山东大学

硕士学位论文

基于PIC24系列微控制器的嵌入式实时操作系统的设计与实现

姓名：张磊

申请学位级别：硕士

专业：检测技术与自动化装置

指导教师：杜晓通

20080510

摘 要

本文以 Microchip 公司生产的高性能的 PIC24 系列微控制器为硬件平台，设计和实现了一种基于 PIC24 系列微控制器的嵌入式实时操作系统，包括操作系统内核和 Bootloader 程序。

随着嵌入式系统的发展，传统的前后台式的软件设计结构已经明显不能满足日益复杂的应用的需求。在嵌入式系统的软件设计过程中，任务级响应时间长、不固定以及因各种资源调度不当而发生死锁、系统可靠性降低等问题越来越突出。要解决这些问题，必须将操作系统的概念引入到嵌入式系统的软件设计过程中。

嵌入式实时操作系统是一种新的系统设计思想和一个开放的软件框架。它具有操作系统的基本功能，可以对整个实时系统的运行进行控制，根据系统中各个任务的轻重缓急，合理地在任务之间分配 CPU 和各种资源。实时操作系统利用信号量、消息等系统功能协调和同步各个任务，降低了任务模块之间的耦合性，提高了系统的稳定性。此外，基于嵌入式实时操作系统的软件设计模式天然地具有良好的可扩展性，便于软件系统的改进和扩展，有效地降低了成本，提高了开发效率。随着嵌入式系统的广泛应用，基于嵌入式实时操作系统的软件设计方法必将得到广泛的应用。

本文设计和实现的嵌入式实时操作系统主要包括调度内核、任务管理、时间管理、任务之间的通信与同步、中断响应管理和 Bootloader 模块。

最后，以山东省肿瘤防治研究院能量管理系统中的能量检测控制器的设计为例，介绍嵌入式实时操作系统在工程中的应用。结合具体的软件设计过程，充分阐述在实际的设计过程中，如何在嵌入式实时操作系统平台上进行软件的设计和实现，以及嵌入式实时操作系统内核功能的正确、高效使用。通过该系统的成功运行，验证了所设计的嵌入式实时操作系统程序具有良好的稳定性和可靠性。

可以预见，由于嵌入式实时操作系统不可替代的优点，其必将大大加快嵌入式应用开发的速度，并有效地提高系统的稳定性、可靠性和可扩展性。

关键词：嵌入式，操作系统，微控制器，**PIC24**，**Bootloader**

ABSTRACT

This thesis mainly introduces a kind of method to design and implement an Embedded Real-Time Operating System based on PIC24 microcontroller, imcluding OS kernel and Bootloader program.

With the development of Embedded System, tranditonal foreground-background software architecture is no longer able to support the more and more complicated application requirement. For the complicated application, task level response time will be longer and not fixed, because of inappropriate scheduling of resourses, and the problem of deadlock and decrease of system reliability will be more outstanding. In order to resolve these problems and combine its powerful function with more demands for IT products, it is necessary to introduce the Embedded Real-Time Operating System into Embedded System design process to take the place of foregroundan-background software architecture.

RTOS is a new system design concept and open software architecture. It decreases the complexity of the routine. It has basic functions of an Operating System that can control the whole system, and assigns CPU time slot and other hardware resources to tasks according to their importance and urgency. It improves the efficiency of CPU usage by providing semaphore, message and other system functions. The usage of RTOS makes the expansion of the application convenient, reduces the cost, and improves the system reliability. It is sure that the software design method on RTOS platform must be widely applied, because of its great advantages.

This thesis induces the design and implementation procedure of Embedded RTOS, including the kernel structure, task and time management, task synchronization and communication, and Bootloader module.

At the end of thesis, the design process of Energy Meamurement and Control Instrument which is part of Shandong Cancer Research Center Main Building Energy Meamurement System is taked as an example to describe how to design the software on RTOS platform correctly and effectively.

Keywords: Embedded System, Operating System, PIC24, Microcontroller, Bootloader

原 创 性 声 明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究所取得的成果。除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的科研成果。对本文的研究作出重要贡献的个人和集体，均已在文中以明确方式标明。本声明的法律责任由本人承担。

论文作者签名：张磊　　日　期：2008.5.10

关于学位论文使用授权的声明

本人同意学校保留或向国家有关部门或机构送交论文的印刷件和电子版，允许论文被查阅和借阅；本人授权山东大学可以将本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、缩印或其他复制手段保存论文和汇编本学位论文。

(保密论文在解密后应遵守此规定)

论文作者签名：张磊　导师签名：　日　期：2008.5.10

前言

近年来，以单片机、数字信号处理器为核心的嵌入式计算系统，以其功耗低、可靠性高、适应面广等优点获得了越来越广泛的应用，并且已经在很大程度上改变了人们的生产、生活方式。未来随着技术的不断革新，可以肯定嵌入式系统的应用将会越来越广泛。

由于软硬件技术的限制，早期的嵌入式系统中并没有操作系统的概念，嵌入式程序通常采用前后台系统结构，以中断来驱动各种应用功能。随着应用的需求越来越复杂，这样的系统结构逐渐显现出了许多问题，如：无法支撑系统结构复杂的应用，难以重用已有代码，不适合团队合作开发等，这些问题的出现使得在嵌入式系统中引入操作系统的概念变得越来越迫切。同时，随着微电子技术的进步，硬件的性能飞速发展，也使得在嵌入式硬件平台上引入操作系统的概念成为了可能。于是从上世纪八十年代开始出现了商用嵌入式实时操作系统。但是许多商业化的嵌入式实时操作系统（如：PSOS，VxWorks，QNX等）不仅价格昂贵而且不公开源代码。即使是一些开放源代码的嵌入式实时操作系统也各有不足之处，难以按照应用的需求移植到自己开发的嵌入式系统中。

PIC24 系列微控制器是 Mrcrochip 公司最新推出的 16 位微控制器，具有较高的性能和丰富的资源，其硬件资源完全可以支持嵌入式实时操作系统的运行。

基于以上原因，同时为了满足实际工程应用的需求，决定针对PIC24系列微控制器设计一个嵌入式实时操作系统。由于该操作系统还不够成熟，在将来的进一步开发中必然需要反复修改和升级，所以在开发嵌入式实时操作系统内核的同时开发了相应的Bootloader程序，方便了该嵌入式实时操作系统和应用软件的继续开发和升级。另外，本文所研究和实现的嵌入式实时操作系统已经成功地应用到若干实际项目中。

文中各章节安排如下：第一章简述论文选题的原因、目的和目标，介绍PIC24系列微控制器和嵌入式实时操作系统的特点、发展状况和前景。第二章详细介绍基于PIC24系列微控制器的嵌入式实时操作系统的设计和实现方法。第三章描述在PIC24微控制器上开发Bootloader程序的过程。第四章介绍本文所设计的嵌入式实时操作系统在工程中的应用。第五章为总结和展望。

第一章 绪论

嵌入式系统是以应用为中心，以计算机技术为基础、软硬件可裁剪，对功能、可靠性、成本、体积、功耗严格要求的专用计算机系统[1]。随着嵌入式系统的应用越来越广泛，功能越来越复杂，系统也变得越来越庞大。在开发成本、系统可靠性和实时性等方面，传统的前后台设计开发模式已经明显无法满足日益增长的应用需求。为了解决这种矛盾，越来越多的开发人员开始意识到在嵌入式系统中引入操作系统概念的必要性。另一方面，微电子技术的迅猛发展使得微控制器的运算性能不断提高，也为在嵌入式平台上引入操作系统创造了条件。目前很多开发人员已经将嵌入式实时操作系统应用到实际的项目中。可以预见，未来在嵌入式系统中应用嵌入式操作系统的软件设计方法必将取代前后台设计方法，成为嵌入式系统设计的发展方向。

1.1 课题设计

1.1.1 课题设计原因

在作者以前开发的项目中，硬件采用 PIC24 系列微控制器，软件结构采用传统的前后台模式。整个程序分为前台程序和后台程序两部分。前台的中断服务子程序负责处理异步事件和实时性要求严格的操作，为了提高中断处理的实时性能，通常仅仅在中断服务程序中标记事件的发生，然后就迅速退出中断。后台程序是一个无限循环，通过查询标志位的变化来调用相应的处理过程，以响应外界异步事件。

前后台程序设计方法主要存在五个缺陷：一是实时性较差。系统在处理中断时，通常仅仅置标志位后就立即退出中断，不考虑该中断所引发应用功能的重要性。即使该功能非常重要，也还是返回原先被中断的过程，直到轮到该应用处理函数运行时才执行。二是可靠性难以保证。当系统功能复杂时，要考虑的各种可能太多，软件的可靠性难以保证。三是各功能函数之间交换信息困难。前后台设计方法一般采用全局变量的方法实现通讯。当功能多、情况复杂时，采用全局变量实现通信就容易出现问题。四是不利于大型系统的开发。随着嵌入式系统复杂度的不断提高，越来越多的系统需要团队的开发才能及时完成。而在前后台系统结构中，各个函数或程序模块之间耦合性极强，无法适应团队分工合作开发模式

的需要。五是可扩展性差。在实际的嵌入式项目开发中，经常要对原先软硬件系统进行扩展和改造。前后台系统中，各个功能模块之间是直接的调用和被调用的关系，对任何一个函数、变量或者程序语句的更改都有可能影响到其他函数，导致无法预料的后果。因此很难实现软件的扩展和改造。

由此可见，前后台设计方法只适合于外设和功能不多，并且实时性要求不高时的场合。如果应用比较复杂，或者要求较高的实时处理能力时，就无能为力了。在这种情况下，必须使用嵌入式实时操作系统的软件设计方法。

1.1.2 课题设计目的

设计嵌入式实时操作系统的目的：首先是要充分发挥 PIC24 系列微控制器的性能，适应人们对嵌入式产品功能更高的要求，使其成为一个高集成、高效率、低功耗嵌入式产品的开发平台。其次，通过实现一个基于 PIC24 系列微控制器的嵌入式操作系统，为应用软件开发人员提供一个友好的开发平台。通过提供多任务调度、时间管理、任务间通信和同步等功能，使得应用程序更加易于设计、开发、维护和扩展。并且能够使产品更加稳定可靠，降低开发成本。

由于该操作系统还不够成熟，在将来的开发中必然需要反复修改和升级，而且在实际项目的开发和部署过程中，也常常需要对已有的应用软件系统进行升级。Microchip 公司为 PIC24 系列微控制器提供了相应的 MPLAB IDE 开发环境和 MPLAB ICD2 编程器/调试器这样的开发工具。但是，这些工具仅仅在实验室环境或者研发过程中是易于使用的，一旦系统部署到实际的应用现场，逐个设备地重新编程将变得十分困难。对于这种情况，最合适的解决方案就是在嵌入式操作系统中加入 Bootloader 模块，这样就可以利用 RS-232、RS-485、CAN 总线等通信方式，方便地在现场实现软件升级。因此，针对 PIC24 系列微控制器开发了专用的 Bootloader 程序。

1.1.3 课题设计目标

原则：在满足系统功能的前提下尽可能紧凑、简单，最大限度地减少资源消耗。

目标：

1、最多能同时调度 16 个实时任务。

2、采用基于优先级的抢占式调度策略，保证系统的实时响应能力。

3、尽可能减少所占用的程序存储器、内存空间和 CPU 的额外开销。

4、PIC24 系列微控制器专用的 Bootloader 程序。

1.2 PIC24 系列微控制器

美国 Microchip 公司最新推出的 16 位 PIC24 系列微控制器采用全新的内核架构和流水线结构，具有卓越的性能。PIC24 系列又分为 PIC24F 和 PIC24H 两个档次，适合在不同领域中的应用。PIC24F 系列最高性能为 16MIPS，适用于低成本和低功耗的应用。PIC24H 系列提供高达 40MIPS 的运算性能，更大的存储空间和更多的外设，适用于高性能的应用。该系列微控制器具有以下特点：

- 最高支持 40MIPS 的高性能内核结构。16 个内核工作寄存器，17X17 单周期硬件乘法器，单周期移位运算指令。
- 全新的中断向量结构。PIC24 系列微控制器内核为每一个中断源提供唯一的中断向量，以及可配置的高达 16 级的中断优先级。
- 丰富的片上外围模块：PIC24F 系列微控制器上集成了较多的片上外围模块，包括：最高 16KB 的 RAM 和 256KB 的 Flash 程序存储器，多个串行通信端口（UART、SPI、I2C），16 位的定时器模块，16 通道的 12 位 500kbps 高速 A/D 转换器以及实时日历时钟模块。
- 低功耗电源管理系统：上电复位和故障保护时钟监视器、采用纳瓦技术的电源管理、片上低压降稳压器以及多种低功耗处理器模式。
- 开发方便：Microchip 公司提供专用的集成开发环境（MPLAB IDE）和硬件仿真器/编程器（MPLAB ICD2），实现程序编写、模拟仿真和在线调试，为用户开发提供了极大的方便。

嵌入式操作系统本身也是一个软件，也需要消耗一定的硬件资源，包括 CPU 时间、RAM 和程序存储区。因此，选择 PIC24FJ32GA002 芯片作为本设计的硬件平台。它具有 8KB 的 RAM 和 32KB 的 Flash 程序存储区，完全能够满足系统设计的需求。

1.3 嵌入式实时操作系统

1.3.1 嵌入式实时操作系统特点

嵌入式实时操作系统作为大多数实时系统的软件平台,它管理系统的硬件资源，为应用软件提供各种必要的服务[4]。用户的应用程序是运行于嵌入式实时操作系统之上的各个任务。嵌入式实时操作系统根据各个任务的需求，进行资源管理、任务调度、异常处理等工作。

嵌入式实时操作系统的基本特点有：

- 执行时间的可确定性：使用实时操作系统的目的，就是要提高系统的执行效率。实时操作系统采用各种算法和策略，保证系统行为的可预测性。
- 可裁剪性：用户可以根据应用系统对功能、可靠性、成本、体积、功耗等的要求对实时操作系统进行裁剪。
- 可移植性：嵌入式设备所采用的微控制器的结构和资源差别很大。因此，嵌入式实时操作系统必须能够方便地在不同的微控制器上移植。
- 多任务抢占：嵌入式实时操作系统根据任务的重要性给不同的任务分配不同的优先级。优先级较高的任务，优先得到 CPU 的使用权。

在嵌入式设备中引入操作系统体现了一种全新的系统设计思想和一个开放的软件框架。首先，嵌入式操作系统保证了系统实时性的需求，使得嵌入式设备运行更加稳定可靠。第二，嵌入式操作系统是最接近硬件的软件模块，直接管理着 CPU、定时器、各种外围模块等系统资源，按照一定的调度策略为各个任务合理地分配资源，既满足各个应用任务的需求，又充分利用了硬件资源，最大程度地发挥了硬件的能力。第三，在采用嵌入式操作系统的软件结构设计中，应用需求被划分为若干个任务，各个任务之间耦合性降低，符合模块化的软件架构设计思想。这种设计方法不仅有利于保证系统的稳定性和可靠性，而且适合团队开发，提高了开发效率。第四，在嵌入式实时操作系统平台上，应用程序的可复用性也大大提高。开发人员可以在不变动其他任务的情况下就能够增加或去掉一个任务，使得系统的更改和扩展变得十分方便和灵活。

当然，使用操作系统进行程序设计也会带来一些负面效应。首先，为了支持嵌入式实时操作系统的运行，必然要消耗一些系统资源。其次，由于所有的软件功能都是以任务的形式出现，各个任务的重要性不同，所需要的硬件资源也不同。因此，如何合理地划分任务，为各个任务分配优先级，以及如何实现任务之间的通信和同步都是需要认真考虑和设计的问题。如果设计不当，不仅无法发挥嵌入式实时操作系统优势，反而会适得其反。

综上所述，虽然在嵌入式系统中采用实时操作系统的设计思想并不能解决一切问题。但是随着软硬件技术的不断发展，负面效应会越来越小而优势变得越来越明显。因此，嵌入式实时操作系统的应用必将是大势所趋。

1.3.2 嵌入式实时操作系统发展

1.3.2.1 嵌入式实时操作系统历史

嵌入式实时操作系统的发展自上世纪六十年代开始可以分为三个阶段：

1. 早期的实时操作系统

早期的实时操作系统是小而简单、带有一定专用性的软件，功能较弱，可以认为是一种实时监控程序。它一般为用户提供对系统的初始化管理以及简单的实时时钟管理。这一时期的实时应用比较简单，实时性要求也不高。应用程序、实时监控程序和硬件平台是紧耦合在一起的。

2. 专用实时操作系统

随着应用规模的扩大和应用复杂度的提高，早期的实时操作系统已经不能满足应用的需求了。有些实时系统的开发者研制了与特定硬件相匹配的实时操作系统。它是用户为满足自身开发需要而研制的，它一般只能适用于特定的硬件环境。因此，可以认为是一种专用的实时操作系统。

3. 通用实时操作系统

由于多任务的机制如基于优先级的调度、实时时钟管理、任务间的通信、同步/互斥机制等基本上是相同的，不同的只是面向各自的硬件环境与应用目标，所以可以把这部分很好的组织起来，形成一个通用的实时操作系统内核。在内核的最底层将不同的硬件特性屏蔽掉。

1.3.2.2 嵌入式实时操作系统现状

在二十世纪八十年代末，许多商业化的嵌入式实时操作系统被开发出来。应用较广泛的有 VXWORKS、PSOS、嵌入式 Linux、QNX、WINCE 等。它们不管是从体系结构上，还是从功能模块和性能指标上都存在着较大的差异。

下面就对几种著名的嵌入式实时操作系统做简要的介绍[6][7]：

- VxWorks

VxWorks操作系统由美国WindRiver公司开发的嵌入式实时操作系统，具有可裁剪微内核结构、高效的任务管理、灵活的任务间通讯、微秒级的中断处理、支持POSIX 1003.1b 实时扩展标准、支持多种物理介质及标准的、完整的TCP/IP网络协议等。但是其价格比较高，并且不提供源代码。由于是专用操作系统，需要专门的技术人员掌握开发技术和维护，所以软件的开发和维护成本都非常高。支持的硬件数量有限。

- Windows CE

Windows CE是一种针对小容量、移动式、智能化、32位的模块化实时嵌入式操作系统平台，为建立针对掌上设备、无线设备的动态应用程序和服务提供了丰富的功能。它能在多种处理器体系结构上运行，是多线程、完整优先权、多任务的操作系统。它的模块化设计允许它对从掌上电脑到专用的工业控制器的用户电子设备进行定制。Windws CE作为嵌入式操作系统有很多的缺陷：没有开放源代码，使应用开发人员很难实现产品的定制；在效率、功耗方面的表现并不出色，而且占用过多的系统内存，应用程序庞大，且价格较昂贵。

◆ 嵌入式Linux

Linux 最大的特点是源代码公开，人们可以任意修改以满足自己的应用。遵从 GPL 协议，无须为每例应用交纳许可证费。有大量的应用软件可用。其中大部分也都遵从 GPL 协议，可以稍加修改后应用于用户自己的系统。优秀的网络功能。稳定是 Linux 本身具备的一个很大优点。内核精悍，运行所需资源少，十分适合嵌入式应用。支持的硬件数量庞大。在嵌入式系统上运行 Linux 的缺点是 Linux 体系提供实时性能需要添加实时软件模块。而这些模块运行的内核空间正是操作系统实现调度策略、硬件中断异常和执行程序的部分。由于这些实时软件模块是在内核空间运行的，因此代码错误可能会破坏操作系统从而影响整个系统的可靠性，这对于实时应用是一个非常严重的弱点。

◆ μC/OS-II

μC/OS-II是著名的源代码公开的实时内核，是专为嵌入式应用设计的，可用于8 位、16 位和32位单片机或数字信号处理器。它的主要特点如下：公开源代码；绝大部分源代码是用C语言写的，便于移植到其他微控制器上；可固化，可裁剪性，可以有选择地使用需要的系统服务，以减少所需的存储空间；多任务，可管理64个任务，任务的优先级必须是不同的，不支持时间片轮转调度法；可确定性，函数调用与服务的执行时间具有其可确定性，不依赖于任务的多少；实用性和可靠性，成功应用该实时内核的实例，是其实用性和可靠性的最好证据。

这些商业化的嵌入式实时操作系统仍然存在着许多问题。首先，应用代码的重用性差。选择不同的嵌入式实时操作开发时，不能保护用户已有的应用投资，给应用开发者带来难题。其次，多数商业化嵌入式实时操作系统普遍价格昂贵。第三，虽然各个嵌入式实时操作系统都将可裁剪性和可移植性作为重要特性，但是目前来看，由于在嵌入式系统中采用的微控制器千差万别，导致在不同的微控

制器平台上移植一个操作系统仍然不是一件容易的事情。第四，不提供源代码。应用者无法了解其细节或者需要专门的熟悉操作系统的人员负责操作系统平台的研发。以上这些都是目前商业化的嵌入式实时操作系统所存在的通病。

