江西师范大学

硕士学位论文

基于8051的嵌入式实时操作系统研究

姓名：甘智

申请学位级别：硕士

专业：计算机系统结构

指导教师：罗杰

20070501

摘　要

8051 微控制器是嵌入式控制领域的一个重要体系。它被广泛用在各种嵌入式系统中。传统上，在 8051 系列微处理器平台上开发的应用均是单任务执行，不能满足同时执行多个任务并要求较好的响应速度的场合。随着软硬件技术的不断进步，在 8051 系列单片机上实现多任务并行不仅是一种可能，而且在很多情况下成为需要。实现多任务的一种方法是在硬件系统和应用程序之间引入一个操作系统。通过操作系统实现多任务调度，使得运行的每个任务并行执行，从而获得良好的响应速度。

本文工作不同于以往的通用操作系统在 8051 上的移植，主要研究的是在保持操作系统通用性与可读性的条件下如何进一步提高 8051 系统的实时性。研究工作集中在两个方面，分别是对现有重入方案和中断系统实时性的研究与改进。

为提高重入函数的执行效率，本文提出了一种基于页的新型 8051 存储器模型。文章对 8051 页变量特征进行了分析，通过将这些特性与重入问题相联系，从理论角度说明了页函数支持多任务重入的原理。本文还结合开发工具对基于页的存储器模型的执行效率进行了定量分析，其结果显示基于页的存储器模型带宽是目前重入堆栈模型带宽的 3.75 倍。这表示新模型使多任务程序执行效率得到较大提升，接近传统单任务的紧凑模式的执行效率。我们在 8051 微控制器上实现了基于页的多任务模型。论文总结了实现过程中的关键技术，存在的问题及相应的对策。

在中断处理方面，本文从影响实时性较大的关中断问题和中断服务程序复杂度两个方向进行了实时性的研究。我们引用了中断延时队列，分离了对核心关键数据的写同步问题，从而避免了关中断操作。同时，为提高中断服务程序的可预测性，我们使用 delta 链对时钟节拍中的定时器管理进行了改进。论文结合 8051 程序执行特征，给出了对 delta 链方式复杂度的证明。证明显示，delta 链的时间复杂度并非在任何情况下都比目前的算法好，而且特殊情况下会更差。但是尽管如此，证明还显示 delta 链的大部分运算是在中断服务程序之外完成的。这使得中断服务程序时间复杂度保持为常数级 O(1)。因此改进后的中断服务程序是可预测的，提高了系统实时性。

本文最后还对进一步的研究方向进行了展望。

关键词：8051、嵌入式操作系统、实时性、重入、中断、可预测性。

Abstract

8051 family of microcontrollers is a leading choice for embedded control, which makes it widely used in the embedded system. Traditionally, the task running on the 8051 platform is single threaded, but this concept can't meet the needs of realtime in multi-tasks development. With the ever lasting development of both software and hardware technology, it becomes more than a possibility to apply realtime multi-tasks in the 8051 family. One simple solution is to employ an operating system to link the software applications and hardware together. It is the operating sytem that schedules the tasks, makes them running parallely, thus it speeds up the response.

To be different from all the transplanting researches of general purpose 8051 operating system, this paper mainly focuses on the improvement of realtime performance while keeping the operating system's generality and readability. Our study can be divided into two aspects, namely the research and improvement in current reentrance method and interrupt system.

To enhance 8051's reentrancy performance, here we present a new method, the page-based reentrant function. Based on the detailed examining of the PDATA's properties, we find the PDATA is a new theoretical solution for C51's reentrant function and can be employed in the multi-tasks. In this paper, we also make a quantitative analysis on the run-time performance of page-based reentrant functions. It shows that our model's bandwidth is 3.75 times of the reentrant stack's, thus the new method makes the executing performance nearly the same as classical compact mode. By taking full advantage of this theory, we carry out a page-based operating system which is suitable for time critical systems. This paper also summarizes the key techniques, main problems and solutions that we meet in the process of building up the page-based operating system.

We also improve the system's realtime performance by interrupt handling. On one hand, we use the interrupt-delay-queue to separate the simultaneous writing of key data, thus avoid the operation of disabling interrupt. On the other hand, we use the delta list to improve the tick management of soft timers, which enhances the predictability of ISR. In this paper, we give proof on the time complexity of the delta list with the running features of 8051 platform. The proof demonstrates that the total time complexity of delta list is not better than current algorithm, sometimes even worse. But due to the main load of operation is done outside TICK, it makes the rest load of soft timers management maintain a constant time complexity O(1) in the

TICK. Therefor, the enhanced ISR is predictable and grants better real time performance.

This paper gives an outlook to the future research in the end.

Key words: 8051, Embedded System, real time,reentrancy, interrupt, predictability

学位论文独创性声明

本人所呈交的学位论文是我在导师的指导下进行的研究工作及取得的研究成果。据我所知，除文中已经注明引用的内容外，本论文不包含其他个人已经发表或撰写过的研究成果。对本文的研究做出重要贡献的个人和集体，均已在文中作了明确说明并表示谢意。

学位论文作者签名：　　签字日期：　　年　　月　　日

学位论文版权使用授权书

本学位论文作者完全了解江西师范大学研究生学院有关保留、使用学位论文的规定。本人授权江西师范大学研究生学院可以保留并向国家有关部门或机构送交论文的复印件和磁盘，可以将学位论文的全部或部分内容编入有关数据库进行检索，可以采用复印、扫描等复制手段保存、汇编学位论文，允许论文被查阅和借阅。

学位论文作者签名：　　导师签名：

签字日期：　　年　　月　　日　　签字日期：　　年　　月　　日

第1章 绪 论

自 1946 年第一台电子计算机诞生以来，不断发展进步的计算机技术和电子技术极大地改变了人们的生活和工作方式。如果说人类用计算机创建了一个新的数字世界，那么嵌入式计算机就是人们进入这个世界的通行证、引路人和代理人。从数字机顶盒到汽车控制系统，从手机到智能家居，人类直接接触的大多是专用嵌入式系统。然而在人们的日常生活之外，嵌入式系统已经广泛应用于科学研究、工程制造和军事技术等各个领域。

1.1 研究背景

随着计算机技术和通信技术的快速发展以及互联网的广泛应用，嵌入式系统已成为计算机应用的新热点[1]。作为嵌入式系统应用产品的一个主要方面，嵌入式信息产品也得到广泛应用。信息家电、智能仪表和智能安保系统等产品已经越来越多的出现在人们的生活中。可以预见，由于满足了人们对舒适、便捷、安全生活环境的需求，嵌入式信息产品的设计、应用将得到快速发展。

8051 系列单片机作为嵌入式信息产品的一个核心部件，为适应这种发展的要求，其使用、设计发生了巨大的变化。在微电子工艺水平发展的推动下，8051 微控制器的处理能力不断提高。在运算速度上看，8051 从最初 12 振荡周期的 1MIPS 提高到了目前单振荡周期的 40MIPS[2]。从数据加工能力上看，部分 8051 微控制器中还集成了数字信号扩展(DSP)或数字协处理器。总体来说，8051 的整体性能提高了几十倍。这些都使得 8051 的功能越来越强大，执行速度越来越快，应用领域进一步得到拓宽。可以说，单片机芯片的硬件性能已经能够满足现代人们对嵌入式信息产品的更高要求，因此基于 8051 单片机的操作系统研究应运而生。嵌入式 RTOS(Real Time Operating System 实时操作系统)的软件设计方法的出现打破了以前的前后台(超循环)的单进程设计方法的限制，并越来越受到人们的重视和应用。

嵌入式操作系统的概念是 1970 年提出的。与实时系统设计一样，嵌入式操作系统设计并不是一个新的话题，也经历了漫长的发展过程。据嵌入式系统杂志(Embedded Systems Programming)的最新报道，世界各国已有 40 多家公司，成功地推出了 200 余种可供嵌入式应用的实时操作系统。其中几个著名的操作系统是:Microtec Research 公司(MRI)的 VRTX，Wind River 公司的 VxWorks 及 pSOS，这些操作系统适用于强实时、多任务应用环境，而且还具有相应的功能齐全的交叉开发环境。[刘波] 实时嵌入式操作系统的出现，为实时系统设计提供了一个有力的工具。最初的实时系统并不需要实时嵌入式操作系统的支持。在那时，由

于受内存大小以及处理器能力的限制，程序设计往往用汇编代码来实现，而对于多个任务的并发执行，则完全靠设计人员的手工安排来保证。随着芯片制造技术以及程序语言的发展，渐渐涌现出许多基于高级语言的编程技术以及基于高级语言的实时嵌入式操作系统。例如 C 语言编译效率的逐渐提高使得 C 语言编写程序能广泛应用于深度嵌入的场合。而内存制造技术的提高，使得嵌入式系统不再受限于几 K、几十 K 的小内存，而向几兆甚至几十兆的范围扩展，这也为嵌入式实时操作系统在实时场合的应用提供了基础。大量的实践经验也证明，实时嵌入式操作系统可以极大地简化系统设计过程，节约人力物力的投入，具有很高的应用价值。

从开发环境上看，最早为 8051 设计的 C51 编译器是在 1985 年出现的[42]。新版的 C51 编译器的编译效率可以达到 1.1。在单片机项目中应用 C51，更重要的是开发周期可以大大缩短。一般说来，一个熟练的 C51 程序员的开发速度是汇编程序员的两信以上，两者一的执行速度相差无几。如果不是有特别苛刻的要求，程序整体用 C51 开发要更快。

虽然人们较早就在 8051 上实现了 C51 开发，也针对 8051 微控制器进行了一些操作系统的研究，其中就包括实时操作系统 RTX51。但 RTX51 主要是使用汇编语言编写的，可读性差，导致了较大的修改和推广难度。直到实时操作系统 μC/OS 以及高性能 8051 微控制器被推出后，在 8051 上使用操作系统才得到长足的进步。这些不仅仅是因为 μC/OS 符合 ANSI-C 标准的，可读性较好，容易得到推广，还因为 μC/OS 是开源的，研究人员很容易免费获得该操作系统的全部源程序，并进行深入研究。

总之，制造工艺的提高、开发环境的改进和通用操作系统 μC/OS 的出现共同推进了 8051 系列微控制器开发平台化、系统化。由此可见，研究嵌入式实时操作系统是嵌入式开发的新需要，也是嵌入式系统开发的新阶段，具有广阔的应用前景。

1.2 嵌入式系统

一般地，嵌入式系统(Embedded System)是指嵌入于各种设备及应用产品内部的计算机及其应用系统。它最早出现在二十世纪 60 年代武器控制中，后来用于军事指挥控制和通信系统，现在广泛用于民用机电一体化产品中[3]。随着电子制造技术的飞速进步，人们可以把所有常用的资源，如存储器、模数转换器、通用输入输出口、定时器等，与 CPU 集成在一个芯片上，其体积小、功耗低、使用方便的特点日益凸显。因此，嵌入式系统可作为一个部件埋藏于所控制的装置中，它提供用户接口、管理有关信息的输入输出、监控设备工作，减少设备及应用系统的功耗和体积，从而使系统具有较高智能和性价比。因此当代嵌入式系统被普

遍定义为：以应用为核心，以计算机技术为基础，其软硬件可配置，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。

1.2.1 嵌入式系统的主要特点

嵌入式技术要求将计算机内装于专用设备或系统中，它的反应速度快，自动化程度高。大多数嵌入式系统是实时系统，该系统要求能在指定的时间内对各种事件，尤其是异步事件，做出正确响应。一般而言，嵌入式系统的特点可归纳为[3]：

①嵌入式系统通常是面向特定应用的。嵌入式系统的专用性很强，其中的软件系统和硬件的结合非常紧密，一般需要针对硬件进行系统的移植。同时针对不同的任务，往往需要对系统进行较大更改，程序的编译下载要和系统相结合，这种修改和通用软件的“升级”是完全不同的概念。

②系统精简。嵌入式系统一般没有系统软件和应用软件的明显区分，不要求其功能设计及实现上过于复杂，这样一方面利于控制系统成本，同时也利于实现系统安全。

③高实时性嵌入式操作系统。这是嵌入式软件的基本要求，而且软件要求固态存储，以提高速度。软件代码要求高质量和高可靠性、实时性。

④为了提高执行速度和系统可靠性，嵌入式系统中的软件一般都固化在存储器芯片或单片机本身中，而不是存贮于磁盘等载体中。

⑤嵌入式软件开发走向标准化。为了合理地调度多任务、利用系统资源、系统函数以及和专家库函数接口，用户必须自行选配 RTOS (Real Time Operating System)开发平台，这样才能保证程序执行的实时性、可靠性，并减少开发时间，保障软件质量。

⑥嵌入式系统本身不具备自举开发能力，即使设计完成以后用户通常也是不能对其中的程序功能进行修改的，必须有一套开发工具和环境才一能进行开发。开发时往往有主机和目标机的概念，主机用于程序的开发，目标机作为最后的执行机，开发时需要交替结合进行。

1.2.2 嵌入式系统的组成

作为计算机系统，嵌入式系统也分为软件部分和硬件部分两部分：硬件以芯片、模板、组件、控制器形式埋藏于设备内部。硬件架构上以嵌入式处理器为中心，配置存储器、I/O 设备、通信模块等必要的外设；嵌入式系统有别于一般的计算机处理系统，它不具备像硬盘那样大容量的存储介质，而大多使用 EPROM, EEPROM 或闪存(Flash Memory)作为存储介质。嵌入式软件通常是实时多任务操作系统和各种应用程序组成的专用软件，一般固化在 ROM 或闪存中。应用程序

控制着系统的运作和行为；而操作系统控制着应用程序编程与硬件的交互作用。向上提供应用编程接口(API)，向下屏蔽具体硬件特性的板级支持包 BSP。嵌入式系统中，软件和硬件紧密配合，协调工作，共同完成系统预定的功能[4]。

1.2.2.1 嵌入式硬件系统

1）嵌入式处理器的特点

在嵌入式系统的硬件设备中，嵌入式处理器芯片是整个系统的核心部件，其性能的好坏直接决定整个系统的运行效果。它通常具备以下特点：

①对实时多任务有很强的支持能力，能完成多任务并且有较短的中断响应时间，从而使内部的代码和实时内核的执行时间减少到最低限度；

②具有功能很强的存储区保护功能。这是由于嵌入式系统的软件结构已模块化，而为了避免在软件模块之间出现错误的交叉作用，需要设计强大的存储区保护功能，同时也有利于软件诊断；

③可扩展的处理器结构，以能最迅速地开展出满足应用的最高性能的嵌入式微处理器；

④嵌入式微处理器必须功耗很低，尤其是用于便携式的无线及移动的计算和通信设备中靠电池供电的嵌入式系统更是如此如需要功耗只有 mW 甚至 μW 级。

2）嵌入式处理器的分类[4]：

①嵌入式微控制器(MCU)

嵌入式微控制器又称单片机，顾名思义，就是将整个计算机系统集成到一块芯片中。微控制器的片上外设资源一般比较丰富，它常以某一种微处理器内核为核心，芯片内部集成 ROM/EPROM, RAM、总线、总线逻辑、定时/计数器、WatchDog、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。适合于控制，因此称微控制器。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[33]。微控制器是目前嵌入式系统工业的主流。目前 MCU 占嵌入式系统约 70%的市场份额。

嵌入式微控制器目前的品种和数量最多，比较有代表性的通用系列包括 8051，P51XA， MCS-251、MCS-96/196/296、C166/167、MC68HC05/11/12/16、68300 等。

②嵌入式 DSP 处理器(Digital Signal Processor)

DSP 处理器对系统结构和指令进行了特殊设计，使其适合于执行 DSP 算法，编译效率较高，指令执行速度也较高。在数字滤波、FFT、谱分析、图像处理等领域，DSP 正在大量进入嵌入式市场。推动嵌入式 DSP 处理器发展的另一个因

素是嵌入式系统的智能化，例如各种带有智能逻辑的消费类产品，生物信息识别终端，带有加解密算法的键盘，ADSL 接入、实时语音压解系统，虚拟现实显示等。这类智能化算法一般都是运算量较大，特别是向量运算、指针线性寻址等较多，而这些正是 DSP 处理器的长处所在。

嵌入式 DSP 处理器比较有代表性的产品是 Texas Instruments 的 TMS320 系列和 Motorola 的 DSP56000 系列。

③嵌入式微处理器(MPU)

嵌入式微处理器的基础是通用计算机中的 CPU。为了满足嵌入式应用的特殊要求，嵌入式微处理器虽然在功能上和标准微处理器基本是一样的，但在工作温度、抗电磁干扰、可靠性等方面一般都做了各种增强。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，但是在电路板上必须包括 ROM, RAM、总线接口、各种外设等器件，从而降低了系统的可靠性，技术保密性也较差。

目前市场最常见的 MPU 系列有 ARM, POWERPC, MIPS 等。

④SOC (System On A Chip)

为了寻求应用系统在片上的最大化解决。还出现了片上系统 SOC。SOC 把处理器和高密度逻辑电路，外围模拟/数字电路混合信号、存储器和通信模块等集成在一块芯片上。各种通用处理器内核将作为 SOC 设计公司的标准库，和许多其它嵌入式系统外设一样，成为 VLSI 设计中一种标准的器件，用标准的 VHDL 等语言描述，存储在器件库中。用户只需定义出整个应用系统，仿真通过后就可以将设计图交给半导体工厂制作样品。这样基本上用一块芯片就完成了早期嵌入式计算机系统的所有功能。应用系统电路板将变得很简洁，对于减小体积和功耗、提高可靠性非常有利。

SOC 可以分为通用和专用两类。通用系列包括 Infineon 的 TriCore, Motorola 的 M-Core, Echelon 和 Motorola 联合研制的 Neuron 芯片等。专用 SOC 一般专用于某个或某类系统中，不为一般用户所知。一个有代表性的产品是 Philips 的 SmartXA，它将 XA 单片机内核和支持超过 2048 位复杂 RSA 算法的 CCU 单元制作在一块硅片上，形成一个可加载 JAVA 或 C 语言的专用的 SOC，可用于公众互联网如 Internet 安全方面。

1.2.2.2 嵌入式软件系统

嵌入式软件系统主要是指控制和操作硬件实现系统功能的指令集合。它是实现嵌入式系统功能的关键。目前，在平台化的嵌入式系统开发中，嵌入式软件可

进一步划分为操作系统和应用程序。嵌入式开发对系统软件和应用软件的要求和通用计算机有所不同，主要有以下几点：

①软件要求固态化存储。为了提高执行速度和系统可靠性，嵌入式系统中的软件一般都固化在存储器芯片或嵌入式微控制器芯片本身中，而不是存贮于磁盘等载体中。

②软件代码要求高质量、高可靠性。尽管半导体技术的发展使处理器速度不断提高、片上存储器容量不断增加，但在大多数应用中受应用环境的限制，或者是出于对批量生产的成本控制，存储空间仍然是宝贵的，再加上实时性的要求。为此要求程序编写和编译质量要高，以减小程序二进制代码长度，提高执行速度。

③操作系统软件（OS）的高实时性是基本要求。在多任务嵌入式系统中，对重要性各不相同的任务进行统筹兼顾的合理调度是保证每个任务及时执行的关键，单纯通过提高处理器速度是无法完成或是盲目的，这种任务调度只能由优化编写的系统软件来完成，因此系统软件的高实时性也是基本要求。

④嵌入式系统软件需要实时多任务操作系统开发平台（RTOS）。通用计算机具有完善的操作系统和应用程序接口。应用程序的开发以及完成后的软件都在 OS 平台上运行，但一般不是实时的。嵌入式系统则不同，应用程序可以没有操作系统直接在芯片上运行;但是为了合理地调度多任务、利用系统资源，选择合适的 RTOS 进行嵌入式系统开发已成为普遍做法。RTOS 保证了程序执行的实时性、可靠性，并能减少开发时间，保障软件质量。

⑤嵌入式系统的自动化程度极高。一方面是因为嵌入式系统的人机交互界面比较简单，另一方面这也是应用的需要，这一点突出体现在军用及航天系统中。一方面因为是针对特定系统的开发，另一方面是因为实时性的要求，嵌入式软件中的应用程序和操作系统结合非常紧密，这就是通常所说的“深度嵌入”[5]。

1.2.3 嵌入式系统的应用

嵌入式系统几乎包括了生活中的所有电器设备：

军用：武器系统 航空航天器等

工业：工业控制设备 高级测试仪器医疗器械等

民用：视频会议 数码相机 打印机 汽车电子 路由器 交换机 手机 小灵通 信息电话智能家电 机顶盒 PDA VCD DVD 等

1.3 实时系统

1.3.1 实时系统的定义与分类

实时系统(Real Time System)是一个能够在指定或者确定的时间内完成系统功能及对外部或内部事件在同步或异步时间内做出响应的系统。这种系统的正确

性不仅依赖于计算的逻辑结果，而且还与计算结果产生的时间有关[6]。

C. M. Krishna 和 Kang G. Shin 在实时系统设计中对实时系统所作的一个定义为[5]："任何一个对外部激励信号需要有一个确定的时间响应的计算机系统就是一个实时系统。"在这里，一个确定的时间响应的意思就是，在一个实时系统中运行的任务必须有一个死限(Deadline)。

根据结果贬值程度的大小，实时系统通常划分为硬实时系统和软实时系统[5][7]。

硬实时系统：如果响应时间超过了系统设计时所设定的死限，那么会造成重大的财产损失。如用计算机来控制的飞机电子系统，如果一系列的响应时间超过了系统设计时所设定的死限，那么就有可能会造成机毁人亡的严重后果。

