南开大学

硕士学位论文

嵌入式操作系统Particula OS系统设计与实现技术研究

姓名：朱鹏辉

申请学位级别：硕士

专业：计算机软件与理论

指导教师：刘景泰

20070501

摘要

嵌入式系统在社会生产和生活中的应用日益广泛，人们对嵌入式系统的功能要求及系统的软硬件复杂程度也日益提高。直接基于硬件的开发方式严重阻碍了生产效率的提高，嵌入式操作系统通过管理和封装硬件，为系统开发者提供了一个快速开发的平台，使得生产效率得到明显提高，维护和开发的成本有效降低。在嵌入式系统中使用操作系统正在成为一种趋势。

嵌入式操作系统 Particula OS 由内核和核外子系统组成。内核支持分时轮转和优先级混合调度，负责中断、上下文切换等底层支持功能。核外子系统包括进程管理、时间管理以及进程间通信。主要向用户提供进程创建、挂起及恢复，延时挂起与恢复，信号量通信和消息队列通信等系统服务。嵌入式操作系统 Particula OS 是南开大学机器人与信息自动化研究所针对嵌入式系统的实际，自行开发的已经获得软件登机证书的嵌入式操作系统，具有可配置、可剪裁、易移植等特点。

任务切换和堆栈技术是 Particula OS 设计和实现中的两个关键技术。任务切换技术是多任务系统的基础，论文结合 PIC18F8621/8622 的实际，详细定义了上下文的内容，剖析上下文保存和上下文恢复的机器执行过程，从而阐明了上下文切换的过程。堆栈对于嵌入式系统紧张的数据内存的影响非常直接，论文在给出了堆栈设计的一般方法的基础上，利用 P89C668 的硬件特点，采用将中断堆栈和任务堆栈相分离的方法，从而优化了堆栈设计。

中断响应时间和任务切换时间是 Particula OS 中具有代表性的两个时间指标。一个标志着对外部事件的反应速度，一个指明了对 CPU 的利用效率。经过在 PIC18F452 平台上的一系列实验，得到了上述两个时间指标分别为 61μs 和 139μs，可以满足应用系统对其的时间要求。

关键词：Particula OS，内核，核外子系统，任务切换，堆栈

Abstract

Embedded system is changing the life and the embedded operating system is changing the embedded system. Today, the functions of the embedded system are more complex. Without the embedded operating system, building a system is hard.

The embedded operating system Particula OS is developed by Institute of Robotics and Automatic Information System, Nankai University, China. It is scalable, portable and configurable.

Particula OS consists of the kernel and subsystems out of the kernel. The kernel is the basis of Particula OS. Subsystems out of the kernel include Process Management, Time Management and Internal Process Communication.

Something difficult in the design and implementation of Particula OS includes context switching and stack. Using the specific hardware, PIC MCU PIC18F8621/8622, the context was caught clearly. Relevantly, the context switching was interpreted easily. The way of using the stack space is critical in the embedded system. An optimal approach using stack space is proposed in the thesis.

Interrupt response time and context switching time are important to Particula OS. In the platform of PIC18F452, the experiments were designed and implemented to get the interrupt response time and context switching time.

Particula OS has the basic components of the embedded operating system. It will upgrade itself in the future.

Key Words: Particula OS, Kernel, Subsystems out of the kernel, Task Switching, Stack

南开大学学位论文版权使用授权书

本人完全了解南开大学关于收集、保存、使用学位论文的规定，同意如下各项内容：按照学校要求提交学位论文的印刷本和电子版本；学校有权保存学位论文的印刷本和电子版，并采用影印、缩印、扫描、数字化或其它手段保存论文；学校有权提供目录检索以及提供本学位论文全文或者部分的阅览服务；学校有权按有关规定向国家有关部门或者机构送交论文的复印件和电子版；在不以赢利为目的的前提下，学校可以适当复制论文的部分或全部内容用于学术活动。

学位论文作者签名：李鹏辉

2007 年 5 月 28 日

经指导教师同意，本学位论文属于保密，在　　年解密后适用本授权书。

指导教师签名：		学位论文作者签名：	李鹏辉
解　密　时　间：	年　　月　　日		

各密级的最长保密年限及书写格式规定如下：

内部　5 年（最长 5 年，可少于 5 年）

秘密★10 年（最长 10 年，可少于 10 年）

机密★20 年（最长 20 年，可少于 20 年）

南开大学学位论文原创性声明

本人郑重声明：所呈交的学位论文，是本人在导师指导下，进行研究工作所取得的成果。除文中已经注明引用的内容外，本学位论文的研究成果不包含任何他人创作的、已公开发表或者没有公开发表的作品的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明。本学位论文原创性声明的法律责任由本人承担。

学位论文作者签名：李晓婷

2007 年 5 月 28 日

第一章 绪论

嵌入式系统在社会生产和生活中的广泛应用，使得嵌入式操作系统的作用越来越重要，在嵌入式系统中使用嵌入式操作系统正在成为一种潮流。

第一节 嵌入式系统

1.1.1 嵌入式系统

嵌入式系统（Embedded System）是当今的新技术、最热门的领域之一，至今为止仍没一个可以普遍接受的定义。但从一般意义上讲，可以这样给它定义：

嵌入式系统是以应用为中心，以计算机技术为基础，并且软硬件可裁剪，适用于应用系统对功能、可靠性、成本、体积、功耗有严格要求的专用计算机系统。

嵌入式系统由嵌入式硬件和嵌入式软件组成。

从20世纪70年代开始出现的单片机到现在的DSP、SOC，嵌入式硬件经历了长足发展；嵌入式软件业也从最初的一体化设计，到现在嵌入式操作系统和应用软件的分离。现在嵌入式系统已经广泛应用于社会产生和生活的各个方面。

1.1.2 嵌入式硬件

一般而言，嵌入式硬件可以分成：处理器、存储器、输入输出（I/O）。其中最核心，最能代表嵌入式系统性能的是嵌入式处理器。

根据其现状，嵌入式处理器可以分成下面几类：

1. 嵌入式微处理器（Micro Processor Unit，MPU）

嵌入式微处理器是由通用计算机中的CPU演变而来的。它的特征是具有32位以上的处理器，具有较高的性能，当然其价格也相应较高。但与计算机处理器不同的是，在实际嵌入式应用中，只保留和嵌入式应用紧密相关的功能硬件，去除其他的冗余功能部分，这样就以最低的功耗和资源实现嵌入式应用的特殊要求。和工业控制计算机相比，嵌入式微处理器具有体积小、重量轻、成本低、可靠性高的优点。目前主要的嵌入式处理器类型有Am186/88、386EX、SC-400、Power PC、68000、MIPS、ARM/StrongARM系列等。

2. 嵌入式微控制器（Microcontroller Unit，MCU）

嵌入式微控制器的典型代表是单片机，从 70 年代末单片机出现到今天，虽然已经经过了 20 多年的历史，但这种 8 位的电子器件目前在嵌入式设备中仍然有着极其广泛的应用。单片机芯片内部集成 ROM/EPROM、RAM、总线、总线逻辑、定时/计数器、看门狗、I/O、串行口、脉宽调制输出、A/D、D/A、Flash RAM、EEPROM 等各种必要功能和外设。和嵌入式微处理器相比，微控制器的最大特点是单片化，体积大大减小，从而使功耗和成本下降、可靠性提高。微控制器是目前嵌入式系统工业的主流。微控制器的片上外设资源一般比较丰富，适合于控制，因此称作微控制器。比较有代表性的包括 8051、MCS-251、MCS-96/196/296、P51XA、C166/167、68K 系列以及 MCU 8XC930/931、C540、C541，并且有支持 I2C、CAN-BUS、LCD 及众多专用 MCU 和兼容系列。目前 MCU 占嵌入式系统约 70%的市场份额。近来 ATMEL 出产的 AVR 单片机由于其集成了 FPGA 等器件，所以具有很高的性价比，势必将推动单片机获得更高的发展。

3. 嵌入式 DSP 处理器（Embedded Digital Signal Processor，EDSP）

DSP 处理器是专门用于信号处理方面的处理器，其在系统结构和指令算法方面进行了特殊设计，具有很高的编译效率和指令的执行速度。在数字滤波、FFT、谱分析等各种仪器上 DSP 获得了大规模的应用。

DSP 的理论算法在 70 年代就已经出现，但是由于专门的 DSP 处理器还未出现，所以这种理论算法只能通过 MPU 等由分立元件实现。MPU 较低的处理速度无法满足 DSP 的算法要求，其应用领域仅仅局限于一些尖端的高科技领域。随着大规模集成电路技术发展，1982 年世界上诞生了首枚 DSP 芯片。其运算速度比 MPU 快了几十倍，在语音合成和编码解码器中得到了广泛应用。至 80 年代中期，随着 CMOS 技术的进步与发展，第二代基于 CMOS 工艺的 DSP 芯片应运而生，其存储容量和运算速度都得到成倍提高，成为语音处理、图像硬件处理技术的基础。到 80 年代后期，DSP 的运算速度进一步提高，应用领域也从上述范围扩大到了通信和计算机方面。90 年代后，DSP 发展到了第五代产品，集成度更高，使用范围也更加广阔。

目前最为广泛应用的是 TI 的 TMS320C2000/C5000 系列，另外如 Intel 的 MCS-296 和 Siemens 的 TriCore 也有各自的应用范围。

4. 嵌入式片上系统（System On Chip）

SOC 追求产品系统最大包容的集成器件，是目前嵌入式应用领域的热门话题之一。SOC 最大的特点是成功实现了软硬件无缝结合，直接在处理器片内嵌入操作系统的代码模块。而且 SOC 具有极高的综合性，在一个硅片内部运用 VHDL 等硬件描述语言，实现一个复杂的系统。用户不需要再像传统的系统设计一样，绘制庞大复杂的电路板，一点点的连接焊制，只需要使用精确的语言，综合时序设计直接在器件库中调用各种通用处理器的标准，然后通过仿真之后就可以直接交付芯片厂商进行生产。由于绝大部分系统构件都是在系统内部，整个系统就特别简洁，不仅减小了系统的体积和功耗，而且提高了系统的可靠性，提高了设计生产效率。

1.1.3 嵌入式软件

嵌入式软件往往可以分为底层系统软件和应用软件。

底层系统软件包括嵌入式操作系统，驱动程序，BSP 等。而嵌入式应用软件会结合具体的应用，具有较强的针对性和专用性。嵌入式系统技术具有非常广阔的应用前景，其应用领域可以包括：

1. 工业控制

基于嵌入式芯片的工业自动化设备将获得长足的发展，目前已经有大量的 8、16、32 位嵌入式微控制器在应用中，网络化是提高生产效率和产品质量、减少人力资源的主要途径，如工业过程控制、数字机床、电力系统、电网安全、电网设备监测、石油化工系统。就传统的工业控制产品而言，低端型采用的往往是 8 位单片机。但是随着技术的发展，32 位、64 位的处理器逐渐成为工业控制设备的核心，在未来几年内必将获得长足的发展。

2. 交通管理

在车辆导航、流量控制、信息监测与汽车服务方面，嵌入式系统技术已经获得了广泛的应用，内嵌 GPS 模块，GSM 模块的移动定位终端已经在各种运输行业获得了成功的使用。目前 GPS 设备已经从尖端产品进入了普通百姓的家庭，只需要几千元，就可以随时随地找到你的位置。

3. 信息家电

这将称为嵌入式系统最大的应用领域，冰箱、空调等的网络化、智能化将引领人们的生活步入一个崭新的空间。即使你不在家里，也可以通过电话线、网络进行远程控制。在这些设备中，嵌入式系统将大有用武之地。

4. 家庭智能管理系统

水、电、煤气表的远程自动抄表，安全防火、防盗系统，其中嵌有的专用控制芯片将代替传统的人工检查，并实现更高，更准确和更安全的性能。目前在服务领域，如远程点菜器等已经体现了嵌入式系统的优势。

5. POS 网络及电子商务

公共交通无接触智能卡发行系统，公共电话卡发行系统，自动售货机，各种智能 ATM 终端将全面走入人们的生活，到时手持一卡就可以行遍天下。

6. 环境工程与自然

水文资料实时监测，防洪体系及水土质量监测、堤坝安全，地震监测网，实时气象信息网，水源和空气污染监测。在很多环境恶劣，地况复杂的地区，嵌入式系统将实现无人监测。

7. 机器人

嵌入式芯片的发展将使机器人在微型化，高智能方面优势更加明显，同时会大幅度降低机器人的价格，使其在工业领域和服务领域获得更广泛的应用。

第二节 嵌入式操作系统

嵌入式操作系统是系统的灵魂，其同时肩负着管理系统硬件资源的功能和为用户提供系统服务的功能。

1.2.1 嵌入式操作系统现状

从 20 世纪 80 年代早期开始，嵌入式系统的程序员开始用商业级的“操作系统”编写嵌入式应用软件，这使得可以获取更短的开发周期，更低的开发资金和更高的开发效率，“嵌入式操作系统”出现了。确切点说，这个时候的操作系统是一个实时核，这个实时核包含了许多传统操作系统的特征，包括任务管理、任务间通讯、同步与相互排斥、中断支持、内存管理等功能。其中比较著名的有 Ready System 公司的 VRTX、Integrated System Incorporation（ISI）的 PSOS 和 IMG 的 VxWorks、QNX 公司的 QNX[1]等。这些嵌入式操作系统都具有嵌入式的典型特点：它们均采用占先式的调度，响应的时间很短，任务执行的时间可以确定；系统内核很小，具有可裁剪，可扩充和可移植性，可以移植到各种处理器上；较强的实时和可靠性，适合嵌入式应用。这些嵌入式实时多任务操作系统的出现，使得应用开发人员得以从小范围的开发中解放出来，同时也促

使嵌入式有了更为广阔的应用空间。

90 年代以后，随着对实时性要求的提高，软件规模不断上升，实时核逐渐发展为实时多任务操作系统（RTOS），并作为一种软件平台逐步成为目前国际嵌入式系统的主流。这时候更多的公司看到了嵌入式系统的广阔发展前景，开始大力发展自己的嵌入式操作系统。除了上面的几家老牌公司以外，还出现了 Palm OS，WinCE，嵌入式 Linux，Lynx，Nucleux，以及国内的 Hopen，Delta OS[2] 等嵌入式操作系统。随着嵌入式技术的发展前景日益广阔，相信会有更多的嵌入式操作系统软件出现。

1.2.2 微内核技术

随着计算机硬件体系结构的不断变化和用户所需功能的不断扩展，操作系统必须解决可扩充性、可移植性、兼容性、效率、可靠性、适应性等问题。传统操作系统是大内核结构的操作系统，具体功能全部放在内核中完成，如文件服务、网络服务等。这种实现方式使得操作系统内核的抽象概念变得混乱、模块性差、不易移植、不易扩充。

所谓微内核[3, 4]技术，就是把操作系统最基本的功能放在一个小核心中实现，其它功能尽量放在核外实现。内核向用户提供基本机制和抽象对象。使用微内核后，就可以在微内核的基础上利用微内核提供的基本功能，在核外实现操作系统的各种特征，完成原内核的所有功能，使原操作系统的应用程序不作任何修改就可以运行在新的操作系统上。

基于微内核的操作系统和传统操作系统相比，传统操作系统的各种服务以过程调用形式相互作用，效率高。微内核是基于消息传递机制的[5]，因此在效率上虽略有降低，但基于微内核的操作系统可扩充性强，增加服务时，只需在核心外增加功能服务；可移植性强，核外各功能基本上与硬件无关；可靠性好，核外各功能和微内核处于不同层，不相互干扰，而且可支持不同的硬件体系结构与应用需求。因此总的来说，微内核结构以其方便灵活、可移植性强和可靠性高的特点，成为下一代操作系统的发展方向。

嵌入式操作系统 Particula OS 正是充分考虑到了嵌入式系统空间有限、对可移植性要求高等特点，而采用微内核结构，极大的满足嵌入式系统开发的实际。

第三节 基本概念

在嵌入式操作系统中，很多概念在不同的论著中有着或多或少的差别，为了保证论文概念的规范，将嵌入式操作系统 Particula OS 设计中涉及到的重要概念作一概要说明。

1.3.1 任务和多任务

1. 任务（Task）

嵌入式应用程序的设计中常常要把待解决的问题分割成多个功能模块，每个模块完成一定的功能。在 Particula OS 中，能够完成一定功能的程序模块是以进程的形态存在和运行的。每个进程负责一定的功能，于是就将能够承担一定功能的、以进程形态存在的程序称为一个任务。

因此，在 Particula OS 中，进程就可以称为任务。

2. 多任务（Multi-Task）

在计算机产生和发展的初期，整个机器将全部的运算能力和系统资源贡献给了正在运行的单一程序。后来人们研究发现以这种方式运行的计算机会因为程序响应外部事件、存取 I/O 等原因而让 CPU 处于空闲等待状态，白白浪费了宝贵的计算资源。

为解决这一问题，提高计算机的利用效率，“多道程序设计”的概念被提了出来。其核心概念是指，在系统中同时运行多道程序，当一个正在执行的程序因某些事件暂时停止运行的时候，让另一个程序投入运行。这样就提高了系统资源利用率。

在 Particula OS 中就使用多任务机制实现了“多道程序设计”的思想，如此在进行嵌入式系统应用开发中，就可以使开发人员将很复杂的应用程序层次化，应用程序将更容易设计和维护。

1.3.2 任务切换

嵌入式操作系统 Particula OS 是一个多任务操作系统，具备不同任务的各个进程的运动过程总是处于运动、停止，运动、停止……，运动、停止的不断变化之中。当一个任务 A 停止运行时，就要将另一个任务 B 投入运行。经过一段时间的运行之后，任务 A 又要重新投入运行时，就要在 CPU 及系统中各必需寄存器导入任务 A 停止运行时的数值，这一过程称为上下文恢复。

与之相对应，当一个任务停止运行时，为了日后的上下文恢复，首先要保存CPU及系统中各必需的寄存器，这个过程称为上下文保存。

在多任务系统中，由于存在着不同任务在运动和停止之间的转换，也就存在着不断的上下文保存和上下文恢复。

于是将一个任务停止时，保存自己的上下文，然后将另一个要投入运行的任务进行上下文恢复的过程，称为任务切换（Task Switch），也称为上下文切换（Context Switch）。

1.3.3 可剥夺内核

内核的可剥夺性（Preemptive）是指，内核可以终止一个正在运行、且尚未运行完成的任务，转而让另一个任务投入运行。

当某个异步事件发生时，激活了一个高优先级任务，这时可剥夺内核就可以立即将低优先级的任务终止，使高优先级任务投入运行，保证了系统具有较高的响应时间。

1.3.4 中断

中断[6]的出现使得处理器可以灵活处理异步事件。当某个事件发生时，会触发中断电路，这一过程被称为中断请求；CPU 注意到这一情况后，就中断正在执行的任务，开始着手来处理这个事件，这一过程被称为中断响应；CPU 执行用户的中断服务子程序的过程，称为中断服务；中断服务程序的最后一条指令是中断返回指令，当执行到该指令时，便返回程序，从断点处继续往下执行，这一过程被称为中断返回。

由上，一个完整的中断处理过程可以总结为四个步骤：中断请求，中断响应，中断服务，中断返回。

在嵌入式系统中，中断响应的快慢是考量系统反应能力的一个重要指标。

第四节 论文内容组织

论文的研究内容是嵌入式操作系统 Particula OS 的设计，以及在实现中若干重要的技术问题。

第一章绪论。介绍嵌入式系统和嵌入式操作系统，叙述了嵌入式操作系统的研究现状，以及若干重要的基本概念，并概述了论文的主要研究内容和论文的

结构安排。

第二章是 Particula OS 系统设计。嵌入式操作系统 Particula OS 由内核和核外子系统组成，可以完成中断，调度，进程管理，时间管理以及进程间通信等功能。