1.3.3 嵌入式实时操作系统应用前景

嵌入式系统在工业控制、商业管理领域、消费类和医疗保健类领域都具有广阔的前景，如智能工控设备、POS/ATM 机、IC 卡机顶盒、WebTV、手机、智能交通等。进入新世纪，嵌入式系统的发展和应用已经如火如荼，国内外众多的厂商开发出了许多各具特色的产品。但是面对客户对产品越来越高的要求，只有功能强大、运行稳定可靠、成本低廉的产品才能在激烈的市场竞争中获胜。而要开发出这样的产品，只有采用嵌入式实时操作系统的设计方式才能实现。

在嵌入式系统中引入嵌入式操作系统，首先提高了系统的可靠性。其次，提高了开发效率，缩短了开发周期。第三，在嵌入式实时操作系统环境下开发的应用程序更加容易扩展。第四，通过嵌入式实时操作系统的协调管理，使得系统资源得到了更好、更高效的利用，充分发挥了微控制器的潜力。

由此可见，嵌入式操作系统的应用已是未来嵌入式系统开发的发展方向。而如何开发出容量小、稳定性高、易于维护且成本低廉的嵌入式实时操作系统就成为了当务之急。可以看出，嵌入式实时操作系统的发展是机遇与挑战并存。

如前所述，美国 Microchip 公司推出 PIC24 系列 16 位高性能微控制器已经具备了运行嵌入式实时操作系统的能力，同时经过数十年的研究和发展，嵌入式实时操作系统的理论研究也逐步完善。所以，在 PIC24 系列微控制器平台上开发自主的嵌入式实时操作系统在技术上是可行的，在市场应用上也是大有前途的。

第二章 嵌入式实时操作系统的设计

本文设计的嵌入式实时操作系统，采用了现代软件工程中的模块化设计思路，将嵌入式实时操作系统分为调度内核、任务管理、通信与同步、中断管理等模块。为了最大程度地提高该嵌入式实时操作系统的可移植性、可扩展性和实时性，将整个嵌入式实时操作系统程序分为硬件相关部分和硬件无关部分。与硬件无关的部分采用 C 语言实现，与硬件相关的部分采用汇编语言编写，同时尽力压缩汇编部分的代码量。这样既保证了系统的实时响应能力，又降低了系统移植的复杂性。由于嵌入式实时操作系统的设计理论很成熟，在设计系统时借鉴了一些开放源代码的实时操作系统的设计思想和代码，如 Linux、μC/OS-II 等。

嵌入式实时操作系统通常包括调度内核、任务管理、任务通信和同步、中断处理等几个主要的模块。本章将逐一介绍这些模块的设计和实现过程。

2.1 PIC24 微控制器 CPU 内核结构

2.1.1 CPU 内核

操作系统的任务就是对 CPU 和各种硬件资源进行管理和调度。例如：任务切换过程本质就是 CPU 寄存器的存取操作，这与 CPU 结构和内存的管理方式密切相关。因此，在设计嵌入式实时操作系统之前，深入理解和认识 CPU 内核的工作模式是十分必要的。

PIC24 微控制器CPU有16 个工作寄存器和7个控制寄存器，如下表所示

寄存器名称	说明
W0 到 W15	工作寄存器阵列
PC	23 位程序计数器
SR	ALU 状态寄存器
SPLIM	堆栈指针边界寄存器
TBLPAG	表存储器页地址寄存器
PSVPAG	程序空间可视页面寄存器
RCOUNT	REPEAT 循环计数寄存器
CORCON	CPU 控制寄存器

2.1.2 软件堆栈

PIC24 微控制器的堆栈操作，包括函数调用、中断和异常处理，都使用由 CPU 在内存中动态分配的软件堆栈空间，并且向上（高地址方向）生长。W14 和 W15 分别作为软件堆栈帧指针和软件堆栈指针专门用于堆栈操作。

W14 是软件堆栈帧指针（FP）。“帧”是当前函数所占用的堆栈段，在函数

的堆栈帧中保存了该函数的返回地址(PC)、函数参数和局部变量。通过使用 LNK（连接）和 ULNK（断开）指令可以使 W14 指向和不指向某个堆栈帧。

W15 是软件堆栈指针（SP），它始终指向下一个可用的堆栈地址，从低地址到高地址生长，并被中断处理、程序调用和返回自动修改。当执行 PUSH 指令时，操作数被压入堆栈，然后 W15 的值加 1；当执行 POP 指令时，W15 的值减 1，然后将 W15 指向的数据弹出堆栈。当执行 CALL 指令时函数的 PC 被自动压入堆栈；当执行程序返回指令 RETURN 时，W15 的值减 1，然后将 W15 指向的数据弹出到 PC 寄存器。

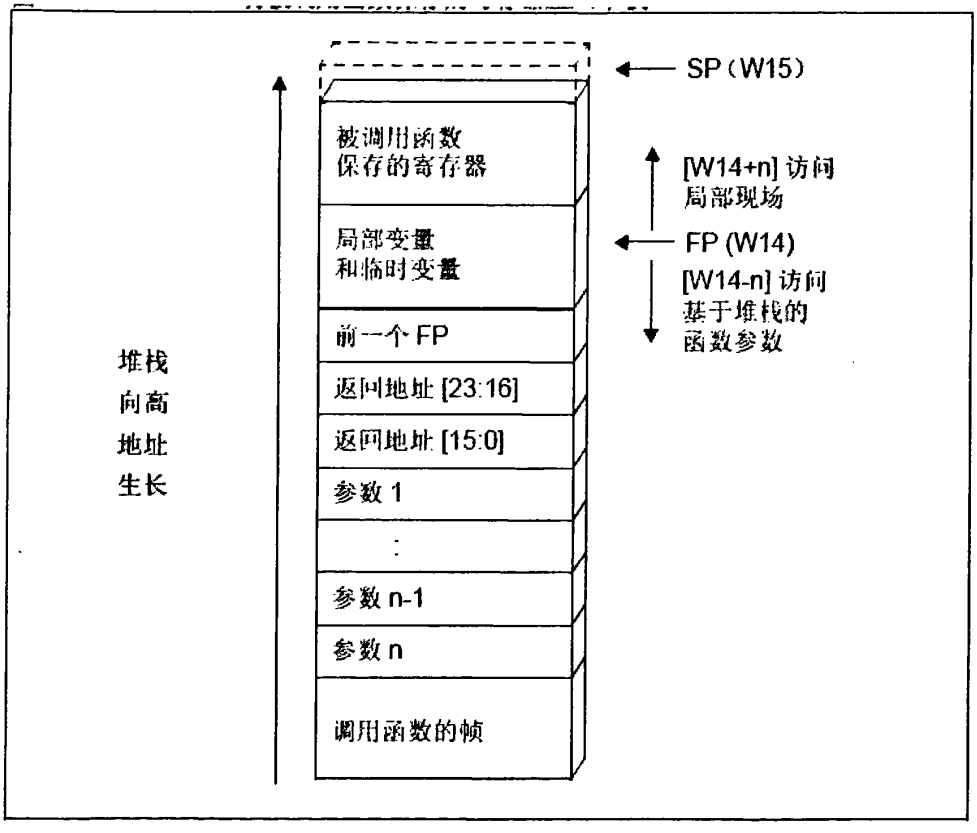


图 2.1 当执行调用函数指令 CALL 后，当前函数的软件堆栈内容

需要注意的是：由于 PC 为 24 位宽，当 PC 被压入堆栈时，PC<15:0> 位被压入第一个可用的堆栈字，然后 PC<22:16> 位被压入第二个可用的堆栈单元。中断处理期间，PC 的 MSB 与 STATUS 寄存器 SR 的低 8 位相连，使 SRL 的内容能被自动保存。

2.2 任务管理

2.2.1 任务定义

任务是嵌入式实时操作系统中最小的运行单位，也是嵌入式实时操作系统中获取资源进行调度的最小单位。系统将为任务分配资源，建立堆栈空间，定义任务的优先级和状态。

任务具有以下基本特征：

◆ 动态性：任务的状态是不断变化的。任务的状态一般分为就绪态、运行态和挂起态等。任务的状态随着系统的运行不断变化。

◆ 并行性：系统中同时存在着多个任务，通过嵌入式实时操作系统的调度，CPU 在各个任务之间切换运行。

◆ 独立性：在传统的前后台系统中，不同函数相互调用，而在嵌入式实时操作系统环境下，一个任务并不知道其他任务的存在，而是由嵌入式实时操作系统在不同的任务之间调度，协调所有任务共同完成应用功能。

从功能上看，任务就是用户的应用程序模块，每个任务都完成一个特定的功能，多个任务之间相互协调共同完成系统功能。从形式上来看，每个任务就是一个 C 语言中的函数，任务与普通函数的不同之处在于任务是不会返回的，而由嵌入式操作系统在不同的任务之间不断调度切换。

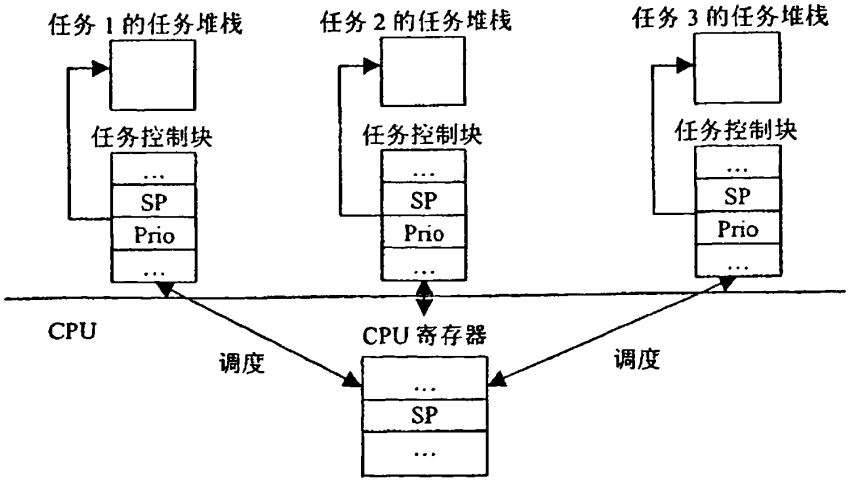


图 2.2 多任务控制

根据应用系统的具体情况，系统中的任务数也不相同，但不能大于规定的最大任务数。本文所设计的嵌入式操作系统中定义的最大任务数为 16。从理论上看，在应用系统中建立多于 16 个的任务也是可以的，但是从实践上看，系统中运行的任务越多，嵌入式操作系统管理任务的负担就越重。因此，为了保证系统的实时性将实时操作系统支持的最大任务数定义为 16。实践证明，这样的定义既保证

了系统的实时响应能力，也能够满足绝大多数应用的需求。

每个任务都有其优先级，任务越重要、实时性要求越强，被赋予的优先级就越高。本文设计的嵌入式操作系统的优先级可以从 0 到 15，值越小代表任务的优先级越高。不同的任务可以有相同的优先级。嵌入式操作系统总是运行进入处于就绪态的优先级最高的任务，而对同优先级的任务采用时间片轮转调度。

2.2.2 任务状态和状态转换

由于任务具有“动态性”的特点，所以在嵌入式实时操作系统中，任务存在着休眠态、就绪态、运行态、挂起态和被中断态共五种状态，并在这五种状态之间不断切换。如图 2.3 所示：

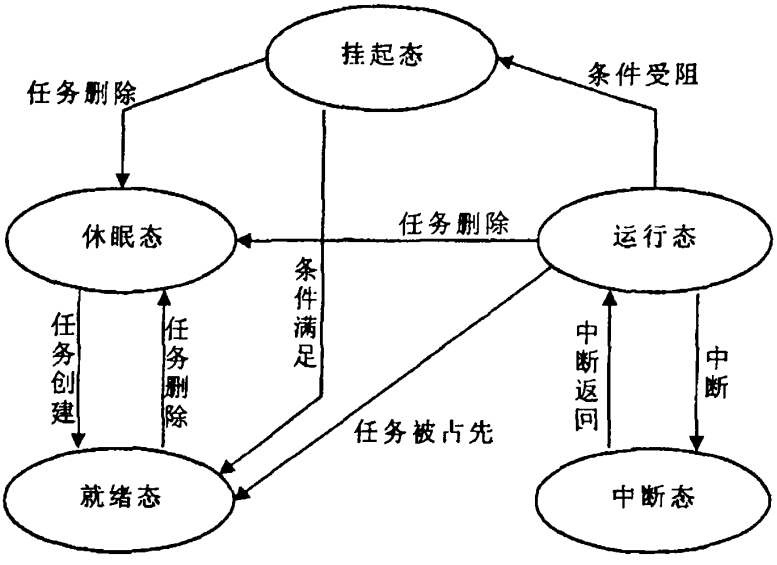


图 2.3 任务状态变迁图

◆ 运行态：任务获得了 CPU 的使用权，正在运行中。

◆ 就绪态：表明任务已经准备好，可以运行，但由于该任务的优先级低于目前运行任务的优先级或者虽然优先级相同，但还没有被轮转调度到，还暂时不能运行。所有处于就绪态的任务，都将加入到就绪任务队列等待被调度。

◆ 挂起态：当任务暂时不需要继续运行时将进入挂起态。任务的 CPU 寄存器环境和程序指针以及其他相关重要信息被保存到任务堆栈当中，等待该任务可以运行时再恢复这些状态信息。

◆ 休眠态：当任务没有运行或者已经运行完毕，暂时不需要再运行时，嵌

入式实时操作系统会将该任务的状态设置为休眠态。此时任务的寄存器环境和程序指针等信息将不会被保存，同时该任务所占用的内存空间也将被系统收回。

◆ 被中断态：当中断发生时，CPU 提供相应的中断服务，原来运行的任务暂时不能运行而进入被中断态。为了保证被中断的任务能够返回中断位置继续运行，进入中断态的任务的寄存器环境和程序指针以及其他相关重要信息同样将被保存到任务堆栈当中，并在必要时被恢复。

当嵌入式操作系统调用任务创建函数 CreateTask()创建一个任务后，被创建的任务的初始默认状态为就绪态，同时该任务加入到就绪队列中等待被任务调度函数 TaskSched()调度。

所有任务创建完毕后，嵌入式操作系统调用 StartTask()函数启动多任务环境。StartTask()调用 TaskSched()函数查找和启动就绪任务队列中优先级最高的任务。

当正在运行的任务由于某个运行条件不满足而无法继续运行时，该任务会调用 TaskSuspend()函数进入挂起态。此时，嵌入式操作系统会调用 TaskSched()函数调度新的就绪任务进入运行态。当原先被挂起任务所等待的运行条件满足时，该任务重新进入就绪态，等待下一次的调度。

另一种使任务挂起的情况是：当前任务在等待某一个任务或者过程的完成（例如等待某个信号量或者消息），此时为了提高系统资源的利用率，当前运行任务会调用 WaitSem()等函数主动将自身挂起，让出 CPU 的使用权。

任务有时需要将自身延迟一段时间再运行，此时任务可以调用 TaskDly()函数启动一个定时器。当任务被延时后，新的高优先级的就绪任务将开始运行，被延时的任务在定时时间到时重新进入就绪态。

如果正在运行的任务被另一个更高优先级的任务抢占了 CPU 使用权，由于原先运行任务的各种运行条件都是满足的，只是被更高优先级的任务占先，所以原先运行的任务不会被挂起而是进入就绪态等待被调度。

中断是嵌入式系统中响应外部异步事件的重要手段。来自内外部中断源的中断请求也有可能打断当前任务的执行，触发任务的状态转换过程。当中断发生时，CPU 会打断正在执行的任务，自动进入中断服务子程序（ISR）。由于中断标志着某个异步事件的发生，中断服务程序可能会使一个或多个任务进入就绪态。因此，中断返回之前必须进行任务调度，以判断被中断了的任务是否还是就绪任务中优先级最高的。如果中断服务子程序使一个优先级更高的任务进入了就绪态，则这

个优先级更高的任务将运行，否则继续运行原来被中断的任务。

任务可以通过调用 DeleteTask()使自己进入休眠态。如果任务已经运行完毕或者暂时不需要再运行，任务所占用的内存空间，如任务堆栈等，都将被系统收回。因为任务的内存空间已被系统收回，所以进入休眠态的任务已不能被操作系统内核调度运行。如果要想重新运行进入休眠态的任务，唯一的办法就是调用任务创建函数 CreateTask()重新创建该任务，重新为任务分配任务堆栈等运行所需的内存空间。这个过程同创建一个新的任务完全一样，被重新创建的任务会进入就绪态，等待被调度运行。

2.2.3 任务创建和任务控制块

2.1.3.1 任务控制块

任务控制块（TCB）是描述任务的数据结构，其中包含了任务信息和任务堆栈。每个任务都有唯一的一个任务控制块，它是任务在系统中存在的唯一标志，所以任务控制块的最大数目与最大任务数相同都是 16 个。

任务控制块由 CreateTask()在创建任务时建立，是从内存中分配的一段空间。当任务被删除时，该内存空间就被系统收回。任务控制块结构包含以下内容：

TaskSP：指向任务堆栈栈顶的指针。系统给每个任务分配独立的任务堆栈，以便在任务切换时保存任务的上下文环境。任务堆栈的大小是固定的。

TaskId：任务 ID 号。这是开发人员为每个任务定义的一个编号，原则上可以是任意的 unsigned char 型整数。不同任务的 ID 号不能相同。

TaskStatus：任务状态。有运行态（RUNNING）、就绪态（READY）、挂起态（SUSPEND）、休眠态（SLEEP）和被中断态（INTERRUPTED）共 5 种。

TaskPrioty：任务优先级，取值范围为 0~15。值越小，任务的优先级越高。

TaskDlyTime：任务被延迟的时钟节拍数。当需要把任务延时若干时钟节拍时要使用这个变量。如果这个变量取值为 0，表示不延时或者延时结束。

TaskEventPtr：指向事件控制块的指针，将在任务通信和同步一节中介绍。

TaskMsgPtr：指向传给任务的消息的指针，将在任务通信和同步一节中介绍。

TaskPCL、TaskPCH：任务的程序入口地址。由于程序计数器（PC）为 24 位宽，所以用两个 unsigned int 型变量表示。TaskPCL 为低字，TaskPCH 为高字。

2.1.3.2 任务创建

操作系统开始调度任务之前，必须要创建至少一个任务，而本设计中的实时

操作系统又限制了最大任务数为16。因此，系统中会存在至少1个，最多16个任务。这些任务各自拥有独立的任务控制块和任务优先级。任务被创建时，初始的默认状态为就绪态，并加入到就绪任务队列中等待被调度。

调用任务创建函数CreateTask()创建任务时，开发人员需要指定任务的名字、ID号、优先级和初始状态。这些信息都是作为函数参数传入的。

函数伪代码：

```
unsigned int CreateTask(void (* TaskName)(), unsigned char Id, unsigned char Prio,
                        unsigned char stat)                                    (1)
{
    TaskPC = (unsigned long) TaskName;
    PCL = TaskPC;
    PCH = TaskPC >> 16;                                                        (2)
    stack = (TaskStack *) malloc(sizeof(TaskStack));                           (3)
    if(NULL == stack)                                                          (4)
        {
            return ERROR;
        }
    调用任务堆栈初始化函数初始化任务堆栈;                                      (5)
    pTcb[Id].TaskSP = (unsigned int) stack;                                    (6)
    pTcb[Id].TaskId = Id;
    pTcb[Id].TaskStatus = stat;
    pTcb[Id].TaskPrioty = Prio;
    pTcb[Id].TaskPCL = PCL;
    pTcb[Id].TaskPCH = PCH;
}
```

(1) 函数参数：taskname是指向任务函数的指针；Id是任务的ID号，操作系统通过任务的ID号来区分不同的任务；taskprio是任务的优先级；stat是用户指定的任务的初始状态，该值默认为READY（就绪态）。函数返回值：反映函数执行结果的错误码。

(2) 获取任务的程序入口地址。程序的入口地址也就是指向该任务的指针，CPU只有获得该任务的程序入口地址，并将该指针写入PC后才能执行该函数。

(3) 调用C库函数中的内存分配函数malloc()从内存中分配size字节大小的存储

区作为任务的堆栈空间。任务的堆栈空间中保存了任务的 CPU 寄存器状态，使得任务在切换时能够恢复到被打断前的状态，正确地从断点处开始继续执行。

(4)malloc()函数的返回值为指向所分配内存区起始地址的指针，若分配失败则返回 NULL。因此通过判断 malloc()返回值来确定任务堆栈空间是否分配成功，如果分配失败则返回错误码并退出函数。

(5) 如果任务堆栈分配成功，则初始化任务堆栈。这一步是非常关键的，因为在 C 语言中，变量的值在未初始化前是一个随机数。如果使用没有正确初始化的变量，有可能导致程序错误。

(6) 根据任务创建函数的参数和任务堆栈分配情况初始化任务控制块。

2.2.4 任务调度

每个任务都是一个功能模块，都需要在一定的情况下、在一段时间内获得 CPU 的使用权。由于 CPU 是唯一的，所以在某一时刻只能有一个任务为运行态。嵌入式操作系统所要完成的最重要的工作就是根据一定的算法，合理地调度和切换任务，使所有任务在 2.1.2 节讲到的五种状态之间转换。