软实时系统：如果响应时间超过了系统设计时所设定的死限，并不会造成致命的后果。大部分常见的嵌入式系统便属于软实时系统，如一个媒体播放器，即使在观看过程中，真的出现了这种情况，也只是短暂的画面的停顿，并不会造成什么人生财产损失，可归为软实时系统。

针对不同的系统，任务死限的计算方式也不同。例如对于一些系统而言，其任务的死限是确定的。同样举汽车刹车控制系统的例子，刹车控制器必须在接到传感器信号后的几个毫秒之内作出具体的响应。而对于另外一些系统，如计算 Pi 值时，如果降低计算精度，那么任务的死限就可以变得小，而提高计算精度，那么任务的死限就需要放长。

由于本文应用背景的特殊性，在本文中所指的实时系统均指硬实时系统。

1.3.2 实时系统的特点

实时系统是在逻辑和时序控制中如果出现偏差将会引起严重后果的系统。对于实时系统来说，它应具备以下几个重要的特性：

实时性。实时系统所产生的结果在时间上有着严格的要求，只有符合时间要求的结果才认为是正确的。在实时系统中，每个任务都有一个截止期限，任务必须在这个截止期限之内完成，以此保证系统所产生的结果在时间上的正确性。

可靠性。可靠性一方面指系统的正确性，即系统所产生的结果在返回值和运行费时上都是正确的;另一方面它指系统的健壮性。也就是说，虽然系统出现了错误或外部环境与预先假定的外部环境不符合，但系统仍然可以处于可预测状态，它仍可以安全地带错运行和平缓地降级。

并行性。一般来说，一个实时系统常常有多个外部输入端口。因此，这就要求系统具有并行处理的能力，以便能同时响应来自不同端口的输入信号。

可预测性[7]。实时系统的实际行为必须处在一定的限度内，而这个限度可以由系统的定义而获得。这意味着系统对来自外部输入的反映必须全部是可预测

的，即使在最坏的条件下，系统也要严格遵守时间的约束。因此，在出现过载时，系统必须能以一种可预测的方式来降级它的性能。

1.3.3 嵌入式系统与实时系统

[陈晗斐]而且由于采用了 C 语言等高级语言的编程模式，使得一个可靠的实时嵌入式操作系统可以进一步提高实时系统的可靠性。在传统的实时系统设计中，完全靠设计人员手工实现这些功能，因此对于系统的升级、维护以及移植都造成了极大的不便。即使在有的实时系统设计中，设计者专门设计了操作系统，但这些操作系统一般都比较简单，和现有的实时嵌入式操作系统功能还是无法相比的。

从根本上来说，实时系统和通用系统之间的差别主要体现在以下方面。首先，实时系统在它们所处的领域更具有专业性，其次，对于故障所带来的影响，实时系统更具有危害性[5]。

一般来说一个实时系统总是为一个专门的应用而设计。这样做的好处是，应用环境的特点以及它的操作环境会比通用系统更精确。因此，系统设计者可以对实时系统作细致的调整，使得系统的运行达到最佳。

从嵌入式系统的发展历史和定义可知，嵌入式系统实质上就是实现某些特定要求的计算机应用系统。它的最大特点是其所具有的目的性或针对性，即每一套嵌入式系统的开发设计都有其特殊的应用场合与特定功能，这也是嵌入式系统与通用的计算机系统最主要的区别。另外，因为主要应用于各种信息管理与信号控制，嵌入式系统与实时性有着天然的联系。并且嵌入式系统是为特定的目的而设计的，常常受到空间、成本、存储、带宽等条件的严格限制，因此设计者也必须最大限度地在硬件上和软件上充分考虑实时性的要求。

1.4 嵌入式操作系统

嵌入式操作系统是嵌入式系统的软件核心。随着嵌入式处理器速度的不断提高，使用操作系统对系统进行资源管理逐渐成为可能甚至是必要。嵌入式系统的用户需求也变得越来越复杂，通常需要多任务并行处理才能完成。这要求开发人员将应用分解成若干独立的任务，通过操作系统来对任务进行管理，以提高系统的运行效率。嵌入式操作系统是实现嵌入式开发平台化最有力的工具[4]。

1.4.1 嵌入式操作系统的发展历史

[黄磊]嵌入式操作系统的概念出现在 1970 年左右。其发展如图 1-1 所示。

1）无操作系统阶段

这一阶段的嵌入式系统只执行一些单线程的程序，因此严格地说还谈不上系统的概念。其主要特点是：系统结构和功能相对单一，处理效率较低，存储容量

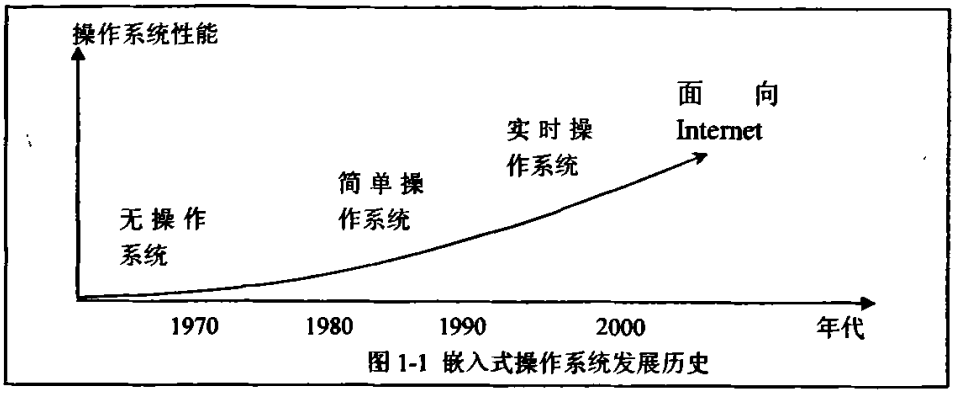

图 1-1 嵌入式操作系统发展历史

较小，几乎没有用户接口。由于这种嵌入式系统使用简便、价格低廉，因而曾经在工业控制领域中得到了非常广泛的应用，但却无法满足现今对执行效率、存储容量都有较高要求的信息家电等场合的需要。

2）简单操作系统阶段

20 世纪 80 年代，随着微电子工艺水平的提高，IC 制造商开始把嵌入式应用中所需要的微处理器、I/O 接口、串行接口以及 RAM、ROM 等部件统统集成到一片 VLSI 中，制造出面向 I/O 设计的微控制器，并一举成为嵌入式系统领域中异军突起的新秀。与此同时，嵌入式系统的程序员也开始基于一些简单的"操作系统"开发嵌入式应用软件，大大缩短了开发周期、提高了开发效率。

这一阶段嵌入式系统的主要特点是：出现了大量高可*、低功耗的嵌入式 CPU（如 Power PC 等），各种简单的嵌入式操作系统开始出现并得到迅速发展。此时的嵌入式操作系统虽然还比较简单，但已经初步具有了一定的兼容性和扩展性，内核精巧且效率高，主要用来控制系统负载以及监控应用程序的运行。

3）实时操作系统阶段

20 世纪 90 年代，在分布控制、柔性制造、数字化通信和信息家电等巨大需求的牵引下，嵌入式系统进一步飞速发展，而面向实时信号处理算法的 DSP 产品则向着高速度、高精度、低功耗的方向发展。随着硬件实时性要求的提高，嵌入式系统的软件规模也不断扩大，逐渐形成了实时多任务操作系统（RTOS），并开始成为嵌入式系统的主流。

这一阶段嵌入式系统的主要特点是：操作系统的实时性得到了很大改善，已经能够运行在各种不同类型的微处理器上，具有高度的模块化和扩展性。此时的嵌入式操作系统已经具备了文件和目录管理、设备管理、多任务、网络、图形用户界面（GUI）等功能，并提供了大量的应用程序接口（API），从而使得应用软件的开发变得更加简单。

4）面向 Internet 阶段

21 世纪无疑将是一个网络的时代，将嵌入式系统应用到各种网络环境中去的

呼声自然也越来越高。目前大多数嵌入式系统还孤立于Internet之外，随着Internet的进一步发展，以及Internet技术与信息家电、工业控制技术等的结合日益紧密，嵌入式设备与Internet的结合才是嵌入式技术的真正未来。

1.4.2 嵌入式实时操作系统的组成与特点

从功能模块上看，一个好的实时操作系统需要具备以下功能：

- ✓ 任务管理(多任务和基于优先级的任务调度)
- ✓ 具备消除优先级倒置的机制
- ✓ 任务间同步和通信(信号量和邮箱等)
- ✓ 实时时钟服务
- ✓ 中断管理服务
- ✓ 操作系统的行为是可知的和可预测的

从性能要求上看，嵌入式系统对操作系统的要求包括[8][9]：

①实时性　对外部事件作出反映的时间必须要快，在某些情况下需要是确定的、可重复出现的，不管当时系统状态如何，都是可预测的；

②有处理异步并发事件的能力　嵌入式实时处理系统处理的外部事件往往不是单一的，这些事件往往同时出现，而且发生的时刻也是随机的，实时软件应有能力对这类事件组进行有效的处理；

③快速启动、并有出错处理和自动复位功能　这一要求对机动性强、环境复杂的智能系统显得特别重要。

1.4.3 实时操作系统与通用操作系统的对比

实时操作系统(RTOS- Real Time Operation System)是指一个能够在指定的时间范围内完成特定的功能或者对外部的异步事件作出响应的操作系统。实时操作系统上的进程执行结果不仅依赖于逻辑判断和逻辑计算的正确性，而且还依赖于执行过程中所花费的时间的长短。即它的计算结果的正确性不仅与数值计算的正确性有关，还与系统计算给出结果的时间有关[10]。如果给出结果的时间超过了系统规定的时间，则计算结果的价值将出现贬值或无效，在有些情况下甚至是负效果。所以实时操作系统必须能够确保其进程对时间的要求，即必须确保在要求的时间内完成指定的任务。在相当一部分嵌入式系统中，系统的实时性对于系统的成败有着至关重要的作用。因此，在嵌入式操作系统中，大部分也是实时操作系统。[8]

从系统角度看，实时领域需要研究的内容是很广泛的。因为所有在计算机体系结构、容错计算以及操作系统中所出现的问题，在实时计算中同样的存在。而

且，在实时领域，这些问题变得更复杂，因为实时系统必须满足更多的限制条件[5]。

举任务调度的例子来说。在通用操作系统中，任务调度的目标是公平原则，也就是说计算机的资源必须在不同的用户之间尽可能地公平地分配，如 ROUND-ROBIN 调度就可以防止一个用户过多地占用计算机系统的资源。但公平调度在实时系统中并不是首要的。举例来说，在载人飞行舱控制系统中，象温度采集与控制这样的生命维持系统的监控任务就非常重要。为防止处理器被其他任务抢占，生命维持系统通常要被赋予更高的优先级，以保证该任务能获得充分的运行时间。

一般桌面系统使用的都是通用操作系统。[11]通用操作系统是由分时操作系统发展而来，大部分都支持多用户和多进程，负责管理众多的进程并为它们分配系统资源。分时操作系统的基本设计原则是：尽量缩短系统的平均响应时间并提高系统的吞吐率，在单位时间内为尽可能多的用户请求提供服务。分时操作系统注重平均表现性能，不注重个体表现性能。对于整个系统来说，注重所有任务的平均响应时间而不关心单个任务的响应时间，对于某个单个任务来说，注重每次执行的平均响应时间而不关心某次特定执行的响应时间。

实时操作系统所遵循的另一个重要的设计原则是始终保证系统行为的可预测性(predictability)。可预测性是指在系统运行的任何时刻，在任何情况下，实时操作系统的资源调配策略都能为争夺资源(包括 CPU、内存、网络带宽等)的多个实时任务合理地分配资源，使每个实时任务的实时性要求都能得到满足。与通用操作系统不同，实时操作系统注重的不是系统的平均表现，而是要求每个实时任务在最坏情况下都要满足其实时性要求，也就是说，实时操作系统注重的是个体表现，更准确地讲是个体最坏情况表现。

因为实时系统故障的后果比通用系统来得更为严重，因此实时系统需要被仔细地定义，而且需要更好地验证它们的执行效率。也就是需要采用专门的方法来对其执行进行评估。我们需要一些方法来评估程序的执行时间、模块化系统软件和硬件、分配任务到不同的处理器并保证它们死限、保证系统能够从一个部件的故障中快速地恢复等。以上的这些问题，或者其它更多的问题，需要在实时系统设计中被仔细地考虑。

总之，实时系统和通用系统之间的差别体现在以下方面。首先，实时系统在它们所处的领域更具有专业性，其次，对于故障所带来的影响，实时系统更具有危害性。

1.4.4 目前世界主流嵌入式操作系统

VxWorks

VxWorks 是目前嵌入式系统领域中使用最广泛、市场占有率最高的操作系统。它支持多种处理器，如 x86, i960, Sun Sparc, Motorola MC58xxx, MIPS, POWER PC 等等。大多数的 VxWorks API 是专有的。采用 GNU 的编译和调试器。VxWorks 的实时性和稳定性在嵌入式操作系统中是第一流的。已经十多年没有发现 Bug。被广泛的用于航空航天，武器，通讯等实时性稳定性要求高的领域。缺点是价格贵，一般一套正版的 VxWorks 需要几十万美金。另外它不提供内核源码。如果选用的芯片不支持 VxWorks，用户一般无能为力。

pSOS

ISI 公司已经被 WinRiver 公司兼并，现在 pSOS 属于 WindRiver 公司的产品。这个系统是一个模块化、高性能的实时操作系统，专为嵌入式微处理器设计，提供一个完全多任务环境，在定制的或是商业化的硬件上提供高性能和高可靠性。可以让开发者根据操作系统的功能和内存需求定制成每一个应用所需的系统。开发者可以利用它来实现从简单的单个独立设备到复杂的、网络化的多处理器系统。它提供的模块很多，如文件系统，网络系统，分布式管理等等。可以按照不同的需求裁减，编译。北京华为的路由器，UT 的小灵通手机里面选用的就是 pSOS。它的缺点也是价格贵，且内核不公开

QNX

QNX 是一个实时的、可扩充的操作系统，它部分遵循 POSIX 相关标准，如：POSIX. 1b 实时扩展。它提供了一个很小的微内核以及一些可选的配合进程。其内核仅提供 4 种服务:进程调度、进程间通信、底层网络通信和中断处理，其进程在独立的地址空间运行。所有其它 OS 服务，都实现为协作的用户进程，因此 QNX 内核非常小巧((QNX4. x 大约为 12Kb)而且运行速度极快。这个灵活的结构可以使用户根据实际的需求，将系统配置成微小的嵌入式操作系统或是包括几百个处理器的超级虚拟机操作系统。

μC/OS -II

μC/OS 是美国电子工程师 Jean J.Labrosse 编写的，它是一个源码公开、可移植、可固化、可裁剪、占先式、支持多任务的实时操作系统。经过了 8 年的发展，μC/OS 已经在许多行业得到成功的应用。它采用占先式调度策略，支持 64 个优先级任务，实时性很强。μC/OS 源码只有 3000 多行，全部公开，用户可以根据需求修改内核。

μC/OS 的内核极为精简稳定，裁减后的最小内核甚至只有 3K。是低端 8 位和 16 位的嵌入式芯片的绝佳选择。早期版本的 μC/OS 是免费的。现在也要收费但不高。μC/OS 已经成功被移植在了 8051 系统中。我们将在第二章对 μC/OS 进行更深入地阐述。

Linux

Linux 内核经过裁减编译可以精简到 300K。

优点:源码公开，可以根据需要修改。开发资料极为丰富，网上可以很容易找到现成的例子。支持的芯片类型最多。有免费版本。

缺点：因为是非占先式调度策略，所以实时性不好。但随着研究的深入已出现实时内核的 Linux 操作系统。内核过于复杂，对于 8 位和 16 位芯片来说仍然偏大，一般用于 32 位芯片的场合。

Windows CE

当嵌入式系统应用的竞争如火如荼地展开时，微软 Windows CE 的推出标志着竞争达到一个尖峰。这也客观地说明了嵌入式系统市场的巨大潜力。

微软推出的 Windows CE 是为嵌入式系统设计的多线程、完整优先权、多任务的操作系统。基本内核占用 200K 以上 ROM。它提供强大的 API 函数调用，特别是有丰富的 GUI 接口。在上可以方便地开发出漂亮地界面。因此被广泛的用于个人消费品如掌上电脑，PDA 中。缺点是实时性差，任务切换往往反应很慢。

它不是一个实时操作系统，不能用在实时性高的场合。价钱也较贵。

1.5 论文的主要内容和结构安排

嵌入式实时操作系统由来已久。但是直到近几年才开始获得重视并在各行各业中获得越来越多的应用。尤其是在嵌入式应用场合，基于 c 语言的实时嵌入式操作系统已经逐渐替代了原来的利用汇编语言加上任务顺序执行的传统编程方法[5]。

嵌入式实时系统被定义为以应用为核心，提高系统实时性满足应用需要是系统设计的首要任务。从上文实时系统的描述中我们可以看到，大部分嵌入式系统都是实时系统，因此提高系统实时性是本文的研究重点。

我们的操作系统应用环境被定义在深度嵌入的实时系统中，因此对通用操作系统进行实时性能改造涉及的范围较广，包括算法性能和存储器等底层硬件性能评估。本文不是介绍通用操作系统的移植，而是在保留实时操作系统 μC/OS-II 的可读性和通用性的条件下，进一步发挥 8051 系统的结构特点，提高当前 8051 系统在多任务开发中的实时性。

关键研究内容包括两方面：

.1）8051 多任务实现技术。当前包括 C51 开发手册和移植实时操作系统的大量文献显示为实现多任务日使用的重入函数效率非常低。因此为提高实时性，8051 多任务及重入技术是我们的一个重要研究目标。

.2）实时性与中断快速响应。在实时系统研究中，中断是事件提交的重要方

式。中断服务程序具有特殊的优先权，其质量与实时系统实时性密切相关。因此，研究嵌入式实时操作系统就不能不重新评价中断及中断服务程序。一般的操作系统以关中断方式实现对核心数据的操作。这将不可避免地造成一些中断信号的挂起甚至是丢失。因此提高中断的响应速度，减轻关中断运行对中断响应的影响，是本系统一项关键技术。

本文第一章介绍背景，理清嵌入式系统与实时系统之间的相互关系。介绍了各种评价系统的性能指标。

第二章具体介绍在 8051 实现多任务技术的原理及概念。在多任务条件下，重新评价了开发平台中软硬件的特点及技术限制。

第三章详细介绍了在目前的开发平台上，如何采用基于页的 8051 多任务模型提高系统性能。以及如何使用中断延时队列和 delta 队列对通用操作系统中的中断处理进行改进的介绍。这一章，我们还结合 8051 的执行效率，特别对 delta 队列的算法时间复杂度进行了证明。

第四章为总结与展望。在总结全文的基础上，我们展望了在 8051 上引入分页技术的前景。

第2章　基于 8051 的嵌入式操作系统开发要素

这一章从操作系统的基本功能、处理器特性和开发工具三个方面介绍嵌入式操作系统。最后通过对现有能在 8051 上运行的操作系统的分析，给出了基于 8051 操作系统开发的关键技术。

嵌入式实时操作系统，既是嵌入式操作系统，又是实时操作系统。作为一种嵌入式操作系统，它具有嵌入式软件共有的可裁剪、低资源占用、低功耗等特点；而作为一种实时操作系统，它与通用操作系统(如 windows, Unix, Linux 等)相比有很大的差别，它除了要满足应用的功能需求以外，更重要的是还要满足应用提出的实时性要求，而实时性和多任务能力在很大程度上取决于它的任务调度机制[1]。因为嵌入式系统只运行相对有限的程序，并且这些程序是在设计时就已知的，这使得在通用系统中难以实现的系统优化在嵌入式系统中能够实现[12]。

实时多任务系统是靠实时操作系统 RTOS 来管理的。应用程序由多个任务组成，这些任务相对独立又相互有关。Linux, Unix 是分时的，有人把它们变成实时的，如实时 μCLinux, Lynx 等，通常的做法是让事件中断的优先级由用户设置，直到高于定时器的优先级。这种系统性能肯定很不错，因为这类操作系统集中了计算机专家们上千人的劳动与智慧。问题是系统太大，内存的开销至少也有几百千字节，无法嵌入到一般的嵌入式应用类产品中去，于是着手对这样的系统进行裁剪。首先，在嵌入式应用中不需要文件系统(即外存管理)，内存管理一般也可以省去，只剩下进程管理和 I/O 设备管理。进程管理在嵌入式系统中就是任务管理。每个 I/O 设备的管理也可以当若干个任务来管理。计算机的操作系统经过这样精简，剩下了面向嵌入式应用的部分，实时性变得更好，这部分通常称作实时内核。在嵌入式应用中，先在嵌入的计算机中运行一个实时内核，然后让这个实时内核管理应用程序中的各个任务，就形成了嵌入式实时多任务系统。从这种意义上说，实时内核就是嵌入式应用中所谓的实时操作系统。

2.1 基本概念

目前操作系统还没有一个精确的定义[12]，操作系统的出现只是为了能够对系统内运行的程序进行跟踪，屏蔽系统底层以实现降低系统编程复杂度。

和普通操作系统一样，嵌入式操作系统也能够从两个不同角度进行理解：机器扩展的角度和资源管理的角度。嵌入式系统也是一个由处理器、存储器、定时器等许多设备组成的一个复杂系统。因此，操作系统的主要工作就是为大量竞争资源的程序合理分配处理器、存储器等各种系统资源，也就是资源调度。因为处理器为整个系统的核心，而且为存储器等其他资源的直接使用者，因此一般情况

