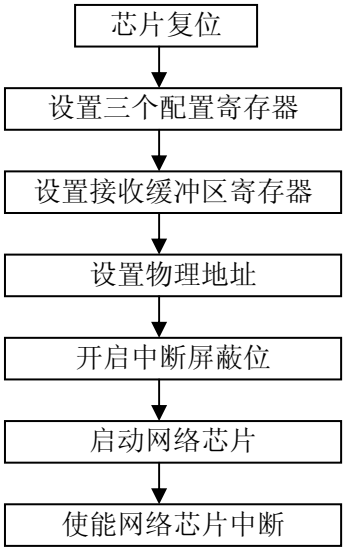

图 4.5 RTL8019AS 初始化流程图
Fig. 4.5 Flow chart of RTL8019AS Initialization

4.3.3 数据接收操作

根据 μC/OS-II 操作系统内存管理的特点，网卡的接收数据若在中断程序中处理，则可能出现把收到的数据从双口 RAM 中向内存中读取时，因为申请不到空闲内存块而只能丢弃当前数据(因为中断处理程序不能挂起)，所以在中断程序先检查数据的有效性，若无效则直接丢弃，有效则设置一个标志，数据的读取和处理由 TCP/IP 协议任务中的网络接口对象完成。在中断处理程序中判断数据有效性可以减少无效数据对协议任务的影响。通常中断程序中的操作要尽量少，所以对数据有效性的检查只是检查数据包头部是否有效，即前 18 个字节，并不读出后面的数据块部分[29]。

接收数据操作设计为两部分，中断处理程序和数据处理程序。中断处理程序属于操

作系统内核部分，数据处理程序在网络接口对象中完成，两者通过全局互斥量 Recv_Sem 来同步。当网卡收到数据后产生中断，中断处理程序给 Recv_Sem 解锁；协议栈任务不断地检查 Recv_Sem，若已解锁则由网络接口对象通过驱动程序从网卡缓冲中读出数据包，并给 Recv_Sem 加锁。协议栈任务在检查互斥量时采用无等待获取状态操作，这样就不会协议栈任务挂起而影响任务中其它操作[30]。接收数据框图如图 4.6 所示。

在网卡中断处理程序中，先判断中断类型，根据中断类型执行不同操作；如果是正常接收数据，则进行的内容包括数据有效性的检查和标志位的设置。网卡中断处理程序原型为：u8_t znet_r8019_int (void)，该函数主要完成把协议需要处理的任务送交给协议主任务处理的功能。而驱动的接收函数是由中断异步调用的，其函数原型为：u16_t znet_r8019_Recv(zbuffer_t *zbuf)，参数 zbuf 是要接收的数据包指针，函数返回读出的字节数。

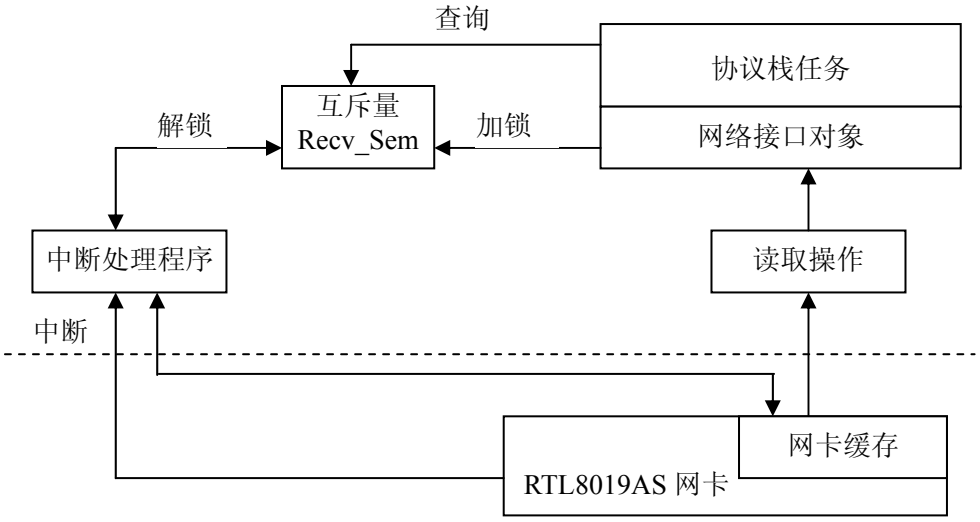

图 4.6 网卡接收处理框图

Fig. 4.6 Netcard receive processing block

4.3.4 数据发送操作

网卡的发送数据过程比较简单，该函数只负责操作网络硬件发送一个准备好的数据。发送操作函数原型为：u8_t znet_r8019_send(znetif_t *pnetif, ipaddr_t *dest_ip)。其操作流程大致为：设置 Remote DMA 写操作的相关寄存器；启动 Remote DMA 操作，写数据到网络芯片；等待 Remote DMA 传输完成；设置发送缓冲区寄存器；最后启动发送命令，发送数据。

4.4 网络层协议实现

4.4.1 IP 协议的实现

本课题设计的嵌入式 TCP/IP 协议栈网络层需要实现的协议有 IP 协议、ICMP 协议和 ARP 协议。

IP 协议提供不可靠、无连接的数据报传送方式。所谓不可靠，是指它不能保证 IP 数据报能成功地到达目的地，另外 IP 数据报的校验仅仅针对数据报头部，对于数据部分没有校验检查[31,32]。要避免这些错误的产生，还需要上一层协议的支持，如 TCP 协议。无连接是指 IP 并不维护任何关于后续数据报的状态信息，每个数据报的处理是相互独立的。

4.4.1.1 IP 协议的简化

不同类型的网络对它的数据链路层的 MTU(最大传输单元)的规定有所不同，当发送数据大于链路层的 MTU 数据时，需要进行 IP 分片处理；当接收到 IP 分片后，需要进行重装处理。

由于嵌入式设备网络通信数据量不大，能够直接通过网络传输，且分片与重组需要复杂的内存管理机制，会影响到通信效率，因此，不需要将输出数据报进行分片。但嵌入式系统设备可能接收到分片的数据，为了与传统计算机网络互连互通，必须实现重组功能。在本课题的 TCP/IP 协议的设计中，不对输出数据报进行分片，只实现对输入数据片的重组。这样既不影响通信的正常进行，又降低了 IP 协议处理的复杂度，精简了代码，提高了效率[33]。

4.4.1.2 IP 协议中定义的数据结构

由于本课题实现的嵌入式 TCP/IP 协议不考虑路由表及分片数据，IP 协议中只定义了 IP 数据报首部。

```
struct ip_hdr{
    u16_t   ver_hl_tos;   /* IP 协议版本号、首部长度及服务类型*/
    u16_t   totle_len; /*总长度*/
    u16_t   id; /*数据报标识*/
    u16_t   offset; /*片偏移*/
    u16_t   ttl_proto; /*生存时间&协议类型*/
    u16_t chksum; /*校验和*/
    ipaddr_t src; /*源 IP 地址*/
    ipaddr_t dest; /*目的 IP 地址*/
    };
```

4.4.1.3 IP 协议的实现模型

IP 协议是嵌入式 TCP/IP 协议实现的中心环节，它为不同网络的主机之间发送数据报的操作序列提供无连接服务，通过在数据报前添加 IP 协议头，使每个数据报具有寻址能力，从而实现对无连接的 IP 数据报的寻径、传送[34]。数据流程图如图 4.7 所示。

IP 协议的实现模型包括的模块有：IP 数据报接收处理模块、IP 数据报发送处理模块、IP 初始化模块和 IP 重组模块。对应的处理函数分别是：ip_input( )、ip_output( )、ip_init( )和 ip_reass( )。

IP 初始化函数声明为 void ip_init (void)，实行功能比较简单，只是对 IP 输入输出过程需要用到的变量、参数赋值。

IP 发送数据的函数原型为：u8_t ip_output(zbuffer_t *pbuf,struct ipaddr_t *src, struct ipaddr_t *dest,unsigned int ttl_proto)。函数 ip_output( )负责接收上层协议产生的数据，由上层的 UDP 发送函数 udp_output( )、TCP 发送函数 tcp_output( )以及同层的 ICMP 协议中送来的数据交给 ip_output( )函数。在 ip_output( )函数中，通常对于不等于本机 IP 地址的数据报应该通过路由表选择下一目的地址进行转发，但本协议不考虑转发情况，所以直接丢弃。

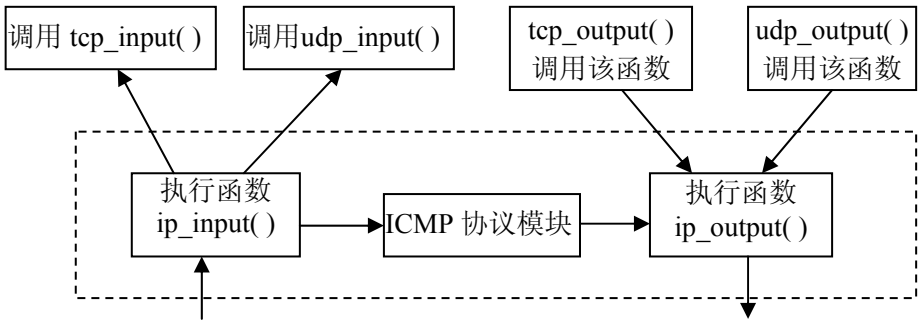

图 4.7 IP 协议的函数关系图

Fig. 4.7 Functional relation chart of IP protocol

IP 接收数据的函数原型为：u8_t ip_input(znetif_t *pnetif, zbuffer_t *pbuffer)。函数 ip_input( )被下层的网络接口层函数调用以接收 IP 数据报。在 ip_input( )处理函数中，首先完成对 IP 版本、头部长度的检查，并计算 IP 校验和，然后检查数据报的目的 IP 地址，如果不是自己的 IP 地址，则丢弃该数据报；如果是自己的 IP 地址，则再检查该数据报的类型，如果是 UDP 报文，则调用上层的 UDP 接收函数 udp_input( )；如果是 TCP 报文，则调用上层的 TCP 接收函数 tcp_input( )；如果是 ICMP 报文，则调用 ICMP 协议实现中的接收函数 icmp_input( )，处理完毕以后重新由 ip_output( )函数输出。