为了充分保证系统实时性要求，本文设计的嵌入式操作系统采用了基于优先级的可抢占式调度策略。前文中提到，每个任务都有自己的优先级，在基于优先级的可抢占式调度策略中，任何时刻占有 CPU 的、正在运行的任务总是当前就绪任务队列中具有最高优先级的任务。如果另一个更高优先级的任务由于某种条件的满足或者某个异步事件的发生而进入就绪态，嵌入式实时操作系统将立刻保存当前运行任务的 CPU 寄存器环境和程序指针以及其他相关重要信息到任务堆栈当中，然后将 CPU 切换到更高优先级的任务。由此可见，在基于优先级的可抢占式调度策略中，嵌入式实时操作系统总是让处于就绪态的最高优先级的任务占有 CPU 使用权。因此，系统的响应时间得到了最优化，实时性最强。

基于优先级的可抢占式调度策略又根据优先级是否可以动态调整而分为静态优先级抢占策略和动态优先级抢占策略。静态优先级策略是指：任务的优先级由开发人员预先定义好，在调用创建任务函数 CreateTask()时将该任务的优先级作为函数参数传入。因此，所有任务的优先级在运行过程中都是固定不变的。与静态优先级策略相反，在动态优先级的系统中，嵌入式操作系统允许任务在运行过程中动态地改变自身或者其他任务的优先级状态。这种策略的好处是使得嵌入式系统具有相当的灵活性，更加适应外部事件的变化，但是这种策略如果不能正确

运用，很容易导致系统死锁甚至崩溃。虽然动态优先级策略具有最大的灵活性，但要消除其不良影响又需要在嵌入式实时操作系统中加入专门的管理算法和策略。由于本文的目的是建立一个简单实用的、结构紧凑的嵌入式操作系统，所以采用了基于静态优先级的抢占式调度算法。对于具有相同优先级的就绪任务，操作系统采用时间片轮转调度策略，顺序运行各个任务。

当需要执行任务调度时，嵌入式操作系统首先调用任务调度函数 TaskSched() 在就绪任务队列中查找优先级最高的任务，然后比较该任务和当前运行任务的优先级。如果当前运行任务的优先级更高，则退出调度过程，继续运行当前任务；如果就绪任务的优先级更高，则将当前任务的状态，包括 CPU 寄存器的内容、程序指针以及其他相关重要信息，保存到任务的堆栈中。入栈工作完成后，将待运行任务的堆栈内容装入 CPU 寄存器，开始新任务的运行。

中断触发的任务调度基本与上述过程类似，但由于中断响应的特殊性仍有一些细节的差别，这将在 2.5 节中介绍。

2.2.4.1 临界区

临界区是操作系统中非常重要的一个概念，是指在运行时不能被打断的一段代码。通常来说这些代码都是关键性的代码，系统必须采用一定的机制来保证这段代码的执行是原子操作。如果代码的执行过程被打断，系统将会产生不可预料的后果，破坏程序的正常执行流程，甚至导致系统崩溃。例如：中断处理服务子程序中的堆栈操作和操作系统中任务调度和切换过程都是不允许被打断的。

处理临界区代码最常用的方法就是关中断，处理完毕后再开中断。显然，关闭中断将影响系统对实时事件的响应速度，因此只在确实必要才使代码进入临界区，并且应使处在临界区内的代码尽量少。

PIC24微控制器CPU中的SR寄存器虽然不是中断控制硬件的组成部分，但它包含控制中断功能的位。SR寄存器中的IPL2:IPL0 位可以设置当前CPU的中断优先级。用户可以通过修改这些位来改变当前CPU 的优先级，从而禁止优先级低于给定优先级的所有中断源。例如，当IPL<2:0> = 111时，CPU优先级为7，高于用户中断的优先级（0-7），所以此时所有外部用户中断就都被禁止了。

可用以下过程禁止所有的外部中断：

1. 使用PUSH 指令将当前的SR 值压入软件堆栈。
2. 将值0xE0 与SRL 进行“或”运算而使CPU 优先级强制设置为7。

如果要允许外部中断，则可以使用POP 指令来恢复先前的SR 值。

本设计中，定义了两个宏 DisInt 和 EnInt 来完成以上操作，实现禁止和允许中断。可以采用如下方法使一段代码进入临界区：

```
void example(void)
 {
   DisInt;
   被保护的用户代码段;
   EnInt;
}
```

2.2.4.2 任务调度函数

任务调度函数 TaskSched( )的伪代码如下：

```
 void TaskSched(void)
{
    DisInt;                                              (1)
    if(0== IntNesting)                                   (2)
    {
        taskid=FindMaxPrioTask()                         (3)
         if(CurrentTaskId!=taskid)
           TaskSuspend (taskid);
           EnInt;
           TaskSw( );

      }
    else
       {
         EnInt;                                          (4)
         Return;
       }
}
```

(1) 进入临界区

(2) 变量 IntNesting 记录中断嵌套层数。如果调用来自中断服务子程序，函数 TaskSched()将退出，不做任务调度。因为中断触发的任务调度和切换有专门的函

数。

(3) 调用最高优先级任务搜索函数 FindMaxPrioTask()在就绪任务队列中查找处于就绪状态的最高优先级的任务。如果找到的任务的优先级高于当前正在运行任务的优先级，则调用 TaskSw( )完成任务切换。否则不调度，直接返回。

(4) 退出临界区。

2.2.4.3 任务级的切换函数

TaskSw( )函数是任务级切换函数，区别于中断服务子程序中调用的任务切换函数IntTaskSw( )。为了提高效率，TaskSw( )的代码采用汇编语言编写。

任务切换是操作系统中最核心的过程。每个任务都有独立的任务堆栈，当发生任务切换时，操作系统将当前运行任务的所有现场信息，包括CPU寄存器、程序计数器（PC）、堆栈指针全部保存到当前任务的堆栈中，然后根据待运行任务的堆栈指针从待运行任务的堆栈中将同样的数据恢复出来。由于被切换任务的现场信息已经保存到了堆栈中，所以当该任务下次运行的时候程序就会从被打断处恢复，继续执行未完成的代码。

TaskSw( )函数的伪代码如下：

```
void TaskSw(void)
{
    获取当前任务的堆栈指针；                                    (1)
    依次将W工作寄存器阵列（W0-W15）、SR、SPLIM、TBLPAG、PSVPAG、
    RCOUNT、CORCON寄存器的内容推入当前任务堆栈；
    保存当前的返回地址（程序计数器PC指针）到任务堆栈；
    获取待运行任务的堆栈指针；                                  (2)
    从待运行任务的堆栈中依次将W工作寄存器阵列（w0-W15）、SR、SPLIM、
    TBLPAG、PSVPAG、RCOUNT、CORCON寄存器的内容恢复到CPU内
    核寄存器；
    从任务堆栈中恢复待运行任务的程序计数器PC到PC寄存器；
    执行子程序返回指令；                                        (3)
}
```

(1) 获取当前运行任务的堆栈指针，并保存当前CPU上下文环境。

(2) 获取新任务的堆栈指针，并从任务堆栈中恢复新任务的CPU上下文环境

(3) 执行子程序返回指令将待运行任务的程序计数器PC的值，即新任务的断点地址（PC值），装入到CPU寄存器中。重新开始运行的任务代码从PC指向的那一点开始运行，从而切换到新任务代码。

2.2.5 任务删除

与任务的创建相反，任务的删除函数 DeleteTask()强制终止任务的运行，然后收回任务控制块所占用的内存空间。需要注意的是，任务只能删除自身而不能删除其他任务，否则将会导致不可预料的后果。

函数伪代码：

```
unsigned int DeleteTask(void)
{
    if (IntNesting>0)
        return (ERR_DELETE_TASK_ISR);                    (1)
    DisInt;
    ListDeleteTask(CurrentTaskId);                       (2)
    Free(pTcb[id].TaskSP);                               (3)
    pTcb[Id]. TaskStatus =SLEEP;                         (4)
    pTcb[Id]. TaskPrioty =16;
    pTcb[Id]. TaskMsgPtr =NULL;                          (5)
    pTcb[Id]. TaskSP=NULL;
    pTcb[Id]. TaskEventPtr=NULL;
    pTcb[Id]. TaskPCL=NULL;
    pTcb[Id]. TaskPCH=NULL;
    EnInt;
    TaskSched( );                                        (6)
    return NO_ERROR;
}
```

(1) 不能在中断服务子程序中删除一个任务，否则返回错误信息。

(2) 从就绪任务队列中删除当前运行任务。

(3) 由于任务的堆栈空间是在任务创建时用malloc()函数在内存中分配的。当任务被删除时，必须调用free()函数释放这段内存空间。否则会导致内存泄露。

(4) 将TCB的TaskStatus域置为SLEEP，使任务进入休眠态，这样任务就不会再被

多任务内核所调度。

(5) 特别需要注意的是：TCB中的所有指针型变量必须赋值为NULL，否则这些变量将变成野指针，这是很危险的。

(6) 调用任务级调度函数TaskSched( )再次调度。

2.2.6 任务挂起

当任务暂时不需要继续运行时，任务挂起函数将任务的状态转为挂起态，以等待执行条件的到来。任务可以挂起自己，也可以挂起其他任务。

函数伪代码：

```
unsigned int TaskSuspend (unsigned int taskid)                    (1)
{
    if (taskid>=MAX_TASKS)
        return(ERR_INVALID_TASK_ID);
    DisInt;
    if (READY== pTcb[Id].TaskStatus)                              (2)
        ListDeleteTask(CuurentTaskId);
    pTcb[Id].TaskStatus =SUSPEND;                                 (3)
    if (CurrentTaskId==taskid)                                    (4)
    {
        EnInt;
        TaskSched();
        return(NO_ERROR);
    }
    EnInt;
    return NO_ERROR;
}
```

(1) 函数参数：taskid 被挂起任务的 ID 号。返回值：反映函数执行结果的错误码。

(2) 如果被挂起的任务处于就绪态，则从就绪任务队列中删除该任务。

(3) 将TCB中TaskStatus设为SUSPEND。

(4) 如果被挂起任务是当前运行任务，则调度运行新的就绪任务。

2.2.7 任务恢复

与任务的挂起相对应，当阻塞任务运行的条件满足时，实时操作系统调用任

务恢复函数将任务状态恢复为就绪态，并将任务加入就绪任务队列中。

函数伪代码：

```
unsigned int TaskResume (unsigned int taskid)                    (1)
{
    if (taskid>=MAXTASKS)
        return(ERR_INVALID_TASK_ID);
    if (SLEEP== pTcb[taskid].TaskStatus)
        return(ERR_TASK_NOT_EX IST);
    DisInt;
    if (CurrentTaskId==taskid)                                   (2)
    {
        EnInt;
        return(ERR_RESUME_RUNNING_TASK);
    }
    if (SUSPEND== pTcb[taskid].TaskStatus)                       (3)
    {
        pTcb[taskid].TaskStatus ==READY                          (4)
        TaskSched( );                                            (5)
        EnInt;
        return(NO_ERROR);
    }
    EnInt;
    return (ERR_TASK_NOT_SUSPENDED);
}
```

(1) 函数参数：taskid是待恢复任务的ID号。返回值：反映函数执行结果的错误码。

(2) 判断要恢复的是不是当前任务，因为当前任务不可能是挂起态

(3) 判断任务状态，确定待恢复的任务是否确实是挂起态。

(4) 修改任务TCB信息，重新恢复该任务为就绪态。

(5) 将任务恢复为就绪态后，再次执行任务调度

2.3 任务间通信和同步

在基于嵌入式实时操作系统平台的应用程序设计过程中，所有的应用功能都

被划分为一个个的任务。任务和任务之间或者中断和任务之间是一种协同工作的关系，因此它们之间就必然存在的信息或者数据的交换。由此可见，嵌入式操作系统必须提供一种通信和同步的机制来支持这种信息交互。

任务间的通信有以下几个用途：

- 一个任务向另一个任务传递数据。在任务之间，可以有数据的依赖关系，即一个任务是数据生产者，另一个任务是数据消费者。
- 一个任务发信号给另一个任务，通知某个事件的发生。
- 一个任务控制另一个任务的执行。任务之间可以有主从或者先后的运行关系，因此任务可以通过发消息给另一个任务来启动另一个任务运行。

通信实际上就是任务之间的信息传递的过程，这种信息传递的过程通常有两种途径：全局变量或者发消息给另一个任务。

使用全局变量进行任务间通信是最简单的办法，用这种方法可以很方便地在任务之间传递数据。提供数据的任务对全局变量进行写操作，使用数据的任务对全局变量进行读操作，从而实现了数据在任务之间的传递。但是由于全局变量是一种共享资源，对全局变量的读写操作有可能发生冲突而产生错误的数据，因此在本设计中不建议和提供这种通信方式。

消息是另一种具有行为同步能力的通信手段。当一个任务发送消息时，实际上只是发送了一个指针，该指针指向消息的首地址。另一个任务接收到这个消息后，通过指针来得到真正的消息内容。从程序实现上来看，虽然保存消息的变量也是收发消息双方的的共享资源，但是操作系统在消息收发函数的具体实现过程中已经解决了对消息的共享冲突问题，所以不会产生冲突。

本设计中用于同步和任务通信的方法是信号量和消息。在本设计中，所有的信号都被看成是事件(event)，用于通信的数据结构为事件控制块(ECB)。

任务或者中断服务子程序可以通过事件控制块来向另外的任务发信号。必须注意，只有在任务中可以等待事件发生，中断服务子程序是不能等待事件发生的，否则中断将一直处于等待状态而无法正常返回。

多个任务可以同时等待同一个事件的发生（见图 2.4）。当该事件发生后，所有等待该事件的任务中优先级最高的任务将得到该事件，并进入就绪态。

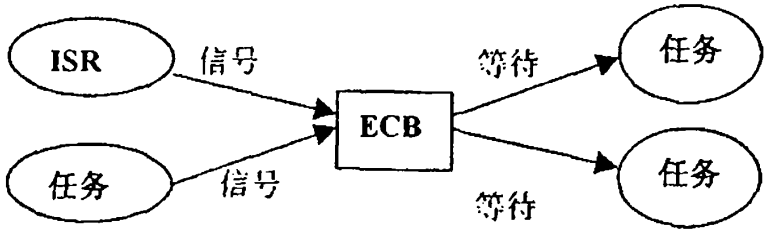


图2.4

2.3.1 事件和事件控制块

事件控制块（ECB）结构包含以下内容：

EventId：标识一个任务的ID号。

EventCnt：当事件是一个信号量时，EventCnt是信号量的计数器。

EventType：事件类型。可以是信号量(SEM)或者消息(MSG)。根据事件的类型来调用相应的系统函数。

EventPtr：当EventType是消息时指向该消息的指针。

WaitList[MAXEVENT][MAXTASK]：等待事件发生的任务队列。

2.3.1.1 任务等待事件

当任务需要等待一个事件时，WaitSem( )或ReceiveMsg( )函数调用WaitEvent ( )函数，将等待该事件的任务挂起，并将任务加入到等待该事件任务队列中。

函数伪代码：

```
unsigned int WaitEvent (unsigned int eventid, unsigned taskid)          (1)
{
 if (eventid>MAXEVENT)
       return(ERR_INVALID_EVENT_ID);
 if(taskid>MAXTASK)
      return(ERR_INVALID_TASK_ID);
 if(SLEEP ==pTcb[taskid].TaskStatus)                                    (2)
       return(ERR_SLEEP_TASK_WAIT)
  if (RUNNING==pTcb[taskid ].TaskStatus)                                (3)
    {
      pTcb[taskid].TaskStatus=SUSPEND;
      WaitList[event][taskid]=WAIT;
      TaskSched();                                                      (4)
     return(NO_ERROR)
```

```
        }
else if(READY==pTcb[taskid].TaskStatus)                    (5)
        {
        pTcb[taskid].TaskStatus=SUSPEND;
        WaitList[event][taskid]=WAIT;
        ListDeleteTask(CuurentTaskId);
        }
return (NO_ERROR);
}
```

(1) 函数参数：eventid是所等待事件的ID号，taskid是等待该事件的任务ID号。返回值：反映函数执行结果的错误码。

(2) 如果等待事件的任务已被删除，则返回错误信息。

(3) 如果等待事件的任务是当前正在运行的任务，则将任务挂起，再将任务加入到等待事件队列中。

(4) 因为当前运行任务已经被挂起，所以需要执行一次任务调度过程。

(5) 如果等待事件的任务处于就绪态，则从就绪任务队列中将该任务删除后加入到等待事件队列中。因为没有影响到当前运行的任务，所以不需要执行任务调度。

2.3.1.2 任务获得事件

当事件发生时，处在等待事件队列中的最高优先级任务将脱离等待队列并获得该事件。SendSem()和SendMsg()调用GetEvent( )以实现该操作。

函数的伪代码如下：

```
unsigned int GetEvent(unsigned int eventid)                (1)
{
    if (eventid>MAXEVENT)
        return(ERR_INVALID_EVENT_ID);
   taskid=FindMaxPrioEVENT(eventid)                        (2)
   if(SUSPEND!=pTcb[taskid].TaskStatus)                    (3)
       return ERR_TASK_NOT_SUSPEND;
  pTcb[taskid].TaskStatus=READY;                           (4)
   WaitList[eventid][taskid]=GET;
   TaskSched();                                            (5)
   return(NO_ERROR)
```

```
}
```

(1) 函数参数：eventid是所等待事件的ID号。返回值：反映函数执行结果的错误码。

(2) 查找等待事件队列，并使优先级最高的任务获得事件。

(3) 等待事件的任务应该处于挂起态，否则返回错误信息。

(4) 由于等待事件的任务得到了该事件，所以任务状态变为就绪态并从等待任务列表中被移除。

(5) 因为有新的任务进入了就绪态，所以执行一次任务调度。

2.3.2 消息

任务可以通过嵌入式操作系统提供的消息发送函数发送一个特定的消息给另一个任务。同样，一个任务也可以等待某个消息的到来。

2.3.2.1 建立消息

使用消息前，必须先调用CreateMsg()函数建立消息。因为在消息创建时事件还没有发生，所以要初始化该消息为空。

函数的伪代码如下：

```
unsigned int CreateMsg (msgid)                          (1)
{
    if (emIntNesting>0)                                 (2)
        return ERR_INT_CREATE_MSG;
    ECB[msgid]. EventPtr=NULL;                          (3)
    ECB[msgid]. EventType=MSG;
    ECB[msgid].EventCnt=0;
    CleanWaitEventList(msgid);                          (4)
    return NO_ERROR;
}
```

(1) 函数参数：msgid是消息ID号。返回值：反映函数执行结果的错误码。

(2) 中断服务子程序不能调用CreateMsg( )函数。

(3) 初始化ECB：将该ECB的事件类型设置成MSG，初始化指向消息的EventPtr指针为NULL，信号量计数值清零。

(4) 消息被创建时没有任何任务在等待它，所以清空等待该消息的任务队列。

2.3.2.2 删除消息

函数原型：unsigned int DeleteMsg(unsigned int msgid);

参数：msgid是消息ID号。

返回值：反映函数执行结果的错误码。

函数DeleteMsg( )的代码与CreateMsg ( )类似，只是对ECB的操作相反。DeleteMsg()函数设置EventType为NULL，指向消息的EventPtr指针为NULL，信号量计数值清零。

2.3.2.3 发送消息

函数的伪代码如下：

```
unsigned int SendMsg(unsigned int taskid, unsigned int *msgptr, unsigned int msgid)   (1)
{
    if (msgid>MAXEVENT)
        return(ERR_INVALID_EVENT_ID);
    if(taskid>MAXTASK)
        return(ERR_INVALID_TASK_ID);
    if(WatiList[msgid] [taskid]==NULLMESG)
        return ERR_MSG_SENDED;
    if (ECB[msgid].EventType!=MSG)
        return(ERR_EVENT_TYPE);
    if (msgid==WatiList[msgid] [taskid])                                    (2)
    {
        WatiList[msgid] [taskid]=NULL;
        ECB[msgid].EventPtr=msgptr;                                         (3)
        GetEvent(msgid)
        return (NO_ERROR);
    }
    else if(NULL!= ECB[msgid].EventPtr)                                     (4)
        return ERR_MSG_EXSITING;
        else
        ECB[msgid].EventPtr=msgptr
    return (NO_ERROR);
}
```

(1) 函数参数：taskid是接收该消息的任务ID号，msgptr 是指向消息的指针，msgid是消息的ID号。返回值：反映函数执行结果的错误码

(2) 如果在事件等待列表中有任务在等待这个消息，则清除该任务的等待状态，调用GetEvent()函数使任务得到这个消息。

(3) 消息的任务控制块中的EventPtr指针指向这个消息。等待这个消息的任务就可以通过这个指针找到消息的具体内容。

(4) 如果没有任务在等待该消息，检查ECB中是否为空。由于ECB中只能保存一则消息，所以如果ECB中已存有消息则返回错误信息，否则将新的消息指针保存。

2.3.2.4 等待消息

函数的伪代码如下：

```
unsigned int   ReceiveMsg (unsigned int msgid ,unsigned int **msgptr)        (1)
{
   if (msgid>MAXEVENT)
        return(ERR_INVALID_EVENT_ID);
    if(IntNesting>0)
    return ERR_INT_RECEIVE_MSG:
    if (ECB[msgid].EventType!=MSG)
        return(ERR_EVENT_TYPE);
    if(NULL！=ECB[msgid].EventPtr)                                            (2)
    {
        pTcb[CurrentTaskId].TaskMsgPtr = ECB[msgid].EventPtr;
        ECB[msgid].EventPtr =NULL;
        WatiList[msgid] [CurrentTaskId]=NULL;

   }
    else
      {
        WaitEvent(msgid);                                                     (3)
       }
}
```

