

JIYU VXWORKS DE QIANRUSHI XITONG JI SHIYAN

基于 **VxWorks** 的
嵌入式系统及实验

徐惠民 主编



北京邮电大学出版社
www.buptpress.com

前 言

进入 21 世纪后,计算机技术发展的一个重要方面就是嵌入式计算机系统的迅速发展。嵌入式计算机的应用领域已经从传统的控制功能发展到通信、网络等各个方面,嵌入式计算机的核心也从简单的单片机发展到专用的嵌入式计算机芯片,系统软件也从简单的监控系统发展到实时多任务操作系统。

在这样的形势下,各个高等院校纷纷开出了关于嵌入式系统的课程。关于嵌入式系统设计的实验竞赛也举行了几次。

本书是我们从 2001 年开始开设的一门课程“嵌入式系统和实验”的总结。

这门课程的技术基础是 ARM 嵌入式芯片和 VxWorks 操作系统。鉴于 ARM 芯片在嵌入式领域中的地位,选择 ARM 芯片作为硬件基础应该不需要做什么说明;而嵌入式课程选择 VxWorks 操作系统作为软件基础的还不是很多。但是在通信和信息技术的一些高级应用中,出于对系统可靠性等性能的考虑,使用 VxWorks 系统还是很多的。正是考虑到学校和学生的这种信息技术背景,本书选择了 VxWorks 操作系统。

本书从内容上讲可以分为两大部分:第 1 章到第 5 章是关于嵌入式系统的基本知识和基本理论;第 6 章到第 9 章是关于嵌入式系统的实验;还有一个附录,包含了一些相关的补充知识和补充材料。

理论基础部分包含了“嵌入式系统概述”、“实时系统的基本理论”、“嵌入式系统硬件基础”、“VxWorks 操作系统”和“嵌入式系统开发环境”。其中“实时系统的基本理论”介绍了关于实时系统的基本概念,特别是关于任务调度方面的一些算法。这些理论在 VxWorks 等操作系统中实际上已经在应用,所以,我们觉得做一些介绍还是很有必要的。

实验部分的目标板是我们自行开发的基于 S3C4510B 芯片的嵌入式实验

开发板 IRDCv3。实验分为内核基本功能实验、基础应用性实验和高级扩展性实验 3 部分。每个实验都有详细说明,也给出了详细的源代码框架。只要在指定部分加入相应的语句,就可以构成完整的源程序。

本书的基本理论部分是实验部分的基础,而实验部分又是对理论部分的实践和应用。通过理论和实践的结合以及两者的互相贯通,读者能比较迅速地掌握嵌入式开发的基本理论和方法。

本书由徐惠民主编,参加教材编写的还有王立磊、赵红强、许馨元。参加相关工作的还有郭洪文、陈萍、周虚、徐雅静等。特别感谢郭洪文在开发实验板过程中所做的贡献,没有他的努力,实验板的开发不可能如期完成。

本书在内容上和叙述上一定还有许多不足之处,希望读者不吝指正。

徐惠民

目 录

第 1 章 嵌入式系统概述

1.1 嵌入式系统的定义	1
1.2 嵌入式系统的特点	1
1.3 嵌入式系统的应用领域	3
1.4 嵌入式系统的发展趋势	5
1.5 嵌入式系统设计面临的挑战——优化设计度量	5
1.6 3 种关键的嵌入式系统技术	7
1.6.1 处理器技术	7
1.6.2 IC 技术	8
1.6.3 设计技术	8

第 2 章 实时系统的基本理论

2.1 实时系统的历史	11
2.2 实时系统的基本概念及其性能衡量指标	12
2.2.1 实时系统的基本概念	12
2.2.2 实时系统的性能衡量指标	14
2.3 实时系统的任务调度	14
2.3.1 任务调度的功能	14
2.3.2 任务调度的性能准则	14
2.3.3 任务调度方式	15
2.3.4 任务调度算法	15
2.4 实时系统任务的互斥与同步	23
2.4.1 任务的互斥	23

2.4.2	任务的同步	24
2.4.3	优先级倒置现象	25
2.4.4	优先级继承算法	27
2.4.5	优先级上限算法	28
2.5	死锁现象	30
2.5.1	死锁的概念	30
2.5.2	死锁的起因	31
2.5.3	解决死锁的策略	31
2.6	实时系统任务的通信机制	32

第 3 章 嵌入式系统硬件基础

3.1	ARM 体系结构	34
3.1.1	ARM 的概念	34
3.1.2	ARM 处理器的架构特点	34
3.1.3	主流 ARM 处理器及使用场合	35
3.1.4	ARM 微处理器的寄存器结构	38
3.1.5	ARM 微处理器的两种指令集比较	39
3.2	S3C4510B 芯片	40
3.2.1	工作特性	40
3.2.2	芯片引脚介绍	44
3.3	ARM/THUMB 指令集及汇编	50
3.3.1	总体介绍	50
3.3.2	ARM 的寻址方式	51
3.3.3	ARM 指令集	54
3.3.4	THUMB 指令及应用	70
3.3.5	ARM 汇编程序设计	70

第 4 章 VxWorks 操作系统

4.1	系统概述	88
4.2	多任务介绍	89
4.2.1	任务概述	89

4.2.2 任务的创建	91
4.2.3 任务名和任务标识	94
4.2.4 任务的删除和保护	95
4.2.5 任务的信息	96
4.2.6 任务的控制	98
4.2.7 任务的错误状态	99
4.3 任务间的通信	100
4.3.1 概述	100
4.3.2 信号量	100
4.3.3 消息队列	109
4.3.4 管道	112
4.3.5 信号	115
4.3.6 事件	117
4.3.7 共享内存	119
4.4 内存管理	120
4.4.1 概述	120
4.4.2 VxWorks 中的内存布局	122
4.4.3 系统函数介绍	123
4.4.4 虚拟内存	124
4.4.5 高速缓存 Cache	125
4.4.6 常用的内存分析工具	125
4.5 中断和异常	126
4.5.1 VxWorks 的中断机制	126
4.5.2 VxWorks 的异常机制	128
4.6 时钟和定时机制	129
4.6.1 VxWorks 操作系统的时钟	129
4.6.2 VxWorks 操作系统的定时机制	131
4.7 I/O 系统	133
4.7.1 I/O 系统概述	133
4.7.2 3 种 I/O 接口介绍	134
4.7.3 I/O 系统的工作机制	139

4.8 文件系统	141
4.8.1 文件系统概述	141
4.8.2 VxWorks 文件系统介绍	141

第 5 章 嵌入式系统开发环境

5.1 Tornado 开发环境介绍	143
5.1.1 Tornado 核心工具	144
5.1.2 Tornado 的工程类型	156
5.1.3 开发环境的使用	156
5.2 IRDC05v3 嵌入式实验箱介绍	162
5.2.1 硬件方案说明	162
5.2.2 系统软件方案说明	164

第 6 章 内核基本功能实验

6.1 实验 1 Hello world	167
6.2 实验 2 信号量	169
6.3 实验 3 消息队列	174
6.4 实验 4 看门狗	178
6.5 实验 5 wind 内核功能	181
6.6 实验 6 中断处理	185
6.7 实验 7 Socket 通信	188
6.8 实验 8 虚拟内存设备驱动	198
6.9 实验 9 任务软调度	208
6.10 实验 10 任务软调度的改进	214

第 7 章 基础应用性实验

7.1 实验 1 LED 灯实验	229
7.2 实验 2 ARM 的串行口实验	238
7.3 实验 3 ARM 的交通灯实验	251
7.4 实验 4 ARM 的键盘接口实验原理实验	257
7.5 实验 5 ARM 的液晶接口实验原理实验	264

7.6 实验 6 液晶键盘实验	275

第 8 章 高级扩展性实验

8.1 实验 1 VxWorks BSP 移植	279
8.2 实验 2 理发师问题实验	286
8.3 实验 3 哲学家吃饭问题实验	296
8.4 实验 4 路由器问题实验	308
8.5 实验 5 构架 WebServer 服务器	321

附 录

附录 A 参考文献	325
附录 B 参考电路图	326
附录 C 标准 C 库函数	335
附录 D WindShell 常用命令	339
附录 E 串口调试助手使用说明	341

第1章 嵌入式系统概述

1.1 嵌入式系统的定义

根据 IEEE(国际电气和电子工程师协会)的定义,嵌入式系统是控制、监视或者辅助设备机器和车间运行的装置(原文为 devices used to control, monitor, or assist the operation of equipment machinery or plants),这主要是从应用上加以定义的,从中可以看出嵌入式系统是软件和硬件的结合体,是具有一定应用功能的系统。

目前国内一个普遍被认同的定义是:以应用为中心,以计算机技术为基础,软件硬件可裁减,适用于应用系统对功能、可靠性、成本、体积、功耗有严格要求的专用计算机系统。

1.2 嵌入式系统的特点

1. 针对性强

嵌入式系统是面向用户、面向产品、面向应用的,不可能独立发展。嵌入式处理器的功耗、体积、成本、可靠性、速度、处理能力、电磁兼容性等方面均受到应用要求的制约。针对用户的具体需求,嵌入式系统的软硬件均可以裁减和添加,面向的用户是不一样的,处处体现了其针对性强的特点。

2. 实时性强

实时性是嵌入式系统的一个基本要求,很难想象嵌入式系统没有实时性会是什么样的情况。当然,单纯靠提高嵌入式处理器的计算性能和工作频率来提高嵌入式系统的实

时性是不科学的,而应该主要靠优化嵌入式软件,保证合理的任务调度等手段来实现。嵌入式系统的实时性分为硬实时和软实时,硬实时规定了任务完成的截止时间,不能有丝毫的延时,软实时则可以将条件放宽一些。

3. 高可靠性

如果嵌入式系统经常出现故障,其众多应用场所将会发生许多灾难性的事故,也会带来非常多的不便。“勇气号”等嵌入式机器人之所以取得很大的成功,一个很重要的原因就在于它们具有克服了各种复杂恶劣环境的高可靠性。嵌入式的软件要求固化存储,这样可以提高软件执行的可靠性和速度。

4. 系统精简

嵌入式系统的操作系统内核容量非常的精简,目的是更可靠、更快速地执行任务程序,并且节省存储空间,使系统的复杂程度降低。外围器件也要求精简可靠,以最大性价比的代价满足客户的需要。

5. 嵌入式系统需要开发工具和环境

由于嵌入式系统本身不具备自主开发能力,即使设计完成以后,用户通常也是不能对其中的程序功能进行修改,因此必须有一套开发工具和环境才能进行开发。这些工具和环境一般是基于通用计算机上的软硬件设备以及各种逻辑分析仪、混合信号示波器等。开发时往往有主机和目标机的概念,主机用于程序的开发、调试,目标机作为最后的执行机构。开发时需要主机和目标机不断交互,所以还要有相应的通信机制。

一个典型的开发模型如图 1-1 所示。

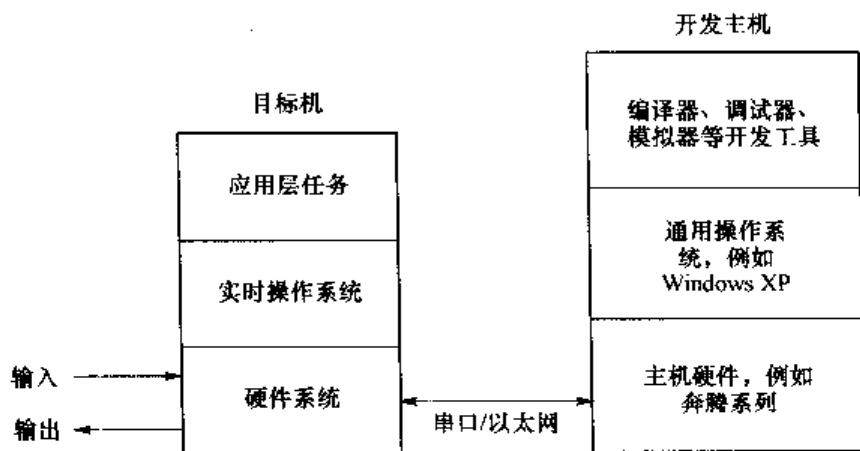


图 1-1 嵌入式开发模型

其中目标机由 3 个层次构成:最底层是硬件系统,中间层是实时操作系统,最高层是应用层任务。实时操作系统起承上启下的作用,调度多任务良好地工作,并且管理系统的所有资源。

开发主机也由3部分构成:最底层是硬件系统,如我们常用的PC;中间层为桌面操作系统,常见的有Windows系列、Linux等;最高层则是集成开发环境,如本书所介绍的针对VxWorks操作系统应用开发的Tornado集成软件,包括编辑器、编译器、调试工具、模拟工具等。

开发主机和目标机通过串口或者网口等方式进行通信,并且具有一定的协议。

1.3 嵌入式系统的应用领域

如今,嵌入式系统的应用非常广泛,如制造工业、过程控制、通信、仪器、仪表、汽车、船舶、航空、航天、军事装备、消费类产品等等,可以说已经融入到了人类生活的方方面面。下面结合图表具体举一些例子,可以使读者更方便地了解。

1. 电子通信方面

嵌入式在电子通信方面应用的一个很典型的例子就是如图1-2所示的手机,手机可以说已经在我们国家非常普及,关于手机的嵌入式软、硬件技术也非常的多。



图 1-2 手机

2. 信息家电方面

嵌入式系统在信息家电方面应用的种类也是非常多,如表1-1所示,但这也只是其中很少的一部分。

表 1-1 信息家电举例

信息家电	代表性产品
网络电视	Microsoft WebTV
网络可视电话	Info Gear IPhone
网络游戏机	Sega Dreamcast
网络智能手持器件	3Com Palm AT&T PocketNet Phone
	Nokia 9000, 9000I, 9110, 7110
NC 终端(clients)	IBM NetStation
PC-中间器件	DIAMON Multimedia Rio Player
	Nuvomedia Rocket e-Book
	Audible Mobile Player

3. 航天航空领域

在航天航空领域,嵌入式系统的应用也是如火如荼,如大家熟悉的“勇气号”(如图1-3所示)就是一个典型的嵌入式应用,其可靠稳定的性能使嵌入式的特点得到了淋漓尽致的

体现。我们国家发射的卫星,包括各种飞机上,均运用了大量的嵌入式技术,如电子雷达、电子通信、各种信号的传感探测等。



图 1-3 “勇气号”火星探索机器人

4. 军事装备

嵌入式在军事方面(如导弹系统、雷达系统、卫星通信系统、红外线探测等方面)得到了很广泛的应用,现代化的战争如果没有嵌入式的应用,很难想象会变成什么样子。各个国家均大力加强嵌入式技术的发展,争取在国家军事装备和国防技术上有重大突破。如图 1-4 所示的“赫耳墨斯”,参加了阿富汗反恐作战,价值 4 万美元,可携带 2 架摄像机,它克服了许多恶劣的环境将前方的实时信息传送回来,在战争中发挥了很好的监视、探测作用。

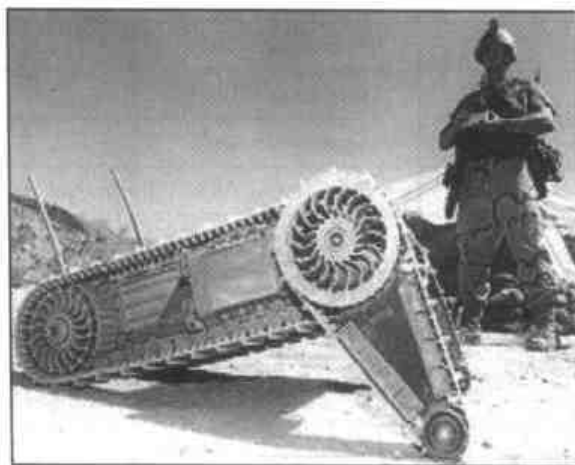


图 1-4 赫耳墨斯

1.4 嵌入式系统的发展趋势

1. 计算能力不断提高

随着半导体工业的大力发展,摩尔定律仍然发挥着自己的效用。嵌入式系统的计算能力在不断发展提高,而市场的需求也要求必须这么做。多媒体等技术的发展对嵌入式系统的计算性能要求越来越高,有更多的复杂算法需要嵌入式系统去计算和处理。

2. 网络功能不断发展

由于网络的普及,越来越多的场合实现了网络化和远程访问,越来越多的客户需要远程监控、远程访问,这就需要嵌入式系统支持网络功能。

3. 软件模块化技术得到广泛发展

基于模块化的嵌入式系统软件开发的优点有很多:可以选用商业成熟的软件组件;可以重用以往经过测试的高可靠性软件模块;能够提高软件质量、缩短产品面市时间;能够降低成本、减少费用等。

4. 集成度越来越高

由于稳定性、可靠性等许多指标的要求,片上 SOC 技术也得到了极大的需求和发展,越来越多的系统集成商和嵌入式产品开发商也选择了 SOC 片上系统作为自己产品的核心模块,由选择单一的器件变为选择更集成的系统模块。

5. 软件方法设计硬件

可编程芯片在嵌入式系统设计中的应用促进了用软件方法设计硬件的发展。硬件设计人员通过 Verilog 语言或者 VHDL 语言可以实现复杂的硬件功能,并且通过软件模拟来确定功能实现是否正确,最后才用硬件来实现。

6. 适应能力增强

切入到市场的时机关系到投资的效益问题,所以要求开发周期尽可能地缩短,这就需要嵌入式系统灵活多变,有很强的适应性,一旦出现新的功能需求,可以在很短的时间内完成系统的设计,开发出产品,投入到市场中。

1.5 嵌入式系统设计面临的挑战——优化设计度量

嵌入式系统设计面临的挑战用一句话来概括就是:必须优化设计度量(Design Metrics)。优化设计度量的目标是构建一个最理想的系统。那么,哪些因素可以称之为设计度量呢?下面我们来讨论这个问题。

设计度量有:

单元成本——生产每个嵌入式系统复制品的费用;

NRE 成本(非重复性工程成本)——设计系统的一次费用;

容量大小——系统所需的物理容量；

性能——系统的执行时间；

功耗——系统消耗的总功率；

可扩展性——系统不增加投资而改变功能的能力；

设计出样品的工期——设计出一个样品所需花费的时间；

投入到市场的工期——开发出产品，并且可以投入到市场的时间；

可维护性——发行第一版产品后，维护产品的能力；

正确性、安全性等等。

但是需要注意的是，优化设计度量中的某一项可能会引起其他项变差。例如开发某个产品，要降低功耗，则需要投入更多的费用，可能系统的性能也会下降。优化度量参数不能仅仅由某一方面的专家进行，而应该综合考虑，从整体性能考虑，选出最优化的方案进行设计。下面以投入市场的开发工期为例说明优化设计度量的重要性。

图 1-5 就是苛刻的投入市场的开发周期因子与收入的分析比较图。可以看出，收入值在某一时间点之前慢慢上升，过了这个时间点后就开始下降。这也就说明，投入市场的开发周期是具有严格的时间限制的，不能太早，也不能太晚，太早则时机还不成熟，太晚则过了收入的高峰值，并且延迟时间越长，收入下降得越厉害。一般来说，投入市场的开发周期大约为 8 个月。

在图 1-6 中，坐标原点表示及时切入市场、投放产品的时间， D 表示延期将产品投入市场的时间， W 表示出现最大收益值的时间， $2W$ 为产品的市场生命周期结束的时间。如果用曲线面积表示整个收益的话，则及时切入市场的收益为 $1/2 \times 2W \times W$ ，延期切入市场的收益为 $1/2 \times (W - D + W) \times (W - D)$ ，损失率为 $[D(3W - D)/2W^2] \times 100\%$ 。例如：

市场生命周期 $2W = 52$ 周，延迟 $D = 4$ 周，则

$$\text{损失} = [4 \times (3 \times 26 - 4) / 2 \times 26^2] = 22\%$$

市场生命周期 $2W = 52$ 周，延迟 $D = 10$ 周，则

$$\text{损失} = [10 \times (3 \times 26 - 10) / 2 \times 26^2] = 50\%$$

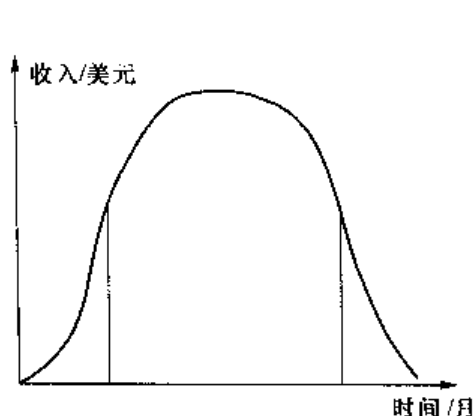


图 1-5 开发周期和收入的关系分析曲线

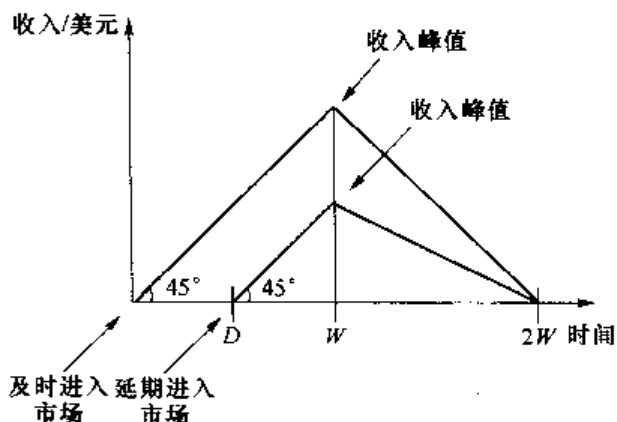


图 1-6 两种不同的切入市场时间引起的效益分析图

由以上例子可以看出延迟的时间越长,损失越严重,延迟10周将产品投入市场的损失甚至达到了50%。

单元设备成本(Cost1)和开发样品成本(Cost2)构成了开发产品的总成本(Cost)。正如前面提到的,单元设备成本为复制样品的成本,相对比较便宜,而且数量(n)越多成本越低,而开发样品的成本一般是比较昂贵的。例如,开发嵌入式产品的总成本

$$\text{Cost} = \text{Cost1} \times n + \text{Cost2} \quad (1.1)$$

平均每个产品的投资(Cost3)

$$\text{Cost3} = \text{Cost} / n \quad (1.2)$$

从式(1.1)可以看出来:假设 n 比较大,总投资成本Cost主要和Cost1、 n 有关系(大到一定程度Cost2可以忽略不计),Cost1和 n 越大,则总投资Cost越多。从式(1.2)可以看出,在总投资Cost不变的情况下,产品的平均投资Cost3也主要由数量 n 决定, n 越大,则平均投资成本越低。

1.6 3种关键的嵌入式系统技术

1.6.1 处理器技术

作为嵌入式系统中最关键的技术之一,处理器技术的发展对于整个嵌入式系统的发展起着至关重要的作用,对于提高整个嵌入式系统的性能也起着非常重要的作用。

处理器从结构和功能上一般分为3种:通用处理器(General Processor)、特定应用场合的处理器(Application-specific Processor)和单一功能的处理器(Single-purpose Processor)。图1-7可以帮助读者更好地理解3种处理器的区别。

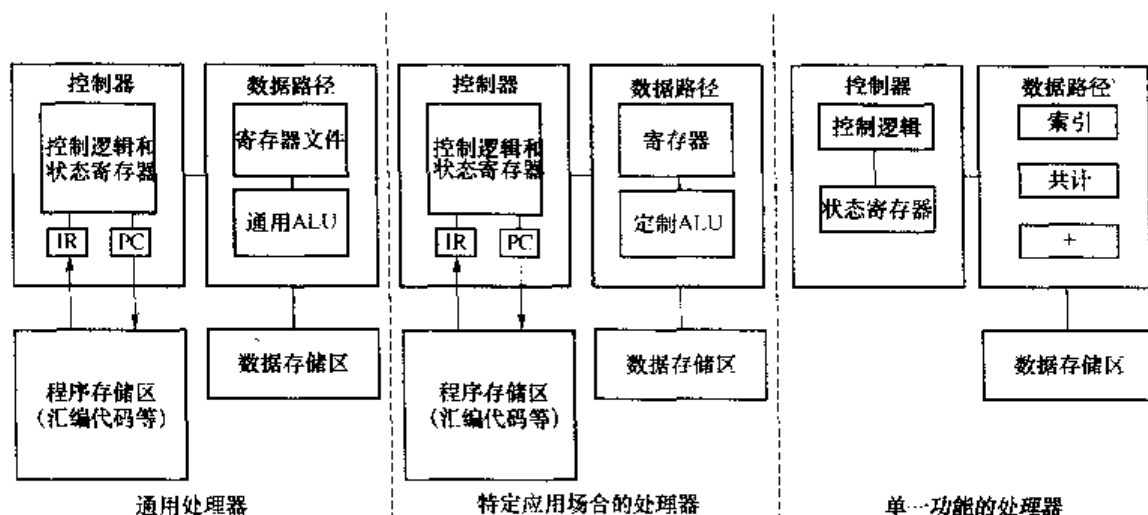


图 1-7 3种不同的处理器结构

由图 1-7 也可以看出,通用处理器结构比较复杂,集中的元件比较多,具有程序存储区、较大的寄存器文件和通用的算术逻辑单元(ALU)。通用处理器很常见,比如我们经常提到的奔腾系列 CPU 就是典型的通用处理器。通用处理器具有较短的开发产品投入市场时间(time-to-market)和较强的扩展性。

单一功能的处理器一般没有程序存储区,仅仅是为了执行某个特定的程序和具备特定的功能,比如我们常见的减速器、协处理器、其他外围器件等。它具有快速、低功耗、小尺寸等特点。

特定应用场合的处理器是介于通用处理器和单一功能的处理器之间的一种折衷处理器,适用于某些特定场合,一般也具有程序存储区,具有一定的扩展能力和良好的性能。

1.6.2 IC 技术

IC 技术就是集成电路(Integrate Circuit)技术,或者集成芯片(Integrate Chip)技术。IC 技术在设计上针对各种客户的需求进行定制,一般采用多层电路板(多者可以达到 10 层或者 10 层以上)。IC 技术可以在结构或者功能上分为 3 种类型:超大规模集成电路(VLSI)、专用集成电路(ASIC)、可编程逻辑设备(PLD)。

VLSI 各层均进行了优化,无论是尺寸、布线等都进行了专门的考虑,特点是性能好、尺寸小、功耗低。缺点是研发费用高,产品投入到市场的周期比较长。

ASIC 不用设计者像设计 VLSI 那样讲究电路各种铜线的路由布线或者布局,也具有性能好、尺寸小的特点,另外它跟 VLSI 相比研发费用较低,一般只有 VLSI 的几分之一。但是 ASIC 也具有研发时间比较长的缺点。

PLD 各层电路已经存在,设计者不用考虑布线,只需要将其中的硬件连接进行重新组织、编程,就可以方便地设计出各种功能的产品。它的优点是灵活、开发周期短,但是 PLD 的执行速度比较慢,尺寸比较大,价钱也比较昂贵。

1.6.3 设计技术

在讲设计技术之前,首先要介绍一下著名的摩尔定律(Moore's Law):集成电路的晶体管容量大概每 18 个月就会翻一番(IC transistor capacity has doubled roughly every 18 months for the past several decades)。这个定律是由大名鼎鼎的 Intel 创始人 Gordon Moore 于 1965 年提出的,如图 1-8 所示。

在图 1-8 中可以看到,集成电路的晶体管容量在 1981 年到 2009 年期间按照稳定的斜率增加,完全符合摩尔定律的预测。当然,摩尔定律也不一定非常准确,但它确实反映了 IC 集成电路的快速发展。

单位芯片中晶体管的数量成指数增加,主要跟技术和需求这两个因素有关系,两者缺一不可。设计技术的进步也是日新月异,由最初的 4004 到后来的 Pentium 4 处理器。技

术的复杂程度在不断增加。很多读者很难想象 IC 技术的发展是如此得快速。

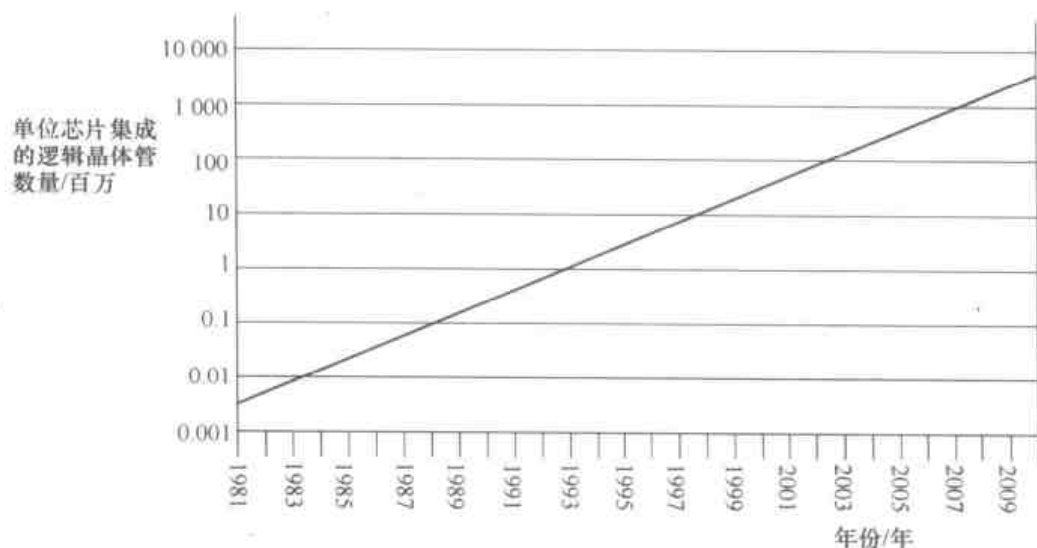


图 1-8 摩尔定律-1

又如图 1-9 所示,2002 年的芯片可以容纳 150 000 000 个晶体管,而 1981 年的主流芯片仅可以容纳 10 000 个晶体管,2002 年的容量是 1981 年的 15 000 倍。另外,设计的能力也有了很大的进步。

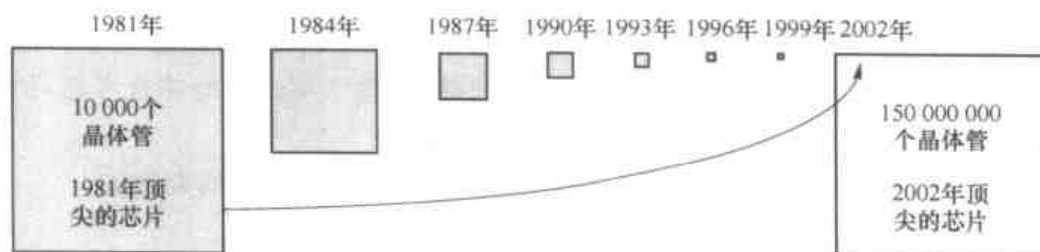


图 1-9 摩尔定律-2

在现代化的嵌入式技术中,越来越重视软硬件协同开发(Co-Design)。在过去,软硬件设计有着非常大的不同,一般都是相对独立的。而如今作为整个嵌入式的一部分,软硬件设计必须综合考虑,为共同完成某项功能相互配合,必须考虑整个设计度量指标:开发周期、成本、尺寸等等。

另外,处理器技术和 IC 技术的相对独立性也是嵌入式设计的一个考虑因素。基本的权衡考虑是:选择通用的还是特殊定制的技术。通用的趋势是朝着扩展性强、可维护性强、开发周期短、成本低发展,而特定技术的趋势是朝着更低的功耗、尺寸、成本高的方向发展。

现在的设计能力比以前可以说有着突飞猛进的发展,但是比 IC 芯片的晶体管容量的提高速度还有很大的差距,如图 1-10 所示。

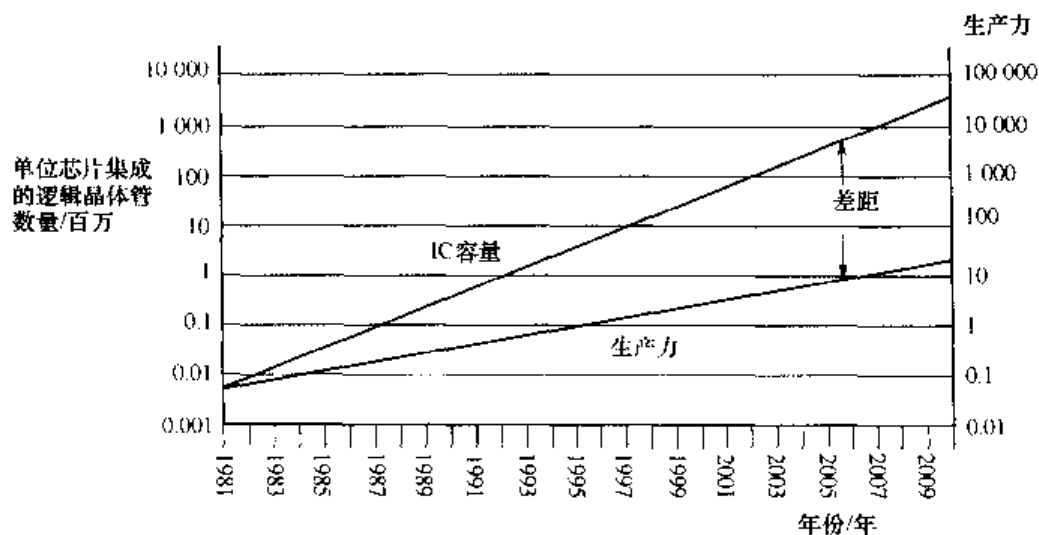


图 1-10 设计能力差距分析曲线

可以从图 1-10 看出这种差距随着时代的进步还在加大。

1981 年制造一个当时先进的芯片需要 100 人员,2002 年则需要 30 000 人员,设计的成本从 1 百万美元增加到了 300 百万美元。设计的能力增加了,但是同时设计的复杂程度也增加了,导致了开发的成本增加,需要投入的人力资本更多。但是需要注意的是,对于同一个项目,并不是说投入的人越多,就会使项目的开发周期缩短。这是因为开发人数越多,人员的管理也就越复杂,设计人员之间需要交流的也就越多,使整个项目的不确定因素增多。所以开发管理者必须充分考虑这个问题,从整个项目进行综合考虑,而不是单纯地投入或考虑某个方面,防止得此失彼。

如图 1-11 所示,设计人数增加到一定程度后,反而会使开发的周期增加。

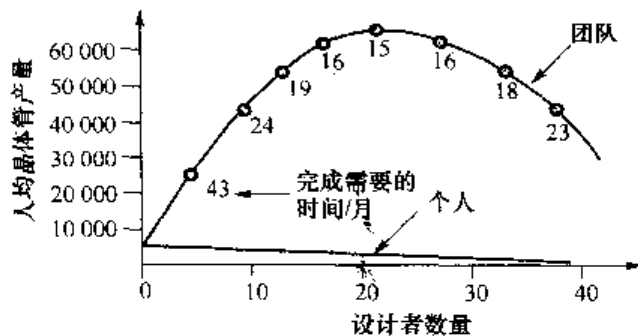


图 1-11 设计人数投入分析曲线

嵌入式系统大多数都是实时系统。学习嵌入式系统还应该具有一定的实时系统的基本知识。下一章将对实时系统做一些基本的介绍。

第2章 实时系统的基本理论

2.1 实时系统的历史

1970年,伊利诺斯州大学的 C. L. Liu 和 Jane 教授创建了实时系统试验室——RTSL (Real-Time Systems Laboratory),可以说是研究实时系统比较早的机构。C. L. Liu教授分析了单调速率调度算法(Rate Monotonic, RM)和时间限调度算法(Deadline),并将其作为 Jet Propulsion 实验室阿波罗项目计划(如图 2-1 所示)中的一部分。这项工作开辟了实时系统分析调度领域的先河。在 1999 年,他获得了 IEEE 实时系统委员会的第一个突出技术成就奖励,表彰他在实时系统计算方面的开创性工作。

1980 年到 1990 年初期间, Jane Liu 和 Kwei-Jay Lin 首先进行了模糊计算技术方面的研究。为了说明这项技术广泛的应用性,他们开发了一个相关的队列处理机,处理越多的数据,则会得到越精确的答案。他们还开发了调度模糊计算的算法,并且评估了这项技术在多媒体交通阻塞控制、控制系统和其他相关领域的效应。Jane Liu 教授发展了多处理器和分布式系统中点对点的调度技术,还发展了多处理器系统中时间不确定情况下的约束效应技术。并且, Jane Liu 教授和他的学生发展了性能评审技术 (Performance Evaluation Review Technique, PERTS)。



图 2-1 阿波罗探月项目

在 C. K. Liu 教授研究的基础上,CMU 的 Liu Sha 教授和他的同事发展了广义单调调度理论——GRMS(generalized rate monotonic scheduling)理论,这项理论提供了一个全面解决许多实时计算问题的方法。美国国防部 1991 年的一个题为“Software Technology Strategy”(“软件技术策略”)的报告中,其中 8~15 页提到:“A major payoff. ……System designers can use this theory to predict whether task deadlines will be met long before the costly implementation phase of a project begins. It also eases the process of making modifications to application software。”其意思就是说:设计者可以应用这项理论来预先判断在项目重要的执行阶段开始之前,是否会有任务截至时间到达的情况;这项理论还可以使得调整应用软件变得容易。可见美国国防部对这项理论有着多么大的评价。GRMS 理论被成功地应用到许多国家的高技术项目中,并且作为许多研究机构的实时计算课程的内容之一,也被大多数商业实时操作系统所支持。Liu Sha 被 IEEE 会员称为“实时计算方面的技术领导者和研究贡献者”。2001 年,他也得到了 IEEE 实时系统委员会的突出技术成就奖励。

实时系统越来越受到重视,来自美国总统信息技术顾问委员会(President's Information Technology Advisory Committee)的说法:国家依赖于软件。Liu Sha 的 Simplex 结构理论早就吸引了一些当时最大的高技术项目的兴趣:“必须开始开发相关的科学技术和方法,来构建一个健壮的系统——可靠的、安全的、可维护的、高成本效益的系统——来面对这些挑战。”尤其在航天航空领域,实时系统更是非常关键。

2.2 实时系统的基本概念及其性能衡量指标

2.2.1 实时系统的基本概念

牛津词典中定义了实时系统的概念:响应时间意义重大的任何系统(Any system in which the time at which output is produced is significant)。这句话表明计算的正确性不仅依靠结果还依靠响应的时间。

任何对外界的激励能够及时响应的系统就是实时系统。也就是说实时系统具有一定的时间限,如果没有了这个要求则肯定就不是实时系统了,可以称之为非实时系统。我们平常生活中有许多实时系统的例子,如将鼠标移动一下,则屏幕上的光标也会相应地移动一下,如果光标过了很长时间才移动,则客户肯定无法忍受,所以说鼠标的控制系统也是个实时系统。典型的实时系统还有:巡航导弹软件系统、智能驾驶舱显示系统、航线预定系统、工业过程控制系统、银行自动取款机系统等。实时系统广泛地应用于以下行业:安防系统、电话系统、自动机车控制、信号处理系统、雷达系统、自动生产系统、卫星系统、电子设备等。

1. 实时系统常用的术语

任务(Task):相关作业的集合,提供一些系统功能,记作 T ,例如接收一个数据帧是提供时间服务任务的一部分内容。

时限(Deadline):作业必须要完成的截止时间。

发起时间(Release time):作业准备好并可以开始执行的时间。

响应时间(Response time):作业开始执行,一直到作业完成(包括等待)所需的时间。

相对时限(Relative deadline):作业允许的最大响应时间。

绝对时限(Absolute deadline):作业必须要完成的具体时刻。

2. 硬实时和软实时系统

在这些实时系统的应用中,如果计算机或者控制电路对外界的激励必须严格地在时限到来前给出响应,不能有任何的延时,称之为强实时系统或者硬实时系统;如果对外界的激励给出的响应超过了时限,而对任务的完成虽然有影响但是没有太严重的后果,则称之为软实时系统。我们坐飞机或者汽车时,飞机的降落或者起飞等控制系统不能有任何的延时,否则可能会发生灾难性的后果;精确制导导弹也必须及时跟踪目标物体,如果有了一定的延时则无法完成任务,所以它们都是硬实时系统。我们开视频会议或者电话会议的时候,由于网络等条件的影响,图像、声音并没有完全实时地传输过来,但对整个会议的进行并没有太大的影响,所以我们可以认为它是软实时系统。参照图 2-2 可以看出两者的区别。

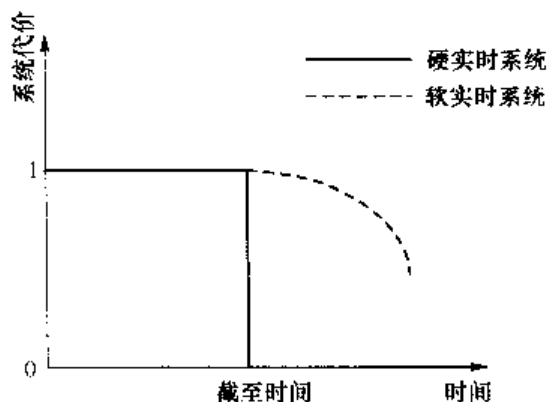


图 2-2 软实时系统和硬实时系统

实时系统的开发过程是比较复杂的,需要较复杂的开发环境和实时操作系统的支持。

作为实时系统最为关键的部分——实时操作系统内核,它的功能主要有:事件管理、时间管理、中断服务、实时调度等,如图 2-3 所示。

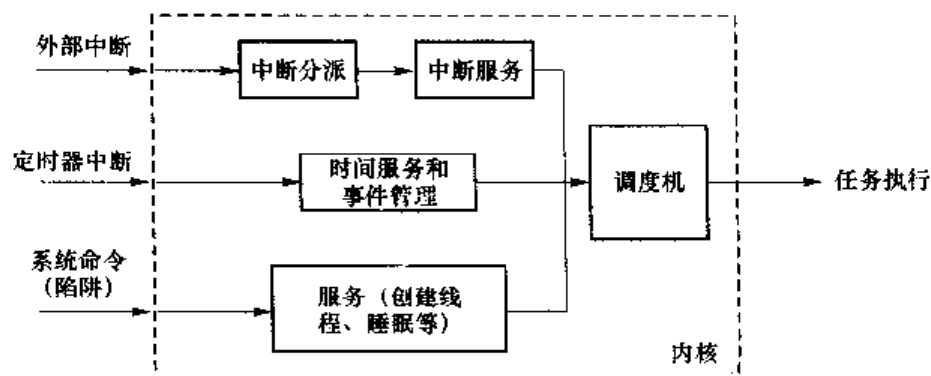


图 2-3 实时操作系统的内核

2.2.2 实时系统的性能衡量指标

有很多的实时系统应用在重要场合中,单纯的以时限来衡量实时系统还是不够的,或者说还没有完全体现出实时系统的特点,我们必须选择一些客观准确的性能评测标准来评测实时系统。

传统的实时系统指标包括可用率、可靠性、吞吐量等。可用率指系统运行时间的比率,即有效执行任务的时间和总时间的比;可靠性指给定时间内系统不发生失效的概率;吞吐量是单位时间内系统能处理的平均指令数。衡量实时系统的指标既要能体现系统各部分的特征,又要能体现各部分之间的相互影响和耦合关系。从这个要求来说,传统的衡量指标还不是很完善。

实时系统的性能应该与其控制的过程所引起的结果等因素密切相关,所以提出了可运行性和代价函数这两个新的指标。可用率和可靠性仅仅粗略地反映了系统的性能,而可运行性指标则需要分析实时系统在各种状态下不同层次的性能,具有更好的预测性和评估性。代价函数则给出了系统在不同状态下的响应性能评估手段。更详细的介绍可以参考 C. M. Krishna 的《实时系统》。

2.3 实时系统的任务调度

实时系统中存在着多个并发任务,它们都要使用处理器资源运行自己的程序。一般任务的调度由操作系统内核负责管理,将处理器资源公平合理地分配给各个任务,但是具体怎么样的分配方法算是公平合理的呢?这个完全取决于调度算法的优劣性。

2.3.1 任务调度的功能

在实时系统运行中,多个任务轮流使用处理器资源,所以说完成分配处理器的调度是系统中最为频繁的工作,任务调度的好坏直接影响着实时系统的性能和响应时间。

任务调度的主要功能有以下几个。(1)记录当前的任务状态,包括每个任务的临时变量、各种状态寄存器、执行时间、任务名等,将这些信息保存到任务控制块(TCB)中,一旦任务具备运行条件后,处理器会将任务控制块中的信息全部取出来继续执行任务。(2)根据调度算法决定就绪任务中哪一个任务能够执行,可以拥有处理器资源。调度算法有先来先服务(FIFO)、基于优先级的调度算法等。调度算法不同,组织就绪任务队列的方式也不同。先来先服务调度算法中,就绪队列要按等待时间从大到小的顺序排队;优先级调度算法中,就绪进程队列要按优先级的升序(或降序)方式排队。(3)分配处理器。将处理器资源分配给调度算法选中的某个任务。(4)任务运行结束后作善后处理工作,如保存当前信息等。

2.3.2 任务调度的性能准则

系统完成任务的调度,最关键的就是调度算法,但是调度算法的好坏怎么样去评价和

衡量呢?这就需要用分析调度算法的性能准则。

(1) 响应时间

这是实时系统中最为关键的一项性能指标。从用户的角度出发,响应时间越快则实时性越好,但对于整个系统的性能而言,常常需要以牺牲系统资源利用率为代价。所以说不能单纯地以响应时间或者某一个指标来评价任务调度算法。

(2) 处理机利用率

处理机作为系统中最关键的资源,其利用率大小是重要的性能指标。如何调度才能使处理机得到最大程度的利用是调度算法常常要考虑的问题。应当尽可能使处理机处于利用状态。

(3) 等待时间

等待时间指当任务处于就绪态后到进入运行态所花的时间,系统的等待时间越小,则调度算法性能越好。

(4) 吞吐量

吞吐量指系统在单位时间完成作业或者任务的平均数量,它可以用来衡量处理机的工作负担和工作效率,所以调度算法应当将吞吐量作为重要的指标进行设计。

(5) 其他指标

2.3.3 任务调度方式

调度方式从任务可否被抢占的角度来区分,可以分为可打断调度和不可打断调度。

1. 可打断调度

可打断调度是许多实时系统一个基本的功能。当一个任务执行时,如果有更高优先级或者更重要的任务就绪,则可以打断当前任务的执行状态,执行高优先级的任务,等到高优先级的任务执行完时,再恢复以前被中断的状态信息,继续执行低优先级的任务。当然,这里需要注意的一个问题就是要防止优先级倒置现象发生,VxWorks中有专门的预防策略(优先级继承,详细的说明见后文)防止这种情况发生。

2. 不可打断调度

不可打断调度指低优先级的任务占用资源后,即使有高优先级的任务处于就绪态,也不能打断低优先级的执行,将资源让出来。这种情况在系统级的任务执行情况中比较常见,任务级的调度中比较少。在这种方式中,一旦任务占用了处理机资源,系统不能剥夺其对处理机的使用,虽然简化了程序设计,但是系统性能较差,并且可靠性非常差,一旦有某个执行的任务出现错误则会导致整个系统瘫痪。

2.3.4 任务调度算法

根据调度的驱动方式,本书介绍两种经典的调度算法:基于时钟驱动的调度算法(Clock-Driven Scheduling)和事件驱动调度算法(Event-Driven Scheduling)。前一种调度一般用于周期性任务,根据任务特定的周期来决定调度哪个任务,执行情况如图2-4

所示。

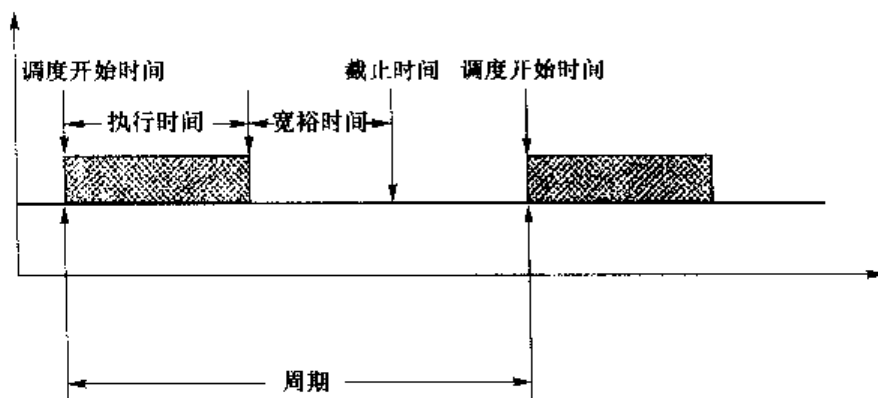


图 2-4 周期任务执行过程

时钟驱动任务调度中常用到的一些参数有：

Φ_i —— 任务 T_i 的起始就绪时间；

p_i —— 任务 T_i 的周期；

e_i —— 任务 T_i 的执行时间；

u_i —— 任务 T_i 的利用率, $u_i = e_i / p_i$ ；

D_i —— 任务 T_i 的时限。

可以用 Φ_i 、 p_i 、 e_i 、 D_i 4 个参数来表示周期性任务 T_i 的特性。例如：(1, 10, 3, 6)。

在默认情况下, 每个任务的起始就绪时间为 0, 相应的时限等同于它的周期, 这样, 用两个参数就可以表示周期任务的特征。例如 (0, 10, 3, 10) 就可以用 (10, 3) 来表示。

多个周期任务的调度也有优先级的问題。

事件驱动调度算法根据事件出现的先后以及任务的优先级安排任务的执行, 优先调度优先级较高的任务。它的任务执行时间一般是非周期性的, 如图 2-5 所示。

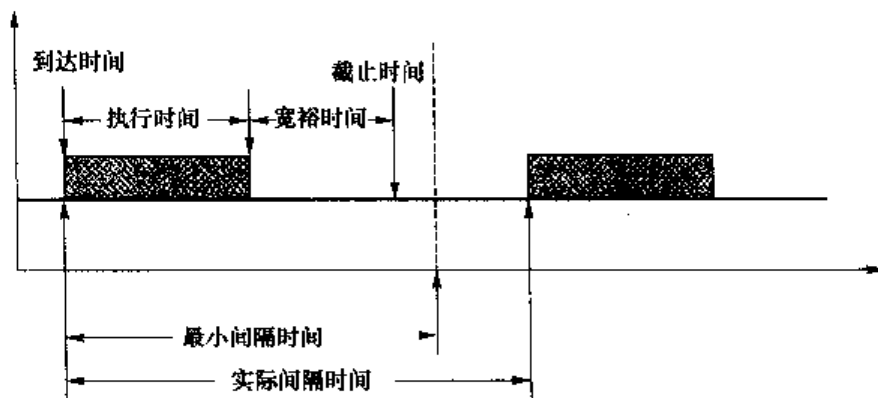


图 2-5 事件驱动的任务执行过程

下面重点介绍时钟驱动的任务调度。

1. 事件驱动调度算法

事件驱动的实现方式是依赖一个外部的硬件设备,通过产生中断的方式来为任务调度提供信号。这种调度方式又可细分为两种:集成事件驱动调度和非集成事件驱动调度。在集成事件驱动的调度系统中,每一个任务的启动都会有一个中断发布给处理器,中断的优先级与任务的优先级相对应,并且该中断只有在其优先级高于正在执行的任务时才会被处理器响应。所谓集成,就是中断的优先级是取决于其对应任务的优先级的。在非集成事件驱动的调度系统中,任务都是通过外部中断启动的。每一个新任务到来时,当前任务都会被中断,即中断的优先级与相关的到达任务的优先级是没有关系的。

2. 基于时钟驱动的调度算法

基于时钟驱动的调度算法一般用于周期性的任务调度,其他空闲时间可以用于非周期性的任务调度。在一个循环内,给周期性任务静态地分配了固定的时间间隔,这个循环时间也称为超周期(Hyperperiod),即各个任务周期的最小公倍数。为了便于任务分配,超周期要划分为同样长度的几个小周期。给定一个任务集和可能存在的大周期和小周期,可以根据最大网络流量的原则来检查是否存在一个可行的周期调度。

用 f 或者 frame(下同)表示小周期时间长度。在静态轮转调度算法中,计算小周期时间间隔有 3 个约束条件。

(1) 理想情况下,每个作业能够在单个 f 时间内开始并且完成执行。即 $f \geq \max(e_i)$, $1 < i \leq n$ 。 e_i 是第 i 个任务的执行时间。

(2) 小周期 f 时间尽可能短, f 应该能被超周期整除。

(3) 由于调度算法的调度执行发生在每个小周期的开始,并且作业在每个小周期内也不会被抢占,为了便于确定在时限到达前任务是否可以完成,要求在任务开始和到达时限之间至少有一个 f 的时间。用公式表达就是

$$2f - \gcd(p_i, f) \leq D_i \text{ for each } 1 \leq i \leq n \text{ (gcd 是最大公约数的缩写)}$$

例 2-1 确定表 2-1 所示的几个任务的小周期。

表 2-1 例 2-1 的几个任务的参数

任务	周期	截止时间	运行时间
T_1	p_1	D_1	e_1
T_1	4	4	1
T_2	5	5	1.8
T_3	20	20	1
T_4	20	20	2

解 在这个例子中,有 4 个任务,超周期 Hyperperiod=20。确定小周期的第一个限制要求 $f \geq 2$;第二个限制是 f 可以除周期的最大公倍数 20,所以 f 必须是 2, 4, 5, 10, 20 中的一个;第三个限制要求 $f \leq 4$ 。所以小周期 f 取值为 4。

例 2-2 需要对任务进行拆分才能正确调度的例子。

本例中 3 个任务的参数如表 2-2 所示。

解 根据第一个限制 $f \leq 4$,而根据小周期的第三个限制 $f \geq 5$ 。 f 无解。

在这种情况下,可以将一个任务拆分为几个子任务。子任务的执行时间之和就是用来任务的执行时间。变化后的任务参数如表 2-3 所示。任务 3 拆分为 3 个子任务,执行时间分别是 1、3、1。

表 2-2 例 2-2 各任务的参数

T_i	p_i	e_i	D_i
T_1	4	1	4
T_2	5	2	5
T_3	20	5	20

表 2-3 例 2-2 任务拆分后的参数

T_i	p_i	e_i	D_i
T_1	4	1	4
T_2	5	2	5
T_{31}	20	1	20
T_{32}	20	3	20
T_{33}	20	1	20

根据确定小周期的 3 个约束条件算得 $f = 4$ 。

可以画出一个超周期中任务调度的结果,如图 2-6 所示。

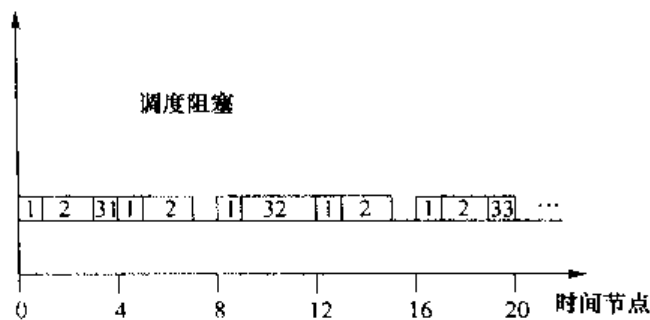


图 2-6 例 2-2 时钟驱动调度的结果

在图 2-6 的结果中,不存在任务被打断的情况,因为任务之间没有优先级的区别,任务调度只要满足时间参数的要求就可以了。

读者如果感兴趣的话,可以参考 1995 年 21 届 Euromicro 会议时的“The end of the line for static cyclic scheduling?”一文,作者是 N. Audsley,

K. Tindell 和 A. Burns。

3. 基于优先级的时钟驱动调度算法

多个周期任务的调度中,也可以有优先级的差别。在这样的任务调度中,除了要满足时间参数的要求外,还要满足优先级高低的要求。下面举几个例子来说明基于优先级的时钟驱动调度。

例 2-3 基于优先级的时钟驱动调度之例一。

各任务的参数如表 2-4 所示。

表 2-4 例 2-3 各任务的参数

任务	周期	截止时间	运行时间	相位
T_i	p_i	D_i	e_i	Φ_i
A(高优先级)	2	2	1	4
B(低优先级)	5	5	1	0

任务调度的结果如图 2-7 所示。

在这个例子中,任务 A 具有较高的优先级,任务 B 具有较低的优先级。任务 A 的执行周期为 2 个时间单位,截止时间为 2 个时间单位,运行时间为 1 个单位,启动就绪时间为 4;任务 B 的执行周期为 5 个时间单位,截止时间为 5 个时间单位,

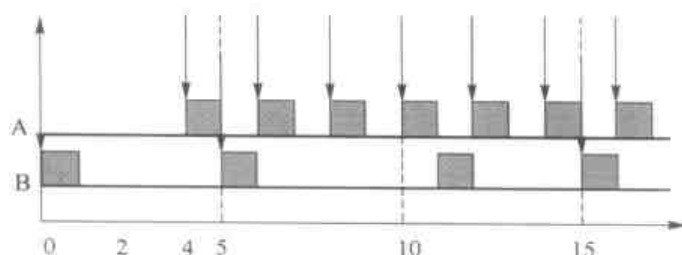


图 2-7 例 2-3 优先级驱动调度

运行时间为 1 个单位,启动就绪时间为 0。在时间点为 10 的时候,任务 A 和任务 B 都需要执行,但由于处理机一次只能处理 1 个任务,这就需要调度机通过调度算法调度任务。由于任务 A 的优先级比任务 B 的优先级高,所以任务 B 的执行时间推迟到了时间点 11 的时候才开始执行,但是,总的任务调度还是成功的,无论是任务的时间要求还是优先级的要求,都得到了满足。

例 2-4 基于优先级的时钟驱动调度之例二。

各个任务的参数如表 2-5 所示。

表 2-5 例 2-4 各任务的参数

任务	周期	截止时间	运行时间	相位
T_i	p_i	D_i	e_i	Φ_i
A(低优先级)	2	2	1	4
B(高优先级)	5	5	1	0

任务调度的结果如图 2-8 所示。

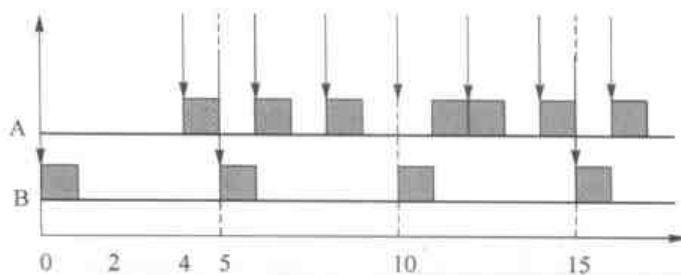


图 2-8 例 2-4 优先级驱动调度

在这个例子中,两个任务的时间参数和例 2-3 相同。只是优先级的分配改为任务 B 具有较高的优先级,任务 A 具有较低的优先级。在时间点为 10 的时候,任务 A 和任务 B 都需要执行,但由于任务 B 的优先级高,所以就先执行任务 B,然后再执行任务 A。

例 2-5 基于优先级的时钟驱动调度之例三。

各个任务的参数如表 2-6 所示。

表 2-6 例 2-5 各任务的参数

任务	周期	截止时间	运行时间	相位
T_i	p_i	D_i	e_i	Φ_i
A(高优先级)	2	2	1	0
B(低优先级)	5	5	2	0

任务调度的结果如图 2-9 所示。

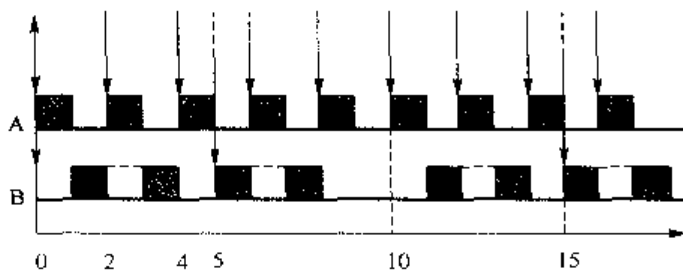


图 2-9 例 2-5 优先级驱动调度

在这个例子中,任务 A 具有较高的优先级,任务 B 具有较低的优先级。任务 B 的执行时间增加到 2,其余的时间参数除了任务 A 的启动就绪时间外,都和前两个例子相同。

在这个调度中,除了出现高优先级抢先的情况外,还出现高优先级打断低优先级的情况。在图 2-9 中,用虚线表示任务 B 被打断的情况。

尽管出现了高优先级的抢先和打断低优先级,但调度的结果,两个任务的时间参数还是得到了满足。

例 2-6 基于优先级的时钟驱动调度之例四。

各个任务的参数如表 2-7 所示。

表 2-7 例 2-6 各任务的参数

任务	周期	截止时间	运行时间	相位
T_i	p_i	D_i	e_i	Φ_i
A(低优先级)	2	2	1	0
B(高优先级)	5	5	2	0

任务调度的结果如图 2-10 所示。其中的“×”表示不能支持执行的任务单元。

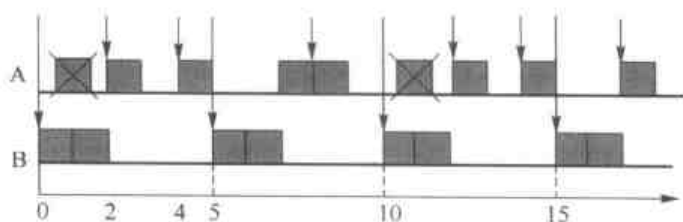


图 2-10 例 2-6 优先级驱动调度

在这个例子中,任务 A 具有较低的优先级,任务 B 具有较高的优先级。任务 A 和任务 B 的时间参数与例 2-5 相同。可以看到任务 A 由于优先级比较低,要被任务 B 抢先执行。任务 B 的执行时间是 2 个时间单元,等到任务 B 完成第一个周期任务的执行时,任务 A 的第一周期已经结束,没有时间留给任务 A 去执行了,这样任务 A 就丢失了一次任务的执行,调度失败。

这个例子说明:在任务的时间参数不变的情况下,由于优先级分配不同,可能导致任务调度的失败。

例 2-7 基于优先级的时钟驱动调度之例五。

各个任务的参数如表 2-8 所示。

表 2-8 例 2-7 各任务参数

任务	周期	截止时间	运行时间	相位
T_i	p_i	D_i	e_i	Φ_i
A(高优先级)	2	2	1	0
B(低优先级)	5	5	2.1	0

任务调度的结果如图 2-11 所示。

在上面这个例子中,任务 A 具有较高的优先级,任务 B 具有较低的优先级。任务 A 的执行周期为 2 个时间单位,截止时间为 2 个时间单位,运行时间为 1 个时间单位,启动时间为 0;任务 B 的执行周期为 5 个时间单位,截止时间为 5 个时间单位,运行时间为 2.1 个时间单位,启动时间为 0。调度结果也出现了失败:任务 B 没有足够的时间来执行任务。在图 2-11 中也用“×”指出了出错的时刻。

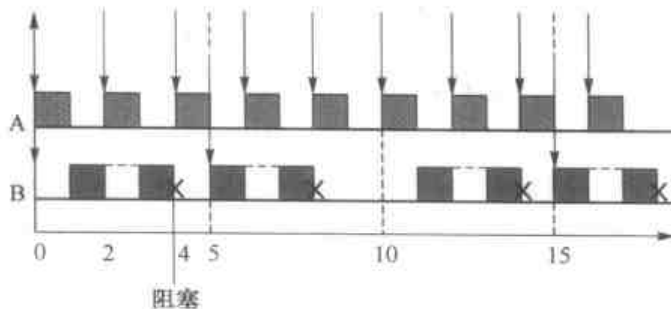


图 2-11 例 2-7 优先级驱动调度

这个例子中所有任务的利用率为: $u = e_1/p_1 + e_2/p_2 = 1/2 + 2.1/5 = 0.92$ 。可以看

出当 $u < 1$ 时,如果采用固定优先级的调度可能会导致一些任务分配不到处理机资源。

这样就引出了以下的概念:基于优先级的调度算法可以分为静态优先级调度算法和动态优先级调度算法。静态优先级调度算法给任务里的所有作业分配同样的优先级,或者说各个作业优先级不随时间发生变化,动态优先级调度算法给任务里的所有作业动态分配不同的优先级,优先级随时间发生变化。静态优先级调度算法又可以分为下面几个算法。

- (1) 根据执行时间(e_i)调度:最小执行时间先执行,最大执行时间先执行。
- (2) 根据周期(p_i)调度:最小周期任务优先执行,最大周期任务优先执行。
- (3) 根据任务利用率(e_i/p_i)调度:最小任务利用率优先,最大任务利用率优先。
- (4) 其他算法。

两个经典的基于优先级的算法:单调速率算法(Rate-Monotonic, RM)和最早时限优先算法(Earliest Deadline First, EDF),它们都是单处理器算法,也都是基于优先级的调度算法。RM 算法是典型的静态优先级算法,EDF 算法是典型的动态优先级算法。

RM 算法根据任务的周期大小进行调度,任务周期越小则优先级越高。^① RM 算法一般用于所有的任务集和时间限制条件都比较清楚的情况,然后产生一个单一的、固定的调度。而 EDF 等动态调度算法虽然知道当前处于激活状态任务的情况,但是并不知道新任务会在什么时间到达。动态调度算法通常在运行时刻进行。

4. 其他算法

(1) 先来先服务(FIFO)调度算法

先来先服务算法是按任务来到的先后次序进行调度的,换句话说,调度算法每次选择的任务是等待时间最久的,而不管任务的运行时间的长短。任务按照就绪队列的先后顺序调度并分配处理机,先进入就绪队列先分配处理机。一旦某个任务或者作业获得了处理机,则其他的任务或者作业就必须等下去,而不管等待任务或者作业的轻重缓急。这种调度算法突出的优点是实现简单,效率较低,在一些实际的系统和一般应用程序中采用这种算法的较多。这种算法很少单独使用,一般配合其他调度算法一起使用。

(2) 短作业优先调度算法

短作业优先调度算法考虑任务或者作业的运行时间,每次总是选择一个运行时间最小的任务或者作业调入内存(系统),给其分配处理机等资源。

在一般情况下这种调度算法比先来先服务调度算法的效率要高一些,而实现相对于先来先服务调度算法要困难些,如果任务或者作业的到来顺序及运行时间不合适,会出现长运行时间的任务一直得不到资源现象。例如,系统中有一个运行时间很长的作业 A 和几个运行时间小的作业,然后,不断地有运行时间小于 A 的作业的到来,这样,作业 A 就一直不可调度。

^① 更详细的说明可以参照 C. L. Liu 和 J. W. Layland 编著的“Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment”, JACM, Vol. 20, No. 1, pages 46-61, 1973.

(3) 可变时间片轮转调度算法

为了克服固定时间片调度算法的缺点,人们设计出一种可变时间片的调度算法,其思想是:时间片的大小是可变的,系统可根据系统中当前的任务数来确定时间片的大小。

这种算法从理论上克服了系统中任务数很少时系统开销大的缺点,但修改时间片的大小,统计系统任务的数量也需要消耗系统时间,还有一个调整时间片大小的周期,如果这个周期太大,系统开销很大,得不偿失。

(4) 均衡调度算法

均衡调度算法是一种更为理想化的调度算法,采用复杂的算法均衡考虑各种因素,而不是像前面几种调度算法只是简单地考虑一二个因素,但是实现起来更加困难,并且算法本身的开销有时会远远大于先来先服务和短运行时间任务优先调度算法的不足。

2.4 实时系统任务的互斥与同步

在多任务系统中存在着多个并发的任务,多个任务共享系统资源,共同完成某项任务和功能,所以任务之间并不是孤立的,而是有很多的相互制约或者依赖的关系,这种关系有两种基本形式:互斥和同步。有了这两种基本关系和相应的机制,任务之间才能更好地协同合作,要不然根本就无法建立起多任务的环境,系统的性能也就无法得到保障。

2.4.1 任务的互斥

多个并发任务可以共享系统资源,这就使得任务之间产生了相互的制约关系。但是如果对于这种关系不采取相应的保护机制或者处理机制的话,会使整个系统处于很危险的状态。比如说有多个任务需要访问某个 I/O 设备 D,任务 A 首先访问 D,但是当任务 A 还没有访问完 D 时,更高优先级的任务 B 也打算要访问 D,根据优先级调度的算法,任务 A 肯定要让出 I/O 设备,但这时候任务 A 还没有访问完设备 D,所以 I/O 设备可能会作出意想不到的反应。这里的 I/O 设备资源可以称之为临界资源。临界资源不但包括硬件资源,如 I/O 设备、内存、CPU 等,也包括软件程序中的变量、数据、BUFFER 区等。如果不对这些临界资源进行保护,采用互斥操作,必然会带来灾难性的后果。在软件程序操作中,如果一个任务在处理变量的时候被另一个任务终止并且同时操作这个变量,则变量的值会变得不可预测。所以访问临界资源必须要采取互斥操作,必须要等到任务处理完后再把资源让出来。如下面的程序:

```
TASK1() //任务 1
{
    for (;;)
    {
```

```

        Get signal;          //等待获取信号量,访问临界资源
        Perform x, y;        //x, y 是临界资源
        Free signal;
        Continue operation;
    }
}

TASK2( ) //任务 2
{
    for (;;)
    {
        Get signal;          //等待获取信号量,访问临界资源
        Perform x, y;        //x, y 是临界资源
        Free signal;
        Continue operation;
    }
}

```

以上的程序清楚地描述了对临界资源的互斥访问。我们可以采取一些假设使读者更清楚地明白这个过程:假如说每个任务获得互斥信号量时都将 $x = x - 1$, 只有成功地将变量 x 减 1 后才能正确地操作下一步,但是如果没有互斥保护,两个任务就有可能同时将 x 变量减 1, 出来的结果为 $x = x - 2$, 而 $x = x - 2$ 对于两个任务的下一步操作可能就是非法的,将会发生灾难性的后果。

2.4.2 任务的同步

早在 1990 年, L. Sha、R. Rajkumar、J. Lehoczky 教授就在《IEEE Transactions on Computers》杂志上,发表了文章“优先级继承算法:实时同步的方法”。同步问题也越来越得到研究人员的重视。

任务之间除了互斥地访问临界资源外,往往还需要同步来完成某项作业,需要同步的任务不能按顺序一直执行,必须要与其他合作的任务同步进行,等其他任务执行产生某个结果后再继续执行。比如下面的程序:

```

TASK1( ) //任务 1
{
    for (;;)
    {
        Perform operation; //执行 1 步操作
    }
}

```

```

Free signal1;          //释放同步信号量 1
Wait for signal2 from task2; //等待任务 2 的信号量再进行下一步操作
Continue operation;
}
}
TASK2()//任务 2
{
    for ( ; ; )
    {
        Perform operation;
        Free signal2; //释放同步信号量 2
        Wait for signal1 from task1; //等待任务 1 的信号量再进行下一步操作
        Continue operation;
    }
}

```

在上面的程序中,存在着两个同步信号量 signal1 和 signal2,每个任务操作完一步后必须等待另外一个协同的任务释放信号量后才可以继续进行。又如我们在软盘上将文本文件进行变换并重新写入软盘的 3 个任务 Task1(读软盘)、Task2(变换文本文件)、Task3(写软盘),这 3 个任务是一个合作的过程,它们互相合作才可以得到正确的结果。Task2 只有在 Task1 完成并且写入缓存后,才可以将缓存中的数据取走并且进行变换,而 Task3 也必须等到 Task2 变换完并且成功写入到缓存后,才可以将数据写入到软盘中,这是个同步的过程,每个任务不能自己单独执行,必须共同合作实现一个目标。

任务无论是互斥还是同步,之间都有一种相互制约的关系,但是它们的不同之处在于:

(1) 互斥的各个任务可以自己独立执行,并且得到正确的结果;而同步的任务则不行,同步的任务必须协同工作才可以得到正确的结果;

(2) 互斥的各个任务执行的时间可以没有时序要求,只是在访问临界资源的时候才必须等待;同步的任务则有严格的时序要求,不能异步进行。

另外,许多应用程序需要联合访问临界资源,为了同步访问一个临界资源,使用信号量进行控制。如果一个任务想要锁住一个已经被另一个任务锁住的二进制信号量,则该任务将被阻塞。这里出现的问题包括:资源共享引起的阻塞对任务调度的可行性的影响有多大;调度算法是不是可以经过调整后支持资源共享的任务调度等。

2.4.3 优先级倒置现象

如果若干任务需要访问临界资源,在临界资源被封锁的情况下,有可能出现优先级低

的任务比优先级高的任务更早的完成,这种现象称为优先级倒置(Priority Inversion)。

例 2-8 优先级倒置示例。

表 2-9 列出了 3 个任务的时间参数。

表 2-9 优先级倒置示例的各任务参数

T_i	Φ_i	P_i	D_i	e_i
T_1	3	50	10	5
T_2	0	500	500	250
T_3	0	3 000	3 000	1 000

假设任务 1 和任务 3 共享某个数据,访问数据需要使用信号量 s ,任务 T_1 具有最高优先级,任务 T_3 具有最低优先级。每个任务执行下面的代码:

```
do local work (L);
sem_wait(s) (P(s));
access shared resource (R);
sem_signal(s) (V(s));
do more local work (L);
```

任务的执行过程将是:

- (1) 任务 T_3 执行 Local Work(L);
- (2) 任务 T_3 访问临界资源(R);
- (3) 任务 T_1 启动,打断任务,执行 Local Work(L);
- (4) 任务 T_1 要访问临界资源,因为被任务 T_3 占用,任务 T_1 阻塞;
- (5) 任务 T_2 启动,打断任务 T_3 的执行,一直持续执行 250 个时间单元;
- (6) 任务 T_3 释放临界资源(R);
- (7) 任务 T_1 占用临界资源,完成任务的执行,但是已经大大超过时限 10;
- (8) 任务 T_3 继续完成 Local Work(L)。

最后的结果是高优先级 T_1 的任务没能首先执行,而是比优先级较低的任务 T_2 结束得还要晚,导致时限大大地超过,产生了优先级倒置。

优先级倒置的现象在同步操作中比较常见。所谓优先级倒置是指临界资源的任务在到达其临界区时,因为临界资源的竞争而造成高优先级任务被低优先级任务阻塞的情况。图 2-12 是这种现象的一般图示。

在图 2-12 中,Task1 具有较高的优先级,Task3 具有较低的优先级,Task2 具有中间的优先级。Task3 锁住了临界资源 R_1 ,当 Task1 准备好时,抢占了 Task3 的处理机资源运行,随后需要访问临界资源 R_1 。但是临界资源此时被 Task3 占据,所以 Task1 阻塞,等待 Task3 释放临界资源 R_1 。这时 Task2 准备好可以运行,于是抢占了 Task3 的处理机资源进行运行,使得 Task1 多等待了一段时间,直到 Task3 释放临界资源 R_1 。

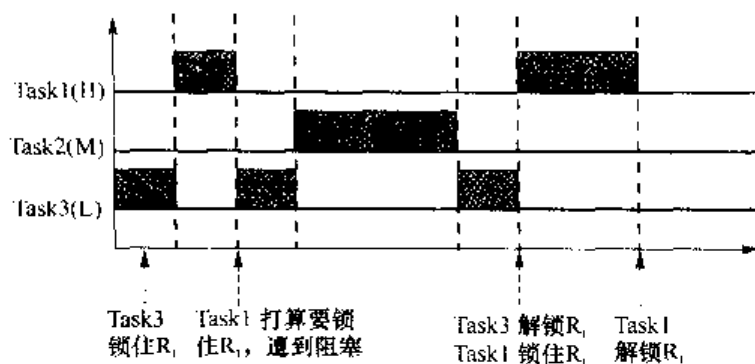


图 2-12 优先级倒置

优先级倒置违反了优先级调度算法的基本原则,必须设法解决。解决优先级倒置的方法很多,下面介绍两个相关的算法。

2.4.4 优先级继承算法

优先级继承协议的基本思想是:当一个任务进程 Task 阻塞一个或多个优先级更高的任务进程时,所有阻塞的任务都被存入一个优先级队列中。将任务 Task 的优先级暂时提高到被它阻塞的所有任务中的最高优先级,从而使保证 Task 对于临界资源的访问不会被打断。目的是让它尽快地结束对临界资源的访问,以便其他任务可以及时使用临界资源。

这种算法不影响与临界资源无关的其他高优先级的任务的执行。当任务 Task 退出临界区时就恢复原来的优先级。

优先级的继承是可以传递(Transitive)的。例如,假设 Task1, Task2, Task3 是 3 个优先级从高到低排列的任务,如果 Task3 阻塞 Task2, Task2 阻塞 Task1, 则 Task3 将通过 Task2 而继承 Task1 的优先级。

优先级继承算法的执行过程可参见图 2-13。

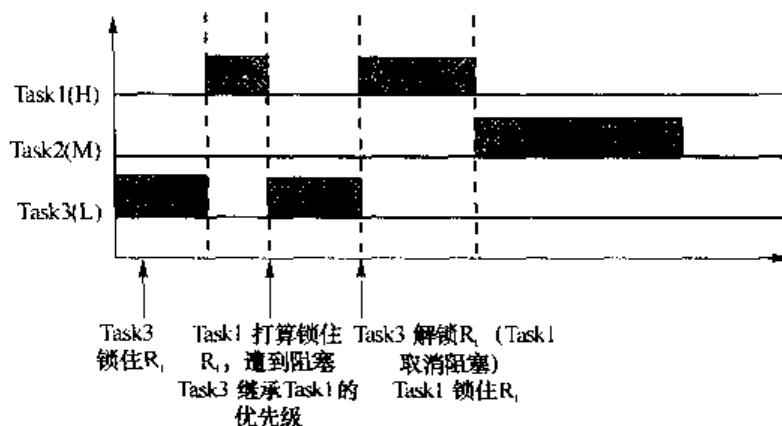


图 2-13 优先级继承

在图 2-13 中, Task1 具有较高的优先级, Task3 具有较低的优先级, Task2 具有中间

的优先级。Task3 锁住了临界资源 R_1 , 当 Task1 准备好时, 抢占了 Task3 的处理机资源运行, 随后需要访问临界资源 R_1 , 但是此刻临界资源被 Task3 占据, 所以 Task1 阻塞, 等待 Task3 释放临界资源 R_1 。这时候因为 Task3 具有优先级继承协议, 所以 Task3 的优先级为动态变化, 被赋予和 Task1 相同的优先级。这时候虽然 Task2 已经准备好可以运行了, 但是前两个任务此刻的优先级都比它高, 所以 Task2 被阻塞。Task1 没有多等待时间, 等 Task3 释放临界资源 R_1 后得以继续执行。

采用优先级继承算法后, 高优先级任务 Task1 虽然也被阻塞了一段时间, 但是它的执行不会受到较低优先级任务 Task2 的影响, 任务执行的结束时间也会大大提前, 基本上避免了优先级倒置现象的出现。

注意, 在优先级继承算法中, 占用临界资源的低优先级任务只是在高优先级任务也要访问临界资源时才会继承高优先级任务的优先级, 而不是占用了临界资源后就立即继承被阻塞高优先级任务的优先级。

阻塞(Blocking)分为两种类型: 直接阻塞(Direct)和间接阻塞(Indirect)。在直接阻塞情况时, 如果低优先级的任务在占有临界资源时, 它会直接阻塞高优先级的任务; 间接阻塞时, 如果低优先级在占有临界资源时, 低优先级的任务继承了高优先级任务的优先级, 中优先级的任务会被阻塞, 这被称为间接阻塞。

优先级继承协议在一定程度上解决了优先级反转的问题, 实现起来也比较简单。优先级的继承和恢复都应该是不可分割的原子操作, 以保证实时系统内部数据结构的一致性。这一过程可以在用户程序中实现, 也可以由内核的调度程序来自动实行。但是, 该方法存在两个重要的缺陷。

首先是会造成死锁(Deadlock)。例如, 假设在 t_1 时刻, 任务 Task2 占用信号量 s_2 , 并进入其临界区; 在 t_2 时刻, 任务 Task2 试图在 s_2 的临界区内占用信号量 s_1 , 以访问另一个资源。若此刻优先级更高的任务 Task1 已经就绪, 并且 Task1 抢先占用了信号量 s_1 , Task2 就被阻塞。若接下来 Task1 试图再占用信号量 s_2 , 由于 s_2 已经被任务 Task2 占用, 所以任务 Task1 也被阻塞。也就是说 Task2 和 Task1 互相阻塞了对方, 发生了死锁。

另外, 由于允许任务在临界区内嵌套地锁定其他信号量, 因此可能引起连续的阻塞链(Blocking Chain), 导致因为阻塞而产生的延迟的积累, 也可能使得高优先级的任务不能及时的执行。

要解决死锁和阻塞链必须对优先级继承算法继续进行改进。

2.4.5 优先级上限算法

优先级上限算法(Priority Ceiling Protocols, PCP)是 Sha 等人在基本的优先级继承算法的基础上提出的一个改进的方案。

与优先级继承算法不同的是, 优先级上限算法定义了信号量的优先级上限(Priority Ceiling)。一个信号量的优先级上限与可能锁定该信号量的所有任务中优先级最高的任

任务的优先级相等。调度任务的时候不仅要比较任务之间的优先级,还要比较信号量的优先级上限。当一个进程J要进入一个临界区的时候,如果J的优先级不比已经被其他进程锁定的所有信号量的优先级上限高,则J被阻塞。

这种协议防止了死锁现象和递归阻塞,支持互斥访问资源。在运行时间,如果一个任务想要锁住一个信号量 s ,它的优先级必须严格地比当前被其他任务锁住的信号量优先级上限高(除非这个任务拥有的信号量具有更高的优先级上限)。如果这个条件不满足,任务就阻塞在 s 上。当任务在信号量 s 上阻塞的时候,拥有 s 信号量的当前任务继承了被阻塞的任务的优先级。

优先级继承协议和优先级上限协议最大的不同是:优先级继承协议允许任务在资源没有被占的时候锁住资源,而优先级上限协议则可能不允许这样的访问。请看以下的例子。

例 2-9 原始优先级上限算法如何防止死锁。

有两个任务:任务A和任务B。任务A有高优先级,启动时间为3;任务B优先级较低,启动时间为0。两个任务都要访问临界资源 R_1 和 R_2 。

两个任务的执行过程是:

任务A: L R_1 R_1 R_2 R_2 L L

任务B: L L R_2 R_2 R_1 R_1 L L

(注:其中L表示正常访问;R表示所访问的临界资源;每一小格表示一个时间单元。下同。)

按照原始优先级上限算法调度的结果如图2-14所示。

这是一个有可能产生两个任务死锁的例子。但是,使用优先级上限算法后,死锁就可以避免。

由于高优先级任务A都要占用对临界资源进行保护的信号量 s_1 、 s_2 ,所以 s_1 、 s_2 的优先级上限就是任务A的优先级。任务执行的过程如下:

(1) 任务B先开始执行,并且占用信号量 s_2 ;

(2) $t=3$ 时,任务A开始执行;

(3) 当任务A要占用信号量 s_1 时,尽管相应的资源并没有被占用,但是因为任务A的优先级并没有高于 s_2 的优先级上限,任务A被阻塞;同时,任务B的优先级将继承任务A的优先级;

(4) 任务B继续占用信号量 s_2 ,释放信号量 s_2 ;占用信号量 s_1 ,再释放 s_1 ;

(5) 任务A开始占用信号量 s_1 ,释放 s_1 ;再占用信号量 s_2 ,释放 s_2 ,直到任务完成;