第三章是 Particula OS 关键实现技术研究。论文对 Particula OS 设计和实现中的两个关键技术进行研究。任务切换技术是多任务系统的基础，论文结合 PIC18F8621/8622 的实际，详细定义了上下文的内容，阐明了上下文切换的过程。堆栈对于嵌入式系统紧张的数据内存的影响非常直接，论文根据 P89C668 的实际特点，给出了一种堆栈的优化方法。

第四章是 Particula OS 时间性能测定与分析。中断响应时间和任务切换时间是 Particula OS 中具有代表性的两个时间指标，一个标志着对外部事件的反应速度，一个指明了对 CPU 的利用效率。经过在 PIC18F452 平台上的一系列实验，得到了上述两个时间指标的具体数值，并进行了分析。

第五章是总结与展望。

第二章 Particula OS 系统设计

嵌入式操作系统 Particula OS 的设计从总体上可以分为内核和核外子系统。本章在给出 Particula OS 的总体设计的基础上，按照各模块逐一给出其详细设计。

第一节 总体设计

2.1.1 背景与目标

随着社会的进步，自动化和信息化程度的提高，嵌入式系统得到了广泛应用。作为支撑软件的嵌入式操作系统，一方面要提供底层支持服务，一方面还要满足嵌入式系统对于时间和空间方面的要求。并且随着消费电子的发展，它们对于实时性的要求并不苛刻，但却提出了对于处理器和各种资源的“虚拟”和“共享”的要求。

正是基于这样的考虑，嵌入式操作系统 Particula OS 的设计目标就是内存占用少，具有软实时性，可分时，可移植，可剪裁及良好的结构性。

嵌入式操作系统 Particula OS 从结构上看，包括内核和核外子系统。Particula OS 的体系结构如图 2.1 所示。

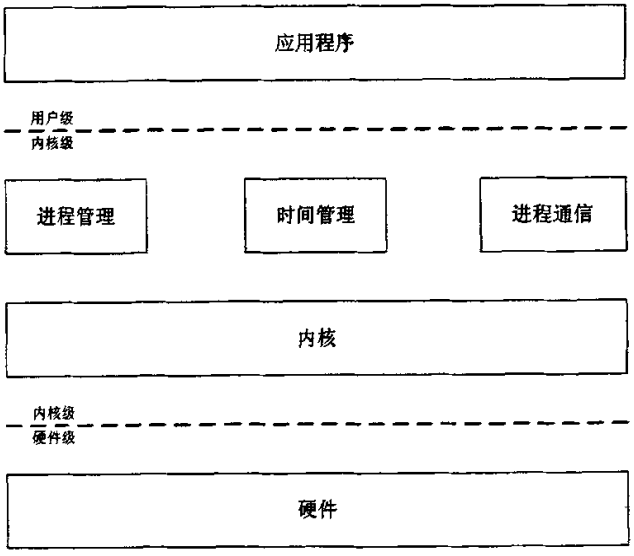


图 2.1 Particula OS 的体系结构

嵌入式操作系统 Particula OS 的内核功能包括调度，中断，上下文切换，进程间通信的底层算法。可选的功能包括片上其它部件的驱动程序。

在内核的基础上，核外各子系统包括进程管理，时间管理，信号量管理，消息队列管理。它们可以使用内核提供的底层功能。

2.1.2 内核设计

1. 进程结构及其数据结构

进程组成：代码，进程控制块（PCB），用户栈

PCB：进程的管理信息，进程状态模型信息，堆栈信息

用户栈：采用连续数组的方式

2. 进程状态模型

状态模型：就绪，运行，睡眠。

进程控制块表：每个需要进程控制块的进程将从该表中申请到一个空间。

就绪表：是一个数组，每个数组元素代表一个优先级。每个数组元素是一个队列。队列中的元素是进程号。

运行队列：设置全局变量

3. 调度算法

每个进程赋以一定的优先级。优先级不同时，按优先级调度；优先级相同时，按分时调度。

最大支持 8 个用户优先级。每个优先级最多支持 8 个进程分时运行。

采用可抢占式调度策略。

4. 上下文切换、中断和栈

上下文切换：PCB 本身和 CPU 各寄存器的内容。

中断：中断处理程序要简短，可将处理内容向用户程序转移。请求不成功时以错误退出，不再等待。

支持中断嵌套。中断时不允许进行上下文切换。中断栈和用户栈分开。

栈：向上增长和向下增长用宏定义来区分。

2.1.3 核外各层设计

1. 进程管理

提供进程的创建，进程挂起，进程恢复，进程数据查询的系统调用。

2．时间管理

提供任务延时，恢复延时的任务，获取系统时间，设置系统时间的系统调用。

3．进程间通信

进程间通信提供信号量，消息队列通信这两种机制。

2.1.4 可移植性和可剪裁性考虑

1．可移植性

尽量减少汇编成份，尽量减少在 C 代码中加入汇编代码。将汇编代码单列封装，提供宏调用接口。

2．可剪裁性

除了内核，其余部分，都设置相应的配置选项，可根据需要适当配置功能的增减。

第二节 内核设计

2.2.1 进程结构

进程组成：代码，进程控制块（PCB），用户栈。

PCB：进程的静态管理信息，进程状态模型信息，堆栈信息。PCB 内容如表 2.1 所示。

用户栈：采用连续数组的方式。

表 2.1 PCB 数据结构

PCB 数据结构		
信息属类	域	含义
堆栈信息	SOSPCBStkPtr	指向当前进程堆栈栈顶的指针
	SOSPCBStkBottom	指向当前进程堆栈栈底的指针
进程的管理信息	SOSPID	进程标识号
	SOSPCBNext	后向指针
	SOSPCBPrev	前向指针
	SOSPCBDly	进程延时时间
	SOSPCBPrio	进程优先级
进程状态模型的动态信息	SOSPCBState	进程状态

2.2.2 进程状态模型

1．进程状态

Particula OS 的进程状态如表 2.2 所示。

表 2.2 进程状态定义

进程状态定义		
进程状态定义量	汉语含义	英语含义
SOS_STATE_RDY	就绪态	READY
SOS_STATE_RUN	运行态	RUN
SOS_STATE_SLP	睡眠态	SLEEP

- 就绪态

进程一旦建立，就进入了就绪态（READY），准备运行。进程的建立可以是在多任务运行开始之前，也可以动态地由一个运行着的进程建立。

- 运行态

调用 SOSStart()可以启动多任务。SOSStart()函数只能在启动时调用一次，该函数运行用户初始化代码中已经建立的、进入就绪态的优先级最高的进程。优先级最高的进程就这样进入了运行态（RUN）。任何时刻只能有一个进程处于运行态。就绪的进程只有当所有优先级高于这个进程的进程都转为睡眠状态（SLEEP），才能进入运行态。

- 睡眠态

正在运行的进程可以通过调用 SOSTmeDly()，将自身延迟一段时间。这个进程于是进入睡眠状态（SLEEP），一直到函数中定义的延迟时间到。该函数会立即强制执行任务切换，让下一个优先级最高的、并进入了就绪态的进程运行。等待的时间过去以后，系统服务函数 SOSTimeTick()使延迟了的进程进入就绪态。

正在运行的进程可能需要等待某一事件的发生，可以通过调用函数 SOSSemPend()和 SOSQPend()。如果某事件并未发生，调用上述函数的进程就进入了睡眠状态（SLEEP），直到等待的事件发生了。当进程因等待事件被挂起时，下一个优先级最高的进程立即得到了 CPU 的使用权。当事件发生了或等待超时，被挂起的进程就进入就绪态。事件发生的报告可能来自另一个进程，也可能来自中断服务子程序。

2. 就绪表

就绪表是数组，每个数组元素代表一个优先级。每个数组元素是一个队列。队列中的元素是进程号（PID）。

3. 数据结构定名

- 就绪表：SOSRdyTbl[]
- 就绪表大小：SOS_RDY_TBL_SIZE

4. 运行队列

使用全局变量 SOSPCBCur 来指向当前运行的进程的进程控制块，不用设置运行队列，因为运行的进程只可能有一个。

2.2.3 内核函数列表

表 2.3 内核函数列表

内核函数详细定义	
函数原型	函数功能
void SOSInit (void)	系统初始化
void SOSIntEnter (void)	中断进入
void SOSIntExit (void)	中断退出
void SOSSched (void)	进程级调度
void SOSSchedLock (void)	调度上锁
void SOSSchedUnLock (void)	调度解锁
void SOSStart (void)	系统多任务启动
void SOSProcIdle (void *ppdata)	空闲进程
void SOSStatInit (void)	统计进程初始化（设置配置开关）
void SOSProcStat (void *ppdata)	统计进程（设置配置开关）
INT8U SOSPCBInit (INT8U prio, OS_STK *ptos, OS_STK *pbos，INT8U *pid)	进程控制块初始化
void *SOSProcStkInit (void (*task)(void *pd), void *ppdata, void *ptos, INT16U opt)	任务堆栈初始化（与 CPU 有关）
void SOSTimeTick (void)	时钟节拍函数，在时钟中断程序中调用
INT16U SOSVersion (void)	提供系统版本信息

2.2.4 系统初始化

函数 SOSInit ()为系统进行初始化。

1. 函数原型

void SOSInit (void);

2. 函数功能

本函数用来初始化 Particula OS 的内核，并且必须在创建任何数据和执行任何操作之前被执行，并且必须在 SOSStart()被调用前执行。

3. 参数和返回值说明

参数：无

返回值：无

4. 程序流程图见附录 A 中图 A1。

2.2.5 中断进入

函数 SOSIntEnter ()是中断进入。

1. 函数原型

void SOSIntEnter (void);

2. 函数功能

这个函数用来向 Particula OS 表明程序正在准备进入中断服务程序。它允许 Particula OS 跟踪中断嵌套，并使得 Particula OS 只能在最后一个中断即将退出时才能发生调度。

3. 参数和返回值说明

参数：无

返回值：无

4. 程序流程图见附录 A 中图 A2。

5. 说明

(1) 中断服务程序可以直接对 SOSIntNesting 进行加一操作，必须确保对 SOSIntNesting 的正确操作，因为 SOSIntNesting 是一个全局变量。

(2) 虽然可以直接操作 SOSIntNesting，但是中断离开还必须调用 SOSIntExit()。

(3) SOSIntEnter() 和 SOSIntExit()的调用必须成对出现，每调用一次 SOSIntEnter()，就必须调用一次 SOSIntExit()。

2.2.6 中断退出

函数 SOSIntExit ()是中断退出。

1. 函数原型

void SOSIntExit (void);

2. 函数功能

这个函数用来向 Particula OS 表明程序正在准备进入中断服务程序。当最后一个嵌套的中断服务程序完成时，Particula OS 将调用调度器来决定是否存在一个新的，高优先级的进程投入运行。

3. 参数和返回值说明

参数：无

返回值：无

4. 程序流程图见附录 A 中图 A3。

5. 说明

(1) SOSIntEnter() 和 SOSIntExit() 的调用必须成对出现，每调用一次 SOSIntEnter()，就必须调用一次 SOSIntExit()。

(2) 当调度器被上锁期间，调度是禁止的。

2.2.7 进程调度

函数 SOSSched()是进程调度。

1. 函数原型

void SOSSched (void);

2. 函数功能

该函数被其它 Particula OS 系统服务调用，用来决定是否存在一个新的，高优先级进程投入运行。

3. 参数和返回值说明

参数：无

返回值：无

4. 程序流程图见附录 A 中图 A4。

5. 说明

(1) 该函数只能在 Particula OS 内部进行调用，应用程序不得使用。

(2) 当调度器被上锁期间，调度禁止。

2.2.8 调度上锁

函数 SOSSchedLock()是进程调度上锁。

1. 函数原型

void SOSSchedLock (void);

2. 函数功能

本函数的功能是阻止调度的发生。如果你想阻止上下文切换，你可以在你的应用程序中使用该函数。

3. 参数和返回值

参数：无

返回值：无

4. 程序流程图见附录 A 中图 A5。

5. 说明

对 SOSSchedLock()和 SOSSchedUnlock()的调用，必须成对儿使用。换句话说，每调用一次 SOSSchedLock()，就必须也调用一次 SOSSchedUnlock()。

2.2.9 调度解锁

函数 SOSSchedUnLock ()是进程调度解锁。

1. 函数原型

void SOSSchedUnLock(void);

2. 函数功能

本函数的功能是为进程调度解锁，使能进程调度。

3. 参数和返回值

参数：无

返回值：无

4. 程序流程图见附录 A 中图 A6。

5. 说明

对 SOSSchedLock()和 SOSSchedUnlock()的调用，必须成对儿使用。换句话说，每调用一次 SOSSchedLock()，就必须也调用一次 SOSSchedUnlock()。

2.2.10 系统多任务启动

函数 SOSStart ()是系统多任务启动。

1. 函数原型

void SOSStart (void);

2. 函数功能

本函数用来启动多任务，对于用户所创建的进程，使用本函数后，进程将进入被管理状态。在使用 SOSStart（）之前，必须使用 SOSInit（），并且需要至少创建一个进程。

3. 参数和返回值

参数：无

返回值：无

4. 程序流程图见附录 A 中图 A7。

5. 说明

SOSStartHighRdy()必须调用 SOSProcSwHook()，并且将 SOSRunning 置为真。

2.2.11 空闲进程

函数 SOSProcIdle ()是空闲进程。

1. 函数原型

void SOSProcIdle (void *ppdata);

2. 函数功能

该进程是 Particula OS 的内部进程，它具有最低的优先级，当其它进程因为等待某些事件发生而挂起时，该进程才会运行。

3. 参数和返回值

参数：1 个。

void *ppdata：传递给空闲进程的数据

返回值：无

4. 程序流程图见附录 A 中图 A8。

2.2.12 统计进程初始化

函数 SOSStatInit ()是统计进程初始化

1. 函数原型

void SOSStatInit (void);

2. 函数功能

该函数用来统计 CPU 的利用率，当它执行时，没有高优先级进程在执行，所以这时它可以统计在 1 秒钟时间内 CPU 可以计数到多少。之后 CPU 的利用率就可以由一个更低优先级的进程来决定，该进程可以跟踪内部计数器当其它进程执行时，如此，CPU 的利用率就可以使用如下公式进行计算。

$$CPU\text{利用率}\ (\%) = 100 * (1 - \frac{\text{OSIdleCtr}}{\text{OSIdleCtrMax}})$$

3. 参数和返回值

参数：无

返回值：无

4. 程序流程图见附录 A 中图 A9。

2.2.13 统计进程

函数 SOSProcStat ()是统计进程。

1．函数原型

void SOSProcStat (void *ppdata);

2．函数功能

这个函数是 Particula OS 的内部函数，可以在多任务环境下进行统计工作。统计进程可以计算 CPU 的使用率，通过如下的公式进行。

$$CPU\text{利用率}\ (\%) = 100 * (1 - \frac{\text{OSIdleCtr}}{\text{OSIdleCtrMax}})$$

3．参数和返回值

参数：1 个

void *ppdata：传递给统计进程的数据

返回值：无

4．程序流程图见附录 A 中图 A10。

5．说明

(1) 该进程的优先级是进高于空闲进程的次低进程。它的优先级是 SOS_IDLE_PRIO-1

(2) 可以取消该进程，通过设置配置位#define SOS_PROC_STAT_EN 为 0

(3) 在可以开始稳定的统计进程前，要留出 5 秒钟的时间，让系统可以稳定运行，同时也让其它进程可以很好的执行创建过程。最低的时间是 2 秒。

2.2.14 进程控制块初始化

函数 SOSPCBInit ()是进程控制块初始化。

1．函数原型

INT8U SOSPCBInit (INT8U prio, OS_STK *ptos, OS_STK *pbos，INT8U *pid);

2．函数功能

这个函数是 Particula OS 的内部函数，用来对进程控制块进行初始化。

3．参数和返回值

参数：3 个。

- prio：被创建进程的优先级
- ptos：进程栈的栈顶指针。该栈用来存放 CPU 的各寄存器。当

SOS_STK_GROWTH 为 1 时，栈顶指针指向内存高地址；当 SOS_STK_GROWTH 为 0 时，栈顶指针指向内存低地址。栈的增长方向是 CPU 紧密相关的。

- pbos：栈底指针。

返回值：3 种。

- SOS_NO_ERR：调用成功时。
- SOS_NO_MORE_PCB：没有进程控制块可以分配。
- SOS_QTS_FULL：QTS 空间满。

4. 程序流程图见附录 A 中图 A11。

5. 说明

该函数是 Particula OS 的内部函数，应用程序不能使用。

2.2.15 任务堆栈初始化

函数 SOSProcStkInit ()是任务堆栈初始化。

1. 函数原型

void *SOSProcStkInit (void (*task)(void *pd), void *ppdata, void *ptos, INT16U opt);

2. 函数功能

这个函数被 SOSProcCreate()或 SOSProcCreateExt()调用，以便初始化新创建任务的堆栈结构。本函数与处理器高度相关。

3. 参数和返回值

参数：4 个。

- task：指向任务代码的指针
- pdata：当任务第一次执行时将要传入任务的用户数据结构指针
- ptos：栈顶指针。ptos 指针被默认为用户堆栈入口指针。如果 SOS_STK_GROWTH 被置 1，那么，ptos 指向用户堆栈的最高有效地址。同样地，如果 SOS_STK_GROWTH 清 0，ptos 将指向用户堆栈的最低有效地址。
- opt：指定可以改变 SOSProcStkInit()行为的选项。

返回值：用户堆栈空间的最低有效地址

4. 程序流程图见附录 A 中图 A12。

2.2.16 时钟节拍函数

函数 SOSTimeTick()是时钟节拍函数。

1．函数原型

void SOSTimeTick (void);

2．函数功能

本函数用来通知 Particula OS 发生了系统时钟，该函数被时钟 ISR 调用，也可以被更高优先级进程调用。

3．参数和返回值

参数：无

返回值：无

4．程序流程图见附录 A 中图 A13。

2.2.17 提供系统版本信息

函数 SOSVersion ()提供系统版本信息。

1．函数原型

INT16U SOSVersion (void);

2．函数功能

该函数返回 Particula OS 的版本。返回值是 Particula OS 的版本号的 100 倍。换句话说，如果版本是 2.00，则返回值就是 200。

3．参数和返回值

参数：无

返回值：版本信息。

4．程序流程图见附录 A 中图 A14。

第三节 进程管理设计

在 Particula OS 的设计中，进程的结构必须是一个无限的循环。进程看起来就是一个 C 函数，具有参数和返回值，只是它从不返回。任务的返回类型必须被定义成 void 型。进程结构必须是如下结构：

```
void YourTask (void *pdata)
```

```
{
    while(1)
    {
        /* 用户代码 */
        调用 Particula OS 的服务例程之一:
            SOSQPend();
            SOSSemPend();
            SOSProcSuspend(SOS_PRIO_SELF);
            SOSTimeDly();
            ......
        /* 用户代码 */
    }
}
```

2.3.1 进程管理系统调用列表

表 2.4 进程管理系统调用列表

进程管理系统调用函数原型	
函数原型	函数功能
SOSProcCreate（）	进程的创建
SOSProcSuspend（）	进程挂起
SOSProcResume（）	进程恢复
SOSProcQuery（）	进程数据查询

2.3.2 进程的创建

函数 SOSProcCreate（）是进程的创建。

1．函数原型

INT8U SOSProcCreate (void (*task)(void *pd), void *ppdata, SOS_STK *ptos, SOS_STK *pbos, INT8U prio,INT8U *pid);