(1) 函数参数：msgid是消息的ID号，msgptr返回指向消息的指针。

(2) 如果ECB中有消息，则取出消息并传递给当前任务，同时在等待事件队列中

移除当前任务，然后将ECB中的EventPtr指针重新设为NULL。任务获得消息后就具备了继续执行的条件，因此没有必要执行任务调度过程。

(3) 如果ECB中没有消息，则调用WaitEvent()函数使任务将进入等待状态，等待另一个任务或者中断服务子程序发出该消息。由于当前任务要等待一个消息的到来才能继续执行下去，因此WaitEvent()函数调用任务调度函数执行任务调度，以寻找当前可以运行的最高优先级的任务。

2.3.3 信号量

信号量是用于控制对共享资源的访问，在任务之间实现同步和互斥的一种手段。从程序实现上看，信号量包含了一个用于指示资源使用情况的计数器和一个等待该资源的任务队列。

每个信号量都对应一个共享资源，当计数器的值为一个不为零的非负整数时，表明当前该共享资源是可用的，等待该信号量的任务可以获得相应资源的使用权，然后将该计数器的值减 1。如果等待该信号量的任务发现计数器的值为 0，则表明该共享资源正忙，任务将加入到等待该信号量的任务队列。当任务使用完共享资源后，任务将该信号量的计数值加 1，然后释放先前获取的信号量。

2.3.3.1 创建信号量

在使用信号量之前，必须先创建信号量并初始化信号量的计数值。如果信号量是用于对一个共享资源的访问那么该信号量的初始值应设为1。如果信号量用来表示允许任务访问n个相同的资源，那么该信号量的初始值设为n。

函数的伪代码如下：

```
unsigned int   CreateSem (unsigned int semid, unsigned int cnt)          (1)
{
    if (emIntNesting>0)                                                  (2)
        return ERR_INT_CREATE_SEM;
    if (msgid>MAXEVENT)
        return(ERR_INVALID_EVENT_ID);
    ECB[msgid]. EventType=SEM;                                           (3)
    ECB[msgid].EventCnt=cnt;
    ECB[msgid]. EventPtr=NULL;
    CleanWaitEventList(msgid);                                           (4)
    return NO_ERROR;
```

```
}
```

(1) 函数参数：semid 是信号量的ID号,cnt是信号量的初始值。返回值：反映函数执行结果的错误码。

(2) 中断服务子程序不能调用CreateSem( )函数，只能在任务级代码或者多任务启动前建立新号量。

(3) 初始化ECB中的变量，设置事件类型EventType为SEM；根据用户输入设置信号量的初始值EventCnt为cnt；初始化信号量型事件中EventPtr为NULL。

(4) 调用CleanWaitEventList()函数清空等待该信号量的任务队列。

2.3.3.2 删除信号量

函数原型：unsigned int DeleteSem(unsigned int msgid);

函数参数：semid是信号量ID号。

返回值：反映函数执行结果的错误码。

函数的代码与CreateSem ( )类似，只是对ECB的操作相反。DeleteSem( )将ECB的事件类型域设置为NULL，信号量计数值清零，清空等待该信号量的任务队列。在删除信号量之前，操作系统需要先判断是否还有任务在等待该信号量，只有在没有任务等待该信号量时才能删除这个信号量，否则返回错误信息提示用户。

2.3.3.3 等待信号量

函数的伪代码如下：

```
unsigned int WaitSem ( EM_EVENT *pevent )                    (1)
{
   if (emIntNesting>0)                                       (2)
        return(ERR_PEND_ISR);
   if (msgid>MAXEVENT)
        return(ERR_INVALID_EVENT_ID);
    if(IntNesting>0)
        return ERR_INT_RECEIVE_MSG:
    if (ECB[semid].EventType!=SEM)
        return(ERR_EVENT_TYPE);
    if(ECB[semid].EventCnt>0)                                (3)
    {
        ECB[semid].EventCnt--;
```

```
            WatiList[msgid] [CurrentTaskId]=NULL;
        }
        else
          {
            WaitEvent(semid);                                    (4)
          }
    }
```

(1) 函数参数：semid是信号量的ID号。

(2) 在中断中不能等待信号量。

(3) 信号量的计数值非0说明该信号量所管理的共享资源当前是可用的，于是将此信号量的计数值减1后，使需要该共享资源的任务得到该信号量并继续运行。

(4) 如果信号量的计数值为0，说明该共享资源当前不可用。WaitEvent()函数使任务将进入等待状态，以等待另一个任务或者中断服务子程序释放该信号量。当前任务被阻塞后，WaitEvent()会调度就绪任务队列中的最高优先级任务开始运行。

2.3.3.4 释放信号量

函数的伪代码如下：

```
unsigned int SendSem(EM_EVENT *pevent)                           (1)
{
  if (semid>MAXEVENT)
        return(ERR_INVALID_EVENT_ID);
  if(taskid>MAXTASK)
      return(ERR_INVALID_TASK_ID);
    if (ECB[msgid].EventType!=SEM)
        return(ERR_EVENT_TYPE);
    if (semid==WaitList[msgid] [taskid])                         (2)
    {
        WatiList[semid] [taskid]=NULL;
        ECB[msgid].EventCnt++;
        GetEvent(semid)
        return (NO_ERROR);
    }
  else
```

```
        ECB[msgid].EventCnt++;                                    (3)
        return (NO_ERROR);
    }
```

(1) 函数参数：pevent是指向ECB的指针, taskid是接收该信号量的任务ID号。

(1) 如果有任务在等待该信号量，则调用GetEvent()使任务得到这个共享资源。

(2) 如果没有任务在等待该信号量，则该信号量的计数值就简单地加1，不进行任何调度工作。

2.4 中断响应处理和中断级调度

中断是一种硬件机制，用于通知 CPU 某个异步事件的发生。当发生中断时，CPU 内核自动打断正在运行的程序，调用专门的中断服务函数来处理异步事件。当中断处理结束时，CPU 内核恢复先前的现场信息并继续执行正常的程序流程。

在嵌入式操作系统中，中断处理过程同样是由操作系统负责管理的。受操作系统管理的中断可以分为以下三个步骤：

1、进入中断：系统首先要保护所有的中断现场信息，然后通知操作系统。嵌入式操作系统为中断嵌套的层数定义了一个全局变量 IntNesting，记录当前中断的嵌套深度。

2、用户中断服务程序：完成具体的中断处理。这部分代码通常由用户自己定义，除了完成本中断服务子程序的功能代码外，还包含了系统通信和同步函数的调用等与其他任务通信或同步的功能。通过调用通信和同步函数使得相应的任务得到信号量或者消息，从而进入就绪状态。

3、退出中断：在操作系统管理的中断过程中，中断在退出之前必须执行规定的退出流程。当退出中断时，嵌入式操作系统首先判断 IntNesting 的值：如果 IntNesting 不为 0，则返回被中断了的较低优先级的中断服务程序，然后将 IntNesting 值减 1；如果 IntNesting 为 0，说明所有被嵌套的中断都已经完成，嵌入式操作系统系统调用中断级任务调度和切换函数，判断是否有更高优先级的任务就绪。如果有更高优先级的任务被中断唤醒而进入了就绪态，则返回到更高优先级的任务，否则返回被中断的任务。

以上进入中断和退出中断的处理流程是固定的，而且与具体的 CPU 结构有关。因此，将进入中断和退出中断的处理过程作为操作系统的一部分，而将用户中断服务程序独立出来，提供一个调用接口以方便用户实现自己的应用功能。

2.4.1 中断处理函数

中断服务子程序ISR( )的伪代码如下：

```
void ISR(void)
{
    DisInt;
    依次将W工作寄存器阵列（W0-W15）、SR、SPLIM、TBLPAG、PSVPAG、
    RCOUNT、CORCON寄存器的内容推入当前任务堆栈；                    (1)
    IntNesting++;                                                  (2)
    if(是时钟节拍中断)
      {
       清零定时器Timer0的中断标志；
       EnInt;                                                      (3)
       重新给定时器Timer0赋初值；
       Tick( );
      }
    可以在此处插入其他中断的处理程序；
    if (IntNesting>0)
       IntNesting--;                                               (4)
    if (IntNesting==0)
       {
       从被中断的任务的TCB中恢复栈指针；                           (5)
       IntSched( );                                                (6)
       }
    EnInt;
    执行中断返回指令；
}
```

(1) 保存中断现场。

(2) 通过将IntNesting加1，通知操作系统内核进入中断服务子程序。

(3) 如果允许中断嵌套，可以重新开中断，因为操作系统内核可以跟踪中断嵌套层数IntNesting。

(4) 将中断嵌套层数减1。当减到0时，所有嵌套的中断就都完成了。

(5) 当中断返回到最外层嵌套时，从被中断任务的TCB中恢复栈指针，将栈空间再从中断嵌套堆栈切换回原来的任务堆栈。

(6) 调用中断级调度函数IntSched( )判断有没有更高优先级或同优先级的其他任务被中断服务子程序或任一级的中断嵌套唤醒而进入就绪态，或者是否需要对同优先级的任务进行时间片轮转调度，函数的详细描述见下一小节。

2.4.2 中断级任务调度和切换函数

函数的设计思路与任务级调度函数 TaskSched( )基本相同，函数的代码如下：

```
void IntSched (void)
  {
     FindMaxPrioTask();
     if(找到优先级高于当前运行任务的就绪任务)
       {
          将当前运行任务的状态转换为被中断态;          (1)
          IntTaskSw( );                              (2)
       }
     else
          不执行任务切换;
  }
```

(1) 这里的处理方式与任务级调度不同。当查找到更高优先级的任务后，原先运行的任务将转为被中断态，而不是挂起态。

(2)调用专门的中断级的任务切换函数。

中断级切换函数IntTaskSw( )与任务级切换函数TaskSw( )的设计思路有相似之处，主要区别在于中断服务子程序已经将CPU寄存器的值（中断现场）存入到了被中断的任务的堆栈中，因此无需再做第二次。函数的伪代码如下：

```
void   IntTaskSw(void)
{
     取当前任务的堆栈指针;
     获取待运行任务的堆栈指针;
     从待运行任务的堆栈中依次将W工作寄存器阵列(w0-W15)、SR、SPLIM、
     TBLPAG、PSVPAG、RCOUNT、CORCON寄存器的内容恢复到CPU内
     核寄存器;
```

从任务堆栈中恢复待运行任务的程序计数器PC到PC寄存器；

执行中断返回指令；

}

2.5 系统时间管理

在嵌入式系统中，延时和超时控制等时间相关的功能是很常用的，因此操作系统必须提供一些时间相关的内核函数来支持应用程序对时间相关服务的要求。

时间管理函数以系统时钟节拍为处理单位。时钟节拍就是特定的周期性中断，这就需要在嵌入式操作系统的设计中提供周期性的信号源，从而产生对微控制器的周期性中断请求。每次时钟中断可以看做是一个抢占剥夺点，当时钟中断到来时，如果有高优先级的任务就绪，则操作系统进行任务调度抢占当前任务的CPU 使用权。当然，由于时钟节拍是由定时中断提供的，所以节拍的频率越高系统的额外开销也就越大。根据不同的应用，通常将时钟节拍取为 10-200ms。

本设计中，采用 PIC24 微控制器内置的定时器 Timer0 作为周期性中断源，时钟节拍率为每秒 100 次，时间间隔为 10ms。

用户程序必须在多任务系统启动以后再启动时钟节拍源计时，也就是在调用TaskStart( )之后做的第一件事是初始化定时器中断。否则，时钟节拍中断有可能在系统启动第一个任务之前发生，此时系统是处在一种不确定的状态之中，用户应用程序有可能会崩溃。

系统中的时钟节拍服务是通过在时钟中断服务子程序中调用Tick( )实现的。在中断服务子程序中处理时钟节拍中断时，调用Tick( )，由它执行具体操作。

2.5.1 任务延时函数

为了等待某个事件的完成，任务经常需要延时一段时间。嵌入式操作系统提供了一个系统函数TaskDly()，任务可以调用这个函数使任务延时一段时间。延时时间的长短是用时钟节拍的数目来表示的。调用延时函数后，操作系统将进行一次任务调度，挂起当前任务从而去执行就绪任务队列中优先级最高的任务。

函数的伪代码如下：

```
void TaskDly (unsigned int taskid, unsigned int ticks)          (1)
{
    if (ticks>0)
    {
```

```
        if(RUNNING!=pTcb[taskid].TaskStatus&&READY==
            pTcb[taskid].TaskStatus)
            return ERR_DLY_INVALID_TASK;                    (2)
        pTcb[taskid]. TaskStatus=SUSPEND                    (3)
        pTcb[taskid]. TaskDlyTime=ticks
        pHtb[taskid].htstatus.IDtimer.Flg=HARD_TIMER_WORK;  (4)
        pHtb[taskid].Lt_value= ticks;
        pHtb[tid].temp=mSeconds;
        if(RUNNING==pTcb[taskid].TaskStatus)                (5)
            {
             TaskSched();
            }
        }
    }
```

(1) 函数参数：taskid 是被延时任务的ID号，ticks是延时节拍数。

(2) 首先判断被延时任务的状态，因为只有处于运行态或者就绪态的任务才能被延时，被中断、被删除和已经被挂起的任务是不能被延时的，也没有意义。

(3) 设置TCB中TaskStatus为SUSPEND以挂起任务，TaskDlyTime记录延时时长。

(4) 在定时器控制块HTB中记录该任务被延时、延时的时间长度、定时器计数值。

(5) 如果当前任务被挂起，执行任务调度寻找当前优先级最高的就绪任务。否则不需要执行任务调度。

2.5.2 恢复延时任务

ResumeDlyTask()函数可以取消延时，使被延时的任务不等待延时期满而重新运行。函数原型为ResumeDlyTask(unsigned int taskid)。ResumeDlyTask()的代码与TaskDly()类似，只是对任务控制块TCB和定时器控制块HTB的操作相反。

(1) ResumeDlyTask()函数首先根据taskid判断要恢复的是不是当前任务，因为当前任务正在运行中没有被延时。

(2) 通过检测TCB的TaskStatus位是否为SUSPEND，以及HTB中的标志位来判断任务是否确实是已被延时。

(3) 如果任务确实被延时，则将任务状态TaskStatus位转变为就绪态，DlyTime设置为0。同时清除HTB中该任务被延时标志位和延时时间变量。

(4) 最后，调用任务级调度函数重新进行任务调度。如果被恢复的任务的优先级比当前运行任务的优先级高则进行任务切换，否则继续执行当前任务。

2.6 系统启动与初始化过程

在基于嵌入式实时操作系统的软件设计中，必须按照一定的规则来创建main函数。系统启动过程典型代码如下

```
void main(void)
{
 HardwareInit();                    (1)
 CreateTask();                      (2)
 CreateMesg();                      (3)
 CreateSem();
 SetHardTimer();                    (5)
 StartTask();                       (6)
}
```

(1) 首先调用 HardwareInit()函数初始化所需的硬件模块。在嵌入式系统中，除了CPU 负责运算功能外，还有许多外围模块，例如：定时器、A/D 转换器、串行接口等，也都是完成系统功能所必不可少的部件。开发人员要在系统中使用这些外围模块，必须事先对这些外围模块进行正确的配置。例如：使用通用异步收发器（UART）之前必须配置好波特率、数据位、校验位、停止位等控制信息，这些配置都是通过设置外围模块的控制寄存器来完成的。因此，在嵌入式系统上电或者复位启动后，必须有一个硬件初始化的过程。HardwareInit()函数就是为开发人员预留的用于初始化硬件模块的系统函数。软件开发人员可以重写 HardwareInit()函数，在 HardwareInit()函数中完成所需的初始化硬件模块的代码。

(2) 调用CreateTask()函数创建所有的任务。前面讲到，任务是嵌入式实时操作系统管理和调度的最小单位，在系统启动之前至少要建立一个任务。由于调用一次CreateTask()函数只能创建一个任务，所以开发人员要多次调用CreateTask()函数逐一创建所有的任务。

(3) 调用CreateMesg()和CreateSem()函数创建消息和信号量。

(4) 调用SetHardTimer()函数设置定时器。

(5) 调用StartTask()函数启动多任务调度过程。多任务启动函数只在系统启动时被

调用一次，一旦多任务系统开始运行则不会再次被调用。除非系统重新启动，重新进入初始化和启动过程。

2.7 程序调试与测试

对于软件开发人员来说，编写代码只是整个开发过程的第一步，在代码编写完成后还要对程序进行调试和测试。通常来说，在软件开发过程中最耗费时间和精力的工作是程序的调试和测试而不是代码的编写过程。这是因为程序编译通过只能代表代码没有语法错误，并不能保证程序能够正确地、可靠地完成所需的功能。要确保程序能够最终实现功能必须靠对程序进行调试和测试来完成。本章所设计的程序在调试过程中同样出现了很多的问题，经过不断的修改和完善才最终完成了该程序。程序的调试是一个需要细心和耐心的工作，只有在认真地观察和思考的前提下借助软件工具认真分析才能最终发现和解决问题。

由于篇幅所限，下面仅举两个在调试过程中有代表性的关键问题为例：

1. 内存溢出错误

在调试过程中出现了这样的问题：内存中软件堆栈在不停地长，以至于最后超出了存储空间而产生堆栈溢出错误。以如下的一段任务代码为例：

```
void Task1(void)
    {
      unsigned char i=1,m=0;                    (1)
      任务功能代码;                             (2)
      TaskSched();                              (3)
     }
```

(1) 任务中定义了两个局部变量i和m。由2.12节介绍可知，PIC24的CPU采用软件堆栈结构。当Task1()函数被调用时，CPU分配一段内存作为该函数的堆栈空间，局部变量i和m就保存在这段堆栈空间中。

(2) 执行该任务的功能代码。

(3) 任务功能执行完成后，调用消息或者信号量函数重新执行任务调度过程。在TaskSched()函数被调用时，该函数的断点地址也被压入 Task1 的软件堆栈空间中。

TaskSched()查找最高优先级的就绪任务，然后完成任务切换过程。TaskSched()函数中只完成了当前任务和待运行任务的堆栈切换，所以这部分的压栈和出栈是平衡的，但是此时在第一步和第三步中分配的局部变量 i、m 和 TaskSched()的断

点地址所占用的内存空间并没有被释放的情况下就退出了 Task1 函数。因此，随着程序的运行，这部分空间就会逐渐积累，最终导致内存溢出。

解决方法：在 TaskSched()函数中加上 ulnk 指令。ulnk 指令用于解除帧指针，使帧指针重新指向上次未处理完的函数。本例中执行 ulnk 指令后，帧指针重新指向了 Task1()函数的软件堆栈，也就是说堆栈指针重新指向了存放变量 i、m 的位置。在随后的压栈过程中，原存放变量和断点指针的位置被覆盖，而接下来的出栈与压栈又是平衡的，从而避免了内存软件堆栈溢出。

2. 定时器中断错误

在程序调试运行过程中发现时钟节拍函数在运行一段时间后会出错，无法产生时钟中断。经调试发现：定时器进入中断 21 次后，无法再进入中断。由于系统的时间函数都是以定时器中断作为时间节拍，所以当定时器无法进入中断后，系统的时间函数随之出错。

问题原因：PIC24 微控制器 CPU 中的 SR 寄存器存放着内核优先级控制位，假如设定为 4，那就意味着优先级低于 4 外部中断将不会触发中断。当内核优先级设置为 7 时，将屏蔽所有的外部中断。在进入中断时，CPU 为了防止另一个中断源触发中断会自动将 SR 中的内核优先级控制位设定为 7，在本中断结束后，再将 SR 恢复为用户设定值。由于 SR 寄存器值是任务控制块中的一个成员，所以在进入中断时，SR 被保存在了任务控制块中，而此时的 SR 寄存器内的内核优先级控制位已被 CPU 置为 7。在下次任务调度中，为了能够从断点处重新开始执行，原先被保存的各寄存器的值被恢复出来。由于先前保存在任务控制块中的 SR 内核优先级位的值为 7，所以 SR 寄存器的内核优先级位被恢复为 7，于是导致外部中断的产生被阻止了。

解决方法：在任务控制块中，不再保存 SR 寄存器状态。其实在中断时，CPU 会自动将 SR 的值在被改变前保存到 PCH 的高字节。在中断结束时，RETFIE 指令会自动将 PCH 的高字节返回到 SR 寄存器中。

第三章 Bootloader模块的设计与实现

启动引导加载程序（Bootloader）是指系统上电启动后所运行的第一段程序代码，也就是说Bootloader是运行在应用程序之前的一段程序。其主要任务就是将应用程序加载到程序存储区中。于是嵌入式开发人员可以摆脱交叉编译环境和硬件编程器，而仅仅利用串口等常用通讯接口与嵌入式设备相连接，然后利用Bootloader软件就可以方便地对嵌入式设备重新编程，从而实现软件升级。

