代　号	10701	学　号	0881490004
分类号	TP316	密　级	公开
U D C		编　号	

题（中、英文）目　嵌入式实时操作系统 FreeRTOS 在 x86 上的移植

The Transplant of Embedded Real-time Operating System FreeRTOS on x86 platform

作 者 姓 名　杨继兰　学校指导教师姓名职称　白丽娜　高工

工 程 领 域　软件工程　企业指导教师姓名职称　董渊　副教授

提交论文日期　二〇一一年六月

西安电子科技大学

学位论文独创性（或创新性）声明

秉承学校严谨的学风和优良的科学道德，本人声明所呈交的论文是我个人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢中所罗列的内容以外，论文中不包含其他人已经发表或撰写过的研究成果；也不包含为获得西安电子科技大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中做了明确的说明并表示了谢意。

申请学位论文与资料若有不实之处，本人承担一切相关的法律责任。

本人签名：_____ 日期_____

西安电子科技大学

关于论文使用授权的说明

本人完全了解西安电子科技大学有关保留和使用学位论文的规定，即：研究生在校攻读学位期间论文工作的知识产权单位属西安电子科技大学。学校有权保留送交论文的复印件，允许查阅和借阅论文；学校可以公布论文的全部或部分内容，可以允许采用影印、缩印或其它复制手段保存论文。同时本人保证，毕业后结合学位论文研究课题再攥写的文章一律署名单位为西安电子科技大学。

（保密的论文在解密后遵守此规定）

本学位论文属于保密，在　　年解密后适用本授权书。

本人签名：_____ 日期 _____

导师签名：_____ 日期 _____

摘　要

在嵌入式领域中，嵌入式实时操作系统正得到越来越广泛的应用。采用嵌入式实时操作系统(RTOS)可以更合理、更有效地利用CPU的资源，简化应用软件的设计，缩短系统开发时间，更好地保证系统的实时性和可靠性。操作系统到特定硬件的移植关系到整个项目能否按期完工。因此需要选择那些可移植性较高的操作系统，从而避免操作系统向硬件移植带来的困难，加速嵌入式系统开发的进度。

本文主要介绍了嵌入式实时操作系统FreeRTOS在X86的移植。首先详细分析了FreeRTOS操作系统的内核，然后设计了系统引导程序bootloader，最后给出了FreeRTOS在X86的模拟器Skyeye上的移植过程。本文选择了Eclipse集成开发环境，以远程调试运行在开源模拟器Skyeye上的FreeRTOS。其中在Linux下使用GCC编译此操作系统时，需要构建makefile文件、修改内联汇编、处理相关函数。内核装入内存后，还需处理调度器和中断。移植成功后，继续对FreeRTOS做裁减。

将FreeRTOS成功移植到X86，使这一开源操作系统得到更广泛的应用。待整个项目即FreeRTOS的目标码覆盖率分析完成后，FreeRTOS在安全关键领域的应用也会越来越多。

关键词：嵌入式操作系统　FREERTOS　X86　SKYEYE

Abstract

In the embedded system,embedded real time operating system is being more widely used. Embedded real time operating system (RTOS) can be more reasonable and more effective use of CPU resources. It can simplify the design of application software and reduce system development time. Also it can ensure the timeliness and reliability of the system. Porting the operating system to the hardware of the entire system can be completed on schedule or not.Therefore we need to select the operating system of the higher portability.And in order to avoid the difficulties caused to hardware migration and to accelerate the progress of the embedded system development.

This paper describes the embedded real time operating system FreeRTOS transplant on X86. First the system operating environment is set up. And then the bootloader is given. Finally the transplant process is given on the X86 emulator Skyeye.The Eclipse integrated development environment is select to debug FreeRTOS running on open-source remote on the emulator skyeye.

After the success of transplantation, the open source operating system will be more widely applied.And after the code coverage analysis is completed, FreeRTOS will be more and more widely used in areas of security.

Keyword: Embedded Operating System FREERTOS X86 SKYEYE

目　录

第一章　绪论

1.1　选题背景及研究意义

本文所在的项目是嵌入式实时操作系统 FreeRTOS 的目标码覆盖率分析。此项目是针对航空系统对于软件的高安全要求而开发。对应用安全苛刻系统的软件做安全认证是保证嵌入式软件安全可靠的手段之一。使用经过安全认证的可靠的实时操作系统开发嵌入式产品，可部分化解系统安全性方面的风险[1]。在高安全性嵌入式应用领域，无一例外地采用认证标准作为保证质量的依据，如 DO-178B、IEC61508、MISRA 等。DO-178B 标准对用于航空设备方面的软件提出了要求。为了符合这一要求，必须尽可能地通过文件描述和测试，展示软件在稳定性和安全性这两方面都符合要求。目前市场上有关软件可信验证的工具相对较少，价格也过于昂贵。传统的结构化覆盖测试时在源代码级别统计覆盖率，统计的是关于源程序的执行情况信息，而在处理器上执行的是目标码。由于编译器的解释过程或编译器的优化可能会导致产生的目标码不能直接追溯到源代码，由此导致在测试完成后即使源代码 100%覆盖了，目标码还有可能存在没有覆盖到的地方。这些没有覆盖到的地方如果不经过严格的测试和确认，难以保证在目标机上执行的目标码不会引入错误，造成程序执行异常。要完成充分的目标码覆盖测试，工作量很大，没有测试工具难以完成，因此需要提供自动化的有效的目标码测试工具的支持。本项目以航空标准 DO-178B 为出发点，认真分析 A 级软件（安全关键）的要求，在此基础上，试图提供一个可以分析高安全性嵌入式系统的目标码覆盖率的工具和方法。最终选择了 Eclipse 集成开发环境，以嵌入式实时操作系统 FreeRTOS 作为例子，使用开源模拟器 Skyeye，来分析这个嵌入式实时操作系统的目标码覆盖率。

本文所做的工作为Eclipse环境的集成和FreeRTOS在X86模拟器Skyeye上的移植，以及FreeRTOS的裁减。一方面，论文利用的是一些开源软件，这就降低了项目的成本。使用Eclipse集成开发环境，Eclipse调试器提供了可用于调试远程应用程序的功能，使它连接到一个运行应用程序的远程虚拟机Skyeye上。这对于目标机资源受限的软件提供了更大的便利性。之所以使用open64编译器而不是默认的GCC，一方面因为，open64也可以生成高质量的代码，目前已经在嵌入式领域有大量的应用；另一方面，虽然通常大家认为编译器是语义安全的，即编译生成的代码应该准确地按照源代码程序的语义而执行。然而，编译器是一种实施精确符号变换的复杂软件，即便经过严格的测试，编译器中的错误依然屡见不鲜。open64

编译器引入问题比其他编译器引入问题更可控。另一方面，完成了本论文，FreeRTOS这一开源嵌入式实时操作系统也就成功的移植到了目前应用最广泛的X86架构上。

1.2 国内外研究现状

尽管嵌入式系统有着无比广阔的市场需求和发展前景，但嵌入式系统的发展过程却痛苦和曲折。随着微处理器的产生，价格低廉、结构小巧的 CPU 和外设连接提供了稳定可靠的硬件架构，那么限制嵌入式系统发展的瓶颈就突出表现在了软件方面。操作系统到硬件的移植关系到整个系统能否按期完工。因此需要选择那些可移植性较高的操作系统，从而避免操作系统向硬件移植带来的困难，加速嵌入式系统开发的进度。而移植又与硬件高度相关。

由于RTOS需占用一定的系统资源(尤其是RAM资源)，只有μC/OS-II、embOS、Salvo、FreeRTOS等少数实时操作系统能在小RAM单片机上运行。相对于μC/OS-II、embOS等商业操作系统，FreeRTOS操作系统是完全免费的操作系统，具有源码公开、可移植、可裁减、调度策略灵活的特点，可以方便地移植到各种单片机上运行。所谓移植，就是使一个实时内核能在其他的微处理器或控制器上运行。FreeRTOS实现了一个微内核，并且已经移植到ARM7、AVR、PIC18、ColdFire、MIPS等众多处理器上，目前在嵌入式操作系统的市场上占有不少的份额。其移植范例可以从官网获得。本项目将此操作系统移植到使用最广泛的Intel 80X86系列的CPU上。80X86 CPU每年的产量有数百万，大部分用于个人计算机，不过用于嵌入式系统的数量也在快速增加[2]。

目前著名的嵌入式操作系统有 VxWorks、μC/OS-II、QNX、WinCE 等。也有中国人自己开发的嵌入式操作系统 HOPEN、DELTA OS[3]和现在谈论比较多的嵌入式 Linux 等。可用于嵌入式系统开发的操作系统很多，关键是如何选择一个适合所开发项目的操作系统。但所选择的操作系统并不一定支持你所使用的硬件平台，这时就首先需要做操作系统的移植工作。使同一操作系统工作在不同的硬件平台上，不仅节省了资源，也减少了重复劳动，不用每开发一个嵌入式系统都先开发一个操作系统。FreeRTOS 的源码实现，除与硬件相关的部分用汇编完成外，其余部分均由 C 语言编写，这就使得移植相对简单。

FreeRTOS 目前已经支持 26 种微处理器，使用人群在快速增加，这从官网每年的下载量上可以看出。几乎每个月都有新的版本发布，目前最新版本为 6.1.1。新版本实时性性能有所提高，并且增加了几个范例程序和对一个新的硬件平台的支持。国内 FreeRTOS 的研究也发展迅速。在各个行业内的应用也越来越广泛。

1.3　主要研究内容

本文主要介绍了嵌入式实时操作系统FreeRTOS在X86的移植。首先详细分析了FreeRTOS操作系统的内核，然后设计了系统引导程序bootloader，最后给出了FreeRTOS在X86的模拟器Skyeye上的移植过程。本文选择了Eclipse集成开发环境，以远程调试运行在开源模拟器Skyeye上的FreeRTOS。其中在Linux下使用GCC编译此操作系统时，需要构建makefile文件、修改内联汇编、处理相关函数。内核装入内存后，还需处理调度器和中断。移植成功后，继续对FreeRTOS做裁减。在整个过程中主要完成以下工作：

1) open64 和 Eclipse 的集成。Eclipse 的最大特点是插件。其中开发的 C/C++项目由 GNU 编译器 gcc/g++编译。现在引入 open64 编译器，在 Eclipse 中开发的 C/C++项目使用 open64 编译。
2) Skyeye 和 Eclipse 的集成。目的是使用 Eclipse 调试器远程调试 Skyeye 模拟器上的程序，这里用来调试 FreeRTOS 操作系统。
3) 构建 makefile 文件。这是移植 FreeRTOS demo pc 的第一步。和 demo pc 相关的文件有 80 多个，写 makefile 非常必要。比较简洁的方式是使用此 demo 原来的编译器 open watcom 编译，根据产生的 log 文件来追溯这个范例涉及到的源文件，再由这些源文件来构建 makefile 文件。
4) 修改内联汇编。在 DOS/Windows 中，386 汇编语言采用 Intel 定义的格式，在 Linux 中，采用的却是由 AT&T 定义的格式。修改内联汇编。使之符合 AT&T 汇编格式。
5) 处理相关函数。有些函数是 DOS 系统提供，在 DOS 下可以直接使用，而在 Linux 下没有对应的函数，此时就需要自己实现这些函数。
6) Bootbloader 的实现。在 Linux 下使用 GCC 编译完成后，要使其在 Skyeye 上运行起来，需提供引导加载程序 bootbloader，将编译后内核装入内存。
7) 处理调度器和中断。要使这个操作系统正常工作，还必须考虑进程的调度和中断的处理，这对操作系统来说，是很重要的工作。
8) 裁减 FreeRTOS。此 demo 以 20 个任务为例子，演示了 FreeRTOS 的主要功能。其中有些可以去掉，只保留可以反映操作系统核心功能的任务。

1.4　论文结构

论文共分为六章，各章主要内容如下：

第一章：绪论。提出了选题背景、研究意义以及国内外的研究现状，介绍了

本论文的主要工作和组织结构。

第二章：相关技术概述。列举并简要描述了项目研究过程中涉及到的关键理论和技术，其中包括嵌入式操作系统的现状，以及编译器和内联汇编等。

第三章：FreeRTOS分析。深入分析了FreeRTOS操作系统的原理与实现，其中包括时间管理、内存管理、任务调度等。

第四章：系统需求分析。对系统需求和每部分的功能给出了详细的介绍。

第五章：设计与实现。首先实现了open64编译器和Eclipse的集成，以及Skyeye和Eclipse的集成。详细描述了FreeRTOS操作系统的demo pc范例的移植过程，包括函数的处理、BIOS的启动、bootloader的实现、内核的装入、调度器和中断的处理、以及FreeRTOS的裁减。使从DOS上开发的一个程序，能在skyeye模拟器上正常运行，为项目的下一步工作即分析其目标码覆盖率做好准备。

第六章：运行与测试。对论文所完成的工作测试其正确性。

第二章　相关技术概述

2.1　嵌入式实时操作系统简介

2.1.1　操作系统的功能

管理 CPU的系统软件是每一个系统所必需的，不同的CPU 平台，差异很大。如果都要应用系统的开发者自己完成，那么每次都要重复开发重复测试，造成不必要的资源浪费，延长了开发周期，增加了开发成本。并且CPU管理是整个系统的基础，任何缺陷都可能引起严重的故障。操作系统的主要作用是管理资源以满足特定应用的需求。传统的时间共享操作系统把要求公平和高效的资源利用率作为目标。

2.1.2　嵌入式操作系统的起源

实时应用要求及时性和确定性，实时操作系统把满足这些需求的实时应用作为主要目标。由此有了嵌入式操作系统的发展。

不像大型机和PC 机，嵌入式系统中应用所在的硬件系统差异性很大，从嵌入式系统的角度看，PC 机只是一种标准硬件—X86 CPU+PC 主板。因此，我们能看到微软的Windows 只是某类特殊硬件上的一个操作系统，同时没有任何关于PC机应用程序的差异性假设条件。因此，Windows 被设计为通用的操作系统。

嵌入式操作系统具有软件代码小，响应速度快等特点，适合于要求实时和多任务的应用体系，最早应用在航天工业，随着这种系统优势的体现，现在已经被广泛的应用于工业、交通、能源、通信、科研、医疗以及日常生活中。尤其是在工业控制、军事设备、航空航天等领域对系统的响应时间有苛刻的要求，更需要使用实时系统。

在计算机嵌入式应用中使用实时操作系统，是因为RTOS将应用分解成多任务，大大简化了应用程序软件的设计；多任务间可能的竞争问题、任务间通讯问题，有操作系统替用户考虑；RTOS使控制系统的实时性得到保证，可以接近理论上能达到的最好水平；良好的多任务设计，有助于提高系统的稳定性和可靠性，也使应用程序更便于测试、维护与扩展。

2.1.3 常见的嵌入式系统

常见的嵌入式系统有Linux、uxLinux、WinCE、PalmOS、Symbian、eCos、μC/OS-II、VxWorks、pSOS、Nucleus、ThreadX、Rtems、QNX、INTEFRITY、OSE、FreeRTOS等。从商业角度讲最为成功的当属美国WindRiver公司的VxWorks，它经过了广泛的验证，系统稳定可靠，并且提供了包括网络通信等丰富的开发组件，得到了许多硬件厂商的支持，可在各种CPU硬件平台上使用，在高端领域里已经被广泛接受，但是它的价格也是嵌入式操作系统里比较昂贵的。WindRiver公司提供基于图形用户界面的集成开发环境Tornado，集成编辑器、编译器、调试器于一体，主要包括集成的源代码编辑器、工程管理工具、集成的C和C++编译器和make工具、浏览器、图形化的增强型调试器、Windsh、Vxsim、Wind—View和可配置的各种选项。