2．函数功能

该函数用来让 Particula OS 管理一个进程的执行。一个进程要么在多任务开始前被创建，要么被一个正在运行中的进程创建。一个进程不能被中断处理程序创建。

3．参数和返回值

参数：5 个。

- task：是一个指向进程代码的指针。
- pdata：是一个可选数据域的指针，该数据区域可以用来向该进程传递参数，当该进程被第一次创建的时候。当进程考虑和使用该参数时，它认为该参数被传递如下形式：

```
void Proc (void *pdata)
{
    while (1)
    {
        Proc code;
    }
}
```

- ptos：是一个指向进程栈顶的指针。如果配置数包含 SOS_STK_GROWTH 被置为 1，则栈就被认为是向下生长的（也就是说，从高存储器地指向低存储器地址方向生长），这时，指针“pstk”将指向最高有效的栈存储器地址。如果 SOS_STK_GROWTH 被置为 0，这时，指针“pstk”将指向最低有效的栈存储器地址，这时栈就向存储器地址增加的方向生长。
- pbos：是一个指针，指向栈底。如果配置数包含 SOS_STK_GROWTH 被置为 1，则栈就被认为是向下生长的（也就是说，从高存储器地指向低存储器地址方向生长），这时，指针“pstk”将指向最高有效的栈存储器地址。如果 SOS_STK_GROWTH 被置为 0，这时，指针“pstk”将指向最低有效的栈存储器地址，这时栈就向存储器地址增加的方向生长。
- prio：是进程的优先级。该优先级只能是在最低优先级和最高优先级之间的一个优先级。

返回值：3 种。

- SOS_NO_ERR：如果该函数成功执行，返回 SOS_NO_ERR
- SOS_PRIO_EXIT：如果进程优先级已经存在
- SOS_PRIO_INVALID：如果你指定的进程优先级高于所允许的最大值，也就是说，进程优先级>= SOS_LOWEST_PRIO。

4. 程序流程图见附录 A 中图 A15。

2.3.3 进程挂起

函数 SOSProcSuspend（）是进程挂起。

1．函数原型

INT8U SOSProcSuspend (INT8U pid);

2．函数功能

该函数终止进程的执行，直至被恢复。

3．参数和返回值

参数：pid

- 是将被挂起的进程号。

返回值：3 种。

- SOS_NO_ERR：如果该进程已经被挂起
- SOS_PROC_SUSPEND_IDLE：试图挂起空闲任务
- SOS_PROC_SUSPEND_PRIO：给出的进程号无效

4．程序流程图见附录 A 中图 A16。

2.3.4 进程恢复

函数 SOSProcResume（）是进程恢复。

1．函数原型

INT8U SOSProcResume（INT8U pid）;

2．函数功能

该函数可用来恢复运行先前挂起的进程，这是一个显式的进程恢复。

3．参数和返回值

参数：pid

是待恢复的进程号。

返回值：3 种。

- SOS_NO_ERR：试图恢复的进程已经恢复
- SOS_PRIO_INVALID：进程无效
- SOS_PROC_NOT_SUSPENDED：试图恢复的进程根本没有挂起

4．程序流程图见附录 A 中图 A17。

2.3.5 进程数据查询

函数 SOSProcQuery（）是进程数据查询。

1．函数原型

INT8U SOSProcQuery（INT8U pid, SOSPCB *ppdata）;

2．函数功能

该函数用来获取一特定进程 PCB 的副本。

3．参数和返回值

参数：2 个。

- INT8U pid

  是要获取的进程号。

- SOSPCB *ppdata

  传递该进程的参数。

返回值：3 种。

- SOS_NO_ERR：查询请求得到有效执行。
- SOS_PRIO_INVALID：如果你指定的优先级高于允许的最大值，也就是说，优先级>= SOS_LOWEST_PRIO。这时的另一种可能情况是，你也没有指定优先级是 SOS_PRIO_SELF。
- SOS_PRIO_ERR：所请求查询的进程不存在。

4．程序流程图见附录 A 中图 A18。

第四节 时间管理设计

Particula OS 提供定时中断，叫做时钟节拍。时钟节拍的实际频率由应用程序根据需要指定。时钟节拍的频率越高，系统的负荷就越重。

2.4.1 时间管理系统调用列表

表 2.5 时间管理系统调用列表

时间管理系统调用详细定义	
函数原型	函数功能
void SOSTimeDly（INT16U ticks）	任务延时（以节拍为单位）
INT8U SOSTimeDlyResume（INT8U pid）	恢复延时的任务
INT32U SOSTimeGet（void）	获取系统时间
void SOSTimeSet（INT32U ticks）	设置系统时间

2.4.2 任务延时

函数 SOSTimeDly ()是任务延时。

1．函数原型

void SOSTimeDly (INT16U ticks);

2．函数功能

该函数用来延时正在运行的进程的执行，直到所指定的系统时钟嘀嗒数期满为止。如此就等同于在进程内延时一段时间一样。

如果所指定的系统时钟嘀嗒数为 0，那么这时将不会有延时产生。如果所指定的系统时钟嘀嗒数大于 0，将会导致发生一个上下文切换。

注意，任务延时功能的使用，只能由正在运行着的进程延时它自己时使用。

3．参数和返回值

参数：1 个。

- ticks：指的是打算延时的系统时钟嘀嗒数。如果将其指定为 0，则进程将不会有延时。

返回值：无。

4．程序流程图见附录 A 中图 A19。

2.4.3 恢复延时的任务

函数 SOSTimeDlyResume（）是恢复延时的任务。

1．函数原型

INT8U SOSTimeDlyResume（INT8U pid）;

2．函数功能

该函数用来恢复运行一个进程，并且该进程是先前使用 SOSTimeDly()而进入挂起的进程，对于这种情况，可以使用 SOSTimeDlyResume()。

注意，不能使用这个函数来恢复一个进程的执行，如果该进程是在等待一个事件，并且该事件已经设置了超时限制。如果在这种情况下，使得进程就如同是一个超时发生一样。

3．参数和返回值

参数：1 个。

- pid：指定需要恢复延时的进程号。

返回值：4 种。

- SOS_NO_ERR：进程恢复得到成功执行。
- SOS_PRIO_INVALID：如果在参数中所指定进程的优先级大于最大允许的优先级，也就是说，优先级>= SOS_LOWEST_PRIO
- SOS_TIME_NOT_DLY：进程延时的原因不是由于调用了 SOSTimeDly()，这时不能使用 SOSTimeDlyResum()。
- SOS_PROC_NOT_EXIST：所期望的进程并不存在。

4. 程序流程图见附录 A 中图 A20。

2.4.4 获取系统时间

函数 SOSTimeGet（）是获取系统时间。

1. 函数原型

INT32U SOSTimeGet（void）;

2. 函数功能

可以从应用程序中调用该函数，从而获得系统时间。该时间的表示，是以一个 32 位的计数器为依据，该计数器记录着系统时钟嘀嗒数。

3. 参数和返回值

参数：无

返回值：返回系统嘀嗒数。

4. 程序流程图见附录 A 中图 A21。

2.4.5 设置系统时间

函数 SOSTimeSet（）是设置系统时间。

1. 函数原型

void SOSTimeSet（INT32U ticks）;

2. 函数功能

对记录着系统时钟嘀嗒数的计数器进行赋值。

3. 参数和返回值

参数：1 个

- ticks：为计数器指定的新值。

返回值：无

4. 程序流程图见附录 A 中图 A22。

第五节 信号量设计

信号量用于进程间的通信，实现进程的同步互斥。

信号量由 2 部分组成：一部分是 16 位的无符号整型信号量的计数值（0~65535）；另一部分是由等待该信号量的任务组织成的等待任务表。

使用信号量保证进程同步的示例如下。一个进程可以调用 SOSSemPend()函数来请求一个信号量，然后通过调用 SOSSemPost()函数来释放一个信号量。

```
void Task1 (void *pdata)
{
    ......
    while (1)
    {
        SOSSemPend();   /* 获得对资源的访问权 */

            /* 进程获得信号量,对资源进行访问 */

        SOSSemPost();   /* 释放对资源的访问权 */
    }
}
```

2.5.1 信号量管理系统调用列表

表 2.6 信号量管理系统调用列表

信号量管理系统调用函数原型	
函数原型	函数功能
SOS_EVENT *SOSSemCreate (INT16U cnt)	创建信号量
INT16U SOSSemAccept (SOS_EVENT *pevent)	无等待请求信号量
void SOSSemPend (SOS_EVENT *pevent, INT16U timeout, INT8U *err)	等待信号量
INT8U SOSSemPost (SOS_EVENT *pevent)	发出信号量
INT8U SOSSemQuery (SOS_EVENT *pevent, SOS_SEM_DATA *ppdata)	查询信号量

2.5.2 创建信号量

函数 SOSSemCreate (INT16U cnt)是创建信号量。

1．函数原型

SOS_EVENT *SOSSemCreate (INT16U cnt);

2. 函数功能

该函数可以创建信号量。

3. 参数和返回值

参数：1 个。

- cnt：是信号量的初始值。如果该值为 0，则没有资源可用，或者是没有事件发生。可以通过指定该参数的值，来初始一个信号量，有多少资源可用。举例来说，如果你有 10 个资源，则你可以为信号量初始化为 10。

返回值：2 种。

- 返回值不为 0：返回值是一个指针，如果不为 0，则表明返回值代表着一个事件控制块。
- 返回值为 0：返回值为 0，表明没有可用的事件控制块。

4. 程序流程图见附录 A 中图 A23。

2.5.3 无等待请求信号量

函数 SOSSemAccept ()是无等待请求信号量。

1. 函数原型

INT16U SOSSemAccept (SOS_EVENT *pevent);

2. 函数功能

该函数检查一个信号量，看其是否有一个资源可用，或是有事件发生。当没有资源可用，或者是没有事件发生时，SOSSemPend()并不像 SOSSemAccept()那样，并不挂起调用该函数的进程。

3. 参数和返回值

参数：1 个。

- pevent：指向事件控制块的指针。

返回值：2 种。

- 大于 0：如果资源可用，或者是事件发生，则信号量就获得该资源。
- 等于 0：如果资源不可用，或者是事件没有发生。

4. 程序流程图见附录 A 中图 A24。

2.5.4 等待信号量

函数 SOSSemPend (SOS_EVENT *pevent, INT16U timeout, INT8U *err)是等待信号量。

1．函数原型

void SOSSemPend (SOS_EVENT *pevent, INT16U timeout, INT8U *err);

2．函数功能

该函数的功能是等待一个信号量。

3．参数和返回值

参数：3 个。

- pevent：是一个指向事件控制块的指针，该指针就是所期望的信号量。
- timeout：是一个可选的超时时间段，以时钟嘀嗒数为单位。如果该数非 0，则进程在等待资源变为可用，或者是等待时间出现的过程中，对等待时间进行考虑，如果所指定的超时时间到，则进程就不再等待。如果该数为 0，则进程将会永远的等待下去，除非资源变为可用，或者是所等待的事件发生了。
- err：是一个指向错误信息的指针。可能的错误信息包括如下这些。

表 2.7 函数 SOSSemPend()参数 err 的意义

错误信息	意义
SOS_NO_ERR	调用是成功的，并且进程得到了需要的资源，或是所等待的事件发生了。
SOS_TIMEOUT	在超时时限内，没有等到所要求的资源，或者是所等待的事件发生。
SOS_ERR_EVENT_TYPE	没有给信号量传递指针。
SOS_ERR_PEND_ISR	如果从中断处理程序中调用了该函数，则将导致挂起。

返回值：无。

4．程序流程图见附录 A 中图 A25。

2.5.5 发出信号量

函数 SOSSemPost ()是发出信号量。

1．函数原型

INT8U SOSSemPost (SOS_EVENT *pevent);

2．函数功能

该函数的功能是向信号量发出一个信号。

3．参数和返回值

参数：1 个。

- pevent：指向事件控制块的指针，该事件控制块是所期望信号量的。

返回值：3 种。

- SOS_NO_ERR：函数调用成功，向信号量发出了一个信号。
- SOS_SEM_OVF：信号量超出了极限。也就是说，向信号量发出的信号，超出了使用 SOSSemAccept()或 SOSSemPend()向该信号量所发出的信号量请求。
- SOS_ERR_EVENT_TYPE：没有传递指针给信号量。

4．程序流程图见附录 A 中图 A26。

2.5.6 查询信号量

函数 SOSSemQuery ()是查询信号量。

1．函数原型

INT8U SOSSemQuery (SOS_EVENT *pevent, SOS_SEM_DATA *ppdata);

2．函数功能

该函数可以获得信号量的信息。

3．参数和信号量

参数：2 个。

- pevent：指向信号量的指针，该信号量是所期望的信号量。
- pdata：指向一个结构的指针，该结构可以包含有关信号量的信息。

返回值：2 种。

- SOS_NO_ERR：调用成功执行，获得了所要求的信号量信息。
- SOS_ERR_EVENT_TYPE：试图获得从一个非信号量数据得到信息。

4．程序流程图见附录 A 中图 A27。

第六节 消息队列设计

消息队列是 Particula OS 中另一种通讯机制，它可以使一个进程或者中断服务子程序向另一个进程发送以指针方式定义的变量。因具体的应用有所不同，每个指针指向的数据结构变量也有所不同。

2.6.1 消息队列管理系统调用列表

表 2.8 消息队列管理系统调用列表

消息队列管理系统调用函数原型	
函数原型	函数功能
void OSQInit (void)	初始化消息队列
OS_EVENT *SOSQCreate（void **start, INT16U size)	创建消息队列
void *SOSQPend（OS_EVENT *pevent, INT16U timeout, INT8U *err）	等待消息队列
INT8U SOSQPost（OS_EVENT *pevent, void *msg）	发出消息队列

2.6.2 初始化消息队列

函数 OSQInit ()是初始化消息队列。

1．函数原型

void OSQInit (void);

2．函数功能

该函数被 Particula OS 调用，用来初始化消息队列模型。作为一个内部函数，用户不能使用该函数。

3．参数和返回值

参数：无。

返回值：无。

4．程序流程图见附录 A 中图 A28。

2.6.3 创建消息队列

函数 SOSQCreate()是创建消息队列。

1．函数原型

OS_EVENT *SOSQCreate（void **start, INT16U size）；

2．函数功能

在事件控制块还有空余的情况下，本函数的功能是创建一个消息队列。

3．参数和返回值

参数：2 个。

- start：指向消息队列存储区域基地址的指针。该存储区域必须是一个指向 void 类型的指针数组。这种指针就如同下面的例子一样。

  void *MessageStorage[size]

- size：存储区域中元素的个数。

返回值：2 种。

- 不等于 0：这表明创建队列成功了，这个不等于 0 的数，就是事件控制块的指针。
- 等于 0：如果没有事件控制块可用，返回 0

4. 程序流程图见附录 A 中图 A29。

2.6.4 等待消息队列

函数 SOSQPend（）是等待消息队列。

1. 函数原型

void *SOSQPend（OS_EVENT *pevent, INT16U timeout, INT8U *err）;

2. 函数功能

本函数等待发给消息队列的一条消息。

3. 参数和返回值

参数：3 个。

- pevent：是一个指针，指向所期望的消息队列的事件控制块
- timeout：是一个超时设定，以系统时钟嘀嗒数为单位。如果是一个非 0 的数，那么，该进程将会等待消息的到来，直到超时时间到。如果超时指定为 0，那么，该进程将会一直等待下去，除非所等待的消息到来。
- err：是一个指向错误信息的指针。错误信息包括

表 2.9 函数 SOSQPend()参数 err 的意义

错误信息	意义
SOS_NO_ERR	调用是成功的，并且进程得到了需要的资源，或是所等待的事件发生了。
SOS_TIMEOUT	在超时时限内，没有等到所要求的资源，或者是所等待的事件没有发生。
SOS_ERR_EVENT_TYPE	没有给信号量传递指针。
SOS_ERR_PEND_ISR	如果从中断处理程序中调用了该函数，则将导致挂起。

返回值：2 种。

- 不等于 0：如果不等于 0，则这个非 0 数就是指向所收到的消息的指针。
- 等于 0：没有消息到达，或者是没有给消息队列传递指针。

4. 程序流程图见附录 A 中图 A30。

2.6.5 发出消息队列

函数 SOSQPost（）是发出消息队列。

1．函数原型

INT8U SOSQPost（OS_EVENT *pevent, void *msg）;

2．函数功能

本函数发出一个消息给消息队列。

3．参数和返回值

参数：2 个。

- pevent：指向一个与期望队列相关的事件控制块
- msg：指向待发送消息的指针。待发送的消息绝不能是一个空指针。

返回值：3 种。

- OS_NO_ERR：调用函数成功，消息被成功发送。
- SOS_Q_FULL：消息队列满，不能再接收任何一个消息了。
- SOS_ERR_EVENT_TYPE：没有给队列传递指针。

4．程序流程图见附录 A 中图 A31。

第三章 Particula OS 关键实现技术研究

在嵌入式操作系统 Particula OS 设计和实现的过程中，任务切换和堆栈问题是很基本的问题，也是很重要的内容，在设计中、尤其是在实现技术中一直是操作系统的关键部分，也是调试中的难点。

第一节 任务切换

正如在第一章中所阐述的任务切换一样，对于一个多任务操作系统，任务切换是一个很基础性的工作。这里就涉及到两个问题，一个是为了保证被切换掉的进程能够在将来的时刻可以恢复在 CPU 上的运行，必须在切换之前保存正确、合适的上下文信息。另一个问题就是如何进行切换，硬件是如何动作的，切换是如何完成的。

3.1.1 上下文的内容

如上所述，任务切换中需要保证上下文的信息是正确、合适的。因此，需要根据处理器的具体情况，定义上下文的具体内容，保证任务切换的正确运行。

从常规意义上来讲，一般的 CPU 寄存器最基本构成应该包括程序计数器（PC），状态寄存器（PSW），通用寄存器，这些寄存器构成了上下文的基本内容。除此之外，不同的开发工具在编译和执行程序时也会使用一些 CPU 中的寄存器，因此，这些内容也成为了上下文的一部分。

下面通过 Particula OS 在 PIC18F8621/8622 上的一个运行实例，如表 3.1 所示，具体说明上下文所包含的内容。

表 3.1 PIC18F8621/8622 中上下文内容

寄存器	作用
PC	程序计数器
STATUS	状态寄存器
BSR	块选择寄存器
FSR0	指向 RAM 的指针，有时作为保存中间结果的寄存器用
FSR1	堆栈指针
FSR2	帧指针
TBLPTR	指向 FLASH 的指针
PROD	乘法结果、返回值和中间计算

需要指出的是，由于 PIC18F8621/8622 在硬件电路设计上的特点，将 CPU 和外设模块用来控制器件操作的寄存器统称为特殊功能寄存器 SFR[7]。换句话说，PIC 的 SFR 是通常 CPU 中 PC、PSW 和通用寄存器的合集。

表 3.1 中列出的寄存器都是 SFR，该上下文的定义在 PIC 中可以满足 Particula OS 的设计需要。

3.1.2 任务切换的实现

有了上下文的具体内容，任务切换的过程就是要将一个进程的上下文保存，以便在将来的某个时刻恢复运行时使用；然后将另一个进程的上下文恢复，交付 CPU 的控制权，并让其运行。