所有的微控制器生产商都为其处理器产品提供相应的交叉编译环境和专用的硬件编程器/调试器，例如：美国Microchip公司为其PIC24系列微控制器提供的MPLAB IDE开发软件和MPLAB ICD2编程器/调试器。利用这些工具在实验室环境开发和调试嵌入式设备无疑是高效和方便的，然而一旦该嵌入式设备被部署到了实际的应用环境中，当需要重新编程或者升级软件时，这些工具就变的不再实用。特别是有大量的嵌入式设备需要升级软件时，利用编程器逐个设备地重新编程更是一个极其困难的任务。

在这种情况下，利用Bootloader软件通过串口甚至以太网接口重新加载应用软件就成为最高效和方便的方法。特别是通过总线对大量嵌入式设备实现软件升级更是事半功倍。另一方面本文设计的嵌入式操作系统还不够成熟，在进一步开发中必然还需要反复修改和升级，所以在开发嵌入式实时操作系统内核的同时开发了相应的Bootloader程序，方便了该嵌入式实时操作系统和应用软件的继续开发和升级。

在嵌入式领域，Bootloader程序是严重依赖于硬件的，不可能实现一个通用的Bootloader软件。因此，本章针对PIC24系列微控制器设计和实现了一个专用的Bootloader软件，经实践测试该程序是稳定和可靠的。

3.1 Bootloader模块概述

在嵌入式系统的开发过程中，通常是利用微控制器生产商提供的交叉编译工具（如：Microchip 公司提供的 MPLAB IDE）在 PC 机上用 C 语言或者汇编语言编写程序，然后将源程序编译成微控制器能够识别的二进制文件，最后通过编程器/调试器等硬件工具将编译后的二进制文件下载到微控制器的程序存储区中。PIC24 系列微控制器的开发过程也是一样的，C 语言或者汇编语言编写的源程序

在经过编译器编译后会在当前工程目录下生成一个以.hex 为后缀的文件，这就是将要下载到微控制器中的应用程序的二进制文件。

本文设计开发的 Bootloader 程序就是要摆脱相关的硬件编程器，通过串口、以太网或其他通用通讯接口，将这个 hex 文件从 PC 机传输到嵌入式设备，然后自动载入到微控制器内部的 Flash 程序存储器中，从而实现应用程序的更新升级。

Bootloader 程序可以分为两个部分：

- 运行在 PC 机上的 Client 程序。该程序是供用户使用的，其作用是根据用户的指令，解析和校验 hex 文件内容，然后通过通讯接口将 hex 文件发送到用户指定的一个或一组目标嵌入式设备。
- 运行在嵌入式设备上的 Server 程序。该程序是被固化在嵌入式微控制器的程序存储器中的，其作用是与 PC 端的 Client 模块通讯，根据从 PC 端接收到的命令，接收和解析 hex 文件并校验，然后将该二进制文件写入嵌入式微控制器的 Flash 程序存储器，从而实现软件的在线更新。

PIC24 系列微控制器内部集成了 UART（通用串行收发器），因此本设计中采用串口（如：RS-232 或者 RS-485）作为 PC 机端 Client 程序与嵌入式设备端 Server 程序的通讯方式。某些 PIC24 系列微控制器的高端型号也内建了 CAN 总线收发器，对于这种器件也可以采用 CAN 总线方式通讯。另外，随着 Intenet 时代的到来，在嵌入式设备中加入以太网接口变得越来越普遍，因此在系统资源和产品成本要求可以满足的情况下，也可以采用 TCP/IP 方式实现通讯。

在本设计中，嵌入式设备端的 Server 程序用 MPLAB IDE 交叉编译器开发，尽管 PIC24 系列微控制器程序存储区容量可达 16KB，但在 Server 程序设计过程中仍然着力减少其程序空间的消耗。其中读写程序存储区等硬件相关部分用汇编语言编写，其他硬件无关部分用 C 语言编写。PC 机端的 Client 程序采用 Microsoft 公司的 Visual C++开发。

3.2 嵌入式设备端程序的设计与实现

3.2.1 Server程序的总体规划

嵌入式设备端Server程序结构如图3.1所示：

Server程序采用PIC24微控制器内置的串口控制器作为通信方式，通过串口实现数据包的收发操作。Server程序根据从串口收到的命令执行相应的操作，如：读/写程序存储区、读器件ID号、执行编程等。为了控制Server程序的各种操作行

为，在程序中定义了一些相关的命令：

```
#define NACK      0x00    //接收错误应答命令
#define ACK       0x01    //接收成功应答命令
#define WritePM   0x02    //写程序存储器命令
#define WriteCM   0x03    //写器件配置位命令
#define Reset     0x04    //重启命令
#define ReadID    0x05    //读设备ID号命令
```

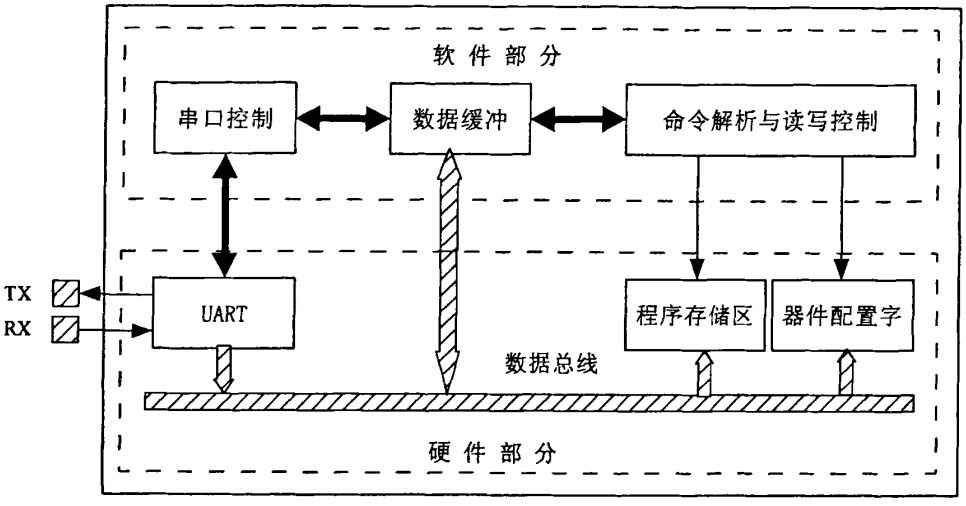


图 3.1 嵌入式端程序结构

Server程序收到待写入程序存储区的数据后并不会立即执行编程操作，而是将数据暂存在RAM中的数据缓冲区内，待收到相应的执行编程命令后再按照特定的Flash程序存储区读写时序进行编程操作。数据缓冲区的另一个作用是通过错误重传机制保证数据的正确性。在工业现场，由于干扰等因素造成通信数据错误是无法避免的，在这种情况下只能采用错误重传的机制来保证所有数据的正确传输。同时，PIC24微控制器程序存储器的编程操作有着严格的时序要求，如果不采用数据缓冲区方式，必然会出现由于等待某个数据包错误重传的时间过长而导致编程时序错误和编程失败，可见数据缓冲区的设置是十分必要的。

Bootloader实现软件在线升级实际上是利用了Flash程序存储器的运行时自编程（RTSP）能力。PIC24 器件有4M x 24 位Flash程序存储器空间。PIC24程序存储器阵列是由多行组成的，每行64 条指令（192 字节），每8行（512条指令）称为1页。RTSP 可以让用户一次擦除8 行（512 条指令），也可以一次编程1行（64 条指令）。从程序存储器起始地址处开始，每8 行擦除页和1行写块分别在

1536 字节和192 字节的边界上边缘对齐。整个Flash程序存储器被划分为用户程序空间和配置空间两部分。用户程序空间（000000h 至7FFFFFh）包含复位向量、中断向量表和程序存储器。器件配置寄存器和器件ID 单元映射在配置空间中，可通过对闪存配置字中的期望值编程来置1 或清零配置位。

在PIC24系列微控制器的程序存储区中，中断向量表（IVT/AIVT）固定地占用了地址从0x000到0x01FF的程序存储空间。但是Bootloader的Server程序并不能紧接着中断向量表存储在0x0200到0x03FF位置，这是因为这段空间与中断向量表处于同一Flash页内，当Server程序擦除Flash程序存储区中的第一页时，将同时擦除这段空间的内容。所以，将Server程序安排在地址0x400开始的程序存储空间内。基于同样的原因，用户的应用程序同样不能存放在紧接着Server程序的程序存储

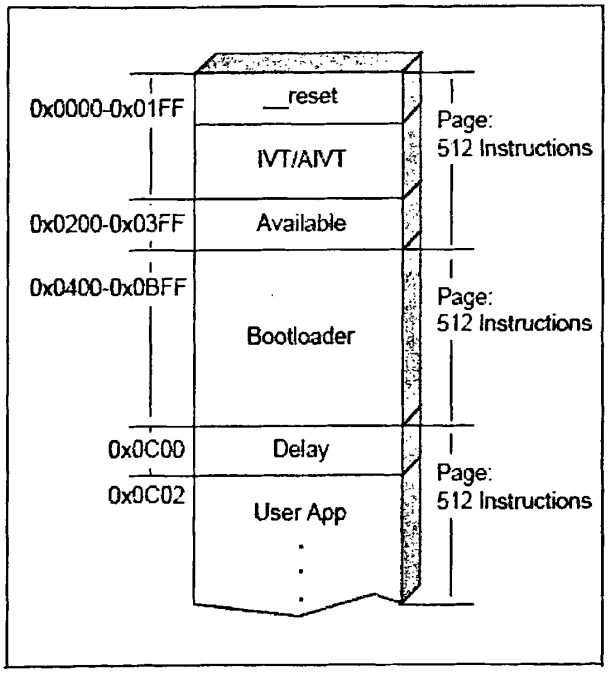


图3.2 程序存储区分配

区内，而必须放在起始地址为0xC00的下一个Flash页开始的空间里。Server程序的启动延迟时间（其值为0到256）存储在0xC00地址处，将Flash程序存储区中0xC02开始的存储空间作为真正的用户应用程序的存储区。

综上所述，PIC24微控制器的Flash程序存储区分配结构如图3.2所示：

当嵌入式设备启动时，Server程序首先读取地址0xC00处存储的数值，如果该值为1到256中的某个数值，例如：X则Server程序等待X秒，在此期间Server程序

不断查询串行通讯端口，以检测是否接收到PC端Client程序发来的Bootloader命令。如果收到，则进入Bootloader过程，如果在定时时间段内没有收到Bootloader命令，则程序跳转到0xC02地址，开始执行应用程序。如果Server程序读取到0xC00地址处存储的数值为0，则Server程序将永远等待下去，直到收到PC端Client程序发来的Bootloader命令。

3.2.2 Server程序的软件实现

3.2.2.1 Server程序流程

嵌入式设备端Server程序流程图如图3.3所示：

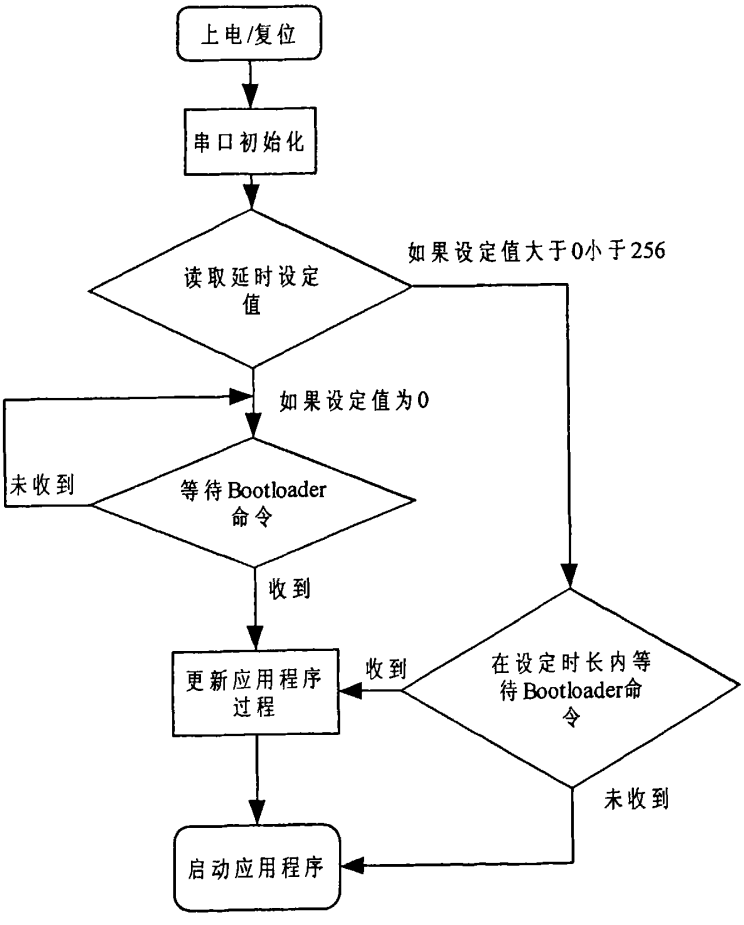


图 3.3 Server程序流程图

PIC24系列微控制器的程序存储器采用的是Flash存储器，Bootloader的Server程序就是利用了Flash程序存储器的可多次读写的特性。首先将Flash程序存储器中原来的内容擦除，然后将从通信端口接收到的二进制程序代码写入0xC02开始的程序存储器中，从而实现了软件自编程。

3.2.2.2 读、写Flash程序存储器

Server程序通过对Flash程序存储区的重新编程实现软件的更新，而在Flash程序存储器的编程过程中，最基本的操作就是读出和写入数据。由于Flash程序存储器的读写操作有严格的时序要求，尽管C30编译器支持C语言直接操作CPU内核寄存器，但是C30编译器无法保证编译后的程序为原子操作，也就无法保证能够满足时序的严格要求，因此本段程序采用汇编语言编写。

程序存储空间由可字寻址的块（block）构成。尽管程序存储空间被视为24 位宽，但可以将它的每个地址看作低位和高位字，高位字的高位字节未实现（以0填充）。低位字总是偶地址，而高位字总是奇地址。程序存储器地址在低位字上总是字对齐的，所以执行代码时地址总是增2或减2。

下图为程序存储器的结构图

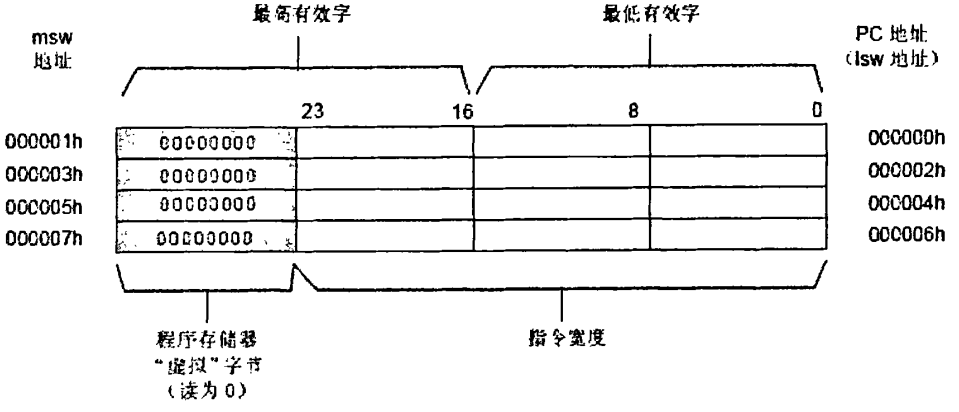


图 3.4 程序存储器的结构

下面分别介绍读出和写入两个基本的操作：

◆ 使用表读指令读程序存储器

表读需要两个步骤。首先，使用TBLPAG 寄存器和一个W 寄存器存器建立一个地址指针。然后读取特定地址单元的程序存储器内容。

1、字模式读取程序如下

```
//首先建立指向程序存储空间的指针
MOV #tblpage(PROG_ADDR),W0      //载入程序存储区页地址
MOV W0,TBLPAG                   //将页地址载入表页地址指针寄存器
MOV #tbloffset(PROG_ADDR),W0    // 载入程序存储区偏移地址
//读取该指针指向的程序存储区
TBLRDH [W0],W3                  //读取高字节到 W3 寄存器
```

```
TBLRDL [W0],W4                    //读取低字节到 W4 寄存器
```

2、字节模式读取程序

```
//首先建立指向程序存储空间的指针
MOV #tblpage(PROG_ADDR),W0      //载入程序存储区页地址
MOV W0,TBLPAG                    //将页地址载入表页地址指针寄存器
MOV #tbloffset(PROG_ADDR),W0   // 载入程序存储区偏移地址
//读取该指针指向的程序存储区
TBLRDH.B [W0],W3                 //读取高字节到 W3 寄存器
TBLRDL.B [W0++],W4               //读取低字节到 W4 寄存器
TBLRDL.B [W0++],W5               //读取中字节到 W5 寄存器
```

◆ 使用表写指令写程序存储器

表写指令不会直接写非易失性程序存储器，而是装入用来存储待写入数据的保持锁存器。当所有保持锁存器都被装入后，通过执行一个特殊的指令序列即可开始实际的存储器编程操作。

1、以字模式写入一个程序存储器锁存单元

```
//读取将要写入的数据到 W 寄存器
MOV #PROG_LOW_WORD,W2
MOV #PROG_HI_BYTE,W3
TBLWTL W2,[W0]                   //将数据写入锁存器
TBLWTH W3,[W0++]
```

2、以字节模式写入一个程序存储器锁存单元

```
//首先建立指向程序存储空间的指针
MOV #tblpage(PROG_ADDR),W0      //载入程序存储区页地址
MOV W0,TBLPAG                    //将页地址载入表页地址指针寄存器
//读取将要写入的数据到 W 寄存器
MOV #LOW_BYTE,W2
MOV #MID_BYTE,W3
MOV #HIGH_BYTE,W4
//将数据写入锁存器
TBLWTH.B W4,[W0]                 //写高字节到锁存器
```

```
TBLWTL.B W2,[W0++]          // 写低字节到锁存器
TBLWTL.B W3,[W0++]          //写中字节到锁存器
```

3.2.2.3 Flash程序存储器的编程操作

对微控制器内部闪存进行编程操作必须使用完整的编程序列对Flash存储器和相关寄存器进行操作，编程操作持续时间的标称值为4 ms，处理器将暂停直到编程操作完成。

程序存储器有一个保持缓冲器，可包含64 条指令的编程数据。实际编程操作前，必须将要写入的数据连续装入保持缓冲器后再执行编程。装入的指令字必须来自64 个边界的组合。

RTSP 编程的基本步骤是先建立一个表指针，然后执行一系列表写指令（TBLWT）将数据装入保持缓冲器。编程操作是通过将控制寄存器（NVMCON）寄存器中的控制位置1 ，然后由CPU自动执行的。执行一次编程总共需要执行64条TBLWTL 和TBLWTH指令。

用户可以一次性地对程序存储器的一行进行编程。要实现编程，必须先擦除包含要编程的行的8 行擦除块。一般过程如下：

1. 读取程序存储器的8 行（512 条指令），并存储在数据RAM 中。
2. 用所需的新数据更新RAM中的程序数据。
3. 擦除该区块：

a） 将NVMOP 位（NVMCON<3:0>）设置为0010 以配置块擦除。将ERASE（NVMCON<6>）和WREN （NVMCON<14>）位置1。

b） 将要擦除的块的起始地址写入TBLPAG 和W 寄存器。

c） 将55h 写入NVMKEY。

d） 将AAh 写入NVMKEY。

e） 将WR 位置1 （NVMCON<15>）。擦除周期开始，在擦除周期中CPU会暂停。当擦除结束时， WR 位会自动清零。

4. 将数据RAM 中的前64 条指令写入程序存储缓冲器。
5. 将程序块写入闪存存储器：

a） 将NVMOP 位设置为0001 以配置行编程。将ERASE 位清零，将WREN 位置1。

b） 将55h 写入NVMKEY。

c） 将AAh 写入NVMKEY。

d） 将WR 位置1。编程周期开始，在写周期中CPU 会暂停。写入闪存存储器结束时，WR位会自动清零。

6. 通过递增TBLPAG 中的值，使用数据RAM 中的下一批可用的64 条指令重复步骤4 和5，直到全部512 条指令写回闪存存储器中。

3.3 PC端程序的设计与实现

3.3.1 Client程序的软件界面

PC端Client程序采用Visual C++编写，采用PC机的串口与嵌入式设备端的Server程序通信。PC端的Client程序运行界面如下图所示：

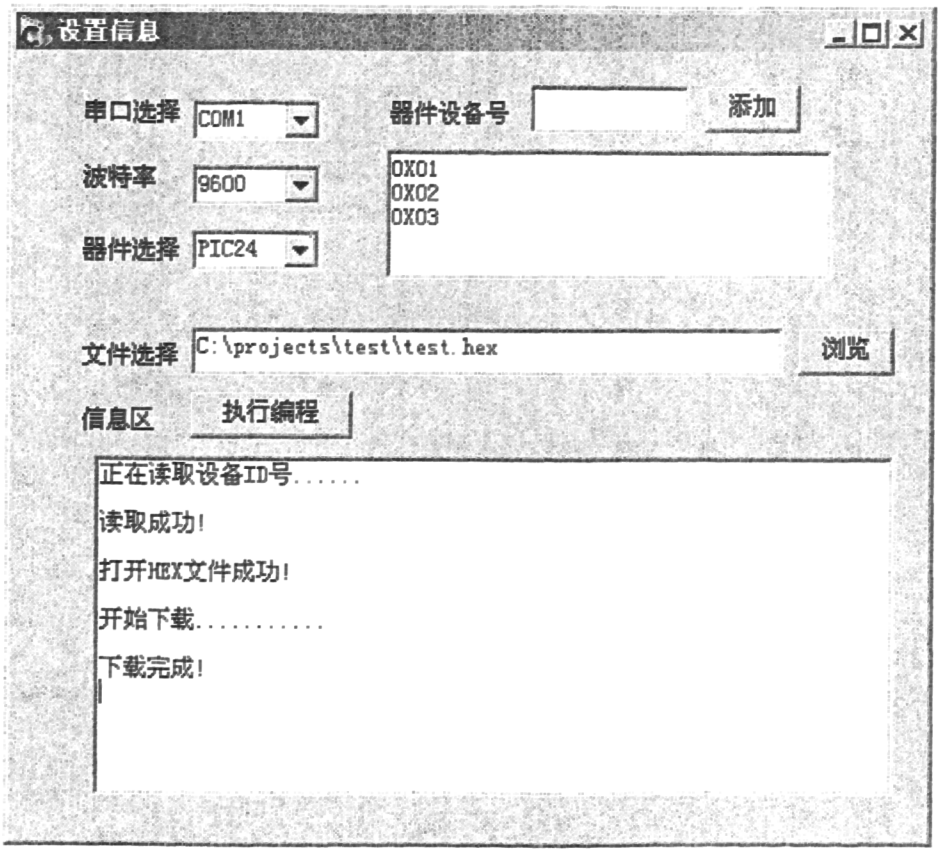


图 3.5 Client程序运行界面

3.3.2 Client程序的软件实现

如 3.1 节中所述，Client 程序的主要功能是根据用户的指令，解析和校验 hex 文件内容，然后通过通讯接口将 hex 文件发送到用户指定的一个或一组目标嵌入式设备。因此，Client 程序除了用户界面外还有两个主要的功能模块，一个是串口通信模块，另一个是 Hex 文件解析模块。Client 程序流程图如下：