(6) 任务B继续完成未完成的任务。

所谓“原始”优先级上限算法,是在任务A被阻塞时,任务B的优先级才上升为任务

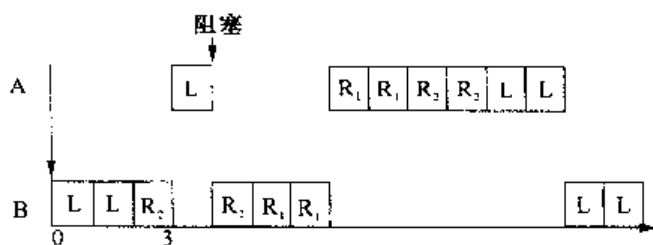


图 2-14 按照原始优先级上限算法调度任务

A 的优先级。

另外一种“立即”优先级上限算法,是在任务 B 占用信号量 s_2 时,它的优先级就“立即”上升为也要使用信号量 s_2 的任务 A 的高优先级。它的效果可以通过以下的例子来说明。

例 2-10 立即优先级上限算法调度任务。

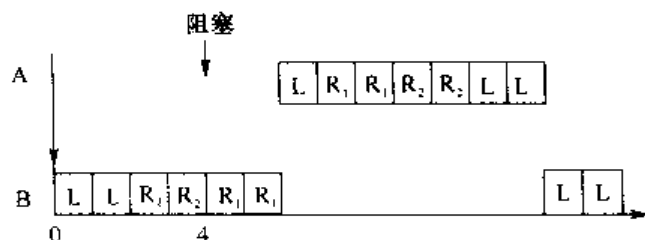


图 2-15 按照立即优先级上限算法调度任务

两个任务的参数和执行的具体要求,都和例 2-8 相同。调度结果如图 2-15 所示。

任务 B 在占用了信号量 s_2 后,就“立即”继承可能被信号量 s_2 阻塞的任务 A 的高优先级一次,当 $t=4$ 时,任务 A 的优先级并没有高于任务 B 的动态优先级,所以任务 A 立即被阻塞。其余的过程和例 2-8 相似。

2.5 死锁现象

上一节提到了死锁现象。死锁作为实时系统资源管理的重要考虑对象,有必要对它进行进一步的研究并且采取相应的措施。

2.5.1 死锁的概念

当两个任务或者进程并发执行时,任务 T_1 占有资源 R_1 、任务 T_2 占用资源 R_2 , T_1 要求 R_2 ,但由于 T_2 已占有 R_2 而得不到 R_2 ,所以 T_1 只有等待;同时 T_2 申请资源 R_1 ,但由于 T_1 已占有 R_1 而得不到 R_1 ,任务 T_2 只有等待,就出现了死等的情况。死锁用一句话总结就是任务或者进程之间由于互相占用资源,想要索取对方资源但对方又不肯让出自己的资源,导致谁也得不到自己想要的资源使系统任务执行不能继续的现象。

用程序来表示就是:

```
TASK1() //假设任务 T1 占有资源 R1
```

```
{
```

```
    for ( ; ; )
```

```
    {
```

```
        Perform operation; //执行 1 步操作
```

```
        Wait for signal2 from task2; //等待任务 2 的信号量再进行下一步操
```

作,想要索取资源 R2

```
        Continue operation;
```

```
    }
```

```

}
TASK2( )// 假设任务 T2 占有资源 R2
{
    for ( ; ; )
    {
        Perform operation;
        Wait for signal1 from task1; //等待任务1的信号量再进行下一步操作, 想要索取资源 R1
        Continue operation;
    }
}

```

2.5.2 死锁的起因

发生死锁是由于资源分配不合理,或者资源不够造成的,具体分析起来,主要有以下几个原因。

1. 互斥条件

产生死锁的一个主要原因是因为多个任务竞争有限的资源。这些资源(如打印机、光驱、键盘等)都是不能共享的临界资源,在一个时刻只能被一个任务或者进程所占有,换句话说就是资源具有互斥性,不可以被多个任务或者进程同时占有。

2. 不可抢占的限制

如果资源是不可以抢占的,也容易形成死锁。否则高优先级的任务可以从低优先级的任务那里获取资源,就不可能形成死锁。

3. 资源分配不完全

产生死锁的原因还在于进程或者任务在运行中只得到了部分需要的资源,同时还要请求其他资源。如果在任务或者进程一开始就分配了所有所需的资源,任务也不会因为等待资源而阻塞,也就不可能发生死锁。

4. 环路条件

任务之间按照各自的速度独立推进而导致它们之间形成了对几个资源的循环等待关系。

2.5.3 解决死锁的策略

解决死锁的策略可以分为以下几个方面:死锁的预防、死锁的避免、死锁的检测和恢复。其本质是打断形成死锁的条件。下面进行具体分析。

1. 死锁的预防

死锁的产生是由于对资源的不合理分配或者资源不够,处理对象是具有独占性的资

源。如果能够对资源进行合理分配,完全可以防止死锁。这里有两种方法可以预防死锁:静态资源分配法和有序资源分配法。

静态资源分配法就是在任务或者进程运行前将其所需的资源全部分配给它,如果资源不能全部满足则不分配。静态资源分配法不会出现部分分配,如果资源够用则分配,不够用则不分配。但是这种分配方法也有很多缺点,比如效率低下,如果任务一直得不到所需的资源则一直等待,并且如果某个任务很长时间占用资源就会导致其他的任务一直等待。有序资源分配法则是对静态资源分配法的改进,并不是一开始就给任务分配好资源,而是动态地分配资源,但这种动态分配是有一定次序的,是按照某种顺序的分配。

2. 死锁的避免

通过破坏产生死锁的必要条件来预防死锁,系统是要付出一定的代价的,会引起资源利用率的下降。实际上,即使存在着发生死锁的某些条件,如果合理地处理好资源的分配,也可以起到避免死锁的作用。避免死锁算法中最有代表性的算法是 Dijkstra E. W 于 1968 年提出的银行家算法。该算法需要检查申请者对资源的最大需求量,如果系统现存的各类资源可以满足申请者的请求,就满足申请者的请求。这样申请者就可很快完成其计算,然后释放它占用的资源,从而保证了系统中的所有任务或者进程都能完成,所以可避免死锁的发生。

3. 死锁的检测和恢复

由于死锁的避免和预防需要很大的系统开销,所以在重视资源利用率的系统中,需要权衡这个问题。当死锁发生时可以有办法检测到并且将任务恢复到原来的状态,这也是一种解决问题的策略和方法。

常用的死锁检测法就是时刻监测任务的运行状态,一旦监测到有异常情况,则立刻进行处理,或者给发生死锁的任务提供它所需的资源,或者将部分任务撤销出来。

2.6 实时系统任务的通信机制

实时系统任务之间的通信机制是非常多的,除了我们在前几节看到的信号量通信外,还有很多种通信机制:消息队列、信号、管道等等。

信号量是实现任务同步和互斥的重要手段,信号量相对于其他的通信手段来说实现速度最快。例如实时系统 VxWorks 提供了 3 种不同功能的信号量:二进制信号量、互斥信号量、计数信号量,广泛地用于任务的同步和互斥。

信号量方式的通信虽然快速,但是还远远不能满足任务之间的通信需要。通过消息队列方式通信也是实时系统中最常见的通信方式。任务之间通过消息队列收发消息,可以起到很好的交互作用。例如 VxWorks 系统中提供了两种消息队列库:msgQLib 和 mqPxLib。msgQLib 提供了标准的 VxWorks 消息队列,而 msPxLib 提供了 POSIX 消息

队列,通过这两个库的支持,VxWorks 操作系统提供了良好的消息队列通信机制。

信号是由事件引发的软中断,即中断并非由外界硬件产生,而是由内部程序自己产生。需要特别注意的是,信号的处理程序并非在中断的上下文中,而是在接收任务本身的上下文中进行处理。由信号引起的任务通信也非常常见。

另外,实时系统的任务之间还可以通过共享内存来实现通信,共享的对象可以是单变量、数组、链表等等,但是要特别注意的是对共享的内存对象操作时,一定要注意保护,否则会发生意想不到的结果。

实时系统的通信机制一般都是非常丰富的,这跟实时系统本身的实时性、可靠性等要求是有关系的,任务开发中应该多结合一些通信手段。具体的例子可以参见 4.3 小节,其中详细讲述了以 VxWorks 操作系统为代表的实时系统的通信机制。

第3章 嵌入式系统硬件基础

3.1 ARM 体系结构

3.1.1 ARM 的概念

ARM(Advanced RISC Machines),顾名思义,是采用高端 RISC 架构的计算机。它既是一个公司的名称,也可以认为是一类技术的名称,或者是一类处理器的总称。

ARM 公司于 1991 年成立于英国剑桥,主要出售芯片设计技术的授权。采用 ARM 技术知识产权(IP)内核的微处理器,即我们通常所说的 ARM 微处理器,已遍及工业控制、消费类电子产品、通信系统、网络系统、无线系统等各类产品市场。基于 ARM 技术的微处理器应用约占据了 32 位 RISC 微处理器 75% 以上的市场份额,可以说 ARM 处理器是日前在嵌入式领域里应用最广泛的 RISC 微处理器结构。ARM 处理器以其低成本、低功耗、高性能的特点占据了嵌入式系统应用领域的领先地位。可以说研究 ARM 技术对于掌握 RISC 架构的处理器是非常有意义的。

但是,ARM 公司本身并不生产微处理器,而是将处理器内核授权给各大公司。授权的公司开发出适用于各种应用场合的处理器,如三星的 S3C4510B,ATMEL 公司的 AT9140162 等等。这样做既使得 ARM 公司可以专注于 ARM 技术的开发,集中优势保证自己核心技术的发展,又使得各大公司可以根据客户的需求,开发出各种类型的产品,使 ARM 技术得到了广泛的支持和认可,更具有竞争力。

3.1.2 ARM 处理器的架构特点

ARM 处理器由于属于 RISC 处理器,所以也具有 RISC 处理器的特性,同时还有一

些通用 RISC 处理器所没有的特性,如具有两种指令状态:32 位的 ARM 指令和 16 位的 THUMB 指令等。

传统的复杂指令集计算机(Complex Instruction Set Computer,CISC)结构有其固有的缺点:随着计算机技术的发展会不断引入新的复杂的指令集,为支持这些新增的指令,计算机的体系结构会越来越复杂,然而,在 CISC 指令集的各种指令中,其使用频率却相差悬殊,大约有 20% 的指令会被频繁使用,占整个程序代码的 80%,而余下的 80% 的指令却不经常使用,在程序设计中只占代码的 20%。显然,这种结构是不太合理的。考虑到这种设计的不合理性,1979 年美国加州大学伯克利分校提出了精简指令集计算机(Reduced Instruction Set Computer,RISC)的概念。RISC 并非只是简单地去减少指令,而是把着眼点放在了如何使计算机的结构更加简单合理,以利于提高运算速度。RISC 结构通过优先选取使用频率最高的简单指令,避免复杂指令;将指令长度固定,指令格式和寻址方式种类减少;以控制逻辑为主,不用或少用微码控制等措施来达到上述目的。虽然 RISC 指令集具有这样或那样的优点,但这并不代表就可以取代 CISC 指令集或者说它比 CISC 指令集更优秀,实际上,两种指令集各有各的优点,主要看应用于哪种场合。现代了指令集发展趋势是尽可能综合两种指令集的优势。

综上所述,ARM 处理器架构的特性有:

- (1) 固定长度的指令结构;
- (2) 两种指令集——ARM 和 THUMB,可以适应于多种应用场合和架构;
- (3) 寻址方式简单,所有的寻址只由寄存器的内容和指令域决定,并且在这个基础上,增加了多寄存器的加载和存储指令,使得数据的操作效率更高;
- (4) 所有指令的条件执行实现了快速的代码操作;
- (5) 大量使用寄存器,数据处理指令只对寄存器进行操作,只有加载/存储指令可以访问存储器,以提高指令的执行效率;
- (6) 在循环处理中使用地址的自动增减来提高运行效率。

3.1.3 主流 ARM 处理器及使用场合

ARM 微处理器发展至今,在 ARM 公司的领导下,已经开发出了多种系列、可以适用于多种场合的产品,主要分为:

- (1) ARM7 系列;
- (2) ARM9 系列;
- (3) ARM9E 系列;
- (4) ARM10E 系列;
- (5) SecurCore 系列;
- (6) Intel 改进型系列。

其中,ARM7、ARM9、ARM9E 和 ARM10 为 4 个通用处理器系列,每一个系列提供一套相

对独特的性能来满足不同应用领域的需求。如 ARM7 属于低端产品,在网络通信、控制领域非常常见;而 ARM9 以上的处理器属于中高档的产品,在多媒体行业比较常见;SecurCore 系列专门为安全要求较高的应用而设计。具体的各种系列特点将在下面进行说明。

1. ARM7 微处理器系列

ARM7 系列微处理器是 32 位的 RISC 处理器,跟其他系列相比,ARM7 是一款比较低端、廉价但性价比比较高的产品,适合于那些功能比较简单,成本要求比较低的产品开发。本书所介绍的 Samsung 公司的 S3C4510B 即属于该系列的处理器。ARM7 系列包括 ARM7TDMI、ARM7TDMI-S、ARM720T、ARM7EJ 4 种内核,S3C4510B 属于第一种 ARM7TDMI 内核。

TDMI 的基本含义为:

T——支持 16 位压缩指令集 THUMB;

D——支持片上调试(Debug),使处理器能够停止以响应调试请求;

M——内嵌硬件乘法器(Multiplier);

I——嵌入式 ICE,支持片上断点和调试点。

总结起来,ARM7 具有以下特点:

- (1) 适应于功耗比较低的场合;
- (2) 代码密度高;
- (3) 具有 ARM/THUMB 双指令集,适应于不同场合;
- (4) 支持包括 VxWorks、Linux 和 Windows CE 等多种实时嵌入式系统,使用户可以有多种选择,满足不同类型的客户;
- (5) ARM7 的指令也可以很好地过渡到更高级别的系统中,方便用户进行升级;
- (6) 主频最高可达 130 MIPS,性价比比较高,可以满足多数应用场合的需要;
- (7) 三级流水线。

2. ARM9 微处理器系列

ARM9 系列微处理器在性能上比 ARM7 高一个档次,属于性能比较好的一个系列,在包括多媒体领域如摄影、图像处理、高端数码设备、MP3 等在内的许多场合都非常适用,它的特点主要有:

- (1) 具有 ARM/THUMB 双指令集,适应于不同场合;
- (2) 支持 MMU 虚拟内存管理,支持更大容量的内存;
- (3) 支持包括 VxWorks、Linux 和 Windows CE 等多种实时嵌入式系统,使用户可以有多种选择,满足不同类型的客户;
- (4) 支持数据 Cache 和指令 Cache,具有更高的指令和数据处理能力;
- (5) 五级流水线。

ARM9 系列微处理器包含 ARM920T、ARM922T 和 ARM940T 3 种类型。

3. ARM9E 微处理器系列

与 ARM9 微处理器相比,ARM9E 系列微处理器更加综合,并且增加了一些新的特

性,主要是包含了 DSP 处理浮点数的算法和能力,适合于那些需要同时使用 DSP 和微控制器的应用场合,并且使用单一的处理器的内核提供了微控制器、DSP、Java 应用系统的解决方案,极大地减少了芯片的面积和系统的复杂程度。当然,ARM9E 也属于比较高端的产品,在数码领域、图像处理等多媒体领域应用比较广泛,得到了众多厂商的支持。

ARM9E 系列微处理器的主要特点如下:

- (1) 支持 DSP 指令,适合需要同时需要浮点操作的应用场合;
- (2) 具有 ARM/THUMB 双指令集,适应于不同场合;
- (3) 支持 MMU 虚拟内存管理,支持更大容量的内存;
- (4) 支持包括 VxWorks、Linux 和 Windows CE 等多种实时嵌入式系统,使用户可以有多种选择,满足不同类型的客户;
- (5) 支持数据 Cache 和指令 Cache,具有更高的指令和数据处理能力;
- (6) 五级流水线;
- (7) 主频最高可达 300 MIPS。

ARM9E 系列微处理器包含 ARM926EJ-S、ARM946E-S 和 ARM966E-S 3 种类型。

4. ARM10E 微处理器系列

ARM10E 系列微处理器可以说是目前所有 ARM 系列中处理性能最高的一款处理器。由于采用了新的体系结构,功耗非常低,但同时又拥有出色的处理能力,ARM10E 处理器可以适用于多种需求的场合,如数码影视、无线传输、图像处理、网络传输等领域。

ARM10E 系列微处理器的主要特点如下:

- (1) 支持 DSP 指令,适合需要同时需要浮点操作的应用场合;
- (2) 具有 ARM/THUMB 双指令集,适用于不同场合;
- (3) 支持 MMU 虚拟内存管理,支持更大容量的内存;
- (4) 支持包括 VxWorks、Linux 和 Windows CE 等多种实时嵌入式系统,使用户可以有多种选择,满足不同类型的客户;
- (5) 支持数据 Cache 和指令 Cache,具有更高的指令和数据处理能力;
- (6) 六级流水线;
- (7) 支持 VFP10 浮点处理协处理器;
- (8) 主频最高可达 400 MIPS;

ARM10E 系列微处理器包含 ARM1020E、ARM1022E 和 ARM1026EJ-S 3 种类型。

5. SecurCore 微处理器系列

SecurCore,名称中有安全的含义,所以顾名思义 SecurCore 系列微处理器专为安全需要而设计,提供了完善的 32 位 RISC 技术的安全解决方案。因此,SecurCore 系列微处理器除了具有 ARM 体系结构的低功耗、高性能的特点外,还具有其独特的优势,即提供了对安全解决方案的支持。SecurCore 系列微处理器在银行系统、网上认证系统、网络安全系统等有着广泛的应用。

SecurCore 系列微处理器除了保持着 ARM 体系的特点,还在系统安全方面具有如下的特点:

- (1) 带有灵活的保护单元,以确保操作系统和应用数据的安全;
- (2) 采用软内核技术,防止外部对其进行扫描探测;
- (3) 可集成用户自己的安全特性和其他协处理器。

SecurCore 系列微处理器包含 SecurCore SC100、SecurCore SC110、SecurCore SC200 和 SecurCore SC210 4 种类型。

6. Intel 改进型微处理器系列

Intel 结合 ARM 技术推出了两个系列的产品。

(1) StrongARM 系列

StrongARM 微处理器在掌上电脑领域应用比较广泛,主要是 Intel 公司结合自身的优势和 ARM 处理器的特点,推出的一款基于 ARMv4 架构的处理器系列。StrongARM 系列处理器主要有 SA1110 和 SA1111 两个版本。SA1110 可以满足便携式、嵌入式应用的要求,作为 32 位 StrongARM RISC 处理器,其最高运行速度可达 206 MHz。SA1111 支持 SA1110 开发平台,并可扩展 SA1110 的开发环境。

(2) Xscale 系列处理器

Xscale 系列处理器是基于 ARMv5TE 体系结构的解决方案,是一款全性能、高性价比、低功耗的处理器,也是 Intel 公司根据自身的优势和 ARM 技术的特色推出的一款较高端的处理器。

据 Intel 表示,“Xscale 与 StrongARM 相比,可以大幅降低工作电压并且获得更高的性能”(Ron Smith)。具体来讲,在目前的 StrongARM 中,在 1.5 V 下工作可以获得 166 MHz 的工作频率,在 2.0 V 下工作可以获得 233 MHz 的工作频率;而采用 Xscale 后,则可以在 0.75 V 时使工作频率达到 150 MHz,在 1.0 V 时使工作频率达到 400 MHz,在 1.65 V 下工作频率则可以达到 800 MHz。当在 0.75 V (10 mW) 下以 50 MHz 的工作频率运行时,相当于用一只 5 号电池可以连续工作一个星期。所以可以看到 Xscale 具有更高的性能和良好的节能能力。

3.1.4 ARM 微处理器的寄存器结构

ARM 处理器共有 37 个寄存器,这些寄存器包括:

- (1) 31 个通用寄存器,其中包括程序计数器(PC 指针),均为 32 位的寄存器;
- (2) 6 个状态寄存器,用以标识 CPU 的工作状态及程序的运行状态,均为 32 位,目前只使用了其中的一部分。

同时,ARM 处理器又有 7 种不同的处理器模式,在每一种处理器模式下均有一组相应的寄存器与之对应。即在任意一种处理器模式下,可访问的寄存器包括 15 个通用寄存器(R0~R14)、一至两个状态寄存器和程序计数器。在所有的寄存器中,有些是在 7 种处

理器模式下共用的同一个物理寄存器,而有些寄存器则是在不同的处理器模式下有不同的物理寄存器。

ARM 微处理器支持 7 种运行模式。

(1) 用户模式(user): ARM 处理器正常的程序执行状态。

(2) 快速中断模式(fiq): 用于高速数据传输或通道处理。

(3) 外部中断模式(irq): 用于通用的中断处理。

(4) 管理模式(svc): 操作系统使用的保护模式。

(5) 数据访问终止模式(abt): 当数据或指令预取终止时进入该模式,可用于虚拟存储及存储保护。

(6) 系统模式(sys): 运行具有特权的操作系统任务。

(7) 未定义指令中止模式(und): 当未定义的指令执行时进入该模式,可用于支持硬件协处理器的软件仿真。

具体如表 3-1 所示。

表 3-1 ARM 状态下的寄存器组织

User(sys)32	Fiq32	Supervisor32	Abort32	IRQ32	Undefined32
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
R8	R8_fiq	R8	R8	R8	R8
R9	R9_fiq	R9	R9	R9	R9
R10	R10_fiq	R10	R10	R10	R10
R11	R11_fiq	R11	R11	R11	R11
R12	R12_fiq	R12	R12	R12	R12
R13(SP)	R13_fiq	R13_svc	R13_abt	R13_irq	R13_und
R14(LR)	R14_fiq	R14_svc	R14_abt	R14_irq	R14_und
R15(PC)	R15(PC)	R15(PC)	R15(PC)	R15(PC)	R15(PC)

3.1.5 ARM 微处理器的两种指令集比较

正如前面所介绍的,ARM 微处理器体系结构支持两种指令集:ARM 指令集和

THUMB 指令集。ARM 指令为 32 位的长度,THUMB 指令为 16 位长度。THUMB 指令集为 ARM 指令集的功能子集,但与等价的 ARM 代码相比较,可节省 30%~40%以上的存储空间,同时具备 32 位代码的多数优点。但这并不是说 THUMB 指令就可以代替 ARM 指令,实际上,ARM 指令在一定场合的运行情况会更好。

据有关专家进行测试分析,两种指令集的性能比较如下:当系统数据总线宽度为 32 位时,ARM 比 THUMB 有更好的性能表现;当系统数据总线宽度为 16/8 位时,THUMB 比 ARM 性能更好;同样一段 C 代码,用 THUMB 指令编译的结果约为 ARM 的 65%,同时具有更低的运行功耗。因此,选择 ARM 还是 THUMB 指令,需要从存储器特性和系统性能要求两个方面考虑,而不是单纯地从存储特性一个方面考虑,否则就会丧失两种指令集架构相互融合的优势。

3.2 S3C4510B 芯片

三星的 S3C4510B 是一款 16/32 位的 RISC 处理器,具有低成本、高性能的特点,并且集成了以太网控制器,可以应用于路由器和集线器等网络产品。

这款处理器嵌有 16/32 位的 ARM7TDMI RISC 核。关于 ARM7TDMI RISC 核的介绍可以参照 3.1 小节的内容,它是一个低功耗、应用于特定场合的集成电路。S3C4510B 芯片具有可配置的 8 K 字节的 Cache\SRAM 功能,而且其外围功能也非常强大。如 2 个带缓冲描述符(Buffer Descriptor)的 HDLC 通道、2 个 UART 通道、2 个 GDMA 通道,2 个 32 位的定时器、18 个可编程的 I/O 口等;片内的逻辑控制电路包括:中断控制器、DRAM/SDRAM 控制器、ROM/SRAM 和 FLASH 控制器、系统管理器、一个内部 32 位系统总线仲裁器、一个外部存储器控制器。

3.2.1 工作特性

S3C4510B 的工作特性描述如下(以下参照三星公司关于 S3C4510B 的产品手册)。

(1) 体系结构

- 用于嵌入式以太网应用的集成系统。
- 完全的 16/32 位的 RISC 架构。
- 支持大端、小端模式。内部架构为大端模式,外部存储器只支持小端模式。大端模式是字数据的高字节存储在低地址中,而字数据的低字节则存放在高地址中;小端模式则相反。
- 内含效率高、功能强的 ARM7TDMI 处理器内核。
- 边界扫描接口。

(2) 系统管理器

- 支持 ROM/SRAM、FLASH、DRAM 和外部 I/O 以 8/16/32 位的方式操作。

- 带一个总线请求/应答引脚的外部总线控制器。
- 支持EDO/常规存储器或SDRAM存储器。
- 可编程的访问周期(可设定0~7个等待周期)。
- 4字的写缓冲。
- 高性价比的存储器到外围的DMA接口。

(3) 一体化的指令/数据Cache

- 双向集关联体系结构、一体化的8K Cache。
- 支持LRC(近期最少使用)替换算法。
- Cache可配置为内部SRAM。

(4) IIC接口

- 仅支持主控模式。
- 串行时钟由波特率发生器生成。

(5) Ethernet 控制器

- 带猝发模式的DMA引擎。
- DMA发送/接收缓冲区(256字节发送,256字节接收)。
- MAC发送/接收FIFO缓冲区(80字节发送,16字节接收)。
- 数据对准逻辑。
- 支持端模式变换。
- 10/100 Mbit/s的工作速率。
- 与IEEE802.3标准完全兼容。
- 提供MII和7线制10 Mbit/s接口。
- 站管理信号生成。
- 片内CAM(可达21个目的地址)。
- 带暂停特性的全双工模式。
- 支持长/短包模式。
- 包拆装PDA生成。

(6) HDLC (High-Level Data Link Control) 高层数据链路协议

- HDLC协议特征:标志检测与同步;零插入与删除;空闲检测和发送;FCS生成和检测(16位);终止检测与发送。
- 地址搜索模式(可扩展到4字节)。
- 可选择CRC模式或非CRC模式。
- 用于时钟恢复的数字PLL模块。
- 波特率生成器。
- 发送和接收支持NRZ/NRZI/FM/曼彻斯特数据格式。
- 回环与自动回波模式。
- 8字的发送和接收FIFO。

- 可选的 1 字或 4 字数据传送方式。
- 数据对准逻辑。
- 可编程中断。
- Modem 接口。
- 高达 10 Mbit/s 的工作速率。
- 基于 8 位位组的 HDLC 帧长度。
- 每个 HDLC 有 2 通道 DMA 缓冲描述符用于发送和接收。

(7) DMA 控制器

- 用于存储器到存储器、存储器到 UATR、UATR 到存储器数据传送的 2 通道通用 DMA 控制器,不受 CPU 干预。
- 可由程序或外部 DMA 请求启动。
- 可增减源地址或目的地址,无论 8 位、16 位或 32 位数据传输。
- 4 种数据猝发模式。

(8) UART

- 2 个可工作于 DMA 方式或中断方式的 UART 模块。
- 支持 5、6、7、8 位的串行数据发送和接收。
- 波特率可编程。
- 1 位或 2 位停止位。
- 奇/偶校验。
- 间隔信号的生成与检测。
- 奇偶校验、覆盖和帧错误检测。
- $\times 16$ 时钟模式。
- 支持红外发送和接收。

(9) 定时器

- 2 个可编程 32 位定时器。
- 间隔模式或触发模式工作。

(10) 可编程 I/O 口

- 18 个可编程 I/O 口。
- 可分别配置为输入模式、输出模式或特殊功能模式。

(11) 中断控制器

- 21 个中断源,包括 4 个外部中断源。
- 正常中断或快速中断模式(IRQ、FIQ)。
- 基于优先级的中断处理。

(12) PLL

- 外部时钟可由片内 PLL 倍频以提高系统时钟。
- 输入频率范围:10~40 MHz。

- 输出频率可以是输入时钟的 5 倍。

(13) 工作电压

- 3.3 V, 偏差不超过 5%。

(14) 工作温度

- 0℃~70℃。

(15) 工作频率

- 最高为 50 MHz。

(16) 封装形式

- 208 脚 QFP 封装。

具体的结构如图 3-1 所示。

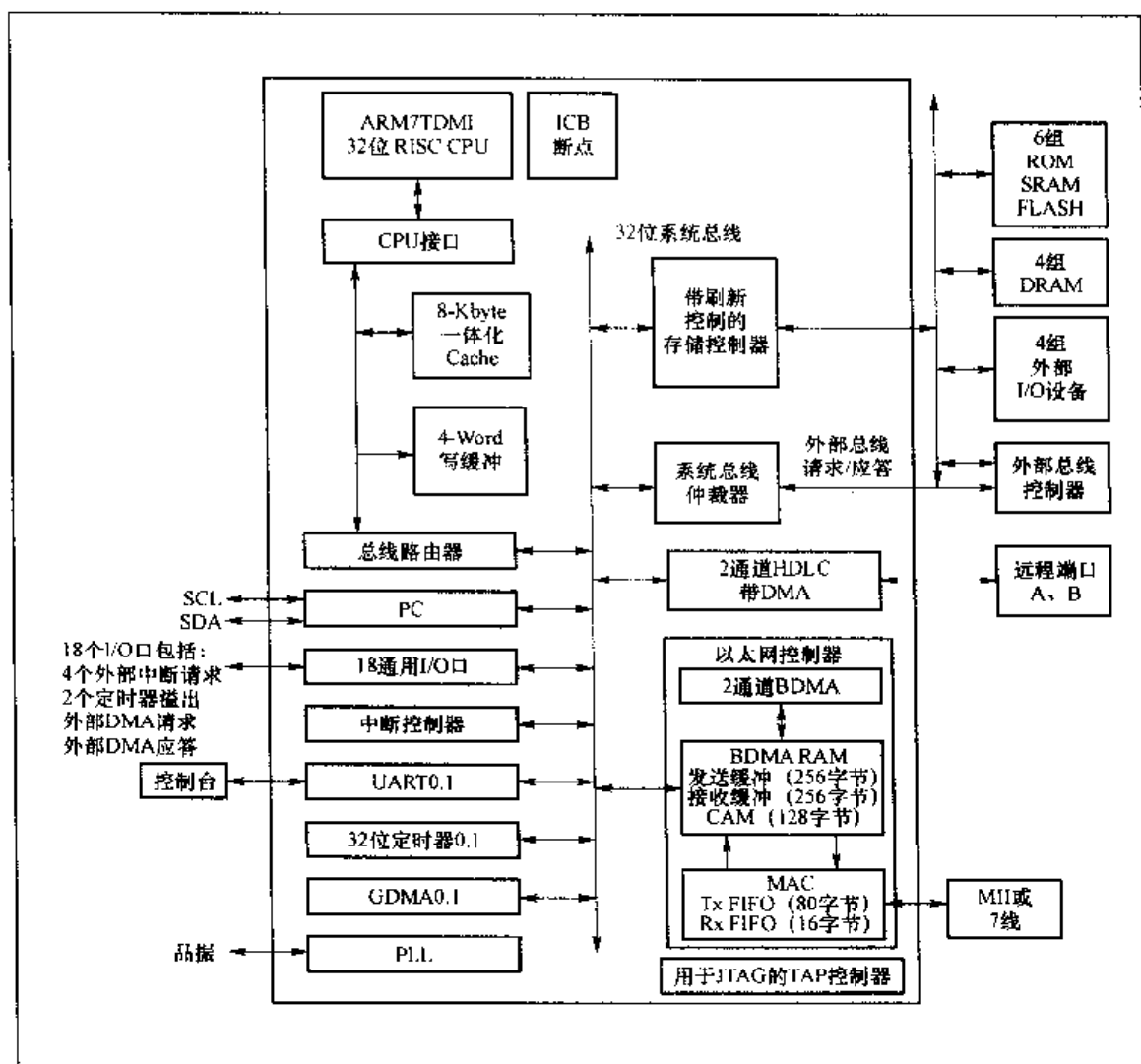


图 3-1 S3C4510B 芯片

3.2.2 芯片引脚介绍

S3C4510B 芯片的引脚分布如图 3-2 所示。

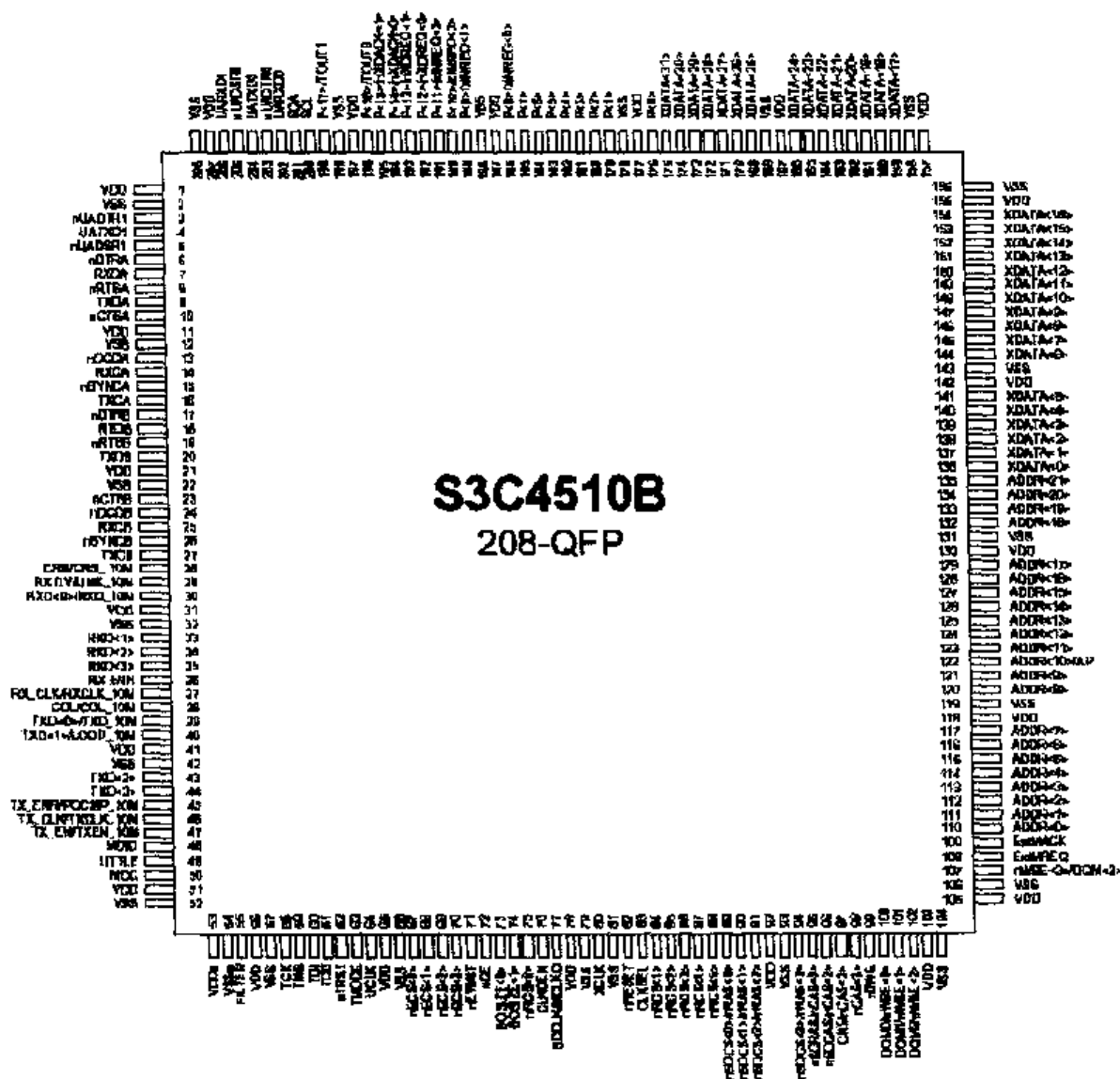


图 3-2 S3C4510B 的引脚分布图

表 3-2 是具体的芯片引脚介绍(参照三星公司的 S3C4510B 芯片手册)。

表 3-2 S3C4510B 芯片引脚介绍

信号	引脚号	类型	描述
XCLK	80	I	S3C4510B 的系统时钟源。如果 CLKSEN 为低电平, 将 PLL 输出时钟作为 S3C4510B 的内部系统时钟。如果 CLKSEN 为高电平, XCLK 直接作为 S3C4510B 的内部系统时钟。
MCLKO/SDCLK	77	O	系统时钟输出。MCLKO 信号可以用来监视内部系统时钟。SDCLK 为 SDRAM 提供时钟信号。
CLKSEL	83	I	时钟选择。如果 CLKSEL 为 0, PLL 输出时钟作为 S3C4510B 的主时钟。如果 CLKSEL 为 1, XCLK 直接作为 S3C4510B 的主时钟。
nRESET	82	I	低有效的复位信号。nRESET 为 S3C4510B 的复位信号, 为配合内部数字滤波, nRESET 必须至少保持 64 个主时钟周期的低电平。
CLKOEN	76	I	时钟输出允许/禁止。高电平允许系统时钟信号输出, 低电平禁止。
TMODE	63	I	测试模式选择。低电平为正常工作模式, 高电平为芯片测试模式。TMODE 也用来改变 PLL 的 MF。
FILTER	55	AI	如果使用 PLL, 应在该引脚和数字地之间接 820 pF 的陶瓷电容。
TCK	58	I	JTAG 测试时钟。JTAG 测试时钟信号用于切换状态信息和检测数据的输入输出。该引脚在片内下拉。
TMS	59	I	JTAG 测试模式选择。该信号控制 S3C4510B 的 JTAG 测试操作。该引脚在片内上拉。
TDI	60	I	JTAG 测试数据输入。在 JTAG 测试操作的过程中, 该信号将指令和数据串行送入 S3C4510B。该引脚在片内上拉。
TDO	61	O	JTAG 测试数据输出。在 JTAG 测试操作的过程中, 该信号将指令和数据串行送出 S3C4510B。
nTRST	62	I	JTAG 复位信号, 低电平复位。异步复位 JTAG 逻辑。该引脚在片内上拉。

续表

信号	引脚号	类型	描述
ADDR[21:0]/ ADDR[10]/AP	117~110 129~120 135~132	O	地址总线。22 位的地址总线可寻址每一个 ROM/ SRAM 组、FLASH 存储器组、DRAM 组和外部 I/O 组的 4 M 字 (64 M 字节) 的地址范围。内部的 23 位地址用来产生 DRAM 地址。
XDATA[31:0]	141~136 154~144 166~159 175~169	I/O	外部数据总线(双向,32 位)。S3C4510B 支持外部 8 位,16 位,32 位的数据宽度。
nRAS[3:0]/ nSDCS[3:0]	94,91, 90,89	O	低有效的 DRAM 行地址锁存信号。S3C4510B 支持最多 4 个 DRAM 组,每个 nRAS 输出控制一组。nSDCS[3:0]用作 SDRAM 的片选信号。
nCAS[3:0] nCAS[0] /nSDRAS nCAS[1] /nSDCAS nCAS[2]/CKE	98,97, 96,95	O	低有效的 DRAM 列地址锁存信号。无论访问哪一个 DRAM 组,4 个 nCAS 输出信号均表示字节选择。nSDRAS 作为 SDRAM 的行地址锁存信号,nSDCAS 作为 SDRAM 的列地址锁存信号,CKE 作为 SDRAM 的时钟使能信号。
nDWE	99	O	DRAM 写使能信号(低有效)。该引脚为 DRAM 组提供写操作信号。(nWBE[3:0]用于为 ROM/SRAM/FLASH 存储器组提供写操作信号。)
nECS[3:0]	70,69, 68,67	O	外部 I/O 片选信号。可以有 4 个外部 I/O 组映射到存储空间,每一个外部 I/O 组的地址范围最大为 16 KB。nECS 提供每一个外部 I/O 组的片选信号。
nEWAIT	71	I	外部等待信号。该信号用于在访问外部 I/O 设备时,由外设插入等待周期。
nRCS[5:0]	88~84,75	O	ROM/SRAM/FLASH 片选信号。S3C4510B 可访问多达 6 个的外部 ROM/SRAM/FLASH 组。
B0SIZE[1:0]	74,73	I	ROM/SRAM/FLASH 存储器组 0 的数据总线宽度设定。ROM/SRAM/FLASH 存储器组 0 常用于程序的启动。01=字节(8 位);10=半字(16 位);11=字(32 位);00=保留。

续表

信号	引脚号	类型	描述
nOE	72	O	输出使能。当对存储器进行访问的时候,该信号控制存储器的输出使能。
nWBE[3:0]/ DQM[3:0]	107, 102~100	O	写字节使能。当对存储器进行写操作时,该信号控制存储器(DRAM除外)的写使能。对于 DRAM 存储器组,由 nCAS [3:0]和 nDWE 控制写操作。DQM 用于 SDRAM 数据输入/输出的屏蔽信号。
ExtMREQ	108	I	外部总线控制器请求信号。外部总线控制器通过该引脚请求控制外部总线,当该信号有效时,S3C4510B 将外部总线置为高阻状态,以便外部总线控制器取得对外部总线的控制。当 ExtMACK 信号为低电平时,S3C4510B 重新取得对外部总线的控制权。
ExtMACK	109	O	外部总线应答信号。
MIXC	50	O	管理数据时钟。该引脚作为 MDIO 数据输入输出时所需的时序参考。
MDIO	48	I/O	管理数据输入/输出。当执行一个读数据的命令时,该引脚输入由物理层产生的数据,当执行一个写数据的命令时,控制层产生的数据由该引脚输出到物理层(PHY)。
LITTLE	49	I	小端模式选择引脚。当该引脚为高电平时,S3C4510B 工作在小端模式,当该引脚为低电平时,工作在大端模式。该引脚在片内已下拉,因此,S3C4510B 缺省工作在大端模式。
COL/COL_10M	38	I	冲突检测/10 M 冲突检测。该引脚显示是否检测到冲突。COL 引脚在 MII 模式时,可以最小延时的异步反映冲突。COL_10 M 可以将 10 M 的物理层检测到的冲突反映出来。
TX_CLK/ TXCLK_10M	46	I	发送时钟/10 M 发送时钟。S3C4510B 在 TX_CLK 的上升沿驱动 TXD[3:0]和 TX_EN,当工作在 MII 模式时,PHY 在 TX_CLK 的上升沿采样 TXD[3:0]和 TX_EN。在发送数据时,TXCLK_10 M 由 10 M 的 PHY 产生。
TXD[3:0] LOOP_10M TXD_10M	44,43, 40,39	O	发送数据/10 M 发送数据/10 M 回环测试。TXD[3:0]为发送数据引脚,TXD_10 M 为 10 M 的 PHY 的发送数据引脚,LOOP_10M 由控制寄存器的回环测试位驱动。
TX_EN/ TXEN_10M	47	O	发送使能/10 M 发送使能。(详细见芯片手册)

续表

信号	引脚号	类型	描述
TX_ERR/ PCOMP_10M	45	O	发送错误/10 M 包压缩使能。(详见芯片手册)
CRS/CRS_10M	28	I	载波侦听/10 M 载波侦听。(详见芯片手册)
RX_CLK/ RXCLK_10M	37	I	接收时钟/10 M 接收时钟。RX_CLK 为连续的时钟信号,当其频率为 25 MHz 时,数据传输速率为 100 M,当其频率为 2.5 MHz 时,数据传输速率为 10 M。在接收数据时,RXCLK_10M 由 10 M 的 PHY 产生。
RXD[3:0] RXD_10M	35,34, 33,30	I	接收数据/10 M 接收数据。(详见芯片手册)
RX_DV/ LINK10M	29	I	接收数据有效/10 M 连接状态。(详见芯片手册)
RX_ERR	36	I	接收错误。
TXDA	9	O	HDLC Ch-A 发送数据。
RXDA	7	I	HDLC Ch-A 接收数据。
nDTRA	6	O	HDLC Ch-A 终端准备就绪。nDTRA 引脚指示数据终端设备准备发送或接收。
nRTSA	8	O	HDLC Ch-A 传送请求。
nCTSA	10	I	HDLC Ch-A 传送清除。
nDCDA	13	I	HDLC Ch-A 数据载波检测。
nSYNCA	15	O	HDLC Ch-A 同步检测。
RXCA	14	I	HDLC Ch-A 接收时钟。
TXCA	16	I/O	HDLC Ch-A 发送时钟。
TXDB	20	O	HDLC Ch-B 发送数据。
RXDB	18	I	HDLC Ch-B 接收数据。
nDTRB	17	O	HDLC Ch-B 终端准备就绪。
nRTSB	19	O	HDLC Ch-B 传送请求。
nCTSB	23	I	HDLC Ch-B 传送清除。
nDCDB	24	I	HDLC Ch-B 数据载波检测。
nSYNCB	26	O	HDLC Ch-B 同步检测。
RXCB	25	I	HDLC Ch-B 接收时钟。
TXCB	27	I/O	HDLC Ch-B 发送时钟。
UCLK	64	I	外部 UART 时钟输入。可由外部输入时钟作为 UART 时钟,通常由系统时钟提供 UART 时钟输入。
UARXD0	202	I	UART0 数据接收。
UATXD0	204	O	UART0 数据发送。

续表

信号	引脚号	类型	描述
nUADTR0	203	I	UART0 数据终端准备就绪。该输入信号通知 S3C4510B, 外设(或其他主机)已准备好发送或接收数据。
nUADSR0	205	O	UART0 数据设备准备就绪。该输出信号通知外设(或其他主机), UART0 已准备好发送或接收数据。
UARXD1	206	I	UART1 数据接收。
UATXD1	4	O	UART1 数据发送。
nUADTR1	3	I	UART1 数据终端准备就绪。参见 nUADTR0。
nUADSR1	5	O	UART1 数据设备准备就绪。参见 nUADSR0。
P[7:0]	185~179, 176	I/O	通用 I/O 口。
XINTREQ[3:0] P[11:8]	191~189, 186	I/O	外部中断请求信号, 或作为通用 I/O 口。
NXDREQ[1:0] /P[13:12]	193, 192	I/O	外部 DAM 请求信号, 或作为通用 I/O 口。
nXDACK[1:0] /P[15:14]	195, 194	I/O	外部 DAM 应答信号, 或作为通用 I/O 口。
TOUT0/P[16]	196	I/O	定时器 0 溢出, 或作为通用 I/O 口。
TOUT1/P[17]	199	I/O	定时器 1 溢出, 或作为通用 I/O 口。
SCL	200	I/O	I ² C 串行时钟。
SDA	201	I/O	I ² C 串行数据。
VDDP	1, 21, 41, 56, 78, 92, 105, 118, 130, 155, 167, 177, 197	Power	I/O 口电源。
VDD1	11, 31, 51, 65, 103, 142, 157, 187, 207	Power	芯片内核电源。
VSSP	2, 22, 42, 57, 79, 81, 93, 106, 119, 131, 156, 168, 178, 198	GND	I/O 口地。
VSS1	12, 32, 52, 66, 104, 143, 158, 188, 208	GND	芯片内核地。
VDDA	53	Power	PLL 模拟电源。
VSSA	54	GND	PLL 模拟地。

注: 有些引脚只介绍了部分功能, 具体的其他功能请参照产品手册。

3.3 ARM/THUMB 指令集及汇编

3.3.1 总体介绍

ARM 微处理器是 RISC 架构的处理器,采用精简指令集。ARM7TMDI(本书主要以 ARM7 为主进行介绍)具有 32 位的 ARM 指令集,也具有 16 位的 THUMB 指令集,具有密集的代码,THUMB 指令和 ARM 指令相互配合,最大功效的发挥作用。

ARM 微处理器的指令集可以分为六大类:

- (1) 跳转指令;
- (2) 数据处理指令;
- (3) 程序状态寄存器(PSR)处理指令;
- (4) 加载/存储指令;
- (5) 协处理器指令;
- (6) 异常指令。

具体的 ARM 指令集及其功能如表 3-3 所示。

表 3-3 ARM 指令集

ADC	带进位加法指令
ADD	加法指令
AND	逻辑与指令
B	跳转指令
BIC	位清零指令
BL	带返回的跳转指令
BLX	带返回和状态切换的跳转指令
BX	带状态切换的跳转指令
CDP	协处理器数据操作指令
CMN	比较反值指令
CMP	比较指令
EOR	异或指令
LDC	存储器到协处理器的数据传输指令
LDM	加载多个寄存器指令
LDR	存储器到寄存器的数据传输指令
MCR	从 ARM 寄存器到协处理器寄存器的数据传输指令

续表

MLA	乘加运算指令
MOV	数据传送指令
MRC	从协处理器寄存器到 ARM 寄存器的数据传输指令
MRS	传送 CPSR 或 SPSR 的内容到通用寄存器指令
MSR	传送通用寄存器到 CPSR 或 SPSR 的指令
MUL	32 位乘法指令
MLA	32 位乘加指令
MVN	数据取反传送指令
ORR	逻辑或指令
RSB	逆向减法指令
RSC	带借位的逆向减法指令
SBC	带借位减法指令
STC	协处理器寄存器写入存储器指令
STM	批量内存字写入指令
STR	寄存器到存储器的数据传输指令
SUB	减法指令
SWI	软件中断指令
SWP	交换指令
TEQ	相等测试指令
TST	位测试指令

3.3.2 ARM 的寻址方式

所谓寻址方式就是处理器根据指令中给出的地址信息来寻找物理地址的方式,目前 ARM 指令系统支持以下几种常见的寻址方式。

1. 立即寻址

立即寻址中的操作数在指令中给出,只要取出指令也就取到了操作数。例如:

```
ADD R0,R0,#5           ;R0←R0+5
ADD R0,R0,#0xFF        ;R0←R0+0xFF
SUB R0,R0,#0x01        ;R0←R0-0x01
```

在以上几条指令中,第二个源操作数即为立即数,要求以“#”为前缀,对于以十六进制表示的立即数,还要求在“#”后加上“0x”。

2. 寄存器寻址

寄存器寻址就是利用寄存器中的数值作为操作数。例如:

ADD R0, R1, R2 ;R0←R1 + R2

SUB R0, R0, R1 ;R0←R0 - R1

第一条指令的执行效果是将寄存器 R1 和 R2 的内容相加,其结果存放在寄存器 R0 中;第二条指令的结果是将其寄存器 R0 的内容与 R1 的内容相减,然后将结果存放在 R0 种。总之要操作的数据放在另外一个寄存器中,而不是直接将操作数据在指令中给出来。

3. 寄存器间接寻址

寄存器间接寻址就是以寄存器中的值作为操作数的地址,而操作数本身存放在内存中,即存储器为存放数据的地方,寄存器充当地址指针。例如:

ADD R1, R1, [R2] ;R0←R1 + [R2]

SUB R1, R2, [R0] ;R1←R2 - [R0]

LDR R0, [R1] ;R0←[R1]

STR R0, [R1] ;[R1]←R0

在第一条指令中,以寄存器 R2 的值作为操作数的地址,在存储器中取得一个操作数后与 R1 相加,结果存入寄存器 R1 中。

第二条指令将寄存器 R0 中的地址作为操作数的地址,然后按照这个地址访问存储器,将存储器中的数据取出来,然后与 R2 相减,将结果存到寄存器 R1 中。

第三条指令以 R1 的值为地址,访问存储器,将其中的数据传送到 R0 中。

第四条指令将 R0 的值传送到以 R1 的值为地址的存储器中。

4. 基址变址寻址

基址变址,顾名思义就是靠一种组合来构成地址,组合中有基地址,有相对的偏移量,根据基地址和偏移量就可以算出要访问的目的地址,这种方式非常灵活,对于访问连续的地址非常方便,形式也非常多,例如:

LDR R0, [R1, #1] ;R0←[R1 + 1]

LDR R0, [R1, #1]! ;R0←[R1 + 1], R1←R1 + 1

LDR R0, [R1], #1 ;R0←[R1], R1←R1 + 1

LDR R0, [R1, R2] ;R0←[R1 + R2]

STR R0, [R1, #1] ;[R1 + 1] ←R0

STR R0, [R1, R2] ;[R1 + R2] ←R0

在第一条指令中,将寄存器 R1 的内容加上 1 形成操作数的有效地址,从而取得操作数存入寄存器 R0 中。

在第二条指令中,将寄存器 R1 的内容加上 1 形成操作数的有效地址,从而取得操作数存入寄存器 R0 中,然后, R1 的内容自增 1 个字节。可以看到上面两条指令的最大不同是第一条指令执行完后,地址指针寄存器中的内容自身并没有变化,而第二条地址指针寄存器中的内容发生了变化,增加了 1。

在第三条指令中,以寄存器 R1 的内容作为操作数的有效地址,从而取得操作数存入

寄存器 R0 中,然后,R1 的内容自增 4 个字节。

在第四条指令中,将寄存器 R1 的内容加上寄存器 R2 的内容形成操作数的有效地址,从而取得操作数存入寄存器 R0 中。可以看出上面两条指令的不同是,第三条指令先将地址指针寄存器中指向的存储器数据传给寄存器 R0,然后再将自己的地址加 1,而第四条寄存器先将自身的地址加上 R2 的数据,然后再操作。两者的地址指针操作过程不一样。

第五、六条指令寻址方式上同前几条指令,只是数据是从寄存器传送到存储器。

5. 多寄存器寻址

采用多寄存器寻址方式,一条指令可以完成多个寄存器值的传送。这种寻址方式可以用一条指令完成传送最多 16 个通用寄存器的值。例如:

```
LDMIA R0!,{R1 - R5}
```

该指令将 R0 指向的存储器中的数据取出来放到 R1 至 R5 中,其中 R0 的内容自动增加,所以可以处理一片连续的内存空间。

```
STMIA R0!,{R3 - R5,R7}
```

将寄存器 R3 到 R5、R7 中的值保存到 R0 指向的存储器中。

6. 相对寻址

相对寻址和基址变址的原理比较相似,只不过相对寻址将当前程序计数器 PC 提供的地址作为基址,在这个基础上将指令中的地址码作为偏移量,然后将两者相加得到了有效地址。接着程序计数器载入有效地址,进行下一步操作。

```
BL TEST2           ;跳转到子程序 TEST2 处执行
```

```
...
```

```
MAIN: LOAD R1,R2
```

```
...
```

```
TEST2: MOV R1,R2    ;子程序
```

7. 堆栈寻址

堆栈是一种有着特定读写顺序的存储结构,支持动态存储区分配,对于被用来存储数据的存储区域,在编译或者汇编时是不知道的。堆栈也经常作为一个线性的数据结构来实现,按先进后出(First In Last Out,FILO)的规则进行存储。使用一个专门的堆栈指针指示当前的存储位置,堆栈指针总是指向栈顶。

当堆栈指针指向最后压入堆栈的数据时,称为满堆栈(Full Stack),而当堆栈指针指向下一个将要放入数据的空位置时,称为空堆栈(Empty Stack)。

同时,根据堆栈的生成方式,又可以分为递增堆栈(Ascending Stack)和递减堆栈(Descending Stack),当堆栈由低地址向高地址生成时,称为递增堆栈,当堆栈由高地址向低地址生成时,称为递减堆栈。这样就有 4 种类型的堆栈工作方式,ARM 微处理器支持这 4 种类型的堆栈工作方式。

- (1) 满递增堆栈:堆栈指针指向最后压入的数据,且由低地址向高地址生成。
- (2) 满递减堆栈:堆栈指针指向最后压入的数据,且由高地址向低地址生成。
- (3) 空递增堆栈:堆栈指针指向下一个将要放入数据的空位置,且由低地址向高地址生成。

(4) 空递减堆栈:堆栈指针指向下一个将要放入数据的空位置,且由高地址向低地址生成。

堆栈操作举例如下:

```
STMFD SP!,{R1 - R5}      ;将 R1~R5 中的数据入栈,满递减堆栈
LDMFD SP!,{R1 - R5}      ;数据出栈,放入 R1~R5 寄存器中,满递减堆栈
```

3.3.3 ARM 指令集

1. 跳转指令

所有的 ARM 处理器支持跳转指令,允许向前或者向后最大 32 MB 地址空间的跳转。另外,由于 PC 为通用的程序计数器,通过向 PC 寄存器中写入一个地址值也可以实现跳转。子程序的跳转有多种方式。

① B : 分支跳转(unconditionally Branch)

指令格式如下:

B{条件}<地址>

B 指令是 CPU 执行这个指令的时候,根据当前的 PC 值和要跳转的偏移量,算出有效值,然后跳转到指向的位置。例如:

```
B LOOP1                ;跳转到 LOOP1 指向的地址空间
BEQ LOOP1               ;相等则跳转到 LOOP1 指向的地址空间
```

② BL : 带连接的分支跳转(Branch with Link)

指令格式如下:

BL{条件}<地址>

BL 是带连接的跳转指令,在跳转之前装入下一条指令的地址到 R14(LR,连接寄存器)中,例如:

```
BL LOOP1                ;跳转到 LOOP1 指向的地址空间
```

③ BX : 指令(Branch with Exchange)

指令格式如下:

BX{条件} 目标地址

BX 指令跳转到指令中所指定的目标地址,目标地址处的指令既可以是 ARM 指令,也可以是 THUMB 指令。在 ARM v4 及 v5 以上架构中,BX(分支和切换)指令将通用寄存器 Rm 的内容拷贝到 PC 中。如果 Rm 位的最低位为 1,则处理器切换到 THUMB 状态,反之则切换到 ARM 状态。例如:

BLX R0 ; 跳转到 R0 指定的地址, 并且根据 R0 的最低位决定程序状态

④ BLX : 带连接和状态切换的分支 (Branch with Link and Exchange)

指令格式如下:

BLX <地址>

BLX 指令跳转到指令中所指定的目标地址, 并将处理器的工作状态由 ARM 状态切换到 THUMB 状态, 该指令同时将 PC 的当前内容保存到寄存器 R14 中。例如:

BLX R0

2. 数据处理指令

数据处理指令分为算术逻辑指令、数据传送指令、比较指令等。算术逻辑指令主要完成数据的算术和逻辑运算, 这些指令将运算结果保存到目的寄存器中, 同时还更新 CPSR 中的相应条件标志位; 比较指令不保存运算结果, 只更新 CPSR 中相应的条件标志位。数据传送指令主要将数据从寄存器和存储器之间传输转移。下面就每一部分进行详细介绍。

(1) 算术和逻辑指令

① ADC : 带进位的加法 (Addition with Carry)

指令格式如下:

ADC{条件}{S}<目的寄存器>, <操作数 1>, <操作数 2>

目的寄存器 = 操作数 1 + 操作数 2 + 进位

(注: 以下统一把目的寄存器以 dest 代替, 操作数 1 以 op1 代替, 操作数 2 以 op2 代替)

ADC 将把两个操作数加起来, 并把结果放置到目的寄存器中。它使用一个进位标志位, 操作的时候可以带进位, 具体的例子有:

ADC R3, R2, R1 ; 将 R2 + R1 + 进位的结果存到寄存器 R3 中

② ADD : 加法 (Addition)

指令格式如下:

ADD{条件}{S}<dest>, <op 1>, <op 2>

dest = op_1 + op_2

ADD 将把两个操作数加起来, 把结果放置到目的寄存器中。操作数 1 是一个寄存器, 操作数 2 可以是一个寄存器、被移位的寄存器或一个立即值, 例如:

ADD R0, R1, R1 ; R0 = R1 + R1 = R1 * 2

ADD R0, R1, #0xFF ; R0 = R1 + 0xFF

ADD R0, R2, R3, LSL#2 ; R0 = R2 + (R3 << 2)

③ AND : 逻辑与 (logical AND)

指令格式如下:

AND{条件}{S}<dest>, <op 1>, <op 2>

dest = op_1 AND op_2

AND 将在两个操作数上进行逻辑与,把结果放置到目的寄存器中。操作数 1 是一个寄存器,操作数 2 可以是一个寄存器、被移位的寄存器或一个立即数,例如:

AND R0, R0, #7 ; R0 = 保持 R0 的低三位,其余各位置 0

AND R0, R1 ; 将 R0 与 R1 相与,然后将结果存入 R0

④ ORR : 逻辑或(logical OR)

指令格式如下:

ORR{条件}{S} <dest>, <op 1>, <op 2>

dest = op_1 OR op_2

ORR 将在两个操作数上进行逻辑或,把结果放置到目的寄存器中;对设置特定的位有用。操作数 1 是一个寄存器,操作数 2 可以是一个寄存器,被移位的寄存器,或一个立即值,例如:

ORR R0, R0, #7 ; 设置 R0 中位 0 和 1,2,其余各位不变

ORR R0, R1 ; 将 R0 与 R1 相或,然后将结果存入 R0

⑤ BIC : 位清除(Bit Clear)

指令格式如下:

BIC{条件}{S} <dest>, <op 1>, <op 2>

dest = op_1 AND (! op_2)

BIC 是在一个字中清除位的一种方法,与 OR 位设置是相反的操作。操作数 2 是一个 32 位掩码(mask)。如果在掩码中设置了某一位,则清除这一位。未设置的掩码位指示此位保持不变。

BIC R0, R0, # %1111 ; 清除 R0 中的位 0、1、2 和 3。保持其余的不变

⑥ EOR : 逻辑异或(logical Exclusive OR)

指令格式如下

EOR{条件}{S} <dest>, <op 1>, <op 2>

dest = op_1 EOR op_2

EOR 将在两个操作数上进行逻辑异或,把结果放置到目的寄存器中;对反转特定的位有用。操作数 1 是一个寄存器,操作数 2 可以是一个寄存器、被移位的寄存器或一个立即值,例如:

EOR R0, R0, #3 ; 反转 R0 中的位 0 和 1

EOR R0, R1 ; 将 R0 与 R1 相异或,然后将结果存入 R0

(2) 传送指令

① MOV : 传送(Move)

指令格式如下:

MOV{条件}{S} <dest>, <op 1>

$\text{dest} = \text{op\_1}$

MOV 从另一个寄存器、被移位的寄存器或一个立即值装载一个值到目的寄存器。可以指定相同的寄存器来实现 NOP 指令的效果,还可以专门移位一个寄存器,例如:

MOV R0, R0 ; R0 = R0, 可以认为是 NOP 指令
MOV R0, R0, LSR#2 ; R0 = R0 / 4, 缩小为原来的 4 倍

如果 R15 是目的寄存器,将修改程序计数器或标志。这用于返回到调用代码,方法是把连接寄存器的内容传送到 R15,例如:

MOV PC, R14 ; 退出到调用者,返回
MOVS PC, R14 ; 退出到调用者并恢复标志位

② MVN : 传送取反的值(Move Negative)

指令格式如下:

MVN{条件}{S}<dest>, <op 1>
 $\text{dest} = ! \text{op\_1}$

MVN 从另一个寄存器、被移位的寄存器或一个立即值装载一个值到目的寄存器。不同之处是在传送之前位被反转了,所以是把一个被取反的值传送到一个寄存器中。取反后的值要按补码来确定它的值,例如:

MVN R0, #4 ; R0 = -5
MVN R0, #0 ; R0 = -1

③ SUB : 减法(Subtraction)

指令格式如下:

SUB{条件}{S}<dest>, <op 1>, <op 2>
 $\text{dest} = \text{op\_1} - \text{op\_2}$

SUB 用操作数 1 减去操作数 2,把结果放置到目的寄存器中。操作数 1 是一个寄存器,操作数 2 可以是一个寄存器、被移位的寄存器或一个立即值,例如:

SUB R0, R1, R3 ; R0 = R1 - R3
SUB R0, R1, #0xFF ; R0 = R1 - 0xFF
SUB R0, R2, R3, LSR#1 ; R0 = R2 - (R3 / 2)

减法可以在有符号和无符号数上进行。

④ SBC : 带借位的减法(Subtraction with Carry)

指令格式如下:

SBC{条件}{S}<dest>, <op 1>, <op 2>
 $\text{dest} = \text{op\_1} - \text{op\_2} - ! \text{carry}$

SBC 做两个操作数的减法,把结果放置到目的寄存器中。它使用进位标志来表示借位,这样就可以实现大于 32 位的减法。SUB 和 SBC 生成进位标志的方式不同于常规,

如果需要借位则清除进位标志。所以,指令要对进位标志进行一个非操作(在指令执行期间自动的反转此位)。

```
SUBS R0, R0, R2
```

```
SUB R1, R1, R3 ;两条指令实现(R0R1) - (R2R3)
```

⑤ RSB : 反向减法(Reverse Subtraction)

指令格式如下:

```
RSB{条件}{S} <dest>, <op 1>, <op 2>
```

```
dest = op_2 - op_1
```

RSB 用操作数 2 减去操作数 1,把结果放置到目的寄存器中。操作数 1 是一个寄存器,操作数 2 可以是一个寄存器、被移位的寄存器或一个立即值,例如:

```
RSB R0, R2, R3 ; R0 = R3 - R2
```

```
RSB R0, R1, #0xFF ; R0 = 0xFF - R1
```

```
RSB R0, R2, R3, LSR#1 ; R0 = (R3 / 2) - R2
```

⑥ RSC : 带借位的反向减法(Reverse Subtraction with Carry)

指令格式如下:

```
RSC{条件}{S} <dest>, <op 1>, <op 2>
```

```
dest = op_2 - op_1 - ! carry
```

同于 SBC,但掉换了两个操作数的前后位置。

(3) 比较指令

① CMP : 比较(Compare)

指令格式如下:

```
CMP{条件}{P} <op 1>, <op 2>
```

```
status = op_1 - op_2
```

CMP 允许把一个寄存器的内容如另一个寄存器的内容或立即值进行比较,更改状态标志来允许进行条件执行。它进行一次减法,但不存储结果,而是正确地更改标志。标志可以反映操作数 1 和操作数 2 比较的结果。例如,若操作数 1 大于操作操作数 2,则此后的有 GT 后缀的指令将可以执行。

② CMN : 比较取负的值(Compare Negative)

指令格式如下:

```
CMN{条件}{P} <op 1>, <op 2>
```

```
status = op_1 - (- op_2)
```

CMN 与 CMP 类似,但它允许与负值(操作数 2 取负值)进行比较,比如 -1(难于用其他方法实现的用于结束列表的数)。这样与 -1 比较将使用,例如:

```
CMN R0, #1 ; 把 R0 与 -1 进行比较
```

③ TST : 测试位 (Test bits)

指令格式如下:

```
TST{条件}{P} <op 1>, <op 2>
      Status = op_1 AND op_2
```

TST 类似于 CMP, 不产生放置到目的寄存器中的结果, 而是在给出的两个操作数上进行操作并把结果反映到状态标志上。使用 TST 来检查是否设置了特定的位。操作数 1 是要测试的数据字而操作数 2 是一个位掩码。例如:

```
TST R0, # %3 ; 测试在 R0 中是否设置了 0,1 位
```

④ TEQ : 测试等价 (Test Equivalence)

指令格式如下:

```
TEQ{条件}{P} <op 1>, <op 2>
      Status = op_1 EOR op_2
```

TEQ 类似于 TST, 区别是这里所使用的运算是 EOR 而不是 AND, 便于比较两个操作数的异同。

3. 移位操作指令

寄存器的移位 ARM 指令集中并不是以单独的指令出现的, 而是作为指令格式中的一个字段。移位的对象既可以是立即数, 也可以是寄存器中的值。

① 逻辑或算术左移 (Logical or Arithmetic Shift Left)

指令格式如下:

```
Rx, LSL #n ; 逻辑左移 n 位
Rx, ASL #n ; 算术左移 N
Rx, LSL Rn ; 按照寄存器 Rn 中的值逻辑左移
Rx, ASL Rn ; 按照寄存器 Rn 中的值算术左移
```

接受 Rx 的内容并按“n”或在寄存器 Rn 中指定的数量向高有效位方向移位。最低有效位用零来填充。除了概念上的第 33 位(就是最后被移出的那一位)之外丢弃移出最左端的高位, 如果逻辑类指令中设置了 S 位, 则此位将成为进位标志的值。

例如:

```
MOV R1, #14
MOV R0, R1, LSL#2
```

在退出时, R0 是 56。例子中先将 14 赋给 R1, 然后将 R1 左移 2 位, 其实就是扩大 4 倍, 然后将 R1 的值传送到 R0 中。

② 逻辑右移 (Logical Shift Right)

指令格式如下:

```
Rx, LSR #n 或者
Rx, LSR Rn
```

它在概念上与左移相对。把所有位向更低有效位方向移动。如果逻辑类指令中设置了 S 位,则把最后被移出最右端的那位放置到进位标志中。例如:

```
MOV    R1, #16
MOV    R0, R1, LSR #2
```

在退出时,R0 是 4。例子中先将 16 赋给 R1,然后将 R1 右移 2 位,其实就是缩小 4 倍,然后将 R1 的值传送到 R0 中。

③ 算术右移 (Arithmetic Shift Right)

指令格式如下:

```
Rx, ASR #n  或者
Rx, ASR Rn
```

类似于 LSR,但是要用被移位的寄存器(Rx)最高位的值来填充高位,用来保护补码表示中的符号位。如果逻辑类指令中设置了 S 位,则把最后被移出最右端的那一位放置到进位标志中。例如:

```
MOV R0, R1, ASR #2           ;将 R1 的内容右移两位后送到 R0 中,左两位用最高位
                               ;的值来填充
```

④ 循环右移 (Rotate Right)

指令格式如下:

```
Rx, ROR #n  或者
Rx, ROR Rn
```

循环右移类似于逻辑右移,但是把从右侧移出去的位放置到左侧,如果逻辑类指令中设置了 S 位,则同时放置到进位标志中,这就是位的“循环”。一个移位量为 32 的操作将导致输出与输入完全一致,因为所有位都被移位了 32 个位置,又回到了开始时的位置。例如:

```
MOV R0, R1, ROR #2           ;将 R1 中的内容循环右移两位后传送到 R0 中
```

⑤ 带扩展的循环右移 (Rotate Right with extend)

指令格式如下:

```
Rx, RRX #n  或者
Rx, RRX Rn
```

这是一个和 ROR 相似的操作。不同之处是,它使用处理器的进位标志来提供一个要被移位的第 33 位的值,也就是进位标志也要参加循环移位。例如:

```
MOV R0, R1, RRX #2           ;将 R1 的内容进行带扩展的循环右移两位后送到 R0
```

4. 乘法指令

① MUL : 乘法 (Multiplication)

指令格式如下:

```
MUL{条件}{S} <dest>, <op 1>, <op 2>
```


$$\text{dest} = \text{op\_1} * \text{op\_2}$$

MLA 将操作数 1 和操作数 2 相乘, 然后再加上第三个操作数, 结果的低 32 位保存到目标寄存器中。例如:

MUL R1, R2, R3 ; 结果 R1 = R2 * R3

② MLA : 带累加的乘法 (Multiplication with Accumulate)

指令格式如下:

MLA{条件}{S} <dest>, <op 1>, <op 2>, <op 3>

$$\text{dest} = (\text{op\_1} * \text{op\_2}) + \text{op\_3}$$

MLA 将操作数 1 和操作数 2 相乘, 然后再加上第三个操作数, 结果的低 32 位保存到目标寄存器中。例如:

MLA R1, R2, R3, R4 ; 结果 R1 = R2 * R3 + R4

③ UMULL : 64 位无符号乘法 (Multiplication)

指令格式如下:

UMULL{条件}{S} <dest1>, <dest2><op 1>, <op 2>

$$\text{dest 1} = \text{op\_1} * \text{op\_2}(\text{低 32 位})$$

$$\text{dest 2} = \text{op\_1} * \text{op\_2}(\text{高 32 位})$$

UMULL 将操作数 1 和操作数 2 相乘, 结果的低 32 位保存到目标寄存器 1 中, 高 32 位保存到目标寄存器 2 中, 例如:

UMULL R1, R2, R3, R4 ; 结果 R1 = R3 * R4(低 32 位)

; 结果 R2 = R3 * R4(高 32 位)

按无符号数对操作数进行相乘。

④ UMLAL : 64 位无符号乘法带累加

指令格式如下:

UMLAL{条件}{S} <dest1>, <dest2><op 1>, <op 2>

$$\text{dest 1} = \text{op\_1} * \text{op\_2} + \text{dest1}(\text{低 32 位})$$

$$\text{dest 2} = \text{op\_1} * \text{op\_2} + \text{dest2}(\text{高 32 位})$$

UMLAL 将操作数 1 和操作数 2 相乘, 然后将结果和 dest1 和 dest2 中的结果进行相加。结果的低 32 位保存到目标寄存器 1 中, 高 32 位保存到目标寄存器 2 中, 例如:

UMLAL R1, R2, R3, R4 ; 结果 R1 = R3 * R4 + R1(低 32 位)

; 结果 R2 = R3 * R4 + R2(高 32 位)

按无符号数对操作数进行相乘。

⑤ SMULL : 64 位有符号乘法 (Multiplication)

指令格式如下:

SMULL{条件}{S} <dest1>, <dest2><op 1>, <op 2>

$$\text{dest 1} = \text{op\_1} * \text{op\_2}(\text{低 32 位})$$

dest 2 = op_1 * op_2(高 32 位)

SMULL 将操作数 1 和操作数 2 相乘,结果的低 32 位保存到目标寄存器 1 中,高 32 位保存到目标寄存器 2 中,例如:

UMULL R1, R2, R3, R4 ;结果 R1 = R3 * R4(低 32 位)

按有符号数对操作数进行相乘。

⑥ SMLAL : 64 位有符号乘法带累加

指令格式如下:

SMLAL{条件}{S} <dest1>, <dest2><op 1>, <op 2>

dest 1 = op_1 * op_2 + dest1(低 32 位)

dest 2 = op_1 * op_2 + dest2(高 32 位)

SMLAL 将操作数 1 和操作数 2 相乘,然后将结果和 dest1 和 dest2 中的结果进行相加。结果的低 32 位保存到目标寄存器 1 中,高 32 位保存到目标寄存器 2 中,例如:

SMLAL R1, R2, R3, R4 ;结果 R1 = R3 * R4 + R1(低 32 位)

;结果 R2 = R3 * R4 + R2(高 32 位)

按有符号数对操作数进行相乘。

5. 程序状态寄存器(PSR)处理指令

ARM 微处理器支持程序状态寄存器访问指令,用于在程序状态寄存器和通用寄存器之间传送数据,程序状态寄存器访问指令包括以下两条:MRS(程序状态寄存器到通用寄存器的数据传送指令)和 MSR(通用寄存器到程序状态寄存器的数据传送指令)。

每个状态寄存器由 4 个 8 位的域构成,能够单独写入,其中 Bits[31:24]表示标志域,Bits[23:16]表示状态域,Bits[15:8]表示扩展域,Bits[7:0]表示控制域。下面具体进行介绍。

① MRS 指令

MRS 指令的格式为:

MRS{条件} 通用寄存器,程序状态寄存器(CPSR 或 SPSR)

MRS 指令用于将程序状态寄存器的内容传送到通用寄存器中。该指令一般用在以下几种情况:

- 当需要改变程序状态寄存器的内容时,可用 MRS 将程序状态寄存器的内容读入通用寄存器,修改后再写回程序状态寄存器。
- 当在异常处理或进程切换时,需要保存程序状态寄存器的值,可先用该指令读出程序状态寄存器的值,然后保存。

指令示例:

MRS R2,CPSR ;传送 CPSR 的内容到 R2

MRS R2,SPSR ; 传送 SPSR 的内容到 R2

② MSR 指令

MSR 指令的格式为:

MSR{条件} 程序状态寄存器(CPSR 或 SPSR)_{域},操作数

MSR 指令用于将操作数的内容传送到程序状态寄存器的特定域中。其中,操作数可以为通用寄存器或立即数。〈域〉用于设置程序状态寄存器中需要操作的位,32 位的程序状态寄存器可分为 4 个域:

- 位[31:24]为条件标志位域,用 f 表示;
- 位[23:16]为状态位域,用 s 表示;
- 位[15:8]为扩展位域,用 x 表示;
- 位[7:0]为控制位域,用 c 表示。

该指令通常用于恢复或改变程序状态寄存器的内容,在使用时,一般要在 MSR 指令中指明将要操作的域。

指令示例:

MSR CPSR,R2 ; 传送 R1 的内容到 CPSR

MSR SPSR,R2 ; 传送 R1 的内容到 SPSR

MSR CPSR_c,R2 ; 传送 R1 的内容到 SPSR,但仅仅修改 CPSR 中的控制位域

6. 加载/存储指令

(1) 传送单一数据

在讲加载/存储指令前,首先介绍一下地址寻址模式的特点,和前面讲的不同,这里重点分析寻址模式的构成:基址寄存器(Base Register)和偏移量(Offset)。

偏移量主要有 3 种形式:立即数偏移量、固定值寄存器偏移量和可变值寄存器偏移量。

立即数偏移量是一个无符号的数,可以加也可以减去基址寄存器的值。立即数偏移量寻址对于访问固定距离的数据元素是非常有用的,例如堆栈偏移、输入输出寄存器。对于字或者无符号字节指令,立即数偏移量是一个 12 位的数,对于半字或者有符号的字节指令,它是一个 8 位数。

固定值寄存器偏移量中的寄存器是一个通用的寄存器,也可以从基址里加或者减,对于访问大批数据块很有用。

可变值寄存器偏移量通过立即数的左移或者右移获得偏移量,仅仅在字或者无符号字节指令中应用。

ARM 系列处理器支持两种类型的从存储器到单个寄存器加载和存储数据的指令:第一种类型可以加载或者存储 32 位的字或者 8 位的无符号字节;第二种类型可以加载或者存储 16 位的无符号半字或者 8 位字节。第二种类型仅仅用于 ARM v4 架构及以下的处理器。

主要的指令有:LDR(字数据加载指令)、LDRB(字节数据加载指令)、LDRH(半字数据加载指令)、STR(字数据存储指令)、STRB(字节数据存储指令)、STRH(半字数据存储指令)。

下面具体进行介绍:

① LDR

LDR 的指令格式为:

LDR{条件} 目的寄存器, <存储器地址>

LDR 是从存储器中的字数据传送到目的寄存器中,然后进行处理。其中目的寄存器不仅可以是通用寄存器,也可以是程序寄存器 PC,当目的寄存器是程序寄存器时,可以用作跳转指令。例如:

```
LDR R0,[R1]           ;将以寄存器 R1 为地址的数据读入到 R0 中
LDR R0,[R1,R2]        ;将 R1 + R2 为地址的数据读入到 R0 中
LDR R0,[R1,R2]!       ;将 R1 + R2 为地址的数据读入到 R0 中,并且将 R1 + R2
                        ;写入到 R1 中
LDR R0,[R1,#4]         ;将 R1 + 4 为地址的数据读入到 R0 中
LDR R0,[R1,#4]!       ;将 R1 + 4 为地址的数据读入到 R0 中,并且将 R1 + 4
                        ;写入到 R1 中
```

② LDRB

LDRB 的指令格式为:

LDRB{条件} 目的寄存器, <存储器地址>

LDRB 指令将存储器中一个 8 位字节的数据传送到目的寄存器中,同时将寄存器的高 24 位清零。其中目的寄存器不仅可以是通用寄存器,也可以是程序寄存器 PC,当目的寄存器是程序寄存器时,可以用作跳转指令。

例如:

```
LDRB R0,[R1]           ;将以寄存器 R1 为地址的数据的低 8 位读入到 R0 中
LDRB R0,[R1,R2]        ;将 R1 + R2 为地址的数据的低 8 位读入到 R0 中
LDRB R0,[R1,R2]!       ;将 R1 + R2 为地址的数据的低 8 位读入到 R0 中,并且将
                        ;R1 + R2 写入到 R1 中
LDRB R0,[R1,#4]         ;将 R1 + 4 为地址的数据的低 8 位读入到 R0 中
LDRB R0,[R1,#4]!       ;将 R1 + 4 为地址的数据的低 8 位读入到 R0 中,并且将
                        ;R1 + 4 写入到 R1 中
```

③ LDRH

LDRH 的指令格式为:

LDRH{条件} 目的寄存器, <存储器地址>

LDRH 指令将存储器中一个 16 位的半字数据传送到目的寄存器中,同时将寄存器

的高16位清零。其中目的寄存器不仅可以是通用寄存器,也可以是程序寄存器PC,当目的寄存器是程序寄存器时,可以用作跳转指令。

例如:

```
LDRH R0,[R1]           ;将以寄存器 R1 为地址的 16 位数据读入到 R0 中
LDRH R0,[R1,R2]        ;将 R1 + R2 为地址的 16 位数据读入到 R0 中
LDRH R0,[R1,R2]!       ;将 R1 + R2 为地址的 16 位数据读入到 R0 中,并且将
                        ;R1 + R2 写入到 R1 中
LDRH R0,[R1,#4]        ;将 R1 + 4 为地址的 16 位数据读入到 R0 中
LDRH R0,[R1,#4]!       ;将 R1 + 4 为地址的 16 位数据读入到 R0 中,并且将
                        ;R1 + 4 写入到 R1 中
```

④ STR

STR 的指令格式为:

STR{条件} 源寄存器,〈存储器地址〉

和 LDR 指令相反,STR 指令用于将源寄存器的 32 位数据值写入到存储器中。

例如:

```
STR R0,[R1,R2]         ;将寄存器中的 32 位数据写入到寄存器 R1 + R2 为地址
                        ;的存储器中
STR R0,[R1],R2         ;将寄存器中的 32 位数据写入到寄存器 R1 为地址
                        ;的存储器中,并且将 R1 + R2 的和值写入到 R1 中
```

⑤ STRB

STRB 的指令格式为:

STRB{条件} 源寄存器,〈存储器地址〉

和 LDRB 指令相反,STRB 指令用于将源寄存器的 8 位数据值写入到存储器中。

例如:

```
STRB R0,[R1,R2]        ;将寄存器中的 8 位数据写入到寄存器 R1 + R2 为地址
                        ;的存储器中
STRB R0,[R1],R2        ;将寄存器中的 8 位数据写入到寄存器 R1 为地址
                        ;的存储器中,并且将 R1 + R2 的和值写入到 R1 中
```

⑥ STRH

STRH 的指令格式为:

STRH{条件} 源寄存器,〈存储器地址〉

和 LDRH 指令相反,STRH 指令用于将源寄存器的 16 位数据值写入到存储器中。

例如:

```
STRH R0,[R1,R2]        ;将寄存器中的 16 位数据写入到寄存器 R1 + R2 为地址
                        ;的存储器中
```

STRH R0,[R1],R2 ;将寄存器中的 16 位数据写入到寄存器 R1 为地址
;的存储器中,并且将 R1 + R2 的和值写入到 R1 中

(2) 批量数据加载/存储指令

使用多数据传送指令(LDM 和 STM)来装载和存储多个字的数据从/到内存。

LDM/STM 的主要用途是把需要保存的寄存器复制到栈上。指令格式是:

xxM{条件}{类型} Rn{!},(寄存器列表){~}

“xx”是 LD 表示装载,或 ST 表示存储。

再加 4 种“类型”就变成了 8 个指令:

栈	其他	说明
LDMED	LDMIB	预先增加装载
LDMFD	LDMIA	过后增加装载
LDMEA	LDMDB	预先减少装载
LDMFA	LDMDA	过后减少装载
STMFA	STMIB	预先增加存储
STMEA	STMIA	过后增加存储
STMFD	STMDB	预先减少存储
STMED	STMDA	过后减少存储

FD、ED、FA、和 EA 指定是满栈还是空栈,是升序栈还是降序栈。一个满栈的栈指针指向上次写的最后一个数据单元,而空栈的栈指针指向第一个空闲单元。一个降序栈是在内存中反向增长(就是说,从应用程序空间结束处开始反向增长),而升序栈在内存中正向增长。