下，调度就是指处理器调度。

2.1.1 任务管理

多任务系统中，实时内核负责管理各个任务，或者说为每个任务分配 CPU 时间，并且负责任务之间的通信。下面就实时内核的任务调度策略和任务通信机制加以论述之前，先交代实时操作系统中的几个重要概念。

任务(Task)

指拥有所有 CPU 资源的简单程序。在进行实时应用设计时通常要把工作分割成多个任务，每个任务处理一部分问题，并被赋予一定的优先级、一套自己的 CPU 寄存器及堆栈。实时系统中的大部分任务是周期的，体现在编程上每个任务则是一个典型的无限循环。RTOS 中的任务可以处于下面几种状态的任何一种状态:休眠态、就绪态、运行态、挂起态(延迟或等待事件)及中断态。不同操作系统可能会定义不同的运行状态，但大体相同。

从这段描述可以看出，嵌入式操作系统中分配资源的单位是任务，而不是进程。这主要是因为嵌入式系统是以应用为中心的。换句话说，和进程相比较，这里所定义的任务在资源分配和资源私有性的角度看是一样的，但从并行角度看，进程颗粒度要小于任务的颗粒度，进程具有更好的并行性。在嵌入式开发中，考虑到嵌入式系统的运算能力比桌面系统要差，程序并行需要使用较大的颗粒度，所以嵌入式系统中任务的概念与进程基本相同。

上下文切换(Context Switch)

当多任务内核调度一个任务时，它将当前运行任务上下文(CPU 寄存器)保存在当前任务上下文存储区，并将即将运行的新任务的上下文从该任务的存储区中恢复过来，继续执行新任务的代码。上下文切换给系统增加了额外开销，CPU 的寄存器越多，开支越大。实时操作系统的性能不应该以内核每秒钟可以处理多少次上下文切换次数来判断，因为每个上下文切换时间由寄存器的数目决定。

调度(Schedule)

调度是内核的主要职责之一，决定任务运行的次序[29]。CPU 调度的基本方式有抢占方式和非抢占方式。基本的调度算法有先来先服务 FCFS，最短周期优先 SBF，优先级法(Priority)，轮转法(Round-Robin)，多级队列法(mufti-level Queues)，多级反馈队列(mufti-level Feedback Queues)等。多数实时内核是基于优先级调度的多种方法的复合。

内核(Kernel)

多任务系统的一部分，负责管理任务，或者说为每个任务分配 CPU 时间，并且负责任务间通信。内核提供的基本服务是上下文切换(context switch)。按内核的可抢占性可分为非抢占内核(non-preemptive)和抢占内核(Preemptive)。非抢

占式内核要求每个任务自动放弃 CPU 的所有权，任务一旦执行，其他任务则不可插入，直至该任务完成或主动交出 CPU;在剥夺式内核中，高优先级的任务一旦就绪，总能够迅速获得 CPU 的控制权。抢占式内核比非抢占式内核复杂得多，实时内核多数是抢占式内核。还可以按内核结构分为微内核(micro kernel)和单内核(monolithic kernel)两种。微内核更具模块性与移植性，但系统额外开销大，是现代实时操作系统的发展方向，但目前的操作系统多为两者的折衷。

优先级(Priority)

每个任务按其重要性被赋予一定的优先级。如果各任务的优先级在程序编译时是已知的，应用程序执行过程中各任务的优先级不变，则称之为静态优先级；如果在应用程序的执行过程中，任务的优先级是可变的，则称之为动态优先级。

基于优先级的系统会出现优先级倒置的问题，解决的方法有优先级继承法(priority inheritance)及优先级天花板协议(priority ceiling protocol)。已开发出多种算法用于实时任务的优先级分配，基本的有单调执行率调度法 RMS 和最早期限优先法 EDF 等。

进程通信(Interprocess Communication)

在多任务系统中，任务之间存在相互制约的关系，或者任务之间需要交换信息，在本文也称其为任务通信。任务之间相互制约的关系主要有互斥关系和同步关系，解决互斥关系主要是临界段的实现问题。任务间通信的方法有信号量、邮箱、事件标志、队列、共享内存等。

可预测性(Predictability)

指在系统运行的任何时刻、任何情况下，实时操作系统的资源调配策略都能为争夺资源(包括 CPU、内存、网络带宽等)的多个实时任务合理地分配资源，使各实时任务的实时性要求都能得到满足。可预测性在实时系统中是一个定性的概念，影响操作系统可预测性的因素是多方面的。传统上，嵌入式实时系统一直使用静态技术以保证可预测性，其中包括静态任务调度技术与静态存储器管理技术[32]。

2.1.2 中断处理

计算机系统接受事件有两种：查询和中断。在多任务操作系统中，由于采用查询方式处理事件或 I/O 请求会消耗大量的系统资源—— CPU 时间。因此，在一般的操作系统或嵌入式操作系统中，采用中断来处理事件或 I/O 请求是计算机系统接受事件最常见的方式。

与中断有关的三个概念是中断响应、时钟节拍和临界代码。

中断响应

中断响应就是指计算机系统开始执行与该中断相关的程序，处理引起中断的

事件。中断响应时间就是指从中断发生到计算机响应改该中断的时间间隔。中断响应速度与实时性密切相关，因此实时系统中中断研究的核心是提高中断响应速度。

在操作系统中，中断是同中断处理程序联系在一起的。以 I/O 操作为例，举例来说：任务 A 发出 I/O 请求后被挂起一操作系统切换到其它任务运行、I/O 设备完成相应的操作并发出中断谕求——操作系统调用相应的中断处理程序解挂任务 A。在嵌入式操作系统中，对中断处理十分重视，可以说多数嵌入式操作系统都是事件驱动的[13]。

在中断处理上一般的操作系统与嵌入式操作系统一个不同之处是现场保护。一般的操作系统的中断现场保护是由操作系统来完成的，在中断处理完成之后，也由操作系统恢复现场。在嵌入式操作系统中，由于受到代码量的限制以及实时性的要求，中断现场的保护往往由中断处理程序来完成。在中断处理程序的入口要保护在中断处理程序中用到的寄存器，在中断处理完成后恢复。中断服务程序(ISR)一般要用到汇编语言，以提高寄存器操作或堆栈操作的效率及中断响应时间，另一方面减少了代码量。但是这样做却损失了系统的安全性，同时也增加了调试的难度。这是在嵌入式操作系统的设计中应该予以关注的问题。另外为了提高系统的实时响应性能，ISR 设计要尽量精简。在中断处理程序(ISR)中常常仅执行一些必要的状态转换，而将主要事务处理放到任务中完成，关中断的时间也要尽量的短，最大限度地降低中断响应时间[14]。

时钟节拍（Time tick）

时钟节拍是一种周期产生的特殊系统中断。时钟节拍是实现抢占性内核的重要技术手段之一。它通过定时发出的中断节拍，中断正在执行的任务，从而使已就绪的高优先级任务能及时得到调度。

时钟节拍可视为系统心脏的跳动。中断周期越短，系统响应速度越快，但开销也越大，并且程序的执行速度越慢。典型中断时间在 10-200ms 之间。

临界代码(Critical Section)

临界代码指一段不可分割的代码，一旦执行，不能被中断。实现代码临界区的常见方法有两种。方法一，屏蔽中断。通常在代码执行前关闭中断，执行后打开中断，只能用于单处理机的情形。方法二是通过使用信号进行临界资源的控制。

显然屏蔽中断的方法程序结构清晰，其实现比方法二简单。但是方法一关闭中断显然会削弱系统实时性。而方法二的实质就是在保证信号操作的原子性的前提下，通过合理编写程序可以在不关闭中断的情况下操作临界资源。

2.2 8051 控制器概述

8051 是美国 Intel 公司的八位高档单片机系列，是在 MCS-48 系列的基础上

发展而成的，也是我国目前应用最广的一种单片机系列。

2.2.1 80C51 系列 MCU 概述[1]

在嵌入式系统低端的单片机领域，从 8 位单片机诞生至今，在百花齐放的单片机家族中，80C51 系列一直扮演着一个独特的角色。自 Intel 公司推出了 8051 系列单片机，不久之后便实施了最彻底的技术开放政策。在众多电器商、半导体商的积极参与下，将 8051 发展成了众多型号系列的 80C51 MCU 家族。8051 经典的体系结构、极好的兼容性和 Intel 公司的开放政策使众多厂家参与发展。

8051 的极佳兼容性，使其在被改造后，还能以不变的指令系统、基本单元的兼容性保持着 80C51 内核的生命延续，并在未来 SOC (System On Chip)发展中，担任 8 位 CPU 内核的重任。

8051 的优点归纳如下：低成本、性价比高、可扩展性好；内核技术成熟，各种 MCU 兼容性强；应用背景好、有强大的技术支持和服务；开发工具齐全、开发周期短；发展前景广阔，有很强的生命力。

(1)嵌入式应用中的 8 位机现象

与从 8 位机迅速向 16 位、32 位、64 位过渡的通用计算机相比，8 位单片机从 20 世纪 70 年代初期诞生至今，虽历经从单片微型计算机到微控制器、MCU 和 SOC 的变迁，但 8 位机始终是嵌入式系统低端应用的主要机型，而且在未来相当长的时间里，仍会保持这个势头。这是因为嵌入式系统与计算机系统有完全不同的应用特性，从而走向完全不同的技术发展道路。

嵌入式系统嵌入到对象体系中，并在对象环境下运行。与对象领域相关的操作主要是对外界物理参数进行采集、处理，对外界对象实现控制，并与操作者进行人机交互等。而对象领域中的物理参数的采集与处理、外部对象的控制以及人机交互所要求的响应速度有限。在 8 位单片机能基本满足其响应速度要求后，数据宽度不是技术发展的主要矛盾。因此 8 位单片机会稳定下来，其技术发展方向转为最大限度地满足对象的采集、控制、可靠性和低功耗等品质要求。

随着现代通信技术的发展，智能化系统对 DSP 需求的增长要求单片机相应提高运算速度。当前 8 位单片机在不扩展数据总线的情况下，提高运行速度仍有潜力可挖。例如，采用 RISC(精简指令集计算机)结构实现并行流水线作业，CISC(复杂指令集计算机)结构的 C8051F 采用 CIP-8051 结构，使单周期指令速度提高到原 80C51 的 12 倍。

鉴于嵌入式低端应用对象的有限响应要求、嵌入式系统低端应用的巨大市场以及 8 位机具有的速度潜力，可以预期在未来相当长的时间内，8 位机仍然是嵌入式应用中的主流机型。

随着半导体技术的发展，8 位单片机在 CPU 结构、CPU 外围、功能外围、

外围接口和集成开发环境方面都会迅速地发展。因此，可以说 8 位单片机虽然"古老"，但又会是一个十分活跃而新兴的嵌入式领域。80C51 系列从 Intel 公司的 8051 发展到今天百花齐放的 80C51 系列 MCU 的过程充分地说明了这一点。

(2) 8051 时序

研究 8051 系列微控制器的程序执行效率时，必须注意到该系列微控制器所使用的振荡器频率。因为大部分 8051 的设计方法是全静态设计，所以该系列的微控制器所能使用的振荡器范围非常广。一般来说，8051 系列微控制器工作的频率范围是 0~40Mhz。但是，在很多情况下，设计者并不会使用最高的工作频率，而是使用一个恰当的工作频率。比如说，为减少系统对外界的电磁干扰，就需要使用较低的工作频率。

因此，在评价 8051 微控制器的程序效率时，我们需要注意以下三个概念：振荡周期、机器周期、指令周期。振荡周期，即时钟周期，就是微控制器振荡电路振荡一次的时间，是 8051 内核工作的基本节拍，也是系统时序中最小的时间单位。机器周期被定义为实现特定功能所需的时间[16]，通常由若干个振荡周期构成。在 8051 微控制器中，机器周期固定由 12 个连续的振荡周期组成，每个机器周期可分 6 个状态，且每个状态又可分为 2 个节拍。众所周知，传统的 8051 微控制器是复杂指令集控制器（CISC），它执行不同的指令所需要的时间并不完全相同。所以，指令周期就是执行一条指令所需要的时间。显然，指令周期是指令时序中最大的时间单位。8051 微控制器指令可分为单周期指令、双周期指令和四周期指令三类。

为了客观评价程序效率，本文使用振荡周期数作为计算比较的指标，而不是使用程序执行完成所需要的实际时间。

2.2.2 8051 存储器系统

作为微型计算机系统，数据是各种控制及运算的依据和加工的对象。因此，存储器性能很大程度上影响着程序的执行效果，这一点尤其体现在程序执行的速度上。而从整体上看，变量的存取速度也影响着系统的实时性。特别地，8051 可以操作的存储器类型非常丰富。正是出于这个原因，为了理解下文中 C51 开发环境为了支持各种类型的存储器而定义的许多 ANSI C 之外的关键字，有必要先熟悉 8051 的存储器类型与结构。

从数据存储的角度看，8051 的存储器分为只读存储器（简称 ROM，即程序存储器）和随机存取存储器（简称 RAM，即数据存储器）。这两种存储器从物理位置的角度进行划分，又分别可以分为片内和片外两类，如图 2-1 所示[15]。

程序存储器中是固化的机器代码。通常一个正常工作的 8051 系统，程序存储器不论片内片外都工做在同一速度上。这表示在程序员看来，系统中可操作的

程序存储器只有一种。但数据存储器却不是这样。

①片内数据存储器

即 8051 内部 RAM，指 8051 控制器芯片中与控制器内核一起封装的一部分存储器。主要是由于成本原因，这部分的存储器容量都非常少，但也能满足一般应用的需要。

内部 RAM 的特点是，存储空间小，存取速度快。

标准 8051 的内部 RAM 为 128 字节。它包含工作寄存器组，即存取速度与寄存器一致，是整个系统中最高速的数据存储器。所以常用于存放实时数据，其中一部分还必须作为数据堆栈，以支持函数调用。

②片外数据存储器

即外部 RAM，指 8051 控制系统中，位于控制器芯片外，连接在 8051 三总线接口（地址总线、数据总线、控制总线）上，能被 8051 控制器内核直接访问的存储器。这部分存储器通常是在内部存储器无法完成数据存储任务时，为了满足应用需要而额外扩充的存储器。

外部 RAM 的特点是：存储空间大，存取速度慢。

标准 8051 的外部数据存储器寻址空间为 0000h-FFFFh，共 64KB。因为 8051 极具代表性的 8 位总线能和通用计算机系统的 ISA 总线基本兼容，所以这部分空间除了作为标准的数据存储器使用，还常被用来作为一些 ISA 设备的地址空间，比如以太网接口芯片 RT8019AS。

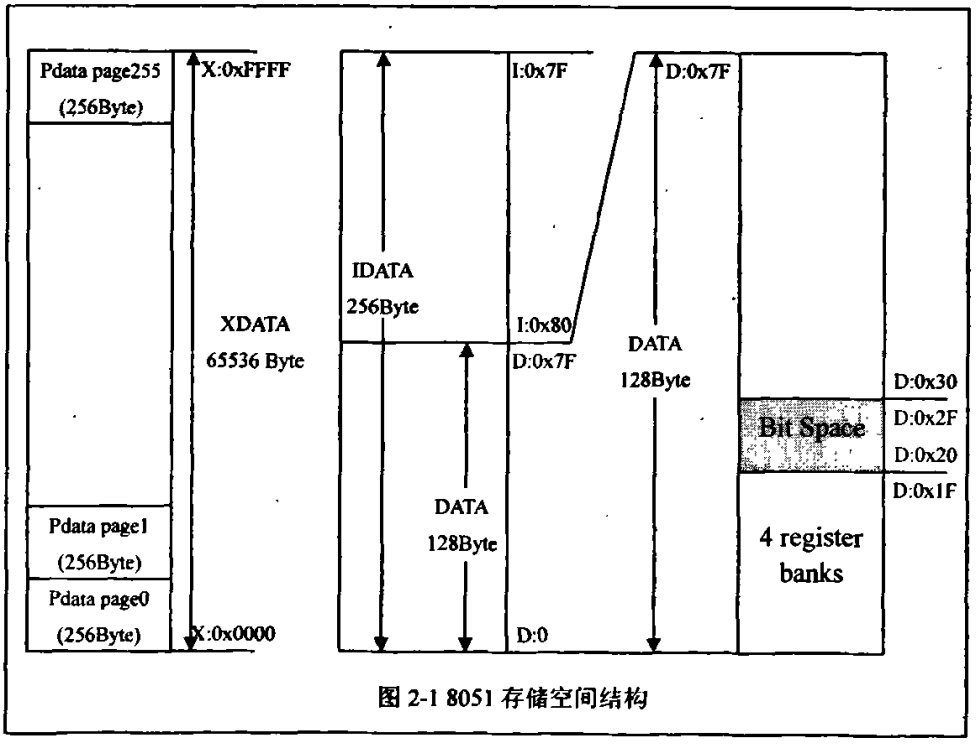

图 2-1 8051 存储空间结构

数据存储器类型

内部数据存储空间最大为 256 字节，标准 8051 仅有内部存储器空间的前 128 字节。从功能上看，标准 8051 简称为内部 RAM 可分为工作寄存器组、位寻址区、和便签区，如图 2-1 所示。

从寻址方式上看，内部数据存储器可分为直接寻址内部存储器和间接寻址内部存储器。直接寻址内部存储器仅覆盖内部数据存储器的前 128 字节。相对地，非直接寻址内部数据存储空间包含整个 256 字节的内部数据存储器。非直接寻址方式下需要先设置寻址寄存器内容，所以一般情况下，使用非直接寻址方式操作内部数据寄存器速度比直接寻址方式要慢。

外部数据存储器存取速度慢的主要原因是外部数据存储器的操作必须通过地址寄存器进行。外部数据存储器的寻址方式分为 8 位寻址方式和 16 位寻址方式两种。因为操作外部数据存储器必须先设置好地址寄存器，设置 8 位地址比 16 位地址稍快，即 8 位寻址方式下，操作外部数据存储器的速度略快于 16 位寻址方式。显然，使用 8 位寻址方式的程序完成同样功能比使用 16 位寻址方式的程序代码也稍短。

2.2.3 8051 的堆栈

必须注意，8051 使用片内 RAM 作为数据堆栈。这直接引起了两个问题：一是受片内存储器容量限制，8051 的程序不能太复杂，函数调用的嵌套层次不能太多；二是在应用操作系统时，保护堆栈的工作量较大。即 8051 缺乏堆栈保护的硬件结构，这导致了多任务操作系统的上下文切换中保护现场的开销比在通用计算机上的开销要大。

2.2.4 中断系统

标准 8051 系列单片机有五个中断源，提供两个中断优先级，可实现一级中断嵌套。这两级优先级遵循下列规则：仅高优先级中断源可中断嵌套低优先级中断源。为实现这一规则，中断系统内部包含两个不可寻址的优先级状态触发器。当特定优先级的某中断源被响应时，相应的触发器即被置位，直到执行了 RETI 指令后，这个触发器才被复位。在此期间，同级和低级中断将被阻止。中断源的中断请求能否得到响应，受中断允许寄存器 IE 的控制。每个中断源的优先级可通过对中断优先级寄存器 IP 编程来设定：或最低，或最高。同一优先级中的各中断源同时请求中断时，由内部查询逻辑确定响应次序。查询次序依次为:外部中断 0（EX0）、定时器中断 0（T0）、外部中断 1（EX1）、定时器中断 1(T1)、串口中断(S)。如果当前指令是 RETI 或是对 IE、IP 操作的指令，将封锁 CPU 对中断的响应，且必须再执行完一条指令之后才会响应中断[17]。

8051 系统采用中断优先级机制来保证紧急中断的快速响应。即高优先级的中断可以中止低优先级的中断处理，而同一优先级或较低优先级的中断不能中止较高优先级的中断处理。

2.3 Keil C51 概述

2.3.1 C 语言与 8051 变量

C51 编译器，以至于之后发展的 Cx51 编译器，就是 ANSI C 的完整实现。C51 的一个最重要的功能就是为 8051 微控制器产生高效紧凑的程序代码。其目的就是为了向用户提供一个既具有 C 语言灵活性又有汇编语言高效性的编程环境[18]。

8051 系列是微控制器体系结构中发展最快的体系结构之一。今天有超过 400 多家芯片制造商在制造 8051 系列控制器。经过扩展的 8051 设备，如 Philips 的 80C51 MX 系列，就是针对超过数兆字节的大型应用而设计的。为适应新型 8051，在 C51 基础上便发展出来了 Cx51 编译器。本文的研究都是在 Cx51 编译器上进行的，除特殊说明外，C51 就是指 Cx51。

C51 为了良好地支持 8051 结构的特点而提供了许多对 ANSI C 的扩展。这些扩展包括了如下几个方面[18]：

- ✧ 存储器空间
- ✧ 存储器类型
- ✧ 存储器模型
- ✧ 存储器指定功能
- ✧ 变量类型指定功能
- ✧ 位寻址及位变量
- ✧ 特殊功能寄存器
- ✧ 指针
- ✧ 函数的特殊属性

2.3.1.1 C51 关键字与 8051 体系结构

从 C51 角度看 8051 存储器空间看按以下三个标准进行划分：

- ✧ 只读空间
- ✧ 读写空间
- ✧ 高速读写空间

CODE——该关键字就是指程序空间。该存储器空间只能读出而无法写入。

标准 8051 的程序空间为 64KB，而 8051 控制器只能执行程序空间中的代码。必须注意需要执行的程序只能放在程序空间中，这包括全部需要使用的自定义函数、C51 提供的标准库以及自定义常量。

