电 子 科 技 大 学

UNIVERSITY OF ELECTRONIC SCIENCE AND TECHNOLOGY OF CHINA

专业学位硕士学位论文

MASTER THESIS FOR PROFESSIONAL DEGREE

论文题目 医院病房语音呼叫系统开发

专业学位类别 工程硕士

学 号 201122250413

作 者 姓 名 魏莱

指 导 教 师 徐军

分类号_____ 密级_____

UDC[注1]_____

学　位　论　文

医院病房语音呼叫系统开发

（题名和副题名）

魏莱

（作者姓名）

指导教师　徐军　教　授

电子科技大学　成　都

刘东　XX

XXXX　成　都

（姓名、职称、单位名称）

申请学位级别　硕士　专业学位类别　工程硕士

工程领域名称　软件工程

提交论文日期　2016.XX.XX　论文答辩日期　2016.XX.XX

学位授予单位和日期　电子科技大学　2016 年 XX 月 XX 日

答辩委员会主席_____

评阅人_____

注1：注明《国际十进分类法UDC》的类号。

Development of Hospital Voice Calling System

A Master Thesis Submitted to

University of Electronic Science and Technology of China

Major: **Software Engineering**

Author: **Wei Lai**

Advisor: **Professor Xu Jun**

School: **School of Physical Electronics**

独创性声明

本人声明所呈交的学位论文是本人在导师指导下进行的研究工作及取得的研究成果。据我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得电子科技大学或其它教育机构的学位或证书而使用过的材料。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谢意。

作者签名：_____ 日期：　　年　　月　　日

论文使用授权

本学位论文作者完全了解电子科技大学有关保留、使用学位论文的规定，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。本人授权电子科技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，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。

（保密的学位论文在解密后应遵守此规定）

作者签名：_____ 导师签名：_____

日期：　　年　　月　　日

摘 要

医院病房语音呼叫系统是医院现代化管理的重要工具和手段，对提高医疗质量、促进资源共享、提高医院竞争力等具有重要的意义。

现有的医院住院病房管理系统大多数采用的是本地化的管理需要人工的去病房巡视，观测病人状态，降低了工作的效率。随着人们对健康的日益关注，不断增加的医疗行为导致了医院现有系统性能逐渐下降，医院住院病房管理系统已经不能满足现在繁忙的业务系统。随着网络技术的发展人们越来越依靠网络技术给人们带来的便利，也是本系统设计的大背景。

本文首先分析了医院病房语言系统的发展现状，指出了先有呼叫系统存在的不足，并根据医院的特殊的业务需求提出一套改进的病房语音呼叫系统解决方案。本系统方案分为嵌入式系统和上位机软件系统两部分，嵌入式系统硬件以STM32为控制核心完成按钮响应、语音采集与播放、网络接入等功能；嵌入式系统软件基于RT-Thread嵌入式实时操作系统开发，负责嵌入式系统硬件的驱动和管理。将上位机软件系统按照功能划分了四个主要业务功能模块和三个管理功能模块，并介绍了各个功能模块相关的典型业务。然后采用面向对象的程序设计技术对系统需求进行了抽象，完成了上位机软件的架构设计，最后使用Visual C#集成开发环境完成了系统的设计和实现。最后，对系统的软件安装、网络部署、参数配置进行了阐述，并对系统功能完整性、正确性和稳定性进行了测试，测试的结果表明相对可以很好地满足医院的语音呼叫的需求，大大提升了医院病房管理的效率。

关键词：语音呼叫系统，嵌入式，上位机，数据库，TCP/IP

ABSTRACT

Hospital voice calling systems are important tools and means of hospital management, to improve medical quality to promote resource sharing and improving competitiveness is of great significance.

Existing management system is used in most hospital wards localized management need to be constructed to ward visits, observation patients, reducing the efficiency of work. With the growing health concerns, increasing medical practice leads to the decrease hospital system performance, management system can no longer meet in hospital wards are now busy business systems. With the development of network people increasingly rely on Internet technology brought convenience to people, but also the background of the system design.

This thesis analyses the development status of the language system of the hospital ward, noted that shortcomings of the calling system, and according to the specific business needs of the hospital ward of an improved voice calling system solutions. This programme is divided into two parts: embedded systems and PC software system, embedded system hardware to STM32 controlled Finish button response, audio capture and playback, Internet access and other features; RT-Thread based embedded real-time operating system for embedded systems software development, responsible for embedded system hardware drivers and management. PC software in accordance with the functions of the system can be divided into four major business function modules and three management modules, and describes the various functional modules of typical business. Using object-oriented programming techniques to abstract the system requirements, complete PC software architecture design, final integrated development environment using Visual C# complete system design and implementation. Finally, the system introduces the software installation, deployment, configuration, and functional integrity, accuracy and stability of the system was tested, test results show that the relative can meet Hospital demand for voice calls, greatly enhance the efficiency of management of hospital wards.

Keywords: Voice Calling System, Embedded System, Upper-side Machine, Database, TCP/IP

目 录

缩略词表

缩略词	英文全称	中文全称
ADC	Analog-to-Digital Converter	模数转换器
AGC	Automatic Gain Control	自动增益控制
BOM	Bill of Material	物料清单
DAC	Digital-Analog Converter	数模转换器
DMA	Direct Memory Access	直接存储访问
DNS	Domain Name System	域名解析系统
LNA	Low Noise Amplifier	低噪声放大器
NRE	Non-Recurring Engineering	一次性的开发
NTP	Network Time Protocol	网络时间协议
OOP	Object Oriented Programming	面向对象的程序设计
OSI	Open System Interconnect	传统的开放式系统互连
SDIO	Secure Digital Input and Output	安全数字输入输出
TCP/IP	Transmission Control Protocol/Internet Protocol	传输控制协议/因特网互联协议
Wi-Fi	Wireless Fidelity	一种基于802.11b标准的无线局域网传输协议

第一章 绪论

1.1 研究背景

随着计算机通讯技术的飞速发展，基于计算机技术和通信技术的新型病房呼叫系统应运而生，其应用可以更有效地帮助医护人员及时掌握患者的突发急危病情，尤其是无人陪护的急病患者的准确呼救信息，对迅速到达现场实施抢救提供了技术保障；同时它也是现代化医院护理，医院医疗管理体系的重要组成部分[1]。

当前市场上存在着许多种型号不一功能各异的医院病房呼叫系统，主要为两大类：有线式和无线式。传统的有线式病房呼叫系统往往采用集中式结构，电源线、数据通信线、语音通信线分开传输，铺设线路较多、成本高、安装调试困难、实时性差、故障率较高等缺点；而无线式病房呼叫系统不存在铺设线路的问题，但是可靠性差，而且无线电波会干扰其它医疗仪器设备，目前大多数医院不采用此类无线呼叫系统。以此，本课题以研究并设计一种简单实用、安全可靠、性能稳定具有良好性价比的病房呼叫系统，对于我国基层医院的现代化建设有十分重要的意义。本课题的研究将以呼叫系统的稳定性、语音数据传递的准确性、设备的复杂性等方面的问题作为主要研究内容[2]。

病房呼叫系统的发展大概可以分为3个阶段，即传统上的口头呼叫、摇铃呼叫和电子按铃呼叫3个阶段。相比而言，国外医院的呼叫系统发展比较迅速，现在已经渐渐出现了朝着护理系统的智能化和可视化发展，而我国的病房呼叫系统起步较晚，还处在电子按铃呼叫的初级阶段，与多年前相比变化不大。病床呼叫系统一般具有声光提示功能，有的系统有通话功能，使医护人员能够了解病人的医护请求。除人性化的辅助功能外，研究重点主要集中在主机与病房呼叫终端之间通信的安全性、可靠性、施工、系统构建、可扩展性等技术问题上进行研究。其通信方式可归纳为：有线通信和无线通信两种方式。

有线通信医疗呼叫系统，主机与病房呼叫终端之间采用导线连接进行通信和供电。20世纪80年代初，医疗呼叫一般都采用内部电话来构成呼叫系统，技术成熟，这种方式优点是：可进行双向呼叫和通话功能。缺点是：每个病床需要连接一根电话线，布线施工麻烦，成本也高。随着总线通信技术的发展和单片机应用的普及，采用CAN总线、RS-485总线、XY●CN总线、二线制或四线制脉冲编码通信总线、电力载波等总线通信技术的医疗呼叫系统得到了广泛应用。无线通信医疗呼叫系统，主机与病房呼叫终端之间采用无线电波或红外光等无线通信手段进

行通信，呼叫终端一般采用电池供电。主机与病房呼叫终端之间不需要布线，具有安装施工简单，能满足移动呼叫的需求。

随着我国医疗行业进入一个以服务为核心的新的发展周期，国内众多学者对于新型呼叫系统的设计与推广进行了广泛地研究。病床呼叫系统是病人请求值班医生或护士进行诊断或护理的紧急呼叫工具。可将病人的请求快速传送给值班医生或护士，是提高医院和病室护理水平的必备设备之一。如何利用先进的信息技术为医院服务，更大程度的提高医院的服务质量及利润，是医院信息化建设中的一个重要着眼点。随着现在计算机技术不断的发展，迫切的希望研发一套医院病房呼叫系统，来应对病房病人的需求[3]。

1.2 国内外发展现状

国外医院的呼叫系统发展比较迅速，现在已经渐渐出现了朝着护理系统的智能化和可视化发展，而我国的病房呼叫系统起步较晚，还处在电子按铃呼叫的初级阶段，与多年前相比变化不大。

国内的多位学者对医院病房呼叫系统都做出了研究曾进辉在《基于DTMF的医院护理呼叫系统的设计与实现》一文中提出了系统以单片机AT89C52为核心，采用信号发送芯片、信号接受芯片、语音播报芯片、发光二极管、数码管显示等外围电路以及相应的控制程序，实现了通过电话拨号进行单呼、群呼、显示、呼叫提示、查询、播报、对讲及护理级别的设置和删除等功能通过编码、传输及解码技术，能确保数据远距离传输且抗干扰能力强，避免了有线寻呼系统传输的不稳定性[4]。

朱艳华、田行军、李夏青等在《基于PL3105的病房呼叫系统设计》一文中提出了针对传统病房呼叫系统存在扩展困难或无线电干扰的现状，设计了基于可编程载波通信芯片PL3105的新型病房呼叫系统。系统分控制器和载波终端两层结构，均以PL3105为主控制单元，并以低压电力线为“总线”传输相应的控制命令。系统软件采用功能模块化设计思想，提高了系统移植性和可靠性。实测证明，新型病房呼叫系统不仅满足医疗单位的功能需求，而且具有扩展性强、维护方便、经济实用等优点，有较强的推广价值[5]。

乔国鹏在《基于ARM的数字化病房呼叫系统》一文中详细介绍了基于ARM的数字化病房呼叫系统的设计。以微控制器STM3210为核心控制终端设备接收和发送，采用RS232、RS485及UDP通信技术，实现了终端设备与服务台之间语音和通讯命令传输。该系统在实际使用中效果良好。系统利用单片机的自动控制特性，使得系统稳定、可靠。系统采用的元器件均是常见的电子元器件，因此系

统硬件成本较低。分机具有较低的功耗，并且具有较好的扩展性。主机与分机的通信稳定，实时性好，能满足各种规模医院的要求，有很好的应用前景。

郭广颂、胡璞在《基于单片机的无线病房呼叫系统设计》一文中详细介绍了在传统病房呼叫系统的基础上提出一种基于单片机实现的无线病房呼叫系统设计方法给出了基于单片机和射频通信芯片nRF401而设计的硬件原理图，并对影响无线通信性能的通信协议的设计、数据帧和防信息碰撞方法的实现及混合信号PCB板设计等做了较详细的探讨[?]。

潘绍明、梁喜幸《基于信号叠加和无线电的病房呼叫系统设计与实现》一文中介绍了包括走廊主机、监控室主机、电脑上位机、手持式监控机和病床呼叫分机的病房呼叫系统。各床位的呼叫信号通过两根既作为信号传输又为各病床呼叫分机提供电源的电线发送到走廊主机，再由走廊主机以无线电的方式向监控室主机和手持式监控机发送。监控室主机和手持式监控机在接收到信号后将会做出相应处理和反应。本设计加入了无线电通信，提高了医务人员工作的灵活性，能在无线电覆盖范围的任何一个位置接收到病人的呼叫信号，保证了病人的呼叫信号能在第一时间得到响应，很大程度上保障了病人的身体健康和生命安全[6]。

通过对上述研究成果的综合分析，和对目前医疗呼叫系统的工程实现的经验可以把当前医疗呼叫系统产品中存在的不足，归纳为以下几类[2]：

1. 有线通信医疗呼叫系统: 病房和值班室之间需要大量的连线，安装布线复杂，检查维修困难，病房扩建不易及费用高，不固定应用场所使用不便。
2. 单工无线通信医疗呼叫系统:可靠性低，抗干扰能力差，发射功率大，对医疗设备有干扰。
3. ZigBee协议无线通信医疗呼叫系统:载波频率高，穿透墙壁能力差，通信距离短;支持ZigBee协议的无线模块和微控制器成本相对较高。

近年来设计生产的呼叫器已普遍采用了单片机，使其功能大为增强。同时，值班室与病房间的连线也大大减少，布线简洁方便，迎合了医院追求环境整洁的需求。即便如此，但还是无法摆脱电线的束缚，布线麻烦，遇到病房扩建或改造，系统则需要重新布线，产品的重复使用率低，致使成本增加，需要新一代的产品来改善。目前市售的各种呼叫器均不具备个性录音功能，而在临床实践中发现，好多的医院需要自己独特的录音内容，这对于医生的护理将有大好处，这也是本文所介绍的病房语音呼叫系统功能上的创新点之一。

1.3 研究内容和意义

伴随着医疗体制改革的不断深化和医疗事业的飞速发展，越来越多的人们需要迅捷、方便地得到医院的各种各样的医疗服务，这必将使医院之间的竞争日趋激烈。这使得衡量一个医院的综合水平高低，不再仅仅局限于软、硬件的建设上，更要比服务。原有的服务体系已不足以适应现代社会需求，谋求适合现代社会需求的客户服务系统，是所有企事业单位计划做的工作。这些工作有利于改善服务量，提高效率并增加企业效益，从而赢得良好的社会声誉。如何利用先进的信息技术为医院服务，更大程度的提高医院的服务质量及利润，是医院信息化建设中的一个重要着眼点[3]。

医院的竞争越来越激烈，商业医院的生存是第一位的，提高档次和服务质量迫在眉睫，陪护问题一直是医患矛盾的主体，也是长期困扰卫生系统服务质量的大问题，使用无线呼叫系统，方便病人更快找到医生，以节约病人的宝贵时间。临床呼叫求助装置是传送临床信息的重要手段，关系病员安危，传统的有限呼叫系统历来受到各大医院的普遍重视。如果采用无线传输，会节约布线和改造线路的资金，为医院节约成本，并且及时、准确、可靠，简便可行，必然比目前的同类产品更能受到医院及病人的认可，有更强的竞争力，必然能大力推广[7]。

本文主要涉及硬件设计、通信技术、软件工程等的基本理论。研究内容包括语音呼叫终端的供电、主控电路、通信接口和通信协议、语音采集与回放、数据存储于访问等[?]。

临床呼叫装置是传送临床信息的重要手段，关系患者的安危。呼叫系统历来受到各大医院的普遍重视。为了方便患者，提高医院服务质量，医用呼叫系统已经成为了国内外各类医院中广泛使用的一种电子设备。它能从根本上解决传统医患之间所存在的一些服务纠纷等问题，可以静化医院的工作环境，避免无谓的争执。既可以帮助病人快速的呼叫医护人员，也可减轻医护人员巡视病房的辛劳，减轻医护人员值班的心理压力，在无呼叫时放心的作好其他医护工作，从而提高了医护效率。因此，医用呼叫系统具有广泛的社会意义与重大的实用价值[2?]。

1.4 研究思路和方法

本课题主要采用文献研究、需求调研与分析、总体方案论证、硬件设计和软件设计等方法，根据EDA/CAD、嵌入式系统和面向对象的程序设计等技术进行指导设计与测试。

呼叫终端的硬件设计是整个病房呼叫系统设计的基础，在系统设计中占有非常重要的作用。而它的硬件设计中，最重要的组成部分就是主控处理器模块，主

控处理器模块直接影响硬件终端的功能、性能优劣。因此，硬件方案的选择，其实就是终端硬件主控处理器模块和其他外设的确定，其决定了整个系统功能的优劣。

本课题涉及两个部分的软件开发过程，上位机部分和下位机部分，下位机为单片机部分，是指病患和医生使用的终端器件，其软件开发是基于模块化的功能实现，需要针对医生和病人设计不同的终端呼叫设备，上位机部分为电脑上运行的软件，或主控部分，或服务器部分，应该针对市面主流操作系统进行开发，完成对整个系统呼叫要求的申请和处理。

在通信方面，本系统设计时主要涉及物理层和媒体接入协议。当前市面上还没有为医院呼叫系统开发出专用的媒体接入协议，但是可以借鉴无线局域网的MAC协议的部分内容。针对静止或者移动缓慢的呼叫终端，可以通过有线网接入服务器，针对需要移动的终端可以选择带无线局域网的接入方式，可以考虑通过中心节点来控制局部网络，形成小型的多层局域网，或者选择微波中继站，或者ZigBee协议，来提高整个通信的一条距离，来减少整个系统的设计难度，当前最通用的无线局域网媒体接入协议可以根据硬件选择IEEE802.11、IEEE802.11b、IEEE802.11a、IEEE802.11g、IEEE802.11n等多种无线协议标准。

课题研究中，将使用Altium Designer设计系统的硬件电路原理图和PCB版图，拟采用STM32单片机作为语音呼叫终端的微控制器，配合国内开源的RT-Thread嵌入式实时操作系统管理所有硬件设备并完成语音的网络传输。嵌入式软件开发工具选择Keil uVision/MDK-ARM，上位机软件采用Microsoft的Visual Studio/C# ，数据库管理采用SQL。