其他形式的 STM/LDM 指令用 IA、IB、DA、DB 简单的描述指令的行为,意思分别是过后增加(Increment After)、预先增加(Increment Before)、过后减少(Decrement After)、预先减少(Decrement Before)。例如:

STMFD R13!, {R0, R4-R12, LR} ;将寄存器列表中的(R0, R4~R12, LR)存入堆栈

LDMFD R13!, {R0, R4 - R12, PC} ;将堆栈内容恢复到寄存器{R0, R4 - R12, PC} 中

7. 数据交换指令

ARM 处理器支持存储器 and 寄存器之间交换数据,主要包括以下两条:

① SWP : 单一数据交换

指令格式如下:

SWP{条件}<目的寄存器>,<操作数 1>,[<操作数 2>]

SWP 指令从操作数 2 所指向的内存装载一个字并把这个字放置到目的寄存器中。把寄存器操作数 1 的内容存储到操作数 2 所指向的地址中。如果目的寄存器和操作数 1 是同一个寄存器,则把寄存器的内容和给定内存位置的内容进行交换。

例如:

SWP R0,R1,[R2]	;将 R2 所指向的存储器中的字数据传送到 R0 中,同
	;时将 R1 中的数据传送到 R2 指向的存储器中
SWP R0,R0,[R1]	;将 R1 所指向的存储器中的字数据传送到 R0 中,
	;同时将 R0 中的数据传送到 R1 指向的存储器中

由指令 2 可以看出如果目的寄存器和寄存器操作数 1 为同一个寄存器时,可以交换寄存器和存储区的数据。

② SWPB: 字节单一数据交换

指令格式如下:

SWPB(条件) <目的寄存器>, <操作数 1>, [<操作数 2>]

SWPB 指令从操作数 2 所指向的内存装载一个字节并把这个字节放置到目的寄存器中。把寄存器操作数 1 的一个字节的内容存储到同一个地址中,如果目的和操作数 1 是同一个寄存器,则把寄存器的内容和给定内存位置的内容进行交换。

例如:

SWPB R0,R1,[R2]	;将 R2 所指向的存储器中的字节数据传送到 R0 中,
	;同时将 R1 中的字节数据传送到 R2 指向的存储器中
SWPB R0,R0,[R1]	;将 R1 所指向的存储器中的字节数据传送到 R0 中,
	;同时将 R0 中的字节数据传送到 R1 指向的存储器中

由指令 2 可以看出如果目的寄存器和寄存器操作数 1 为同一个寄存器时,可以交换寄存器和存储区的一个字节的数据。

8. 协处理器指令

ARM 微处理器可支持多达 16 个协处理器,用于各种协处理操作,在程序执行的过程中,每个协处理器只执行针对自身的协处理指令,忽略 ARM 处理器和其他协处理器的指令。

ARM 指令集提供 3 种类型的指令和协处理器通信。这些指令集允许:ARM 处理器启动协处理器的数据处理操作、ARM 寄存器和协处理器寄存器之间的数据传输 ARM 处理器产生协处理器需要加载或者存储所需的地址。在每个协处理器指令中,可以利用一个 4 位的域区分最多 16 个协处理器,每个协处理器被分配一个专门的号码。

ARM 的协处理器指令主要用于 ARM 处理器初始化 ARM 协处理器的数据处理操作,以及在 ARM 处理器的寄存器和协处理器的寄存器之间传送数据,和在 ARM 协处理器的寄存器和存储器之间传送数据。

协处理器执行同样的指令流,忽略 ARM 指令集和协处理器指令的差别。协处理器能够执行部分指令,并且造成异常。这对于处理运行异常是非常有用的,如除零产生溢出等。但是,部分的执行指令是在协处理器内部,对于 ARM 处理器是不可见的。

ARM 协处理器指令包括以下 5 条:

- CDP 协处理器数操作指令;

- LDC 协处理器数据加载指令；
- STC 协处理器数据存储指令；
- MCR ARM 处理器寄存器到协处理器寄存器的数据传送指令；
- MRC 协处理器寄存器到 ARM 处理器寄存器的数据传送指令。

① CDP 指令

CDP 指令的格式为：

CDP{条件} 协处理器编码,协处理器操作码 1,目的寄存器,源寄存器 1,源寄存器 2,
协处理器操作码 2

CDP 指令用于 ARM 处理器通知 ARM 协处理器执行特定的操作,若协处理器不能成功完成特定的操作,则产生未定义指令异常。其中协处理器操作码 1 和协处理器操作码 2 为协处理器将要执行的操作,目的寄存器和源寄存器均为协处理器的寄存器,指令不涉及 ARM 处理器的寄存器和存储器。

指令示例：

CDP P3,1,C12,C10,C3,1 ;该指令完成协处理器 P3 的初始化

② LDC 指令

LDC 指令的格式为：

LDC{条件}{L} 协处理器编码,目的寄存器,[源寄存器]

LDC 指令用于将源寄存器所指向的存储器中的字数据传送到目的寄存器中,若协处理器不能成功完成传送操作,则产生未定义指令异常。其中,{L}选项表示指令为长读取操作,如用于双精度数据的传输。

指令示例：

LDC P1,C2,[R1] ;将 ARM 处理器的寄存器 R1 所指向的存储器中的
;字数据传送到协处理器 P1 的寄存器 C2 中

③ STC 指令

STC 指令的格式为：

STC{条件}{L} 协处理器编码,源寄存器,[目的寄存器]

STC 指令用于将源寄存器中的字数据传送到目的寄存器所指向的存储器中,若协处理器不能成功完成传送操作,则产生未定义指令异常。其中,{L}选项表示指令为长读取操作,如用于双精度数据的传输。

指令示例：

STC P1,C2,[R1] ;将协处理器 P1 的寄存器 C2 中的字数据
;传送到 ARM 处理器的寄存器 R1 所指向的存储器中

④ MCR 指令

MCR 指令的格式为：

MCR{条件}协处理器编码,协处理器操作码 1,源寄存器,目的寄存器 1,目的寄存器 2,协处理器操作码 2

MCR 指令用于将 ARM 处理器寄存器中的数据传送到协处理器寄存器中,若协处理器不能成功完成操作,则产生未定义指令异常。其中协处理器操作码 1 和协处理器操作码 2 为协处理器将要执行的操作,源寄存器为 ARM 处理器的寄存器,目的寄存器 1 和目的寄存器 2 均为协处理器的寄存器。

指令示例:

MCR P1,3,R1,C2,C3,6 ;该指令将 ARM 处理器寄存器 R1 中的数据传
;送到协处理器 P1 的寄存器 C2 和 C3 中

⑤ MRC 指令

MRC 指令的格式为:

MRC{条件}协处理器编码,协处理器操作码 1,目的寄存器,源寄存器 1,源寄存器 2,协处理器操作码 2

MRC 指令用于将协处理器寄存器中的数据传送到 ARM 处理器寄存器中,若协处理器不能成功完成操作,则产生未定义指令异常。其中协处理器操作码 1 和协处理器操作码 2 为协处理器将要执行的操作,目的寄存器为 ARM 处理器的寄存器,源寄存器 1 和源寄存器 2 均为协处理器的寄存器。

指令示例:

MRC P1,3,R1,C1,C2,6 ;该指令将协处理器 P1 的寄存器中的数据
;传送到 ARM 处理器寄存器

9. 异常指令

ARM 微处理器所支持的异常指令有如下两条:

- SWI 软件中断指令;
- BKPT 断点中断指令。

① SWI 指令

SWI 指令的格式为:

SWI{条件} 24 位的立即数

SWI 指令用于产生软件中断,以便用户程序能调用操作系统的系统例程。操作系统在 SWI 的异常处理程序中提供相应的系统服务,指令中 24 位的立即数指定用户程序调用系统例程的类型,相关参数通过通用寄存器传递,当指令中 24 位的立即数被忽略时,用户程序调用系统例程的类型由通用寄存器 R0 的内容决定,同时,参数通过其他通用寄存器传递。

指令示例:

SWI 0x05 ;该指令调用操作系统编号位 05 的系统例程

② BKPT 指令

BKPT 指令的格式为:

BKPT 16 位的立即数

BKPT 指令产生软件断点中断,可用于程序的调试。

3.3.4 THUMB 指令及应用

为兼容数据总线宽度为 16 位的应用系统,ARM 体系结构除了支持执行效率很高的 32 位 ARM 指令集以外,同时支持 16 位的 THUMB 指令集。THUMB 指令集是 ARM 指令集的一个子集,允许指令编码为 16 位的长度。与等价的 32 位代码相比较,THUMB 指令集在保留 32 位代码优势的同时,大大的节省了系统的存储空间。

所有的 THUMB 指令都有对应的 ARM 指令,而且 THUMB 的编程模型也对应于 ARM 的编程模型,在应用程序的编写过程中,只要遵循一定调用的规则,THUMB 子程序和 ARM 子程序就可以互相调用。当处理器在执行 ARM 程序段时,称 ARM 处理器处于 ARM 工作状态,当处理器在执行 THUMB 程序段时,称 ARM 处理器处于 THUMB 工作状态。

与 ARM 指令集相比较,THUMB 指令集中的数据处理指令的操作数仍然是 32 位,指令地址也为 32 位,但 THUMB 指令集为实现 16 位的指令长度,舍弃了 ARM 指令集的一些特性,如大多数的 THUMB 指令是无条件执行的,而几乎所有的 ARM 指令都是有条件执行的;大多数的 THUMB 数据处理指令的目的寄存器与其中一个源寄存器相同。

由于 THUMB 指令的长度为 16 位,即只用 ARM 指令一半的位数来实现同样的功能,所以,要实现特定的程序功能,所需的 THUMB 指令的条数较 ARM 指令多。在一般的情况下,THUMB 指令与 ARM 指令的时间效率和空间效率关系为:

- THUMB 代码所需的存储空间约为 ARM 代码的 60%~70%;
- THUMB 代码使用的指令数比 ARM 代码多约 30%~40%;
- 若使用 32 位的存储器,ARM 代码比 THUMB 代码快约 40%;
- 若使用 16 位的存储器,THUMB 代码比 ARM 代码快约 40%~50%;
- 与 ARM 代码相比较,使用 THUMB 代码,存储器的功耗会降低约 30%。

由上面的分析可以看出,两种指令适应于不同场合和需求,必须适当加与区分,选择最适合的方式去设计程序。

3.3.5 ARM 汇编程序设计

如果读者有单片机的设计基础,那么了解 ARM 汇编语言就相对容易多了,它跟 51 单片机或者 x86 处理器一样,有专门的 ARM 汇编调试开发软件或者开发环境,在 Tornado 或者 ADS 里,都是可以调试底层的汇编语言。本节主要介绍 ARM 汇编的伪指

令,语法格式和程序结构。

1. ARM 汇编器所支持的伪指令

在多数汇编语言里,如 51 汇编、8086 汇编等,都有不同的助记符。当然,助记符并不是在运行的时候执行,而是编译器在运行之前就进行处理了。在 ARM 汇编语言程序里,有一些特殊指令助记符,这些助记符与指令系统的助记符不同,没有相对应的操作码,通常称这些特殊指令助记符为伪指令,它们所完成的操作称为伪操作。伪指令在源程序中的作用是为完成汇编程序作各种准备工作,这些伪指令仅在汇编过程中起作用,一旦汇编结束,伪指令的使命就完成了。

在 ARM 的汇编程序中,有如下几种伪指令:符号定义伪指令、数据定义伪指令、汇编控制伪指令、宏指令以及其他伪指令。

(1) 符号定义(Symbol Definition)伪指令

符号定义伪指令用于定义 ARM 汇编程序中的变量、对变量赋值以及定义寄存器的别名等操作。符号定义对于汇编程序非常有好处,可以使程序的可读性、可维护性增强,如果需要改某个变量,则只需要修改伪指令的值就可以了,而不用修改程序的每个地方。常见的符号定义伪指令有如下几种:用于定义全局变量的伪指令,如 GBLA、GBLL 和 GBLS;用于定义局部变量的伪指令,如 LCLA、LCLL 和 LCLS;用于对变量赋值的伪指令,如 SETA、SETL、SETS 等;为通用寄存器列表定义名称的伪指令,如 RLIST;以及为协处理器命名的 CP 等其他伪指令。下面分别进行介绍。

① 定义全局变量的符号伪指令(GBLA、GBLL 和 GBLS)

GBLA、GBLL 和 GBLS 伪指令用于定义一个 ARM 程序中的全局变量,并将其初始化。

语法格式:

GBLA(GBLL、GBLS) Global Variant(全局变量)

GBLA 伪指令用于定义一个全局的数字变量,并初始化为 0;GBLL 伪指令用于定义一个全局的逻辑变量,并初始化为 F(假);GBLS 伪指令用于定义一个全局的字符串变量,并初始化为空。由于以上 3 条伪指令用于定义全局变量,因此在整个程序范围内变量名必须唯一。如果不唯一,则编译器在编译的时候就会提示错误。

3 种类型举例如下:

GBLA TMP	; 定义一个全局的数字变量,变量名为 TMP,并且初始化为 0
GBLL TMP	; 定义一个全局的逻辑变量,变量名为 TMP,并初始化为 F(假)
GBLS TMP	; 定义一个全局的字符串变量,变量名为 TMP,并初始化为空

② 定义局部变量的伪指令(LCLA、LCLL 和 LCLS)

LCLA、LCLL 和 LCLS 伪指令用于定义一个 ARM 程序中的局部变量,并将其初始化,跟定义全局变量的伪指令非常相似。

语法格式如下:

LCLA(LCLL,LCLS) Local Variant(局部变量名)

其中 LCLA 伪指令用于定义一个局部的数字变量,并初始化为 0;LCLL 伪指令用于定义一个局部的逻辑变量,并初始化为 F(假);LCLS 伪指令用于定义一个局部的字符串变量,并初始化为空。以上 3 条伪指令用于声明局部变量,在其作用范围内变量名必须唯一。如果不唯一,则编译器在编译的时候就会提示错误。

3 种类型举例如下:

LCLA TMP ;定义一个局部的数字变量,变量名为 TMP,并且初始化为 0

LCLL TMP ;定义一个局部的逻辑变量,变量名为 TMP,并初始化为 F(假)

LCLS TMP ;定义一个局部的字符串变量,变量名为 TMP,并初始化为空

③ 对变量赋值的伪指令(SETA、SETL 和 SETS)

伪指令 SETA、SETL、SETS 用于给一个已经定义的全局变量或局部变量赋值,跟全局变量和局部变量里的用法非常相似。

语法格式如下:

变量名 SETA(SETL 或 SETS) 表达式

SETA 伪指令用于给一个数值变量赋值;SETL 伪指令用于给一个逻辑变量赋值;SETS 伪指令用于给一个字符串变量赋值;其中变量名为已经定义过的全局变量或局部变量,表达式为将要赋给变量的值。

使用示例:

TMP SETA 0xFF ;将该变量赋值为 0xFF

TMP SETL {TRUE} ;将该变量赋值为真

TMP SETS "Hello!" ;将该变量赋值为"Hello!"

④ 其他伪指令

- 对通用寄存器列表命名伪指令 RLIST

RLIST 语法格式如下:

Label RLIST {reglist}

其中 reglist 为通用寄存器列表,RLIST 伪指令可用于对一个通用寄存器列表定义名称。

使用示例:

RegListTest RLIST {R0 - R10} ;将寄存器列表名称定义为 RegListTest

- 对协处理器命名伪指令 CP

CP 伪指令语法格式如下:

Name CP Number

Name 为要定义的协处理器名称,Number 为协处理器的编号,数值范围为 0~15。

本语法为一个协处理器定义名称。举例如下：

CoProcessor CP 1

将1号协处理器取名为 CoProcessor。

- 对协处理器的寄存器命名伪指令 CN

CN 伪指令语法格式如下：

Name CN Number

Name 为协处理器的寄存器定义名称；Number 为协处理器的寄存器编号，数值范围为 0~15。

本语法为一个协处理器的寄存器定义名称。举例如下：

CoProcessorReg1 CN 1

将协处理器的寄存器 1 取名为 CoProcessorReg1。

- 对浮点处理器命名伪指令 FN

FN 伪指令语法格式如下：

Name FN Number

Name 为要定义的浮点处理器名称；Number 为浮点协处理器的编号，数值范围为 0~7。

本语法为一个浮点处理器定义名称。举例如下：

FProcessor FN 1

将1号浮点处理器取名为 FProcessor。

(注：还有一些不经常使用的伪指令不再列举。)

(2) 数据定义(Data Definition)伪指令

数据定义伪指令用于数据空间分配、定义数据表、同时完成初始化等。常见的数据定义伪指令有如下几种。

- SPACE 用于分配一块内存空间(连续)。
- MAP 用于定义一个结构化的内存表首地址。
- FIELD 用于定义一个结构化的内存表的数据域。
- DCB 用于分配一段以字节为单位的连续内存空间，并用指定的数据初始化。
- DCW 用于分配一段以半字为单位的连续内存单元，并用指定的数据初始化。DCWU和它用法相同。
- DCD 用于分配一段以字为单位的连续内存空间，并用指定的数据初始化。DCDU和它用法相同。
- DCFD 用于为双精度的浮点数分配一段以字为单位的连续内存空间，并用指定的浮点数据初始化。DCFUDU和它用法相同。
- DCFS 用于为单精度的浮点数分配一段以字为单位的连续内存空间，并用指定的浮点数据初始化。DCFSU和它用法相同。
- DCQ 用于分配一段以8字节为单位的连续内存单元，并用指定的数据初始化。

DCQU 和它用法相同。

下面具体进行介绍。

① SPACE 伪指令

语法格式如下:

{Label} SPACE(%) Expression

SPACE 伪指令用于分配一片连续的、字节数为 Expression 的存储区域并初始化为 0。
SPACE 等价于“%”。

举例如下:

Test1 SPACE 200 ;分配 200 字节的连续的内存空间,并初始化为 0

Test1 % 200 ;分配 200 字节的连续的内存空间,并初始化为 0

② MAP 伪指令

语法格式如下:

{MAP}(^) 表达式[,基址寄存器]

MAP 伪指令用于定义一个结构化的内存表的首地址。MAP 等价于“~”。

表达式可以为程序中的标号或数学表达式,基址寄存器为可选项,当基址寄存器选项不存在时,表达式的值即为内存表的首地址,当该选项存在时,内存表的首地址为表达式的值与基址寄存器的和。

使用示例:

MAP 0x300,R1 ;定义结构化内存表首地址的值为 0x300 + R1

~ 0x300,R1 ;定义结构化内存表首地址的值为 0x300 + R1

③ FIELD 伪指令

语法格式:

{Label} FIELD(#) Expression

FIELD 伪指令用于定义一个结构化内存表中的数据域。FIELD 等价于“#”。

表达式的值为当前数据域在内存表中所占的字节数。FIELD 伪指令常与 MAP 伪指令配合使用来定义结构化的内存表。MAP 伪指令定义内存表的首地址,FIELD 伪指令定义内存表中的各个数据域,并可以为每个数据域指定一个标号供其他的指令引用。

使用示例:

MAP 0x4000 ;定义结构化内存表首地址的值为 0x4000

Area1 FIELD 16 ;定义 Area1 的长度为 16 字节,位置为 0x4010

MAP 0x5000 ;定义结构化内存表首地址的值为 0x5000

Area4 # 16 ;定义 Area4 的长度为 16 字节,位置为 0x5010

④ DCB 伪指令

语法格式如下:

Label DCB (=) Expression {,Expression}{,Expression}...

(DCB)伪指令用于分配一段以字节为单位的内存单元,并用伪指令中指定的表达式初始化。一般 Expression 为数字或字符串,DCB 等同于“=”。

使用示例:

```
AreaByte DCB 180           ;定义一个字节数据,并初始化为 180
AreaByte = 200             ;定义一个字节数据,并初始化为 200
```

⑤ DCW(DCWU)伪指令

语法格式:

```
{Label} DCW(DCWU) Expression{,Expression}{,Expression}...
```

DCW(DCWU)伪指令用于分配一段以半字为单位的内存空间,并用伪指令中指定的 Expression 初始化。其中,Expression 一般为数字表达式。用 DCW 分配的字存储单元是半字对齐的,而用 DCWU 分配的半字内存空间不需要半字对齐。要注意两者的区别。

使用示例:

```
DCW 1,3,44                ;分配一片连续的半字存储单元并初始化为 1,3,44
```

⑥ DCD(或 DCDU)伪指令

语法格式:

```
{Label} DCD(或 DCDU)Expression{,Expression}{,Expression}...
```

DCD(或 DCDU)伪指令用于分配一段以字为单位的内存空间,并用伪指令中指定的 Expression 初始化。其中,Expression 可以为程序标号或数字表达式。DCD 等同于“&”。用 DCD 分配的字存储单元是字对齐的,而用 DCDU 分配的字内存单元不需要字对齐。要注意两者的区别。

使用示例:

```
ResetTest DCD Reset       ;Reset 是程序中的一个标号,相当于一个 32 位二进制数
ResetTest DCDU Reset
ResetTest & Reset
```

⑦ DCQ(或 DCQU)伪指令

语法格式:

```
{Label} DCQ(或 DCQU)Expression{,Expression}{,Expression}...
```

DCQ(或 DCQU)伪指令用于分配一段以 8 字节为单位的连续内存空间,并用伪指令中指定的表达式(8 字节)初始化。用 DCQ 分配的存储单元是字对齐的,而用 DCQU 分配的存储单元并不严格字对齐。

使用示例:

```
DataTest DCQ 33            ;分配一段连续的字内存空间,并初始化为指定的值 33
DCQU 1222,3333             ;分配一段连续的字内存空间,并初始化为指定的值
                           ;1222,3333
```

⑧ DCFD(或 DCFDU)伪指令

语法格式:

```
{Label} DCFD(或 DCFDU)Expression{,Expression}{,Expression}...
```

DCFD(或 DCFDU)伪指令用于分配一段以双字为单位的连续内存空间,并用伪指令中指定的表达式(双精度浮点)初始化。用 DCFD 分配的字存储单元是字对齐的,而用 DCFDU 分配的字存储单元并不需要字对齐。要注意两者的区别。

使用示例:

```
FTest DCFD 4E6 ;定义一个 64 位的双精度浮点数,并初始化为 4E6
```

```
FTest DCFDU 4E6 ;定义一个 64 位的双精度浮点数,并初始化为 4E6
```

⑨ DCFS(或 DCFSU)

语法格式:

```
{Label} DCFS(或 DCFSU)Expression{,Expression}{,Expression}...
```

DCFS(或 DCFSU)伪指令用于分配一段以字为单位的连续内存空间,并用伪指令中指定的表达式(单精度)初始化。用 DCFS 分配的字内存空间是字对齐的,而用 DCFSU 分配的字内存空间并不需要字对齐。要注意两者的区别。

使用示例:

```
FDataTest DCFS 4E6 ;定义一个 32 位的单精度浮点数,并初始化为 4E6
```

```
FDataTest DCFSU 4E6 ;定义一个 32 位的单精度浮点数,并初始化为 4E6
```

(3) 汇编控制伪指令

汇编控制伪指令用于条件汇编、宏定义、重复汇编控制等,常用的汇编控制伪指令包括以下几条:条件类(如 IF、ELSE、ENDIF)、循环类(如 WHILE、WEND)、宏定义类(如 MACRO、MEND)、退出类(如 MEXIT)。

① 条件类

语法格式如下:

```
IF {逻辑代码}
```

```
指令代码 1
```

```
ELSE
```

```
指令代码 2
```

```
ENDIF
```

有 C 语言基础的读者可能立刻感觉到,这跟标准 C 里的用法几乎一样。若 IF 里的逻辑代码成立,执行指令代码 1,否则不成立则执行代码 2。跟标准 C 一样,可以嵌套使用。

举例如下:

```
...
```

```
IF R0
```



```

        B CODE1
ELSE
        B CODE2
ENDIF
...
CODE1;
        代码 1
CODE2;
        代码 2

```

如果 R0 中的值为 0, 则跳出判断执行 CODE2, 否则为真则执行 CODE1。

② 循环类

语法格式如下:

```

WHILE 逻辑表达式
指令代码
WEND

```

WHILE、WEND 伪指令也跟标准 C 里的 while 用法一样, 首先判断 WHILE 后面的逻辑代码是否成立, 如果成立则执行下面的指令代码, 执行完以后再判断逻辑代码, 直到逻辑代码不成立, 则跳出循环, 执行循环外面的代码指令。WHILE、WEND 伪指令也可以嵌套使用。

举例如下:

```

...
WHILE Test < 9
Test = Test + 2
代码指令
WEND

```

如果变量 Test 的值小于 9, 则执行下面的代码指令, 否则一直加 2, 一直到 WHILE 后面的逻辑不成立, 跳出循环。

③ 宏定义类

语法格式如下:

```

MACRO
$Label 宏定义的名称 { $parameter1 } { $parameter2 } ...
宏表达式或者指令
MEND

```

跟标准 C 里的宏定义用法相似, MACRO、MEND 伪指令可以将一段代码定义为一

个整体,称为宏指令,然后就可以在程序中通过宏指令多次调用该段代码。其中,\$ 标号在宏指令被展开时,标号会被替换为用户定义的符号,宏指令可以使用一个或多个参数,当宏指令被展开时,这些参数被相应的值替换。

包含在 MACRO 和 MEND 之间的指令序列称为宏定义体,在宏定义体的第一行应声明宏的原型(包含宏名、所需的参数),然后就可以在汇编程序中通过宏名来调用该指令序列。在源程序被编译时,汇编器将宏调用展开,用宏定义中的指令序列代替程序中的宏调用,并将实际参数的值传递给宏定义中的形式参数。MACRO、MEND 伪指令可以嵌套使用。

举例如下:

```
MACRO
$HandlerLabel HANDLER $HandlerLabel
$HandlerLabel
sub      sp,sp,#4           ;减少 sp
stmfd    sp!,{r0}          ;将工作寄存器的内容压入堆栈
ldr      r0,=$HandlerLabel ;将 HandleXXX 的地址装载给寄存器 r0
ldr      r0,[r0]            ;装载 r0 中的内容
str      r0,[sp,#4]         ;给堆栈里存储 HandleXXX 的内容
ldmfd    sp!,{r0,pc}       ;将堆栈的值恢复给工作寄存器和 PC
MEND
```

④ 退出类

语法格式如下:

MEXIT

MEXIT 用于从宏定义中退出来。

(4) 其他常用的伪指令

还有一些伪指令如段定义伪指令、引入声明、ARM 还有 THUMB 代码等,主要包括以下几条:边界对齐(ALIGN)、段定义(AREA) 指令集类型定义(CODE16/ CODE32)、程序入口(ENTRY)、结束声明(END)、常量定义(EQU)、声明符号可以被其他文件引用(EXPORT/GLOBAL)、声明外部符号(IMPORT/ EXTERN)、包含文件(GET/INCLUDE)、包含不被汇编的文件(INCBIN)、给特定的寄存器命名(RN)、标记局部标号使用范围的界限(ROUT)。

① 边界对齐伪指令:ALIGN

指令语法如下:

ALIGN {Expression{,offset}}

ALIGN 伪指令满足一定的对齐方式。其中,表达式的值用于指定对齐方式,可能的

取值为2的幂,如1、2、4、8、16等。默认为字对齐。偏移量为一个数字表达式,若使用该字段,则当前位置的对齐方式为:2的表达式次幂+偏移量。

在ARM代码中,要求地址标号为字对齐,而在THUMB中要求代码是半字对齐。可以使用ALIGN 4和ALIGN 2来调整各自的对齐格式。举例如下:

```
AREA TEST_CODE, CODE, READWRITE, ALIGN = 2      ;声明代码段,指定后面的
                                                    ;指令为4字节对齐,可读写
AREA TEST_DATA, DATA, READONLY, ALIGN = 3      ;声明数据段,指定后面的
                                                    ;指令为8字节对齐,只读
```

② 段定义伪指令:AREA

指令语法如下:

AREA 段名称 AREA 段名 属性1,属性2,属性3等

AREA伪指令用于定义一个代码段或数据段。其中,段名若以数字开头,则该段名需用“|”括起来,如|1-CODE|。相关属性用逗号分隔,具体介绍如下。

- ALIGN属性:具体用法上面已经介绍,主要为对齐方式的设置。在默认情况下,ELF(可执行连接文件)的代码段和数据段是按字对齐的,并且取值范围为0~31。如ALIGN=2,表示4字节对齐,可以参照上面的例子。对齐的格式为2的幂次字节。
- CODE属性:用于定义代码段,默认为READONLY。
- COMMON属性:该属性定义一个通用的段,不包含任何的用户代码和数据,初始化为零。各源文件中同名的COMMON段共享同一个内存空间。
- DATA属性:用于定义数据段,默认为READWRITE。
- READONLY属性:指定本段为只读,代码段默认为READONLY。
- READWRITE属性:指定本段为可读可写,数据段的默认属性为READWRITE。

ARM汇编中,需要指定代码段和数据段的属性,分配好空间。举例如下:

```
AREA TEST_CODE, CODE, READWRITE, ALIGN = 2      ;声明代码段,指定后面的
                                                    ;指令为4字节对齐,可读写
AREA TEST_DATA, DATA, READONLY, ALIGN = 3      ;声明数据段,指定后面的
                                                    ;指令为8字节对齐,只读
```

③ 指令集类型定义伪指令:CODE16、CODE32

指令语法如下:

CODE16

THUMB 指令

CODE32

ARM 指令

可用 CODE16 伪指令通知编译器,其后的指令序列为 16 位的 THUMB 指令,用 CODE32 伪指令通知编译器,其后的指令序列为 32 位的 ARM 指令。但他们只通知编译器后面指令的类型,并不能对处理器进行状态的切换。如果要切换可以使用 BX 指令进行切换。举例如下:

```
AREA TEST_CODE, CODE, READWRITE, ALIGN = 2
```

```
CODE16
```

```
具体代码 1
```

```
AREA TEST_DATA, DATA, READONLY, ALIGN = 3 ;
```

```
CODE32
```

```
具体代码 2
```

④ 指定入口点伪指令: ENTRY

ENTRY 伪指令用于指定汇编程序的入口点。

指令语法如下:

```
ENTRY
```

在一个完整的汇编程序中至少要有一个 ENTRY,但在一个源文件里最多只能有一个 ENTRY,也可以没有入口点。

举例如下:

```
AREA TEST_CODE, CODE, READWRITE, ALIGN = 2 ; 定义代码段
```

```
ENTRY ; 指定应用程序的入口点
```

```
CODE16
```

```
主程序代码
```

⑤ 程序结束伪指令: END

指令语法如下:

源文件代码正文

```
END
```

END 伪指令表示源文件的程序代码结束,每个源文件必须包含一个结束伪指令。

举例如下:

```
AREA TEST_CODE, CODE, READWRITE, ALIGN = 2 ; 定义代码段
```

```
ENTRY ; 指定应用程序的入口点
```

```
CODE32
```

```
主程序代码
```

```
END ; 表示源文件的代码结束
```

⑥ 常量定义伪指令: EQU

语法如下:

```
Name EQU Expression(, Type)
```

跟标准 C 语言的“define”语法一样,EQU 主要为数字常量、标号、变量等定义一个名称,其中,Type 表示表达式中的数据类型,可以是 CODE16、CODE32、DATA。

举例如下:

```
TEST EQU 0x1122, CODE16      ;定义 TEST 的值为 0x1122,且该处
                               ;为 16 位的 THUMB 指令
TEST EQU base_addr, CODE32   ;定义 TEST 的值为 base_addr,且该处
                               ;为 32 位的 ARM 指令
```

⑦ 声明全局变量伪指令 1: EXPORT

语法如下:

```
EXPORT Label {[WEAK]}
```

EXPORT 伪指令用于在程序中声明一个全局的标号,该标号可在其他的文件中引用。标号在程序中区分大小写,[WEAK]选项声明其他的同名标号优先于该标号被引用。

举例如下:

```
AREA TEST_CODE, CODE, READWRITE, ALIGN = 2
EXPORT TMP                      ;声明一个可全局引用的标号 TMP
```

⑧ 声明全局变量伪指令 2: GLOBAL

语法如下:

```
GLOBAL Label {[WEAK]}
```

GLOBAL 伪指令用于在程序中声明一个全局的标号,该标号可在其他的文件中引用。标号在程序中区分大小写,[WEAK]选项声明其他的同名标号优先于该标号被引用。和 EXPORT 指令语法一样。

举例如下:

```
AREA TEST_CODE, CODE, READWRITE, ALIGN = 2
GLOBAL TMP                      ;声明一个可全局引用的标号 TMP
```

⑨ IMPORT 伪指令

语法如下:

```
IMPORT Label {[WEAK]}
```

IMPORT 伪指令定义的标号是别的文件定义,可能会在本文件引用。

标号在程序中区分大小写,[WEAK]选项表示当所有的源文件都没有定义这样一个标号时,编译器也不给出错误信息,在多数情况下将该标号置为 0,若该标号为 B 或 BL 指令引用,则将 B 或 BL 指令置为 NOP 操作。

举例如下:

```
AREA TEST_CODE, CODE, READWRITE, ALIGN = 2
IMPORT Init                    ;通知编译器当前文件要引用标号 Init,
```

;但 Init 在其他源文件中定义

⑩ EXTERN 伪指令

语法如下:

EXTERN Label {[WEAK]}

EXTERN 伪指令定义的标号是别的文件定义,可能会在本文件引用。和 IMPORT 语法一样。标号在程序中区分大小写,[WEAK]选项表示当所有的源文件都没有定义这样一个标号时,编译器也不给出错误信息,在多数情况下将该标号置为 0,若该标号为 B 或 BL 指令引用,则将 B 或 BL 指令置为 NOP 操作。

举例如下:

AREA TEST_CODE, CODE, READWRITE, ALIGN = 2

EXTERN Init ;通知编译器当前文件要引用标号 Init,
;但 Init 在其他源文件中定义

⑪ 包含伪指令 1: GET 和 INCLUDE

语法如下:

GET/INCLUDE 文件名 (GET 和 INCLUDE 语法一样)

GET/INCLUDE 伪指令使用方法与标准 C 语言中的“include”非常相似,用于将一个源文件包含到当前的源文件中。但是需要注意的是:GET/INCLUDE 伪指令只能用于包含源文件,包含目标文件需要使用 INCBIN 伪指令。

举例如下:

INCLUDE test.s

GET test1.s

⑫ 包含伪指令 2: INCBIN

语法如下:

INCBIN 文件名

INCBIN 伪指令用于将一个目标文件或数据文件包含到当前文件中,被包含的文件代码或者数据不作任何变动的存放在当前文件中,编译器从伪指令后面开始继续处理。

举例如下:

INCBIN test.bin ;通知编译器当前源文件包含文件 a1.dat

主程序

⑬ RN 伪指令

语法如下:

Name RN Expression

RN 为寄存器命名。

举例如下:

W11_R6 RN R6 ;将 R6 定义一个别名 W11_R6

W11_R8 RN R8

;将 R8 定义一个别名 W11_R8

⑭ ROUT 伪指令

语法如下:

{Name} ROUT

ROUT 伪指令为局部标号定义作用范围。在程序中未使用该伪指令时,局部标号的作用范围为所在的段,而使用 ROUT 后,局部标号的作用范围为当前 ROUT 和下一个 ROUT 之间。

举例如下:

Test1 ROUT

代码 1

Test2 ROUT

局部有效范围的名称为 Test1,新的有效范围为 Test2。

2. 汇编语言规范

(1) ARM/THUMB 汇编语句格式

ARM(THUMB)汇编语言的语句格式为:

{标号} {指令或|条件} {操作数} {;注释}

ARM 汇编对符号大小写敏感,区分大小写字母。并且所有的指令/伪指令可以全部为大写字母或者小写字母,但不能大小写混用。注释使用符号“;”。所有指令不能顶格书写。跟其他语法规则一样,可以使用字符“\”续接单行写不完的部分内容。不允许顶格用“:”符号。举例如下:

```
/* disable interrupts and force SVC32 mode, just like reset */
Code1 MRSr1, cpsr
      BIC r1, r1, #MASK_MODE
      ORR r1, r1, #MODE_SVC32 | I_BIT | F_BIT
      MSR cpsr, r1

/* CPU INTERRUPTS DISABLED */
Code2 LDR r2, L$_SndsIntmsk /* R2→interrupt controller */
      MVN r1, #0 /* &FFFFFFFF */
      STR r1, [r2] /* disable all interrupt sources */
```

上面的例子中指令不允许顶格,标号在顶格,可以使用注释。

(2) 标号使用规范

标号代表代码地址,可以分为全局标号和局部标号。全局标号一般表示分支的目标地址或者绝对地址,局部标号作用于局部范围代码内,作用域为当前段或者也可用在 ROUT 伪指令定义的范围。

局部标号的语法格式为:

$\% \{F|B\} \{A|T\} N\{name\}$

N 为局部标号名, name 为局部标号作用范围的名称。“%”表示局部标号引用操作;“F”表示编译器只向前搜索;“B”表示编译器只向后搜索;“A”表示编译器搜索宏的所有嵌套层次;“T”指示编译器搜索宏的当前层次。例如:

```
CODE ROUT
代码 1
1CODE
      BNQ  % 2CODE
2CODE
代码 2
```

(3) 符号使用规范

在 ARM/THUMB 汇编中,符号可以表示地址、变量和常量等,如果表示地址则称为标号,如上面的介绍。符号的使用规范有:符号是由大小写字母、数字和下划线组成,且一般不能以数字开头;区分大小写,要注意两者的区别;符号在其作用范围内必须唯一;符号不能与系统预留的符号重名;符号名不应与指令或伪指令同名。

(4) 变量表示规范

变量主要有以下几种:数字变量、逻辑变量、字符串变量、字符变量。下面分别进行介绍。

① 数字变量。跟标准 C 一样,数字变量用于在程序的运行中保存数字值,但注意数字值的大小不应超出数字变量所能表示的范围。

② 逻辑变量。跟标准 C 一样,逻辑变量用于在程序的运行中保存逻辑值,逻辑值只有两种取值情况:真或假。

③ 字符串变量。字符串变量用于在程序的运行中保存一个字符串。

④ 字符变量。字符变量用于在程序的运行中保存一个字符。

(5) 常量表示规范

常量也主要有以下几种:数字常量、逻辑常量、字符串常量、字符常量。下面分别进行介绍。

① 数字常量。跟标准 C 一样,数字常量用于在程序的运行中保存数字值,但注意数字值的大小不应超出数字常量所能表示的范围。

② 逻辑常量。跟标准 C 一样,逻辑常量用于在程序的运行中保存逻辑值,逻辑值只有两种取值情况:真或假。

③ 字符串常量。字符串常量用于在程序的运行中保存一个字符串。

④ 字符常量。字符常量用于在程序的运行中保存一个字符。

(6) 段定义

ARM 汇编中至少需要一个代码段,当然可以有多个代码段和数据段。通常代码段为只读的,数据段为可读写的。

(7) 运算符使用规范

ARM 汇编中,运算次序遵循如下的优先级:

- ① 括号运算符的优先级最高。
- ② 相邻的单目运算符的运算顺序为从右到左,且单目运算符的优先级高于其他运算符。
- ③ 优先级相同的双目运算符的运算顺序为从左到右。

3. 汇编语言的程序结构

(1) 汇编语言的程序结构

在 ARM(THUMB)汇编语言程序中,主要的单位是段,分为代码段和数据段,在 AREA 里进行设置。只要有一个代码段,多个段共同构成程序,由编译器编译成可执行文件。一般来说,代码段都是只读的,数据段可以读写。

(2) 汇编语言的子程序调用

在 ARM 汇编语言程序中,子程序的调用一般是通过 BL 指令来实现的。在程序中,使用指令:

BL 子程序名

即可完成子程序的调用。但是要特别注意寄存器的使用规则:子程序间通过寄存器 R0~R3 来传递参数;子程序使用寄存器 R4~R11 来保存局部变量;寄存器 R12 用作过程调用中间临时寄存器;寄存器 R13 用作堆栈指针;寄存器 R14 用作连接寄存器 LR,在程序调用回来的时候,LR 中保存了返回地址,需要将 LR 拷回到 PC 中;寄存器 R15 是程序计数器。

(3) 汇编语言与 C/C++ 的混合编程

汇编语言虽然执行速度比较快,对底层的操作效率比较高,但是汇编程序不容易编写大程序、复杂的程序,结构不好控制,而且汇编语言可维护性差,不是作者的话很难看懂源程序。另外,汇编程序也不利于移植,因为它的语法跟体系结构有关系。一套完整的嵌入式程序一般由汇编语言和高级语言 C/C++ 互相嵌套完成。举例如下:

C 语言程序:

```
extern void keyled(void);  
int main(void)  
{  
    初始化程序;  
    keyled();  
}
```

ARM 汇编程序:

keyled	; 由 C 跳入汇编的入口
START	
CHECK LDR R2, = PDATG	; 查询键盘有无按键按下动作
LDR R3, [R2]	
AND R3, R3, # 0X10	
CMP R3, # 0X10	
BEQ CHECK	
(4) 汇编语言程序示例	
AREA STEP, CODE, READONLY	
EXPORT STEPCTRL	
TEST	
START MOV R0, # 0	
ADR R2, TABLE	; 将表头地址存到 R2 中
loop LDR R5, = 0x0a000000	; bank5
LDRB R1, [R2], # 1	; 注意这里是先将 R2 中的数据(一个字节)
	; 送入 R1 后 R2 的地址再加 1
STRB R1, [R5]	; 将 R1 中的数据送入 R5 中的数据所表示的地
	; 址中
MOV R1, # 0	; 从本句开始到 DELAYOVER 为两极循环嵌
	; 套延时
REPEAT ADD R1, R1, # 1	; 将 R1 中的值加 1 再送入 R1 中
CMP R1, # 400	; [R1] 和 400 比较
BNE NEXT1	; 如果上面的比较结果是不等则跳转
B DELAYOVER	
NEXT1 MOV R3, # 0	
NEXT2 ADD R3, R3, # 1	
CMP R3, # 40	; 将 R3 中的数据和 40 进行比较, 不等则
BNE NEXT2	; 跳转
B REPEAT	
DELAYOVER	
ADD R0, R0, # 1	; 控制循环八次
CMP R0, # 8	

```
BNE loop
B    START
TABLE
    DCB 0x10,0x30,0x20,0x60,0x40,0xc0,0x80,0x90
END
```

(注:本章节内容参考了 ARM 公司的 ARM Architecture Reference Manual 和三星公司关于 S3C4510B 芯片的用户手册以及李煦光的《ARM 应用系统开发详解——基于 S3C4510B 的系统设计》等资料。)

第 4 章 VxWorks 操作系统

4.1 系统概述

VxWorks 操作系统是美国 Wind River 公司(风河公司,或者简称为 WR 公司)推出的一个实时操作系统。风河公司组建于 1981 年,地点在美国加利福尼亚州,是一个专门从事实时操作系统研发的软件公司,并且提供了强大的嵌入式集成开发平台 Tornado 等。利用 Tornado 等开发工具的强大支持,客户可以方便地开发自己的产品。风河公司推出的实时操作系统在业界有很好的声誉。

VxWorks 是一个运行在目标机上的高性能、可裁减的嵌入式实时操作系统。VxWorks 以其良好的可靠性和卓越的实时性被广泛地应用在通信、军事、航空、航天等高精尖技术及实时性要求极高的领域中,如卫星通信、军事演习、弹道制导、飞机导航等。在美国的 F-16、FA-18 战斗机、B-2 隐形轰炸机和爱国者导弹上,甚至连 1997 年 4 月在火星表面登陆的火星探测器上也使用了 VxWorks。当然,在民用方面,VxWorks 也占有很大一部分市场。可以说,VxWorks 操作系统在很多领域和行业都得到了极大的认可和广泛的应用。

VxWorks 实时操作系统的特点主要有以下几个方面。

(1) 良好的实时性和稳定性

VxWorks 操作系统支持优先级抢先调度和时间片轮转调度策略,而且任务本身的开销也非常小,实现了很好的实时性;

稳定性是衡量一个实时操作系统的重要指标,VxWorks 操作系统在这方面一直都处于领先地位。强大的市场占有率、在国防军事领域的多种应用充分体现了 VxWorks 操作系统强大的稳定性和可靠性。

(2) 高效的任务管理

支持多任务环境,具有优先级为 0~255 的 256 个优先级;

每个任务都有自己的任务控制块 TCB,负责管理任务的所有信息;

强大的任务调度算法和排队策略;

任务之间的上下文切换速度快。

(3) 灵活的任务间通信

支持多种通信方式和手段,如信号量、消息队列、共享内存、信号、事件等。

(4) 微秒级的中断处理

中断处理时间的多少是衡量嵌入式实时系统的一个重要指标。VxWorks 操作系统在这方面非常出色,达到了微妙级的中断处理,使整个操作系统获得了快速、高效的性能。

(5) 方便移植、构建的 VxWorks 板支持包(BSP)

风河公司提供了支持多种体系结构的板极支持包(BSP)。在 Tornado 等工具的支持下,用户可以方便地进行 BSP 包的设计、调试和移植,大大缩短了产品的开发周期和费用。

(6) 支持多种标准

支持标准 ANSI C;

支持 POSIX 规范;

支持其他多种标准,体现了 VxWorks 操作系统良好的兼容性,使得系统的移植非常方便,适应于各种风格的开发者。

(7) 丰富的网络功能

随着网络的普及和网络技术的快速发展,VxWorks 操作系统本身也具有丰富的网络功能,可以应用在远程控制、网络通信等许多领域。它的结构如图 4-1 所示。

总之,VxWorks 操作系统是一个集许多优势于一体的商业实时操作系统。

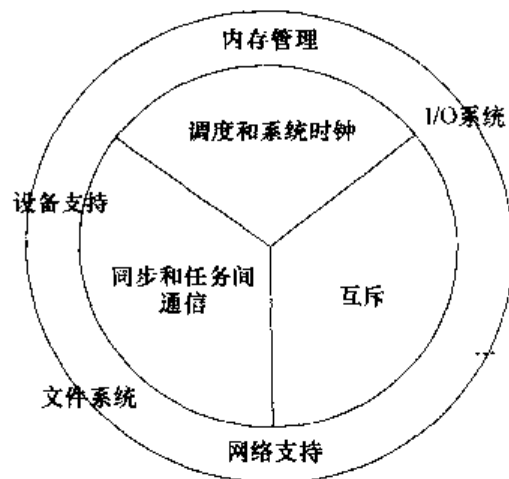


图 4-1 VxWorks 操作系统构成

4.2 多任务介绍

4.2.1 任务概述

在 VxWorks 中,任务是竞争资源的最小单位。操作系统最主要的功能就是为任务服务,提供一个多任务的环境。在每个实时系统中均有很多的任务,协助合作、共同完成

某些功能。

在每个任务中,都有一个任务控制块(Task Control Block,TCB),主要用于记录任务的状态、参数、临时变量。任务控制块的主要内容包括:

- (1) 任务的程序计数器;
- (2) 处理器的通用寄存器;
- (3) 局部变量和堆栈;
- (4) 时间片定时器;
- (5) 出错信息;
- (6) 断点列表。

任务一旦创建,就处在有限的几个状态之一:挂起状态(Suspend)、准备状态(Ready)、阻塞状态(Pend)、延迟状态(Delay)和运行状态(Running)。在 Tornado 环境中专门的工具可以分析任务在任何时刻所处的状态,具体可以参考附录的 Shell 命令集。由于处理器的独占性,在一个时刻只允许一个任务运行,享用处理器资源。任务之间切换的时候会发生所谓的上下文切换,但是需要注意的是除了运行状态外,任务在其他状态之间切换并不会发生上下文切换。图 4-2 为一个在 Tornado 中显示任务状态的例子。

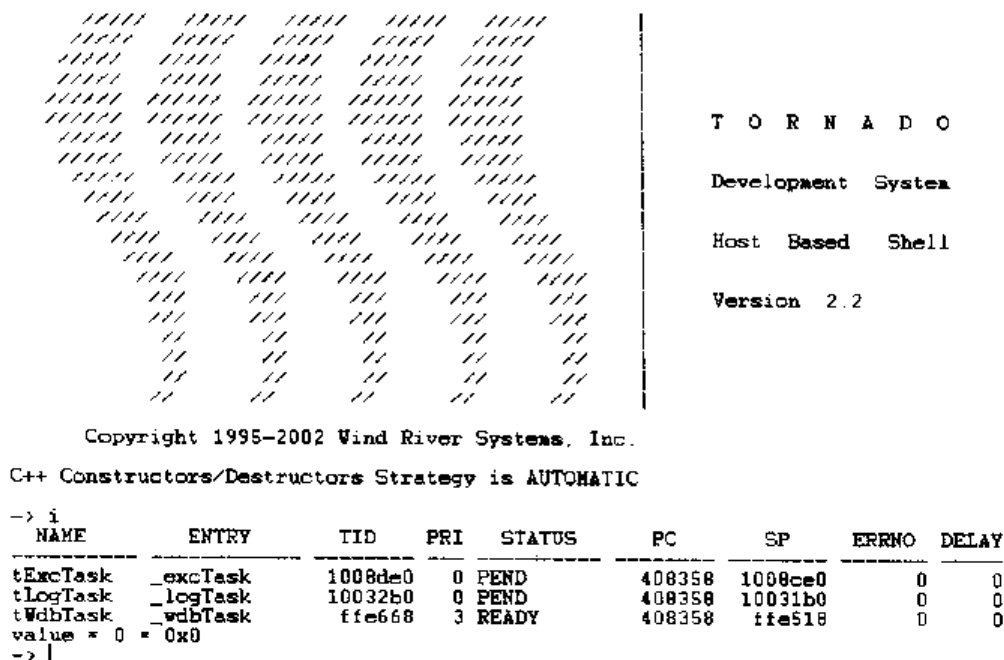


图 4-2 Tornado 中分析任务状态

决定任务之间调度的是 VxWorks 中的调度算法,具体调度算法可以参照前面第 2 章的内容。VxWorks 中支持时间片调度算法和优先级调度算法,默认的是优先级调度算法。一般情况下,每个任务在初始化的时候设置一个固定优先级(动态优先级一般

不常用,系统比较复杂,不好控制),在 0~255 之间,然后操作系统会根据每个任务的优先级进行调度。当然,在 Tornado 中也可以查看任务的优先级,图 4-2 中也显示了任务的优先级。任务 tExcTask 和 tLogTask 的优先级为 0,任务 tWdbTask 的优先级为 3。

4.2.2 任务的创建

在 VxWorks 中,任务的创建主要由以下几个函数来完成:taskSpawn()、taskInit()、taskActivate()。下面分别进行介绍。

(1) taskSpawn()

taskSpawn():创建一个任务。

函数原型为:

```
int taskSpawn
(
    char *   name,          /* 创建新任务的名称 */
    int      priority,      /* 新任务的优先级 */
    int      options,       /* 任务的可选项 */
    int      stackSize,     /* 任务的堆栈大小 */
    FUNCPTR  entryPt,       /* 新任务的函数入口 */
    int      arg1,          /* 以下为入口函数的 1~10 个参数 */
    int      arg2,
    int      arg3,
    int      arg4,
    int      arg5,
    int      arg6,
    int      arg7,
    int      arg8,
    int      arg9,
    int      arg10
)
```

利用 taskSpawn 函数创建完任务后,如果创建成功则返回一个任务的 ID,否则返回 -1。

VxWorks 限制任务名称长度最大到 10 个字符,并且在任务运行后给它们加上前缀“t”。如果名称为空,系统会默认指定一个名称,格式为“tn”,其中 n 为创建任务的序号。

其中 options 的可选参数有以下几个。

① VX_FP_TASK(0x0008)

运行时使用浮点计算器,将浮点寄存器作为任务上下文的一部分。任何涉及到浮点的操作或者在函数参数中使用浮点变量时必须包含此选项。

② VX_PRIVATE_ENV (0x0080)

支持私有环境。

③ VX_NO_STACK_FILL (0x0100)

任务创建时不以 0xEE 填充堆栈,不能使用 checkStack()。

④ VX_UNBREAKABLE (0x0002)

一旦调试成功,返回任务的 ID,否则返回 ERROR(-1),并利用全局变量 errno 告知错误原因,不允许断点调试。

使用 taskOptionGet()来查询任务的 options,使用 taskOptionSet()来设置或复位任务的 options。

分配给新任务的唯一资源就是指定大小的堆栈,堆栈都来自系统内存池。堆栈大小必须为偶数。下面给出创建任务的范例代码。

```
#include "VxWorks.h"
#include "taskLib.h"
#include "stdio.h"
STATUS SerialTask(void);
int tyCoTest()
{
    int taskId;
    taskId = taskSpawn("SerialTask", 120, 0, 10240, (FUNCPTR)SerialTask, 0,
0,0,0,0,0,0,0,0,0,0);
}
STATUS SerialTask(void)
{
    while(1)
    {

    }
}
```

在这里,举一个最简单的例子,成功创建后,可以在 Shell 中看到创建好的名字 SerialTask。如图 4-3 所示。


```
Shell vxsim@vll
```

```
TORNADO  
Development System  
  
Host Based Shell  
  
Version 2.2
```

```
Copyright 1995-2002 Wind River Systems, Inc.
```

```
C++ Constructors/Destructors Strategy is AUTOMATIC
```

```
-> tyCoTest  
value = 16715616 = 0xff0f60  
->
```

	NAME	ENTRY	TID	PRI	STATUS	PC	SP	ERRNO	DELAY
tExcTask	_excTask		1008d0	0	PEND	408358	1008ce0	0	0
tLogTask	_logTask		10032b0	0	PEND	408358	10031b0	0	0
tVdbTask	_vdbTask		f1e668	3	READY	408358	f1e518	0	0
SerialTask	_SerialTask		f10f60	120	READY	e453b4	f10eec	0	0

```
value = 0 = 0x0  
->
```

图 4-3 用 taskSpawn 创建好的任务

(2) taskInit()

函数原型为：

STATUS taskInit


```
WIND_TCB * pTcb,          /* 任务 TCB 的地址 */
char *      name,          /* 任务的名称 */
int         priority,      /* 任务的优先级 */
int         options,       /* 任务的可选项 */
char *      pStackBase,    /* 任务堆栈的基址 */
int         stackSize,     /* 堆栈所需的大小 */
FUNCPTR     entryPt,       /* 任务的函数入口 */
int         arg1,          /* 下面 10 个为传递到入口函数的 10 个参数 */
int         arg2,
int         arg3,
int         arg4,
int         arg5,
int         arg6,
int         arg7,
int         arg8,
```

```

        int          arg9,
        int          arg10
    )

```

可以看到 taskInit 函数比 taskSpawn 函数增加了需要指定任务的 TCB 指针和堆栈指针,其他参数基本上一样。这就需要开发人员熟悉内存的使用情况。

(3) taskActivate()

函数原型为:

```

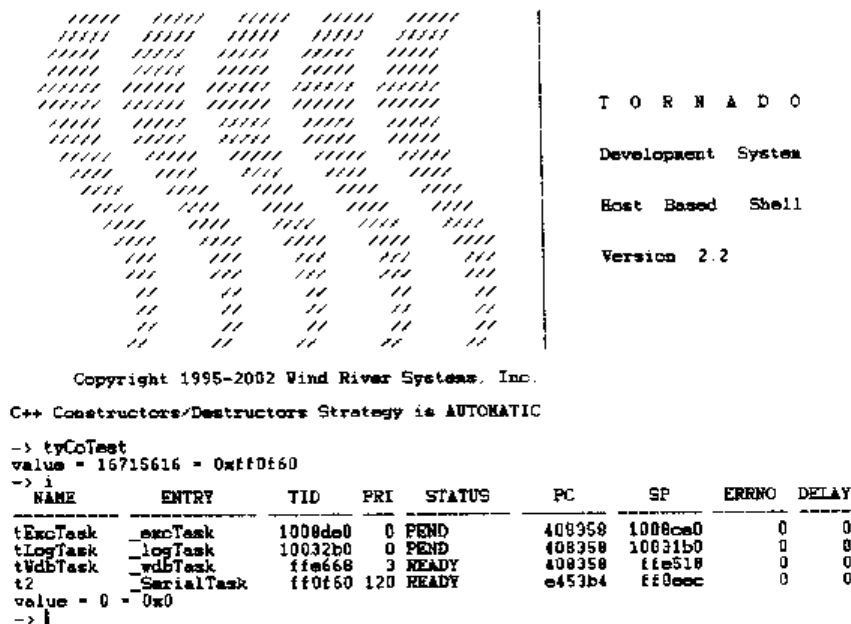
STATUS taskActivate
{
    int tid /* 任务的 ID */
}

```

当使用 taskInit 创建任务后,任务并没有开始执行,必须再调用 taskActivate 函数才可以激活任务,使任务运行。

4.2.3 任务名和任务标识

任务名是一个任意长度的 ASCII 码字符串,一般来说需要指定任务的名称,这将大大方便调试和管理任务,如果不指定任务名称的话,VxWorks 操作系统会分配一个默认的名字“tn”,其中 n 是任务的序号,按照增序赋值。如图 4-4 所示。



```

T O R N A D O
Development System
Host Based Shell
Version 2.2

Copyright 1995-2002 Wind River Systems, Inc.
C++ Constructors/Destructors Strategy is AUTOMATIC
-> tyCoTest
value = 16715616 = 0x1f0f60
-> 1

```

NAME	ENTRY	TID	PRI	STATUS	PC	SP	ERRNO	DELAY
tExcTask	_excTask	1000dc0	0	PEND	400958	1000cc0	0	0
tLogTask	_logTask	10032b0	0	PEND	408358	10031b0	0	0
tVdbTask	_vdbTask	ffe668	3	READY	400358	ffe518	0	0
t2	_SerialTask	ff0f60	120	READY	e453b4	ffe0ec	0	0

```

value = 0 = 0x0
-> |

```

图 4-4 创建任务没有指定名称

任务的 ID 其实就是任务的 TCB 的地址指针,这可以从 taskInit 函数中看出来。

相关的函数有以下几个。

(1) **taskName()**

函数原型如下：

```
char * taskName
(
    int tid          /* 任务的 ID */
)
```

描述：如果任务有名字，通过此函数返回指向名称的字符指针，否则返回一个空字符串。

(2) **taskNameToId()**

函数原型如下：

```
int taskNameToId
(
    char * name /* 指向任务名称的指针 */
)
```

此函数返回指定名称任务的 ID。

(3) **taskIdSelf()**

函数原型如下：

```
int taskIdSelf (void)
```

返回调用任务的 ID。

(4) **taskIdVerify()**

函数原型如下：

```
STATUS taskIdVerify
(
    int tid          /* 任务的 ID */
)
```

此函数检查是否存在具有指定 ID 的任务。

4.2.4 任务的删除和保护

可以用 `exit( )`、`taskDelete( )`、`taskDeleteForce( )` 3 个函数来删除任务。任务删除时，收回其所占用的资源：TCB 和堆栈。

但是，如果一个任务 A 调用了 `taskSafe( )` 函数进行保护，则其他任务试图删除它时，会处于阻塞状态，直到任务 A 解除了保护。任务解除保护可以通过调用 `taskUnsafe( )` 来进行。

这几种函数的具体用法如下：

```
STATUS taskDelete
```

```
(
    int tid          /* 要删除的任务的 ID */
)

void exit
(
    int code         /* 删除函数需要的参数,存储在 TCB 中 */
)

STATUS taskSafe(void)
/* 对任务采取保护函数 */

STATUS taskUnsafe(void)
/* 取消任务保护函数 */

STATUS taskDeleteForce
(
    int tid          /* 要删除的任务的 ID */
)
```

4.2.5 任务的信息

任务的不同信息可以通过调用系统函数来获得,主要的相关函数有以下几个。

(1) **taskInfoGet()**: 获得任务的相关信息

函数原型如下:

```
STATUS taskInfoGet
(
    int      tid,          /* 要查询的任务的 ID */
    TASK_DESC * pTaskDesc
)
```

(2) **taskIdListGet()**: 获取所有活动任务的 ID

函数原型如下:

```
int taskIdListGet
(
    int idList[],          /* 所有 ID 数组 */

```

```

    int maxTasks          /* 查询的最大任务数 */
)

```

(3) **taskPriorityGet()**: 获得任务的优先级

函数原型如下:

```
STATUS taskPriorityGet
```

```

(
    int tid,              /* 任务的 ID */
    int * pPriority        /* 返回优先级 */
)

```

(4) **taskRegsGet()**: 获得指定任务的寄存器内容

函数原型如下:

```
STATUS taskRegsGet
```

```

(
    int      tid,          /* 任务的 ID */
    REG_SET * pRegs        /* 返回的寄存器内容指针 */
)

```

(5) **taskRegsSet()**: 设置指定任务的寄存器

函数原型如下:

```
STATUS taskRegsSet
```

```

(
    int      tid,          /* 任务的 ID */
    REG_SET * pRegs        /* 设置寄存器内容的指针 */
)

```

(6) **taskIsSuspended()**: 查看任务是否处于 Suspended 状态

函数原型如下:

```
BOOL taskIsSuspended
```

```

(
    int tid                /* 任务的 ID */
)

```

(7) **taskIsReady()**: 查看任务是否处于 Ready 状态

函数原型如下:

```
BOOL taskIsReady
```

```

(
    int tid                /* 任务的 ID */
)

```

(8) **taskTcb()**: 获得任务 TCB 指针

函数原型如下:

```
WIND_TCB * taskTcb
(
    int tid                /* 任务的 ID */
)
```

4.2.6 任务的控制

VxWorks 中提供了很多任务控制功能,具体有以下几种。

(1) **taskSuspend()**: 挂起一个任务

函数原型如下:

```
STATUS taskSuspend
(
    int tid                /* 要挂起的任务的 ID */
)
```

(2) **taskResume()**: 恢复挂起的任务

函数原型如下:

```
STATUS taskResume
(
    int tid                /* 要恢复的任务的 ID */
)
```

(3) **taskRestart()**: 重新启动任务

函数原型如下:

```
STATUS taskRestart
(
    int tid                /* 要重启的任务的 ID */
)
```

(4) **taskDelay()**: 将指定任务延迟一段时间

函数原型如下:

```
STATUS taskDelay
(
    int ticks              /* 要延迟的 tick 数 */
)
```

(5) **taskLock()**: 禁止任务被重新调度

函数原型如下:

```
STATUS taskLock (void)
```


4.3 任务间的通信

4.3.1 概述

VxWorks 中任务之间的通信手段是非常丰富的,支持信号量、消息队列、管道、信号、事件、共享内存等。不同的通信方式具有不同的特点,在具体使用的时候需要注意。

4.3.2 信号量

信号量是实现任务同步和互斥的重要手段,在第 2 章讲实时系统的时候也提到过。相比其他的通信手段,信号量实现速度最快。

VxWorks 提供了 3 种不同功能的信号量:二进制信号量、互斥信号量、计数信号量。下面就具体介绍这 3 种信号量的用法。

1. 二进制信号量

二进制信号量速度最快,可以实现任务之间的同步和互斥操作。VxWorks 提供的系统函数有以下几个。

(1) **semBCreate()**:创建二进制信号量

函数原型如下:

```
SEM_ID semBCreate
```

```
{
```

```
    int          options,          /* 信号量选项 */
```

```
    SEM_B_STATE initialState      /* 初始化信号量状态 */
```

```
}
```

options 选项为 EM_Q_PRIORITY (0x1)或者 SEM_Q_FIFO (0x0),前者是基于优先级的队列,后者是基于先进先出的队列。

initialState 可以为 SEM_FULL (1) 或者 SEM_EMPTY (0)。前者是将信号量初始化为满,后者则是初始化为空。

(2) **semTake()**:获取信号量

函数原型如下:

```
STATUS semTake
```

```
{
```

```
    SEM_ID semId,                  /* 信号量 */
```

```
    int      timeout               /* 等待信号量的时间 */
```

```
}
```


其中 timeout 选项可以为 WAIT_FOREVER (-1) 或者 NO_WAIT (0), 前者表示任务一直等待信号量直到可用, 后者表示不等待。如果获取信号量的时候信号量为空, 则不获取, 直接进行下一步操作。还可以为指定的一个 tick 数。

(3) semGive(): 释放信号量

函数原型如下:

```
STATUS semGive
(
    SEM_ID semId          /* 释放的信号量 */
)
```

(4) semFlush(): 唤醒信号量

函数原型如下:

```
STATUS semFlush
(
    SEM_ID semId          /* 要释放的信号量 */
)
```

此函数可以释放所有阻塞在指定信号量上的任务。

(5) semDelete(): 删除信号量

函数原型如下:

```
STATUS semDelete
(
    SEM_ID semId          /* 要删除的信号量 */
)
```

(6) semInfo(): 得到阻塞在信号量上的所有任务 ID 列表

函数原型如下:

```
int semInfo
(
    SEM_ID semId,          /* 相关的信号量 */
    int idList[],          /* 要显示的任务 ID 数组 */
    int maxTasks           /* 要显示的任务最大数 */
)
```

(7) semShow(): 显示信号量的相关信息

函数原型如下:

```
STATUS semShow
(
    SEM_ID semId,          /* 要显示的信号量 */

```

```

        int    level                /* 0 = 概括, 1 = 详细 */
    )

```

例 4-1 二进制信号量的实例。

```

#include "VxWorks.h"
#include "taskLib.h"
#include "semLib.h"
#include "stdio.h"
/* 声明两个任务 */
STATUS task1(void);
STATUS task2(void);
/* 全局的变量 */
int taskId1, taskId2;
SEM_ID semTest1, semTest2;

int initTask()
{
    taskId1 = taskSpawn("task1", 120, 0, 10240, (FUNCPTR)task1, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0);
    taskId2 = taskSpawn("task2", 120, 0, 10240, (FUNCPTR)task2, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0);
    semTest1 = semBCreate(SEM_Q_PRIORITY, SEM_EMPTY);
    semTest2 = semBCreate(SEM_Q_PRIORITY, SEM_EMPTY);
    if(semTest1 == NULL)
    {
        return ERROR;
    }

    return OK;
}

STATUS task1(void)
{
    while(1)
    {
        semGive(semTest1);

```

```
        printf("Task1 free the semTest1! \n");
        semTake(semTest2, WAIT_FOREVER);
        printf("Task1 got the semTest2! \n");
    }
}

STATUS task2(void)
{
    while(1)
    {
        semTake(semTest1, WAIT_FOREVER);
        printf("task2 got the semTest1! \n");
        semGive(semTest2);
        printf("task2 free the semTest2! \n");
    }
}

/* 删除任务 */
int deleteTask(void)
{
    if(taskDelete(taskId1) == ERROR)
    {
        return ERROR;
    }

    if(taskDelete(taskId2) == ERROR)
    {
        return ERROR;
    }

    if(semDelete(semTest1) == ERROR)
    {
        return ERROR;
    }

    if(semDelete(semTest2) == ERROR)
    {
        return ERROR;
    }
}
```

```

    }
    return OK;
}

```

这个例子是应用二进制信号量实现任务同步。两个信号量 SemTest1 和 SemTest2 都初始化为空。任务 1(task1)先释放信号量 SemTest1,使得任务 2(task2)可以执行具体的功能;然后任务 2 释放信号量 SemTest2,使得任务 1 继续执行具体的功能,实现了任务间的同步。读者可以自己练习一下,当然也可以根据本书后半部分的实验部分专门进行练习。

二进制信号量也可以用来进行互斥控制。用于互斥控制的信号量初始化为“满”,表示临界资源处于可用状态。当一个任务需要访问临界资源时,先占用这个信号量,从而排斥了其他任务对临界资源的访问。只有当这个任务结束了对于临界资源的访问,并且释放信号量后,其他任务才可以获取这个信号量,并访问临界资源。

2. 互斥信号量

互斥信号量跟二进制信号量进行互斥操作相比,增加了一些功能,例如防止优先级倒置、递归访问等。跟它相关的函数主要有以下几个。

(1) semMCreate:创建互斥信号量

函数原型如下:

```

SEM_ID semMCreate
(
    int options                /* 互斥信号量选项 */
)

```

其中,options 有下列可选项。

- SEM_Q_PRIORITY (0x1):需要获取信号量的任务基于优先级原则排列。
- SEM_Q_FIFO (0x0):需要获取信号量的任务基于先进先出原则排列。
- SEM_DELETE_SAFE (0x4):保护任务防止意外删除,当执行 semTask 操作时会默认执行 taskSafe 操作,当执行 semGive 操作时会默认执行 taskUnsafe 操作。
- SEM_INVERSION_SAFE (0x8):保护系统,防止系统出现优先级倒置现象。
- SEM_EVENTSEND_ERR_NOTIFY (0x10):任务发送事件失败时,会发送错误号。

(2) semMGiveForce:强制释放互斥信号量

函数原型如下:

```

STATUS semMGiveForce
(
    SEM_ID semId              /* 要释放的互斥信号量 */
)

```

此函数强制释放互斥信号量,但是它仅仅用于系统调试。

例 4-2 互斥信号量的实例。

```
#include "VxWorks.h"
#include "taskLib.h"
#include "semLib.h"
#include "stdio.h"
/* 声明两个任务 */
STATUS task1(void);
STATUS task2(void);
/* 全局的变量 */
int taskId1,taskId2;
SEM_ID semTest1,semTest2;

int initTask()
{
    taskId1 = taskSpawn("task1", 120, 0, 10240, (FUNCPTR)task1, 0,0,0,0,0,
0,0,0,0,0);
    taskId2 = taskSpawn("task2", 120, 0, 10240, (FUNCPTR)task2, 0,0,0,0,0,
0,0,0,0,0);
    semTest1 = semMCreate(SEM_Q_PRIORITY,);
    semTest2 = semMCreate(SEM_Q_PRIORITY,);
    if( semTest1 == NULL)
    {
        return ERROR;
    }
    return OK;
}

STATUS task1(void)
{
    while(1)
    {
        semGive(semTest1);
        printf("Task1 free the semTest1! \n");
        semTake(semTest2, WAIT_FOREVER);
    }
}
```

```

        printf("Task1 got the semTest2! \n");
    }
}

STATUS task2(void)
{
    while(1)
    {
        semTake(semTest1, WAIT_FOREVER);
        printf("task2 got the semTest1! \n");
        semGive(semTest2);
        printf("task2 free the semTest2! \n");
    }
}

/* 删除任务 */
int deleteTask(void)
{
    if(taskDelete(taskId1) == ERROR)
    {
        return ERROR;
    }

    if(taskDelete(taskId2) == ERROR)
    {
        return ERROR;
    }

    if(semDelete(semTest1) == ERROR)
    {
        return ERROR;
    }

    if(semDelete(semTest2) == ERROR)
    {
        return ERROR;
    }

    return OK;
}

```

```
}
```

通过上述例子,读者可以分析对比一下二进制信号量和互斥信号量的不同。

3. 计数信号量

计数信号量在很多实时系统中也很常见,主要用于收集某些信号量。跟它相关的系统函数主要有:

```
SEM_ID semCCreate
(
    int options,                /* 信号量属性或者类型 */
    int initialCount            /* 初始化计数 */
)
```

同样,此函数的 options 选项有:SEM_Q_PRIORITY (0x1),SEM_Q_FIFO (0x0),SEM_EVENTSEND_ERR_NOTIFY (0x10)。

例 4-3 计数信号量的实例。

```
#include "VxWorks.h"
#include "taskLib.h"
#include "semLib.h"
#include "stdio.h"
#define MAX_COUNT 2

STATUS task1(void);
STATUS task2(void);
STATUS task3(void);

int taskId1,taskId2,taskId3;
SEM_ID semTest1,semTest2;
int count;

int initTask()
{
    taskId1 = taskSpawn("task1", 120, 0, 10240, (FUNCPTR)task1, 0,0,0,0,0,
0,0,0,0,0);
    taskId2 = taskSpawn("task2", 110, 0, 10240, (FUNCPTR)task2, 0,0,0,0,0,
0,0,0,0,0);
```

```
taskId3 = taskSpawn("task3", 150, 0, 10240, (FUNCPTR)task3, 0,0,0,0,0,
0,0,0,0,0);
```

```
semTest1 = semCCreate(SEM_Q_PRIORITY,0);
semTest2 = semBCreate(SEM_Q_PRIORITY,SEM_EMPTY);
count = 0;
if(semTest1 == NULL)
{
    return ERROR;
}
return OK;
}
```

```
STATUS task1(void)
{
    semGive(semTest1);
    semTake(semTest2, WAIT_FOREVER);
    while(1)
    {
        printf("task1 is beginning! \n");
        taskDelay(1);
    }
}
```

```
STATUS task2(void)
{
    semGive(semTest1);
    semTake(semTest2, WAIT_FOREVER);
    while(1)
    {
        printf("task2 is beginning! \n");
        taskDelay(3);
    }
}
```



```

STATUS task3(void)
{
    while (count >= MAX_COUNT)
    {
        if(semTake(semTest1, WAIT_FOREVER) != OK)
        {
            printf("Wait! \n");
            count++;
        }
    }
    semFlush(semTest2);
    while(1)
    {
        printf("task3 is beginning! \n");
    }
}

```

在上面的例子中可以看到,task3 实际上起到了一个管理者的作用。刚开始的时候等待 task1 和 task2 释放计数信号量,然后等待二进制信号量。task3 在收齐信号量时,释放二进制信号量,3 个任务进入正常工作状态。

4.3.3 消息队列

信号量方式的通信虽然快速,但是还远远不能满足任务之间的交互通信。通过消息队列方式通信也是 VxWorks 多任务系统中最常见的通信方式。任务之间通过消息队列收发消息,可以起到很好的交互作用。

VxWorks 系统中提供了两种消息队列库:msgQLib 和 mqPxLib。msgQLib 提供了标准的 VxWorks 消息队列,而 msPxLib 提供了 POSIX 消息队列。我们这里主要介绍 VxWorks 消息队列。

跟消息队列相关的系统函数主要有以下几个。

(1) **msgQCreate()**:创建或者初始化消息队列

函数原型为:

MSG_Q_ID msgQCreate

(

int maxMsgs,

/* 最大消息数 */

int maxMsgLength,

/* 消息的最大字节数 */

```
int options                /* 信息的选项 */
)
```

其中 options 可以为:

- MSG_Q_FIFO (0x00): 任务以 FIFO 原则排队;
- MSG_Q_PRIORITY (0x01): 任务以基于优先级的原则排队;
- MSG_Q_EVENTSEND_ERR_NOTIFY (0x02): 消息队列向它的注册任务发送事件失败时, 将会返回一个 ERROR 的值, 并且正确地设置 errno。

成功创建消息队列后会返回消息队列的 ID。

(2) **msgQDelete()**: 删除消息队列

函数原型为:

```
STATUS msgQDelete
(
    MSG_Q_ID msgQId          /* 要删除的消息队列 */
)
```

(3) **msgQSend()**: 发送信息给消息队列

```
STATUS msgQSend
(
    MSG_Q_ID msgQId,          /* 要发送的消息队列 */
    char *   buffer,          /* 要发送的消息指针 */
    UINT     nBytes,          /* 消息的长度 */
    int      timeout,          /* 等待的 tick 数 */
    int      priority          /* 普通还是紧急模式 */
)
```

此函数将消息发送到指定消息队列中, 如果有任务在消息队列中排队等待的话, 任务就按照 FIFO 规则或者优先级规则接受消息。如果 Priority 为紧急模式(MSG_PRI_URGENT (1))的话将消息发到消息队列的头部, 为普通模式(MSG_PRI_NORMAL (0))就发送到消息队列的尾部排队。其中 timeout 选项为等待的 tick 数, 可以为 NO_WAIT (0), 即使消息没有发送出去, 也立刻返回; WAIT_FOREVER (-1), 一直等待发送, 直到消息发送出去。

(4) **msgQReceive()**: 从消息队列中收取消息

函数原型为:

```
int msgQReceive
(
    MSG_Q_ID msgQId,          /* 接收消息的消息队列 */
    char *   buffer,          /* 接收消息的 buffer 指针 */

```

```

    UINT    maxNBytes,          /* buffer 最大字节数 */
    int     timeout             /* 等待的 tick 数 */
)

```

将消息拷贝到指定的 buffer 中,如果消息超过了 buffer 的最大字节数,则超出的部分将被丢弃。

(5) **msgQNumMsgs()**:得到排队到消息队列中的消息数

函数原型为:

```

int msgQNumMsgs
(
    MSG_Q_ID msgQId           /* 消息队列的 ID */
)

```

该函数返回排队到消息队列中的消息数。

例 4-4 消息队列的实例。

```

#include "VxWorks.h"
#include "taskLib.h"
#include "msgQLib.h"
#include "stdio.h"
#define MSG "hello"
STATUS task1(void);
STATUS task2(void);

int taskId1,taskId2; /* 两个收发任务的 ID */
MSG_Q_ID msgQId1;    /* 消息队列 ID */
char buffer[50];     /* 接收消息的 buffer */

int initTask()
{
    taskId1 = taskSpawn("task1", 120, 0, 10240, (FUNCPTR)task1, 0,0,0,0,0,
0,0,0,0,0);
    taskId2 = taskSpawn("task2", 110, 0, 10240, (FUNCPTR)task2, 0,0,0,0,0,
0,0,0,0,0);
    msgQId1 = msgQCreate(20,50,MSG_Q_PRIORITY);
    if(msgQId1 == NULL)
    {

```

```

        return ERROR;
    }
    return OK;
}

STATUS task1(void)
{
    while(1)
    {
        if(msgQSend(msgQId1,MSG,sizeof(MSG),WAIT_FOREVER,
                    MSG_PRI_NORMAL) == ERROR)
        {
            return ERROR;
        }
        taskDelay(10);
    }
}

STATUS task2(void)
{
    while(1)
    {
        if(msgQReceive(msgQId1,buffer,50,WAIT_FOREVER) == ERROR)
        {
            return ERROR;
        }
        printf("the message is %s\n",buffer);
    }
}

```

在这个例子中,共有两个任务 task1 和 task2,task1 给 task2 发送消息,task2 将消息接收下来,然后打印出来。

4.3.4 管道

管道是一种虚拟的 I/O 设备,所以对管道可以通过标准的 I/O 函数(open()、close()、write()、ioctl()、read())进行操作。

跟管道有关的函数有以下几个。

(1) **pipeDevCreate()**: 创建管道

函数原型为:

```
STATUS pipeDevCreate
(
    char * name,      /* 管道名 */
    int    nMessages, /* 管道中的最大数目 */
    int    nBytes     /* 每个消息的字节数 */
)
```

(2) **pipeDevDelete()**: 删除管道

函数原型为:

```
STATUS pipeDevDelete
(
    char * name,      /* 要删除的管道名 */
    BOOL  force       /* 是否强制删除 */
)
```

(3) **pipeDrv()**: 初始化管道

函数原型为:

```
STATUS pipeDrv (void)
```

例 4-5 管道应用的实例。

```
#include "VxWorks.h"
#include "ioLib.h"
#include "pipeDrv.h"
#include "stdio.h"
#define MSG "hello"
/* 声明两个实时任务 */
STATUS task1(void);
STATUS task2(void);
/* 声明一些全局变量 */
int taskId1, taskId2; /* 返回的任务的 ID */
int pipeId; /* 返回的管道文件描述符 */
int temp;
char buffer[50];
/* 初始化任务, 创建管道 */
```

```

int initTask()
{
    taskId1 = taskSpawn("task1", 120, 0, 10240, (FUNCPTR)task1, 0,0,0,0,0,
0,0,0,0,0);
    taskId2 = taskSpawn("task2", 110, 0, 10240, (FUNCPTR)task2, 0,0,0,0,0,
0,0,0,0,0);
    temp = pipeDevCreate("~/pipe/pipeTest",50,50);/* 创建管道 */
    if(temp == ERROR)
    {
        return ERROR;
    }
    pipeId = open("~/pipe/pipeTest",O_RDWR,0);/* 打开管道 */
    if(pipeId == ERROR)
    {
        return ERROR;
    }
    return OK;
}

STATUS task1(void)
{
    while(1)
    {
        if( write(pipeId,MSG,sizeof(MSG)) == ERROR)/* 向管道里写信息 */
        {
            return ERROR;
        }
        taskDelay(10);
    }
}

STATUS task2(void)
{
    while(1)
    {

```

```

        if(read(pipeId,buffer,50) == ERROR)/* 向管道里读信息 */
        {
            return ERROR;
        }
        printf("the message is %s\n",buffer);
    }
}

```

在上面的例子中,任务1给管道写信息,任务2从管道里读信息,两者通过管道进行通信。另外可以看出,对管道进行的操作是标准的I/O操作。

4.3.5 信号

信号是由事件引发的软中断,中断并非由外界硬件产生,而是由内部程序自己产生。需要特别注意的是,信号的处理程序并非在中断上下文中进行处理,而是在接收任务本身的上下文中进行处理。

跟信号相关的函数有以下几个。

(1) **signal()**:将信号与特定的处理程序绑定

函数原型为:

```

void (* signal
(
    int      signo,          /* 信号 */
    void(* pHandler)() /* 信号处理程序 */
))

```

(2) **kill()**:向指定的任务发送信号

函数原型为:

```

int kill
(
    int tid,                /* 要发送的任务 ID */
    int signo                /* 要发送的信号 */
)

```

例 4-6 信号应用的实例。

```

#include "VxWorks.h"
#include "semLib.h"
#include "sigLib.h"
#include "stdio.h"

```

```

#define MSG "hello"
/* 声明任务 */
STATUS task1(void);
STATUS task2(void);
void sigISR(int sig);
/* 声明全局变量 */
int taskId1,taskId2;
int signalTest = 5;
/* 初始化任务 */
int initTask()
{
    taskId1 = taskSpawn("task1", 120, 0, 10240, (FUNCPTR)task1, 0,0,0,0,0,
0,0,0,0,0);
    taskId2 = taskSpawn("task2", 110, 0, 10240, (FUNCPTR)task2, 0,0,0,0,0,
0,0,0,0,0);
    return OK;
}

STATUS task1(void)
{
    signal(signalTest,sigISR);/* 将信号与信号处理程序绑定 */
    while(1)
    {

    }
}

void sigISR(int sig)
{
    if(sig == signalTest)
    {
        logMsg("The signal num is %d\n",sig,0,0,0,0,0);/* 在控制台打印信
息 */
    }
    return;
}

```



```

STATUS task2(void)
{
    while(1)
    {
        kill(taskId1,signalTest);/* 发送信号给任务 task1 */
        taskDelay(10);
    }
}

```

在上面的程序中,task2 给 task1 发送特定信号 signalTest,task2 收到信号后,转向信号处理程序进行处理。这里需要注意的是打印信息函数 logMsg,而不是 printf 函数,主要是因为调用 printf 函数可能会引起中断阻塞,这对系统来讲是非常致命的,可能引起系统的瘫痪。

4.3.6 事件

事件也是 VxWorks 支持的通信方式,可以用于任务之间或者任务与 ISR,或者任务与 VxWorks 系统(如队列和信号量等)之间。任务最多可以注册 24 个事件,每个任务的 TCB 中有专门的事件寄存器(32 位):其中高 8 位系统保留,低 24 位用于记录用户的事件信息。但是此寄存器不可以直接访问,有事件发生变化的时候会改变相应的寄存器值。

跟事件相关的系统函数有以下几个。

(1) **eventReceive()**:接收事件

函数原型为:

```

STATUS eventReceive
(
    UINT32    events,          /* 等待的事件号 */
    UINT8     options,        /* 可选项 */
    int       timeout,         /* 等待的事件 */
    UINT32 *   pEventsReceived/* 反映接收到的事件变量 */
)