如前文存储器类型所介绍的，8051 的内部数据存储器可分为三个独特的存储器类型——直接寻址区、间接寻址区和位寻址区。分别对应 C51 关键字——data、idata 和 bdata。

data 关键字对应的是内部数据区的直接寻址空间。idata 关键字对应的是内部数据区的非直接寻址空间。bdata 关键字指位于内部 RAM 中 0x20-0x2F 的位寻址区，只有 16 字节。其中每个字节都可以单独进行位操作。

C51 使用关键字 pdata 和 xdata 分别对应 8 位和 16 位寻址方式下的外部数据存储器。其中 xdata 所指代的空间覆盖了全部 64kB 的外部数据存储空间。而 pdata 仅指代外部数据存储空间中的一页，即 256 字节[18]。

按速度和容量，C51 的存储器模型归纳起来如表所示。其中 fosc 为 8051 控制器工作的主频。

	data	idata	Pdata	xdata	重入堆栈[1]
容量(字节)[2]	128	256	256	65536	65536
存取所需机器周期[3]	2	2	4	5	15
带宽 (字节/秒)[4]	fosc/24	fosc/24	fosc/48	fosc/60	fosc/180

表 2-1 8051 存储器性能对比

2.3.1.2 C51 与运行模式

C51 的特点之一是为了尽可能地产生高效率的程序代码。其编译结果的运行效率取决于两个参数，一个是程序存储器编译模式（Code Rom Size），一个是数据存储器编译模式（Memory Model）。这两个参数的设置界面在“project”菜单下的“options for target”选项中。在配置该选项之前首先要按 C51 工程管理的方法建立好一个工程项目，然后才能配置这些选项。这两个参数的配置，决定了编译器对函数及变量缺省方式下的分配策略。从下文的分析可以得出结论，这两个编译参数中，数据存储器编译模式对程序执行的速度影响是起决定性作用的。

程序存储器编译模式有小型(small)、紧凑型(compact)和巨型(large)三种，并且相同的 C 程序使用这三种类型编译所产生的代码依次变大，效率依次降低，

[1] 重入堆栈的内容在后文的 3.1.3.1 一节中。
[2] 这里的容量是从指令上看的容量，实际容量随 8051 型号或系统配置的不同各不相同。
[3] 这里是指单字节存取方式下操作相关存储器所需的机器周期数，其中还包括了外部程序存储器所需的地址寄存器赋值所需要的时间（机器周期）。
[4] 这里的带宽是指单字节操作下的存取带宽。

但其差别不大。这样说的理由是各跳转指令所需的时间基本相同，都为 2 个机器周期。程序存储器编译模式根据实际拥有的程序存储器空间进行选择。

数据存储器编译模式也分为小型(small)、紧凑型(compact)和巨型(large)三种。

小型模式下，编译器缺省地将函数变量指定为 data 类型，即内部 RAM 型。这样程序可以以最高速度操作变量，从而使程序的执行速度最快，实时性最好。但是因为 256 字节的内部 RAM 空间过于狭小，并且系统堆栈必须在内部 RAM 空间分配，实际所需堆栈大小与程序嵌套的层次数密切相关，这导致小型模式下堆栈空间的大小成为一个十分关键的问题。

紧凑模式下，编译器缺省地将函数变量指定为 pdata 类型，即 8 位寻址方式的外部数据类型。根据 Intel 8051 说明书，我们也可称之为页类型。在 C51 环境中，pdata 数据只有 256 字节，大小与内部 RAM 类型一样。pdata 带宽只有 data 型的一半，但必须注意，在 8051 系统中 pdata 的速度是仅次于 data 型的。关键是，pdata 与堆栈不相关，不存在与堆栈争用空间的问题。

巨型模式下，编译器缺省地将函数变量指定为 xdata 类型，即 16 位寻址方式下的外部数据类型。在重入堆栈出现之前，这是最低效的存储类型了。但其最大 64KB 的存储空间却极大地缓解了“寸土必争”的 8051 存储空间问题。该模式使得 8051 控制器能以较高的效率处理更复杂的问题，分析较为复杂的数据包，比如如使用通用操作系统和以太网数据包。

2.3.2 C51 与重入

在熟悉了C51的各种特性之后，我们可以讨论C51与操作系统之间的关系了。这时必须提到C51链接/定位器（linker/locator）在对变量进行定位时的一些特征。首先从 Keil C51 与 ANSI C 的区别开始。

Keil C51 为能产生高效率的 8 位单片机代码，所以设计的时候与 ANSI C 标准还是有一些区别。而以 μC/OS 为代表的操作系统，常常要求编译器符合 ANSI C 标准。

查阅 Keil C51 的用户手册，我们知道 Keil C 的编译器与 ANSI C 的区别有两点：一是对 16 位长字符的支持；二是对递归调用的支持[18]。这里必须对递归调用做进一步讨论。

我们已知 8051 系统的堆栈必须在内部 RAM 中[15]，而内部 RAM 的容量非常小，仅有 256 字节。通过查阅链接/定位器用户手册，我们还发现因为 8051 系统的堆栈寻址功能效率低，使用堆栈方式传递函数的参数当然就比直接使用绝对地址方式传递参数慢。正是因为这些原因，Keil C51 链接/定位器将临时变量和函数参数一起定位于固定内存地址，而不是使用堆栈地址。并且，为了最大化利用存储器空间，链接/定位器使用静态覆盖技术（overlaying）对函数参数及临时变

量进行分析，以保证不同函数的变量能使用同一内存地址，从而实现数据的静态覆盖。简单地说，在传统方式的单任务 8051 开发中，只要两个函数没有相互调用，那么他们的临时变量就能共享同一块内存。

实践证明，静态覆盖技术的应用非常好，并且确实比使用堆栈方式参数和变量的效率要高。因为编译时栈分布就已经确定，所以该技术也被称为“编译时”栈技术。

但是使用这种静态覆盖技术的函数在发生递归调用时，固定地址的临时变量将被改写，因此，缺省情况下，Keil C51 编译的函数都不支持递归。而在操作系统中，由于多任务的出现，类似递归这样的现象更普遍，并抽象为另一个更普遍的现象——重入（Reentrancy）。

重入的详细描述如下。一个函数在调用过程中，在第一次调用还没有返回的情形下又发生了第二次或更多次调用，那么就称为发生了函数的重入或发生了重入函数。重入函数过程中如果使用了全局变量，那么很显然，在第二次进入后会破坏第二次运行过程中的变量，等返回之后第一次的调用时的变量内容必然发生无法预测的改变，破坏了算法的确定性与正确性。如果在函数中避免使用全局变量，就可以使函数具有可重入性，ANSI C 中的局部变量都是存放在堆栈之中的，因此自然就可以满足重入性的要求，只要在函数中只使用局部变量就可以了。而在 Keil C51 中，由于考虑到单片机的内部硬件堆栈非常小(大约只能有 128 字节)，无法满足子函数对堆栈容量的需求，因此 Keil C51 将函数的局部变量也当作静态变量来处理，固定地占据一部分数据空间，从存储模式上来说和全局变量是相同的，因此默认的 Keil C51 下的函数均不可以重入[19][20][21]。

为了解决这个问题，Keil C51 专门提供了一个关键字 reentrant 供用户使用，这一关键字会告诉编译器，将在一个特别开辟的“重入堆栈（reentrant stack）”中临时存放重入函数的局部变量和参数。因为重入堆栈的指针和数据都必须由软件来维护，所以该堆栈也被称为“仿真堆栈”。

2.3.3 Keil C51 的函数库

Keil C51 的运行时(run-time)函数库为用户提供了 100 多个用于 8051 C 语言开发的预定义函数或宏[18]。这些库函数和微控制器的运行模式密切相关。只有正确合理地选择运行模式以及相应的库函数才能最大程度地发挥微控制器性能。库文件就是 Keil C51 预定义函数或宏的一个集合。

一般来说，Keil C51 库中的函数是符合 ANSI C 标准的。但是为了合理利用 8051 的体系结构，有少量函数还是做了些修改，结果是和 ANSI C 标准有些微不同。对 ANSI C 标准的改动也是为了能最大程度地发挥控制器的性能以及减少程序代码。

2.3.3.1 库文件的选择

由于 8051 衍生产品非常多，为了充分发挥控制器的特殊性能，Keil C51 提供了许多库文件。表 为常见的六个[5]库文件，其中每个文件都对应着一种编译模式。

库文件名	说明
C51S.LIB	无浮点运算的小型（small）模式库
C51FPS.LIB	带浮点运算的小型（small）模式库
C51S.LIB	无浮点运算的紧凑型（compact）模式库
C51FPC.LIB	带浮点运算的紧凑型（compact）模式库
C51L.LIB	无浮点运算的巨型（large）模式库
C51FPL.LIB	无浮点运算的巨型（large）模式库

表 2-2 库文件说明

一般地，当在项目中根据芯片产商选定产品类型，并设置好编译模式后，Keil C51 开发环境会自动选择对应的库文件，以缺省方式参加整个工程的编译和链接。但是在一些特殊情况下，用户还是必须手工向工程中明确指定所需要的库文件，比如后文中，我们使用一个新型 8051 控制器，且开发工具还未及时提供相应库文件。

我们只能从 Keil C51 的官方网站上找到一些关于库文件的说明。经过整理，我们看到确定项目中的库文件需要根据以下三个要素加以选择：芯片产商名称、是否需要浮点运算及运行模式。其中产商名称位于库文件前端，可看做文件名的前缀；而浮点运算及运算模式为文件名后缀。如 C51S.lib 表示的是标准 8051 的微型模式库，CD51FPL.lib 则是属于 Dallas 公司产品带浮点运算巨型模式的库文件。

2.3.3.2 函数库与重入

有丰富的库文件可供利用，是开发者选择 C51 进行 8051 开发的主要原因之一。在上文中我们看到，向 8051 系统中引入多任务开发则一定需要解决代码重入问题。同样，在使用函数库时也需要考虑代码重入。

C51 用户手册有以下说明[18]：许多标准库函数具有可重入性，但用户手册中特别说明的程序外，标准库函数都没有重入性。其中包括部分数学运算函数、大部分的字符串处理函数、流处理函数以及全部的内存分配函数。这要求使用者在多任务开发中必须明确其使用的函数是否为可重入的，增加了使用者的工作量。

[5] 作者所使用的 Keil C51 （7.00 版本）中，LIB 子目录下的库文件已增加到了 50 个。

若多任务中使用了非重入函数，则必须将该函数改写为重入函数[43]。必须注意，这种修改库文件标准函数的过程并不是简单地在函数名后使用 reentrant 关键字就能完成的。

2.3.4 C51 的局限性

当然，C51 在使用过程中也有一些限制[22]。例如，通常函数中的局部变量生成单独的数据段，位变量要生成位段。为了使程序模块化，我们应编制很多自定义函数，可是由于位段与数据段的数量有一定的限制，所以自定义函数的数量也不允许太多。此外，条件编译指令的最大深度限制不能超过 20，宏限制最多允许嵌套 8 级等等。

总的来说，Keil C51 编译器对传统模式的 8051 开发是非常有效的工具，但对它在多任务重入所产生的代码效率非常低，值得研究改进。

2.4 基于 8051 的嵌入式实时操作系统

8051 控制器的主要特性是它优秀的控制性能，和一般桌面系统中的处理器相比，8051 的运算能力就显得非常薄弱和简陋了。所以一般操作系统难以在 8051 上移植。

2.4.1 μC/OS-II

2.4.1.1 μC/OS-II 移植的条件

μC/OS-II 实时操作系统并不是针对 8051 核的嵌入式微控制器设计的，为此必须结合 8051 微控制器内核结构和开发工具的特点对 μC/OS-II 进行必要修改，使之能在该微控制器上运行。这种保持原程序主要结构及算法特征，使之能在另一种运行环境下工作的开发过程就是“移植”。μC/OS-II 在设计时就已经充分考虑了可移植性，其能够成功移植的前提条件是目标处理器必须满足以下条件[23]：

1)处理器的 C 编译器能产生可重入代码；

2)用 C 语言就可以打开和关闭中断；

3)处理器支持中断，并且能产生定时中断(通常在 10-100Hz 之间)；

4)处理器支持能够容纳一定数量的硬件堆栈；

5)处理器有将堆栈指针和其它 CPU 寄存器读出和存储到堆栈或内存中的命令。

通过上文的介绍，我们知道 Keil C51 可以满足条件 1 和 2，且 8051 作为移植 μC/OS-II 的目标处理器，它满足移植条件 3，4，5。

2.4.1.2 μC/OS-II 的任务管理

如图 2-2，我们以任务状态转移图的方式可以得到 μC/OS-II 的任务管理流程。

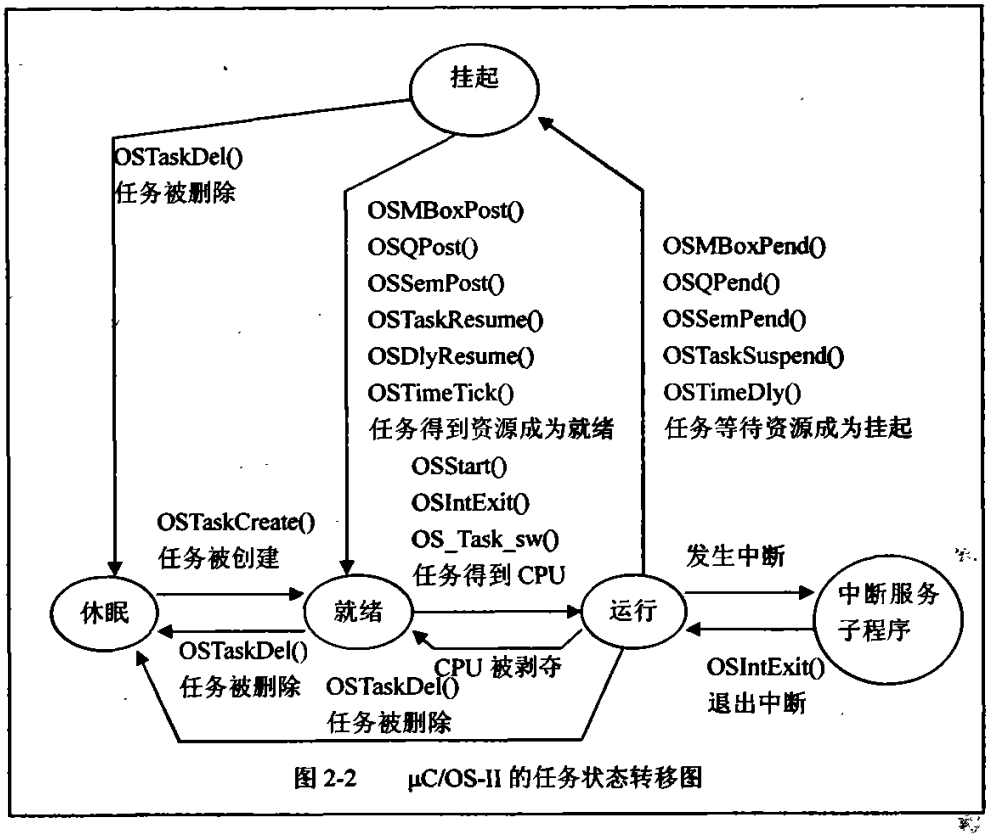

图 2-2　μC/OS-II 的任务状态转移图

2.4.1.3 μC/OS-II 在 80C51 系列 MCU 上移植的关键技术

以下是 μC/OS-II 移植到基于 8051 微控制器上的一些关键技术问题的分析：

(1)使用多组 Keil C51 重入堆栈

为了满足多任务开发中的重入问题，使用 Keil C51 编译移植 μC/OS-II 都是使用 Keil C51 提供的重入函数功能[靳文兵 101]。为了分离不同任务的临时变量，在实际开发中，必须为每个任务在存储器中开辟一个单独空间作为重入堆栈使用。

(2)上下文切换

因为 8051 系列微控制器没有软中断指令，所以原 μC/OS-II 中软中断方式的上下文切换程序 OSCtxSw()必须改写为非中断方式的函数[44]。

因为 μC/OS-II 为抢占式调度，非中断过程中的上下文切换被设计成与中断方式上下文切换使用同一段代码，即 OSCtxIntSw()基本直接对应 OSCtxSw()。这就

要求非中断过程中对现场的保护必须与中断现场保护完全一致。一般情况下使用 Keil C51 开发中断服务程序只需要指定关键字 interrupt 即可，因为编译器将自动根据编译结果将使用到的寄存器保存起来。

必须注意到，不同中断服务程序由于其复杂度不一样，则保存在现场中的寄存器的数量或入栈顺序不一样。在多任务开发中，由于非中断过程和各中断过程使用同样的上下文切换代码，则上下文切换的现场保护必须统一起来。即不论中断服务程序，还是非中断过程中的调度程序，必须将所有可能使用到的寄存器以同一顺序推入堆栈中保存起来。

(3)一些关键的系统函数实现[1][24]

除了上下文切换程序 OSCtxSw()和 OSCtxSw()，还必须根据 8051 微控制器特点，修改“初始任务调度函数 OSStartHighRdy()”和“时钟节拍中断函数 OSTickISR()”。这些关键技术在许多文章中都有描述，读者可以查阅参考文献[23,24,43]。

2.4.2 RTX51

RTX51 是德国 Keil 公司开发的一种应用于 8051 系列单片机的实时多任务操作系统，它可以工作在所有 8051 单片机以及派生家庭中[25]。它有两个版本：RTX51 Full 和 RTX51 Tiny。

总的来说，RTX51 的特点是：内核代码全部由汇编语言编写，因此可读性差，但效率高。

RTX51 Full 支持事件驱动的抢先式多任务调度策略，允许 4 个优先权任务的循环和切换，并且还能并行的利用中断功能。它支持信号传递，以及与系统邮箱和信号量进行消息传递。RTX51 Full 的 os_wait 函数可以等待以下事件：中断、时间到、来自任务或中断的信号、来自任务或中断的消息、信号量。

RTX51 Tiny 是 RTX51 Full 的一个子集。RTX51 Tiny 可以很容易地运行在没有扩展外部存储器的单片机系统上。但是，使用 RTX51 Tiny 的程序可以访问外部存储器。RTX51 Tiny 允许最大 16 个任务循环切换，并且支持信号传递，还能并行的利用中断功能。但它不能进行消息处理，不支持存储区的分配和释放，不支持抢先式调度。RTX51 Tiny 是一个很小的内核，完全集成在 Keil C51 编译器中。更重要的是，它仅占用 800 字节左右的程序存储空间。RTX51 Tiny 的 os_wait 函数可以等待以下事件：时间到、时间间隔、来自任务或者中断的信号[26][27]。

2.4.2.1 任务管理

如图 2-3，RTX51 的任务共分为四个状态[28]：

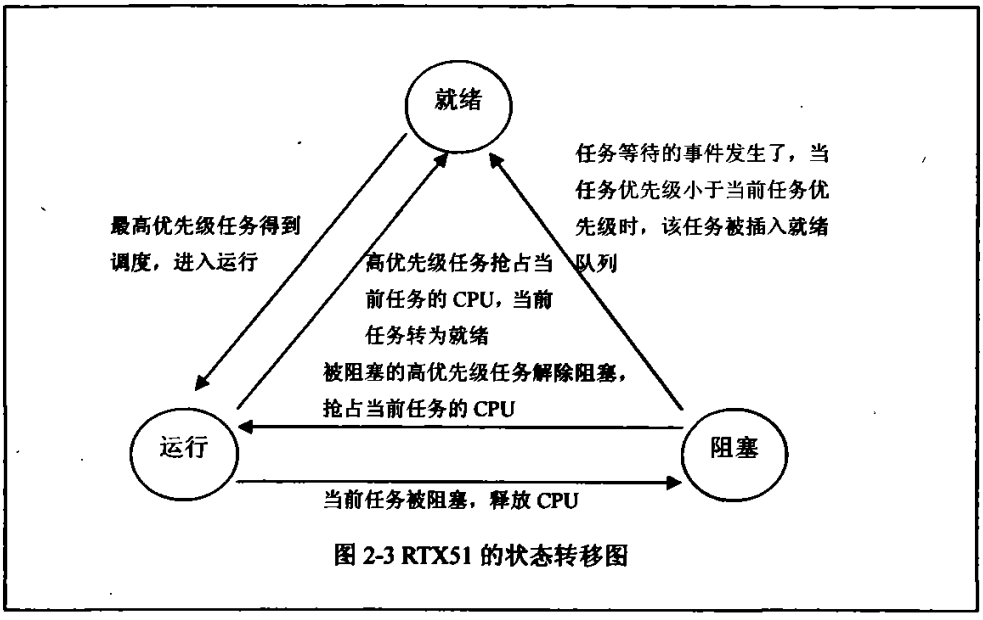

图 2-3 RTX51 的状态转移图

运行 (Running)：任务正处于运行中，同一时刻只有一个任务可以处于“RUNN NIG”状态。

就绪 (Ready)：等待运行的任务处于“就绪”状态。在当前运行的任务退出运行状态后，就绪队列中的任务根据调度策略被调度执行 ,进入到运行状态。

阻塞 (Blocked)：等待一个事件的任务处于“阻塞”状态。如果等待的事件发生，则此任务进入“Ready”状态，等待被调度。

休眠 (Sleeping)：被申明过但没有开始运行的任务处于“休眠”状态。运行过但已经被删除的任务也处在“休眠”状态中。

处于“READY/TIMEOUT”、“RUNNING”和“BLOCKED”状态的任务被认为是激活的状态，它们的转换关系如图 2-3。“SLEEPNIG”状态的任务是非激活的，不能被执行或认为已经终止。