1.5 本文的结构安排

第一章，绪论，本章主要对系统研究的背景以及系统研究的国内外发展现状作了详细的分析，对本文的研究意义以及本文所做的工作做了详细的分析。

第二章，语音呼叫系统的相关技术研究，本章对系统所用的到相关技术作了详细的介绍，对系统设计的软硬件技术以及系统的传输协议等进行了论述。

第三章，系统的硬件设计与实现，本章对系统的下位机硬件进行了详细的设计，实现了下位机的通信，通过对通信协议的设计优化保证了整个系统的通信可靠性。

第四章，系统的软件设计与实现，对上位机和下位机系统主要功能模块的设计和实现进行了详细阐述，并对上下位机进行联合调试，实现了整个系统的所有功能模块。

第五章，系统的部署与联合测试，对整个系统进行部署，对系统部署所需要的软硬件的环境进行了论述，以及系统的整个功能进行了测试，测试了系统功能的可用性以及系统的稳定性。

第六章，总结与展望，总结了本文的工作，对系统发展的前景作了论述。

本文提出了一种医院病房语音呼叫系统的解决方案，方便了医院对病房和病人的管理，解决了人工巡视病房的缺点，让病人随时可以方便地呼叫医护人员。系统硬件上采用无线传输以及微控制器电路设计，保证了系统的简洁、灵活、稳定、功耗低，底层硬件和顶层上位机系统通过无线网络实现互联，方便了系统的使用与部署。

第二章 语音呼叫系统的相关技术

2.1 嵌入式系统概述

2.1.1 嵌入式系统的定义

嵌入式系统是以应用为中心、以计算机技术为基础、软件硬件可裁剪、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。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。

嵌入式系统包含有计算机，但又不是通用计算机的计算机应用系统。通用计算机与嵌入式系统对比见表 2–1

表 2–1 通用计算机与嵌入式系统对比

特征	通用计算机	嵌入式系统
形式和类型	• 看得见的计算机。 • 按其体系结构、运算速度和结构规模等因素分为大、中、小型机和微机。	• 看不见的计算机。 • 形式多样，应用领域广泛，按应用来分。
组成	• 通用处理器、标准总线和外设。 • 软件和硬件相对独立。	• 面向应用的嵌入式微处理器，总线和外部接口多集成在处理器内部。 • 软件与硬件是紧密集成在一起的。
开发方式	• 开发平台和运行平台都是通用计算机。	• 采用交叉开发方式，开发平台一般是通用计算机，运行平台是嵌入式系统。
二次开发性	• 应用程序可重新编制	• 一般不能再编程

2.1.2 嵌入式系统的组成

嵌入式系统一般由嵌入式硬件和软件组成。硬件以微处理器为核心集成存储器和系统专用的输入/输出设备；软件包括：初始化代码及驱动、嵌入式操作系统和应用程序等，这些软件有机地结合在一起，形成系统特定的一体化软件。典型的嵌入式系统框架如图 2–1 所示。

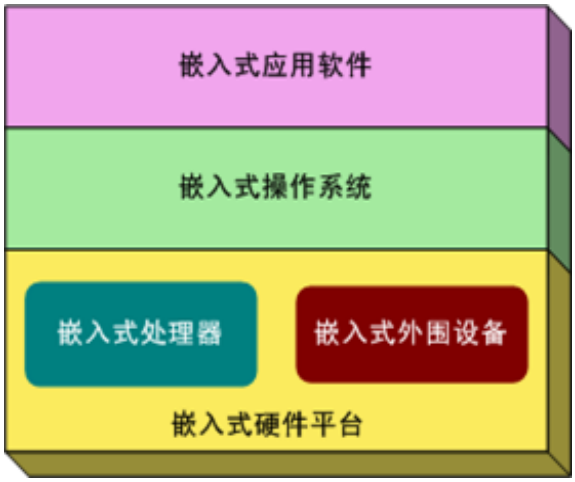


图 2–1 典型的嵌入式系统框架

2.1.3 嵌入式系统的特点

嵌入式系统具有以下特点[8? -10]：

1. 嵌入式系统通常是形式多样、面向特定应用的

   一般用于特定的任务，其硬件和软件都必须高效率地设计，量体裁衣、去除冗余，而通用计算机则是一个通用的计算平台。它通常都具有低功耗、体积小、集成度高等特点，能够把通用微处理器中许多由板卡完成的任务集成在芯片内部。嵌入式软件是应用程序和操作系统两种软件的一体化程序。

2. 嵌入式系统得到多种类型的处理器和处理器体系结构的支持

   通用计算机采用少数的处理器类型和体系结构，而且主要掌握在少数大公司手里。嵌入式系统可采用多种类型的处理器和处理器体系结构。

   在嵌入式微处理器产业链上，IP设计、面向应用的特定嵌入式微处理器的设计、芯片的制造已相成巨大的产业。大家分工协作，形成多赢模式。有上千种的嵌入式微处理器和几十种嵌入式微处理器体系结构可以选择。

3. 嵌入式系统通常极其关注成本；

   嵌入式系统通常需要注意的成本是系统成本，特别是量大的消费类数字化产品，其成本是产品竞争的关键因素之一。嵌入式的系统成本包括:

- 一次性的开发（Non-Recurring Engineering，NRE）成本；
- 产品成本:硬件物料（Bill of Material，BOM）成本、外壳包装和软件版税等；
- 批量产品的总体成本=NRE成本+每个产品成本*产品总量；
- 每个产品的最后成本=总体成本/产品总量=NRE成本/产品总量+每个产品成本。

4. 嵌入式系统有实时性和可靠性的要求

   一方面大多数实时系统都是嵌入式系统，另一方面嵌入式系统多数有实时性的要求，软件一般是固化运行或直接加载到内存中运行，具有快速启动的功能。并对实时的强度要求各不一样，可分为硬实时和软实时。嵌入式系统一般要求具有出错处理和自动复位功能，特别是对于一些在极端环境下运行的嵌入式系统而言，其可靠性设计尤其重要。在大多数嵌入式系统的软件中一般都包括一些机制，比如硬件的看门狗定时器，软件的内存保护和重启动机制。

5. 嵌入式系统使用的操作系统一般是适应多种处理器、可剪裁、轻量型、实时可靠、可固化的嵌入式操作系统

   由于嵌入式系统应用的特点，像嵌入式微处理器一样，嵌入式操作系统也是多姿多彩的。大多数商业嵌入式操作系统可同时支持不同种类的嵌入式微处理器。可根据应用的情况进行剪裁、配置。嵌入式操作系统规模小，所需的资源有限如内核规模在几十KB，能与应用软件一样固化运行。一般包括一个实时内核，其调度算法一般采用基于优先级的可抢占的调度算法。高可靠嵌入式操作系统：时、空、数据隔离。

6. 嵌入式系统开发需要专门工具和特殊方法

   多数嵌入式系统开发意味着软件与硬件的并行设计和开发，其开发过程一般分为几个阶段：产品定义、软件与硬件设计与实现、软件与硬件集成、产品测试与发布、维护与升级。由于嵌入式系统资源有限，一般不具备自主开发能力，产品发布后用户通常也不能对其中的软件进行修改，必须有一套专门的开发环境。该开发环境包括专门的开发工具（包括设计、编译、调试、测试等工具），采用交叉开发的方式进行，交叉开发环境如图2–2 所示。

2.2 面向对象的程序设计技术

面向对象的程序设计（Object Oriented Programming，OOP）是一种广泛使用

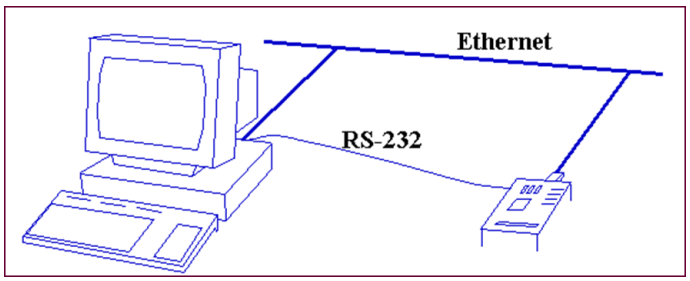


图 2–2 嵌入式系统的交叉开发环境

的计算机编程架构。面向对象的程序设计的一条最基本的原则就是计算机程序是由单个的能够起到子程序作用的单元或对象组合而成的。

2.2.1 对象

面向对象程序设计既是一种程序设计范型，同时也是一种程序开发的方法。对象指的是类的实例。OOP将对象作为程序的基本单元，将程序和数据封装其中，以提高软件的重用性、灵活性和扩展性。

一个具体对象属性的值被称作它的 “状态”。（系统给对象分配内存空间，而不会给类分配内存空间。这很好理解，类是抽象的系统不可能给抽象的东西分配空间，而对象则是具体的。

2.2.2 组件

即数据和功能一起在运行着的计算机程序中形成单元，组件在面向对象的程序设计所设计的计算机程序中是模块化和结构化的基础。

2.2.3 封装

也叫做信息封装，即确保组件不会以某种不可预期的方式改变其它组件的内部运行状态；只有在那些提供了内部运行状态改变方法的组件中，才可以访问其内部运行状态。每类组件都提供了一个与其它组件进行联系的接口，并且规定了其它组件对其进行调用的方法。

2.2.4 消息传递

一个对象通过接受消息、处理消息、传出消息或使用其他类的方法来实现一定功能，这叫做消息传递机制（Message Passing）。

面向对象程序设计可以看作一种在程序中包含各种独立而又互相调用的对象的思想，这与传统的思想刚好相反：传统的程序设计主张将程序看作一系列函数的集合，或者直接就是一系列对电脑下达的指令。为了实现整体运算，面向对象程序设计中的每一个对象都应该能够接受数据、处理数据并将数据传达给其它对象，因此它们都可以被看作一个小型的 “机器”，即对象。

2.2.5 OOP的主要特性

为什么病房语音呼叫系统的开发要选用面向对象的程序设计呢？因为面向对象的程序设计实现了软件工程的三个主要目标：重用性、灵活性和扩展性。

图4-1 程序设计的层次面向对象的程序设计具备下列特性[11]：

抽象性——即程序有能力忽略正在处理中的某些信息的某些方面，即对信息的主要方面进行关注的能力。

多态性——即组件的引用以及类集会涉及到许多其它不同类型的组件，而且引用组件所产生的结果必须依据实际调用的类型。

继承性——即允许在现存的组件基础之上创建子类组件，这统一并且增强了程序的多态性和封装性。简单地说来就是使用类来对组件进行划分，而且还可以定义新类作为现存类的扩展，于是这样就可以将类组织成树形结构或者网状结构，同时这也体现了动作的通用性。

正是因为面向对象的程序设计具有以上的优越特点，才使得它满足病房语音呼叫系统的开发需求，方便程序接口的设计。

2.3 TCP/IP协议

TCP/IP是Transmission Control Protocol/Internet Protocol的简写，中译名为传输控制协议/因特网互联协议，又名网络通讯协议，是Internet最基本的协议、Internet国际互联网络的基础，由网络层的IP协议和传输层的TCP协议组成。TCP/IP定义了电子设备如何连入因特网，以及数据如何在它们之间传输的标准。协议采用了4层的层级结构，每一层都呼叫它的下一层所提供的协议来完成自己的需求。通俗而言：TCP负责发现传输的问题，一有问题就发出信号，要求重新传输，直到所有数据安全正确地传输到目的地。而IP是给因特网的每一台联网设备规定一个地址[12, 13]。

2.3.1 TCP/IP协议的层次

TCP/IP协议不是TCP和IP这两个协议的合称，而是指因特网整个TCP/IP协议族。

从协议分层模型方面来讲，TCP/IP由四个层次组成：网络接口层、网络层、传输层、应用层。

TCP/IP协议并不完全符合OSI（，Open System Interconnect）的七层参考模型，OSI是传统的开放式系统互连参考模型，是一种通信协议的7层抽象的参考模型，其中每一层执行某一特定任务。该模型的目的是使各种硬件在相同的层次

上相互通信。这7层是：物理层、数据链路层（网络接口层）、网络层（网络层）、传输层（传输层）、会话层、表示层和应用层（应用层）。而TCP/IP通讯协议采用了4层的层级结构，每一层都呼叫它的下一层所提供的网络来完成自己的需求。由于ARPANET的设计者注重的是网络互联，允许通信子网（网络接口层）采用已有的或是将来有的各种协议，所以这个层次中没有提供专门的协议。实际上，TCP/IP协议可以通过网络接口层连接到任何网络上，例如X.25交换网或IEEE802局域网。

2.3.2 IP

IP层接收由更低层（网络接口层例如以太网设备驱动程序）发来的数据包，并把该数据包发送到更高层--TCP或UDP层；相反，IP层也把从TCP或UDP层接收来的数据包传送到更低层。IP数据包是不可靠的，因为IP并没有做任何事情来确认数据包是否按顺序发送的或者有没有被破坏，IP数据包中含有发送它的主机的地址（源地址）和接收它的主机的地址（目的地址）[14]。

高层的TCP和UDP服务在接收数据包时，通常假设包中的源地址是有效的。也可以这样说，IP地址形成了许多服务的认证基础，这些服务相信数据包是从一个有效的主机发送来的。IP确认包含一个选项，叫作IP source routing，可以用来指定一条源地址和目的地址之间的直接路径。对于一些TCP和UDP的服务来说，使用了该选项的IP包好像是从路径上的最后一个系统传递过来的，而不是来自于它的真实地点。这个选项是为了测试而存在的，说明了它可以被用来欺骗系统来进行平常是被禁止的连接。那么，许多依靠IP源地址做确认的服务将产生问题并且会被非法入侵[14]。

2.3.3 TCP

TCP是面向连接的通信协议，通过三次握手建立连接，通讯完成时要拆除连接，由于TCP是面向连接的所以只能用于端到端的通讯。

TCP提供的是一种可靠的数据流服务，采用“带重传的肯定确认”技术来实现传输的可靠性。TCP还采用一种称为“滑动窗口”的方式进行流量控制，所谓窗口实际表示接收能力，用以限制发送方的发送速度。

如果IP数据包中有已经封好的TCP数据包，那么IP将把它们向‘上’传送到TCP层。TCP将包排序并进行错误检查，同时实现虚电路间的连接。TCP数据包中包括序号和确认，所以未按照顺序收到的包可以被排序，而损坏的包可以被重传[14, 15]

TCP将它的信息送到更高层的应用程序，例如Telnet的服务程序和客户程序。应用程序轮流将信息送回TCP层，TCP层便将它们向下传送到IP层，设备驱动程序和物理介质，最后到接收方。

面向连接的服务（例如Telnet、FTP、rlogin、X Windows和SMTP）需要高度的可靠性，所以它们使用了TCP。DNS（Domain Name System，域名解析系统）在某些情况下使用TCP（发送和接收域名数据库），但使用UDP传送有关单个主机的信息。

2.3.4 UDP

UDP是面向无连接的通讯协议，UDP数据包括目的端口号和源端口号信息，由于通讯不需要连接，所以可以实现广播发送。

UDP通讯时不需要接收方确认，属于不可靠的传输，可能会出现丢包现象，实际应用中要求程序员编程验证。

UDP与TCP位于同一层，但它不管数据包的顺序、错误或重发。因此，UDP不被应用于那些使用虚电路的面向连接的服务，UDP主要用于那些面向查询--应答的服务，例如NFS。相对于FTP或Telnet，这些服务需要交换的信息量较小。使用UDP的服务包括NTP（Network Time Protocol，网络时间协议）和DNS（DNS也使用TCP）。

欺骗UDP包比欺骗TCP包更容易，因为UDP没有建立初始化连接（也可以称为握手，因为在两个系统间没有虚电路），也就是说，与UDP相关的服务面临着更大的危险[14, 15]。

2.4 数据库技术

上位机系统需要通过计算机通信接口采集所有下位机用户的用电数据，这些数据随时间积累会变得很庞大，好在它们都是有固定格式的数据，使用数据库对其进行管理便自然而然地成了最佳选择[?]。

数据库技术是信息系统的一个核心技术。是一种计算机辅助管理数据的方法，它研究如何组织和存储数据，如何高效地获取和处理数据。是通过研究数据库的结构、存储、设计、管理以及应用的基本理论和实现方法，并利用这些理论来实现对数据库中的数据进行处理、分析和理解的技术。即：数据库技术是研究、管理和应用数据库的一门软件科学[?]。

数据库技术是现代信息科学与技术的重要组成部分，是计算机数据处理与信息管理系统的核心。数据库技术研究和解决了计算机信息处理过程中大量数据有

效地组织和存储的问题，在数据库系统中减少数据存储冗余、实现数据共享、保障数据安全以及高效地检索数据和处理数据。

数据库技术研究和管理的对象是数据，所以数据库技术所涉及的具体内容主要包括：通过对数据的统一组织和管理，按照指定的结构建立相应的数据库和数据仓库；利用数据库管理系统和数据挖掘系统设计出能够实现对数据库中的数据进行添加、修改、删除、处理、分析、理解、报表和打印等多种功能的数据管理和数据挖掘应用系统；并利用应用管理系统最终实现对数据的处理、分析和理解[16]。

2.5 本章小结

本章主要对病房语音呼叫系统中需要涉及到的主要相关技术及其相关概念进行了简要介绍和概括：嵌入式系统、面向对象的程序设计技术、TCP/IP协议、数据库技术等。

第三章 系统硬件的设计与实现

3.1 系统硬件的总体方案设计

任何通信都涉及到信道和传输媒介的问题，考虑到有线通信方式在系统部署时需要铺设大量的通信电缆，产生巨大的安装成本而且也不利于设备维护，影响外观。而目前 Wi-Fi（Wireless Fidelity）的覆盖越来越广泛，使用Wi-Fi作为病房语音呼叫系统接入网络的方式就成了不言而喻的选择。系统硬件上以嵌入式微控制器为核心，通过外设IO与各个电路或模块互联，总体方案框图如图 3–1所示。