μC/OS-II的使用也很广泛，它公开源代码，教学上使用免费，是在校学生完成论文和毕业设计的很好的工具。它适用于各种微处理器和微控制器，并且已经被移植到40多种处理器架构中，支持从8位到64位的各种CPU。有大量的移植范例可以从网上免费下载。只是它的开发组件比较少，需要工程师自己做很多扩展工作，不过也由此带来了代码量小和占用系统资源少的优势，适用于投资少要求低的项目中。

相对于μC/OS-II、 embOS等商业操作系统，FreeRTOS操作系统是完全免费的操作系统，具有源码公开、可移植、可裁减、调度策略灵活的特点，可以方便地移植到各种单片机上运行。

2.1.4 通用与专用嵌入式操作系统

一些嵌入式操作系统不针对具体应用领域，这类操作系统称为通用实时嵌入式操作系统[4]。如嵌入式Linux和Windows CE。还有一些操作系统，为特定应用而专门实现发布所必需的必要运行时支持。这些操作系统称为专用嵌入式操作系统，它们在某些特殊的应用领域有些国际标准，如航空工业的ARINC-653 标准， 汽车工业的OSEK/VDK，无线传感器网络的TinyOS，这些操作系统各不相同。

2.1.5 实时操作系统的特点

实时操作系统(RTOS)的使用使得实时应用程序的设计和扩展变得容易，不需要大的改动就可以增加新的功能。通过将应用程序分割成若干独立的任务，RTOS使得应用程序的设计过程大为简化。使用可剥夺性内核时，所有时间要求苛刻的

事件都得到了尽可能快捷、有效的处理。通过有效的服务，如信号量、邮箱、队列、延时、超时等，RTOS使得资源得到更好的利用。

几乎每一个嵌入式项目都可以从实时操作系统（RTOS）中获益， 不管该系统是自行开发的， 或者是VxWorks、 Neutrino、ThreadX、Nucleus 等商业实时操作系统，还是Linux、eCos 之类的开源/自由实时操作系统。事实上，程序员对于操作系统的选型是整个项目成败的关键所在。

2.1.6　嵌入式操作系统产品链

完整的嵌入系统产品链应当包括芯片制造商、嵌入系统提供商（又包括系统设计工具、实时操作系统、开发和编程工具、测试工具、验证和确认工具）、嵌入系统设计公司、制造公司等，这些技术公司背后同样可以看到投资公司和其他服务公司的身影[5]。几乎所有这些实时操作系统参与方都是很小的软件公司，作为整个嵌入系统产品链的一个关节，实时操作系统是关键，但是根本无法成为真正的驱动力。

2.1.7　嵌入式实时操作系统的未来趋势

1) 特定应用领域的整合

在航空和汽车工业等特定应用领域中已经可以看到逐步整合的趋势。这一特点直接反应在相关工业产业中已经通过相应工业标准并且为整个产业所认可，比如航空工业的ARINC-653 和汽车工业的OSEK/VDK。我们将会看到沿着这个方向的进一步发展。

不过，即便有标准，也这并不意味着在该应用领域中只需要该标准的唯一一个系统实现。通常情况下，在同一个领域中总是有遵循相同标准的很多不同系统实现，相互竞争、共生共存。

2) 应用领域发展中的扩散

随着计算技术在更多应用领域的普及，嵌入实时操作系统发展的大趋势是发散，即多样性。这就意味着由于新应用领域新的需求，市场上将出现更多全新的嵌入实时操作系统。

同样的，今天仍然有很多通用和领域专用的编程语言不断出现，可以预料，未来仍将开发完成更多的嵌入实时操作系统。

2.2 编译器

简单地说，编译器是一个程序，它读入用某种语言编写的程序并将其翻译成一个与之等价的另一种语言程序。在翻译的过程中，编译器能够向用户报告源程序中出现的错误。目前，世界上存在着数千种源语言，既有Fortran和Pascal这样的传统语言，也有各计算机应用领域中出现的专用语言[6]。目标语言也同样广泛，可以是另一种程序设计语言或是从微处理机到超级计算机的任何机器语言。不同语言需要不同的编译器。编译由两部分组成：分析与综合。分析部分将源程序分成一些基本块并形成源程序的中间表示，综合部分把源程序的中间表示转换为所需的目标程序。

为了建立可执行的目标程序，除了编译器以外，还需要几个其他的程序：源程序可能被分成模块存储在不同的文件中，把存储在不同文件中的程序模块集成为一个完整的源程序，这个任务由预处理器完成。预处理器也能够把源程序中的宏扩展为原始语句加入到源程序中。图2.1给出了一个典型的编译过程。由编译器创建的目标程序需要进一步处理才能运行。图中的编译器产生汇编代码，然后与一些库程序连接在一起形成可在计算机上运行的代码。

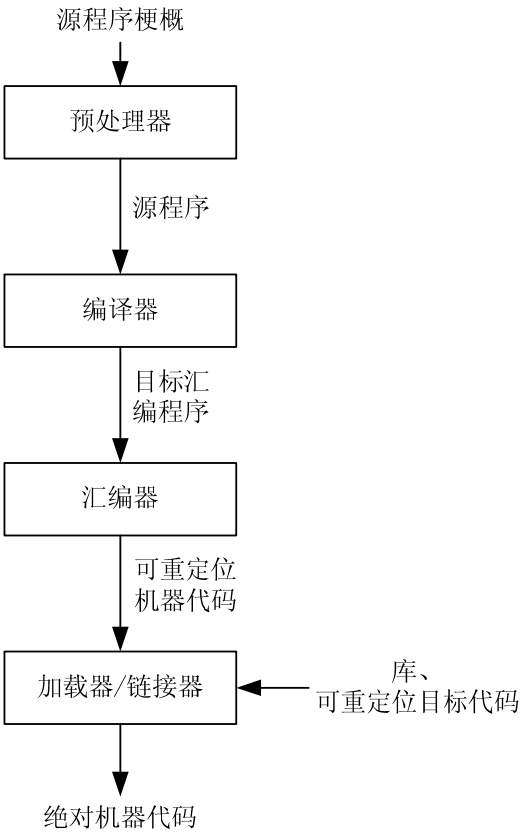

图2.1 一个语言处理系统

从图中可以看到，编译器的输入可能由一个或者多个预处理器产生，而输出

也可能需要进一步的处理才能成为可执行的机器代码。简单介绍一下和编译器的输入输出相关的工具。

2.2.1　预处理器

预处理器产生编译器的输入，一般具有以下功能：

1) 宏处理。预处理器允许用户在源程序中定义宏。宏是经常使用的较长结构的缩写。
2) 文件包含。预处理器可以把头文件包含到程序正文中。例如，C语言预处理器能够用<dos.h>文件的内用替代源程序中的语句#include <dos.h>。
3) 理性预处理器。这些处理器能够把现代控制流和数据结构化机制添加到比较老式的语言中。例如，如果一种语言没有while和if这样的控制结构，理性预处理器可以用内部宏定义向用户提供这类控制结构。
4) 语言扩充。这类处理器通过大量的内部宏定义来增强语言的能力。

宏处理器处理两种类型的语句：宏定义和宏引用。宏定义由具有唯一性的字符或者关键字来标识，如define或macro。

2.2.2　汇编器

某些编译器产生汇编代码，汇编代码需要交给汇编器做进一步的处理。也有些编译器能够完成汇编器的工作，产生可重定位的机器代码，交给加载器loader或链接器linker处理。汇编代码是机器代码的容易记忆的形式。汇编代码使用名称而不是二进制代码来表示操作，存储地址也用名称来表示。

2.2.3　链接器和加载器

链接器和加载器的基本工作很简单：将更抽象的名字与更底层的名字绑定起来，以便程序员使用更抽象的名字编写代码。由于计算机可以执行的基本操作极其简单，有用的程序都是由可以执行更高级、更复杂操作的子程序组成的。通常使用的计算机都带有一些预先编写好的子例程库，程序员可以将这些子例程加载到主程序中以构成一个完整的程序，而不需要编写所有的子程序。在操作系统出现以后，程序就必须和操作系统甚至其他程序共享计算机内存。这意味着在操作系统将程序加载到内存之前是无法确定程序的确切地址的，并将最终的地址绑定从链接时推延到了加载时。链接器对每一个程序的部分地址进行绑定并分配相对地址，加载器完成最后的重定位并分配实际的物理地址。

2.3 内联汇编

汇编语言大家肯定都有所耳闻，但提到汇编语言，经常使人想到低级的位移动和各种各样的指令格式。随着高级语言开发工具的快速发展，不少人认为“汇编语言程序设计已经死亡”。然而，汇编语言还远远没到死亡的时候。每种高级语言在能够连接为可执行程序之前都必须被编译为汇编语言程序。编译器使用编译器的设计者定义的规则来确定如何正确的转换高级语言语句。很多程序员只是编写高级语言程序并且假设编译器会创建正确的可执行代码。但是，情况并非总是如此。当编译器把高级语言程序转换为汇编语言代码时，意想不到的事情时有发生。另外，编译器往往遵循非常特别的转换规则，以至于最终的汇编代码不一定是最节省时间的，而对于一些编写不良的高级例程，也很难加以改善。这种情况下，汇编的知识就有用武之地了。

什么是内联汇编呢？内联汇编即在C和C++程序之内使用汇编语言代码。在原始的C代码内创建函数的汇编语言代码，然后使用标准C编译器进行编译。这种方法对于最终程序在汇编语言级别如何实现特定的函数，给予程序员更多的控制权。内联汇编技术通过把汇编语言函数放置在C或C++程序中，把程序变量传递给汇编语言代码，然后把汇编语言代码生成的输出存放到C程序的变量中。C语言的asm语句包含汇编语言代码，这些代码被放进从C程序代码编译出的汇编语言程序中。虽然可以在C程序中的任何位置放置内联汇编代码，但大多数时候还是把内联汇编代码用作宏函数。在主程序中引用宏时，宏被扩展为宏定义的完整函数。

第三章　FreeRTOS 分析

3.1　FreeRTOS 简介

FreeRTOS是一个轻量级的开源嵌入式实时操作系统，其源码可从官网http://www.freertos.org/获得[7]。它实现了一个微内核，并且移植到ARM7、avr、pic18、coldfire、MIPS等众多处理器上，目前已经在嵌入式操作系统的市场上占有不少的份额。它的目标在于低性能小RAM的处理器上。内核只有4个文件，外加上一些移植范例的和处理器相关的一些文件，实现比较简洁。

FreeRTOS提供的功能包括：任务管理、时间管理、信号量、消息队列、内存管理。其内核支持优先级调度算法，每个任务可根据重要程度的不同被赋予一定的优先级，CPU总是让处于就绪态的、 优先级最高的任务先运行。FreeRT0S内核同时支持轮换调度算法，系统允许不同的任务使用相同的优先级，在没有更高优先级任务就绪的情况下，同一优先级的任务共享CPU的使用时间。这一点是和μC/OS-II不同的。

FreeRTOS既可以配置为可抢占内核也可以配置为不可抢占内核。当FreeRTOS被设置为可剥夺型内核时，处于就绪态的高优先级任务能剥夺低优先级任务的CPU使用权，这样可保证系统满足实时性的要求；当被设置为不可剥夺型内核时，处于就绪态的高优先级任务只有等当前运行任务主动释放CPU的使用权后才能获得运行，这样可提高CPU的运行效率。

3.2　FreeRTOS 源码分析

图3.1是FreeRTOS的源码目录树。

其中，Source中包含内核4个内核源文件，tasks.c、list.c、queue.c和croutine.c，FreeRTOS内核的实现主要由这4个文件组成。list.c 是一个链表的实现，主要供给内核调度器使用； queue.c 是一个队列的实现，支持中断环境和信号量控制；croutine.c 和task.c是两种任务的组织实现。对于croutine.c，所有任务共享同一个堆栈，使RAM的需求进一步缩小。而tasks.c则是传统的任务实现，各任务使用各自的堆栈，支持完全的抢占式调度。include 中包含所需的头文件。Portable包含和范例程序相关的编译器和处理器信息。

Demo中主要是一些范例包，以及这些范例的公用程序。其中PC范例是我们重点关注的。

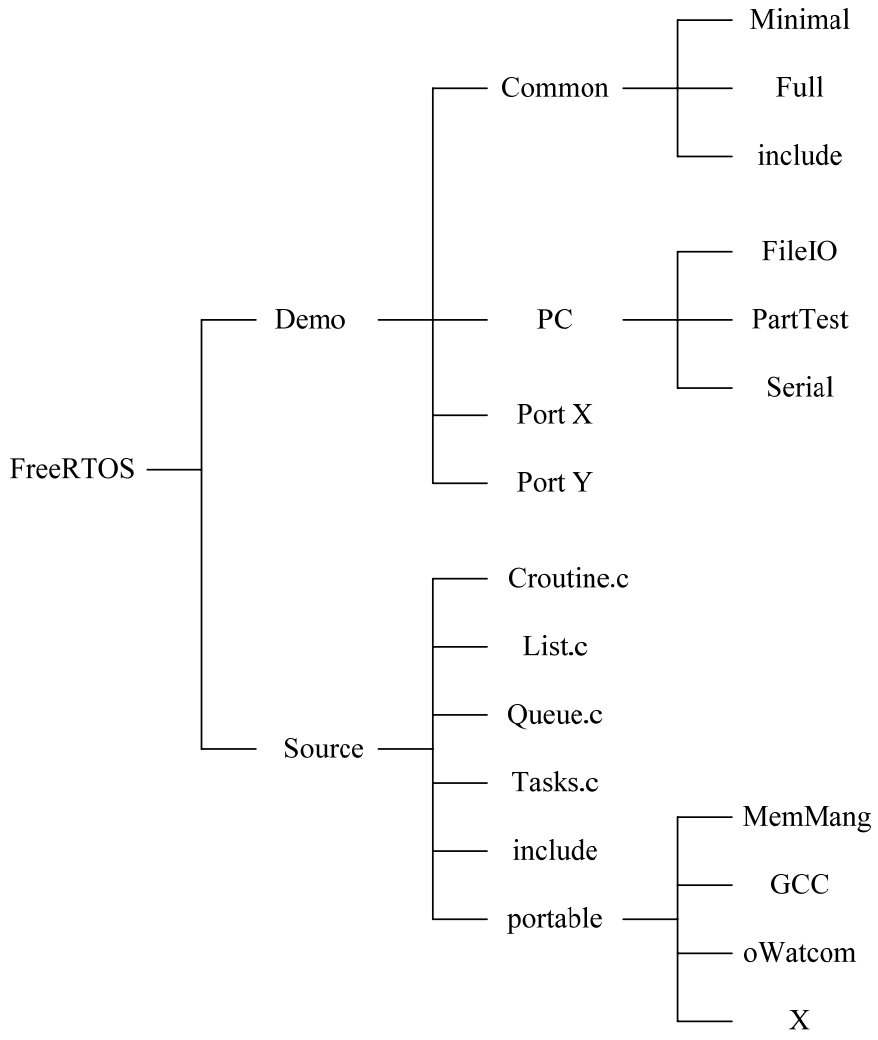

图 3.1 FreeRTOS 的源码目录树

3.3 FreeRTOS 操作系统的原理与实现

3.3.1 任务调度机制的实现

任务调度机制是嵌入式实时操作系统的一个重要概念，也是其核心技术。对于可剥夺型内核，优先级高的任务一旦就绪就能剥夺优先级较低任务的CPU使用权，提高了系统的实时响应能力。不同于μC/OS-II，FreeRTOS对系统任务的数量没有限制，既支持优先级调度算法也支持轮换调度算法，因此FreeRTOS采用双向链表而不是采用查任务就绪表的方法来进行任务调度。