IP 重组的函数原型为：static struct _zbuffer *ipreass(zbuffer_t *pbuffer)。为了对数据报进行重组，定义一个数组来保存数据报片，数组中每个元素对应一个有数据报片到达的数据报，该元素包含数据报的源地址和 ID 字段，并且包含一个指针，指向此数据报片的分片链表。链表的表项是所有已到达的数据报片。片重组函数 ip_reass( )使用数据报首部中的信息来判断对应的数据报片链表。如果几个数据报片的源地址和 IP 标识符中的字段相等，那么这些数据报片就属于同一个数据报。ip_reass( )接收数据报片，在分片表中搜索，看表中是否有该数据报所属的数据报相对应的表项，对这个表项，它比较

两者的源地址和标识符字段，如果找到匹配的表项，就向链表中加入该数据报片，然后看是否全部的数据报片已到齐，可以重组数据报。如果没有发现对应信息的全面维护匹配的表项，就分配数组的第一个空闲表项，拷贝源地址和标识符字段，并将数据报片置入一个新分配的队列中。

4.4.2 ICMP 协议的实现

4.4.2.1 ICMP 协议的简化

ICMP 协议的类型很多，不同的类型由报文首部的 8 位类型字段和 8 位代码字段来共同决定。虽然这些功能都很有用，但嵌入式应用系统的资源有限，而且有些报文的处理很复杂，所以只实现其中最必需的功能。在本课题的嵌入式 TCP/IP 协议中，只实现目的不可达、端口不可达、回显请求/应答，其他 ICMP 报文直接丢弃。这些在网络数据传输过程中比较常用且更为实用。如回显报文可以快速判断网络是否畅通，端口不可达可以通知发送者需要检测接收端口是否正确[35]。

4.4.2.2 ICMP 协议中定义的数据结构

ICMP 协议中定义了 ICMP 数据报首部。

```
struct icmp_hdr{
   u8_t   type;   /*报文类型*/
   u8_t   code; /*代码*/
   u16_t   chksum; /*校验和*/
   u16_t   id; /*标识符*/
   u16_t   seqno; /*序列号*/
}ip_header_t;
```

4.4.2.3 ICMP 协议的功能实现

由于目的不可达和端口不可达报文都是输出报文，相对比较独立，所以设计 icmp_dest_unreach 和 icmp_port_unreach 分别发送这两种报文；对于回显请求/应答的处理，设计了 icmp_input 过程进行处理。

回显请求/应答的处理的函数原型为：int icmp_input(znetif_t *pnetif, zbuffer_t *pbuffer)。ICMP 输入过程只供 IP 协议调用，当 IP 协议模块收到 ICMP 报文时，通过 icmp_input 处理该包。对于 ICMP 报文，本协议只对回显请求报文处理，其它类型直接丢弃，所谓回显请求报文，就是常说的 ping 命令发出的报文。在应答时把收到的报文目的 IP 地址改为源 IP 地址，目的物理地址改为源物理地址，源 IP 地址和物理地址改为本机对应地址，然后修改报文类型为回显应答，并重新计算校验和，最后调用 ip_output( ) 发出。

发送 ICMP 目的不可达报文实现函数原型为 void icmp_dest_unreach(znetif_t *pnetif, zbuffer_t *pbuffer, enum icmp_dur_type t)。本函数也是供 IP 协议调用，当 IP 过程收到的报文目的地址不是本机地址或广播地址时，就给源地址发送 ICMP 目的不可达报文。icmp_dest_unreach 根据输入的参数解析出源地址信息，然后生成一个目的不可达报文，并调用 ip_output 发出。

发送 ICMP 端口不可达报文实现函数原型为 void icmp_port_unreach(znetif_t *pnetif, zbuffer_t *pbuffer)。本函数是在 UDP 输入处理时调用的，当 UDP 输入过程发现目的端口与本地接收端口都不一致时，就给源地址发送 ICMP 端口不可达报文。icmp_dest_unreach 根据输入的参数解析出源地址信息和目的端口号，然后生成一个端口不可达报文，并调用 ip_output 发出。

4.4.3 ARP 协议的实现

4.4.3.1 ARP 协议的简化

ARP 协议的功能是进行 IP 地址与以太网地址之间的转换。因为我们在向对方主机发送数据的时候，知道对方的 IP 地址，但可能并不知道其以太网物理地址，而要想发送基于以太网的 IP 包就必须知道目的物理地址，这个任务就是由 ARP 协议来完成的。本课题实现的 ARP 协议的具体工作包括发送 ARP 请求和响应对方的 ARP 请求，以及动态的维护一个 ARP 高速缓存。ARP 的实现可以分为输入模块、输出模块和 ARP 缓冲管理模块[36]。

输入模块：处理来自网络的 ARP 分组。

输出模块：发送数据时，网络接口软件调用输出模块，输出模块将 IP 地址与物理地址绑定，并返回一个绑定，网络接口利用这个绑定封装和发送数据。

ARP 地址映射管理模块：因为 IP 地址和物理地址的绑定不是一成不变的，随着时间的变化其绑定也有可能发生变化，因此需要对地址绑定表进行维护，主要是对 ARP 表进行查找，添加、删除以及更新操作。

4.4.3.2 ARP 协议的主要数据结构

ARP 协议中主要的数据结构有 ARP 数据报文结构和 ARP 缓冲信息结构。

(1) ARP 数据报文结构

```
struct etharp_hdr{
   u16_t   hwtype; /*硬件类型*/
   u16_t   proto; /*协议类型*/
   u16_t   hwlen_ protolen; /*硬件地址长度和协议地址长度*/
   u16_t   opcode; /* ARP 报文类型*/
```

```
    struct ethaddr_t shwaddr; /*源硬件地址*/
    struct ipaddr_t sipaddr; /*源协议地址*/
    struct ethaddr_t dhwaddr; /*目的硬件地址*/
    struct ipaddr_t dipaddr; /*目的协议地址*/
}arp_header_t;
```

(2) ARP 缓冲信息结构

```
Struct etharp_entry{
    struct ipaddr_t    ipaddr; /*IP 地址*/
    struct ethaddr_t    ethaddr; /*物理地址*/
    enum etharp_state    state; /*表项状态*/
    u16_t    ctime; /*生存时间*/
};
```

(3) 映射表表项状态

```
enum etharp_ state{
    ETHARP_STATS_EMPTY, /*空*/
    ETHARP_STATE_PENDING, /*待定*/
    ETHARP_STATE_STABLE /*确定*/
};
```

4.4.3.3 ARP 协议的功能实现

(1) ARP 输入模块实现

对于 ARP 输入，通常 TCP/IP 协议主要是为了处理 ARP 请求或应答报文，并及时更新地址映射表。结合嵌入式系统实际情况和 RTOS 对于超时的要求通常都较为严格，如果按照普通 TCP/IP 协议的设计方法，可能得不到理想的响应时间。所以，为了及时更新地址映射表，不仅使用 ARP 应答报文，还利用 IP 报文更新映射表。这样，从机可以很快获得主机地址，不会出现第一次发给主机的数据因为得不到映射而延迟发送；还有就是在正常通信过程中，由于不断利用 IP 报文更新地址映射表，各表项的有效性也大大提高，避免了由于长时间得不到更新而导致表项超时，并进而需要使用 ARP 请求报文。很显然，这样处理也有缺点，那就是 ARP 地址映射表会频繁地被更新，但相对于提高映射可靠性等优点，还是利大于弊[37]。

ARP 报文输入函数原型为 void arp_arp_input(znetif_t *pnetif, zbuffer_t *pbuffer)。ARP 报文分为请求和应答两种，对于请求报文，需要把本地的物理地址 ethaddr 填入报文中并把报文修改成应答类型，然后发出；对于应答报文，先从报文中解析出发送者的 IP 地址和物理地址，再调用 arp_update_entry 进行更新。

IP 报文输入函数原型为 void arp_ip_input(znetif_t *pnetif)。对于 IP 报文的处理与 ARP 应答报文类似，只是这时报文的格式不同。

(2) ARP 输出模块实现

对于 ARP 输出则比较简单，对外主要就是把地址映射表中没有的表项或过期的表项重新发送 ARP 请求报文，对内需要把目的主机的 IP 地址转换成物理地址。ARP 输出部分仅供网络接口对象调用，其它协议和应用程序通常不直接使用该模块；当需要发送数据时，网络接口对象调用该模块把目的 IP 地址变换成物理地址，再把填入物理地址的数据发出。

输出模块通常只会发出 ARP 请求报文，为了给网络接口对象提供地址映射功能，还设计了 arp_output 函数供其调用。ARP 请求功能函数的原型为 void arp_query(ethaddr_t *hwaddr, ipaddr_t *ipaddr)，该函数的功能是根据要请求的目的 IP，封装一个 ARP 请求报文并发出。地址解析接口函数定义为 u8_t arp_output(ipaddr_t *ipaddr, zbuffer_t *q)，根据 ipaddr 在映射表中遍历查找，若找到则把对应物理地址写入带发送的数据报 q 中，并返回发送结果；若映射失败，则调用 arp_query，把数据报暂存到 arp_pending_buf 中，并把映射表中对应表项的状态置为 ETHARP_STATE_PENDING。

(3) ARP 地址映射管理模块实现

ARP 地址映射表管理模块包括两部分，一是定期检查映射表中超时表项，并将其删除；二是更新映射表表项状态。同时该模块还要完成地址映射表初始化、查找空表项等功能[38]。

在 ARP 协议模块的设计中，比较关键的一块就是 ARP 缓冲的设计。在通用计算机系统中，ARP 一般被设计为双向数据链的形式，这样，整个缓冲可以方便地动态增减。但是这种非线性存储的链表式缓冲结构，在进行表项匹配时比较费时，不适合用于嵌入式系统[39]。在本课题的嵌入式 TCP/IP 设计中采用的是线性数组形式的 ARP 缓冲结构，不同于链表结构的是它在内存中是连续线性存储的，这样查找起来速度很快。在删除缓冲中的表项时，并没有真正的从表项中删除，而是将该表项的状态标志位清零即可。同样在向 ARP 表项中添加表项时，只是查找该表的状态标志位为零的表项，然后替换即可。由于缓冲大小有限，如果没有空闲的表项空间，按规定从缓冲区的第一个表项开始替换。