相对地，在 RTX51 Tiny 中人们将用户任务划分为五个状态——运行、就绪、阻塞、休眠和超时(Timeout) [27]。其中前面四个与 RTX51 完全一致，“超时”表示任务由于时间片用完而处于“Time out”状态，并等待再次运行。该状态与“READY”状态相似，但由于是内部操作过程使一个循环任务被切换，因而单独算作一个状态。我们将非激活的休眠状态和图 2-3 的状态组合在一起，便得到了图 2-4 所示 RTX51 Tiny 的完整状态转换图。其中虚线部分为本文作者补充的部分。

对比图 2-2 和图 2-4 中我们可以看到 RTX51 与 μC/OS-II 的任务在状态转换上基本相同。

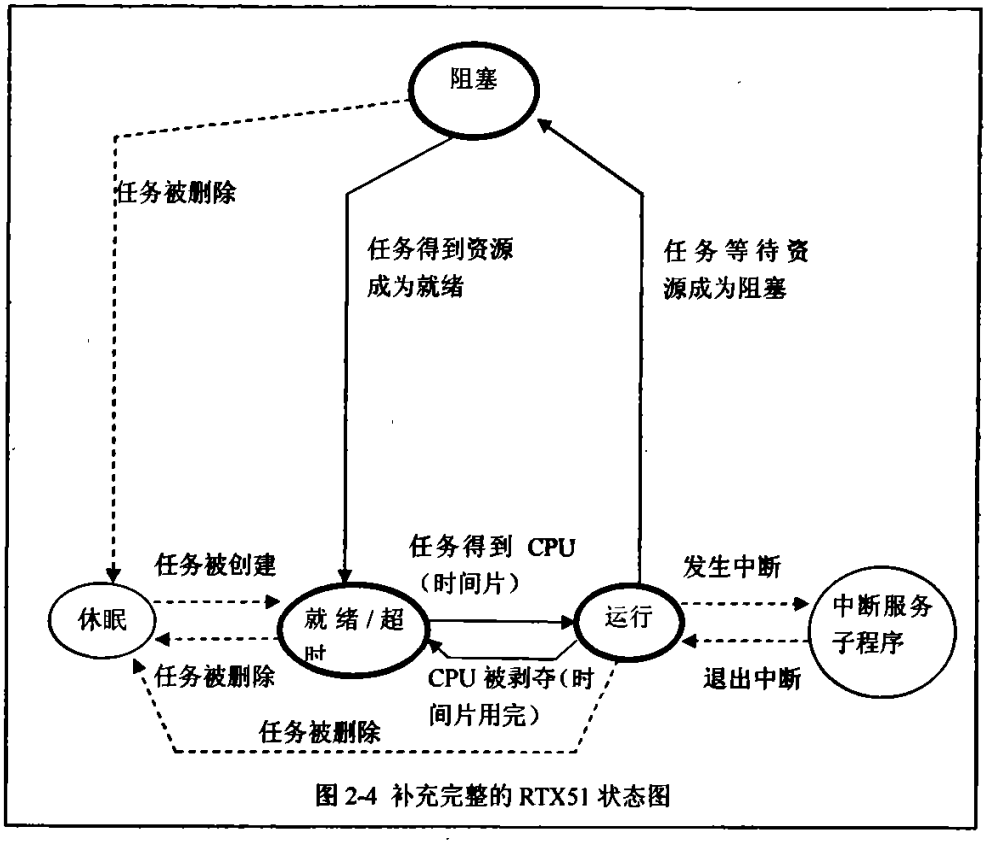

图 2-4 补充完整的 RTX51 状态图

2.4.2.2 中断处理

RTX51 Tiny 提供两种方法来处理中断：一种是 C51 中断函数，包括软件中断和硬件中断。对于硬件中断，用户不能再使用定时器 T0。当中断发生的时候，程序跳到相应的中断处理函数。中断处理过程与正在运行的任务是相互独立的，即中断处理过程在 RTX51 系统内核之外和任务切换规则没有关联。但中断处理过程可以同 RTX51 任务互发信号或交换数据；另一种是 RTX51 的任务中断。如果中断发生，等待中断的任务就从“等待”状态进入到“就绪”状态，并按照任务切换规则进行切换。这种中断处理是完全集成在 RTX51 内部，与硬件中断事件的处理以及信号、消息的处理是完全相同的。

两种方法可以在应用程序中混合使用。在系统响应时间上 C51 中断函数是最快的。RTX51 必须完全控制中断使能寄存器，这样才能遵守任务的切换规则并保证中断程序的无误进行。必须注意中断使能寄存器是由 RTX51 完全控制的，用户不能在程序中修改，否则可能会影响内核的正常运行。

第3章　基于 8051 的嵌入式实时操作系统设计与实现

上一章中，我们介绍了大量基于 8051 的嵌入式实时操作系统的基本概念和技术手段。一方面，专门为 8051 开发的 RTX51 操作系统使用静态技术，各任务的临时变量独占存储资源，即所有任务从一开始就将存储器占用，而不管该任务是否为激活状态。这样做严重限制了系统的可扩放性。另一方面，我们看到使用通用操作系统 μC/OS 实现基于 8051 的多任务时，因为微控制器硬件性能还不能有效地支持这样一种通用的操作系统，所以不得不使用纯软件的“模拟堆栈”技术。这使得系统的整体运行效率大打折扣，严重影响了系统的实时性。鉴于此，对目前简单移植方法的 8051 操作系统方案进行深入分析，充分利用现有技术和成果，研究实现更好的解决方案十分必要。

3.1 重入问题的分析与解决

使用多任务操作系统，通常需要解决重入问题。在 8051 系统上移植多任务操作系统的过程中，我们发现 RTX51Tiny 和 μC/OS 使用了两种不同的方案。μC/OS 使用的是 Keil C51 编译器提供的“重入函数”，而 RTX51 代码中却没有重入函数。其原因是 RTX51 Tiny 使用的是非抢占调度方式。这表示除非是进程主动释放控制器，否则不会被调度器中断执行。因此，各代码都完整执行，没有重入问题。显然，这种操作系统的实时性较差。

相对地，μC/OS 使用的是抢占方式调度。抢占方式下，调度器以中断方式停止函数执行，并将处理器分配给其他任务。显然，使用抢占方式的操作系统必然需要解决重入问题。目前，在 8051 上移植 μC/OS 的文章所使用的方案都是利用 Keil C51 提供的重入函数。其原因是跟 8051 的系统结构以及 Keil C51 的编译链接技术有关，在前文的 Keil C51 一节中已作详细介绍。

3.1.1 Keil C51 重入函数原理

3.1.1.1 技术原理

只要在 Keil C 函数后使用关键字 reentrant，就可将该函数声明为重入函数。Keil C51 重入函数的关键技术是使用了“重入堆栈”。编译器在系统存储区专门设置一个空间给重入堆栈。重入函数的临时变量分配将在重入堆栈中进行，而不再是和普通函数一样用静态覆盖技术在存储器空间分配，如图 3-1 所示。因为缺乏 8051 硬件的直接支持，重入堆栈的操作完全使用软件方式实现，因此，重入堆栈的效率很低。

Keil C51 用户手册对重入函数的说明是[6]：因为 8051 没有合适的寻址方式支持“重入堆栈”，这导致了重入堆栈仿真结构的低效性，但这样做是必要的。

3.1.1.2 代码分析

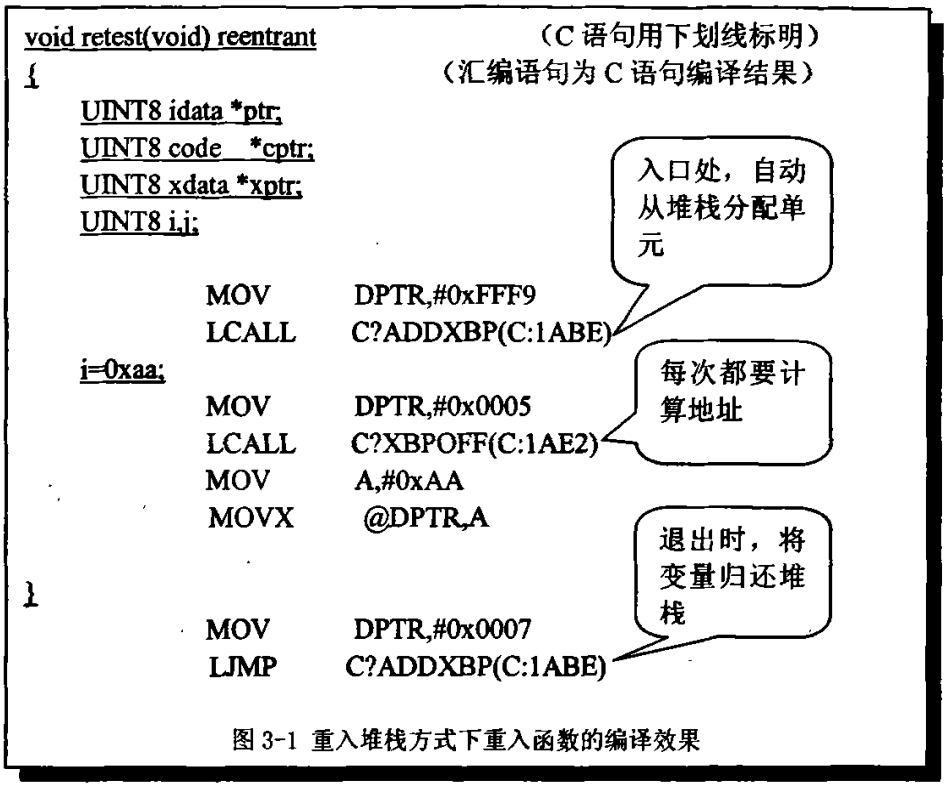

图 3-1 重入堆栈方式下重入函数的编译效果

图 3-1 给出了一个简单的重入函数代码。其中 C 语言代码用下划线标明，其余为编译后的汇编代码，也就是需要执行的机器码。这个例子清晰地展现了重入函数的执行过程：

首先，在函数入口处由编译器自动添加了一段程序（C?ADDXBP）以实现函数临时变量的分配。分配的对象是重入堆栈。具体做法是将当前重入堆栈指针向下移动，移动的偏移量正好为该重入函数所有临时变量的字节数。

分配完临时变量之后，才是真正由用户编写的 C 语句代码。该代码使用临时变量完成实质性的函数功能。必须注意，函数操作这些临时变量必须先计算变量地址“MOV DPTR,#0x0005”“LCALL C?XBPOFF(C:1AE2)”。

[6] Keil Elektronik GmbH. and Keil Software, Inc.. Cx51 Compiler Optimizing C Compiler and Library Reference for Classic and Extended 8051 Microcontrollers User's Guide [M] 09.2001: 130

最后，重入函数在返回之前还需要将临时变量归还重入堆栈。这里归还变量的操作与分配变量的操作一样，只是重入堆栈指针移动的偏移量正好是入口处偏移量的补码。

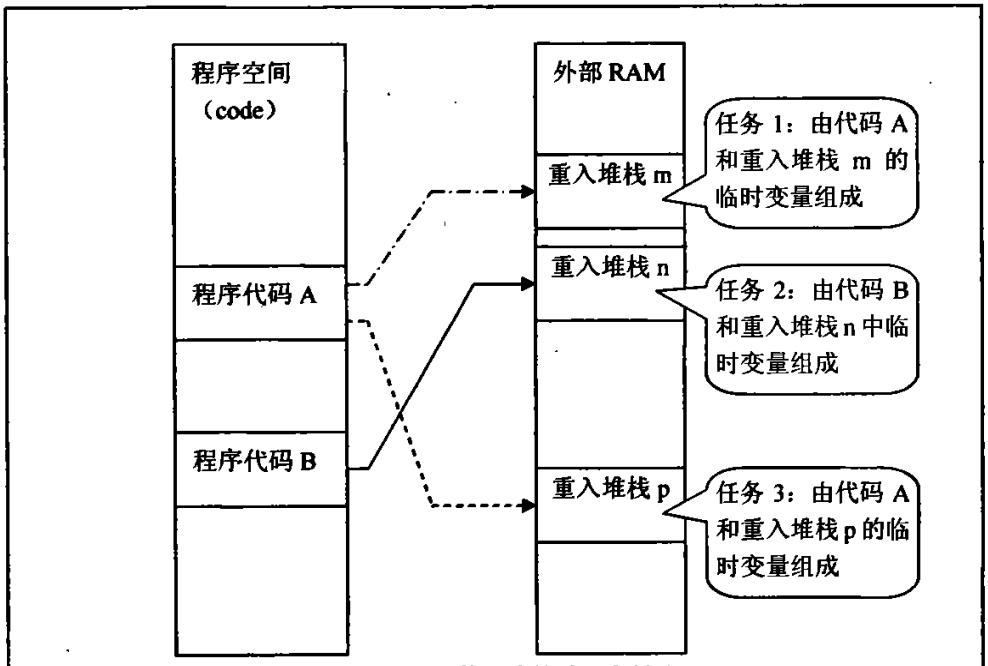

图 3-2 重入堆栈方式的重入变量分配

重入函数存在的理由自然是为了在当前 8051 和 Keil C51 环境下实现函数重入，但使用改方案则整体系统付出的代价是巨大的。重入函数的主要缺陷表现在实时性、易失性（volatile）和递归性三个方面。

①对实时性的影响。重入函数对实时性的影响表现在以下两点：一、重入函数有变量分配和回收的额外开销；二、每次使用临时变量都需要计算变量地址，大大降低了变量存取速度。最糟糕的是因为重入堆栈是纯软件实现的，所有这些的堆栈操作都必须由软件完成，消耗了处理器时间。严重影响了系统的实时性。

仔细观察分配过程，我们发现为了保证在 8 位控制器上实现 16 位运算的原子性，分配函数可能会临时关闭中断。这样直接引出两个不良后果：一是程序执行时间无法预测，二是中断临时关闭的时刻无法预测，导致中断过程及中断响应时间的计算更复杂。由于可预测性也是衡量系统实时性优劣的依据之一，重入函数的不可预测性客观地削弱了系统实时性。

②易失性。我们在图 3-1 中看到，程序结束前将使用过的单元放回重入堆栈。这说明，该重入函数无法使用 static 类型的变量。

③递归性。从图 3-1 的栈操作，可以看出重入的局限性。重入函数必须以先进后出的方式实现重入，而多任务系统中，任务的产生与结束具有随机性和多样

性，根本没有办法用先进后出的方法调度。简单地说，递归调用是重入方式之一。为满足任意方式的重入，操作系统要为每个任务安排一个私有重入堆栈，比如μCOS-II。在这种分组管理方式下，每个私有重入堆栈内部的空间碎片相互独立，会降低整体存储器的利用率。

在以上各种重入函数的缺陷中，对实时性的影响是最主要问题。而影响实时性的各个方面中，每次引用临时变量都需要计算变量地址是问题的主要矛盾所在。

3.1.2 基于页的多任务模型原理与实现

抛开现有方案的束缚，我们直接从研究重入现象本身以及8051存储器的自然特征入手。归根到底，解决函数重入的关键在于局部变量的分配，我们试图提出一种更高效率的存储器模型以满足重入问题。

3.1.2.1 原理

如第2.3.1节所介绍的，8051控制器能直接寻址的存储器类型很丰富。在这些类型中，我们注意到了页变量(pdata)及其与众不同的特性。

(1)页变量pdata

根据Intel用户手册，8051的页存储器就是指8位地址方式操作的外部存储器[3]。因为外部存储器统一使用16位地址编址。使用8位地址方式时，允许使用P2对外部数据来分页[30][31]。从指令上看，该方式的操作指令为(Movx @Ri)，其中R0或R1提供页内地址（低8位地址），P2寄存器隐藏地提供页地址（高8位地址）。

如第二章图2-1所示，单字节地址方式将8051的64KB外部存储器分成256个页面。其硬件特征如下：①系统的256个页面的存储结构完全一致；②工作页面可以被指定为这256页中的任意一页；③页地址由P2隐式提供给地址总线，且页地址可以由程序修改。

工具方面，Keil C51专用关键字pdata表示单字节方式寻址的外部数据类型，pdata变量即页变量。编译后的页变量具有以下特性：①页变量操作严格对应单字节地址方式。②页变量全部转换成了静态页内地址。

(2)页函数

根据页变量性质，只要函数的所有局部变量都被指定为页变量类型，则函数所生成的代码可以工作于系统任一页面上。

当操作系统为一个函数分配多个页面时，该函数与每一个页面上的局部变量均构成一个任务[7]，如图 3-3 所示，程序代码 A 分别与第 i 和第 k 页的页变量构成相互独立的任务（进程）。

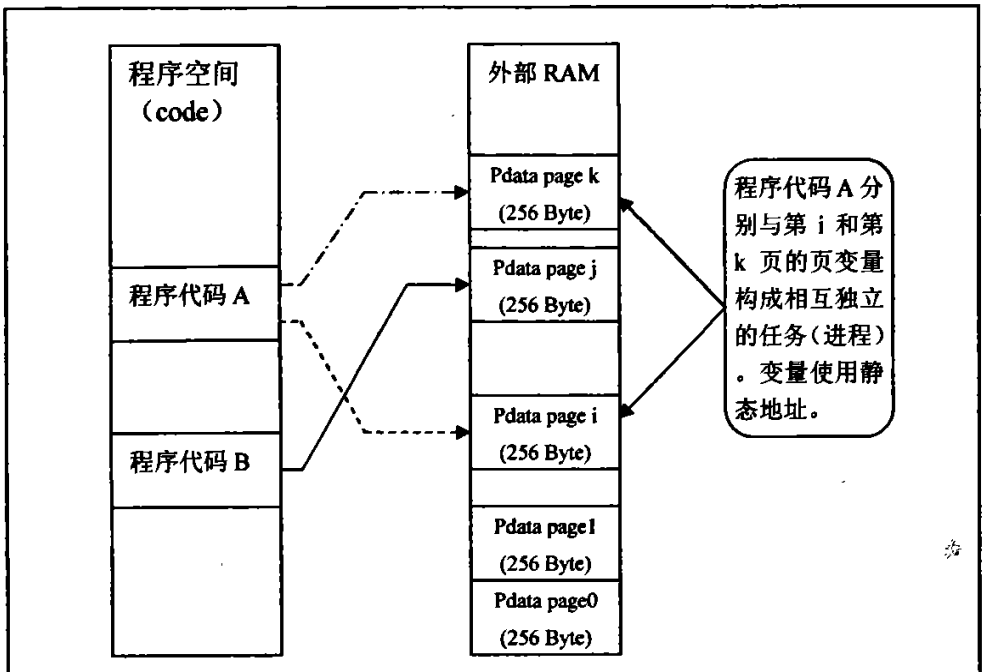

图 3-3 页变量方式的重入变量分配

与重入函数不同，这类函数本身不能自动分配变量，因此没有重入性。只有在操作系统的协助下，为其分配工作页后，页函数才是可重入的。页函数中变量使用静态地址，因此其存取速度得到大幅提升，改善了系统实时性。

为区别 Keil C51 定义的重入函数（Reentrant Function），我们称这种只使用页变量的函数为基于页的重入函数(Page-based Reentrant Function)，简称页函数。

3.1.2.2 实现的关键技术

从原理分析中看到，页变量的软件和硬件特征都指向多任务的可行性。据此，我们设计了名为 Celia 的基于页的占先式 8051 多任务调度内核。因为基于页的多任务模型与调度算法、调度流程彼此没有矛盾。我们沿用了 μC/OS-II 的结构框图与处理流程，以使基于页的多任务模型能更快运行。也可以说，我们开发的 Celia 操作系统是 μC/OS-II 在“分页多任务模型”上的移植。

从宏观上看，分页多任务开发必须从任务和操作系统两个方面进行修改，这

[7] 请注意嵌入式系统中进程与任务概念相近。这里若将任务换成进程，则该说法也是成立的。详见第二章 2.1.1

是因为，分页多任务的实现必须得到操作系统的支持。

(1)任务方面

任务使用虚拟存储的概念。每个任务都有自己的工作页面。程序员编写任务时，该工作页面可称为逻辑页面。

任务必须将任务中的私有变量声明为页变量类型。这些变量允许是临时的（可覆盖的），也可以是静态的（不可覆盖的）。同理，为了保证重入，任务所调用的子程序也必须指定为页变量。

值得一提的是，Keil C51 专门为 pdata 提供了一个 COMPACT 编译模式，它将变量缺省地分配在页变量空间。

(2)操作系统方面

操作系统必须将任务需要的系统调用按照分页方式编写。

操作系统还需要在任务控制块（TCB）中增加一个 page 字段。在执行任务上下文切换时把 page 值写入 P2 端口以完成旧页面的换出和新页面的换入。这也是任务的逻辑页面映射到物理页面的过程。

基于页的多任务操作系统 Celia 完全不使用 Keil C51 提供的重入函数，这是她与 μC/OS-II 的最大区别。经过在 W78P438 为核心的平台上实测，该系统调度正常，程序运行正确。这证明该重入方案可行，以下对方案性能进行分析对比。

3.1.3 基于页的多任务操作系统性能分析

基于页的多任务操作系统 Celia 的程序结构与流程与 μC/OS-II 基本一致，其根本区别在于使用的重入模型不一样，Celia 使用页函数实现重入，而 μC/OS 使用 Keil C51 的重入函数。因此，我们主要分析页变量存取的性能。图 3-4 为与重入堆栈方式的重入函数与基于页的重入函数编译效果对比。