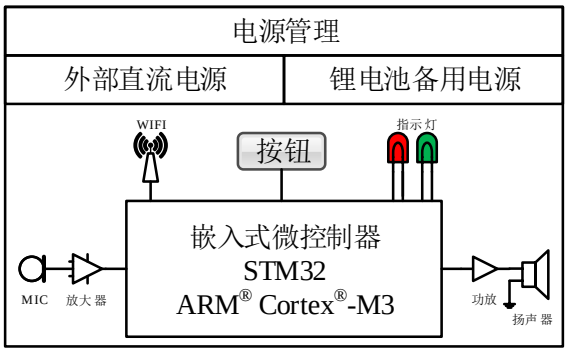


图 3–1 系统硬件的总体方案框图

3.2 系统电源设计

考虑到病房语音呼叫的特殊性，系统硬件一般情况下使用外部直流电源供电，遇到停电等特殊情况时使用内部电池作为后备电源。外部电源供电时通过TP4056锂电池电源管理芯片给电池充电以延长电池寿命。为尽量降低功耗，系统电源芯片RT8008在常态下是关闭的，只在三种情况下才开启：（1）定时唤醒；（2）语音呼叫按钮被按下；（3）防破坏拆除按钮被断开。任何一种情况都会是RT8008的使能信号Power_en有效，系统电源开启，详见第 3.7 节阐述。

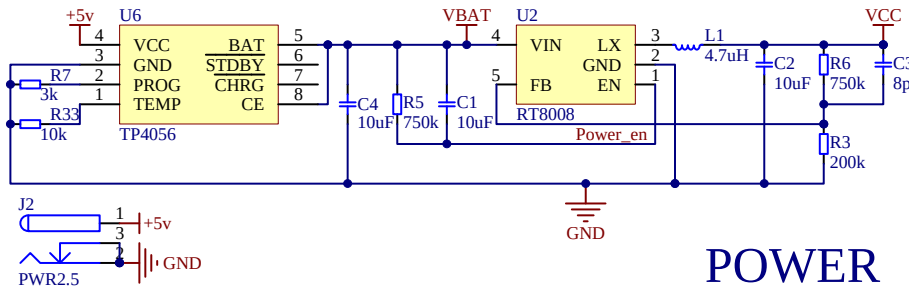


图 3–2 系统电源电路原理图

3.3 微控制器最小系统设计

3.3.1 STM32嵌入式微控制器简介

STM32系列嵌入式微控制器基于专为要求高性能、低成本、低功耗的嵌入式应用专门设计的ARM®Cortex®-M3内核。ARM®Cortex®-M3 处理器是行业领先的 32 位处理器，适用于具有较高确定性的实时应用，它经过专门开发，可使合作伙伴针对广泛的设备（包括微控制器、汽车车身系统、工业控制系统以及无线网络和传感器）开发高性能低成本平台。此处理器具有出色的计算性能以及对事件的优异系统响应能力，同时可应实际中对低动态和静态功率需求的挑战。此处理器配置十分灵活，从而支持广泛的实现形式（从需要内存保护和强大 trace 技术的实现形式，直至需要极小面积的成本敏感型设备）[17]。

STM32系列嵌入式微控制器按性能分成两个不同的系列：STM32F103“增强型”系列和STM32F101“基本型”系列。增强型系列时钟频率达到72MHz，是同类产品中性能最高的产品；基本型时钟频率为36MHz，以16位产品的价格得到比16位产品大幅提升的性能，是16位产品用户的最佳选择。两个系列都内置32K到128K的闪存，不同的是SRAM的最大容量和外设接口的组合。时钟频率72MHz时，从闪存执行代码，STM32功耗36mA，是32位市场上功耗最低的产品，相当于0.5mA/MHz[18]。

3.3.2 基于STM32的嵌入式最小系统

事实上STM32单片机最小系统是不需要外部时钟的，其内部集成了RC高速振荡器可作为系统时钟源，只不过考虑到语音采样或者还原需要比较精确的时钟控制，选择了片外晶体振荡器为STM32提供系统时钟。预留的SWD接口为STM32的编程调试接口，用官方的ST-LINK或者第三方工具J-LINK可以方便地对STM32进行在线调试或编程。一红一绿两个LED用于指示系统工作状态：通常情况下，绿色指示灯每1s闪烁一次表示系统工作正常且处于待机状态；用户按下语音呼叫按钮后红色LED闪烁表示正在呼叫中；语音呼叫接通后，绿色LED长亮表示处于语音通话状态；如果长时间不能接通，则红色LED长亮表示通信故障；语音通话结束后，回到待机状态，绿色LED每秒闪烁一次。

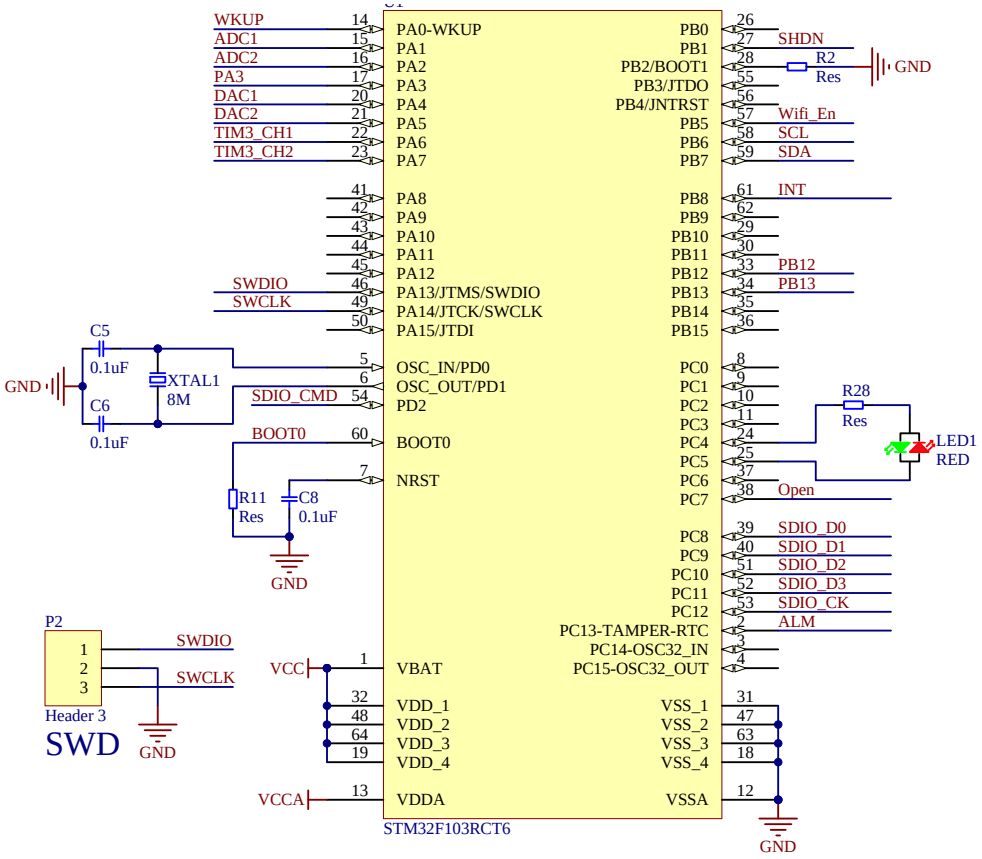


图 3–3 STM32最小系统电路原理图

3.4 语音输入电路

语音输入电路如图 3–4所示，语音通过驻极话筒转化为电信号，经电容耦合进入放大电路，微弱的语音信号经过两级运算放大器从mV级别放大到约2.5 Vpp左右，送入STM32的模拟信号输入引脚，最后经其内部的ADC（Analog-to-Digital Converter，模拟数字转换器）转化为数字语音信号。

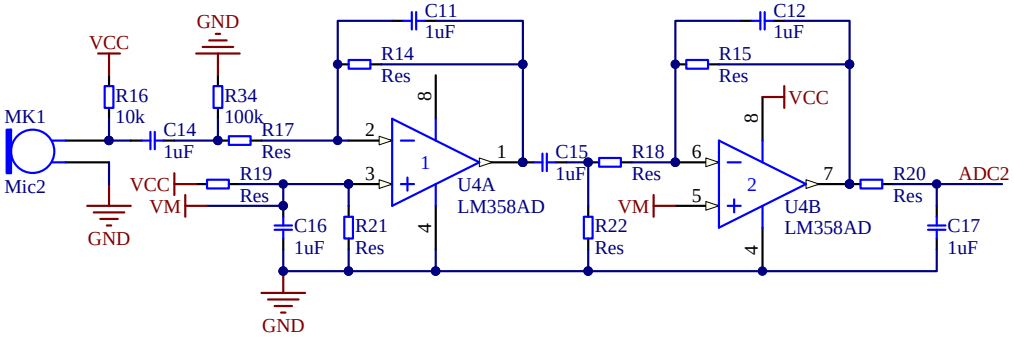


图 3–4 语音输入电路原理图

3.5 语音输出电路

对方用户端采集和处理后的数字语音信号通过网络实时传输到接收端，接收端的STM32微控制器则将语音数据流通过其片内的DAC（Digital-Analog Converter，数字模拟转换器））按特定的速率转换为模拟语音电信号，再经过低通滤波去除高频数字噪声、功率放大后驱动扬声器还原成声音。语音功放芯片带有使能引脚SHTN，接高电平时正常工作，接低电平时停止工作，消耗极小的电流。STM32微控制器通过PB1引脚控制其工作状态，待机时关闭语音功放，语音通话时开启语音功放。

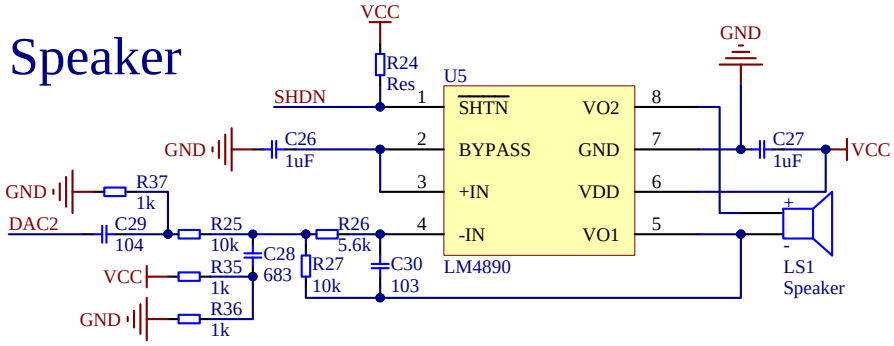


图 3–5 语音输出电路原理图

3.6 Wi-Fi模块电路

Wi-Fi模块WM-G-MR-08外围电路也比较简单，主要包括天线匹配电路、接收低噪声放大器（Low Noise Amplifier, LNA）的自动增益控制（Automatic Gain Control, AGC）设置。Wi-Fi模块通过 SDIO（Secure Digital Input and Output，安全数字输入输出）接口与STM32微控制器通信，SDIO_CMD为命令信号，SDIO_CK为时钟信号，SDIO_D[3：0]为数据信号。为降低功耗，Wi-Fi模块也有掉电控制引脚PDn，低电平时模块进入掉电模式，耗电极低，高电平时模块进入工作模式。由STM32微控制器的PB5引脚控制Wi-Fi模块掉电或工作。

3.7 呼叫按钮与掉电管理电路

考虑到病房语音呼叫的特殊性，系统硬件一般情况下使用外部直流电源供电，遇到停电等特殊情况时使用内部电池作为后备电源。为尽量降低功耗，延长电池寿命，使用PCF8563作为定时掉电管理芯片。常态下系统电源关闭，在三种情况下系统会被唤醒：（1）PCF8563经过程序预定的时间后输出一唤醒信号（类似于关机闹钟）；（2）语音呼叫按钮被按下；（3）防破坏拆除按钮被断开。任何一种

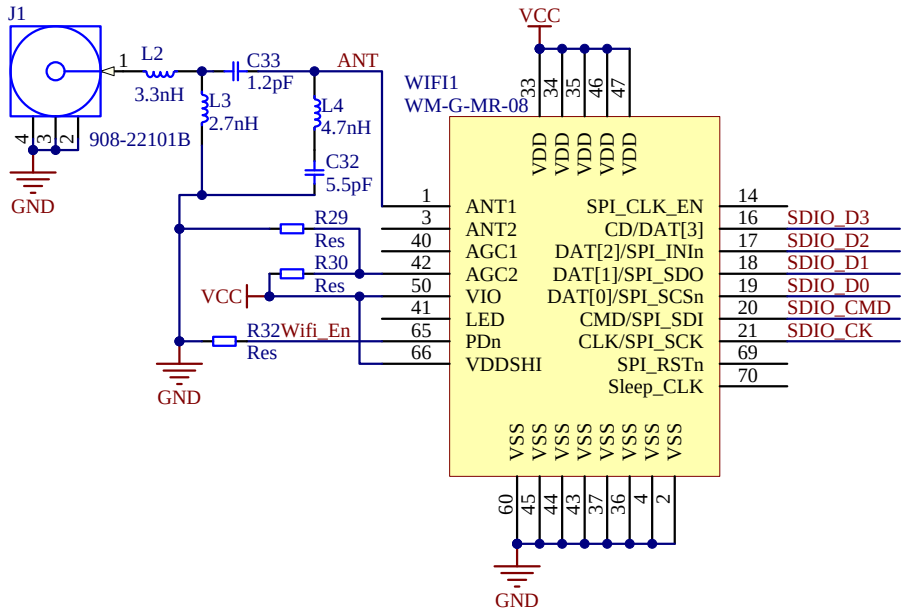


图 3–6 Wi-Fi模块电路原理图

情况都会使系统电源开启，STM32微控制器上电。STM32开始工作后首先判断唤醒原因并执行不同的程序：（1）定时唤醒，使能Wi-Fi模块，尝试向服务器发送心跳数据包表明自己工作正常并判断网络是否通畅；（2）语音呼叫按钮S2被按下，定时唤醒，使能Wi-Fi模块，尝试向服务器发送语音呼叫请求；（3）防破坏拆除按钮S1被断开，说明设备被人为破坏，警报音响起，同时使能Wi-Fi模块，尝试向服务器发送报警数据包。

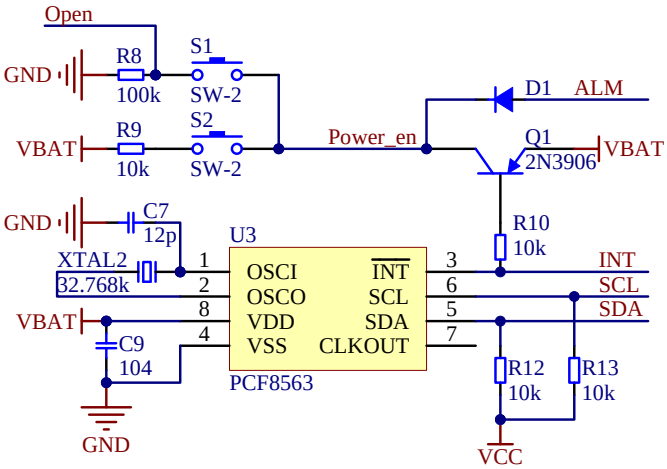
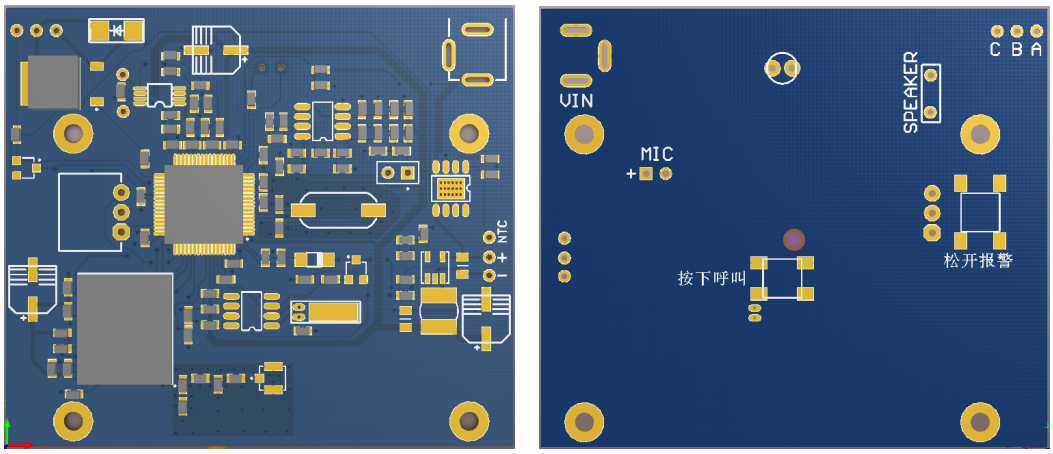


图 3–7 呼叫按钮与掉电管理电路原理图

3.8 系统硬件的PCB版图

系统硬件的原理图和PCB版图均使用Altium Designer设计，PCB三维效果图如图 3–8所示。PCB尺寸65 mm × 56 mm，外壳使用标准“八六盒”（86 mm × 86 mm）。



(a) (b)

图 3–8 系统硬件的PCB版图

(a)Toplayer；(b)Bottomlayer

3.9 本章小结

本章主要阐述了系统硬件的总体设计方案，对系统电源、微控制器、语音输入输出电路、Wi-Fi模块电路等各个功能的硬件电路进行了分别详细的阐述，并给出了对应的电路原理图。

第四章 系统软件的设计与实现

本文拟设计与的实现的病房语音呼叫系统，研究的重点除了硬件上的设计之外，再者就是软件上的创新。众所周之，任何一个自动化系统都是由硬件系统与软件系统两部分组成的。硬件是系统实现功能的基础，软件则是系统功能实现的核心。软件系统的可靠稳定性也将对系统的整体性能产生重要影响。

本系统的设计特别考虑了在不同的工作环境下，如何保证病房语音呼叫系统正常工作，这就要求系统要具备强健的抗干扰能力。此系统的抗干扰问题的解决与抗干扰功能的设计也就成为了软件系统设计的重要内容。通常来讲，监测系统上的抗干扰设计既可以通过硬件来实现，也可以通过软件来实现。本文探讨的重点是如何通过软件和硬件结合来解决系统的抗干扰问题。