1. 任务切换之前的数据状况

借鉴文献[8]的表现形式，在图 3.1 给出了 Particula OS 在进行上下文切换之前，两个进程 A、B，以及 CPU 内部的数据状况。

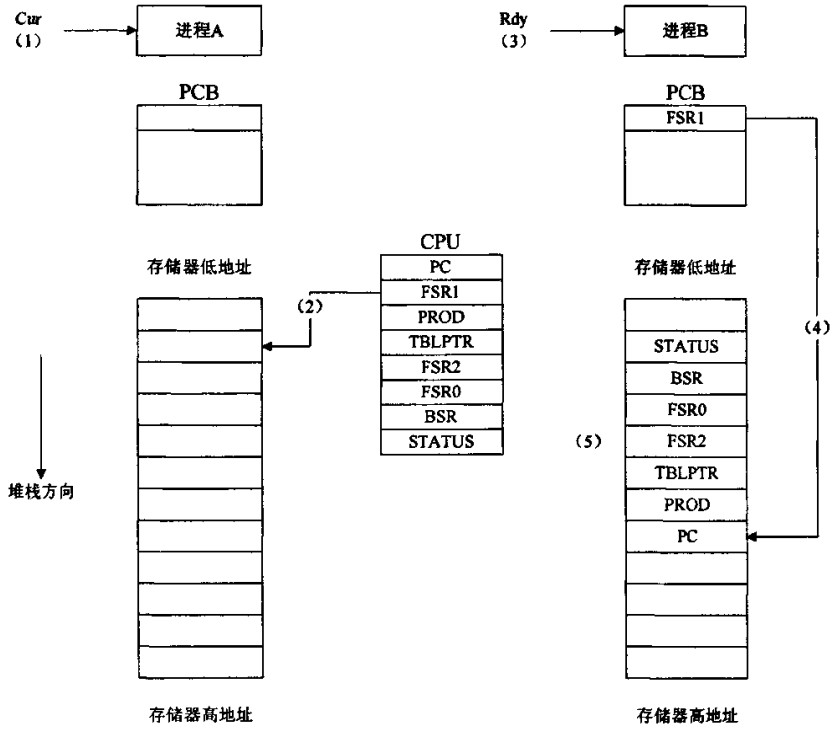


图 3.1 Particula OS 任务切换之前数据状况

在图 3.1 中，存在两个进程 A 和 B：

(1) Cur 指向即将被挂起的进程 A。

(2) 在 CPU 上运行的进程 A，其 FSR1 指向了进程堆栈的堆栈顶。

(3) Rdy 指向了一个就绪进程 B，即将投入运行。

(4) 进程 B 存储着指向进程堆栈的栈顶指针。

(5) 即将运行的进程 B，在其进程堆栈中存储着在以前某个时刻保存的上下文，并即将恢复到 CPU 的各个寄存器中去。

2. 上下文保存

在给出了 Particula OS 在进行任务切换之前的进程和 CPU 的数据状况之后，任务切换首先就是上下文保存，就是要将正在运行的进程的运行环境，保存在进程堆栈中，以便在以后的某个时刻恢复。见图 3.2。

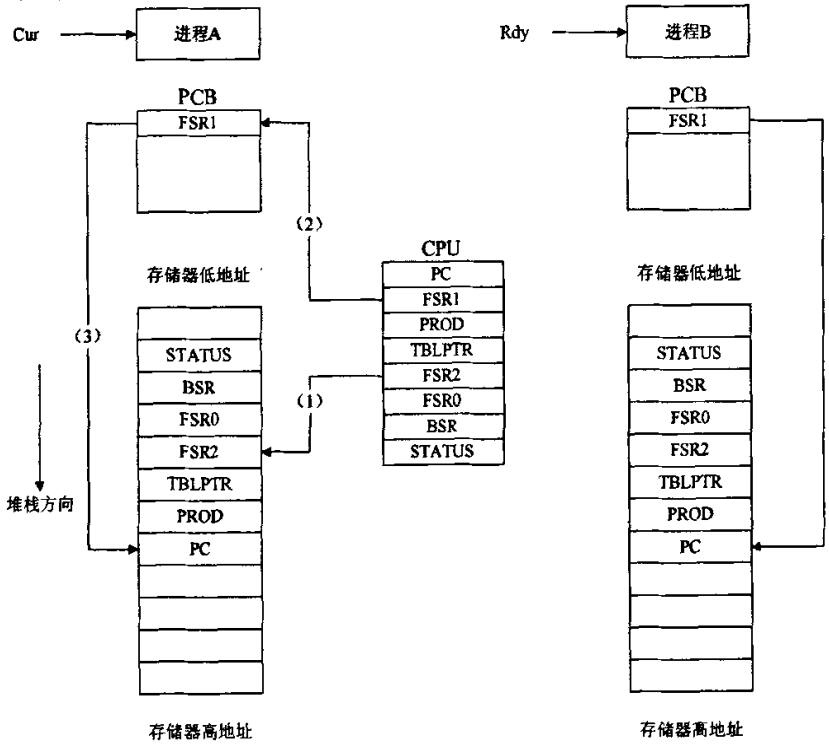


图 3.2 上下文保存

在图 3.2 中，保存进程 A 的上下文：

(1) 保存 CPU 中的寄存器到进程 A 的堆栈中。保存顺序依次是 STATUS、BSR、FSR0、FSR2、TBLPTR、PROD 和 PC。

(2) 然后将进程堆栈指针 FSR1 保存在进程 A 的 PCB 中。

(3) 这时 FSR1 指向进程堆栈的栈顶。若在以后的某个时刻需要恢复进程 A 的执行时，即可通过 FSR1 找到先前被保存的上下文。

3. 上下文恢复

在完成了一个进程的上下文保存之后，就需要将即将投入运行的进程的上下文进行恢复，装入 CPU 的各个寄存器中，见图 3.3。

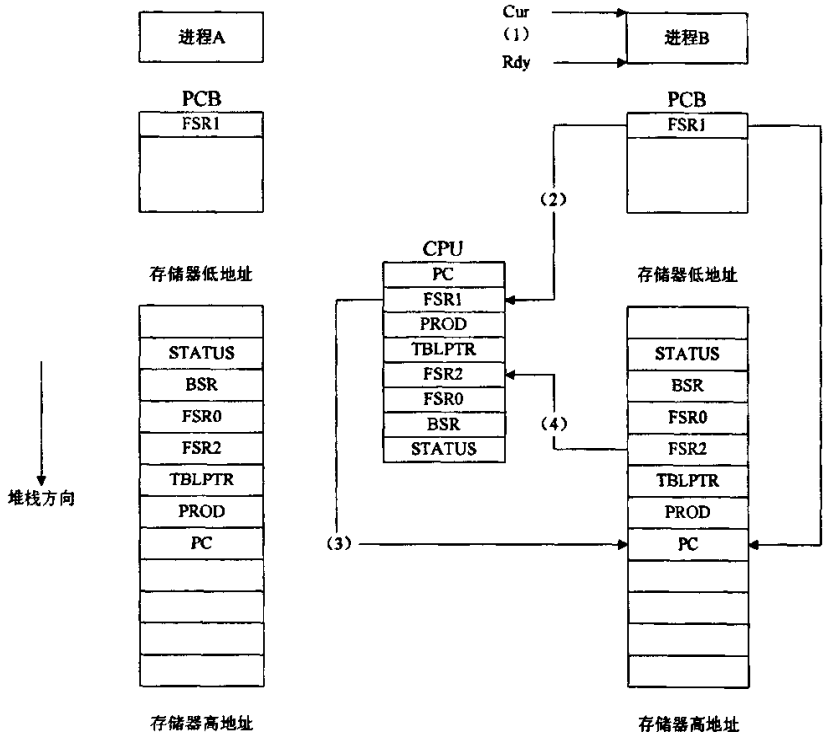


图 3.3 上下文恢复

图 3.3 表明了上下文的恢复过程：

(1) 指针 Cur 指向进程 B。由于进程 B 是即将要投入运行的进程，也就是说，下一时刻在 CPU 上运行的进程是进程 B。

(2) 将保存在进程 B 的 PCB 中的堆栈指针 FSR1 恢复到 CPU 的寄存器 FSR1 中。

(3) CPU 在得到 FSR1 之后，就得到了堆栈的栈顶。

(4) 按照和先前压栈的相反顺序进行出栈操作，将堆栈中的值恢复到 CPU 的

各个寄存器中。

在经过了上述步骤之后，进程 B 的上下文已恢复到 CPU 中，这时，在 CPU 的下一个指令周期到来时，会根据 PC 的值取出指令，而这条指令恰是先前进程 B 被暂停执行的断点。如此，进程 B 就可以从断点处继续执行。至此，就完成了一次完整的任务切换过程。

第二节 堆栈的设置和使用

从上述的任务切换的过程中可以知道，每个进程也都需要使用一定数量的数据空间来存储自身在运行过程中所需要的临时信息，亦即进程堆栈。

一方面，凡一个进程，就需要一个进程堆栈；另一方面，嵌入式系统中的数据内存往往非常有限。因此，一个进程堆栈需要占据多大空间，如何尽可能减少进程堆栈的空间占用，却又必须保证能够容纳足够多的信息，成为在嵌入式操作系统设计和实现中的一个重要问题。

3.2.1 堆栈大小对内存的影响

如前所述，系统中的每一个进程，都需要一个进程堆栈。但是嵌入式系统中的数据内存往往非常有限。如果进程堆栈占用空间太大，则空间有限，势必影响应用程序的运行。如果占用空间太小，就有可能丢失进程运行过程中的运行数据。

根据 PIC18F8621 的硬件特点，其数据类型定义如表 3.2 所示。

表 3.2 PIC18F8621 中数据类型

数据类型	大小（字节）
字符型	1
整型	2
Short long int	3
长整型	4
数据存储器指针	2
Near 程序存储器指针	2
Far 程序存储器指针	3

根据上表，估算 Particula OS 占用数据内存约 1400 字节。假定在某个应用中（为了说明问题，特举一个极端的例子以作说明），驱动程序、中间件和应用程序的数据内存占用量如表 3.3 所示。

从该表中可以看出，堆栈所占用的数据内存只能是余量内存的部分或全部。

换句话说，整个内存空间是由 Particula OS，驱动程序以及中间件，应用程序和堆栈所瓜分的了。从表 3.3 所举的例子中可以得知，堆栈的大小大约为 300 字节左右。受空间所限，即便假定最多只有 8 个进程（尽管 Particula OS 最多可以支持 64 个进程），那么每个进程的堆栈大小不会超过 38 字节。

表 3.3 PIC18F8621 中 RAM 占用情况

类型	大小（字节）
PIC18F8621 的 RAM	3840
Particula OS 占用量	1400
驱动程序和中间件	300
应用程序	1840
余量	300

由 3.1.1 节可知，一个上下文所包含的内容及其所占用的数据内存大小如表 3.4 所示。

表 3.4 Particula OS 上下文占用字节情况

寄存器	大小（字节）
PC	3
STATUS	1
BSR	1
FSR0	2
FSR1	2
FSR2	2
TBLPTR	3
PROD	2
合计	16

由此可知，一个进程堆栈最多可以容纳两帧上下文，也就是说只能容纳一次中断嵌套。这种结果不论在任何情况下，都是无法让人可以认为这是一个安全的数值。可见，堆栈的大小是必须要认真考虑的。

3.2.2 堆栈设计及优化

从上一节的讨论可知，堆栈的大小在嵌入式系统中是需要认真考虑的一个因素。那么如果硬件的数据内存足够大，这个问题就会迎刃而解。然而事实上，在嵌入式系统中，出于系统功能要求、硬件选型实际和成本的考虑，数据内存永远是一种紧缺资源，所以就有必要在堆栈设计上加以认真考虑，并进行优化。

在 Particula OS 中，堆栈的一般设计方法如图 3.4 所示。

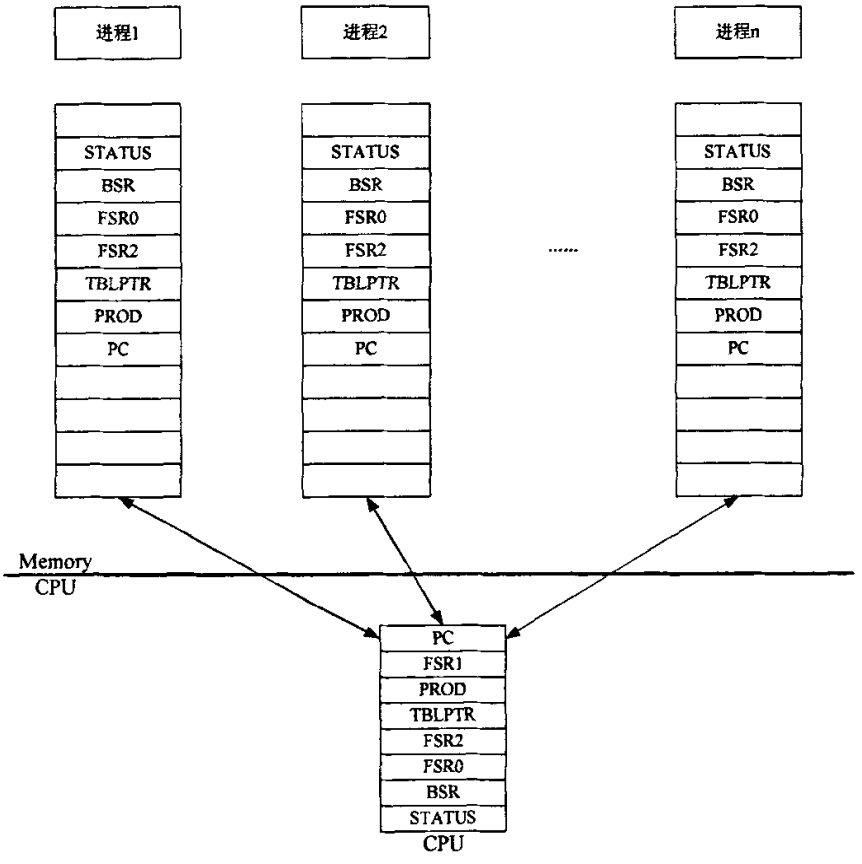


图 3.4 Particula OS 堆栈设置示意图

从这种堆栈设置方法中可以看出，进程堆栈必须要考虑中断嵌套对堆栈大小的占用[9, 10]，所以需要占用较大的空间。然而在实际中，考虑到硬件特点，这一问题将会得到较大改进和优化。

从 Particula OS 在 P89C668 上的应用实际来看，由于结合了硬件特点[11]，堆栈的空间占用问题得到了较好的优化和解决。

Particula OS 在 P89C668 上的堆栈设置示意图如图 3.5。

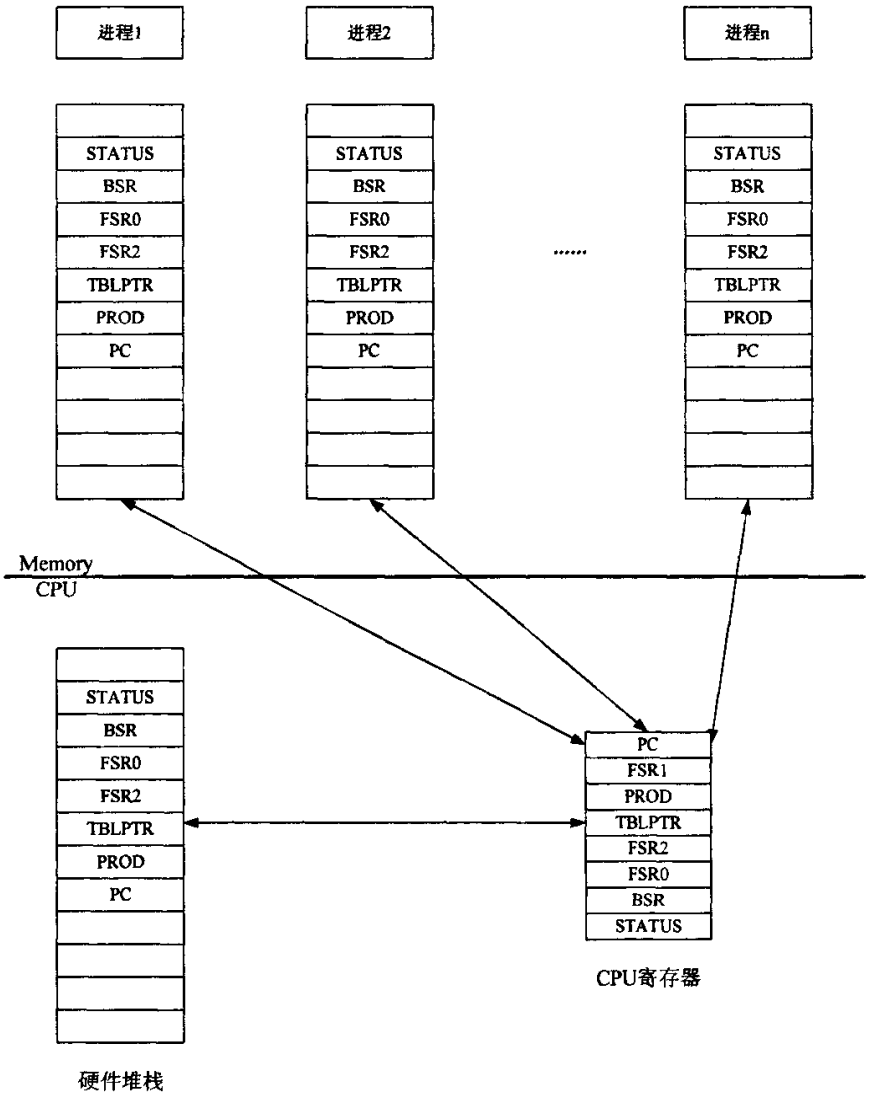


图 3.5 Particula OS 在 P89C668 上的堆栈设置示意图

如图 3.5，P89C668 自身带有一个硬件堆栈。利用这一特性，可以在中断发生时的现场保存在硬件堆栈中。这样一方面可以提高速度，一方面可以减少了进程堆栈的空间需求。

上述设计方法，由于充分考虑到了硬件的实际特点，很好的解决了进程堆栈的空间占用问题，达到了优化的效果。

第四章 Particula OS 时间性能测定与分析

不论在理论研究，还是工程实践中，实时性都是嵌入式操作系统的重要特征，是设计和使用中需要考虑的重要因素。主要来说，实时性从外部主要表现为对于外部异步事件反应的敏捷程度上，而内部则主要表现在由于操作系统的存在所带来的 CPU 额外代价的大小上。

本章首先给出时间性能概念，然后描述了在不同硬件平台上为测定时间性能的实验方案设计，最后根据得到的实验数据，给出 Particula OS 时间性能的分析。

第一节 时间性能概念

在 Particula OS 中有两个重要的时间性能[12, 13]指标。中断响应时间主要反映 Particula OS 对外部异步事件的反应速度，任务切换时间[14]主要反映了为完成任务切换所花费的额外 CPU 代价。

4.1.1 中断响应时间

论文在 1.3.4 中断这一节已经指出，“中断的出现使得处理器可以灵活处理异步事件。当某个事件发生时，会触发中断电路，这一过程被称为中断请求；CPU 注意到这一情况后，就中断正在执行的任务，开始着手来处理这个事件，这一过程被称为中断响应”。

因此，中断响应是指，从中断事件发生，到开始执行用户的中断服务子程序代码来处理这个中断的过程。这一过程所花费的时间就是中断响应时间，记为 T_{IR}（其中 IR 表示 Interrupt Response），中断响应时间测量原理图如图 4.1 所示。

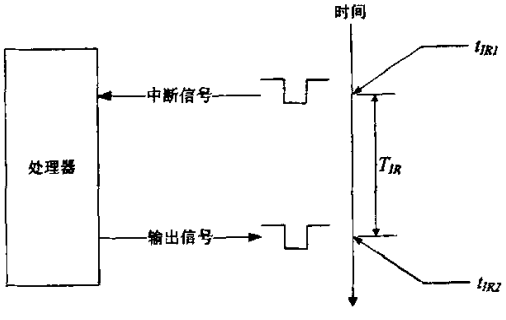


图 4.1 中断响应时间测量原理图

中断事件的发生，就会触发中断电路，引起处理器某引脚的电平变化。此为中断发生的计时起点，将这一时刻记为t_{IR1}。

每一个中断事件总意味着外部发生了某种事件，需要系统做出反应，进行相应的处理。这些处理事件的代码，通常是中断服务子程序的主角。如图 4.1，可以在这时改变处理器某引脚的电平，并将发生这一电平变化的时刻记为t_{IR2}。