```

其中 events 为等待的事件号,options 可选项为:

- EVENTS_WAIT_ANY (0x1):等待任何一个事件发生了就继续;
- EVENTS_WAIT_ALL (0x0):等待所有的事件都发生了再继续;
- EVENTS_RETURN_ALL (0x2):返回所有等待的和不等待的事件;
- EVENTS_FETCH (0x80)pEventsReceived 变量设置已经接收到的事件相应位,并且立刻返回。

(2) **eventSend()**:发送事件

函数原型为:

```
STATUS eventSend
(
    int      taskId,          /* 要发送的任务 */
    UINT32 events             /* 要发送的事件 */
)
```

(3) **eventClear()**:清除当前任务的所有事件

函数原型为:

```
STATUS eventClear(void)
```

(4) **semGive()**:将事件拷贝到在该信号量注册的事件寄存器

(5) **msgQSend()**:将事件拷贝到在该队列注册的事件寄存器

例 4-7 事件应用的实例。

```
#include "VxWorks.h"
#include "semLib.h"
#include "eventLib.h"
#include "stdio.h"

/* 声明两个任务函数 */
STATUS task1(void);
STATUS task2(void);
/* 声明变量 */
int taskId1,taskId2;
SEM_ID semId1; /* 促使事件发生的资源 */
/* 初始化任务 */
int initTask()
{
    taskId1 = taskSpawn("task1", 120, 0, 10240, (FUNCPTR)task1, 0,0,0,0,0,
0,0,0,0,0);
    taskId2 = taskSpawn("task2", 110, 0, 10240, (FUNCPTR)task2, 0,0,0,0,0,
0,0,0,0,0);
    semId1 = semBCreate(SEM_Q_PRIORITY,SEM_EMPTY);/* 创建二进制信号量 */
    if(semId1 == NULL)
    {
        return ERROR;
    }
    return OK;
}
```

```

    }

    STATUS task1(void)
    {
        UINT32 tmp_receive;
        semEvStart(semId1, VXEVO1, EVENTS_OPTIONS_NONE); /* 任务1在资源:二进制信号量 semId1 上注册,如果资源发生变化,则任务会收到相应的事件 */
        while(1)
        {
            eventReceive(VXEVO1, EVENTS_WAIT_ANY, WAIT_FOREVER, &tmp_receive);
            /* 等待1号事件发生 */
            if((tmp_receive & VXEVO1) != 0)
            {
                printf("The semaphore become useful! \n"); /* 如果收到1号事件,则打印信息出来 */
            }
        }
    }

    STATUS task2(void)
    {
        while(1)
        {
            semGive(semId1);
            semTake(semId1, NO_WAIT);
            taskDelay(10);
        }
    }

```

在上面的例子中,task1 在信号量 semId1 上注册事件,同时等待事件;task2 使信号量的可用状态发生变化。所以 task1 可以收到相应的事件,然后作相应的处理。

4.3.7 共享内存

VxWorks 中的任务之间还可以通过共享内存来实现通信,共享的对象可以是单变量、数组、链表等等。但是要特别注意的是对共享的内存对象操作时,一定要注意保护,否则会发生意想不到的结果。由于共享内存的使用比较简单,这里就不举例了。

4.4 内存管理

4.4.1 概述

内存的概念是相对于外存而言的。外存通常是指 FLASH、ROM、NVRAM 等操作系统不经常访问的,存放程序和启动代码的存储器。在实时系统中,绝大多数的指令和数据操作都是在 RAM 中进行的,这就是所谓的内存。内存的管理是非常重要的,也是必不可少的。嵌入式开发者必须非常清楚内存的使用才能有效地进行内存管理。从另一方面讲,内存的费用也相当的高,如何合理地、最大限度地利用内存也是必须要考虑的问题。

嵌入式系统中的内存管理主要有以下几个功能。

(1) 分配可用内存

操作系统对于用户或者任务的内存需要,采取什么样的策略进行内存分配是关系到是否能够有效地利用内存的关键问题。操作系统必须采取有效的办法使内存得到最大限度的利用。

在 VxWorks 中,应该尽量地使用静态内存。这种预处理可以加快系统的响应速度,还可以对内存的使用有良好的估计。否则动态分配内存太多很容易出现内存碎片太多,影响系统性能,另外也不容易计算内存的使用量。

要合理地分配内存,就必须有效地管理内存。对于内存中的空闲内存块一般可以通过表格或者链表的形式进行管理。表格管理速度快,但是需要占用一部分内存,并且表格大小不好确定。链表容易删除和增加,VxWorks 中使用链表进行内存管理。

内存分配算法也有很多,主要有以下几种。

① 最先适应算法(First Fit)

最先适应算法要求内存块按照地址递增的顺序排列,在 VxWorks 中使用的就是此类算法,在这种算法中,一旦找到可以满足用户需求的内存块就停止搜索。这种算法具有有效的利用地址空间、性能好等特点,不需要变动内存块位置。

② 最佳适应法(Best Fit)

最佳适应法需要内存块按照内存块从小到大的次序排列,一旦找到适合用户大小的内存就进行分配,停止内存块搜索。此类算法内存使用率比较高。

③ 最坏适应法(Worst Fit)

最坏适应法需要内存块按照从大到小的次序进行排列,如果第一个内存块不能满足需要,则分配失败。

(2) 回收释放的内存

将释放的内存进行回收,重新按照某种算法进行排列。

(3) 共享内存

因为内存是系统中最紧张的资源,所以很多时候任务之间的通信通过共享内存进行,这样可以大大地节省空间,减少系统开支。

(4) 对内存中的程序和代码进行保护

在内存中,有各种类型的应用对象:堆栈、程序代码、操作系统等等,要防止各种数据遭到非法使用或者破坏,所以要对内存中的程序和代码,尤其是临界资源进行保护。

一般来说,内存管理必须满足以下几个要求^[8]。

(1) 实时性

从实时性的角度出发,要求内存分配过程要尽可能地快。因此,在嵌入式系统中,不可能采用通用操作系统的一些复杂而完善的内存分配策略,一般没有段页式的虚存管理机制;而是采用简单、快速的内存分配方案,其分配方案也因程序对实时性的要求而异。例如,VxWorks 系统采用简单的“首次适应,立即聚合”方法。

(2) 可靠性

嵌入式系统应用的环境千变万化。在有些特定情况下,对系统的可靠性要求极高,内存分配的请求必须得到满足,如果分配失败则可能会带来灾难性的后果。比如,飞机的燃油检测系统。在飞机飞行过程中,如果燃料发生泄漏,系统应该立即检测到,并发出相应的警报,等待飞行员及时处理。如果因为内存分配失败而不能进行相应的操作,就可能发生机毁人亡的事故。

(3) 高效性

内存分配要尽可能地减少浪费。不可能为了保证满足所有的内存分配请求而将内存配置得很大。一方面,嵌入式系统对成本的要求使得内存存在其中只是一种很有限的资源;另一方面,即使不考虑成本的因素,系统硬件环境有限的空间和有限的印刷电路板面积也决定了可配置的内存容量是很有限的。

针对以上 3 个约束条件,市场上主流的嵌入式操作系统,如 VxWorks、嵌入式 Linux 等均提出了一套有效的解决方案。一般的嵌入式系统中最基本的内存管理方案有两种:静态分配和动态分配。

静态分配是指在编译或链接时将程序所需的内存空间分配好。采用这种分配方案的程序段,其大小一般在编译时就能够确定;而动态分配是指系统运行时根据需要动态地分配内存。这两种策略的选取一直是嵌入式系统设计中一个令人头痛的问题。

一般的嵌入式系统都支持静态分配,因为像中断向量表、操作系统映像这类的程序段,其程序大小在编译和链接时是可以确定的。而是否支持动态分配主要基于两个方面的考虑:首先是实时性和可靠性的要求,其次是成本的要求。对于实时性和可靠性要求极高的系统(硬实时系统),不允许延时或者分配失效,必须采用静态内存分配,如航天器上的嵌入式系统多采用静态内存分配。除了基于成本的考虑外,用于汽车电子和工业自动化领域的一些系统也没有动态内存分配,比如 Wind River 著名的 OSEKWorks 系统。然

而,仅仅采用静态分配,使系统失去了灵活性。必须在设计阶段就预先知道所需要的内存并对之作出分配;必须在设计阶段就预先考虑到所有可能的情况,因为一旦出现没有考虑到的情况,正在运行的系统就无法处理。这样的分配方案必然导致很大的浪费。因为内存分配必须按照最坏情况进行最大的配置,而实际运行时很可能只使用其中的一小部分;而且在硬件平台不变的情况下,不可能灵活地为系统添加功能,从而使得系统的升级变得困难。

虽然动态内存分配会导致响应和执行时间不确定、内存碎片等问题,但是它的实现机制灵活,为程序实现带来了极大的方便,有的应用环境中动态内存分配甚至是必不可少的。比如,嵌入式系统中使用的网络协议栈。在特定的平台下,为了比较灵活地调整系统的功能,在系统中各个功能之间作出权衡,必须支持动态内存分配。例如,为了使系统能够及时地在支持的 VLAN 数和支持的路由条目数之间作出调整,或者为了使不同的版本支持不同的协议,类似于 malloc 和 free 这类的函数是必不可少的。

大多数的系统是硬实时和软实时综合。系统中的一部分任务有严格的时限要求,而另一部分只是要求完成得越快越好。按照 RMS(Rate Monotonous Scheduling)理论,这样的系统必须采用抢先式任务调度。而在这样的系统中,就可以采用动态内存分配来满足部分对可靠性和实时性要求不高的任务。采用动态内存分配的最大好处就是给设计带来很大的灵活性,可以方便地将原来运行于非嵌入式操作系统的程序移植到嵌入式系统中。

4.4.2 VxWorks 中的内存布局

VxWorks 中的内存分为低端内存区、VxWorks 区、Target Server 内存池、系统内存池和用户保留区等,如图 4-6 所示。

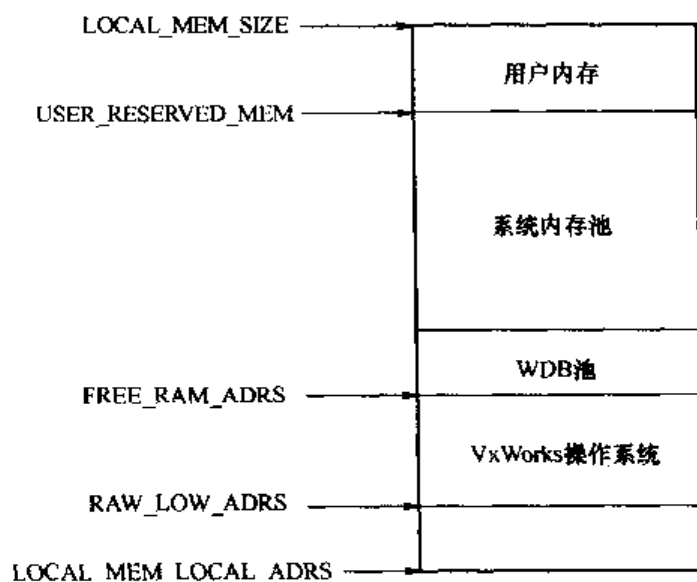


图 4-6 VxWorks 内存的典型布局

下面进行详细介绍。

(1) 低端内存区

低端内存区通常是系统的中断向量表、引导配置等。LOCAL_MEM_LOCAL_ADR 是低端内存区的起始地址。

(2) VxWorks 区

VxWorks 区是存放 VxWorks 操作系统映像的区域。

(3) 系统内存池

系统内存池用于堆的分配、任务的 TCB 控制块、临时变量等。

(4) Target Server 内存池

Target Server 内存池用于动态下载目标模块,主要用于调试使用,由 Target Server 管理。

(5) 用户保留区

用户保留区主要用于用户特定服务的区域。

4.4.3 系统函数介绍

1. 内存区的申请和释放函数介绍

(1) malloc(): 动态申请内存

函数原型如下:

```
void * malloc
(
    size_t nBytes           /* 分配的字节数 */
)
```

(2) free(): 释放内存

函数原型如下:

```
void free
(
    void * ptr              /* 要释放的内存指针 */
)
```

(3) memPartFree(): 从指定内存分区中释放内存

函数原型如下:

```
STATUS memPartFree
(
    PART_ID partId,         /* 要释放的内存分区号 */
    char * pBlock           /* 要释放的内存指针 */
)
```

(4) **memPartAlloc()**:从指定内存分区中分配内存

函数原型如下:

```
void * memPartAlloc
(
    PART_ID partId,          /* 要分配的分区号 */
    unsigned nBytes          /* 要分配的字节数 */
)
```

(5) **memPartAlignedAlloc()**:从指定内存分区中分配内存,分配的内存是 2 的幂次方

函数原型如下:

```
void * memPartAlignedAlloc
(
    PART_ID partId,          /* 要分配的内存分区 */
    unsigned nBytes,         /* 分配的内存大小 */
    unsigned alignment       /* 排列的边界 */
)
```

2. 内存分区函数介绍

(1) **memPartAddToPool()**:从指定的内存分区中分配内存

函数原型如下:

```
STATUS memPartAddToPool
(
    PART_ID partId,          /* 指定的分区 */
    char * pPool,           /* 返回的内存块指针 */
    unsigned poolSize        /* 块大小 */
)
```

(2) **memPartCreate()**:创建内存分区

函数原型如下:

```
void * memPartAlloc
(
    PART_ID partId,          /* 分配的分区号 */
    unsigned nBytes          /* 要分配的分区字节数 */
)
```

4.4.4 虚拟内存

在基于段页式的内存管理体系结构中,必然要包含内存管理单元(Memory Management

Unit,MMU)功能,可以将虚拟地址翻译成实际的物理地址从而实现了内存的控制访问。

VxWorks 提供了两种虚拟内存的实现接口,一种是 VxWorks 绑定的虚拟内存接口,一种是由 VxVMI 组件提供的虚拟内存接口,可以实现 MMU 功能。在 VxWorks 中如果实现虚拟内存功能,需要在 BSP 中或者 Tornado 中进行相应的配置或者在 tornado→VxWorks→hardware→memory→MMU 中进行配置。当然,VxVMI 组件不是 VxWorks 默认支持的组件,需要用户自行购买,它所支持的虚拟内存功能要更加强大,支持内存访问保护等功能。

4.4.5 高速缓存 Cache

Cache 是一种缓冲机制,是为了系统更快速地工作,匹配 CPU 的工作频率和内存读写频率的巨大差别。VxWorks 在系统加电的时候是禁止 Cache 的。

VxWorks 操作系统是支持 Cache 操作的,但要特别注意在 Cache 中的数据 and 内存中的数据之间的一致性问题。防止内存已经更新了,而 Cache 中数据还没有更新,出现不一致的情况。对 Cache 的操作主要有 flush 和 invalidate 两种:冲洗缓冲区和使缓冲区无效。当 CPU 向外设发送数据时,执行 flush 功能,可以使 Cache 和 RAM 中的数据保持一致;当从外设读取数据时,执行 invalidate 功能,可以使 Cache 中的数据得到更新,和 RAM 保持一致。

4.4.6 常用的内存分析工具

Browser 中有很多种可以管理查看内存的工具,如图 4-7 所示的 Stack Check 选项,还有 Memory Usage、Module Information、Target Information、Object Information 等。

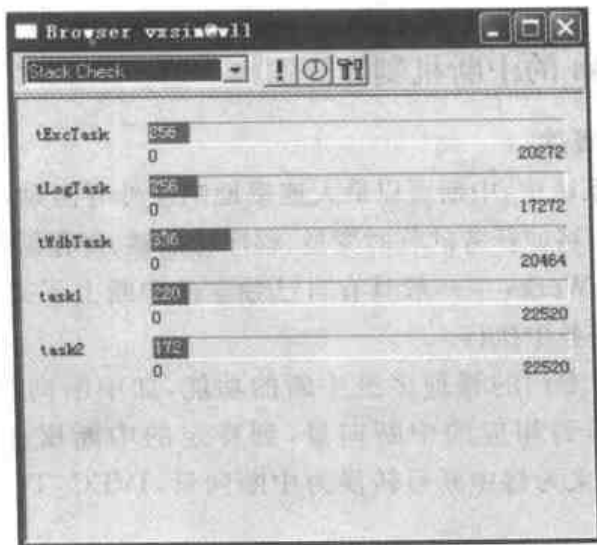
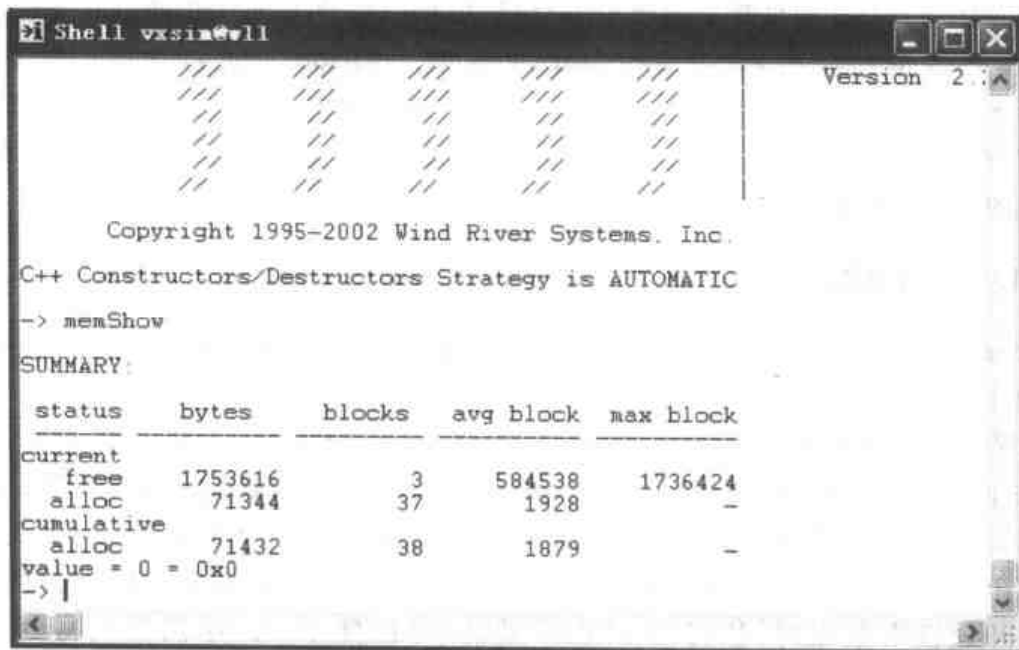


图 4-7 Browser 内存管理工具

图 4-8 为 Shell 中查看内存使用情况,及时地分析内存使用状况可以更好地了解系统的性能和运行情况,便于找出问题。



```

Shell vxsim@vll
Version 2.1

Copyright 1995-2002 Wind River Systems, Inc.
C++ Constructors/Destructors Strategy is AUTOMATIC
-> memShow
SUMMARY:

```

status	bytes	blocks	avg block	max block
current				
free	1753616	3	584538	1736424
alloc	71344	37	1928	-
cumulative				
alloc	71432	38	1879	-
value = 0 = 0x0				

```

-> |

```

图 4-8 Shell 中显示内存情况

4.5 中断和异常

4.5.1 VxWorks 的中断机制

1. VxWorks 中断概述

在 VxWorks 实时系统中,中断可以最大速率地响应外界激励,并且提供服务。中断执行的时候会保护好现场,将原任务的临时变量、程序计数器、寄存器等压到堆栈里,等到中断执行完成后再恢复。VxWorks 中一般具有自己独立的中断上下文。当然如果体系结构不支持,中断上下文将在任务中执行。

在 VxWorks 中,有专门的管理多级中断的功能,如中断向量和中断号的互相转换等。通过中断号可以得到相应的中断向量,到特定的中断服务程序中去执行。其中 INUM_TO_IVEC 宏定义为将中断号转换为中断向量,IVEC_TO_INUM 宏定义将中断向量转换为中断号。

在 VxWorks 中断服务程序中,有些操作是禁止或者不提倡的。如申请某些可能会引起阻塞的资源,中断服务程序过于冗长等。这些操作有可能造成系统瘫痪。另外,在中

断程序中还有必要进行中断保护,防止中断执行的时候出现异常情况。

2. VxWorks 的中断处理函数

VxWorks 中关于中断的处理函数是非常多的,下面进行详细地说明。

(1) `intLock()`:禁止中断

函数原型为:

```
int intLock(void)
```

此函数可以禁止其他中断打断正在执行的程序。

(2) `intUnlock()`:允许中断

函数原型为:

```
int intUnlock(void)
```

和 `intLock` 函数配套使用,允许中断。

(3) `intConnect()`:将中断向量与特定的服务程序绑定

函数原型为:

```
STATUS intConnect
(
    VOIDFUNCPTR * vector,      /* 要绑定的中断向量 */
    VOIDFUNCPTR routine,      /* 要绑定的中断服务程序入口 */
    int parameter             /* 传递到中断服务程序中的参数 */
)
```

成功执行后返回 OK,否则返回 ERROR。

例 4-8 将中断向量与特定服务程序绑定的实例。

```
void sysHwInit2(void)
{
    ...
    /* connect generic VME interrupts handler */
    intConnect(INT_VEC_VME, sysVmeIntr, NULL);
    ...
}
```

(4) `intContext()`:检查当前的状态是在中断上下文中还是在任务上下文中

函数原型为:

```
BOOL intContext(void)
```

(5) `intCount()`:得到中断的嵌套深度

函数原型为:

```
int intCount(void)
```

4.5.2 VxWorks 的异常机制

1. VxWorks 异常概述

异常和中断的概念非常相似。从某种程度来讲,异常和中断的概念一样的,都是运行正常的程序时被打断,转而去执行特殊处理程序的过程。异常通常是指由于程序内部的非法访问或者错误处理导致发生特殊处理过程,如内部程序的除零、数据溢出等。中断一般由外界因素激发,如外设发出中断请求等等。在 VxWorks 中对于异常的处理和中断很相似,有特定的异常处理程序。可以默认使用系统自身的程序,也可以使用用户自己的程序作为异常处理程序。

2. VxWorks 异常处理函数

在 VxWorks 中,对于异常的处理函数主要有以下几个。

(1) **excVecInit()**:初始化中断、异常向量

函数原型为:

```
STATUS excVecInit (void)
```

(2) **excConnect()**:将 C 程序与异常向量绑定

函数原型为:

```
STATUS excConnect
```

```
(
```

```
    VOIDFUNCPTR * vector,          /* 要绑定的异常向量 */
```

```
    VOIDFUNCPTR routine           /* 要绑定的异常服务程序 */
```

```
)
```

(3) **excIntConnect()**:将 C 程序与异步异常向量绑定

函数原型为:

```
STATUS excIntConnect
```

```
(
```

```
    VOIDFUNCPTR * vector,          /* 要绑定的异常向量 */
```

```
    VOIDFUNCPTR routine           /* 要绑定的程序 */
```

```
)
```

(4) **excCrtConnect()**:将 C 程序与关键的异常向量绑定

函数原型为:

```
STATUS excCrtConnect
```

```
(
```

```
    VOIDFUNCPTR * vector,          /* 要绑定的异常向量 */
```

```
    VOIDFUNCPTR routine           /* 要绑定的程序 */
```

```
)
```

(5) **excVecSet()**: 设置一个 CPU 异常向量

函数原型为:

```
void excVecSet
(
    FUNCPTR * vector,           /* vector offset */
    FUNCPTR  function           /* address to place in vector */
)
```

(6) **excVecGet()**: 获得一个 CPU 异常向量

函数原型为:

```
FUNCPTR excVecGet
(
    FUNCPTR * vector           /* vector offset */
)
```

4.6 时钟和定时机制

4.6.1 VxWorks 操作系统的时钟

在 VxWorks 实时系统中, 进行规定时间的操作是很常见的。比如周期性执行某个任务或者在某个时间点进行某项规定的操作, 这就需要有非常精确的定时器作为时间基准。在 VxWorks 内部, 任务之间的调度运行激励也需要有非常精确的定时, 内核周期性地执行时钟中断。在 VxWorks 操作系统中, 提供两种基于硬件的定时时钟: 系统时钟 (System Clock) 和系统辅助时钟 (System Auxiliary Clock)。当然, VxWorks 中也支持 ANSI C 和 POSIX 标准接口的时钟机制。

1. 系统时钟

主要的相关函数有以下几个。

(1) **sysClkConnect()**: 绑定时钟中断程序

函数原型为:

```
STATUS sysClkConnect
(
    FUNCPTR routine,           /* 要绑定的时钟处理程序 */
    int    arg                 /* 处理程序的参数 */
)
```

(2) **sysClkDisable()**: 关闭系统中断

函数原型为:

```
void sysClkDisable (void)
```

(3) **sysClkEnable()**: 打开系统中断

函数原型为:

```
void sysClkEnable (void)
```

(4) **sysClkRateGet()**: 获得系统时钟频率

函数原型为:

```
void sysClkRateGet(void)
```

(5) **sysClkRateSet()**: 设置系统时钟频率

函数原型为:

```
STATUS sysClkRateSet
```

```
(  
    int ticksPerSecond          /* 每秒的时钟中断数或者 tick 数 */  
)
```

2. 系统辅助时钟

辅助时钟,顾名思义,就是对于系统时钟的辅助和补充,它在实时系统中也经常被使用。当需要较高的时钟频率时,由于系统时钟还有很多其他的用途,如为系统调度提供基准等,因此其频率不能够设置得太高,这时可以使用辅助时钟。辅助时钟也是基于硬件定时器的,所以比较精确。辅助时钟如果可用则系统时钟也必须可用。辅助时钟在其他一些参考书上被称为主时钟的“影子”,形象地说明了它的作用。类似的,跟辅助时钟相关的函数主要有以下几个。

(1) **sysAuxClkConnect()**: 绑定辅助时钟中断处理程序

函数原型为:

```
STATUS sysAuxClkConnect
```

```
(  
    FUNCPTR routine,          /* 要绑定的时钟处理程序 */  
    int    arg                /* 处理程序的参数 */  
)
```

(2) **sysAuxClkDisable()**: 关闭辅助时钟中断

函数原型为:

```
void sysAuxClkDisable (void)
```

(3) **sysAuxClkEnable()**: 打开辅助时钟中断

函数原型为:

```
void sysAuxClkEnable (void)
```

(4) **sysAuxClkRateGet()**: 获取辅助时钟中断频率

函数原型为:

```
void sysAuxClkRateGet(void)
```

(5) **sysAuxClkRateSet()**: 设置辅助时钟中断频率

函数原型为:

```
STATUS sysAuxClkRateSet
```

```
(
    int ticksPerSecond           /* 每秒的时钟中断数或者 tick 数 */
)
```

从上面的辅助时钟系统函数可以看出,它们和系统时钟函数非常相似,是对系统时钟很好的补充和辅助。

4.6.2 VxWorks 操作系统的定时机制

正如 4.6.1 小节所介绍的,VxWorks 操作系统的系统时钟和辅助时钟起很好的定时作用。另外还需要介绍一种很重要的定时机制:看门狗定时。看门狗定时在实时系统中非常常见,它不仅可以定时,更重要的是可以起到监控整个软件系统运行、发现异常情况进行软复位的作用,很像“看门狗”的作用。

主要的相关函数有以下几个。

(1) **wdCreate()**: 创建一个看门狗定时器

函数原型如下:

```
WDOG_ID wdCreate(void)
```

(2) **wdDelete()**: 删除看门狗定时器

函数原型如下:

```
STATUS wdDelete
```

```
(
    WDOG_ID wdId                /* 要删除的看门狗的 ID */
)
```

(3) **wdStart()**: 启动看门狗定时器

函数原型如下:

```
STATUS wdStart
```

```
(
    WDOG_ID wdId,                /* 看门狗 ID */
    int delay,                   /* 延迟时间 */
    FUNCPTR pRoutine,            /* 超时以后调用的函数 */
    int parameter                /* 传递给调用函数的参数 */
)
```

)

(4) **wdCancel()**: 停止当前的看门狗定时器计数

函数原型如下:

STATUS wdCancel

(

WDOG_ID wdId /* 看门狗的 ID */

)

例 4-9 看门狗计时的实例。

```
#include "VxWorks.h"
#include "wdLib.h"
#include "stdio.h"
/* 声明函数 */
STATUS task1(void);
void WDOG_ISR(void);
int taskId1; /* 任务 ID */
int i;
WDOG_ID wdog_test; /* 看门狗变量 */

int initTask()
{
    taskId1 = taskSpawn("task1", 120, 0, 10240, (FUNCPTR)task1, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0);
    wdog_test = wdCreate(); /* 创建看门狗 */
    i = 0;
    if(wdog_test == NULL)
    {
        return ERROR;
    }
    if(wdStart(wdog_test, 50, (FUNCPTR)WDOG_ISR, 0) == ERROR) /* 启动看门狗 */
    {
        return ERROR;
    }
    return OK;
}

/* 任务执行函数 */
```



```

STATUS task1(void)
{
    while(1)
    {
        if(i == 1)
        {
            i = 0;
            wdStart(wdog_test,50,(FUNCPTR)WDOG_ISR,0);/*再启动看门狗函数*/
        }
    }
}

void WDOG_ISR(void)
{
    logMsg("WDOG is beginning! \n",0,0,0,0,0,0);/*进入看门狗处理程序后打印消息*/
    i = 1;
}

```

在上面的例子中,首先创建了一个看门狗。当看门狗计时到了以后,进入看门狗服务程序,打印一条消息出来,表示进行了相应的操作处理,然后将变量 *i* 的值赋为 1。task1 一直在监测变量 *i* 的值,如果 *i* 的值变为 1 了,则重新启动看门狗,并且将 *i* 的值赋为 0。

4.7 I/O 系统

4.7.1 I/O 系统概述

I/O 系统全称为(Input/Output)输入输出系统。在 VxWorks 中。I/O 系统向用户屏蔽了硬件层,为用户提供了一个统一的标准接口。对于应用层开发者来说,只要了解了 I/O 系统的标准接口使用方法,就可以正确地操作外部设备。但对于底层驱动工程师而言,必须要全面地了解 I/O 系统,才能够更好地为上层应用开发提供良好的支持。

如果从数据的流向来讲,I/O 系统可以分为 3 个层次:面向用户的 I/O 接口层、硬件驱动层和具体硬件。这 3 个层次一起构成了 I/O 系统。

VxWorks 支持的 I/O 设备基本上可以分为字符设备和块设备。字符设备主要指以字符为单位进行数据操作的设备,如串口等;块设备是以大批量的“块”为单位进行数据操作的设备,如网络设备、存储设备等。字符设备直接通过 I/O 接口层和应用层进行交互,

而块设备则要通过文件系统与 I/O 接口层进行交互。

VxWorks 提供 3 种类型的 I/O 接口:基本 I/O 接口、带缓冲的 I/O 接口和格式化 I/O 接口。

4.7.2 3 种 I/O 接口介绍

1. 基本 I/O 接口

在 VxWorks 中,设备都是按照文件来操作的,打开一个设备需要知道它的文件名。当成功地打开某个设备时,会返回一个文件描述符。文件描述符是一个全局的句柄。成功得到某个设备的文件描述符后,对该设备的操作都将针对这个文件描述符进行。

在 VxWorks 中,文件描述符的资源是有限的,应该尽量节约使用。有一部分文件描述符是系统保留的,如:

- 0——标准输入;
- 1——标准输出;
- 2——标准错误。

VxWorks 中可以对文件描述符进行重定向使用,但是要特别注意不要将文件描述符混淆,从而导致设备的访问出现混乱。

基本的 I/O 函数有以下几个。

(1) **creat()**:创建一个文件

函数原型为:

```
int creat
(
    const char * name,          /* 创建的文件名 */
    int         flag            /* O_RDONLY, O_WRONLY, or O_RDWR */
)
```

creat()函数在指定的设备上创建一个文件并且打开这个文件,创建成功后则返回文件描述符。其中 flag 选项可以为 O_RDONLY(只读操作), O_WRONLY(只写操作),或 O_RDWR(可读可写操作)。

(2) **open()**:打开一个文件

函数原型为:

```
int open
(
    const char * name,          /* 要打开的文件名 */
    int         flag,          /* O_RDONLY, O_WRONLY, O_RDWR, or
                                O_CREAT */
    int         mode            /* 文件模式 */
)
```

通常,用 `open()` 函数打开的是已经存在的文件,对于 NFS 文件,需要配置 `mode` 参数。其中 `flag` 选项可以为 `O_RDONLY`(只读操作), `O_WRONLY`(只写操作), `O_RDWR`(可读可写操作),或 `CREAT`(创建文件)。例如:

```
fd = open("/usr/myFile", O_CREAT | O_RDWR, 0644)
```

这个例子创建了一个 UNIX 文件模式的文件,返回正确的文件描述符。

(3) **remove()**:删除一个文件

函数原型为:

```
STATUS remove
```

```
(
    const char * name           /* 要删除的文件名 */
)
```

删除一个文件,如上面的 `remove("/usr/myFile")`;

(4) **close()**:关闭一个文件

函数原型为:

```
STATUS close
```

```
(
    int fd                     /* 要关闭的文件描述符 */
)
```

如 `close(fd)`,成功操作后返回 OK。

(5) **read()**:从文件或者设备上读数据

函数原型为:

```
int read
(
    int    fd,                /* 要读的文件描述符 */
    char * buffer,            /* 接收数据的 buffer 指针 */
    size_t maxbytes           /* 读到缓冲中的最大字节数 */
)
```

如果文件描述符不存在、没有读操作的驱动函数、设备读操作返回错误都会返回 ERROR,否则返回所读的字节数。

(6) **write()**:向一个文件写数据

函数原型为:

```
int write
(
    int    fd,                /* 要写的文件描述符 */

```

```

    char * buffer,           /* 要输出的数据缓冲指针 */
    size_t nbytes           /* 要写的字节数 */
)

```

如果文件描述符不存在、没有写操作的驱动函数、设备写操作返回错误都会返回 ERROR, 否则返回所写入的字节数。

(7) **ioctl()**: 执行 I/O 控制函数

函数原型为:

```

int ioctl
(
    int fd,                 /* 文件描述符 */
    int function,           /* 函数代码 */
    int arg                 /* 传入的参数 */
)

```

例如下面所示, 用“newname”对某个目录文件进行了重命名:

```
ioctl (fd, FIORENAME, "newname")
```

例 4-10 基本 I/O 使用举例。

```

#include "VxWorks.h"
#include "selectLib.h"
#include "taskLib.h"
#include "sysLib.h"
#include "ioLib.h"
#include "stdio.h"
.....

/* 声明函数变量 */
STATUS SerialTask(void);

/* 主函数 */
STATUS tyCoTest()
{
    /* 串口初始化 */
    .....

    if((sd = open("/tyCo/0", O_RDONLY, 0)) == ERROR)
    {
        return ERROR;
    }
}

```

```

sd = open("/tyCo/0", O_RDWR, 0); /* 打开串口 */
status = ioctl(sd, SIO_BAUD_SET, 9600); /* 设置波特率 */
status = ioctl(sd, FIOSETOPTIONS, OPT_RAW); /* 串口模式 */
status = ioctl(sd, FIOFLUSH, 0); /* 刷新串口收发缓冲 */
taskId = taskSpawn("SerialTask", 120, 0, 10240, (FUNCPTR)SerialTask, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0); /* 创建一个任务 */
return OK;
}

STATUS SerialTask(void)
{
.....
while(1)
{
    if ( ERROR == ( read(sd, c + length, 1) ) ) /* 接收一个字符 */
    {
        return ERROR;
    }
    .....
    write(sd, send_buffer[i], 10); /* 往指定的文件描述符里写数据 */
    .....
}
}

```

在上面的例子中,有对串口设备进行操作的基本函数,可以使读者更加直观地了解基本 I/O 的用法。

2. 带缓存的 I/O 接口

VxWorks 中除了支持基本的 I/O 接口外,还支持带缓存的 I/O 接口。基本的 I/O 接口在访问频率很高的情况下,性能是比较差的,这就需要有缓存机制。有了缓存机制后,可以大大提高系统访问速度。在 VxWorks 中缓存对于用户完全是透明的,不用关心。要使用 VxWorks 中的带缓存的 I/O 包,需要在配置文件中选择 INCLUDE_ANSI_STDIO 选项。缓存 I/O 使用文件指针代替文件描述符以实现对文件或者设备的访问,例如下面的例子:

```
FILE *fp = fopen("/mydocument", "r")
```

返回的是一个文件指针,得到文件指针后,就可以对文件进行操作,使用 fopen() 或者 fwrite() 进行文件读写。关闭文件后,其缓存将会被释放。但是需要注意的是,文件

缓存没有相应的保护机制,所以当不同的任务要访问相同的文件指针时要特别地注意,应该尽量避免这种做法。

相关的系统函数有以下几个。

(1) **fopen()**:打开文件

函数原型如下:

```
FILE * fopen
(
    const char * file,          /* 文件名 */
    const char * mode          /* 模式 */
)
```

(2) **fread()**:读文件

```
int fread
(
    void * buf,                /* 要保存的数据指针 */
    size_t size,              /* 成员大小 */
    size_t count,             /* 成员数 */
    FILE * fp                 /* 要读的文件指针 */
)
```

(3) **fwrite()**:写文件

```
int fwrite
(
    const void * buf,          /* 要输出的数据指针 */
    size_t size,              /* 成员大小 */
    size_t count,             /* 成员数 */
    FILE * fp                 /* 要写的文件指针 */
)
```

更多的内容可以参照帮助文档,以上为几个典型的函数。

3. 格式化 I/O 接口

在 VxWorks 中,格式化 I/O 属于 fioLib 库,它提供的是格式化的、无缓存的接口。其中典型的函数有以下几个。

(1) **printErr()**:将标准的信息流输出到标准的错误流中

原型函数如下:

```
Int printErr
(
    const char * fmt,          /* 要输出的格式流 */
    ...
)
```

```

...          /* 可选的格式化参数 */
)
(2) logMsg():输出格式化信息
原型函数如下:
int logMsg
(
    char * fmt,          /* 要打印的信息流格式 */
    int   arg1,          /* 下面是6个可打印的参数变量 */
    int   arg2,
    int   arg3,
    int   arg4,
    int   arg5,
    int   arg6
)

```

logMsg()函数在中断、异常服务程序中应用非常频繁,主要用于调试,输出变量信息。

4.7.3 I/O 系统的工作机制

正如前面所介绍的,I/O系统包括3个层次:I/O接口、驱动程序、I/O设备。I/O设备分为块设备和字符设备。字符设备可以和I/O接口直接交互,而块设备则需要通过文件系统进行交互。在介绍I/O系统的工作机制前,首先介绍几个基本概念。

1. I/O 系统的初始化

VxWorks 操作系统在加载的过程中,由根任务usrRoot调用iosInit()函数进行I/O设备的初始化。其中包括最多能安装的设备个数,同时能够打开的文件数、空设备的名称等。函数原型如下:

```

STATUS iosInit
(
    int    max_drivers,    /* 允许安装的最大设备数 */
    int    max_files,     /* 允许打开的最大文件数 */
    char * nullDevName     /* 空设备的名称 */
)

```

2. 驱动程序的初始化

驱动程序的初始化是由函数iosDrvInstall()完成的,将7个标准I/O函数加入到驱动表中,然后返回驱动号。这个函数的原型如下:

```
int iosDrvInstall
```

```
(
    FUNCPTR pCreate,           /* 创建设备函数指针 */
    FUNCPTR pDelete,          /* 删除设备函数指针 */
    FUNCPTR pOpen,            /* 打开设备函数指针 */
    FUNCPTR pClose,           /* 关闭设备函数指针 */
    FUNCPTR pRead,            /* 读设备函数指针 */
    FUNCPTR pWrite,           /* 写设备函数指针 */
    FUNCPTR pIoctl            /* I/O 设备控制函数指针 */
)
```

3. 设备的创建和访问

在 I/O 系统中,具有专门存储设备信息的设备链表。链表之间通过设备链表地址相互连接。每个表项叫做设备描述符,里面包括驱动号、设备名称、设备头等信息。相关的系统函数为:

(1) **iosDevAdd()**:增加设备到 I/O 系统中

函数原型为:

STATUS iosDevAdd

```
(
    DEV_HDR * pDevHdr,        /* 设备描述符指针 */
    char *    name,           /* 设备名 */
    int       drvnum          /* 服务的设备号 */
)
```

(2) **iosDevDelete()**:从 I/O 系统中删除设备

函数原型为:

void iosDevDelete

```
(
    DEV_HDR * pDevHdr        /* 设备头指针 */
)
```

需要注意的是,如果设备描述链表中没有此项设备,则此项操作会发生不可预料的结果。

(3) **iosDevFind()**:搜索某个设备

函数原型为:

DEV_HDR * iosDevFind

```
(
    char * name,              /* 设备名 */
    char * * pNameTail
)
```


)

如果搜索成功,则返回设备的设备头。

4. 文件描述符

文件描述符是文件的索引号,每个文件描述符中包括驱动号和设备描述符地址。

有了上述几个概念后,I/O 系统的工作过程也非常容易理解了。一般来说,由 VxWorks 操作系统首先完成 I/O 系统的初始化、驱动程序的安装和设备的安装。用户想访问 I/O 设备时,通过 open() 函数打开设备,由系统将相应的驱动号、设备描述符地址装入到文件描述符链表中,然后返回文件索引号,这样就可以方便地对设备文件进行操作了。当然,这里说的是字符设备,如果是块设备,则是将文件驱动函数列表装入到驱动表中。这些过程对于开发者是透明的,最终还是返回文件索引号。开发者只需要对文件索引号进行操作就可以了。

4.8 文件系统

4.8.1 文件系统概述

VxWorks 中的文件系统主要是为块设备的访问服务的。从某种角度考虑,文件系统可以认为是 I/O 系统的一部分。I/O 系统通过访问文件系统操作设备,文件系统和块设备之间采用标准的接口,使得文件系统可以方便地在驱动程序上进行移植。

4.8.2 VxWorks 文件系统介绍

1. dosFs 文件系统

dosFs 文件系统可以灵活地适应实时系统的需要,也是 VxWorks 操作系统中用得最多的一种文件系统。它的特点主要有:

- (1) 文件和目录分级,在同一个磁盘上可以创建任意多个文件;
- (2) 文件的组织方式可以是连续的也可以是不连续的;
- (3) 兼容 MS DOS 文件系统,支持各种文件系统格式;
- (4) 支持多种存储介质,如 磁带、Flash、CDROM 等;
- (5) 支持长文件名。

具体的使用函数可以见 dosFsLib,典型的使用函数有:

- dosFsLibInit()——准备使用 dosFs 文件系统;
- dosFsDevCreate()——创建 dosFs 文件的设备;
- dosFsShow()——显示 dosFs 文件系统的配置信息。

创建 dosFs 文件系统时,需要配置 CBIO。

2. rawFs 文件系统

rawFs 文件系统将整个磁盘作为一个文件进行管理,速度快,操作简单,对于文件和目录没有特别高级的管理。具体的操作函数见 rawFsLib 库,典型的有:

- rawFsDevInit() ——用 rawFs 文件系统初始化设备;
- rawFsInit() ——初始化 rawFs 文件系统。

3. rt11Fs 文件系统

rt11Fs 文件系统是为了支持 RT-11 文件系统而提出的,这在老的 VxWorks 版本中比较常见,新的 VxWorks 中已经用得差不多了。

4. tapeFs 文件系统

tapeFs 文件系统支持磁带存储设备,但不使用标准的文件和目录结构。跟 rawFs 文件系统一样,将整个磁带作为一个文件进行管理,磁带可以写入也可以读。对于磁带的文件和目录也没有一个更高级的管理,除非借助应用层进行管理。具体的操作见 tapeFsLib 库,典型的系统函数有:

- tapeFsDevInit() ——用 tapeFs 文件格式初始化指定的设备;
- tapeFsInit() ——初始化 tapeFs 文件卷库。

5. cdromFs 文件系统

cdromFs 文件系统支持 ISO 9660 标准格式的光盘文件,使用标准的 POSIX I/O 命令从光驱中读取文件。典型的系统函数如:

- cdromFsInit() ——初始化 cdromFs 文件系统;
- cdromFsVolConfigShow() ——显示卷文件信息;
- cdromFsDevCreate() ——创建一个 cdromLib 设备。

更多内容请参考 cdromFsLib 库。

第 5 章 嵌入式系统开发环境

本章介绍基于 VxWorks 的集成开发环境 Tornado, 以及基于 VxWorks 和 S3C4510B 芯片的 IRDX05v3 实验板。

5.1 Tornado 开发环境介绍

Tornado 是嵌入式实时领域里的开发调试环境之一, 它为嵌入式系统开发人员提供了一个不受目标机资源限制的超级开发和调试环境(如图 5-1 所示)。Tornado 包含 3 个高度集成的部分, 它们分别是:

- (1) 运行在宿主机和目标机上的强有力的交叉开发工具和实用程序;
- (2) 运行在目标机上的高性能、可裁剪的实时操作系统 VxWorks;
- (3) 连接宿主机和目标机的多种通信方式, 如: 以太网、串口线、仿真器等。

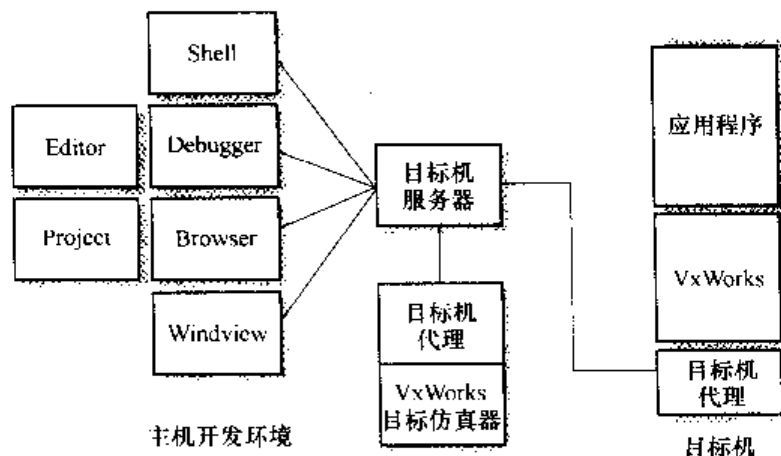


图 5-1 Tornado 主从开发架构

Tornado 是专门为解决嵌入式开发人员所面临的诸多问题而设计的,充分考虑到了开发者在进行嵌入式开发时所遇到的困难以及他们的需求。其一,对于不同的目标机,Tornado 为开发者提供了一个一致的图形接口和人机界面。当使用 Tornado 的开发人员转向新的目标机时,不必再花费时间学习或适应新的工具。其二,Tornado 所有的工具都是驻留在开发平台上的。在嵌入式系统工具发展历史上,Tornado 是第一个实现了当目标机资源有限时开发工具仍可使用而且功能齐全的开发环境。其三,所有工具都通过一个中央服务器(Target Server)处理与目标机的通信。所以无论连接方式是以太网、串口线还是仿真器,所有工具均可使用。

5.1.1 Tornado 核心工具

1. 集成仿真器 VxSim

VxSim 对于各种不同体系结构的主机来说是 VxWorks 的一个端口。VxSim 是一个原型仿真器,主要是使开发者在没有实际目标硬件的情况下,先进行原型机应用程序的开发,包括网络设计和基于多处理器的设计。在多数情况下,VxSim 模拟下的系统性能和运行在目标硬件上真实的 VxWorks 系统性能是一样的。开发者在 VxSim 中链接应用程序和编译 VxWorks 映像,这个过程和使用标准 BSP 的交叉开发环境是一样的。VxSim 可以使开发者在开发周期中较早地进行大部分的应用软件测试,使开发者能够以较小的代价纠正错误。

VxSim 环境下和真实的目标环境下的区别是:在 VxSim 中,映像文件在宿主机上执行,而真实环境下是在目标机中执行。通信机制允许 VxSim 获取 IP 地址,并且使用 VxWorks 网络工具和主机上的 Tornado 工具通信。

由于不可能和目标机的硬件进行交互,所以在 VxSim 中不适合开发设备驱动。然而,VxSim 中支持任务的交互以及优先级调度和抢占机制。

Tornado 工具中包含 VxSim 的基本功能,并且允许对仿真目标机的立即访问。集成的模拟器不包括网络部分,并且仅提供单实例使用。如果要使用 VxSim 完全功能的仿真,必须要购买风河公司的其他产品。

可以概括 VxSim 和 VxWorks 的最大区别有以下几点。

(1) 驱动

因为设备驱动要求直接的硬件交互,大多数 VxWorks 设备驱动在 VxSim 中无法使用。

(2) 文件系统

VxSim 默认使用 passFs 文件系统直接访问文件。大多数 VxWorks 目标机默认使

用 netDrv 访问主机上的文件。

(3) 网络

网络功能在基本产品上是不包含的,要使用全部的仿真必须购买组件。

下面介绍一下 VxSim 的使用。

首先建立一个 downloadable 工程(具体的工程类型选择可以参照本章 5.1.2 小节的内容),如图 5-2 所示。

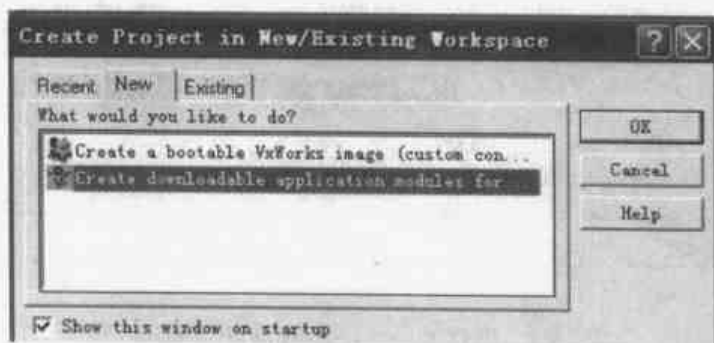


图 5-2 创建 downloadable 应用工程-1

然后选择工程的存放路径和名称,如图 5-3 所示。

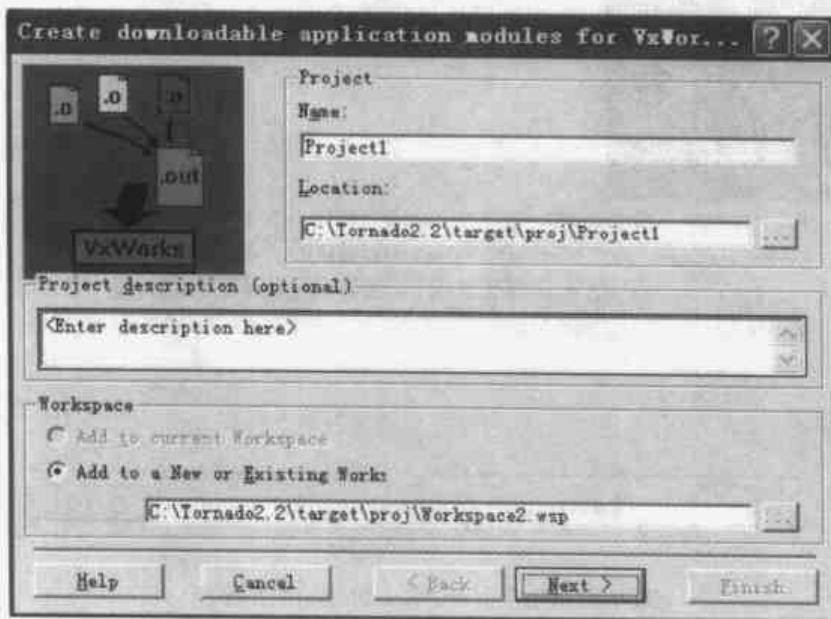


图 5-3 创建 downloadable 应用工程-2

选择模拟工具链 SIMNTgnu,如图 5-4 所示。



图 5-4 创建 downloadable 应用工程-3

创建好后,点击“Finish”按钮,如图 5-5 所示。

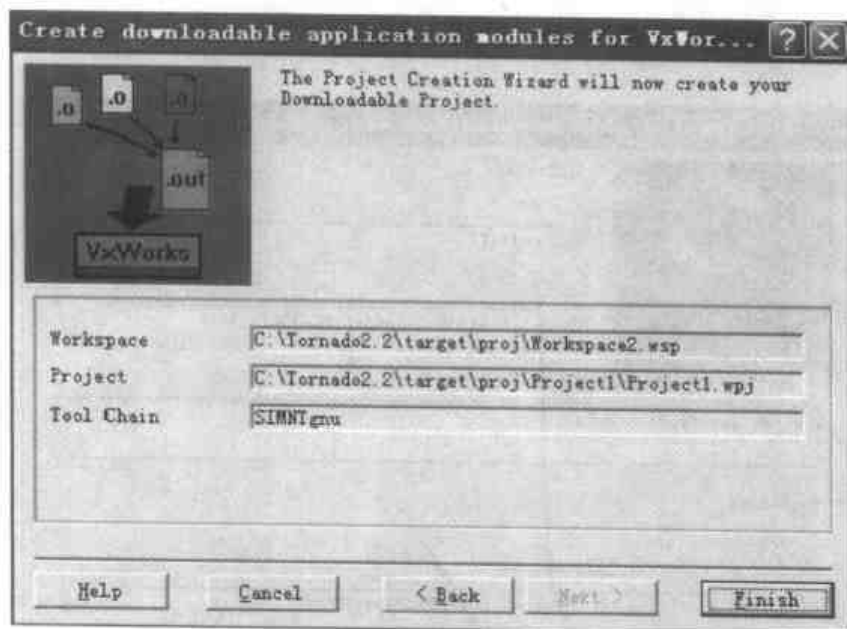


图 5-5 创建 downloadable 应用工程-4

按照上面的步骤就成功创建了 VxSim 环境下的工程。将程序编译好后,就可以下载映像文件到 VxSim 中。点击工具栏上的 VxSim 图标或者点击 Tools 菜单→Simulator 选项,如图 5-6 所示。

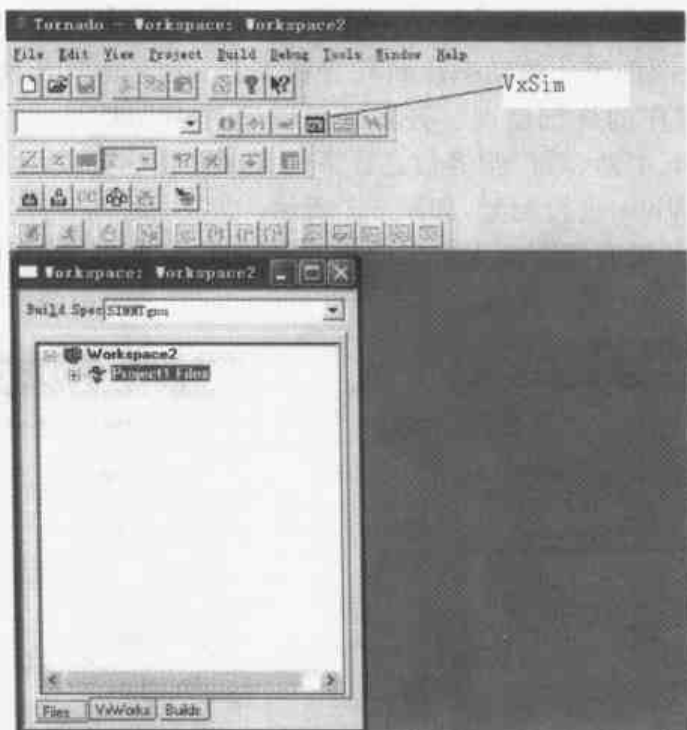


图 5-6 选择 VxSim 工具

点击 VxSim 工具后,可以看到如图 5-7 所示的对话框。



图 5-7 设置 VxSim 工具参数

然后点击“OK”,就可以进行应用程序的仿真模拟了。仿真器成功启动后,会在任务栏上显示红色的图标。下载程序等操作和在真实环境下的操作是一样的。

2. 软件逻辑分析仪 WindView

WindView 是一个图形化的动态诊断和分析工具,主要是向开发者提供目标机硬件上实际运行的应用程序的详细情况。开发者可以通过 WindView 看到目标机的任务切换、信号量、消息队列、中断、看门狗等信息。下面详细介绍它的使用方法。

首先要对 WindView 进行配置,如图 5-8 所示。

一般默认即可,如果有特殊要求可以选择“Properties”和“Reset”按钮进行详细配置,如图 5-9 所示。



图 5-8 设置 WindView 配置参数-1

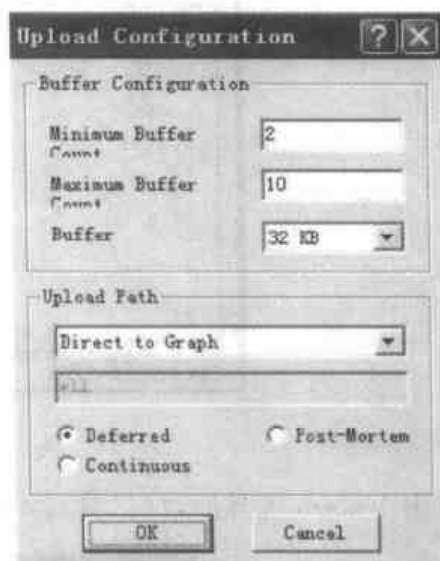


图 5-9 设置 WindView 配置参数-2

配置好 WindView 参数后,就可以开始使用了。它的启动方式也有两种,一种是直接在工具栏图标种选择,另一种是在 Tools→WindView→Launch 中选择。如图 5-10 所示。



图 5-10 启动 WindView

启动 WindView 之后,进入 WindView 的控制界面,如图 5-11 所示。

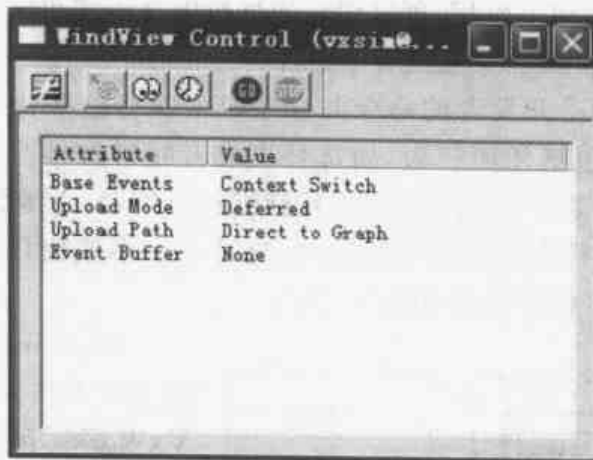


图 5-11 WindView 的控制界面

其中可以看到以前的几个基本设置参数,如基本事件类型、升级模式、升级路径等。点击“GO”按钮后,就进入了 WindView 的逻辑分析界面,如图 5-12 所示。

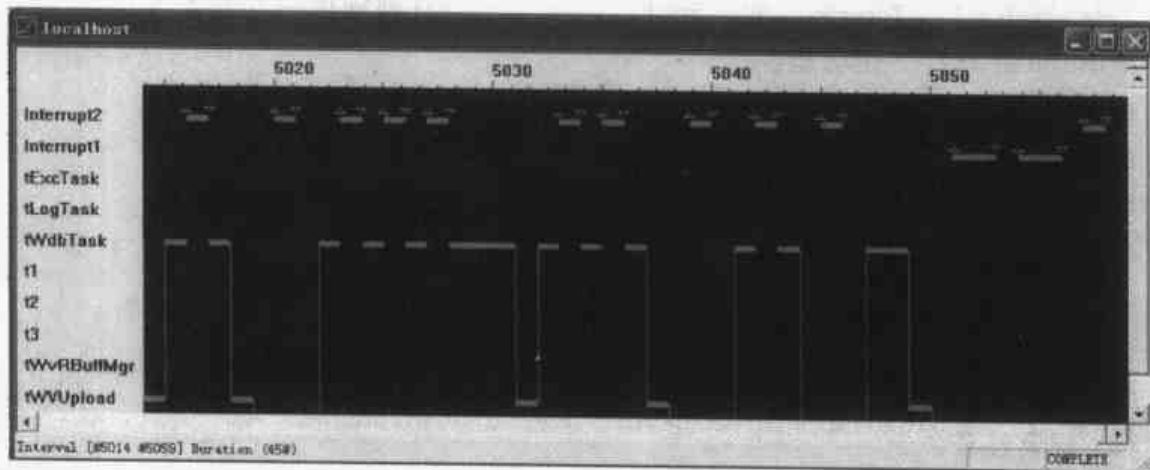


图 5-12 WindView 的逻辑分析界面

3. 命令执行工具 WindShell

WindShell 允许开发者下载应用模块、调用 VxWorks 和应用模块子程序。这项功能在许多场合都要使用,例如在 VxWorks 操作系统下的交互性开发、测试等。WindShell 能够解释大多数 C 语言表达式,能够解析和执行多数 C 操作符。开发者还可以通过 Tcl 解释器和 WindShell 交互,Tcl 解释器提供了一套完整的控制结构并能对目标设备进行更底层的访问。更详细的解释可以参考 Tcl 接口。

WindShell 在开发主机上执行,而不是在目标机上。但是它允许开发者启动任务读写目标设备,实现对目标机的完全控制。由于 WindShell 在开发主机上执行,开发者可以

最小程度地占用目标机资源。和其他的 Tornado 工具一样,仅仅要求目标机具有 Target Agent。因此,Shell 尽量保证可用,可以用它来进行项目测试和试验。

Shell 操作涉及到 Tornado 系统的 3 个组件:WindShell、Target Server 和 Target Agent。Shell 可以直接控制、接受开发者的命令并在主机上执行这些命令,发送请求给 Target Server。这些请求可以包括任何行为,如符号表或者驻留在目标机上的程序或者数据。Target Server 管理符号表并处理它和远端目标机的通信,发送函数调用并回送它们的结果。Target Agent 是运行在目标机上的组件,它是一个小监控程序,管理主机到目标机内存和其他工具的访问(如图 5-13 所示)。

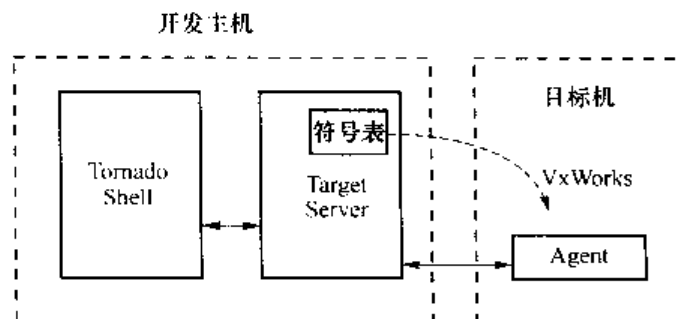


图 5-13 Tornado 和 WindShell

WindShell 具有双重角色,一方面它可以作为命令解释器,通过调用 VxWorks 函数提供对所有 VxWorks 功能的访问;另一方面它可以作为应用开发人员的原型和调试工具。开发人员可以调用任何应用函数来运行应用模块。WindShell 提供任何硬件异常的警告。

WindShell 具有下述功能:

- (1) 特定任务的断点;
- (2) 特定任务的单步执行;
- (3) 符号反汇编;
- (4) 任务和系统信息功能;
- (5) 调用用户函数的能力;
- (6) 创建和检查变量的能力;
- (7) 检查和调整内存的能力;
- (8) 异常捕获。

WindShell 的启动和前面几种工具相似,可以在菜单中选择,也可以在工具栏图标中选择,如图 5-14 所示。

选择好 Target 类型后,就可以点击“OK”按钮进入 Shell 界面,如图 5-15 所示。

无论用什么方法启动 WindShell,只要在 WindShell 中执行 `exit()` 或者 `quit()` 命令或者在键盘上按下 `CTRL+D` 就可以退出 WinShell 界面。WindShell 的功能上文已经介绍过。例如如果要从 Shell 中下载程序,在 Shell 命令行中输入 `ld(目标文件)` 就可以了。关于 Shell 中的命令,这里就不一一讲解了,更详细地可以参照 Tornado 帮助文件。

4. 目标机系统浏览器 Browser

Tornado 中的 Browser 工具可以方便地监视目标机的状态。主要的 Browser 窗口概括了活动中的系统和应用任务、内存消耗、当前目标机内存映射的情况。使用 Browser T

具,开发者还可以检查:

- (1) 详细的任务信息;
- (2) 信号量;



图 5-14 启动 WindShell

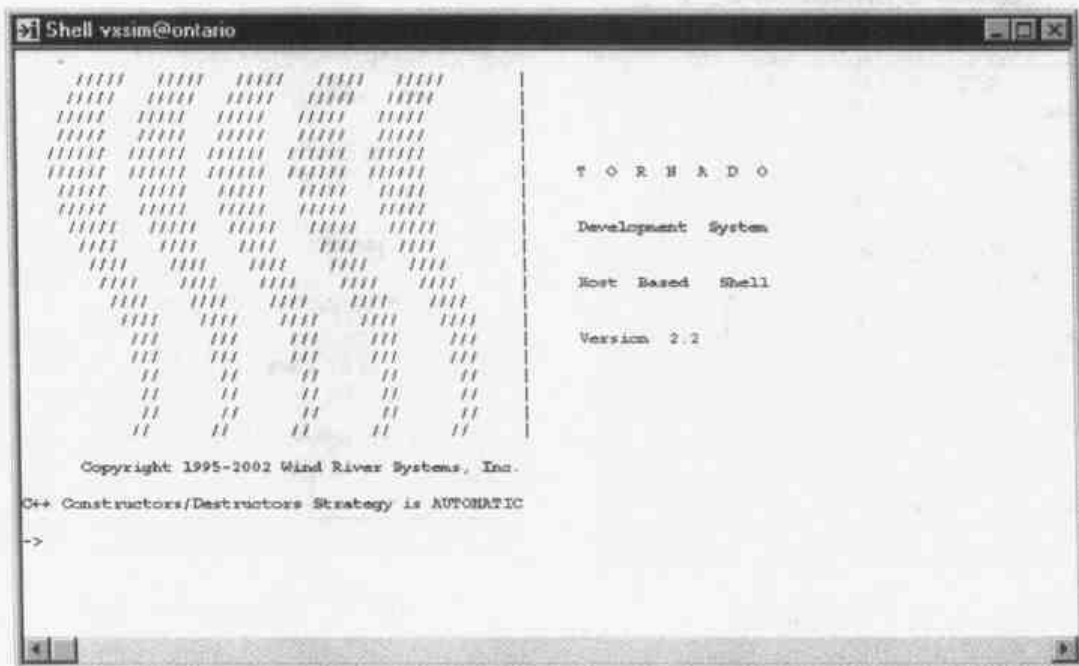


图 5-15 WindShell 显示界面

- (3) 消息队列;
- (4) 内存分区;
- (5) 看门狗定时器;
- (6) 所有任务的堆栈使用情况;
- (7) 目标机的 cpu 使用情况;
- (8) 目标模块结构和符号;
- (9) 中断向量。

这些信息捕捉了当前的信息。可以动态更新信息,当显示信息有任何变化时,Browser 将以高亮状态显示。

Browser 工具的启动也有两种方式,一种是在工具菜单里选择,另外一种是在工具栏图标中选择启动。如图 5-16 所示。

在图 5-16 中,可以看到 Launch Browser 有 3 个部分:数据区,显示目标机各种信息;目标机选择区,选择哪个目标机;图形工具区。可以对 Browser 进行配置,如图 5-17 所示。



图 5-16 Browser 启动界面

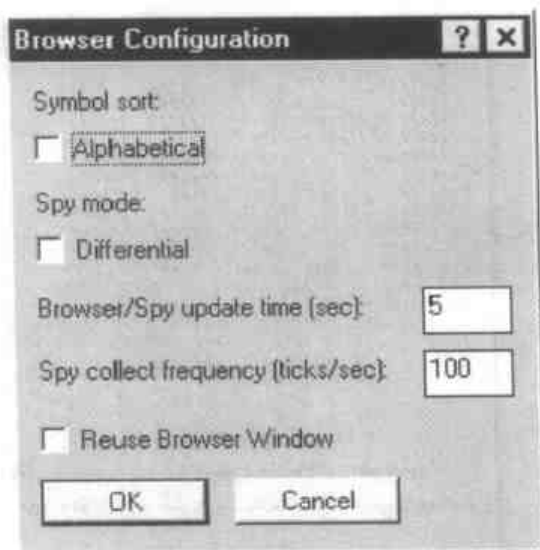


图 5-17 Browser 配置界面

配置了 Browser 工具的参数后,就可看到 Browser 的各种目标机相关信息界面,如图 5-18~图 5-21 所示。

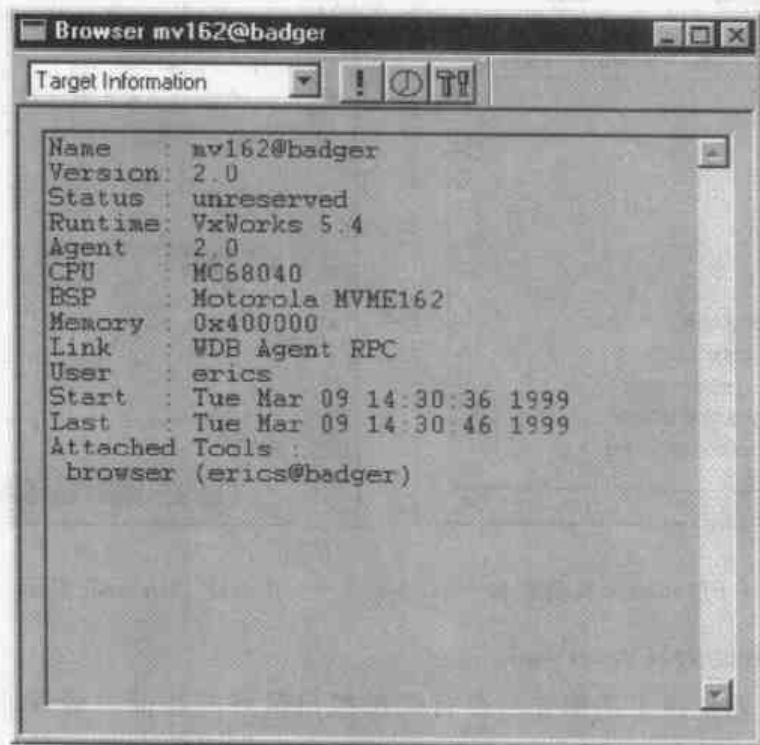


图 5-18 Browser 显示目标机信息界面

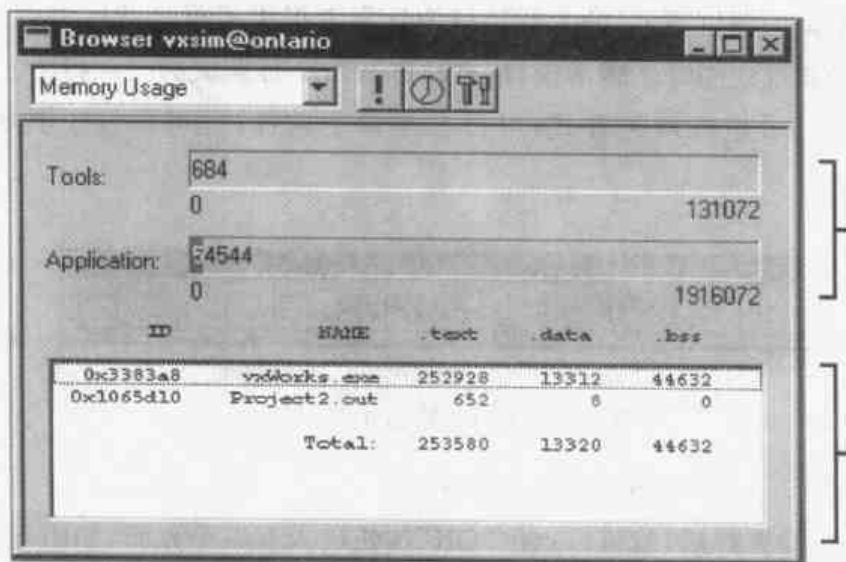


图 5-19 Browser 显示目标机内存使用信息-1

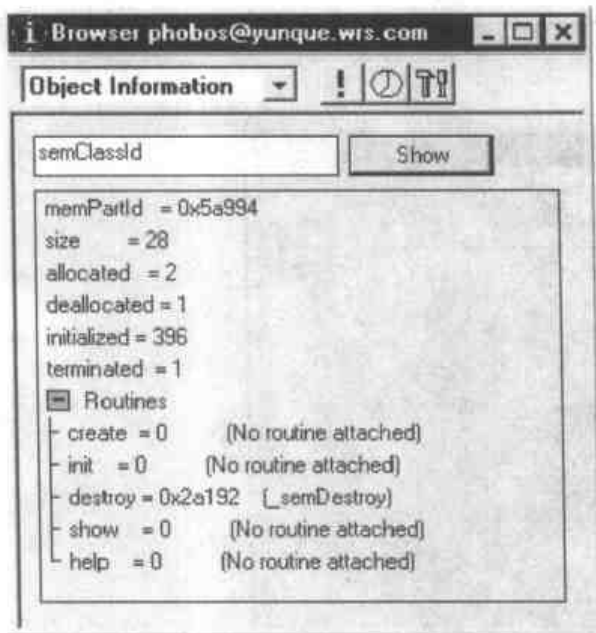


图 5-20 Browser 显示目标机信息-2

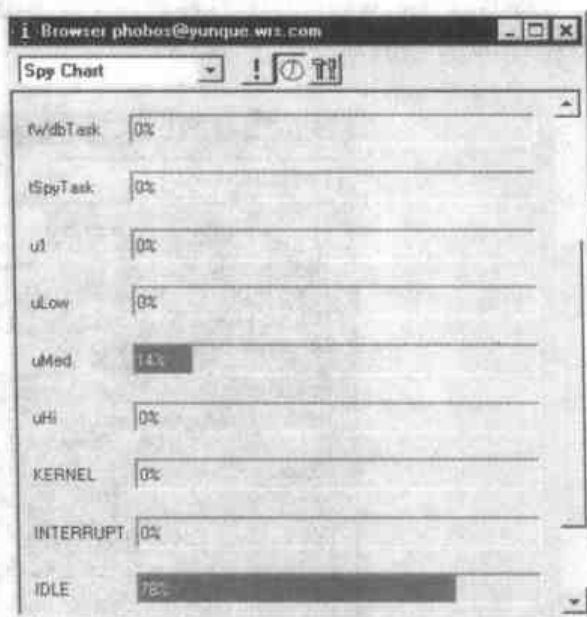


图 5-21 Browser 显示目标机信息-3

5. 调试器 Debugger(CrossWind)

Tornado 调试器集成了图形和命令行调试接口最好的特征。最普通的调试行为,如设置断点、控制程序执行可以在 CrossWind 中执行。类似地,程序列表和数据监视窗口为应用程序关键信息提供了一个可视的内容界面。

为了满足复杂的调试需要,命令行接口为开发者提供了很多特殊的调试命令。开发者可以扩展或者通过使用 Tcl 脚本接口,自动进行命令行调试的交互行为。

使用 CrossWind 也有两种方式,可以在菜单中选择,也可以在工具栏上选择。如图 5-22 所示。

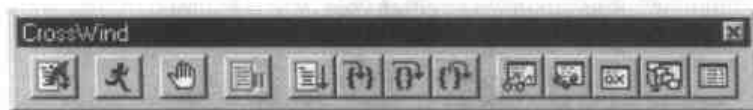


图 5-22 CrossWind 工具栏界面

如图 5-23 所示为 Debugger 的启动界面。

将 Debugger 配置好后,就可以点击“OK”按钮进入下一个界面,如图 5-24 所示。

进入运行界面,要选择任务的入口点,如图 5-24 所示,然后才可以进行单步调试界面,如图 5-25 所示。



图 5-23 CrossWind 启动界面

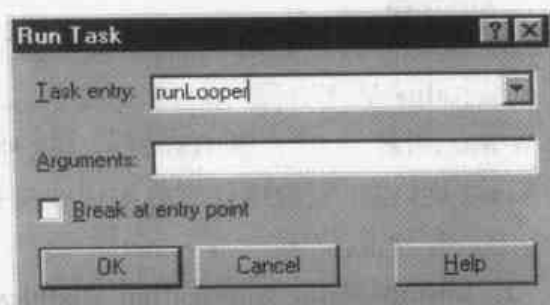


图 5-24 CrossWind 运行任务界面

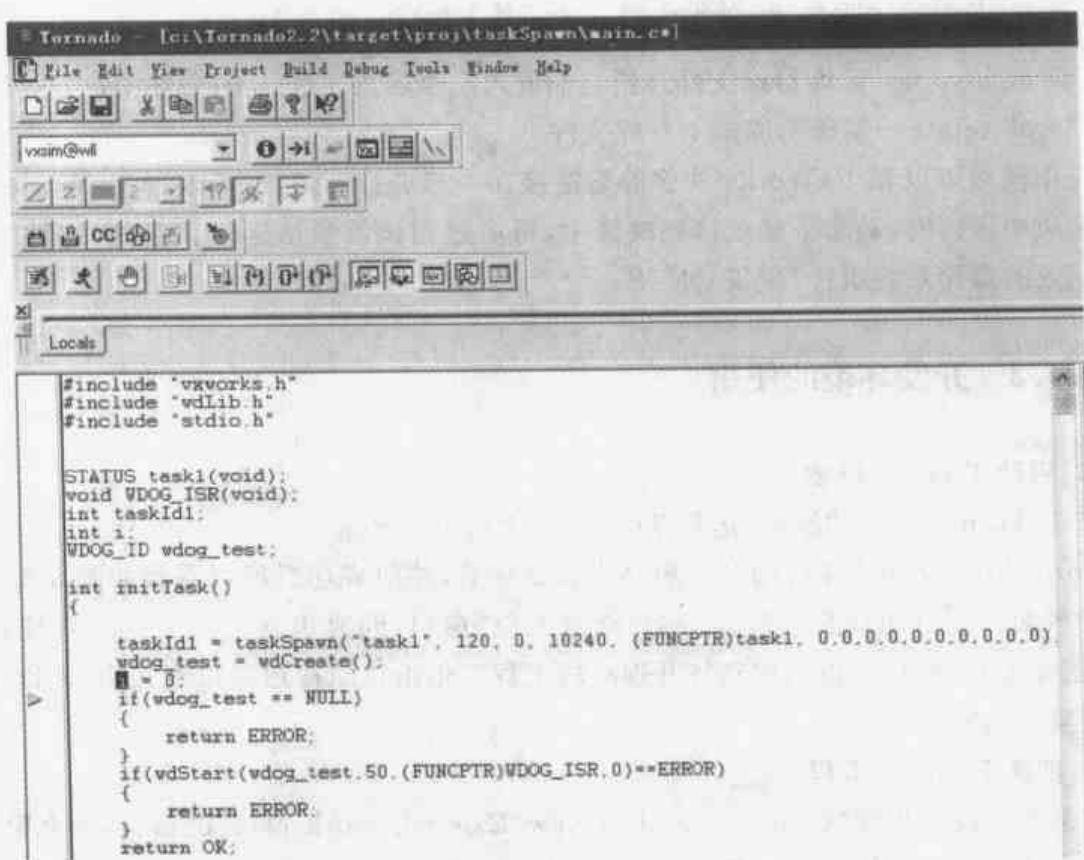


图 5-25 CrossWind 调试界面

5.1.2 Tornado 的工程类型

在 Tornado 开发环境下,可以建立两种工程类型,分别是 Bootable VxWorks Image 型工程和 Downloadable Application Modules 型工程。下面将分别介绍这两种不同类型的工程的特点。

1. Bootable VxWorks Image 型工程

Bootable VxWorks Image 类型的工程是可下载的 VxWorks 映像,包含了必需的 VxWorks 内核、组件以及 BSP。另外这种工程是可配置的,开发者不仅可以根据需要修改编译规则,在工程的组件列表中增加或删除组件,还可以选择特性项对相关组件内部的一些宏进行赋值。

2. Downloadable Application Modules 型工程

Downloadable Application Modules 类型的工程可以编译为可下载的应用模块。在此类型工程的 rules 项有 3 种选项:

- (1) 工程.out——编译目标文件并把它们链接而形成的文件;
- (2) archive——编译目标文件并把它们放入一个名为工程.a 的文件中;
- (3) objects——编译形成多个目标文件。

应用模块可以和 VxWorks 映像静态链接在一起进行下载,也可以当 VxWorks 映像在目标版中运行时,动态下载链接到映像中,再通过调试器激活运行。但是,这种工程没有可配置的操作系统组件,仅仅是应用。

5.1.3 开发环境的使用

1. 启动 Tornado 环境

点击 Tornado2.2 的图标,运行 Tornado2.2 的开发环境。

Tornado 开发环境在启动时一般会先尝试注册,注册成功后即可看到如图 5-26 所示的启动画面。在启动画面中包含一个“创建工程”窗口,同时包含大量工具的快捷按钮。利用“创建工程”窗口可以创建和打开现有的工程。单击快捷按钮可以直接启动 Tornado 开发环境的内嵌工具组件。

2. 创建工作区和工程

如果系统没有出现“Create Project in New/Existing Workspace”的窗口,那么第一步是打开文件(File)菜单,然后选择“Open Workspace”。

在弹出的 Open Workspace 对话框中选择“New”,将出现如图 5-27 所示的两种类型的工程。第一种表示创建可引导映像,第二种表示创建可下载的应用程序模块。选择其