ARP 缓冲的表项数目为 ARP_BUFSIZE,这是一个全局常量。可以在协议栈编译前，根据具体的需求由嵌入式 TCP/IP 协议的配置项来灵活的设置。需要注意的是 ARP_BUFSIZE 不易设置的太大，否则会占用大量的内存资源。

ARP 缓冲除了在接收到 ARP 响应包和 ARP 请求包时需要更新外，还需要定时更新。这主要是考虑到别的主机在更换了硬件网络接口并重启后，会导致本机 ARP 缓冲中存

在错误表项，进而导致错误的数据包的发送。在 ARP 缓冲表中的表项生存时间 ctime 字段标志着该表项可以存活的时间，其中 ctime 字段值是由系统的定时器按照一定的时间间隔来进行更新的，每次将 ctime 减 1,当 ctime 的值为 0 时，说明该表项过期。可以丢弃掉。而新加入表项的 ctime 值会按照需要初始化为一个统一的值。

ARP 地址映射表管理模块初始化函数原型为 void arp_init(void)，在协议栈初始化时调用，实现对 ARP 数据报文结构的初始化以及对 ARP 映射表的初始化。

清空过期表项的函数原型为 void arp_ tmr (void)，映射表应定期进行检查，若检查到有表项 ctime 已经为 0，就清空该表项。在清空过期表项时，若该表项的状态为 ETHARP_STATE_PENDING，还需要把发送缓冲中包含相同 IP 地址的节点清空，并释放该节点指向的数据报空间。

查找空表项的函数原型为 u_8t arp_find_entry(void)，查找需遍历映射表，若找到空表项则返回表项序号，否则返回-l。

更新表项函数定义为 void* arp_update_entry (struct ipaddr_t *ipaddr,struct ethaddr_t *ethaddr)；当收到 ARP 应答报文或 IP 报文后，需要更新表项。更新过程先遍历映射表，若找到对应表项则更新，否则调用 arp_find_entry 增加一个表项。调用 arp_find_entry 可能会返回-1，说明找不到空闲表项，这时就该放弃更新。

4.5 传输层协议的实现

传输层实现的是面向无连接的 UDP 协议和面向连接的 TCP 协议。

4.5.1 UDP 协议的实现

UDP 协议是建立在 IP 协议之上，同 IP 一样提供了无连接的数据报传输。其实 UDP 报文就是基于 IP 报文的，只是对 IP 报文又一次进行了封装。UDP 报文在发送前不需要建立连接，这样使得 UDP 协议的通信效率比较高，很适合传输数据内容不是很大的突发数据报[40]。

4.5.1.1 UDP 协议的简化

UDP 协议用来提供不面向连接的、尽最大努力传输的数据流传输服务。它只是简单的将数据报从一台主机发送到另一台主机，并不保证该数据报能到达另一端。任何必须的可靠性，必须由应用层来提供。UDP 协议本身较为简单，无需要简化。但 UDP 协议通过配置项对连接的用户做了限定，最多可以处理五个 UDP 连接，默认值为一个 UDP 连接。UDP 提供的协议端口能够区分在一台机器上运行的多个程序，也就是说，每个 UDP 报文不仅传输用户数据，还包括目的端口号和源端口号，这使得目的机器上的 UDP 软件能够把报文送到正确的接收进程，而接收进程也能回送应答报文。

在同一台计算机上可以有多个进程同时使用 UDP 协议收发数据而互不影响，因为 UDP 协议使用端口号来区分不同的进程。各进程使用的端口号互不相同，当收到 UDP 报文时，先对报文进行有效性检查，然后根据接收方端口确定把数据转给那个应用程序；如果没有一个对应的进程，就丢弃报文，并给发送方发送端口不可达报文[41]。

虽然 UDP 协议收发数据不用建立连接，但协议可选择使用校验和的手段尽量保证数据的正确性。UDP 数据报的校验和是可选的，但通常都会使用。与 IP 校验和不同，UDP 数据报的校验和不仅包括对头部的校验，还包括数据部分的校验，这样也弥补了 IP 校验和的不足。UDP 数据报还包含了一个 12 字节长的伪头部，这时为了计算校验和而设置的，它包括源 IP 地址、目的 IP 地址、协议和数据报长度。其中 IP 地址在 IP 头部中也包含，目的是为了让 UDP 两次检查数据是否已正确到达目的地。

4.5.1.2 UDP 协议的实现模型

UDP 协议由 UDP 数据接收处理模块和 UDP 数据发送处理模块构成。对应的函数分别为 udp_input( )和 udp_output( )。其实现模型如图 4.8 所示。

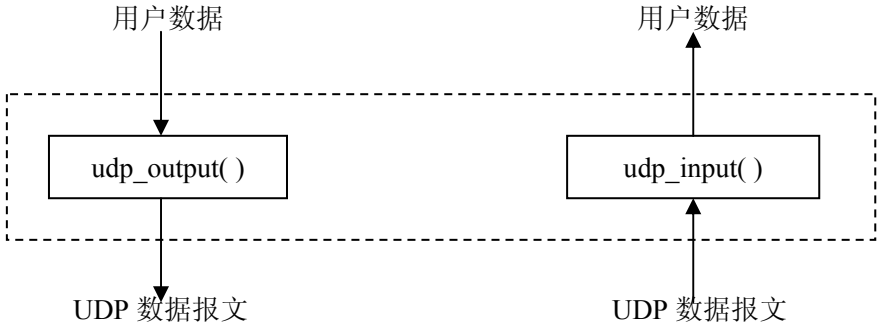

图 4.8 UDP 协议实现模型
Fig. 4.8 Realization model of UDP protocol

当 IP 模块接收到 UDP 数据报时，去掉 IP 报头，调用 udp_intput( )函数将 UDP 数据报提交到 UDP 模块。在应用程序使用 UDP 协议通信之前，首先要获得一个本地 UDP 端口号，当应用程序生成 UDP 输出之后，以目的 IP 地址、目的端口号、分组的缓冲区地址、数据长度和校验和标志等参数调用 udp_output( )函数来发送 UDP 数据报。

4.5.1.3 UDP 协议的主要数据结构

(1) UDP 头部结构

```
strct udp_hdr{
    u16_t   source_port; /*源端口号*/
    u16_t   dest_port; /*目的端口号*/
    u16_t   length; /*UDP 长度*/
    u16_t   chksum; /*校验和*/
```

```
};
```

(2) UDP 控制块

```
struct udp_pcb{
   zsocket_t  *_psocket; /*UDP 对应的 socket*/
   sys_sem_t   _sem; /*UDP 对应的事件对象*/
   u8_t   _udp_state; /*UDP 状态*/
   zbuffer_t  *_data[UDP_BUFFER_LEN]; /*UDP 数据链*/
   u8_t   _rindex; /*读数据索引*/
   u8_t   _windex; /*写数据索引*/
   } udp_pcb_t;
```

4.5.1.4 UDP 协议的功能实现

(1) UDP 输入过程实现

UDP 输入过程都在 UDP 输入函数中处理，处理后如果是有效的数据报就通过 UDP 接收函数继续向应用程序传递。UDP 输入函数原型为 void udp_input(znetif_t *pnetif, zbuffer_t *pbuffer)。输入函数的功能就是把收到的数据进行检查并送到相应的接收任务中，即先检查校验和和长度是否正确，再根据解析得出的源和目的 IP 地址及端口号遍历 udp_pcb；若找到 remote_ip 和 remote_port 与收到的数据报源地址和端口相同则通过 pcb->recv 把数据和源、目的信息传给接收任务；若找不到接收任务，则调用 icmp_port_unreach 发送 ICMP 端口不可达报文。

UDP 接收函数原型为 void udp_recv(zsocket_t *psocket , ipaddr_t *remote_ip, u16_t *remote_port, zbuffer_t **pdata, u8_t flags)，用户任务调用该函数，但是要注意原子操作，因为该函数执行的时候，协议任务可能打断该函数的执行。原子操作语句都是在 sys_enter_critical( )和 sys_out_critical( )之间。函数首先查看接收缓冲区是否有数据，如果有就从接收缓冲区中得到实际数据，并通过邮箱消息进行通知，否则进入睡眠状态等待事件唤醒。

(2) UDP 输出过程实现

UDP 的输出比较简单，其功能仅仅是封装 UDP 数据，并把该数据发送出去。

UDP 发送函数原型为 u8_t udp_send(zsocket_t *psocket , ipaddr_t *remote_ip, u16_t *remote_port, void *pdata, u16_t *psize)。发送过程首先需要得到对方的 MAC 地址，得到对方主机 MAC 地址后还需要修改输入的数据块 pdata 并封装成 UDP 数据报格式，重新计算校验和，最后通过 ip_output 发出。应用程序在调用 udp_send 之前必须进行过连接，否则无法得知目的地址和端口。

本课题实现的 UDP 都是阻塞形式的，即用户任务调用接收函数后，如果没有接收

到数据，用户任务就处于睡眠状态，其实还可以通过增加 API 调用，加入一些 socket 状态变量，实现一些非阻塞的调用，例如实现 select。

4.5.2 TCP 协议的实现

TCP 协议提供了面向连接的流传输，具有很高的可靠性。TCP 套接字在传输数据之前，必须在发送端和接收端建立一条连接，如果建立连接不成功，发送端就不会接收端发出数据。为了保证可靠性，TCP 在传输过程中，对每一份报文都要接收端确认，这样不但提高了可靠性，还保证了发送的数据都能按发送顺序到达[42]。

4.5.2.1 TCP 协议的简化

TCP 协议是 TCP/IP 协议中最复杂的协议，也是最难实现的协议。由于许多应用层协议都采用 TCP 作为传输层协议，所以，其实现的好坏直接影响到整个协议栈性能。对 TCP 协议进行简化的原则是：不改变其面向连接的特性。本课题实现的嵌入式 TCP/IP 协议在以下几个方面做了简化：

(1) 限定连接

在标准的 TCP/IP 协议中，将端口号、IP 地址、序列号、窗口尺寸和响应的传输控制块 TCB 结合，来标志不同的连接，可以为不同用户建立多个连接，并发执行。对于嵌入式应用系统，由于处理能力和可利用资源的限制，TCP 协议通过配置项对连接的用户做了限定，最多可以处理五个 TCP 连接，默认值为一个 TCP 连接。