显然，中断响应时间可以由式（4.1）给出：

$$T_{IR} = t_{IR2} - t_{IR1} \tag{4.1}$$

4.1.2 任务切换时间

任务切换（Task Switch）是指，当一个任务停止运行时，保存自己的上下文，然后将另一个要投入运行的任务进行上下文恢复的过程。任务切换也称为上下文切换（Context Switch）。那么任务切换时间T_{CS}（其中 CS 表示 Context Switch）就是完成任务切换的时间。

需要再次说明的是，T_{CS}仅指上下文切换的时间。而一个任务停止运行后，如何确定是哪一个任务需要投入运行等等，这些过程所花费的时间不包括在T_{CS}内。

任务切换时间测量原理图如图 4.2 所示。

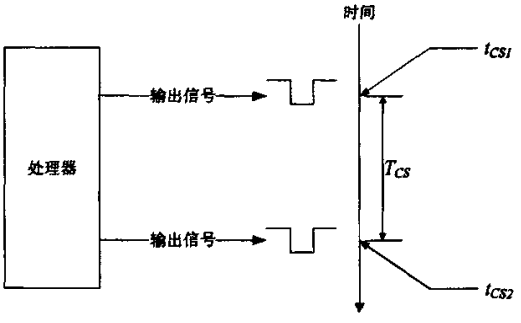


图 4.2 任务切换时间测量原理图

如图 4.2，当开始执行任务切换之前，改变处理器某引脚的电平，并将发生这一电平变化的时刻记为t_{CS1}。

当完成了任务切换之后，同样，改变处理器某引脚的电平，并将发生这一电平变化的时刻记为t_{CS2}。

则，任务切换时间可以由式（4.2）给出：

$$T_{CS} = t_{CS2} - t_{CS1} \tag{4.2}$$

第二节 基于 PIC18F452 的实验设计与实验数据

PIC18F452 是一种 RISC 单片机，共有 75 条指令，大多指令都能在一个指令周期内完成。其硬件结构图[15]如图 4.3 所示。

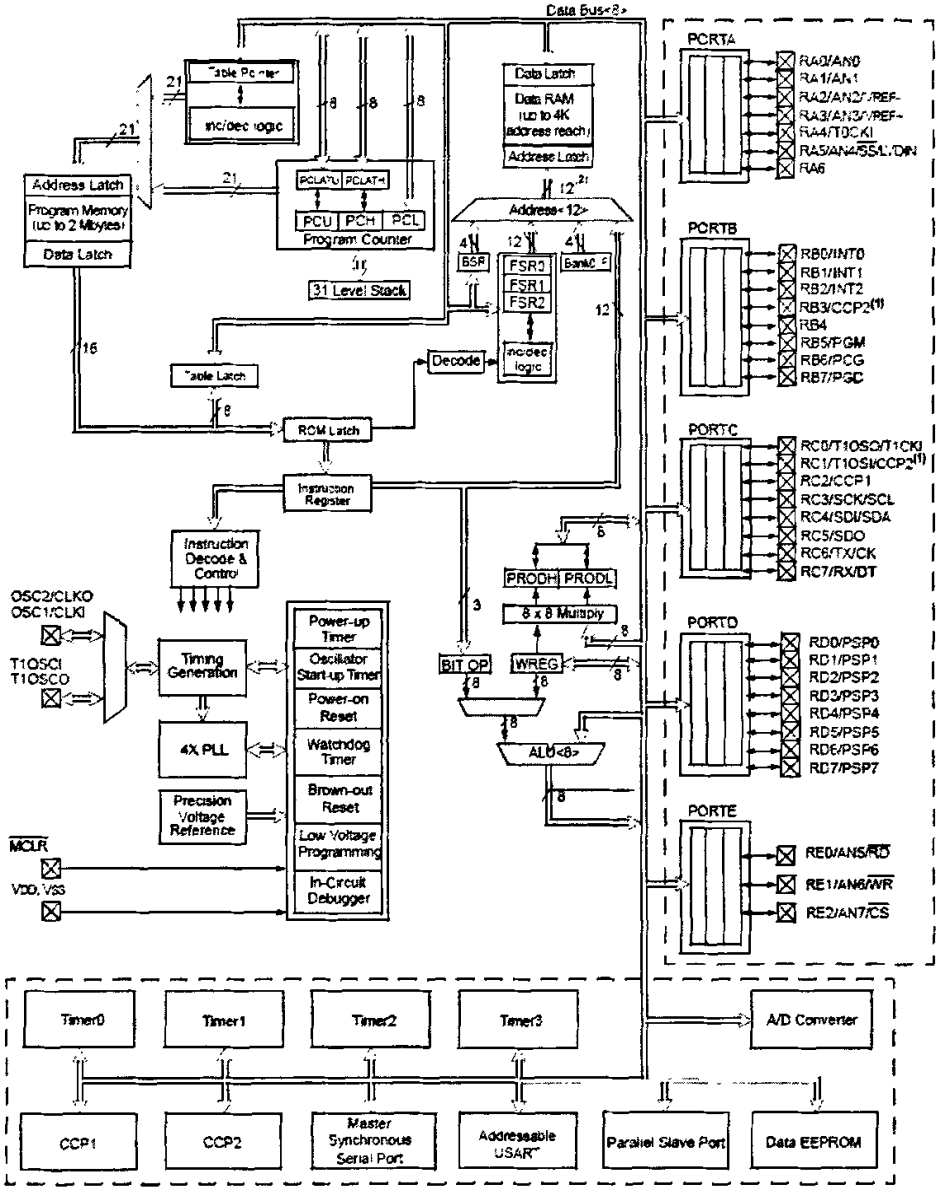


图 4.3 PIC18F452 结构图

PIC18F452 具有多种封装方式，实验中使用 DIP 封装形式。图 4.4 中给出了 PIC18F452 的引脚图[15]。

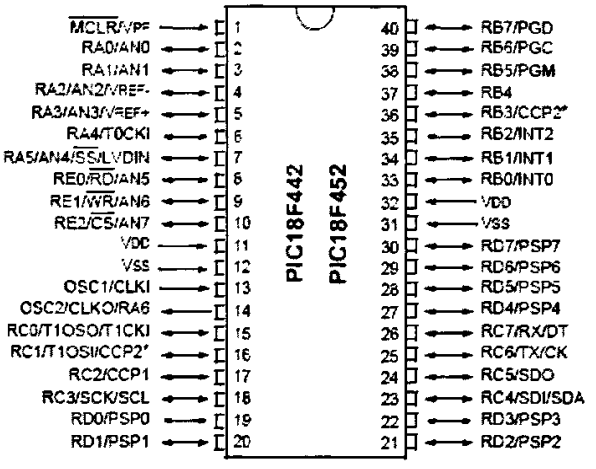


图 4.4 PIC18F452 引脚图（DIP）

实验所用的电路板实物[16]，如图 4.5 所示。

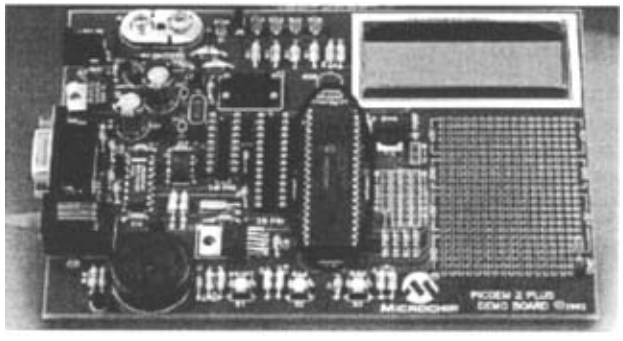


图 4.5 基于 PIC18F452 实验电路板实物图

图中以红色标出部分是单片机 PIC18F452，其采用 DIP 封装形式，共有引脚 40pin。除此之外，在实验中还用到了电路板上的 4 个 LED，它们与 PIC18F452 的 PORTB 相连接。在后面的实验中再详细介绍。

4.2.1 中断响应时间实验设计

1．中断源

中断源采用 PIC18F452 中的 Timer0 模块，通过对相应寄存器的配置，设定

Timer0 的中断频率为 50Hz。

2. 实验的设计原理

如前所述，式（4.1）给出了中断响应时间的计算方法

$$T_{IR} = t_{IR2} - t_{IR1}$$

当 PIC18F452 得知中断事件发生时，硬件逻辑就跳转到 0x08 的位置执行，而这一位置被通常理解为“中断向量”，而硬件逻辑跳转到 0x08 的位置的时刻就是t_{IR1}。

此后，在经过中断识别、中断向量跳转、保存当前处理器各个寄存器内容等过程之后，就会执行中断服务，而这一时刻就是t_{IR2}。

得到t_{IR1}和t_{IR2}之后，就可以根据式（4.1）计算出中断响应时间了。

3. 实验方法和实验环境

实验中，使用 PIC18F452 的 PORTB 端口电平变化来表示中断响应过程。具体来说，就是在系统初始化的过程中，将 PORTB<3>置为低电平。在t_{IR1}时刻将 PORTB<3>置为高电平，在t_{IR2}时刻将 PORTB<3>再次置为低电平。从而 PORTB<3>保持高电平的时间，就是中断响应时间。

通过观察图 4.6 和图 4.7 可知，PIC18F452 的引脚 PORTB<3>连接到发光二极管 RB3。

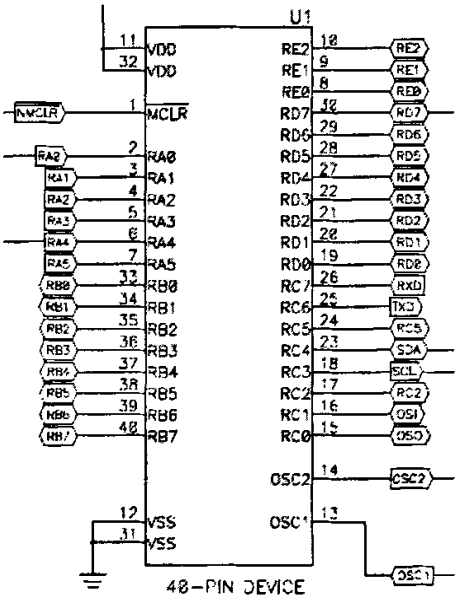


图 4.6 基于 PIC18F452 实验电路引脚图

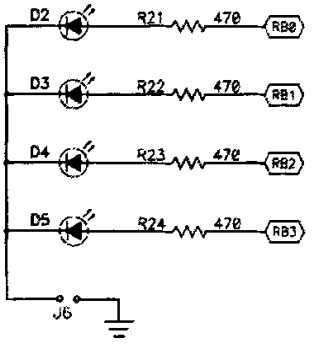


图 4.7 基于 PIC18F452 实验电路 LED 图

因此，为了测量的方便，可以直接测量 RB3，其高电平的保持时间就是中断响应时间。

4. 实验内容

如前所述，在 t_{IR1} 时刻将 PORTB<3>置为高电平，相应的汇编代码如下：

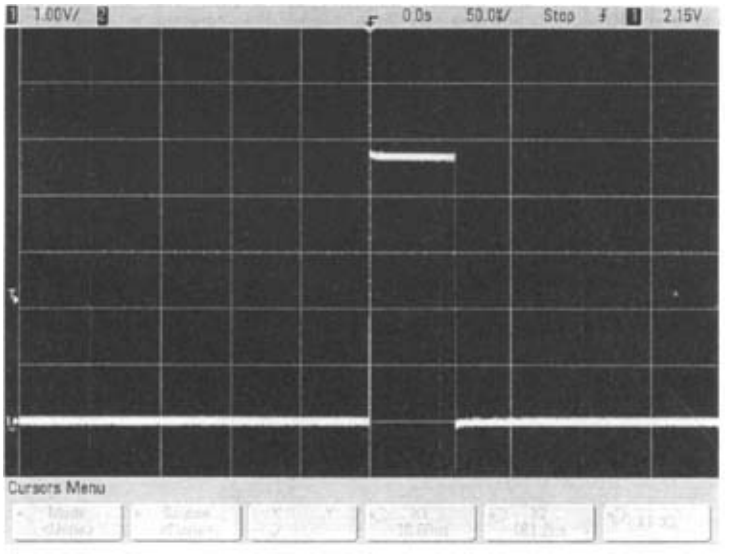
```
InterruptVector  CODE  0x08          ;中断发生时硬件自动跳转到 0x08
HighInterrutpVector
    BSF     PORTB,3,0                ;将 PORTB<3>置为高电平
    GOTO HighInterruptHandler        ;中断向量跳转
```

在 t_{IR2} 时刻将 PORTB<3>置为低电平，相应的汇编代码如下：

```
.....
BCF     PORTB,3,0                    ;将 PORTB<3>置为低电平
CALL    SOSTimeTick                  ;提供中断服务
.....
```

5. 实验结果

通过实验测定，Particula OS 的中断响应时间结果如图 4.8 所示。



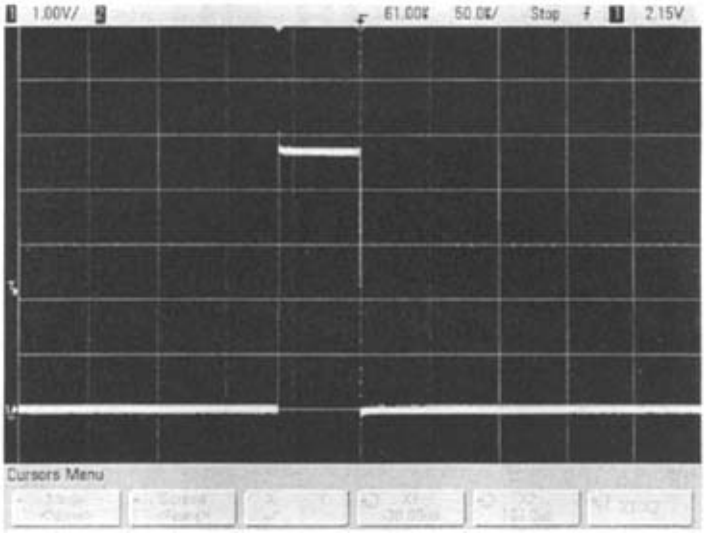
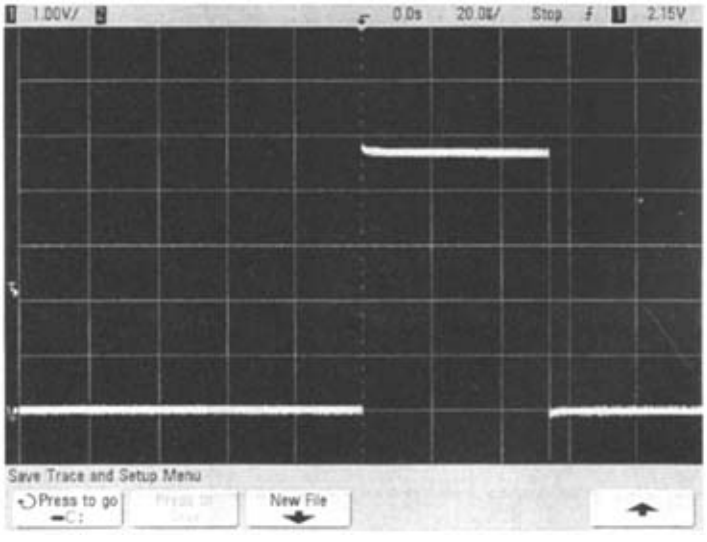


图 4.8 基于 PIC18F452 的 Particula OS 中断响应时间为 61μs

由图 4.8 可知，Particula OS 的中断响应时间是 61μs。

同时基于相同的思想和设计方法，通过实验得到 μC/OS-Ⅱ的中断响应时间[17]结果如图 4.9 所示。



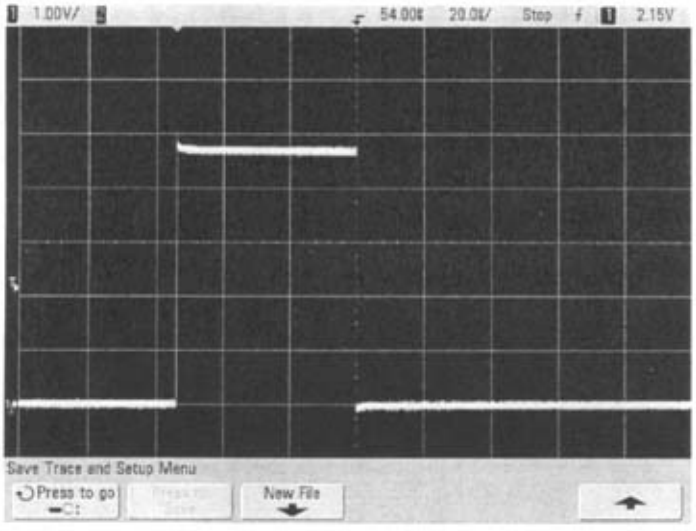


图 4.9 基于 PIC18F452 的 μC/OS-Ⅱ中断响应时间为 54μs

由图 4.9 可知，μC/OS-Ⅱ的中断响应时间是 54μs。

4.2.2 任务切换时间实验设计

1．实验的设计原理

在 4.1.2 中指出了任务切换时间的计算方法，具体的式（4.2）如下：

$$T_{CS} = t_{CS2} - t_{CS1}$$

在 Particula OS 中，任务切换功能由单独的函数来执行，函数的开始执行时刻为t_{CS1}。

同样，当完成了任务切换之后，准备执行返回指令的时刻为t_{CS2}。

得到t_{CS1}和t_{CS2}之后，就可以根据式（4.2）计算出任务切换时间了。

2．实验方法

根据实验设计原理，在系统初始化时，将 PORTB<3>置为低电平。当开始执行任务切换时，也就是在t_{CS1}时刻，将 PORTB<3>置为高电平；当任务切换完成之后，准备执行返回指令时，也就是在t_{CS2}时刻，将 PORTB<3>再次置为低电平。从而 PORTB<3>保持高电平的时间，就是任务切换时间。

根据图 4.6、图 4.7 可知，PIC18F452 的 PORTB<3>连接到发光二极管 RB3。因此其高电平的保持时间就是任务切换时间。

3．实验内容

如前所述，在t_{CS1}时刻将 PORTB<3>置为高电平，相应的汇编代码如下：

```
void SOSCtxSw(void)          ;执行任务切换功能的函数
{
_asm                         ;在 C 语言代码中插入汇编代码的关键字
    BSF     PORTB,3,0        ;将 PORTB<3>置为高电平
_endasm
……                           ;函数内的其它代码
```

在t_{CS2}时刻将 PORTB<3>置为低电平，相应的汇编代码如下：

```
……                           ;函数内的其它代码
_asm
    BCF     PORTB,3,  0      ;将 PORTB<3>置为低电平
_endasm

_asm
    RETFIE 0                 ;使用汇编代码执行“返回”功能
_endasm
}
```

在上面的代码中，需要说明的是将 PORTB<3>置为低电平的时机问题。因为按照通常的理解，C 语言中的函数以左括号开始，以右括号结束。如果没有显式使用 return 语句，C 语言编译器会自动将右括号翻译成 return 语句，从而结束函数调用。

然而在上面的代码中却使用了汇编指令 RETFIE[15]作返回功能。也就是说当处理器执行了 RETFIE 指令之后，就已经发生了程序跳转，而不再执行 C 编译器所自动产生的 return 语句了。所以 RETFIE 指令就是函数 SOSCtxSw ()的最后一条指令，也就是上面所述的t_{CS2}时刻了。所以应当在执行这条指令之前将 PORTB<3>置为低电平。