FreeRTOS定义就绪任务链表数组为xList pxReady—TasksLists[portMAX_PRIORITIES]。其中portMAX_PRIORITIES为系统定义的最大优先级。若想使优先

级为n的任务进入就绪态，需要把此任务对应的TCB中的结点xGenericListltem插入到链表pxReadyTasksLiStS[n]中，还要把xGenericListItem中的pvContainer指向pxReadyTasksLists[n]方可实现。

当进行任务调度时，调度算法首先实现优先级调度[8]。系统按照优先级从高到低的顺序从就绪任务链表数组中寻找usNumberOfItems第一个不为0的优先级，此优先级即为当前最高就绪优先级，据此实现优先级调度。若此优先级下只有一个就绪任务，则此就绪任务进入运行态；若此优先级下有多个就绪任务，则需采用轮换调度算法实现多任务轮流执行。

若在优先级n下执行轮换调度算法，系统先通过执行(pxReadyTasksLists[n])→pxIndex=(pxReadyTasks-Lists[n])→pxlndex→pxNext语句得到当前结点所指向的下一个结点，再通过此结点的pvOwner指针得到对应的任务控制块，最后使此任务控制块对应的任务进入运行态。

实现多个任务的有效管理是操作系统的主要功能。FreeRTOS下可实现创建任务、删除任务、挂起任务、恢复任务、设定任务优先级、获得任务相关信息等功能。下面主要讨论FreeRTOS下任务创建和任务删除的实现。当调用sTaskCreate()函数创建一个新的任务时，FreeRTOS首先为新任务分配所需的内存。若内存分配成功，则初始化任务控制块的任务名称、堆栈深度和任务优先级，然后根据堆栈的增长方向初始化任务控制块的堆栈。接着，FreeRTOS把当前创建的任务加入到就绪任务链表。若当前此任务的优先级为最高，则把此优先级赋值给变量ucTopReadyPriorlty。若任务调度程序已经运行且当前创建的任务优先级为最高，则进行任务切换。图3.2为任务调度示意图。

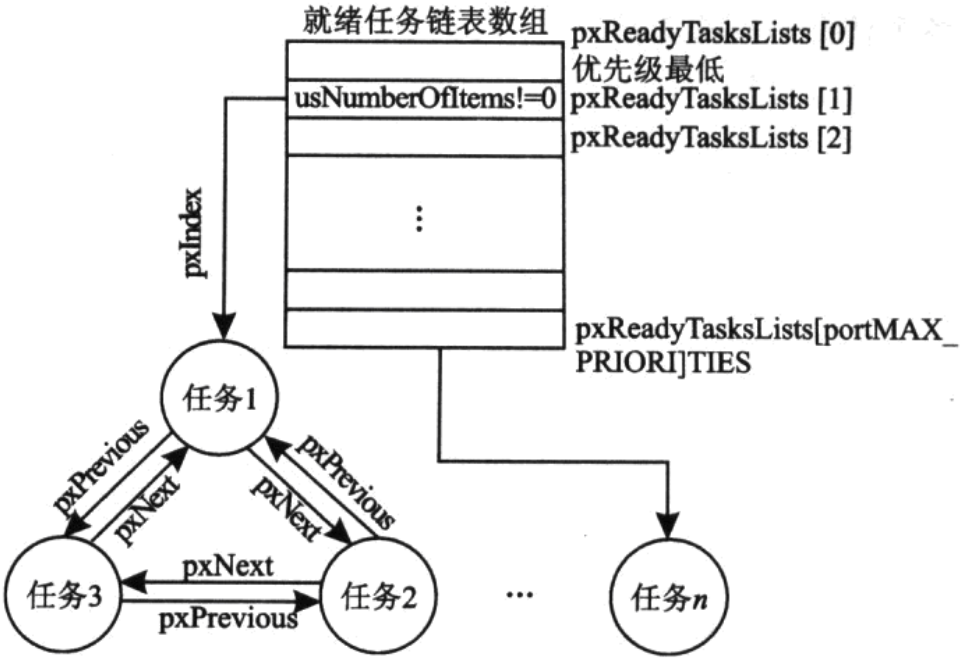

图3.2　任务调度示意图

不同于μC/OS-II，FreeRTOS下任务删除分两步进行。当用户调用vTaskDelete()函数后，执行任务删除的第一步：FreeRTOS先把要删除的任务从就绪任务链表和事件等待链表中删除，然后把此任务添加到任务删除链表，若删除的任务是当前运行任务，系统就执行任务调度函数，至此完成任务删除的第一步。当系统空闲任务即prvldleTask()函数运行时，若发现任务删除链表中有等待删除的任务，则进行任务删除的第二步，即释放该任务占用的内存空间，并把该任务从任务删除链表中删除，这样才彻底删除了这个任务。

通过比较μC/OS-II和FreeRTOS的具体代码发现，采用两步删除的策略有利于减少内核关中断时间，减少任务删除函数的执行时间，尤其是当删除多个任务的时候。

任务调度时，会有上下文切换。任务执行时，它使用处理器或微控制器的寄存器来实现对ROM或RAM的访问。处理器的寄存器以及堆栈等资源，就构成了任务执行的上下文。在FreeRTOS中，每个任务都有自己的堆栈，这使得上下文切换变得简单，只需要将寄存器的值压入堆栈中来保存上下文。上下文保存是使用函数portSAVE_CONTEXT()实现，这个函数的具体实现是一个宏。

3.3.2 时间管理

FreeRTOS 提供的典型时间管理函数是 vTaskDelay()，调用此函数可以实现将任务延时一段特定时间的功能。在 FreeRT0S 中，若一个任务要延时 xTicksToDelay 个时钟节拍，系统内核会把当前系统已运行的时钟节拍总数(定义为 xTickCount，32 位长度)加上 xTicksToDelay 得到任务下次唤醒时的时钟节拍数 xTimeToWake。然后，内核把此任务的任务控制块从就绪链表中删除，把 xTimeToWake 作为结点值赋予任务的 xItemValue，再根据 xTimeToWake 的值把任务控制块按照顺序插入不同的链表。若 xTimeToWake>xTickCount，即计算中没有出现溢出，内核把任务控制块插入到 pxDelayedTaskList 链表；若 xTimeToWake<xTickCount，即在计算过程中出现溢出，内核把任务控制块插入到 pxOverflowDelayed-Taskust 链表。

每发生一个时钟节拍，内核就会把当前的 xTickCount 加 1。若 xTickCount 的结果为 0，即发生溢出，内核会把 pxOverflowDelayedTaskList 作为当前链表；否则，内核把 pxDelaycdTaskList 作为当前链表。内核依次比较 xTickCotlrtt 和链表各个结点的 xTimcToWake。若 xTick-Count 等于或大于 xTimeToWake，说明延时时间已到，应该把任务从等待链表中删除，加入就绪链表。

由此可见，不同于 μC/OS-II，FreeRTOS 采用“加”的方式实现时间管理。其优点是时间节拍函数的执行时间与任务数量基本无关，而 μC/OS-II 的

OSTimcTick()的执行时间正比于应用程序中建立的任务数。因此当任务较多时，FreeRTOS 采用的时间管理方式能有效加快时钟节拍中断程序的执行速度。

3.3.3　内存管理

创建任务、队列和信号量时，FreeRTOS要求分配一定的RAM。与内存管理相关的文件有三个，heap_1.c ，heap_2.c，heap_3.c。其优缺点分别如下：

heap_1.c按照需求内存的大小简单地把一大块内存分割为若干小块，每个小块的大小对应于所需求内存的大小。这样做的好处是比较简单，执行时间可严格确定，适用于任务和队列全部创建完毕后再进行内核调度的系统；缺点是，由于内存不能有效释放，系统运行时应用程序并不能实现删除任务或队列。

heap_2.c采用链表分配内存，可实现动态的创建、删除任务或队列。系统根据空闲内存块的大小按从小到大的顺序组织空闲内存链表。当应用程序申请一块内存时，系统根据申请内存的大小按顺序搜索空闲内存链表，找到满足申请内存要求的最小空闲内存块。为了提高内存的使用效率，在空闲内存块比申请内存大的情况下，系统会把此空闲内存块一分为二。一块用于满足申请内存的要求，一块作为新的空闲内存块插入到链表中。这种方法是用的最普遍的。其优点是，能根据任务需要高效率地使用内存，尤其是当不同的任务需要不同大小的内存的时候；缺点是，不能把应用程序释放的内存和原有的空闲内存混合为一体，因此，若应用程序频繁申请与释放“随机”大小的内存，就可能造成大量的内存碎片。

Heap_1.c和heap_2.c实现了MALLOC和FREE函数，用于内存的分配使用和回收。Heap_3.c用的是编译器本身自带的标准MALLOC和FREE函数。不过标准库中的malloc和free函数存在以下缺点：并不是在所有的嵌入式系统中都可用，并且要占用不确定的内存，可重入性欠缺以及执行时间具有不可确定性，而且多次反复调用可能导致严重的内存碎片。

3.4　FreeRTOS 的不足

相对于常见的μC/OS-II操作系统，FreeRTOS操作系统既有优点也存在不足。其不足之处，一方面体现在系统的服务功能上，如FreeRTOS只提供了消息队列和信号量的实现，无法以后进先出的顺序向消息队列发送消息；另一方面，FreeRTOS只是一个操作系统内核，需外扩第三方的GUI（图形用户界面）、TCP/IP协议栈、FS（文件系统）等才能实现一个较复杂的系统，不像μC/OS-II可以和μC/GUI、μC/FS、μC/TCP-IP等无缝结合。但FreeRTOS在不断完善其功能，以更好地满足人们对嵌入式操作系统实时性、可靠性、易用性的要求。

第四章　系统需求分析

4.1　系统需求分析

现代飞机对软件的可信要求越来越高，仅靠开发过程中各阶段的方法和技术不能满足高质量软件开发的需要。因此，需要一些辅助工具来保证软件的可靠性。航空领域采用DO-178B标准作为保证质量的依据。为了符合这一要求，必须尽可能地通过文件描述和测试，展示软件在稳定性和安全性这两方面都符合要求。目前市场上有关软件可信验证的工具相对较少，价格也过于昂贵。因此，本项目试图使用一些开源工具，来开发出一套针对航空领域做质量认证的系统。

对于A级（安全关键性）应用软件，DO-178B标准要求软件的开发者必须进行目标码的验证工作，虽然这只是应用系统中的一部分，但是它仍然需要进行大量的测试工作，需要投入大量的人力物力。因此，自动化的，不依赖于编译器的验证过程可以节约大量的时间和经费。而任何代码的覆盖率测试都离不开工具的支持（除微小型软件）。

对应用安全苛刻系统的软件做安全认证是保证嵌入式软件安全可靠的手段之一。使用经过安全认证的可靠的实时操作系统开发嵌入式产品，可部分化解系统安全性方面的风险。

目前市场上有关软件可信验证的工具相对较少，价格也过于昂贵。传统的结构化覆盖测试时在源代码级别统计覆盖率，统计的是关于源程序的执行情况信息，而在处理器上执行的是目标码[9]。由于编译器的解释过程或编译器的优化可能会导致产生的目标码不能直接追溯到源代码，由此导致在测试完成后即使源代码100%覆盖了，目标码还有可能存在没有覆盖到的地方。这些没有覆盖到的地方如果不经过严格的测试和确认，难以保证在目标机上执行的目标码不会引入错误，造成程序执行异常。要完成充分的目标码覆盖测试，工作量很大，没有测试工具难以完成，因此需要提供自动化的有效的目标码测试工具的支持。本项目以航空标准DO-178B为出发点，认真分析A级软件（安全关键）的要求，在此基础上，试图提供一个可以分析高安全性嵌入式系统的目标码覆盖率的工具和方法。整个系统要做的工作如图4.1所示。

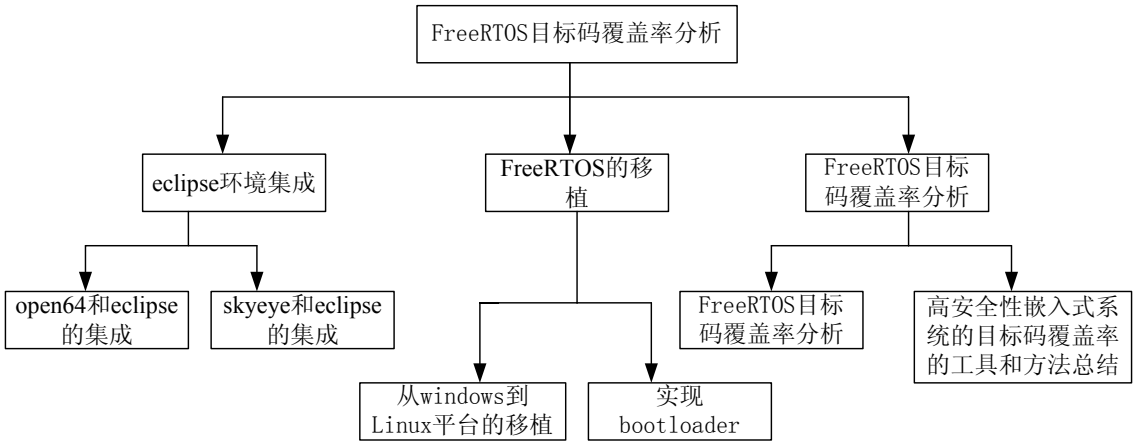

图4.1 系统总体工作

4.2 系统功能分析

下面分别介绍每部分的工作。

4.2.1 Eclipse 环境集成

Eclipse的最大特点是插件。本文实现了Eclipse的两个插件，open64和skyeye。使用open64编译器来替代Eclipse中的GCC来编译C程序。Skyeye用来远程调试运行在目标机上的程序。

1) open64和Eclipse的集成

Linux系统下的程序多用C语言编写，目前Linux系统下最常用的C语言编译器是GCC(GNU Compiler Collection)，它是GNU项目中符合ANSI C标准的编译系统，能够编译C、C++和Object C程序。GCC不仅功能强大，结构也很灵活，它可以通过不同的前端模块来支持各种语言，如Java、Fortran、Pascal、Modula-3和Ada。

编译的多个阶段可以分为前端和后端两个大的阶段。前端包括依赖于源语言并在很大程度上独立于目标机器的某些阶段或者某些阶段的某些部分。前端一般包括词法分析、语法分析、符号表的建立、语义分析、中间代码生成以及相关的错误处理。相当一部分代码的优化工作也在前端完成。后端包括编译器中依赖于目标机器的阶段或某些阶段的某些部分。一般来说，后端完成的任务不依赖与源语言而只依赖于中间语言。后端主要包括代码优化、代码生成以及相关的错误处理和符号表操作。为不同的机器编写相同的源语言程序的编译器时，首先为所有的机器编写相同的前端，然后为每个机器编写编译器的后端。也可以将不同的源语言编译成同一中间语言，对不同的前端使用相同的后端，从而得到同一机器上的不同编译器。

open64编译器，其前端和GCC 4兼容，支持多种高级语言，后端支持多种体系

结构。在正确性和性能方面，open64编译器均接近或达到与GCC编译器相当的水平，可以生成高质量的代码，目前已经在嵌入式领域有大量的应用。

2) Skyeye和Eclipse的集成

嵌入式操作系统移植时，系统调试是开发过程中必不可少且花费时间最多的环节。通用的桌面操作系统与嵌入式操作系统的调试有很大不同。前者，调试器与被调试的程序通常是运行在同一台机器上，调试器通过操作系统提供的接口访问控制被调试的程序。而在嵌入式系统中，调试器运行在宿主机的通用操作系统之上，被调试的进程运行在目标机的嵌入式操作系统中，调试器和被调试进程通过串口或者网络进行通信，调试器可以控制、访问被调试进程，读取被调试进程的当前状态，并能够改变被调试进程的运行状态。这种调试方法称为远程交叉调试。这就带来了以下问题：调试器与被调试程序如何通信，调试器如何控制访问被调试程序，调试器如何处理与目标硬件平台相关的信息。

