Y 771235

学校代码： 10246
学　　号： 022021214

復旦大學

硕 士 学 位 论 文

ARM平台嵌入式实时操作系统MiscabOS的设计与实现

院　　系： 信息科学与工程学院

专　　业： 计算机应用

姓　　名： 吴立明

指 导 教 师： 钟亦平　教授

完 成 日 期： 2005年5月20日

未经作者、导师同意
勿全文公布

摘 要

在 21 世纪，随着电子制造工艺和大规模集成电路技术的迅速发展，嵌入式微控制器逐渐走向成熟。其不断增长的计算能力和可嵌入的特点极大地拓宽了嵌入式微控制器的应用领域，广泛应用于工业控制、仪器仪表、信息家电、移动终端和无线通信领域。嵌入式微控制器计算能力的增长和对于互连能力的更高要求迫切需要有一个系统软件来管理嵌入式系统的计算能力、系统资源和外设，这个系统软件就是嵌入式操作系统。嵌入式操作系统是当前嵌入式领域研究的热点之一，相关的研究包括嵌入式操作系统的实时性、通用性、扩展性和安全性。在这些研究成果的基础上，嵌入式领域产生了相当多的嵌入式操作系统，例如商业化的 VxWorks 和开放源代码的 uC/OS 等。这些操作系统吸收了传统操作系统和嵌入式操作系统的理论研究成果，形成了各自的特色，发展了各自的专业应用领域。但彼此之间的不兼容以及专用的应用领域极大地限制了嵌入式系统用户和制造商对嵌入式操作系统的选择。

基于这一考虑，结合 ARM-GUI 项目，我们开发了 ARM 平台通用的 MiscabOS 嵌入式操作系统。本文围绕着 MiscabOS 的开发过程和实现技术展开讨论，详述了具有创新意义的三个议题：ARM 平台系统保护机制和系统调用模型的实现、可延时的中断服务程序和硬件抽象层的优化设计。本文首先提出了 MiscabOS 的系统架构，然后阐述了 MiscabOS 的具体实现细节，主要包括 MiscabOS 主要功能模块的核心数据结构和关键机制的算法实现。作为论文实验的一部分，论文介绍了 MiscabOS 的开发环境和实验环境，对 MiscabOS 性能与其他相关系统做了对比分析，给出了对比性的图形分析结果，显示出 MiscabOS 操作系统在性能上的优势。

论文的开发工作主要完成了 MiscabOS 的第一个版本，下一步的工作将开发遵循 POSIX 标准的用户程序设计接口 API 和 ROMFS 文件系统的移植，这一工作在论文结论中给出了展望。

关键词： ARM　实时操作系统　嵌入式操作系统　可延时中断服务程序

ABSTRACT

In 21st century, great progress is achieved in the field of electronic manufacture and large scale integrated circuit technology, which directly resulted in the booming market of embedded system. The embedded system with increasing power and embed ability opened up much wider application areas than ever before. Nowadays, embedded system is widely used in industrial control, measurement and meters, information home appliances, mobile terminals and wireless communications. However, application requirements for the embedded system become more and more complex, making it impossible to integrate everything into a single system any more. Obviously, powerful system software is needed to control all the computational power, system resources and peripheral devices of the embedded system. The embedded OS is one of the main system software. At present, almost all the research work focuses on the embedded OS. Possible research areas include real-time embedded OS, general-purpose embedded OS, secure and extendable embedded OS. Consequently, the research comes up with a lot of embedded OSes, such as the most successful commercial VxWorks, the open source uC/OS and eCos. Currently, there are more than 100 kinds of embedded OS available for building embedded systems, but unfortunately, each embedded OS has its own outstanding features, develop its own application fields and is incompatible with each other, which make it very hard for the users to select embedded OS for their embedded applications.

Regarding the above consideration, we developed a general purpose embedded OS for ARM architecture called "MiscabOS" as a part the "ARM-GUI" project. The thesis details three novel ideas around the MiscabOS: protection mechanism and system call model for ARM architecture, deferred interrupt service routine and optimized hardware abstract layer. Firstly, we propose the architecture of MiscabOS and describe each function module in detail at design level. Secondly, we discuss the implementation of each function module of MiscabOS at code level, mainly including kernel data structures and algorithms. Finally, as a part of the study, we introduce the development and experiment environment of MiscabOS, analyze the performance of MiscabOS compared with other embedded OSes, typically eCos and uC/OS-II and at the end of the thesis, illustrate the advantages

MiscabOS have over eCos and uC/OS-II.

So far, MiscabOS version 1 is completed, what we are going to do include the design of POSIX-conformed user interfaces API and porting of ROMFS to MiscabOS. Future works and research directions are also mentioned. The research output of this paper can guide the research of embedded OS in both theory and practice, and the experiment analysis and architecture design have verified that the research output can be put into effect.

Key words: ARM（Advanced RISC Machines） RTOS（Real-time Operating System） Embedded OS (Embedded Operating System) DSR (Deferred Interrupt Service Routine)

第1章 前言

1.1. 研究背景

随着大规模集成电路制造工艺和嵌入式微控制器技术的突飞猛进，以往以单片机为核心的工业控制领域进入了嵌入式系统时代，此外，嵌入式系统也被广泛应用在信息家电，移动终端、无线通信和军事科技等领域。在市场的推动下，嵌入式系统相关领域的研究俨然已成为当前的研究热点。

嵌入式系统被定义为：以应用为中心、以计算机技术为基础、软件硬件可裁减、适应应用系统，对功能、可靠性、成本、体积和功耗要求极为严格的计算机系统。嵌入式系统是将先进的计算机技术、半导体技术、电子技术和各个行业的具体应用相结合后的产物，这一点就决定了嵌入式系统必然是一个技术密集、资金密集、高度分散，不断创新的知识集成系统。根据嵌入式系统的构建过程，嵌入式系统可分为五个层次：芯片层，嵌入式操作系统层，中间件层，APS 层和应用软件层，其结构图如下所示：

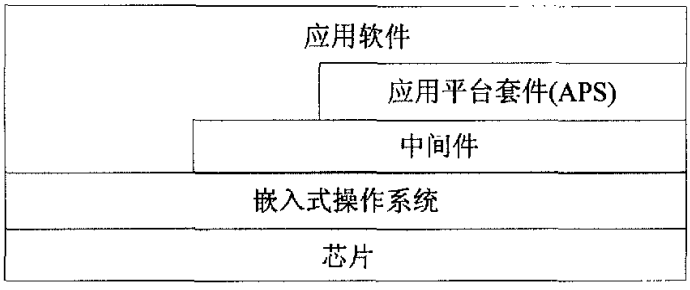

图 1-1 嵌入式系统结构层次

如图 1-1 所示，结构图的最底层是芯片层，也就是微控制器，这一层包含了嵌入式系统的“芯”，例如一些 MCU，还有 ARM 处理器，MIPS 处理器等。芯片之上是嵌入式操作系统层，嵌入式操作系统管理计算能力、系统资源和外设，向下隐藏不同芯片的细节，向上提供标准的用户程序设计接口。中间件是构建于嵌入式操作系统之上的应用程序，为上层的 APS 和应用程序提供某种服务，例如 JAVA 虚拟机提供 JAVA 代码的执行服务。APS 是 Application Platform Suite（应用平台套件）的缩写，是面向于特定应用的开发工具，例如 OpenTV 是针对机顶盒应用软件开发的套件。最上层是应用程序，其开发可以完全基于嵌入式操作系统提供的接口，也可以利用中间件或者 APS 提供的服务和工具。由图可见，嵌入式操作系统在整个嵌入式系统的结构层次中处于最核心的地位，其控制着所有的硬件资源，管理着所有应用程序，良好的嵌入式

操作系统设计能够极大地提升嵌入式系统的应用价值，拓宽嵌入式系统的应用领域。目前，市场上已经有十分成熟的嵌入式操作系统，例如商业化的 VxWorks【35】，开放源代码的 eCos【34】和 μC/OS【33】等等。但业界在构建嵌入式系统时对处于核心低位的嵌入式操作系统的选择并不多。商业化的 VxWorks 功能强大，处理高效，但价格昂贵；μC/OS 短小精悍，但扩展性差，功能较弱，无法支持关键性系统应用，而且提供的接口与整个开放源代码社区的风格不同，应用程序开发人员较难适应；eCos 是开放源代码社区的杰作，但是 GPL 的协议约束使得 eCos 的研究性大于实用性。基于这些考虑，笔者开发了 ARM 平台通用的嵌入式操作系统 MiscabOS。MiscabOS 是笔者组织的一个实验性项目 ARMGUI 的一部分，该项目的目的是为目前最通用的 ARM 嵌入式处理平台开发一个支撑性的实时操作系统 MiscabOS，并在该实时操作系统之上实现具有商业应用价值的 GUI 标准库。笔者负责 MiscabOS 实时操作系统的开发，目前已经实现了第一版，后续的改进和 GUI 标准库的开发正在进行当中。本文将结合 MiscabOS 操作系统，探讨一下如何在嵌入式处理平台上构建实时操作系统。

1.2. 研究目标

本文在嵌入式操作系统 MiscabOS 实现的基础上，着重于研究嵌入式操作系统实现的四个方面：实时性、扩展性、通用性和安全性。研究的目标是在综合目前嵌入式平台操作系统主要技术的基础上，对实时性、扩展性、通用性和安全性四个主要方面做出改进和优化。

1.3. 论文内容与结构

本文第二章描述了 ARM 平台的体系结构【15】，着重阐述 ARM 微处理器的优良设计给操作系统开发带来的便利，其中，对于与 MiscabOS 设计联系紧密的硬件特性给出重点说明。在第二章最后一节，提出了独创性的 MiscabOS 保护机制以及系统调用模型的设计思想。

第三章给出了嵌入式操作系统 MiscabOS 的系统功能框架和系统服务模型，之后各个小节剖析每个功能的实现思想。重点描述 MiscabOS 硬件抽象层的设计和可延时中断服务程序实现。

第四章着重于分析 MiscabOS 的具体实现细节。本章将列出典型的代码来说明 MiscabOS 的数据结构和系统变量。

接下来第五章介绍了 MiscabOS 的开发环境和实验环境，为前两章的内容

提供实验支持。

本文的最后，第六章对所做的理论和实践工作进行总结，提出了改进的想法和后续的研究开发计划。

1.4. 创新与成果

在开发 MiscabOS 之前，笔者花了大量的时间学习一些现有操作系统的源代码，例如 Linux，μC/OS 和 eCos。在充分理解现有实时操作系统优缺点的基础上，结合 ARM 的硬件平台特性，在 MiscabOS 的开发过程中进行了很多的创新设计和实现。这些创新点主要体现在以下三个方面：

1. 优化的硬件抽象层设计。根据 ARM 硬件平台的特性，MiscabOS 的硬件抽象层封装了系统初始化、任务切换、时钟管理和中断控制器管理。硬件抽象层的描述在第 3 章。
2. ARM 平台的保护机制实现。利用 ARM 平台提供的七种模式，实现了用户空间和内核空间的隔离，保障了系统的安全性，并在保护机制的基础上，建立了系统调用的模型。保护机制和系统调用模型的设计思想将在第 2 章详细阐述。
3. 延时中断机制的实现。为了避免中断处理程序关闭中断时间过长，通过把中断处理程序切割成两部分分别执行，成功地缩短了中断响应时间，提高了系统的实时性。在第 3 章的中断系统设计小节讨论了延时中断机制。

第2章 ARM 体系结构简介

ARM【22】(Advanced RISC Machines) 是一家微处理器公司，1990 年在英国剑桥成立，致力于精简指令集嵌入式处理器的研究和设计。1991 年推出第一个 RISC 的微处理器设计 ARM6，之后的几年中陆续推出 ARM7，ARM8，ARM9，ARM10 和 ARM11 系列微处理器，其体系结构版本也由 v1 演化到 v6，处理能力也达到了惊人的 1000MIPS（百万指令每秒）。目前 ARM 家族的处理器分为三类：应用处理器核，嵌入式处理器核和安全核。应用处理器核是在嵌入式处理器核上扩展 Cache，扩展 MMU 以及 I/O 核外围总线而构成的微处理器，主要应用在移动设备，手持终端，路由器等网络设备中；安全核是为了实现安全计算环境而设计的微处理器，几乎支持所有的加密算法，广泛应用在 SmartPhone、SmartCard、指纹识别以及各种安全扫描应用上；嵌入式处理器核则仅仅包含 ALU、移位器、调试支持部件和乘法器等基本运算功能器件的处理器内核，主要用于给第三方制造商提供基本的计算能力支持，在此基础上，很多第三方的半导体制造商推出了基于 ARM 嵌入式处理器核的应用处理器，例如 Conexant 的路由器设备芯片和 Triscend 的 A7 系列处理器。

MiscabOS 选择了 ARM 的嵌入式处理器核 ARM7TDMI【16】，这是因为 ARM7TDMI 的体系结构版本为 v4T，包含了 Thumb 指令集。而 v4 版架构是目前应用最广泛的体系结构，ARM7、ARM8、ARM9 和 StrongARM 都采用 v4 版架构，具有相当的通用性。

2.1. ARM 体系架构

ARM 处理器属于精简指令集的处理器，因此其体系结构比复杂指令集的处理器要简单很多。以下各小节分别从体系结构、存储器结构、I/O 结构、AMBA【15】总线结构四个方面介绍 ARM 体系架构。

2.1.1. ARM 体系结构

ARM 由 32 位 ALU、地址加法器、31 个 32 位通用寄存器及 6 个 32 位状态寄存器、32×8 位乘法器、32×32 位桶形移位器、指令译码器和数据/地址寄存器组成。其体系结构图如下所示：

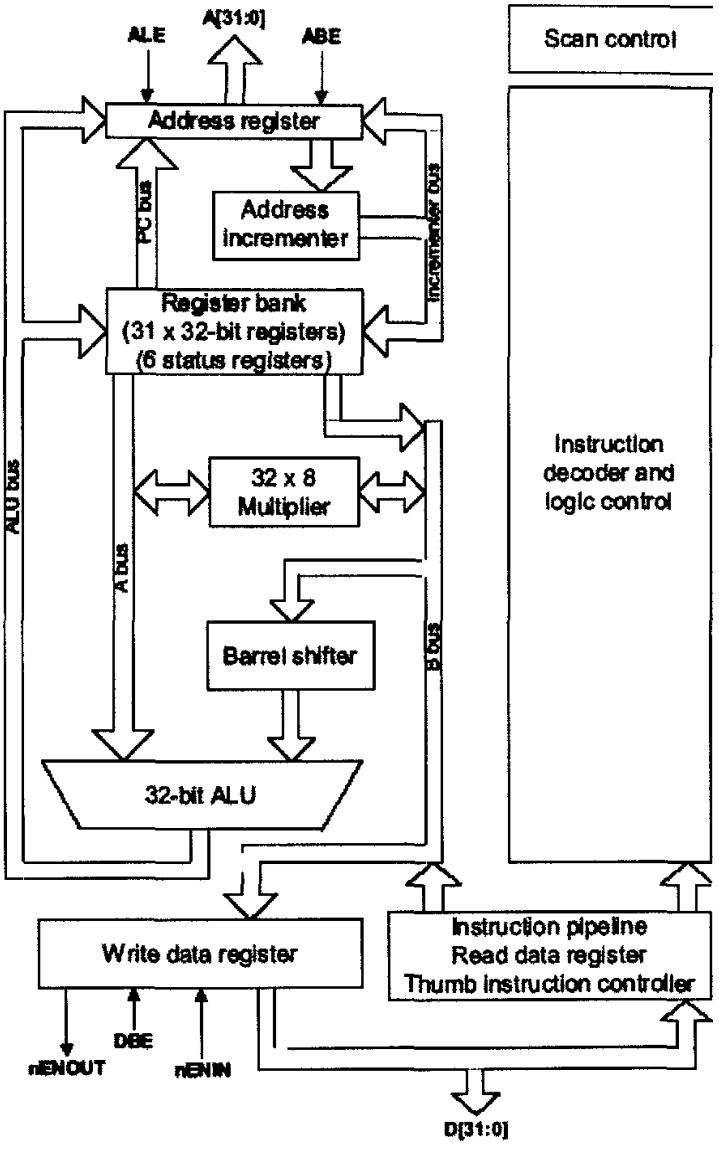

图 2-1 ARM7TDMI 体系结构图

ARM7TDMI 内部实现了 3 级流水线，采用流水线互锁、旁路和分支预测克服了流水线中的相关，极大地提高了流水线的吞吐量。此外，独立的地址加法器也节省了 ALU 的时间，地址的加法通过加法器来完成，ALU 只负责操作数的各种运算。最有特色的桶形移位器，可以把乘法操作或者加减法操作转换为移位操作，缩短了算术指令的执行时间。ARM7TDMI 还包含了对硬件调试的支持，通过标准的 JTAG【21】协议，能够对目标代码进行在线调试，缩短了软件的开发时间，降低了开发成本。