中一种,单击“OK”按钮。

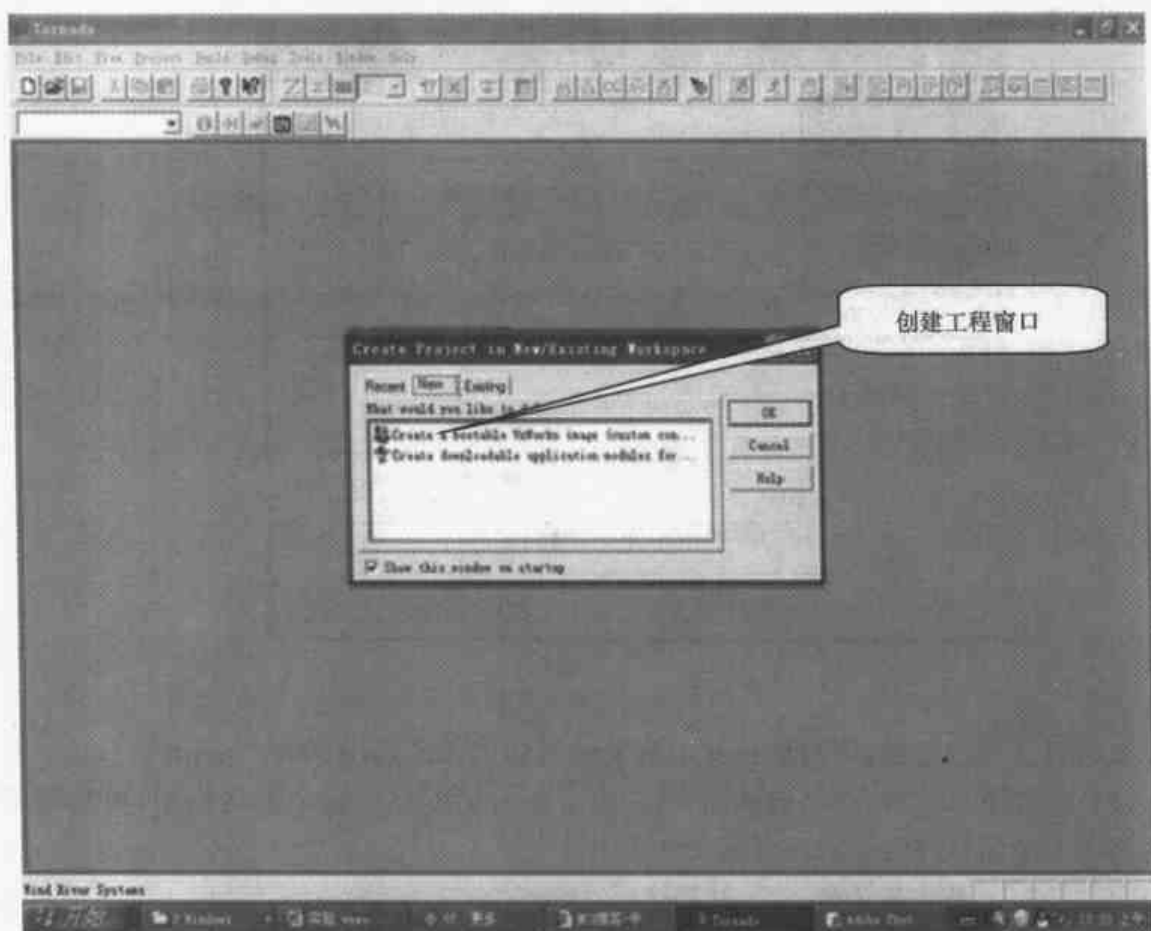


图 5-26 创建工程窗口-1

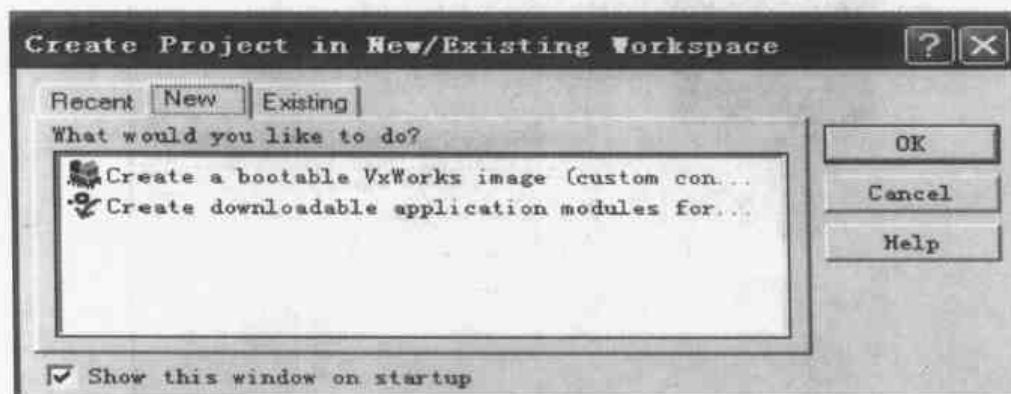


图 5-27 创建工程窗口-2

下面以创建 bootable VxWorks Image 为例进行说明。

(1) 点击“OK”后将弹出如图 5-28 所示的对话框。

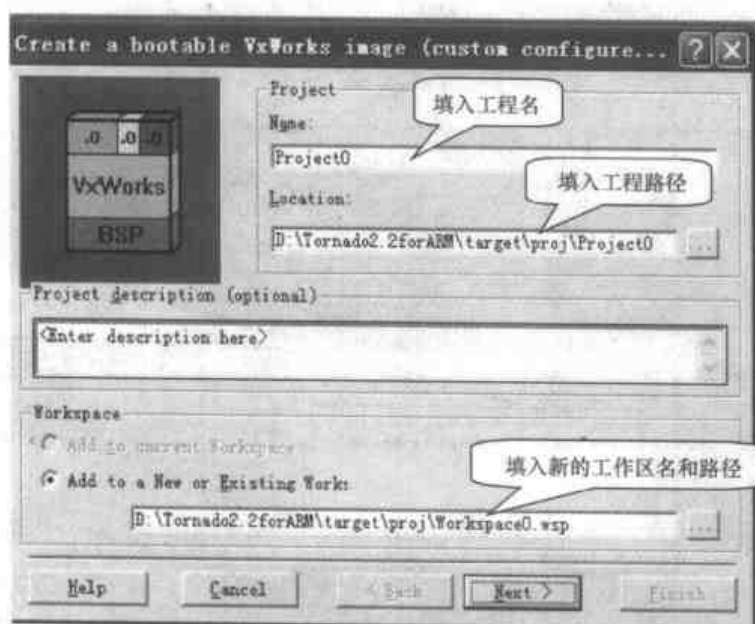


图 5-28 创建工程窗口-3

如果只是创建工程,则可以选择工程要加入的工作区,最后单击“Next”。

(2) 点击“Next”后,将出现如图 5-29 所示的对话框。在这个对话框中,可选择新工程是基于一个已存在的工程还是基于一个 BSP。

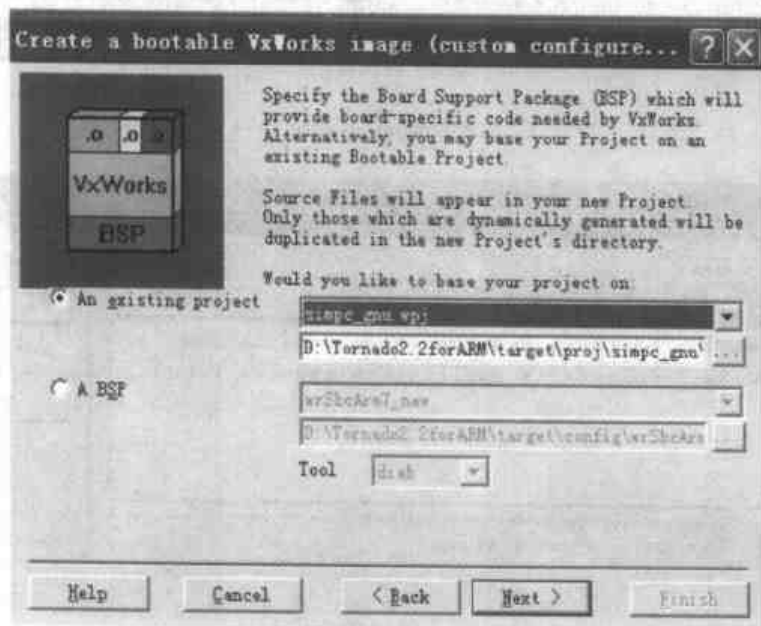


图 5-29 创建工程窗口-4

(3) 若选择基于一个已存在的工程, 点击“Next”将出现如图 5-30 所示的对话框。在这个对话框中列出了工作区、工程、工程基于已存在的工程还是基于 BSP 的信息。如果需要修改, 点击“Back”返回上一步; 否则点击“Finish”。这样, 新的工作区和工程就创建完成了。

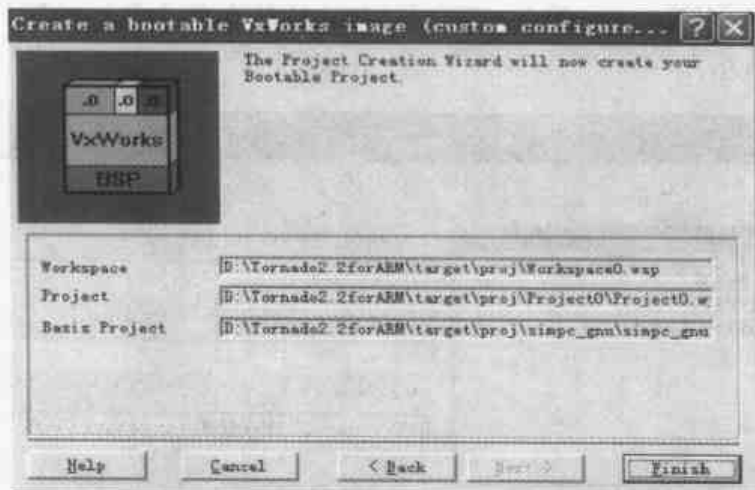


图 5-30 创建工程窗口-5

(4) 点击“Finish”后, 出现了工作区窗口, 如图 5-31 所示。工作区窗口中包含了工作区的名字以及包含的文件。基于此工作区的窗口, 便可以进行应用程序的开发。

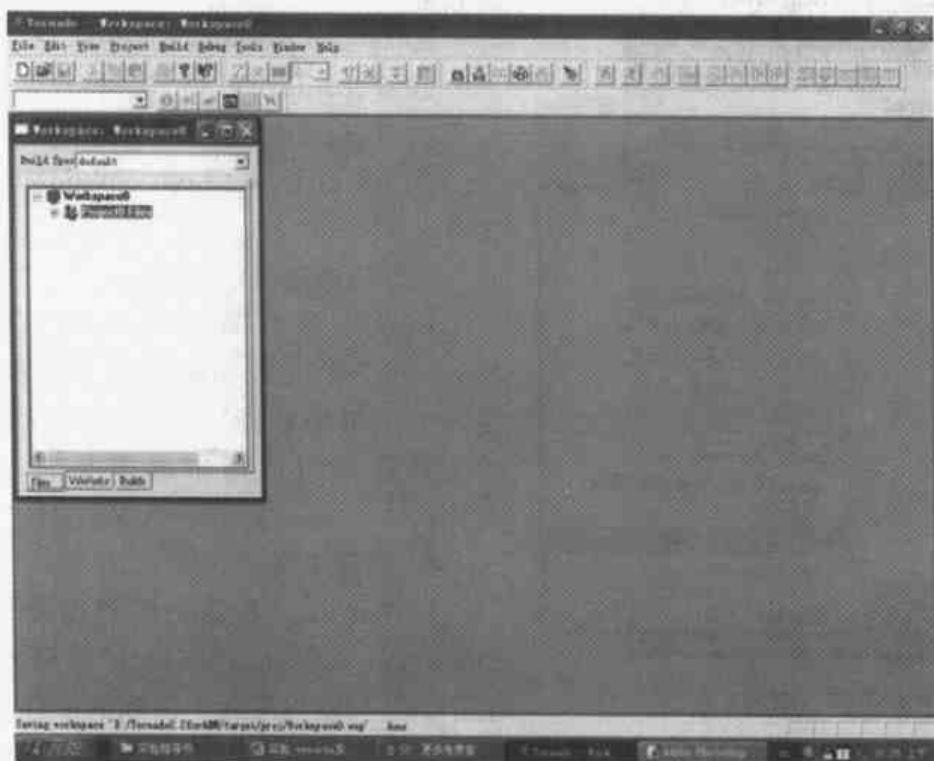


图 5-31 创建工程窗口

3. 添加文件到工作区和工程

(1) 建立新的文件

在菜单栏的“File”菜单中点击“New”项进入如图 5-32 所示的对话框。在此对话框中可以指明要建立的程序文件的类型、文件名称、文件所在的工程以及文件在磁盘上的具体位置。然后单击“OK”。

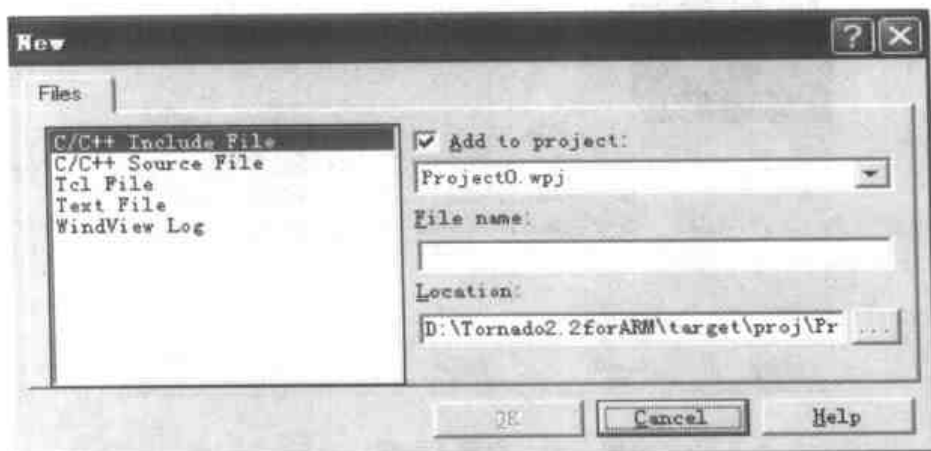


图 5-32 创建新源文件

① C/C++ Include File, C/C++ 头文件;



图 5-33 增加文件

② C/C++ Source File, C/C++ 源文件;

③ Tcl File, Tcl 格式文件;

④ Text File, 文本文件;

⑤ WindView Log, WindView 的日志文件。

(2) 添加已有的文件

用鼠标右击 Workspace0() 工作区内的“Project0”, 将出现如图 5-33 所示的浮动菜单, 在这个菜单中选择“Add Files”。然后在出现的文件选择对话框中选择要加入该工程的文件。

4. 编译工程

VxWorks 中所说的编译不是通常所说的 compile 的概念, 指的是 build 的概念, 它包括了编译、汇编、连接等几个阶段的工作。

(1) 在编译工程以前可以浏览编译规则

选择工作区中的 builds 选项卡, 然后单击 SIMNTgnu 的属性窗口, 如图 5-34 所示,

这个属性窗口列出了 SIMNTgnu 的若干特性,如编译规则、连接器、连接次序、汇编器、编译宏定义、C/C++ 编译器。

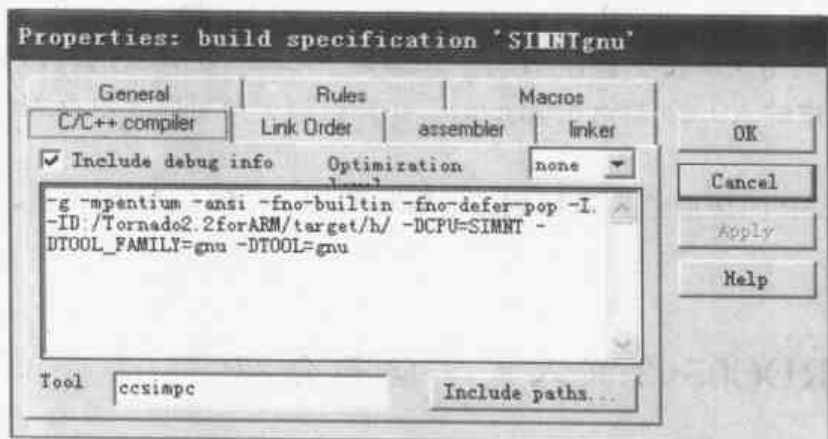


图 5-34 设计编译规则

(2) 编译工作区的工程

右击 SIMNTgnu,在浮动的弹出菜单中选择“build project×××”。如果是第一次编译源文件,系统会自动弹出所编译文件的依赖关系对话框。编译结束后,系统在 build out 窗口显示编译出错信息和成功信息。

5. 下载工程到 VxWorks 的目标模拟器

(1) 找到要下载的工程文件,右击此工程文件,在弹出的菜单中选择“download project ×××”。

(2) 点击 download 后系统会自动提示是否启动 VxWorks 中的目标模拟器(这是因为我们在前面创建工程时,选择了 SIMNTgnu 作为工程所基于的工具链),选择“是”。

(3) 接下来系统将弹出目标模拟器的可配置选项,按系统默认值即可。

(4) 目标模拟器启动成功后,屏幕右下方将出现如图 5-35 所示的标志。

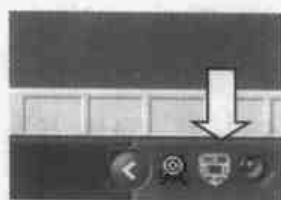


图 5-35 模拟器成功显示情况

6. 在线帮助的使用

(1) 运行 Tornado 和 VxWorks 的帮助文件,启动 Tornado 的 IDE → Start → Tornado2.2 → Tornado。

(2) 从主菜单栏中选择 Help → Manuals content,会弹出系统默认的浏览器并在上面显示 Tornado 的在线手册。

(3) 从 VxWorks API 参考手册开始。这个参考手册中包括 VxWorks 的程序和库的相关信息。在打开的浏览器中选择 VxWorks API Reference。从一个库的信息开始

看。点击 OS Libraries, 选择 bLib。可以查看关于 buffer 操作的库。

(4) 在文档中查看 read() 程序。如果知道程序 read() 属于 ioLib 中, 则选择 VxWorks API Reference → OS Libraries → ioLib → read()。如果不知道 read() 的具体路径, 在 Tornado 的主菜单栏中选择 Help → Manuals index, 在弹出的对话框中选择 Index 标签, 在文本框中输入 read 后点击 display。在浏览器中便可以显示有关程序 read() 的帮助信息。

(5) 查看 BSP 的帮助信息返回到 Tornado Online Manuals 的主页, 选择 VxWorks BSP Reference。

5.2 IRDC05v3 嵌入式实验箱介绍

IRDC-ARM 学生实验开发板所实现的平台在当前嵌入式应用中是一个非常实用的技术, 试验板在硬件上实现了双以太网控制器、USB 主/从双设备接入、低速的双 UART 串行通信端口、人机交互界面(键盘的输入、液品的输出)。软件上在移植了实时操作系统 VxWorks 的基础上, 进一步实现了 ttfS 以及 dos 文件系统的支持, 再加上平台上开源移植的支持 asp 脚本服务的 WebServer, 适当裁减就可以非常方便地应用到很多嵌入式网络产品中去(包括很多的无线网络应用如 AP 无线接入点等)。这些技术也可以很方便地被继承到其他嵌入式课题中去。为了配合教学任务的完成, 在完善开发板功能的同时, 在此基础上进行了学生实验的编写工作。具体实验分为 VxWorks 系统部分和硬件调试两个部分。经过 3 个版本的开发调试, 新的实验板现在已经稳定运行, 应该已经可以完全满足学生的实验要求。以下从硬件方案、系统软件方案两个方面进行说明。

5.2.1 硬件方案说明

根据系统的应用环境, 实验板硬件上必须满足 4 种接口要求: 高速设备多端口接入实现路由控制、低速控制网络、人机界面的输入、输出显示和 INTERNET 接入访问及控制显示。在嵌入式设计中, 虽然它们都属于产品的外设部分, 但是, 它们决定了系统的处理器和内存的选型。一般情况下嵌入式产品设计包括 3 个部分: 处理器、内存和外设。所以实验板硬件方案也从这 3 个方面来分析。下面, 通过这 3 个部分的具体介绍, 来讨论硬件平台的具体构成和每个部分的器件选型。

1. CPU

在实验板应用中, 需要满足 4 种接口要求。因此, 处理器必须具备高速的处理能力, 适合于网络应用, 能满足众多的外设接口要求, 并且还要有比较高的性价比。目前, 结合嵌入式的当前发展状况, 实验板选择了三星公司基于 ARM 的 S3C4510B 芯片作为处理器芯片, 该芯片以其优异的网络性能和高性价比而被广范应用在各种嵌入式网络设备中。

S3C4510B 主频 50 MHz, 具有 64 M 的寻址范围, 内置单以太网控制器。

2. 双(多)以太网接口

此模块属于外设部分。实验板通过外接具有 Mac 层功能的以太网接入芯片可以方便而高速地扩展内置以太网接口。同时, 实验板要求物理上实现连接两个(在此基础上可以扩展)网段, 因此, 它必须在物理上具备两个网络接口, 满足系统应用方案中“高速设备接入”和“INTERNET 接入”以及路由的要求。在两个以太网口中, 一个用于连接家庭内部网络, 实现家庭内部网络的管理; 另一个用于高速 INTERNET 接入, 实现家庭内部网络设备的 INTERNET 访问和远程管理。对于实现单个物理设备的多个 IP 的管理, 通过其上的系统平台(我们选用 VxWorks)实现。由于 CPU S3C4510B 内置一个以太网控制器, 只需外加一个以太网收发芯片、网络变压器和 RJ45 接头, 即可实现一路以太网接口。对于需要增加的第二路以太网接口, 它的器件选型需要满足接口简单、资料丰富、软件移植性好等特点, 方便实验板的设计。目前结合 CPU S3C4510B 的实际处理能力, 以太网接口芯片 RTL8019AS 是比较好的选择。它具有 10 M 的传输速度, 兼容 NE2000 网卡寄存器结构, 并且 RTL8019AS 的网上资料丰富, 芯片价格也比较便宜。

3. 双串行接口

属于外设部分。它可以外接低速控制设备, 满足系统应用方案中“低速控制网络”的要求。两个串口, 一个用于连接低速控制总线, 实现家庭内部低速控制、调试网络; 另一个可以外接 MODEM 用于低速 INTERNET 接入。由于 CPU 内置了两个 UART 控制器, 串行口的设计比较简单, 外接一个传统的串行接口芯片(例如 MAX232)即可。对于其他的串行总线, 更换对应的接口电平转换芯片即可(例如对于 485 总线, 增加 MAX485 接口芯片即可)。

4. 液晶和键盘

属于外设部分。实现简单的人机界面, 实现系统应用方案中“人机界面”的要求。本课题中对液晶和键盘没有特别严格的要求。现在市场上液晶的可选择种类比较多, 大多也都能满足本课题的要求。分析了几种液晶产品, 考虑到显示速度、连接接口、价格以及稳定性等因素, 最终选择了北京青云创新公司的 T6963c 控制器的一款 128×64 液晶。键盘的选型比较简单, 它是低速设备, 接口简单, 没有很多性能、指标的要求, 最后考虑到外观, 选择了一个 4×4 小键盘。

5. 存储设备

存储设备对于嵌入式设计是一个非常关键的部分, 它决定了系统的性能能否得到最大程度的发挥。在本实验板的设计中, 由于系统外设接口较多, 功能较多, 必须要选择容量大而且速度快的内存和 Flash 以提高系统的性能。在内存总线上, 设计了 32 位内存总线从而提高系统的整体速度。内存的选型综合了速度、容量和成本的考虑, 选择了 SDRAM 内存。SDRAM 的最大特点就是可以与 CPU 的外频同步, 可以取消等待周期, 减少了数据传输的延迟, 提高了存取速度。另外 SDRAM 又具有 DRAM 的大容量的特

点,因此 SDRAM 是本课题的理想选择。在本课题中,使用了两片 HY57V281620 SDRAM 组成 $8\text{M} \times 32$ 的内存。Flash 芯片选择相对简单,没有本质上的差异,实验板选择了两种 Flash 芯片分别用来存储 VxWorks 的启动映像和文件系统。这两种 Flash 芯片分别是 sst39vf040 和 sst39vf160。通过跳线选择可以使用不同的芯片作为启动设备,从而实现了多种系统的装载(VxWorks 或 Linux)。目前默认模式为 VxWorks 系统启动,另一个芯片实现 tffs 文件系统。

通过上面硬件方案的分析,可以得出如图 5-36 所示的实验板硬件功能扩展的系统结构图。

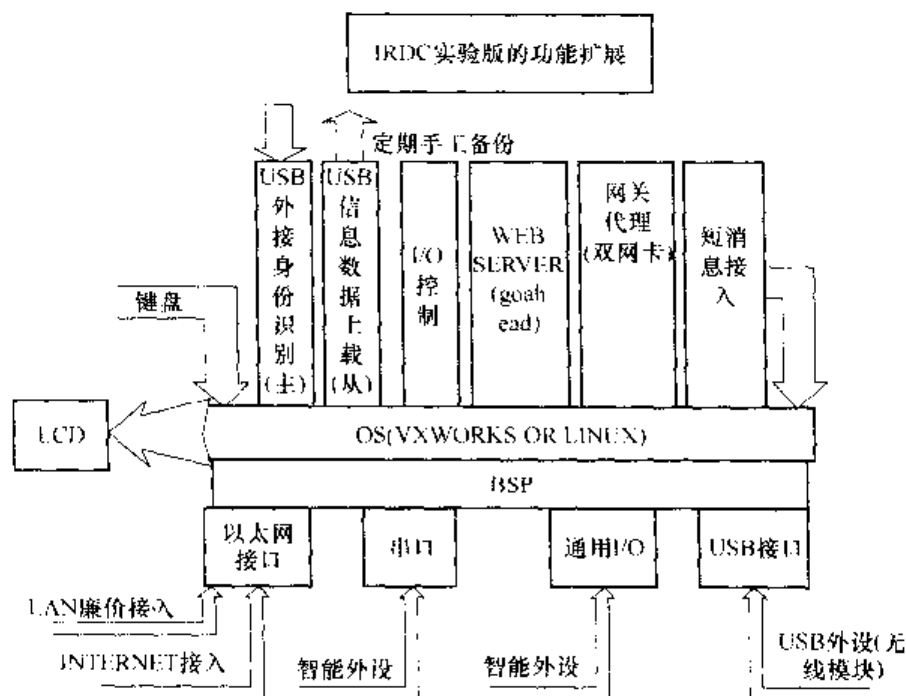


图 5-36 系统结构图

5.2.2 系统软件方案说明

1. 嵌入式 OS

实验板既要实现网络控制中心的作用又要兼顾学生实验的功能,在软件上必须有嵌入式操作系统(Operating System, OS)的支持。当前流行的嵌入式 OS 很多,像 Linux、UC/OS2、WINCE、VxWorks 等等。它们都比较成功,分别都有自己的应用领域。其中 VxWorks 操作系统是 Wind River 公司推出的嵌入式强实时操作系统,VxWorks 操作系统有许多显著的特点,非常适合实验板的应用。第一,VxWorks 的微内核 Wind 是一个高效的实时系统内核,具有任务间切换时间短、中断时间短、网络流量大等特点,非常适合于实时应用;第二,VxWorks 系统具有非常好的可裁减能力,可以根据实验和应用的实际需求进行裁减,非常适合于资源有限的嵌入式应用;第三,VxWorks 的集成开发环境

Tornado 集成了非常丰富的调试工具,支持应用程序的动态连接和动态下载,可以方便地搭成主机和目标机的开发环境,方便开发和调试,缩短开发周期。系统启动或复位时的系统寄存器分布如图 5-37 所示。

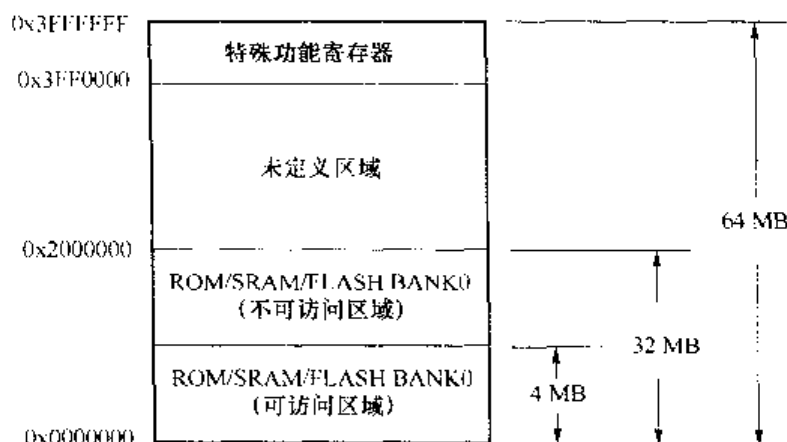


图 5-37 系统启动或复位时的系统寄存器分布图

2. 软件方案分析

软件方案建立在硬件方案的基础上,它的形成需要和硬件系统、实验板的实际应用密切相关。根据前面嵌入式操作系统的介绍和硬件平台的系统方案分析,可以得出系统的软件方案层次图,如图 5-38 所示。图中系统软件方案主要包括以下几部分:Board Support Package(BSP)模块、嵌入式操作系统 VxWorks 模块、嵌入式的 webServer 模块、各个设备的驱动模块和应用程序模块。这些模块中,BSP 模块和驱动模块与硬件平台密切相关,并且 VxWorks 模块和 webServer 模块也直接依赖于这两个模块。应用模块建立在这上面 4 个模块的基础上,解决实际的用户需求。

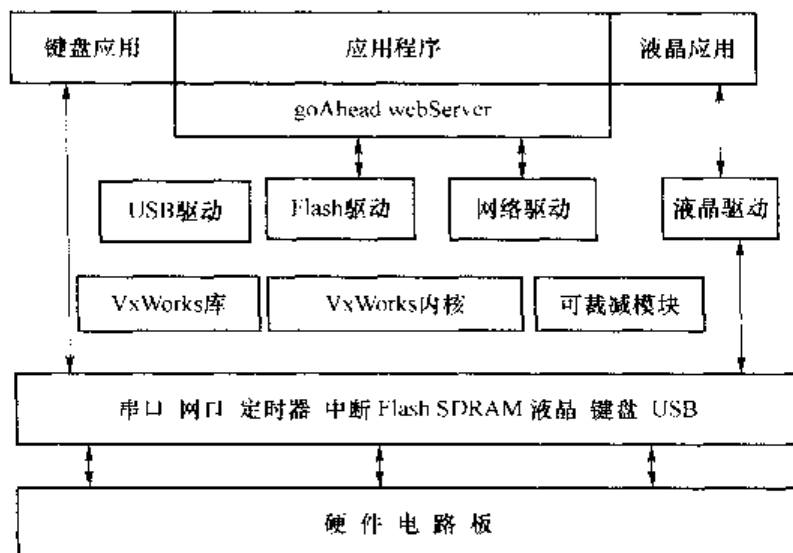


图 5-38 扩展功能方案

在整个软件系统中 BSP 模块是基础。VxWorks 的一个强项就是它可以使应用程序编码在很大程度上与目标板的硬件和结构无关,这都要归功于 VxWorks 的 BSP。实验板软件部分工作的第一步就是移植 BSP。在 BSP 成功移植的基础上才能启动 VxWorks 操作系统,加载各种驱动程序和应用。实验板中 BSP 的移植是一个非常大的工作,它涉及到硬件平台中所有硬件的初始化,以及内存管理等工作。说它工作量大,还有一个原因,就是移植 BSP 没有特别方便的调试工具,每一次都需要“修改代码→下载到 Flash→察看调试信息→再修改代码”这个过程,比较费时。

对于嵌入式的 webServer 模块,在实验板中,选择移植一个开源的嵌入式 goAhead webServer。它的下层需要文件系统和以太网驱动的支持。因此实验板中必须加载基于 Flash 的文件系统和双以太网驱动。webServer 模块在嵌入式应用中是一个非常实用的功能,可以应用在很多的嵌入式设备上。在实验板中,它能方便地响应来自 INTERNET 网络的请求,实现家庭内部网络管理和网络平台设备自身管理的功能。

驱动模块在 BSP 的基础上实现各个外设的驱动:Flash 文件系统驱动、双以太网驱动、液晶驱动等。这一部分与硬件密切相关,是一个比较细致并且费时的的工作,它需要对各个硬件芯片、BSP、VxWorks 和调试工具都相当地熟悉才能完成。

第 6 章 内核基本功能实验

本章是关于 VxWorks 和 Tornado 开发环境的基本实验。通过这些实验可以熟悉开发环境中各种工具的使用,熟悉整个开发过程,并且对于 VxWorks 操作系统中任务之间的通信和同步的机制有较深入的了解。

6.1 实验 1 Hello world

1. 实验目的

- (1) 掌握基于 VxWorks 的工程开发流程;
- (2) 熟悉 Tornado 工具的使用。

2. 实验内容

- (1) 阅读 Tornado 帮助文档或者第 5 章内容,熟悉 Tornado 工具的使用;
- (2) 编写打印出“Hello world!”字符串的标准 C 代码;
- (3) 使用 Tornado 集成开发环境来下载、调试程序。

3. 预备知识

- (1) 熟悉标准 C 语言的基本语法;
- (2) 了解 VxWorks 操作系统。

4. 实验设备和工具

- (1) 主机一台;
- (2) Win2000/XP 操作系统;
- (3) Tornado2.2 for arm 集成开发环境。

5. 实验步骤

- (1) 建立一个 downloadable 工程,如图 6-1 所示。

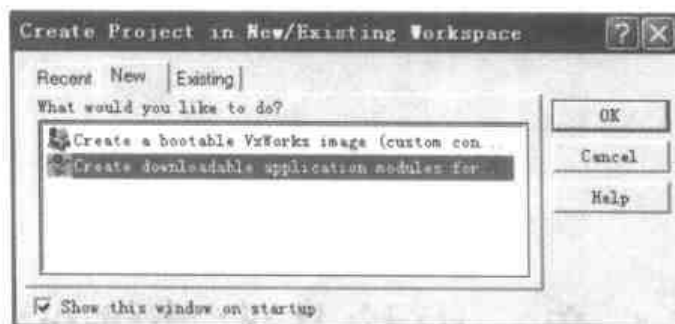


图 6-1 建立工程

(2) 选择 VxSim 模拟工具, 如图 6-2 所示。



图 6-2 选择 SIMNTgnu 工具链

(3) 编写主程序。

```

/ ***** hello.c ***** /
int main(void)
{
    printf("Hello world! \n");
}
/ ***** /
    
```

(4) 运行调试。

① 编译程序

- 在所要编译的文件上单击鼠标右键, 将弹出如图 6-3 所示的对话框, 选择“compile hello.c”(hello.c 为此例所用的源程序文件);

- 如果程序通过编译,会弹出完成的对话框,否则将报错;
- 在上面弹出的菜单中选择“build hello_world.out”生成.out文件。



图 6-3 编译文件

② 下载工程文件

单击“download hello_world.out”。这时候会启动 VxSim 工具,成功设置后会在任务栏右下角看到一个红色的图标。成功启动 VxSim 后会将执行文件下载到虚拟目标机中。

③ 运行程序

在 Shell 中输入主函数名 main,然后回车。成功运行后会返回 Value 值。

④ 观察、分析运行结果

注意观察结果,看看是否有“Hello world!”字符串打印到 Shell 中或者控制台中(具体在哪里显示取决于打印所定向的端口)。

另外,可以在 Shell 中通过输入一些 shell 命令查看程序执行情况。

6. 思考题

- (1) 在 VxSim 中调试程序和和在试验板中调试程序有什么不同,各自的优缺点是什么?
- (2) 如何开发大型的工程项目?

6.2 实验 2 信号量

1. 实验目的

- (1) 了解 VxWorks 的同步和互斥机制;
- (2) 熟悉信号量原理和使用方法;

- (3) 掌握并学会使用信号量的创建、打开和释放函数；
- (4) 能够在今后的实践中灵活运用信号量实现任务的同步和互斥。

2. 实验内容

实时操作系统中对于临界资源的保护与任务的同步协作，一般都提供了信号量机制。通常将信号量分为 3 种：二进制信号量、计数信号量、互斥信号量。二进制信号量常用于相互协作任务之间的同步。计数信号量常用于管理多个临界资源的使用。互斥信号量常用于对单一临界资源的保护。本实验使用 VxWorks 的开发环境，通过分别编写二进制信号量、计数器信号量和互斥信号量的 3 个程序，在 VxSim 环境下模拟信号量的执行过程，并使用 Shell 工具执行函数、创建任务及观察系统任务的运行情况。

3. 预备知识

- (1) C 语言编程的基础知识；
- (2) 虚拟环境 VxSim 的启动；
- (3) VxWorks 中 Shell 工具的使用。

4. 实验设备和工具

- (1) 主机一台；
- (2) Win2000/XP 操作系统；
- (3) Tornado2.2 for arm 集成开发环境。

5. 信号量原理

(1) 二进制信号量

- 对应于事件；
- 存在置位和清除两种状态，分别对应于事件的开始和结束；
- 信号量创建 `semBCreate()`；
- 任务调度：
 - 获取信号量 `semTake()` ——若事件已经发生则该任务进入阻塞状态；
 - 释放信号量 `semGive()` ——等待该事件的任务队列中的第一个任务解除阻塞。

(2) 计数信号量

在 VxWorks 中使用的比较少，这是一个针对计数的信号量，已经预封装了互斥机制。

(3) 互斥信号量

- 针对某个被保护的资源而创建的，用来实现对临界资源的保护；
- 信号量创建 `semMCreate()`；
- 任务调度同二进制信号量；
- 拥有临界资源的任务可重复获得资源的使用权，而且只有资源的所有者才能释放该资源。

(4) 系统实现及函数原型

Wind 信号量对于各种类型的信号量的控制提供了同一规范化的接口，在创建函数时需要特别指明信号量类型。

```

semBCreate()          //创建(产生并激活)一个二进制信号量
SEM_ID      semBcreat
{
    int      options    //信号量选项
    SEM_B_STATE      initialState    //信号量初始化状态
}

semMCreate()          //创建(产生并激活)一个互斥信号量
semCCreate()          //创建(产生并激活)制一个计数信号量
SEM_ID      semCCreat
{
    int      options    //信号量选项
    int      initialCount //信号量初始化计数值
}

semDelete()           //中止并自由信号量
STATUS      semDelete
{
    SEM_ID    semId    //要删除的信号量 ID 号
}

semTake( )            //获得信号量
STATUS      semTake
{
    SEM_ID    semId    //要得到的信号量 ID 号
    Int      timeout    //等待时间
}

semGive()             //给出信号量
STATUS      semGive
{
    SEM_ID    semId    //要给出的信号量 ID 号
}

semFlush()            //解锁所有正在等待某一信号量的任务

```

6. 实验步骤

- (1) 建立一个 downloadable 工程(具体的工程创建可以参考本书第 5 章或者本章试验 1)
- (2) 编写主程序(以二进制信号量和计数器信号量为例)

① 二进制信号量 Task_semB 的实验

编程思想:编写基本的收发任务,接受任务收到信息后将信息打印出来。

Task_give (parameter)//用于释放信号量的任务

```

{
    FOREVER /* 任务为一个无限循环 */
    {
        初始化和准备工作
        ...
        semGive(synSem); /* 信号量释放后,唤醒较高优先权的执行任务 */
    }
}

```

Task_take (parameter) //用于获取信号量的任务

```

{
    FOREVER
    {
        semTake (synSem);
        ...
        执行具体的操作
        ...
    } /* 开始下一个循环时,由于得不到信号量,阻塞 */
}

```

Task_test (parameter) //程序入口函数

```

{
    parameter = semBCreate(...);
    //semBCreate: 用于创建并初始化二进制信号量
}

```

② 计数信号量 Task_scmC 的实验

编程思想:定义资源数目 nCount。

Task_give (parameter)//用于释放信号量的任务

```

{
    FOREVER /* 任务为一个无限循环 */
    {
        初始化和准备工作
        ...
        semGive(synSem); /* 信号量释放后,唤醒较高优先权的执行任务 */
        nCount ++ ;
    }
}

```



```

Task_take (parameter) //用于获取信号量的任务
{
    FOREVER
    {
        nCount--;
        semTake (synSem);
        ...
        执行具体的操作
        ...
    }
    /* 开始下一个循环时,由于得不到信号量,阻塞 */
}

Task_test (parameter) //程序入口函数
{
    parameter = semCCreate(...);
    //semCCreate: 用于创建并初始化计数信号量
}

```

(3) 运行调试

① 编译程序

- 在所要编译的文件上单击鼠标右键,将弹出如图 6-4 所示的对话框,选择“compile sem_B.c”(sem_B.c 为此例所用的源程序文件);

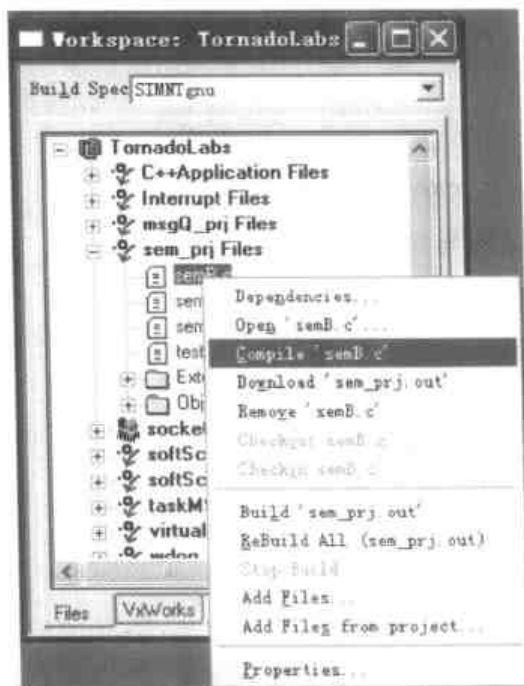


图 6-4 编译文件

- 如果程序通过编译,会弹出完成的对话框,否则将报错;
- 在上面弹出的菜单中选择“build sem_prj.out”生成.out 文件。

② 下载工程文件

单击“download sem_prj.out”。如果有符号表不统一的情况发生,下载的时候会有相应的提示信息。正常情况下没有任何额外信息。

③ 运行结果

在 Shell 中输入启动函数名:Task_test。如果执行不正确则会打印出相应的错误信息和错误号。

④ 观察,分析运行结果

注意观察结果,看看是否有消息打印到 Shell 中或者控制台中(具体在哪里显示取决于打印所定向的端口)。

另外,可以在 Shell 中通过输入一些 shell 命令或者启动 WindView 等调试工具观察任务执行的情况,比如任务的优先级、堆栈、ID 号是否正常,任务之间的交互是否正常,有没有错误信息打印出来等。

7. 思考题

- (1) 二进制信号量、计数信号量和互斥信号量的联系、区别以及各自的适用范围。
- (2) 在掌握了二进制信号量和计数信号量的基础上,学习互斥信号量的原理,并编写调试运行互斥信号量的程序。

6.3 实验 3 消息队列

1. 实验目的

- (1) 了解 VxWorks 中任务的划分;
- (2) 熟悉消息队列的原理和使用方法;
- (3) 初步掌握程序的调试方法。

2. 实验内容

- (1) 编写程序,利用消息队列实现任务间的通信;
- (2) 调试程序并用 VxSim 运行。

3. 预备知识

- (1) 任务的相关知识;
- (2) C 语言的基础知识;
- (3) 消息队列的原理和实现方法。

4. 实验设备和工具

- (1) 主机一台;

- (2) Win2000/XP 操作系统;
- (3) Tornado2.2 for arm 集成开发环境。

5. 实验原理

(1) VxWorks 中任务的特点和划分原则

① 任务不仅在表面上具有与普通函数相似的格式,而且还有着自己较明显的特点:

- 具有任务初始化的起点;
- 具有存放执行内容的私用数据区;
- 任务的主体结构表现为一个无限循环体或有明确的终止条件,它不同于函数,一般不需要返回。

② 任务划分的原则:

- 时间——两个任务所依赖的周期条件具有不同的频率和时间段;
- 异步性——两个任务所依赖的条件没有相互的时间关系;
- 优先级——两个任务所依赖的条件需要有不同的优先级;
- 清晰性/可维护性——两个任务可以在功能上或逻辑上互相分开。

(2) 关于消息队列

① 消息机制使用一个被各个有关进程共享的消息队列,任务之间经由这个消息队列发送和接收消息;

② 全双工队列(如图 6-5 所示);

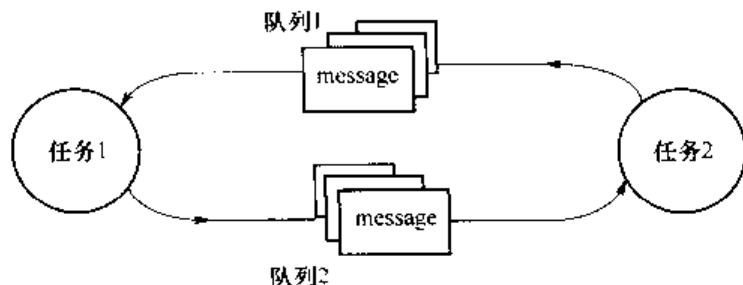


图 6-5 全双工工作原理

③ 实现:分别建立用于发送消息和接收消息的任务。

(3) Wind 消息队列管理

- msgQCreate():创建(产生并激活)消息队列

MSG_Q_ID msgQCreate

```
(
    int maxMsgs,           //队列所能容纳的最大消息数目
    int  maxMsgLength,     //每一消息的最大长度
    int  options           //消息的入列方式
)
```

- msgQSend(): 向消息队列发送消息

STATUS msgQSend

```
(
    MSG_Q_ID msgQId,          //所发向的消息队列名
    char * buffer,            //消息包所在缓冲区指针
    UINT      nBytes,         //消息包长度
    int       timeout,        //等待的时间长度
    int       priority        //优先级
)
```

- msgQDelete(): 中止并删除信号量

STATUS msgQDelete

```
(
    MSG_Q_ID msgQId,          //要删除的消息队列 ID 号
)
```

- msgQReceive(): 从消息队列接收消息

int msgQReceive

```
(
    MSG_Q_ID msgQId,          //接收消息的消息队列 ID 号
    char * buffer,            //接收消息的缓冲区指针
    UINT      maxNBytes,      //缓冲区长度
    int       timeout         //等待时间
)
```

6. 实验步骤

- (1) 建立一个 downloadable 工程;
- (2) 在新建的工程中添加 .c 文件;
- (3) 编写主程序。

设计程序流程图如图 6-6 所示。

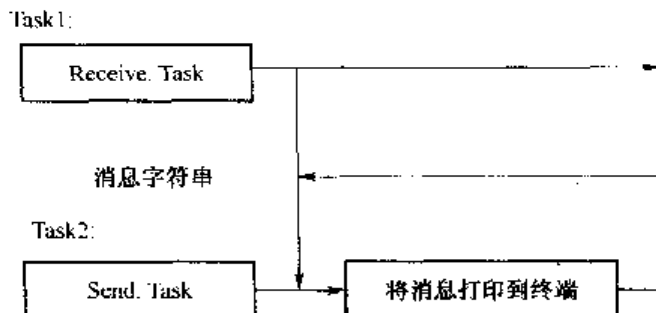


图 6-6 设计流程图

编程思想:在主函数中创建两个任务 Task1(Receive.task 用于接收消息)和 Task2 (Send.Task 用于发送消息)。这两个任务通过消息队列 messageQue 实现关联。Task1 任务等待 Task2 任务通过消息队列发来的消息,接到消息以后,Task1 便在终端打印接收到的字符串。

```
void Send.task(void) //用于实现消息的发送任务
{
    int maxMsgs = 64; //队列允许的最大消息数
    int maxMsgLength = 1024; //消息中的最大字节长度
    int options = MSG_Q_FIFO; //消息队列属性选项
    //定义消息队列参数

    msgQCreate(maxMsgs,maxMsgLength,options);
    //创建消息队列

    for(循环参数) //无限发送消息
    {
        if(loop_num == value)
        {
            ...
            msgQSend(parameter);
            ...
        }
    }

    void receive.task(void) //用于实现消息的接收任务
    {
        for(;;) //无限等待消息的到来
        {
            ...
            msgQReceive(parameter);
            ...
        }
    }

    void CreateTask(void) //用于实现消息的发送和接收任务的创建
    {
```

```
...  
    taskSpawn("sendMSGTask",100,VX_PRIVATE_ENV,512,  
    发送任务函数名,0,0,0,0,0,0,0,0,0,0);  
...  
}
```

(4) 运行调试

在 Shell 中输入启动函数名:CreateTask。不正确的执行会打印出相应的错误信息和错误号。

(5) 观察、分析运行结果

注意观察结果,看看是否有消息打印到 Shell 中或者控制台中(具体在哪显示取决于打印所定向的端口)。

另外,可以在 Shell 中通过输入一些 shell 命令或者启动 WindView 等调试工具观察任务执行的情况,比如任务的优先级、堆栈、ID 号是否正常,任务之间的交互是否正常,有没有错误信息打印出来。

7. 思考题

- (1) 消息队列的优势和用途。
- (2) 实现任务间通信还有哪些其他的方法,它们和消息队列相比有什么优势?

6.4 实验 4 看门狗

1. 实验目的

- (1) 了解看门狗(watchdog)定时器的原理和工作方式;
- (2) 了解 VxWorks 的定时管理机制。

2. 实验内容

掌握 VxWorks 系统提供的有关看门狗定时器的相关函数,并编写程序来实现和体会看门狗定时器的运行过程。

3. 预备知识

- (1) 中断服务程序的相关知识;
- (2) C 语言的基础知识;
- (3) 看门狗定时器的原理。

4. 实验设备和工具

- (1) 主机一台;
- (2) Win2000/XP 操作系统;
- (3) Tornado2.2 for arm 集成开发环境。

5. 实验原理

关于看门狗定时器

VxWorks 看门狗定时器属于系统时钟中断服务程序的一部分,允许 C 语言函数指明某一时间延迟。一般来说,被看门狗定时器激活的函数运行在系统时钟中断级。如果内核不能立即运行该函数,函数被放入 tExcTask 工作队列中。在 tExcTask 工作队列中的任务运行在最高优先级 0。

看门狗定时器调用函数:

wdCreate() 分配并初始化看门狗定时器

WDOG_ID wdCreate(void)

wdDelete() 中止并解除看门狗定时器

STATUS wdDelete

(

WDOG_ID wdId /* 被删除的定时器 ID 号 */

)

wdStart() 启动看门狗定时器

STATUS wdStart

(

WDOG_ID wdId, /* 看门狗定时器 ID 号 */

int delay, /* 延迟值,以滴答计 */

FUNCPTR pRoutine, /* 超时函数 */

int parameter /* 超时函数的参数 */

)

wdCancel() 取消当前正在计数的看门狗定时器

STATUS wdCancel

(

WDOG_ID wdId /* 被取消的定时器 ID 号 */

) 该函数只是让定时器的延迟值为零来取消其工作。

6. 实验步骤

(1) 建立一个 downloadable 工程(同前 3 个例子,都是在 VxSim 中执行)

(2) 编写主程序

编程思想:编写一个基本的看门狗服务程序,一旦看门狗中断到后就执行看门狗中断服务程序,在中断服务程序中打印一些信息,表示执行中断服务程序。

```
#define DELAY_SECONDS 5

WDOG_ID wid;

void watchdog_service (unsigned int zero) //看门狗超时服务程序
{
    在模拟器上输出相关信息
    loop_sec = zero;
}

Void watchdog_Routine() //用户程序
{
    运行程序;
    loop_sec ++; //在服务程序中初始化为零
    for(i=0;i<loop_sec;i++)
    {
        taskDelay(clk_rate);
        if(loop_sec) printf("%d...",i+1);
    }
}

void wdogStart() //启动看门狗
{
    loop_sec = 0;
    wid = wdCreate(); //创建定时器

    clk_rate = sysClkRateGet(); //记录系统时钟
    delay_sec = clk_rate * DELAY_SECONDS;

    FOREVER
    {
        wdStart (wid, delay_sec, (FUNCPTR)watchdog_service, 0);
        //未超时执行用户程序,超时执行服务程序
        Watchdog_Routine ();
        wdCancel(wid); //取消定时器
    }
}
```



```
}

```

(3) 编译、下载并运行程序

① 编译程序

在所要编译的文件上单击鼠标右键,将弹出对话框,选择源代码文件进行编译。

- 如果程序通过编译,会弹出完成的对话框,否则将报错;
- 生成.out 文件。

② 下载工程文件

下载.out 文件。如果符号表不统一或者不解析会有相应的提示信息。

③ 运行程序、观察结果

在 Shell 中输入本例程启动函数 wdogStart() 名,然后回车。

注意观察结果,看看是否有数字打印到 Shell 中或者控制台中(具体在哪显示取决于打印所定向的端口)。否则会打出错误号或者其他错误信息。看门狗超时执行看门狗中断服务程序,不超时执行用户程序。

另外,可以在 Shell 中通过输入一些 shell 命令(如 i,可以显示任务的基本信息)或者启动 WindView 等调试工具观察任务执行的情况,比如任务的优先级、堆栈、ID 号是否正常,任务之间的交互是否正常,有没有错误信息打印出来。

7. 思考题

- (1) 软件看门狗和硬件看门狗的优缺点。
- (2) 考虑设计退出任务的程序 wdogExit()。

6.5 实验 5 wind 内核功能

1. 实验目的

结合前面的几个实验,测试 wind 内核提供的系统调用。

2. 实验内容

结合前面几个关于 VxWorks 的基本实验,运用相关的系统函数,通过编写程序实现任务的调度,测试了 wind 内核中一些不同功能的组件的使用。其中包括信号量的同步、互斥原语的使用、任务控制中任务挂起和恢复等控制功能的使用,消息队列的使用,看门狗在任务中定时情况下的应用。

3. 预备知识

- (1) 信号量的相关知识;
- (2) 看门狗的相关知识;
- (3) 任务控制的相关知识;
- (4) 消息队列的相关知识;

(5) 与任务(task)相关函数的使用。

VxWorks 内核的任务管理提供了动态创建、删除和控制任务的功能,具体实现通过如下一些系统调用(关于以下函数的具体使用可以参照第 4 章):

- taskSpawn():创建(产生并激活)新任务。
- taskInit():初始化一个新任务。
- taskActivate():激活一个已初始化的任务。
- taskDelete():中止指定任务并自由内存(仅任务堆栈和控制块)。
- taskSuspend():悬置一个任务。
- taskResume():恢复一个任务。
- taskRestart():重启一个任务。
- taskDelay():延迟一个任务。

4. 实验设备和工具

- (1) 主机一台;
- (2) Win2000/XP 操作系统;
- (3) Tornado2.2 for arm 集成开发环境。

5. 实验原理

为了使用并且体会实验内容中提到的一些内核功能,在本实验中设置了两个任务,并为它们设置了不同的优先级。其中,高优先级任务在运行时需要用到一些系统暂时无法提供的资源,因此,高优先级的任务会因为缺乏资源而阻塞。与此同时,低优先级任务将会从高优先级的任务手中获得处理机的控制权,它在执行任务的过程中同时释放一些高优先级任务等待的资源,这样高优先级任务又可以运行。用这种方法,高优先级的任务和低优先级的任务交替执行,就会体现 VxWorks 系统中任务上下文切换、任务的调度、任务就绪队列以及定时器队列的变换等工作过程。因此,可以通过 Tornado 的内置工具来观察和体会这些工作过程,从而达到对 wind 内核功能的测试的目的。

测试这些内核功能的原因是因为大多应用场合经常用到同步、互斥、任务控制、任务间通信以及定时器的功能。这些功能不仅是具体应用的主要组成部分,也是 VxWorks 操作系统本身的重要组成部分。另外,熟练地掌握这些基本功能也是学好 VxWorks 的关键。

6. 实验步骤

- (1) 建立一个 downloadable 工程
- (2) 在新建的工程中添加.c 文件
- (3) 编写主程序

程序中高优先级任务和低优先级任务之间交互的流程如下:

- ① 主程序创建信号量对象;
- ② 高优先级任务分配信号量,同时获得此信号量;

- ③ 高优先级任务再次尝试获得此信号量,从而发生等待;
- ④ 低优先级任务分配信号量,以唤醒刚刚挂起的高优先级任务;
- ⑤ 高优先级任务发送消息到消息队列中,同时获得此消息;
- ⑥ 高优先级任务再次尝试从消息队列中获得消息,从而发生等待;
- ⑦ 低优先级任务发送消息到消息队列,以唤醒刚刚挂起的高优先级任务;
- ⑧ 启动 wdstart()。

编程思想:在本实验中设置了两个任务,并为它们设置了不同的优先级。其中,高优先级任务在运行时需要用到一些系统暂时无法提供的资源。因此,高优先级的任务会因为缺乏资源而阻塞。与此同时,低优先级任务将会从高优先级的任务手中获得处理机的控制权。它在执行任务的过程中同时释放一些高优先级任务等待的资源。这样高优先级任务又可以运行。用这种方法,高优先级的任务和低优先级的任务交替执行。设计两个任务入口函数为 tHighPrior()、tLowPrior()。

```
/* 定义 */
消息队列、时钟、任务优先级的相关参数
/* 函数声明 */
void tHighPrior (para.);
void tLowPrior (para.);

/* 编写主函数 */
/ *****
* windDemo - 用于产生子任务的父任务
* 此任务调用 taskHighPri()和 taskLowPri()开始实际测试操作,然后将自己挂起
* 任务恢复由低优先级任务完成
***** /

void tmain
(
/* 创建子任务使用的对象,包括消息队列、信号量和看门狗定时器 */
msgQId  = msgQCreate (MAX_MSG, MSG_SIZE, MSG_Q_FIFO);
semId   = semBCreate (SEM_Q_PRIORITY, SEM_FULL);
wdId    = wdCreate ();

/* 子任务开始执行内核例程 */
FOREVER
{}
```

```
LOCAL void tHighPrior
```

```
{
```

```
for (.....)
```

```
{
```

(1) 获取并给出信号量——不包括任务的上下文切换

如果信号量不可用则发生任务上下文切换,挂起任务

(2) 发送消息

若阻塞在消息队列上,则等待低优先级任务发来的消息

当真正有消息到来时,便将其打印

(3) 测试看门狗定时器

```
}
```

```
}
```

```
LOCAL void tLowPrior
```

```
{
```

```
for (.....)
```

```
{
```

(1) 释放信号量,激活高优先级的信号量

(2) 发送消息

(3) 返回主函数

```
}
```

(4) 编译、下载并运行程序

① 编译程序

在所要编译的文件上单击鼠标右键,将弹出对话框,选择源代码文件进行编译。

- 如果程序通过编译,会弹出完成的对话框,否则将报错;
- 生成.out 文件。

② 下载工程文件

下载.out 文件。如果符号表不统一或者不解析会有相应的提示信息。

③ 运行程序、观察结果

在 Shell 中输入本例程启动函数 tmain() 名,然后回车。

注意观察结果,看看是否有消息打印到 Shell 中或者控制台中(具体在哪显示取决于打印所定向的端口)。否则会打出错误号或者其他错误信息。观察高优先级的任务和低

优先级的任务是否交替执行,是如何访问临界资源的。

另外,可以在 Shell 中通过输入一些 shell 命令(如 i, 可以显示任务的基本信息)或者启动 WindView 等调试工具观察任务执行的情况,比如任务的优先级、堆栈、ID 号是否正常,任务之间的交互是否正常,有没有错误信息打印出来。

7. 思考题

频繁的任务切换带来的弊端,有没有改进的办法?

6.6 实验6 中断处理

1. 实验目的

通过实验例程,熟悉 Tornado 中各种工具的使用。

2. 实验内容

理解实验例程,通过 Tornado 中的 Shell, Brower, CrossWind 工具观察实验例程运行过程中系统的变化。

3. 预备知识

(1) WindSh: Tornado 外壳

WindSh 是一个驻留在主机内的 C 语言解释器,通过它可运行下载到目标机上的所有函数,包括 VxWorks 和应用函数。Tornado 外壳还能解释常规的工具命令语言 Tcl。

(2) Browser 浏览器

Tornado 浏览器可查看内存分配情况、系统目标(如任务、消息队列、信号量等)。这些信息可周期性地更新。

(3) CrossWind: 源码级调试器

源码级调试器 CrossWind 提供了图形和命令行方式来调试,可进行指定任务或系统级断点设置、单步执行、异常处理。

4. 实验设备和工具

(1) 主机一台;

(2) Win2000/XP 操作系统;

(3) Tornado2.2 for arm 集成开发环境。

5. 实验原理

本实验设计了 3 个任务,分别用于模拟中断服务程序、数据处理和打印信息。

当有数据到来的时候,任务 1(中断服务程序)用来分配一个信号量,并通过中断来控制任务 2(数据处理)的执行。任务 2 得到信号量后对接收到的数据进行处理,然后将处理结果告知任务 3(打印信息)。最后由任务 3 打印出相关信息。

本实例处理流程如图 6-7 所示。

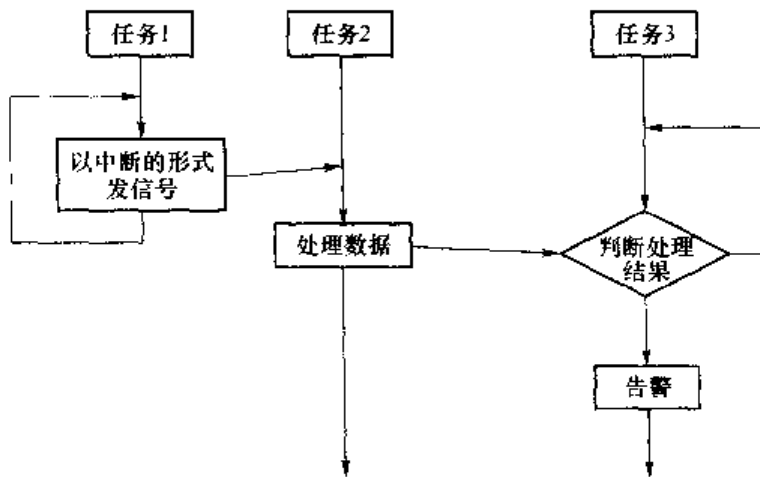


图 6-7 实例处理流程

6. 实验步骤

- (1) 建立一个 downloadable 工程
- (2) 设计主程序

编程思想：本实验设计了 3 个任务，分别用于模拟中断服务程序的函数 void tIS (void)、数据处理和打印信息的函数 void tDataprocess(void)、void tPrinter(void)。

入口函数 progStart(此函数用于启动其他例程)

progStart (void)

```

{
    (1) 创建信号量
    (2) 启动各个任务,设置各个任务的优先级
}
    
```

任务一

void tIS(void)

```

{
    while(任务启动){
        接收数据
    else
        等待有效数据

        semGive();/* 利用信号量和延迟保证临界数据资源不被抢占 */
        taskDelay();
    }
}
    
```

```

    }
    改变任务状态
    释放信号量
}

```

任务二

```

void tDataprocess(void){
    while(.....){
        获取信号量
        while(有数据)
        {
            处理数据
        }
        释放信号量
    }
    挂起任务
}

```

任务三

```

void tPrinter(void)
{
    while(任务启动)
    {
        if(不符合条件)
            printf("WARNING;HOT! \n");
        else if(符合条件)
            printf("OK\n");
    }
}

```

(3) 编译、下载、运行程序

① 编译程序

在所要编译的文件上单击鼠标右键编译文件。

- 如果程序通过编译,会弹出完成的对话框,否则将报错;
- 在弹出的菜单中选择编译生成.out 文件。

② 下载工程文件

下载.out 文件到 VxSim 中。如果有符号表不统一的情况发生,下载的时候会有相

应的提示信息。正常没有任何额外信息。

③ 运行结果

在 Shell 中输入启动函数名:progStart。不正确的执行会打印出相应的错误信息和错误号。

④ 观察、分析运行结果

注意观察结果,看看是否有消息打印到 Shell 中或者控制台中(具体在哪显示取决于打印所定向的端口)。如果符合条件,将打印出字符串“OK”,否则打印出字符串“WARNING:HOT!”。

另外,可以在 Shell 中通过输入一些 shell 命令或者启动 WindView 等调试工具观察任务执行的情况,比如任务的优先级、堆栈、ID 号是否正常,任务之间的交互是否正常,有没有错误信息打印出来。

7. 思考题

(1) 用 VxWorks 系统中的看门狗定时器代替例程中的中断服务程序是否可以? 可以的话,程序应如何改写?

(2) 总结 Shell,Browser,CrossWind 等 Tornado 工具在程序调试过程中的应用。

6.7 实验 7 Socket 通信

1. 实验目的

通过双向传输实验进一步理解 VxWorks 实时操作系统的基本概念和特点,进一步掌握多任务调度的相关知识和技巧。通过实验,学习 VxWorks 嵌入式实时操作系统,掌握嵌入式实时系统的基本原理和开发过程,熟悉 Tornado 的开发环境,掌握使用 C 语言在 Tornado 上编制基于 VxWorks 操作系统应用层程序的方法。掌握使用 Tornado 提供的工具对所编程序进行调试与测试的方法。使用 TCP/IP 协议在 VxWorks 操作系统上实现多机间同时进行收发双工通信。最后达到运用嵌入式系统进行开发项目的目的。

2. 实验内容

使用 TCP/IP 协议在 VxWorks 操作系统上实现多机同时进行收发双工通信。

3. 预备知识

(1) Socket 定义

Socket(套接字)是通过标准的 Unix 文件描述符和其他程序通信的一个方法。每一个套接字都用一个半相关描述——{协议,本地地址、本地端口}——来表示。一个完整的套接字则用一个相关描述——{协议,本地地址、本地端口、远程地址、远程端口}。每一个套接字都有一个本地的由操作系统分配的唯一的套接字号。

(2) Socket 类型

① 流式 Socket(SOCK_STREAM)

流式套接字提供可靠的、面向连接的通信流；它使用 TCP 协议，从而保证了数据传输的正确性和顺序。

② 数据报 Socket(SOCK_DGRAM)

数据报套接字定义了一种无连接的服务，数据通过相互独立的报文进行传输，是无序的，并且不保证可靠、无差错。它使用数据报协议 UDP。

③ 原始 Socket

原始套接字允许对底层协议如 IP 或 ICMP 直接访问，它功能强大但使用较为不便，主要用于一些协议的开发。

(3) Socket 中的数据结构

- ```

① struct sockaddr { //用于存储套接字地址
 unsigned short sa_family; //地址类型
 char sa_data[14]; //14 字节的协议地址
};

② struct sockaddr_in{ //in 代表 internet
 short int sin_family; //internet 协议族
 unsigned short int sin_port; //端口号,必须是网络字节顺序
 struct in_addr sin_addr; //internet 地址,必须是网络字节顺序
 unsigned char sin_zero; //添 0(和 struct sockaddr 一样大小)
};

③ struct in_addr{
 unsigned long s_addr;
};

```

## (4) 网络字节顺序及其转换函数

## ① 网络字节顺序

每一台机器内部对变量的字节存储顺序不同，而网络传输的数据是一定要统一顺序的。所以对内部字节表示顺序与网络字节顺序不同的机器，一定要对数据进行转换，从程序的可移植性要求来讲，就算本机的内部字节表示顺序与网络字节顺序相同也应该在传输数据以前先调用数据转换函数，以便程序移植到其他机器上后能正确执行。到底转换还是不转换是由系统函数自己来决定的。

## ② 有关的转换函数

- unsigned short int htons(unsigned short int hostshort): 主机字节顺序转换成网络字节顺序,对无符号短型进行操作 4 bytes。
- unsigned long int htonl(unsigned long int hostlong): 主机字节顺序转换成网络字

节顺序,对无符号长型进行操作 8 bytes。

- unsigned short int ntohs(unsigned short int netshort):网络字节顺序转换成主机字节顺序,对无符号短型进行操作 4 bytes。
- unsigned long int ntohl(unsigned long int netlong):网络字节顺序转换成主机字节顺序,对无符号长型进行操作 8 bytes。

(注:以上函数原型定义在 netinet/in.h 里。)

#### (5) IP 地址转换

有 3 个函数将数字点形式表示的字符串 IP 地址与 32 位网络字节顺序的二进制形式的 IP 地址进行转换。

##### ① unsigned long int inet\_addr(const char \* cp)

该函数把一个用数字和点表示的 IP 地址的字符串转换成一个无符号长整型,如:

```
struct sockaddr_in ina
```

```
ina.sin_addr.s_addr = inet_addr("202.206.17.101")
```

该函数成功时,返回转换结果;失败时返回常量 INADDR\_NONE,该常量为 -1,二进制的无符号整数 -1 相当于 255.255.255.255,这是一个广播地址,所以在程序中调用 inet\_addr()时,一定要人为地对调用失败进行处理。由于该函数不能处理广播地址,所以在程序中应该使用函数 inet\_aton()。

##### ② int inet\_aton(const char \* cp,struct in\_addr \* inp)

此函数将字符串形式的 IP 地址转换成二进制形式的 IP 地址;成功时返回 1,否则返回 0,转换后的 IP 地址存储在参数 inp 中。

##### ③ char \* inet\_ntoa(struct in\_addr in)

将 32 位二进制形式的 IP 地址转换为数字点形式的 IP 地址,结果在函数返回值中返回,返回的是一个指向字符串的指针。

#### (6) 字节处理函数

Socket 地址是多字节数据,不是以空字符结尾的,这和 C 语言中的字符串是不同的。Linux 提供了两组函数来处理多字节数据,一组以 b(byte)开头,是和 BSD 系统兼容的函数;另一组以 mem(内存)开头,是 ANSI C 提供的函数。

以 b 开头的函数有:

##### ① void bzero(void \* s,int n)

将参数 s 指定的内存的前 n 个字节设置为 0,通常它用来将套接字地址清 0。

##### ② void bcopy(const void \* src,void \* dest,int n)

从参数 src 指定的内存区域拷贝指定数目的字节内容到参数 dest 指定的内存区域。

##### ③ int bcmp(const void \* s1,const void \* s2,int n)

比较参数 s1 指定的内存区域和参数 s2 指定的内存区域的前 n 个字节内容,如果相同则返回 0,否则返回非 0。

#### 4. 实验设备和工具

两台或者多台 IRDCv03 试验箱(连接在同一个局域网内),一台主机,Tornado 开发平台。

#### 5. 实验步骤

##### (1) 设计主程序

##### ① 多机之间的通信架构

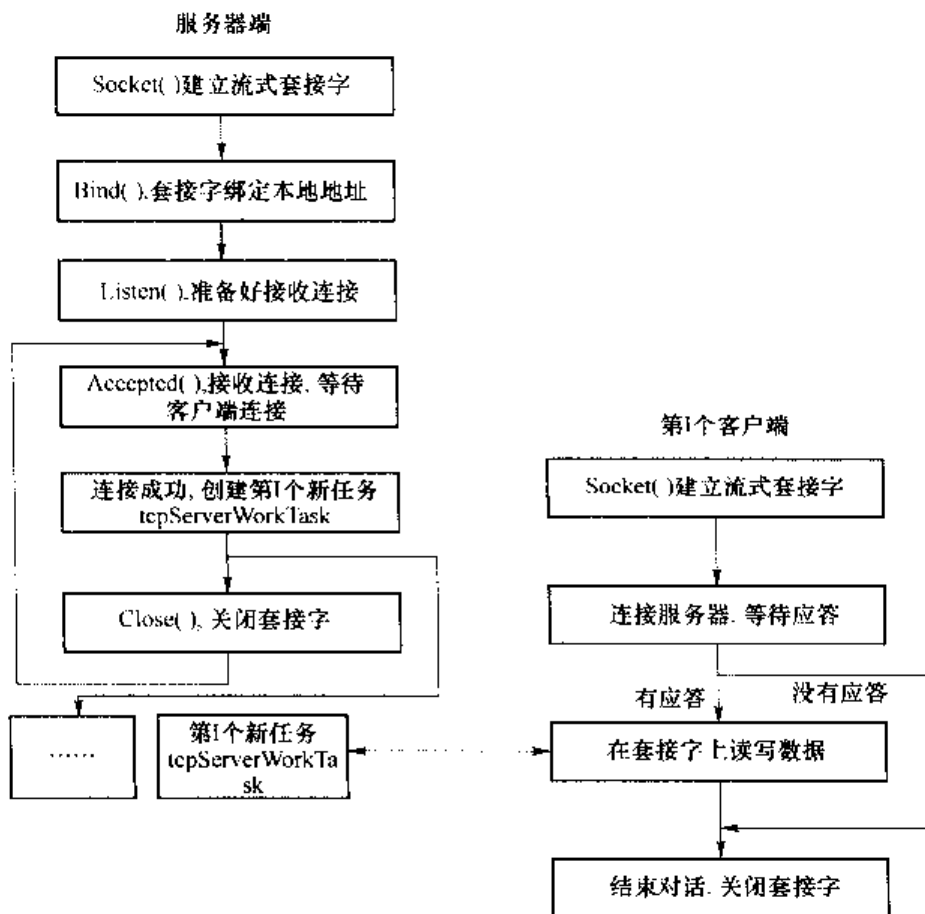


图 6-8 Socket 通信 Server/Client 架构

##### ② 编程思想

主要的设计流程可以参照图 6-8,底层的网卡驱动安装好后,可以调用标准的 Socket 套接字通过 TCP/IP 协议进行通信。服务器端等待服务,一旦有客户端的通信请求,如果通信协议符合,则建立连接,进行通信。

##### ③ 源程序

```
/* tcpClient.c - TCP client example */
```

```
/* TcpClient 是下载到 Client 端的程序,其完成的功能是建立连接,通过套节字进
```

```

行数据传输,传输的数据包括信息,是否回应。出错,返回错误信息 */
#include "vxWorks.h"
#include "sockLib.h"
#include "inetLib.h"
#include "stdioLib.h"
#include "strLib.h"
#include "hostLib.h"
#include "ioLib.h"
/*****
#include "tcpExample.h"
*****/
STATUS tcpClient (char * serverName) /* Server 机的 IP 地址 */
{
 struct request myRequest; /* 定义的结构体,保存信息和回复
 请求以及信息长度 */
 struct sockaddr_in serverAddr; /* Server 端的套节字地址 */
 char replyBuf[REPLY_MSG_SIZE]; /* 保存回复信息的数
 组 */
 char reply; /* 回复请求 */
 int sockAddrSize; /* Server 端的套节字地址结构
 体的长度 */
 int sFd; /* 保存套节字的设备 */
 int mlen; /* 信息长度 */
 char chRequest[500];

 /* 建立套节字,用于在两机之间进行数据传输 */
 if ((sFd = socket (AF_INET, SOCK_STREAM, 0)) == ERROR)
 {
 perror ("socket");
 return (ERROR);
 } /* 套节字建立失败,返回错误信息 */

 sockAddrSize = sizeof (struct sockaddr_in);
 bzero ((char *) &serverAddr, sockAddrSize);

```

```

serverAddr.sin_family = AF_INET;
serverAddr.sin_len = (u_char) sockAddrSize;
serverAddr.sin_port = htons (SERVER_PORT_NUM);
if (((serverAddr.sin_addr.s_addr = inet_addr (serverName)) == ERROR) &&
 ((serverAddr.sin_addr.s_addr = hostGetByName (serverName)) == ERROR))
{
 perror ("unknown server name");
 close (sFd);
 return (ERROR);
} /* 服务端不存在,返回错误信息 */

if (connect (sFd, (struct sockaddr *) &serverAddr, sockAddrSize) == ERROR)
{
 perror ("connect");
 close (sFd);
 return (ERROR);
} /* 连接客户机与目标机,如果失败返回错误信息 */
/* 修改的语句 */
memset((char *)&myRequest, '/0', 500);
printf ("Message to send: \nI am Client 1.1...", "");
mlen = read (STD_IN, chRequest, REQUEST_MSG_SIZE);
/* 从输入设备检测保存到 myRequest.message 中 */
sprintf(myRequest.message, "I am Client ... %s", chRequest);
myRequest.msgLen = strlen(myRequest.message) + 1;
/* myRequest.message[myRequest.msgLen] = ""; */
printf ("Would you like a reply (Y or N): \n");
read (STD_IN, &reply, 1); /* 从输入设备检测是否要求服务器端回应 */
switch (reply)
{
 case 'y':
 case 'Y': myRequest.reply = TRUE;
 break;
 default: myRequest.reply = FALSE;
 break;
} /* 如果输入 Y 或 y 则 myRequest.reply = TRUE,反之为 FALSE */

```

```

 if (write (sFd, (char *) &myRequest, sizeof (myRequest)) == ERROR)
 {
 perror ("write");
 close (sFd);
 return (ERROR);
 } /* 将 message 和 reply 保存到 sFd 中 */

 /* 以上过程完成将数据传到服务器端 */

 if (myRequest.reply) /* if expecting reply, read and display it */
 {
 if (read (sFd, replyBuf, REPLY_MSG_SIZE) < 0)
 {
 perror ("read");
 close (sFd);
 return (ERROR);
 } /* 如果服务器端接收到数据, 执行下一句, 反之报错 */
 printf ("MESSAGE FROM SERVER:\n %s\n", replyBuf);
 }

 close (sFd);
 return (OK);
}

/*****
/* tcpServer.c - TCP server example */
*****/

#include "vxWorks.h"
#include "sockLib.h"
#include "inetLib.h"
#include "taskLib.h"
#include "stdioLib.h"
#include "strLib.h"
#include "ioLib.h"
#include "fioLib.h"

```

```

#include "tcpExample.h"

VOID tcpServerWorkTask (int sFd, char * address, u_short port);
VOID tcpServeraddTask();

STATUS tcpServer (void)
{
 if(taskSpawn("Server", SERVER_WORK_PRIORITY, 0, SERVER_STACK_SIZE,
 (FUNCPTR) tcpServeraddTask, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0) == ERROR)
 {
 /* 调用函数,发起一个新的任务,错误返回错误信息 */
 printf("Server task created error! \n");
 }
 return (OK);
}

VOID tcpServeraddTask()
{
 struct sockaddr_in clientAddr;
 int ix = 0; /* counter for work task names */
 char workName[16]; /* 建立的任务名称 */
 int sockAddrSize;
 int newFd;
 struct sockaddr_in serverAddr;
 int sFd;

 sockAddrSize = sizeof (struct sockaddr_in);
 bzero ((char *) &serverAddr, sockAddrSize);
 serverAddr.sin_family = AF_INET;
 serverAddr.sin_len = (u_char) sockAddrSize;
 serverAddr.sin_port = htons (SERVER_PORT_NUM);
 serverAddr.sin_addr.s_addr = htonl (INADDR_ANY);

```

```

/* 建立套节字 */
if ((sFd = socket (AF_INET, SOCK_STREAM, 0)) == ERROR)
{
 perror ("socket");
 return (ERROR);
}

if (bind (sFd, (struct sockaddr *) &serverAddr, sockAddrSize) == ERROR)
{
 perror ("bind");
 printf("error in Server Bind");
 close (sFd);
 return (ERROR);
}

if (listen (sFd, SERVER_MAX_CONNECTIONS) == ERROR)
{
 perror ("listen");
 close (sFd);
 return (ERROR);
} /* 建立失败返回错误信息 */

sockAddrSize = sizeof (struct sockaddr_in);

FOREVER
{
 if ((newFd = accept (sFd, (struct sockaddr *) &clientAddr,
 &sockAddrSize)) == ERROR)
 { /* 判断是否有信息传输,若建立失败,返回错误信息 */
 perror ("accept");
 close (sFd);
 return (ERROR);
 }

 sprintf (workName, "tTcpWork %d", ix++);
 if (taskSpawn(workName, SERVER_WORK_PRIORITY, 0, SERVER_STACK_SIZE,

```



```

(FUNCPTR) tcpServerWorkTask, newFd,
(int) inet_ntoa (clientAddr.sin_addr), ntohs (clientAddr.sin_port),
0, 0, 0, 0, 0, 0, 0) == ERROR)
{
 /* 调用函数,发起一个新的任务,错误返回错误信息 */
 perror ("taskSpawn");
 close (newFd);
}
}

}

VOID tcpServerWorkTask /* 被调用的函数 */
/* 在建立的循环中,如果套节字的字长大于0,读取数据,反之退出 */
(
 int sFd,
 char * address,
 u_short port
)
{
 struct request clientRequest;
 int nRead;
 char replyMsg[] = "I am Server1.1...I have received your message! Replay ...";
 int mlen;

 while ((nRead = fioRead (sFd, (char *) &clientRequest,
 sizeof (clientRequest))) > 0)
 {
 printf ("MESSAGE FROM CLIENT (Internet Address %s, port %d):\n%s\n",
 address, port, clientRequest.message);
 free (address);
 if (clientRequest.reply)
 {
 if (write (sFd, replyMsg, sizeof (replyMsg)) == ERROR)
 perror ("write");
 }
 }
}

```

```

 }
 }
 if (nRead == ERROR)
 perror ("read");
 close (sFd);
}

```

## (2) 下载运行程序

① 将客户端程序和服务器端程序分别下载到不同的试验板中。如果条件不容许,客户端程序可以下载到 Flash 中直接运行,服务器端程序可以在线下载,这样比较方便观看通信的信息。当然,如果条件容许,客户端程序在线下载更好,可以同时监视客户端和服务端端的通信状况。

② 运行服务器端程序和客户端程序。

③ 观察运行结果。注意观察结果,看看是否有客户端信息打印到 Shell 中或者控制台中(具体在哪显示取决于打印所定向的端口)。否则会打出错误号或者其他错误信息。操作正确的话,可以在服务器端看到“MESSAGE FROM CLIENT (IP 地址,端口号)”等字符信息。

另外,可以在 Shell 中通过输入一些 shell 命令(如 i,可以显示任务的基本信息)或者启动 WindView 等调试工具观察任务执行的情况,比如任务的优先级、堆栈、ID 号是否正常,任务之间的交互是否正常,有没有错误信息打印出来。

## 6. 思考题

(1) 双机通信唯一的问题是,实验说明所给的代码中 SERVER 起来以后就关不掉,即没有退出机制。用 CTRL+C 强行退出以后,原来绑定的 Socket 资源没有释放,后面的网络操作函数会出现问题。如何解决这个问题?