4．实验结果

通过实验测定，Particula OS 的任务切换时间结果如图 4.10 所示，可知 Particula OS 的任务切换时间是 139μs。

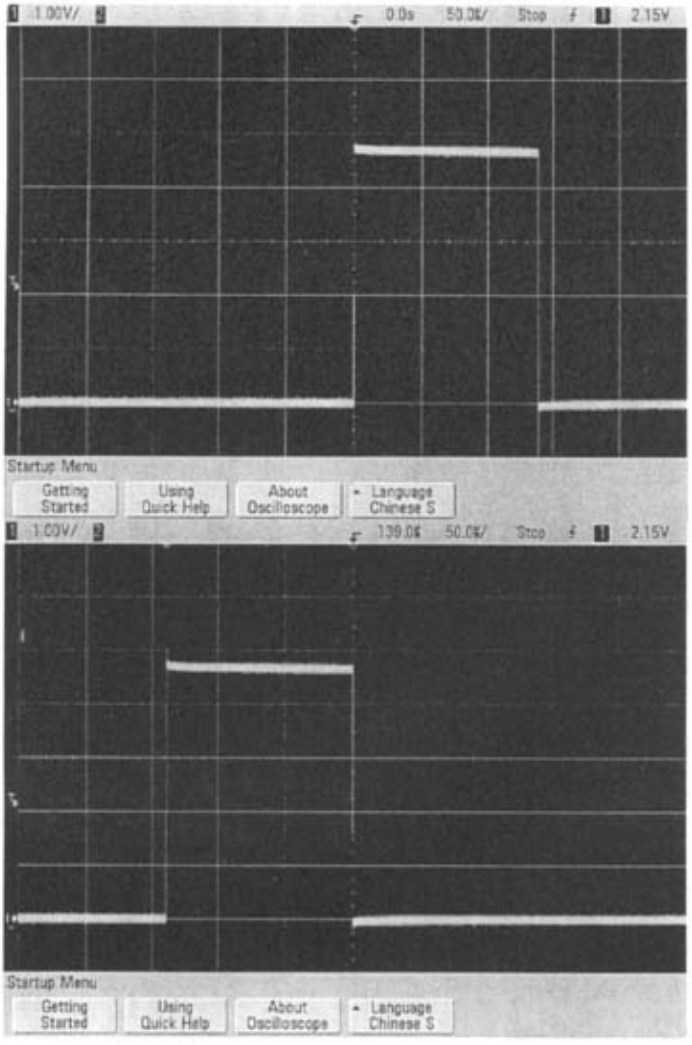


图 4.10 基于 PIC18F452 的 Particula OS 任务切换时间为 139μs

基于相同的思想和设计方法，通过实验得到 μC/OS-Ⅱ的任务切换时间[17]结果如图 4.11 所示，可知 μC/OS-Ⅱ的任务切换时间是 139μs。

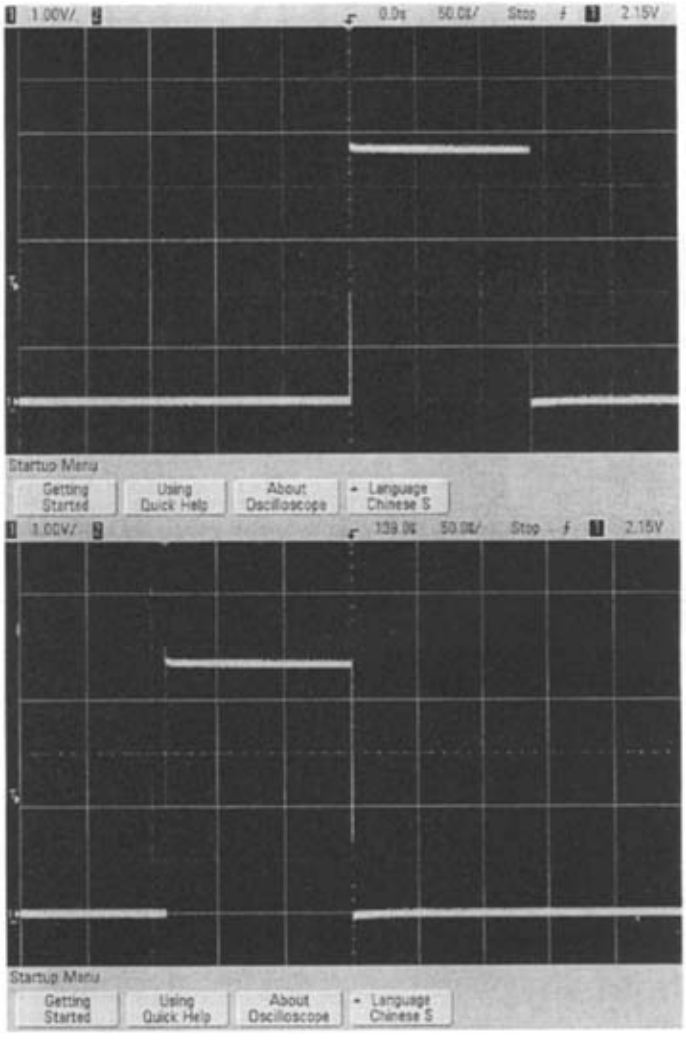


图 4.11 基于 PIC18F452 的 μC/OS-II 任务切换时间为 139μs

第三节 实验结果分析

4.3.1 实时性

由实验可知，实验的硬件平台采用的是 PIC18F452，实验电路的振荡源频率

为 4MHz[16]。由于 PIC18F452 的大多指令都能在一个指令周期内完成，并且一个指令周期可以分为 4 个振荡周期，所以，PIC18F452 中的一条指令的执行时间是 1μs。

嵌入式操作系统 Particula OS 和 μC/OS-II 的实验结果如表 4.1 所示。

表 4.1 时间性能实验结果

实时性指标	Particula OS	μC/OS-II
中断响应时间	61μs	54μs
任务切换时间	139μs	139μs

从数据中可以得知，就中断响应时间和任务切换时间这两个指标而言，Particula OS 和 μC/OS-II 相差不大。因此，从这个角度来说，它们在实时性上并无大的差距。

4.3.2 中断响应时间

从上表中可知，Particula OS 响应中断的时间为 61μs，那么依据此前有关中断和中断响应时间的定义可知，从中断发生到最终处理完中断并返回进程执行，将需要更多的时间。

因此，在实际应用 Particula OS 时，尤其是对中断的响应和处理有时间要求的时侯，就需要认真考虑，当一个中断事件发生时，究竟应该在 ISR 中放入多少处理代码，能够满足时间要求；而在某个相关的进程中放入多少处理代码，能够满足时间要求。

从另一个角度来讲，由于本实验中的 PIC18F452 的指令周期是 1μs，因此可以认为中断响应的过程中，大约执行了 61 条指令。因此，如果实际应用中需要更短的中断响应时间的话，可以考虑提高硬件速度。如果本实验中采用 20MHz 的振荡源的话，那么 Particula OS 的中断响应时间就是 12.2μs。

4.3.3 任务切换时间

从上述数据中可以看出，相比于中断响应时间而言，任务切换时间耗时较长。并且由于在从一个任务转到另一个任务的过程中，还要花费调度的时间等，因此从一个任务转到另一个任务的时间花费要比任务切换时间更大。

既然如此，明白了任务切换是一个比较费时的操作后，在实际应用中就要特别注意任务的划分，尽量使执行内容相关的代码都放在一个任务之内，尽量减少任务切换的次数，同时也就是减少了任务切换时间。

如前所述，任务切换时间是一个表征为完成任务切换所花费的额外 CPU 代价。那么减少任务切换，减少各个任务间切换的时间花费，实际上就是降低了 CPU 的额外花费，提高了 CPU 的利用率，同时也就提高了应用系统的效率。

第五章 总结与展望

在论文的这部分，对 Particula OS 的研发作一简要总结，并对后续工作进行了展望。

嵌入式操作系统的设计与开发工作不但包含了软件设计和实现，同时也要求对计算机硬件工作原理甚至具体芯片相当熟悉。在 Particula OS 的研究和开发过程中，在 Philips 公司的微控制器 P89C668 和 PIC 公司的微控制器上做了大量的实践。这部分工作也占据了 Particula OS 研发相当一部分的工作量，起了很重要的作用。但限于论文的论述重点和篇幅限制，对硬件部分，只对最必须部分进行论述，其余内容可参阅相关文献。

第一节 工作回顾

Particula OS 作为嵌入式操作系统，它是嵌入式软件的灵魂。论文从社会对嵌入式系统的实际需求出发，指明了嵌入式操作系统的重要作用。

嵌入式操作系统 Particula OS 由内核和核外子系统组成。

内核负责底层支持功能。根据微内核技术的基本思想，将 Particula OS 需要向用户提供的各种功能以系统调用的形式给出，而只保留最基本的底层支持，如中断、调度等功能。

核外子系统包括进程管理，时间管理以及进程间通信等功能。

进程管理包括进程创建、进程挂起与恢复等功能，时间管理包括延时、恢复、时间设置与获取时间等功能。

进程间通信包括信号量和消息队列。信号量为用户提供了进程间的同步互斥手段，可以有效控制进程的执行节奏，保持各个进程间的同步关系。消息队列是进程间交换信息的有效手段。

论文给出了上述各个部分的设计，在此基础上，对设计和实现中的两个关键技术进行研究。任务切换技术是多任务系统的基础，论文结合 PIC 公司微控制器 PIC18F8621/8622 的实际，详细定义了上下文的内容，阐明了上下文切换的过程。堆栈对于嵌入式系统紧张的数据内存的影响非常直接，论文根据 P89C668 的实际特点，给出了一种堆栈的优化方法。

中断响应时间和任务切换时间是 Particula OS 中具有代表性的两个时间指标。一个标志着对外部事件的反应速度，一个指明了对 CPU 的利用效率。经过在 PIC18F452 平台上的一系列实验，得到了上述两个时间指标的具体数值，并进行了分析。

清晰起见，将 Particula OS 的特点列出如表 5.1，并以 μC/OS-Ⅱ作为参照。

表 5.1 Particula OS 的特点及其与 μC/OS-Ⅱ的对比

特点	Particula OS	μC/OS-Ⅱ
调度方式	分时，优先级	优先级
大小	程序内存约 10~20KB	程序内存约 6~20KB
	数据内存约 2~3KB	数据内存约 1~2KB
可移植	良好的移植性	良好的移植性
可剪裁	良好的可剪裁性	良好的可剪裁性
可配置	良好的可配置性	良好的可配置性
实时性	几十到几百微秒	几十到几百微秒
可管理的进程数量	32	64
提供的功能	进程管理，时间管理，信号量管理，消息队列管理	任务管理，时间管理，信号量管理，互斥型信号量管理，时间标志组管理，消息邮箱管理，消息队列管理，内存管理

注意，表 5.1 中的实时性，是以中断响应时间和任务切换时间为代表的。如前所述，任务切换是一个相当费时的操作，所以它基本上是嵌入式操作系统 Particula OS 中最耗时的底层操作了。

Particula OS 满足了嵌入式系统开发和应用实际中对于嵌入式操作系统的基本要求，是一个精简而相对完备的系统，可以说“麻雀虽小，五脏俱全”。

第二节 展望

如上所述，嵌入式操作系统 Particula OS 是一个麻雀，但毕竟只是一只“小”麻雀。随着嵌入式系统的发展，人们对于嵌入式系统在低功耗，网络及无线，安全性等方面提出了更高的要求。

Particula OS 在这些方面仍面临着新的课题，不但软件上要充实对这些功能的支持，更要把握硬件发展和应用的实际，结合硬件的特点，一方面简化软件设计，另一方面也提高了软硬件的结合度，从而提高系统的性能。

参考文献

[1] 何小庆. 嵌入式实时操作系统的现状和未来. 单片机与嵌入式系统应用，2001（3）：12~13

[2] 钟锡昌. 嵌入式操作系统在中国的发展现状与前景. 信息技术与应用，2002（6）：6~9

[3] 胡祖宣. 微内核操作系统（上）. 电子科技导报，1996（5）：20~22

[4] 胡祖宣. 微内核操作系统（下）. 电子科技导报，1996（6）：16~19

[5] 付长冬，孟庆余，潘清. 基于微内核的操作系统综述. 计算机工程与科学，1997，Vol. 19（3）：72~75

[6] 陆志才. 微型计算机组成原理. 北京：高等教育出版社，2003. 133~137

[7] Microchip Technology Inc. PIC18F6525/6621/8525/8621 Data Sheet

[8] [美]Jean J. Labrosse. 嵌入式实时操作系统 μC/OS-Ⅱ（第 2 版）（邵贝贝等译）. 北京：北京航空航天大学出版社，2003. 93~95

[9] 杨少军，李杭生. μC/OS-Ⅱ任务栈处理的改进设计. 单片机与嵌入式系统应用，2004（5）：73~74

[10] 倪敏，周怡颐，杨继堂. μC/OS-Ⅱ的任务切换机理及中断调度优化. 单片机与嵌入式系统应用，2003（10）：26~30

[11] Philips Semiconductor. P89C668 Data Sheet. 2001，JUL19

[12] 李庆诚，顾健. 嵌入式实时操作系统性能测试方法研究. 单片机与嵌入式系统应用，2005（8）：19~21

[13] 刘轶欧. 嵌入式实时系统测试的充分性研究与实践：[硕士学位论文]. 成都：电子科技大学，2004

[14] 程红蓉. 一种实时嵌入式操作系统内核 DeltaCORE 的设计与实现：[硕士学位论文]. 成都：电子科技大学，2001

[15] Microchip Technology Inc. PIC18FXX2 Data Sheet

[16] Microchip Technology Inc. PICDEM™ 2 Plus User's Guide

[17] 张谦，竹利平. μC/OS-Ⅱ实时嵌入式操作系统的实时性分析与测试. 计算机工程与设计，2005，Vol.26（9）：2422~2424

致谢

选择南开大学，选择机器人所是我的幸运。在南开的生活和学习，浓厚的人文环境不但丰富了我的生命，而且也从另一个方面促进了对专业能力的培养。机器人所有着很好的科研环境，在这里使我得到了极大提高。

能够在刘景泰教授的指导下完成我的三年研究生活，也是我的幸运。我要特别感谢刘老师，没有他的指导，我的研究生生涯可能就是另一番景象了。刘老师精深的学术造诣，敏锐的眼光，开阔的视野，严谨的治学风格，乐观进取的态度感染着身边的每一个人。不论是机器人学的课堂上，工作讨论会、小组学习中，还是与我们的单独交流中，总能恰到好处的给予我们帮助，使我在人生道路上走得更好。在这里，祝愿刘老师在以后的工作和生活中身体健康，幸福快乐。

感谢孙雷老师，在刘老师留美的这些日子里，孙雷老师既是良师，又是诤友。他时刻关心我们的研究学习情况，对于我存在的问题和困难及时帮助，并且与我们小组分享着快乐。

感谢张蕾老师、张建勋老师，他们为我创造了很多有利条件，对我的工作、科研和学习帮助很大。他们也十分关心我的就业情况，给了我很多信息和帮助。我要感谢他们。

感谢卢桂章教授，赵新老师，方勇纯老师，辛运帏老师，王晓娜老师，以及实验室所有老师对我学习和生活上的热心帮助。

感谢赵春颖在课题之初求索路上给予我的帮助。当这个课题圆满完成时，却只能通过电子邮件与她分享喜果了，谨以本论文祝愿在大洋彼岸的她一切顺利！

感谢郝铭师兄无私的帮助。与李加俊、杨蕊、张星星愉快的合作经历，让人真想南开一直呆下去。感谢段跃新，陈涛，邴志刚，黄兴博，孙巍巍，杜江，张伟，王鸿鹏，景国辉，菅炜，刘征，陈新伟，李海峰，宋银灏，韩社丰，齐晓轩，郝龙的热心帮助。和小组同学们的三年生活，不仅让人想起“恰同学少年，风华正茂，书生意气，挥斥方遒。”

感谢天津南开太阳高技术有限公司的所有朋友们。

感谢我的父母们和妻子。人生一世，草木一春，唯亲情、爱情永存！

最后，感谢所有帮助过我的老师、同学和朋友们！

朱鹏辉

2007 年 4 月 16 日

附录 A Particula OS 详细设计流程图

在附录 A 中是 Particula OS 各模块的详细设计流程图，它们是详细设计的重要组成部分。论文中主要论述 Particula OS 的系统设计，而将流程图汇集于此，以作参考。