在本项目中，使用了Eclipse调试器远程调试SkyEye上运行的应用程序。Skyeye是一个开源的硬件仿真平台，支持多架构，多核，并且高度可扩展，当前主要由核心库libcommon.so 和基于核心库的一系列插件组成。支持的体系结构有ARM、Blackfin、Coldfire、PowerPC、MIPS、Sparc、X86 等。SkyEye 的官方网站为www.skyeye.org，其源代码通过svn仓库进行维护，其地址为http://skyeye.sourceforge.net/。

4.2.2 FreeRTOS 的移植

在FreeRTOS的源码中，有很多的范例程序。本项目针对的是demo pc。Demo pc是使用open watcom编译器编译的运行于DOS上的一个范例程序。即编译器生成的是DOS下的可执行文件，在Windows的DOS窗口中运行。Open watcom最早是DOS开发环境中的C/C++编译器，并且是首个支持intel 80386保护模式的编译器。为使这个范例程序在Linux下成功运行，首先就需要使用GCC编译通过。这个过程就有很多工作要做。详见实现部分。编译通过后，接下来的工作就是使其在skyeye上运行起来。这就需要引导加载程序bootloader。这部分的工作见表4.1。

4.2.3 FreeRTOS 目标码覆盖率分析

航空标准RTCA DO-178B中这样描述目标码验证：结构化语言的覆盖率分析可以在源代码级进行，但是如果是A级软件并且编译器产生的目标码不能直接追踪到源代码中的语句，那么，其他的验证要在目标码中执行，这样来确定产生的代码序列的正确性。在目标码中检查编译器生成的数组的边界就是目标码不能直接追

踪到源代码的一个例子[10]。目标码的控制流结构和源码的不一致产生的原因有很多，如编译器的解释和优化等。对于A级（安全关键性）应用软件，DO-178B标准要求软件的开发者必须进行目标码的验证工作，虽然这只是应用系统中的一部分，但是它仍然需要进行大量的测试工作，需要投入大量的人力物力。因此，自动化的，不依赖于编译器的验证过程可以节约大量的时间和经费。而任何代码的覆盖率测试都离不开工具的支持（除微小型软件）。

表4.1 FreeRTOS的移植分析

工作	描述
构建 makefile	组织和 demo pc 相关的 80 多个文件
修改内联汇编	将 Intel 格式的汇编转换为 AT&T 格式的汇编
相关函数的处理	DOS 系统提供 Linux 下没有的函数，需要自己实现
实现 bootloader	提供引导加载程序 bootbloader，将编译后内核装入内存
处理调度器和中断	处理调度器和中断，使操作系统在 skyeye 上运行起来
裁减 demo pc	裁减 demo pc,只保留可以反映操作系统核心功能的任务

对应用安全苛刻系统的软件做安全认证是保证嵌入式软件安全可靠的手段之一。使用经过安全认证的可靠的实时操作系统开发嵌入式产品，可部分化解系统安全性方面的风险。目前市场上有关软件可信验证的工具相对较少，价格也过于昂贵。传统的结构化覆盖测试时在源代码级别统计覆盖率，统计的是关于源程序的执行情况信息，而在处理器上执行的是目标码。由于编译器的解释过程或编译器的优化可能会导致产生的目标码不能直接追溯到源代码，由此导致在测试完成后即使源代码100%覆盖了，目标码还有可能存在没有覆盖到的地方[11]。这些没有覆盖到的地方如果不经过严格的测试和确认，难以保证在目标机上执行的目标码不会引入错误，造成程序执行异常。要完成充分的目标码覆盖测试，工作量很大，没有测试工具难以完成，因此需要提供自动化的有效的目标码测试工具的支持。整个项目以航空标准DO-178B为出发点，认真分析A级软件（安全关键）的要求，在此基础上，试图提供一个可以分析高安全性嵌入式系统的目标码覆盖率的工具和方法。

在目标码覆盖率测试中，基于硬件的一种测试方法其工作原理如下：测试工具采用总线监视技术，在不插桩、不干涉被测软件目标码的情况下，通过对数据线和地址线上的信息进行采集，存储，之后对数据进行分析，得出软件运行的情况[12]。总线上的数据都是二进制的机器代码，而每条汇编语句也对应着一个二进制机器代码。如 51 汇编语言“MOV AX，0x09”对应的目标代码就为“E5 09”；如果数据总线上存在“E5 09”则认为 CPU 执行了这样一条语句。

本项目使用 skyeye 作目标码覆盖率分析。其原理为：因为执行的每一条 PC

都可以看到并记录下来，只需要把这些执行过的指令和地址进行一些分析就可以知道对应的汇编代码或者 C 程序哪些执行了，哪些没有执行。

本论文完成了Eclipse环境集成和FreeRTOS的移植两部分工作。

第五章　设计与实现

系统试图提供一个可以分析高安全性嵌入式系统的目标码覆盖率的工具和方法，选择了以FreeRTOS为例子，分析这个嵌入式实时操作系统的目标码覆盖率。分析目标码覆盖率的工具是使用skyeye。为了实现预期目标，系统选择将FreeRTOS移植到应用最广泛的X86架构上，然后分析其目标码覆盖率。在FreeRTOS的源码中，有很多的范例程序。在分析FreeRTOS的基础上，最终选择了demo pc范例，将其移植到X86的模拟器skyeye上。demo pc是运行于DOS下使用open watcom编译的一个程序。这个范例程序创建了16个任务，在后期做裁减时会有相关任务的详细描述。裁减后，只保留能充分反映此操作系统特点的任务，为接下来的工作即做FreeRTOS目标码覆盖分析减少工作量。

考虑到某些安全关键领域的条件限制，其宿主机和目标机运行在不同的环境中，通常目标机资源有限，需要在宿主机上远程调试运行于目标机上的程序。本论文选择了在Eclipse开发环境中集成skyeye，使用skyeye来远程调试目标机上的程序。同时在Eclipse中集成了open64编译器，来提高高安全性领域的软件对安全性的需求。

综合以上分析，本论文给出这样的设计：

第一步：open64和skyeye集成到Eclipse环境。

第二步：选择FreeRTOS的demo pc范例，将其从windows下移植到Linux下。使用Linux下的编译器GCC编译通过这一程序。

第三步：实现系统引导加载程序bootloader，将这一程序加载至内存。

第四步：处理调度器和中断，使程序在X86的模拟器skyeye上运行起来，准确进行任务的调度和中断的处理。

第五步：裁减FreeRTOS。对16个任务做裁减，只保留最能反映操作系统特性的任务。这减轻了分析目标码覆盖率的工作量。并且也没有必要对16个任务全做分析。

以上五步就是本论文要做的工作。在完成以上工作的基础上，系统继续对FreeRTOS做目标码覆盖率分析。图5.1是本文的工作顺序图。

下面说明环境的搭建和使用方法。

5.1　open64 和 Eclipse 的集成

Eclipse集成开发环境除了可以编辑运行Java程序之外，还支持了许多语言，比

如 C/C++/Fortran。在Eclipse中开发的C/C++项目由GNU编译器gcc/g++编译。

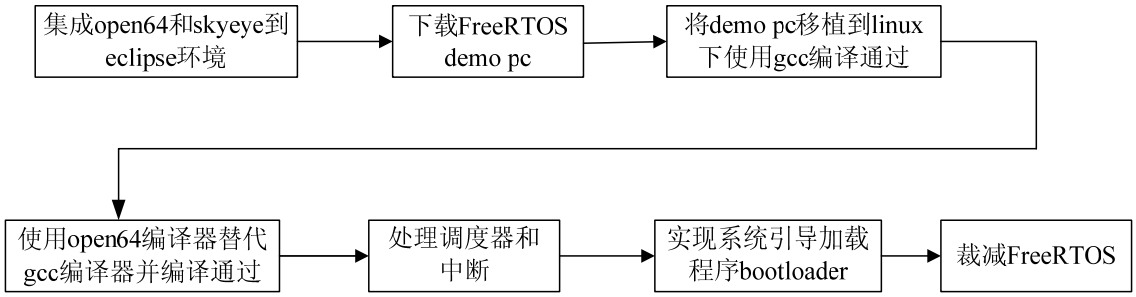

图5.1 工作顺序图

我们现在引入open64编译器，在Eclipse中开发的C/C++项目使用open64编译。下面说明环境的搭建和使用方法。

环境：ubuntu9.10

下载：

1) JDK1.5；
2) Eclipse SDK 3.2.2;（open64 的插件在 Eclipse3.3 上暂时还不能正常工作。）
3) Eclipse CDT3.1.2；
4) Eclipse fortran；
5) open64。（推荐使用版本 4.2.1。下载地址 www.open64.net。rpm 包或 bz2 包或 Source code 都可以，只是 Source code 需要编译。）

安装：

1) JDK 直接双击安装即可；
2) 挂载 CDT
   解压 CDT，将 features 和 plugins 目录下的文件分别拷到 Eclipse 的目录下对应的 feature 和 plugins 目录中。
3) 挂载 fortran
   解压 fortran，将 features 和 plugins 目录下的文件分别拷到 Eclipse 的目录下对应的 feature 和 plugins 目录中。
4) 安装 open64
   设置环境变量 export TOOLROOT=$HOME/mycompiler； 即设置 open64 的安装路径，其中 mycompiler 为自己指定的一个目录
   运行 export PATH=$TOOLROOT/bin:$PATH
   运行 source $HOME/.bashrc ；使更改有效
   解压 open64
   运行 INSTALL.sh

现在，在$TOOLROOT/bin目录下可以看到C，C++，Fortran的编译器，分别是

opencc，openCC，openf90。

完成以上步骤后，继续下载Eclipse的插件 open64-Eclipse-plugin。解压文件到Eclipse目录下，然后执行命令 ./Eclipse －clean，启动Eclipse。

完成了以上设置后，使用时只要在Project Type的下拉菜单中选择Executable(open64 C/C++)就可以了。

5.2 skyeye 和 Eclipse 的集成

嵌入式操作系统移植时，系统调试是开发过程中必不可少且花费时间最多的环节。通用的桌面操作系统与嵌入式操作系统的调试有很大不同。前者，调试器与被调试的程序通常是运行在同一台机器上，调试器通过操作系统提供的接口访问控制被调试的程序。而在嵌入式系统中，调试器运行在宿主机的通用操作系统之上，被调试的进程运行在目标机的嵌入式操作系统中，调试器和被调试进程通过串口或者网络进行通信，调试器可以控制、访问被调试进程，读取被调试进程的当前状态，并能够改变被调试进程的运行状态。这种调试方法称为远程交叉调试。这就带来了以下问题：调试器与被调试程序如何通信，调试器如何控制访问被调试程序，调试器如何处理与目标硬件平台相关的信息。

在本项目中，使用了Eclipse调试器远程调试SkyEye上运行的应用程序。SkyEye是一个开源的硬件仿真平台，支持多架构，多核，并且高度可扩展，当前主要由核心库libcommon.so 和基于核心库的一系列插件组成。支持的体系结构有ARM、Blackfin、Coldfire、PowerPC、MIPS、Sparc、X86等。SkyEye 的官方网站为www.skyeye.org，其源代码通过svn仓库进行维护，其地址为http://skyeye.sourceforge.net/。下载后解压，然后运行如下命令编译安装：./configure，make lib，make，make install，其中make lib用来编译第三方的库，make用来编译skyeye的源代码。默认将skyeye安装到/opt目录下。在/opt/skyeye目录下，直接输入/opt/skyeye/bin/skyeye即可启动。

下面介绍如何利用Eclipse调试器远程调试SkyEye上的应用程序。

Eclipse调试器提供了可用于调试远程应用程序的选项，它可以连接到一个运行应用程序的远程虚拟机上（我们用skyeye）。Eclipse通过CDT支持C/C++，CDT通过调试C/C++代码的功能扩展了标准的Eclipse debug视图，同时CDT debug视图允许开发人员在工作台中管理C/C++项目的调试。CDT不包含其内部调试器，但它向GNU GDB调试器提供了一个前端。处理远程调试与本地调试类似，远程调试配置要求对Run>Debug...窗口进行相应的设置。下载并安装CDT之后，只需切换到Debug视图，就可以调试当前的C/C++项目了。

在Eclipse中，创建工程，设置编译选项，接下来是设置debug选项。根据SkyEye

和GDB的通信协议及端口为TCP/IP、12345，故设置步骤如下：

1) 在工程文件位置单击鼠标右键，点击Run As->Open Run Dialog，如图5.2：

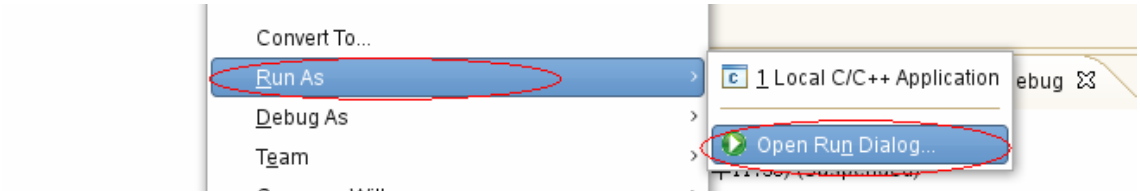

图5.2 打开Debug设置窗口

2) 在弹出的窗口中，一般Main菜单默认即可；

3) 继续点击菜单Debugger，如图5.3，设置断点main、在子菜单Main中选择GDB debugger，其他可默认；

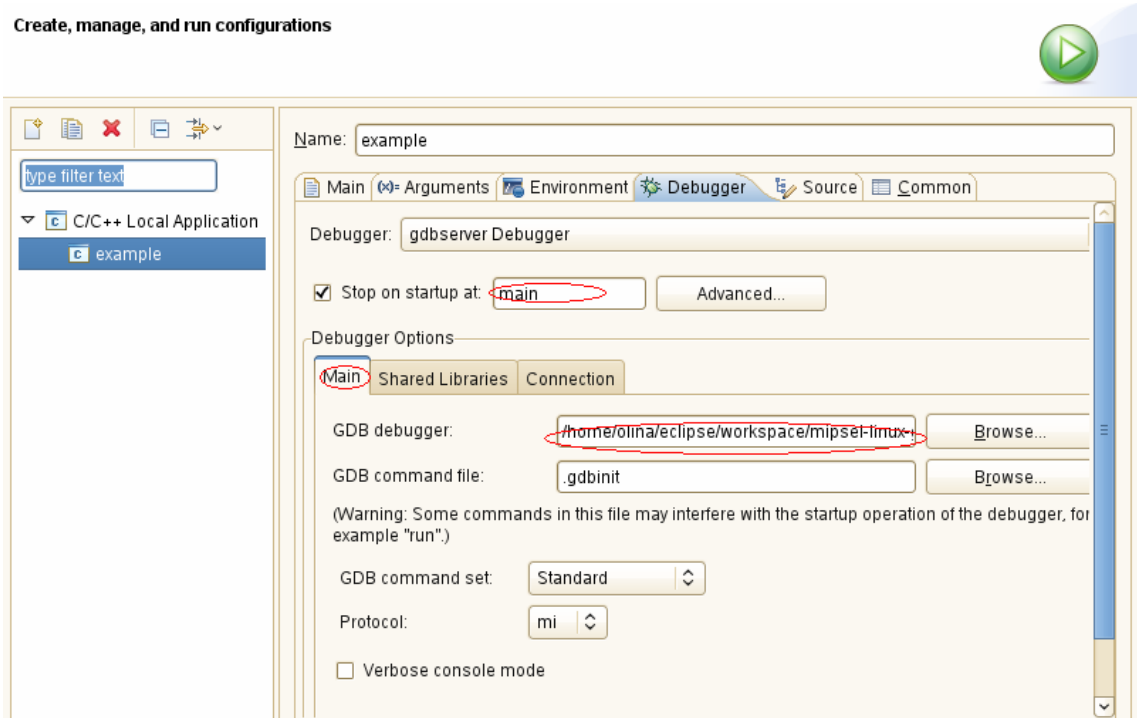

图5.3 Debugger中的Main设置

4) 选中子菜单Connection，如图5.4，依次设置Type: TCP、 Host name or IP address: localhost、 Port number: 12345。