2.2. ARM 存储器结构

2.2.1. ARM Cache 结构

ARM 的地址线是 32 位的，所以可以寻址的空间是 4GB。ARM 架构的处理器一般片内没有配置 RAM 和 ROM，因此，系统所需要的 RAM 和 ROM 都要通过总线来外接。 此外，一些高端的应用处理器还配置的 Cache。常用的 Cache 有指令和数据统一的 Cache 和指令和数据分离的 I-Cache 和 D-Cache。下面给出 Cache 结构和性能之间的一个关系表：

表 2-1 Cache 性能表

Cache 结构	性能
无 Cache	1
仅 I-Cache	1.95
仅 D-Cache	1.13
I-Cache 和 D-Cache	2.5

目前新型的 ARM 处理器大多数都采用独立的指令和数据 Cache 的 Cache 结构。例如 Intel 公司的 SA1110 处理器内含有 16K 字节的 I-Cache 和 8-K 字节的 D-Cache。MiscabOS 选择的目标平台包含有 8KB 的 4 路组相联的 Cache 结构，其 Cache 既用来存放数据，也用来存放指令。

2.2.2. ARM I/O 结构

ARM 架构处理器核内部没有 I/O 模块，I/O 功能必须通过 AMBA 总线来扩充。本节将结合应用处理器来说明 I/O 的功能。ARM 系列的应用处理器以及第三方的应用处理器的 I/O 结构一般有三种：存储器映像 I/O、直接存储器存取 DMA、中断 IRQ 和快速中断 FIQ。

MiscabOS 的实现中采用了存储器映像 I/O 和中断 IRQ 两种 I/O 结构，用来支持设备驱动程序的 I/O。然而，GUI 标准库的实现需要 DMA 的支持，例如支撑 LCD 数据的大批量传送。这一特性会在 MiscabOS 的后续版本中加入。

2.3. ARM 指令系统

ARM7TDMI 处理器同时支持两种指令集，分别是 32 位的 ARM 指令集和 16 位的 Thumb 指令集。Thumb 指令集是 ARM 指令集的一个子集，包含了 ARM

指令集中最常用的指令。在我们的嵌入式应用程序中，可以根据应用程序运行的硬件环境来选择合适的指令集。就嵌入式操作系统运行的硬件环境而言，内存子系统的配置有两个参数：容量和总线宽度。容量表示内存的大小，比如8MB，32MB；总线宽度表示内存的一次读或写操作能够传输的比特数，比如16 位，32 位。当运行环境的内存容量非常小时，Thumb 指令集是最好的选择，因为 32 位的空间，能够容纳两条 Thumb 指令，却只能容纳一条 ARM 的指令。当运行环境的总线宽度是 16 位时，Thumb 指令集的性能要优于 ARM 指令集，因为 ARM 指令的长度是 32 位，每取一条 ARM 指令，需要访问两次内存；而 Thumb 指令的长度是 16 位，每取一条 Thumb 指令，只需要访问一次内存。一般而言，Thumb 代码的尺寸是相同功能 ARM 代码的 65%，而且 Thumb 代码在 16 位总线上的性能是 ARM 代码的 160% 【16】。但 Thumb 指令集毕竟只是 ARM 指令集的子集，很多 ARM 指令在 Thumb 代码中无法使用，功能上弱于 ARM 指令集。由于 ARM 平台同时支持 ARM 和 Thumb 指令集，我们可以在应用程序中混合使用 ARM 和 Thumb 指令进行程序设计，但这会带来额外的状态切换开销。所以如果内存容量相当，并且采用 32 位地址线时，应毫不犹豫选择 ARM 指令集。具体地，对于 MiscabOS，其运行的硬件环境是 32 位地址线，8MB 的 Flash，32MB 的 SDRAM；为此，MiscabOS 完全采用 ARM 指令集。

2.4. ARM 程序设计模型

ARM 的程序设计模型与 MiscabOS 的开发联系十分紧密，所以本节将会用较大的篇幅来说明。ARM 程序设计模型包括 ARM 的寄存器，尤其是程序状态寄存器，ARM 处理器的状态，ARM 处理器的运行模式和 ARM 的异常模型和异常处理机制。

2.4.1. 寄存器

ARM 有 37 个 32 位的寄存器，其中包括 31 个通用寄存器和 6 个状态寄存器。

2.4.2. 处理器状态

ARM 处理器能够运行在两种状态，ARM 状态和 Thumb 状态。当处理器处于 ARM 状态时，每一个指令周期从内存中取出 32 位的内容送到 ARM 指令

译码器中；处于 Thumb 状态时，每一个指令周期从内存中取 16 位的内容送到 Thumb 指令译码器中。如果应用程序完全由 ARM 指令构成，那么在程序的运行过程中不需要切换处理器的状态，否则需要使用 ARM 指令 BX【16】来进行状态之间的切换。

2.4.3. 处理器模式

处理器模式是 ARM 处理器最鲜明的特点之一。ARM7TDMI 处理器支持七种处理器模式：

1. 用户模式
2. 系统模式
3. 管理模式
4. 中断模式
5. 快中断模式
6. 中止模式
7. 未定义模式

除开用户模式之外的其它六种模式都属于特权模式，能够执行所有指令，访问所有资源。用户模式则是受限模式，访问限制资源的时候必须先按照特殊的规定切换到特权模式，在下一节，将对这种特殊的规定做出详尽的解释。用户模式和特权模式类似于 Linux 系统中的用户模式和系统模式【10】【11】，也类似于 Windows 系统中的 Ring3 和 Ring0【4】【5】的运行级别。MiscabOS 的实现中应用程序运行在用户模式，而操作系统运行在特权模式。

2.4.4. 异常模型和异常处理机制

ARM 处理器的异常模型和其运行模式是紧密联系的。ARM 处理器实现了七种异常，分别为：

1. 重启异常
2. 数据中止异常
3. 预取中止异常
4. 未定义指令异常
5. 软中断异常
6. 中断异常
7. 快中断异常

在 Intel 系列的 CPU 中，CPU 运行模式有两种，系统模式和用户模式，分

别对应于 Ring0 和 Ring3，保存在段寄存器，通过比较段寄存器 DPL 和段选择子中的 RPL 值来确定访问是否合法【10】【11】，而在 ARM 模式中，运行模式则是由程序状态寄存器 CPSR 的模式位表示的，通过对模式位的比较，就能够确定访问的合法性。同样在 Intel 系列 CPU 中，中断服务程序，系统调用都是运行在 Ring0，即系统模式；而 ARM 则把中断服务程序和系统调用与系统模式分离，中断服务程序运行在中断模式，系统调用运行在管理模式，通过提供更高粒度的模式，使得应用程序的设计变得简单，保护机制的实现相对简单。以下简单介绍 ARM 的异常处理机制，之后讨论在 ARM 平台上的保护机制的实现。

重启异常是用户开启电源或者按下重启键之后发生的异常，ARM 处理器在收到重启异常之后，进入管理模式，之后跳到异常向量表中，选择重启异常的异常处理程序执行。

数据中止异常在存储器访问失败的情况下发生，ARM 处理器进入中止模式，之后进入异常向量表，选择数据中止异常的异常处理程序执行。数据中止异常可以用来实现虚拟内存的机制【15】【16】。

预取中止异常是当取指令失败的情况下发生的，此时 ARM 处理器进入中止模式，之后进入异常向量表，选择预取中止异常的异常处理程序执行。

未定义指定异常是译码器无法解释当前指令时发生的。ARM 处理器进入未定义模式，然后选择异常向量表中的未定义异常处理程序执行。未定义指令异常可以用来实现软件模拟的协处理器【15】。

中断是当外部设备发出中断信号后产生的异常，ARM 处理器进入中断模式，并执行保存在异常向量表中的中断服务程序执行。中断异常实现中断驱动 I/O 的基础，在 MiscabOS 的 I/O 系统的实现完全依赖于中断异常。

ARM 处理器广泛应用于嵌入式系统领域的一个原因就是在 ARM 的处理器上能够很方便地构建实时系统。ARM 处理器提供的快中断异常是针对那些对实时性要求十分高的系统的。通过快中断，系统能够在最短的时间内进入快中断处理程序。与中断异常相似，快中断异常也是由于外部设备中断源引发的，之后 ARM 处理器进入快中断模式，并选择快中断异常处理程序执行。

ARM 处理器的异常向量表如下所示：

表 2-2 ARM 处理器异常向量表

异常	入口地址	处理器模式	优先级
重启	0x00000000	管理模式	0
未定义指令	0x00000004	未定义模式	5
软中断	0x00000008	管理模式	5

预取中止	0x0000000C	中止模式	4
数据终止	0x00000010	中止模式	1
保留	0x00000014		
中断	0x00000018	中断模式	3
快中断	0x0000001C	快中断模式	2

从表 2-2 可以看出，在异常向量表中的每个入口地址上都存放着一条跳转指令，当异常发生时，处理器执行跳转指令，就能跳到异常处理程序处执行。而值得关注的是，快中断异常的入口在异常向量表的最末。程序设计者可以直接把快中断异常处理程序放在入口地址处，就不需要通过跳转指令跳转到快中断异常的处理程序了，进一步地缩短了中断响应时间。但一般而言，除非对于特别紧急的工业控制应用，否则不需要使用快中断异常，而且由于快中断异常无法利用中断控制器进行中断分级，也就是说无法在一个快中断上挂载多个外设，所以在 MiscabOS 的设计中，我们选择使用了中断异常，没有使用快中断异常，我们把中断异常的服务程序直接存放到 0x00000018 的地址处，这样在中断发生时，中断异常处理程序立刻就会被执行。

嵌入式操作系统有别于标准的桌面系统，后者往往包含众多复杂的功能，例如文件系统、内存管理、协议栈等等，而且从程序执行角度来看，还包含程序加载器和程序重定位器，所有的可执行文件存放在文件系统里，利用程序加载器和程序重定位器把可执行文件加载进内存并执行。而嵌入式操作系统相对而言则要简单得多，仅包含标准桌面操作系统的关键功能，例如任务管理、中断管理和设备管理。在嵌入式操作系统中，用户程序是和操作系统绑定在一起的，形成单一的可执行文件，在系统运行时该文件被执行，用户程序在系统初始化完成后开始运行。在这样的操作系统的实现中，没有用户空间和内核空间的概念，所有的代码都工作在内核空间，用户代码可以访问系统的任何资源，可以直接控制设备，系统没有任何的保护机制和安全性，典型的 uC/OS-II 和 eCos，就是这类操作系统的代表。而 MiscabOS 利用 ARM 平台提供的异常模式实现了用户空间和内核空间的隔离，并在空间分离的基础上，建立了系统调用的服务请求模型，用户代码对于系统关键资源的访问完全通过系统调用来完成，极大地提升了系统的安全性和健壮性。

MiscabOS 保护机制的原理框图如下所示：

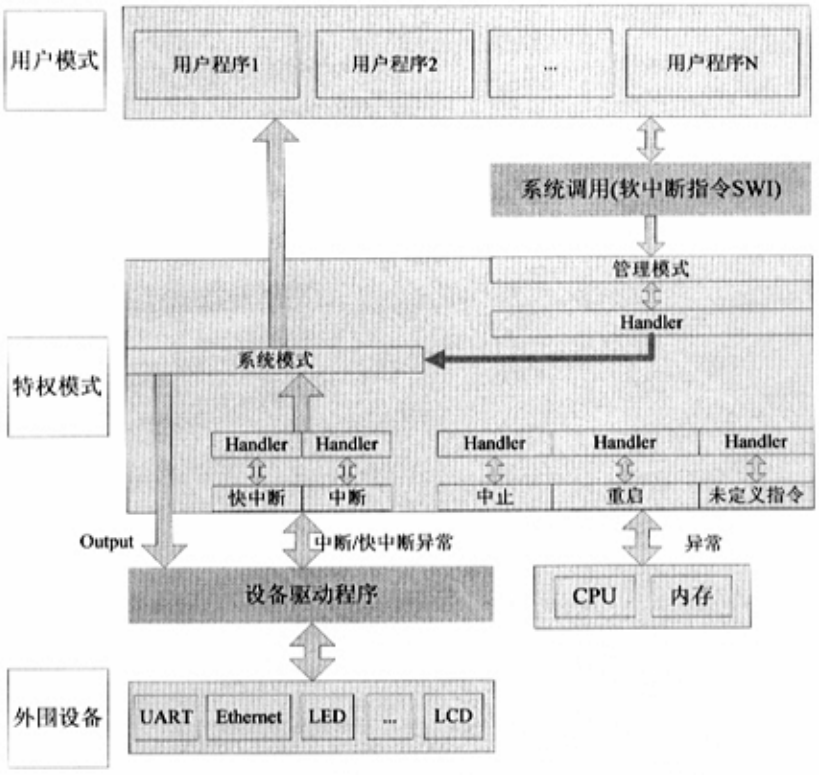

图 2-2 ARM 处理器保护模式的实现机制

在讨论保护机制之前，我们对 ARM 处理器的七种模式作进一步的讨论。ARM 处理器有其中模式，其中用户模式是受限模式，其他模式为特权模式。在讨论了异常之后，我们需要再次划分所有特权模式。在六种特权模式中，系统模式并非是因为异常发生而导致处理器进入的模式，除此之外，其余五种模式都是因为相应的异常发生，而导致处理器模式的切换。异常的发生具有不确定性，处理器在执行异常处理程序时，可能再次发生异常，而导致处理器重新进入异常处理程序，前面的执行结果将会丢失，执行现场被破坏。为了避免这种情况，异常模式应该迅速切换到系统模式执行，然后执行异常模式的处理程序。

在图 2-2 中，我们把嵌入式操作系统的保护机制分为三个层次：用户模式，特权模式和外围设备。用户程序在用户模式运行，当需要请求系统服务时，调用操作系统提供的系统调用接口，系统调用会产生一个软中断异常，处理器随之进入特权模式，在特权模式中，系统完成用户请求的服务，并返回用户模式执行。由上面的讨论知道系统在服务用户程序之前，必需要先切换到系统模式。切换完成之后，我们就可以进行一些动作来服务用户程序的请求，完成之后，

由系统模式再切换回用户模式。举个简单的例子，用户程序需要进行 I/O 操作，首先必须调用软中断指令进入管理模式，之后切换到系统模式，最后在系统模式下，调用驱动程序提供的接口函数进行输入输出，把结果存放在用户指定的内存地址处，最后返回到用户模式。这一过程也是典型的系统调用的执行过程，也正是在保护机制的基础上，MiscabOS 实现了系统调用的服务请求模型。

第3章 MiscabOS 的系统框架

本章给出 MiscabOS 的系统功能框架，并从体系结构角度给出每一个模块的功能描述和设计思想，在第四章中将会结合设计思想和论文的开发实践来讨论具体的实现。

3.1. MiscabOS 综述

MiscabOS 是在 ARM 嵌入式处理器平台上开发的实时操作系统，包含有特定于 ARM 平台的硬件抽象层、任务调度器、任务管理器、任务间通信、I/O 系统以及设备驱动程序，其结构图如图 3-1 所示：