4.1 系统嵌入式软件的设计与实现

4.1.1 RT-Thread嵌入式实时操作系统简介

RT-Thread是一个开放源代码的实时操作系统，并且商业许可证非常宽松的实时操作系统。图 4–1是RT-Thread及外围组件的基本框架图。

RT-Thread Kernel内核部分包括了RT-Thread的核心代码，包括对象管理器，线程管理及调度，线程间通信等的微小内核实现（最小能够到达4k ROM，1k RAM体积占用）。内核库是为了保证内核能够独立运作的一套小型类似C库实现（这部分根据编译器自带C库的情况会有些不同，使用GCC编译器时，携带更多的标准C库实现）。CPU及板级支持包包含了RT-Thread支持的各个平台移植代码，通常会包含两个汇编文件，一个是系统启动初始化文件，一个是线程进行上下文切换的文件，其他的都是C源文件。

RT-Thread实时操作系统核心是一个高效的硬实时核心，它具备非常优异的实时性、稳定性、可剪裁性。最小可以到3KB ROM占用、1KB RAM占用。

② 图取自《RT-Thread实时操作系统编程指南0.3.0》P25

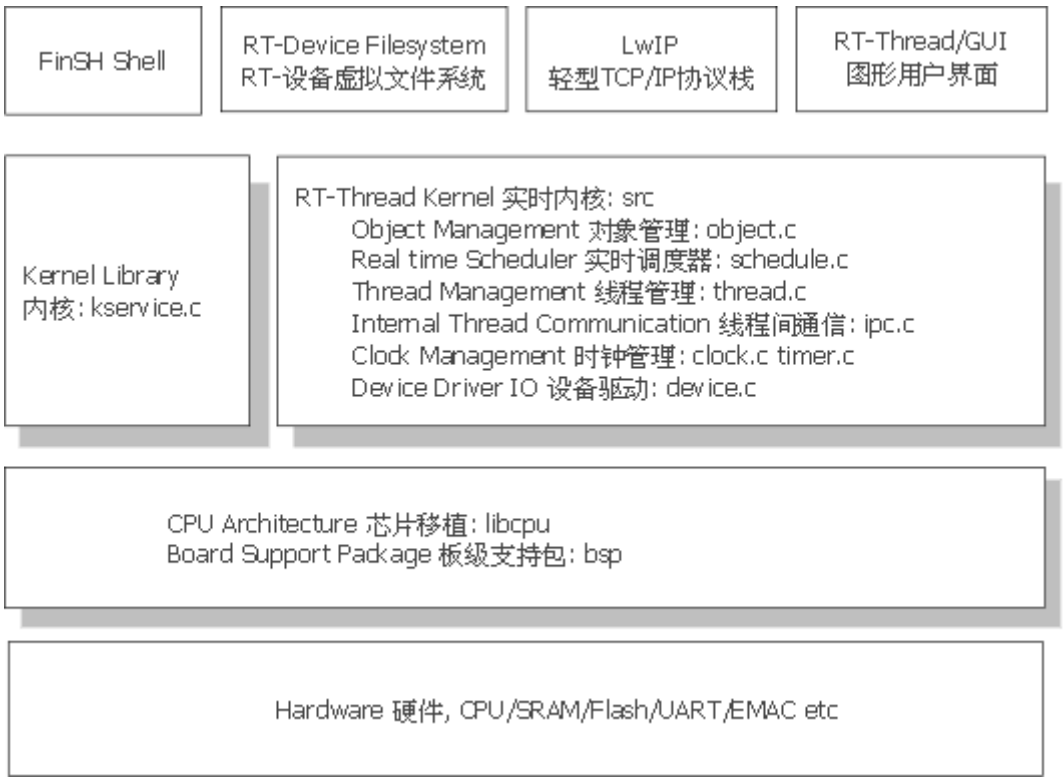


图 4–1 RT-Thread及外围组件基本框架②

4.1.1.1 内核对象系统

实时线程操作系统内部采用面向对象的方式设计，内建内核对象管理系统，能够访问/管理所有内核对象。内核对象包含了内核中绝大部分设施，而这些内核对象可以是静态分配的静态对象，也可以是从系统内存堆中分配的动态对象。通过内核对象系统，RT-Thread可以做到不依赖于具体的内存分配方式，伸缩性得到极大的加强[19]。

4.1.1.2 任务/线程调度

支持以线程为基本调度单位的多任务系统。调度算法是基于优先级的全抢占式线程调度，支持256个线程优先级（亦可配置成32个线程优先级），0优先级代表最高优先级，255优先级留给空闲线程使用；相同优先级上支持多个线程，这些相同优先级的线程采用可设置时间片长度的时间片轮转调度；调度器寻找下一个最 高优先级就绪线程的时间是恒定的(O(1))。系统不限制线程数量的多少，只与物理平台的具体内存相关[19]。

4.1.1.3 同步机制

系统支持semaphore，mutex等线程间同步机制。mutex采用优先级继存方式以防止优先级翻转。semaphore释放动作可安全用于中断服务例程中。同步机制支持线程按优先级等待或按先进先出方式获取信号量或互斥锁[19]。

4.1.1.4 通信机制

系统支持event，mailbox，message queue通信机制等。event支持多事件”或触发”及”与触发”，适合于线程等待多个事件情况。mailbox中一个mail的长度固定为4字节，效率较messagequeu高。通信设施中的发送动作可安全用于中断服务例程中。通信机制支持线程按优先级等待或按先进先出方式获取[19]。

4.1.1.5 时钟/定时器

系统默认使用时钟节拍来完成同优先级任务的时间片轮转调度；线程对内核对象的时间敏感性是通过系统定时器来实现的；定时器又分成了硬定时器和软定时器，一次定时及周期性定时[19]。

4.1.1.6 内存管理

系统支持静态内存池管理及动态内存堆管理。从静态内存池中获取/释放内存块时间恒定，而当内存池空时，可根据申请线程请求把申请线程挂起、立刻返回、或等待一段时间仍未获得返回。当其他线程释内存块到内存池时，将把挂起的线程唤醒。对于系统内存紧张的系统，RT-Thread也提供了小型的伙伴内存管理算 法。而对于拥有大一些内存的嵌入式系统，RT-Thread提供了性能非常高效的SLAB内存管理系统[19, 20]。

4.1.1.7 诊断

通过系统提供的FinSH shell系统，能够查看到线程，信号量，互斥锁，事件，邮箱，消息队列的运行情况，以及各个线程的栈使用情况[19, 21]。

4.1.2 RT-Thread在STM32上的移植与配置

ARM® Cortex®-M3微处理器可以说是和ARM7TDMI微处理器完全不同的体系结构，在进行RT-Thread移植时首先需要把线程的上下文切换移植好。通常的ARM移植，RT-Thread需要手动的保存当前模式下几乎所有寄存器，R0 –R13，LR，PC，CPSR，SPSR等。在Cortex®-M3微处理器中，则不需要保存全部的寄存器到栈中，因为当一个异常触发时，Cortex®-M3硬件能够自动的完成部分的寄存器保存。

当要进行切换时（假设从Thread [from] 切换到Thread [to]），通过rt_hw_context_switch函数触发一个PenSV异常。异常产生时，Cortex®-M3会把PSR，PC，LR，R0 –R3，R12自动压入当前线程的栈中，然后切换到PenSV异常处理。到PenSV异常后，Cortex®-M3工作模式切换到Handler模式，由函数rt_hw_pend_sv进行处理。

rt_hw_pend_sv函数会载入切换出线程和切换到线程的栈指针，如果切换出线程的栈指针是0那么表示这是第一次线程上下文切换，不需要对切换出线程做压栈动作。如果切换出线程栈指针非零，则把剩余未压栈的R4 – R11寄存器依次压栈；然后从切换到线程栈中恢复R4 – R11寄存器。当从PendSV异常返回时，PSR，PC，LR，R0 – R3，R12等寄存器由Cortex® M3自动恢复。

当中断达到时，当前线程会被中断并把PC，PSR，R0 –R3，R12等压到当前线程栈中，工作模式切换到Handler模式。在运行中断服务例程期间，如果发生了线程切换（调用rt_schedule），会先判断当前工作模式是否是Handler模式（依赖于全局变量rt_interrupt_nest），如果是则调用rt_hw_context_switch_interrupt函数进行伪切换[?]：

在rt_hw_context_switch_interrupt函数中，将把当前线程栈指针赋值到rt_interrupt_from_thread变量上，把要切换过去的线程栈指针赋值到rt_interrupt_to_thread 变量上，并设置中断中线程切换标志 rt thread_switch_interrput_flag 为1。

在最后一个中断服务例程结束时，Cortex®-M3将去处理PendSV异常，因为PendSV异常的优先级是最低的，所以只有触发过PendSV异常，它将总是在最后进行处理[22]。

这里就不进一步详细讨论RT-Thread在STM32上的移植了。事实上，RT-Thread开发组已经做了好STM32系列微控制器的大部分移植工作，从官网下载源码后作简单修改即可完成符合自身硬件的移植工作。

前面讨论了，嵌入式操作系统是有很高的可裁剪性的，RT-Thread也不例外，通过启用或者关闭rtconfig.h头文件中的宏定义就可以很方便地保留或者裁剪系统的功能代码。本系统中将保留RT-Thread的默认配置选项：

- 线程优先级支持，32优先级
- 内核对象支持命名，4字符
- 操作系统节拍单位，10毫秒（100拍/秒）
- 支持钩子函数
- 支持信号量、互斥锁
- 支持事件、邮箱、消息队列
- 支持内存池
- 支持RT-Thread自带的动态堆内存分配器

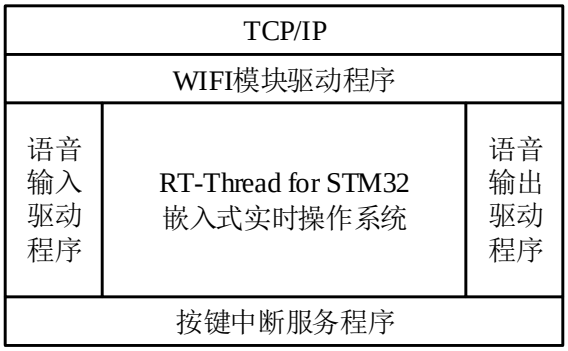


图 4–2 系统嵌入式软件的总体框架

4.1.3 系统嵌入式软件的总体框架

4.1.4 建立系统主要线程

本应用中嵌入式系统主要有两个线程：LED指示灯线程和UDP通信线程。采用事件驱动方式：无事件发生时系统空闲，执行操作系统的自带的idle线程；有事件发生时发送信号量或者消息完成进程间同步。

4.1.4.1 LED指示灯线程

为LED线程分配堆栈和声明线程：

```
01 static rt_uint8_t led_stack[ RT_STACK_SIZE ];
02 static struct rt_thread led_thread;
```

定义LED线程入口并完成功能程序：

```
01 static void led_thread_entry(void* parameter)
02 {
03     rt_uint32_t led_mode=*((rt_uint32_t)parameter);
04     rt_hw_led_init();//LED对应IO初始化
05     //限于篇幅，以下仅给出主要逻辑框架，不列举完整代码
06     while (1)
07     {
08         switch(led_mode)
09         {
10             case LED_MODE_BOTH_ON:
11                 //红色LED亮;绿色LED亮;
12             break;
13             case LED_MODE_BOTH_OFF:
```

```
14                  //红色LED灭;绿色LED灭;
15              break;
16              case LED_MODE_RED_ON:
17                  //红色LED亮;绿色LED灭;
18              break;
19              case LED_MODE_GREEN_ON:
20                  //红色LED灭;绿色LED亮;
21              break;
22              case LED_MODE_RED_TOGGLE:
23                  //红色LED取反;绿色LED灭;
24              break;
25              case LED_MODE_GREEN_TOGGLE:
26                  //红色LED灭;绿色LED取反;
27              break;
28              case LED_MODE_BOTH_TOGGLE:
29                  //红色LED取反;绿色LED取反;
30              break;
31          }
32          rt_thread_delay( RT_TICK_PER_SECOND/2 );
33      }
34  }
```

在rt_application_init()中初始化并启动LED线程：

```
01 int rt_application_init()
02 {
03     rt_thread_t init_thread;
04     rt_err_t    result;
05
06     /* init led thread */
07     result = rt_thread_init(&led_thread,
08                             "led",
09                             led_thread_entry, RT_NULL,
10                             (rt_uint8_t*)&led_stack[0], sizeof(led_stack),
```

```
30, 1);
11      if (result == RT_EOK)
12      {
13          rt_thread_startup(&led_thread);
14      }
15      /*创建一个用于初始化其他线程的线程"init"*/
16      init_thread = rt_thread_create("init",
17                                     rt_init_thread_entry, RT_NULL,
18                                     2048, 8, 20);
19
20      if (init_thread != RT_NULL)
21          rt_thread_startup(init_thread);
22
23      return 0;
24 }
```

以上程序专门创建了一个“init”线程用来对其他用户线程完成初始化工作。

4.1.4.2 UDP通信线程

为UDP通信线程分配堆栈和声明线程：

```
01 static rt_uint8_t udp1_stack[ RT_STACK_SIZE ];
02 static struct rt_thread udp1_thread;
```

定义UDP通信线程入口并完成功能程序：

```
01 static void udp1_thread_entry(void* parameter)
02 {
03      //限于篇幅，这部分代码仅给出大致框架，不列举全部内容;
04      //变量定义、声明和初始化;
05      //初始化WIFI模块配置;
06      //尝试通过WIFI连接到路由器;
07      //创建UDP连接到服务器;
08      //更新当前命令;
09      while (1)
10      {
11          pt[1]=butn_cmd;
```

```
12          switch(butn_cmd)
13          {
14              case BUTN_CMD_REPORT_RQST:
15                  //向服务器报告系统状态;
16              break;
17              case BUTN_CMD_VOICE_RQST:
18                  //向服务器发出语音呼叫请求;
19              break;
20              case BUTN_CMD_VOICE_DATA:
21                  //向服务器发送本机采集到的语音数据流;
22              break;
23              case BUTN_CMD_CONFIG_RQST:
24                  //向服务器发出系统参数配置请求;
25              break;
26              case BUTN_CMD_CONFIG_RPLY:
27                  //向服务器恢复已经收到参数配置信息;
28              break;
29              case BUTN_CMD_SET_ALARM:
30                  //向服务器发出语音呼叫请求;
31              break;
32              case BUTN_CMD_NONE:
33                  //空命令;
34              break;
35              case BUTN_CMD_UDP_RQST://UDP打洞请求
36                  //向服务器发出UDP打洞请求（用于系统与服务器不在同一
网段的情况）;
37              break;
38              case BUTN_CMD_SYS_RESET:
39                  //系统出错时进行复位;
40              break;
41          }
42      }
43  }
```

在rt_init_thread_entry中初始化并启动UDP线程：

```
    result = rt_thread_init(&udp1_thread,
                            "udp1",
                            udp1_thread_entry, RT_NULL,
                            (rt_uint8_t*)&udp1_stack[0], sizeof(udp1_stack),
10, 20);
    if (result == RT_EOK)
    {
        rt_thread_startup(&udp1_thread);
    }
```

4.1.5 按键中断服务程序

配置按键对应的IO引脚为带外部中断的输入模式:

```
void key_io_init(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA|RCC_APB2Periph_GPIOB
|RCC_APB2Periph_AFIO, ENABLE);
    //以下为按键输入引脚，用于判断外部事件
    GPIO_InitStructure.GPIO_Pin = OPEN_KEY|ALARM_KEY;//OPEN_KEY:PB11
& ALARM_KEY:PB10
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
    GPIO_Init(GPIOB, &GPIO_InitStructure);
    GPIO_EXTILineConfig(GPIO_PortSourceGPIOB, GPIO_PinSource10);
}
```

使能按键对应的外部中断线，并配置为上升沿或下降沿触发中断:

```
void key_exti_init(void)
{
    EXTI_InitStruct.EXTI_Line=EXTI_Line10;//ALARM_KEY
```

```
04      EXTI_InitStruct.EXTI_Mode=EXTI_Mode_Interrupt;
05      EXTI_InitStruct.EXTI_Trigger=EXTI_Trigger_Falling;
06      EXTI_InitStruct.EXTI_LineCmd=ENABLE;
07      EXTI_Init(&EXTI_InitStruct);
08  }
```

配置按键对应外部中断的优先级:

```
01  void key_nvic_init(void)
02  {
03      NVIC_InitTypeDef NVIC_InitStructure;
04      //使能按键所在的外部中断通道
05      NVIC_InitStructure.NVIC_IRQChannel = EXTI15_10_IRQn;
06      //先占优先级4位,共16级
07      NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0x02;
08      //先占优先级0位,从优先级4位
09      NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0x02;
10      //使能外部中断通道
11      NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
12      NVIC_Init(&NVIC_InitStructure);
13  }
```

编写按键中断服务程序:

```
01  void EXTI15_10_IRQHandler(void)
02  {
03      u8 event;
04      /* enter interrupt */
05      rt_interrupt_enter();//RT-Thread中断嵌套
06      if(SET == EXTI_GetITStatus(EXTI_Line10)//OPEN_KEY &
07       ||SET == EXTI_GetITStatus(EXTI_Line11))// ALARM_KEY
08      {
09          event=get_key_event();
10          if(event == EVENT_ALARM)//语音呼叫请求
11              butn_cmd=BUTN_CMD_VOICE_RQST;
```

```
12          else if(event == EVENT_OPEN)//防破坏拆除报警
13              butn_cmd=BUTN_CMD_SET_ALARM;
14          if(server_mode==SERVER_MODE_NAT)//打洞服务器
15          {
16              butn_cmd=BUTN_CMD_UDP_RQST;//先打洞
17              next_cmd=butn_cmd;//后连接目标服务器
18          }
19          EXTI_ClearITPendingBit(EXTI_Line11|EXTI_Line10);
20      }
21      /* leave interrupt */
22      rt_interrupt_leave();//RT-Thread中断嵌套
23  }
```