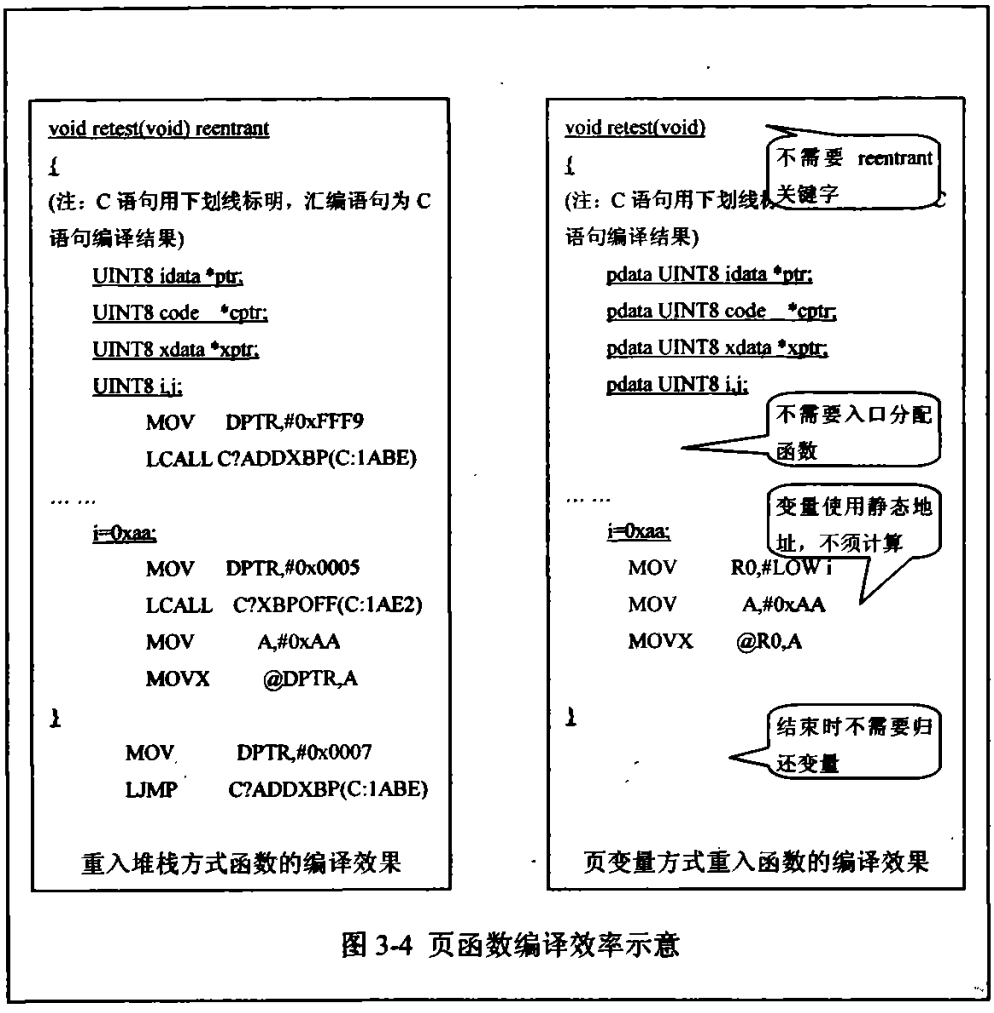

图 3-4 页函数编译效率示意

3.1.3.1 优点

(1)更高的变量存取带宽

8051 为 8 位总线接口，单字节存取是其基本操作。研究单字节变量的存取带宽可以从根本上说明页变量的优点。

8051 系统操作外部存储器使用的是 Movx 指令。执行该指令需要两个机器周期。在标准 8051 中，一个机器周期为 12/fosc。则总线带宽如(3.1.1)式所示，其中 fosc 为晶振频率。在 Keil C51 中对任意位置的外部数据区完成一次存/取操作需要 3 条汇编指令，共 5 个机器周期。因为是对任意地址的存取操作，这 3 条指令中包含了向地址寄存器 DPTR 赋值的操作。2 式所表示的就是普通 xdata 变量的存储带宽。

$$总线带宽=1B\times\frac{fosc}{12\times2} \tag{3.1.1}$$

而从图 3-1 中我们看到，在重入函数中对单字节变量执行一次存（或取）操作需要执行 11 条指令，共 15 个机器周期。进一步分析可知，其中仅计算变量地址并写入地址寄存器就需要 12 个机器周期。因此，重入堆栈的实际存取带宽如(3.1.2)式所示。

$$重入堆栈存取带宽=1B\times\frac{fosc}{12\times15} \tag{3.1.2}$$

相对地，页函数中变量地址是确定的。因为不需要计算地址的额外操作，其变量操作速度比重入堆栈有大幅提高。如图 3-4，包括设置地址寄存器，页函数中操作任意位置的单字节变量只需要 3 条指令（“MOV R0,#LOW i”“MOV A,#0xAA” “MOVX @R0,A”），共 4 个机器周期。故页变量的实际存取带宽如(3.1.3)式。

$$页变量存取带宽=1B\times\frac{fosc}{12\times4} \tag{3.1.3}$$

进一步分析可知，操作多字节变量时，重入函数也只需要计算一次变量地址。故进行单字节变量存取时，重入堆栈的存取带宽就是最低值。故存取单字节变量过程中，页函数对重入堆栈的带宽加速比达到最大值（Rmax）3.75，如(3.1.4)式所示。

$$Rmax=\left(\frac{fosc}{48}\right)\Big/\left(\frac{fosc}{180}\right)=3.75 \tag{3.1.4}$$

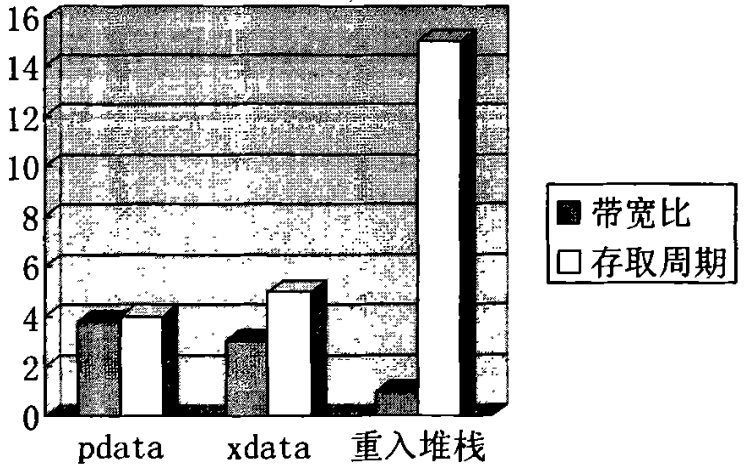

图 3-5 页变量、外部 RAM、重入堆栈的速度对比

另一方面，页函数不需要计算和操作重入堆栈指针，则图 3-4 中函数入口和出口处的指针操作(C?ADDXBP)不需要，可再次节省 22 个(首尾各 11)指令周期。

附录 1 给出了 C?ADDXBP 和 C?XBPOFF 函数的汇编代码，这是我们计算指令周期的依据。

综上，相对于过去的重入函数，页函数既能满足重入需要，又具有更高的执行效率和实时性。

(2)更好的安全性

配置页面的工作是操作系统完成的。在使用基于页的多任务开发中，任务本身不需要更换页面[8]。因此，私有变量的操作只在当前页进行，不会影响到其他页或其他任务。这样的程序封装体现了较好的安全性。

(3)更高的存储器利用率

分页多任务使用虚拟存储概念。每个任务都有自己的静态变量空间，即逻辑页面。所有的任务只在被操作系统建立时才分配物理页面，提高了系统存储器的利用率。而且，这种规格化的虚拟页使换页功能得以实现。必要时，可以将一些任务的工作页面换到更低速的辅助存储器中，以提高系统内存的利用率。关于换页策略，我们可以参考桌面系统中已经比较成熟的算法和理论。

为保证系统实时性，我们将任务划分为硬实时与软实时。对于硬实时任务可以禁止操作系统对它执行换页操作。

(4)非易失性

从重入堆栈的运行方式看，重入函数在入口处为局部变量申请存储单元，相应地，在函数返回前，需要将用过的单元归还给重入堆栈。显然，该方式下，局部变量的生存期只在函数内，即该方式下的局部变量不可能具备 static 属性。当遇到函数需要 static 类型局部变量时，情况会如何呢？

通过实验，我们注意到堆栈方式的重入函数中允许声明 static 变量。该变量并不在重入堆栈中分配，而是被分配了一个确定的位置。当重入发生时，该变量成为一个静态全局变量。显然，声明 static 变量的重入函数就失去了重入性。但这样的做法却不会引起编译和连接的失败或警告。

与堆栈方式不同，分页方式下允许在重入函数内声明 static 变量。局部变量使用 static 属性时，链接器可以保证该变量在逻辑页面上不被覆盖。因此，static 变量的生存期就等于任务的生存期。换句话说，对于任务 static 变量是非易失的。显然，逻辑页中的 static 变量依然是私有的，满足重入的需要。

3.1.3.2 存在的问题与解决方法

(1)容量的限制与颗粒度

[8] 因为 8051 存储器管理没有提供特殊保护机构，所以任意程序都可以修改页地址寄存器内容，从而可能引起系统崩溃。但这是属于编程的质量问题。

8051 的硬件决定了页面大小为 256 字节，不可变更。这使得“页面容量限制”成为基于页的多任务开发中最需要考虑的问题。

存储器的一页为 256 字节，与最小模式下 8051 的内部数据空间（IDATA 空间）大小相同。因此，我们认为页变量的 256 字节能满足最小模式任务的需要。

对于需求超过 256 字节的任务，可在页面外的外部存储器中申请后备空间。只要指向后备空间的指针在页面内，则该后备空间仍是私有的，满足重入条件。其结构如图 3-6。虽然后备存取区使用指针操作，但是不需要计算变量地址。因此后备存储区存取速度优于重入堆栈。

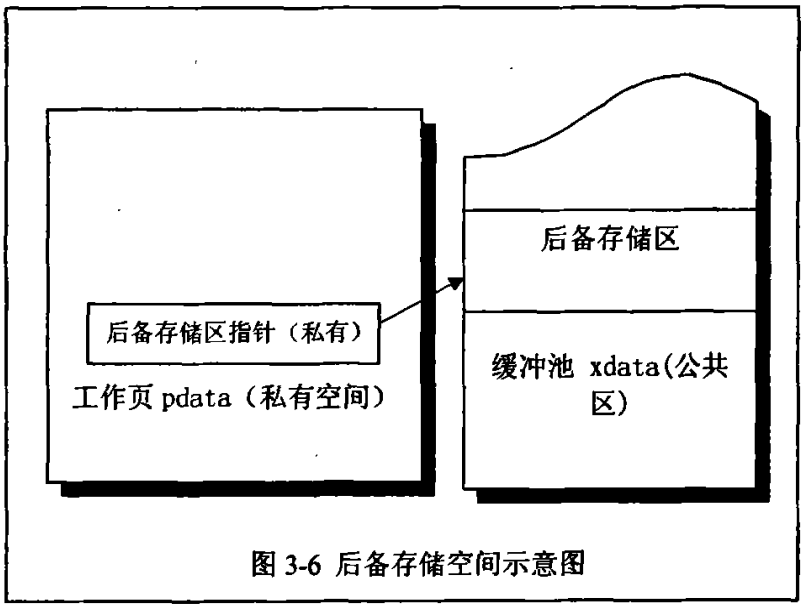

图 3-6 后备存储空间示意图

(2) 工具的限制

目前，Keil C51 开发工具不提供多任务以及多页面支持。这主要体现在两点：

①需要新的函数库。现有的大量函数库不支持页函数方式重入。准确地说，大部分函数库不支持任何方式的重入——即使使用重入堆栈，函数库问题也依然存在。目前的解决方法只有程序员根据需要编写新的页函数库。

②一个项目中只允许存在一个 pdata 页面，不能按任务将变量安排在不同的逻辑页面上。其后果是，多任务开发中，编译链接工具只在同一个页面中分配所有任务的页变量，导致存储器迅速溢出。

问题 2 的解决方案是：为使每个任务具有自己的工作页，我们为每个任务单独建立工程，并使用 COMPACT 方式进行编译。各个任务工程之间和操作系统之间使用绝对地址表传递系统调用和任务入口地址。绝对地址表是我们对一些系统调用的约定地址。

这些不便之处是暂时的、可克服的。

3.1.4 小结

8051主要应用领域为实时控制，因此努力提高系统实时性是开发者不断追求的目标。我们从提高实时性的角度出发，提出了一种基于页的多任务模型。相对于目前重入堆栈多任务模型，它具有较好的实时性，也是处理重入问题的一种新思路。

本模型已经过Keil C51仿真工具的一般性测试，并在W78P438芯片上实测成功，相信其结果适用于全部8051及兼容系列。考虑到8051依然广泛应用在多任务开发中，本文中的新模型值得推广。

3.2 中断系统设计

中断系统的设计，硬件方面需要考虑中断的触发模式、系统中断堆栈的分配以及生长方向，软件方面则关系到整个软件系统中中断的管理与设计。因为8051中断堆栈与普通程序调用所使用的堆栈同在内部存储器中，所以在硬件上，中断堆栈设计没有选择的余地。与其他计算机系统一样，在中断问题上，嵌入式系统中软件质量的优劣对系统实时性所产生的影响是主要方面。

为提高系统实时性，我们可以从缩短中断响应时间和提高中断服务程序的可预测性两个方面加以改进。

3.2.1 中断响应时间与临界区

中断响应时间是指中断从发生到执行中断程序之间的时间间隔。中断响应时间值越小，则中断快速响应性能越好。中断快速响应是评价实时系统性能的重要指标，它一般由硬件和软件共同决定。

3.2.1.1 操作系统对中断响应时间的影响

影响中断响应时间的一个因素是中断上下文切换。对于不同硬件体系结构，操作系统必须保证中断处理程序运行在合适的上下文中。在嵌入式操作系统中，由于硬件体系结构的不同，当中断上下文建立在任务环境之上时，响应速度比较快，但是却会对用户堆栈大小造成影响。而建立单独的中断上下文会对中断响应时间有一定的影响。

影响中断响应时间的另一个因素是关中断，这也是我们将要解决的主要问题。我们都知道操作系统中有临界区问题。为了避免由于多个任务对共享资源同时访问，从而造成数据的不一致性，需要用信号量机制来加以避免。对于中断来说，这个问题同样存在。当中断服务程序和任务的系统调用对同一个数据进行操作时，我们就说发生了碰撞。

```
void   OS_Sched (void)
{
    INT8U       y;
    OS_ENTER_CRITICAL();        //关中断，进入临界区
    if ((OSIntNesting == 0) && (OSLockNesting == 0)) { /* Sched. only if all ISRs
done & not locked      */
        y = OSUnMapTbl[OSRdyGrp];   /*Get pointer to HPT ready to run */
        OSPrioHighRdy = (INT8U)((y << 3) + OSUnMapTbl[OSRdyTbl[y]]);
        if (OSPrioHighRdy != OSPrioCur) {
            /* No Ctx Sw if current task is highest rdy */
            OSTCBHighRdy = OSTCBPrioTbl[OSPrioHighRdy];
            OSCtxSwCtr++;               /* Increment context switch counter   */
            OS_TASK_SW();               /* Perform a context switch */
        }
    }
    OS_EXIT_CRITICAL();         //开中断，退出临界区
}
```

图 3-7 μC/OS-II 的临界区处理

在可见源码的传统操作系统中，对于操作系统的内部调用，解决中断干扰的处理方法通常是关中断。例如，在 μC/OS-II 中，其任务调度需要在关闭中断的情况下进行，如图 3-7 所示：

图 3-7 中，关键数据结构就是系统任务列表（OSTCBPrioTbl[]）及相关公共指针(OSTCBHighRdy,OSTCBCurrPrio,...)。我们知道 μC/OS-II 是一个抢占式的操作系统。当任务正在执行“任务调度”这一系统调用时，这时系统使用公共指针扫描任务列表，以得到下一个执行的任务（TCB）。该过程中，若有中断发生且中断中有新的任务转为就绪状态时，中断服务程序就必须通过公共指针修改系统任务列表。这时便发生了碰撞。而 μC/OS-II 的做法就是在操作系统任务列表的时候禁止中断服务程序的执行，也就是我们所看到的关中断方式。

在这段代码中，OS_ ENTER_CRITICAL()这个函数就是 μC/OS-II 用来进入临界区保护的 P 操作。而这个函数的实现，在 μC/OS-II 中，只是简单地关闭了中断。这样一来，整个操作系统的任务调度工作就全部都在关中断的情况下进行。这样做的好处是可以简化程序设计，使得程序代码量和复杂性降低，但是另一方面，却造成了中断的响应时间有可能被无限期地延长。如果系统中的任务数量很多，而操作系统在相关队列中插入一个任务而需要进行整个队列扫描时，中断延时时间就有可能不是由用户能够控制的了。

为了减少中断响应时间，以保证系统设计的确定性，我们需要有一种机制，能够保证即不破坏操作系统的关键数据的一致性，又能够使得操作系统在进行关

键数据操作时，开放中断资源，使得中断能够获得尽可能快的响应时间，从而提高系统的性能。

由于中断均是由外部事件触发的，因此，CPU 在运行过程中必须开放中断控制位才能保证中断的响应。而 CPU 一般均是在当前运行的指令结束后才对中断信号进行响应，通过纯硬件的方法，来保存一些必须的环境变量，然后跳转到所指定的中断向量中去，因此，硬件体系结构的不同，也会造成中断响应时间的不同。

在可见源码的传统操作系统中，对中断干扰的处理方法是关闭中断。如在 Linux 中，当内核态进程需要访问关键数据结构时，就直接关闭中断来实现，由于 Linux 是单内核结构的操作系统，其内核功能非常强大，因此也造成了内核中多处存在着关内核运行从而保证内核与中断的互斥操作，可是，这也造成了 Linux 中断响应时间在最坏情况下会达到 100ms 左右。对 Linux 进行实时性能改造的首要工作就是缩短其中断响应时间，这是嵌入式实时操作系统重要的工作之一。我们看到，在 μC/OS-II [J.L. Jean2001]实时嵌入式操作系统中，同样采用关闭中断的方法来保证中断响应。

我们将设计一套新的中断处理机制，主要目标是避免使用关闭中断的方法处理临界区问题，并以此提高系统的中断响应时间。

3.2.1.2 中断延时队列

3.2.1.2.1 原理

解决问题的关键在于分离对核心数据的读写操作。不论是中断服务程序中，还是在一般系统调用中，对核心数据操作都是在系统态完成的。

我们发现 μC/OS 是支持中断嵌套的。μC/OS-II 中[34]，中断服务程序将全部 CPU 寄存器推入当前任务栈。μC/OS-II 允许中断嵌套，由中断嵌套计数器 OSIntNesting 跟踪嵌套层数。

处理完中断服务后，μC/OS-II 调用函数 OSIntExit()通知内核，到了返回任务级代码的时候了，于是 OSIntExit()将 OSIntNesting 减一。当 OSIntNesting 为零时，μC/OS-II 判定有没有优先级较高的任务进入了就绪态，若有，μC/OS-II 就返回到那个高优先级的任务，如果调度被禁止了(OSIntNesting>0)，μC/OS-II 将被返回到被中断了的任务。

可以看到高优先级中断抢占低优先级中断处理器的过程和中断抢占系统程序的过程是相似的，但却没有使用关中断的方法。这种处理方法的实质是通过 OSIntNesting 记录下系统运行的状态，同时将实质性的操作延迟到最低优先级中

断结束的时刻执行。受这种中断嵌套方式的启发，当我们将中断过程可看作一种特殊的系统调用过程时，则可以将中断引起的核心数据操作延迟到系统调用结束的时刻完成。既然中断这种特殊的系统调用可以以延迟的方式实现嵌套，那么任务级调度这一最低级别的系统调用可以以同样方式实现嵌套。如图 3-8 所示改进前的中断嵌套中已有关键数据结构的临界问题。这说明了不使用关闭中断进行核心数据操作的可行性。

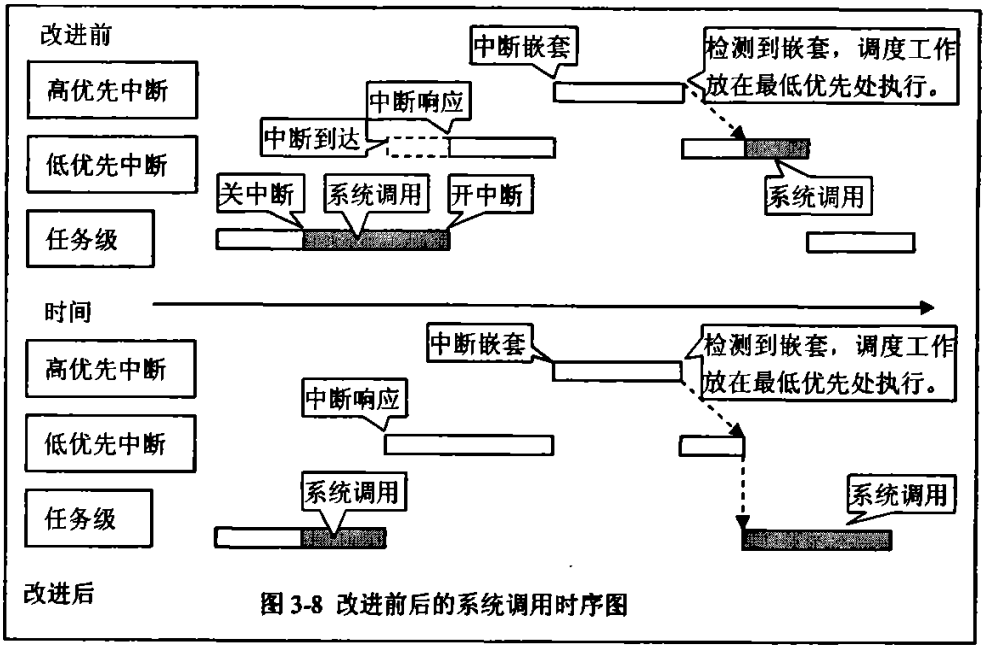

图 3-8 改进前后的系统调用时序图

根据这一思路，若中断发生在系统调用时间外，则中断结束时正常执行调度。若中断发生在系统调用的时间内，则使用 OSTCBDlyList 来记录中断中新就绪的任务 TCB，将原来在中断结束时的系统调用延迟到任务级的系统调用中。这一方法与[陈晗斐]中断延时队列 workQ 的思路完全一致。可以在不关闭中断的情况下对核心数据进行操作。从图 3-8 中可以看到，低优先中断的响应时间提前了。这样做并不会影响中断的实时性，因为事件处理已经在中断中完成，只是处理结果尚未及时提交。提交的时刻正是在系统调用的返回时刻，如图 3-8 所示。