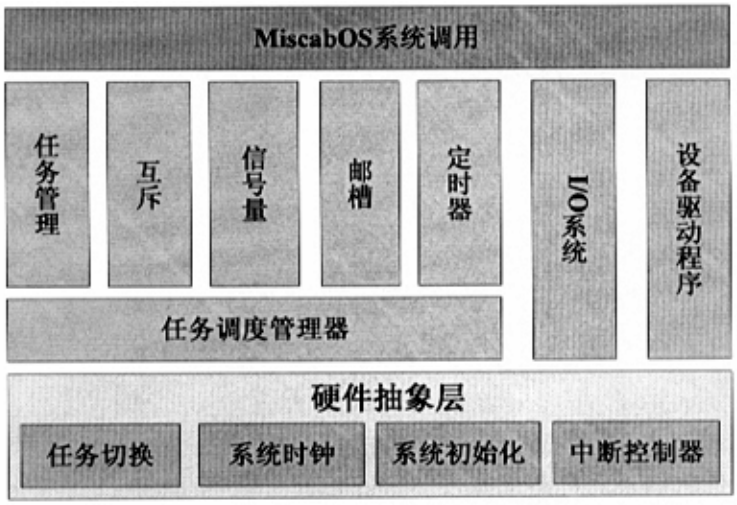

图 3-1 MiscabOS 体系结构图

在下面的章节中将会针对 MiscabOS 体系结构的每一部分进行分析，提炼出设计思想。

3.2. 硬件抽象层

从图 1-1 中可以看到，在嵌入式系统的结构层次中，嵌入式操作系统直接控制着底层的硬件设备，充当用户和计算机系统的界面和接口，使得用户不需要关心底层的硬件细节就可以很方便地操纵整个系统。根据是否直接与特定的硬件交互的标准，可以把操作系统分为两个部分，分别是与硬件有关的硬件抽象层和与硬件无关的操作系统内核。硬件抽象层与操作系统的内核关系如图

3-2 所示：

OS内核	OS内核	OS内核	OS内核
x86的HAL	ARM的HAL	MIPS的HAL	……
x86	ARM	MIPS	……

图 3-2 操作系统内核与 HAL 关系图

通过把操作系统内核与硬件抽象层的隔离，在移植操作系统到其他处理器平台时，只需要重写硬件抽象层，极大地提高嵌入式操作系统的可移植性和通用性。尤其是在当前嵌入式领域标准混乱，互不兼容的局面下，优化地设计硬件抽象层具有重要的现实意义。

由图 3-1，MiscabOS 的硬件抽象层分为四部分：任务切换、时钟管理、系统初始化和中断控制器。分别在以下子章节进行说明。

3.2.1. 任务切换

多任务和分时是现代操作系统的重要标志，MiscabOS 也支持这一特性。多任务操作系统的一个重要的职责就是在适当的时候进行任务切换，也就是保存即将换出任务的执行上下文，然后恢复即将换入任务的上下文。由于不同处理器的寄存器设置不同，导致了任务的上下文在不同的处理器平台上也有不同的定义。在 ARM 平台上，任务的上下文是指 ARM 处理器的通用寄存器 r0-r12, r14, r15 和程序状态寄存器 cpsr 组成的。任务的切换有两种情况：任务和任务之间的切换以及中断服务程序和任务之间的切换，我们先来探讨第一种情况。任务和任务的切换是通过对任务的堆栈进行操作完成的。任务的数据结构描述任务控制块(TCB)中包含有其堆栈的指针，把两个任务的堆栈指针传递给任务切换函数。任务切换函数根据换出任务的堆栈指针，把当前的上下文压栈，保存现场，之后，利用换入任务的堆栈，把其上下文从栈弹出到 ARM 寄存器中，就完成了现场恢复。中断服务程序和任务之间的切换要复杂一些。当中断发生时，处理器内部产生中断异常，系统进入中断模式，并跳入中断服务程序执行。在中断服务程序里面，首先需要保存被中断任务的现场。而被中断的任务是在用户模式或者系统模式下执行的，所以必须切换模式到用户模式或者系统模式，然后保存相应的 ARM 寄存器，之后返回到中断模式，执行中断服务程序。在中断服务程序执行完毕后，根据被中断任务的堆栈指针恢复其执行现场。

3.2.2. 时钟管理

MiscabOS 的任务支持延时执行，也就是说我们可以挂起一个任务，然后在一段时间间隔后，恢复其执行，这个功能的实现需要系统时钟的支持。在系统启动时，MiscabOS 以给定的时间间隔初始化时钟，之后系统时钟经过每一个时间间隔时就会中断处理器，处理器转而执行时钟中断服务程序。在中断服务程序中修改被挂起任务的延时时间，如果延时时间变为零，就把任务转移到就绪队列中等待调度。对系统时钟的一些操作以内核函数形式存在，供用户程序和系统程序调用。时钟初始化包括设置中断时间间隔，注册时钟中断服务程序，启用时钟中断。

3.2.3. 系统初始化

ARM 处理器在加电后产生重启异常，进入管理模式并执行重启异常处理程序。初始化代码放在重启异常处理程序中，主要进行不同模式的堆栈分配。MiscabOS 支持用户模式/系统模式(共享堆栈)【15】、管理模式和中断模式，所以需要为三种模式分配堆栈空间。之后加载操作系统初始化函数，进行存储器初始化、时钟初始化、中断控制器初始化、加载设备驱动程序、初始化外围设备、挂载文件系统、创建用户任务和空闲任务以及设置系统运行模式，最后启动任务调度器，这时操作系统才真正运行起来。系统启动的顺序如图 3-3 所示：

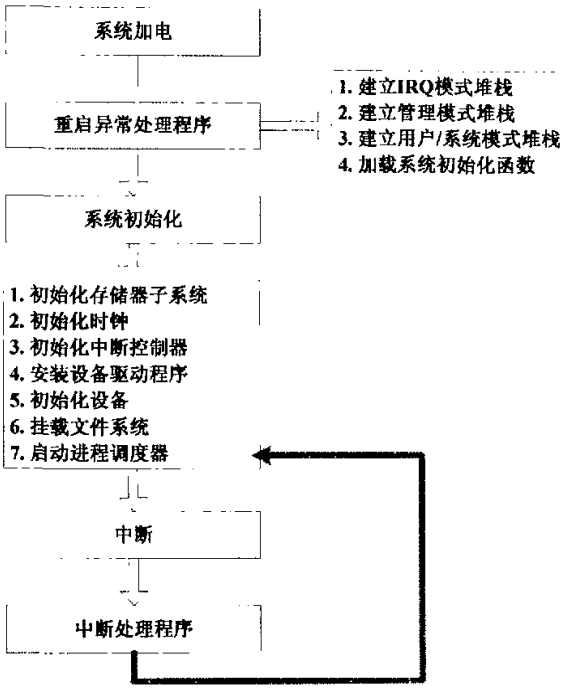

图 3-3 系统启动顺序图

3.2.4. 中断控制器

ARM 处理器核 ARM7TDMI 仅有一个中断信号(IRQ)引脚，与外部的中断控制器相连【23】【28】。MiscabOS 选用了 Triscend 公司【29】的 A7S20 开发板【23】【25】，支持多达十六级的中断优先级别，并支持中断的屏蔽。在 2.2.2 节中提到 MiscabOS 完全采用了中断驱动的 I/O 和 DMA 两种 I/O 机制，而中断驱动 I/O 的实现正是借助于 A7S20 开发板上的中断控制器实现的。中断控制器的初始化包括检测中断控制器的工作状态、注册中断服务程序和启动中断。A7S20 开发板的中断控制器还提供三个可编程的中断向量【28】，在开发板上，有一个可编程的中断向量连接到板载的 24 键键盘，这样除了可以通过串口和 MiscabOS 交互外，还可以利用键盘。

3.3. 任务模型设计

结合传统操作系统的设计理念，首先定义了 MiscabOS 中的任务状态，并给出了状态之间转换的模型。任务状态图如图 3-4 所示：

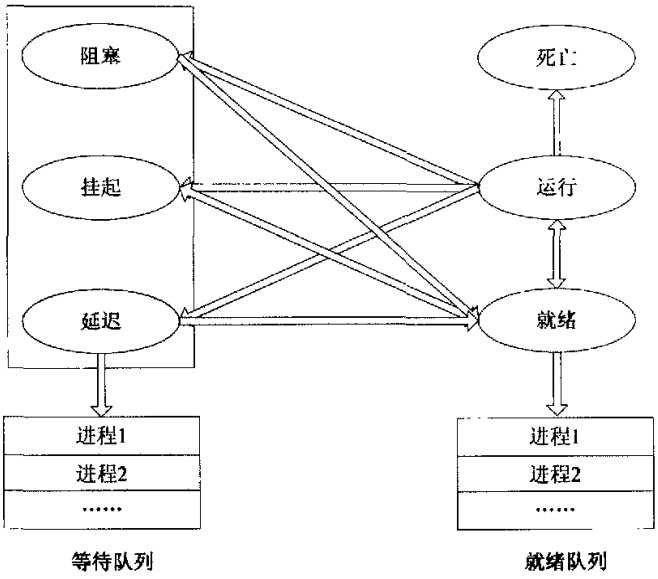

图 3-4 任务状态图

以下是任务之间转换的机理：

1. 运行-死亡：任务执行完毕，任务进入死亡态；
2. 运行-就绪：运行任务被具有更高优先级的任务或者中断服务程序抢占，这也是实时操作系统最重要的一个特征；
3. 运行-阻塞：任务因为等待一些事件的发生而导致无法继续执行。这些事件可以是 I/O 操作的完成，互斥量被解锁，信号量被释放；
4. 运行-挂起：任务可以挂起它本身，也可以挂起任何就绪的、延迟的和阻塞的任务。但已经挂起的任务不能再次被挂起；
5. 运行-延迟：当前运行的任务希望在一定时间间隔后重新运行；
6. 就绪-运行：任务被调度器选中执行；
7. 就绪-挂起：就绪的任务被其他任务挂起；
8. 阻塞-就绪：阻塞任务等待的事件已经发生；
9. 挂起-就绪：挂起的任务重新被重置进就绪队列；
10. 挂起-就绪：当延时的时钟滴答过去以后，任务重新进入就绪队列等待调度。

丰富的任务状态满足了各种各样的复杂应用的需求，使得用户可以更灵活地开发嵌入式应用。除了任务状态以外，必须要有优先级来指明任务的重要性。此外还有用于指向任务栈顶的堆栈指针。在任务状态的转换过程中，有两个非常重要的队列：等待队列和就绪队列。当任务的状态变为阻塞、挂起、延迟或者三者的任意组合时，任务被放入等待队列。在中断服务程序中会扫描这个队

列，找出能够运行的任务，置入就绪队列。调度器仅仅扫描就绪队列，每次从就绪队列选出一个任务执行。如何在就绪队列中选出要执行的任务属于调度策略的范畴，将会在 3.5 节详细阐述，任务的数据结构定义将在第四章说明。大多数桌面机系统，例如 Windows 和 Linux 都有数个队列，分别用于存放各种不同状态的任务，这些系统的调度器和中断服务程序设计十分复杂，而 MiscabOS 仅采用两个队列，简化了调度器的设计，也使得开发人员编制设备驱动程序的时候相对容易得多。

3.4. 中断机制设计

中断机制的设计是嵌入式操作系统最为关键的部分之一。嵌入式操作系统有两个最为重要的衡量标准：其一是系统的实时性，其二是可抢占式的调度策略。实时性是指的中断响应时间，ARM 处理器的设计为快速的中断响应提供了硬件基础，MiscabOS 很好地利用了这一基础，并在中断机制的设计上做出创新，进一步缩短了系统的中断响应时间，提供了优秀的实时性。

在几乎所有的操作系统实现中，中断服务程序的执行必须是在中断关闭的情况下执行的，待到中断服务程序执行完毕后，中断才会被重新打开。这样就导致了一个问题：如果中断服务程序执行时间过长，那么中断就会长时间处于关闭中断，此时若有新的中断信号到达，则得不到响应。如图 3-5 描绘了传统中断机制的原理图，可见在中断关闭的期间，外围设备发生的中断信号将得不到处理器的响应。

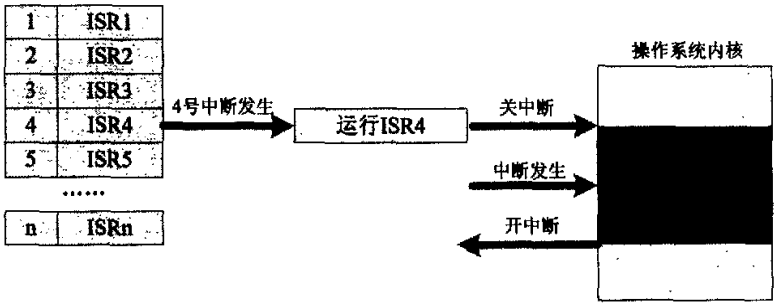

图 3-5 传统中断服务模型

在嵌入式系统中，中断信号的丢失是非常危险的。对于电厂控制来说，汽轮机的转子转速、锅炉内的气压是整个系统良好运行的最关键的标准。如果转子超速，锅炉过压产生的中断信号得不到系统响应的话，将会酿成巨大的灾难。所以在嵌入式系统的中断机制实现上，必须以最短的中断响应时间作为最高准则。MiscabOS 的中断机制通过把中断服务程序分为两部分来解决可能的中断丢失问题。MiscabOS 的中断机制模型如图 3-6 所示：

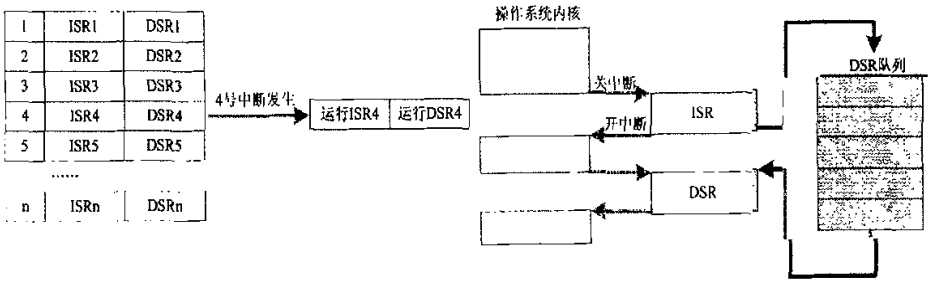

图 3-6 MiscabOS 的中断机制模型

在 MiscabOS 中，中断服务程序被分为两部分：ISR 和 DSR。DSR 是 Deferred Interrupt Service Routine 的缩写，意思是可延迟的中断服务程序。ISR 在中断关闭状态下执行，仅仅进行一些关键部分的设置，操纵一些临界的数据结构，之后把繁重的任务，例如 I/O 操作，信号量操作等交给 DSR，把 DSR 放入 DSR 队列，最后打开中断。DSR 是在中断打开状态下执行，被当作普通任务进行调度，但具有比普通任务高的优先级，在 DSR 队列不空的时候，优先执行 DSR 队列中的元素。中断服务程序执行的具体过程如下：

1. 关闭中断
2. 转入中断服务程序 ISR
3. 在 ISR 执行完毕，返回之前，将 ISR 对应的 DSR 置入 DSR 队列
4. 从 ISR 返回

如果没有中断发生，调度器选择 DSR 队列中的 DSR 执行，直到队列为空通过在 MiscabOS 中部署这样的中断机制，成功地减少了中断关闭的时间，避免了可能的中断信号丢失。

在系统中引入新的机制，无疑会带来性能上的损失。在可延时的中断服务程序模型中，对于 DSR 的调度和 DSR 队列的管理都需要耗费处理器时间。但是考虑到系统的实时性和中断响应时间，进行这样的设计是利大于弊的。尤其是当硬件系统的运行速度和计算能力大幅提高时，引入 DSR 带来的系统开销可以忽略不计。

DSR 的引入为驱动程序的编制者提供了方便。MiscabOS 完全采用中断驱动的 I/O，在开发驱动程序时，开发者需要为要驱动的设备注册相应的中断服务程序。如果该设备的 I/O 操作非常快，可以考虑把 I/O 操作完全放在 ISR 中，避开 DSR，以节省不必要的系统开销；如果设备的 I/O 操作速度慢，则可以把中断服务程序分为 ISR 和 DSR，把繁重的 I/O 任务交给 DSR 完成。这种灵活性在嵌入式系统应用的设计中显得尤为重要。