获取按键外部中断对应的事件类型：

```
01 u8 key_get_event()
02 {
03      u32 open,alarm;
04      open=GPIO_ReadInputDataBit(GPIOB,OPEN_KEY);
05      alarm=GPIO_ReadInputDataBit(GPIOB,ALARM_KEY);
06      if(!open)
07          return EVENT_ALARM;//语音呼叫按键被按下
08      else if(!alarm)
09          return EVENT_OPEN;//硬件放破坏拆除按键被断开
10      else//以上两种情况都不是，
11          return EVENT_TIMER;//那么肯定是被定时唤醒的
12  }
```

4.1.6 语音输入驱动程序

配置语音输入对应的IO引脚为模拟输入模式，配置ADC的采样分辨率为8 bit，采样速率为8 KSPS，为ADC启动DMA（Direct Memory Access, 直接存储访问）传输，采集到的语音数据流将被DMA交替地搬运到两个缓冲区：

```
01 void audio_input_init(void)
```

```
02 {
03     GPIO_InitTypeDef GPIO_InitStructure;
04     NVIC_InitTypeDef NVIC_InitStructure;
05     TIM_TimeBaseInitTypeDef  TIM_TimeBaseStructure;
06     ADC_InitTypeDef ADC_InitStructure;
07     DMA_InitTypeDef DMA_InitStructure;
08
09     /* Enable peripheral clocks */
10     /* Enable DMA clock */
11     RCC_AHBPeriphClockCmd(RCC_AHBPeriph_DMA1, ENABLE);
12     RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3, ENABLE);
13     /* Enable ADC1 and GPIOA clock */
14     RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1 | RCC_APB2Periph_GPIOA,
ENABLE);
15     RCC_ADCCLKConfig(RCC_PCLK2_Div8);//72M/8=9M,ADCCLK最大不超过14M
16
17     /* GPIO configuration */
18     //配置IO为模拟输入模式;
19     /* Time base configuration */
20     //通过配置TIM时间基准来配置ADC采样率为8 KSPS;
21     //配置DMA1_CH1 NVIC优先级;
22     /* DMA channel1 configuration */
23     //DMA传输相关配置;
24     /* Enable DMA channel1 */
25     DMA_Cmd(DMA1_Channel1, ENABLE);
26     /* ADC1 configuration */
27     //ADC相关配置：时钟、转换时间、分辨率、数据对齐方式等;
28     //使能定时器、ADC和DMA
29 }
```

4.1.7 语音输出驱动程序

配置语音输入对应的IO引脚为模拟输入模式，配置DAC的采样分辨率为8 bit，转换速率为8 KSPS，为DAC启动DMA传输，从UDP服务端接收到的语音数据流将

被交替地存储到两个缓冲区，DMA则会按转换速率依次启动DAC将数据流转换成对应的模拟语音信号输出：

```
01 void audio_output_init(void)
02 {
03     GPIO_InitTypeDef GPIO_InitStructure;
04     TIM_TimeBaseInitTypeDef  TIM_TimeBaseStructure;
05     NVIC_InitTypeDef NVIC_InitStructure;
06     DMA_InitTypeDef DMA_InitStructure;
07     DAC_InitTypeDef DAC_InitStructure;
08
09     //使能相关外设的时钟：GPIO/TIM/DAC/DMA;
10     //配置音频输出对应IO引脚为模拟输出模式;
11     //通过配置TIM配置DAC的转换速率为8 KSPS;
12     //DAC初始化：转换时间、分辨率、数据对齐方式等;
13     //配置DMA2_CH4 NVIC中断优先级;
14     //为DAC配置DMA传输条件：源地址、目的地址、位宽等;
15     //使能TIM、DAC、DMA;
16 }
```

语音通话建立后开启语音输入输出功能：

```
01 void audio_on()
02 {
03     audio_input_init();//语音输入初始化
04     audio_output_init();//语音输出初始化
05     GPIO_SetBits(GPIOB,AUDIO_PA);//开启音频功放
06 }
```

语音通话结束后关闭语音输入输出功能：

```
01 void audio_off()
02 {
03     GPIO_ResetBits(GPIOB,AUDIO_PA);//关闭音频功放
04     ADC_Cmd(ADC1, DISABLE);//关闭ADC1
05     ADC_DMACmd(ADC1, DISABLE);//关闭ADC1的DMA功能
```

```
06      DMA_Cmd(DMA1_Channel1,DISABLE);//关闭DMA1第1通道
07      DMA_Cmd(DMA2_Channel4,DISABLE);//关闭DMA2第4通道
08      //DAC_Cmd(DAC_Channel_1,DISABLE);//关闭DAC第1通道
09      DAC_Cmd(DAC_Channel_2,DISABLE);//关闭DAC第2通道
10      //DAC_DMACmd(DAC_Channel_1, DISABLE);//关闭DAC第1通道的DMA功能
11      DAC_DMACmd(DAC_Channel_2, DISABLE);//关闭DAC第2通道的DMA功能
12      TIM_Cmd(TIM2,DISABLE);//关闭ADC触发时钟
13      TIM_Cmd(TIM3,DISABLE);//关闭DAC触发时钟
14  }
```

4.1.8 WIFI模块驱动程序

由于WIFI模块驱动程序非常庞大，在此仅示意性地给出WIFI初始化顶层API的部分程序：

```
01  void wifi_init()
02  {
03      wifi_power_on();//使能WIFI模块
04      timerbase_config();//SDIO接口时基配置
05      sysflag.status=on_connect;
06      sysflag.waittime=160;
07      load_netcfg();//加载网络参数配置信息
08      mem_init();//LWIP内存初始化
09      //初始化marvell模块驱动,包括SDIO设备枚举，加载设备固件等操作
10      priv = init_marvell_driver();
11      //扫描wifi网，结果存放在priv->network数据域中，使用bss_descriptor结
构体描述
12      lbs_scan_worker(priv);
13      //rt_kprintf("scanning finish...\r\n");
14      psk_out = mem_malloc(32);//为结构体申请内存
15      marvel_assoc_network(priv,marvel_ssid,marvel_key,marvel_mode);//关
联WIFI网络
16      mem_free(psk_out);//释放内存
17  }
```

至此，嵌入式软件的设计与实现部分阐述完毕。

4.2 系统上位机软件的设计与实现

4.2.1 开发工具简介

病房语音呼叫系统的上位机软件拟采用Visual C#集成开发环境来设计和完成。Visual C#是美国Microsoft公司设计和推出的一款面向Windows操作系统的可视化编程工具，它不需要开发者编写大量的程序代码，常用的工具都是以控件的方式集成在开发包当中，当用户需要使用相关的控件时，只需要将控件拖到相对应的位置即可。Visual C#的最大优势是比较容易上手，开发者无论是刚刚接触编程的初级开发人员，还是资深的程序员开发者，都可以很快掌握Visual C#的开发方式和方法。使用Visual C#来开发基于接近开关的测试系统上位机软件使得系统的风险得到了大大的降低，还使得整个系统的开发效率得到了极大的提高，系统的维护和升级也变得相对简单和容易。

Visual C#的主要技术特点为：

1. 遵循面向对象的程序开发思想

   Visual C#循面向对象的程序开发思想其核心思想就是将复杂的系统设计分解成若干个具有独立功能且相对容易的对象集合。

2. 数据库交互能力十分强大

   Visual C#内部集成了多种数据访问控件，对于常规的ACCESS、SQL Server、Oracle等数据库都有相对应的控件，开发者在使用的时候只需要调用相关的控件即可完成对相关数据的读写等相关的操作。

3. 具有强大的事件驱动能力

   Visual C#具有强大的事件驱动能力，Visual C#是一种面向Windows操作系统的开发工具，所以它可以很好的响应Windows环境下的各种事件等操作。

4. 采用结构化的编程风格

   Visual C#常用的工具都是以控件的方式集成在开发包当中，当用户需要使用相关的控件时，只需要将控件拖到相对应的位置即可。Visual C#的最大优势是比较容易上手，开发者无论是刚刚接触编程的初级开发人员，还是资深的程序员开发者，都可以很快掌握Visual C#的开发方式和方法。

4.2.2 上位机软件的主要框架

医院病房语音呼叫系统在上位机软件设计过程中应该遵循系统兼容性好、开发周期短、操作简便等特点，故采用了C#作为开发语言。病房语音呼叫系统在上位机软件可以完成对测试系统多种工作参数的编程设置以及对测试数据的读取。

病房语音呼叫系统的上位机软件功能结构如图 4–3 所示，从下到上按功能分为通信接口层、设备驱动层、中间件和数据库、应用层、用户接口层。最底层为通信接口层，上位机与嵌入式系统通过TCP/IP协议进行数据通信。设备驱动层负责语音输入输出设备的初始化、数据采集和转换、语音输出、网络设备控制等。中间件负责应用层和设备层之间的信息交换，数据库与中间件并列在中间层，顶层和底层需要长期存储的数据需要存储在数据库中时，也由中间件负责控制。应用层是语音呼叫系统上位机软件的各个核心功能模块所在层，所有应用模块通过中间件访问数据库或者底层设备，并将信息输出到用户界面。

<table>
<tr><td colspan="7">图形化用户界面</td></tr>
<tr><td>普通病房模块</td><td>重症病房模块</td><td>产房监护模块</td><td>普通病房模块</td><td>系统管理模块</td><td>语音控制模块</td><td>信息查询模块</td></tr>
<tr><td colspan="7">中间件和数据库</td></tr>
<tr><td>语音输入</td><td colspan="5">设备驱动层</td><td>语音输出</td></tr>
<tr><td colspan="7">TCP/IP通信接口层</td></tr>
</table>

图 4–3 系统上位机软件的总体框架

4.2.3 通信接口层

通信接口层主要负责上位机系统与语音呼叫终端之间的数据交换和通信链路管理，同时负责语音的输入与输出。

4.2.3.1 通信协议

通信接口层首先应该规定好通信协议，通信双方按照通信协议进行数据的传输。由于RT-Thread已经对LwIP进行了移植，考虑到语音呼叫系统点（服务器）对面（大量的终端设备）的通信需求，采用UDP通信协议。由于UDP通信协议并不是面向连接的，协议层也不对数据传输的有效性进行检查，虽然这对语音通信数据来讲影响不大（语音数据如果偶尔发生丢包会表现出语音断续、卡顿，但并不会严重影响通话质量，除非因网络拥堵导致严重丢包），但是对一些关键数据（例

如设备配置信息）的传输时，有必要引入数据校验机制，接收方需要对接收到的数据进行校验，校验失败的数据予以丢弃。

通信数据包格式定义如表 4–1所示，一个完整的数据包由1字节包头（服务端为0xAA，客户端为0x55）、2字节包长度、1字节包类型、可变字节数的数据域和2字节校验码组成

表 4–1 通信数据包格式

组成	包头	包长度	包类型	数据域	校验码
字节数	1	2	1	长度可变	2
服务端	0xAA	LENGTH	TYPE	DATA	CHECKSUM
客户端	0x55	LENGTH	TYPE	DATA	CHECKSUM

4.2.3.2 程序设计

由于上位机代码量极大，限于这里仅简要介绍部分程序或给出代码框架，并不一一罗列。

启动对话框程序时初始化并创建UDP通信连接：

```
01 public partial class myUdpClient : Form
02 {
03     //创建一个Thread实例
04     private Thread thread1;
05     private Thread thread2;
06     //创建一个UdpClient实例
07     private UdpClient udpReceive;
08     private byte[] receive_bytes;
09     private byte[] send_bytes;
10     int trys;
11     //private DialogResult result;
12
13     public myUdpClient()
14     {
15         int i;
16         InitializeComponent();
17         CheckForIllegalCrossThreadCalls = false;
18         comboBox2.SelectedIndex = 0;
```

```
19          comboBox3.SelectedIndex = 0;
20          udpReceive = new UdpClient(60000);//新建UDP连接，端口60000
21          thread1 = new Thread(new ThreadStart(ReceiveMessage));
22          thread1.Start();//新建并启动UDP接收线程
23          button1.Text = "停止监听";
24          send_bytes = null;
25          comboBox1.Text = "没有待配置设备";
26          comboBox4.SelectedIndex = 0;
27          comboBox5.SelectedIndex = 0 ;
28      }
29  }
```

程序启动时创建UDP接收线程，单击发送数据或者配置终端设备等按钮时，创建UDP发送线程：

```
01 private void myUdpClient_Load(object sender, EventArgs e)
02 {
03     //程序启动时创建UDP接收线程
04     thread1 = new Thread(new ThreadStart(ReceiveMessage));
05     thread1.Start();
06 }
07
08 private void BtnSend_Click(object sender, EventArgs e)
09 {
10     //单击按钮时创建UDP发送线程
11     thread2 = new Thread(SendMessage);
12     thread2.Start();
13 }
```

以下程序通过按钮开启或关闭UDP端口监听功能：

```
01 private void button1_Click(object sender, EventArgs e)
02 {
03     if (button1.Text == "开始监听")
04     {
```

```
05          udpReceive = new UdpClient(60000);
06          thread1 = new Thread(new ThreadStart(ReceiveMessage));
07          thread1.Start();
08          button1.Text = "停止监听";
09      }
10      else if (button1.Text == "停止监听")
11      {
12          //关闭UDP连接
13          udpReceive.Close();
14          //终止UDP接收线程
15          thread1.Abort();
16          button1.Text = "开始监听";
17      }
18  }
```

4.2.4 数据库管理层

数据库管理层主要负责语音呼叫终端的参数、病房信息、病员信息、用药信息等数据的存储，以及以上信息的检索与重定位。

SQL提供了非常丰富的语句对数据库进行查询、排序、增加、删除等操作，因此数据库管理层的代码量也非常庞大，限于篇幅，这里仅给出基本SQL操作语句的C#代码作为示例[23]。

```
01 public class SQLHelper
02 {
03      //新建SQLHelper类（支持基本QL语句）
04      private SqlConnection sqlCon = null;
05      private SqlCommand cmd = null;
06      private SqlDataReader sdr = null;
07      public SQLHelper()
08      {
09          sqlCon = new SqlConnection(ConfigurationManager .ConnectionStrings["Conn
10      }
11      /// <summary>
```

```
12      /// 打开数据库连接
13      /// </summary>
14      /// <returns></returns>
15      private SqlConnection GetCon()
16      {
17          if (sqlCon.State==ConnectionState.Closed)
18          {
19              sqlCon.Open();
20          }
21          return sqlCon;
22      }
23      /// <summary>
24      /// 执行不带参数的增删改sql语句或存储过程
25      /// </summary>
26      /// <param name="cmdText">增删改sql语句或存储过程</param>
27      /// <param name="ct">命令类型</param>
28      /// <returns></returns>
29      public int ExecuteNonQuery(string cmdText, CommandType ct)
30      {
31          int rex;
32          try
33          {
34              SqlCommand sqlcom = new SqlCommand(cmdText,GetCon());
35              sqlcom.CommandType = ct;
36              rex =sqlcom.ExecuteNonQuery();
37          }
38          catch (Exception ex)
39          {
40
41              throw ex;
42          }
43          finally
44          {
```

```
45              if (sqlCon.State==ConnectionState.Open)
46              {
47                  sqlCon.Close();
48              }
49          }
50          return rex;
51      }
52      /// <summary>
53      ///  执行带参数的增删改SQL语句或存储过程
54      /// </summary>
55      /// <param name="cmdText">增删改SQL语句或存储过程</param>
56      /// <param name="ct">命令类型</param>
57      /// <returns></returns>
58      public int ExecuteNonQuery(string cmdText, SqlParameter[] paras,
CommandType ct)
59      {
60          int res;
61          using (cmd = new SqlCommand(cmdText, GetCon()))
62          {
63              cmd.CommandType = ct;
64              cmd.Parameters.AddRange(paras);
65              res = cmd.ExecuteNonQuery();
66          }
67          return res;
68      }
69      /// <summary>
70      ///  执行带参数的查询SQL语句或存储过程
71      /// </summary>
72      /// <param name="cmdText">查询SQL语句或存储过程</param>
73      /// <param name="paras">参数集合</param>
74      /// <param name="ct">命令类型</param>
75      /// <returns></returns>
76      public DataTable ExecuteQuery(string cmdText, SqlParameter[] paras,
```

```
CommandType ct)
77      {
78          DataTable dt = new DataTable();
79          cmd = new SqlCommand(cmdText,GetCon());
80          cmd.CommandType = ct;
81          cmd.Parameters.AddRange(paras);
82          using (sdr = cmd.ExecuteReader(CommandBehavior.CloseConnection))
83          {
84              dt.Load(sdr);
85          }
86          return dt;
87      }
88      /// <summary>
89      /// 执行不带参数的查询SQL语句或存储过程
90      /// </summary>
91      /// <param name="cmdText">查询SQL语句或存储过程</param>
92      /// <param name="ct">命令类型</param>
93      /// <returns></returns>
94      public DataTable ExecuteQuery(string cmdText, CommandType ct)
95      {
96          DataTable dt = new DataTable();
97          cmd = new SqlCommand(cmdText, GetCon());
98          cmd.CommandType = ct;
99          using (sdr = cmd.ExecuteReader(CommandBehavior.CloseConnection))
100          {
101              dt.Load(sdr);
102          }
103          return dt;
104      }
105 }
```