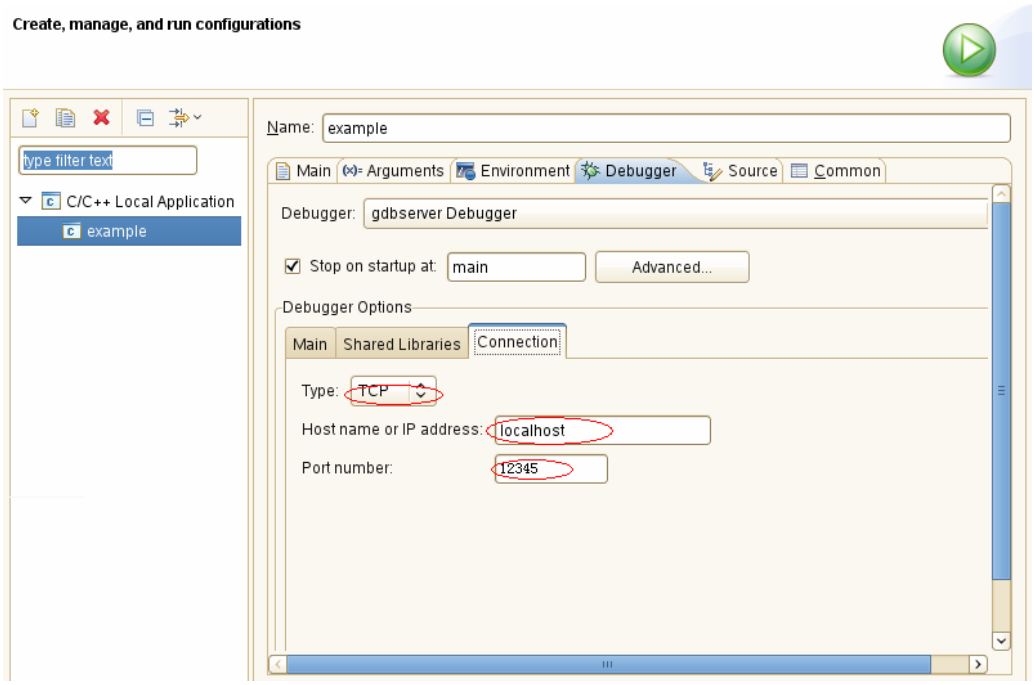

图5.4　Debugger中的Connection设置

Eclipse的设置到此结束。至此，环境的搭建和设置全部完成。

5.3　构建 makefile

先简单介绍一下makefile。在Linux环境下使用GNU的make工具能够比较容易的构建一个大的工程，整个工程的编译只需要一个命令make就可以完成。不过这需要我们花一些时间去写makefile文件，这个文件是make命令正常工作的基础。Makefile文件描述了整个工程的编译、链接等规则，其中包括：工程中的哪些文件需要编译以及如何编译、需要创建哪些库文件以及如何创建这些库文件、如何产生我们想要的可执行文件等。尽管makefile的编写可能比较麻烦，不过完成了makefile后，编译整个工程唯一要做的就是在shell提示符下输入make命令。并且，当某个或某些文件被修改后，make命令会自动根据makefile指定的规则来决定是否执行定义的动作，即那些修改过的文件将会被重新编译。这大大提高了效率。

现在开始在Linux下使用GCC编译FreeRTOS的demo pc的第一步：写makefile文件。和demo pc相关的文件有80多个，写makefile非常必要。比较简洁的方式是使用此demo原来的编译器open watcom编译，根据产生的log文件来追溯这个范例涉及到的源文件，再由这些源文件来构建makefile文件。由于文件中没有需要解释的程序，只是列出了这个demo工程需要的源文件以及依赖的规则和最终需要的目标文件，在此不再给出源码，整个系统完整的makefile文件见附录A。

5.4 修改内联汇编

修改内联汇编指令，使之符合AT&T汇编格式。在DOS/Windows中，386汇编语言采用Intel定义的格式，在Linux中，采用的却是由AT&T定义的格式。两种汇编主要的差异如下：

1) Intel 格式中大多使用大写字母，而在 AT&T 格式中都使用小写字母。
2) 在 AT&T 格式中，寄存器名前要加上“%”作为前缀，而在 Intel 格式中不带前缀。
3) 在 AT&T 格式中，指令的源操作数与目标操作数的顺序与在 Intel 格式中刚好相反。在 Intel 中是目标在前，源在后。
4) 在 AT&T 格式中，访内指令的操作数大小由操作码的最后一个字母(即操作码的后缀)来决定。如用 b 表示 8 位，w 表示 16 位，l 表示 32 位。而在 Intel 格式中，则是在表示内存单元的操作数前面加上“BYTE PTR”，“WORD PTR”，或“DWORD PTR”来表示。如将 FOO 所指内存单元中的字节取入 8 位的寄存器 AL，则：

   MOV AL，BYTE PTR FOO　　(Intel 格式)

   movb FOO，%al　　(AT&T 格式)
5) 在 AT&T 格式中，直接操作数前要加上“$”作为前缀，而在 Intel 格式中则不用。
6) 在 AT&T 格式中，绝对转移或调用指令 jump/call 的操作数要加上“*”作为前缀，Intel 中则不用。
7) 远程转移指令和子程序调用指令不同。
8) 间接寻址的格式不同。

虽然FreeRTOS大部分源码是C程序，但仍有少部分和硬件相关的操作是由汇编完成。其中主要有以下几个方面：上下文切换的实现、内核中直接操作硬件的指令、开关中断指令及某些函数的实现等。重要的汇编函数的实现见附录B。

5.5 相关函数的处理

有些函数是DOS系统提供，在DOS下可以直接使用，而在Linux下没有对应的函数，此时就需要自己实现这些函数。例如_dos_getvect()，_dos_setvect()，是DOS下的两个函数，其功能分别为：保存原来的中断向量和装载新中断向量。就需要自己实现其功能。也有些函数，在Linux下不能直接使用，不过有相同或相似的函数，此时需要替换成对应的函数。

dos.h中定义的两个函数，getvect() 函数用来保存原来的中断向量，setvect() 函数用来装载新中断向量。这里给出这两个函数的实现。

```
extern void (*_dos_getvect( int __intnum ))(){return __intnum*4；}
extern void _dos_setvect( int __intnum，  void (*__handler)() ){*(int *) (void *)
(__intnum*4)=(int) (__handler)；}
```

在port.c文件中的使用到了这两个函数，这里也给出这两个函数的使用。

```
portBASE_TYPE xPortStartScheduler( void )
{
  pxISR pxOriginalTickISR；
//调用这个函数时要在关中断状态下
//先记下当前使用的中断向量以便以后需要时可返回
  pxOldSwitchISR = _dos_getvect( 0x80 )；
  pxOriginalTickISR = _dos_getvect( portTIMER_INT_NUMBER )；
  pxOldSwitchISRPlus1 = _dos_getvect( 0x80 + 1 )；
  prvSetTickFrequency( configTICK_RATE_HZ )；
  prvYieldProcessor()；
//设置新的中断向量
_dos_setvect( 0x80，  prvYieldProcessor )；
_dos_setvect( 0x80 + 1，  pxOriginalTickISR )；
if configUSE_PREEMPTION == 1
{
   _dos_setvect(portTIMER_INT_NUMBER，  prvPreemptiveTick )；
}
else
{
   _dos_setvect(portTIMER_INT_NUMBER，  prvNonPreemptiveTick )；
}
endif
//  设置时钟计数以调用中断
sDOSTickCounter = portTICKS_PER_DOS_TICK；
sSchedulerRunning = pdTRUE；
portFIRST_CONTEXT()；
return sSchedulerRunning；
 }
```

X86中，I/O空间与内存空间相互独立，这样I/O空间的访问需要使用专门的I/O函数。Windows下是使用inp(port) 和outp(port，value)来实现对某个端口的读写操作。Linux下，inb(port)从port参数指定的端口读入并返回一个字节；outb(value，port)将value输出到port端口并返回该值。使用时需要将port参数设置成相应的端口。且Linux下使用端口访问设备之前需调用ioperm函数允许用户操作端口。这两个函数在conio.h中。由于windows2000后，由于系统采用保护模式，用户不能直接访问，需要自己实现一些底层的IO驱动。

5.6 系统的启动和初始化

本节主要讲解从开机到操作系统运行成功需要做的工作。将按照图5.5的顺序来说明。

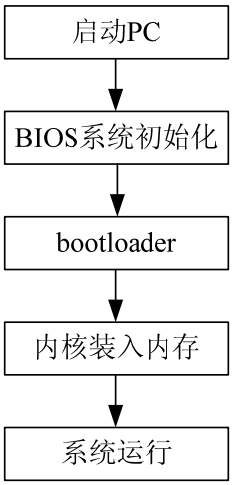

图5.5 系统启动过程

5.6.1 BIOS 系统初始化

在PC启动时，首先会在实模式下运行BIOS，skyeye使用了一个镜像文件BIOS-bochs-latest来模拟真实的BIOS，在启动时此镜像文件的内容被加载到物理内存0x000F0000到0x00100000位置处。在开始时内存中没有任何其他程序可以执行，于是将CS设置为0xF000，将IP设置为0xFFF0，物理地址为0x000FFFF0，这保证了BIOS在刚启动时得到控制权。在BIOS得到控制权后便会对系统进行一系列的初始化。

Skyeye启动的各种参数在bochsrc.txt中，凭字面意思很容易理解，这里列出了主要参数的设置。

```
    //filename of ROM image
romimage: file=./BIOS-bochs-latest
cpu: count=1， ips=10000000， reset_on_triple_fault=0
    //how much memory the emulated will have
    megs: 32
vgaromimage: file=./VGABIOS-lgpl-latest
vga: extension=none
    //what disk images will be used
floppya: 1_44=/dev/fd0， status=inserted
floppyb: 1_44=b.img， status=inserted
    //hard disk
ata0: enabled=1， ioaddr1=0x1f0， ioaddr2=0x3f0， irq=14
ata0-master: type=disk， mode=flat， path="FreeRTOS.img"， cylinders=100，
heads=10， spt=10
    //choose the boot disk
    boot: disk
floppy_bootsig_check: disabled=0
    //where do we send log messages?
log: bochsout.txt
```

在skyeye执行输出过程中，看到”Booting from Hard Disk”，说明BIOS判断系统应该从硬盘启动，接着BIOS将bootloader从硬盘读到内存并把控制权交给该程序，BIOS的任务结束。

5.6.2　bootloader 的实现

在Linux下使用GCC编译完成后，要使其在skyeye上运行起来，需提供引导加载程序bootbloader，将编译后内核装入内存。Bootloader的实现分两步，图5.6为bootloader stage1。

硬盘被划分为一个个大小为512字节的扇区，即每次对硬盘的读写只能对一个或多个扇区来操作。若操作系统从硬盘启动，则硬盘的第一个扇区即为“启动扇区”，bootloader就放在这个扇区。在BIOS找到启动硬盘后，便将启动扇区的512字节的内容装载到物理内存的0x7c00到0x7dff处，接着执行一个跳转指令将CS设置为0x0000，IP设置为0x7c00，把控制权交给bootloader。由于每个扇区的大小为512字节，所以bootloader的大小不能超过512字节。bootloader程序主要由bootasm.S和bootmain.c两个文件组成。先分析一下bootasm.S。

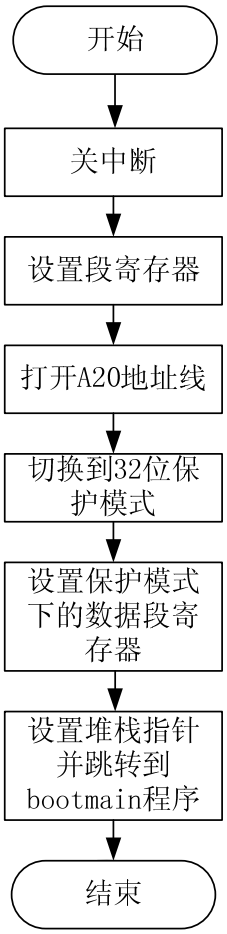

图5.6 bootloader stage1

首先启动CPU，切换到32位保护模式，由汇编程序转换到C程序。实模式和保护模式的主要区别是寻址方式的不同。实模式没有地址空间的保护机制，对于每一个由段寄存器的内容确定的基地址，一个进程总是能够访问从此开始的64K字节的连续地址空间，而无法加以限制。不能对一个进程的内存访问加以限制，也就谈不上对其他进程以及系统本身的保护。

Intel从80286开始实现保护模式。在386中增加了两个寄存器，全局描述符表寄存器GDTR和局部描述符表寄存器LDTR。GDTR和LDTR中的段描述符表指针和段寄存器给出的下标一起，决定了具体的段描述表项在内存中的具体位置。因此也就无法修改表项的内容，从而起到了保护的作用。

BIOS从硬盘上第一个扇区加载以下bootasm.S中的代码到内存物理地址 cs=0 ip=7c00，然后开始在实模式下运行。

```
#include "asm.h"

#define SEG_KCODE 1                    // kernel code
#define SEG_KDATA 2                    // kernel data+stack
```

```
#define CR0_PE      1                    // protected mode enable bit
  .code16                                //Assemble for 16-bit mode
  .globl start
  start:
  cli                                    //Disable interrupts

 //设置重要的段寄存器(DS，  ES，  SS).
  xorw      %ax，%ax                     //Segment number zero
  movw      %ax，%ds                     //-> Data Segment
  movw      %ax，%es                     //-> Extra Segment
  movw      %ax，%ss                     //-> Stack Segment
```

以上这段程序的作用是首先boot程序会进行初始化，先把代码段选择子与数据段选择子以及保护模式移动标志设置为常量，然后关中断，并把ds，es，ss这些段寄存器清零。

接下来打开A20地址线。默认情况下，第20根地址线一直为0，目的是为了向下兼容早期的PC。为了访问所有的内存，我们需要把A20打开，开机时它默认是关闭的。

```
seta20.1:
  inb       $0x64，%al                 // Wait for not busy
  testb     $0x2，%al
  jnz       seta20.1

  movb      $0xd1，%al                 // 0xd1 -> port 0x64
  outb      %al，$0x64

seta20.2:
  inb       $0x64，%al                 // Wait for not busy
  testb     $0x2，%al
  jnz       seta20.2

  movb      $0xdf，%al                  // 0xdf -> port 0x60
  outb      %al，$0x60
```

然后从实模式切换到保护模式。首先用“lgdt gdtdesc”这条指令将GDT表的首地址加载到GDTR，再将CR0寄存器的PE位置1，即系统进入保护模式。

```
  lgdt      gdtdesc
  movl      %cr0，  %eax
  orl       $CR0_PE，  %eax
  movl      %eax，  %cr0

  //跳转到下一条指令同时切换到32位保护模式
  ljmp      $(SEG_KCODE<<3)，  $start32
```