内核设计

系统初始化程序流程图如图 A1 所示。

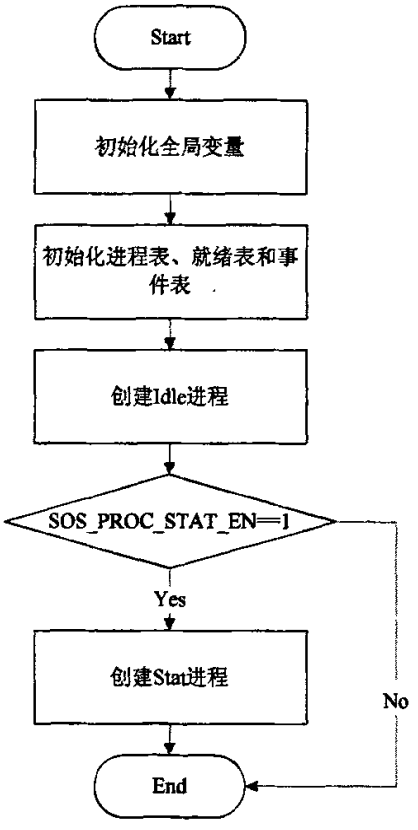


图 A1 系统初始化程序流程图

中断进入程序流程图如图 A2 所示。

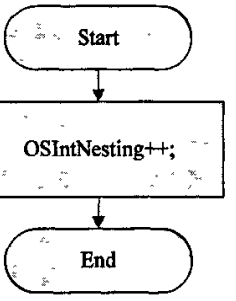


图 A2 中断进入程序流程图

中断退出程序流程图如图 A3 所示。

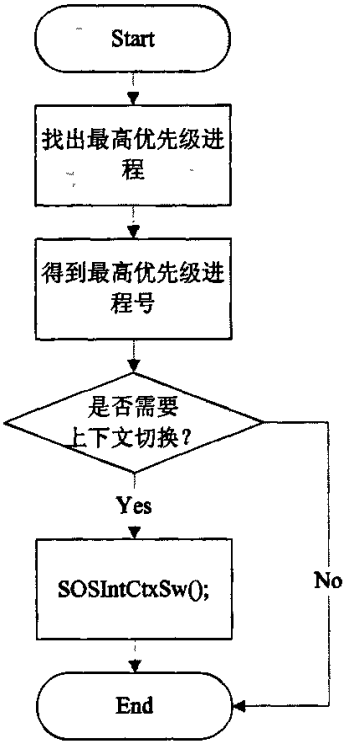


图 A3 中断退出程序流程图

进程调度程序流程图如图 A4 所示。

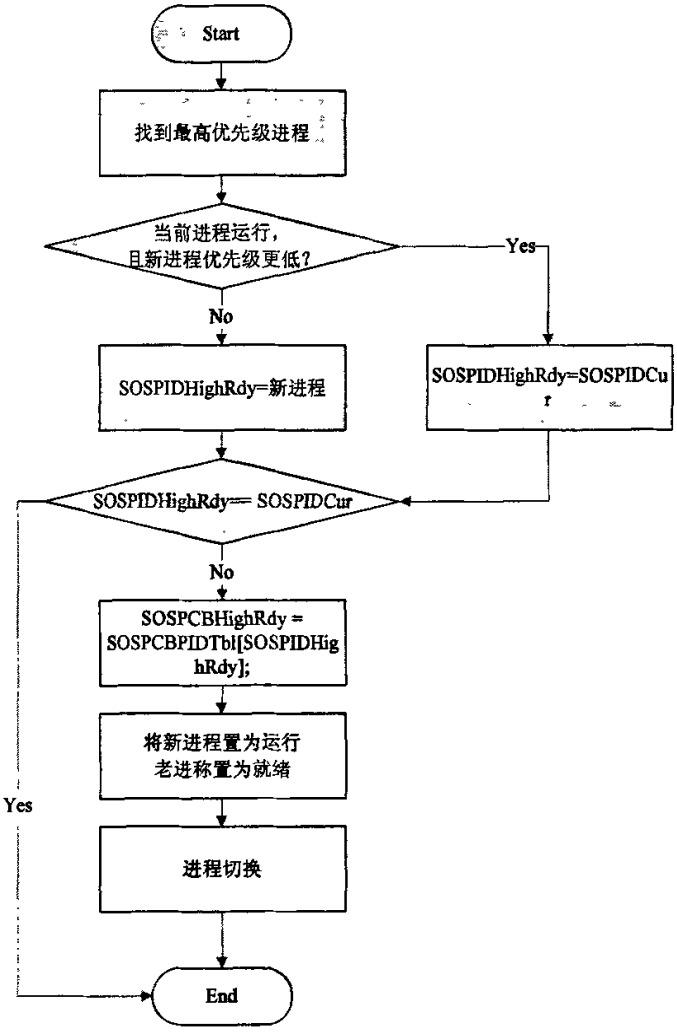


图 A4 进程调度程序流程图

调度上锁程序流程图如图 A5 所示。

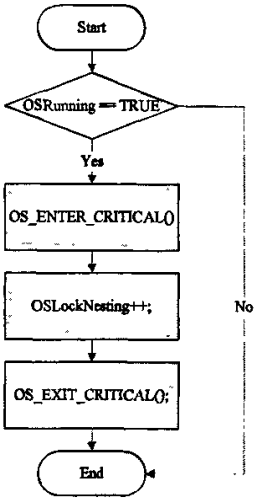


图 A5 调度上锁程序流程图

调度解锁程序流程图如图 A6 所示。

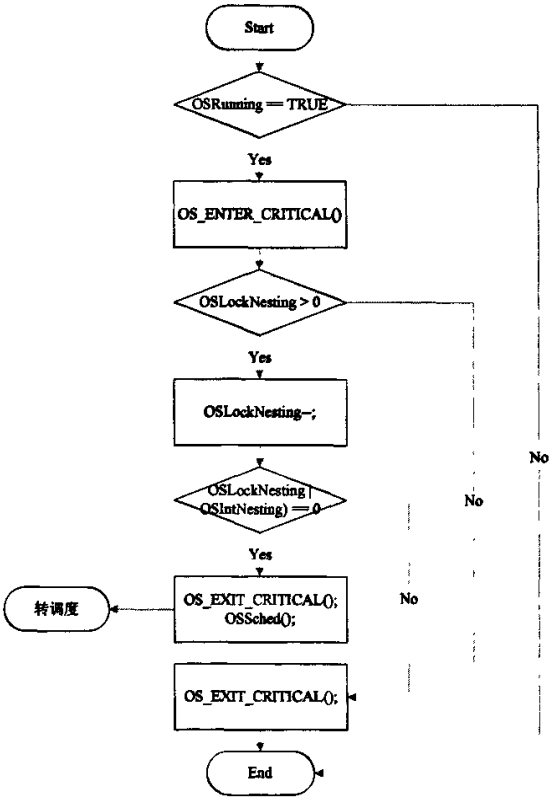


图 A6 调度解锁程序流程图

系统多任务启动程序流程图如图 A7 所示。

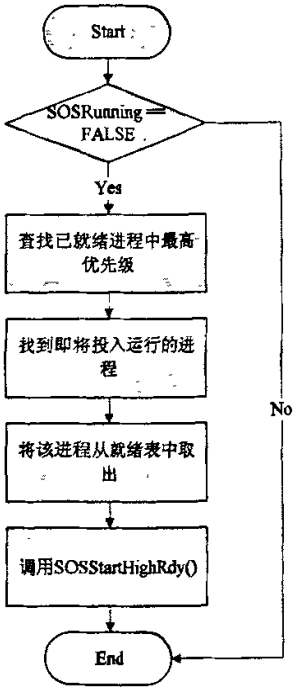


图 A7 系统多任务启动程序流程图

空闲进程程序流程图如图 A8 所示。

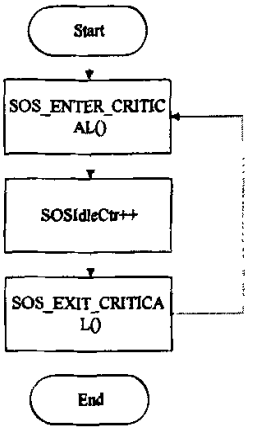


图 A8 空闲进程程序流程图

统计进程初始化流程图如图 A9 所示。

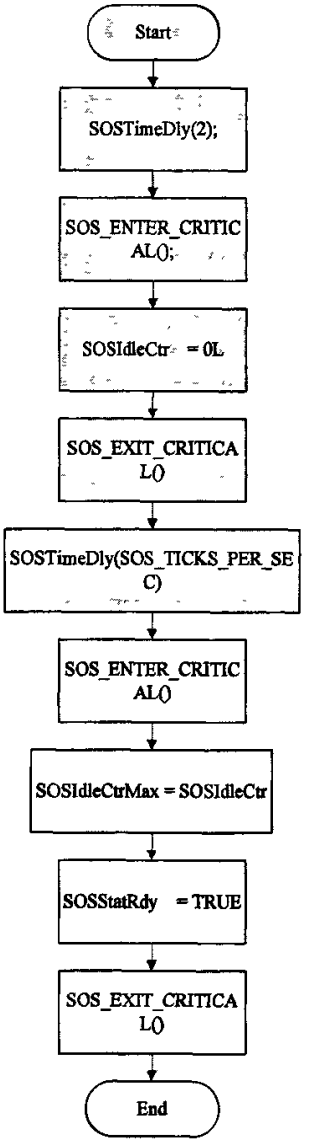


图 A9 统计进程初始化程序流程图

统计进程程序流程图如图 A10 所示。

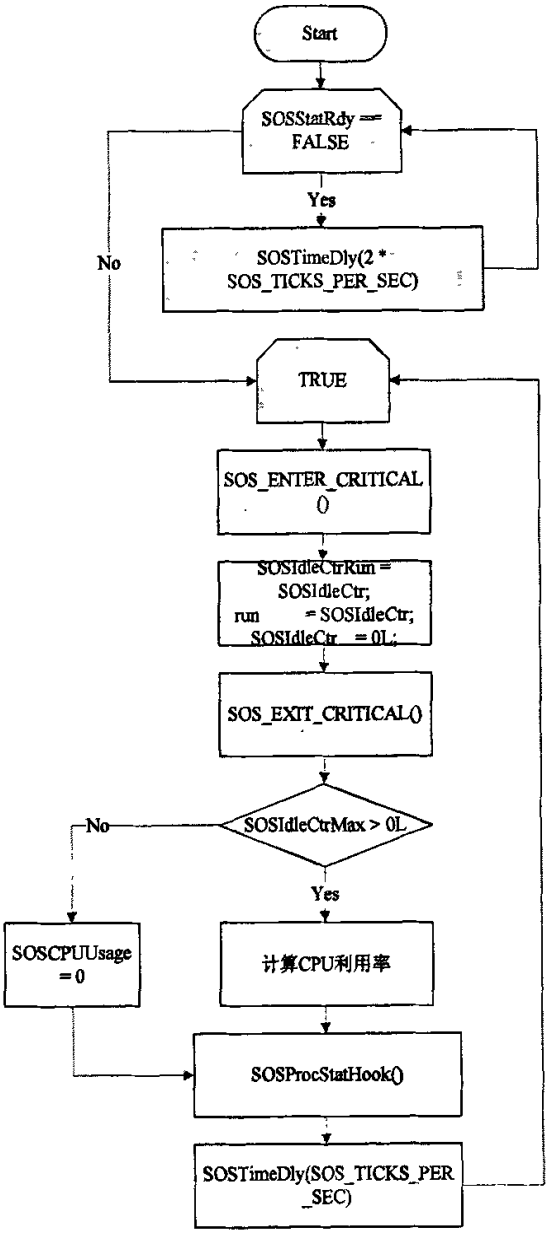


图 A10 统计进程程序流程图

进程控制块初始化程序流程图如图 A11 所示。

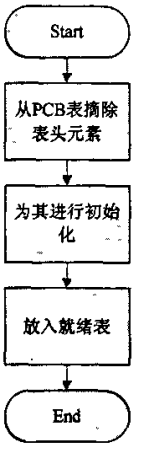


图 A11 进程控制块初始化程序流程图

任务堆栈初始化程序流程图如图 A12 所示。

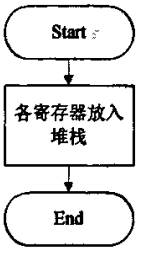


图 A12 任务堆栈初始化程序流程图

时钟节拍函数程序流程图如图 A13 所示。

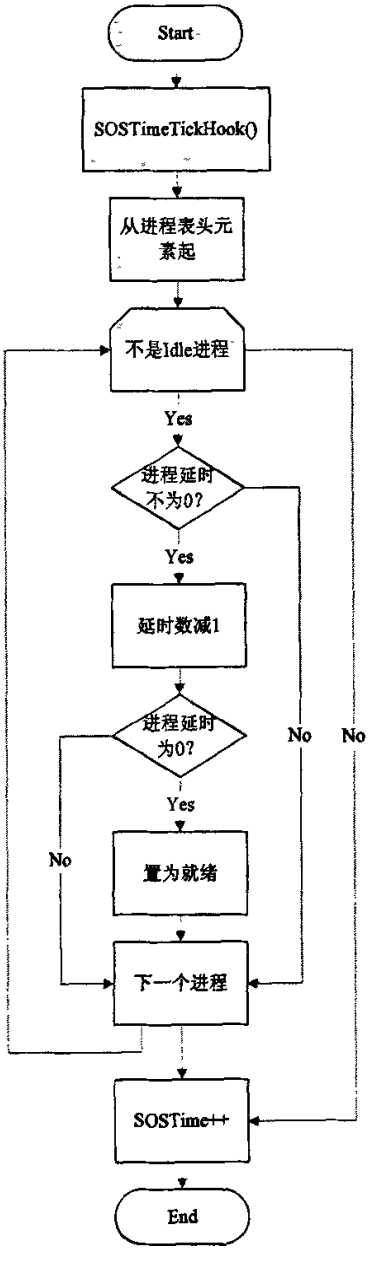


图 A13 时钟节拍函数程序流程图

提供系统版本信息程序流程图如图 A14 所示。

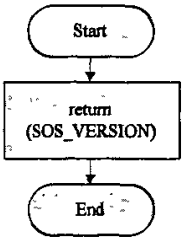


图 A14 提供系统版本信息程序流程图

进程管理

进程的创建程序流程图如图 A15 所示。

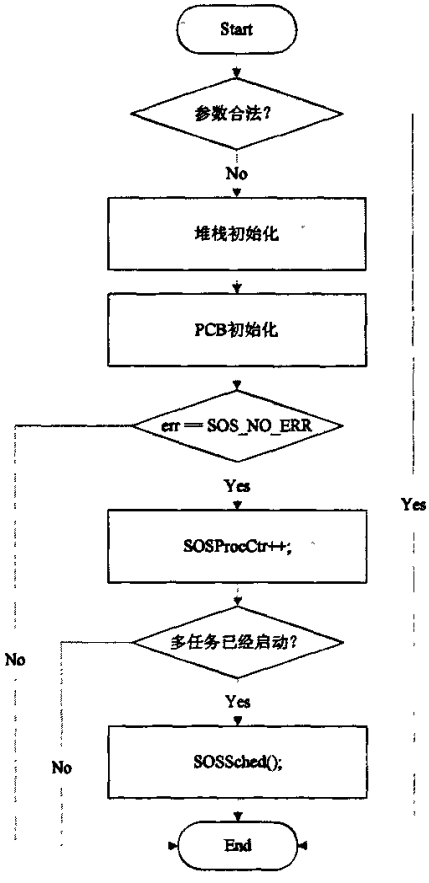


图 A15 进程的创建程序流程图

进程挂起程序流程图如图 A16 所示。

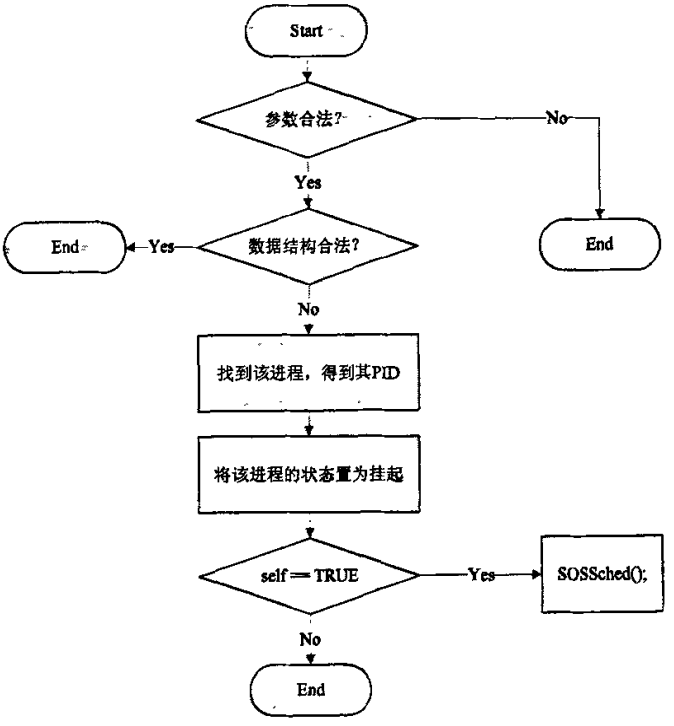