4.2.5 图形化用户界面

图形化用户界面是提供给用户的人机接口，负责接收用户的输入操作，并将

相关信息作为结果反馈给用户。图形化用户界面应该做得极度简洁，一目了然，避免太过繁杂，否则反而不便于用户操作。基于这一准则设计的用户界面如图 4–4 所示。

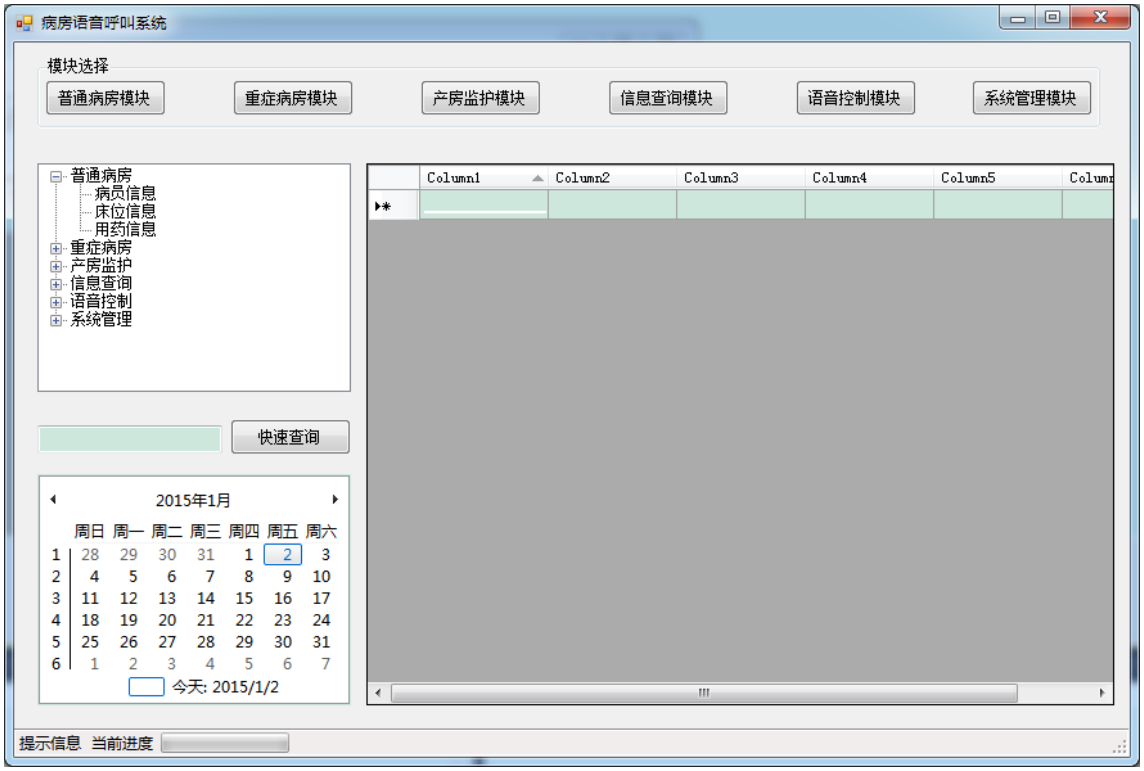


图 4–4 图形化用户界面

用户界面为用户提供了每一个主功能模块的快捷按钮，并额外添加了一个快捷搜索按钮，用户可以通过搜索框输入关键词快速地进行信息检索。左侧以树状图显示了各个功能模块的层次信息，用户也可以通过展开功能树直接点击相应功能。考虑到各项工作的开展、记录和管理都离不开时间，特地在主界面添加了一个日历月视图。

4.2.5.1 普通病房模块

普通病房模块是对普通的医院住院病人进行监护和管理的功能模块，主要针对普通病房进行巡检信息、病人用药信息以及联系方式等基本信息的管理，该模块可以及时回应病人的呼叫请求。普通病房模块的功能框图见图 4–5。

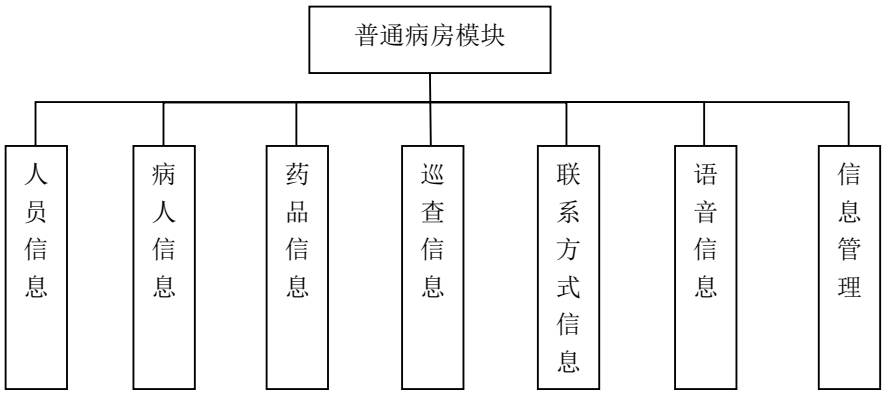


图 4–5 普通病房模块功能框图

4.2.5.2 重症病房模块

重症监护病房模块是本系统重要的功能模块之一，也是医院住院病房重点关注的地方，该模块分为病人信息、床位信息、用药信息等。通过重症病房模块，医护人员可以随时监护重症病人的情况并及时响应病人或病人家属的语音呼叫。重症病房模块的功能框图见图 4–6。

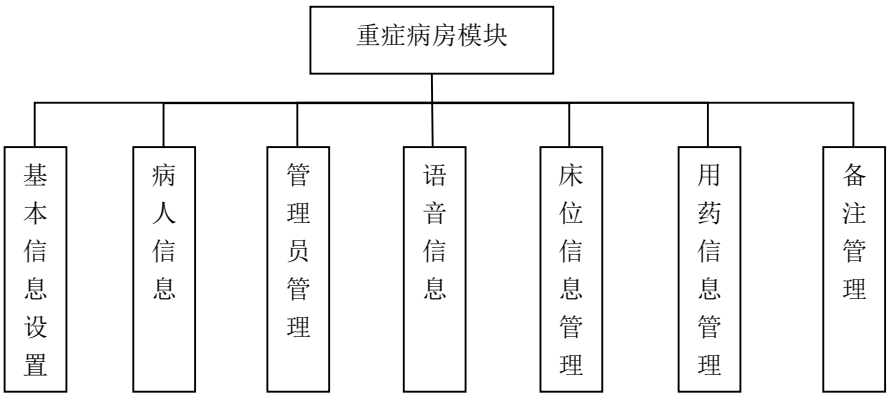


图 4–6 重症病房模块功能框图

4.2.5.3 产房监护模块

产房监护模块是语音系统的又一个重要模功能块，该模块不仅负责维护产妇的信息，同时也负责及时地关注婴儿的信息，对产妇信息和婴儿信息进行双重的维护。产房监护模块的具体功能框图如图 4–7 所示。

4.2.5.4 系统管理模块

系统管理模块是对整个系统的稳定运行以及数据进行管理的重要功能模块，主要包对系统数据、语音参数、病人信息、系统日志以及用户权限申请等进行监视和管理。具体功能模块图如图 4–8 所示。

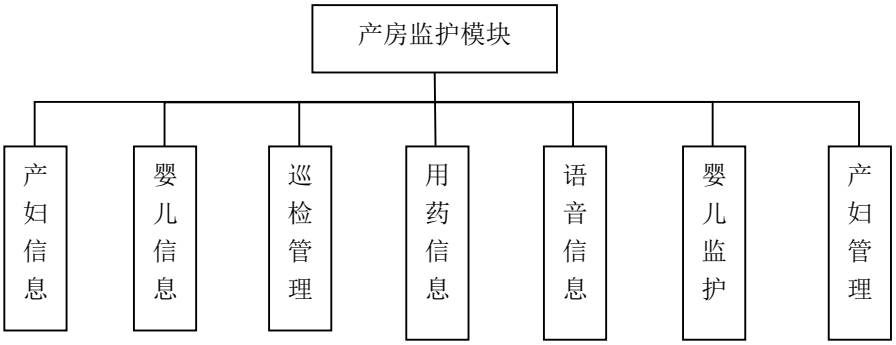


图 4–7 产房监护模块功能框图

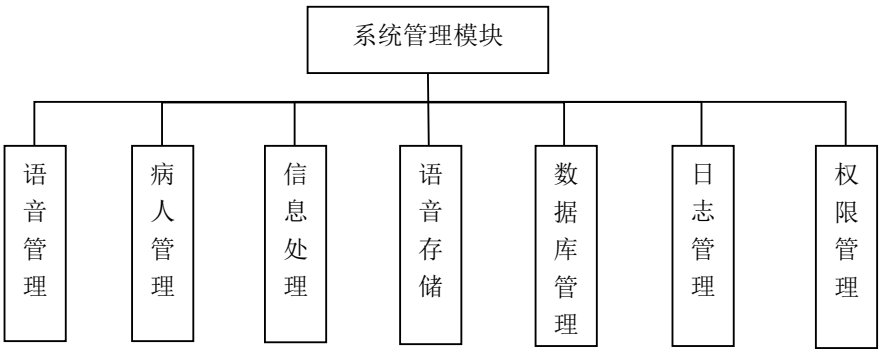


图 4–8 系统管理模块功能框图

4.2.5.5 语音控制模块

语音控制模块主要负责对语音通信进行控制和参数配置，比如没有病人的病房或者没有病人使用的床位，语音控制模块可以控制对应病房或者床位的语音呼叫设备关闭语音模块。用户也可以通过语音控制模块确认是哪个床位发起的语音呼叫，可以对语音进行存储。所有语音呼叫设备的设备号、语音采样率、服务器IP和端口号等通信参数配置也是通过语音控制模块进行的。因此，语音控制模块是整个系统语音控制中枢，具体的功能模块图如图 4–9 所示。

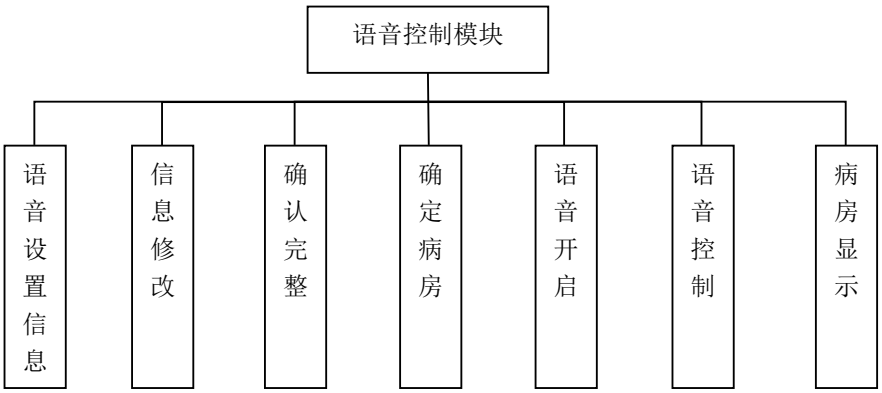


图 4–9 语音控制模块功能框图

4.2.5.6 信息查询模块

为方便用户以任意关键词进行信息查询，系统还提供了信息查询功能。用户可以对系统中的各种信息可以进行查询，例如设备编号、设备状态、病人信息、语音信息等。同时信息查询模块还可以以报表的形式导出和打印相关的查询结果。具体功能如图 4–10所示。

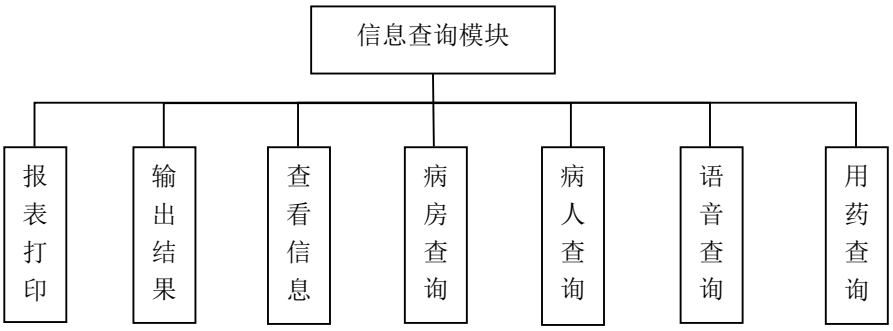


图 4–10 信息查询模块功能框图

4.3 本章小结

本章主要完成了病房语音呼叫系统软件部分的详细设计与实现。首先对系统软件设计所应遵循的总体原则进行了介绍和说明，再以逐步推进的方式分别对系统嵌入式软件和上位机软件进行了设计阐述。嵌入式软件部分对操作系统、数据采集及处理等、UDP通信等进行了详细的论述。上位机软件部分对病房语音呼叫系统的主要框架和各个功能模块进行了阐述。

第五章 系统的部署与联合测试

5.1 系统软件安装

系统部署和测试之前，首先应该给目标计算机安装系统上位机软件，因为后续的设备参数配置、联合调试都倚赖系统的上位机软件。安装方法和步骤跟常规软件的安装完全相同，在此不做赘述。

5.2 系统网络部署

系统部署时需要给系统终端提供外部 +5 V 直流电源，因此首先需要铺设统一供电的电缆或者给每个终端设备提供一个单独的小型化AC220V/DC+5V电源适配器。

由于系统使用Wi-Fi作为网络接入方式，所以省去了大量铺设通信电缆的成本和人力，系统网络部署时只需要确保终端设备的工作环境（病房）有良好的Wi-Fi覆盖，并保证终端设备天线连接可靠，无金属障碍物遮挡即可。Wi-Fi接入点（AP，通常为路由器）的设置参考 5.3 节。

5.3 设备参数配置

系统终端设备供电和Wi-Fi覆盖都满足后，便可对设备进行参数配置，以便设备顺利接入网络。用户可以使用串口或者Wi-Fi对设备进行参数配置。由于初始情况下，设备只能按默认参数进行Wi-Fi接入，若果使用Wi-Fi进行配置，那么应将接入点按照设备默认参数进行设置，即使用“router_ssid”作为路由器的SSID，并且使用默认的用户名“admin”和密码“admin”，终端设备才可以自动进行连接，否则应该通过串口将Wi-Fi用户名和密码下发给终端设备，随后设备会保存用户名和密码、自动重启并以指定的用户名和密码接入Wi-Fi。此时，运行上位机软件，选择【设备管理】→【参数配置】，弹出对话窗口如图 5–1 所示。

下发给中断设备的参数还包括设备的IP获取方式：DHCP或静态IP，DHCP由路由器动态分配IP，IP地址不固定，如果希望每个终端设备有固定IP，则应选择静态IP方式，此时还应该为其指定IP地址、子网掩码、默认网关和DNS地址。设备的参数配置还包括给设备指定设备名称、设备定时向服务器汇报自身情况（例如电源电量、有无系统异常等）的时间间隔、指定服务器IP（对于固定IP服务器）或服务器URL（对于使用域名解析的服务器）、服务器端口和服务器名称。这些参

图 5–1 设备参数配置对话框

数配置完成后，设备首先将参数存储到FLASH中并向配置端返回参数配置成功的确认包，随后自动重启并按新的参数初始化系统，自动接入Wi-Fi并连接服务器，定时向服务器发送心跳汇报包。

参数配置完毕后，应观察设备的指示灯是否正常指示系统的工作状态，是否能够正常连接服务器并发送汇报包，如图 5–2。若不正常应该检查原因或者重新进行参数配置。待以上项目都正常以后按下语音呼叫按钮，服务器端应该收到语音呼叫请求，并正确显示呼叫设备的名称、IP地址等相关信息，如图 5–3 所示。确认以上信息均为正确后点击确认接受呼叫请求，测试能否正常地建立语音连接和通话质量。最后结束语音通话，检查终端设备是否正常关闭了语音输出，设备指示灯恢复到待机状态并定时向服务器汇报自身状态。

参数配置到此完毕。

5.4 系统联合测试

系统软件安装、系统网络部署、设备参数配置都完成后便可开始系统联合测试，此项测试只旨在检验系统在实际运行环境中多设备共存、服务器多业务并发执行、多客户端并发访问、数据库频繁大量的数据操作等情况下的稳定性。系统联合测试按如下方法进行：

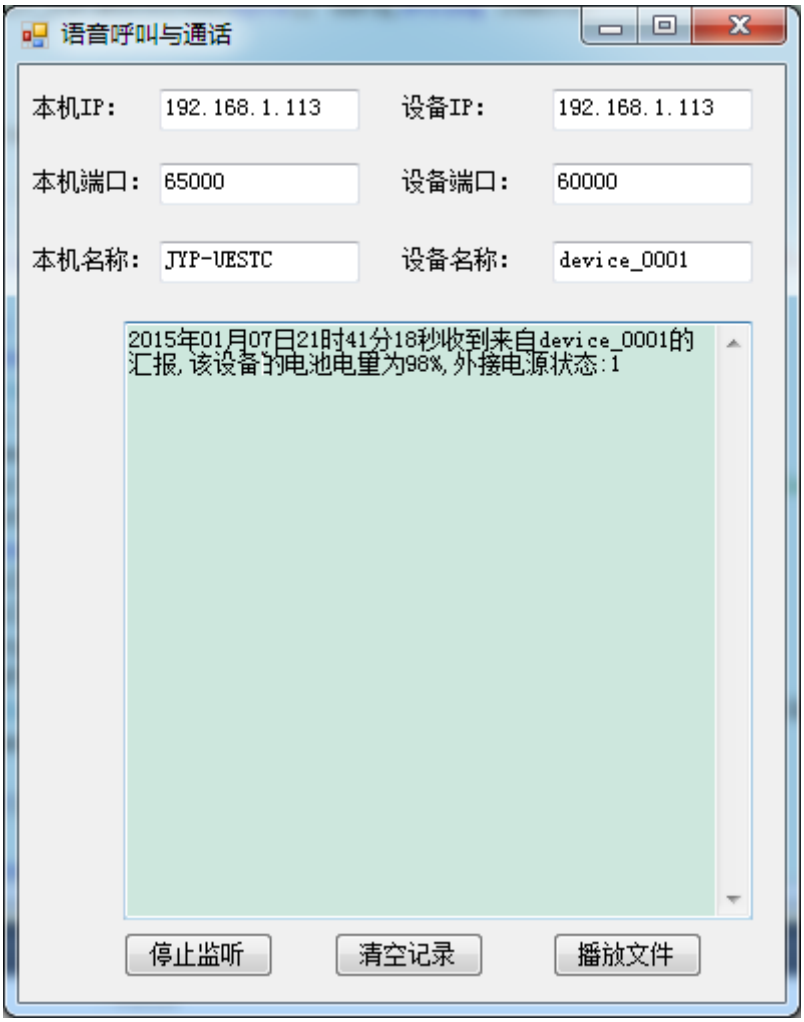


图 5–2 服务器收到设备状态汇报

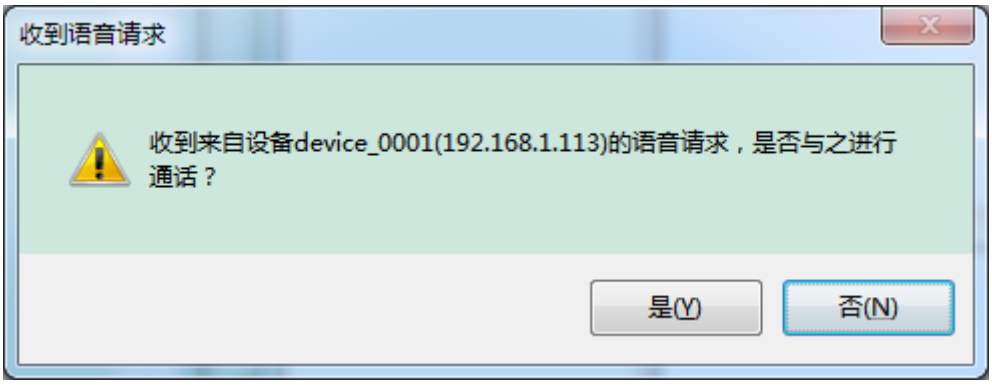


图 5–3 服务器收到语音呼叫请求

1. 根据实际需求选择部署的终端设备数量，这决定了系统网络的规模（为保证测试的有效性，建议设备节点数在100以上），完成设备参数配置和网络部署，并保证网络参数分配合理，同一局域网中绝对不允许有两个或两个以上相同设备IP和设备名称。
2. 开启无线接入点并保证网络畅通，运行系统上位机软件并启动系统服务器，最后给所有的终端设备供电。此时服务器应该能收到每一个终端设备周期性的心跳汇报包，对 24 h（或更长时间）内每一个设备的汇报包进行统计，考察是否有长时的“心跳停止”。
3. 尽可能同时按下多个终端设备的语音呼叫按钮，考察服务端是否能同时响应这些设备的语音呼叫请求，重复测试，统计并发呼叫终端设备和服务器实际响应的语音呼叫请求情况，考察是否有无语音呼叫请求被丢失。
4. 重复发起语音呼叫请求并建立语音连接（建议每个终端设备至少重复100次），测试语音呼叫结束后终端设备是否正常关闭了语音输出，设备指示灯是否恢复到待机状态并定时向服务器汇报自身状态。
5. 其他上位机界面的操作和病房信息管理等功能测试在短时间内只能做基本功能测试，稳定性方面只能在系统的实际运行过程中注意观察，一旦有故障发生应该记录故障现象和诱因以便后续故障定位和修复。