3.5. 任务调度模型

任务调度是现代多任务操作系统的重要组成部分，也是对操作系统分类的根据之一。任务调度模型设计的主要目的是缩短任务调度时间，提高系统整体运行效率。考虑到嵌入式系统的应用特点，MiscabOS 的任务调度采用了两种调度机制：基于动态优先级调度和时间片轮转调度。

基于动态优先级的调度策略是指每个任务在创建时根据其重要性被赋予一个优先级，重要的任务赋予高的优先级，调度器总是从系统的等待队列中选择具有最高优先级的任务执行，如图 3-7 所示。在基于动态优先级的调度策略中，任务的优先级是可以被改变的，通过修改任务的优先级，就可以更精确地控制任务的运行。 时间片轮转也叫 Round-Robin【1】【3】【4】，该策略是指在系统的等待队列中，对于具有相同优先级的任务，让每个任务轮流执行一段指定时间，来保证相同优先级任务间的公平性，如图 3-7 所示。

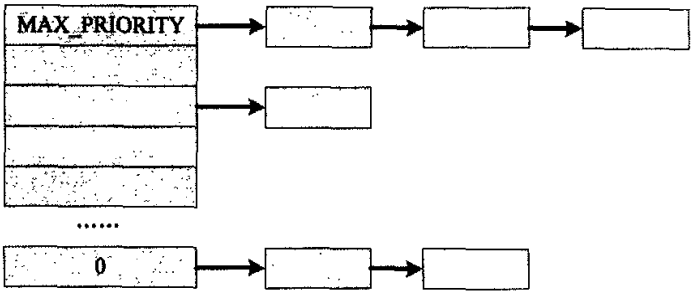

图 3-7 任务就绪队列图

MiscabOS 调度器调度的对象除了等待队列中的任务之外，还有 DSR 队列。DSR 队列中的可延迟中断服务程序优先级高于一般任务，调度器会优先调度 DSR 队列中的元素执行。这是因为在系统的等待队列中可能有任务在等待 I/O 操作的完成，而 I/O 操作的执行一般都是放在 DSR 中进行的，所以优先执行 DSR 能够保证尽可能多的任务处于就绪状态。

3.6. 任务间通信机制

在现代的多任务操作系统中，IPC(进程间通信)机制是进程之间的同步协作的理论基础。为了方便进程之间交换信息，一般的操作系统都提供一种以上的进程间通信机制。以 Linux 操作系统为例，支持六种 IPC，分别为：

1. 管道
2. FIFO(命名管道)

3. 共享内存
4. 文件映射
5. 消息队列(邮槽)
6. 套结字

uC/OS 系统操作系统仅仅提供了消息队列一种进程间通信机制，而 MiscabOS 支持互斥、信号量、Mailbox 和定时器四种，极大地方便了多任务应用程序的开发和驱动程序的编写。以下分别描述这四种进程间通信机制，并给出具体的应用。本文中，任务和进程是指同一个概念，在以下的描述中会混用。

3.6.1. 互斥

互斥是指多个进程共同访问同一系统资源时产生的竟态问题，这种资源叫做临界区【4】【5】，MiscabOS 的互斥机制通过把多个进程的访问序列化来解决这一问题。进程在访问临界区时首先需要对临界区上锁，如果上锁成功，那么进程进入临界区；如果临界区已经被其他进程锁住，那么当前进程必须等待，直到临界区解锁为止。MiscabOS 对于互斥的解决方法包括临界区解锁和上锁两部分，以下给出具体的步骤。对临界区的上锁包括以下步骤：

1. 判断临界区是否被锁住
2. 如果已经锁住，则阻塞当前进程，把该进程置入互斥量的等待队列
3. 重新启动调度器

对临界区的解锁包括以下步骤：

1. 释放临界区
2. 从互斥量的等待队列选出等待的进程，把等待的进程置入就绪队列
3. 重新启动调度器

在第四章会从代码的级别对进程间通信的机制做出进一步阐述。

在完全基于优先级的调度系统中，对于互斥量的使用应该注意一个问题，那就是可能出现优先级反转(Priority Inversion)【9】，也就是可能导致地优先级的任务在运行，而高优先级的任务被阻塞，违背了实时操作系统调度器的设计准则。图 3-8 是出现优先级反转情形的一个图例：

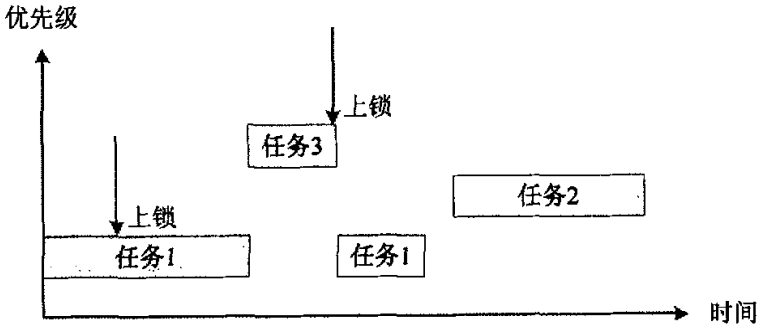

图 3-8 优先级反转图例

图 3-8 是任务优先级发生反转的一个图例，其中 X 轴是时间，Y 轴是优先级。从图中可以看出，图例中有三个任务：任务 1、任务 2 和任务 3。优先级最高的是任务 3，最低的是任务 1。以下给出发生优先级反转的时间顺序流：

1. 任务 1 被调度器调度运行
2. 任务 1 需要访问临界区，对临界区上锁
3. 任务 3 优先级高于任务 1，于是抢占了任务 1，任务 1 被置入就绪队列
4. 任务 3 也需要访问临界区，因临界区已上锁而阻塞而被置入等待队列
5. 调度器调度就绪队列，选择任务 1 执行
6. 任务 2 优先级高于任务 1，于是抢占了任务 1，任务 1 被置入就绪队列
7. 任务 2 运行

在本例中，调度器工作的结果是低优先级的任务 2 在运行，而高优先级的任务 3 被阻塞，这就是优先级反转的概念。优先级反转违背了基于优先级的调度策略的设计思想，是所有可抢占式的实时系统共同面临的问题。对于优先级反转的解决方案有两种：分别是优先级继承和优先级置顶。MiscabOS 支持两种策略，可以由用户根据具体的应用系统进行定制选择。优先级继承的图例如下：

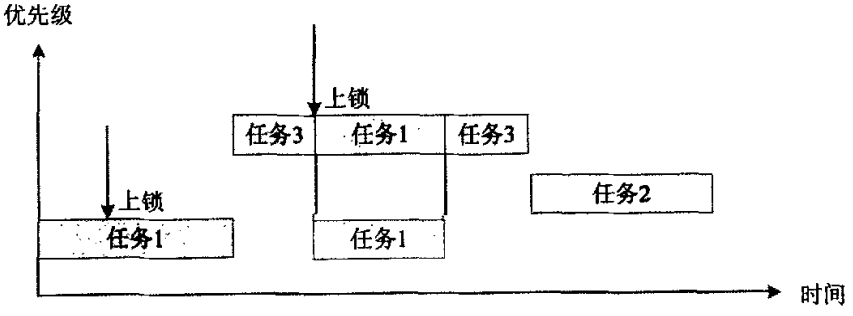

图 3-9 优先级继承图例

采用优先级继承策略后，系统任务调度器的工作顺序流程如下：

1. 任务 1 被调度器调度运行
2. 任务 1 锁住临界区
3. 任务 3 优先级高于任务 1，抢占任务 1 并把任务 1 置入就绪队列
4. 任务 3 尝试锁住临界区，失败后阻塞，并把任务 1 的优先级修改为任务 3 的优先级
5. 任务 1 被调度运行，运行完毕之后释放临界区
6. 任务 3 访问临界区
7. 任务 3 运行完毕只有任务 2 被调度运行

可以看出，采用优先级继承策略后，在任意时间点，当前运行的任务的优先级都是就绪队列中最高的，遵循了基于优先级调度策略的思想。此外，我们通过把任务 1 的优先级提高到任务 3 的优先级，避免了优先级反转问题的出现，这里我们称任务 1 继承了任务 3 的优先级，其核心思想是让锁住临界区的任务优先被调度，这样临界区就可以尽可能快地被释放。还可以看出，虽然任务创建时的优先级是固定的，但 MiscabOS 允许在执行过程中动态改变任务的优先级，这就是动态优先级的概念。

优先级置顶策略是优先级继承策略的一种极端情况。在上例中，把任务 1 的优先级置为系统能够支持的最大优先级，保证了任务 1 在没有中断出现的情况下最先被调度执行，从而可以在最短的时间内释放临界区锁。

在两种抗优先级反转策略的实现中，都动态地修改了任务的优先级，在完成了临界区的操作之后，所有被修改的优先级都应该被恢复到之前的状况。在任务的进程数据结构描述中，MiscabOS 采用两个变量分别保存修改前和修改后的优先级，从而可以在执行结束后恢复任务的优先级。在 MiscabOS 的实现中，用户可以根据不同的应用，通过更改配置文件来定制系统支持哪一种抗优先级反转的策略。

3.6.2. 信号量

互斥是用来顺序化任务对临界区的访问，而信号量则主要用来实现一些任务同步模型，例如读者写者模型，生产者消费者模型等等。在实现带缓冲的 I/O 模块时，这些模型非常高效。MiscabOS 信号量的实现十分简单，其数据结构包括一个整型计数器和一个等待队列，操作数据结构的方法包括 wait 和 signal。下面以打印机的等待队列为例来说明如何使用信号量。

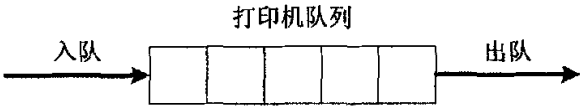

图 3-10 打印机队列的信号量实现图

对打印机队列而言，生产者是用户程序，消费者是打印机。当打印机队列未满时，生产者将打印任务放入打印机队列；当打印机队列不空时，打印机从队列中取出打印任务执行。对于这样的一个应用，我们应该维持一个全局的变量用来表示打印机队列中的任务数量，还有一个等待队列，当任务阻塞时被置入该队列，此外，打印机队列的任务数量属于临界资源，应该用互斥来顺序化多个任务对它的访问。基于这一点，生产者的实现应该有如下的结构：

1. 生产者询问打印机队列是否已满
2. 若打印机队列已满，生产者阻塞，进入等待队列
3. 若打印机队列未满，生产者锁住临界区，将打印任务数量增加 1，然后解锁
4. 唤醒消费者

消费者的实现类似于生产者：

1. 消费者询问打印队列是否为空
2. 若打印队列为空，消费者阻塞，进入等待队列
3. 若打印队列不空，消费者锁住临界区，将打印任务减少 1，然后解锁
4. 唤醒生产者

生产者和消费者的模型在操作系统中比较普遍，在 MiscabOS 中，对于各种设备的 I/O 缓冲区都是用生产者消费者模型来实现的。MiscabOS 对于生产者和消费者的模型的实现进行了封装，系统程序和用户程序开发人员可以方便地调用信号量提供的接口方法在程序中引入信号量的任务同步机制。

3.6.3. Mailbox

Mailbox 类似于消息队列，通过内核提供的服务，任务或中断服务子程序可以将一则消息(该消息指针)放入消息队列。同样，一个或多个任务可以通过内核提供的服务从消息队列中获取消息，消息队列的结构图如下所示：

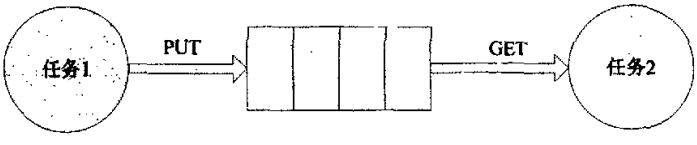

图 3-11 Mailbox 结构图

发送和接收消息的任务约定，传递的消息实际上是传递的指针，指针指向消息真正的内容。通常，先进入消息队列的消息先传给任务，也就是一种 FIFO 的消息队列，而 MiscabOS 也支持 LIFO 的消息队列。

像使用信号量那样，当一个以上的任务要从消息队列接收消息时，每个消息队列有一张等待消息的任务的等待列表。如果消息队列中没有消息，即消息队列为空，则等待消息的任务被阻塞，并放入等待消息的任务列表中，直到有消息到来。一旦一则消息放入消息队列，该则消息将传递给等待消息任务列表中优先级最好的任务，或是最先进入等待消息任务列表的任务。MiscabOS 支持前一种机制，将消息传递给等待消息列表中优先级最好的任务。

3.6.4. 定时器

定时器为系统任务的执行提供了时间上的参照。定时器在初始化后会以给定的时间间隔周期性地中断处理器，使之转入定时器的中断处理程序中执行。在中断处理程序中，可以进行计时、统计以及对延时任务进行处理。MiscabOS 的定时器的结构图如图 3-12 所示：

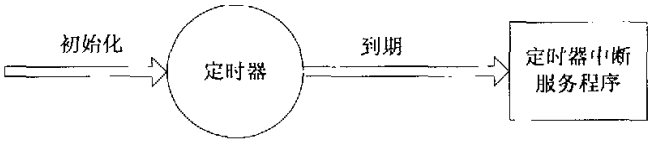

图 3-12 MiscabOS 定时器结构图

以上各小节分别描述了 MiscabOS 实现的各种进程间通信机制，值得一提的是 MiscabOS 在实现这些机制时，充分考虑到了系统和用户程序设计者的需要，把这些机制接口化。当用户在程序设计中需要用到这些机制时，仅仅需要调用系统提供的接口。

3.7. I/O 系统

在操作系统的设计过程中，I/O 系统的设计是最复杂的部分，因为操作系统必须要能够支持尽可能多的外部设备，实现尽可能多的 I/O 机制，另外还必须提供一定程度的抽象机制，使得对于所有 I/O 设备的访问能够有相同的形式。对大多数操作系统来说，I/O 部分的代码占据了整个操作系统的相当部分。在 I/O 系统的设计上，Linux 操作系统十分成功。Linux 把所有的设备分为三类：字符设备、块设备和网络设备[引用]。除了网络设备的处理方法稍显特殊以外，Linux 把字符设备和网络设备都抽象成文件，使得访问这些设备就像访问文件一样容易。MiscabOS 的 I/O 系统实现借鉴了 Linux 的实现方法，对上层的用

户程序提供统一的访问接口，对下层的驱动程序规定标准的接口名。MiscabOS 的 I/O 系统结构图如下所示：

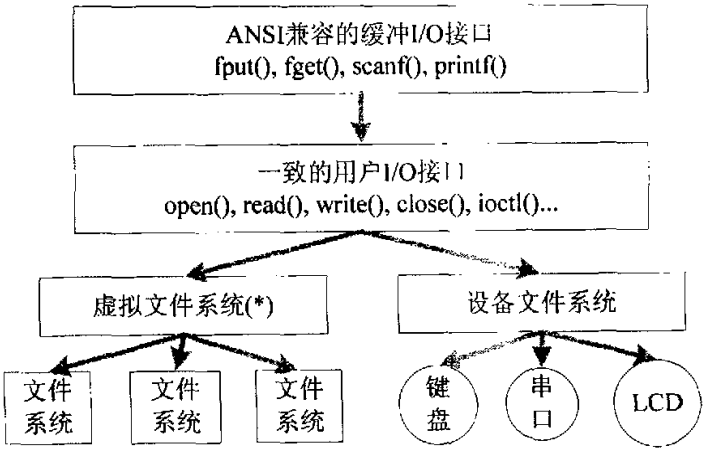

图 3-13 MiscabOS IO 系统结构图

在图 3-13 中，处于最上层的是操作系统提供的缓冲 I/O 接口，这些接口符合 ANSI 标准，为用户提供高性能的输入输出，一般作为操作系统之上的库函数实现；第二层是操作系统的标准用户接口，利用这些标准接口，可以用来操纵普通文件和设备，这些接口一般以系统调用或者内核函数的形式存在；第三层根据用户要操作的文件类型分为两部分：操纵常规文件需要虚拟文件系统，操纵设备文件则需要通过设备文件系统进行。虚拟文件系统是对所有文件系统的一层抽象，通过虚拟文件系统，可以很方便地把各种通用的文件系统或者自己实现的专用文件系统挂载到虚拟文件系统，实现文件系统的扩展。目前 MiscabOS 没有实现虚拟文件系统，在以后的版本中会加入对虚拟文件系统的支持。设备文件系统是 MiscabOS 对 I/O 设备的文件抽象，在 I/O 系统结构中，用户通过缓冲 I/O 接口或者一致的用户 I/O 接口对不同的设备进行输入输出控制。这种分层的 I/O 结构体现了面向对象的思想，使得操作系统的设计变得相对简单。