(2) 流量控制

TCP 协议是一个高可信协议，其中的流量控制机制是保证其可靠性的一个重要措施。如果没有流量控制，则可能因接收缓冲区溢出而丢失大量数据导致许多超时重传。TCP 采用滑动窗口机制，利用可变窗口来进行流量控制。窗口的大小表示在最近收到的确认号后允许传送的数据长度。如果接收方通告窗口为 0，则表示在当前的接收方没有能力接收另外的数据，必须等待新的确认信息来改变窗口大小[43]。标准 TCP/IP 协议在流量控制的实现上过于复杂需要消耗大量的系统资源，本课题实现的嵌入式 TCP 协议依据 RFC 文档的要求，对流量控制机制进行了重新设计。

(3) 拥塞控制

拥塞控制是 TCP 协议的一个重要功能，它对防止网络拥塞，改变网络性能起到了一定的作用，但同时也增加了 TCP 协议的复杂性，本课题的 TCP 协议对传统的拥塞控制算发进行了改进，降低了其实现的复杂性，提高了效率。

(4) 重发处理

重发处理是 TCP 协议保证可靠性传输的重要机制。它分为快速重发和超时重发两种。快速重发是指当通信的一方连续收到 3 次相同的确认序号，立即将还没有被确认的

数据重发出去。因此，快速重发与 TCP 数据序号管理紧密相关。对于资源受限的嵌入式应用系统，进行序号管理不容易，且快速重发机制并不是必不可少的。所以，在嵌入式 TCP/IP 协议的实现中，将快速重发作为可选功能。为保证传输的正确性，超时重发必不可少。每次发送 TCP 报文段后，就启动等待确认定时器，如果定时器超时，就将还没有确认的数据重发[44]。

(5) TCB 的简化

TCB(TCP transmission control block)是指 TCP 传输控制块。它是一个数据结构，TCP 通过该数据结构来为每个 TCP 连接协调发送、接收和重发动作，TCP 为每个活跃的连接保持了一个 TCB。TCB 中包含了有关连接的所有信息，标准的 TCP 协议实现中 TCB 较大，而本课题对 TCP 协议的 TCB 进行了简化，只保留了用于控制面向连接的数据收发所必需的基本信息。

4.5.2.2 TCP 协议的实现模型

TCP 协议为两个任意处理速率、使用不可靠 IP 链接机制的机器之间的通信提供可靠的、具有流量控制的、端到端的数据流服务。TCP 协议实现模型包括的模块可以划分为：输入处理模块、输出处理模块、有限状态机处理模块和定时模块。具体如图 4.9 所示。

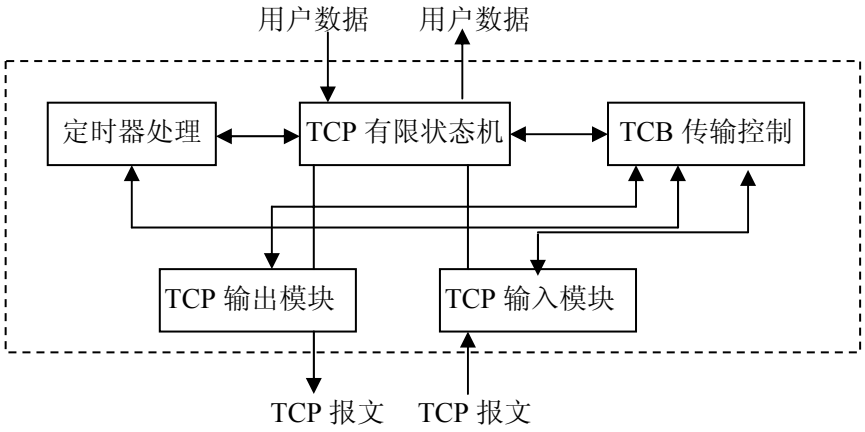

图 4.9 TCP 协议实现模型

Fig. 4.9 Realization model of TCP protocol

输入模块接收数据后，从 TCP 首部提取信息存入 TCB 传输控制快，然后将 TCP 首部交给 TCP 有限状态机模块处理，根据 TCB 内容进行有限状态机处理，更新 TCB 内容。最后将 TCP 数据交给应用程序或是进入输出处理模块发送 TCP 报文段，这完全取决于 TCP 有限状态机处理结果。输入模块还需要对相关定时器进行刷新处理。

当 TCP 建立后，应用程序控制 TCP 有限状态机处理模块处理，即关闭 TCP 连接还是发送数据，然后由输出处理模块形成 TCP 报文段提交给 IP 层处理。此时，重发超时定时器开始工作，如果等待确认超时则启动重发过程。

4.5.2.3 TCP 协议的主要数据结构

(1) TCP 状态

```
enum tcp_state{
    CLOSED, LISTEN, SYN_SENT, SYN_RCVD, ESTABLISHED, FIN_
    WAIT_1,FIN_WAIT_2, CLOSE_WAIT, CLOSING, LAST_ACK, TIME_ WAIT
};
```

(2) TCP 头部结构

```
struct tcp_hdr{
   u16_t   src; /*TCP 源端口*/
   u16_t   dest; /*TCP 目的端口*/
   u32_t   seqno; /*序号*/
   u32_t   ackno; /*确认序号*/
   u16_t   hdrlen_rsvd_ flags; /*头部长度及状态标志*/
   u16_t   wnd; /*窗口大小*/
   u16_t   chksum; /*校验和*/
   u16_t   urge; /*紧急指针*/
   };
```

(3) TCP 控制字

TCP 通过传输控制块(TCB)这个数据结构类来维护和管理 TCP 数据传输的细节。TCP 通过 TCB 为每个 TCP 连接协调传送、接收和重发动作，TCB 为所有进程所共享。TCP 为每个活跃的连接保留一个独立的 TCB。所有的 TCP 协议处理与 TCB 有关，它是实现 TCP 的关键。由于 RFC 文档中没有对 TCB 做强制性的要求，所以它的定义和处理非常灵活。

```
struct tcp_pcb{
struct tcp_pcb *next; /*指向下一个 tcp_pcb 的指针*/
enum tcp_state state; /* TCP 的状态*/
zsocket_t *_psocket; /*TCP_PCB 对应的 socket*/
sys_sem_t user_sem; /* API 层使用，用与用户的事件通知*/
u8_t user_state; /* API 层函数使用，可以控制使用查询还是阻塞方式*/
s8_t accept; /*TCP Server 使用的变量*/
struct tcp_pcb *server;      /*Server TCP 所使用*/
u8_t flags; /*状态信息*/
u32_t rcv_nxt; /*下一个希望收到的序号*/
```

```
u16_t rcv_wnd; /*接收窗口*/
u16_t rtime; /*快速重传时钟 */
u16_t mss; /*最大报文段长度*/
u16_t rttest; /*估计的往返时间，以 500ms 为单位 */
u32_t rtseq; /* 开始计时点*/
u16_t rto; /*重传超时*/
u8_t nrtx; /* 重传次数*/
u32_t lastack; /*最后确认的序号*/
u32_t snd_nxt; /*下一个要发送的序号*/
u32_t snd_max; /*已发送最大序号*/
u16_t acked; /*已确认*/
u16_t snd_buf; /*发送缓冲 */
u8_t snd_queuelen; /*发送队列长度 */
struct tcp_seg *unsent; /*未发送段链*/
struct tcp_seg *unacked: /*已发送但未被确认段链*/
struct tcp_seg *ooseq; /*收到失序的段链*/
zbuffer_t *recv_pbuf; /*接收数据缓冲区*/
}tcp_pcb_t;
```

(4) TCP 发送分段段结构

```
struct tcp_seg{
struct tcp_seg *next; /* 指向下一个段结构*/
zbuffer_t *pbuffer; /*包含了 TCP 头部和发送数据的缓冲区 */
struct tcp_hdr *tcphdr; /* TCP 头部指针*/
void *dataptr; /* 实际数据指针*/
u16_t len; /*分段长度*/
u8_t retry_time; /*超时重传次数 */
}tcp_seg_t;
```

(5) 超时结构

```
typedef void (*tcp_timeout_handler)(void *arg);
struct tcp_timeout{
   struct tcp_timeout *next; /*指向下一个超时结构*/
   u32_t time; /*超时时间*/
   tcp_timeout_ handler h; /*定时时间到后执行的回调函数*/
```

```
    void *arg; /*执行回调函数的参数*/
};
```

(6) 超时数组

```
struct tcp_timeout tcp_timeouts[MAX_TCP_TASK];
```

其中 MAX_TCP_TASK 为最大 TCP 连接数量，用户可根据实际情况进行修改，MAX_TCP_TASK 默认值为 5。

4.5.2.4 TCP 协议的主要功能实现

(1) TCP 输入过程实现

TCP 输入过程都在 TCP 输入函数中处理，处理过程比较复杂，因为 TCP 的连接按照 TCP/IP 协议定义了 11 种状态，针对每种状态收到不同的数据会产生不同的输出，并可能产生状态变迁，为此设计了 TCP 状态处理过程用以对 TCP 有限状态机进行处理。处理后如果需要给应用程序的数据就通过 TCP 接收函数继续向应用程序传递。

TCP 输入函数定义为 void tcp_input(znetif_t *pnetif, zbuffer_t *pbuffer)。输入函数的功能就是把收到的数据进行检查并送到相应的接收任务中，它首先检查校验和和长度是否正确；解析数据报，得出源和目的的 IP 地址及端口号，如果目的地址是广播地址则直接丢弃；然后遍历 tcp_pcb，找到对应的 TCP 连接，若找到则交给 tcp_process 过程进一步处理，若找不到匹配 TCP 连接，则给发送者发送 RST 消息。

TCP 状态处理过程函数定义为 static u8_t tcp_process(struct tcp_pcb *pcb)。该过程根据目前 TCP 连接所处的不同状态及收到的报文确定其下一步操作。TCP 状态变迁图如图 4.10 所示。在图中，椭圆表示连接状态，箭头表示状态的迁移。箭头旁的标签标识发生该迁移时收到的数据段以及对数据段做出的反应。