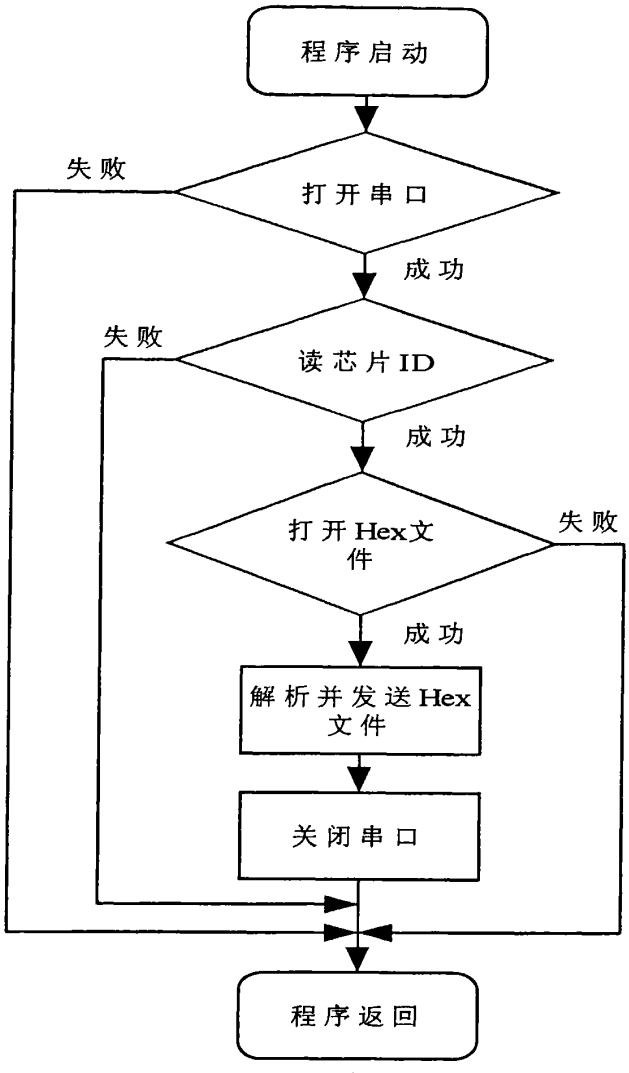


图 3.6 Client 程序流程图

下面分别介绍这两个主要功能模块的实现过程。

3.3.2.1 通讯模块

上一小节描述了PC机端Client程序的软件界面。本节介绍该Client程序的设计和实现过程。

如前所述，PC机端的Client程序采用串口与嵌入式设备端的Server程序实现通信。在Windows系统中，实现串口通信编程通常有如下四种方式。

- 使用Win32 API实现：通过调用Windows SDK提供的API函数接口实现。
- 使用Active X控件：采用Microsoft公司提供的MSComm控件实现。
- 直接嵌入汇编语言：此种方法不能在Windows NT系统下使用，只能在Windows98系统下，利用C/C++嵌入汇编的功能实现。

◆ 通过VXD或者WDM驱动程序实现：该方法较复杂也很少采用。

本文中Client程序采用的是第一种方法，即通过Windows系统的SDK提供的API接口函数方法实现。在32位的Windows系统中，串口和其他通信设备是作为文件来处理的。串口的打开、关闭、读取和写入所用的函数与操作文件的函数完全一致。

下面分别介绍串口编程相关的主要API函数。

1、打开串口

CreateFile()函数打开串口，该函数返回一个句柄。

函数原型： HANDLE CreateFile(LPCTSTR lpszName,DWORD fdwAccess,DWORD fdwShareMode,LPSECURITY_ATTRIBUTES lpsa,DWORD fdwCreate,DWORD fdwAttrsAndFlags,HANDLE hTemplateFile)

参数：lpszName：指定要打开的串口名，如 COM1、COM2； fdwAcess：指定串口访问类型；fdwShareMode：指定串口的共享属性；lpsa：引用安全属性结构（SECURITY_ARRTIBUTES）； fdwCreate：指定如果 CreateFile（）函数正在被其他程序调用时应采取的动作；fdwAttrsAndFlags：描述端口的属性；hTemplateFile:指向模板的句柄。

返回值：指向已经打开串口的句柄。

2、设置发送和接收缓冲区

SetupComm()函数分配和初始化发送和接收缓冲区。

函数原型:BOOL SetupComm(HANDLE hFile,DWORD dwInQueue,DWORD dwOutQueue)

参数：hFile：串口句柄；dwInQueue：输入缓冲区的大小；dwOutQueue：输出缓冲区大小。

返回：成功True，失败False

3、获取串口当前配置参数

BOOL GetCommState()函数用来获取串口当前的配置参数。

函数原型：BOOL GetCommState（HANDLE hFile， LPDCB lpDCB）

参数：hFile：串口句柄； lpDCB：串口设备对应的设备控制块，该控制块描述了串口设备的各种参数和属性

返回值：成功True，失败False

4、设置串口参数

当需要更改串口的配置属性时，可以先通过GetCommState()函数获得当前的设备控制块（DCB），更改相应参数后，调用SetCommState()函数重新配置串口。

函数原型：BOOL SetCommState（HANDLE hFile，LPDCB lpDCB）

参数：hFile：串口句柄；lpDCB：串口设备对应的设备控制块。

5、缓冲区操作

PurgeComm()函数用来对缓冲区进行操作。

函数原型：BOOL PurgeComm（HANDLE hFile，DWORD dwFlags）

参数：hFile：串口句柄；dwFlags：执行的操作，如清除缓冲区、终止发送/接收操作。

6、读取串口数据

读取串口数据可以用ReadFile()函数，像读取文件一样从串口读取数据。

函数原型：BOOL ReadFile（HANDLE hFile，LPVOID lpBuffer，DWORD nNumberOfBytesToRead，LPDWORD lpNumberOfBytesRead,LPOVERLAPPED lpOverLapped ）

参数:hFile：串口句柄；lpBuffer：指向接收缓冲区的指针

nNumberOfBytesToRead：要从串口读取的字节数；

lpNumberOfBytesRead：调用该函数实际读出的字节数；lpOverlapped:指向OVERLAPPED结构的指针。

7、从串口发送数据

从串口发送数据实际就是写串口缓冲区操作。调用WriteFile()函数像写文件一样向串口写入数据。

函数原型：BOOL WriteFile（HANDLE hFile，LPCVOID lpBuffer，DWORD nNumberOfBytesToWrite，LPDWORD lpNumberOfBytesWritten，LPOVERLAPPED lpOverlapped）

参数：hFile：串口句柄；lpBuffer：指向存放待发送数据缓冲区的指针；nNumberOfBytesToWrite：要向串口写入的字节数；lpNumberOfBytesWritten：调用该函数以写入的字节数；lpOverlapped：指向OVERLAPPED结构的指针。

在本设计中，将串口操作的相关API函数封装到了四个函数中：

◆ HANDLE OpenConnection (HANDLE *pComDev, char *pPortName, char *pBaudRate);

函数功能：打开和设置用户选定的串口，初始化数据缓冲区。

参数：pComDev是指向该串口的句柄；pPortName是端口名（COM1或者COM2）；pBaudRate是用户设置的波特率。

返回值：函数成功则返回 pComDev，函数失败则由assert函数强制终止程序。

函数伪代码如下：

```
HANDLE OpenConnection(HANDLE * pComDev, char * pPortName, char *
                      pBaudRate)
{

    sscanf(pBaudRate, "%d", &BaudRate);
    *pComDev = CreateFile(pPortName, GENERIC_READ |
                          GENERIC_WRITE, 0, NULL, OPEN_EXISTING,
                          FILE_ATTRIBUTE_NORMAL |
                          FILE_FLAG_OVERLAPPED, NULL);
     if(*pComDev == NULL)
    {
        return (*pComDev);
    }
    SetCommMask(*pComDev, EV_RXCHAR);
    SetupComm(*pComDev, BUFFER_SIZE, BUFFER_SIZE);
    PurgeComm(*pComDev, PURGE_TXABORT | PURGE_RXABORT |
PURGE_TXCLEAR | PURGE_RXCLEAR);
    初始化IO模块;
    assert(SetCommTimeouts(*pComDev, &CommTimeOuts) == TRUE);
    Dcb.DCBlength = sizeof(DCB);
    assert(GetCommState(*pComDev, &Dcb) == TRUE);
    初始化DCB块;
    assert(SetCommState(*pComDev, &Dcb) == TRUE);
    return (*pComDev);
}
```

◆ BOOL WriteCommBlock (HANDLE *pComdDev, char *pBuffer , int

BytesToWrite);

函数功能： 将数据写入串口发送缓冲区。

参数：pComDev是指向该串口的句柄；pBuffer是指向将要发送的数据的指针；BytesToWrite是数据字节数。

返回值：成功返回True，失败则终止程序。

函数伪代码：

```
BOOL WriteCommBlock(HANDLE * pComDev, char *pBuffer , int
                                BytesToWrite)
{
  if(WriteFile(*pComDev,pBuffer,BytesToWrite,&BytesWritten,&osWrite)
     == FALSE)
   {
      assert(GetLastError() == ERROR_IO_PENDING);
      return (FALSE);
   }
   return (TRUE);
}
```

◆ int ReadCommBlock (HANDLE *pComdDev, char *pBuffer, int MaxLength);

函数功能：从串口读取数据。

参数：pComDev是指向该串口的句柄；pBuffer是指向接收数据缓冲区的指针；MaxLength是数据字节数。

返回值：函数执行成功则返回读取的数据长度，失败则终止程序。

函数代码如下：

```
int ReadCommBlock(HANDLE *pComDev, char * pBuffer, int
                          MaxLength )
{

   ClearCommError(*pComDev, &ErrorFlags, &ComStat);
   Length = min((DWORD)MaxLength, ComStat.cbInQue);
   if(Length > 0)
   {
```

```
        if(ReadFile(*pComDev, pBuffer, Length, &Length, &osRead) ==
            FALSE)
        {
            Length = 0 ;
            ClearCommError(*pComDev, &ErrorFlags, &ComStat);
            assert(ErrorFlags == 0);
        }
    }
    else
    {
        Length = 0;
        Sleep(1);
    }
    return (Length);
}
```

- BOOL CloseConnection(HANDLE *pComdDev);

  函数功能：关闭串口。

  参数：pComDev是指向该串口的句柄。

  返回值：成功True，失败False。

  函数代码如下：

```
BOOL CloseConnection(HANDLE * pComDev)
{
PurgeComm(*pComDev,PURGE_TXABORT | PURGE_RXABORT |
            PURGE_TXCLEAR | PURGE_RXCLEAR);
return(CloseHandle(*pComDev));
}
```

3.3.2.1 HEX文件解析模块

软件开发人员用C语言或者汇编语言编写的应用程序，经过C30编译器编译后会生成相应的HEX文件，该文件中包含了以十六进制格式数据存储的编译后的源程序、器件配置信息等内容。PC机端的Client程序在通过串口发送HEX文件之前，首先要对该HEX文件的内容进行解析。

HEX文件解析和发送模块程序流程图如图3.7所示：

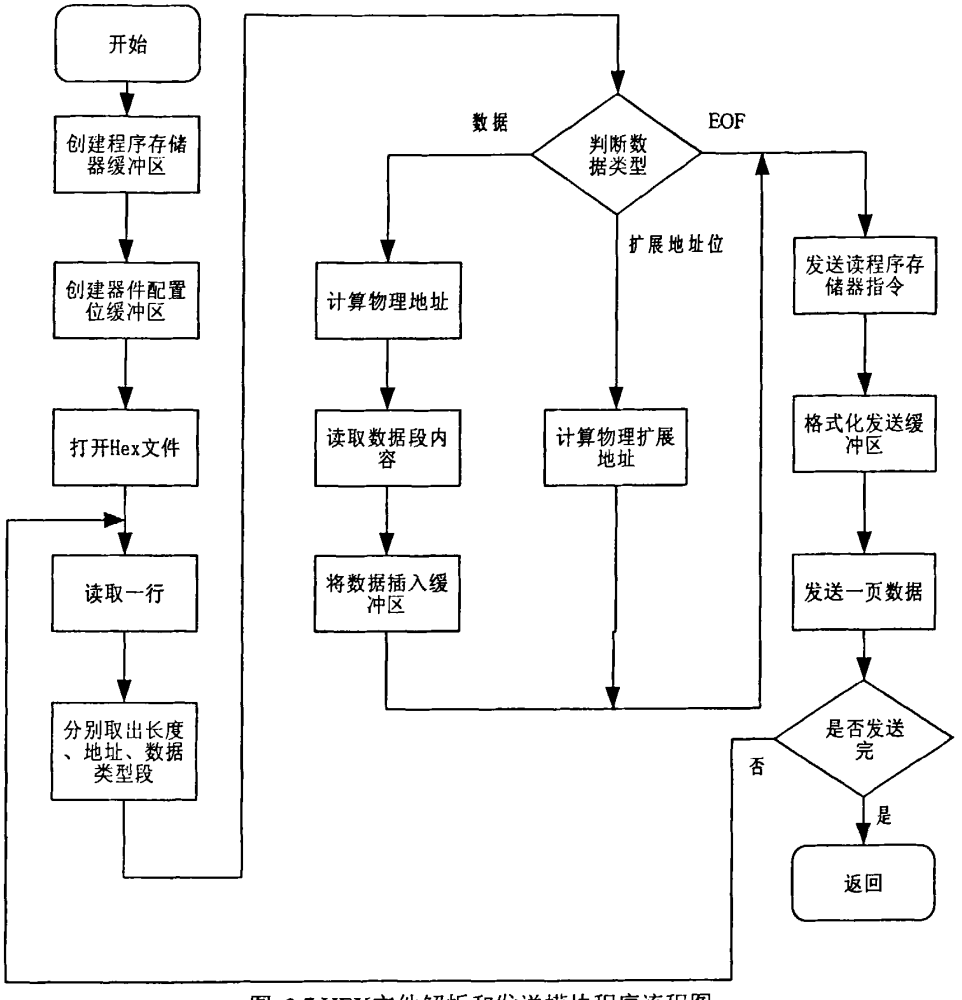


图 3.7 HEX文件解析和发送模块程序流程图

标准HEX文件的基本格式如下：

```
:BBAAAAATTHHHH...HHHHCC
```

可见每一行的数据都是由一个由9个字符组成的前导符(:BBAAAATT)开始，并以2个字符的校验和（CC）结束。

- 前导符的第一个符号为冒号“：”，标志着一行的开始。
- BB：该符号是一个两位的十六进制数，指示了该行的数据长度。
- AAAA：该符号是一个四位的十六进制数，指示了该行数据的起始地址。
- TT：该符号是一个两位的十六进制数，指示了该行的数据类型。“00”表示该行为数据，“01”表示文件结束符（EOF），“04”表示该行为扩展地址记录
- HHHH：数据段，即编译后的程序或者器件配置位等信息。

◆ CC：该行所有数据的校验和。

HEX文件解析和发送模块首先为程序存储区和器件配置位区开辟对应大小的缓冲区，然后打开用户选定的、待发送的HEX文件，读取第一行。HEX文件中每一行记录的前导码中包含有数据长度、程序存储器地址和数据记录类型字段等信息。Client程序将这些内容取出后进行判断，如果该行记录为结束符（EOF）则直接发送当前发送缓冲区中的内容。如果该行记录为为扩展地址，则计算实际的物理地址。如果该行记录为数据，则取出数据段内容。HEX文件将编译后的程序按照一定的格式存储在HEX文件的数据段中，为了使数据段内容能够被微控制器正确写入到程序存储器，必须在通过串口发送这些数据前，按照程序存储器的编程结构重新组织数据，并将格式化后的数据保存到发送缓冲区中。当解析完512条指令（即Flash存储器一页数据）后，通过串口发送缓冲区中的数据。重复以上过程直至所有的数据都发送结束，程序返回

3.4 Bootloader与操作系统的结合

根据前面的介绍知道，只有在系统重启后才会执行 Bootloader 程序。通常来说使系统重启可以通过两种方式：一种是利用电路板上的复位按钮或者给设备重新上电的手工方式；另一种是在程序中调用 RESET 汇编指令使微控制器自动重启。本章设计 Bootloader 程序的目的就是要通过自动化的方式来减少开发人员的工作量，显然第一种复位方式是不可取的，只能采用第二种方式。

在用户应用程序运行过程中，整个系统都是在嵌入式实时操作系统的管理之下的。因此，在嵌入式实时操作系统中必须有一种机制能够响应 Client 程序向嵌入式设备端发送 Bootloader 命令这样一个异步事件。

由于串行数据接收是在串行接收中断中处理的，因此在串行接收中断中增加一段功能以判断接收到的数据是否是 Bootloader 命令。如果收到 Bootloader 命令则调用汇编指令 RESET 以重启系统。

```
Void UartISR(void)
{
    if(接收到 Bootloader 命令)
        {
          SaveUserData();                                    (1)
          asm volatile ("RESET");                            (2)
```

```
        }
    else
     执行其他操作;
}
```

(1)由于 PC 端 Client 程序随时可以发送 Bootloader 命令，而此时嵌入式设备正在嵌入式实时操作系统的控制下运行用户的应用程序。所以调用 SaveUserData()函数保护相关的数据，例如可以将当前数据采集过程中的数据暂时写入非易失存储器。开发人员可以重写 SaveUserData()函数的代码以完成所需的处理工作。

(2)采用在 C 代码中插入行内汇编的方式调用汇编指令。MPLAB C30 编译器支持在 C 函数中使用 asm 语句将一行汇编代码插入到编译器生成的汇编语言中。使用下面的语法在 C 代码中插入汇编指令：asm ("instruction");。另外，可通过在 asm 后写关键字 volatile 避免 asm 指令在编译和优化过程中被编译器删除、移动或组合。

第四章 嵌入式实时操作系统的工程应用

本章以山东省肿瘤防治研究院能量管理系统中的能量检测控制器的设计为例，介绍嵌入式实时操作系统在工程中的应用。

为了使人们的工作生活更加舒适，现代大型建筑中大量安装了中央空调、装饰照明灯具等高耗能的电器设备，使得建筑能耗大大增加。近年来，能源短缺问题已经日益严重，节能技术也随之越来越受到世界各国政府和科研机构的重视。因此，如何有效降低建筑能耗也逐渐成为节能技术研究的热点之一。

目前建筑节能技术的研究大多着眼于新型建筑材料的研发和单个耗能设备的结构改进上，缺乏一种宏观的、整体的、系统的观念，忽略了能量的使用是为人服务的以及有多少能量被真正有效利用这个关键问题。因此，单纯考虑能耗是不全面的，应该以“能量”的观点来考虑能量与人的利用率之间的相互关系。