MiscabOS 的 I/O 机制包括中断驱动的 I/O 和直接内存访问 DMA。中断驱动的 I/O 是在轮询 I/O 的基础上发展起来的，通过在系统中配置中断控制器与外部设备相连，在外部设备发生中断时，处理器就暂停当前的任务，转而执行中断服务例程。中断驱动的 I/O 在性能上要远远优于轮训的 I/O，是目前所有计算机系统的标准 I/O 机制。MiscabOS 的目标平台 Triscend A7S20 的中断控制器提供了多达 16 级的分级中断支持，也就是说可以连接最多 16 个的外部设备，大大提升了系统的可扩展性。直接内存访问是指设备的 I/O 操作可以在没有处理器支持的情况下完成。在没有 DMA 支持时，外部设备的数据传输都要

经由处理器的寄存器，然后被存放到内存中，慢速的 I/O 会耗费大量的 CPU 时间，导致系统的性能下降。而直接内存访问，所有的外部设备与存储器之间的数据传输都由 DMA 控制器负责。CPU 只需将数据传输的源地址和目标地址以及要传输数据的字节数通知 DMA 控制器，所有的数据传输都将在没有 CPU 参与的情况下完成，只是当数据传输完成时，DMA 会中断 CPU，以便 CPU 做进一步的处理。Triscend A7S20 带有四个独立的 DMA 通道，在程序的设制下，每个通道可以进行外设到存储器的数据传输，甚至存储器到存储器的数据传输。MiscabOS 目前仅实现了中断驱动的 I/O 机制，DMA 机制的支持将在 ARM GUI 项目中的 GUI 子系统的开发中加入。

3.8. 设备驱动程序

设备驱动程序和 I/O 系统联系紧密，本节将结合 I/O 系统来说明 MiscabOS 的设备驱动程序结构。从图 3-13 中我们可以看到对设备的访问都要通过设备文件系统来进行。针对标准的 read，open 等用户 I/O 接口，设备文件系统也提供了 device_read，device_open 等设备接口。MiscabOS 将二者进行映射，当用户调用 read 接口时，read 会转而调用 device_read 接口。而对于设备驱动程序的开发者而言，device_read 和 device_open 只是标准的设备文件抽象接口，必须编写操纵具体设备的读、写、打开、关闭等控制例程，然后把具体设备的控制例程和设备文件系统的标准抽象例程映射起来。例如，针对键盘的设备，需要编写 kb_open，kb_read，kb_ioctl 等控制例程，然后映射 device_open 到 kb_open，device_read 到 kb_read，通过这种方式，就可以实现设备的抽象。MiscabOS 为驱动程序的编写提供了极大的便利，驱动程序开发人员只需要定义相应的数据结构，编写具体设备的控制例程，然后调用 MiscabOS 的内核函数 driver_install 就可以完成映射等所有的工作。

3.9. 系统调用模型

MiscabOS 利用 ARM 的异常模式实现了用户空间和系统空间隔离的保护机制，用户空间代码对于系统空间数据结构和变量的访问必须要通过系统调用来实现，因此系统调用模型实质上是保护机制的一个配套实现。在 ARM 平台中，系统调用模型是利用软中断指令 SWI 来实现的，图 3-15 是系统调用模型的框架图：

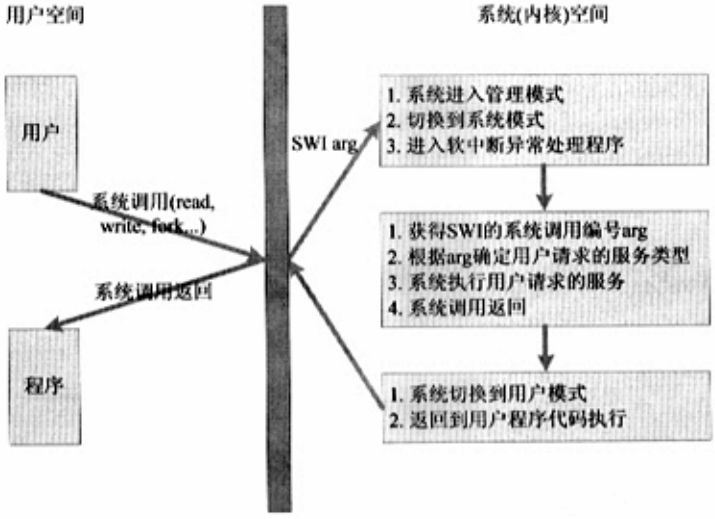

图 3-14 MiscabOS 的系统调用模型图

在图 3-14 中，用户程序在用户空间(用户模式)运行，当需要系统提供服务时，例如 I/O 或定时器服务，执行系统预定义的系统调用函数。系统调用会产生一条 SWI 指令，其中包含系统调用的编号。SWI 指定的执行导致处理器进入管理模式，由第 2 章对 ARM 处理器异常模式的阐述知道，在执行用户请求的服务之前，必须先切换到系统模式。切换到系统模式以后，处理器从异常向量表中选择软中断异常处理程序执行。在软中断异常的处理程序中，首先获得系统调用编号，表示用户请求的服务类型，然后根据调用编号和用户传递的参数执行用户请求的服务。在服务完成后，从系统调用返回。系统调用返回包括两部分：首先将处理器模式置为用户模式，然后返回用户程序代码执行。在第 4 章，将通过代码说明这一机制的实现。

系统调用模型成功地实现了操作系统代码和用户的代码的隔离，极大地提升了系统的安全性和健壮性，是嵌入式操作系统发展的必然趋势。

3.10. 系统配置

相比标准 PC 系统，嵌入式系统最大的特色就是运行环境的不确定性。标准 PC 系统运行在标准的 PC 机上，统一的硬件电气特性，统一的软硬件接口。但嵌入式系统则需要支持多样化的运行环境，必须要能够根据应用的需要对全功能的系统进行相应的裁减。为了应付这种不确定性，在 MiscabOS 的设计中加入了较多的条件包含语句来使得用户可以根据需要对整个系统的关键策略和实现方法进行配置。

MiscabOS 可配置的参数非常多，下面列出最常用，也最重要的一些参数：

1. 系统的最大任务数：MiscabOS 缺省配置成支持十六个任务，用户可以根据应用的需求改变缺省配置。
2. 系统的最大堆栈：每个任务的运行必须要有堆栈，根据任务的性质不同选择不同的堆栈大小。对于科学计算型的任务，需要分配较多的堆栈，而对于一般任务，则采用默认的配置即可满足要求。最大堆栈是和最大任务数相联系的，改变了系统最大任务数后，需要改变系统的最大堆栈。
3. 堆栈增长方向：缺省的堆栈增长方向是向下。这一参数的设置是为了支持各种不同的内存子系统。
4. 任务的最大优先级：缺省为 32。也就是说系统支持从 0 到 31 的任务优先级。
5. 最大的互斥量个数：互斥量属于紧要的系统资源。在用户编制多任务协作同步程序的时候，应相应增加互斥量的个数。
6. 消息队列容量：与最大的互斥量个数类似，消息队列的容量根据任务间通信的消息负载来决定。
7. 时间片大小：在时间片轮转调度算法中，当某个任务执行了一个时间片以后，调度器就会选择下一个任务执行。时间片若设的太短，会导致过多的任务切换，降低了处理器的效率，设得太长又可能对短的交互请求的响应变差。通常没有一个太好的选择，需要根据目标平台处理器的运行速度来决定。一般而言，对于嵌入式处理器的时间片大小设置要远大于 PC 系统时间片的设置。1000ms 是 MiscabOS 缺省的选择。
8. 抗优先级反转的策略：MiscabOS 采用了优先级继承和优先级置顶两种方案来解决可能出现的优先级反转问题。用户可以自行选择不理会优先级反转、采用优先级继承来处理优先级反转或者是采用优先级置顶来处理优先级反转。
9. 最大的设备个数：由于 MiscabOS 没有程序加载器，没有内存分配器，所有的系统资源都必须要预先分配好。所以必须配置最大的设备个数，以便系统分配相应的资源来存储这些设备所对应的设备文件，为构建设备文件系统提供资源基础。
10. 可延时的中断服务程序队列容量：MiscabOS 中把终端服务程序分为 ISR 和 DSR 两部分，所有要执行的 DSR 都被放在 DSR 队列里面，当 ISR 执行完毕后，调度器就会从 DSR 队列中逐个取出 DSR 执行。对

于挂载设备较多的系统，I/O 操作均采用可延时中断服务程序的情况下，需要指定相对大一点的 DSR 队列的容量。

在重新配置了操作系统的参数后，用户必须重新编译整个操作系统。目前，操作系统参数配置只能让用户手动修改文件，在将来的版本中，将会提供交互式的参数配置或者 GUI 的系统配置工具，方便用户定制嵌入式系统的应用。

整个第三章对 MiscabOS 系统框架中的各个组成部分都作了较为详细的介绍，在理解第三章描述的理论和机制的基础上，本文在第四章将结合代码来进一步说明这些理论和机制。

第4章 MiscabOS 的实现

本章将在第 3 章的理论基础上，简要地探讨 MiscabOS 的具体实现，对于前面的内容作出实验上的证明。第 4 章的结构和第 3 章相应，每一个小节说明对应的一个主题。

4.1. MiscabOS 实现综述

MiscabOS 是用 ANSI 标准 C 语言和 ARM 平台汇编语言开发而成，代码总量 93.4KB，文件总数 59 个，其中平台有关的文件 14 个，平台无关的文件 45 个，代码行数 2311 行。以下是 MiscabOS 的目录组织结构：

1. hal：硬件抽象层的实现。包括系统初始化代码、时钟初始化代码、中断控制器初始化代码以及任务切换代码。
2. drivers：设备驱动程序。目前包括串口驱动程序和 24 键键盘驱动程序。
3. fs：文件系统实现，目前仅包含设备文件系统代码和抽象文件系统代码。
4. kernel：内核实现，包含任务控制块、调度器、IPC、等待/就绪队列和 DSR 队列及调度的实现。
5. libc：包含 ANSI 兼容的 C 标准库函数实现。
6. config：包含系统的配置文件。

下面各小节将以 MiscabOS 的系统框架图为线索，介绍每一个功能模块。

4.2. 硬件抽象层

硬件抽象层的实现使得 MiscabOS 能够很容易地移植到其他 ARM 平台上，提高了嵌入式操作系统的通用性。MiscabOS 的硬件抽象层包含系统初始化、时钟初始化、中断控制器初始化以及任务切换。

4.2.1. 系统初始化

系统初始化的代码采用 ARM 汇编语言编写而成，主要工作是初始化异常向量表，注册异常处理程序，建立系统的模式堆栈。在 MiscabOS 的实现中，仅仅处理重启异常、中断异常和软中断异常，因此在异常向量表中，重

启异常、中断异常和软中断异常的入口地址处存放一条跳转指令，其他入口皆置为一条无效指令。代码如下所示：

```
ENTRY
LDR     PC, Reset_Addr       ; 重启异常
LDR     PC, Undefined_Addr   ; 未定义指令异常
LDR     PC, SWI_Addr         ; 软中断异常
LDR     PC, Prefetch_Addr    ; 预取中止异常
LDR     PC, Abort_Addr       ; 数据中止异常
NOP                          ; 保留
LDR     PC, IRQ_Add          ; 中断异常
LDR     PC, FIQ_Addr         ; 快中断异常
```

其中 ENTRY 标志着系统加电后将从其后面的第一条指令开始执行。这样系统加电时，LDR PC, Reset_Addr 指令将被执行，Reset_Addr 正指向重启异常的异常处理程序 Reset_Handler，因此 Reset_Handler 开始执行，所有的初始化代码都放在 Reset_Handler 中，这里是系统运行的起点。

4.2.2. 时钟初始化

在 Reset_Handler 进行了堆栈设置之后，就开始进行硬件抽象层其它部分的初始化，首先初始化时钟。时钟的初始化通过控制 Triscend A7S20 定时器的寄存器 TIMER0_CONTROL_REG 和 TIMER0_LOAD_REG【28】来完成，具体信息请参考 Triscend A7S20 硬件手册。

4.2.3. 中断控制器初始化

中断控制器初始化通过设定 INT_IRQ_ENABLE_CLEAR_REG【28】寄存器的初值来完成。中断发生后的中断号通过读取 INT_IRQ_STATUS_REG【28】寄存器获得。中断控制器初始化后整个系统的硬件初始化告完成，之后的工作是建立系统空闲任务和用户任务，并启用任务调度器。

4.2.4. 任务切换

MiscabOS 的任务切换通过任务的堆栈指针来进行。在 ARM 体系结构中，寄存器 r13 被用作堆栈指针寄存器，因此在任务切换时，需要把换出任

务的现场保存到 r13 所指向的任务堆栈，然后更新 r13 的值，使其指向换入任务的堆栈并恢复该任务的执行现场。实现代码如下所示：

```
switch_context
  stmfd sp!,{lr}
  stmfd sp!,{r0-r12,lr}
  mrs r2,cpsr
  stmfd sp!,{r2}
  str sp,[r1]
load_context
  MSR   CPSR_c, #Mode_SVC:OR:I_Bit:OR:F_Bit
  mov sp,r0
  ldmfd sp!,{r0}
  msr spsr_cxsf,r0
  ldmfd sp!,{r0-r12,lr,pc}^
  END
```

switch_context 用于执行两个任务之间的切换，switch_context 前一部分代码保存换出任务的执行现场，然后调用 load_context 恢复换入任务的执行现场。

4.3. 任务模型的实现

MiscabOS 中的一个任务是用被成为任务控制块(TCB)的数据结构来描述的，在 TCB 中，包含了实现了各种机制和算法的数据元素，MiscabOS 的 TCB 结构如下所示：

```
struct task {
int32 stp;
struct task* next;
struct task* prev;
int32 delay;
void* queue;
enum task_status status;
int8 pri;
int8 pri_old;
int8 slice;
};
```

stp 是用于任务切换的堆栈指针；next 和 prev 用来把任务连接起来，存

放在队列中；delay 是任务的延时定时器，通过 delay 可以实现把任务延迟指定的时间后再执行；queue 指向该任务目前所在的队列；status 是任务的状态；pri 和 pri_old 是用来实现抗优先级反转策略的。在动态修改任务的优先级后，修改前的优先级被保存在 pri_old 中，便于任务对临界区的访问结束后恢复任务的优先级；slice 是时间片，用于时间片轮转调度算法。TCB 的结构十分简单，保证了任务调度和任务切换的高效率。

4.4. 中断机制的实现

在第 3 章中给出了 MiscabOS 中断机制的详细描述，其中分析了可延时的中断服务程序所带来的一些调度机制和中断机制上的变化。在这里，将对这些机制作代码级的分析。当中断发生时，系统进入中断模式，然后处理器转而执行中断异常处理程序，在中断异常处理程序中，保存被中断任务的现场，然后进入中断异常服务程序。由于中断的分级是由中断控制器实现的，所以处理器执行的中断异常服务程序实际上只能被称为中断转发服务程序。在中断转发服务程序中，读取中断控制器的 INT_IRQ_STATUS_REG 寄存器，获知发生的中断编号，然后根据中断编号，再调用具体的中断服务程序。中断转发服务程序如下：

```
void ISR_Dispatcher(int32 stp) {
 int32 num=(*(volatile unsigned *)INT_IRQ_STATUS_REG);
 void* result;
 int8 i=0;
 while (num!=0x1) {
     i++;
     num >>=1;
 }
 result=isr_handlers[i]();
 reschedule_from_isr(stp,dsr_handlers[i],result);
}
```