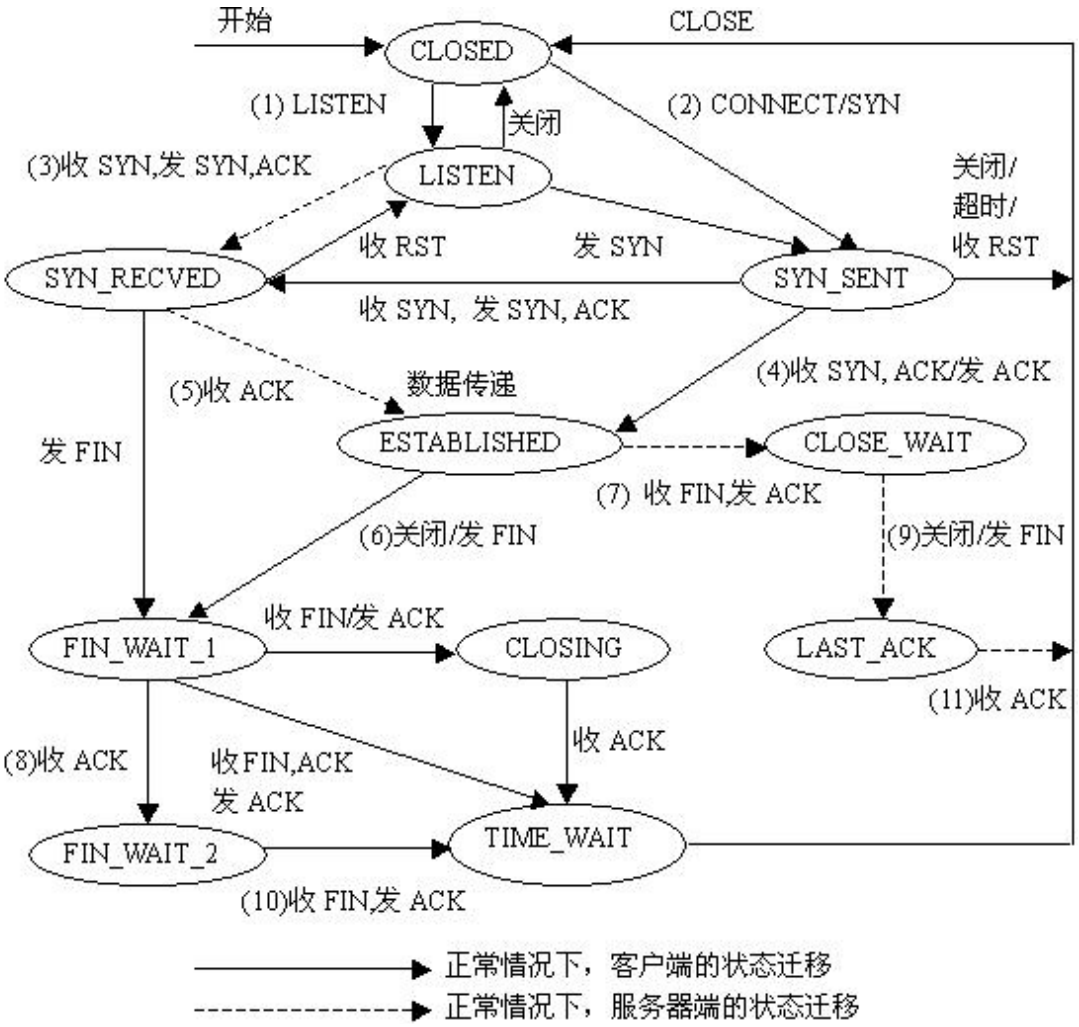

图 4.10 TCP 有限状态机

Fig. 4.10 TCP limited status machine

对每一个状态，某些事件是合法的。当一个合法的事件发生后，可能引起某种动作和状态的变迁。每个连接均开始于 CLOSED 状态。当一方执行了被动的连接原语 LISTEN 或主动的连接原语 CONNECT 时，它便会脱离 CLOSED 状态。如果此时另一方执行了相对应的原语，连接便建立了，并且状态变为 ESTABLISHED 此时双方就可以进行数据通信了。任何一方可以请求释放连接，当连接释放时，状态又回到 CLOSED 状态[45]。

不论是客户端还是服务器端，他们都要为通信创建一个端点，都具有一份有限状态机的拷贝，正常的 TCP 连接与关闭可以分为以下几个步骤：

(1) 服务器执行 LISTEN 原语首先发起一个被动打开的操作，这将导致服务器的有限状态进入 LISTEN 状态，该服务器在 LISTEN 状态中等待客户端连接；

(2) 当客户端的一个应用程序发起主动打开 CONNECT 请求后，导致该机器上的 TCP 软件为其创建 TCB 并发送一个 SYN 报文段给服务器，然后进 SYN_SENT 状态；

(3) 当正在 LISTEN 状态中等待的服务器收到 SYN 报文段后，它以一个 SYN 和一

个 ACK 报文段作为应答，并创建一个新的 TCB，将新的 TCB 置于 SYN_RECVED 状态；

(4) 客户端收到 SYN 加上 ACK 报文段，该机器上的 TCP 软件给服务器发出三次握手的最后一个 ACK 报文段，并且转为 ESTABLISHED 状态；

(5) 当客户端的 ACK 报文段到达服务器新创建的 TCB 后，该该服务器的 SYN 得到确认，服务器端 TCP 进入 ESTABLISHED 状态，三次握手完成。此时，就可以进行数据通信了。当一个应用程序完成数据收发任务后，它要关闭 TCP 连接(假设仍由客户端发起主动关闭连接)；

(6) 客户端执行 CLOSE 原语，TCH 发送一个 FIN 报文段并等待响应的确认(进入状态 FIN_WAIT_1)；

(7) 服务器收到一个 FIN 报文段，它确认客户端的请求发回一个 ACK 报文段，进入 CLOSE_WAIT 状态；

(8) 客户端收到确认 ACK 报文段，就转移到 FIN_WAIT_2 状态，此时连接在一个方向就断开了；

(9) 服务器端应用得到通告后，也执行 CLOSE 原语关闭另一个方向上的接连，该机器上的 TCP 软件向客户端发送一个 FIN 报文段，并进入 LAST_ACK 状态，等待最后一个 ACK 确认报文段；

(10) 客户端收到 FIN 报文段并确认，进入 TIME_WAIT 状态，现在双方均已断开连接，但 TCP 要等待一个最大分组生命周期，确保该连接的所有分组全部消失，以防止出现确认丢失的情况，当定时器超时后，TCP 删除该 TCB，返回到初始状态(CLOSED)；

(11) 服务器收到最后一个确认 ACK 报文段，TCP 删除该 TCB，返回到初始状态(CLOSED)。

TCP 接收函数定义为 void tcp_recv(zsocket_t *psocket, zbuffer_t **ppbuffer, u8_t flags)。该函数主要是从对应的 TCP_PCB 中维护的接收数据缓冲区中取数据，不过取数据的时候也要注意使用原子操作。

(2) TCP 输出过程实现

对于应用程序来说，是用 TCP 发送过程来发送数据的。实际上这个数据报是先经过 TCP 分段处理后进入发送队列，然后才通过 TCP 输出过程发出去。所以发送过程实际上是由这三部分组成。

TCP 发送过程函数定义为 u8_t tcp_write(znetif_t *pnetif, ipaddr_t *pdest_ip, ipaddr_t *psrc_ip,zbuffer_t *pbuffer)。发送过程是由应用程序调用，所以这里需要对 pcb 的状态进行检查。应用程序只可能在 TCP 处在 SYN_SENT, SYN_RCVD, ESTABLISHED 或

CLOSE_WAIT 状态时发数据，其他状态只能是 TCP 协议内部才会使用，对应用程序是透明的。如果状态正确，则调用 tcp_enqueue 过程进行分段。这个过程并不是真正把数据发出去，如果需要，可以在调用 tcp_write 之后再调用 tcp_output( )把数据实际发出。

TCP 分段过程函数定义为 u8_t tcp_enqueue(struct tcp_pcb *pcb, void *arg,u16_t len, u8_t flags，u8_t *optdata, u8_t opt_len)。该函数先检查带发送队列中最后一项是否达到最大传输长度 MSS 限制，若没有则用新数据先填满；然后把要发送数据按 MSS 大小分块，每一块都以 tcp_seg 结构添加到待发送队列，并为段初始化 TCP 头部和其他必要信息，如段序号；如果有选项信息 optdata，把这些信息加到 TCP 头部中。

TCP 输出函数定义为 u8_t tcp_output(struct tcp_pcb *pcb)。输出过程主要就是从待发送队列中取出第一个 TCP 段，通过 ip_out 发送出去，并把该段添加到待应答队列尾部。

4.6 SOCKET API 接口实现

由于用户程序一般不直接使用 ARP、ICMP 协议，因此本课题将这两个模块在协议栈主任务中直接实现了，并没有提供给用户交互的 API。而 UDP、TCP 是开放给用户使用的，即是提供给 μC/OS-II 其他任务以实现网络应用的接口，所以本课题仿照一般 OS 些 Socket 编程的结构，提供一个比较简单的 API 层[46]。

4.6.1 API 层定义的主要数据结构

```
typedef enum{
   data_packet;  /*UDP 程序使用*/
   data_stream;  /*TCP 程序使用*/
}protocol_t;
typedef struct zsocket{
   s8_t    id;  /*该 socket 的标识符*/
   znetif *pnetif;  /*  网络设备*/
   task_t task_id;  /*  使用该 socket 的任务标号*/
   protocol_t proto;  /*  协议类型（UDP or TCP）*/
   u16_t l_port;  /*  本地所使用的端口*/
   ip_addr l_ipaddr;  /*本地所使用的 IP 地址*/
   u16_t r_port;  /*TCP 连接的话，必须指定远端的端口*/
   id_addr r_ipaddr;  /*  TCP 连接对应的远端 IP 地址*/
   u8_t error;  /*API 系统调用错误返回号*/
}zsocket_t;
```

4.6.2 API 主要功能函数的实现

4.6.2.1 提供给协议栈用户使用的 API 的实现

(1) void zsocket_init(void)，主要实现用户连接信息初始化。

(2) s8_t zsocket(protocol_t type)，主要是用来建立一个 zsocket，具体是什么类型的 socket，依据参数 type 来判断。