如果以“能量”的概念从宏观上抽象建筑中所有耗能元素（如：采暖、空调、照明、炊事、热水等），可以将能耗分为能量传输效率和能量利用效率。

能量传输效率是指用户获得的总能量与输出总能量之比，用来表示传输到房间、大厅等人们活动场所的能量是否为人充分利用。比如若房间中没有人，此时空调系统仍在运行，这时的能量利用效率就为零，或虽然有人，但输入的能量大于人们的需求，超出需求的部分能量利用效率也为零。若全部为人所用，没有浪费则能量利用效率为 100%。对于能量传输效率，若以 Eo 表示空调系统和照明系统的总能量输出，E 表示输送到用户处的总能量，那么能量传输效率 η_t 可用下式表示：

$$\eta_t = \frac{E}{E_0} \times 100\%$$

能量利用效率与人的活动紧密相关，首先用离散的方法表示建筑物中不同区域的能量利用效率，然后再将其求和获得能量利用效率。由于能量的使用还与时间有关，所以能量利用效率应针对不同时间段进行统计。为了简单起见，若能量被人们利用则设利用效率为 1(100%)，否则为 0，中间状态暂不考虑。若将建筑物中的区域最多分为 n 个，时间段以小时为单位，那么能量利用效率可用下列公式表示：

$$\delta_{kt} \in (1,0) \qquad \text{其中} \quad k \in (0,n) \quad t \in (0,23)$$

$$\eta_{ut} = (\sum_{0}^{n} \delta_{kt} \div n) \times 100\% \quad \text{其中} \quad k \in (0,n) \quad t \in (0,23)$$

如果能量全部被利用了，那么能量传输效率和能量利用效率为 100%，事实上这是不可能的。那么怎样来提高能效呢？能量传输效率的提高主要通过技术手段就可以实现。能量利用效率是与空调系统和照明系统的管理和运行模式有关，这部分效率的提高不仅仅要依靠技术进步，更重要的是通过能量管理、能量调度才能提高能量的利用效率。

能量管理系统就是这样一套完整的基于嵌入式系统的针对建筑物内空调和照明系统能效运行状况的检测和管理系统，通过对能效的分析并结合建筑物自身的特点，对建筑物能量使用状况进行检测。根据检测的结果，通过与国际上相关标准相比较得到建筑的节能潜力。再根据系统检测到的空调和照明系统中能耗设备运行效率和评估得到的空调照明系统能量转换效率，提出节能的指导建议和具体的实施方案。也就是说能量管理系统的设计出发点是在保证人们舒适度的前提下如何提高能量的利用效率，而不是单纯地降低能量的总消耗数量。

能量管理系统主要由三个子系统组成：

1.能量检测控制器

能量检测系统是整个能量管理系统的基础和依据。通过对各种能量信息的采集，系统可以实时地获得实际的能量使用状况。能量检测系统通过安装在典型区域内的流量、温度、湿度、照度、感知等传感器获取当前温度、湿度、照度等各种环境数据以及区域内人员活动情况，并将这些数据进行初步的分析和处理后提供给能量管理系统作为各种控制策略的依据。本章重点介绍该子系统的软件设计。

2.能量处理服务器

能量处理系统主要负责管理分布在一定区域内的能量检测控制器，收集由能量检测控制器采集到的数据，经处理后发送给能量管理控制中心。同时根据能量管理控制中心的节能控制策略，控制对应的能量检测控制器调节耗能设备的运行状态。该子系统的设计不做介绍。

3.能量管理控制中心

能量管理控制中心包括中央空调水系统效率检测与管理模块、中央空调风系统能效分析模块和远程运行指导模块等。能量管理系统采用检测系统获得的能量

感知数据，分析人员活动状况与温度湿度变化之间的关系，结合相关的控制算法即可判断当前的能量利用效率，并提出可行且具有较好投资回报周期的节能措施。该子系统是运行PC机上的一套管理软件，用Visual Basic软件编写。

能量管理系统的结构图如图4.1所示：

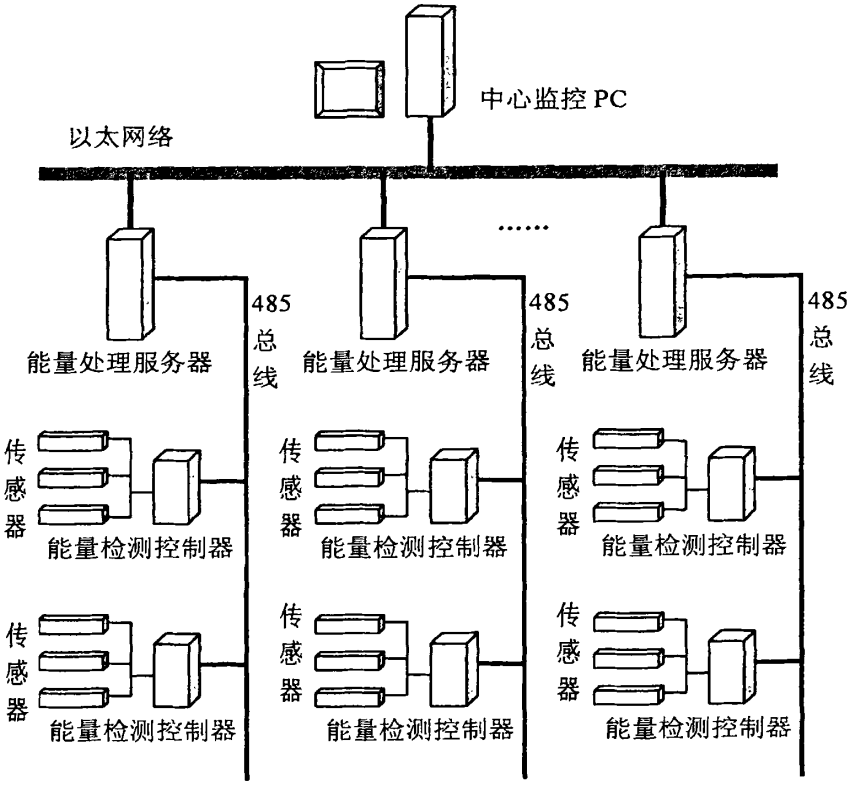


图 4.1 能量管理系统结构示意图

根据能量管理系统的设计思想，能量最终是为人服务的，因此能量检测控制器需要检测建筑物内能量的使用情况和人的活动状态。在现代大型建筑中，整个建筑物根据应用的需求被分割为不同功能和不同面积的房间。以山东省肿瘤防治研究院门诊医技综合楼为例，该建筑门诊、医技检查、科研、办公为一体，总建筑面积26785平方米。设地下一层，地上十三层，地下一～地上四层为裙楼，地下一层为放射科及楼内变电室，一层为急诊门诊及药库等用房，二层为肿瘤内/外科、放疗科、特检科等，三层为检验科与中心实验室等，四层为门诊手术室、查体中心、学术报告厅等；地上五到十三层为主楼，五层一八层为医院办公用房，九层－十层为领导办公层，十一、十二为客房层。可见，在每个房间内安装能量

检测控制器成本会很高，而且也没有必要。最合理的安装方式就是采用典型区域的思路，综合考虑房间的用途、楼层、面积和朝向等因素，选取若干房间作为典型区域安装能量检测控制器，以典型区域内采集到的数据代表该类型房间能耗状况和人的活动情况。

分布在各个典型区域内的能量检测控制器定时采集温度、湿度、照度和红外感知共四路传感器信号。红外感知传感器利用红外线探测的原理，通过记录一段时间内红外传感器输出的开关量信号即可知道当前房间内的人员活动状况。温度、湿度、照度传感器分别采用热敏电阻 pt100、湿敏电容和光敏电阻 PGM5506 以获取当前房间内的环境状况数据。这样，能量检测控制器既获取了反映房间内能量使用状况的环境数据，又得到了人员活动的相关信息。能量检测控制器通过 RS-485 总线与负责管理该片区所有能量检测控制器的能量处理服务器通讯。能量处理服务器定时发送查询命令，逐个轮询被管理的能量检测控制器。能量检测控制器收到查询命令后将采集到的数据处理和打包后发送给能量处理服务器。能量处理服务器将收到的数据通过以太网发送给能量管理控制中心，作为控制中心分析能耗状况和做出节能决策方案的依据。除了环境状况数据检测外，能量检测控制器还具备控制耗能设备运行的功能。能量检测控制器硬件上以继电器作为数字量输出模块，可以控制空调系统的风机盘管、冷冻水泵、新风机组、风阀、冷却塔、变频器、照明电源线路等设备。本文重点介绍基于嵌入式实时操作系统的软件设计，因此对于节能控制策略和对耗能设备的具体控制过程不再赘述，仅以对中央空调器风机盘管的控制为例简单代表。

4.1 硬件设计

能量检测控制器结构

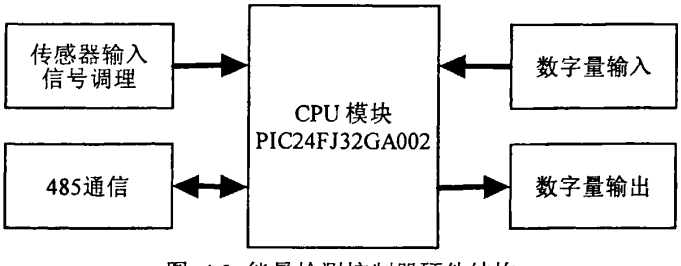


图 4.2 能量检测控制器硬件结构

1.CPU 模块

采用 PIC24FJ32GA002 微控制器，该 CPU 与本文所设计的嵌入式实时操作系统的开发调试平台是相同的。

2.传感器与输入信号调理模块

该模块连接所有传感器，如：温度、湿度、照度、红外感知传感器，对传感器的输出信号进行调理和滤波。

3.数字量输入/输出

数字量输入模块采用光耦采集数字量传感器信号，输出模块采用继电器控制阀门、照明电源线路等。

4.RS-485 通信模块

485 通信模块用来与能量处理服务器通信。

4.2 嵌入式实时操作系统的应用

4.2.1 基于操作系统的软件设计

4.2.1.1 划分任务

由第二章介绍可知，在基于嵌入式实时操作系统的软件设计过程中，首先要做的就是划分任务。任务划分的合理与否对系统的运行效率、实时性和吞吐量影响极大。任务分解过细会引起任务频繁切换，增加系统开销。而任务分解不够彻底会造成原本可以并行的操作只能按顺序串行完成，从而减少了系统的吞吐量。为了达到系统效率和吞吐量之间的平衡与折衷，在应用设计中进行合理的任务分解非常重要[9]。通常来说，在考虑如何将整个系统功能划分为不同的任务时有以下几个原则：

- 各个任务所依赖的周期条件具有不同的频率和时间段。
- 各个任务的功能相对独立。
- 由于通常的输入输出设备的运行速度比 CPU 运行速度低，所以输入输出相关的操作单独划分为一个任务。操作系统可以充分利用等待输入输出完成的时间调度其他任务执行，从而提高系统运行效率。
- 关键性的、实时性要求高的功能单独作为一个高优先级的任务。
- 消耗时间较多的数据处理过程划分为一个任务。
- 若干按固定时序完成的功能合并为一个任务，以减少任务间的通讯量。

根据以上设计原则，首先可以将 485 通信过程划分为一个任务。因为 485 通信功能是一个输入输出过程，只有在收到能量处理服务器的查询命令后才会启动通信功能。PIC24 微控制器内建有串行收发控制器，CPU 不需要等待和控制耗时的串行收发过程，只需将待发送数据存入发送缓冲区就可以被操作系统重新调度

去执行其他就绪任务了，而具体的发送过程由串行收发控制器自动完成。查询命令是由能量处理服务器主动发出是一个异步的过程，能量检测控制器并不知道何时会收到查询命令，所以将接收数据过程放在串口接收中断中处理。

能量检测控制器最主要的功能就是环境数据和人员活动状况的采集。数据采集后还要进行一定的处理，而且数据采集和数据处理功能关联性较大。这里的“关联性”包括两个含义：一是数据采集和数据处理操作的同一组数据对象；二是数据采集和处理在运行时序上有先后关系。如果将这两个关联性较高的功能划分为两个任务，则实时操作系统必须在这两个任务之间执行反复的、频繁的数据同步和任务通信操作，无形中增加了实时操作系统的开销。虽然从理论上来说，数据采集功能的实时性要求高一些，而数据处理相对耗时多，实时性较低，但是考虑到本项目中的数据处理功能没有很复杂的计算过程，实际上并不会影响数据采集功能的实时性。因此最好将数据采集和处理功能合并为一个任务，这样减少了这两个功能之间的切换和通信开销，通过合理安排两个功能在同一个任务中的运行顺序也避免了实时操作系统对共享数据的同步操作。可见，将数据采集和处理划分为一个任务是高效的、合理的。

能量管理的最终目的是通过控制耗能设备的运行实现节能，所以风机盘管的控制功能是系统的一个关键任务，而且独立于其他功能，其执行定时周期也与其他功能不同。因此将对风机盘管的控制功能做为一个单独的任务。

本设计中所有的任务都是定时执行的，且定时周期各不相同。数据采集任务由固定的时间定时器启动，485 数据通信任务只有接收主设备的查询命令才启动数据发送。任务之间没有固定的事情发生先后顺序，比如采集完数据后并不是马上就要执行数据的通信，因此任务完成并将自己挂起后，程序跳转到什么位置去执行是需要考虑的问题。采用一个任务专门负责对所有定时节拍的管理，当定时时间到时，该任务调度执行相应的任务。因此，设计一个底层循环任务作为主任务，在此任务中将不同的事情交给其他任务去完成。设计这么一个底层循环任务，它可以使系统在处理完一个任务后回到这个默认任务中，查询时间节拍状态，等待下一个任务的执行。

综上所述，在本设计中将整个系统功能划分为 4 个任务：底层消息循环任务、数据采集任务、风机盘管控制任务和 485 通信任务。

4.2.1.2 分配优先级

任务划分完后，下一步需要考虑的就是各个任务的优先级分配问题。通常来说，在考虑优先级分配时有以下几个原则：

- 中断关联性：与中断服务子程序（ISR）相关联的任务通常实时性要求较高，应该安排尽可能较高的优先级以及时响应异步事件的发生。
- 关键性：任务越关键则优先级越高，以保障其执行机会
- 传递性：数据传递路径上的上游任务的优先级应该高于下游任务，例如数据采集任务的优先级应该高于数据处理任务的优先级。
- 快速性：运行耗时较短的任务应当安排较高的优先级，以减少其他就绪任务的等待时间。

首先，在已经划分好的四个任务中，485 通信任务是由串口接收中断触发的，当 485 通信任务收到能量处理服务器发来的查询命令时，必须及时做出响应，在最短的时间内将数据发送出去，否则将会影响能量处理服务器对下一个能量检测控制器的查询。而且串行数据的收发过程是绝对不应该被其他任务占先的，否则会导致遗漏接收内容或者发送错误的数据。可见 485 通信任务是最紧迫的、实时性要求也最高，因此将 485 通信任务的优先级设为 1，使之成为最高优先级任务。这样即使在 485 任务运行过程中有别的任务进入就绪态，也不会抢占 485 通讯任务的运行，保证了数据收发过程的正确完成。

再考虑风机盘管控制任务和数据采集处理任务。风机盘管控制任务定时每 5 分钟执行一次，数据采集和处理任务的定时周期为 2.7 秒，可见数据采集和处理任务的执行要比风机盘管控制任务更加频繁。但是如果将数据采集和处理任务的优先级高于风机盘管控制任务，则有可能造成风机盘管控制任务在一段时间内长期得不到执行的机会。根据能量管理理论，能量最终是为人服务的，无论数据采集处理还是节能策略的实施都是为了使人获得舒适的工作生活环境和降低能耗，而调节室内环境和节能就是通过控制诸如风机盘管、照明电源线路、冷冻水泵这些设备的运行来实现的。可见风机盘管控制任务是整个系统的关键任务，必须保证其运行的机会。所以风机盘管控制任务的优先级应该高于数据采集和处理任务。所以设定风机盘管控制任务的优先级为 3，仅次于 485 通信任务。

数据采集和处理任务也是定时执行的，其定时周期为 2.7 秒，与其他任务的定时周期不同。虽然通讯任务执行的一个前提条件是待发送的数据已经被完整采集和正确处理，但是实际上温度、湿度、照度等环境参数变化是一个缓慢的过程，

而且数据采集到后还要进行数据处理操作。所以在每天采集大量数据的前提下，即使数据采集任务遗漏个别数据也不会影响最终的节能策略的判断和执行。所以定义数据采集和处理任务的优先级为 6，低于风机盘管控制任务。

显然，作为系统默认任务的底层循环任务不负责任何实际功能的实现，因此优先级应该是最低的。设定该任务优先级为 9。

综上所述，485 通信任务、风机盘管控制任务、数据采集和处理任务以及底层循环任务的优先级分为为 1、3、6、9。可见这四个任务的优先级并不是连续分配的，这是基于可扩展性的考虑。本章所介绍的应用于山东省肿瘤防治研究院的能量管理系统已经是该系统的第二代产品，未来肯定还会继续研发功能更加完善、系统更加稳定的新产品。因此，在本项目中将各个任务的优先级安排比较宽松，在将来的开发过程中不需要修改现有任务的优先级分配方案就可以很容易地找到一个合适的空闲优先级，方便地添加更多的任务。这种设计方式也符合嵌入式实时操作系统设计中“可扩展性”的理念。

4.2.1.3 任务通信与同步

任务划分完毕、任务优先级分配完成后，下一个需要考虑的就是任务间的同步与通信机制。如第三章 2.3 节所述，在本文设计的嵌入式实时操作系统中，任务之间采用信号量的方式来保证对共享资源的有序访问，采用消息传递的方式来实现通信。

在本项目中，所有的任务都是定时执行且定时周期各不相同。这就要求当定时周期到时，实时操作系统发送特定的消息以唤醒对应的待执行的任务。第三章 2.5 节中讲到，本文设计的嵌入式实时操作系统中采用 PIC24 微控制器内置的定时器 Timer0 作为周期性中断源，产生用户所需的定时周期节拍。但是如果在定时器中断中进行计数，再判断用户所需定时是否到，然后分发特定的消息给任务，必然会导致中断处理过程耗时过长。这与中断处理子程序应当精简高效的设计原则是违背的，也必然会影响系统的实时响应能力。所以，在本项目中，将所有的判断用户定时周期是否到和分发特定的消息的功能都放在底层循环任务中。

数据采集任务、风机盘管控制任务和 485 通讯任务都要对环境状况数据进行操作。因此，应该有数据同步的机制来确保数据被正确地读写，但是由于各个任务间采用消息传递的方式进行通信，而且执行的定时周期各不相同，例如：只有当风机盘管收到特定的消息时，才会读取数据采集和处理任务活动的环境参数，

可见通过消息的传递机制已经保证的各个任务在彼此安全隔离的前提下进行数据交换。因此，在本项目中没有采用专门的数据同步机制。

4.2.2 程序实现

4.2.2.1 主程序设计

主函数的代码如下：

```
int main(void)
{
  SetTimer0();
  UartInit();
  ADInit();                                                    (1)
  //任务名，ID 号，优先级（数越小优先级越高），状态
  CreateTask(Task1,1,9,READY);      //底层
  CreateTask(Task2,2,6,READY);      //检测采样
  CreateTask(Task3,3,3,READY);      //盘管控制
  CreateTask(Task4,4,1,READY);      //SD 通信                    (2)

  CreateMesg(0,4);
  CreateMesg(1,4);
  CreateMesg(2,4);                  //创建消息                   (3)

  SetHardTimer(1,27);               //采样时间设定 2.7s
  SetHardTimer(2,10);               //485 接收定时 1s
  SetHardTimer(3,300);              //盘管控制 5 分钟定时 3000   (4)
  StartHardTimer(1);                //定时器触发                 (5)
  StartHardTimer(3);

  StartTask();                                                 (6)
  return 0;
}
```

(1) 由于在系统中要用 A/D 转换器做数据采集，用 UART 串行控制器完成 485 通信功能，所以在主函数中首先对硬件模块进行配置。在本文第二章 2.5 节时间管

理中提到，实时操作系统采用 PIC24 微控制器的 Timer0 作为时钟节拍源，因此在这里也要根据晶振频率设置 Timer0 的配置寄存器。

(2) 调用操作系统的系统函数 CreateTask（）创建所有任务：Task1 为底层循环任务，任务 ID 为 1，优先级为 4；Task2 为数据采集任务，任务 ID 为 2，优先级为 3；Task3 为风机盘管控制任务，任务 ID 为 3，优先级为 2；Task4 为 485 通信任务，任务 ID 为 4，优先级为 1。可见整个系统的优先级分配中，Task4（485 通信任务）>Task3（风机盘管控制任务）>Task2（数据采集任务）>Task1（底层循环任务）。

(3) 本设计中采用消息作为任务间通信的方式，又由本文第二章 2.3 节知道，在使用消息进行任务间通信之前，实时操作系统必须预先创建相应的消息，然后任务才能收发消息。一共有 3 个消息被创建出来：消息 0 为触发底层采样任务消息；消息 1 为触发盘管监测任务消息;消息 2 为触发与能量处理服务器通讯任务消息。

(4) 数据采集任务和风机盘管控制任务都是定时启动的，因此调用实时操作系统函数 SetHardTimer()分别设置三个定时器：

用于启动 Task2 的定时器 1（定时 2.7 秒）：感知传感器检测人的活动情况是利用红外探测原理记录其报警输出信号。报警信号的持续时间大约为 3s。因此，对感知传感器的数据采集设定其采样周期为 2.7 秒（小于 3 秒）。这样就不会遗漏红外感知传感器的输出信号。