在中断转发程序中，首先读取中断状态寄存器的值，然后根据这个值来判断哪个设备发生了中断，调用相应的中断服务程序，之后在函数 reschedule_from_isr 中把 DSR 部分的处理例程添加到 DSR 队列中。值得一提的是 ISR_Dispatcher 函数的参数 stp。这个 stp 是被中断任务的堆栈指针，在中断执行完毕之后，调度器应该根据 stp 来恢复这个任务的执行现场。

4.5. 任务调度模型

参考图 3-7，MiscabOS 的任务调度模型是基于动态优先级的任务调度策略。这个调度策略是在就绪队列的操作函数 ready_queue_get 中实现的，其代码如下：

```
struct task* ready_queue_get(struct ready_queue* rq) {
  int32 i;
  for (i=MAX_PRIORITY-1;i>=0;i--) {
      if (rq->head[i]!=0) break;
  }
  if (i<0) return 0;
  return rq->head[i];
}
```

这一串代码将从最高优先级开始扫描，直到找到一个合适的任务或者到达队列尾部为止。MiscabOS 把任务调度的模型实现放在 ready_queue_get 函数中，这个函数将从就绪队列中选出需要执行的任务。如果嵌入式系统应用需要实现更为复杂的调度机制时，只需要适当修改这个函数的代码，便可以实现新的调度策略。这样的设计使得任务调度机制的部署变得十分简单。

4.6. 任务间通信机制

MiscabOS 支持四种任务间通信机制，包括互斥量、信号量、消息队列和定时器。互斥量需要实现抗优先级反转的进程调度策略，在互斥量的实现代码中加入了大量的条件编译语句，因此其实现最为复杂，也难以理解，本节将通过具体描述信号的实现机制来说明 MiscabOS 的任务通信机制。

信号量由一个整形计数器和一个等待队列构成，其代码如下所示：

```
struct semaphore {
    int32 count;
    struct wqueue wait_queue;
    };
```

信号量支持两种操作：wait 和 signal，signal 操作的代码如下所示，本节将会以打印机队列的生产者和消费者模型来说明 wait 和 signal 代码：

```
void semaphore_signal(struct semaphore* ps) {
struct task* pt;
dsr_lock();
```

```
ps->count++;
if(ps->count==1) {
    if ((pt=wqueue_get(&ps->wait_queue))!=0) {
        wqueue_remove(&ps->wait_queue,pt);
        pt->status &=~(BLOCK|DELAY);
        if (pt->status==READY) {
            scheduler_add(pt);
            rescheduler();
        }
    }
}
dsr_unlock();
}
```

以打印机队列中的生产者为例，首先锁住临界区，然后把队列中的元素个数累加，如果元素个数为 1，说明消费者处于睡眠，这时需要把消费者唤醒，并置入就绪队列。在这里，signal 操作首先锁住对 DSR 队列的调度，然后对信号量的整形计数器进行累加，若整形计数器的值为 1，则说明有进程在该信号量上睡眠，通过调用 wqueue_get 函数得到需要唤醒的任务，修改状态，置入就绪队列。

wait 操作的代码如下所示：

```
void semaphore_wait(struct semaphore* ps) {
dsr_lock();
while (ps->count==0) {
    scheduler_remove(current);
    wqueue_add(&ps->wait_queue,current);
    current->status=BLOCK;
    rescheduler();
}
ps->count--;
dsr_unlock();
}
```

以打印机队列中的消费者为例，首先锁住临界区，判断队列中的元素个数是否为 0，如果为 0，则说明队列中没有可供消费的打印任务，需要阻塞自身并等待生产者的唤醒。如果队列中的元素不为 0，则将其递减，

释放临界区并返回。在这里，wait 操作首先锁住对 DSR 队列的调度，然后判断信号两的整形变量是否为 0，如果是则说明需要阻塞当前任务，于是修改状态为阻塞态，然后从就绪队列中删除当前任务，并置入信号量的等待队列。如果整形变量不为 0，那么将其递减，释放临界区之后返回。

4.7. I/O 和文件系统

如图 3-13，MiscabOS 的 MiscabOS 的 I/O 系统实现了两层的抽象机制，第一层是设备到设备文件系统的抽象，把所有的设备注册到设备文件系统统一的接口上，那么就能够调用一致的设备文件系统例程来访问具体的硬件设备；第二层是设备文件系统到抽象文件系统的抽象，把所有实现的文件系统都挂载(Mount)到抽象文件系统的统一接口上，那么通过调用抽象文件系统的统一接口，我们就能访问不同的具体的文件系统。包括操作系统提供的统一的抽象文件系统接口，其结构定义如下：

```
struct abstract_fs {
 char name[10];
 int (*creat_entry)(const char*,int);
 int (*open_entry)(const char*,int,int);
 int (*read_entry)(int,char*,int);
 int (*write_entry)(int,char*,int);
 int (*ioctl_entry)(int,int,int);
 int (*close_entry)(int);
 int (*lseek_entry)(int,int,int);
 int (*delete_entry)(const char*);
};
```

具体文件系统通过 mount_fs 函数调用挂载到抽象文件系统。针对设备文件系统，也定义了设备信息的数据结构和接口函数，如下所示：

```
struct device_info {
char name[10];
void (*init_entry)();
int (*open_entry)();
int (*read_entry)(char*,int);
int (*write_entry)(char*,int);
int (*ioctl_entry)(int,int);
int (*close_entry)();
};
```

通过两个数据结构的 name 成员来映射抽象的文件系统接口和设备文件系统接口。MiscabOS 提供了 device_driver_register 函数来建立这种映射关系。

4.8. 设备驱动程序

设备驱动程序需要编写具体的设备控制例程，然后调用 device_driver_register 来注册设备驱动程序，建立起设备文件系统的标准接口和具体设备控制例程之间的映射。以串口驱动为例，在驱动程序中，提供了 uart_read，uart_write，uart_ioctl 等串口控制例程，其设备驱动程序安装的代码如下：

```
void uart_install() {
struct device_info di;
k_strcpy(di.name,"uart");
di.init_entry=uart0_init;
di.open_entry=0;
di.read_entry=uart0_read;
di.write_entry=uart0_write;
di.ioctl_entry=0;
di.close_entry=0;
device_driver_register(&di);
}
```

在代码中，首先建立一个 device_info 的设备文件系统结构，然后用具体设备的控制例程初始化设备文件系统的接口，建立起设备文件系统接口和控制例程之间的映射，再调用 device_driver_register 把驱动程序注册到设备文件系统。

4.9. 系统调用模型的实现

系统调用通过软中断指令 SWI 来实现。SWI 指令具有如下的格式：

```
SWI 0x00002100
```

其中 0x00002100 是软中断的编号，我们利用这个编号来确定用户需要请求哪一类服务。在调用 SWI 指令后，系统进入管理模式，然后执行软中断异常处理程序 SWI_Handler，其代码如下：

```
SWI_Handler
        STR LR, TMP1
        MRS LR, SPSR
        STR LR, TMP2
        MSR CPSR_c, #Mode_SYS:OR:I_Bit:OR:F_Bit
        STMFD SP!,{R0-R12}
        LDR R0, TMP1
```

```
        STMFD SP!, {R0}
        LDR R0, TMP2
        STMFD SP!, {R0}
        LDR r0, [lr, #-4]
        BIC r0, r0, #0xFF000000
        LDR sp,=SWI_Stack
        BL SWI_Dispatcher
        LDMFD SP!, {R1}
        MSR SPSR, R1
        LDMFD SP!, {R0-R12, PC}^
TMP1    DCD     0
TMP2    DCD     0
```

在软中断异常处理程序中，首先保存当前状态和返回地址，之后确定软中断的编号，调用 SWI_Dispatcher 函数，并把软中断编号作为参数传递给 SWI_Dispatcher 函数。SWI_Dispatcher 是用 C 语言编写的软中断转发程序，接受两个参数，第一个参数是系统调用编号，另外一个是系统调用的参数。在 SWI_Dispatcher 将会根据软中断的编号，调用相应的服务程序，其代码是一个 switch-case 的结构。如下所示：

```
void SWI_Dispatcher( int32 num, void *parg)
{
    switch(num)
    {
        case SYSCALL_READ:
                do_read(parg);
                break;
        case SYSCALL_WRITE:
                do_write(parg);
                break;
        case SYSCALL_OPEN:
                do_open(parg);
                    break;
        case SYSCALL_CLOSE:
                do_close(parg);
                break;
```

```
                ……
            default:
                        do_error(parg);
        }
    }
```

SWI_Dispatcher 函数是系统调用转发程序，根据用户的请求编号，来执行相应的服务程序。SYSCALL_OPEN，SYSCALL_CLOSE，SYSCALL_READ，SYSCALL_WRITE 是系统预定义的宏，用来定义与 POSIX 兼容的系统调用。MiscabOS 的系统调用模型的设计实现了极大的可扩展性，用户可以通过定义新的软中断编号，映射用户自定义的函数来扩展系统调用。

在第 4 章，我们从代码的角度进一步说明了 MiscabOS 各个功能模块的实现机制，在第 5 章，将介绍 MiscabOS 的开发环境和实验环境。

第5章 实验环境和系统测试

5.1. MiscabOS 开发的软件配置

MiscabOS 的开发工具是 ARM 公司的 ADS 1.2【17】集成开发环境，该集成开发环境包括交叉编译器、连接工具、汇编工具、调试器等，用户界面如图 5-1 所示：

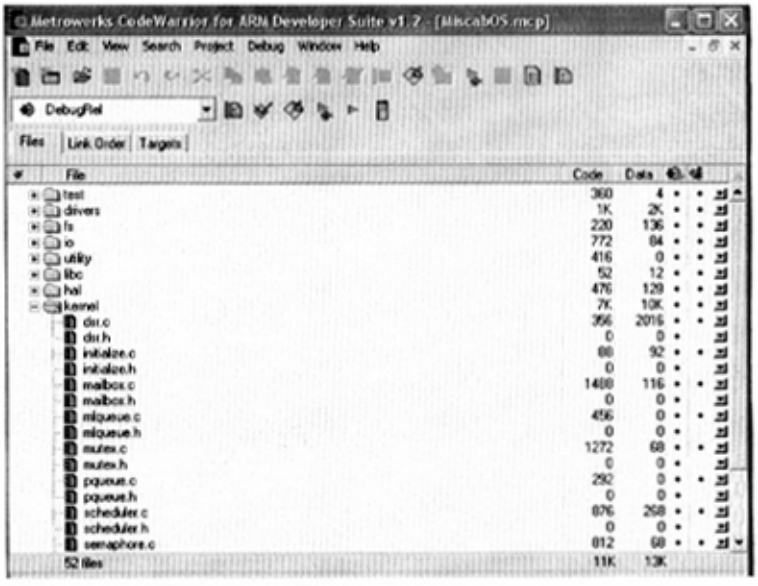

图 5-1 ADS 集成开发环境

此外 MiscabOS 在烧录到目标平台时还采用了可重构设计的开发工具套件和调试工具，其中包括 Cygwin、可重构设计开发套件(FastChip3.0 和 Triscend_SDK3.0)以及 VHDL 综合工具(Synplify)。

5.2. MiscabOS 开发的硬件配置

MiscabOS 的开发采用了 ARM7 实验箱，实验箱集成 Triscend A7S20 应用处理器，内含 ARM7TDMI 处理器内核，包含两万多个 FPGA 逻辑门，结合 ARM&TDMI 和 FPGA 可以进行完备的可重构开发实验。其硬件配置表如表 5-1 所示：

表 5-1 实验箱配置表

编号	内容	说明

1	CPU	ARM7TDMI 内核，60MHz 主频。含有可重构内部总线，多达 25,600 个逻辑门的内部可配置 FPGA 逻辑阵列
2	Flash	可选的 8/16M，可扩展至 256M。该容量满足固化包含 GUI 系统的 uClinux 操作系统
3	SDRAM	32M，可扩展至 256M，可运行 uC/OS II 和 uClinux 操作系统
4	RS232 串口	2 个 RS232 串口，最高波特率 115200bps
5	JTAG	20 针标准 JTAG 调试接口，支持 Wiggler、Raven、VisionPROBEII 接口协议
6	LED	可编程的 LED 指示灯组，3 个 LED，可用 PWM 驱动
7	7-SEG-LED	8 个七段数码显示管，74LS164 锁存驱动
8	BUTTON	2 个外部中断按钮，可设为 FIQ 和 IRQ。一个系统复位按钮，产生复位信号。
9	BUZZER	蜂鸣器可用 PWM 驱动，也可以 I/O 电平驱动，可模拟 D/A 转换输出 Audio
10	Ethernet	RJ-45 10Base-T 以太网接口，支持 ARM 平台的网路应用，包括 TCP/IP、telnet 和 ftp 等
11	USB	1 个 USB1.1 协议的接口，可重构为 Host 或 Client
12	SPI	扩展 2 个 8×8 数码点阵显示管，可显示 8×16 的点阵图形，也可显示广告牌效果的汉字
13	LCD	Samsung 产 65536 真彩色 320×240TFT－LCD，带触摸屏，白色背光，具有 PDA 效果
15	IIC	1 个 IIC 接口的 EEPROM，可进行扩展存储试验
16	Keyboard	24 键键盘，4×6，可进行键盘驱动和中断捕获以及扫描实验
17	PWM	可驱动两路立体声声道，逼真播放 wav 数

		据流，附立体声耳机接口
18	AC'97	AC'97 音频接口，可录、放音，可进行 Audio 开发实验，mp3 解码，MPEG 压缩等实验
19	VGA	标准 VGA 接口，最大刷新频率 50MHz，可驱动 VGA 显示器
20	PS/2	标准 PS/2 接口，可外接 PS/2 键盘鼠标

除表中列举的配置之外，系统还提供时钟源、中断控制器和两个定时器，以及一个看门狗定时器。其外观如图 5-1 所示：

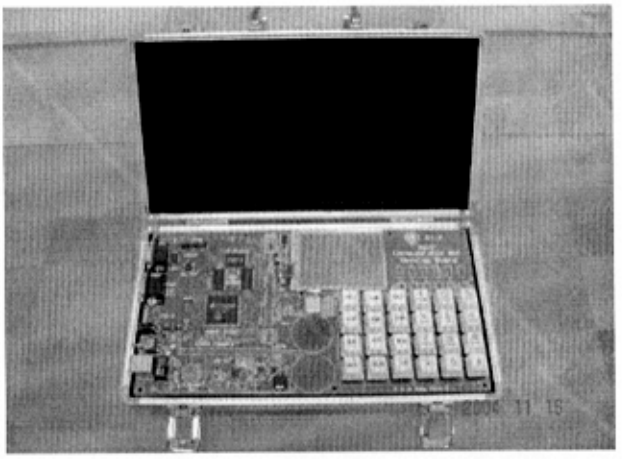

图 5-1 MiscabOS 的硬件环境

5.3. MiscabOS 的系统测试

MiscabOS 的系统测试主要包括操作系统功能测试、互斥信号量等 IPC 机制的测试、优先级反转测试和串口驱动测试，其中串口驱动测试包含在操作系统功能测试中。测试方法是编写测试程序，与 MiscabOS 编译链接，将生成的可执行文件烧录到目标版，通过串口观察输出，并和 MiscabOS 系统进行交互。