(3) u8_t zbind(s8_t sid, ip_addr *localip, u16_t *localport)，主要实现绑定本地 IP 和端口的功能。

(4) u8_t zclose(s8_t sid)，主要是用来关闭该 socket 对应的连接。

(5) u8_t zshutdown(s8_t sid)，主要是释放 socket。

(6) u8_t zioctl(s8_t sid, u8_t request, u8_t *argp)，主要实现在 socket 上通过 io 控制的设置操作。

(7) zsocket_t *query_zsocket(s8_t sid)，主要是用来返回对应 socket 的信息。

4.6.2.2 TCP 对应的 API 实现

(1) u8_t zsent(s8_t sid, u8_t *pdata, u16_t *data_len, u8_t flags)，主要实现 TCP 连接数据的发送操作，最后是调用 tcp_sent( )进行数据发送。

(2) u8_t zrecv(s8_t sid, zbuf **pbuf, u8_t flags)，主要实现从 TCP 远端接收数据，最后是调用 tcp_recv( )进行数据接收。

(3) u8_t zlisten(s8_t sid)，主要是用来侦听是否有 tcp 连接，调用 tcp_listen( )返回侦听结果。

(4) s8_t zaccept(s8_t sid)，主要实现接受新的连接，调用 tcp_accept( )返回远端 socket 的 id。

4.6.2.3 UDP 对应的 API 实现

(1) u8_t zrecvfrom(s8_t sid, zbuf **pbuf,ip_addr *rip, u16_t *rport, u8_t flags)，主要是用来实现 UDP 数据包的接收，最后是调用 udp_recv( )来进行包的接收。

(2) u8_t zsendto(s8_t sid, u8_t *pdata, u16_t *len,ip_addr *rip, u16_t *rport)，主要是用来实现 UDP 数据包的发送，最后是调用 udp_send( )来进行包的发送。

4.7 嵌入式 TCP/IP 协议通用接口层的实现

在实现的嵌入式应用中，无论是操作系统还是嵌入式硬件设备都具有很强的独立性，如果我们实现的嵌入式 TCP/IP 协议不能移植到需要同样功能的其它系统中，就减少了代码的重用，从而提高了开发成本。因此，嵌入式 TCP/IP 协议不仅要实现完善的功能，而且还要考虑使其具有较好的可移植性，这样的开发才更有意义。

移植性的问题主要是由于底层的硬件或操作系统不兼容而引起的，要使上层的系统

具有可移植性，就应该设法屏蔽底层的不同。为了使本课题设计的嵌入式 TCP/IP 协议具有良好的移植性，在协议栈中重新定义了数据类型并包装了 μC/OS-II 的接口，协议中使用了以下几个 μC/OS-II 接口[47]。

```
sys_time_t sys_get_time(void); /*查询当前时间*/
sys_sem_t sys_new_sem(u8_t value); /*发送和接收事件*/
u8_t sys_signal_sem(sys_sem_t sem); /*信号量 signal 操作*/
void sys_wait_sem(sys_sem_t sem, u16_t timeout, u8_t *err);/*信号量 wait 操作*/
void sys_reset_sem(sys_sem_t sem, u8_t value); /*  信号量归零操作*/
u8_t sys_current_task(void); /*查询当前运行的任务 ID */
void sys_delay(u16_t dtime); /*系统定时睡眠*/
#define sys_sleep(tid)    OSTaskSuspend((tid)) /*使某个任务进入休眠状态*/
#define sys_wackup(tid)  OSTaskResume((tid)) /*唤醒某个任务*/
```

由于本课题使用的 RTOS 是 μC/OS-II，所以仅给出本课题的具体实现方式。如果需要移植到其他 OS 上去，只要给出这些函数的实现就可以。

(1) 采用 μC/OS-II 中提供的 OSTimeGet( )得到全局的时钟变量。

```
sys_time_t sys_get_time()
{ return OSTimeGet(); }
```

(2) 采用 μC/OS-II 中提供的 OSSemCreate(value)创建一个信号量。

```
sys_sem_t      sys_new_sem(u8_t value)
{ return OSSemCreate(value);}
```

(3) 采用 μC/OS-II 中提供的 OSSemPend(sem,timeout,err)把信号量减 1。

```
void sys_wait_sem(sys_sem_t sem, u16_t timeout, u8_t *err)
{    OSSemPend(sem, timeout, err);
     if ( *err == OS_NO_ERR)
     {*err = 0;}
}
```

(4) 采用 μC/OS-II 中提供的 OSSemPost(sem,timeout,err)把信号量加 1。

```
u8_t sys_signal_sem(sys_sem_t sem)
{if ( OSSemPost(sem) == OS_NO_ERR )
          return 0;
     else
          return -1;
}
```

(5) 直接赋值给 μC/OS-II 中的 OSEventCnt 即信号量计数器。

```
void sys_reset_sem(sys_sem_t sem, u8_t value)
{   sem->OSEventCnt = value;
    return 0;
}
```

(6) 通过 OSTCBCur->OSTCBPrio 得到运行任务的 ID，即当前任务的优先级别。

```
u8_t sys_current_task(void)
{   task_t    task;
    sys_enter_critical();
    task = OSTCBCur->OSTCBPrio;
    sys_exit_critical();
    return task;
}
```

(7) 通过 OSTimeDly(dtime)得到定时的睡眠实现。

```
void sys_delay(u16_t dtime)
{OSTimeDly(dtime);}
```

第五章 嵌入式 TCP/IP 协议栈测试

5.1 测试环境

由于时间和条件所限，协议栈的测试用了两台 PC，一台是试验机 1，一台是试验机 2，试验机 2 用来充当网关连接局域网和 Internet。一块 S3C44B0X 的 ARM7 开发板，网卡为 RTL8019AS，开发板作为目标机并在其 FLASH 中固化了本课题实现的 μC/OS-II 及其协议栈。这两台 PC 和 ARM 开发板都是通过交换机连接起来的。

5.2 嵌入式协议的功能测试

5.2.1 IP 层及其以下模块的测试

IP 层及其以下的模块有 ARP、IP、ICMP 模块，ICMP 协议处于 IP 模块的上层，可以通过发送 ICMP 回送请求和接收来自其他机器的 ICMP 回送应答来测试各个模块和接收来自其他机器的 ICMP 回送应答来测试各个模块的功能。如果 ICMP 模块测试成功，则说明 IP 模块的收发功能以及 ARP 模块的地址绑定功能也是正确的。

测试方法：在目标机上运行 ping 命令，目的地址为实验机的 IP 地址。此时，目标机将发送 ARP 请求包获得实验机的 MAC 地址，然后发送 ping 包。

测试结果：在目标机一方的终端显示窗口中看到收到 ICMP 请求应答报文的提示信息，包括该报文的字节数、响应时间等。这些信息表明已经正确收到 ICMP 请求应答报文，测试结果如图 5.1 所示。

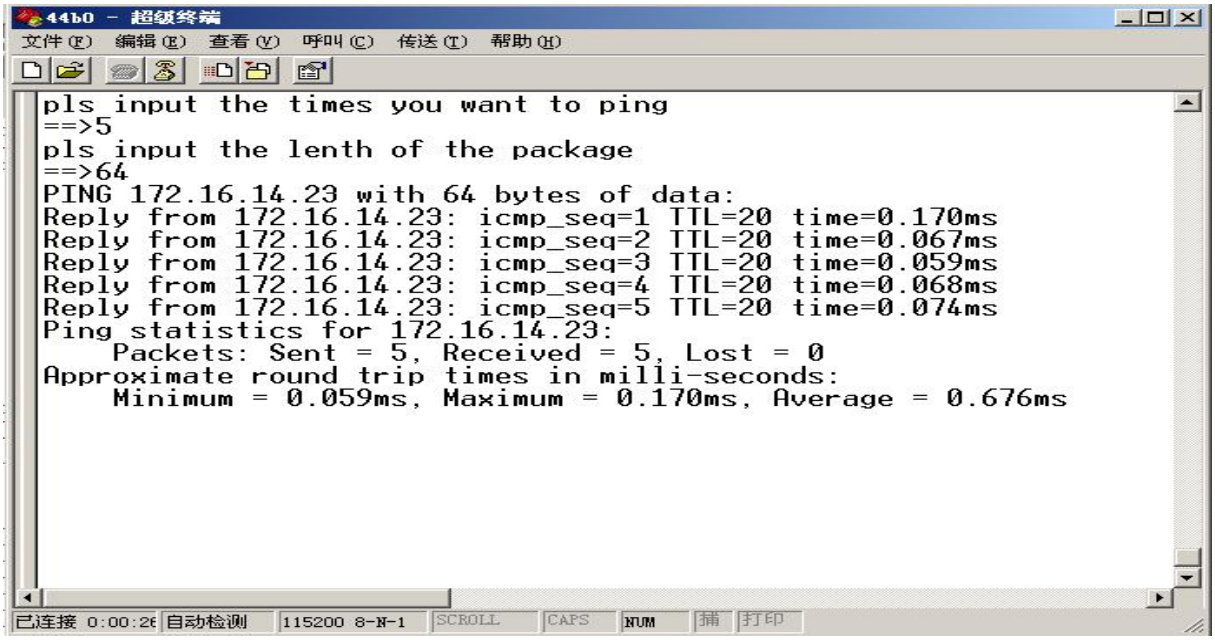

图 5.1 ICMP 协议测试结果

Fig. 5.1 The result for testing ICMP protocol

5.2.2 IP 层以上模块的测试

5.2.2.1 TCP 模块的测试

在目标机中使用协议栈中 API 实现一个只提供简单网页传输服务的简单 Web Server，在试验机上通过浏览器，可以访问运行于 S3C44B0X 硬件平台上简单的 Web Server 提供的网页内容，实验显示结果如图 5.2 所示。