(2) 画出通过 UDP 方式通信的 Server/Client 架构图,并且编写相应的代码。

(3) 在缺乏试验板资源的情况下,本试验也可以通过 ULIP 适配器在多台 PC 机之间仿真实现,查阅相关资料,试实现之。

# 6.8 实验 8 虚拟内存设备驱动

## 1. 实验目的

- (1) 了解设备驱动程序的编写方法和过程;
- (2) 初步掌握设备安装、打开、读写、I/O 控制等函数的编写;
- (3) 利用基本函数可以编写简单设备的驱动程序。

## 2. 实验内容

- (1) 通过给出的例程,了解虚拟内存设备的驱动程序的编写过程和方法;
- (2) 在掌握后,模拟编写磁盘驱动程序。

## 3. 预备知识

- (1) VxWorks 下设备驱动程序的编写规范;
- (2) 设备驱动程序的 7 个基本函数;
- (3) C 语言的基础。

## 4. 实验设备和工具

- (1) 主机一台;
- (2) Win2000/XP 操作系统;
- (3) Tornado2.2 for arm 集成开发环境。

## 5. 实验原理

本实验以虚拟内存设备为例,介绍了设备驱动程序的编写方法。

VxWorks 系统的 I/O 系统可以通过虚拟内存驱动程序将硬件系统的内存当作虚拟 I/O 设备直接访问。在创建虚拟 I/O 设备时,指定了内存的位置和大小。因此,当需要在 VxWorks 引导期间保存关键的数据或者需要在 CPU 之间共享数据时,使用虚拟 I/O 设备是非常方便的。此外,虚拟内存设备驱动程序可以将一些文件编译为 VxWorks 二进制映像,然后将它们挂接到一个文件系统上。

本实验中介绍的虚拟内存驱动程序利用 I/O 系统调用为硬件系统中字节流读写提供了一种简单而又高效的方法。也可以利用本程序实现一个简单的只读文件系统。

本例中的驱动程序,有 4 个函数可以直接调用。分别是函数 mem\_Initial()用于初始化此驱动程序,函数 mem\_Create()和 mem\_CreateDir()用于创建设备,而函数 mem\_Delete()则用来删除设备。

## 6. 实验步骤

- (1) 创建工程(同上面几个例子)
- (2) 设计主程序

### ① 用到的头文件

```
#include "VxWorks.h"
#include "ioLib.h"
#include "iosLib.h"
#include "stat.h"
#include "dirent.h"
#include "memLib.h"
#include "errnoLib.h"
#include "string.h"
```

```
#include "stdlib.h"
#include "stdio.h"
#include "memDrv.h"
```

## ② 用到的结构体定义

### a. 内存设备描述符

```
typedef struct
{
 DEV_HDR devHdr;
 MEM_DRV_DIRENTRY dir; /* 虚拟内存设备的内容 */
 int allowOffset; /* 允许文件以偏移量打开的标志 */
}mem_DEV_DESC;
```

### b. 内存文件描述符

```
typedef struct
{
 MEM_DEV * pDeceive; /* 文件的内存设备 */
 MEM_DRV_DIRENTRY * pDir; /* 文件的目录入口 */
 int offset; /* 当前位置 */
 int mode; /* 模式包括 O_RDONLY, O_WRONLY, O_RDWR */
}mem_FILE_DESC;
```

## ③ 用到的函数

```
LOCAL int memDrvNum;
LOCAL MEM_DRV_DIRENTRY * memFindFile();
LOCAL mem_FILE_DESC * memOpen();
LOCAL STATUS memFileStartGet();
LOCAL int memRead();
LOCAL int memWrite();
LOCAL int memClose();
LOCAL STATUS memIoctl();
extern STATUS mem_initial(void);
extern STATUS mem_Create(char * name, char * base, int length);
extern STATUS mem_CreateDir(char * name, MEM_DRV_DIRENTRY * files, int numFiles);
extern STATUS mem_Delete(char * name);
```

## ④ 主要函数的编程思想

### a. mem\_Initial()函数

此函数的任务是初始化内存驱动程序。在调用此驱动的其他函数之前,必须首先调

用此函数。返回值:正常时返回 OK,如果 I/O 系统不能安装驱动返回 ERROR。

```
STATUS memDrv(void)
{
 if(memDrvNum>0)
 return(OK); /* 驱动已经安装,返回 OK */
 memDrvNum = isoDrvInstall((FUNCPTR)memOpen,(FUNCPTR)NULL,
(FUNCPTR)memOpen,memClose,memRead,memWrite,memIoctl);
 return (memDrvNum == ERROR? ERROR:OK);
}
```

#### b. mem\_Create()函数

此函数用于创建一个内存设备,包含一个单一的文件,分配给设备的内存是绝对内存。它起始于“base”,内存的大小由参数“length”指定。返回值:正常情况下返回 OK,如果内存不足或者 I/O 系统不可添加设备返回 ERROR,ERRNO 为 S\_ioLib\_NO\_DRIVER。

```
STATUS mem_Create(char * name, char * base, int length)
{
 STATUS status;
 FAST MEM_DEV * pMemDv;

 if(memDrvNum<1)
 {
 errnoSet(S_ioLib_NO_DRIVER);
 return(ERROR);
 }

 if((pMemDv = (MEM_DEV *)calloc(1,sizeof(MEM_DEV))) == NULL)
 return(ERROR);

 pMemDv->dir.name = "";
 pMemDv->dir.base = base;
 pMemDv->dir.pDir = NULL;
 pMemDv->dir.length = length;
 pMemDv->allowOffset = 1;

 /* 声明字节、字以及长字方式的访问 */
```

```

 status = iosDevAdd((DEV_HDR *)pMemDv, name, memDrvNum);

 if(status == ERROR)
 free((char *)pMemDv);
 return (status);
}

```

#### c. mem\_CreateDir()函数

此函数用于内存设备的目录创建函数。返回值:正常情况下返回 OK,如果内存不足或者 I/O 系统不可添加设备则返回 ERROR,ERRNO 为 S\_ioLib\_NO\_DRIVER。

```

STATUS mem_CreateDir(char * name, MEM_DRV_DIRENTRY * files, int numFiles)
{
 STATUS status;
 FAST MEM_DEV * pMemDv;

 if (memDrvNum < 1)
 {
 errnoSet(S_ioLib_NO_DRIVER);
 return(ERROR);
 }

 if((pMemDv = (MEM_DEV *)calloc(1, sizeof(MEM_DEV))) == NULL)
 return (ERROR);

 pMemDv->dir.name = "";
 pMemDv->dir.base = NULL;
 pMemDv->dir.pDir = files;
 pMemDv->dir.length = numFiles;
 pMemDv->allowOffset = 0; /* 对于打开的文件,偏移量直接赋 0 */
 status = iosDevAdd((DEV_HDR *)pMemDv, name, memDrvNum);
 if(status == ERROR)
 free((char *)pMemDv);

 return (ERROR);
}

```

## d. mem\_Delete()函数

此函数用于删除一个内存设备。返回值:正常情况下返回 OK,如果设备不存在则返回 ERROR。

```
STATUS mem_Delete(char * name)
{
 DEV_HDR * pDevHdr;

 /* 获取内存设备名字所对应设备的指针 */
 if((pDevHdr = iosDevFind(name, NULL)) == NULL)
 return (ERROR);
 /* 在 VxWorks 系统的 I/O 系统删除这个设备 */
 iosDevDelete(pDevHdr);
 /* 释放设备指针 */
 free((MEM_DEV *)pDevHdr);
 return (OK);
}
```

## c. memRead 函数

此函数用来读取一个内存文件。返回值:正常情况下返回读取的字节数,这个值可能比请求读取的字节数小。如果文件是以只写模式打开则返回 ERROR,如果文件中没有数据则返回 0,其 ERRNO 是 EINVAL 和 EISDIR。

```
LOCAL int memRead(MEM_FILE_DESC * pfd, char * buffer, int maxbytes)
{
 MEM_DRV_DIRENTRY * pDir = pfd->pDir;
 int lLeft;
 /* 如果模式不对或者它是一个目录则返回错误 */
 if(pfd->mode == O_WRONLY)
 {
 errnoSet(EINVAL);
 return(ERROR);
 }
 if(pDir == NULL || pDir->pDir! = NULL)
 {
 errnoSet(EINVAL);
 return(ERROR);
 }
}
```

```

 }
 /* 计算还剩多少字节要读取 */
 lLeft = pDir->length - pfd->offset;
 if(lLeft<0) lLeft = 0;
 if(maxbytes>lLeft) maxbytes = lLeft;
 if(maxbytes>0)
 {
 bcopy(pDir->base + pfd->offset,buffer,maxbytes);
 pfd->offset += maxbytes;
 }

 return (maxbytes);
}

```

#### f. memWrite()函数

此函数用来写入一个内存文件。返回值:正常情况下返回已经写入的字节数,如果传递的字节数超出内存的边界或者打开文件的模式是只读模式则返回 ERROR,其 ERRNO 是 EINVAL 和 EISDIR。

```

LOCAL int memWrite(MEM_FILE_DESC * pfd, char * buffer, int nbytes)
{
 MEM_DRV_DIRENTRY * pDir = pfd->pDir; /* 指向文件信息的指针 */
 int lLeft;
 /* 如果模式不对或者它是一个目录则返回错误 */
 if(pfd->mode == O_RDONLY)
 {
 errnoSet(EINVAL);
 return(ERROR);
 }

 if(pDir == NULL || pDir->pDir != NULL)
 {
 errnoSet(EINVAL);
 return(ERROR);
 }

 /* 计算还剩多少字节要写入 */

```



```

lLeft = pDir->length - pfd->offset;
if(lLeft < 0) lLeft = 0;
/* 如果写入太多,对其进行裁减 */
if(nbytes > lLeft) nbytes = lLeft;

if(nbytes > 0)
{
 bcopy(buffer, pDir->base + pfd->offset, nbytes);
}

return (nbytes);
}

```

#### g. memIoctl()函数

此函数用于实现特定的设备控制功能(本例中实现的功能仅限于几项)。返回值:正常情况下返回 OK,如果搜索超出了内存的边界则返回 ERROR,其 ERRNO 可以是 EINVAL 和 S\_ioLib\_UNKNOWN\_REQUEST。

```

LOCAL STATUS memIoctl(pfd, function, arg)
FAST MEM_FILE_DESC * pfd;
FAST int function;
int arg;
{
 MEM_DRV_DIRENTRY * pDir = pfd->pDir; /* 指向文件信息的指针 */
 DIR * dirp;
 int isDir = 0;
 int status = OK;

 if(pDir == NULL)
 {
 errnoSet(EINVAL);
 status = ERROR;
 }
 else
 {
 isDir = (pDir->pDir != NULL);
 switch(function)
 {

```

```

case FIONREAD: /* 读取功能 */
 if(isDir)
 {
 errnoSet(EINVAL);
 status = ERROR;
 }
 else
 * ((int *)arg) = pDir->length - pfd->offset;
 break;

case FIOSEEK: /* 查找功能 */
 if(isDir || arg > pDir->length)
 {
 errnoSet(EINVAL);
 status = ERROR;
 }
 else
 pfd->offset = arg;
 break;

case FIOWHERE:
 if(isDir)
 status = 0;
 else
 status = pfd->offset;
 break;

case FIOGETFL:
 * ((int *)arg) = pfd->mode;
 break;

case FIOFSTATGET: /* 获取状态 */
 status = memFileStatGet(pfd, (struct stat *)arg);
 break;

```

```

 case FIOREADDIR: /* 目录读取功能 */
 dirp = (DIR *)arg;
 if(! isDir || dirp->dd_cookie >= pDir->length)
 {
 errnoSet(EINVAL);
 status = ERROR;

 }
 else
 {
 strcpy(dirp->dd_dirent.d_name, pDir->pDir[dirp->dd_cookie].name);
 ++ dirp->dd_cookie;
 }
 break;

 default:
 errnoSet(S_ioLib_UNKNOWN_REQUEST);
 status = ERROR;
 break;
 }
}

return (status);
}

```

### (3) 编译、下载、运行程序

#### ① 编译程序

在所要编译的文件上单击鼠标右键编译文件。

- a. 如果程序通过编译,会弹出完成的对话框,否则将报错;
- b. 在弹出的菜单中选择编译生成.out 文件。

#### ② 下载工程文件

下载.out 文件到目标机中。如果有符号表不统一的情况发生,下载的时候会有相应的提示信息,如果正常则没有任何额外信息。

#### ③ 运行结果

有4个函数可以直接调用。分别是函数 mem\_Initial()用于初始化此驱动程序,函数 mem\_Create()和 mem\_CreateDir()用于创建设备,而函数 mem\_Delete()则用来删除设备。不正确的执行会打印出相应的错误信息和错误号。

#### ④ 观察、分析运行结果

注意观察结果,看看是否有消息打印到 Shell 中或者控制台中(具体在哪显示取决于打印所定向的端口)。观察程序是如何初始化设备的,占用资源的情况如何。

另外,可以在 Shell 中通过输入一些 shell 命令或者启动 WindView 等调试工具观察任务执行的情况,比如任务的优先级、堆栈、ID 号是否正常,任务之间的交互是否正常,有没有错误信息打印出来。

#### 7. 思考题

根据对实验例程的理解,编写另一个设备驱动程序,如磁盘驱动程序、打印机等。

## 6.9 实验 9 任务软调度

### 1. 实验目的

了解任务软调度的概念、过程和优点。

### 2. 实验内容

- (1) 简单模拟任务软调度的过程;
- (2) 验证在挂接在任务下的进程相互间可以公平共享处理器资源。

### 3. 预备知识

- (1) 消息队列的应用;
- (2) 任务的调度方法;
- (3) VxWorks 中调试工具的使用;
- (4) 任务软调度的概念。

虽然 VxWorks 系统中提供了基于任务调度的灵活机制,但在许多具体的应用场合,不可能无限制地创建任务来处理复杂的应用,因为任务的数量过多,将造成操作系统的上下文切换过于频繁,这样实际上降低了系统 CPU 的利用率。

在任务的基础上引入“进程”的概念,利用进程完成任务中的单元事务。即在 VxWorks 系统调度任务执行的基础上,任务将处理器资源直接分配给进程来享用,在任务内实现“软调度”。

### 4. 实验设备和工具

- (1) 主机一台;
- (2) Win2000/XP 操作系统;
- (3) Tornado2.2 for arm 集成开发环境。

### 5. 实验原理

本例程的核心思想是让 VxWorks 的任务挂载在内核创建的消息队列上,当发现有消息时立即转入任务内的进程调用,查询进程消息队列内是否有消息存在。为此,在系统中创建

了 3 个任务和 4 个进程。第一个任务是 sendMSG\_Task, 它用于为应用系统中的其他任务和进程发送消息。第二个和第三个任务分别是 SCHETask1 和 SCHETask2, 创建这两个任务的目的是利用这两个任务调度内部进程运作。挂接在 SCHETask1 上的进程是 Process1 和 Process2, 而挂接在 SCHETask2 上的进程是 Process3 和 Process4。

为了验证挂接在任务下的进程相互间可以公平共享处理器资源, 将调度信息打印在终端上, 如果各个进程打印在终端上的调度频率数据大小相近, 或者随着进程被调度次数的增多, 其消息调度次数越来越相近, 那么可以判定进程之间实现了公平调度。

进程之间执行流程图如图 6-9 所示。

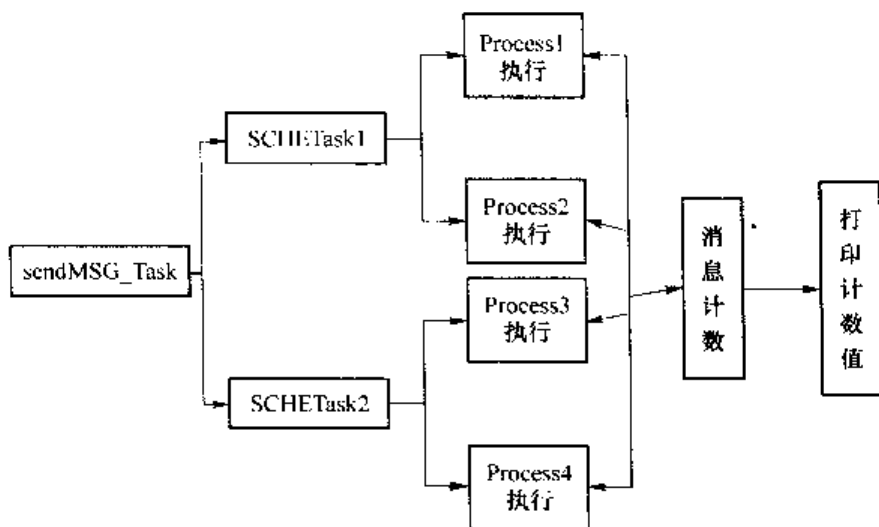


图 6-9 执行流程

## 6. 实验步骤

(1) 创建工程(同上面几个例子)

(2) 设计主程序

① 需要定义的 VxWorks 头文件

```

#include "VxWorks.h"
#include "msgQlib.h"
#include "taskLib.h"
#include "timers.h"
#include "string.h"
#include "stdio.h"
#include "stdlib.h"
#include "math.h"

```

② 需要定义的函数

```

void sendMSG_Task(void); /* 发送消息任务 */

```

```

void CreateTask(void); /* 程序入口函数 */
void ScheduleTask(int taskId); /* 调度任务 */
void Process1Entry(void); /* 各进程入口 */
void Process2Entry(void);
void Process3Entry(void);
void Process4Entry(void);

③ 需要定义的变量

int MaxMsgBytes; /* 消息最大字节数 */
int timeout; /* 等待消息的时间 */
int maxMsgs; /* 消息队列中的最大消息数 */
/* 下面定义的缓冲区用于接收和发送消息 */
char MSGbuffer[MAXMSG_LENGTH];
char MSGbufferA[MAXMSG_LENGTH];
char MSGbufferB[MAXMSG_LENGTH];
char MSGbuffer1[MAXMSG_LENGTH];
char MSGbuffer2[MAXMSG_LENGTH];
char MSGbuffer3[MAXMSG_LENGTH];
char MSGbuffer4[MAXMSG_LENGTH];
/* 消息的调度次数 */
unsigned long int Process1MSG;
unsigned long int Process2MSG;
unsigned long int Process3MSG;
unsigned long int Process4MSG;
/* 任务和进程使用的消息队列 */
MSG_Q_ID ScheduleTask1MSGId;
MSG_Q_ID ScheduleTask2MSGId;
MSG_Q_ID ScheduleProcess1MSGId;
MSG_Q_ID ScheduleProcess2MSGId;
MSG_Q_ID ScheduleProcess3MSGId;
MSG_Q_ID ScheduleProcess4MSGId;

```

#### ④ 主要函数编程思想

##### a. sendMSG\_Task 用于发送消息

其功能是周期性地将消息地送到任务或者进程等待的消息队列中。

```

void sendMSG_Task(void)
{

```

```

int maxMsgLength; /* 消息中允许的最大字节数 */
int options; /* 消息队列的选项 */

long loop_num;
int randnum;
maxMsgs = 64;
maxMsgLength = 1024;
options = MSG_Q_FIFO;

/* 任务 SCHETask1 使用的消息队列 */
ScheduleTask1MSGId = msgQCreate(maxMsgs,maxMsgLength,options);
if(ScheduleTask1MSGId == NULL)
{
 printf("ERROR occurs in task1 msgQCreate! \n");
 return;
}
/* 任务 SCHETask2 使用的消息队列 */(参考 SCHETask1 使用的消息队列)

/* 进程 Process1 使用的消息队列 */
ScheduleProcess1MSGId = msgQCreate(maxMsgs,maxMsgLength,options);
if(ScheduleProcess1MSGId == NULL)
{
 printf("ERROR occurs in Process1 msgQCreate! \n");
 return;
}
/* 进程 Process2 使用的消息队列 */
/* 进程 Process3 使用的消息队列 */
/* 进程 Process4 使用的消息队列 */
/* 消息发送循环 */
for (loop_num = 0;;loop_num++)
{
 /* 获取随机数 */
 /* 发送随机数 */
 /* 发送消息 */
 msgQSend(ScheduleProcessMSGId,MSGbuffer,sizeof(MSGbuffer),timeout,

```

```

MSG_PRI_NORMAL);
 randnum++;

 randnum = fmod(randnum,MOD_CONSTANT);
 if(randnum == 0)
 randnum = MOD_CONSTANT;
 strcpy(MSGbuffer,"randnum");
}

```

#### b. CreateTask 主函数

用于创建应用程序的所有任务。

```

void CreateTask(void)
{
 /* 用于发送消息的任务的初始化 */
 RetValue = taskSpawn("sdMSGTask",100,VX_PRIVATE_ENV,512,(FUNCPTR) sendMSG_Task,0,0,0,0,0,0,0,0,0,0,0);
 /* 调度任务的初始化 */
 RetValue = taskSpawn("SCHETask1",100,VX_PRIVATE_ENV,512,(FUNCPTR) ScheduleTask,1,0,0,0,0,0,0,0,0,0,0);
}

```

#### c. ScheduleTask 实现调度任务的功能

此函数实现的调度任务直接调度挂接在他下面的进程执行。调度任务的原则是实现进程的公平调度。

```

void ScheduleTask(int taskId)
{

 /* 如果任务 ID 为 1,则只调度 process1 和 process2 执行 */
 if(taskId == 1)
 {
 for(;;)
 {
 /* 读取消息队列 */
 revBytes = msgQReceive(ScheduleTask1MSGId,MSGbufferA,MaxMsgBytes,timeout);
 if(0 != revBytes)
 {
 revBytes = msgQReceive(ScheduleProcess1MSGId,MSGbuffer1,MaxMsgBytes,timeout);
 if(0 != revBytes)

```



```

 Process1Entry(); /* 运行进程 1 */

 revBytes = msgQReceive(ScheduleProcess2MSGId, MSGbuffer2, MaxMsgBytes,
 timeout);

 if(0 != revBytes)
 Process2Entry(); /* 运行进程 2 */
 }
 else
 printf("ERROR occurs in SCHETask1 Scheduling MSG\n");
 }
}

/* 如果任务 ID 为 2, 则只调度 process3 和 process4 执行 */
else if(tId == 2)
{

}

```

#### d. ProcessEntry 进程的入口

此进程用于计算消息的调度频率。

```

void ProcessEntry(void)
{
 msgnum = msgQNumMsgs(ScheduleProcess1MSGId);
 /* 遍历消息队列计算消息的调度次数 */
 for (loop = 0; loop < msgnum; loop++)
 {
 revBytes = msgQReceive(ScheduleProcess1MSGId, MSGbuffer1, MaxMsgBytes,
 timeout);
 /* 增加相应的计数器的值 */
 if(strcmp(MSGbuffer1, "1"))
 Process1MSG++;

 }

 /* 如果消息调度次数有所增加, 则打印到终端 */
 return;
}

```

### (3) 编译、下载、运行程序

#### ① 编译程序

在所要编译的文件上单击鼠标右键编译文件。

- 如果程序通过编译,会弹出完成的对话框,否则将报错;
- 在弹出的菜单中选择编译生成.out 文件。

#### ② 下载工程文件

下载.out 文件到 VxSim 中。如果有符号表不统一的情况发生,下载的时候会有相应的提示信息。如果正常则没有任何额外信息。

#### ③ 运行结果

在 Shell 中输入启动函数名:CreateTask。不正确的执行会打印出相应的错误信息和错误号。

#### ④ 观察、分析运行结果

注意观察结果,看看是否有消息打印到 Shell 中或者控制台中(具体在哪显示取决于打印所定向的端口)。观察任务 SCHETask1 和 SCHETask2 之间是如何交互的,进程之间有什么消息通信。

另外,可以在 Shell 中通过输入一些 shell 命令或者启动 WindView 等调试工具观察任务执行的情况,比如任务的优先级、堆栈、ID 号是否正常,任务之间的交互是否正常,有没有错误信息打印出来。

在 VxWorks Simulator 上观察运行结果并通过运行结果对进程调度的公平性等方面做出评价。程序运行大约需要 6 分钟。

### 7. 思考题

本例程在模拟任务软调度工程中存在的缺陷,应该从哪几方面进行改进。

## 6.10 实验 10 任务软调度的改进

### 1. 实验目的

进一步熟悉和理解任务软调度的方法。

### 2. 实验内容

在理解任务软调度概念的基础上,针对实验 9 中任务软调度程序的缺陷,进一步对程序做出改进,写出一个更高效的任务软调度程序。另外,参考实验例程,学习消息队列的创建。

### 3. 预备知识

- (1) 消息队列的应用;
- (2) 任务的调度方法;

(3) VxWorks 中调试工具的使用。

#### 4. 实验设备和工具

- (1) 主机一台;
- (2) Win2000/XP 操作系统;
- (3) Tornado2.2 for arm 集成开发环境。

#### 5. 实验原理

在实验 9 介绍的例程中,虽然可以实现任务内和任务间各个进程的调度平衡,但是它们的调度效率比较低,原因在于程序中调用操作系统中消息队列的系统调用过于频繁。

在本节的参考例程中,进程不使用 VxWorks 系统提供的消息队列。而是通过自己创建的简单消息队列完成消息的分配和管理,因而既可以实现任务内和任务间的各个进程的调度平衡,又具有较高的调度效率。

设计本例的目的是为了验证挂接在任务下的进程相互间可以公平共享处理器资源,并且具有较高的效率。将进度调度的频率打印在终端上,如果各个进程打印在终端上的频率大小相近,那么可以判定进程之间实现了公平调度。同时可以在根据输出窗口的打印频率判定其时间特性。

#### 6. 实验步骤

(1) 创建工程

(2) 设计主程序

① 需要用到的头文件

```
#include "VxWorks.h"
#include "msgQLib.h"
#include "taskLib.h"
#include "timers.h"
#include "string.h"
#include "stdio.h"
#include "stdlib.h"
#include "math.h"
```

② 需要用到的函数

```
/* 所有任务入口的声明 */
void sendMSG_Task(void);
void CreateTask(void);
void ScheduleTask(int taskId);

/* 所有进程入口的声明 */
void Process1Entry(void);
```

```
void Process2Entry(void);
void Process3Entry(void);
void Process4Entry(void);

/* 特定应用函数的声明 */
int DataModel(int data1);
void PostMessageOnProcess(int processNo, int messageNo);
int MessageValid(int revMsg, int processNo);
void PrintfStatistic(int Process, int ProcessId);
int GetMessage(int processNo);
```

### ③ 消息结构的定义

```
typedef struct Tag_Message
{
 /* messageNo(1 - 64)发送给 process1 */
 /* messageNo(65 - 128)发送给 process2 */
 /* messageNo(129 - 192)发送给 process3 */
 /* messageNo(193 - 256)发送给 process4 */

 char messageNo;
 char messagePara1;
 char messagePara2;
 char messagePara3;
}T_Message;

/* 定义发送给进程的消息单元的定义 */
typedef struct Tag_MessageElement
{
 char messageNo;
 char pPara;
 int prevMessage;
 int nextMessage;
 int IsUsed;
 int prevIdle; /* 闲态队列中的上一元素 */
 int nextIdle; /* 闲态队列中的下一元素 */
}T_MessageElement;
```

/\* 数据结构 T\_PCD 用于保存本应用所创建进程的控制管理数据  
实际上,它的大部分内用时管理消息队列的字段 \*/

```
typedef struct Tag_PCD
{
 /* 管理有效消息队列的字段 */
 int messageSum;
 int firstMessage;
 int lastMessage;

 /* 管理闲态消息队列的字段 */
 int idleSum;
 int firstIdle;
 int lastIdle;
 T_MessageElement messageArray[MAXMSG_NUM];
}T_PCD;

T_PCD ProcessPCDArray[PROCESSNUM];
```

#### ④ 主要函数编程思想

##### a. sendMSG\_Task 用于发送消息

此函数的功能是周期性地将消息地送到任务等待的消息队列中。

```
void sendMSG_Task(void)
{
 int options; /* 消息队列选项 */
 long loop_num;
 int randnum;

 char MSGbuffer[4];
 T_Message sendMessage;

 sendMessage.messageNo = 0;
 sendMessage.messagePara1 = 0;
 sendMessage.messagePara2 = 0;
 sendMessage.messagePara3 = 0;
```

```

options = MSG_Q_FIFO;
/* SCHETASK 使用的消息队列 */
ScheduleTaskMSGQId = msgQCreate(maxMsgs,MaxMsgLength,options);
if(ScheduleTaskMSGQId == NULL)
{
 printf("ERROR occurs in task1 msgQCreate! \n");
 return;
}

for (loop_num = 0;;loop_num++)
{
 if(loop_num == 0x0f34e078)
 {
 randnum = rand();
 sendMessage.messageNo = (char)DataModel(randnum);

 memcpy(MSGbuffer,&(sendMessage),sizeof(sendMessage));
 msgQSend(ScheduleTaskMSGQId,MSGbuffer,sizeof(MSGbuffer),timeout,MSG_
PRI_NORMAL);
 }
}

```

#### b. CreateTask 主函数

此函数创建本应用程序的所有任务。

```

void CreateTask(void)
{
 timeout = 100;
 maxMsgs = MAXMSG_NUM;
 MaxMsgLength = MAXMSG_LENGTH;

 /* 为各进程初始化调度计数器 */
 Process1Id = 0;
 Process2Id = 0;

```

```

Process3Id = 0;
Process4Id = 0;

for (loop_i1 = 0; loop_i1 < PROCESSNUM; loop_i1++)
{
 ProcessPCDArray[loop_i1].idleSum = MAXMSG_NUM;
 ProcessPCDArray[loop_i1].firstIdle = 0;
 ProcessPCDArray[loop_i1].lastIdle = MAXMSG_NUM - 1;

 for (loop_i2 = 1; loop_i2 < MAXMSG_NUM - 1; loop_i2++)
 {
 ProcessPCDArray[loop_i1].messageArray[loop_i2].prevIdle = loop_i2 - 1;
 ProcessPCDArray[loop_i1].messageArray[loop_i2].nextIdle = loop_i2 + 1;
 }
 ProcessPCDArray[loop_i1].messageArray[0].prevIdle = 0;
 ProcessPCDArray[loop_i1].messageArray[0].nextIdle = 1;

 ProcessPCDArray[loop_i1].messageArray[MAXMSG_NUM - 1].prevIdle = MAXMSG_NUM - 2;
 ProcessPCDArray[loop_i1].messageArray[MAXMSG_NUM - 1].nextIdle = MAXMSG_NUM - 1;
}

RetVal = taskSpawn("sdMSGTask", 100, VX_PRIVATE_ENV, 512, (FUNCPTR) sendMSG_Task, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0);
if (RetVal == ERROR)
 printf("ERROR occurs in sendMSG Task Creating\n");

RetVal = taskSpawn("SCHETask1", 100, VX_PRIVATE_ENV, 512, (FUNCPTR) ScheduleTask, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0);
if (RetVal == ERROR)
 printf("ERROR occurs in SCHETask1 Creating\n");

RetVal = taskSpawn("SCHETask2", 100, VX_PRIVATE_ENV, 512, (FUNCPTR) ScheduleTask, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0);
if (RetVal == ERROR)
 printf("ERROR occurs in SCHETask2 Creating\n");

```

}

## c. ScheduleTask

此函数实现的调度任务直接调度挂接在它下面的进程执行。调度任务的原则是实现进程结案的公平调度。

```
void ScheduleTask(int taskId)
{
 char MSGbufferA[4];
 char MSGbufferB[4];
 int timeout;
 int tId;
 int revBytes;
 Sint imessageNo;

 timeout = 100;
 tId = taskId;

 switch(tId)
 {
 case FIRST_TASK;
 for(;;)
 {
 /* 如果 Process1 有消息要处理,调用 processEntry1 */
 if(ProcessPCDArray[1].messageSum! = 0)
 Process1Entry();

 /* 如果 Process2 有消息要处理,调用 process2Entry */
 if(ProcessPCDArray[2].messageSum! = 0)
 Process2Entry();

 /* 如果无消息处理,则检查任务的消息队列 */
 revBytes = msgQReceive(ScheduleTask1MSGQId, MSGbufferA, MaxMsgLength,
 timeout);

 imessageNo = (int)MSGbufferA[0];
 if((imessageNo>0)&&(imessageNo<= MOD_SEED))
```



```

 {
 if((imessageNo>0)&&(imessageNo<=MOD_SEED/2))
/* 为 process1 张贴消息 */
 PostMessageOnProcess(1, imessageNo);
 else
 PostMessageOnProcess(2, imessageNo);
 }
 printf("An ERROR occurs in task scheduling\n");
 break;
 }

case SECOND_TASK:
 for(;;)
 {
 /* 如果 Process3 有消息要处理,调用 process3Entry */
 if(ProcessPCDArray[3].messageSum!= 0)
 Process3Entry();
 /* 如果 Process4 有消息要处理,调用 process4Entry */
 if(ProcessPCDArray[4].messageSum!= 0)
 Process4Entry();

 /* 如果无消息处理,则检查任务的消息队列 */
 revBytes = msgQReceive(ScheduleTask2MSGQId,MSGbufferB,MaxMsgLength,timeout);
 imessageNo = (int)MSGbufferB[0];
 if((imessageNo>MOD_SEED)&&(imessageNo<=RAND_MAX))
 {
 if((imessageNo>MOD_SEED)&&(imessageNo<=(MOD_SEED+RAND_MAX)/2))
 PostMessageOnProcess(3, imessageNo);
 else
 PostMessageOnProcess(4, imessageNo);
 }
 printf("An ERROR occurs in task scheduling\n");
 break;
 }

```

```

 }

 default:
 {
 printf("An ERROR occurs in task scheduling\n");
 break;
 }
 }
 return;
}

```

#### d. ProcessEntry 进程入口函数

用于消耗有效消息队列中的消息和记录消耗的的次数消耗次数表示了进程的调度信息。此函数以 10 为频率将调度信息打印到终端。

```

void ProcessEntry(void)
{
 int revMsg;
 revMsg = GetMessage(1);

 if(MessageValid(revMsg, 1))
 Process1Id++;
 PrintfStatistic(1, Process1Id);

 return;
}

```

#### e. DataModel 用于计算数据的模

```

int DataModel(int data1)
{
 int randnum;

 randnum = data1;
 randnum = fmod(randnum, MOD_SEED);
 if(randnum == 0)

```

```

 {
 randnum = MOD_SEED;
 }

 return randnum;
}

```

#### f. PostMessageOnProcess

此函数为给定的进程张贴一条消息,从闲态队列的头部获取一个消息块,并且将它放置在有效的消息队列的尾部。

```

void PostMessageOnProcess(int processNo, int messageNo)
{
 int process;
 int message;
 int messageBlock;
 int last;
 int first;

 process = processNo;
 message = messageNo;

 if
 ((ProcessPCDArray[process].messageSum >= MAXMSG_NUM) || (ProcessPCDArray
[process].idleSum <= 0))
 return;

 messageBlock = ProcessPCDArray[process].firstIdle;

 ProcessPCDArray[process].messageSum++;
 last = ProcessPCDArray[process].lastMessage;
 ProcessPCDArray[process].lastMessage = messageBlock;

 ProcessPCDArray[process].firstIdle = ProcessPCDArray[process].mes-
sageArray[messageBlock].nextIdle;
 ProcessPCDArray[process].idleSum--;
}

```

```

/* 修改空闲队列,摘取它的第一个元素 */
first = ProcessPCDArray[process].firstIdle;
ProcessPCDArray[process].messageArray[first].prevIdle = first;

if(ProcessPCDArray[process].idleSum == 1)
 ProcessPCDArray[process].messageArray[first].nextIdle = first;

/* 修改有效消息队列,在消息队列的尾部添加一元素 */
ProcessPCDArray[process].messageArray[messageBlock].prevMessage = last;
ProcessPCDArray[process].messageArray[messageBlock].nextMessage =
messageBlock;
ProcessPCDArray[process].messageArray[last].nextMessage = messageBlock;

/* 设置消息元素的属性 */
ProcessPCDArray[process].messageArray[messageBlock].IsUsed = 1;
ProcessPCDArray[process].messageArray[messageBlock].messageNo = message;

return;

}

```

#### g. GetMessage

用于从给定的进程获取消息并将其消耗,从有效消息队列的头部获取一个消息块,并且将它放置在闲态队列的尾部。

```

int GetMessage(int processNo)
{
 int process;
 int message; /* 返回消息 */
 int messageBlock;
 int last;
 int first;

 process = processNo;

 if ((ProcessPCDArray[process].messageSum < 0) || (ProcessPCDArray

```

```

[process].idleSum<0))
 return FALSE;

messageBlock = ProcessPCDArray[process].firstMessage;

if(ProcessPCDArray[process].messageSum == 0)
 ProcessPCDArray[process].firstMessage = 0;
last = ProcessPCDArray[process].lastIdle;
ProcessPCDArray[process].lastIdle = messageBlock;
ProcessPCDArray[process].idleSum++;

/* 修改有效消息队列,在消息队列的尾部添加一元素 */
ProcessPCDArray[process].firstMessage = ProcessPCDArray[process].messageArray[messageBlock].nextMessage;
ProcessPCDArray[process].messageSum--;

ProcessPCDArray[process].messageArray[last].nextIdle = messageBlock;
ProcessPCDArray[process].messageArray[messageBlock].prevIdle = last;
ProcessPCDArray[process].messageArray[messageBlock].nextIdle = messageBlock;

/* 修改空闲队列,摘取它的第一个元素 */
first = ProcessPCDArray[process].firstMessage;
ProcessPCDArray[process].messageArray[first].prevIdle = first;
if(ProcessPCDArray[process].messageSum == 1)
 ProcessPCDArray[process].messageArray[first].nextMessage = first;

/* 设置消息单元的属性 */
ProcessPCDArray[process].messageArray[messageBlock].IsUsed = 0;
message = ProcessPCDArray[process].messageArray[messageBlock].messageNo;

return message;
}

```

## h. MessageValid

```

/* MessageValid --- 为一个给定的进程进行消息的有效性验证 */
int MessageValid(int revMsg, int processNo)
{
 int imessage;
 int iprocess;

 imessage = revMsg;
 iprocess = processNo;

 switch(iprocess)
 {
 /* 发送到 process4 的消息 */
 case 4:
 {
 if
 (imessage > (MOD_SEED + RAND_MAX)/2 && (imessage <= RAND_MAX))
 return SUCCESS;
 else
 return FALSE;
 break;
 }
 /* 发送到 process3 的消息 */
 case 3:
 {
 if
 ((imessage > MOD_SEED) && (imessage <= (RAND_MAX + MOD_SEED)/2))
 return SUCCESS;
 else
 return FALSE;
 break;
 }

 /* 发送到 process2 的消息 */
 case 2:
 {

```

```

 if ((imessage > MOD_SEED/2) && (imessage <= MOD_SEED))
 return SUCCESS;
 else
 return FALSE;
 break;
 }

 /* 发送到 process1 的消息 */
 case 1:
 {
 if ((imessage > 0) && (imessage <= MOD_SEED/2))
 return SUCCESS;
 else
 return FALSE;
 break;
 }
 default:
 {
 return FALSE;
 break;
 }
}
}

```

#### i. PrintfStatistic

/\* PrintfStatistic —— 用于输出调试信息 \*/

void PrintfStatistic(int Process, int ProcessId)

```

{
 int count;
 int pro;

 pro = Process;
 count = ProcessId;

```

/\* 判断是否输出, 防止打印过于频繁 \*/

```
if((fmod(count,PRINT_FREQ) == 0)&&(count! = 0))
 printf("Process %d is scheduled %d times.\n",pro,count);
return;
}
```

### (3) 编译、下载、运行程序

#### ① 编译程序

在所要编译的文件上单击鼠标右键编译文件。

- 如果程序通过编译,会弹出完成的对话框,否则将报错;
- 在弹出的菜单中选择编译生成.out 文件。

#### ② 下载工程文件

下载.out 文件到 VxSim 中。如果有符号表不统一的情况发生,下载的时候会有相应的提示信息,如果正常则没有任何额外信息。

#### ③ 运行结果

在 Shell 中输入启动函数名:CreateTask。不正确的执行会打印出相应的错误信息和错误号。

#### ④ 观察、分析运行结果

注意观察结果,看看是否有消息打印到 Shell 中或者控制台中(具体在哪显示取决于打印所定向的端口)。

另外,可以在 Shell 中通过输入一些 shell 命令或者启动 WindView 等调试工具观察任务执行的情况,比如任务的优先级、堆栈、ID 号是否正常,任务之间的交互是否正常,有没有错误信息打印出来。

在 VxWorks Simulator 上观察运行结果并通过运行结果对进程调度的公平性等方面做出评价。

### 7. 思考题

想一些其他可行的方法用于提高任务调度的效率。



## 第7章 基础应用性实验

本章的实验属于基本的应用实验。需要使用实验板上的一些接口和简单的外设。通过这些实验,可以学习在 VxWorks 环境下的接口编程。

### 7.1 实验1 LED 灯实验

#### 1. 实验目的

(1) 掌握 ARM 嵌入式微处理器的通用 I/O 口工作原理。利用北京邮电大学电信工程学院 IRDC 实验室开发的 IRDC05v3 实验板上的通用 I/O 口输入输出控制。

(2) 学习并掌握基于 ARM 开发板对 I/O 编程的方法。并学习通过通用 I/O 口实现 LED 显示原理。

(3) ARM 的通用 I/O 口相关寄存器的功能,熟悉 ARM 系统硬件的通用 I/O 口工作原理相关接口的复用方法。

(4) 学习使用 WindRiver Tornado 2.0 above Integrated Development Environment for ARM 开发工具在 VxWorks 操作系平台上 I/O 接口的输入输出功能程序的编写和发起一个应用程序任务。

#### 2. 实验内容

(1) 使用 VxWorks 操作系统的 WindRiver Tornado 集成开发环境,编写和下载调试程序。熟悉 VxWorks 应用程序的框架结构及其开发流程,VxWorks 任务的发起。学习 WindRiver Tornado 集成开发环境下硬件的编程流程、方法。

(2) 阅读 ARM 芯片文档,掌握 ARM 的通用 I/O 口相关寄存器的功能,熟悉 ARM 系统硬件的通用 I/O 口工作原理相关接口的复用方法。运行一个“Hello world”闪灯程序,交替点亮接在 S3C4510B 的通用 I/O 口上的指示灯。利用实验板上 S3C4510B 提供

的 18 个可编程的通用 I/O 端口(用户可将每个端口配置为输入模式、输出模式)学习通用 I/O 口工作原理及其寄存器状态字的定义内容。了解可编程的通用 I/O 端口控制器的工作过程。

### 3. 预备知识

(1) 熟练使用 VxWorks 操作系统的 WindRiver Tornado 集成开发环境,编写和下载调试程序的基本过程。

(2) 熟悉 VxWorks 应用程序的框架结构及其开发流程,VxWorks 任务的发起。

(3) 了解通用 I/O 口工作原理及其使用方法。

(4) 熟悉 ARM 的通用 I/O 口相关寄存器的功能,熟悉 ARM 系统硬件的通用 I/O 口工作原理及相关接口的使用方法。

### 4. 实验设备和工具

#### (1) 硬件环境

- ① 北京邮电大学电信工程学院 IRDC 实验室开发的 CS4510 教学实验/评估板;
- ② PC;
- ③ 10/100 M 以太网连接(可选双网口) + 直连串口连接;
- ④ 用于 ARM7TDMI 的 JTAG 仿真器(可选)。

#### (2) 软件开发环境

- ① WindRiver Tornado 2.0 above Integrated Development Environment for ARM;
- ② Win98、Win2000 或 WinXP 操作系统;
- ③ ARM SDT 2.51 集成开发环境、仿真器驱动程序(汇编方式调试可选);
- ④ 超级终端通信程序(C 语言/ARM 汇编语言)。

### 5. 实验原理

(1) I/O 接口:指示灯(发光二极管)接在通用 I/O 的 P17、P16 引脚,如图 7-1 所示。

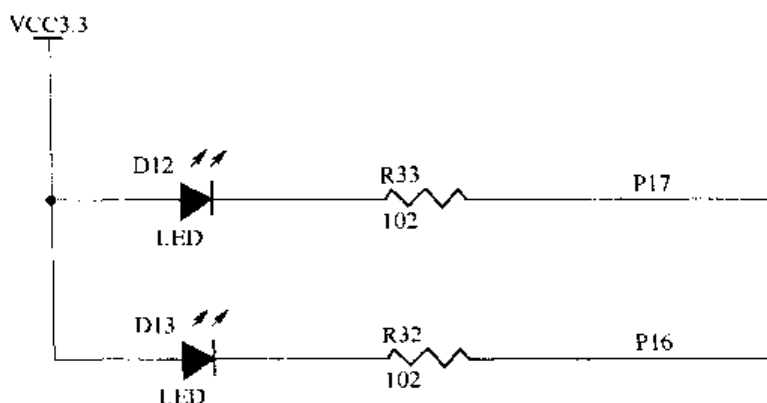


图 7-1 发光二极管电路图

S3C4510B 芯片有 18 个通用可编程 I/O 口,根据功能的不同可以配置成输入模式、

输出模式和特殊功能模式。本实验中只需把 P16 口和 P17 口设置为输出并输出低电平即可点亮 LED 发光二极管。

(2) S3C4510B 的 I/O 寄存器: S3C4510B 有 3 个 I/O 寄存器 IOPMOD、IOPCON 和 IOPDATA, 分别设定它们, 可以定义 18 个通用可编程 I/O 口的功能。其中: IOPMOD 定义通用 I/O 口作为输入或是输出; IOPCON 定义 PORT8~PORT17 作为特殊功能; IOPDATA 反映 18 个相应 I/O 口的信号电平, 如果是输出则设定相应位, 如果是输入则读入相应位。

## 6. 实验步骤

### (1) 创建 bootable 的 hello 工程

启动 Tornado, 创建新工程, 如图 7-2 所示。选择创建 bootable VxWorks image, 这样应用程序包含在 VxWorks image 中, 随 image 一同下载, 也可以选择应用程序是否自动启动。

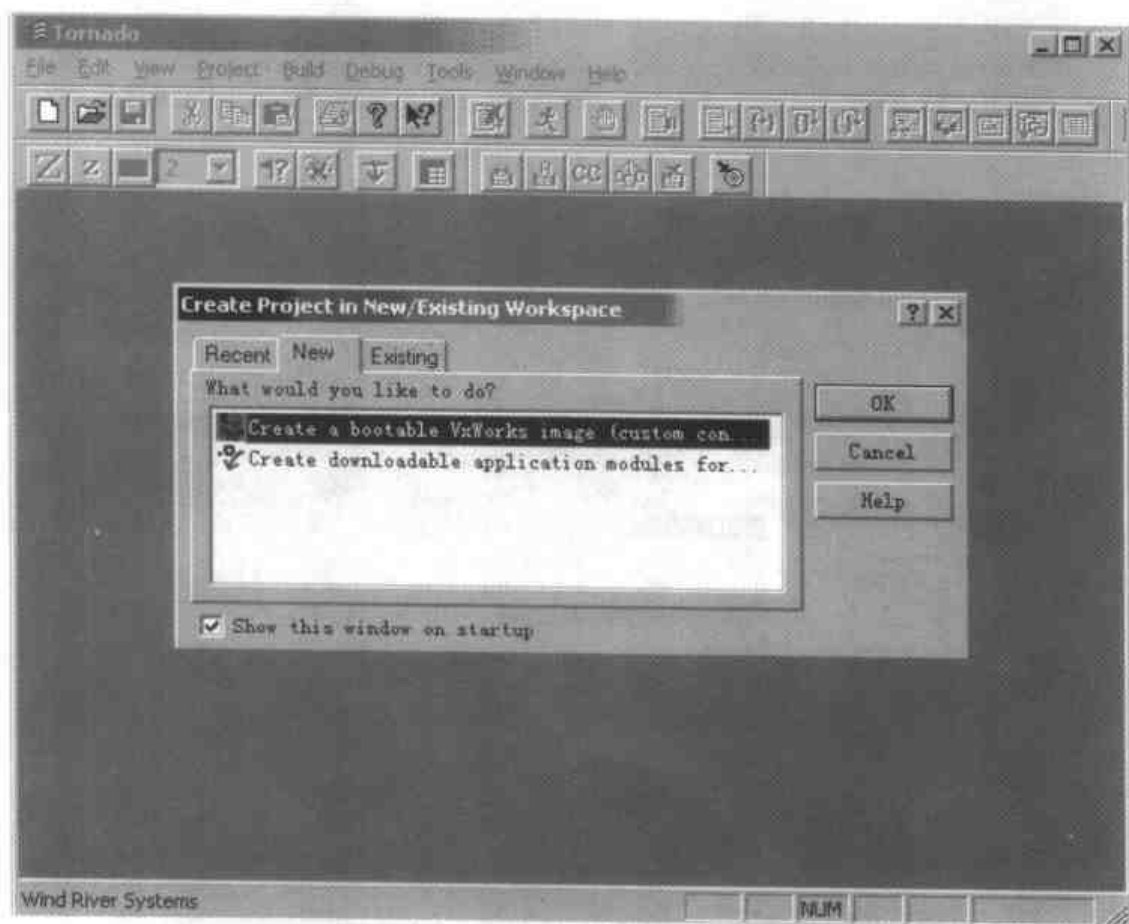


图 7-2 创建工程

建立名为 hello 的工程, 选择相应 BSP 和 GNU 工具链如图 7-3 所示。

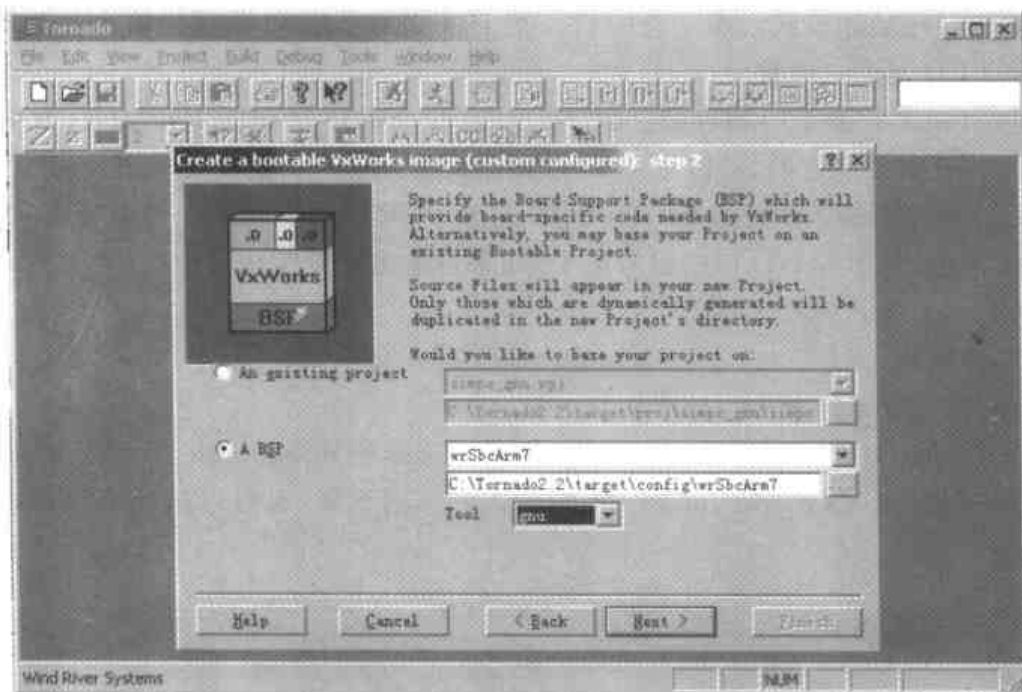


图 7-3 选择 BSP

在工程中,新建文件 hello.c,如图 7-4 所示。



图 7-4 增加文件

编写 hello.c, 然后编译工程, 如图 7-5 所示。没有错误即可启动开发板下载。

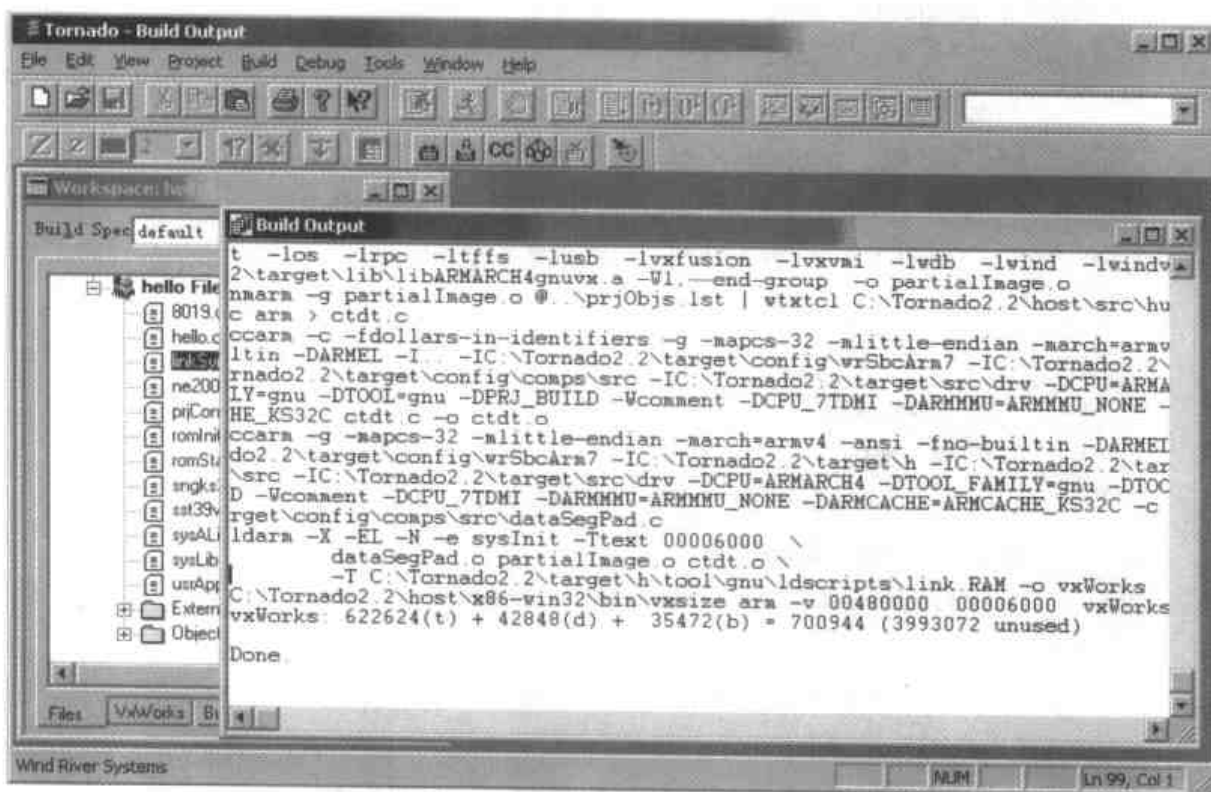


图 7-5 编译文件

hello.c

```

/***** hello.c start *****/
#include "VxWorks.h"
#include "config.h"
#include "intLib.h"
#include "logLib.h"
#include "stdio.h"
#include "semLib.h"

/* * 定义 I/O 寄存器 * */
#define IOPMOD 0x3ff5000
#define IOPCON 0x3ff5004
#define IOPDATA 0x3ff5008

#define STACK_SIZE 20000
int tidled, flag;

```

```

void led_light(void);
STATUS progStart (void)
{
 flag = 0;
 tidled = taskSpawn ("tLed", 200, 0, STACK_SIZE,
 (FUNCPTR) led_light, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0);

 return (OK);
}

void led_init(void)
{
 UINT32 temp;
 temp = * (volatile UINT32 *) IOPMOD;
 temp = temp | 0x30000;
 * (volatile UINT32 *) IOPMOD = temp;
}

/ ***** /
/* * hello world * */
/ ***** /

void led_light(void)
{
 UINT32 temp;
 int k = 0x3;
 led_init();

 for(;;)
 {
 if(flag == 0)
 {
 taskDelay(sysClkRateGet());

 / *****
 ***** 需要读者添加 *****
 * 设置 IOPDATA 端口, 并且进行移位操作 *
 ***** /

```

```

 if(k<=0)k=3;
 else k--;

 *(volatile UINT32 *)IOPDATA = temp;
}
else break;
}

void progStop(void)
{ flag = 1;
}

/ ***** hello.c end ***** /

```

## (2) 工程下载并执行

- ① 连接配套的开发板电源、以太网电缆和串口线。
- ② 启动主机上的 ftp server(在 tornado 的安装目录下)。选中菜单中“security”下的“users/rights...”,新建用户名 target,同时设定用户名和密码都为 target,最后设定主机目录,就是刚才编译生成的 VxWorks image 的目录“C:\Tornado2.2\target\proj\hello\default”如图 7-6 所示。

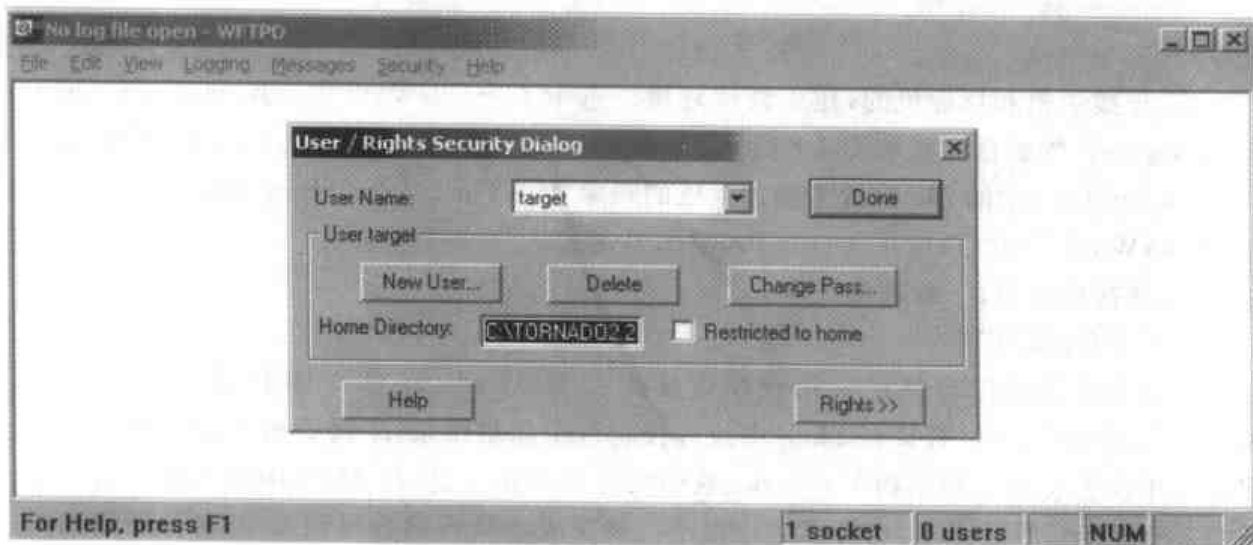


图 7-6 设置 ftp server

③ 启动超级终端或串口调试助手。

④ 开发板上电启动,下载 VxWorks image,如果下载成功,则如图 7-7 所示。

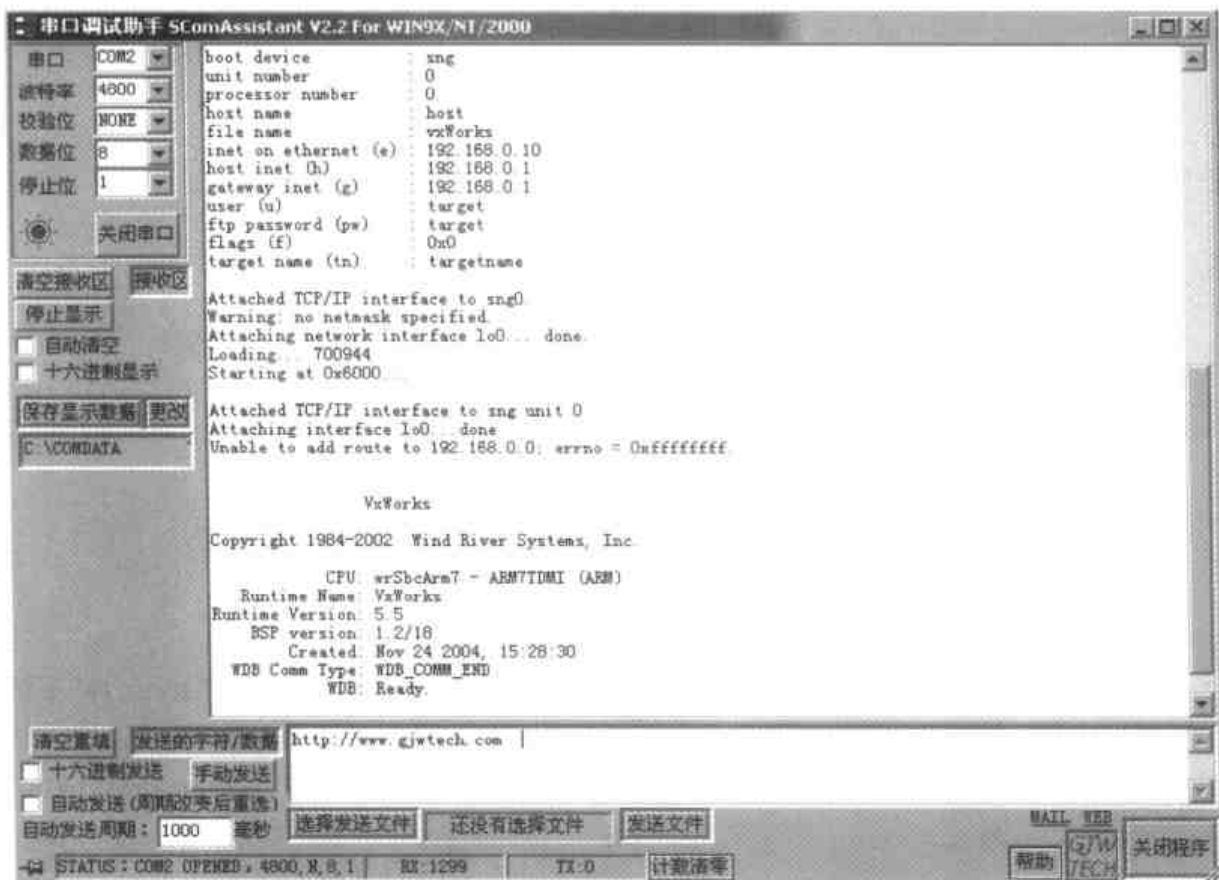


图 7-7 启动串口调试工具

⑤ 连接主机和目标机的,建立调试环境。选中 tornado 菜单中 tools→target server→configure。配置目标机如图 7-8 所示:主要是将目标机 ip 地址“192.168.0.10”和 core file and symbols 中的 file 设置为刚才生成的映像“C:\Tornado2.2\target\proj\hello\default\VxWorks”(在 Target Server Properti 中修改)。

⑥ 连接成功后,下载映像文件。

### (3) 下载运行程序

上面所介绍的内容是把应用程序直接加在 VxWorks 映像文件中,将映像下载后 hello.c 自动运行,可以直接在 Shell 中或者控制台中观察结果,不需要再手动下载应用程序。如果没有将应用程序和 VxWorks 映像文件捆绑在一起,在 Shell 中输入任务启动函数名 progStart,然后回车,也可以运行程序。如果成功执行则会返回 0,否则会看到错误号或者其他错误信息,如图 7-9 所示。



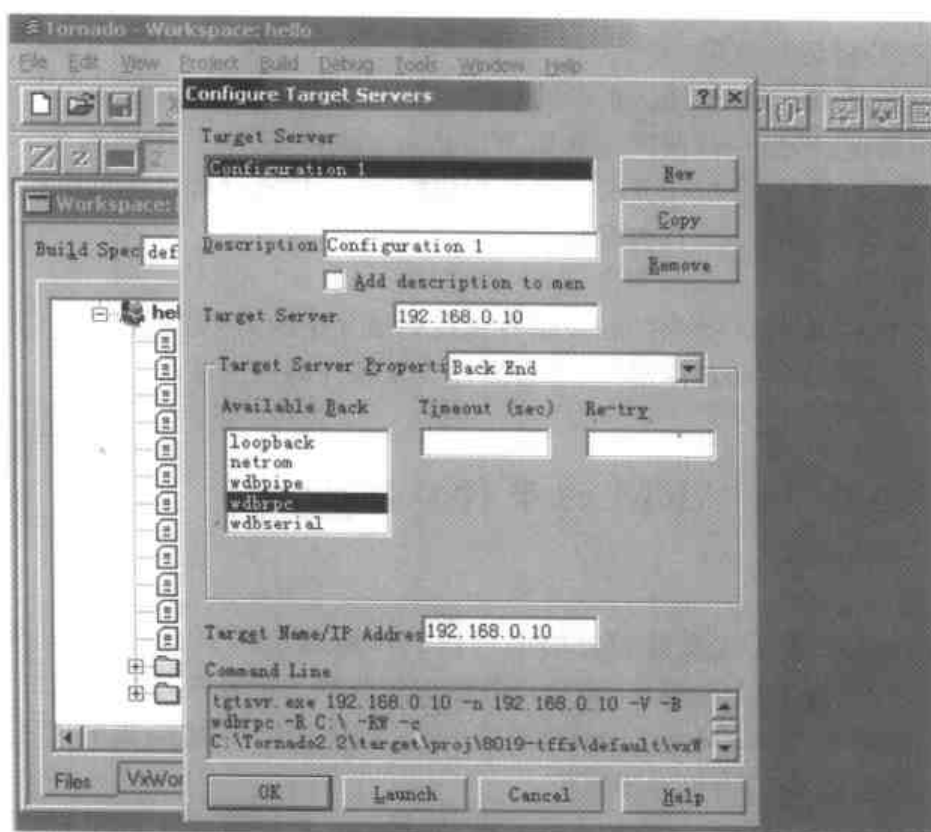


图 7-8 配置 Target Server

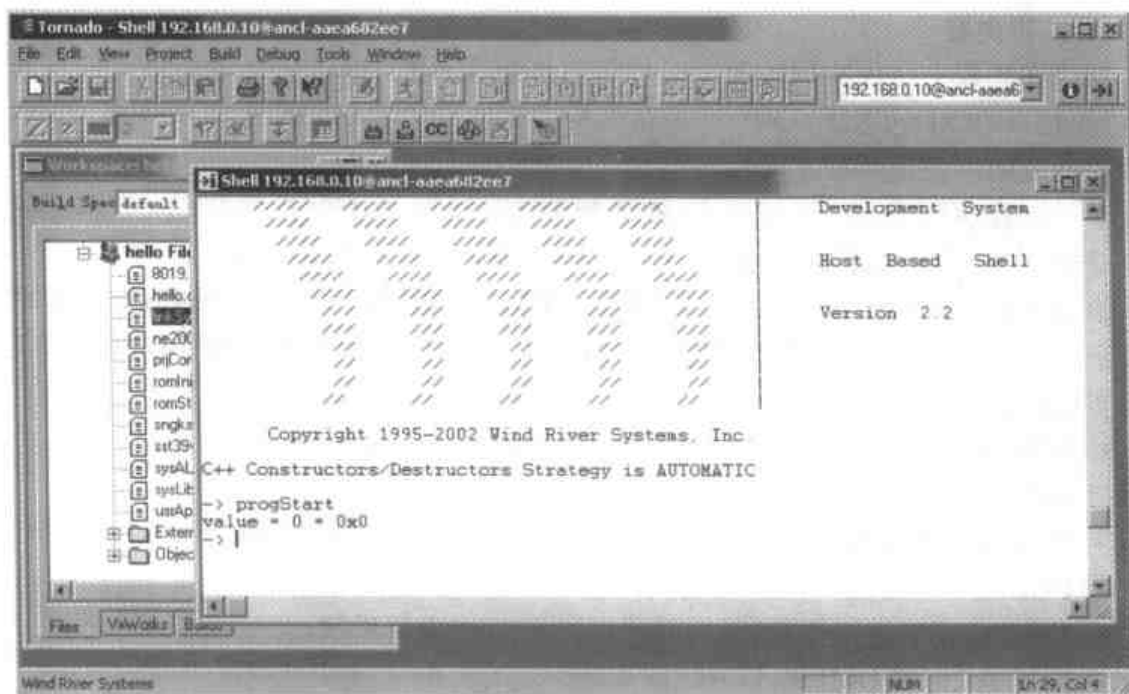


图 7-9 启动 Shell

#### (4) 观察、分析运行结果

观察开发板的 led 发光情况,正确的结果应该是轮流点亮;观察 Shell 中或者控制台上的消息显示情况,借助一些调试工具如 WindView 或者在 Shell 中输入 Shell 命令等方法监视任务执行的情况,分析任务的一些参数和特性。另外,分析目标机中任务占用资源的情况(可以启动 Browser 工具查看)。

#### 7. 思考题

- (1) 试利用 VxWorks 提供的相应函数实现更加复杂的 I/O 控制。
- (2) 复习 VxWorks 的程序编写、下载流程以及硬件调试方法。

## 7.2 实验 2 ARM 的串行口实验

### 1. 实验目的

(1) 掌握 ARM 嵌入式微处理器的串行口工作原理。利用 IRDC05v3 实验板上的 UART 驱动软件实现 ECHO 操作,将从接收端口接收到的字符通过发送端口送出。

(2) 学习编程实现 ARM 的 UART 通信。通过 IRDC05v3 实验板的 UART 端口与 PC 的串行口实现 UART 协议异步通信。

(3) 掌握 CPU 利用串口通信的方法。PC 通过仿真终端程序实现字符输入和输出,并在进一步的基础上可实现文件传送。

### 2. 实验内容

利用实验板上已经实现的两个串行端口进行 UART 协议全双工通信。硬件接口方式:三线接口(RXD、TXD、GND)。通信波特率为 4 800 bit/s。发送和接收以 DMA 方式下的查询或中断任选一种实现均可。

学习串行通信原理,了解串行通信控制器,阅读 ARM 芯片文档,掌握 ARM 的 UART 相关寄存器的功能,熟悉 ARM 系统硬件的 UART 相关接口。

编程实现 ARM 和计算机的串行通信:ARM 监视串行口,将接收到的字符在液晶屏上显示出来(可选)。(计算机向串口发送数据是通过计算机上附带的“超级终端”结合键盘来实现的。)

### 3. 实验设备

#### (1) 硬件环境

- 北京邮电大学电信工程学院 IRDC 实验室开发的 CS4510 教学实验/评估板;
- PC;
- 10/100 M 以太网连接(可选双网口) + 直连串口连接;
- 用于 ARM7TDMI 的 JTAG 仿真器。

#### (2) 软件开发环境

- WindRiver Tornado 2.0 above Integrated Development Environment for ARM;

- Win98、Win2000 或 WinXP 操作系统；
- ARM SDT 2.51 集成开发环境、仿真器驱动程序(汇编方式调试可选)；
- 超级终端通信程序(C语言/ARM 汇编语言)。

#### 4. 预备知识(见本实验附录)

(1) 熟练使用 VxWorks 操作系统的 WindRiver Tornado 集成开发环境,熟悉编写和下载调试程序的基本过程。

(2) 熟悉 VxWorks 应用程序的框架结构及其开发流程以及 VxWorks 任务的发起。

(3) 了解串行总线结构及其通信方法。

(4) 熟悉 ARM 嵌入式微处理器的串行口工作原理及其寄存器的定义内容。

#### 5. ARMS3C4510B 的串行口寄存器状态字介绍

S3C4510B 的 UART 单元提供两个独立的异步串行 I/O 口(Asynchronous Serial I/O, SIO),各带有 16 字节的 FIFO(先入先出寄存器),最大波特率 115.2 K。每个 UART 有 7 种状态:溢出错误、校验错误、帧错误、暂停态、接收缓冲区准备好、发送缓冲区空、发送移位缓冲器空。这些状态可以由相应的 UTRSTAT<sub>n</sub>/UERSTAT<sub>n</sub> 表示。与发送接收缓冲区相对应的有错误缓冲区。每个通信口均可工作在中断模式或 DMA 模式,也即 UART 能产生内部中断请求或 DMA 请求在 CPU 和串行 I/O 口之间传送数据。具体模块可以参考图 7-10。

S3C4510B 的 UART 单元特性包括:波特率可编程;支持红外发送与接收;1~2 个停止位;5、6、7 或 8 个数据位;奇偶校验。

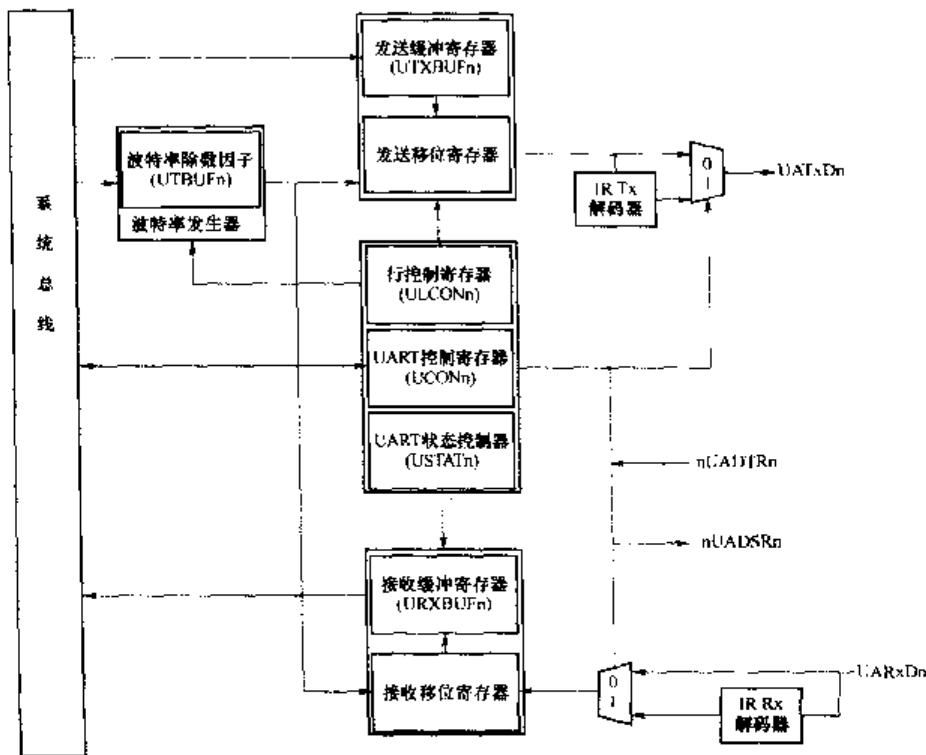


图 7-10 ARM 串行口功能模块

与 UART 有关的寄存器功能定义如表 7-1~表 7-12 以及图 7-11~图 7-16 所示。

表 7-1 UART 特殊功能寄存器的地址、功能、缺省复位值表

| 寄存器     | 偏移地址   | 操作  | 功能描述             | 复位值  |
|---------|--------|-----|------------------|------|
| ULCON0  | 0xD000 | 读/写 | UART0 行控制寄存器     | 0x00 |
| ULCON1  | 0xE000 | 读/写 | UART1 行控制寄存器     | 0x00 |
| UCON0   | 0xD004 | 读/写 | UART0 控制寄存器      | 0x00 |
| UCON1   | 0xE004 | 读/写 | UART1 控制寄存器      | 0x00 |
| USTAT0  | 0xD008 | 读   | UART0 状态寄存器      | 0xC0 |
| USTAT1  | 0xE008 | 读   | UART1 状态寄存器      | 0xC0 |
| UTXBUF0 | 0xD00C | 写   | UART0 发送保持寄存器    | 未定义  |
| UTXBUF1 | 0xE00C | 写   | UART1 发送保持寄存器    | 未定义  |
| URXBUF0 | 0xD010 | 读   | UART0 接收缓冲寄存器    | 未定义  |
| URXBUF1 | 0xE010 | 读   | UART1 接收缓冲寄存器    | 未定义  |
| UBRDIV0 | 0xD014 | 读/写 | UART0 波特率除数因子寄存器 | 0x00 |
| UBRDIV1 | 0xE014 | 读/写 | UART1 波特率除数因子寄存器 | 0x00 |
| BRDCNT0 | 0xD018 | 写   | UART0 波特率计数寄存器   | 0x00 |
| BRDCNT1 | 0xE018 | 写   | UART1 波特率计数寄存器   | 0x00 |
| BRDCLK0 | 0xD01C | 写   | UART0 波特率时钟监视器   | 0x00 |
| BRDCLK1 | 0xE01C | 写   | UART1 波特率时钟监视器   | 0x00 |

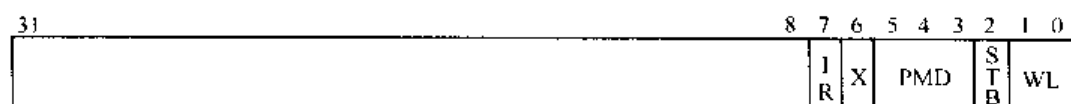


图 7-11 UART 行控制寄存器(UART Line Control Registers, ULCON0、ULCON1)

表 7-2 UART 行控制寄存器描述-1

| 寄存器    | 偏移地址   | 操作  | 功能描述         | 复位值         |
|--------|--------|-----|--------------|-------------|
| ULCON0 | 0xD000 | 读/写 | UART0 行控制寄存器 | 0x0000,0000 |
| ULCON1 | 0xE000 | 读/写 | UART1 行控制寄存器 | 0x0000,0000 |

表 7-3 UART 行控制寄存器描述-2

| 位     | 位名     | 功能描述                                                                                          |
|-------|--------|-----------------------------------------------------------------------------------------------|
| [1:0] | 每帧字长   | 该两位指示发送或接收的每帧的数据位;00=5位,01=6位,10=7位,11=8位。                                                    |
| [2]   | 停止位数   | 该位指示每帧数据的停止位数;0=每帧一个停止位,1=每帧两个停止位。                                                            |
| [5:3] | 校验模式   | 该三位指示在数据的发送与接收过程中如何生成校验并进行检测:<br>0XX=无校验,100=奇校验,101=偶校验,100=奇校验,110=校验强制/检测为1,111=校验强制/检测为0。 |
| [6]   | 串行时钟选择 | 该位用于选择时钟源:<br>0=内部时钟(MCLK),<br>1=外部时钟(UCLK)。                                                  |
| [7]   | 红外模式选择 | S3C4510B的UART模块支持红外(IR)发送与接收。要使能红外模式,需设置ULCON[7]为1,否则选择正常UART模式。                              |

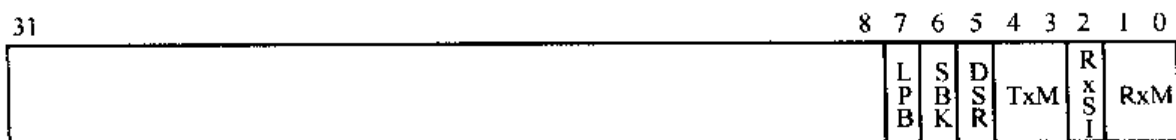


图 7-12 UART 控制寄存器(UART Control Registers,UCON0,UCON1)

表 7-4 UART 控制寄存器描述-1

| 寄存器   | 偏移地址   | 操作  | 功能描述        | 复位值         |
|-------|--------|-----|-------------|-------------|
| UCON0 | 0xD004 | 读/写 | UART0 控制寄存器 | 0x0000,0000 |
| UCON1 | 0xE004 | 读/写 | UART1 控制寄存器 | 0x0000,0000 |

表 7-5 UART 控制寄存器描述-2

| 位     | 位名       | 功能描述                                                                                                           |
|-------|----------|----------------------------------------------------------------------------------------------------------------|
| [1:0] | 接收模式选择   | 该两位的值决定当从 UART 接收缓冲寄存器中读取数据时的当前功能：<br>00=禁止 Rx 模式，<br>01=产生中断请求，<br>10=产生 GDMA 通道 0 请求，<br>11=产生 GDMA 通道 1 请求。 |
| [2]   | 接收状态中断使能 | 该位决定当在数据的接收过程中发生异常(如发生中断、帧错误、校验错误或 Overrun 错误)时是否让 UART 产生中断请求：<br>0= 不产生接收状态中断请求，<br>1= 产生接收状态中断请求。           |
| [4:3] | 发送模式选择   | 该两位的值决定当写数据到 UART 发送缓冲寄存器中时的当前功能：<br>00=禁止 TX 模式，<br>01=产生中断请求，<br>10=产生 GDMA 通道 0 请求，<br>11=产生 GDMA 通道 1 请求。  |
| [5]   | 数据设备准备好  | 该位选择是否产生数据设备准备好(DSR)信号输出：<br>0 不产生 DSR 输出，<br>1= 产生 DSR 输出(nUADSR 引脚)。                                         |
| [6]   | 发送间隔     | 该位选择是否发送间隔信号：<br>0= 不发送间隔信号，<br>1 = 发送间隔信号。                                                                    |
| [7]   | 回环模式选择   | 该位选择 UART 是否进入回环模式。在回环模式下，发送的数据的输出置为高电平，发送缓冲寄存器 UTXBUF 在内部直接连接到接收缓冲寄存器 URXBUF。<br>回环模式仅用于测试目的，在正常操作模式下，该位应为 0。 |

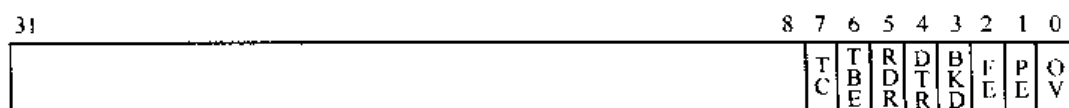


图 7-13 UART 状态寄存器(UART Status Registers, USTAT0、USTAT1)

表 7-6 UART 状态寄存器描述-1

| 寄存器    | 偏移地址   | 操作 | 功能描述        | 复位值         |
|--------|--------|----|-------------|-------------|
| USTAT0 | 0xD008 | 读  | UART0 状态寄存器 | 0x0000,00C0 |
| USTAT1 | 0xE008 | 读  | UART1 状态寄存器 | 0x0000,00C0 |

表 7-7 UART 状态寄存器描述-2

| 位   | 位名            | 功能描述                                                                                                                                                                                            |
|-----|---------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| [0] | Overrun 错误    | <p>在接收串行数据的操作中,当发生 Overrun 错误时,该位自动置为 1。在先前接收到的数据还未读出,而被新收到的数据所覆盖时发生 Overrun 错误。</p> <p>如果接收状态中断使能位 UCON[2]为 1,则当 Overrun 错误发生时产生接收状态中断。</p> <p>任何时候读取 UART 状态寄存器 USTAT,该位都会自动清零。</p>           |
| [1] | 校验错误          | <p>在接收串行数据的操作中,当发生校验错误时,该位自动置为 1。</p> <p>如果接收状态中断使能位 UCON[2]为 1,则当校验错误发生时产生接收状态中断。</p> <p>任何时候读取 UART 状态寄存器 USTAT,该位都会自动清零。</p>                                                                 |
| [2] | 帧错误           | <p>在接收串行数据的操作中,当发生数据帧错误时,该位自动置为 1。当检测到停止位为 0 时发生帧错误。</p> <p>如果接收状态中断使能位 UCON[2]为 1,则当帧错误发生时产生接收状态中断。</p> <p>任何时候读取 UART 状态寄存器 USTAT,该位都会自动清零。</p>                                               |
| [3] | 间隔中断          | <p>当接收到间隔信号时,该位自动置为 1。</p> <p>如果接收状态中断使能位 UCON[2]为 1,则当接收到间隔信号时产生接收状态中断。</p> <p>任何时候读取 UART 状态寄存器 USTAT,该位都会自动清零。</p>                                                                           |
| [4] | 数据终端准备好 (DTR) | <p>该位指示数据终端准备好引脚(nUADTR)的当前信号电平。</p> <p>当该位为 1 时,数据终端准备好引脚(nUADTR)为低电平,当该位为 0 时,数据终端准备好引脚(nUADTR)为高电平。</p>                                                                                      |
| [5] | 接收数据准备好       | <p>当接收缓冲寄存器 URXBUF 从串行口接收到一个有效数据时,该位自动置为 1,然后就可以从 URXBUF 中读取接收到的有效数据。</p> <p>当该位为 0 时表示还未接收到一个有效的数据。</p> <p>当该位为 1 时,根据当前 UART 接收模式位 UCON[1:0]的设置,可以产生中断或 DMA 请求。</p>                           |
| [6] | 发送缓冲寄存器空      | <p>当发送缓冲寄存器 UTXBUF 为空时,该位自动置为 1,此时可以向 UTXBUF 写入下一个要发送的数据。</p> <p>当该位为 0 时,表示 UTXBUF 中的数据还未全部拷贝到发送移位寄存器中,此时不能向 UTXBUF 中写入新的发送数据。</p> <p>当该位为 1 时,根据当前 SIO 发送模式位 UCON[4:3]的设置,可以产生中断或 DMA 请求。</p> |
| [7] | 发送完成(TC)      | <p>当发送缓冲寄存器 UTXBUF 为空且发送移位寄存器也为空时,该位自动置为 1。该位用于指示软件是否可以在此时禁用发送功能模块。</p>                                                                                                                         |

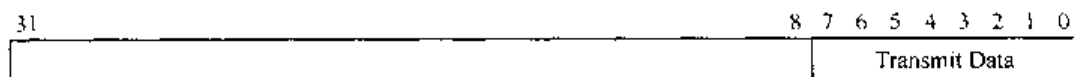


图 7-14 UART 发送缓冲寄存器

表 7-8 UART 发送缓冲寄存器描述

| 寄存器     | 偏移地址   | 操作 | 功能描述          | 复位值 |
|---------|--------|----|---------------|-----|
| UTXBUF0 | 0xD00C | 写  | UART0 发送缓冲寄存器 | 未定义 |
| UTXBUF1 | 0xE00C | 写  | UART1 发送缓冲寄存器 | 未定义 |

表 7-9 UART 发送缓冲寄存器描述

| 位     | 位名   | 功能描述                                                                                                                         |
|-------|------|------------------------------------------------------------------------------------------------------------------------------|
| [7:0] | 发送数据 | 该位存放要发送的数据。<br>当向该寄存器写入数据时,状态寄存器中的发送缓冲寄存器空位 USTAT[6]清 0,以防 UTXBUF 中的数据被覆盖。<br>任何时候向 UTXBUF 中写入新的数据,发送缓冲寄存器空位 USTAT[6]都会被清 0。 |