用于启动 Task3 的定时器 3（定时 5 分钟）：Task3 是风机盘管控制任务，由于对风机的控制不宜过于频繁，否则会损害电机。而且空调调节室内温湿度是一个典型的大时滞过程，能量检测控制器执行风机调节动作后，必须经过一段时间后室内的温湿度环境才会反映出风机调节动作的效果，因此该任务的定时间隔不宜过短。目前暂定定时间隔为 5 分钟。

485 接收等待定时为定时器 2（定时 1 秒）：由前述可知，485 通讯任务是由串口通讯中断触发的，所以这个定时器并不是用于启动 485 通讯任务的。在开发调试的过程中，我们发现了这样的问题：在本项目中，由能量处理服务器主动向能量检测控制器发出查询命令，当 485 总线上能量检测控制器收到查询命令后，继续接收能量处理服务器发出的待查询能量检测控制器的 ID 号。能量检测控制器通过比较收到的待查询设备的 ID 号与自身 ID 是否匹配来决定是否向能量处理服务器上传数据。由于在空闲状态下，485 总线上的所有设备都处于接收状态，

所以被查询能量检测控制器上传数据时，其他的所有未被查询能量检测控制器也能收到数据。此时就有可能出现上传数据中的某个字节与某个能量检测控制器的ID号相匹配，从而错误触发该能量检测控制器的数据上传过程的情况。这种情况会干扰原先正常的数据上传过程。因此为了解决这个问题特别定义了一个2秒的定时器，当能量检测控制器判断发现待查询的设备ID号与自身不匹配时就启动该定时器，在接下来的2秒内禁止串口接收中断，等待2秒定时到后再打开串口。这样既避免了485总线上设备之间的相互干扰问题，也不影响下次查询。

(5) 启动定时器1和3。定时器2用于收到查询命令后定时等待，因此没有启动。

(6) 所有任务、消息和定时器都创建完成后，调用StartTask()启动多任务调度过程。

4.2.2.2 任务设计

1. Task1（底层循环任务）

该任务可以看做是整个系统的一个默认任务，Task1永远处在就绪状态，但不执行任何实际功能。在所有的任务中，Task1的任务优先级最低，所以只有在就绪任务队列中没有其他任何任务时才运行。由于其他任务都是定时执行的，所以在Task1中程序不断调用系统函数CheckHardTimer()轮询定时器，以判断其他任务的触发条件是否满足。当程序发现某个定时周期到时调用TxMessage()函数发送特定的消息给对应的任务，同时重启定时器以进入下一个计时周期。然后程序调用实时操作系统调度函数TaskSched()执行实时操作系统的调度过程，Task1让出CPU使用权，执行被触发的任务。当其他任务执行完毕后，程序再回到此任务继续循环，再去判断、触发下一个任务的发生。

程序伪代码：

```
void Task1(void)
{
  while(1)
  {
   if（定时周期到）
     {
       调用 TxMessage()函数发送特定消息给相应的任务；
       重启定时器；
     }
  TaskSched ();
```

```
    }
  }
```

2. Task2（数据采集与处理）

该任务完成对传感器输出数据的采集和处理工作。该任务执行定时周期为2.7秒，当2.7秒定时周期到时Task1将发送消息给Task2，Task2的任务状态将转换为就绪态。由于该任务优先级为2高于Task1的优先级，经过实时操作系统的任务调度后进入运行态。该任务调用 RxMessage()函数判断接收到的消息后，就执行具体的数据采集和处理功能。该任务执行完后，调用函数TaskSuspend()将自身挂起，然后再调用实时操作系统的调度函数TaskSched ()再次进行任务的调度。

程序伪代码：

```
void Task2(void)
{
   if(RxMessage(2))
   {
      /*执行该任务的功能*/
    }
    TaskSuspend ();
    TaskSched ();
}
```

3. Task3（风机盘管控制任务）

该任务完成对风机盘管的控制功能。该任务的基本程序流程与Task2 是相似的。当Task3收到5分钟定时到消息后，执行相应的功能，最后将自身挂起，重新进行任务的调度。

4. Task4（485通信任务）

该任务完成485通信功能，该任务的基本程序流程与Task2 类似。当Task4收到消息后，执行相应的功能，最后将自身挂起，重新进行任务的调度。需要注意的是，Task2和Task3的消息都是由Task1发出的，而Task4的消息是由串口接收中断发出的。能量检测控制器的485通信任务实际上是由能量处理服务器发出的数据查询命令触发的，因此这是一个异步的事件，而且上传数据的过程是一个高优先级的关键任务。在嵌入式实时操作系统中，中断过程是收到实时操作系统管理的，但是中断的发生是对异步事件的响应是不受实时操作系统控制的。因此，

为了及时响应查询命令在串口接收中断服务子程序中发送消息给 Task4，同时 Task4 还具有最高的优先级，这样就能保证 485 通信任务的实时响应和及时执行。

综上所述，程序流程如图 4.3 所示：

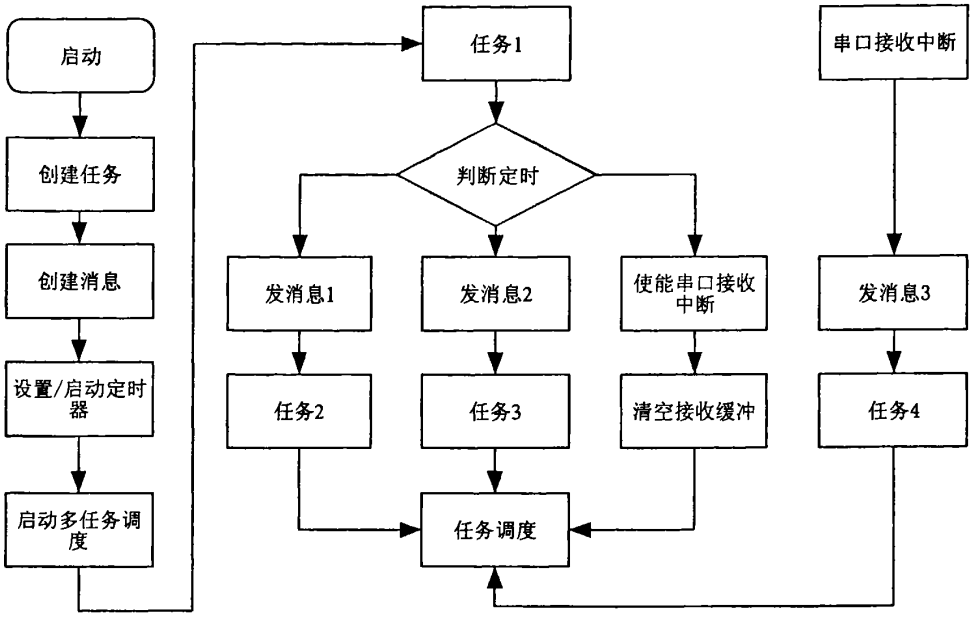


图 4.3 程序流程图

4.3 Bootloader软件应用与测试

如前所述，Bootloader 软件具有巨大的实践应用意义。在嵌入式设备中预置 Bootloader 软件后，只需一台 PC 机即可在应用现场为大量嵌入式设备编程或者升级软件系统。通过这种方式，不仅大大减轻了开发人员的工作量，节省了时间，而且显著降低了开发成本。

第三章介绍的 Bootloader 软件也已经成功应用到了山东省肿瘤防治研究院能量管理项目中。图 4.4 是该项目的逻辑连接图。由图可见，大量的能量检测控制器通过 RS-485 总线相互连接。当开发人员需要对全部或者部分设备升级软件系统时，只需要将 PC 机连接到 RS-485 总线上，然后将待升级设备的 ID 编号输入 PC 机上的 Client 程序中，即可方便地对这些设备编程。

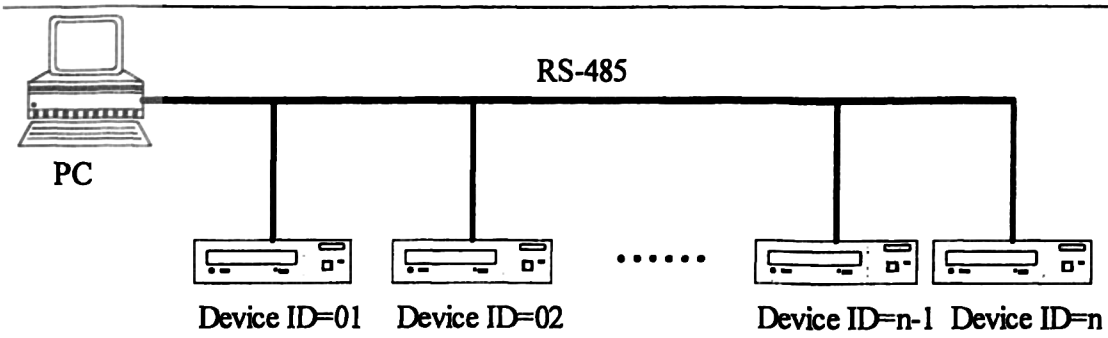


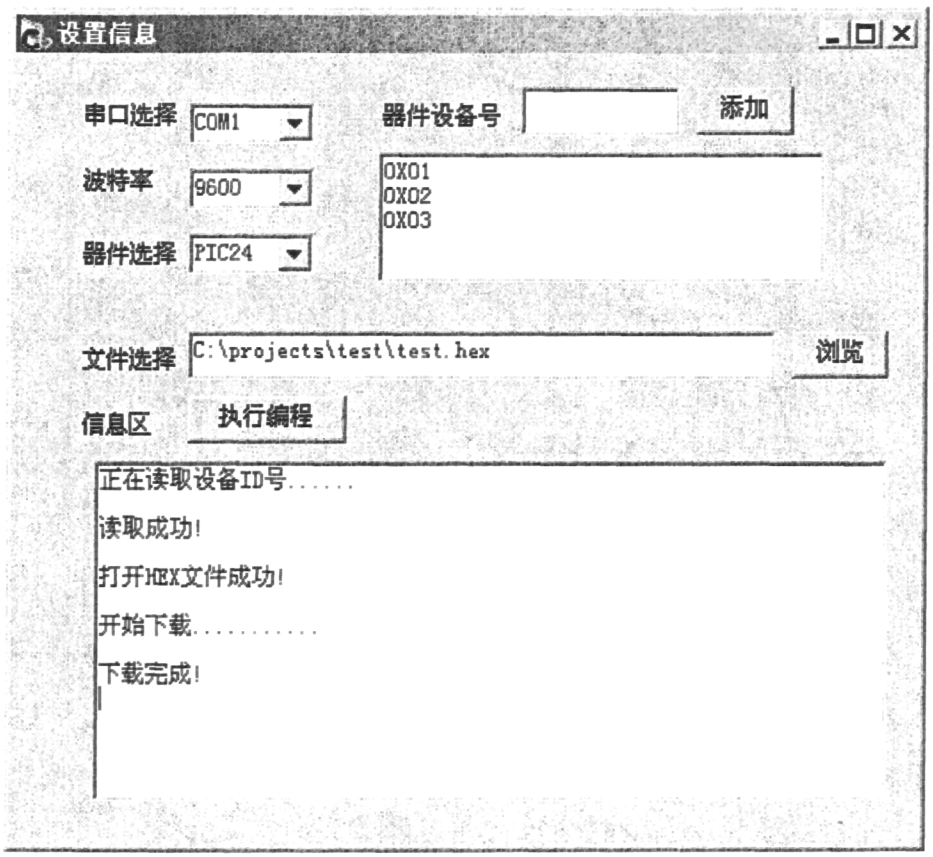
图 4.4 能量管理系统逻辑连接图

具体操作过程如下：

首先，在Client程序中选择待升级设备的ID编号。

然后，选择串口、波特率和待下载的HEX文件。

最后，启动Bootloader过程。



由此可见，经过在实际项目中的测试和检验，Bootloader 程序运行稳定可靠，操作简便。实现了最初的设计目标。

第五章 总结与展望

本文设计和实现了一种基于PIC24系列微控制器的嵌入式实时操作系统，包括操作系统内核和Bootloader程序。经过实际项目的成功应用，可以认为该嵌入式实时操作系统运行稳定可靠，Bootloader软件操作简便实用。

由于本文设计的嵌入式操作系统采用了模块化的设计思路，在具体的应用项目中，可根据实际需要方便地对系统进行裁剪。整个系统分为硬件相关和硬件无关两大部分。其中，硬件无关部分采用标准的C语言编写，该部分可以方便地移植到其他微控制器上，硬件相关部分虽然采用汇编语言编写，但是在本设计中着力减少该部分的代码量，只在必要的情况下才采用汇编语言。因此，根据本文中介绍的软件设计思路，硬件相关部分代码也可以根据具体情况，比较容易地移植到其他处理器平台上。这样的设计方法，充分体现了嵌入式实时操作系统设计理论中可裁剪性和可移植性的要求。未经裁剪的完全版的嵌入式实时操作系统经编译后的大小仅为2KB。可见，该嵌入式操作系统是精简和高效的，完全可以运行在系统资源较紧张的中低端微控制器上，具有较高的实用价值。

本文还设计和实现了与嵌入式实时操作系统相配套的Bootloader软件。虽然本文的设计初衷是将Bootloader软件作为嵌入式是实时操作系统的一个模块。但是，由于整个设计过程也贯穿了模块化设计的思路，因此完全可以根据实际项目的应用需要，将Bootloader软件独立出来，作为一个独立的软件使用。本文中的Bootloader软件是以串行口（RS-232或者RS-485）作为通信连接的，这是由于串行口是目前工控领域内使用最广泛的通信方式，而且PIC24系列微控制器内置了串行通信控制器。同样由于模块化设计思路的采用，根据实际项目的需要，在系统资源和设计成本允许的情况下，只需要更改Bootloader程序中通讯模块的代码就可以很方便地加入对CAN总线甚至TCP/IP等其他通信方式的支持。因此，该Bootloader软件也具有良好的通用性和可扩展性。

任何软件的设计都是一个逐渐完善的过程。作为一个软件产品，本文中设计的嵌入式实时操作系统也有一些地方需要进一步的改进。首先，在软件开发领域中，软件测试的概念已经越来越深入人心。这是因为，通过完善的软件测试过程可以发现软件中隐藏的缺陷，从而保证了软件产品的质量。目前的业界普遍采用软件测试多是采用第三方的测试工具来自动生成测试用例，进而对软件进行全面

完善的测试。但是由于时间仓促以及客观条件的限制，本文中设计的嵌入式实时操作系统和Bootloader软件无法实现如此严格的测试过程，只能在实验室中完成单元测试，以及在实际的项目应用中测试软件的功能和性能。要提高软件的质量，还需要更加严格的测试。第二，随着嵌入式操作系统理论和应用的发展，越来越多的嵌入式系统开始采用操作系统的模式进行开发，但是这些嵌入式设备都是基于不同的嵌入式实时操作系统开发的，而不同嵌入式操作系统又提供不同的系统调用，这就给应用软件的移植带来了很大的困难。面对这种情况，嵌入式操作系统的标准化研究开始受到重视。为了解决嵌入式应用软件移植性的问题，美国IEEE协会提出了POSIX标准。尽管各种嵌入式操作系统的内部实现原理和过程各不相同，但是POSIX标准试图定义一些标准的系统调用接口和功能。本文中设计的嵌入式实时操作系统在设计之初并未考虑标准化和通用性问题，而仅仅以实现功能和性能需求为出发点。因此，从面向未来的发展的角度看，按照POSIX的要求向标准化方向发展也是下一步工作中的要点之一。

尽管还有一些需要改进和完善的地方，但总体来说本文所设计的嵌入式实时操作系统已经实现了最初的设计目标，并在工程实践中获得了成功的应用。

参考文献

【1】 沈胜庆。嵌入式操作系统的内核研究。微计算机信息，2006，05

【2】 谭浩强。C程序设计。北京:清华大学出版社，1999，12

【3】 Jean J.Labrosse著，邵贝贝等译。嵌入式实时操作系统μC/OS-II，第2版。北京：北京航空航天大学出版社，2003，5

【4】 朱珍民，隋雪青，段斌编著。嵌入式实时操作系统及其应用开发。北京：北京邮电大学出版社，2006，12

【5】 郁发新。常用嵌入式实时操作系统比较分析。计算机应用，2006，04

【6】 胡曙辉，陈健。几种嵌入式实时操作系统的分析与比较。单片机与嵌入式系统应用，2007，05

【7】 宋延昭。嵌入式操作系统介绍及选型原则。工业控制计算机，2005，07

【8】 于国华。嵌入式实时操作系统中的程序设计。洛阳工业高等专科学校学报，2003，03

【9】 周航慈，吴光文著。基于嵌入式实时操作系统的程序设计技术。北京：北京航空航天大学出版社，2006，11

【10】 William Stallings著。操作系统－内核与设计原理，第四版。北京：电子工业出版社，2001，6

【11】 蒋句平编著。嵌入式可配置实时操作系统eCos开发与应用。北京：机械工业出版社，2004

【12】唐寅编著。实时操作系统应用开发指南。北京：中国电力出版社，2002，7

【13】 陈明计，周立功等编著。嵌入式实时操作系统Small RTOS51原理及应用。北京：北京航空航天大学出版社，2004

【14】 刘启中，李荣正，王力生，王威编著。PIC单片机原理及应用。北京：北京航空航天大学出版社，2003

【15】 戴梅萼，史嘉权编著。微型计算机技术及应用。北京：清华大学出版社，2000，2

【16】Microchip Technology Incorporated. dsPIC33F & PIC24H Flash Programming Specification. Microchip Technology Incorporated，2007

【17】Microchip Technology Incorporated. PIC24F Family Reference Manual. Microchip Technology Incorporated，2007

【18】Microchip Technology Incorporated. MPLAB C30 C Compiler Getting Started. Microchip Technology Incorporated，2007

【19】Microchip Technology Incorporated. MPLAB C30 C Compiler User's Guide. Microchip Technology Incorporated，2007

【20】Microchip Technology Incorporated. MPLAB IDE v6.xx快速入门指南。Microchip Technology Incorporated，2002

【21】Microchip Technology Incorporated. MPLAB ASM30, MPLAB LINK30 and Utilities User's Guide. Microchip Technology Incorporated，2007

【22】Microchip Technology Incorporated. High-Performance PIC24 Microcontroller Family. Microchip Technology Incorporated，2007

【23】Microchip Technology Incorporated. PIC24FJ32GA002 Family Data Sheet. Microchip Technology Incorporated，2007

【24】Microchip Technology Incorporated. Multi-Tasking on the PIC16F877 with the Salvo™ RTOS. Microchip Technology Incorporated，2001

【25】 龚黎明，辜承林。基于PIC18F系列单片机的嵌入式系统设计。微计算机信息，2004，8

【26】 Jane W.S.Liu著，姬孟洛等译。实时系统。北京：高等教育出版社，2003，12

【27】 潘琢金。基于8位微控制器的嵌入式Internet技术。电子产品世界，2004，2

【28】 万柳，郭玉东。嵌入式RTOS中就绪任务查找算法和优先级反转的解决方案。计算机应用，2003，6

【29】 李飞。几种源码开放的实时操作系统的比较。电子世界，2003,10

【30】 李立清，路海。基于嵌入式系统的TCP/IP协议栈的实现。计算机工程，2004，10

【31】 徐胜，管庆。TMS320C5000 的 Bootloader 技术。IC 与元器件，2003.3

【32】 马学文，朱明日，程小辉。嵌入式系统中 Bootloader 的设计与实现。计算机工程，2005，4

【33】Leonard Elevich, Veena Kvdua. Bootloader for dsPIC30F33F and PIC24F24H Devices. Microchip Technology, Inc

【34】Brant Ivey. A Serial Bootloader for PIC24F Devices. Microchip Technology, Inc

致　谢

在本文即将结束之际，我要衷心地感谢所有关心我，帮助我的老师、同学和朋友们。

本论文是在杜晓通老师的悉心指导下顺利完成的。从课题的方案选择、可行性论证到软硬件平台的设计等，杜老师都给予了精心的指导。杜老师知识渊博、治学严谨、思路开阔，他的许多创新性、开拓性思维使我获益匪浅。在这三年的学习过程中，杜老师不仅教会了我很多知识，更教给了我认识问题、分析问题和解决问题的方法。这是我三年学习过程中最大的收获，也将是我获益终生的宝贵财富。

感谢在学习、工作、生活中给予我帮助的同学和朋友们。感谢实验室的冯立业、杨彬彬、张娜、李忠意等同学。感谢他们在设计和调试过程中的大力帮助，没有他们的帮助，我无法顺利完成本论文。

感谢我们的父母，他们无私的爱和支持是我前进的动力和力量的源泉。

最后，感谢对本论文进行评阅的老师和专家们，感谢你们的辛勤工作。

衷心地感谢你们！

攻读硕士期间研究工作及参与的工程项目

【1】 参与寿光市商务小区智能化系统工程管理项目，2006.9-2007.7

【2】 参与胜利油田油井监控系统的研究与开发，2007.9-2008.3

【3】 参与嵌入式实时操作系统的研究，2006.9-至今