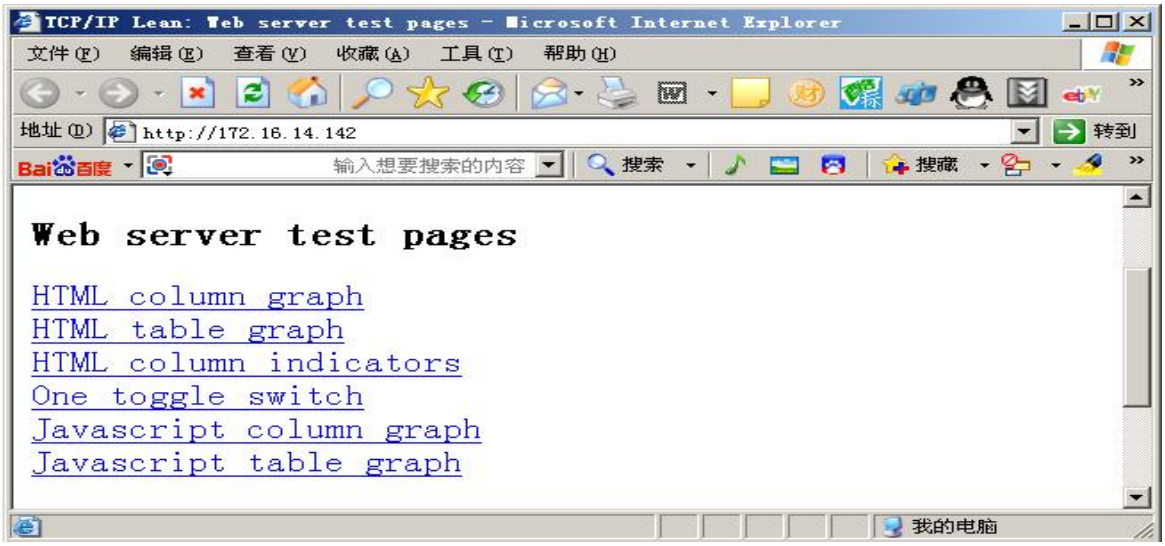

图 5.2 访问目标机上 web server 结果显示

Fig. 5.2 The result view for visiting regart machine web server

5.2.2.2 UDP 模块的测试

在目标机上实现一个基于 UDP 协议的 TFTP 测试用例，通过试验机往目标机传送一个文件，传送完后在终端显示结果如图 5.3 所示。

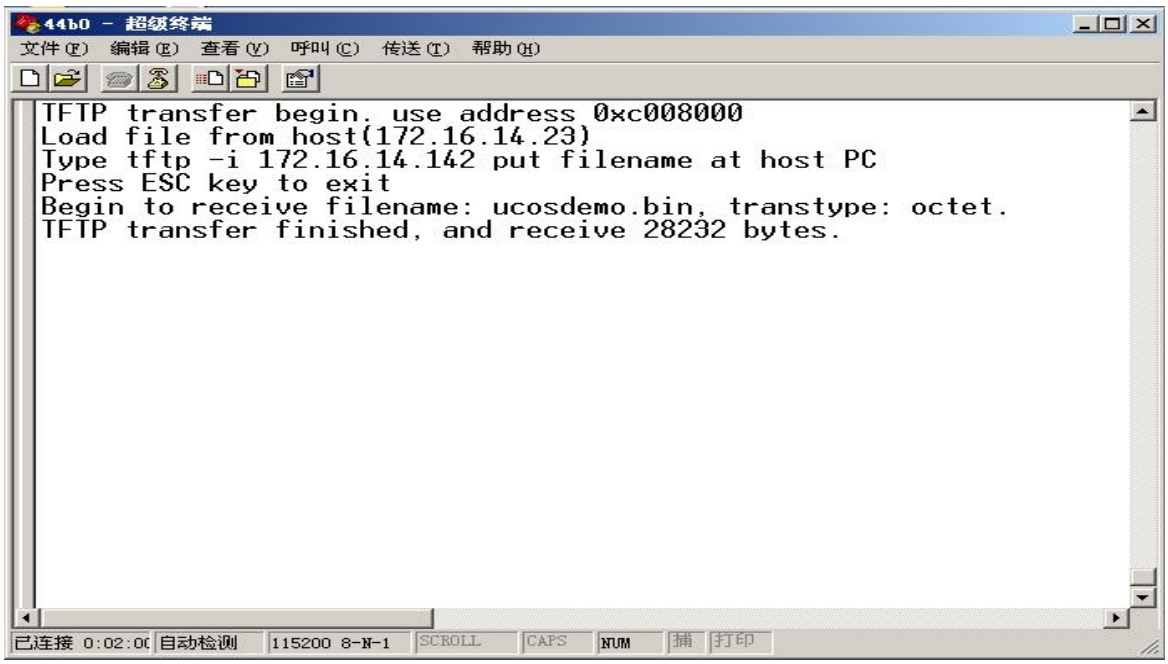

图 5.3 TFTP 协议测试结果

Fig. 5.3 The result for testing TFTP protocol

5.3 嵌入式协议的性能测试

在嵌入式 TCP/IP 协议的性能测试中，我们选择了 TCP 协议进行测试。从 TCP 建立连接的响应时间、TCP 关闭连接的响应时间以及 TCP 协议的数据传输率三方面进行了简单测试，通过与标准的 TCP/IP 协议进行比较，总结了其在性能方面的优势与不足。

5.3.1 TCP 建立连接的响应时间

以实验机 1 为 TCP 的客户端，分别向目标机和实验机 2 发出连接请求，计算从 TCP 客户端发出 TCP 连接请求到连接建立之间的时间。该值可以通过在连接建立前后各加一条语句得到当前时间，然后相减得到。实验数据见表 5.1。

表 5.1 TCP 建立连接的响应时间
Table 5.1 Response time of TCP establish connection

TCP 建立连接响应的实现(单位：ms)								
执行次数	1	2	3	…	18	19	20	平均时间
嵌入式 TCP/IP	4.2	5.6	7.0	…	6.5	5.3	7.6	5.0
标准 TCP/IP	11.5	9.2	13.7	…	8.5	9.8	10.2	10.8

5.3.2 TCP 关闭连接的响应时间

以实验机 1 为 TCP 的客户端，分别向目标机和实验机 2 发出关闭连接请求，计算从发出关闭连接请求到连接断开的时间。该值可以通过在关闭前后各加一条语句得到当前的时间，然后相减可以得到。实验数据见表 5.2。

表 5.2 TCP 关闭连接的响应时间
Table 5.2 Response time of TCP close connection

TCP 关闭连接响应的实现(单位：ms)								
执行次数	1	2	3	…	18	19	20	平均时间
嵌入式 TCP/IP	12.2	12.6	7.0	…	9.5	13.3	8.8	10.0
标准 TCP/IP	10.05	12.2	16.7	…	16.0	17.8	15.2	15.0

5.3.3 TCP 协议的数据传输率

以实验机 1 为客户机，分别与目标机、实验机 2 建立连接，在连接建立成功后，发送一个数据包，直到接收方给该段的确认之间的时间，在此时间内所传输的数据量除以此时间.测试的结果如图 5.4 所示，从图中可以看出，标准 TCP/IP 协议的数据传输率约为 380Kbps，而轻量级 TCP/IP 协议的数据传输率约为 365Kbps。

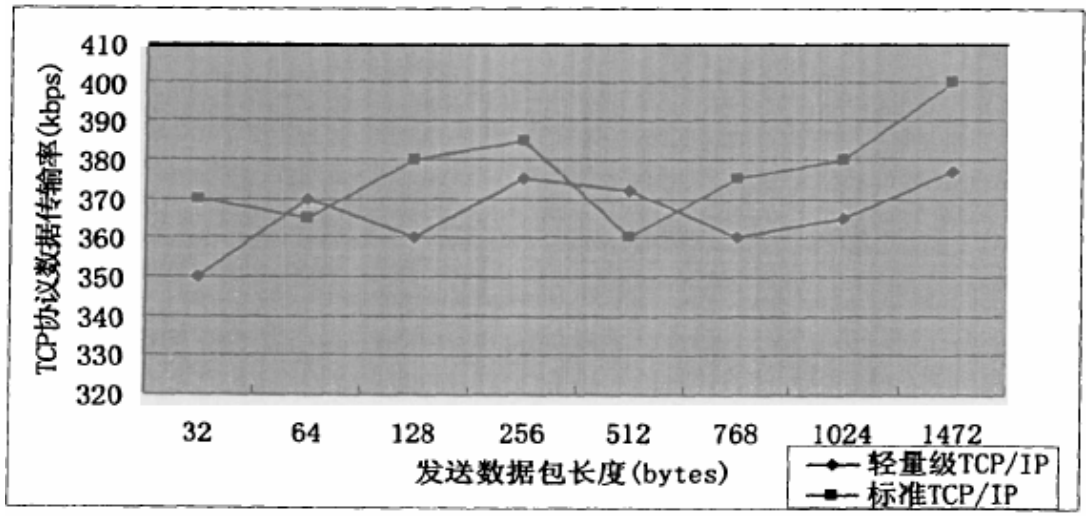

图 5.4 TCP 协议的数据传输率
Fig. 5.4 Data transmission rate of TCP protocol

5.4 测试小结

本章通过一系列实验对本课题实现的嵌入式 TCP/IP 的功能和性能进行了测试，实验结果表明，本课题所实现的嵌入式 TCP/IP 协议完成了系统设计中提出的功能需求，提供了一个较全面的嵌入式系统的通信协议栈。它的目标代码量要远远小于标准的 TCP/IP 协议。在性能方面，嵌入式 TCP 协议在建立 TCP 连接和关闭 TCP 连接上的耗时比标准 TCP 协议要短，这主要是嵌入式 TCP 协议只实现了标准 TCP 协议的部分功能，而且在实现机制上做了改进。可以在更短的时间内建立和关闭 TCP 连接。但是，在数据传输率上，有待进一部提高。

同时，测试的过程中也发现了不少问题，协议栈有待于进一步完善。

第六章 结论与展望

6.1 结论

随着 Internet 的飞速发展和信息家电及后 PC 概念的提出，使得嵌入式设备网络化成为不可阻挡的趋势。在实现嵌入式系统接入 Internet 技术中，如何构建一个稳定、高效、安全、可移植和可裁减的 TCP/IP 协议栈是其中最为关键的的组成部分。正鉴于此，本课题以实现一个稳定、高效、易移植和易裁减的轻量级嵌入式 TCP/IP 协议栈为研究方向和目标，制定了课题的基本解决方案并实现。现在将本课题研究工作的重点、成果和创新点描述如下：

(1) 在研究分析了标准的 TCP/IP 协议基础上提出了轻量级嵌入式 TCP/IP 协议的简化原则，其简化原则是在保证基本的通信要求和协议基本功能的基础上，减少协议数量，简化一些协议功能以满足嵌入式系统网络化的基本通信需求。