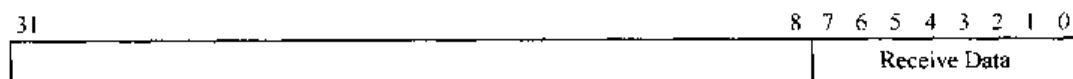


图 7-15 UART 接收缓冲寄存器

表 7-10 UART 接收缓冲寄存器描述-1

| 寄存器     | 偏移地址   | 操作 | 功能描述          | 复位值 |
|---------|--------|----|---------------|-----|
| URXBUF0 | 0xD010 | 读  | UART0 接收缓冲寄存器 | 未定义 |
| URXBUF1 | 0xE010 | 读  | UART1 接收缓冲寄存器 | 未定义 |

表 7-11 UART 接收缓冲寄存器描述-2

| 位     | 位名   | 功能描述                                                                                                                         |
|-------|------|------------------------------------------------------------------------------------------------------------------------------|
| [7:0] | 接收数据 | 该位存放接收到的数据。<br>当接收完一帧数据时,UART 状态寄存器中的接收数据准备好位 USTAT[5]被置为 1,以防从 URXBUF 中的读出的数据无效。<br>任何时候读取 URXBUF,接收数据准备好位 USTAT[5]都会自动清 0。 |

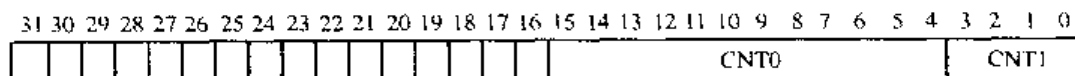


图 7-16 UART 波特率除数因子寄存器



表 7-12 UART 波特率除数因子寄存器 UBRDIV0、UBRDIV1

| 寄存器     | 偏移地址   | 操作  | 功能描述             | 复位值         |
|---------|--------|-----|------------------|-------------|
| UBRDIV0 | 0xD014 | 读/写 | UART0 波特率除数因子寄存器 | 0x0000,0000 |
| UBRDIV1 | 0xE014 | 读/写 | UART1 波特率除数因子寄存器 | 0x0000,0000 |

注:[3:0] 波特率除数因子值(CNT1)

xxx0—除 1

xxx1—除 16

[15:4]时间常数值(CNT0)

## 6. 实验步骤

(1) 学习附录中提供的串行通信原理,了解 ARM 上相应寄存器的功能和各位的意义。计算设定波特率的控制值,送入相应的控制波特率寄存器(UBRDIV<sub>n</sub>),计算公式如下所示。

CNT0 的计算公式

$$CNT0 = MCLK / (32 \times BR) - 1$$

MCLK:系统的工作频率

BR:通信的波特率

UART 波特率示例。

UART 的波特率发生器的输入时钟可以为系统时钟,也可以从外部引入时钟信号。若选用系统时钟为波特率发生器的输入时钟,当系统时钟为 50 MHz 时,则最大的波特率时钟输出为  $MCLK/16 (= 1.5625 \text{ MHz})$ ,其中 MCLK2 为系统时钟 MCLK 除以 2。

(2) 在 WindRiver Tornado 创建相应的工程,定义与 UART 有关的各个寄存器地址和一些特殊的位命令。主要有以下各寄存器。

```
/* UART 的全部功能寄存器 */
#define ULCON0 (* (volatile unsigned *)0x03FFD000) //UART channel0
line control register
#define UCON0 (* (volatile unsigned *)0x03FFD004) //UART channel0
control register
#define USTAT0 (* (volatile unsigned *)0x03FFD008) //UART channel0
status register
#define UTXBUF0 (* (volatile unsigned *)0x03FFD00c) //UART channel0
transmit holding register
#define URXBUF0 (* (volatile unsigned *)0x03FFD010) //UART channel0
receive buffer register
#define UBRDIV0 (* (volatile unsigned *)0x03FFD014) //Baud rate di-
```

```

visor register0
#define ULCON1 (* (volatile unsigned *)0x03FFE000) //UART channel1
line control register
#define UCON1 (* (volatile unsigned *)0x03FFE004) //UART channel1
control register
#define USTAT1 (* (volatile unsigned *)0x03FFE008) //UART channel1
status register
#define UTXBUF1 (* (volatile unsigned *)0x03FFE00c) //UART channel1
transmit holding register
#define URXBUF1 (* (volatile unsigned *)0x03FFE010) //UART channel1
receive buffer register
#define UBRDIV1 (* (volatile unsigned *)0x03FFE014) //Baud rate divi-
sor register1

```

将上述内容定义为一个有一定功能的头文件,并且加入到工程中。

(3) 在程序的文件中添加下面的执行函数,并且建立一个文件(\*.C)包含各函数。这些函数的功能主要为以下几个。

① 在主函数中首先初始化串行口。

```
void InitUART(int Port, int Baudrate);
```

传入参数 Port 为 RDC-CS4510 实验板上的串行端口的选择,取值为 0、1;Baudrate 为选择的串行端口波特率设置,通过附录中的公式计算。

② 发送函数。

```
void PrintUART(int Port, char * s);
```

传入参数 Port 为 RDC-CS4510 实验板上的串行端口的选择,取值为 0、1;s 为选择的串行端口发送的字符串指针。

③ 接收函数(可选)。

④ 在完成了基本函数的编写以后,完成主函数的编写,发起任务(可以是 DMA 方式下的查询或中断)。

```

int uart()
{
 InitUART(1,0xxxx); //19200bps CPU 工作频率 50MHz 0 = COM1;1 = COM2
 for(int i = 0;i<100;i++)
 {
 PrintUART(0,"Communcation Testting! \r\n");
 }
 return(0);
}

```

## (4) 编译、下载、运行程序

可以把应用程序直接加在 VxWorks 映像文件中,将映像下载后应用程序会自动运行,可以直接在 Shell 中或者控制台中观察结果,不需要再手动下载应用程序。如果没有将应用程序和 VxWorks 映像文件捆绑在一起,在 Shell 中输入任务启动函数名,然后回车,也可以运行程序。如果成功执行则会返回 0,否则会看到错误号或者其他错误信息。

## (5) 观察、分析运行结果

观察开发板的串口通信情况,看传过来的字符信息是否是自己想要的信息;试着改变串口参数,看看有什么变化。

观察 Shell 中或者控制台上的消息显示情况,借助一些调试工具如 WindView 或者在 Shell 中输入 shell 命令等方法监视任务执行的情况,分析任务的一些参数和特性。另外,分析目标机中任务占用资源的情况(可以启动 Browser 工具查看)。

## 7. 思考题

(1) ARM 的串行口串行通信可以采用几种方法? 实现以第二种方式的串行通信。

(2) 该考虑串行端口在嵌入式应用中的使用场合。

## 8. 附录内容

## (1) 异步串行 I/O

串行通信是微型计算机之间一种常见的近距离通信手段,因使用方便、编程简单而广泛使用,几乎所有的微控制器、PC 都提供串行通信接口。异步串行方式是将传输数据的每个字符一位接一位(例如先低位、后高位)地传送。数据的各不同位可以分时使用同一传输通道,因此串行 I/O 可以减少信号连线,最少用一对线即可进行。接收方对于同一根线上一连串的数字信号,首先要分割成位,再按位组成字符。为了恢复发送的信息,双方必须协调工作。在微型计算机中大量使用异步串行 I/O 方式,双方使用各自的时钟信号,而且允许时钟频率有一定误差,因此实现较容易。但是由于每个字符都要独立确定起始和结束(即每个字符都要重新同步),字符和字符间还可能有长度不定的空闲时间,因此效率较低(如图 7-17 所示)。

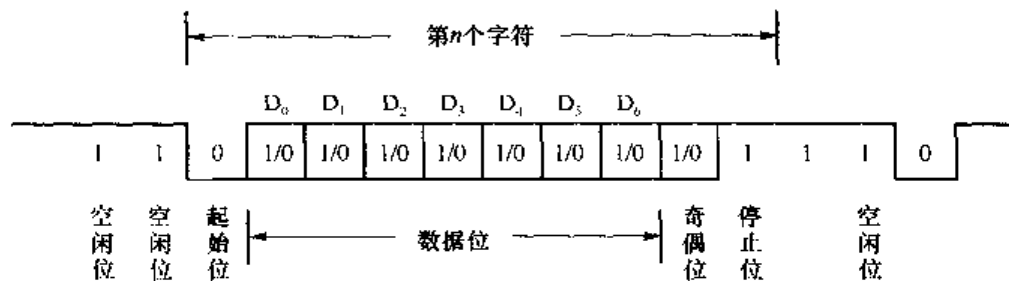


图 7-17 异步串行通信中一个字符的传送格式

开始前,线路处于空闲状态,送出连续 1。传送开始时首先发一个 0 作为起始位,然后出现在通信线上的是字符的二进制编码数据。每个字符的数据位长可以约定为 5 位、6 位、7 位或 8 位,一般采用 ASCII 编码。后面是奇偶校验位,根据约定,用奇偶校验位将所传字符中为 1 的位数凑成奇数个或偶数个。也可以约定不要奇偶校验,这样就取消奇偶

校验位。最后是表示停止位的 1 信号,这个停止位可以约定持续 1 位、1.5 位或 2 位的时间宽度。至此一个字符传送完毕,线路又进入空闲,持续为 1。经过一段随机的时间后,下一个字符开始传送才又发出起始位。

每一个数据位的宽度等于传送波特率的倒数。微机异步串行通信中,常用的波特率为 50、95、110、150、300、600、1 200、2 400、4 800、9 600 等。

接收方按约定的格式接收数据,并进行检查,可以查出以下 3 种错误。

- ① 奇偶错:在约定奇偶检查的情况下,接收到的字符奇偶状态和约定不符。
- ② 帧格式错:一个字符从起始位到停止位的总位数不对。
- ③ 溢出错:若先接收的字符尚未被微机读取,后面的字符又传送过来,则产生溢出错。

每一种错误都会给出相应的出错信息,提示用户处理。

提示:RDC-CS4510 实验板上的 UART 通信格式采用:4 800 波特、8 位数据、1 位停止、无校验。

## (2) 串行接口的物理层标准

通用的串行 I/O 接口有许多种,现仅就 RDC-CS4510 实验板上采用的 EIA RS-232C 标准作简单介绍。

EIA RS-232C 标准美国电子工业协会推荐的一种标准(Electronic industries Association Recoil-mended Standard)。它在一种 25 针接插件(DB-25)上定义了串行通信的有关信号。这个标准后来被世界各国所接受并使用到计算机的 I/O 接口中。

### ① 信号连线

在实际异步串行通信中,并不要求用全部的 RS-232C 信号,许多 PC/XT 兼容机仅用 15 针接插件(DB-15)来引出其异步串行 I/O 信号,而 PC 中更是大量采用 9 针接插件(DB 9)来担当此任。

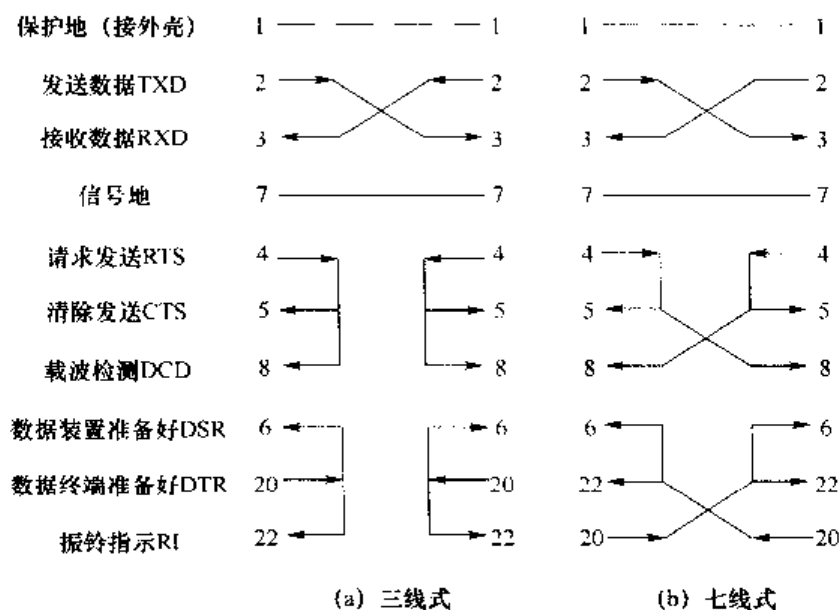


图 7-18 串口连线方式

提示:实验板上的 UART 通信格式采用三线接口(RXD、TXD、GND)(如图 7-18 所示)。

TXD/RXD 是一对数据线, TXD 称发送数据输出, RXD 称接收数据输入。当两台微机以全双工方式直接通信(无 MODEM 方式)时, 双方的这两根线应交叉连接(扭接)。

## ② 信号地

所有的信号都要通过信号地线构成耦合回路。通信线有以上 3 条(TXD、RXD 和信号地)就能工作了。其余信号主要用于双方设备通信过程中的联络(握手信号), 而且有些信号仅用于和 MODEM 的联络。若设置成不要任何联络信号, 则其他线都可不接。有时在通信线的同一端将相关信号短接以“自握手”方式满足联络要求。

## (3) 实例程序清单

```
#define ULCON0 (* (volatile unsigned *)0x03FFD000) //UART channel0
line control register

#define UCON0 (* (volatile unsigned *)0x03FFD004) //UART channel0
control register

#define USTAT0 (* (volatile unsigned *)0x03FFD008) //UART channel0
status register

#define UTXBUF0 (* (volatile unsigned *)0x03FFD00c) //UART channel0
transmit holding register

#define URXBUF0 (* (volatile unsigned *)0x03FFD010) //UART channel0
receive buffer register

#define UBRDIV0 (* (volatile unsigned *)0x03FFD014) //Baud rate divi-
sor register0

#define ULCON1 (* (volatile unsigned *)0x03FFE000) //UART channel1
line control register

#define UCON1 (* (volatile unsigned *)0x03FFE004) //UART channel1
control register

#define USTAT1 (* (volatile unsigned *)0x03FFE008) //UART channel1
status register

#define UTXBUF1 (* (volatile unsigned *)0x03FFE00c) //UART channel1
transmit holding register

#define URXBUF1 (* (volatile unsigned *)0x03FFE010) //UART channel1
recieve buffer register

#define UBRDIV1 (* (volatile unsigned *)0x03FFE014) //Baud rate divi-
sor register1

void InitUART(int Port,int Baudrate);
void PrintUART(int Port,char *s);
int uart()
```

```

{
InitUART(1,0x1440); //19200bps CPU 工作频率 50MHz 0 = COM1;1 = COM2
for(int i=0;i<100;i++){
PrintUART(0,"Communcation Testting! \r\n");
}
return(0);
}

void PrintUART(int Port,char *s)
{
if(Port == 0)
for(; *s! = '\0';s++)
{

/ *****
***** 需要读者添加 *****
* 将数据送到串行口 Buffer 中 *
***** /

if(Port == 1)
for(; *s! = '\0';s++)
{

/ *****
***** 需要读者添加 *****
* 将数据送到串行口 Buffer 中 *
***** /

}
}

void InitUART(int Port,int Baudrate)
{
if(Port == 0)
{
ULCON0 = 0x03;
UCON0 = 0x09;
UBRDIV0 = Baudrate;

```

```

 }
 if(Port == 1)
 {
 ULCON1 = 0x03;
 UCON1 = 0x09;
 UBRDIV1 = Baudrate;
 }
}

```

### 7.3 实验3 ARM 的交通灯实验

#### 1. 实验目的

(1) 掌握 ARM 嵌入式微处理器的通用 I/O 口工作原理。利用北京邮电大学电信工程学院 IRDC 实验室开发的 IRDC05v3 实验板上的通用 I/O 口实现模拟交通灯的定时控制。

(2) 学习并掌握基于 ARM 开发板对 I/O 编程的方法。并学习通过通用 I/O 口实现 LED 显示原理。

(3) ARM 的通用 I/O 口相关寄存器的功能,熟悉 ARM 系统硬件的通用 I/O 口工作原理相关接口的复用方法。

#### 2. 实验内容

(1) 利用实验板上 S3C4510B 提供的 18 个可编程的通用 I/O 端口(用户可将每个端口配置为输入模式、输出模式)学习通用 I/O 口工作原理及其寄存器状态字的定义内容。进一步了解可编程的通用 I/O 端口控制器的工作过程。

(2) 阅读 ARM 芯片文档,掌握 ARM 的通用 I/O 口相关寄存器的功能,熟悉 ARM 系统硬件的通用 I/O 口工作原理相关接口的复用方法。

(3) 利用 IRDC05v3 实验板上的通用 I/O 口编程实现交通灯的定时控制,通过控制相应的 I/O 口的输入输出的工作方式,以及输出相应的高低电平使接在 P16~P17 口的 LED 显示器轮流被点亮。再此基础上实现进一步地复杂控制。

#### 3. 实验设备

##### (1) 硬件环境

- 北京邮电大学电信工程学院 IRDC 实验室开发的 CS4510 教学实验/评估板;
- PC;
- 10/100 M 以太网连接(可选双网口) + 直连串口连接;
- 用于 ARM7TDMI 的 JTAG 仿真器(可选)。

##### (2) 软件开发环境

- WindRiver Tornado 2.0 above Integrated Development Environment for ARM;

- Win98、Win2000 或 WinXP 操作系统；
- ARM SDT 2.51 集成开发环境、仿真器驱动程序(汇编方式调试可选)；
- 超级终端通信程序(C 语言/ARM 汇编语言)。

#### 4. 预备知识(见附录)

(1) 熟练使用 VxWorks 操作系统的 WindRiver Tornado 集成开发环境, 熟悉编写和下载调试程序的基本过程。

(2) 熟悉 VxWorks 应用程序的框架结构及其开发流程以及 VxWorks 任务的发起。

(3) 了解通用 I/O 口工作原理及其复用方法。

(4) 熟悉 ARM 的通用 I/O 口相关寄存器的功能, 熟悉 ARM 系统硬件的通用 I/O 口工作原理相关接口的复用方法。

## 5. ARMS3C4510B 的通用 I/O 口相关寄存器状态字介绍

S3C4510B 提供了 18 个可编程的通用 I/O 端口,用户可将每个端口配置为输入模式、输出模式或特殊功能模式,由片内的特殊功能寄存器 IOPMOD 和 IOPCON 控制。输入模式数据流向由外到内,用于数据的输入、检测。输出模式数据流向由外到内,用于数据的输出、控制。特殊功能模式与其他的功能复用,完成更为复杂的控制。具体来说,端口 0~7 的工作模式仅由 IOPMOD 寄存器控制;但通过设置 IOPCON 寄存器,端口 8~11 可用作外部中断请求 INTREQ0~INTREQ3 的输入;端口 12、端口 13 可用作外部 DMA 请求 XDREQ0、XDREQ1 的输入;端口 14、端口 15 可作为外部 DMA 请求的应答信号 XDACK0、XDACK1;端口 16 可作为定时器 0 的溢出 TOUT0;端口 17 可作为定时器 1 的溢出 TOUT1。所以要想真正掌握 18 个可编程的通用 I/O 端口的编程工作,就要首先熟悉通用 I/O 口相关寄存器状态字以及其工作流程(如图 7-19 所示)。

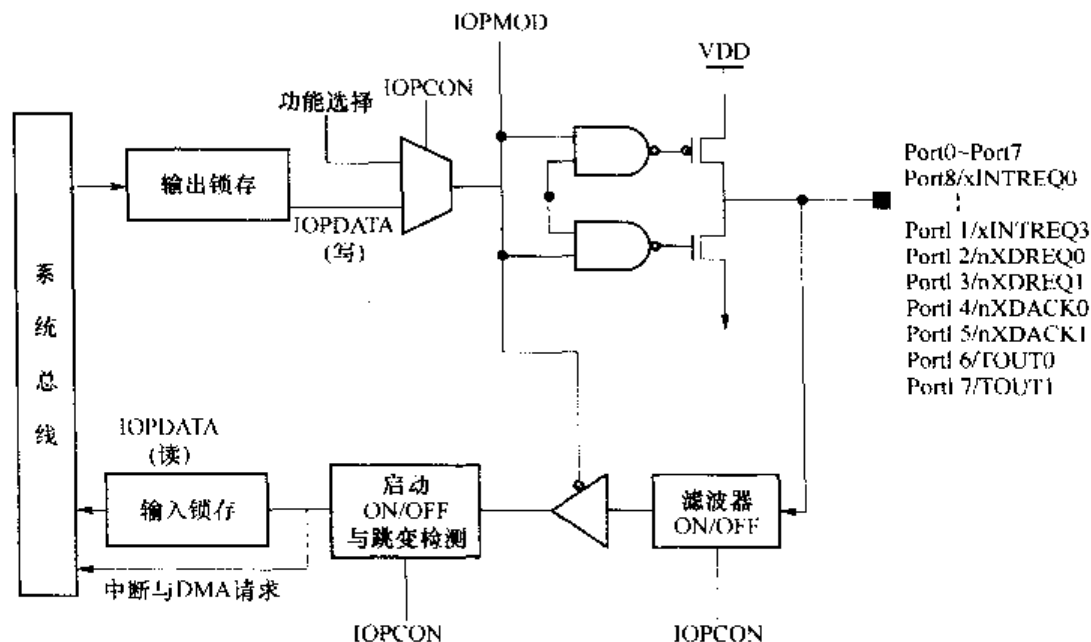


图 7-19 I/O 工作流程



## (1) I/O 口模式寄存器(IOPMOD)

I/O 口模式寄存器 IOPMOD 用于配置 P17~P0, 如表 7-13 与图 7-20 所示。

表 7-13 I/O 口模式寄存器

| 寄存器    | 偏移地址   | 操作  | 功能描述       | 复位值         |
|--------|--------|-----|------------|-------------|
| IOPMOD | 0x5000 | 读/写 | I/O 口模式寄存器 | 0x0000,0000 |

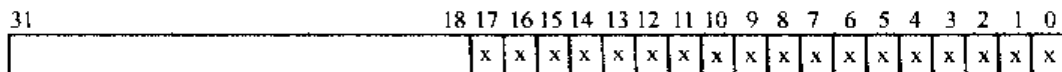


图 7-20 I/O 口模式寄存器

I/O 口模式寄存器的各位描述如下。

[0]: P0 口的 I/O 模式位。0=输入, 1=输出。

[1]: P1 口的 I/O 模式位。0=输入, 1=输出。

[2]: P2 口的 I/O 模式位。0=输入, 1=输出。

[3~17]: P3~P17 口的 I/O 模式位。0=输入, 1=输出。

## (2) I/O 口控制寄存器(IOPCON)

I/O 口控制寄存器 IOPCON 用于配置端口 P8~P17 的特殊功能, 当这些端口用作特殊功能(如外部中断请求、外部中断请求应答、外部 DMA 请求或应答、定时器溢出)时, 其工作模式由 IOPCON 寄存器控制, 而不再由 IOPMOD 寄存器。对于特殊功能输入端口, S3C4510B 提供了一个滤波器用于检测特殊功能信号的输入, 如果输入信号电平宽度等于三个系统时钟周期, 该信号被认为是诸如外部中断请求或外部 DMA 请求等特殊功能的信号(具体如表 7-14 与图 7-21 所示)。

表 7-14 I/O 口控制寄存器

| 寄存器    | 偏移地址   | 操作  | 功能描述       | 复位值         |
|--------|--------|-----|------------|-------------|
| IOPCON | 0x5004 | 读/写 | I/O 口控制寄存器 | 0x0000,0000 |

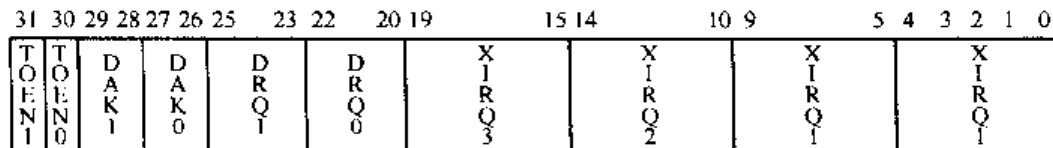


图 7-21 I/O 口控制寄存器

[4:0] 控制端口 8 的外部中断请求信号 0(xIRQ0)输入

[4] 端口 8 用作外部中断请求信号 0

- 0 = 禁止, 1 = 使能
- [3] 0 = 低电平有效, 1 = 高电平有效
- [2] 0 = 滤波器关, 1 = 滤波器开
  
- [1:0] 00 = 电平检测, 01 = 上升沿检测  
10 = 下降沿检测, 11 = 上升、下降沿均检测
  
- [9:5] 控制端口 9 的外部中断请求信号 1(xIRQ1)输入, 使用方法同端口 8
- [14:10] 控制端口 10 的外部中断请求信号 2(xIRQ2)输入, 使用方法同端口 8
- [19:15] 控制端口 11 的外部中断请求信号 3(xIRQ3)输入, 使用方法同端口 8
- [22:20] 控制端口 12 的外部 DMA 请求信号 0(DRQ0)输入
- [22] 端口 12 用作外部 DMA 请求信号 0(nXDREQ0)  
0 = 禁止, 1 = 使能
- [21] 0 = 滤波器关, 1 = 滤波器开
- [20] 0 = 低电平有效, 1 = 高电平有效
- [25:23] 控制端口 13 的外部 DMA 请求信号 1(DRQ1)输入
- [25] 端口 13 用作外部 DMA 请求信号 1(nXDREQ1)  
0 = 禁止, 1 = 使能
- [24] 0 = 滤波器关, 1 = 滤波器开
- [23] 0 = 低电平有效, 1 = 高电平有效
- [27:26] 控制端口 14 的外部 DMA 应答信号 0(DAK0)输出
- [27] 端口 14 用作外部 DMA 信号 0(nXDACK0)  
0 = 禁止, 1 = 使能
  
- [26] 0 = 低电平有效, 1 = 高电平有效
- [29:28] 控制端口 15 的外部 DMA 应答信号 1(DAK1)输出
- [29] 端口 15 用作外部 DMA 信号 1(nXDACK1)  
0 = 禁止, 1 = 使能
- [28] 0 = 低电平有效, 1 = 高电平有效
- [30] 控制端口 16 作为定时器 0 溢出信号(TOEN0)  
0 = 禁止, 1 = 使能
- [31] 控制端口 17 作为定时器 1 溢出信号(TOEN1)  
0 = 禁止, 1 = 使能

### (3) I/O 口数据寄存器(IOPDATA)

当配置为输入模式时, 读取 I/O 口数据寄存器 IOPDATA 的每一位对应输入状态, 当配置为输出模式时, 写每一位对应输出状态。位[17:0]对应于 18 个 I/O 引脚 P17~P0(如表



nel0 transmit holding register

```
#define URXBUF0 (* (volatile unsigned *)0x03FFD010) //UART chan-
```

nel0 receive buffer register

```
#define UBRDIV0 (* (volatile unsigned *)0x03FFD014) //Baud rate
```

divisor register0

```
#define ULCON1 (* (volatile unsigned *)0x03FFE000) //UART chan-
```

nel1 line control register

```
#define UCON1 (* (volatile unsigned *)0x03FFE004) //UART chan-
```

nel1 control register

```
#define USTAT1 (* (volatile unsigned *)0x03FFE008) //UART chan-
```

nel1 status

register

```
#define UTXBUF1 (* (volatile unsigned *)0x03FFE00c) //UART chan-
```

nel1 transmit holding register

```
#define URXBUF1 (* (volatile unsigned *)0x03FFE010) //UART chan-
```

nel1 receive buffer register

```
#define UBRDIV1 (* (volatile unsigned *)0x03FFE014) //Baud rate
```

divisor register1

将上述内容定义为一个有一定功能的头文件,并且加入到工程中。

(3) 在程序的文件中添加下面的执行函数,并且建立一个文件(\*.c)包含各函数。这些函数的功能主要为如下几个。

① 在主函数中首先初始化通用 I/O 口相关寄存器

```
void InitIO();
```

通过查询状态字具体含义完成 I/O 口模式寄存器(IOPMOD)以及 I/O 口控制寄存器(IOPCON)将实验需要使用 IO 口相应的工作方式,如置为输出模式,将复用功能禁止。

② 点灯程序

```
void IOEN()
```

通过查询状态字具体意义了解 I/O 口数据寄存器(IOPDATA),将置为输出模式的 I/O 口,送入相应数据。

③ 定时函数(可选)

```
void Delay(unsigned int x)
```

传入参数  $x$  为相应的延时值。定时函数可以自选或者使用系统提供的函数。

(4) 在完成了基本函数的编写以后,完成主函数的编写,发起任务。

```
int uart()
```

```
{
```

```
 InitIO();
```

```

void IOEN()
Delay(unsigned int x)
return(0);
}

```

#### (5) 编译、下载、运行程序

可以把应用程序直接加在 VxWorks 映像文件中,将映像下载后应用程序会自动运行,可以直接在 Shell 中或者控制台中观察结果,不需要再手动下载应用程序。如果没有将应用程序和 VxWorks 映像文件捆绑在一起,在 Shell 中输入任务启动函数名,然后回车,也可以运行程序。如果成功执行则会返回 0,否则会看到错误号或者其他错误信息。

#### (6) 观察、分析运行结果

观察开发板的 led 灯是否能被智能控制,模拟交通灯显示情况。

观察 Shell 中或者控制台上的消息显示情况,借助一些调试工具如 WindView 或者在 Shell 中输入 shell 命令等方法监视任务执行的情况,分析任务的一些参数和特性。另外,分析目标机中任务占用资源的情况(可以启动 Browser 工具查看)。

### 7. 思考题

- (1) 利用 VxWorks 提供的相应函数实现更加复杂的 I/O 控制。
- (2) 有没有更好的控制策略算法?

## 7.4 实验 4 ARM 的键盘接口实验原理实验

### 1. 实验目的

(1) 掌握 ARM 嵌入式微处理器 S3C4510B 的通用 I/O 口工作原理。利用北京邮电大学电信工程学院 IRDC 实验室开发的 IRDC05v3 实验板上的通用 I/O 口学习 S3C4510B 的 I/O 扩展方式以及学习它的编址方式和总线宽度控制。

(2) 学习并掌握基于键盘工作的原理和相关寄存器编程的方法,学习 S3C4510B 的中断接口的使用方法。

(3) 学习在 VxWorks 下硬件编写一个中断程序的方法。

(4) 学习信号量的使用。

### 2. 实验内容

(1) 利用实验板上 S3C4510B 提供的 18 个可编程的通用 I/O 端口(用户可将每个端口配置为输入模式、输出模式)学习通用 I/O 口工作原理及其寄存器状态字的定义内容,进一步了解可编程的通用 I/O 端口控制器的工作过程。

(2) 根据键盘的接口电路图和键盘的原理写一个中断方式的键盘程序。实现在键盘上按键,在主机的高级终端或串口调试助手上有相应的按键输出。

### 3. 预备知识

(1) 熟练使用 VxWorks 操作系统的 WindRiver Tornado 集成开发环境,熟悉编写和

下载调试程序的基本过程。

(2) 熟悉 VxWorks 应用程序的框架结构及其开发流程以及 VxWorks 任务的发起。

(3) 熟悉 S3C4510B 的通用 I/O 接口及 I/O 寄存器工作原理及其复用方法。

(4) 熟悉 S3C4510B 中断寄存器以及中断程序、信号量的使用。

(5) 熟悉 ARM 的通用 I/O 口相关寄存器的功能,熟悉 ARM 系统硬件的通用 I/O 口工作原理以及相关接口的复用方法。

#### 4. 实验设备

##### (1) 硬件环境

- 北京邮电大学电信工程学院 IRDC 实验室开发的 CS4510 教学实验/评估板;
- PC;
- 10/100 M 以太网连接(可选双网口) + 直连串口连接;
- 用于 ARM7TDMI 的 JTAG 仿真器(可选)。

##### (2) 软件开发环境

- WindRiver Tornado 2.0 above Integrated Development Environment for ARM;
- Win98、Win2000 或 WinXP 操作系统;
- ARM SDT 2.51 集成开发环境、仿真器驱动程序(汇编方式调试可选);
- 超级终端通信程序(C 语言/ARM 汇编语言)。

#### 5. 实验原理

##### (1) 键盘接口

如图 7-23 所示,4×4 键盘工作于中断方式,它有 8 个引出脚,分别接在 CPU 通用 I/O 的 p7~p0 管脚,其中,键盘列线 p3~p0 通过与逻辑形成 CPU 的中断 3 信号线(也就是 CPU 通用 I/O 的 p11 脚)。

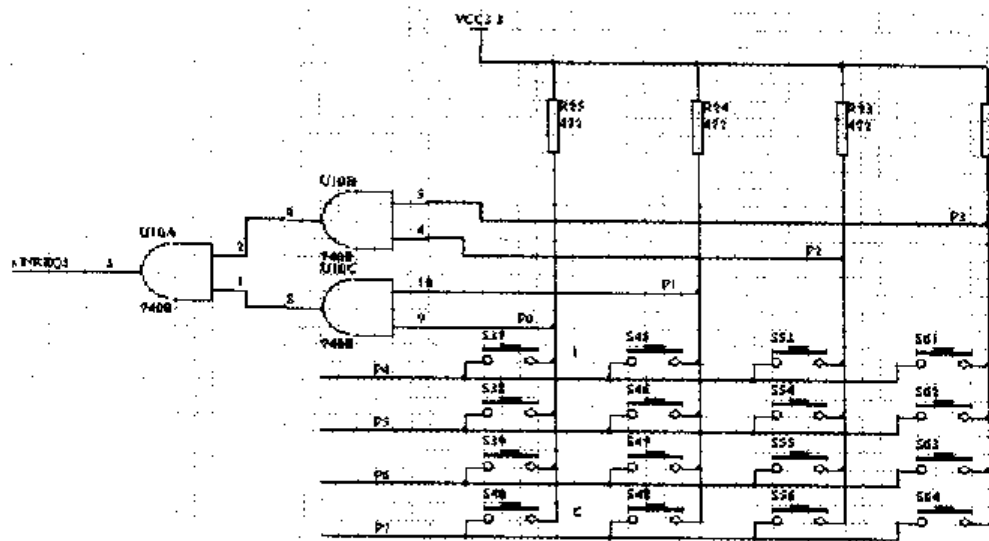


图 7-23 键盘接口硬件连接图

S3C4510B 有 18 个通用可编程 I/O 口,根据功能的不同可以配置成输入模式、输出模式和特殊功能模式。本实验中需把 p4~p7 设置为输出,p3~p0 设置为输入,p11 设置为中断输入,低电平有效。

S3C4510B 的 I/O 寄存器:S3C4510B 有 3 个 I/O 寄存器 IOPMOD、IOPCON 和 IOPDATA,分别设定它们,可以定义 18 个通用可编程 I/O 口的功能。(具体定义参见前面的试验)其中:IOPMOD 定义通用 I/O 口作为输入或是输出;IOPCON 定义 PORT8~PORT17 作为特殊功能;IOPDATA 反映 18 个相应 I/O 口的信号电平,如是输出设定相应位,如果是输入则读入相应位。

### (2) 键盘工作原理

实现键盘有两种方案:一是采用现有的一些专用的芯片实现键盘扫描;二是用软件实现键盘扫描。作为一个嵌入系统设计人员,总会关心产品成本,键盘扫描的软件实现方法有助于缩减一个系统的重复开发成本,且也只需要很少的 CPU 开销。目前嵌入式控制器的功能很强,可以充分利用这一资源,这里就介绍一下我们这个开发板使用的软件键盘方案。

参考键盘接口图,键盘工作于中断方式。空闲时,p4~p7 设置为输出,并且为 0,当有键被按下时,立即会使中断信号有效,产生中断。然后,CPU 会在 p4~p7 口依次输出 1110、1101、1011、0111,分别读入 p0~p3,就可以知道究竟是哪一个键被按下。

注意:如图 7-23 所示,4×4 键盘中每个键使用了一个瞬时接触开关的简单电路,微处理直接检测按键的闭合状态。当开关打开时,通过处理器的 I/O 口的一个上拉电阻提供逻辑 1;当开关闭合时,处理器的 I/O 口的输入将被拉低到逻辑 0。但是,这个开关太不完善,当它们被按下或者被释放时,并不能够产生一个明确的 1 或者 0,而是要有 5~30 ms 之间的按键抖动,这将直接影响到微处理器的判断。因此,必须通过软件方式消除按键抖动。

### (3) 中断控制器

总共有 21 个中断源,主要设置 4 个中断寄存器。

- INTMOD 中断模式寄存器,确定每个中断源为 FIQ 中断或是 IRQ 中断。
- INTPND 中断 pending 寄存器,反映是否有中断请求。
- INTMSK 中断屏蔽寄存器,屏蔽相应的中断源。
- INTPRI0~5 中断优先级寄存器,设定各个中断的优先级。

## 6. 实验步骤

### (1) 建立一个 downloadable 工程

启动 Tornado,新建一个 downloadable 工程,选择工具链 ARMARCH4gnu。

### (2) 工程建立成功后,添加文件 key4\_4.c,编写键盘程序(如图 7-24 所示)。

这里有 3 点需要学习:

- 加载中断函数,见前面的电路图,选用中断 3;

- 信号量负责中断函数和键盘扫描函数的交互；
- 键盘去抖。

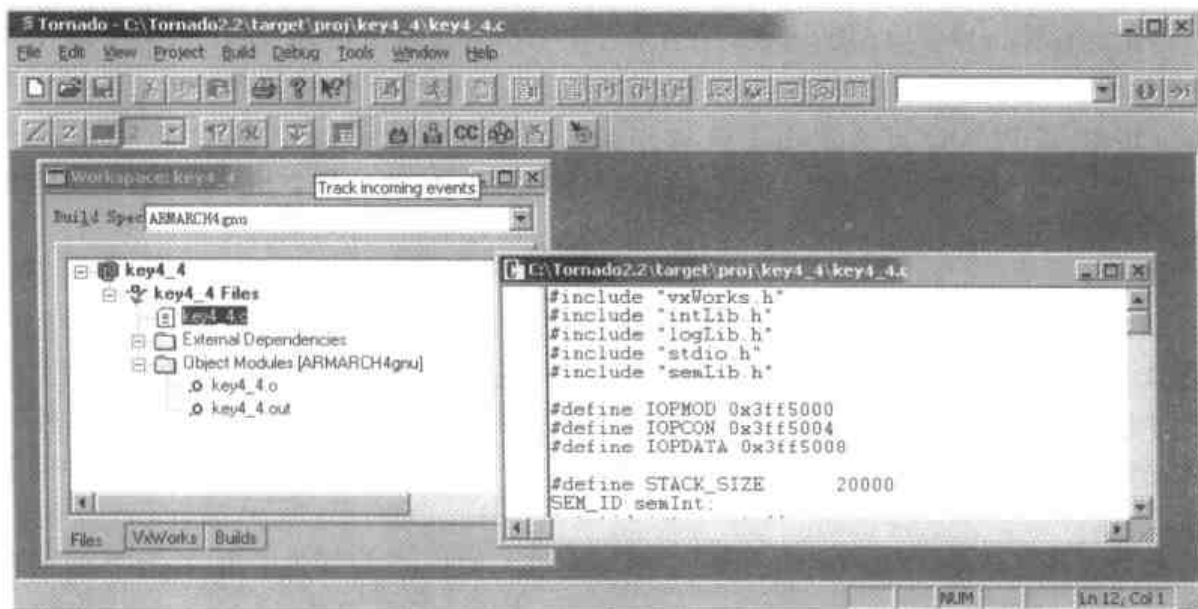


图 7-24 工程实例

key4\_4.c

```

/ ***** key4_4.c start ***** /
#include "VxWorks.h"
#include "intLib.h"
#include "logLib.h"
#include "stdio.h"
#include "semLib.h"

#define IOPMOD 0x3ff5000
#define IOPCON 0x3ff5004
#define IOPDATA 0x3ff5008
#define STACK_SIZE 20000
SEM_ID semInt;

int tid_interrupt, flag;
void key_check(void);
char KEY_code;
STATUS progStart (void)
{
 flag = 0;

```



```

semInt = semBCreate(SEM_Q_FIFO, SEM_FULL);
semTake(semInt, WAIT_FOREVER);
tid_interrupt = taskSpawn("tInterrupt", 199, 0, STACK_SIZE,
 (FUNCPTR) key_check, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0);
return(OK);
}
/* key program */
void out_key_scan(char scan)
{ int temp;
 temp = * (UINT32 *)IOPDATA;
 temp = temp & 0xfffff0f; /* active low, falling edge detection */
 temp = temp | scan;
 * (UINT32 *)IOPDATA = temp;
}
void interruptkey(int arg)
{ int lockkey, i;
 lockkey = intLock();
 logMsg("into interrupt\n", 0, 0, 0, 0, 0, 0);
 /* for(i = 0; i < counter_i; i++)
 for(arg = 0; arg < 60000; arg++); */
 * (UINT32 *)0x3ff4004 = 0x08;
 semGive(semInt);
 intUnlock(lockkey);
}
void key_check(void)
{ int temp, i, j, k, lockkey;
 char scan;
 /* interrupt 3 iopcon enable */
 temp = * (UINT32 *)IOPCON;
 temp = temp & 0xffff07fff;
 temp = temp | 0xb0000; /* active low, falling edge detection */
 * (UINT32 *)IOPCON = temp;
 temp = * (UINT32 *)IOPMOD;
 temp = temp | 0xf0; /* p3~p7 output */
 * (UINT32 *)IOPMOD = temp;
}

```

```

intEnable(3);
temp = * (UINT32 *)0x3ff4008; /* interrupt mask */
temp = temp & 0x0fff7; /* active low, falling edge detection */
* (UINT32 *)0x3ff4008 = temp;
out_key_scan(0x00);

/ *****
***** 需要读者添加 *****
* 将中断向量和中断服务程序挂接 *
***** /

{ logMsg("intConnect failed\n",0,0,0,0,0,0);
 return 0;
!
logMsg("intConnect succeeded\n",0,0,0,0,0,0);
for(;;)
{
 if(flag == 1)break;
 semTake(semInt, WAIT_FOREVER);
 KEY_code = 0xff;
 for(i = 0; i < 100; i++);
 lockkey = intLock();
 scan = 0xef;
 for(ii = 0; ii < 4; ii++)
 {
 out_key_scan(Iscan);
 scan = scan << 1;
 for(k = 0; k < 100; k++);
 temp = * (UINT32 *)IOPDATA;
 switch(temp & 0xf)
 { case 0xe ;
 if(i == 0)KEY_code = 0x1;
 if(i == 1)KEY_code = 0x5;
 if(i == 2)KEY_code = 0x9;
 if(i == 3)KEY_code = 'c';
 break;

```

```

 /* printf("p %d-p0\n",4+i); */
 case 0xd :
 if(i == 0)KEY_code = 0x2;
 if(i == 1)KEY_code = 0x6;
 if(i == 2)KEY_code = 0x0;
 if(i == 3)KEY_code = 'd';
 break; /* printf("p %d-p1\n",4+i); */
 case 0xb :
 if(i == 0)KEY_code = 0x3;
 if(i == 1)KEY_code = 0x7;
 if(i == 2)KEY_code = 'a';
 if(i == 3)KEY_code = 'e';
 break; /* printf("p %d-p2\n",4+i); */
 case 0x7 :
 if(i == 0)KEY_code = 0x4;
 if(i == 1)KEY_code = 0x8;
 if(i == 2)KEY_code = 'b';
 if(i == 3)KEY_code = 'f';
 break; /* printf("p %d-p3\n",4+i); */
 default,
 /* printf("multi key"); */
 }
}

for(j = 0;j<10;j++)for(k = 0;k<65530;k++);
*(UINT32 *)0x3ff4004 = 0x08;
if(KEY_code<= 9)printf("key is %d\n",KEY_code);
else printf("key is %c\n",KEY_code);
intUnlock(lockkey);
out_key_scan(0x00);
}

}

void progStop(void)
{
 flag = 1;
}

/ ***** key4_4.c stop ***** /

```

(4) 编译成功后,下载执行。在 Shell 下运行 progStart。在键盘上键入任何按键,在超级终端上会有相应的按键输出,如图 7-25 所示。

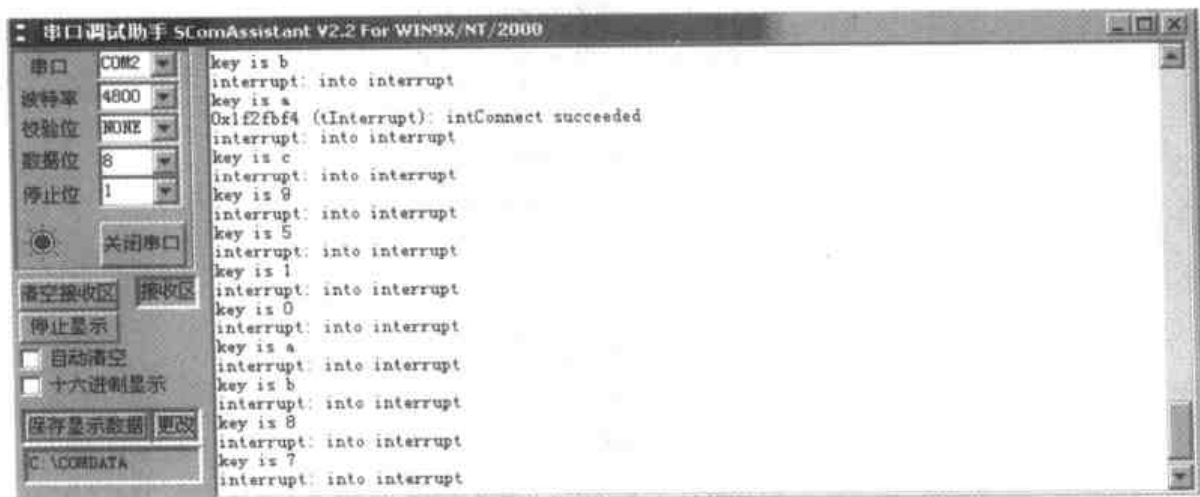


图 7-25 串口显示

借助一些调试工具如 WindView 或者在 Shell 中输入 shell 命令等方法监视任务执行的情况,分析任务的一些参数和特性。另外,分析目标机中任务占用资源的情况(可以启动 Browser 工具查看)。

## 7. 思考题

- (1) 查询方式的键盘驱动的实现。
- (2) 第二功能键的编程实现。

## 7.5 实验 5 ARM 的液晶接口实验原理实验

### 1. 实验目的

(1) 掌握 ARM 嵌入式微处理器的通用 I/O 口工作原理。利用北京邮电大学电信工程学院 IRDC 实验室开发的 IRDC05v3 实验板上的通用 I/O 口学习 S3C4510B 的扩展 I/O 方式以及学习它的编址方式和总线宽度控制。

(2) 学习并掌握基于学习 T6963C 液晶控制器的原理和相关寄存器编程的方法。

(3) 学习总线方式对液晶控制器的驱动方法以及进一步熟悉 ARM 的通用 I/O 口相关寄存器的功能,熟悉 ARM 系统硬件的通用 I/O 口工作原理相关接口的复用方法。

### 2. 实验内容

(1) 利用实验板上 S3C4510B 提供的 18 个可编程的通用 I/O 端口(用户可将每个端口配置为输入模式、输出模式)学习通用 I/O 口工作原理及其寄存器状态字的定义内容,

进一步了解可编程的通用 I/O 端口控制器的工作过程。

(2) 阅读 ARM 和 T6963C 液晶控制器芯片文档,掌握 ARM 的通用 I/O 口相关寄存器的功能,熟悉 ARM 系统硬件的通用 I/O 口工作原理相关接口的复用方法。根据液晶的接口电路图和 T6963C 液晶控制器的工作方式写一个液晶显示程序,实现对液晶的光标和字体显示方式进行控制。

(3) 编程实现利用 IRDC05v3 实验板上的通用 I/O 口实现液晶显示控制,并且通过控制相应的 I/O 口的输入输出的工作方式,实现一定的时间间隔全屏显示 0~F,并有正向和负向两种显示方式,最后显示一句欢迎词“welcome you”,再此基础上实现进一步的复杂控制。

### 3. 实验设备

#### (1) 硬件环境

- 北京邮电大学电信工程学院 IRDC 实验室开发的 CS4510 教学实验/评估板;
- PC;
- 10/100 M 以太网连接(可选双网口) + 直连串口连接;
- 用于 ARM7TDMI 的 JTAG 仿真器(可选)。

#### (2) 软件开发环境

- WindRiver Tornado 2.0 above Integrated Development Environment for ARM;
- Win98、Win2000 或 WinXP 操作系统;
- ARM SDT 2.51 集成开发环境、仿真器驱动程序(汇编方式调试可选);
- 超级终端通信程序(C 语言/ARM 汇编语言)。

### 4. 预备知识

(1) 熟练使用 VxWorks 操作系统的 WindRiver Tornado 集成开发环境,熟悉编写和下载调试程序的基本过程。

(2) 熟悉 VxWorks 应用程序的框架结构及其开发流程以及 VxWorks 任务的发起。

(3) 进一步熟悉 T6963C 液晶控制器的原理和接口寄存器工作原理及其复用方法。

(4) 熟悉 ARM 的通用 I/O 口相关寄存器的功能,熟悉 ARM 系统硬件的通用 I/O 口工作原理相关接口的复用方法。

### 5. T6963C 液晶控制器的接口定义及实验板的接口原理图

#### (1) 液晶原理

LCD 显示器的基本原理就是通过给不同的液晶单元供电,控制其光线的通过与否,从而达到显示的目的。因此,LCD 的驱动控制归于对每个液晶单元的通断电的控制,每个液晶单元都对应着一个电极,对其通电,便可使光线通过,也有刚好相反的,即不通电时光线通过,通电时光线不通过(详细资料参见液晶的相关资料和书籍)。

一般市面上出售的 LCD 有两种类型:一种是带有驱动电路的 LCD 显示模块(如本实

验中这款液晶采用 T6963C 控制器),这种 LCD 可以方便地与各种微处理器进行接口,如 8051、S3C4510B 等,但是由于硬件驱动电路的存在,体积比较大,这种模式常常使用总线方式来驱动;另一种是 LCD 显示屏,没有驱动电路,需要与驱动电路配合使用。特点是体积小,但却需要另外的驱动芯片,也可以使用带有 LCD 驱动能力的高档 MCU 驱动,如 ARM 系列的 S3C44BOX。在中规模图形式液晶显示模块中,内置 T6963C 控制器的液晶显示模块是目前较为常用的一种。该液晶显示模块由液晶显示控制器 T6963C 及其周边电路、行驱动器 T6A40 组、列驱动器 T6A39 组、液晶驱动偏压电路、显示存储器以及液晶屏组成。显示模块对外仅是一个 20 芯的双列扁平电缆接口。

## (2) 实验板选用的液晶模块的功能特点

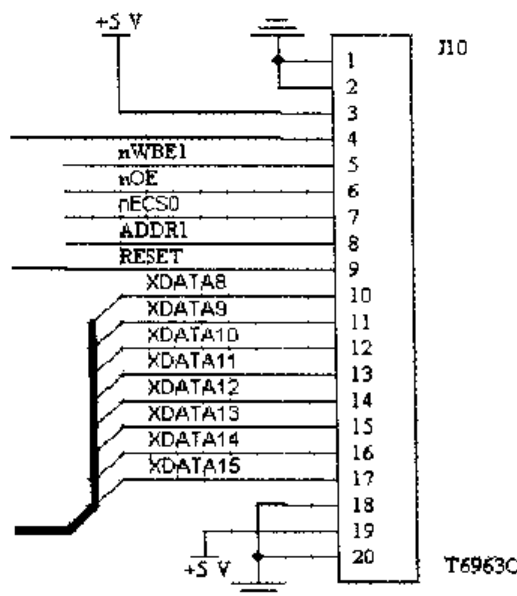


图 7-26 实验板的液晶接口

对于使用内置 T6963C 控制器的液晶显示模块的用户而言,用户无需了解 T6963C 对液晶屏的显示驱动、点阵扫描、显示存储器管理等操作,这一切都由 T6963C 自动进行。用户需要了解的是 T6963C 各种数据/指令格式、显示存储器的区间划分和接口引脚的功能定义。对微处理器而言显示模块对外的接口与一片普通的 I/O 接口芯片(如 8255A)类似,按引脚功能分别连接于微处理器的三大总线上即可(具体的电路如图 7-26 所示)。微处理器对 T6963C 的操作有 4 类共 13 条:一是读状态字指令(1 条),在 T6963C 中有一个一字节的字状态字,微处理器在对 T6963C 的每一次操作前都要读这个状态字并对其中的相应位加以判断,以决定是否可以对 T6963C 进行操作;二是设置指令(9 条),该类指令用于设置显示的区域、方式及数据地址指针,设置光标的形状和数据的读写方式等;三是位操作指令(1 条),该指令专用于对液晶屏上的像素(点)操作;四是数据的读/写指令(2 条),该指令读/写的的数据即为液晶屏上所显示的内容。

表 7-16 T6963C 液晶控制器的接口定义

| PIN NO | SYMBOL  | LEVEL | FUNCTION                                               |
|--------|---------|-------|--------------------------------------------------------|
| 1      | FG      |       | 框地                                                     |
| 2      | Vss     | 0 V   | 地                                                      |
| 3      | Vdd     | +5 V  | 工作电压                                                   |
| 4      | VO      |       | LCD屏操作电压(调节VO的值可以调节显示对比度)                              |
| 5      | /WR     | L     | 信号写                                                    |
| 6      | /RD     | L     | 信号读                                                    |
| 7      | /CE     | L     | 片选                                                     |
| 8      | C/D     | L/H   | WR=L: C/D=H 命令写 C/D=L 数据写<br>RD=L: C/D=H 状态读 C/D=L 数据读 |
| 9      | /RST    | L/H   | L 初始化 T6963C<br>H 正常                                   |
| 10~17  | DB0-DB7 | L/H   | 数据线                                                    |
| 18     | FS      |       | 字体选择L: 8X8 字体 H: 6X8 字体                                |
| 19     | SLA     |       | LED 背光源正极(见参数表)                                        |
| 20     | SLK     | 0 V   | LED 背光源负极                                              |
| 21     | VOUT    |       | 负电压输出                                                  |

如图 7-26 和表 7-16 所示,液晶控制器的数据线 DB7~DB0 接在 S3C4510B 的高 8 位数据线 XDATA8~XDATA15;液晶控制器的数据命令线(C/D)接在 S3C4510B 的地址线 ADDR1;液晶控制器的片选线接在 S3C4510B 的 nECS0。

### (3) 地址定义及工作原理

S3C4510B 中涉及到扩展 I/O 控制有 3 个寄存器 EXTACON0~EXTACON1、EXTDBWTH、REFEXTCON,分别设定它们,可以确定扩展 I/O 的时序、总线宽度和 I/O 基地址。其中:

- EXTACON0 确定扩展 I/O bank0 和 bank1 的访问时间;
- EXTDWBWTH 确定各个扩展外设的总线宽度,这里液晶控制器所在的 bank0 总线宽度设置为 16 位;
- REFEXTCON 确定扩展 I/O 的基地址,这里设置为 0x3fd0000。

正常情况下,以上 3 个寄存器会在 VxWorks 的 BSP 中指定,如果在 BSP 中没有设置。可以在液晶初始化时重新设定。

在液晶显示模块中配备有显示存储器(RAM),T6963C 最大可控制 64 K。该存储器经设置指令设置(区域、方式)后,存储器中被设置的空间内的每一个“位”都与液晶屏上的一个像素(点)相对应,而“位”的二值性就表示液晶屏上像素是否“显现”。T6963C 则将存储器中设置区域的内容不断地、扫描式地送向液晶屏,用户则通过显示模块对外的接口将需显示的“数据”送入存储器中的设置区域即可。

### (4) T6963C 控制器对显示存储器的分配和管理

液晶显示控制器的主要工作是将计算机写到显示存储器的数据以某种形式作为显示数据传给液晶显示驱动系统,该工作的功能强弱决定了控制器性能的优劣。T6963C 控

制器有两种形式将数据传给液晶显示驱动系统,两种形式转送的数据分别存放在两个不同的显示区中,即文本显示区和图形显示区或文本属性区。文本显示区用于文本显示方式。在此方式下文本显示区单元内的数据被控制部确认为字符代码。一个字节的数据对应显示屏上的  $8 \times 8$  点阵(像素)字符块。

### ① 图形显示区

用于图形显示方式。在此方式下图形显示区单元内的数据被控制部确认为显示屏上对应点的显示状态。1 表示显示,0 表示不显示。一个字节的数据对应显示屏上  $1 \times 8$  点阵(像素)条。

### ② 文本属性区

用于文本显示方式。该区单元内的数据被控制部确认为相应文本显示区单元所显示的字符的显示特性,如正向显示、负向显示和闪烁显示等。

图形显示区和文本属性区被划定在同一片显示存储区中,使用这片显示存储区时只能是一种性质的显示存储区,即若作为图形显示区便无文本属性,反之亦然。

嵌入式处理器对显示存储器访问实质上都是通过 T6963C 控制器进行的,通常分两步进行:送地址指针、读/写显示数据( $C/D=0$ );或送地址指针、送操作指令( $C/D=1$ )。嵌入式处理器与 T6963C 控制器的每一次数据前都要进行 T6963C 控制器的状态判断。

### (5) T6963C 指令集

T6963C 使用了硬件初始化设置,使它的指令功能集中于显示功能的设置上,从而加强了 T6963C 的显示控制能力。指令集如表 7-17 所示(更详细的信息可以参考电子工业出版社的《液晶显示应用技术》相关部分)。

表 7-17 T6963C 指令集

| 指令名称      | 控制状态 |     |     | 指令代码 |    |    |    |    |    |    |    | 参数 |
|-----------|------|-----|-----|------|----|----|----|----|----|----|----|----|
|           | C/D  | /RD | /WR | D7   | D6 | D5 | D4 | D3 | D2 | D1 | D0 |    |
| 读状态字      | 1    | 0   | 1   | S7   | S6 | S5 | S4 | S3 | S2 | S1 | S0 | 无  |
| 地址指针设置    | 1    | 1   | 0   | 0    | 0  | 1  | 0  | 0  | N2 | N1 | N0 | 2  |
| 显示区域设置    | 1    | 1   | 0   | 0    | 1  | 0  | 0  | 0  | 0  | N1 | N0 | 2  |
| 显示方式设置    | 1    | 1   | 0   | 1    | 0  | 0  | 0  | CG | N2 | N1 | N0 | 无  |
| 显示状态设置    | 1    | 1   | 0   | 1    | 0  | 0  | 1  | N3 | N2 | N1 | N0 | 无  |
| 光标形状设置    | 1    | 1   | 0   | 1    | 1  | 0  | 0  | 0  | N2 | N1 | N0 | 无  |
| 数据自动读写设置  | 1    | 1   | 0   | 1    | 0  | 1  | 1  | 0  | 0  | N1 | N0 | 无  |
| 数据一次读写设置  | 1    | 1   | 0   | 1    | 1  | 0  | 0  | 0  | N2 | N1 | N0 | 1  |
| 屏读(一字节)设置 | 1    | 1   | 0   | 1    | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 无  |
| 屏拷贝(一行)设置 | 1    | 1   | 0   | 1    | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 无  |
| 位操作       | 1    | 1   | 0   | 1    | 1  | 1  | 1  | N3 | N2 | N1 | N0 | 无  |
| 数据写操作     | 0    | 1   | 0   | 数据   |    |    |    |    |    |    |    | 无  |
| 数据读操作     | 0    | 0   | 1   | 数据   |    |    |    |    |    |    |    | 无  |



以上第5小节部分简述了 S3C4510B 的通用 I/O 口的基本工作原理,更详细的内容可参考 S3C4510B 的用户手册。

## 6. 实验步骤

(1) 明确实验板的地址分配,使用状态字设置 I/O 口以及 T6963C 的寄存器初始化硬件。在了解 ARM 上相应寄存器的功能和各位的意义前提下学习通用 I/O 口以及 T6963C 的寄存器状态字的定义内容。进一步了解可编程的通用 I/O 口以及 T6963C 的工作过程。设定寄存器的控制值,送入相应的控制寄存器 IOPMOD 和 IOPCON 控制。

(2) 在 WindRiver Tornado 创建相应的工程,建立一个 downloadable 工程。选择工具链 ARMARCH4gnu。

(3) 工程建立成功后,添加文件 yejing.c,即编写液晶显示程序。嵌入式处理器对显示存储器访问实质上都是通过 T6963C 控制器进行的,首先完成系统初始化的工作,在系统已经能够正常工作以后,具体的指令操作通常分两步进行:送地址指针、读/写显示数据(C/D=0);或送地址指针、送操作指令(C/D=1)。嵌入式处理器与 T6963C 控制器的每一次数据前都要进行 T6963C 控制器的状态判断。下面是参考程序。

```
#include "VxWorks.h"
#include "logLib.h"
#include "stdio.h"
#define d_addr (short *)0x7fd0000 /* 地址 注意 S3C4510B 内外地址的转换 */
#define i_addr (short *)0x7fd0004 /* 地址 注意 S3C4510B 内外地址的转换 */
#define STACK_SIZE 20000
int tidlcd;
void lcd_work(void);
STATUS progStart(void)
{
 tidlcd = taskSpawn("tLcd", 198, 0, STACK_SIZE,
 (FUNCPTR) lcd_work, 0, 0, 0, 0, 0, 0, 0, 0, 0);
 return (OK);
}
/* 显示字符 welcome you */
char dtab[] = {0x37, 0x45, 0x4c, 0x43, 0x4f, 0x4d, 0x45, 0x00, 0x39, 0x4f, 0x55};
/* read stat function */
char r_stat(void)
{
 volatile short temp;
 temp = * i_addr;
 temp = temp >> 8;
 return ((char)temp);
}
```

```

 }
 /* judge stat.0 and stat.1 ;station of instruction r/w and data r/w */
 void st01(void)
 { int i;
 volatile short temp;
 for(;;)
 { temp = r_stat();
 if((temp&0x1)&&(temp&0x2))
 break;
 }
 }

 /* judge stat.2 ;auto data read */
 void st2(void)
 { int i;
 short volatile temp;
 for(;;)
 { temp = r_stat();
 if(temp&0x4)break;
 }
 }

 /* judge stat.3 ;auto data write */
 void st3(void)
 { int i;
 volatile short temp;
 for(;;)
 { temp = r_stat();
 if(temp&0x8)break;
 }
 }

 /* judge stat.6 ;stat of screen read/copy */
 void st6(void)
 { int i;
 short temp;
 for(;;)
 { temp = r_stat();

```

```

 if(temp&0x0)break;
 }
}

/* data write function */
void out_data(char data1)
{ volatile short temp;
 temp = data1;
 temp = temp<<8;
 st01();
 * d_addr = temp;
}

/* command write function */
void out_cmd(char command)
{ volatile short temp;
 temp = command;
 temp = temp<<8;
 st01();
 * i_addr = temp;
}

/* data read function */
char in_data(void)
{ volatile short temp;
 temp = * d_addr;
 temp = temp>>8;
 st01();
 return ((char)temp);
}

/***** /
void lcd_clear(char code,char attr)
{ int i;
/***** write data *****/
 out_data(0);
 out_data(0);
 out_cmd(0x24);/* set first address of display sram */

```

```

 out_cmd(0xb0); /* set auto write */

 for(i = 0; i < 0x100; i++)
 {
 st3();
 out_data(code);
 }
 out_cmd(0xb2);
 /****** write attribute *****/
 out_data(0);
 out_data(8);
 out_cmd(0x24); /* set first address of display sram */

 out_cmd(0xb0); /* set auto write */

 for(i = 0; i < 0x100; i++)
 {
 st3();
 out_data(attr);
 }
 out_cmd(0xb2);
 }
 /* initialization function */
 char attr,o_y,o_x;
 void lcd_work(void)
 {
 char code,prog_count;
 int i,k,j;
 *(volatile int *)0x3ff3008 = 0x0fff0fff;
 out_data(0);
 out_data(0);
 out_cmd(0x40); /* set first address of text zone */

 out_data(0x20);
 out_data(0);
 out_cmd(0x41); /* set display width of text zone,byte counter of a line */
 }

```

```

out_data(0);
out_data(8);
out_cmd(0x42);/* set first address of graph zone or first addr of text zone */

out_data(0x20);
out_data(0);
out_cmd(0x43);/* set width of graph display zone or width of text zone */

out_cmd(0xa7);/* set cursor shape */

out_cmd(0x84);/* set display mode ;logic or */

out_cmd(0x9f);/* set display switch ;switch on text and graph display */

/ *****
***** 需要读者添加 *****
* 循环显示 0~F *
***** /

out_cmd(0x84); /* set text display model */
out_cmd(0x9f); /* switch on text,graph and cursor flash display */

out_data(0x04);
out_data(0x01);
out_cmd(0x21); /* set location of text cursor */
for(i = 0;i<11;i++)
{
 code = dtab[i];
 out_data(0x3 + i); /* 显示字符位置 */
 out_data(0x00); /* 显示字符位置 */
 out_cmd(0x24);
 out_data(code); /* 显示字符 */
 out_cmd(0xc4);
 out_data(0x3 + i); /* 显示字符属性位置 */
 out_data(8); /* 显示字符属性位置 */
 out_cmd(0x24);

```

```

 out_data(0); /* 显示字符属性位置 */
 out_cmd(0xc4);
 }
 for(i = 0; i < 11; i++)
 {
 code = dtab[i];
 out_data(0x43 + i); /* 显示字符位置 */
 out_data(0x00); /* 显示字符位置 */
 out_cmd(0x24);
 out_data(code); /* 显示字符 */
 out_cmd(0xc4);
 out_data(0x43 + i); /* 显示字符属性位置 */
 out_data(8); /* 显示字符属性位置 */
 out_cmd(0x24);
 out_data(5); /* 显示字符属性位置 */
 out_cmd(0xc4);
 }
}

/*****

```

(4) 在完成了基本函数的编写以后,完成主函数的编写,发起任务。

(5) 编译成功后,下载执行。在 Shell 下运行 progStart,液晶显示屏将出现 0-9-A-F 的显示,最后显示“welcome you”。

(6) 借助一些调试工具如 WindView,或者用在 Shell 中输入 shell 命令等方法监视任务执行的情况,分析任务的一些参数和特性。另外,分析目标机中任务占用资源的情况(可以启动 Browser 工具查看)。

(7) 实验注意事项。

① 要避免在处理机械振动和对模块施加外力,都可能使屏不显示或损坏。

② 不能用手或坚硬工具或物体接触、按压、摩擦显示屏,否则屏上的偏振片被物体划坏。

③ 如果屏破裂液晶材料外漏,液晶可以通过空气被吸入,而且要避免液晶与皮肤接触,如果接触应立即用酒精冲洗,然后再用水彻底冲洗。

④ 要防止高压静电产生的放电,将损坏模块中的 CMOS 电路。避免水蒸气凝结,否则将导致屏或电极开路。

## 7. 思考题

(1) 试考虑汉字如何显示。

(2) 试考虑怎样在液晶上显示图形?

## 7.6 实验6 液晶键盘实验

### 1. 实验目的

(1) 复习掌握 ARM 嵌入式微处理器的通用 I/O 口工作原理。利用北京邮电大学电信工程学院 IRDC 实验室开发的 IRDC05v3 实验板上的通用 I/O 口学习 S3C4510B 的扩展 I/O 方式以及学习它的编址方式和总线宽度控制。

(2) 复习并掌握基于学习键盘工作的原理和相关寄存器编程的方法。复习 S3C4510B 的中断接口的使用方法。复习在 VxWorks 下编写一个中断程序的方法。

(3) 复习 T6963C 液晶控制器的原理和使用,复习键盘的驱动和软件去抖,复习 S3C4510B 的键盘和液晶的接口,复习信号量和使用信号量进行键盘任务和液晶任务的交互。

### 2. 实验内容

(1) 利用实验板上 S3C4510B 提供的 18 个可编程的通用 I/O 端口(用户可将每个端口配置为输入模式、输出模式)学习通用 I/O 口工作原理及其寄存器状态字的定义内容,进一步了解可编程的通用 I/O 端口控制器的工作过程。

(2) 参考前面两个液晶和键盘实验,本实验要编码液晶和键盘两个程序,并使用信号量实现两个程序的交互。要求在键盘上输入相应的数字或字符,在液晶显示器上有对应的数字或字符输出。

### 3. 预备知识

(1) 熟练使用 VxWorks 操作系统的 WindRiver Tornado 集成开发环境,熟悉编写和下载调试程序的基本过程。

(2) 熟悉 VxWorks 应用程序的框架结构及其开发流程以及 VxWorks 任务的发起。

(3) 进一步熟悉 T6963C 液晶控制器的原理和接口寄存器工作原理及其复用方法。

(4) 熟悉 ARM 的通用 I/O 口相关寄存器的功能,熟悉 ARM 系统硬件的通用 I/O 口工作原理相关接口的复用方法。

### 4. 实验设备

#### (1) 硬件环境

- 北京邮电大学电信工程学院 IRDC 实验室开发的 CS4510 教学实验/评估板;
- PC;
- 10/100 M 以太网连接(可选双网口) + 直连串口连接;
- 用于 ARM7TDMI 的 JTAG 仿真器(可选)。

#### (2) 软件开发环境

- WindRiver Tornado 2.0 above Integrated Development Environment for ARM;

- Win98、Win2000 或 WinXP 操作系统；
- ARM SDT 2.51 集成开发环境、仿真器驱动程序(汇编方式调试可选)；
- 超级终端通信程序(C 语言/ARM 汇编语言)。

## 5. 实验原理

### (1) 键盘驱动

参见前面键盘实验, 4×4 键盘工作于中断方式, 当有键按下时, 响应中断, 要求执行按键去抖, 判断键值, 送液晶显示。

## (2) 液晶驱动

参见前面液晶实验, VxWorks 发起液晶显示程序, 等待键盘的消息, 如有按键值, 则启动液晶显示。

### (3) 信号域

负责键盘中断程序和键盘键值判断程序之间,键盘驱动程序和液晶驱动程序之间传递消息。也可以使用其他方式,如消息队列。

## 6. 实验步骤

(1) 启动 Tornado, 新建一个 downloadable 工程, 选择工具链 ARMARCH4gnu。

(2) 工程建立成功后,编写键盘驱动和液晶驱动,并用信号量实现任务间的交互,在 VxWorks 主程序中,创建信号量并且发起这两个任务。

### ① 程序框架

STATUS progStart (void)

{

```
flag = 0; /* 程序执行标志 */
```

```
semLcd = semBCreate(SEM_Q_FIFO, SEM_FULL); /* 信号量 负责键盘程序和液晶
程序进行交互 */
```

```
semInt = semBCreate(SEM_Q_FIFO, SEM_FULL); /* 信号量 负责中断程序和键盘
扫描判断程序进行交互 */
```

```
semTake(semInt, WAIT FOREVER);
```

```
semTake(semLcd, WAIT FOREVER);
```

```
tidled = taskSpawn (~tLcd~, 198, 0, STACK SIZE,
```

```
(FUNCPTR) lcd work,0,0,0,0,0,0,0,0,0);/* 发起液晶程序 lcd work */
```

```
tid interrupt = taskSpawn ("tInterrupt", 199, 0, STACK_SIZE,
```

```
(FUNCPTR) key check,0,0,0,0,0,0,0,0,0,0);/* 发起键盘程序 key check */
```

```
return (OK);
```

}

## ② 液晶程序

先进行初始化,设置显示模式、文本宽度、光标属性,然后等待新的键值,执行显示。



```

void lcd_work(void)
{
 lcd_init(); /* 液晶初始化 */
 for(;;)
 {
 if(flag == 1) break;
 semTake(semLcd, WAIT_FOREVER); /* 等待新的键值 */
 lcd_show(); /* 液晶显示 */
 }
}

```

### ③ 键盘程序

等待键盘中断,软件去抖,然后判断键值,传送给液晶显示。

```

void key_check(void)
{
 key_init(); /* 键盘初始化 */
 Key_interrupt(); /* 初始化键盘中断程序 */
 for(;;)
 {
 if(flag == 1) break;
 semTake(semInt, WAIT_FOREVER); /* 等待键盘中断 */

 / *****
 ***** 需要读者添加 *****
 * 软件去抖 *
 ***** /

 key_value(); /* 键值判断程序 */
 semGive(semLcd); /* 释放信号量 */
 }
}

```

函数终止,设置标志位 flag 为 1,终止 VxWorks 发起的键盘任务和液晶任务。

```

void progStop(void)
{
 flag = 1;
}

```

(3) 调试编译通过后下载进入目标机,在 Shell 中运行 progStart,要求实现在键盘上输入相应键值,在液晶上有相应的键值输出。

(4) 借助一些调试工具如 WindView 或者在 Shell 中输入 shell 命令等方法监视任务执行的情况,分析任务的一些参数和特性。另外,分析目标机中任务占用资源的情况(可以启动 Browser 工具查看)。

#### 7. 思考题

- (1) 试用其他方法实现程序的交互。
- (2) 试在液晶上实现汉字及图形显示。

## 第 8 章 高级扩展性实验

本章的实验可以作为选做。关于 BSP 的移植,主要是将我们移植过程中的一些体会写出来供大家参考。其他几个实验的综合性和应用性都比较强。

### 8.1 实验 1 VxWorks BSP 移植

#### 1. VxWorks BSP 概述

VxWorks 板级支持包(BSP)是为 VxWorks 操作系统与硬件平台之间提供一个接口的软件包,它可以使 VxWorks 运行于特定的硬件平台,如 ARM、PPC、X86 等,它包含了一系列与硬件相关的函数,完成对于硬件的基本输入输出操作,可以对上层程序员实现底层的硬件透明。例如,一般它完成以下操作:

- 目标板硬件初始化
- 内存控制器初始化
- 堆栈初始化
- 外围设备初始化(I/O、Interrupt……)
- 异常向量处理
- CACHE 操作
- 硬件设备的底层驱动
- 定时器驱动
- 串口驱动
- END 驱动
- FLASH 驱动
- LCD 驱动

一般情况下,板级支持包(BSP)由芯片提供商随着芯片的开发板一同提供,但不是所有的 CPU 都提供 VxWorks 的 BSP,因此我们常常需要对 BSP 进行移植,即使芯片提供商发布了 BSP,在实际使用中我们也需要对该 BSP 进行修改和添加它的功能。下面通过 target/config/BSPname 目录下的文件和启动顺序两个方面来分析一个 BSP 的组成。

### (1) 文件组成

① config.h:是某一个具体 BSP 的配置文件,可以通过它重新定义 configAll.h 的参数和对 VxWorks 进行裁减,其中关于启动行和各内存地址的定义尤为重要。它包含 BSPname.h 文件,在移植一个 BSP 时,需要对这两个文件重新定义。

② BSPname.h:这里定义了涉及硬件的基本参数、内存地址接口、I/O 地址接口、时钟、中断等。

③ romInit.s:包含用于初始化的汇编语言源代码,其中 romInit()是开机执行的第一段程序。主要是由初始化堆栈指针和其他寄存器来执行 C 语言程序,并跳转到 romStart()。

④ SysALib.s:特定目标板的源程序,在 BSP 中需要用汇编来实现的程序可以存放在这个文件中。

⑤ sysLib.c:BSP 的 C 语言程序,包含驱动器的设置和初始化,它包含 sysSerial.c、sysNet.c、sysScsi.h、sngks32cEnd.c 等一系列外设初始化函数。

⑥ target.nr:包含了 BSP 所运行的目标板的信息。

⑦ ne2000.c:ne2000 网卡驱动程序,同样也可以包含其他外设的驱动文件,如 LCD.c 或 key.c 等。

⑧ Makefile:是 bootrom 映像的编译规则文件,以命令行方式控制映像 image 的生成,它定义了 CPU 的结构、主机工具链和目标板目录名等,其中几个与内存地址相关的宏定义一定要与 config.h 保持一致。

⑨ 衍生文件:从原文件生成的一些 .o 文件如:romInit.o、sysLib.c 等和 bootrom 启动文件 bootrom\_uncmp.bin、bootrom 等。

### (2) 启动顺序

明白了启动顺序能更好地解读 BSP。如图 8-1 所示,bootRom image 和 VxWorks image 启动顺序的不同在于,bootRom 完成最小的系统初始化,主要用于装载 VxWorks image,它调用 bootConfig.c 中的 usrInit();而 VxWorks image 执行完备的初始化,并加载用户程序,它调用 usrConfig.c 中的 usrInit()。

① romInit()是系统入口函数,完成少部分的初始化,如禁止中断、初始化堆栈以及少量 CPU 寄存器设置,为调用 C 函数做准备,之后则进入第一个 C 程序 romStart()。

② romStart()是第一个 C 程序。一般情况下,不做太多的初始化工作,只完成清内存和从 rom 到 ram 的拷贝。

③ sysInit()是 VxWorks 的入口函数,保证成功调入 image,它重复实现 romInit()的部分功能,如禁止中断和初始化栈等,然后跳转到 usrInit()。

④ `usrInit()` 是保存引导类型信息, 配置硬件, 然后启动操作系统内核, 把控制权交给操作系统。至此, BSP 的主要任务完成。

⑤ `usrRoot()` 是操作系统的第一个任务, 初始化 I/O 系统, 创建 Devices, 并初始化网络系统。对于 bootRom image 来说, 需要通过 ftp 下载 VxWorks image; 对于 VxWorks image, 则会调用用户应用程序。

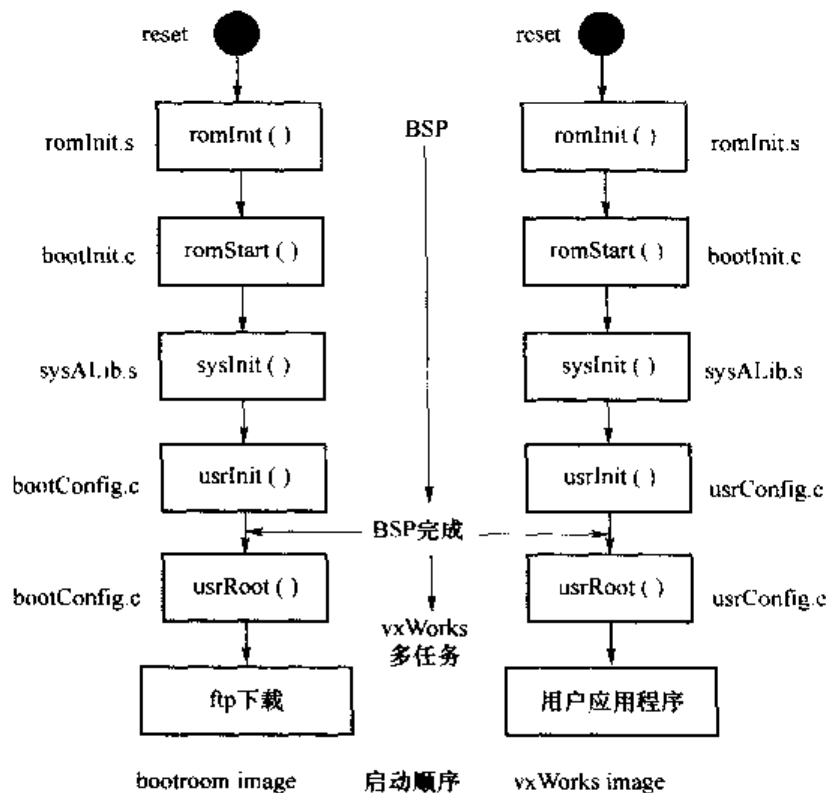


图 8-1 VxWorks 启动顺序

实际中绝大部分情况并不需要我们去写一个 BSP, 只需对原有 BSP 进行修改, 这样我们对于 BSP 的理解就可以简单得多。下面根据一个实例来分析 BSP 的移植和修改。

## 2. 基于 S3C4510B 的 BSP 的移植实例

移植 BSP, 最好有一个可参考的模板, 在此基础上修改 BSP 相关的文件。我们参考的模板为 WINDRIVER FOR 4510 BSP wrSbcArm7, 在 tornado2.2 中有该模板 (在 tornado2.0 下可以参考三星的 snds100)。我们参考的 S3C4510B 开发板主要参数为: 32 M bytes sdram(32 bits)、4 M bytes flash(32 bits)、1 M bytes flash(16 bits)、双网口、液晶接口等。移植一个 BSP, 首先是修改配置文件, 生成 bootrom。

### (1) 配置文件

配置文件 makefile、config.h、BSPname.h (这里为 wrSbcArm7.h), 这里 config.h、wrSbcArm7.h 包含了硬件和软件模块的基本配置信息 (内存、I/O、驱动设备)。接下来以

上面提到的 S3C4510B 的开发板为例讨论一下配置文件的修改。

### ① 定义启动行

```
#define DEFAULT_BOOT_LINE
```

“sng(0,0)host;VxWorks” \sng 为引导设备,内置的以太网控制器;(0,0)在软盘 fd、硬盘 ata 和 tffs 启动时有意义;host 为连接主机;VxWorks 为下载映像名,绝对路径在主机 ftp server 中指定。

```
“h=192.168.0.1” \\host 机 ip 地址
```

```
“e=192.168.0.10” \\目标机 ip 地址
```

```
“g=192.168.0.1” \\网关
```

```
“u=target” \\用户名
```

```
“pw=target” \\密码
```

```
“tn=targetname” \\目标板名
```

### ② 内存配置

将内存配置成与自己的开发板一致。

a. RAM 设置如图 8-2 所示(image 非压缩)

```
#define local_mem_local_adrs 0x0 \\表示板上内存的起始地址
```

```
#define local_mem_size 0x2000000 \\表示静态内存大小(32 M)
```

```
#undef local_mem_autosize \\表示运行时动态内存大小 undef 不指定
```

```
#define user_reserved_mem 0x1000 \\表示保留给用户的内存大小,存储出错信息
```

```
#define ram_high_adrs 0x48000 \\对于非压缩的 image,bootrom 拷贝自身
 代码到这个位置,然后跳到 ram 中执行。
```

```
#define ram_low_adrs 0x6000 \\加载 VxWorks image 入口地址
```

b. ROM 配置

```
#define ROM_BASE_SIZE 0x02000000 \\表示 rom/flash 起始地址低 32 M 为 sram
```

```
#define ROM_TEXT_ADRS ROM_BASE_SIZE \\表示 boot rom 的入口地址
```

```
#define ROM_SIZE \\表示 0x100000 rom/flash 分配给 VxWorks image 的大小
```

```
#define ROM_COPY_SIZE \\ROM_SIZE 启动时拷贝到 RAM 的大小即为 image 大小
```

```
#define ROM_SIZE_TOTAL \\0x00100000 rom/flash 的大小 1 M
```

注意:makefile 文件与 config.h 内存相关宏定义保持一致。

### ③ 4510 寄存器设置

这部分设置在 sbccarm7.h 中,定义寄存器、中断、时钟、I/O 参数等。这里应该注意的地方是 S3C4510B 在 map 前后内存地址的映射分布,如图 8-3 所示,我们可以参照预先确定的 ROMCON0、ROMCON1 和 DRAMCON0 在 map 前后内存分布的变化,对 sbccarm7.h 中的原定义进行修改(详情参见 S3C4510B.pdf 芯片手册)。这里需要注意以下几个设置。

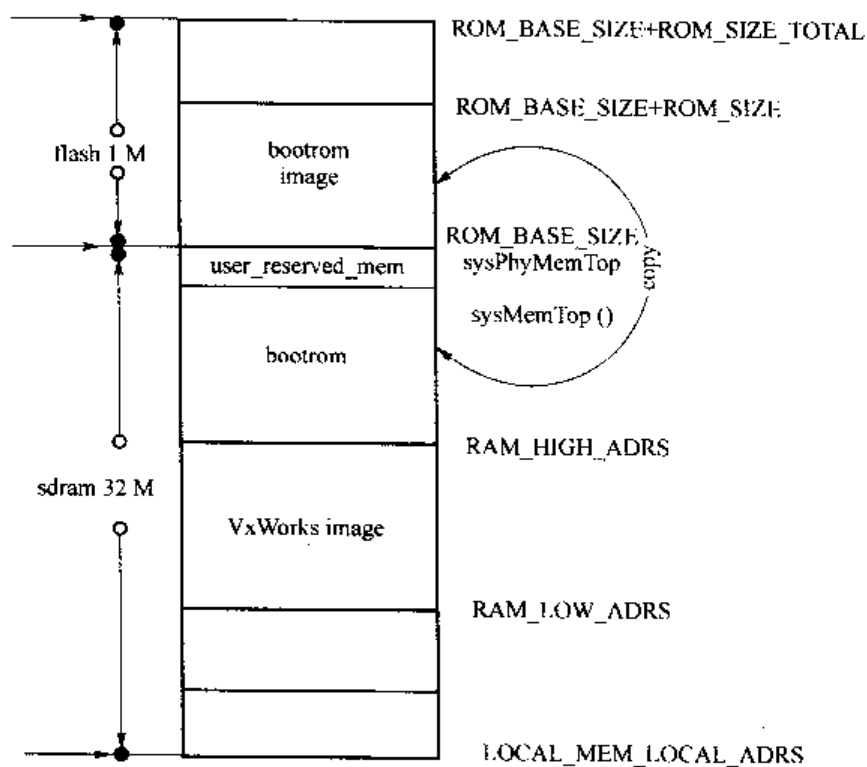


图 8-2 VxWorks 内存映射

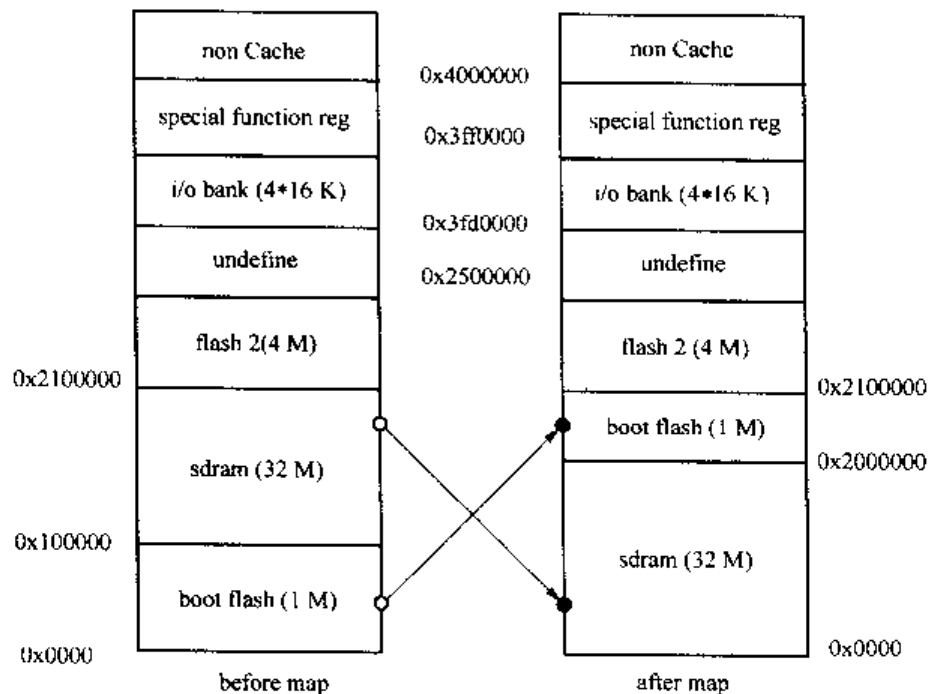


图 8-3 VxWorks 内存映射 2

a. 确定 non Cache 区域。

```
#define NON_CACHE_REGION 0x4000000
```

b. 确定 flash sdram I/O 各个外设的总线宽度, 这些总线宽度定义于 EXTDBWTH 寄存器中。

```
#define DSR0 (2<<<0) /* ROM0 2;16 位 boot flash 1 M */
#define DSR1 (3<<<2) /* ROM1 3;32 位 flash2 4 M */
#define DSR2 (1<<<4) /* ROM2 */
#define DSR3 (0<<<6) /* ROM3 */
#define DSR4 (0<<<8) /* ROM4 */
#define DSR5 (0<<<10) /* ROM5 */
#define DSD0 (3<<<12) /* DRAM0 32 位 主内存 32 M */
#define DSD1 (0<<<14) /* DRAM1 */
#define DSD2 (0<<<16) /* DRAM2 */
#define DSD3 (0<<<18) /* DRAM3 */
#define DSX0 (2<<<20) /* EXTIO0 LCD 接口 16 位 */
#define DSX1 (1<<<22) /* EXTIO1 USB 接口 8 位 */
#define DSX2 (2<<<24) /* EXTIO2 以太网接口 16 位 */
#define DSX3 (1<<<26) /* EXTIO3 */
#define rEXTDBWTH
(DSR0 + DSR1 + DSR2 + DSR3 + DSR4 + DSR5 + DSD0 + DSD1 + DSD2 + DSD3 + DSX0 + DSX1
+ DSX2 + DSX3)
```

其中 DSR0 由外部硬件跳线决定。

c. 确定 External I/O 的基地址, 定义于寄存器 REFEXTCON 中, 该地址为 0x3fd。

d. 分别确定 ROMCON0、ROMCON1 和 DRAMCON0 在 map 前后的取值。

一般情况下, 完成上面几步即可成功生成 VxWorks 的 boot image, 实现从网口引导 VxWorks 映像。但如果要实现进一步的功能, 如添加第二块以太网口、增加 LCD 控制、实现 tffs 文件系统等等都需要进一步增加 BSP 的功能。

下面以以太网为例说明 BSP 功能增强的步骤。

(2) 添加第二块网卡— RTL8019as 以太网控制器

RTL8019as 与 NE2000 网卡兼容, T2.2 中有它的驱动。我们只需把它移植到 BSP 中即可, 下面分析一下移植步骤。

① 修改 BSP 的 config.h 文件, 应该有下列定义。

```
#ifndef INCLUDE_ENE
#define INCLUDE_ENE //加载 ne2000 设备
#define IO_ADRS_ENE //ADRS_8019_BASE + NON_CACHE_REGION 设置以太网控制
```



器基地址

//并定义在 non cache 区域,否则 8019as 控制器将不能工作。

```
#define INT_LVL_ENE //INT_LVL_EXTINT1 设置以太网控制器的中断及向量
#define INT_VEC_ENE //INT_VEC_EXTINT1
#endif
```

② 修改 BSP 的 configNet.h 的 END\_TBL\_ENTRY endDevTbl 结构体。

```
END_TBL_ENTRY endDevTbl [] =
{
#ifdef INCLUDE_SNGKS32C_END
{ 0, SBCARM7_LOAD_FUNC_0, SBCARM7_LOAD_STRING_0, SBCARM7_LOAN, NULL,
FALSE},
#endif /* INCLUDE_SNGKS32C_END */
#ifdef INCLUDE_SNGKS32C_HDLC_END
{ 1, SBCARM7_LOAD_FUNC_1, SBCARM7_LOAD_STRING_1, SBCARM7_LOAN, NULL, FALSE},
{ 2, SBCARM7_LOAD_FUNC_2, SBCARM7_LOAD_STRING_2, SBCARM7_LOAN, NULL, FALSE},
#endif /* INCLUDE_SNGKS32C_HDLC_END 0x18 --> 0x1 */
#ifdef INCLUDE_ENE
{ 0, ne2000EndLoad, "0x7fdc200:0x1:0x1:2:0:2", 1, NULL, FALSE},
#endif
{ 0, END_TBL_END, NULL, 0, NULL, FALSE},
};
```

其中 #ifdef INCLUDE\_ENE {...} #endif 为新添加的网络设备;第一个参数 0 代表同类型的设备序号;ne2000EndLoad 为 ne2000 驱动的入口函数;0x1:0x1 为中断号和中断向量;第一个 2 代表总线宽度为 16 位;0 代表 mac 地址的获取;最后一个 2 代表内存对齐方式为字对齐。

③ 重写对于 rtl8019as 操作的基本输入输出函数同时允许中断输入。

```
sysInWordString (ULONG port, char * pData, int count)
sysOutWordString (ULONG port, char * pData, int count)
sysOutByte(ULONG port, UCHAR data)
sysInByte (ULONG port)
```

由于在 arm 体系结构中内存和 I/O 统一编址,二者的访问方式一样,上述几个函数可与内存操作类似。

```
sysIntEnablePIC(int intLevel)
sysIntDisablePIC
```

参照 S3C4510B 手册确定相应中断的允许和禁止。

④ 在 Tornado 中创建 VxWorks 的工程, 设置 IP\_MAX\_UNITS, 使其大于等于 2。VxWorks 启动完毕后, 可以在 usrAppInit 中, 以如下的方式来启动和配置每个网卡。

```
ipAttach(0, "ene"); ene 为 ne2000 设备标号
```

```
ifMaskSet("ene", 0xffffffff00);
```

```
ifAddrSet("ene", "192.168.0.100");
```

重新编译工程即可生成带有 rtl8019 驱动的 VxWorks 映像。

### (3) 观察移植效果

将程序下载运行, 看是否能够 PING 通目标机, 如果可以, 表示移植成功; 否则再检查移植步骤。

### 3. 问题思考

(1) 总结一下 BSP 移植的关键步骤, 并回顾一下 VxWorks 中 BSP 的特点。

(2) 操作系统的移植应该注意什么?