5.5 本章小结

本章主要介绍了系统软件安装、系统网络部署、设备参数配置以及系统联合测试的详细方法与步骤，给出了样例测试结果。经过实际测试，系统的功能完整正常，系统稳定性良好，功能和性能均达到了设计要求。

第六章 全文总结与展望

6.1 本文的主要贡献

本论文主要论述了医院住院病房语音呼叫系统的研究与开发工作，从系统目前的运行效果和测试结果来看，系统的各个功能模块都达到了预期的要求，实现了病房语音呼叫系统。主要完成了以下工作：

1. 详细分析了本系统的需求，针对系统终端设备嵌入式硬件、嵌入式软件和上位机软件分别制定了总体方案框架，编写了详细的需求分析文档。
2. 完成了系统终端设备的嵌入式硬件电路设计，给出了详细的设计思路和电路图，并完成了制作和调试。
3. 完成了系统终端设备的嵌入式软件设计，采用了RT-Thread嵌入式实时系统，保证了终端设备的稳定性和实时性。
4. 完成了系统上位机软件的设计与测试，使用SQL server 2005建立起系统需要的数据库和数据表，配合JSP所提供的验证的内建数据表，并利用了ASP所内建的 SqlMembershipProvider 和 RoleProvider 模型来简化对成员与角色的管理。
5. 系统总体上采用B/S系统体系结构。采用三层架构模式，充分利用网络的自身优势，将系统部署网络上进行应用；设置专门的应用服务器和数据服务器保证系统的正常运行，保证系统大规模使用时能很好的提供服务；重新设计的三层架构模式对系统的数据传输和数据的处理有很大的提高，提升了医院住院语音系统的模式。
6. 浏览器模式可以更加方便的进行访问，减少了系统的维护费用，方便了系统的推广以及使用。在数据传输和数据库共享上，相比客户端模式有着较大的改善。系统数据的安全性以及语音呼叫工能都得到了很大的改善，对系统的数据分析以及病房管理等功能都进行了改进。

6.2 存在的不足

本文提出的病房语音呼叫系统实现了各个功能模块以及系统的数据统计与分析和数据输出功能，但是对系统的参数设置以及系统的数据分析预测功能还有待完善，对不同的数据分析还有待提高。

6.3 展望

随着单片机技术和通信技术的发展，病房呼叫系统的功能也越来越多，越来越完善，朝着人性化、智能化、网络化的方向发展。

虽然本文设计实现的病房语音呼叫系统仍然存在一定的不足，相信经过更多的优化和完善，以及其他学者的进一步探索，医院病房的语音呼叫和管理系统并将越来越完善而智能化。相信在不久，病房呼叫系统会与医院信息系统联合起来，在发挥更重要的作用。

致 谢

本文历时一年有余终于完成，欣喜之余，作者要首先感谢徐军和姜宝钧两位指导老师，他们为人随和热情，治学严谨细心。当时选定这个题目看重了它的创新性和实用性，虽然一开始以为不是太难，但是深入后发现其实并没有想象的那么容易。两位指导老师始终认真负责地给予作者深刻而细致地指导，帮助作者开拓研究思路，精心点拨、热忱鼓励，本文最终能够完成很大程度上得益于两位导师的鼎力相助。

同时，作者要衷心感谢的父母和其他亲朋好友对作者的关心、支持和理解，没有他们对作者的关心、鼓励和支持，作者无法完成现在的学业。

最后，感谢曾经教育和帮助过作者的所有老师。衷心地感谢电子科技大学对本人的培养，感谢为评阅本论文而付出宝贵时间和辛勤劳动的专家和教授们!

参考文献

[1] 沈凯. 病房呼叫系统的开发与设计[D]. 江苏: 江苏大学, 2010.

[2] 杨进宝. 无线医疗呼叫系统的设计[D]. 湖南: 湖南师范大学, 2009.

[3] 宋昆飞. 电生理检测自动报警智能医院护理呼叫系统设计[D]. 上海: 同济大学, 2006.

[4] 周丽红. 基于DTMF的医院护理呼叫对讲系统的设计与实现[D]. 无锡: 江南大学, 2008.

[5] 朱艳华, 等. 基于PL3105的病房呼叫系统设计[J]. 北京石油化工学院学报, 2009.

[6] 潘绍明, 等. 基于信号叠加和无线电的病房呼叫系统设计与实现[J]. 电子技术应用, 2011.

[7] 梅彬运, 等. 基于nRF401芯片的医院无线护理呼叫系统的设计[J]. 现代电子技术, 2007.

[8] 吴敏. 基于嵌入式的家庭网关控制平台的研究与设计[D]. 武汉: 武汉理工大学, 2007.

[9] 周磊. 基于蓝牙和GPS技术的车辆管控系统关键技术的研究[D]. 南京: 东南大学, 2007.

[10] 宋丹. 基于eCos的嵌入式GIS系统的研究与实现[D]. 成都: 电子科技大学, 2008.

[11] 李婷婷. 薄木生产流程管理系统设计与实现[D]. 济南: 山东大学, 2009.

[12] 刘哲. 网络游戏“未来之王”的研究与设计[D]. 厦门: 厦门大学, 2013.

[13] 马慧娟. 煤矿井下水情综合监控系统管理软件的开发与设计[D]. 太原: 太原理工大学, 2010.

[14] 陈红波. 大型桥梁健康监测分布式数据采集系统的设计与实现[D]. 兰州: 兰州理工大学, 2006.

[15] 沈丰年. 无线电监测系统模型分析及一种新的查找信号的方法研究[D]. 重庆: 重庆大学, 2009.

[16] 魏大海. M地区在用电梯动态监管系统研究[D]. 济南: 山东大学, 2011.

[17] 崔振科. 基于ARM的多功能汽车行驶记录仪的研制[D]. 青岛: 青岛科技大学, 2014.

[18] 李宗方. 基于Cortex内核处理器的USB高速数据采集技术研究[D]. 北京: 北京工业大学, 2009.

[19] 邱祎. 嵌入式实时操作系统RT-Thread的设计与实现[D]. 成都: 电子科技大学, 2007.

[20] 金鑫. 燃气锅炉控制系统的研究与设计[D]. 益阳: 湖南大学, 2014.

[21] 杨泽明. 智慧照明系统设计与实现[D]. 成都: 电子科技大学, 2014.

[22] 李明. 全自动生化分析仪测控系统设计[D]. 南京: 南京理工大学, 2014.

[23] 张臻. 西南季节性干旱区农业资源与环境要素数据库设计与应用[D]. 重庆: 西南大学, 2011.

[24] 魏莱. 本文为初稿，暂未列出实际参考文献，后续会补充[M]. 成都: 电子科技大学出版社, 2016.

[25] 吴援明, 唐军. 模拟电路分析与设计基础[M]. 北京: 科学出版社, 2006.

[26] 张朝. 基于嵌入式及GPRS技术的无线监测终端研究[D]. 西安: 西北工业大学, 2007.

[27] 罗莹. 基于.NET技术的中小企业生产执行管理系统设计与实现[D]. 厦门: 厦门大学, 2012.

[28] 梁志勇. 基于LM3S8962处理器煤矿监测系统的研究[D]. 北京: 北京交通大学, 2010.

[29] 王家君, 崔舜, 林晨光,等. 一种超细钼粉或超细钨粉表面包覆金属铜的制备方法：中国，CN200710177119.6[P/OL]. 北京: 北京有色金属研究总院, 2007.11.09. http://www.google.com/patents/CN101428345A?cl=zh.

[30] 郭广颂, 等. 基于单片机的无线病房呼叫系统设计[J]. 电子技术应用, 2012.

[31] 王广德, 等. 医用局域有线呼叫系统的硬件设计[J]. 吉林师范大学学报（自然科学版）, 2010.

[32] K.-L. Ngo-Wah, J. Goel, Y.-C. Chou, et al. A V-band eight-way combined solid-state power amplifier with 12.8 Watt output power[C]//Microwave Symposium Digest, 2005 IEEE MTT-S International.[S.l.]: IEEE Publishing, 2005:4 pp.1371–1374.

[33] I. ANSYS. ANSYS FLUENT User's Guide[M]. Canonsburg: ANSYS, Inc., 2011.

[34] I. ANSYS. ANSYS Workbench User's Guide[M]. Canonsburg: ANSYS, Inc., 2009.

[35] I. ANSYS. ANSYS Icepak Tutorials[M]. Canonsburg: ANSYS, Inc., 2012.

[36] I. ANSYS. ANSYS Icepak User's Guide[M]. Canonsburg: ANSYS, Inc., 2012.

[37] G. Eggers, R. Kohl. 2 GHz Bandwidth Predistortion Linearizer for Microwave Power Amplifiers at Ku-Band[C]//24th European Microwave Conference, 1994. Cannes: IEEE Publishing, 1994，2:1501–1505.

[38] K. Yamauchi, M. Nakayama, Y. Ikeda, et al. An 18 GHz-band MMIC linearizer using a parallel diode with a bias feed resistance and a parallel capacitor[C]//Microwave Symposium Digest, 2000 IEEE MTT-S International. Boston: IEEE Publishing, 2000，3:1507–1510.

[39] K. Yamauchi, H. Noto, S. Ishizaka, et al. Series anti-parallel diode linearizer for class-B power amplifiers with a gain expansion[C]//Asia-Pacific Microwave Conference, 2006. Yokohama: IEEE Publishing, 2006:883–886.

[40] A. Katz, S. Moochalla, J. Klatskin. Passive FET MMIC linearizers for C, X, and Ku-band satellite applications[C]//Microwave Symposium Digest, 1993 IEEE MTT-S International. Atlanta: IEEE Publishing, 1993，1:353–356.

[41] M. T. Hickson, D. Paul, P. Gardner, et al. High Efficiency Feedforward Linearizers[C]//24th European Microwave Conference, 1994. Cannes: IEEE Publishing, 1994，1:819–824.

[42] L. W. Epp, A. R. Khan, D. J. Hoppe, et al. 24-Way Radial Power Combiner/Divider for 31 to 36 GHz[R]. Pasadena: NASA's Jet Propulsion Laboratory, 2008. http://www.techbriefs.com/component/content/article/1264-ntb/tech-briefs/electronics-and-computers/2784.

[43] J. Nevarez, G. Herskowitz. Output Power and Loss Analysis of 2^n Injection-Locked Oscillators Combined through an Ideal and Symmetric Hybrid Combiner[J]. IEEE Transactions on Microwave Theory and Techniques, 1969, 17(1):2–10.

[44] L. Epp, D. Hoppe, A. Khan, et al. A High-Power Ka-Band (31-36 GHz) Solid-State Amplifier Based on Low-Loss Corporate Waveguide Combining[J]. IEEE Transactions on Microwave Theory and Techniques, 2008, 56(8):1899–1908.

[45] Y. Wang, Y. Fu. A broadband waveguide power splitter and combiner using in spatial power combining amplifier[C]//2011 International Conference on Electronics, Communications and Control (ICECC). Ningbo: IEEE Publishing, 2011:4074–4077.

[46] R. Kline. Harold Black and the negative-feedback amplifier[J]. IEEE Control Systems, 1993, 13(4):82–85.

[47] H. Black. Inventing the negative feedback amplifier: Six years of persistent search helped the author conceive the idea “in a flash” aboard the old Lackawanna Ferry[J]. IEEE Spectrum, 1977, 14(12):55–60.

[48] H. S. Black. Stabilized Feed-Back Amplifiers[J]. Transactions of the American Institute of Electrical Engineers, 1934, 53(1):114–120.

[49] 互联网资源. ARM课程考试相关知识点[EB/OL]. http://blog.sina.com.cn/s/blog_b8ca1232010188v1.html.

[50] H. S. Black. Wave translation system：US，US2102671[P/OL].[S.l.]: Google Patents. http://www.google.com/patents/US2102671.