(2) 依据嵌入式 TCP/IP 协议的简化原则，选取了 IP、ICMP、ARP、UDP 和 TCP 协议并在不违反标准 TCP/IP 协议的前提下，对它们进行重新设计并实现；采用中断方式实现 RTL8019AS 芯片驱动，相比查询方式的驱动可以更好地与实时操作系统 μC/OS-II 相结合，更好地体现出实时操作系统的实时性。

(3) 在 ARM7 开发板上移植 μC/OS-II，把本课题实现的轻量级嵌入式 TCP/IP 协议栈整合到 μC/OS-II 中，并设计了一个良好的通用接口层和配置项，以使协议栈能很好地整合到其他 OS 中。为了适应 μC/OS-II 这样的实时操作系统，协议栈整体作为任务的形式实现，使用基于中断驱动的模式，API 层可以适应多个网络程序同时运行；缓冲区管理的实现参考 Lwip 缓冲管理机制并对其进行改进，采用全局管理方法优化协议栈的缓冲区管理；同时采用“一次拷贝”技术，大大减少了系统资源开销。

6.2 展望

嵌入式系统和 Internet 的接合有着广泛的应用前景，目前已经引起了国内外计算机通信控制领域的关注。但作为一项新兴技术，仍然有大量应用课题需要研究。本课题虽然取得了一定的研究成果，但是还是存在一定的不足，首先就是 TCP 协议的稳定性和完备性还欠缺，API 接口层实现简单化；其次就是没有太多考虑协议通信过程中的安全性；再次就是协议栈的缓存区管理技术还有待进一步提供其性能；最后就是协议栈对异型网络的支持也没有考虑，这个方向将是本课题以后工作的首要方向。

参考文献

1. 王学龙. 嵌入式 Linux 系统设计与应用[M]，北京：清华大学出版社，2001,1-2
2. 探矽工作室. 嵌入式系统开发圣经（第二版）[M]，北京：中国铁道出版社,2003,1-3
3. 李阳明，齐志强，师丽彩. 嵌入式 Internet 应用研究[J]，电子工程师，2005,31(7):59-60
4. 张赘慧，陈泉林. 源码公开的 TCP/IP 协议栈在远程监测中的应用[J]，单片机与嵌入式系统应用，2004,5(11):61-64
5. 李素侠，段友祥. 嵌入式 TCP/IP 协议的分析与应用[J]，微计算机信息，2005,21(7):52-54
6. 李仁发，周祖德，李方敏等. 虚拟实验室网络体系结构研究[J]，系统仿真学报，2002,14(3):359-362
7. 赵海. 嵌入式 Internet—21 世纪的一场信息技术革命[M]，北京：清华大学出版社，2001,1-27
8. 吴明晖. 基于 ARM 的嵌入式系统开发与应用[M]，北京：人民邮电出版社，2004,143-145
9. Filman Robert E. Embedded Internet Syetems Come Home[J], IEEE Internet Computing, 2001, 5(1): 52-53
10. 韩光洁. Embedded Internet 技术及其综述[J]，小型微型计算机系统，2004,25(5):1-4
11. 袁兵，付宏，吴鸿韬. 嵌入式 Internet 与扩展 Internet 研究[J]，北京邮电大学学报，2003,26(增刊):126-129
12. Wilson A. The Challenge of Embedded Internet[J], Electronic Product Design, 1998, January: 31-34
13. Clarke Eslcr, Christopher S.Songtag. Embedded Internet Connectivity for 8-and16-bit Microcontrollers[J], Infineon Technologies Development Tools Partners Magazine, 1999, 2(5): 33-47
14. 李明. 嵌入式互联网络接口的设计与开发[J]，工业控制计算机，2002,15(8):39-42
15. 叶朝辉. 智能家庭网络研究综述[J]，计算机应用研究，2001,18(9):1-6
16. 彭少熙. 家庭网络中的嵌入式 Internet 方案[J]，电子技术应用，2001,10(10):47-50
17. Rihijarvi J, Mahonen P, Saaranen M.J. Providing Network Connectivity for small Appliances: A unctionally minimized Embedded Web Server[J], IEEE Communicanons Magazine, 2001, 39(10): 74-79
18. 刘汉伦，方华京. 嵌入式系统的 Internet 接入实现[J]，工业控制计算机，

2003,16(12):21-23

19. 金欢，阮冠春，徐凌宇，赵海. 基于嵌入式 Internet 技术的 Webit 体系结构研究与实现[J]，控制与决策，2002,17(5):541-549
20. 王勇，陈抗生. 嵌入式 Internet 中的协议选择[J]，电信科学，2002,15(4):1-4
21. 史治国. 嵌入式 Internet 中 TCP 协议的实现[J]，计算机工程与应用，2001,8(6):1-4
22. Jean J.Labrosse 著，邵贝贝译. 嵌入式实时操作系统 μC/OS-II（第二版）[M]，北京：北京航空航天大学出版社，2003
23. 杨哗. 实时操作系统的 μC/OS-II 下 TCP/IP 协议栈的实现[J]，单片机与嵌入式系统应用，2003,7(3):20-23
24. Adam Dunkels. Design and Implementation of the LWIP TCP/IP Stack[S], Lwip source code, 2001
25. W. Richard Stevens 著，范建华等译. TCP/IP 详解（卷 1：协议）[M]，北京：机械工业出版社，2000
26. REALTEK SEMICONDUCTOR CORP. RTL8019AS Realtek Full-Duplex Ethernet Controller with Plug and Play Function (RealPNP) specification[S], 2001
27. J.Bentham 著，陈向群 译. 嵌入式系统 web 服务器 TCP/IP Lean（第二版）[M]，北京：机械工业出版社，2003:10-15
28. ARM7TDMI Technical Reference Manual[S], 2001
29. Agranat lan Douglas. Engineering Web Technologies for Embedded Applications[J], IEEE Internet Computing, 1998, 2(3): 40-45
30. Wood Mike, Tom Barrett. Embedded System Programming[J], A Real-Time Primer, 1990, 3(4), 20-28
31. Josef Fleischmann, Klaus Buchenrieder. Prototyping Networked Embedded Systems[J], Computer, 1999, 6(2): 116-119
32. Marina del Rey. Internet Protocol[S], RFC791, September 1981
33. Babak A S. Stability of Networked Control Systems in the presence of Packet Losses[A], Proceedings of 2003 IEEE Conference on Decision and Control[C], December 2003, 115-118
34. Jon C. Snader. Effective TCP/IP Programming[J], Addison-Wesley Professional, 2000, 1(2): 27-32
35. J. Postel. Internet Control Message Protocol[S], RFC792, September 1981
36. David C. Plummer. An Ethernet Address Resolution Protocol[S], November 1982
37. 陈武，雷航. 基于精简 TCP/IP 协议栈的信息家电网络服务器[J]，单片机与嵌入式系

统应用，2004,6(6):62-65

38. 郝洁，王慕坤等. 嵌入式 TCP/IP 技术研究及应用[J]，哈尔滨理工大学学报，2004,9(2):100-102
39. 王瑞朋，黄琼等. 嵌入式 TCP/IP 协议的内存管理技术[J]，微计算机信息，2006,22(3):69-72
40. J. Postel. User Datagram Protocol[S], RFC768, August 1980
41. Tong Xiaoyang. Transformer Online Fault Monitoring And Diagnosis Embedded System Based On TCP/IP And Pub/Sub New Technology[J], Institute of Electrical and Electronics Engineers, 2003, (4): 25-28
42. Marina del Rey. Transmission Control Protocol[S], RFC793, September 1981
43. James W H. An efficient and lightweight embedded web server for web-based network element management[J], International Journal of network management, 2000, (3): 261-275
44. 李立清，路海. 一种嵌入式 TCP/IP 的实现.信息与电子工程[J]，2003,1(4):293-296
45. 何宁等. 基于 TCP/IP 协议的嵌入式网络系统的设计与应用[J]，山东科技大学学报，2004, 23(1):55-57
46. 刘 鹏，张翔，戴国骏. 基于 μC/OS-II 的嵌入式 μC/IP 协议研究[J]，杭州电子工业学院学报，2005,24(1):60-63
47. Jae-Ho Lee. Implementing priorit inheritance semaphore on uC/OS-II real-time kernel Software Technologies for Future Embedded Systems[J], IEEE Internet Computing, 2003, 5(6): 15-16

致 谢

这篇论文能够顺利的完成，首先要感谢我的导师傅仲逑教授。

傅老师在我的学习期间始终关心我的学习和生活，帮助我克服了许多困难，给我提供了一个良好的科研和工程实验环境，使我得以全身心地投入课题研究并顺利完成学业。在毕业设计过程中，从选题、论证、开题、设计以及论文审核，傅老师都帮我准确地把握课题发展方向和进度、保证课题质量，并提出许多独到的建议。正是在傅老师的严格要求和悉心指导下，课题得以顺利完成。除了学业方面的指导，傅老师在人生道路的指引上也使我获益匪浅，傅老师严谨的治学态度，一丝不苟的工作作风，渊博的知识，广阔的视野和敏锐的洞察力无一不在潜移默化地影响着我，我将为此而受益终生。

其次我要感谢才书训老师，王翠荣老师，丁顺利老师，党群老师，刘杰民老师和研究生办公室的刘晓红老师，他们为我的课题提供了大量的宝贵意见和先进思想，开阔了我的技术视野。

我还要感谢我们实验室的付胜利、熊蜀峰、孙墉懋、陈强，还有韩礼国，杜春春，王娟和我的室友朱俊和章文，感谢他们在学习和生活中给我的帮助。

最后，感谢我的家人和朋友给我的关心和鼓励。他们不仅在生活上给我无微不至的关怀，在精神上也给我巨大的动力。他们是我身后最坚强的后盾，我的每一分进步都离不开他们的支持与鼓励，他们是我奋发向上的动力和幸福的源泉。

攻读硕士期间发表的论文

1 钟芳伟 熊蜀峰 傅仲逑，嵌入式 TCP/IP 协议栈 LWIP 的分析和研究，2006 年北京地区研究生学术交流会－信息与通信技术会议论文集，2006，12

2 付胜利 钟芳伟 傅仲逑，基于嵌入式 Linux 的 HTTP 代理服务器的研究，2006 年北京地区研究生学术交流会－信息与通信技术会议论文集，2006，12