图 A16 进程挂起程序流程图

进程恢复程序流程图如图 A17 所示。

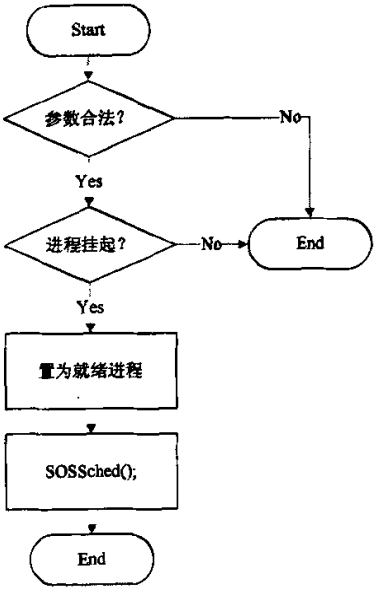


图 A17 进程恢复程序流程图

进程数据查询程序流程图如图 A18 所示。

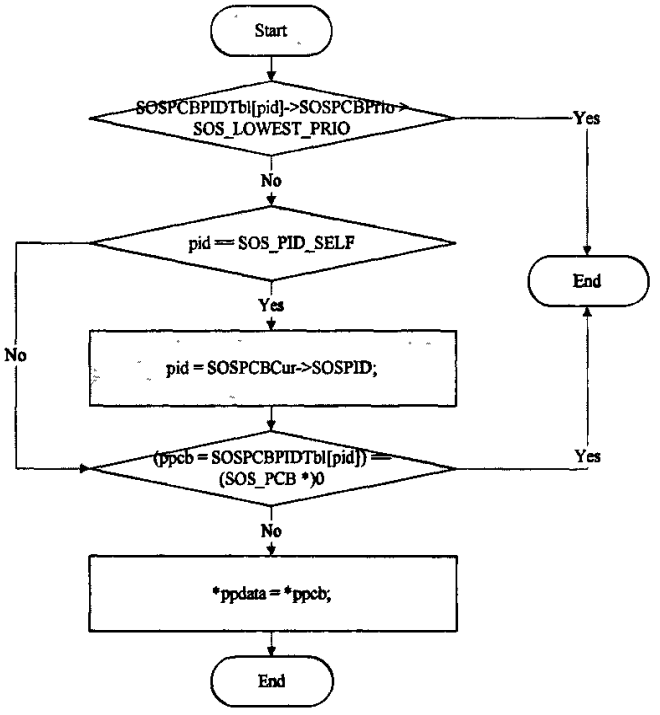


图 A18 进程数据查询程序流程图

时间管理

任务延时程序流程图如图 A19 所示。

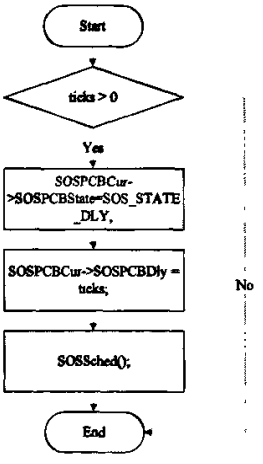


图 A19 任务延时程序流程图

恢复延时的任务程序流程图如图 A20 所示。

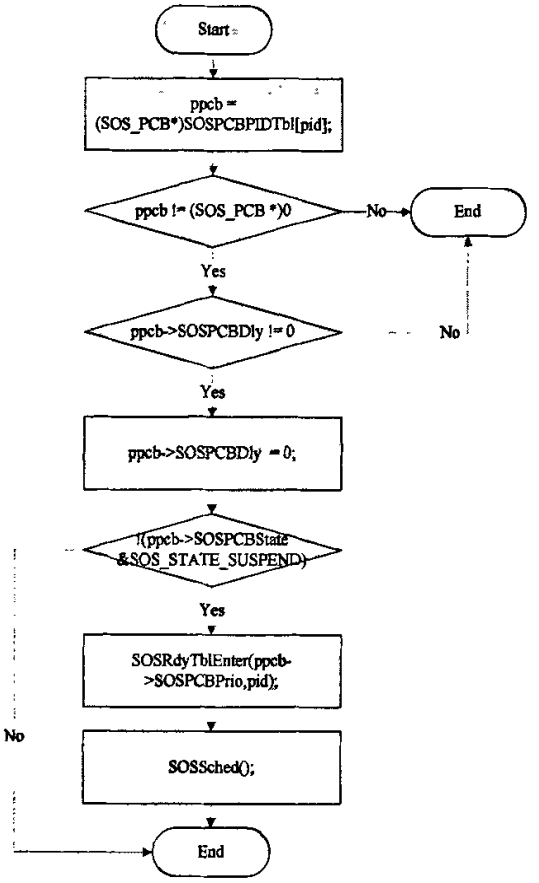


图 A20 恢复延时的任务程序流程图

获取系统时间程序流程图如图 A21 所示。

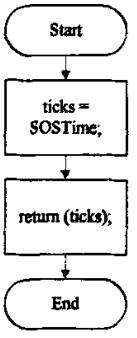


图 A21 获取系统时间程序流程图

设置系统时间程序流程图如图 A22 所示。

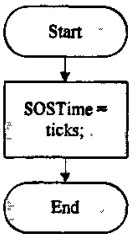


图 A22 设置系统时间程序流程图

信号量管理

创建信号量程序流程图如图 A23 所示。

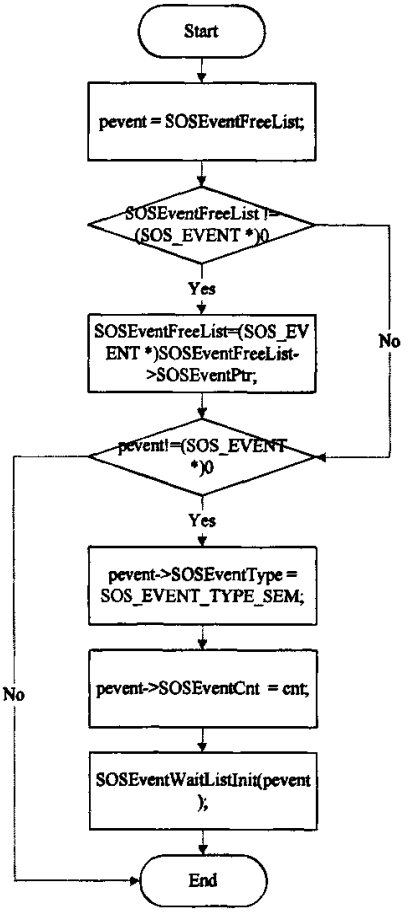


图 A23 创建信号量程序流程图

无等待请求信号量程序流程图如图 A24 所示。

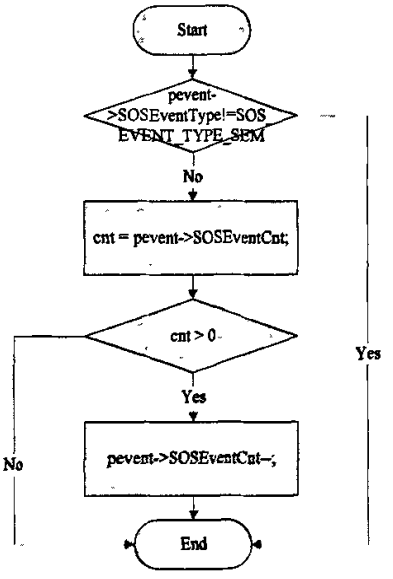


图 A24 无等待请求信号量程序流程图

等待信号量程序流程图如图 A25 所示。

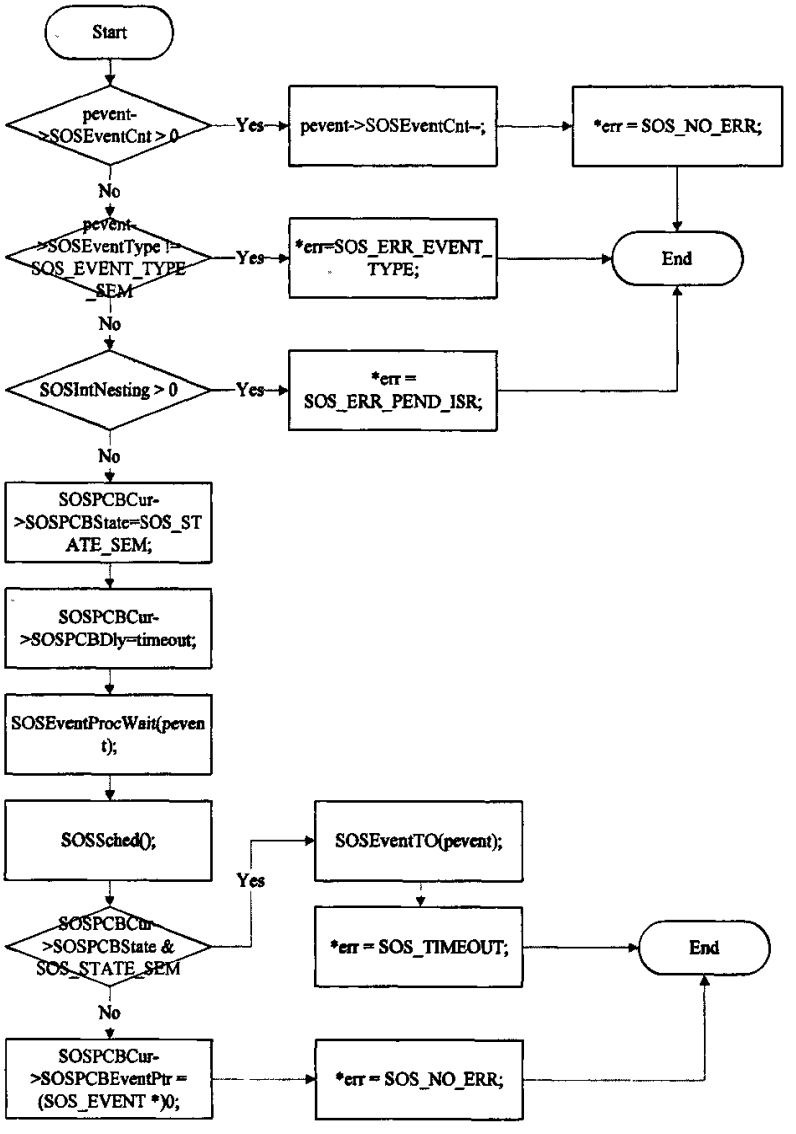


图 A25 等待信号量程序流程图

发出信号量程序流程图如图 A26 所示。

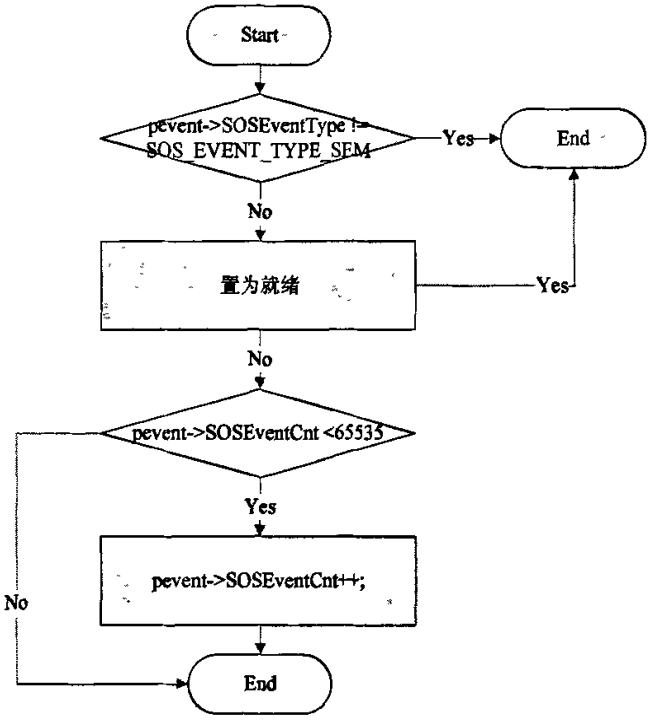


图 A26 发出信号量程序流程图

查询信号量程序流程图如图 A27 所示。

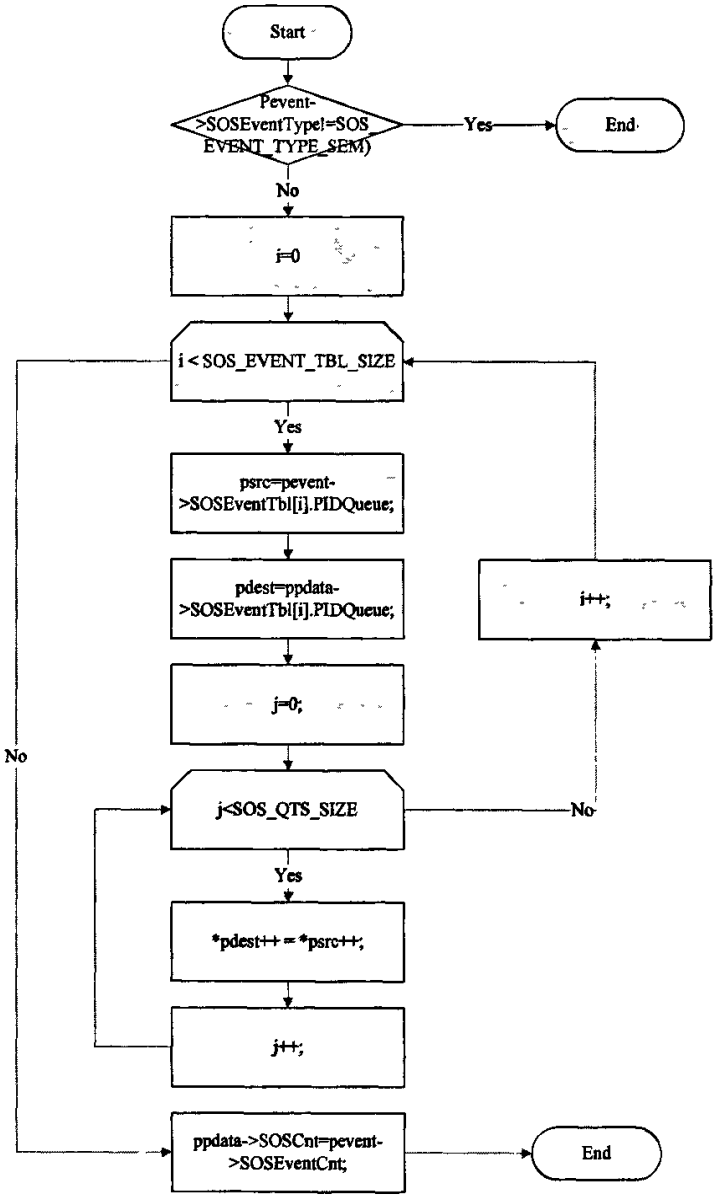


图 A27 查询信号量程序流程图

消息队列管理

初始化消息队列程序流程图如图 A28 所示。

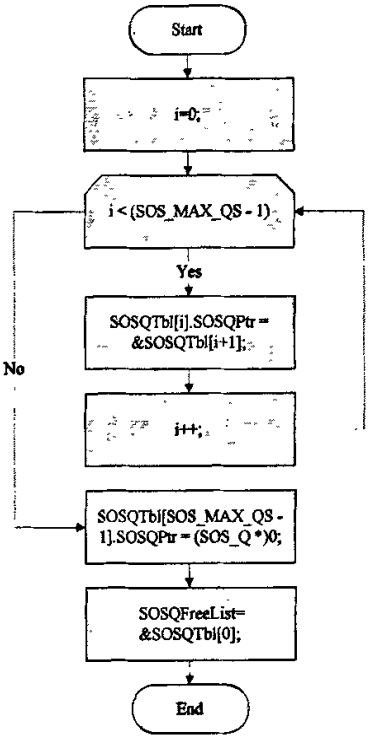


图 A28 初始化消息队列程序流程图

创建消息队列程序流程图如图 A29 所示。

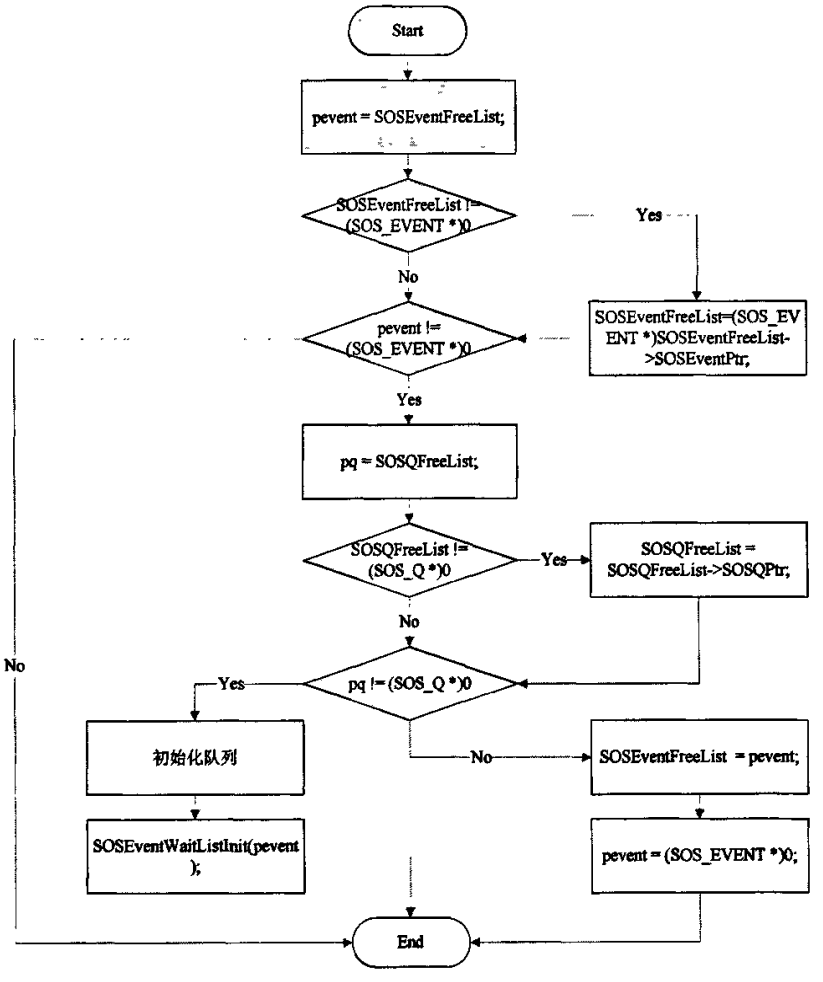


图 A29 创建消息队列程序流程图

等待消息队列程序流程图如图 A30 所示。

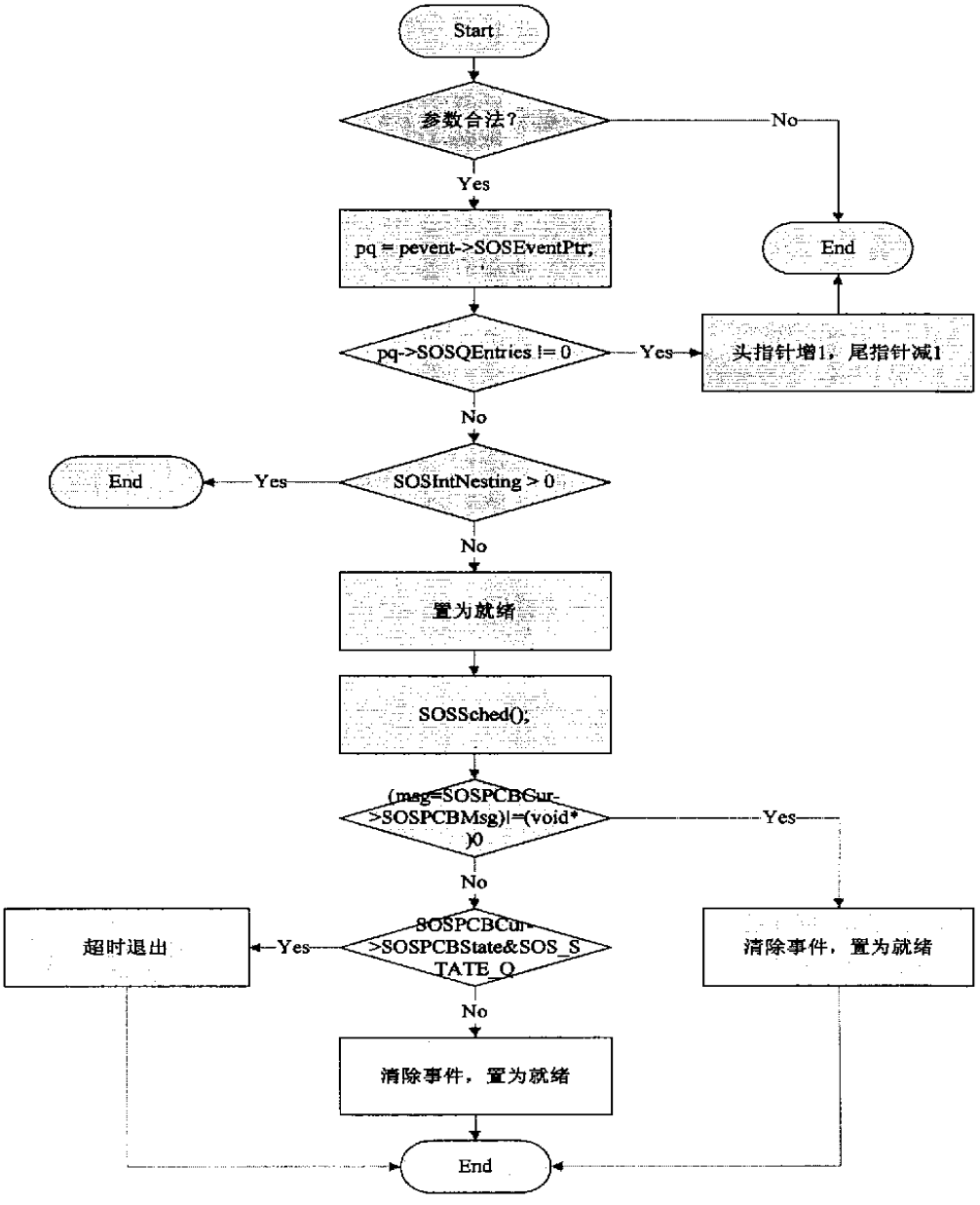


图 A30 等待消息队列程序流程图

发出消息队列程序流程图如图 A31 所示。

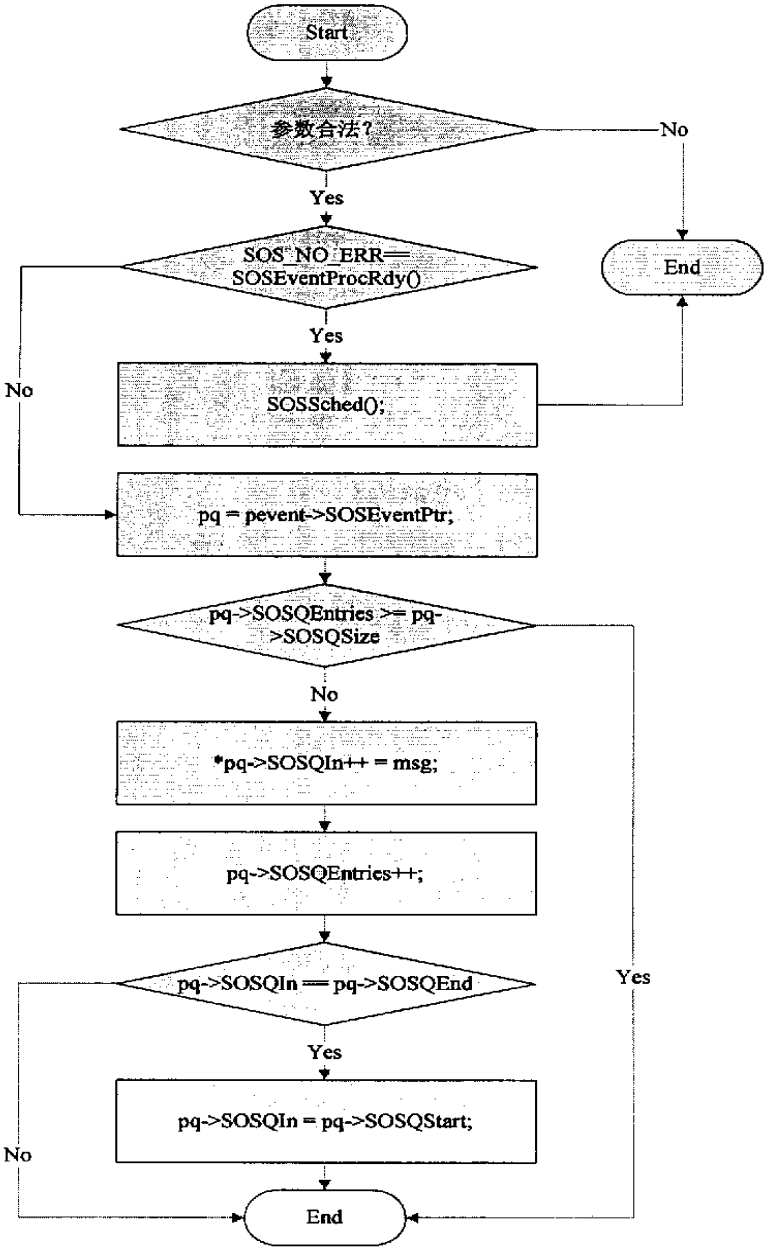


图 A31 发出消息队列程序流程图