附录 A RT-Thread线程API相关源码

```
01 /*
02  * File      : thread.c
03  * This file is part of RT-Thread RTOS
04  * COPYRIGHT (C) 2006 - 2012, RT-Thread Development Team
05  *
06  * The license and distribution terms for this file may be
07  * found in the file LICENSE in this distribution or at
08  * http://www.rt-thread.org/license/LICENSE
09  *
10  * Change Logs:
11  * Date           Author       Notes
12  * 2006-03-28     Bernard      first version
13  * 2006-04-29     Bernard      implement thread timer
14  * 2006-04-30     Bernard      added THREAD_DEBUG
15  * 2006-05-27     Bernard      fixed the rt_thread_yield bug
16  * 2006-06-03     Bernard      fixed the thread timer init bug
17  * 2006-08-10     Bernard      fixed the timer bug in thread_sleep
18  * 2006-09-03     Bernard      changed rt_timer_delete to rt_timer_detach
19  * 2006-09-03     Bernard      implement rt_thread_detach
20  * 2008-02-16     Bernard      fixed the rt_thread_timeout bug
21  * 2010-03-21     Bernard      change the errno of rt_thread_delay/sleep
to
22  *                             RT_EOK.
23  * 2010-11-10     Bernard      add cleanup callback function in thread
exit.
24  * 2011-09-01     Bernard      fixed rt_thread_exit issue when the current
25  *                             thread preempted, which reported by
Jiaxing Lee.
26  * 2011-09-08     Bernard      fixed the scheduling issue in rt_thread_startup.
27  * 2012-12-29     Bernard      fixed compiling warning.
```

```
28  */
29
30 #include <rtthread.h>
31 #include <rthw.h>
32
33 extern rt_list_t rt_thread_priority_table[RT_THREAD_PRIORITY_MAX];
34 extern struct rt_thread *rt_current_thread;
35 extern rt_list_t rt_thread_defunct;
36
37 static void rt_thread_exit(void)
38 {
39     struct rt_thread *thread;
40     register rt_base_t level;
41
42     /* get current thread */
43     thread = rt_current_thread;
44
45     /* disable interrupt */
46     level = rt_hw_interrupt_disable();
47
48     /* remove from schedule */
49     rt_schedule_remove_thread(thread);
50     /* change stat */
51     thread->stat = RT_THREAD_CLOSE;
52
53     /* remove it from timer list */
54     rt_list_remove(&(thread->thread_timer.list));
55     rt_object_detach((rt_object_t)&(thread->thread_timer));
56
57     if ((rt_object_is_systemobject((rt_object_t)thread) == RT_TRUE) &&
58             thread->cleanup == RT_NULL)
59     {
60         rt_object_detach((rt_object_t)thread);
```

```
61     }
62     else
63     {
64         /* insert to defunct thread list */
65         rt_list_insert_after(&rt_thread_defunct, &(thread->tlist));
66     }
67 
68     /* enable interrupt */
69     rt_hw_interrupt_enable(level);
70 
71     /* switch to next task */
72     rt_schedule();
73 }
74 
75 static rt_err_t _rt_thread_init(struct rt_thread *thread,
76                                 const char       *name,
77                                 void (*entry)(void *parameter),
78                                 void             *parameter,
79                                 void             *stack_start,
80                                 rt_uint32_t       stack_size,
81                                 rt_uint8_t        priority,
82                                 rt_uint32_t       tick)
83 {
84     /* init thread list */
85     rt_list_init(&(thread->tlist));
86 
87     thread->entry = (void *)entry;
88     thread->parameter = parameter;
89 
90     /* stack init */
91     thread->stack_addr = stack_start;
92     thread->stack_size = (rt_uint16_t)stack_size;
93 
```

```
94      /* init thread stack */
95      rt_memset(thread->stack_addr, '#', thread->stack_size);
96      thread->sp = (void *)rt_hw_stack_init(thread->entry, thread->parameter,
97                                            (void *)((char *)thread->stack_addr
+ thread->stack_size - 4),
98                                            (void *)rt_thread_exit);
99
100     /* priority init */
101     RT_ASSERT(priority < RT_THREAD_PRIORITY_MAX);
102     thread->init_priority    = priority;
103     thread->current_priority = priority;
104
105     /* tick init */
106     thread->init_tick      = tick;
107     thread->remaining_tick = tick;
108
109     /* error and flags */
110     thread->error = RT_EOK;
111     thread->stat  = RT_THREAD_INIT;
112
113     /* initialize cleanup function and user data */
114     thread->cleanup   = 0;
115     thread->user_data = 0;
116
117     /* init thread timer */
118     rt_timer_init(&(thread->thread_timer),
119                   thread->name,
120                   rt_thread_timeout,
121                   thread,
122                   0,
123                   RT_TIMER_FLAG_ONE_SHOT);
124
125     return RT_EOK;
```

```
126 }
127
128 /**
129  * @addtogroup Thread
130  */
131
132 /*@{*/
133
134 /**
135  * This function will initialize a thread, normally it's used to initialize
a
136  * static thread object.
137  *
138  * @param thread the static thread object
139  * @param name the name of thread, which shall be unique
140  * @param entry the entry function of thread
141  * @param parameter the parameter of thread enter function
142  * @param stack_start the start address of thread stack
143  * @param stack_size the size of thread stack
144  * @param priority the priority of thread
145  * @param tick the time slice if there are same priority thread
146  *
147  * @return the operation status, RT_EOK on OK, -RT_ERROR on error
148  */
149 rt_err_t rt_thread_init(struct rt_thread *thread,
150                         const char       *name,
151                         void (*entry)(void *parameter),
152                         void             *parameter,
153                         void             *stack_start,
154                         rt_uint32_t       stack_size,
155                         rt_uint8_t        priority,
156                         rt_uint32_t       tick)
157 {
```

```
158     /* thread check */
159     RT_ASSERT(thread != RT_NULL);
160     RT_ASSERT(stack_start != RT_NULL);
161 
162     /* init thread object */
163     rt_object_init((rt_object_t)thread, RT_Object_Class_Thread, name);
164 
165     return _rt_thread_init(thread,
166                            name,
167                            entry,
168                            parameter,
169                            stack_start,
170                            stack_size,
171                            priority,
172                            tick);
173 }
174 RTM_EXPORT(rt_thread_init);
175 
176 /**
177  * This function will return self thread object
178  *
179  * @return the self thread object
180  */
181 rt_thread_t rt_thread_self(void)
182 {
183     return rt_current_thread;
184 }
185 RTM_EXPORT(rt_thread_self);
186 
187 /**
188  * This function will start a thread and put it to system ready queue
189  *
190  * @param thread the thread to be started
```

```
191  *
192  * @return the operation status, RT_EOK on OK, -RT_ERROR on error
193  */
194 rt_err_t rt_thread_startup(rt_thread_t thread)
195 {
196     /* thread check */
197     RT_ASSERT(thread != RT_NULL);
198     RT_ASSERT(thread->stat == RT_THREAD_INIT);
199
200     /* set current priority to init priority */
201     thread->current_priority = thread->init_priority;
202
203     /* calculate priority attribute */
204 #if RT_THREAD_PRIORITY_MAX > 32
205     thread->number      = thread->current_priority >> 3;            /*
5bit */
206     thread->number_mask = 1L << thread->number;
207     thread->high_mask   = 1L << (thread->current_priority & 0x07);
/* 3bit */
208 #else
209     thread->number_mask = 1L << thread->current_priority;
210 #endif
211
212     RT_DEBUG_LOG(RT_DEBUG_THREAD, ("startup a thread:%s with priority:%d\n",
213                                    thread->name, thread->init_priority));
214     /* change thread stat */
215     thread->stat = RT_THREAD_SUSPEND;
216     /* then resume it */
217     rt_thread_resume(thread);
218     if (rt_thread_self() != RT_NULL)
219     {
220         /* do a scheduling */
221         rt_schedule();
```

```
222     }
223
224     return RT_EOK;
225 }
226 RTM_EXPORT(rt_thread_startup);
227
228 /**
229  * This function will detach a thread. The thread object will be removed
from
230  * thread queue and detached/deleted from system object management.
231  *
232  * @param thread the thread to be deleted
233  *
234  * @return the operation status, RT_EOK on OK, -RT_ERROR on error
235  */
236 rt_err_t rt_thread_detach(rt_thread_t thread)
237 {
238     rt_base_t lock;
239
240     /* thread check */
241     RT_ASSERT(thread != RT_NULL);
242
243     /* remove from schedule */
244     rt_schedule_remove_thread(thread);
245
246     /* release thread timer */
247     rt_timer_detach(&(thread->thread_timer));
248
249     /* change stat */
250     thread->stat = RT_THREAD_CLOSE;
251
252     /* detach object */
253     rt_object_detach((rt_object_t)thread);
```

```
254
255     if (thread->cleanup != RT_NULL)
256     {
257         /* disable interrupt */
258         lock = rt_hw_interrupt_disable();
259
260         /* insert to defunct thread list */
261         rt_list_insert_after(&rt_thread_defunct, &(thread->tlist));
262
263         /* enable interrupt */
264         rt_hw_interrupt_enable(lock);
265     }
266
267     return RT_EOK;
268 }
269 RTM_EXPORT(rt_thread_detach);
270
271
272 #ifdef RT_USING_HEAP
273 /**
274  * This function will create a thread object and allocate thread object
memory
275  * and stack.
276  *
277  * @param name the name of thread, which shall be unique
278  * @param entry the entry function of thread
279  * @param parameter the parameter of thread enter function
280  * @param stack_size the size of thread stack
281  * @param priority the priority of thread
282  * @param tick the time slice if there are same priority thread
283  *
284  * @return the created thread object
285  */
```

```
286 rt_thread_t rt_thread_create(const char *name,
287                                 void (*entry)(void *parameter),
288                                 void       *parameter,
289                                 rt_uint32_t stack_size,
290                                 rt_uint8_t  priority,
291                                 rt_uint32_t tick)
292 {
293     struct rt_thread *thread;
294     void *stack_start;
295 
296     thread = (struct rt_thread *)rt_object_allocate(RT_Object_Class_Thread,
297               name);
298     if (thread == RT_NULL)
299         return RT_NULL;
300 
301     stack_start = (void *)rt_malloc(stack_size);
302     if (stack_start == RT_NULL)
303     {
304         /* allocate stack failure */
305         rt_object_delete((rt_object_t)thread);
306 
307         return RT_NULL;
308     }
309 
310     _rt_thread_init(thread,
311                     name,
312                     entry,
313                     parameter,
314                     stack_start,
315                     stack_size,
316                     priority,
317                     tick);
318 
```

```
319     return thread;
320 }
321 RTM_EXPORT(rt_thread_create);
322
323 /**
324  * This function will delete a thread. The thread object will be removed
from
325  * thread queue and detached/deleted from system object management.
326  *
327  * @param thread the thread to be deleted
328  *
329  * @return the operation status, RT_EOK on OK, -RT_ERROR on error
330  */
331 rt_err_t rt_thread_delete(rt_thread_t thread)
332 {
333     rt_base_t lock;
334
335     /* thread check */
336     RT_ASSERT(thread != RT_NULL);
337
338     /* remove from schedule */
339     rt_schedule_remove_thread(thread);
340
341     /* release thread timer */
342     rt_timer_detach(&(thread->thread_timer));
343
344     /* change stat */
345     thread->stat = RT_THREAD_CLOSE;
346
347     /* disable interrupt */
348     lock = rt_hw_interrupt_disable();
349
350     /* insert to defunct thread list */
```

```
351     rt_list_insert_after(&rt_thread_defunct, &(thread->tlist));
352 
353     /* enable interrupt */
354     rt_hw_interrupt_enable(lock);
355 
356     return RT_EOK;
357 }
358 RTM_EXPORT(rt_thread_delete);
359 #endif
360 
361 /**
362  * This function will let current thread yield processor, and scheduler
will
363  * choose a highest thread to run. After yield processor, the current
thread
364  * is still in READY state.
365  *
366  * @return RT_EOK
367  */
368 rt_err_t rt_thread_yield(void)
369 {
370     register rt_base_t level;
371     struct rt_thread *thread;
372 
373     /* disable interrupt */
374     level = rt_hw_interrupt_disable();
375 
376     /* set to current thread */
377     thread = rt_current_thread;
378 
379     /* if the thread stat is READY and on ready queue list */
380     if (thread->stat == RT_THREAD_READY &&
381             thread->tlist.next != thread->tlist.prev)
```

```
382     {
383         /* remove thread from thread list */
384         rt_list_remove(&(thread->tlist));
385 
386         /* put thread to end of ready queue */
387         rt_list_insert_before(&(rt_thread_priority_table[thread->current_priorit
388                                 &(thread->tlist));
389 
390         /* enable interrupt */
391         rt_hw_interrupt_enable(level);
392 
393         rt_schedule();
394 
395         return RT_EOK;
396     }
397 
398     /* enable interrupt */
399     rt_hw_interrupt_enable(level);
400 
401     return RT_EOK;
402 }
403 RTM_EXPORT(rt_thread_yield);
404 
405 /**
406  * This function will let current thread sleep for some ticks.
407  *
408  * @param tick the sleep ticks
409  *
410  * @return RT_EOK
411  */
412 rt_err_t rt_thread_sleep(rt_tick_t tick)
413 {
414     register rt_base_t temp;
```

```
415     struct rt_thread *thread;
416 
417     /* disable interrupt */
418     temp = rt_hw_interrupt_disable();
419     /* set to current thread */
420     thread = rt_current_thread;
421     RT_ASSERT(thread != RT_NULL);
422 
423     /* suspend thread */
424     rt_thread_suspend(thread);
425 
426     /* reset the timeout of thread timer and start it */
427     rt_timer_control(&(thread->thread_timer), RT_TIMER_CTRL_SET_TIME,
&tick);
428     rt_timer_start(&(thread->thread_timer));
429 
430     /* enable interrupt */
431     rt_hw_interrupt_enable(temp);
432 
433     rt_schedule();
434 
435     /* clear error number of this thread to RT_EOK */
436     if (thread->error == -RT_ETIMEOUT)
437         thread->error = RT_EOK;
438 
439     return RT_EOK;
440 }
441 
442 /**
443  * This function will let current thread delay for some ticks.
444  *
445  * @param tick the delay ticks
446  *
```

```
447  * @return RT_EOK
448  */
449 rt_err_t rt_thread_delay(rt_tick_t tick)
450 {
451     return rt_thread_sleep(tick);
452 }
453 RTM_EXPORT(rt_thread_delay);
454
455 /**
456  * This function will control thread behaviors according to control
command.
457  *
458  * @param thread the specified thread to be controlled
459  * @param cmd the control command, which includes
460  *  RT_THREAD_CTRL_CHANGE_PRIORITY for changing priority level of thread;
461  *  RT_THREAD_CTRL_STARTUP for starting a thread;
462  *  RT_THREAD_CTRL_CLOSE for delete a thread.
463  * @param arg the argument of control command
464  *
465  * @return RT_EOK
466  */
467 rt_err_t rt_thread_control(rt_thread_t thread, rt_uint8_t cmd, void *arg)
468 {
469     register rt_base_t temp;
470
471     /* thread check */
472     RT_ASSERT(thread != RT_NULL);
473
474     switch (cmd)
475     {
476     case RT_THREAD_CTRL_CHANGE_PRIORITY:
477         /* disable interrupt */
478         temp = rt_hw_interrupt_disable();
```

```
479
480         /* for ready thread, change queue */
481         if (thread->stat == RT_THREAD_READY)
482         {
483             /* remove thread from schedule queue first */
484             rt_schedule_remove_thread(thread);
485
486             /* change thread priority */
487             thread->current_priority = *(rt_uint8_t *)arg;
488
489             /* recalculate priority attribute */
490 #if RT_THREAD_PRIORITY_MAX > 32
491             thread->number      = thread->current_priority >> 3;
/* 5bit */
492             thread->number_mask = 1 << thread->number;
493             thread->high_mask   = 1 << (thread->current_priority
& 0x07);   /* 3bit */
494 #else
495             thread->number_mask = 1 << thread->current_priority;
496 #endif
497
498             /* insert thread to schedule queue again */
499             rt_schedule_insert_thread(thread);
500         }
501         else
502         {
503             thread->current_priority = *(rt_uint8_t *)arg;
504
505             /* recalculate priority attribute */
506 #if RT_THREAD_PRIORITY_MAX > 32
507             thread->number      = thread->current_priority >> 3;
/* 5bit */
508             thread->number_mask = 1 << thread->number;
```

```
509             thread->high_mask   = 1 << (thread->current_priority
& 0x07);   /* 3bit */
510 #else
511             thread->number_mask = 1 << thread->current_priority;
512 #endif
513         }
514
515         /* enable interrupt */
516         rt_hw_interrupt_enable(temp);
517         break;
518
519     case RT_THREAD_CTRL_STARTUP:
520         return rt_thread_startup(thread);
521
522 #ifdef RT_USING_HEAP
523     case RT_THREAD_CTRL_CLOSE:
524         return rt_thread_delete(thread);
525 #endif
526
527     default:
528         break;
529     }
530
531     return RT_EOK;
532 }
533 RTM_EXPORT(rt_thread_control);
534
535 /**
536  * This function will suspend the specified thread.
537  *
538  * @param thread the thread to be suspended
539  *
540  * @return the operation status, RT_EOK on OK, -RT_ERROR on error
```

```
541  *
542  * @note if suspend self thread, after this function call, the
543  * rt_schedule() must be invoked.
544  */
545 rt_err_t rt_thread_suspend(rt_thread_t thread)
546 {
547     register rt_base_t temp;
548 
549     /* thread check */
550     RT_ASSERT(thread != RT_NULL);
551 
552     RT_DEBUG_LOG(RT_DEBUG_THREAD, ("thread suspend:  %s\n", thread->name));
553 
554     if (thread->stat != RT_THREAD_READY)
555     {
556         RT_DEBUG_LOG(RT_DEBUG_THREAD, ("thread suspend: thread disorder,
%d\n",
557                                        thread->stat));
558 
559         return -RT_ERROR;
560     }
561 
562     /* disable interrupt */
563     temp = rt_hw_interrupt_disable();
564 
565     /* change thread stat */
566     thread->stat = RT_THREAD_SUSPEND;
567     rt_schedule_remove_thread(thread);
568 
569     /* enable interrupt */
570     rt_hw_interrupt_enable(temp);
571 
572     return RT_EOK;
```

```
573 }
574 RTM_EXPORT(rt_thread_suspend);
575
576 /**
577  * This function will resume a thread and put it to system ready queue.
578  *
579  * @param thread the thread to be resumed
580  *
581  * @return the operation status, RT_EOK on OK, -RT_ERROR on error
582  */
583 rt_err_t rt_thread_resume(rt_thread_t thread)
584 {
585     register rt_base_t temp;
586
587     /* thread check */
588     RT_ASSERT(thread != RT_NULL);
589
590     RT_DEBUG_LOG(RT_DEBUG_THREAD, ("thread resume:  %s\n", thread->name));
591
592     if (thread->stat != RT_THREAD_SUSPEND)
593     {
594         RT_DEBUG_LOG(RT_DEBUG_THREAD, ("thread resume: thread disorder,
%d\n",
595                                        thread->stat));
596
597         return -RT_ERROR;
598     }
599
600     /* disable interrupt */
601     temp = rt_hw_interrupt_disable();
602
603     /* remove from suspend list */
604     rt_list_remove(&(thread->tlist));
```

```
605
606     /* remove thread timer */
607     rt_list_remove(&(thread->thread_timer.list));
608
609     /* change timer state */
610     thread->thread_timer.parent.flag &= ~RT_TIMER_FLAG_ACTIVATED;
611
612     /* enable interrupt */
613     rt_hw_interrupt_enable(temp);
614
615     /* insert to schedule ready list */
616     rt_schedule_insert_thread(thread);
617
618     return RT_EOK;
619 }
620 RTM_EXPORT(rt_thread_resume);
621
622 /**
623  * This function is the timeout function for thread, normally which
is invoked
624  * when thread is timeout to wait some resource.
625  *
626  * @param parameter the parameter of thread timeout function
627  */
628 void rt_thread_timeout(void *parameter)
629 {
630     struct rt_thread *thread;
631
632     thread = (struct rt_thread *)parameter;
633
634     /* thread check */
635     RT_ASSERT(thread != RT_NULL);
636     RT_ASSERT(thread->stat == RT_THREAD_SUSPEND);
```

```
637
638     /* set error number */
639     thread->error = -RT_ETIMEOUT;
640
641     /* remove from suspend list */
642     rt_list_remove(&(thread->tlist));
643
644     /* insert to schedule ready list */
645     rt_schedule_insert_thread(thread);
646
647     /* do schedule */
648     rt_schedule();
649 }
650 RTM_EXPORT(rt_thread_timeout);
651
652 /**
653  * This function will find the specified thread.
654  *
655  * @param name the name of thread finding
656  *
657  * @return the found thread
658  *
659  * @note please don't invoke this function in interrupt status.
660  */
661 rt_thread_t rt_thread_find(char *name)
662 {
663     struct rt_object_information *information;
664     struct rt_object *object;
665     struct rt_list_node *node;
666
667     extern struct rt_object_information rt_object_container[];
668
669     /* enter critical */
```

```
670     if (rt_thread_self() != RT_NULL)
671         rt_enter_critical();
672 
673     /* try to find device object */
674     information = &rt_object_container[RT_Object_Class_Thread];
675     for (node  = information->object_list.next;
676             node != &(information->object_list);
677             node  = node->next)
678     {
679         object = rt_list_entry(node, struct rt_object, list);
680         if (rt_strncmp(object->name, name, RT_NAME_MAX) == 0)
681         {
682             /* leave critical */
683             if (rt_thread_self() != RT_NULL)
684                 rt_exit_critical();
685 
686             return (rt_thread_t)object;
687         }
688     }
689 
690     /* leave critical */
691     if (rt_thread_self() != RT_NULL)
692         rt_exit_critical();
693 
694     /* not found */
695     return RT_NULL;
696 }
697 RTM_EXPORT(rt_thread_find);
698 
699 /*@}*/
700 
```