在切换到保护模式后，程序重新对段寄存器初始化后调用bootmain函数。Bootmain后是一个死循环，即程序跳转到bootmain后不再返回。

```
.code32                                          //32位模式下的汇编
start32:
  //设置保护模式下的数据段寄存器
  movw      $(SEG_KDATA<<3)，  %ax      //数据段选择子
  movw      %ax，  %ds
  movw      %ax，  %es
  movw      %ax，  %ss
  movw      $0，  %ax
  movw      %ax，  %fs
  movw      %ax，  %gs

  //设置堆栈指针并且跳转到bootmain程序
  movl      $start，  %esp
  call      bootmain

  movw      $0x8a00，  %ax              // 放置0x8a00 到0x8a00端口
  movw      %ax，  %dx
  outw      %ax，  %dx
  movw      $0x8e00，  %ax              //放置0x8e00 到0x8a00端口
  outw      %ax，  %dx
  spin:
    jmp       spin

  //GDT表
  .p2align 2                                    //GDT表4字节对齐
```

最后定义了GDT表，GDT有三个表项，其中SEG_NULLASM和SEG_ASM()是两个宏。

```
gdt:
   SEG_NULLASM                                  //空表项
   SEG_ASM(STA_X|STA_R，  0x0，  0xffffffff)    //代码段表项
   SEG_ASM(STA_W，  0x0，  0xffffffff)          //数据段表项

gdtdesc:
   .word     (gdtdesc - gdt - 1)                // GDT表长度 － 1
.long     gdt                                   //GDT表物理地址
```

从bootloader程序开始，就可以使用Skyeye进行调试。Skyeye的调试和bochs的使用类似，提供设置断点、单步执行、反汇编某一物理内存的值为指令、显示当前处理器的寄存器的值等功能，是一个调试操作系统内核的很有用的工具。

5.7　内核的装入

Bootloader除了将系统从实模式切换到保护模式外，还需要把内核的可执行程序加载到内存。我们在makefile文件中规定了这个可执行文件是ELF文件。下面先简单介绍一下ELF文件，文件格式如表5.1所示：

表5.1　ELF文件结构

ELF 文件头
程序头表
.text 节
.rodata 节
.stab 节
.stabstr 节
.data 节
.bss 节
.comment 节
节头表

.text节：可执行指令部分。

.rodata节：只读全局变量部分。

.stab：符号表部分，这部分的功能是程序报错时可提供错误信息。

.stabstr：符号表字符串部分。

.data：可读可写的全局变量部分。

.bss：未初始化的全局变量部分。这部分不会占存储空间，默认被初始化为0。

.comment：注释部分。这部分不会被加载到内存。

可以用“objdump －h RTOSDemo.elf”命令才查看elf文件的每个节的信息。

了解了elf文件，下面继续分析Bootloader是如何把内核的可执行文件ELF文件加载到内存中的。在此是使用bootmain.c文件来实现此功能的。程序流程图如图5.7所示。

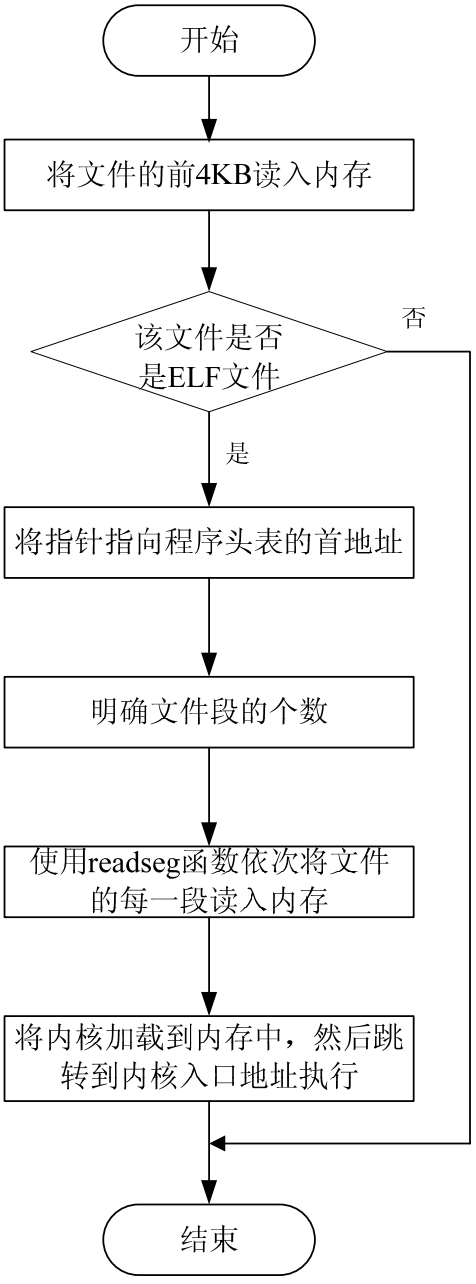

图5.7　bootmain.c程序流程图

这里给出bootmain.c的具体实现与分析。文件中有两个重要的函数readseg()和readsect()。Readseg()依次将文件的每一段读入内存中相应的位置。readsect()用来读取磁盘上的一个扇区。

```
#include "types.h"
#include "elf.h"
#include "x86.h"
#define SECTSIZE   512

void readseg(uchar*，  uint，  uint)；

void bootmain(void)
{
  struct elfhdr *elf;
  struct proghdr *ph，  *eph；
  void (*entry)(void)；
  uchar* va；

  elf = (struct elfhdr*)0x10000；        // 定义一个指向内存中ELF文件头存放位置的结构体指针

  readseg((uchar*)elf，  4096，  0)；  //将文件的前4KB读入内存
  if(elf->magic != ELF_MAGIC)        //判断该文件是否为ELF文件
                    return；

  ph = (struct proghdr*)((uchar*)elf + elf->phoff)；  //将指针指向程序头表的首地址
  eph = ph + elf->phnum；                  //明确文件段的个数
  for(；  ph < eph；  ph++)
    {
          va = (uchar*)(ph->va & 0xFFFFFF)；
           readseg(va，  ph->filesz，  ph->offset)；//使用readseg函数依次将文件的每一段读入内存中相应的位置
           if(ph->memsz > ph->filesz)
              stosb(va + ph->filesz，  0，  ph->memsz - ph->filesz)；
     }

    //将内核加载到内存中，然后跳转到内核入口地址处执行，并且不再返回
  entry = (void(*)(void))(elf->entry & 0xFFFFFF)；
```

```
    entry();
  }
void   waitdisk(void)
  {
    while((inb(0x1F7) & 0xC0) != 0x40);   //等待硬盘准备就绪
  }

// readsect()用来读取磁盘上的一个扇区，其中offset参数代表硬盘的第几个扇区，dst表示这512字节的数据应该存放在内存中的哪个位置。
void   readsect(void *dst,  uint offset)
  {
    waitdisk();
    outb(0x1F2,  1);
    outb(0x1F3,  offset)
    outb(0x1F4,  offset >> 8);
    outb(0x1F5,  offset >> 16);
    outb(0x1F6,  (offset >> 24) | 0xE0)
    outb(0x1F7,  0x20);
    waitdisk();
    insl(0x1F0,  dst,  SECTSIZE/4);
  }

void   readseg(uchar* va,  uint count,  uint offset)
  {
    uchar* eva;
    eva = va + count;
    va -= offset % SECTSIZE;
    offset = (offset / SECTSIZE) + 1;   //将在硬盘中的偏移由字节数转换成扇区数，由于内核可执行程序是从磁盘的第二个扇区开始存储，所以加1。

    for(;  va < eva;  va += SECTSIZE,  offset++)
      readsect(va,  offset);
  }
```

readseg()读取elf文件中的一段。Elf文件头放在0x10000处。

其中elf.h，type.h，asm.h，x86.h是三个头文件。elf.h中定义了ELF文件的格式，x86.h中使用内联汇编实现了bootasm.S中的重要函数。

此时继续完善makefile文件，以完成内核的装入。

```
FreeRTOS.img: bootblock RTOSDemo.elf
dd if=/dev/zero of=FreeRTOS.img count=10000
dd if=bootblock of=FreeRTOS.img conv=notrunc
dd if=RTOSDemo.elf of=FreeRTOS.img seek=1 conv=notrunc


bootblock: ./loader/bootasm.S ./loader/bootmain.c
$(CC) $(OPTIM) -fno-pic -O -nostdinc -I. -c ./loader/bootmain.c
$(CC) $(OPTIM) -fno-pic -nostdinc -I. -c ./loader/bootasm.S
$(LDSCRIPT)    -N -e start -Ttext 0x7C00 -o bootblock.o bootasm.o bootmain.o
$(OBJDUMP) -S bootblock.o > bootblock.asm
$(OBJCOPY) -S -O binary -j .text bootblock.o bootblock
./sign.pl bootblock
```

最终需要生成内核镜像文件FreeRTOS.img，dd命令是Linux下一个非常有用的命令，作用是用指定大小的块拷贝一个文件，在拷贝的同时进行指定的转换。

if=文件名的作用是输入文件名。即指定源文件。of=文件名：输出文件名。即指定目的文件。

seek=1是指从输出文件开头跳过1个块后再开始复制。

conv= notrunc指不截短输出文件。

在bootblock文件的生成规则中，　-Ttext　0x7C00是bootloader的链接地址。./sign.pl是perl脚本文件，作用是确保bootloader的大小不超过512个字节，并且最后4个字节为“55AA”。

到此，系统完成了初始化和内核的加载工作。

5.8　调度器和中断的处理

要使这个操作系统正常工作，还必须考虑进程的调度和中断的处理[13]，这对操作系统来说，是很重要的工作。

5.8.1 调度器

在同一时刻，往往有若干进程同时处于就绪状态。这时就必须依照某种机制来决定哪些进程优先占用处理器。当进行任务调度时，调度算法首先实现优先级

调度。系统按照优先级从高到低的顺序从就绪任务链表数组中寻找usNumberOfItems第一个不为0的优先级，此优先级即为当前最高就绪优先级，据此实现优先级调度。若此优先级下只有一个就绪任务，则此就绪任务进入运行态；若此优先级下有多个就绪任务，则需采用轮换调度算法实现多任务轮流执行。在此项目中，经过裁减后，由原来的20个任务减为7个。

5.8.2 中断的处理

中断是CPU实时地处理内部或外部事件的一种内部机制。当某种内部或外部事件发生时，单片机的中断系统将迫使CPU暂停正在执行的程序，转而去进行中断事件的处理，中断处理完毕后，又返回被中断的程序处，继续执行下去。中断的处理过程为：关中断、保护现场、执行中断服务程序、恢复现场、开中断。在实时环境中，关中断的时间应尽量的短，关中断影响中断响应时间，关中断时间太长可能会引起中断丢失。中断服务的处理时间应该尽可能的短，中断服务所做的事情应该尽可能的少，应把大部分工作留给任务去做。

在DOS中断中，总是把功能号放在AH寄存器中，所有其他一些相关信息放在其他寄存器里。信息放置完毕后，通过INT 21H执行功能调用。查DOS系统功能调用表(INT 21H)可知，25H用来设置中断向量，AH=25H，AL=中断向量号，DS:DX=中断向量。35H用来读取中断向量，AH=35H，AL=中断向量号，ES:BX=存储中断向量地址。通过int 0x80指令产生0x80号软中断。

一个新向量通过使用汇编程序和DOS功能调用被加到中断向量表中。程序首先通过使用INT 21H功能调用号35H读当前向量来保存旧中断向量。此项目中使用_dos_getvect()和_dos_setvect()这两个函数，分别用来保存旧中断向量和设置新中断向量，向量表的地址为0000:0000。原来需要陷入DOS系统调用的中断，在此不再依赖于DOS系统。

系统时钟为操作系统内核提供以下功能：增加ticks计数值；以一定的时间间隔或在特定的时间间隔调度任务；处理任务调度中的时间片轮转；更新延时或超时计数器。

当然要成功移植到skyeye上，除了以上工作，还需要考虑的问题有：字长、数据类型、字节对齐等。不过和以上工作相比，这些问题都是小细节，在此不再一一列出。

5.9 裁减 demo pc

此demo以20个任务为例子，演示了FreeRTOS的主要功能。其中有些可以去掉，

只保留可以反映操作系统核心功能的任务。如数学计算的任务，其功能是为了证明FreeRTOS的上下文切换的正确性，而这在其他任务中也有体现，因此可以去掉。裁减后可适当的减小后续工作（分析FreeRTOS目标码覆盖率）的工作量。这里给出20个任务的分析。

1) prvStartMathTasks()；

调用interger.c中的StartIntegerMathTasks(tskIDLE_PRIORITY)或vStartMathTasks(tskIDLE_PRIORITY)，对调度器的上下文切换机制进行很好的测试。处理器通过两或四次操作来对一个32位的变量进行访问。这些任务的优先级低，即很有可能会在计算的过程中发生上下文切换。与flop.c类似，只不过使用的变量时长整型而不是浮点数。

interger.c与flop.c类似，只不过使用的变量是长整型而不是浮点数。

flop.c的功能为：创建8个任务，每一个不断循环进行（模拟）浮点数运算。所有任务都运行于idle任务的优先级并且不会阻塞或让出处理器。这使得八个任务与idle任务一起进行时间片调度。运行于idle优先级意味着任务会在任何其他任务就绪或时间片到时被抢占。更常见的是在计算的中途被抢占，这样可以对调度器上下文切换进行很好的测试——如果某个计算产生一个意料之外的结果就意味着某个任务的上下文切换出现问题。

interger.c与flop.c类似，只不过使用的变量是长整型而不是浮点数。

2) vStartComTestTasks(mainCOM_TEST_PRIORITY , serCOM1 , ser115200)；

任务的实现在Comtest.c。

创建两个任务用于操作一个中断驱动的串口。要使用一个环形链接器来保证任何发送出去的数据会被接收。此串口没有使用任何流控制。在一个标准的9针串口连接座中2脚与3脚应该连起来。

第一个任务将重复向队列发送字符串，每次一个字符。串口中断将清空队列并且发送字符。任务在每次重发字符串前会挂起一个伪随机的时间。

第二个任务在请求队列时挂起来等待一个字符接收的到来。字符被串口中断服务程序接收并且发送到队列——使任务从挂起恢复到就绪状态。如果是这样则最高优先级的就绪任务将立刻运行——在中断服务程序的最后进行上下文切换。接收字符的任务被赋予一个比发送者更高的优先级。通过适当的环形连接，一个任务发送的字符串将会立刻被另一个任务接收。因为接收的任务知道将收到的是什么字符串，所以可以发现错误。这里同时创建了第三个任务，用于测试在中断服务程序中使用信号量而不做其他的事情。

3) vStartPolledQueueTasks(mainQUEUE_POLL_PRIORITY)；

这是一个很简单的队列测试，其实现在PollQ.c中。更复杂的版本可以参考

BlockQ.c。

由创建两个任务，使用单个队列进行通讯。其中一个任务为生产者，另一个为消费者。生产者循环运行3次，每次循环发送一个递增的数据到队列。然后延时一个预定的时间再继续进行上述的操作。 消费者循环运行，清空队列。它会检查每个从队列中清除的条目，以保证包含的是一个预期的值。当队列为空的时候，它会挂起一个预定的时间，然后再按照上述的方式运行。

所有的队列访问都是无阻塞的进行。消费者每次运行都会完全清空队列，所以生产者永远也不会发现队列为满。

如果消费者得到一个非预期的值或者生产者发现队列为满，一个错误会标识出来。

4) vStartBlockingQueueTasks(mainQUEUE_BLOCK_PRIORITY);