3.2.1.2.2 具体实现

如图 3-9 和图 3-10，我们以流程图的方式给出本文中中断延时队列的具体实现过程。其效果见前文图 3-8。更详细的理论说明请见参考文献[5]，陈晗斐《实时操作系统的若干关键问题研究》的第 44~第 52 页。

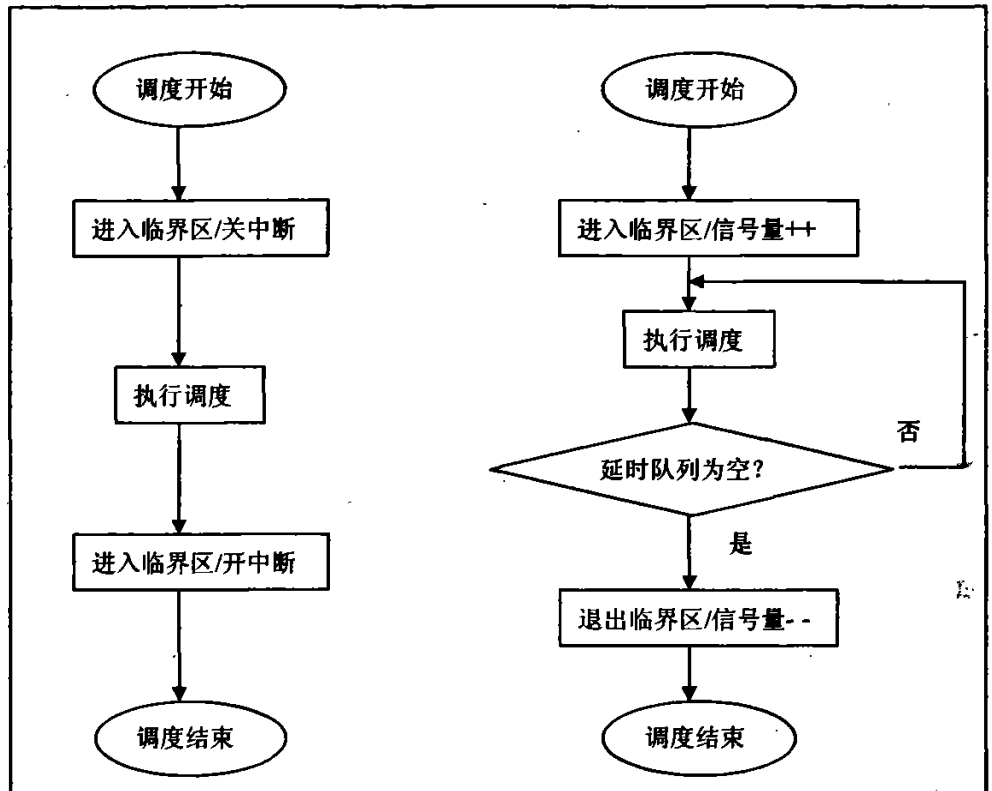

图 3-9 修改前后的（任务级）调度程序

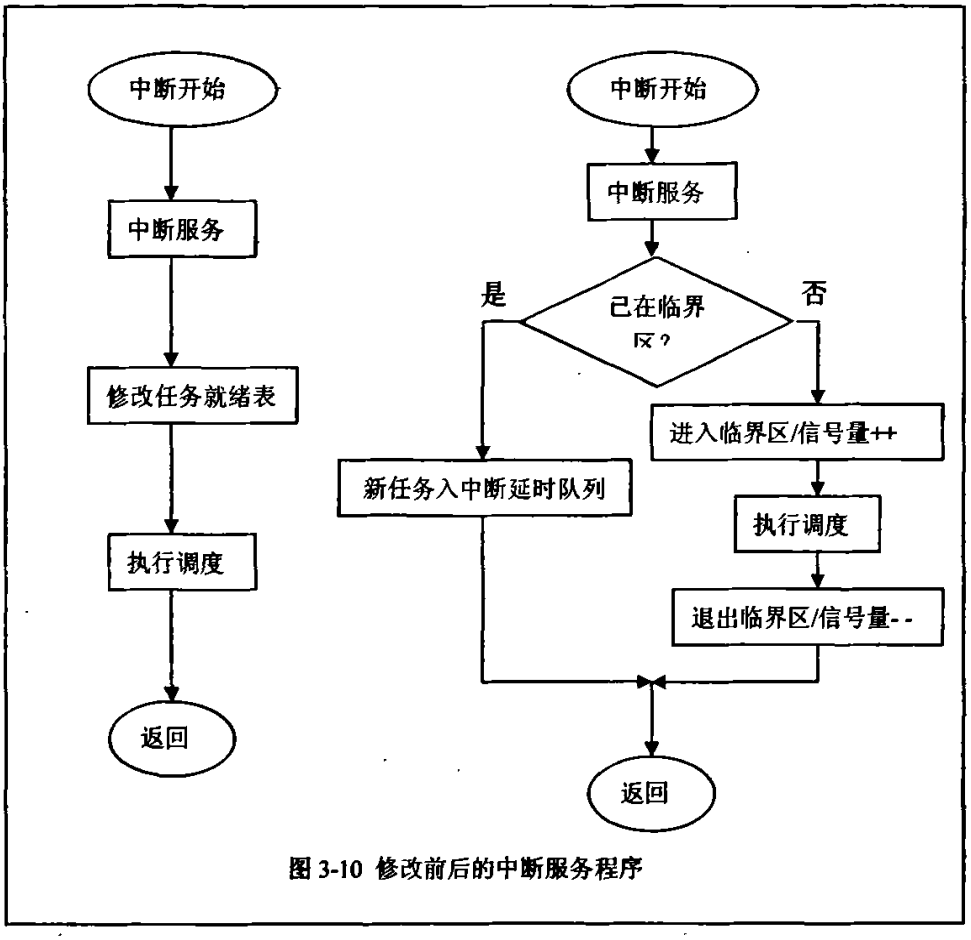

图 3-10 修改前后的中断服务程序

3.2.1.2.3 关键技术介绍

(1)临界区进出操作必须注意原子性

一般情况下，判断系统程序的工作状态需要三种涉及临界区的操作，即进入临界区、退出临界区与判断是否在临界区。关中断处理临界区问题的方案，系统只需要两个操作，即进入临界区“OS_ENTER_CRITICAL()”和退出临界区“OS_EXIT_CRITICAL()”。这两个操作是分别只执行一条指令实现关中断和开中断。这种方式下，关闭中断使程序不再出现无法预测的分支，则判断是否在临界区的操作就不需要了。

使用中断延时队列，则上述三种临界区操作都是必需的。与处理中断嵌套的方法一样，我们使用一个字节变量 wos_crit_count 记录临界状态。wos_crit_count 只有 256 个值。用 0 值表示非临界状态，则 wos_crit_count 只能记录 255 个临界嵌套。使用单字节变量记录临界嵌套是必须的，因为 8051 是 8 位微控制器，只有单字节变量的操作才能在不关中断的条件下具有原子性。

(2)分离核心数据的写同步

使用中断延时队列可以合理地分离核心数据的写操作。我们使用一个链表记录中断中发生的事件，比如在中断中唤醒的任务号。作为生产/消费者问题，新产生的事件被放入该链表内。中断延时队列工作于先进先出方式。只有生产者(中断)能修改队列内容，并维护队列的写指针。消费者(中断外的系统调用)只能读取待处理的事件，而这种操作只需要修改读指针就可以完成。

一般的非中断方式的系统调用允许在进入临界区后直接修改核心数据。该调用不需要判断是否已在临界状态，但修改核心数据前后需要标记进入和退出临界状态。

中断服务程序则不同。[陈晗斐]当中断服务程序只是访问的数据并没有被操作系统内核占用时，中断服务程序可以顺利地执行下去。但是当中断服务程序需要访问被内核占用的数据时，如系统队列，则中断服务程序向 OSTCBDlyList，也就是中断延时处理队列投递一个消息，然后就顺利地退出中断服务程序。由于操作系统只有在处于内核态时才会占用关键数据结构，因此，当操作系统完成对关键数据结构的操作，将要退出内核之前，首先会扫描中断延时队列。如果队列是有未完成的任务，则将所有任务完成之后再进行必要的调度工作，然后退出内核态。通过这种方式，操作系统不需要在关中断的方式下就可以完成关键数据的操作，从而大大地减少了关闭中断的时间，从而提高了中断响应时间。

(3)调度的时刻

即依据中断延时队列更新核心数据的时刻。我们选择在任务级系统调用结束的时刻完成这一操作，这正是任务级系统调用结束的第一时刻。

3.2.1.2.4 小结

中断延时队列是建立在系统内核之中的一个消息队列。通过对操作系统内核程序的合理设计与安排，使中断程序中对关键数据结构的处理均在操作系统退出内核态之前得到处理。因此从理论上来讲操作系统是不需要关中断运行的，从而将操作系统对中断响应时间的影响降低到了最低。采用中断延时队列的方法，在理论上来说可以，使得内核可以在不关中断的情况下进行内核操作，从而使得中断延时时间仅由硬件来决定，这对于实时系统来说是一个很大的进步。从中断服务程序的角度看，对核心数据的操作，由于临界区问题而被延时了。但该方案利用了系统运行状态[35]的区别，使延迟的时间最小。

以上我们从中断响应时间的角度对核心数据的关中断处理方式进行了改进。接下来我们从改善程序可预测性的角度来提高中断系统的性能。

3.2.2 可预测性与定时器管理程序

前面已经提到，实时操作系统除了要满足应用的功能需求以外，更重要的是还要满足应用提出的实时性要求。与通用操作系统不同，实时操作系统注重的不是系统的平均表现，而是要求每个实时任务在最坏情况下都要满足其实时性要求。而实时操作系统所遵循的最重要的一条设计原则就是：采用各种算法和策略，始终保证系统行为的可预测性(predictability)，其中就包括时间可预测性。

3.2.2.1 可预测性与实时系统

时间可预测性是指时间确定性，它要求在任意上下文环境中完成给定计算所需要的时间一致[36]。对于实时系统来说，设计者必须事先知道所有实时任务的运行时间，从而能够计算出实时任务是否能够满足它们的死限要求(要求可预测性)。

可预测性是命令与控制系统的一个重要课题。它要求实时系统具备能对其性能进行评估的能力[37]，是实时性的一项重要性能指标。在实时系统中，可预测性是如此的重要，以至于为了满足它，甚至需要作出速度上的让步。例如，处理器缓存技术使得实时任务的运行时间变得不可预期，从而可能造成严重的后果。因此在实时系统中从设计者的观点来看，处理器缓存所带来的负面影响可能会比它所带来的好处要大。

3.2.2.2 定时器及背景概述

在计算机系统中，类似于“屏幕保护”这种定时触发的活动数不胜数。这些都归功于定时器，它使系统内核能记录下时间轨迹。定时器系统包括硬件定时器和软件定时器。本文讨论的是软件定时器管理的改进算法。软件定时器就是一个软件工具，它允许在将来某个时刻，当给定时间间隔用完时调用函数。

为实现定时器的精确触发，其管理操作通常需要在时钟中断中完成。但是随着进程的变迁，系统需要维护的定时器数量变化很大。这种随机变化使传统的中断服务程序的定时器遍历时间难以预测。因此实时系统不宜直接使用一般操作系统的定时器管理算法。我们首先需要介绍一下时钟节拍这种特殊的中断。

时钟节拍是特定的周期性中断。每个时钟节拍到来的时候，操作系统利用相应的时钟节拍中断服务程序维持系统时间，完成记帐，监督系统运行情况，以及完成各任务的调度。

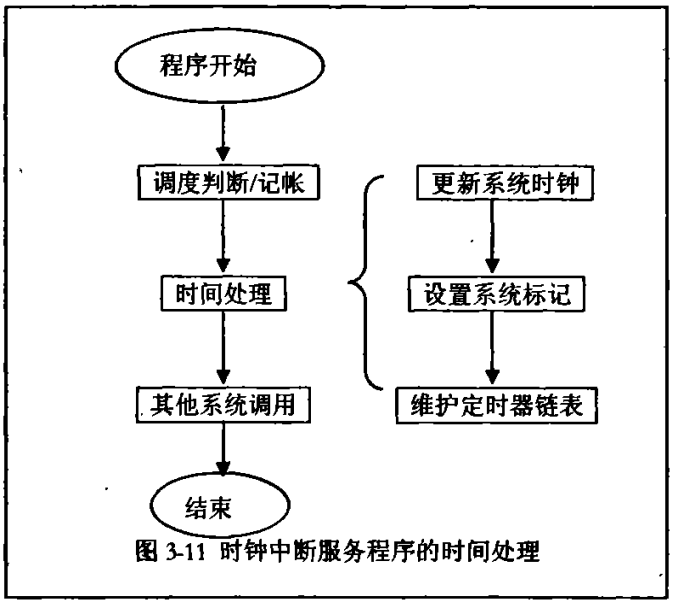

图 3-11 时钟中断服务程序的时间处理

如图 3-11 所示，时间处理包括更新系统时间，实质性的中断任务以及定时器的维护操作。定时器维护(timer management)是指操作系统依次检查定时器链表中的触发值，并判断该定时器是否被触发的一系列软件过程。

```
void   OSTimeTick (void){
     OSTime++;
     ptcb = OSTCBList;
     while  (ptcb->OSTCBPrio  !=  OS_TASK_IDLE_PRIO)
{… …
     if (ptcb->OSTCBDly != 0) {
     --ptcb->OSTCBDly; … … }
}                          //       μ C/OS-II "os_core.c"

static inline void run_old_timers(void){          … …
     for (mask = 1, tp = timer_table+0 ; mask ; tp++,mask +=
mask) {    … …
          if (tp->expires > jiffies)
               continue;
          tp->fn(); … …
     }
}                               //       μ clinux   "sched.c"
```

图 3-12 两种嵌入式操作系统的定时器遍历程序[算法 3.2.1]

图 3-12[算法 3.2.1]列出了嵌入式操作 μ C/OS-II 和 μ clinux 维护定时器的核心代码。下划线标注的循环语句显示，这两种操作系统在维护时所使用的算法都是遍历整个链表。若链表有 n 个定时器，维护一次的时间复杂度为 O(n)。

我们看到为保证定时器精确触发，其维护程序与系统时间中断密切相关，属于中断级代码，具有极高的优先级。

一般情况下，操作系统使用时间复杂度为 O(n)的中断服务程序是可以接受的。例如，[Daniel P. Bovet]就认为维护一个排序的链表效率也不高，并且插入与删除也相当费时，从而依然使用遍历方式维护静态定时器。

但是在实时操作系统中，值得对这种中断服务程序加以研究改进，缩短最坏情况下时钟中断的执行时间，提高中断执行的可预测性。

3.2.2.3 改进方案

由于时钟节拍的运行频率非常高，且属于中断级任务，因此有许多的研究工作都致力于减少时钟节拍的运行时间，即降低该任务的时间复杂度。以下是常见方案。

① "下半区函数(bottom half)" 法

在 μclinux 中，下半区函数是一个低优先级函数，通常与中断处理相关。确切地说，内核将在一个方便的时刻运行下半区函数[38]。

因为维护定时器链表总是由下半区函数完成，而下半区函数被激活后很长时

间才能被执行，即内核不能确保定时器正好在定时到期开始执行，而只能保证在适当的时间执行它们，或者假定延迟到几百毫秒之后执行它们[38]。所以对于必须严格遵守时间的那些实时应用，“下半区函数”法不适合。

②预测法

[屠征]提出了一种对嵌入式时钟中断服务程序的改进方案。方案的核心是使用最早触发的定时器值来设置时钟，该方案通过这种预测触发时间的方法，避免不必要的时钟中断，从而降低了系统开销。其原理类似硬件上的 oneshot 定时器。oneshot 模式就是每次都把时钟芯片设置为下一个事件要发生的时间的工作模式。

事实上，预测法牺牲了时钟节拍。文章在最后说明了使用此方案将无法精确获得系统时间。显然这种损失在实时系统中是无法接受的。

③delta 链[12][40][41]

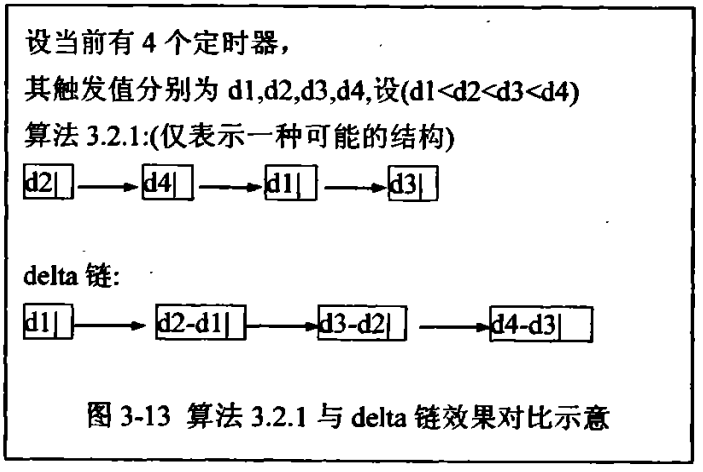

图 3-13 算法 3.2.1 与 delta 链效果对比示意

利用操作系统中定时器双链表结构，可将定时器按触发值从小到大排序，得到一个定时器有序链表。对该链表做进一步处理：除了头单元，修改其余各定时器值为该单元与其直接前驱的触发值之差，如图 3-13 所示。这种调整触发值后的有序链表即 delta 链。图 3-13 为 delta 链与算法 3.2.1 的链表结构对比示意。

这里需要对图 3-13 进一步说明。算法 3.2.1 产生的链表，其单元排列顺序与各定时器在系统中产生顺序有关，因此图 3-13 只表示了算法 3.2.1 的一种可能链表结构。Delta链中各定时器在链表中的顺序只与其触发值的大小关系有关，而与定时器产生顺序无关。

```
t 为待插入元素，t_list 为链表指针，程序开始时指向链表中
的表头元素。
while(t->dly>=t_list->dly){
    t->dly-=t_list->dly;
    t_list=t_list->next;
    ...
}  //插入定时器队列 其执行时间代价计入 L

void  OSTimeTick (void){
    OSTime++;    tcba=OSTCBDlyList;
    if(tcba!=0){ ...
        if(--tcba->dly==0){... ...}
    }
}// (中断)维护定时器，其执行时间计入 I
```

图 3-14 delta 链算法的核心程序

图 3-14 为实现 delta 链的核心代码。链表插入算法是其中最关键部分，详细描述如下：若定时器链表为空，则插入单元直接放入链表。若定时器链表为非空，则：

①初始化，链表表头单元为当前单元；

②若待插入值 < 当前单元，执行③。否则：待插入值 -= 当前单元，选择链表下一单元为当前单元。重复②；

③当前单元 -= 待插入值，待插入值成为当前单元直接前驱。程序退出。

有序链表使定时器管理相对算法 3.2.1 更简单。只需要对定时器链表头单元进行减法和判断便完成一次维护。

Delta 链的时间开销分为两部分。一是将新产生的定时器插入链表所需计算开销，记为 L；另一部分是维护链表所需要的计算开销，在中断中执行，记为 I。若将算法的全部开销记为 T，则 T = L + I。

3.2.2.4 delta 链的性能分析

3.2.2.4.1 复杂度分析

delta 链的链表结构与算法 3.2.1 同样，即空间复杂度相同。以下对比这两者时间的复杂度。

算法 3.2.1(见图 3-12)的基本操作为：①触发值减 1 的时间代价 ts；②判断触发值是否为零的时间代价 tc。

相应地，delta 链的基本操作为：①.触发值比较的时间代价 tc'；②触发值相

减的时间代价 ts'。

触发值为无符号数。对于无符号数，我们认为算法 3.2.1 与 delta 链在基本操作上的时间代价差别足够小，即 tc = tc', ts = ts'。并设基本操作的时间代价 ts + tc 为 1。

3.2.2.4.2 时间复杂度的比较与证明

设系统有 n 个定时器需要维护，延迟时间分别为 $d_1 \ldots d_n$。产生的 delta 链可记为 $Q\{q_1,\ldots,q_n\}$。由定时器的定义可得（3.2.1）式。

$d_i \geq 1 (i=1..n)$ (3.2.1)