5.3.1. 测试 1：系统功能测试

MiscabOS 的系统功能测试以调度器为主要目标，附带测试驱动程序，设备文件系统，主要目的在于验证系统功能的正确性和可靠性。系统测试包含了任务创建、挂起、延迟、恢复、阻塞、消息队列、接口 API 测试和串口驱动测试等等各项测试，其测试的代码如下所示：

```
char echo_buffer[200];
```

```
char greeting[] =
   "Welcome to MiscabOS Ver1.0\n\r"
   "    MiscabOS is a real time operating system \r\n"
   "    Maintained by wlm@fudan.edu.cn\r\n"
   ;
char menu[]="1)Test the task creation.\n\r"
            "2)Test for task suspend and resume.\n\r"
            "3)Test for mutex.\n\r"
            "4)Test for semaphore.\n\r"
            "5)Test for mailbox.\n\r"
            "6)Test for delay (10 seconds).\n\r"
            "7)Test for echo.\n\r"
            "Please choose one(1-7):";
void demo(void* p) {
   int fd;
   int i;
   char buf='c';
   fd=open("/dev/uart0",0,0);
   redirect_stdout(fd);

   while(1) {
        printf(greeting);
        printf(menu);
        read(fd,&buf,1);
        write(fd,&buf,1);
        printf("\n\r");
        switch (buf) {
        case '1':
             task_delay(5, 1);
             printf("Create task1 .......OK\n\r");
             task_delay(5, 1);
             printf("Create task2 .......OK\n\r");
             task_delay(5, 1);
             printf("Create task3 .......OK\n\r");
```

```
            printf("The test of task creation has completed.\n\rPress
any key return to menu...\n\r");
            read(fd,&buf,1);
            break;
        case '2':
            task_delay(5, 2);
            printf("Suspend task1 ..... OK\n\r");
            task_delay(5, 2);
            printf("Suspend task2 ..... OK\n\r");
            task_delay(5, 2);
            printf("Suspend task3 ..... OK\n\r");
            task_delay(5, 2);
            printf("Resume task1 ..... OK\n\r");
            task_delay(5, 2);
            printf("Resume task2 ..... OK\n\r");
            task_delay(5, 2);
            printf("Resume task3 ..... OK\n\r");
            printf("The test of task suspend and resume has
completed\n\rPress any key return to menu...\n\r");
            read(fd,&buf,1);
            break;
        case '3':
            task_delay(5, 3);
            printf("Create mutex ..... OK\n\r");
            task_delay(5, 3);
            printf("Lock mutex ..... OK\n\r");
            task_delay(5, 3);
            printf("Unlock mutex ..... OK\n\r");
            task_delay(5, 3);
            printf("Try lock mutex ..... OK\n\r");
            printf("The test of mutex has completed\n\rPress any key
return to menu...\n\r");
            read(fd,&buf,1);
            break;
```

```
        case '4':
            task_delay(5, 4);
            printf("Create semaphore ..... OK\n\r");
            task_delay(5, 4);
            printf("Wait semaphore ..... OK\n\r");
            task_delay(5, 4);
            printf("Signal mutex ..... OK\n\r");
            task_delay(5, 4);
            printf("Try wait mutex ..... OK\n\r");
            printf("The test of semaphore has completed\n\rPress any
key return to menu...\n\r");
            read(fd,&buf,1);
            break;
        case '5':
            task_delay(5, 5);
            printf("Create mailbox ..... OK\n\r");
            task_delay(5, 5);
            printf("put mailbox ..... OK\n\r");
            task_delay(5, 5);
            printf("get mutex ..... OK\n\r");
            task_delay(5, 5);
            printf("Try put mailbox ..... OK\n\r");
            task_delay(5, 5);
            printf("Try get mailbox ..... OK\n\r");
            printf("The test of mailbox has completed\n\rPress any
key return to menu...\n\r");
            read(fd,&buf,1);
            break;
        case '6':
            printf("Delay begin:\n\r");
            for (i=48;i<48+10;i++) {
                write(fd,(char*)&i,1);
                printf("\n\r");
                task_delay(5, 6);
```

```
            }
            printf("Press any key return to menu...\n\r");
            read(fd,&buf,1);
            break;
        case '7':
            i=0;
            read(fd,&buf,1);
            while (buf!='\r') {
                echo_buffer[i]=buf;
                write(fd,&buf,1);
                read(fd,&buf,1);
                i++;
            }
            buf='\n';
            write(fd,&buf,1);
            buf='\r';
            write(fd,&buf,1);
            echo_buffer[i++]='\n';
            echo_buffer[i++]='\r';
            echo_buffer[i++]='\0';

            printf(echo_buffer);
            printf("Press any key return to menu...\n\r");
            read(fd,&buf,1);
            break;
        default:
            printf("Error input!!!\n\r");
            printf("Press any key return to menu...\n\r");
            read(fd,&buf,1);
        }
    }
}

int main() {
```

```
        task_create(demo,0,3,200);
        scheduler_start();
    }
```

与操作系统编译链接之后，将程序烧录到目标平台，运行程序后用户可以通过串口和系统进行交互。图 5-1 是程序输出菜单后，用户先后选择菜单 1 和 2 的输出：

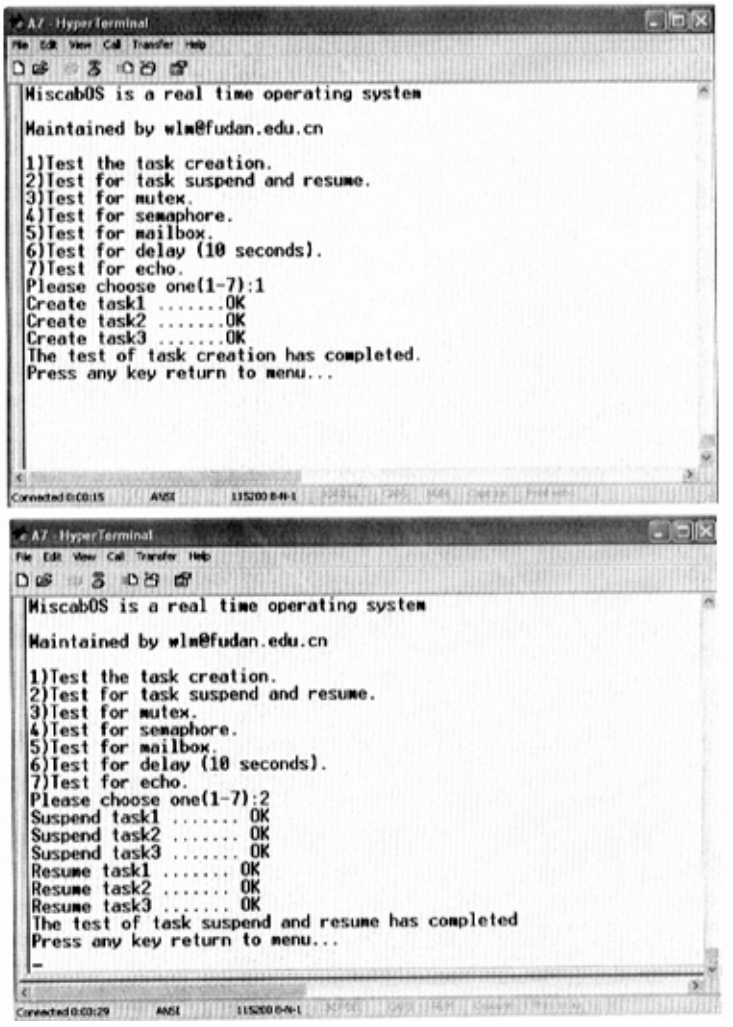

图 5-1 MiscabOS 系统测试输出示意图

系统测试比较全面地测试了作为一个操作系统所应该具有的功能，测试的结果显示 MiscabOS 嵌入式操作系统的实现达到了预期的目标。

5.3.2. 测试 2：IPC 机制测试

IPC 机制主要包括互斥和信号量，我们采用生产者和消费者模型来对 MiscabOS 进行测试。在测试模型中，包含一个缓冲区和描述可用缓冲区空间容量的变量，生产者将持续写数据到缓冲区，减小缓冲区的容量，消费者将从缓冲区读取数据，增大缓冲区的容量。对缓冲区容量的操作通过互斥来控制。测试模型将会创建生产者和消费者两个任务，然后开启调度器。具体的测试代码如下所示：

```
#define MAX_BUFFER 10
struct mutex* pm;
struct semaphore* pse, *psf;
struct buffer {
    char data[MAX_BUFFER];
    int pos;
};
struct buffer buf;
void producer(void* p) {
    while(1) {
        semaphore_wait(pse);
        mutex_lock(pm);
        printf("producer %c,%d\n",'A'+buf.pos,buf.pos);
        buf.data[buf.pos]='A'+buf.pos;
        buf.pos++;
        semaphore_signal(psf);
        mutex_unlock(pm);
    }
}
void consumer(void* p) {
    while(1) {
        semaphore_wait(psf);
        mutex_lock(pm);
        printf("consumer
%c,%d\n",buf.data[buf.pos-1],buf.pos-1);
        buf.data[buf.pos-1]='0';
```

```
            buf.pos--;
            semaphore_signal(pse);
            mutex_unlock(pm);
        }
    }
    int main() {
        task_create(&producer,0,2,100);
        task_create(&consumer,0,2,100);
        pse=semaphore_create(MAX_BUFFER);
        psf=semaphore_create(0);
        pm=mutex_create();
        scheduler_start();
    }
```

可以看到在 main 函数中，首先创建两个任务 producer 和 consumer，然后创建两个信号量和一个互斥量用于任务之间的同步，最后开启调度器。在测试程序成功开始运行后，程序的输出如图 5-2 所示：

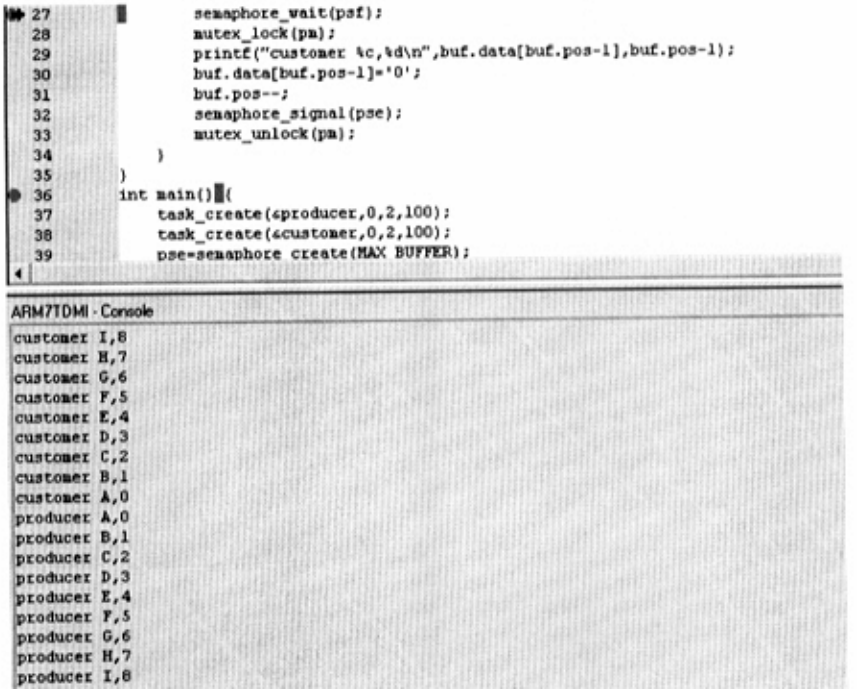

图 5-2 信号量的输出示意图

5.3.3. 测试 3：优先级反转测试

优先级反转测试制造了一个优先级反转问题，然后观察调度器利用优先级继承策略对优先级反转处理的结果，通过观察调度器的输出来验证策略的正确性，测试代码如下所示：

```
extern struct task* volatile current;
struct mutex* pm;
void f(void* p) {
   task_delay(5);
   printf("task3 try to get mutex!\n");
   mutex_lock(pm);
   printf("task3 got the mutex!\n");
   task_suspend(current);
}
void g(void* p) {
   int i;
   mutex_lock(pm);
   printf("Task1 : %d\n",current->pri);
   task_set_pri(current,10);
   printf("Task1 : %d\n",current->pri);
   mutex_unlock(pm);
   printf("task2 unlock the mutex");
   printf("Task1 : %d\n",current->pri);
}

int main() {
   pm=mutex_create();
   task_create(&f,0,16,200);
   task_create(&g,0,4,200);
   scheduler_start();
}
```

程序的输出如图 5-3 所示：

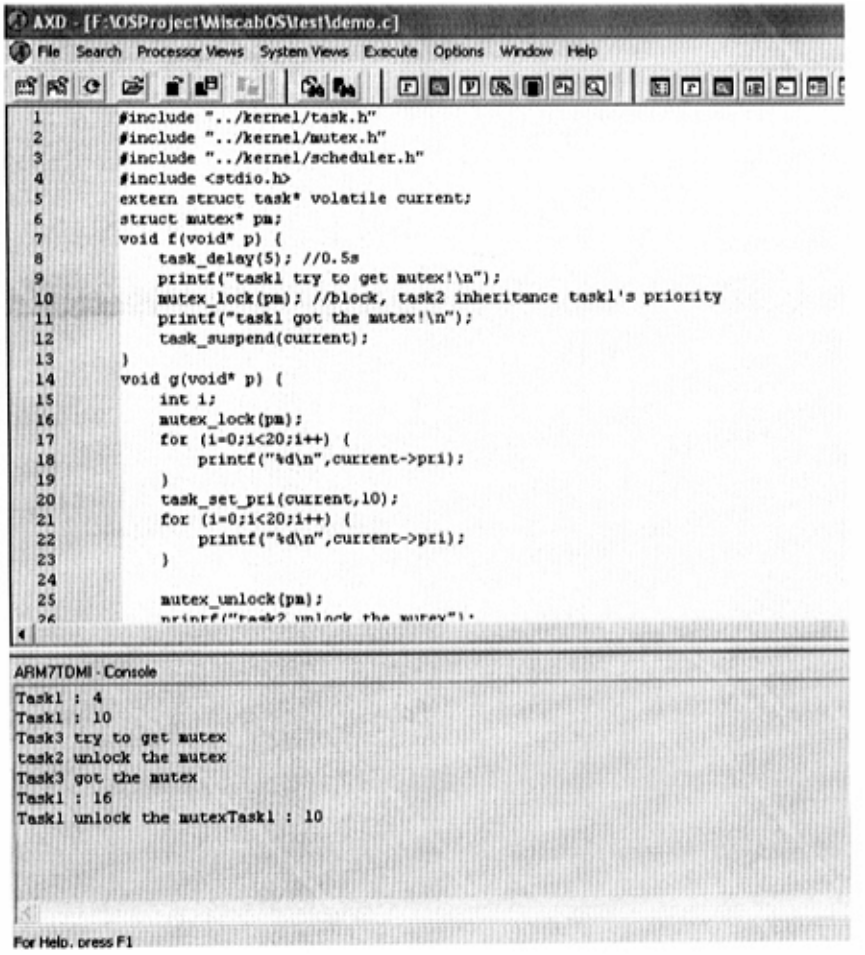

图 5-3 优先级继承策略测试程序输出示意图

从程序的输出我们可以看到，优先级为 4 的任务 1 首先运行，然后自行修改自己的优先级为 10，之后锁住互斥量 pm。这个时候优先级为 16 的任务 3 被调度执行，然后尝试上锁互斥量 pm。由于 pm 已经被任务 1 锁住，所以任务 3 阻塞，并修改任务 1 的优先级为 16，任务 1 随即被调度运行，释放互斥量，任务 1 结束后调度任务 3 运行，任务 3 成功地上锁互斥量 pm，得以继续运行。该例子很好地说明了采用优先级继承的机制来抗优先级反转的策略。

5.4. 与其他相关系统的主要对比分析

嵌入式操作系统的四个重要的衡量标准是实时性、通用性、安全性和扩展性，本节将从操作系统的实现角度分别对比 MiscabOS 和其他嵌入式操作系统，包括内核调度机制、任务间的同步和通信机制、实时性、保护机制以及操作系

统所支持的硬件。给出了 MiscabOS 在这些具体的方面和其他操作系统的对比之后，我们根据四个衡量标准对分析的结果进行了总结。典型地，与 MiscabOS 对比的嵌入式操作系统选择了在嵌入式系统中应用最广泛的 uC/OS-II 2.52 和 eCos。

5.4.1. 内核调度机制的比较

嵌入式操作系统的核心是调度器。当系统包含多个任务时，必须使用调度器来决定执行哪一个任务或线程。调度器对线程的运行进行控制，并为线程提供一种同步机制。表 5-1 列出了 MiscabOS 和 uC/OS-II 以及 eCos 调度器的比较。

表 5-1 调度器比较

操作系统	调度方法	同优先级调度	优先级数/个	任务数量/个
uC/OS-II	固定优先级抢占式调度	无	64(8 个保留)	56
eCos:位图	动态优先级抢占式调度	无	32	32
eCos: 多级队列	动态优先级抢占式调度	Round-Robin	32	依赖内存容量
MiscabOS	动态优先级抢占式调度	Round-Robin	用户定义 缺省 32	依赖内存容量 缺省 32