这也是一个队列测试，其实现在BlockQ.c中，只不过比PollQ.c复杂一些。创建了6个任务，用于对3个队列进行如下操作： 前两个任务发送（接收）一个递增的数字到（从）一个队列。一个任务作为生产者而另一个作为消费者。消费者拥有更高的优先级，并且读队列是挂起。队列只有一个条目的空间——一旦生产者向队列发送一个消息，消费者将从挂起状态恢复，抢占生产者，并且删除条目。
还有两个任务以相反的方向工作。这个队列也是只有放下一个条目的空间。这一次生产者比消费者有更高的优先级。生产者尝试发送条目到队列，当队列满时会挂起。当消费者唤醒时，它会从队列移除条目，这将导致生产者从挂起恢复，抢占消费者，并且立刻重新填满队列。

最后两个任务使用相同的队列生产者与消费者函数。这一次队列有足够的空间用于存放大量条目并且任务运行于相同的优先级。生产者将运行，向队列放置条目。消费者将在队列满（导致生产者挂起）或上下文切换发生（同优先级的任务会用时间片调度）。

在做裁减时，可以把vStartPolledQueueTasks()去掉，留下此任务。

5) vCreateBlockTimeTasks();

对任务做测试，以确保任务不过早地退出队列。其实现在blocktim.c中。

6) vStartGenericQueueTasks(mainGENERIC_QUEUE_PRIORITY);

其实现在GenQTest.c中。用以测试FreeRTOS v4.5.0以后版本引入的队列的功能。

7) vStartSemaphoreTasks(mainSEMAPHORE_TASK_PRIORITY);

其实现在semtest.c中。创建两组任务，每组内部的任务共享一个变量，通过一个信号量来监控其访问。

每个任务以尝试获取信号量来开始。获取到信号量的任务会检查变量是否是一个预期中的值，然后把变量清零，接着每次把变量递增1，直到其初始值。在每

次递增后任务会检查变量是否为刚刚设定的值。当变量值达到初始值时任务会释放信号量使得其他任务有机会来进行同样的事情。初始值必须足够大，使得时间片中断可能在变量增加的循环中发生。

如果在处理的过程中发现共享的变量包含的不是预期中的值，一个错误会标识出来。这表明在访问受限的变量时互斥机制发生错误。

两组任务中的第一组以轮询的方式访问信号量，第二组用阻塞的方式。

8)　vStartDynamicPriorityTasks()；

其实现在dynamic.c中。第一个测试首先创建3个任务——两个计数任务（一个连续计数，一个受限计数）与一个控制任务。三个任务间共享一个“计数”。两个计数任务不会在同一时间处于“就绪”状态。控制任务与连续计数任务运行于同一优先级，并且比受限的计数任务优先级低。

其中一个计数任务不定时的循环，在每次循环中增加共享计数变量的值。为了可以独占的访问变量，它会在增加变量前把自身的优先级提升到比控制任务高，访问完后再恢复到原来的优先级，然后进行下一次循环。

另一个计数任务会在每次循环中增加共享计数变量的值，直到变量增加到0xff——此时它会挂起自身。直到控制任务通过调用vTaskResume ()再次将其置为“就绪”状态，才会开始新的循环。这个计数任务比控制任务有更高的优先级，因此不用担心访问计数变量的互斥问题。

控制任务由两部分组成。第一部分控制与监控连续计数任务，当此部分工作时，受限的计数任务是暂停的。同样地，第二部分控制和监控受限的计数任务，当此部分工作时连续计数任务暂停。

控制任务的第一部分首先会复制共享计数变量 。为了独占的访问计数变量，控制任务会先挂起连续计数任务，复制完变量后再将其恢复。然后控制任务休眠一个固定的时间，期间连续计数任务会运行并且增加共享变量的值。控制任务从睡眠中唤醒后会通过对比计数变量此时的值与先前保存的值来检查连续计数任务的运行。此时，为了保证互斥，调用vTaskSuspendAll ()来挂起调度器。这只是作为演示用途，并不是一个推荐的方法，因为这样没有效率。

在一个预定数量的循环后，控制任务挂起连续计数任务，进入第二部分。

在第二部分的开头共享变量被清零。通过调用vTaskResume ()，受限的计数任务从挂起唤醒。因为这个计数任务运行与一个比控制任务更高的优先级，因此控制任务在共享变量增加到受限值、计数任务自己挂起前不会运行。因此在调用vTaskResume ()后就检查共享变量来保证一切都在按照预期的方式运行。

第二个测试有一些简单的任务组成，它们会在调度器挂起时发送数据到队列。增加这个测试是为了弥补第一个测试没有做到的调度器部分的训练。

9)　vStartMultiEventTasks()；

其实现在events.c中。多个任务阻塞等待同一个事件。共创建5个任务，其中4个事件任务，1个控制任务。4个事件任务有不同的优先级，都在阻塞状态，等待读取同一个队列。这个控制任务向队列写数据，并检查哪个事件任务从队列读取数据。通过有选择的挂起和恢复事件任务，这个控制任务确保队列中阻塞的最高优先级的任务从队列中读取数据。

10) vStartQueuePeekTasks();

其实现在QPeek.c中。创建三个不同优先级的任务，测试当有数据到来时，哪个任务被唤醒。

11) vStartCountingSemaphoreTasks();

其实现在countsem.c中。简单的说明计数信号量的使用。

12) vStartAltGenericQueueTasks(mainGENERIC_QUEUE_PRIORITY);

其实现在AltQTest.c 中。功能与GenTest.c相同。

13) vCreateAltBlockTimeTasks();

其实现在AltBlock.c中。功能与BlockTim.c相同。

14) vStartAltBlockingQueueTasks(mainQUEUE_BLOCK_PRIORITY);

其实现在AltBlckQ.c中，功能与vStartBlockingQueueTasks()相同，只是其实现使用了另一API函数。

15) vStartAltPolledQueueTasks(mainQUEUE_POLL_PRIORITY)

其实现在AltPollQ.c中，其功能与vStartPolledQueueTasks()相同。

16) vStartRecursiveMutexTasks()

其实现在recmutex.c中，创建三个任务，以访问同一递归互斥信号量。

17) "Print" task

管理一个等待显示的字符串队列，这用于控制台输出时的互斥。

如果一个任务希望显示一个消息它会调用vPrintDisplayMessage ()，使用一个字符串指针作为参数。这个指针会发送到xPrintQueue队列。

main.c 中的任务会在 xPrintQueue 上阻塞。当一个消息可用会调用 pcPrintGetNextMessage ()来获取下一个字符串的指针，然后使用硬件层FileIO.c 文件中定义的函数来发送信息。

18) vCreateSuicidalTasks()

其实现在death.c中。创建一个固定的任务用于周期性的动态创建另外4个任务。前面的任务称为创建者任务，新建的4个任务称为自杀任务。

新创建的自杀任务中的两个会各自杀掉一个自杀任务然后自杀——只留下最初的一个任务。

创建者任务会一直检查正在受调度控制的任务数量，并且会再次创建其他的范例任务。正常情况下创建者任务所期望的正在运行的任务数量永远不会比其创

建任务时的数量加上4要多，否则就会标识出一个错误。

19) vStartFlashCoRoutines()

其实现在crflash.c中。这个范例程序示范了使用队列来进行联合程序间数据传递，还示范了联合程序索引参量的使用。

创建N个“预定延时”的联合程序以一个预定的优先级阻塞然后发送一个数量的LED到一个队列。每个这种联合程序使用其索引参数作为数组的索引，用于LED闪烁的时间。另一个已创建的独立的“闪烁”联合程序会在同一个队列上阻塞，等待其为下一个LED闪烁的编号。在接收到编号后它会简单的闪烁指示LED然后在同一队列上再次挂起。这样，从LED0到LED N-1的LED会以一个不同的速度闪烁。“预定延时”的联合程序创建为联合程序优先级0，而闪烁联合程序为优先级1：这意味着队列永远也不会包含多余1个条目。因为发送到队列会是有更高优先级的“闪烁”联合程序从挂起恢复，只有当队列为空时才会再次阻塞。如果尝试向队列发送数据失败会指示一个错误——指示队列已经满。

20) vStartHookCoRoutines()

其实现在crhook.c中。这个范例示范了如何在中断服务程序与联合程序见发送数据。一个时间滴答钩子函数用于定时的在RTOS时间片与一定数量的“钩子”联合程序见发送数据。

将创建hookNUM_HOOK_CO_ROUTINES个联合程序，每个联合程序均挂起等待在中断服务程序中从队列接收一个字符，检验保证接收到的为预料中的字符，然后把编号从另外一个队列发回中断服务程序。

时间片中断程序检查从“钩子”联合程序接收到的编号是否与先前发送的吻合。

无论在任何时候队列函数返回一个非期望的值，或者时间片钩子或联合程序接收到一个错误的数据，都会产生一个错误。

这个例子依赖于在每个hookTICK_CALLS_BEFORE_POST时间片中断间运行的‘钩子’联合程序。这样繁复的在中断服务程序中使用队列可能会导致在一个低速的目标上由于时间问题而检测到有错误发生。

通过以上对demo pc中20个任务的分析，最终决定裁减掉任务1、3、5、6、9、10、11、12、13、14、15、18、19，留下的7个是比较能体现操作系统特性的任务。创建完任务后，开始任务调度。

第六章　运行与测试

软件测试是保证软件质量的主要手段。软件测试的目的是发现软件中隐藏的错误和缺陷。统计表明，软件测试的工作量占整个软件产品开发工作量的40%以上。良好的软件测试方案和测试方法，对于减少软件开发中的重复劳动，降低软件工作量作用显著。

通过前几章对系统的分析设计和实现，最终使一个DOS下的程序能在skyeye模拟器上运行起来。这章将对这一成果进行测试。

6.1　测试环境

表6.1是测试的软硬件环境。

表6.1 测试的软硬件环境

硬件环境	Intel Core Duo CPU p8600 2.4GHz 4G 内存
软件环境	windows XP ubuntu 9.10 GCC 4.4 GDB Eclipse SDK 3.2.2 Eclipse CDT 3.1.2 open64 skyeye
描述	测试引导加载程序bootloader及 FreeRTOS 在模拟器上的运行情况

6.2　bootloader 测试

Eclipse 安装完成后，挂载 open64 和 skyeye 到 Eclipse 的集成环境。首先从 sourceforge 网站上下载 Skyeye 的 1.3.0 的源代码，解压

tar xzvf skyeye-1.2.9_rc1.tar.gz

然后运行如下命令编译。

./configure

make lib

make

其中make lib来编译第三方的库，make来编译Skyeye的源代码

最后安装Skyeye到 /opt目录下。

make install_lib

make install

安装之后，/opt/skyeye有以下目录：

bin conf include info lib testsuite

其中 bin 目录存放了 Skyeye 的二进制程序。

直接输入/opt/skyeye/bin/skyeye会启动Skyeye的命令行应用程序，并进入Skyeye的命令行接口，显示如下：

```
ubuntu@yang:/opt/skyeye> /opt/skyeye/bin/skyeye
SkyEye is an Open Source project under GPL. All rights of different parts or modules are reserved by their author. Any modification or redistributions of SkyEye should note remove or modify the annoucement of SkyEye copyright.
Get more information about it, please visit the homepage http://www.skyeye.org.
Type "help" to get command list.
(skyeye)
```

首先启动Skyeye上的应用程序。通过命令行进入相应的example/Debug目录，运行如下命令启动应用程序example：

$skyeye －d －e example －c skyeye.conf

接下来在Eclipse下的工程文件位置点击鼠标右键在弹出菜单中选择Debug As->1 Local C/C++ Application开始跟踪调试。图6.1为debug状态下截取的运行图。

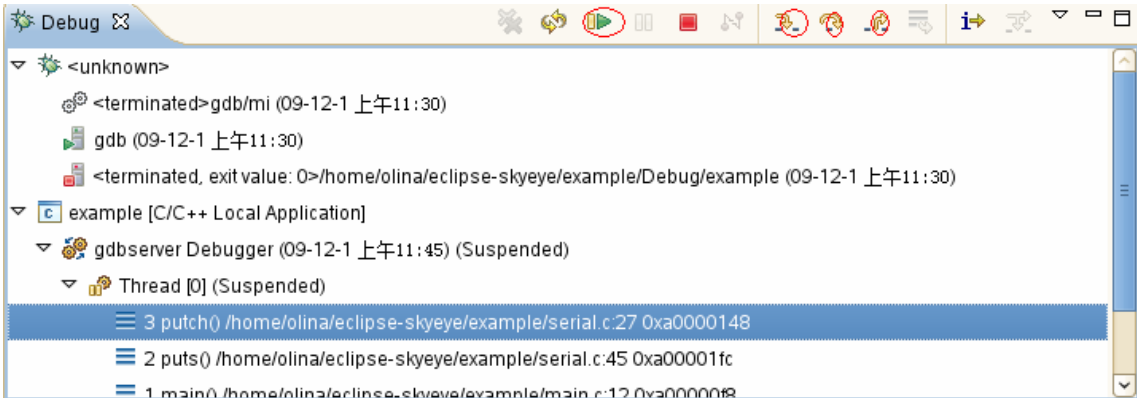

图 6.1 Debug 状态下的运行图

图6.2是调试时可以设置的选项。

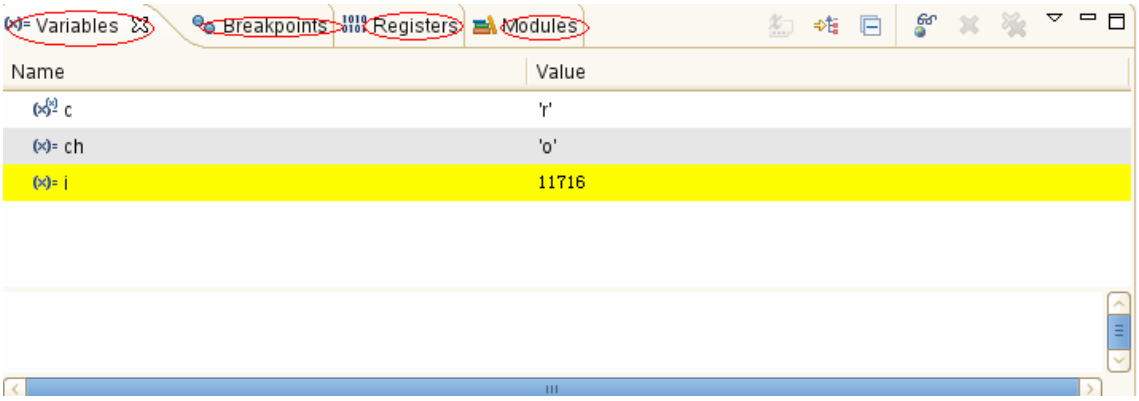

图6.2　调试中可设置的选项

至此，bootloader 已经成功运行。

6.3　FreeRTOS 在模拟器上的运行测试

经过裁减后，demo pc 的 16 个任务只剩下了 7 个。选择第一个任务 prvStartMathTasks()进行测试，以此来验证任务调度和中断处理的正确性。