这里提出假设条件“设 $d_1..d_n$ 两两互不相等”，即(3.2.2)。根据这一假设可得(3.2.3)式。下一节将取消这个假设，并推广到一般情况讨论。

$\forall di \neq \forall dj (i \neq j; i, j \in (1..n))$ (3.2.2)

$q_i \geq 1 (i=1..n)$ (3.2.3)

设算法 3.2.1 的总时间复杂度为 T'，可直接看 T'出满足(3.2.4)式。

$$T'(n) = \sum_{i=1}^{n} di \tag{3.2.4}$$

设 delta 链方式的总时间复杂度为 T。根据 3.2.2.2 节的分析，有 T(n) = L(n) + I(n)。delta 链需要一直管理定时器链表直到最后一个定时器单元触发，因此 I 的时间代价就是当前链表各单元值的总合 Σq，也就是 $\max\{d_1,\ldots,d_n\}$。得(3.2.5)式。

$$I(n) = \sum_{i=1}^{n} q_i \tag{3.2.5}$$

我们通过数学归纳法证明 delta 链方式优于算法 3.2.1，即证(3.2.6)式成立。

$T(n) \leqslant T'(n)$ (3.2.6)

证明：

1.基础步骤：

①当 n = 1 时，待插入单元为 d_1。

$Q(1) = \{q_1\}$;

$L(1) = 0$; $I(1) = q_1 = d_1$;

$T(1) = d_1 = T'(1)$; (3.2.6)式成立;

②当 n = 2 时，待插入单元为 d_2。

若 $d_2 > d_1$ 则 $Q(2) = \{d_1, d_2 - d_1\}$，反之 $Q(2) = \{d_2, d_1 - d_2\}$。

比较和减法各执行一次，因此 $L(2) = L(1) + 1 = 1$。$I(2) = \Sigma q = \max\{d_1,d_2\}$。

$T(2) = L(2) + I(2) = 1 + \max\{d_1, d_2\}$;

$T'(2) = T'(1) + d_2 = d_1 + d_2$;

依据(3.2.1)可知 $T(2) \leqslant T'(2)$;

(3.2.6)式成立;

2.归纳步骤：

设 $n = k$ 时，$T(k) \leqslant T'(k)$成立。

设此时定时器队列为 $Q\{q_1,q_2,\ldots,q_k\}$; 不妨设 $d_i=\min\{d_1,\ldots,d_k\}$, $d_j=\max\{d_1,\ldots,d_k\}$;可得, $d_i = q_1$; $d_j = \Sigma q$;

当 $n = k+1$ 时，待插入单元为 d_{k+1}，分以下三种情况讨论。

①$d_{k+1} < d_i$

$Q(k+1) = \{d_{k+1}, q_1 - d_{k+1}, \cdots, q_k\}$

∵只执行了 1 次比较和减法操作。

$L(k+1) = L(k) + 1$ 且 $I(k+1) = I(k)$;

$\Rightarrow T(k+1) = T(k) + 1$

$T'(k+1) = T'(k) + d_{k+1}$;

∵归纳法假设条件 $T(k) \leqslant T'(k)$成立

且根据(3.2.1) $d_{k+1} \geqslant 1$ 成立

∴$T(k+1) \leqslant T'(k)$ (3.2.6)式成立

②$d_{k+1} > dj$

$Q(k+1) = \{q_1, \cdots, q_k, d_k-q_k-\cdots-q_1\}$

共执行了 k 次比较和减法操作。建立队列的时间开销为 k，中断中增加的时间开销为（$d_k - \Sigma q$）。

$T(k+1) = T(k) + k - (d_{k+1} - \Sigma q)$

$\Rightarrow T(k+1) = [T(k) + d_{k+1}] + (k - \Sigma q)$

$\Rightarrow T(k+1) = T'(k+1) + (k - \Sigma q)$

∵$\forall q \geqslant 1$ ∴$\Sigma q \geqslant k \Rightarrow k - \Sigma q \leqslant 0$;

∴$T(k+1) \leqslant T'(k+1)$, (3.2.6)式成立;

③$d_i < d_{k+1} < d_j$

设 $Q(k+1) = \{q_1, \cdots, q_m, d_{k+1}-q_1-\cdots-q_m, q_{m+1}, \cdots, q_k\}$

Q（k+1）表示插入位置为 qm 单元之后，则该插入算法只需要执行 m 次比较和减法，可知 $d_{k+1} > (q_1+q_2+\cdots+q_m)$。

∵L(k+1) = L(k)+m, I(k+1) = I(k)

∴T(k+1) = L(k+1) + I(k+1) = T(k) +m;

=>T(k+1) = T'(k) + m;

由 d_{k+1} > (q_1+q_2+…+q_m) 且依据(3.2.2)可得: d_{k+1} < m

∴T(k+1) = T'(k) + m ≤ T'(k+1),

(3.2.6)式成立;

3. 结论

根据基础步骤和归纳步骤的证明，可知对于任意自然数 n，(3.2.6)式都成立，delta 链方式时间复杂度优于算法 3.2.1。证毕。

3.2.2.4.3 最坏情况

在 delta 链的插入程序中，(3.2.2)式保证了每次与链表单元的比较和减法具有“有效性”。“有效性”是指该减法所消耗的时间代价从定时器维护角度看是必要的，因为它是“触发值减小到零”和“比较判断”所必须的。

只有当以下两个条件同时出现时，delta 链的时间代价与算法 3.2.1 相等，达到最坏情况。

①定时器产生的顺序，按触发值看正好为从小到大。这是整体上的最坏情况：每次要插入链表的定时器为当前最大值，插入算法需要遍历整个链表。

②定时器触发值(自然数)，正好为其在链表中的位置号。这是局部上的最坏情况：插入过程中，每次比较执行的减法为 1。

只要①②中有一个不成立，则 delta 链的时间开销就小于算法 3.2.1。

3.2.2.4.4 delta 链的特殊优点

在实时系统中，delta 链有两个特殊优点：①.中断中定时器维护只需处理链表头单元，最坏复杂度仅为 O(1)；②定时器插入链表的开销 L 与中断服务程序无关。

优点①使中断的时间代价可预测，它将管理定时器的大部分运算转移到了中断服务程序之外。通过链表排序，只需要判断 delta 链的最小触发值，便能保证各定时器的精度触发，中断服务程序的最坏时间复杂度保持为 O(1)，比算法 3.2.1 的 O(n)有较大提高。

3.2.2.5 算法实现与进一步讨论

μC/OS-II 是一个源码公开、符合算法要求的实时操作系统。图 3-14 是在该系统上实现 delta 链的两段关键程序。从程序中可以看到，相对于图 3-12 算法 3.2.1，图 3-14 中的 OSTimeTick 的时间复杂度就是 O(1)。

进一步分析可知，实际应用中，假设条件(3.2.2)式是特例。一般情况下各定时器触发值为任意，可以相等。

取消(3.2.2)式将导致(3.2.3)式不成立。因此，当 qi = 0 为比较值时，delta 链算法中执行的“减零”运算没有使触发值变小，该操作不符合“最坏情况”里所提出的“有效性”，属于额外开销。

qi = 0 表示当前有多个定时器在同一时刻触发。依据[Daniel P. Bovet]：“大部分设备驱动程序利用定时器检测反常情况”，这表示定时器具有较强的随机性。因此大量定时器在同一时刻触发的概率较小，qi ≠ 0 在链表中占多数。则 delta 链算法中“非零值”的减法是主要的。又考虑到“减零运算”这种额外开销是在中断外进行的，算法对实时系统中断性能的改善是主要方面，因此值得在实时系统中应用。

3.2.2.6 小结

定时器是操作系统中的常用软件设备，也是其时间系统的重要组成部分。为保证定时器精确触发，操作系统需要在时钟中断维护定时器。本文介绍的改进算法将定时器的维护分成两个部分，使大量数值计算在非中断过程中完成，从而将中断服务程序中定时器维护工作降为常数级 O(1)，提高了中断服务程序的可预测性。

我们在 μC/OS-II 上实现了该算法，并列出了关键代码。最后，在实际情况下也可能出现总时间代价比算法 3.2.1 更差的情况，但额外开销不属于中断过程。改进后中断服务程序在维护定时器所需要的时间始终保持 O(1)。

第4章　总结与展望

实时操作系统研究是一个古老的课题，但在 8051 系列上实现多任务却是近年才出现的研究热点。我们从硬件结构和软件算法上进行了实时性的研究。

8051 的体系结构在实现多任务方面存在许多限制。其中最主要的问题是 8051 堆栈系统只能设置在内部 RAM 中，容量太小。在软件平台上，虽然 C51 对传统单任务开发来说是功能强大的开发工具，但是因为嵌入式应用的特殊性，C51 编译中专门针对 8051 体系结构的优化功能反而限制了 C 语言在多任务开发中的程序效率。

本文的最大创新在于突破了开发工具的拘囿，提出一种新的基于页的多任务开发模型。该模型屏弃了传统 8051 多任务的重入堆栈的存储器模型，这使得本文有别于以往 8051 上移植操作系统的工作。另一方面，建立在该模型上的操作系统基本使用 C 语言编写，又保持了通用操作系统 μC/OS-II 的易读性与通用性。同时，它能在现有 C51 平台上较好地处理 8051 重入与效率的矛盾，又具有更好的实时性。本文的设计达到了预期目标。

在算法设计上，我们主要研究了中断的处理。通过使用中断延时队列分离了对关键数据的读写，从而避免了关中断的操作，提高了中断响应速度和系统实时性。我们还引用动态排序的 delta 链算法对定时器管理进行了改进，提高了时钟节拍这种特殊的中断服务程序的可预测性。

总结成果，展望未来。一个操作系统的设计是很复杂的，往往需要花费大量的人力与物力，经过多年的知识积累与研究才能设计出一个性能优越的操作系统。无论是 UNIX、WINDOWS，还是现在的 LINUX，它们的开发过程都可以证明这一点。而实时嵌入式操作系统虽然从代码量上来说并没有这些通用操作系统大，但是由于它牵涉到系统的可靠性，牵涉到人生与财产的安全，因此它的设计与实现更是一个复杂的过程，需要对实时理论以及系统运行特点有深入的了解。因此一个成熟可靠的实时嵌入式操作系统一定需要经过众多应用的考验并逐步完善，需要集中许多行业与软件专家的知识与经验。

基于 8051 的实时多任务开发也是如此。我们已经完成了任务调度和中断处理，但是要完善一个实时操作系统还需要取得内存管理和文件管理的支持。在当前情况下，对分页模式建立一个完善的函数库则是重中之重。这些工作都需要一个较长的设计和验证过程。

从小分页走向大分页，从分页变量走向分页函数。在新型微控制器 DS80C400 上我们看到，8051 微控制器已经实现了程序分页，设计了更大的系统堆栈空间，并且可以工作于“联合模式（Combined Program/Data Memory Access）[45]”，以实现程序的动态载入。在 DS80C400 中，其 16MB 存储器空间可按 64KB 的空间

分页，是大分页模式。这使得基于页的多任务开发将不在受到页变量 256 字节的限制，64KB 的私有页空间甚至可以满足巨型模式的要求。这种存储器结构更适合传统 8051 单任务开发模式的存储器结构。并且其程序空间也能分页，这使得位于不同程序页的程序相互独立。与变量分页一样，程序分页独占一个连续的 64K 程序空间，具备更清晰的程序结构和更好的安全性和数据私密性。可以看出 8051 多任务开发可以从目前变量分页走向程序分页。届时多任务开发会更方便，任务也可以象桌面系统一样，只在被执行时才调入存储器中，从而进一步提高嵌入式系统中有限存储器的利用率。因此，值得对基于 8051 的实时操作系统及分页技术做进一步的研究。

参考文献

[1]周晓中 基于单片机的实时内核设计及其应用研究 硕士论文 中南大学 2005 年 4 月: 1,22,29

[2]Ramtron. Versa 8051 MCU VRS51L3074. Datasheet Rev. 1.4: 1.

[3]王田苗 嵌入式系统设计与实例开发 第二版 北京:清华大学出版社，2003. 10

[4]黄磊 单片机与嵌入式系统开发平台化的研究 硕士论文 南京航空航天大学 2004 年 2 月: 3

[5]陈晗斐 实时操作系统的若干关键问题研究 浙江大学博士论文 2004.7: 39~41,44~52,56,57

[6] Ramamritham K, Stankovic J A. Scheduling Algrithms and Operating Systems Support for Real-time Systems. Engineering Journal, 1994. 83(1): 55-67

[7]朱瑾. 基于 Linux 嵌入式操作系统实时性的研究. 硕士学位论文 沈阳工业大学. 2005.3: 8

[8]齐俊生 嵌入式 Linux 硬实时性的研究与实现 硕士论文 西安理工大学 2003.3: 8

[9]刘波,马连川,张建明 嵌入式实时操作系统选用的初步分析 北方交通大学学报 2000 年 10 月 第 25 卷第 5 期: 106-107

[10]John A.Stankovic, et al. Strategic Directions in Real-Time and Embedded Systems ACM Computing Surveys, Vol. 28, No. 4, December 1996: 751

[11]王宏智 基于嵌入式系统的 Internet 接入技术的研究 硕士论文 大连海事大学 2006 年 3 月: 4-5

[12] A. Tanenbaum. Modern Operating System.北京:机械工业出版社，2002: 2,257-304

[13] 李江 常葆林 嵌入式操作系统设计中的若干问题 计算机工程2000.6 第20卷第6期: 89

[14] 刘从新 曾维鲁 51 单片机实时操作系统的构建 三峡大学学报 2003.10 第 25 卷 第 5 期: 446

[15] Keil Elektronik GmbH. and Keil Software, Inc.. Macro Assembler and Utilities User's Guide 02.2001: 29,261,280

[16] 胡汉才 单片机原理及其接口技术 清华大学出版社 1996:68

[17] 钟建坡 司光宇 MCS-51 系统中断优先级的软扩展 单片机与嵌入式系统应用 2004.4: 22

[18] Keil Elektronik GmbH. and Keil Software, Inc.. Cx51 Compiler Optimizing C Compiler and Library Reference for Classic and Extended 8051 Microcontrollers User's Guide [M] 09.2001: 15,89,92,130,209,218,353

[19]余丽霞, 虞鹤松, 刘昱欣. μC/OS-II 在 C8051F020 中的移植[J] 《电子技术》 2003 年第 7 期: 7.

[20]吴光文 周航慈 51 单片机操作系统开发中的问题与技巧 东华理工学院 单片机与嵌入

式系统应用 2004.11: 76

[21]陈超华,王会进 Keil C51 开发大型嵌入式程序 计算机应用 2003.11: 140-143

[22]张晓华,陈红军,付庄,王卓军 C51 编译器在单片机系统开发中的若干问题 电子技术应用 1997 年第 5 期: 18

[23]J. Labrosse. 嵌入式实时操作系统 μC/OS-II 邵贝贝译. 北京:中国电力出版社 2002: 49-204,186

[24]靳文兵,白凤娥,潘志栋 嵌入式操作系统μC/OS-II 单片机上的移植 太原理工大学学报 第 37 卷第 1 期,2006.1: 101,102

[25]蔡林骥 李清宝 RTX51 嵌入式实时操作系统分析 计算机应用与软件 2005.6 90-92

[26]苑广军,胡冬梅,孙继元,任辉 RTX51 Tiny 任务切换的分析 气象水文海洋仪器 2005.6: 67

[27]刘玉宏 KEIL RTX51 TINY 内核的分析与应用 单片机与嵌入式系统应用 2003.10: 23

[28]Keil Elektronik GmbH. and Keil Software, Inc.. RTX-51 RTX-251 Real-Time Multitasking Executives for the 8051 and MCS 251 Microcontrollers User's Guide[M] 09.1997: 16

[29]周洁,李正凡 嵌入式实时操作系统的调度策略 华东交通大学学报 2005 年 8 月 第 22 卷 第 4 期: 110-111

[30]Intel. 8XC51RA/RB/RC Hardware Description [M] February 1995: 11.

[31]Intel. MCS®51 MICROCONTROLLER FAMILY USER'S MANUAL [M] February 1994 : 1-8, 3-9.

[32] Roger Henriksson. Predictable Automatic Memory Management for Embedded Systems. Department of Computer Science, Lund University 1997

[33]J. Kreuzinger, A. Schulz, M. Pfeffer, Th. Ungerer, et al. Real-time Scheduling on Multithreaded Processors Institute for Computer Design and Fault Tolerance University of Karlsruhe 2000: 1

[34]桂陈 μC/OS 的内核结构及系统研究 重庆大学硕士论文 2005.5: 4,5,6,10

[35]Mohit Aron, Peter Druschel. Soft Timers: Efficient Microsecond Software Timer Support for Network Processing. ACM Transactions on Computer Systems, Vol. 18, No. 3, August 2000, Pages 197–228.

[36]Thomas A. Henzinger, Christoph M. Kirsch "The Embedded Machine:Predictable, Portable Real-Time Code." ACM 2002: 322

[37]Will C. Meilander, Johnnie W. Baker, Jerry Potter. Predictability for Real-Time Command and Control, IEEE-2001 [J].

[38]Daniel P. Bovet, Marco Cesati. Understanding the Linux Kernel, 2nd Edition O'Reilly Dec. 2002: 144,166,169,165.

[39]屠征，谢康林. 嵌入式实时操作系统中对时钟中断服务程序的改进,计算机工程. 2003 年

4 月,第 29 卷第 6 期:79.

[40]王哲华,冯冬芹. FF 通信协议栈中定时器管理机制的设计与实现. 工业仪表与自动化装置. 2005 年第 3 期: 3

[41]George Varghese, Tony Lauck. Hashed and Hierarchial Timing Wheels: Efficient Data Structures for Implementing a Timer Facility. Feb. 1996: 4-20

[42]石春和,乔宇,王江. 单片机 C51 开发新技术的研究. 湘潭矿业学院学报 2000.3 第 15 卷第 1 期: 70

[43]黄亮亮, 朱欣华 基于实时多任务操作系统μCOS-II的C8051F系列单片机应用系统开发 测控技术 2005 年第 24 卷第 9 期: 41

[44]刘洁涓,彭永进 实时操作系统μC/OS-II 在 MCS51 中的移植 计算技术与自动化 第 22 卷第 1 期 2003 年 3 月: 80-81

[45]Dallas Semiconductor. DS80C400 Network Microcontroller. Data sheet REV.102103: 43.

[46]Liu, C.L., and Layland, J.W. "Scheduling Algorithms for Multi-Programming in a Hard Real-Time Environment". Journal of the Association for Computing Machinery Vol. 20, 1 (January 1973): pp. 46-61.

附录 1

```
C?ADDXBP:
C:0x00146F  E509      MOV     A,0x09
C:0x001471  2582      ADD     A,DPL(0x82)
C:0x001473  F582      MOV     DPL(0x82),A
C:0x001475  E508      MOV     A,?C_XBP(0x08)
C:0x001477  3583      ADDC    A,DPH(0x83)
C:0x001479  F583      MOV     DPH(0x83),A
C:0x00147B  B50804    CJNE    A,?C_XBP(0x08),C:1482
C:0x00147E  858209    MOV     0x09,DPL(0x82)
C:0x001481  22        RET
C:0x001482  10AF06    JBC     EA(0xA8.7),C:148B
C:0x001485  858209    MOV     0x09,DPL(0x82)
C:0x001488  F508      MOV     ?C_XBP(0x08),A
C:0x00148A  22        RET
C:0x00148B  858209    MOV     0x09,DPL(0x82)
C:0x00148E  F508      MOV     ?C_XBP(0x08),A
C:0x001490  D2AF      SETB    EA(0xA8.7)
C:0x001492  22        RET

C?XBPOFF:
C:0x001493  E509      MOV     A,0x09
C:0x001495  2582      ADD     A,DPL(0x82)
C:0x001497  F582      MOV     DPL(0x82),A
C:0x001499  E508      MOV     A,?C_XBP(0x08)
C:0x00149B  3583      ADDC    A,DPH(0x83)
C:0x00149D  F583      MOV     DPH(0x83),A
C:0x00149F  22        RET
```

致谢

值此论文完成之际，谨向我的导师罗杰教授表示最诚挚的感谢。感谢他在论文选题、方案确定和论文进展上所做的大量细致工作，以及为我创造了优越的科研和学习环境。论文的研究工作自始至终在他的悉心指导下进行。他渊博的知识、敏锐的思路、严谨的作风、开拓创新的精神以及孜孜不倦的治学态度给我留下了深刻的印象，并将使我受益终身。

在此感谢江西师范大学计算机信息工程学院的老师。他们是王明文教授、甘登文教授、李云清教授和余敏教授等授业老师。感谢他们的辛勤劳动和无私奉献。正是在这样一个优秀的集体中，使我能不断地成长。作者要特别感谢计算机系统结构专业的所有老师和同学。在这里，有良好的科研氛围，有很多高水平及经验丰富的老师，有年轻有为、充满激情的同学，促使我在专业上有了很大的进步。

感谢和我一起完成项目的张剑虹、宋金华、胡荣群和 05 级的吕莉及其他同学。他们的共同支持使我能顺利地完成自己的研究。本文中也凝结着他们的智慧和汗水。

我由衷地感谢父母和我的亲友对我一直的培养与关爱，感谢他们对我的关心、照顾和鼓励。

最后，诚挚地感谢为评阅本文而付出辛勤劳动的各位专家和学者！

在校期间发表的论文

甘智，张剑虹，罗杰 《基于页的 8051 多任务模型》 单片机与嵌入式系统应用 2007.3: 32

个人简历

甘智 （1977.5~ ）1998.7 毕业于南昌航空工业学院电子工程系 获学士学位