MiscabOS 提供了动态优先级和抢占式调度的策略，并可以让用户实现自定义的调度策略，提供了极大的通用性。此外对于同优先级的调度实现了 Round-Robin 调度方法。优先级的个数通过 MAX_PRIORITY 宏来指定，用户可以修改最大的优先级，系统缺省为 32。系统支持最大的任务数量缺省为 32，用户可以根据内存容量进行修改。从表 5-1 可以看出 MiscabOS 在调度器的设计上明显优于 uC/OS-II，此外，由于 MiscabOS 提供了可配置的内核调度器，因此比 uC/OS-II 和 eCos 拥有更好的通用性。

5.4.2. 任务间同步、通信机制

嵌入式操作系统的功能一般需要通过若干任务和中断服务程序共同完成，

任务与任务之间、任务与中断服务程序之间必须协调动作，互相配合，这就牵涉到任务间的同步与通信问题。表 5-2 为 MiscabOS 和另外两种操作系统同步与通信机制的比较。

表 5-2 同步与通信机制的比较

	uC/OS-II	eCos	MiscabOS
同步与通信机制	邮箱 消息队列	互斥 邮箱 条件变量 计数型信号量 事件标志	邮箱 消息队列(Message Queue) 互斥 计数型信号量 消息(Signal)

从表 5-2 可以看出，MiscabOS 比 uC/OS-II 提供了更丰富的进程间通信机制，方便了应用程序的开发。另外 MiscabOS 借鉴了 Linux 系统的 Signal 任务通信机制，在实现任务间同步时可以有灵活的选择。

5.4.3. 实时性

任务切换时间和中断延迟时间是评估嵌入式操作系统性能的两个重要指标。任务切换时间可以反映出嵌入式操作系统执行任务的速度，而中断延迟时间可以反映出嵌入式操作系统对外界变化的反应速度。但直接测量系统的中断延迟和任务切换十分苦难，需要特殊的设备和软件支持，因此我们考虑通过计算斐波那契数列【8】的值来测试系统的整体性能。

测试原理是同时运行三个计算分别计算某个数的斐波那契数列值的程序，然后开启时钟中断，在时钟中断服务程序里面保存当前计算程序的现场，进行计时统计，并调度下一个计算程序执行。这样当三个程序执行完毕之后，比较所有的执行时间，就可以得出整体性能上的一个比较结果。

考虑到数值 50 的斐波那契数列值的计算在配备了奔腾 4 的标准 PC 上的执行时间都需要 5 分钟以上，所以我们的测试套仅仅选用了数值 10，20，30，40 的四组计算。测试的方法是分别将三种嵌入式操作系统和统一的计算程序烧录到 ARM 开发板，开机后运行计算程序，程序运行完毕后从串口输出统计时间。

斐波那契数列值的计算程序如下：

```
int fibonacci(int n)
{
    if(n==0 || n == 1)
```

```
        return 1;
    return fibonacci(n-1) + fibonacci(n-2);
}
```

测试主程序的代码如下：

```
int main()
{
    int i ;
    for( i = 1; i <= 4; i++)
    {
     output_time();
     fib(i *10);
     output_time();
    }
}
```

其中 output_time 是把系统当前时间从串口输出。实际上系统运行后，会维持一个 CPU 时钟滴答的计数器，时钟中断之后在时钟中断服务程序里面会把计数加 1，这样通过记录运行程序前的时钟滴答和运行程序后的时钟滴答，就可以确定程序的运行时间，测试环境选择了 60MHz 的 ARM 处理器，每个时钟滴答为毫秒，也就说精确度是 0.1 秒。

实验的结果如表 5-3 所示：

表 5-3 斐波那契数列数值计算程序时间表

	数值	uC/OS-II(秒)	eCos(秒)	MiscabOS(秒)
测试套 1	10	2.3	2.0	1.9
测试套 2	20	3.9	3.6	3.1
测试套 3	30	15.2	13.4	12.2
测试套 4	40	329	316	282

由于数值计算的值比较大，因此用在任务切换和中断延迟上的时间几乎可以忽略，该测试只能表达总体上的性能优势，如果要更精确地测试三个操作系统在任务切换上的时间和中断延迟时间，必须采用更高精度的时钟源。下一步的工作将对继续这一实验。

5.4.4. 保护机制

uC/OS-II 和 eCos 没有实现保护机制，在 ARM 平台上，所有的代码都运

行在特权模式，而 MiscabOS 利用 ARM 平台提供的异常模式实现了用户模式和特权模式的隔离，在安全性方面胜过 uC/OS-II 和 eCos。目前，嵌入式系统应用对于安全性的要求越来越高，对于安全性的研究也是当前的热点之一，MiscabOS 从操作系统角度探讨了安全机制的实现，无疑有着极大的现实意义。

5.4.5. 对硬件的支持

uC/OS-II 和 eCos 支持当前流行的大部分嵌入式 CPU，都具有很好的移植性。uC/OS-II 支持从 8 位到 32 位的 CPU；而 eCos 可以在 16 位、32 位和 64 位等不同体系结构之间移植。MiscabOS 目前只支持 16 位和 32 位的 ARM 处理器，但是由于优化设计的硬件抽象层，移植到其他平台时只需要重写硬件抽象层。在 MiscabOS 中，硬件相关的代码不到代码总量的三分之一。uC/OS-II 和 eCos 内核较小，经过裁减后代码最小分别为大约 10KB 和 40KB，MiscabOS 编译链接后生成的可执行映像大小为 22KB(含调试信息)，除去调试信息仅有 9 KB，需要的最小内存仅有 14KB。因此 MiscabOS 对硬件的要求很低，具有很高的经济性。通过比较，MiscabOS 在硬件支持方面逊于 uC/OS-II 和 eCos，但 MiscabOS 最为新兴开发的嵌入式操作系统，将会支持越来越多的硬件平台。

5.4.6. 比较总结

第 5 章描述了 MiscabOS 的开发环境和实现环境，并对 MiscabOS 和 uC/OS-II 以及 eCos 做了各方面的比较，以下根据嵌入式操作系统的四个标准：实时性、通用性、安全性和可扩展性做出一个总结。MiscabOS 和其他嵌入式 OS 的比较如图 5-1 所示：

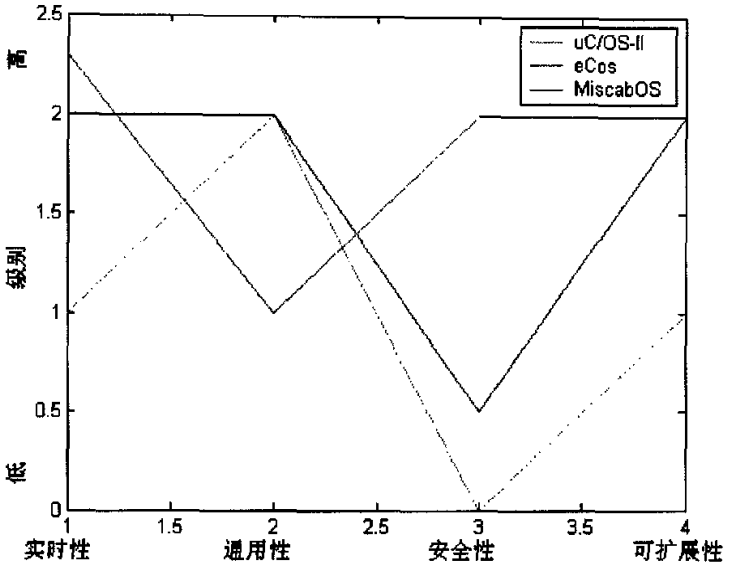

图 5-1 MiscabOS、uC/OS-II 和 eCos 比较

图中横坐标 1、2、3 和 4 分别表示四个比较标准：实时性、通用性、安全性和可扩展性，纵坐标表示各个比较标准的级别。由图 5-1，MiscabOS 除了在通用性方面逊于 uC/OS-II 和 eCos 之外，其他方面都要优于 uC/OS-II，在可扩展性和通用性方面和 eCos 齐平。此外，eCos 虽然是开放源代码，但由于收到 GPL 协议的限制，对 eCos 的使用并不是完全免费的。

第6章 结论与展望

6.1. 结论

近年来，嵌入式系统市场日益升温，对于嵌入式领域的研究也越来越深入，尤其是对于嵌入式系统的核心嵌入式操作系统，更是有了各种各样的解决方案。本文首先分析现有嵌入式操作系统领域的现状，然后描述了各类操作系统的优势和不足，之后以自主开发的 MiscabOS 嵌入式操作系统为基础，全面地介绍了嵌入式操作系统的理论机制和具体实现。本文首先给出了 MiscabOS 的框架图，然后分析了框架图中各个功能模块的设计思想，尤其是具有创新意义的 ARM 平台嵌入式操作系统保护机制的设计、可延时中断服务程序的设计以及硬件抽象层的设计。

在文中的第 5 章，进行了一系列的实验验证了 MiscabOS 各种机制的正确性，并对 MiscabOS 和其他操作系统做了各方面的比较和测试，测试结果显示 MiscabOS 在整体性能、安全性、可扩展性方面都要优于 uC/OS-II 和 eCos。对于测试系统的实时性，本文采用斐波那契模型对系统进行了测试，显示除了 MiscabOS 的优良性能，但对在高精度时钟源上的任务切换时间和中断响应时间的测试，有待进一步的研究。

6.2. 展望

本文在嵌入式操作系统 MiscabOS 设计和开发的基础上对现有嵌入式操作系统的一些主要功能实现作出了创新、优化和改进，这些成果可用作为进一步研究的参考。

同时，MiscabOS 的实现还不完整，不能支持文件系统和内存管理，这些问题将在下一步的计划中完成。具体地，下一步的工作包括开发遵循 POSIX 标准的用户接口 API，并把 ROMFS 移植到 MiscabOS 中来，在 MiscabOS 中加入文件系统的支持。此外，GUI 子系统的开发也将是下一步工作的重点。

今后需要继续密切跟踪嵌入式领域各项的最新发展，并将各项关键技术引入到 MiscabOS 设计开发和构建中，使得 MiscabOS 嵌入式操作系统能随着技术的发展而不断发展完善。

参考文献

[1] H. Huang, P. Pillai, and K. G. Shin, "Improving Synchronization-Free Algorithms for Interprocess Communication in Embedded Real-Time Systems," in Proceedings of USENIX Technical Conference, June 2002.

[2] Y. Hodique and I. Simplot-Ryl， Secure Extensible Type System for Efficient Embedded Operating System by Using Metatypes. G. Grimaud, Jul. 2005

[3] Friedrich Schön, Wolfgang Schröder-Preikschat, Olaf Spinczyk, and Ute Spinczyk: "On Interrupt-Transparent Synchronization in an Embedded Object-Oriented Operating System", Proceedings of the 3rd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2000), Newport Beach, CA, USA, March 15-17, 2000.

[4] Gary Nutt. Operating Systems: A Modern Perspective. ISBN: 7-111-13530 北京 机械工业出版社 2002

[5] Gary Nutt. Kernel Projects For Linux. ISBN: 7-111-10099-9 北京 机械工业出版社 2004

[6] Abraham Silberschatz etc. Applied Operating System Concepts. ISBN: 0471263141 北京 高等教育出版社 2003

[7] Karim Yaghmour. Building Embedded Linux Systems. ISBN: 0-596-00222-X. 美国 O'Reilly 2003

[8] Brian W. Kernighan, Dennis M. Ritchie. The C Programming Language. ISBN: 0131101633 美国 O'Reilly

[9] Jean J. Labrosse. MicroC OS II: The Real Time Kernel. ISBN: 1578201039. 北京 北京航空航天大学出版社 2003

[10]陈莉君 深入分析 Linux 内核源代码 ISBN: 7115105251. 北京 人民邮电出版社 2002

[11]赵炯 Linux 内核完全注释 ISBN: 7111149688 北京 机械工业出版社 2004

[12]陈章龙,涂时亮 嵌入式系统 ISBN: 7810772406 北京 北京航空航天大学出版社 2003

[13]周立功 ARM 微控制器基础与实战 ISBN: 7810773836 北京 北京航空航天大学出版社 2003

[14]Alessandro Rubini, Jonathan Corbet 等 Linux 设备驱动程序 北京 中国电力出版社 2002

[15]ARM Ltd. ARM Architecture Reference Manual

[16]ARM Ltd. ARM7TDMI(Rev 4) Technical Reference Manual

[17]ARM Ltd. ARM® Developer Suite (Version 1.2) CodeWarrior® IDE Guide

[18]ARM Ltd. ARM® Developer Suite (Version 1.2) Assembler Guide

[19]ARM Ltd. ARM® Developer Suite (Version 1.2) Linker and Utilities Guide

[20]ARM Ltd. ARM® Developer Suite (Version 1.2) Developer Guide

[21]ARM Ltd. ARM® Developer Suite (Version 1.2) Debug Target Guide

[22]ARM Ltd. http://www.arm.com

[23]Triscend®. A7 Configuration Update

[24]Triscend®. Triscend SDK Install Guide

[25]Triscend®. A7 SDK Users Guide

[26]Triscend®. Using GDB

[27]Triscend®. A7HAL API Guide

[28]Triscend®. Triscend A7S Data Sheet

[29]Triscend Corporation. http://www.triscend.com

[30]Linux Kernel. http://www.kernel.org

[31]ARM Linux Project. http://www.arm.linux.org.uk/

[32]Embedded Linux/Microcontroller Project. http://www.uclinux.org

[33]μC/OS RTOS. http://www.ucos-ii.com

[34]RedHat eCos. http://sources.redhat.com/ecos/

[35]VxWorks. http://www.windriver.com

[36]ITRON. http://www.ertl.jp/ITRON/home-e.html

[37]ITRON Committee, TRON Association. μITRON4.0 Specification

[38]RISC vs. CISC.
http://cse.stanford.edu/class/sophomore-college/projects-00/risc/risccisc/

致谢

首先我要感谢我的导师钟亦平教授和张世永教授。在这三年的研究生学习生活中，钟老师和张老师尽可能地为我们提供良好的学习和实验环境，而且在各方面都给我们提供了很大的帮助。尤其是在撰写论文的过程中，两位老师一直给予我鼓励和支持，并多次从百忙之中抽出宝贵的时间给予认真而细致的指导，提出了很多建设性的意见，为我的研究指点迷津。

其次我要感谢所有帮助过我的老师和同学。在我学习和撰写论文的过程中，很多老师和同学都给了我很大的启发，多次给予热心的指导和帮助。尤其要感谢傅维明老师、吕智慧博士、顾华平、周爱东以及刘晓涛同学，傅老师为我们的论文撰写提供了方方面面的意见和资源，使得我的论文能够在技术上与最新研究同步，在规范能够符合格式上的各种标准。

吕博士作为实验室的课题负责人，在项目研究和论文撰写上给予了许多有帮助的指点；而顾华平、周爱东和刘晓涛同学在信息传达和相互的沟通上给予了许多帮助。十分感谢他们的热心指点和帮助。

最后我要感谢我的父母和女朋友。在这漫长的三年学习生涯中，是他们的鼓励和支持为我解除后顾之忧，使我能考入复旦继续深造，并能安心地完成我的学业和毕业论文。在此我向他们的关爱表示深深的感谢。

谨以此文献给所有关心我的家人和朋友。

论文独创性声明

本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。论文中除了特别加以标注和致谢的地方外，不包含其他人或其它机构已经发表或撰写过的研究成果。其他同志对本研究的启发和所做的贡献均已在论文中作了明确的声明并表示了谢意。

作者签名：吴立明　　日期：2005.6.4

论文使用授权声明

本人完全了解复旦大学有关保留、使用学位论文的规定，即：学校有权保留送交论文的复印件，允许论文被查阅和借阅；学校可以公布论文的全部或部分内容，可以采用影印、缩印或其它复制手段保存论文。保密的论文在解密后遵守此规定。

作者签名：吴立明　　导师签名：钟仲华　　日期：2005-6.4