图 6.3 是调试过程中查看内存地址信息的截图。

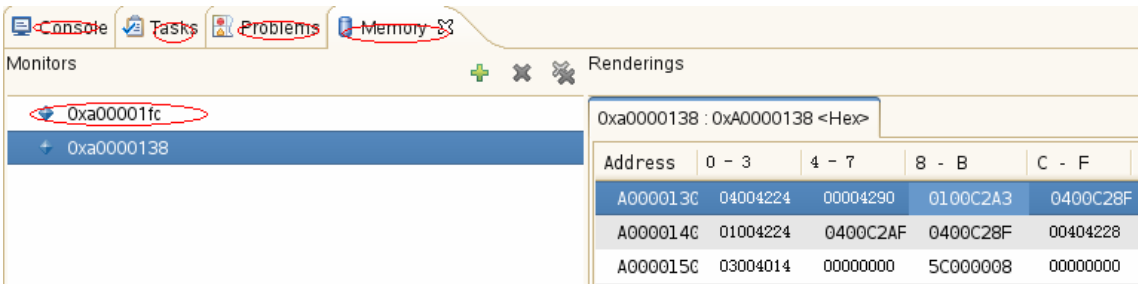

图6.3　查看内存地址信息

测试结果表明，系统对任务的调度和中断的处理正常。扩展到 7 个任务，系统也运行正常。

至此，FreeRTOS 已成功移植到 X86 的模拟器 skyeye 上。接下来，可以继续使用 skyeye 对其做目标码覆盖率分析。

结束语

对于软件安全关键领域，对软件做安全认证是保证软件安全可靠的手段之一。然而，仅靠开发过程中各阶段的方法和技术仍不能满足高质量软件开发的需要。这就需要一些辅助工具来保证软件的可靠性。目前市场占有率较高的高安全性应用开发环境，如法国爱斯特尔的SCADE，价格昂贵。

在嵌入式系统中，被调试的程序通常运行在资源受限的目标机上。调试器和被调试的程序不在同一台机器上，通过串口或者网络进行通信。这就需要远程交叉调试。论文选择了Eclipse集成开发环境，以远程调试运行在模拟器skyeye上的FreeRTOS操作系统。这解决了远程调试的问题，对于目标机资源受限的软件提供了更大的便利性。同时论文利用的是一些开源软件，这就降低了项目的成本。

FreeRTOS移植到X86上后，经过裁减，继续对这一操作系统作目标码覆盖率分析。FreeRTOS的目标码覆盖率分析完成后，如果没有覆盖，修改其内核，使之完全覆盖。这一工作完成后，FreeRTOS在安全关键领域的应用也会越来越多。同时也提供了一个分析软件目标码覆盖率分析的方法和工具，为开发高质量软件提供了一个保障。

然而，编译器即便经过严格的测试，编译器中的错误依然屡见不鲜。这些错误可能会导致编译器的编译时崩溃，而更可怕的是不声不响地将原本正确的源程序变换为错误的可执行程序。

对于低安全保证软件而言，只需经过测试来确认即可，编译器的错误影响甚微。所测试的是编译器所生成的可执行代码，严格的测试在发现源程序错误的同时应该也可以发现编译器引入的错误。需要特别关注的是编译器引入的错误通常都是那种极难发现、极难定位的错误。而对于安全关键的高安全保证的软件，情况则大不相同。此时，单纯的测试手段已无能为力，需要增加或者完全采用模型检测、静态分析和程序证明等一系列形式化方法来实施软件确认。然而，几乎无一例外地，大家都只在程序的源代码级运用各种形式化验证工具，因此，已验证源代码很有可能由于编译器内部的小错误而变成一个有问题的可执行程序，人们煞费苦心通过形式化方法得到的的所有特性都将随着编译器的错误而付之东流。在不久的将来，当形式化方法在源程序级别中普遍应用的时候，编译器有可能成为从程序规格说明到可执行代码这个完整变换链条中最为薄弱的一环。安全关键软件产业已经开始意识到问题严重性，并已经开始使用诸如“不进行任何优化并对编译生成汇编代码进行人工检查”等各种技术来缓减之[14]。然而，即便是付出巨大的开发时间和程序性能成本，这些技术还是治标不治本，无法从根本上解决

这个难题。

一个显而易见的好办法是应用形式化方法对编译器自身进行验证，以保证编译器能够在进行翻译变换的同时保持源代码的语义。目前CompCert编译器，是具备这一特性的真实的、已验证的编译器。称其为已验证，是指这个编译器同时带有可机器自动检查的证明，该证明表明该编译器具有语义保持特性，即所生成的机器代码准确地按照源代码程序的语义执行。称其为真实，我们是指这个编译器可以实实在在地应用于关键软件的生产过程，也就是说，该编译器处理的是关键嵌入软件常用的语言，既不是JAVA，也不是ML，更不是汇编代码，而是一个很大的C语言子集；而且该编译器生成的代码将运行于嵌入系统常用的处理器，即航空工业中非常普及的PowerPC处理器；最后，该编译器所生成的代码足够优化、足够紧凑，能够满足关键嵌入系统的需要。也就意味着，这是一个支持多遍编译、具备良好的寄存器分配算法和基本优化能力的编译器。CompCert编译器将Clight（一个C语言的较大子集）翻译为PowerPC汇编，使用coq证明辅助工具来编写编译器并完成其正确性证明。对于关注关键软件及其形式化验证的情况下，此编译器非常有用：编译器的验证保证源代码级证明过的安全性质能够一直保持到编译所产生的可执行代码。

本项目是运行在X86平台上，而compcert编译器暂时还不支持这一平台。作为未来的展望，可以把这一项目移植到PowerPC平台上，然后使用compcert编译器。对软件做目标码覆盖率分析，也只是对安全关键软件做安全认证的一个方面，可以做的工作还有很多，如形式化验证等[15]。

致　谢

非常感谢清华大学计算机系的董渊老师，在我实习期间，对于我所做工作的指导，他为此花费了大量的时间和精力。

感谢校内导师白丽娜老师对我论文的指导，她对工作极其认真负责。

感谢实验室的朋友们，这一年多来对我学习和生活上的无私帮助，让我在困难时坚持下来。

感谢周围帮助过我的同学们。

参考文献

[1] RTCA Inc， Software Considerations in Airborne Systems and Equipment Certification， RTCA/DO- 178B/ED-12B. (December 1992)

[2] Moonju Park， Non-preemptive fixed priority scheduling of hard-time periodic tasks，Computational Science-ICCS 2007

[3] Ming-Yuan Zhu， Lei Luo， and Guang-Zhe Xiong. A Provably Correct Operating System: delta-Core. Operating Systems Review， 35(1):17–33，January 2001.

[4] Krodel， J. and Romanski， G. (2007). Real-time operating systems and component integration considerations in integrated modular avionics systems report. Technical report， U.S. Department of Transportation - Federal Aviation Administration.

[5] Ming-Yuan Zhu，The Next Hundreds Embedded Real-time Operating System， CoreTek Systems，Inc 2010

[6] T. A. Henzinger， B. Horowitz， and C. M. Kirsch， “Giotto: A Time-Triggered Language for Embedded Programming，” Proc. of EMSOFT 2001， Tahoe City，CA， LNCS 2211， Springer-Verlag，October， 2001.

[7] FreeRTOS webpage www.FreeRTOS.org

[8] G Salenby， D Lundgren， Comparison of Scheduling in FreeRTOS and eCos， November 19， 2006

[9] Horgan，J.，London，S.，Lyu，M.，"Achieving Software Quality with Testing Coverage Measures"，IEEE Computer， Vol. 27， No. 9，September 1994

[10] Y.K.Malaiya，N.Li，F.Karcich， and R.Skibbe， The Relationship between Test Coverage and Reliability， Proceedings of the Fifth International Symposium on Software Reliability Engineering，Monterey，CA， Nov 6-9 1994

[11] Malaiya，Y.K.，Li，M.N. and Bieman，J.M， 2002. Software reliability growth with test coverage. IEEE Transactions on Reliability 51(2002) (4)

[12] J.J. Chilenski and S.P. Miller. Applicability of Modified

Condition/Decision Coverage to Software Testing. Copyrightr 1993， The Boeing Company and Rockwell International Corporation， Submitted for publication.

[13] D. I. Katcher ， H. Arakawa ， J. K. Strosnider， Engineering and Analysis of Fixed Priority Schedulers， IEEE Transactions on Software Engineering， September 1993 [doi>10.1109/32.241774]

[14] James H. Andrews ， Lionel C. Briand ， Yvan Labiche ， Akbar Siami Namin， Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria， IEEE Transactions on Software Engineering， v.32 n.8， p.608-624， August 2006 [doi>10.1109/TSE.2006.83]

[15] Lars Asplund ， Kristina Lundqvist， Safety critical systems based on formal models， ACM SIGAda Ada Letters， v.XX n.4， p.32-39， Dec.， 2000

附　录

附录 A　makefile 文件

```
RTOS_SOURCE_DIR=../../Source
DEMO_COMMON_DIR=../Common
DEMO_COMMON_FULL_DIR=../Common/Full
DEMO_INCLUDE_DIR=../Common/include
CC=gcc
OBJCOPY=objcopy
LDSCRIPT=ld
OBJDUMP =objdump
-mcmodel=medium

LINKER_FLAGS=   -nostartfiles -o RTOSDemo.elf
LD_FLAGS= -L/usr/lib/gcc/i486-Linux-gnu/4.4.1 -L/usr/lib/gcc/i486-Linux-gnu/4.4.1
-L/usr/lib/gcc/i486-Linux-gnu/4.4.1/../../../../lib -L/lib/../lib -L/usr/lib/../lib
-L/usr/lib/gcc/i486-Linux-gnu/4.4.1/../../.. -L/usr/lib/i486-Linux-gnu
LD_FLAGS+= --start-group -lgcc -lgcc_eh -lc --end-group
DEBUG=-g
OPTIM=-static -O2 -std=c99 -D _M_IX86 -D USE_STDIO -D DEBUG_BUILD -D
GCC_PC_PORT

CFLAGS= $(DEBUG) \
        $(OPTIM) \
         -I .
         -I ./include   \
         -I $(RTOS_SOURCE_DIR)/include \
         -I $(RTOS_SOURCE_DIR)/portable/GCC/Linux/pc \
         -I $(RTOS_SOURCE_DIR)/portable/GCC/Linux/common \
         -I $(DEMO_INCLUDE_DIR) \
         -fomit-frame-pointer
```

```
PC_SOURCE= \
main.c \
./FileIO/fileIO.c    \
./ParTest/ParTest.c \
./serial/serial.c    \
$(DEMO_COMMON_DIR)/Full/BlockQ.c    \
$(DEMO_COMMON_DIR)/Full/comtest.c   \
$(DEMO_COMMON_DIR)/Full/death.c     \
$(DEMO_COMMON_DIR)/Full/dynamic.c   \
$(DEMO_COMMON_DIR)/Full/events.c    \
$(DEMO_COMMON_DIR)/Full/flop.c      \
$(DEMO_COMMON_DIR)/Full/integer.c   \
$(DEMO_COMMON_DIR)/Full/PollQ.c     \
$(DEMO_COMMON_DIR)/Full/print.c     \
$(DEMO_COMMON_DIR)/Full/semtest.c   \
$(DEMO_COMMON_DIR)/Minimal/AltBlckQ.c\
$(DEMO_COMMON_DIR)/Minimal/AltBlock.c \
$(DEMO_COMMON_DIR)/Minimal/AltPollQ.c \
$(DEMO_COMMON_DIR)/Minimal/AltQTest.c \
$(DEMO_COMMON_DIR)/Minimal/blocktim.c \
$(DEMO_COMMON_DIR)/Minimal/countsem.c \
$(DEMO_COMMON_DIR)/Minimal/crflash.c   \
$(DEMO_COMMON_DIR)/Minimal/crhook.c    \
$(DEMO_COMMON_DIR)/Minimal/GenQTest.c \
$(DEMO_COMMON_DIR)/Minimal/QPeek.c     \
$(DEMO_COMMON_DIR)/Minimal/recmutex.c \
$(RTOS_SOURCE_DIR)/list.c   \
$(RTOS_SOURCE_DIR)/queue.c \
$(RTOS_SOURCE_DIR)/tasks.c \
$(RTOS_SOURCE_DIR)/croutine.c \
$(RTOS_SOURCE_DIR)/portable/GCC/Linux/pc/port.c \
$(RTOS_SOURCE_DIR)/portable/GCC/Linux/common/portcomn.c \
$(RTOS_SOURCE_DIR)/portable/MemMang/heap_2.c

PC_OBJS = $(PC_SOURCE:.c=.o)
```

```
FreeRTOS.img: bootblock RTOSDemo.elf
    dd if=/dev/zero of=FreeRTOS.img count=10000
    dd if=bootblock of=FreeRTOS.img conv=notrunc
    dd if=RTOSDemo.elf of=FreeRTOS.img seek=1 conv=notrunc

    bootblock: ./loader/bootasm.S ./loader/bootmain.c
    $(CC) $(OPTIM) -fno-pic -O -nostdinc -I. -c ./loader/bootmain.c
    $(CC) $(OPTIM) -fno-pic -nostdinc -I. -c ./loader/bootasm.S
    $(LDSCRIPT)   -N -e start -Ttext 0x7C00 -o bootblock.o bootasm.o bootmain.o
    $(OBJDUMP) -S bootblock.o > bootblock.asm
    $(OBJCOPY) -S -O binary -j .text bootblock.o bootblock
    ./sign.pl bootblock

all: RTOSDemo.bin

RTOSDemo.bin : RTOSDemo.hex
        $(OBJCOPY) RTOSDemo.elf -O binary RTOSDemo.bin

RTOSDemo.hex : RTOSDemo.elf
        $(OBJCOPY) RTOSDemo.elf -O ihex RTOSDemo.hex

RTOSDemo.elf : $(PC_OBJS) Makefile
        #$(CC) $(CFLAGS) $(LINKER_FLAGS) $(PC_OBJS) $(LIBS)
        $(LDSCRIPT) -static -Ttext 0x100000 -e main -o RTOSDemo.elf $(PC_OBJS)
$(LD_FLAGS)
        $(OBJDUMP) -S RTOSDemo.elf > RTOSDemo.asm
        $(PC_OBJS) : %.o : %.c Makefile FreeRTOSConfig.h
        $(CC) -c $(CFLAGS) $< -o $@

        clean :
        rm -f $(PC_OBJS)   RTOSDemo.elf\
          bootblock          bootblock.asm          bootasm.o          bootmain.o
bootblock.o   FreeRTOS.img   RTOSDemo.asm
```

附录 B　重要的汇编函数的实现

```
void portSWITCH_CONTEXT( void );
/*
* Load the stack pointer from the TCB of the task which is going to be first
* to execute.  Then force an IRET so the registers and IP are popped off the
* stack.
*/
void portFIRST_CONTEXT( void );

/* There are slightly different versions depending on whether you are building to include debugger information.  If debugger information is used then there are a couple of extra bytes left of the ISR stack (presumably for use by the debugger).  The true stack pointer is then stored in the bp register.  We add 2 to the stack pointer to remove the extra bytes before we restore our context. */

#define                    portSWITCH_CONTEXT()                    __asm__
__volatile__("mov    pxCurrentTCB ,  %ax \n\t"
                    "mov    %ax，%ds \n\t"
                    "les    pxCurrentTCB，%bx \n\t"
                    "mov    %ss，%es:0x2(%bx) \n\t"
                    "mov    %sp，%es:(%bx) \n\t"
                    "call    *vTaskSwitchContext \n\t"
                    "mov    pxCurrentTCB，%ax \n\t"
                    "mov    %ax，%ds \n\t"
                    "lesl    pxCurrentTCB，%bx \n\t"
                    "mov    %es:0x2(%bx)，%ss \n\t"
                    "mov    %es:(%bx)，%sp \n\t"
                    "mov    %sp，%bp \n\t"
                    "add    $0x0002，  %bp"   )

#define                    portFIRST_CONTEXT()                    __asm__
__volatile__(″mov    pxCurrentTCB，  %ax \n\t″
                ″mov     %ax，%ds \n\t″
```

```
"lesl    pxCurrentTCB, %bx \n\t"
"mov    %es:0x2(%bx), %ss \n\t"
"mov    %es:(%bx), %sp \n\t"
"add    $0x0002, %sp \n\t"
"popw    %ax \n\t"
"popw    %ax \n\t"
"popw    %es \n\t"
"popw    %ds \n\t"
"popa \n\t"
 "iret" )
```