## 8.2 实验 2 理发师问题实验

### 1. 问题描述

有一个理发师, 一个工作室和一个休息室。理发师在工作室一次只能为一名顾客理发, 休息室总共可以容纳 5 名顾客等待。顾客的光临是随机的, 当新到的顾客发现休息室已满的时候就离开。如果休息室没满, 则进入休息室等待。理发完毕的顾客马上离开, 然后下一位顾客接着理发。

### 2. 实验要求

要求给出问题分析、算法说明和源代码, 并且在 Tornado 环境中调试成功。

### 3. 设计方案

对于这个实验有很多种方法, 下面例举两个方案进行说明。

#### (1) 方案 1

##### ① 问题分析

这是一个需要使用到消息队列的问题。在主程序 go 中我们创建了两个任务, 一个是理发师的任务, 名为 Barber; 另一个是顾客的任务, 名为 Customer。创建了一个消息队列 msgCustomer。

Barber 的任务是从消息队列接收消息, 即从排队的顾客中挑选最前面的顾客进行服务。如果消息队列为空, 表示没有顾客在等待。理发师服务完一位顾客后就马上为第一位等待的顾客服务, 并将相应的信息打印输出, 同时假设一位顾客服务的时间为 200 ticks。

在 Customer 任务中, 顾客光临的时间间隔是随机的, 每来一位新顾客就使 Custom-

crID 加 1,同时打印输出,把进入排队的顾客的 ID 号作为一个消息发送到消息队列。同时计算出消息队列里面的消息数目,即等待的顾客数目,将其打印输出。

### ② 算法实现

程序创建了一个任务堆栈,并将其作为一个环形链表使用。顾客到来时将任务插入到环形链表当中,如果没有空余位置,则顾客离开。理发师每次从链表中取出一个进行服务。理发师的服务时间是固定的,顾客到来的时间是随机的。理发师总共服务 20 位顾客,算法结束。

### ③ 代码分析

```
#include <vxWorks.h>
#include <sysLib.h>
#include <stdioLib.h>
#include <msgQLib.h>
#define MAX_MESSAGES 5
#define MAX_MESSAGE_LENGTH 20
MSG_Q_ID msgCustomer;

/* 定义一个消息结构体,用于向消息队列发送和从消息队列接收顾客的 ID 号 */
struct msg
{
 int CustomerID;
};
int currentCustomerID = 0; /* 表示当前正在理发的顾客的 ID */
int CustomerID = 0; /* 表示光临的顾客的 ID */

/* 理发师的任务 */
void Barber()
{
 struct msg CustomerIDbuf;
 FOREVER
 {
 printf("*****\n");
 /* 从消息队列接收消息,即从排队的顾客中挑选最前面的顾客进行服务;如果消息队列为空,表示没有顾客在等待;服务完一位顾客后就马上为第一位等待的顾客服务,并将相应的信息打印输出;假设一位顾客服务的时间为 200 ticks */
```

```

/*****
***** 需要读者添加 *****
***** /

 printf(" ***** \n");
 taskDelay(200);
}
}

/* 顾客的任务 */
void Customer()
{
 struct msg msgCustomerID;
 FOREVER
 {
 /* 顾客光临的时间间隔是随机的 */
 /* 每来一位新顾客就使 CustomerID 加 1,同时打印输出 */
 /* 把进入排队的顾客的 ID 号作为一个消息发送到消息队列 */
 /* 计算出消息队列里面的消息数目,即等待的顾客数目 */
 taskDelay((rand()) % 500);
 printf(" ***** \n");
 printf("Customer %d is coming\n", ++CustomerID);
 msgCustomerID.CustomerID = CustomerID;
 if (msgQSend(msgCustomer, (char *) &msgCustomerID, sizeof(
struct msg), NO_WAIT, 0) == ERROR)
 printf("a new Customer came but the waiting room was full,so he left\n");
 printf("%dCustomers are waiting\n", msgQNumMsgs(msgCustomer));
 printf(" ***** \n");
 }
}

/* 主程序,创建了一个消息队列,同时创建了 Barber 和 Customer 两个任务 */
int go()
{
 msgCustomer = msgQCreate(MAX_MESSAGES, MAX_MESSAGE_LENGTH, MSG_Q_FIFO);
 taskSpawn("Barber", 105, 0, 5000, Barber, 0, 0, 0, 0, 0, 0, 0, 0);

```

```
taskSpawn("Customer",105,0,5000,Customer,0,0,0,0,0,0,0,0,0,0);
}
```

#### ④ 运行程序

程序运行在 Tornado Simulator 仿真器中。仿真环境如图 8-4 所示。

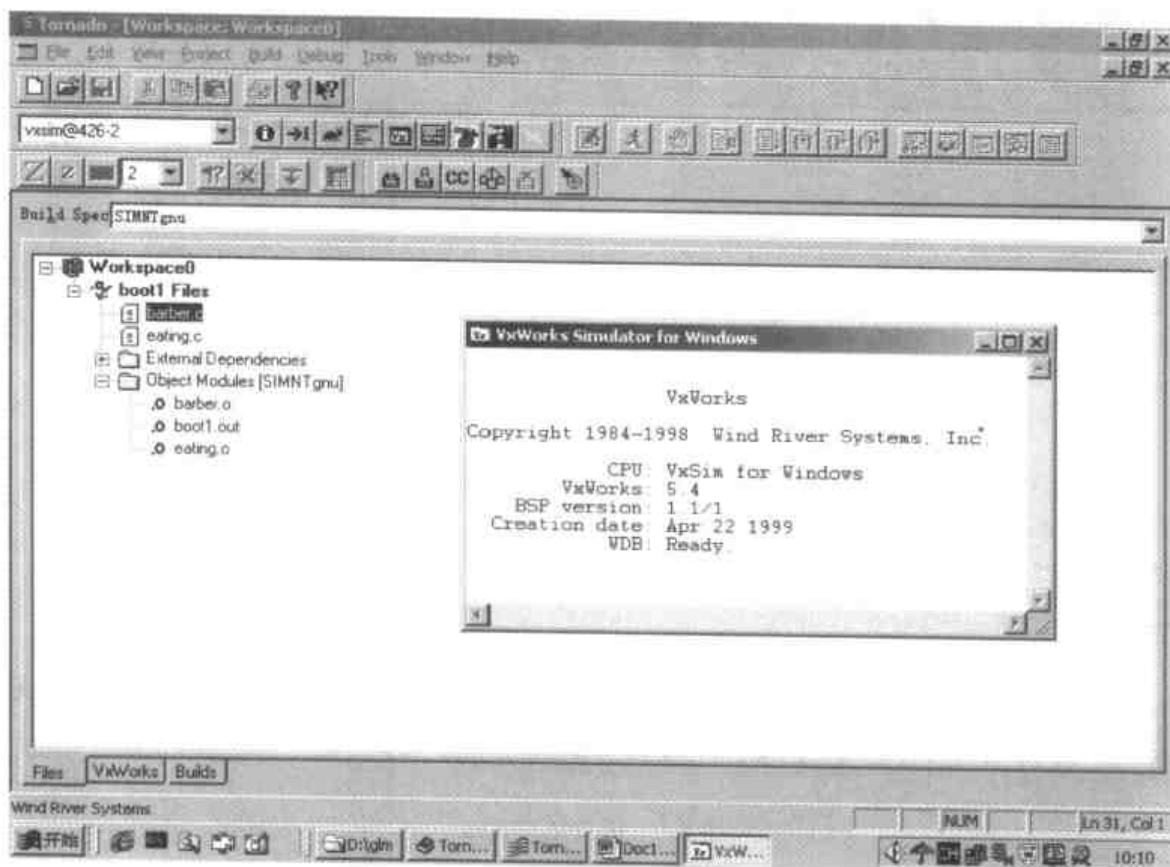


图 8-4 启动 VxSim

运行结果如图 8-5 所示。

#### (2) 方案 2

##### ① 问题分析

在本题中,顾客进程和理发师进程之间存在着多种同步关系。

a. 只有在理发椅空闲时,顾客才能坐到理发椅上等待理发师理发,否则顾客便必须等待;只有当理发椅上有顾客时,理发师才开始理发,否则他也必须等待。这种同步关系类似于单缓冲(对应于理发椅)的生产者-消费者问题中的同步关系,故可通过信号量 empty 和 full 来控制。

b. 理发师为顾客理发时,顾客必须等待理发的完成,并在理发完成后由理发师唤醒他,这可单独使用一个信号量 cut 来控制。

c. 顾客理完发后必须向理发师付费,并等理发师收费后顾客才能离开;而理发师则

需等待顾客付费,并在收费后唤醒顾客以允许他离开,这可以分别通过两个信号量 payment 和 receipt 来控制。

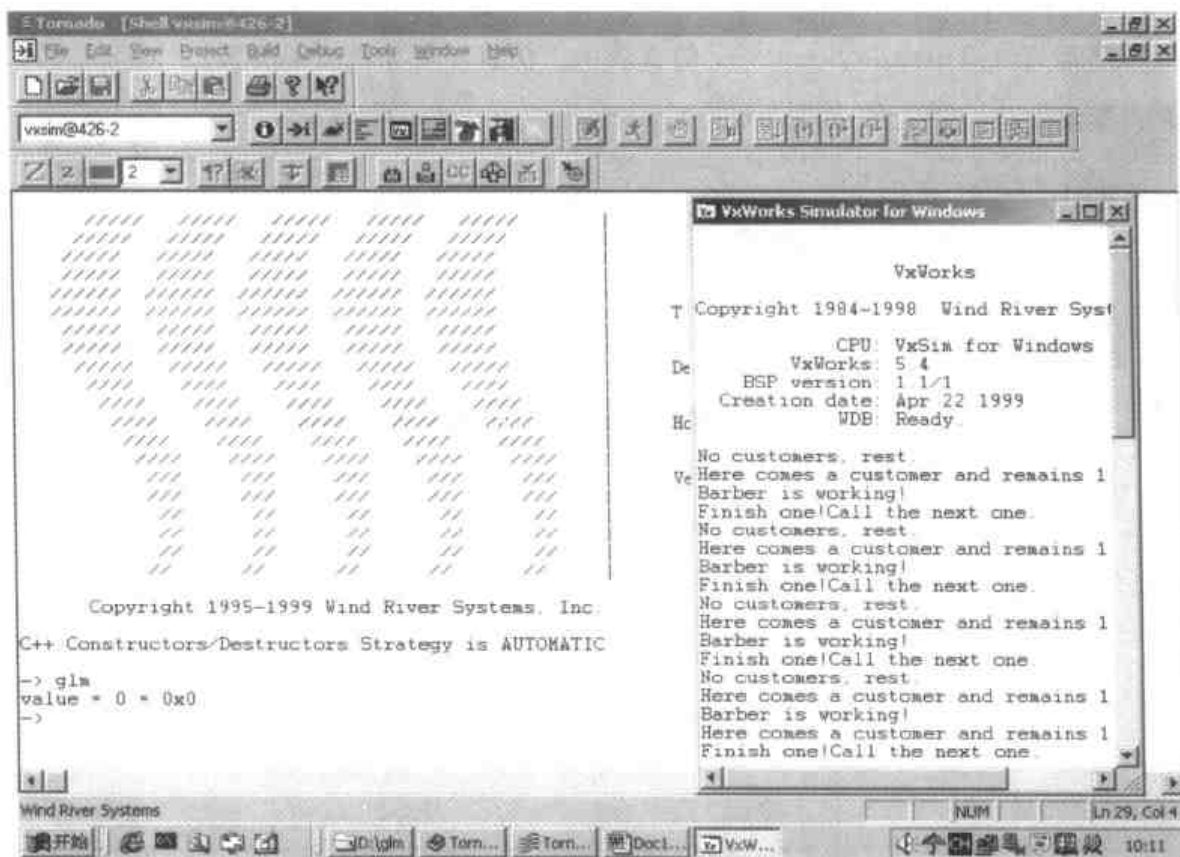


图 8-5 运行结果显示

d. 等候室中的 5 张沙发是顾客进程竞争的资源,故还需为它们设置一个资源信号量 sofa。

e. 为了控制顾客的人数,使顾客能在所有的沙发都被占用时离开理发店,还必须设置一个整型变量 count 来对理发店中的顾客进行计数,该变量可被多个顾客进程互斥地访问并修改,这可通过一个互斥信号量 mutex 来实现。

## ② 算法实现

```
Var count; integer; = 0
```

```
Mutex, sofa, empty, full; semaphore; = 1, N = 5, 1, 0;
```

```
Cut, payment, receipt; semaphore; = 0, 0, 0;
```

```
begin
```

```
parbegin
```

```
guest; begin
```

```
 wait(mutex);
```

```

if(count>N) then
begin
 Signal(mutex);
 exit shop;
end
else
begin
 count; = count + 1;
 if(count>1) then
begin
 wait(sofa);
 sit on sofa;
 wait(empty);
 get up from sofa;
 signal(sofa);
end
else
 wait(empty);
 sit on the barber chair;
 signal (full);
 wait(cut);
 pay;
 signal(payment);
 wait(receipt);
 get up from the barber_chair;
 signal(empty);
 wait(mutex);
 count; = count - 1;
 signal(mutex);
 exit shop;
end
end
Barber; begin
 repeat
 wait(full);

```

```

 cut hair;
 signal(cut);
 wait(payment);
 accept payment
 signal(receipt);
 until false;
end
parend
end

```

### ③ 源代码实现

```

#include <semLib.h>
#include <msgQLib.h>
#include <taskLib.h>
#include <stdlib.h>
#include <stdio.h>
#include <kernelLib.h>

#define NUM_CUST 5 /* the number of the seats */
#define RANDMAX 100 /* the max interval of customer's arrival */
#define BARBERTIME 25 /* barber time */
#define TASK_PRIORITY 101 /* 任务的优先级 */

//顾客结构
typedef struct CCustomer
{
 int cusid;
}Customers;

MSG_Q_ID msgQId;
SEM_ID semId; /* sleeping,barber take it! */
void barber();
void customer();
int taskid2;

```

////////////////////////////////////



```

//程序入口
//
////////////////////////////////////
STATUS demo()
{
 int taskid1;

 //设置任务轮询调度时间片为 200 个 CPU 周期
 kernelTimeSlice(200);

 //创建二进制信号量
 semId = semBCreate(SEM_Q_FIFO,SEM_FULL);

 //创建消息队列
 msgQId = msgQCreate(NUM_CUST,sizeof (Customers),MSG_Q_FIFO);

 //启动一个顾客进程
 taskid1 = taskSpawn("taskcustomer",TASK_PRIORITY,0x100,2000,(FUNCPTR)
customer,0,0,0,0,0,0,0,0,0,0);

 return(OK);
}

////////////////////////////////////
//理发师进程
//
////////////////////////////////////
void barber()
{
 Customers cust;

 //如果有客户等待则连续工作
 while(msgQReceive(msgQId,(char *)&cust,sizeof(Customers),NO_WAIT)!= ERROR)
 {
 //理发过程
 }
}

```

```

 printf("Barber is hairdressing for customer %d...\n",cust.cusid);
 taskDelay(BARBERTIME);
 printf("Finished customer %d! Send him out.Call another.\n",cust.cusid);
 }

 //没有客户等待,睡觉
 printf("No customer! Sleep! \n");
 semGive(semId);
 taskSuspend(taskid2);
}

////////////////////////////////////
//顾客进程
//
////////////////////////////////////
void customer()
{
 Customers custom;
 int i=0,j;
 taskid2 = taskSpawn("taskbarber",TASK_PRIORITY,0x100,2000,(FUNCPTR)
barber,0,0,0,0,0,0,0,0,0,0);
 FOREVER
 {
 custom.cusid= ++ i;

 //顾客随机到达
 taskDelay(rand() % RANDMAX);
 /* i = (i+1) % numCust; */

 / *****
 ***** 需要读者添加 *****
 * 查看理发店是否人满(椅子满)
 * 人满离开,打印信息
 ***** /

```

```

//有空位,等待理发
printf("Here comes a customer and remains %d.\n",msgQNumMsgs(msgQId));

if(semTake(semId,NO_WAIT) == OK)
 /* For the first customer,wake up the barber */
 taskRestart(taskid2);
}
}

```

#### ④ 结果分析

服务时间=100,顾客之间最大间隔=100,如图 8-6 所示。

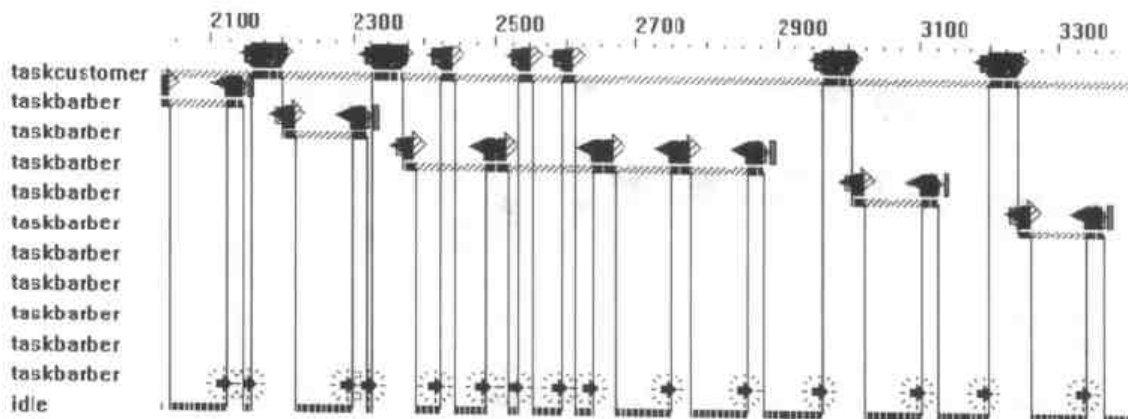


图 8-6 WindView 结果分析-1

服务时间=50,顾客之间最大间隔=100,如图 8-7 所示。

服务时间=10,顾客之间最大间隔=100,如图 8-8 所示。

从实验仿真图可以看出:

- 理发师进程(taskbarber)把消息队列读空后,会自动挂起;
- 新的顾客进程(taskcustomer)运行后,原先挂起的理发师进程(taskbarber)会被销毁,重新创建一个新的理发师进程(taskbarber);
- 一个理发师进程(taskbarber)从被创建到被挂起的时间与此时间内顾客进程(taskcustomer)运行次数成正比;
- 服务时间与顾客之间最大间隔的关系为,服务时间越长,此时间内到达的平均顾客数就越多,反之亦然。

其中有两种极限情况:一种情况是服务时间比顾客之间最大间隔大很多,则消息队列

几乎一直处于满的状态,理发师进程(taskbarber)不会被挂起;另一种情况是服务时间比顾客之间最大间隔小很多,则消息队列最多只有一个消息,而大部分时间为空,理发师进程(taskbarber)每执行一次就挂起一次,等顾客进程(taskcustomer)的唤醒。

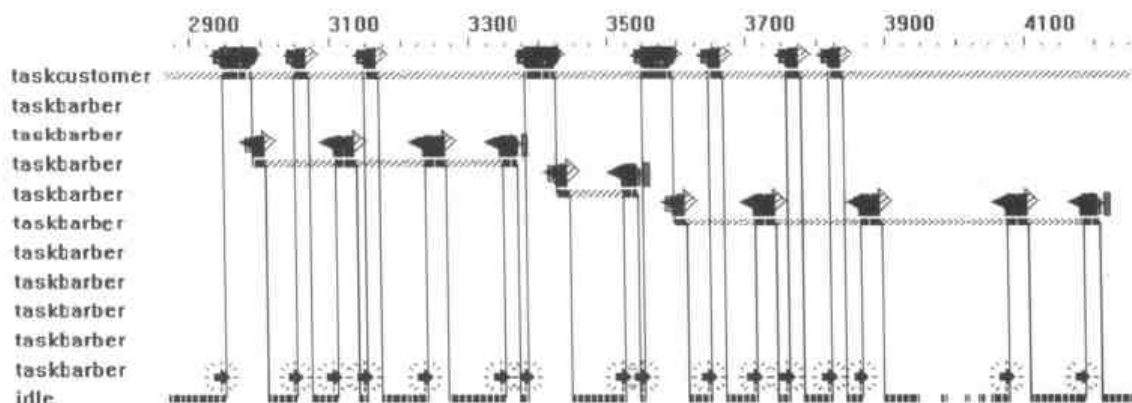


图 8-7 WindView 结果分析-2

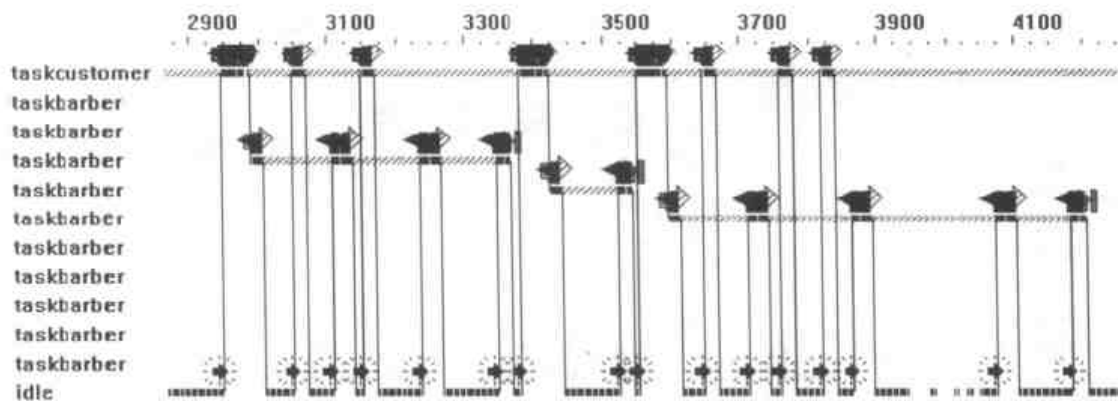


图 8-8 WindView 结果分析-3

#### 4. 问题思考

上述哪种方案的算法最好,进行对比分析,得出结论。

### 8.3 实验 3 哲学家吃饭问题实验

#### 1. 问题描述

5 个哲学家围坐在一个圆桌前,每个哲学家的生活就是思考、饥饿,然后进食。每个人的左右都只有一根筷子,桌上总共只有 5 根筷子。而哲学家就餐时需要两根筷子,即其左右的筷子。而一次只能取一根筷子,所以 5 个哲学家不能同时就餐。设计一个算法模

拟哲学家就餐的问题。

## 2. 实验要求

要求给出问题分析、算法说明和源代码,并且在 Tornado 环境中调试成功。

## 3. 设计方案

对于这个实验有很多种方法,下面例举两个方案进行说明。

### (1) 方案 1

#### ① 问题分析

解决哲学家进餐问题,就是要想办法解决操作系统中互斥和死锁的问题。两个哲学家中间只有一根筷子,一根筷子不能同时为两个哲学家所用,这就是互斥。即一组并发进程中的一个或多个程序段,因共享某一共有资源而导致它们必须作为一个不允许交叉执行的单位进行。

哲学家要进餐必须要用两根筷子。在特定的情况下,会出现每个哲学家都拿到了一根筷子,而又都在等待别的哲学家放下筷子的局面。在这种情况下,5 根筷子都得不到释放,而 5 个科学家都只能永远停留在等待状态不能进餐,这就是死锁。在操作系统中定义为,各并发进程彼此互相等待对方所拥有的资源,且这些并发进程在得到对方的资源之前不会释放自己所拥有的资源,从而造成大家都想得到资源而又都得不到资源,各并发进程不能继续向前推进的状态。

#### ② 算法实现

操作系统技术中是利用 P、V 原语和信号量来解决并发进程的互斥问题的。一个进程在进入临界区之前必须先对信号量执行 P 原语,在对临界区完成操作之后再对信号量执行 V 原语。如果在一个进程进入临界区且未离开之前,另一个进程又执行 P 原语申请进入临界区,第二个进程将会被阻塞。

在实验中,我们引入一个互斥信号量 mutex,在对哲学家状态进行读取之前先执行 semTake(mutex, TIME)以保证各进程不能同时存取哲学家状态。否则两个相邻哲学家可能同时检测到对方不为 eating,于是同时将各自置为 eating(即一把叉子为两人所用)。若成功再判断能否获得筷子,否则被阻塞,等待别的哲学家进程释放互斥信号量 mutex。在用完筷子后,还要执行 semGive(mutex)释放 mutex 信号量。这样就不会出现两个哲学家同时占有一个筷子的错误情况。

解决死锁的方法一般可分为预防、避免、检测与恢复等 3 种。预防是采用某种策略,限制并发进程对资源的请求,从而使得死锁的必要条件在系统执行的任何时间都不满足。避免则是指系统在分配资源时,根据资源的使用情况提前做出预测,从而避免死锁的发生。死锁的检测与恢复是指系统设有专门的机构,当死锁发生时,该机构能够检测到死锁发生的位置和原因,并能通过外力破坏死锁发生的必要条件,从而使得并发进程从死锁状态中恢复出来。

本实验使用的是预防死锁的方法,即在进程获得信号量之前加入条件判断。把哲学

家定为 3 个状态: thinking, hungry 和 eating。在哲学家获得筷子之前先判断, 只有该哲学家的左右两人均不在 eating 状态时才能获得两根筷子, 否则不允许拿任何一根筷子。

### ③ 代码分析

在程序中, 我们定义了一个互斥信号量 mutex, 一个二元的哲学家信号量数组 s[5], 以及一个整型数组 state[5]用以记录各哲学家的状态。

在主程序 Dinning 中, 用 takeSpawn 函数生成 5 个哲学家进程。每个哲学家进程循环执行思考、取筷子、进食、放下筷子、再思考的过程, 分别对应着 think(), take\_forks(), eat(), put\_forks() 函数。

think() 函数打印哲学家在思考的信息, 并随机延时一段时间。之后哲学家进入 hungry 状态。

take\_forks() 函数先调用 semTake(mutex, TIME) 来取得 mutex 互斥信号量, 以保证各进程不能同时访问 state[] 数组。若不成功则等待, 否则执行 test() 函数进行判断。若 state[LEFT] 及 state[RIGHT] 都不为 eating 时 state[i] 置为 eating, 并释放 s[i] 信号量, 表示该哲学家进程可以获得筷子。再用 semGive(mutex) 释放 mutex 信号量, 并调用 semTake(s[i], TIME) 取得 s[i] 信号量, 表示该哲学家已获得两根筷子, 可以进食了。

eating() 函数与 thinking() 函数相似, 打印哲学家进食的信息, 并随机延时一段时间。

进食完后执行 put\_forks() 函数。这时仍需调用 semTake(mutex, TIME) 来取得 mutex 互斥信号量, 将 state[i] 置为 thinking, 表示哲学家已吃完又进入思考状态。并调用 test() 检测左右两位哲学家, 将满足条件的置为 eating 并释放其信号量。然后释放 mutex 离开临界区。具体源代码如下。

```
barber.c
#include "vxWorks.h"
#include "taskLib.h"
#include "sysLib.h"
#include "stdio.h"
#include "stdlib.h"
#include "kernelLib.h"

#include "sysLib.h"
#include "semLib.h"

#define N 50
#define MAXCHAIRS 5
#define SLEEPING 0
```

```

#define WORKING 1

int state;
SEM_ID customers;
SEM_ID mutex;
int waiting; /* number of customers */
int i;
void Customers(void);
void Barber(void);
void Work(void);
void Get_haircut(void);
void Haircut(void)
{
 int task1, task2;
 kernelTimeSlice(1);
 barber = semBCreate(SEM_Q_FIFO, SEM_FULL);
 customers = semCCreate(SEM_Q_FIFO, 0); /* option = ?? */
 mutex = semMCreate(SEM_FULL);
 task2 = taskSpawn("task2", 101, 0x100, 2000, (FUNCPTR)Customers, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0);
 task1 = taskSpawn("task1", 102, 0x100, 2000, (FUNCPTR)Barber, 0, 0, 0, 0, 0, 0, 0,
 0, 0);
}
void Barber(void)
{
 while(TRUE){
 if(waiting == 0){
 printf("The barber is sleeping.\n");
 state = SLEEPING;
 }

 / *****
 ***** 需要读者添加 *****
 * 等待 customers 和 mutex 信号量 *
 * 得不到就阻塞 *

```

```

***** /

 waiting-- = 1;
 semGive(mutex);
 Work(); /* haircut for customer */
}
}
void Customers(void)
{
for(i = 1; i <= N; i++){
 taskDelay(rand()/400); /* delay */
 printf("Customer %d comes. %d waiting.\n", i, waiting);
 if(waiting + 1 > MAXCHAIRS)
 printf("Customer %d leaves.\n", i);
 else
 Get_haircut();
}
}
void Work(void)
{
printf("The barber is serving a customer.\n");
taskDelay(rand()/200); /* delay */

}
void Get_haircut(void)
{
if(state == SLEEPING){
 printf("The barber is waken up.\n");
 state = WORKING;
}
semTake(mutex, WAIT_FOREVER);
waiting += 1;
semGive(mutex);
semGive(customers);
}

```



## (2) 方案2

## ① 问题分析

该问题中,哲学家争用的临界资源是叉子,因此需为每只叉子分别设置初值为1的互斥信号量。

```
Var fork:array[0,...,4] of semaphore; = 1,1,1,1,1;
repeat
 wait(fork[i]);
 wait(fork[(i+1)mod 5]);
 ...
 eat;
 ...
 signal(fork[i]);
 signal(fork[(i+1)mod 5]);
 ...
 think;
 until false;
```

上述算法虽然可以保证互斥地使用叉子,但可能造成死锁。假如5个哲学家同时拿起各自左边的叉子,便会出现循环等待的局面,发生死锁现象。具体解决办法有:

- a. 至多只允许4个哲学家同时进餐;
- b. 仅当哲学家左右两边的叉子都可用时,才允许他进餐;
- c. 规定奇数号哲学家先拿起左边的叉子,再拿起右边的叉子;而偶数号哲学家则先拿右边的叉子,再拿左边的叉子。

## ② 算法实现

我们用的算法是只允许4个哲学家同时进餐,以保证至少有一个哲学家可以进餐,最终才可能由他释放出其所用过的两只叉子,从而使更多的哲学家可以进餐。为此,需设置一个信号量Sm来限制同时进餐的哲学家数目,使它不超过4。故Sm的初值也应置为4。这样,第*i*个哲学家的活动可描述为:

```
repeat
 wait(Sm);
 wait(fork[i]);
 wait(fork[(i+1) mod 5]);
 end
 signal(fork[i]);
 signal(fork[(i+1) mod 5]);
 signal(Sm);
```

```
 think;
until false;
```

### ③ 代码分析

```
#include "vxworks.h"
#include "stdio.h"
#include "stdlib.h"
#include "semLib.h"
#include "kernelLib.h"
#include "taskLib.h"
#include "sysLib.h"
#include "logLib.h"

#define TASK_PRIORITY 101 //任务的优先级
#define TASK_STACK_SIZE 5000 //堆栈大小
#define NUMBER_OF_PHILOSOPHER 5 //哲学家人数
#define EATING_TIME 200 //吃饭时间
#define RANDMAX 300 //最大时延

/* number of i's left neighbor */
#define LEFT(x)int((x - 1 + NUMBER_OF_PHILOSOPHER) % NUMBER_OF_PHILOSOPHER)
/* number of i's right neighbor */
#define RIGHT(x)int((x + 1) % NUMBER_OF_PHILOSOPHER)

//初始化
LOCAL int forks[NUMBER_OF_PHILOSOPHER]; //每个哲学家可以使用的叉子数
LOCAL SEM_ID semId[NUMBER_OF_PHILOSOPHER];
LOCAL SEM_ID mutex;
LOCAL STATUS pickup(int x);
LOCAL STATUS putdown(int x);
LOCAL STATUS philosopher0(int p_num);

STATUS test(int x);

////////////////////////////////////
//程序入口
```

```

//
////////////////////////////////////
STATUS demo()
{
 //kernelTimeSlice(1);
 /* 设置系统的时间片长度 */
 int time_slice = sysClkRateGet()/2;
 if(kernelTimeSlice(time_slice) == OK)
 {
 printf("The timeslice is 1/2 seconds.\n");
 printf("----- \n");
 }

 int i;
 char taskname[11] = "professor0~"; //进程名字

 //创建 NUMBER_OF_PHILOSOPHER 个互斥信号量
 for(i = 0; i < NUMBER_OF_PHILOSOPHER; i++)
 {
 forks[i] = 2;
 if((semId[i] = semBCreate(SEM_Q_FIFO, SEM_EMPTY)) == NULL)
 {
 perror("Error in creating binary semaphore");
 return(ERROR);
 }
 }

 ///////////////////////////////////
 //创建互斥信号量,用于保护程序完整性(原语)
 if((mutex = semBCreate(SEM_Q_FIFO, SEM_FULL)) == NULL)
 {
 perror("Error in creating binary semaphore mutex");
 return(ERROR);
 }

 /* 创建 5 个 professor 任务 */
 for(i = 0; i < 5; i++)

```

```

{
 taskname[9] = i + '0';
 if(taskSpawn(taskname, TASK_PRIORITY, 0x100, 20000, (FUNCPTR)philosopher0, i,
0,0,0,0,0,0,0,0,0,0) == ERROR)
 {
 printf("taskSpawn %s failed\n", taskname);
 return(ERROR);
 }
}

return(1);
}

////////////////////////////////////
//哲学家进程
//
////////////////////////////////////
STATUS philosopher0(int p_num)
{
 FOREVER
 {
 int i;
 i = rand() % RANDMAX;
 taskDelay(i);
 // printf("Philosopher No. %d is now hungry(thinking)! \n\n", p_num);
 pickup(p_num); //拿起1两边的叉子
 printf("Philosopher No. %d is now eating... forks = %d\n\n", p_num, forks
[p_num]);
 taskDelay(EATING_TIME); //EATTING
 putdown(p_num); //放下两把叉子
 // printf("Philosopher No. %d has finished eating and he begins to
thinking! \n\n", p_num);
 }
}

```

```

////////////////////////////////////
//拿起叉子函数
//
////////////////////////////////////
STATUS pickup(int x)
{

/ *****
***** 需要读者添加 *****
* 获取互斥信号,进入保护区 *
* 尝试获得叉子 *
* 释放互斥信号,退出保护区 *
***** /

semTake(semId[x],WAIT_FOREVER); //wait for eat
printf("Philosopher No. %d is now eating~~~forks = %d\n\n",x,forks[x]);
printf("Ph. 0 = %d! Ph. 1 = %d! Ph. 2 = %d! Ph. 3 = %d! Ph. 4 = %d! \n\n",
forks[0],forks[1],forks[2],forks[3],forks[4]);
return(1);
}
////////////////////////////////////
//放下叉子函数
//
////////////////////////////////////
STATUS putdown(int x)
{
//进入保护区
semTake(mutex,WAIT_FOREVER); //wait for signal

forks[RIGHT(x)] = forks[RIGHT(x)] + 1; //put down the right fork
forks[LEFT(x)] = forks[LEFT(x)] + 1; //put down the left fork
printf("Philosopher No. %d get mutex to putdown! -! -! \n\n",x);
printf("Ph. 0 = %d! Ph. 1 = %d! Ph. 2 = %d! Ph. 3 = %d! Ph. 4 = %d! \n\n",
forks[0],forks[1],forks[2],forks[3],forks[4]);

```

```

//通知邻居可以进食
test (LEFT(x));
test (RIGHT(x));

//退出保护区
semGive(mutex);
return(1);
}

////////////////////////////////////
//查看左右是否正在进食
//
////////////////////////////////////
STATUS test(int x)
{
 int temp;

 //判断是否可以进食(左右是否没人在进食)
 if((forks[x] == 2)and(forks[RIGHT(x)] >= 1)and(forks[LEFT(x)] >= 1))
 {
 forks[RIGHT(x)] = (forks[RIGHT(x)] - 1); //take the right fork
 forks[LEFT(x)] = (forks[LEFT(x)] - 1); //take the left fork

 if((forks[RIGHT(x)] < 0) or (forks[LEFT(x)] < 0))
 {
 scanf("Error . %d", &temp);
 }
 printf("Ph. 0 = %d! Ph. 1 = %d! Ph. 2 = %d! Ph. 3 = %d! Ph. 4 = %d! \n\n",
 forks[0], forks[1], forks[2], forks[3], forks[4]);

 //释放哲学家进程信号量
 semGive(semId[x]);
 printf("Philosopher No. %d is now ! ~~~~! \n\n", x);
 }
 return OK;
}

```

④ 结果分析

```

#define EATING_TIME 200 //吃饭时间

```

```
#define RANDMAX
```

```
300
```

```
//最大时延
```

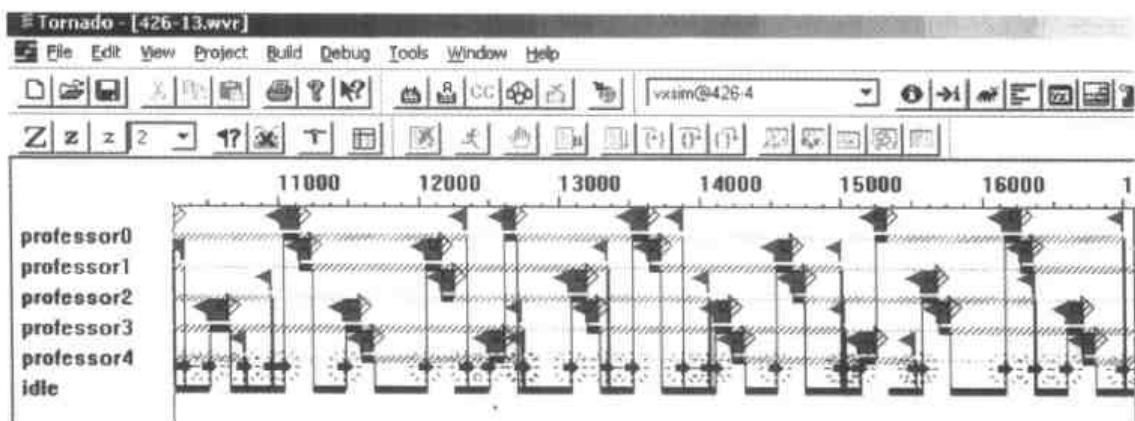


图 8-9 WindView 显示结果-1

```
#define EATING_TIME
```

```
20
```

```
//吃饭时间
```

```
#define RANDMAX
```

```
300
```

```
//最大时延
```

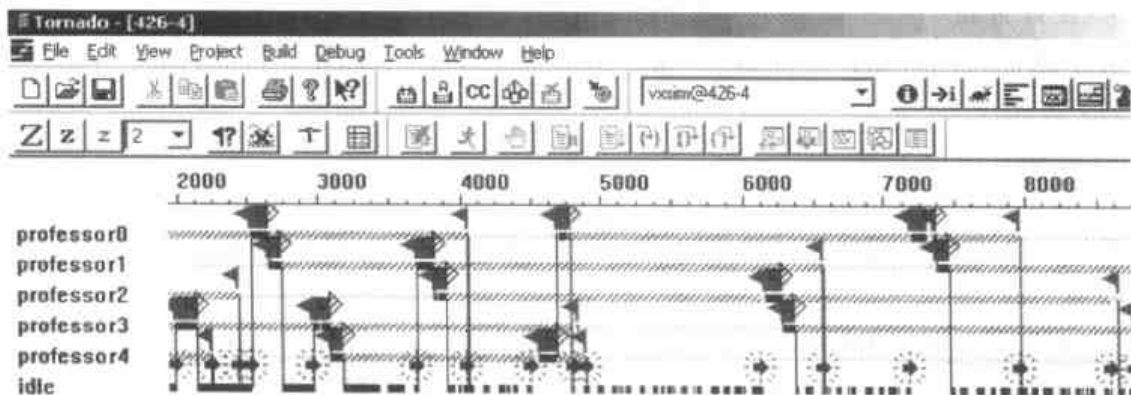


图 8-10 WindView 显示结果-2

我们采用“仅当哲学家左右两边地叉子都可用时,才允许他进餐”方法来解这个题目。从图 8-9 和图 8-10 可以看出:

- 5 个哲学家进程没有出现死锁,并且有最大的并行度(允许 2 个进程同时运行)。
- 当一个哲学家在进餐时,他左右的哲学家都无法进餐(进程被阻塞),直到他进餐完毕,放下叉子。
- 吃饭时间与哲学家饥饿的最大时延的关系。
  - 如果吃饭时间与最大时延大体相当,则进程的执行相当“紧凑”,每一时刻都有两个哲学家在进餐,每个进程执行完毕释放资源后,立刻会有别的进程被唤醒。
  - 如果吃饭时间比最大时延小很多时,则进程的执行相对“闲散”,有时候只有一个哲学家在进餐,甚至一个也没有。资源利用率较低,进程被阻塞的概率小。

#### 4. 问题思考

上述哪种方案的算法最好,进行分析比对,得出结论。

### 8.4 实验 4 路由器问题实验

#### 1. 问题描述

一个星形网络,有一个路由器、 $N$  个发送设备和  $M$  个接收设备。每个发送设备和接收设备都有唯一的标识。发送设备发送到接收设备的信息,必须经过路由器中转。路由器及接收设备的接入均有一定处理上限,当已经有超过一定数量的信息等待处理时,网络接入端繁忙,不能再向该设备发送信息。每个设备均有固定的信息处理时间。信息在网络中传输会引入随机的时延。

#### 2. 实验要求

要求给出问题分析、算法说明和源代码,并且在 Tornado 环境中调试成功。

#### 3. 设计方案

对于这个实验有很多种方法,下面例举两个方案进行说明。

##### (1) 方案 1

##### ① 问题分析

a. 整个系统可以分成两部分,从发送设备到路由器的过程和从路由器到接收设备的过程。从发送设备到路由器是一个多对一的过程,从路由器到接收设备是一个一对多的过程。

b. 发送设备到路由器这一端,发送设备随机产生信息,如果路由器接入繁忙则放弃,否则则发送;对于路由器,如果有信息过来则转发,没有的话就等待。可见,这是一个多缓冲的生产者-消费者问题,即多个生产者一个消费者。

c. 由 b,所以路由器的接入端为一个多缓冲的信息队列,这个信息队列是发送进程竞争的资源,故还需为它们设置一个资源信号量 count。

d. 路由器到接收设备这一端,在实际中,一个接收设备一次只能接收一个信息,所以对于某个特定的接收设备,它与路由器间的关系应为单缓冲的生产者-消费者问题。而在本系统中有多接收设备,所以是一个多重的生产者-消费者问题。

e. 由 d,所以路由器的接入端为  $M$  个单缓冲的信息队列,每个信息队列都是一个临界资源,所以要分别为它们设置一个互斥信号量 `semID[x]`。

##### ② 算法实现

参照问题描述。

##### ③ 代码分析

```
#include <semLib.h>
```



```

#include <msgQLib.h>
#include <taskLib.h>
#include <stdlib.h>
#include <stdio.h>
#include <kernelLib.h>

#define NUM_SENDER 2 /* the number of the NUM_SENDER */
#define NUM_RECEIVER 2 /* the number of the NUM_RECEIVER */
#define NUM_MSG 3 /* the max number the switcher can receive */
#define RANDMAX 50 /* the max interval of customer's arrival */
#define TASK_PRIORITY 101 /* 任务的优先级 */

//消息包结构
typedef struct S_MESSAGE
{
 int s_id; //sender id
 int r_id; //receiver id
}MESSAGE;

MSG_Q_ID msgQId;
MSG_Q_ID msgRId[NUM_RECEIVER];
SEM_ID semId[NUM_RECEIVER]; /* free receiver take it! */

STATUS sender(int i);
STATUS receiver(int r_num);
STATUS switcher();
STATUS msg_send(int s_id,int r_id);
//程序入口
STATUS go()
{
 kernelTimeSlice(200);

 int taskid1,taskid2,i;
 char taskSname[] = "tasksender0";
 char taskRname[] = "taskreceiver0";

```

```

////////////////////////////////////
//创建 NUM_RECEIVER 个互斥信号量 semId[x]和单缓冲消息队列 msgRld
//及接收进程
////////////////////////////////////
for(i = 0; i < NUM_RECEIVER; i++)
{
 if((semId[i] = semBCreate(SEM_Q_FIFO, SEM_FULL)) == NULL)
 {
 printf("ERROR!");
 return(ERROR);
 }
 if((msgRld[i] = msgQCreate(1, sizeof (MESSAGE), MSG_Q_FIFO)) == NULL)
 {
 printf("ERROR!");
 return(ERROR);
 }
 taskRname[12] = '0' + i;
 if(NULL == (taskid2 = taskSpawn(taskRname, TASK_PRIORITY, 0x100, 2000,
(FUNCPTR)receiver, i, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)))
 {
 printf("ERROR!");
 return(ERROR);
 }
}

//创建多缓冲消息队列
msgQId = msgQCreate(NUM_MSG, sizeof (MESSAGE), MSG_Q_FIFO);

//创建路由进程
taskid1 = taskSpawn("taskswitcher", TASK_PRIORITY, 0x100, 2000, (FUNCPTR)
switcher, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0);

//创建 NUM_SENDER 个发送进程
for(i = 0; i < NUM_SENDER; i++)
{

```

```

 taskSname[10] = '0' + i;
 taskid2 = taskSpawn(taskSname, TASK_PRIORITY, 0x100, 2000, (FUNCPTR) sender, i,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0);
 }
 return(OK);
}

////////////////////////////////////
//路由进程
//
////////////////////////////////////
STATUS switcher()
{
 MESSAGE msg;
 int taskNum = 0;
 FOREVER
 {
 //从消息队列接收发送进程发过来的消息
 if(msgQReceive(msgQId, (char *) &msg, sizeof(MESSAGE), NO_WAIT) !=
ERROR)
 {
 printf("Msg from S%d to R%d has arrived Switcher\n!", msg.s_
id, msg.r_id);

 / *****
 ***** 需要读者添加 *****
 * 按照消息目的地标签,把消息转发给被要求的接收进程 *
 * 线路繁忙则抛弃该消息包 *
 ***** /

 }
 //消息处理时间
 taskDelay(RANDMAX/5);
 }
 return(OK);
}

```

```

 }

 //////////////////////////////////////

 //发送设备进程
 //
 //////////////////////////////////////

 STATUS sender(int i)
 {
 int target; //目标地
 MESSAGE msg;
 msg.s_id = i;
 FOREVER
 {
 taskDelay(rand() % RANDMAX); //随机时延
 target = rand() % NUM_RECEIVER; //随机产生目标地址
 msg.r_id = target; //设置消息的目标地址域
 //如果接入线路繁忙则等待一定时间后再尝试发送
 while(msgQSend(msgQId,(char *)&msg,sizeof(MESSAGE),NO_WAIT,MSG_Q
_FIFO) == ERROR)
 {
 //线路忙
 printf("Sender %d trying to call Receiver %d,but switcher busy!
\n",i,target);

 //等待一定个随机时延
 taskDelay(rand() % RANDMAX);
 }
 printf("Sender %d trying to call Receiver %d,msg has send to switc-
her! \n",i,target);
 }
 return(OK);
 }

 //////////////////////////////////////

 //接收设备进程
 //

```

```

////////////////////////////////////

STATUS receiver(int r_num)
{
 MESSAGE msg;
 FOREVER
 {

 / *****
 ***** 需要读者添加 *****
 * 查看是否有消息发过来 *
 ***** /

 {
 //判别是否有送错的消息
 if(r_num == msg.r_id)
 printf("Recever %d received msg from Sender %d ... \n", r_num, msg.s_id);
 else
 printf("ERROR, Recever %d received msg to Recever %d ... \n", r_
num, msg.r_id);

 //释放自己的互斥信号量
 semGive(semId[r_num]);
 }

 //消息处理时间
 taskDelay(RANDMAX/2);
 }
 return(OK);
}

////////////////////////////////////
//消息发送进程
//
////////////////////////////////////
STATUS msg_send(int s_id, int r_id)
{

```

```

MESSAGE msg;
msg.s_id = s_id;
msg.r_id = r_id;

//等待目的地消息队列的互斥信号量
semTake(semId[r_id],WAIT_FOREVER);

//发送消息
msgQSend(msgRId[r_id],(char *)&msg,sizeof(MESSAGE),WAIT_FOREVER
,MSG_Q_FIFO);

//释放信号量
semGive(semId[r_id]);
return(OK);
}

```

#### ④ 运行结果分析(如图 8-11 所示)

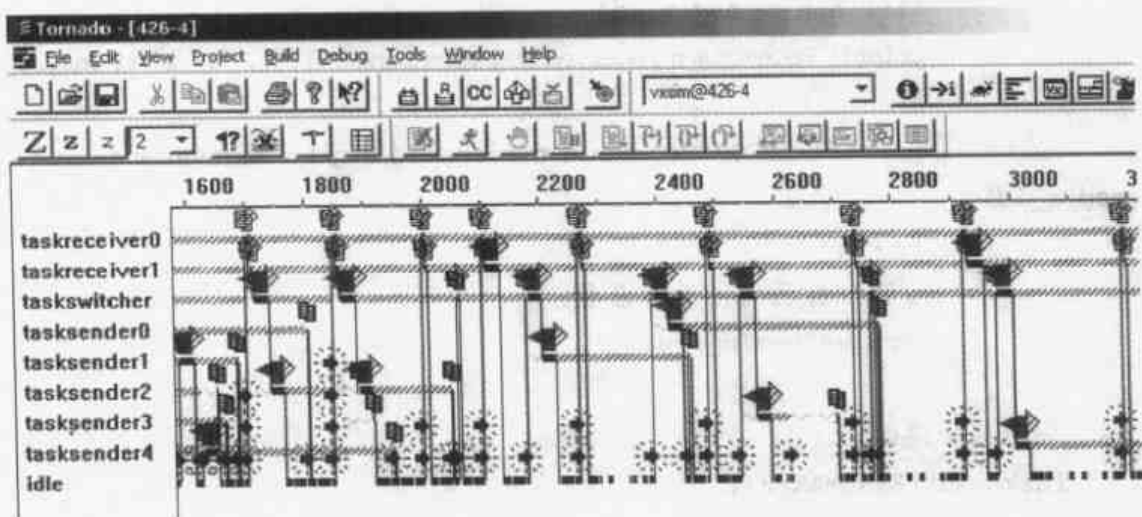


图 8-11 WindView 结果分析

a. 发送进程(tasksender)准备发送消息时,如果消息队列满了,则该进程会被阻塞,直到路由进程(taskswitcher)把消息队列中的消息读走。

b. 只有发送进程(tasksender)会被阻塞,这是因为这个模型是这样设计的:发送进程在尝试发送消息时,发现消息队列满了,会一直等待;而路由进程往接收进程发消息时,如果消息队列满了,则放弃发送。

c. 接收进程(taskreceiver)执行的时间有时候长有时候短,这是因为接收进程(taskreceiver)运行时,如果发现此时并没有消息需要处理就会立即让出 CPU,执行时间就短;

反之,则需要加上消息处理时间,执行时间就长。

d. 有很多时候 CPU 处于 IDLE 状态,这是因为我们设的进程轮询调度时间片较长(200),而进程的一次任务执行时间较短。

## (2) 方案 2

### ① 问题分析

本方案的内容是仿真提供 QoS 区分服务(DifferServ)的路由器对分组进行着色标记的过程,所用的算法是经过简化的令牌桶算法。该程序的仿真是在作了一系列简化的基础上进行的。

先简要介绍一下方案的背景。目前,IP 网络的 QoS 问题,信息网络研究的热点之一。而人们比较看好的方案是 IETF 提出的 DifferServ 模型,该模型旨在改进 IP 网络的“尽力而为”的服务方式,通过对业务流设置优先级,求保证高优先级的业务流得到较高的服务质量,具体而言,就是较低的时延、较小的时延抖动等等。而实现 DifferServ 一般要包括分类、标记、调度等过程,本方案主要仿真的是标记过程,对一个给定的业务流进行监督,如果该业务流没有超出流量的约束,所有的分组就标为绿色,否则,具有一部分分组被标记为红色。红色意味着这些不合法的分组在网络拥塞时会被丢弃。因为这些算法本身都比较复杂,所以本程序进行了很大的简化,只仿真一个宏观的模型。具体说明如下:

#### a. 业务流

该程序中仿真的业务流是突发的、变长的分组流。Void Stream()任务完成这一过程。这些分组流到达的间隔、每个分组流包含的分组个数以及分组的大小均是随机的。Void Stream()通过调用 PacketGenerator()来产生所要处理的分组流。

#### b. 缓冲区

分组流进入路由器后被存放在的缓冲区中,该缓冲区有程序中的类 Buffer 来实现,该 Buffer 是一个环形队列,体现先到先服务的原则。类 Buffer 有以下成员函数:buffer(void){}、Void Clear()用于清空缓冲区,long ShowSize()用于查询缓冲区中的分组数,int OutputOnePacket()用于取出一个分组,Void InputOnePacket(int&temp)用于放入一个分组。用类来设计缓冲区,可以方便地进行扩展,如可以同时仿真几个业务流。

#### c. 简化的令牌桶算法

路由器对分组进行着色标记的过程较为简单,即简化的令牌桶算法。令牌实际上表征着路由器和网络的资源。在该程序中,令牌桶有规律地得到补充,同时又不断地被到达路由器的分组流所消耗,使得到达路由器的分组被标为红色或绿色。绿色意味着该分组是符合业务流规范的,而红色意味着分组流超过了业务流允许的范围,将在后续的传输中遇到网络拥塞时被丢弃。令牌桶算法在本程序中被简化,因为主要的目的是仿真一个软实时系统的整体特征,而不是研究算法的优劣。Void TokenBucket()是进行分组标记的任务,Void TokenAdd()是补充令牌桶的任务。

d. 标记结果的报告

Void Report()用来实现这一功能。

② 算法分析

参照问题分析。

③ 源程序

```
#include<vxworks.h>
#include<taskLib.h>
#include<semLib.h>
#include<sysLib.h>
#include<stdio.h>
#include<wdLib.h>
#include<stdLib.h>
#include<kernelLib.h>
#include<logLib.h>
//defines
#define TASK_STACK_SIZE 2000 //任务堆栈的大小
#define Token_ADD_INTERVAL 15 //令牌增加的间隔
#define Token_ADD_MAX 320000 //每次增加的令牌
#define BUFFER_SIZE 60000 //环形缓冲区的大小
#define ADJUSTOR1 0.94 //调节令牌增加的速度
#define ADJUSTOR2 0 //调节分组标记器的执行速度
#define ADJUSTOR3 420000 //调节突发业务流到达间隔
#define ADJUSTOR4 6436 //调节业务流中分组的间隔时间
#define PR1_TA 101 //任务 TokenAdd()的优先级
#define PR1_TB 102 //任务 TokenBucket()的优先级
#define PR1_STRM 102 //任务 Stream()的优先级
#define PR1_RP 102 //任务 Report()的优先级
#define PACKET_LENGTH_MAX 1500 //分组产生的最大长度
#define PACKET_STRM_MAX 40000 //每个突发分组流的分组数
//globals
int num[3] = {0,0,0}; //num[0]接收分组总数,num[1]绿色
分//组个数,num[2]红色分组个数;
long Token_Num = 0; //跟踪令牌桶的容量
SEM_ID mutex; //修改分组统计结果时的互斥信号量
//declarations
```



```

void PacketGenerator(int i); //被任务反复调用
//class Buffer
class buffer
{
 long head;
 long tail;
 long size;
 int packet[BUFFER_SIZE];
public:
 buffer(void); //拷贝构造函数
 voidClear(); //清空缓冲区
 long ShowSize(); //查询缓冲区中的分组数
 int OutputOnePacket(); //取出一个分组
 void InputOnePacket(int&temp); //放入一个分组
}
//成员函数的实现
buffer::buffer():head(0),tail(0),size(0)
{}
Void buffer::Clear()
{
 Head = 0;
 Tail = 0;
 Size = 0;
}
long buffer::ShowSize()
{
 Return(size);
}
int buffer::OutputOnePacket(void)
{
 int temp = 0;
 if(size == 0)
 {
 //printf("non packets\n");
 //exit(1);
 }
}

```

```

 }

 Temp = packet[head];
 Size -- ;
 head = (head + 1) % BUFFER_SIZE;
 return temp;
}

void buffer::InputOnePacket(int &temp)
{

 / *****
 ***** 需要读者添加 *****
 * 输入一个包 *
 ***** /

}

//由类生成一个对象,为一环形缓冲区
buffer mybuffer;
//tasks
void TokenBucket()
{
 int temp;
 long temp2;
 int flag = 1;
 while(1)
 {
 flag = mybuffer.ShowSize();
 if(! flag) continue;
 temp = mybuffer.OutputOnePacket();
 if(temp != 0) {num[0] ++ ;}
 if(temp >= Token_Num)
 (//printf("tokens are all consumed! \n");
 num[2] ++ ;}
 if(temp != 0 && temp <= Token_Num)
 {Token_Num -= temp;
 num[1] ++ ;}
 }
}

```

```

//调节分组处理速度的参量
for(temp2 = 0;temp2<ADJUSTOR2;temp2 ++);
 }
}
//令牌桶增加的任务
void TokenAdd()
{
 while(1)
 {
 taskLock();
 printf("***** ADDING TOKEN ***** \n");
 printf("unadded Token_Num: %d\n",Token_Num);
 Token_Num += (long)(ADJUSTOR1 * Token_ADD_MAX);
 printf("added Token_Num: %ld\n",Token_Num);
 taskUnlock();
 taskDelay(Token_ADD_INTERVAL);
 }
}
//突发分组流的模拟程序,通过调用 PacketGenerator()来实现。
void Stream(void)
{int temp;
 long temp1;
 long delay1;
 while(1)
 { temp1 = (long)(0.2 * ADJUSTOR3) + rand() % ADJUSTOR3;
 for(delay1 = 0;delay1<temp1;delay1 ++);
 temp = rand() % PACKET_STRM_MAX;
 PacketGenrator(temp);
 }
}
//报告标记结果的任务
void Report()
{
 int i;
 while(1)

```

```

 {
 taskLock();
 logMsg("\n ***** START REPORT ***** \n",0,0,
0,0,0,0);
 logMsg("The whole packetsnumber: % d,green number: % d,red number: % d\n",num
[0],num[1],num[2],0,0,0);
 for(i = 0; i < 3; i++) num[i] = 0;
 logMsg("The result is clear,calculate again\n",0,0,0,0,0,0);
 logMsg("\n ***** END REPORT ***** \n\n",0,0,0,0,0,0);
 taskUnlock();
 taskDelay(5);
 }
}

void PacketGenerator(int i)
{
 int j;
 int temp2,delay2;
 int flag = 0;
 int packet_len;
 for(j = 0; j < i; j++)
 {
 temp2 = (long)(0.2 * ADJUSTOR4) + rand() % ADJUSTOR4;
 for(delay2 = 0; delay2 < temp2; delay2++);
 packet_len = rand() % PACKET_LENGTH_MAX;
 flag++;
 mybuffer.InputOnePacket(packet_len);
 }
 logMsg("% d packet produced! \n",flag,0,0,0,0,0);
}

//main
void QosMarker(void)
{
 taskSpawn("TokenBucket",PR1_TB,0x0,TASK_STACK_SIZE,(FUNCPTR)Token-
Bucket,0,0,0,0,0,0,0,0,0);
 taskSpawn("TokenAdd",PR1_TA,0x0,TASK_STACK_SIZE,(FUNCPTR)TokenAdd,0,0,0,

```

```

0,0,0,0,0,0,0,0);
 taskSpawn(~Report~,PR1_RP,0x0,TASK_STACK_SIZE,(FUNCPTR)Report,0,0,0,
0,0,0,0,0,0,0,0);
 taskSpawn(~Stream~,PR1_STRM,0x0,TASK_STACK_SIZE,(FUNCPTR)Stream,0,0,
0,0,0,0,0,0,0,0);
 kernelTimeSlice(50);
}

```

#### 4. 问题思考

(1) 原来的设计并没有考虑消息处理时间,所以在路由进程(taskswitcher)和接收进程(taskreceiver)中引入延时,这样虽然能正常运行,但是无法采集数据获得仿真图。这是因为,不设延时的话,进程会不断运行,产生的数据量特别大(用 windview control 窗口采集数据时会很快弹出缓存区溢出的提示),更无法加载数据到仿真图。如何解决这个问题?

(2) 比较两个方案的优缺点。

## 8.5 实验5 构架 WebServer 服务器

### 1. 实验目的

掌握基于 VxWorks 操作系统构架 WebServer 服务器的技术。

### 2. 实验内容

(1) 在 Tornado 中调试配置 goahead WebServer 源码。

(2) 通过 PC 主机访问 WebServer 服务器,查看 WebServer 页面访问效果。

### 3. 预备知识

#### goahead WebServer 介绍

goahead WebServer 是源代码开放的嵌入式 Web 服务器系统,在提供用户免费使用的同时,还向用户提供了程序源代码,提供了开发具备 Web 功能的智能产品的基础,用户可以根据自己的需要对该系统进行调整或对不足之处进行改进。

goahead WebServer 是基于开放标准的 Web 服务器系统,能够提供对嵌入式 Web 服务器多种标准的支持。与许多同类产品通过定义权限实现动态数据插入的 Web 页操作不同,goahead WebServer 采用了微软公司先进的 active server pages 技术,加上开放的源代码,可以自由地对数据进行相关处理操作,消除了定义权限的实现方法中二次开发遇到的操作权限问题。同时,goahead WebServer 还提供了方便的远程 Web 浏览器管理和访问功能,因而特别适合用户开发智能设备。

另外,goahead 支持 asp、嵌入式 javascript、CGI 以及静态页面 html 格式;快速响应,

每秒可处理超过 50 个请求;如果不包含 SSI,仅要求 60 K 的内存;包含 SSI,要求 500 K 内存;goahead 支持多种操作系统,包括 VxWorks、Linux、Windows、WinCE、pSOS 等。

#### 4. 实验设备和工具

北京邮电大学电信工程学院 IRDC 实验室开发的 IRDC05v3 教学实验/评估板,PC 机,Tornado 开发环境,Win2000/XP 平台。

#### 5. 实验步骤

##### (1) 设计思路

将 goahead WebServer 的源码(免费公开)进行配置修改,建立 VxWorks+tffs+goahcad+ARM 的 WebServer 环境。

##### (2) 创建工程,调试程序

① 建立基于 BSP 的 downloadable 工程,名称为 goahcadweb。可以通过 downloadable 工程来架设 GoAhead WebServer。当然也可以用类似的方法把它集成到 bootable 工程中,将应用程序和 VxWorks 映像文件捆绑到一起下载到目标机中。

② 将下列下载的源文件加入到工程中。

balloc.c  
base64.c  
default.c  
ejlex.c  
ejparse.c  
emfdb.c  
form.c  
h.c  
handler.c  
md5c.c  
mime.c  
misc.c  
page.c  
ringq.c  
rom.c  
security.c  
sock.c  
sockGen.c  
sym.c  
uemf.c  
um.c

```

umui.c
url.c
value.c
webrom.c
webs.c
asp.c
websuemf.c
cgi.c
/vxworks/main.c

```

③ 参考/vxworks/makefile 来修改工程的 makefile。

在 makefile 中增加定义 `-DWEBS-DUEMF-DOS="VXWORKS" -DVXWORKS` 如下所示。

```

CFLAGS = -g -m486 -ansi -nostdinc -DRW_MULTI_THREAD -D_REENTRANT -
DWEBS -DUEMF -DOS="VXWORKS" -DVXWORKS -fvolatile -nostdlib -fno-builtin
-fno-defer-pop -I. -IE:/Tornado-x86/target/h -DCPU=I80486

```

如果需要增加用户管理、存取控制支持,则还需要增加下列定义

```

-DUSER_MANAGEMENT_SUPPORT
-DDIGEST_ACCESS_SUPPORT

```

具体可以参考原 makefile 进行。

④ 根据文档修改/vxworks/main.c。

设置根目录,例如

```
#define ROOT_DIR T("~/tffs0/webs")
```

设置缺省主页

```
websSetDefaultPage(T("index.asp"));
```

利用 ftp 等工具将 index.asp 等主页下到 tffs0 中。

⑤ 编译为 goaheadweb.out 下载。

⑥ 在 WindShell 下加载 WebServer:输入 `sp websvxmain`。

在浏览器端键入 `http://192.168.0.2/index.asp` 就可以访问/tffs0/webs/index.asp 文件了。

⑦ Asp 编程。

GoAhead Asp 编程首先需要预先定义 Asp 函数,在 main.c 的函数 `initWebs()` 中有如下的例子:

```
websAspDefine(T("aspTest"), aspTest);
```

这里将 `aspTest()` 定义为 Asp 函数,用户可提供相关的函数来进行想要完成的功能,

例如:

```
/*
 * Test Javascript binding for ASP. This will be invoked when "aspTest" is
 * embedded in an ASP page. See web/asp.asp for usage. Set browser to
 * "localhost/asp.asp" to test.
 */
static int aspTest(int eid, webs_t wp, int argc, char_t * * argv)
{
 char_t * name, * address;

 if (ejArgs(argc, argv, T("%s %s"), &name, &address) < 2) {
 websError(wp, 400, T("Insufficient args\n"));
 return -1;
 }
 return websWrite(wp, T("Name: %s, Address %s"), name, address);
}
```

然后在 Asp 的页面中调用 Asp 函数即可,如调用上述例子:

```
<h2>Expanded ASP data:< % aspTest("Peter Smith", "112 Merry Way"); % ></h2>
```

注意:如果要停止或重启 WebServer,请不要使用 taskDelete 或者 td 来删除 WebServer 任务,而是调用 kill()来给 WebServer 任务发送 9 或者 15 信号。

### (3) 查看程序运行结果

通过 PC 或者局域网内其他主机访问嵌入式 Web 主机,看是否有测试页面出现。如果不行,检查是否可以 PING 通,WebServer 目标机设置是否正确。

还可以借助一些调试工具如 Browser 等监视资源使用情况,用 WindView 或者 Shell 等工具分析任务执行情况。

## 6. 思考题

- (1) 如何构架可以远程访问的 WebServer 服务器?
- (2) 有没有别的方式实现 WebServer 服务器?



# 附 录

## 附录 A 参考文献

- [1] C. M. Krishna, Kang G. Shin. 戴琼海,译. 实时系统 Real-time Systems. 清华大学出版社,2004
- [2] 陈智育,温严军,等. VxWorks 程序开发实践. 人民邮电出版社,2004
- [3] 邝坚. Tornado/VxWorks 入门与提高. 科学出版社,2004
- [4] 徐德明. 操作系统原理 Linux 篇. 国防工业出版社,2004
- [5] 周立功,等. ARM 微控制器基础与实战. 北京航天航空出版社,2003
- [6] Steve Furber. 田泽等译. ARM SoC 体系结构. 北京航天航空出版社,2003
- [7] 周启平,张杨,等. VxWorks 开发指南与 Tornado 实用手册. 中国电力出版社,2004
- [8] 曾非一. 嵌入式系统内存管理方案研究. 单片机及嵌入式系统应用,2005 (1)5-7

## 附录 B 参考电路图

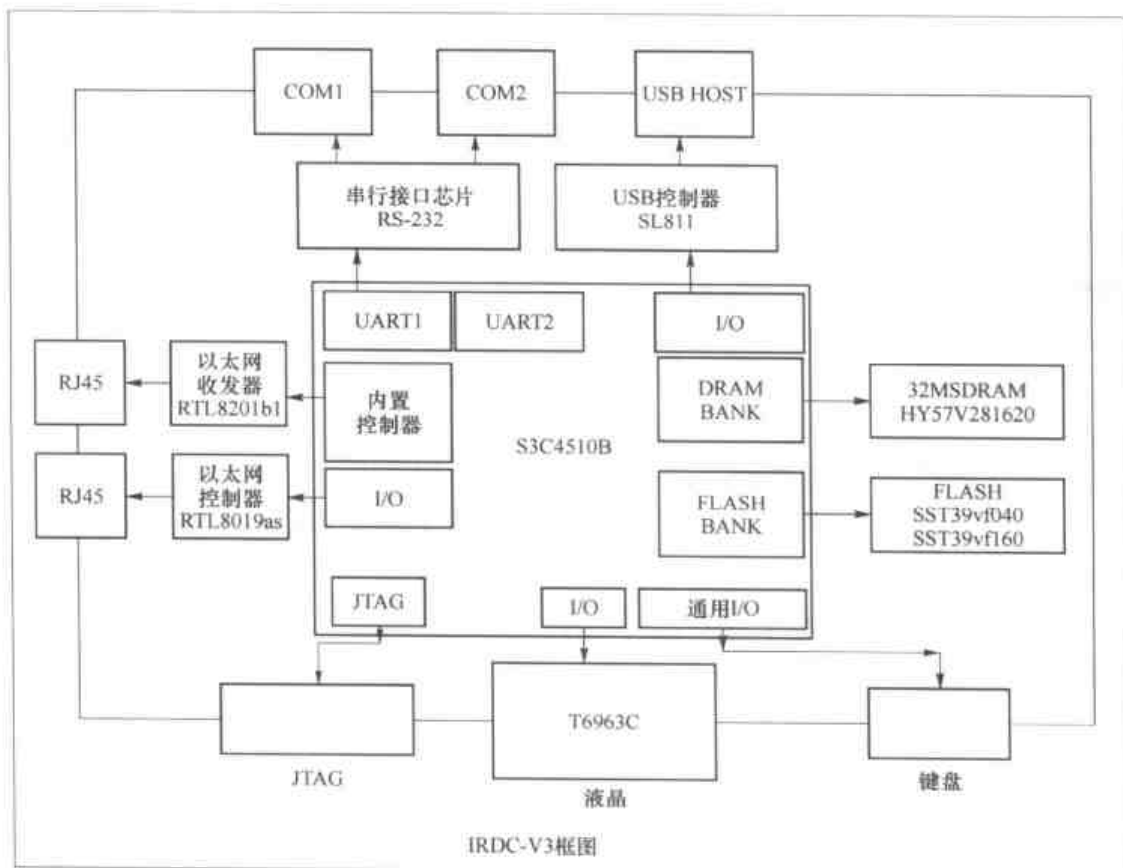


图 B-1 电源电路

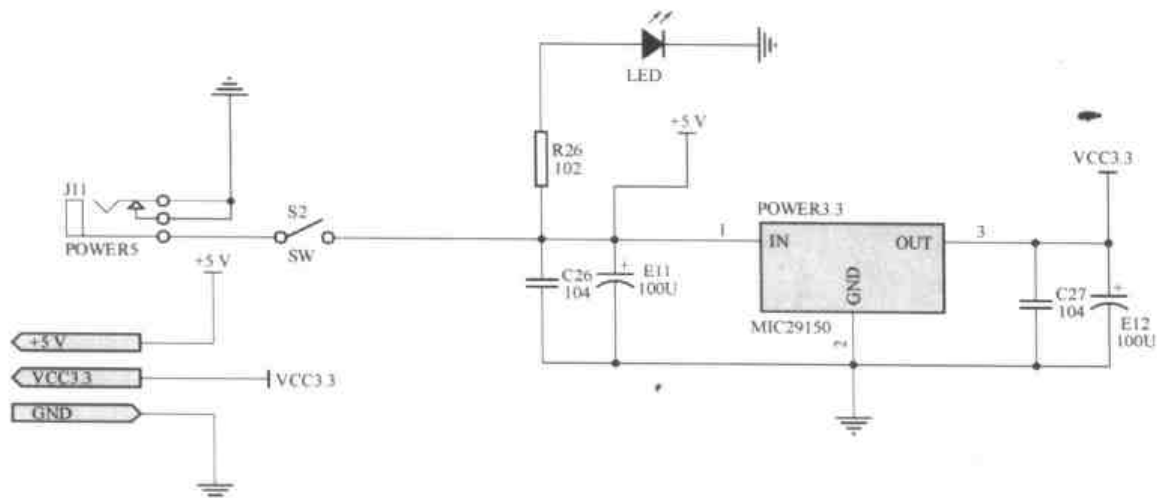


图 B-2 复位电路

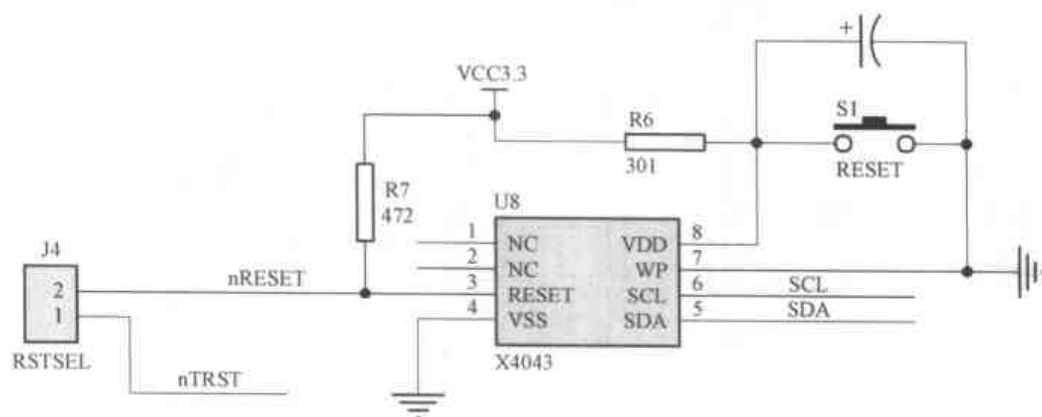


图 B-3 时钟晶振电路

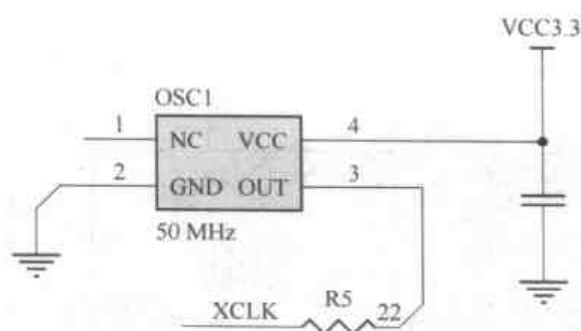


图 B-4 JTAG 接口

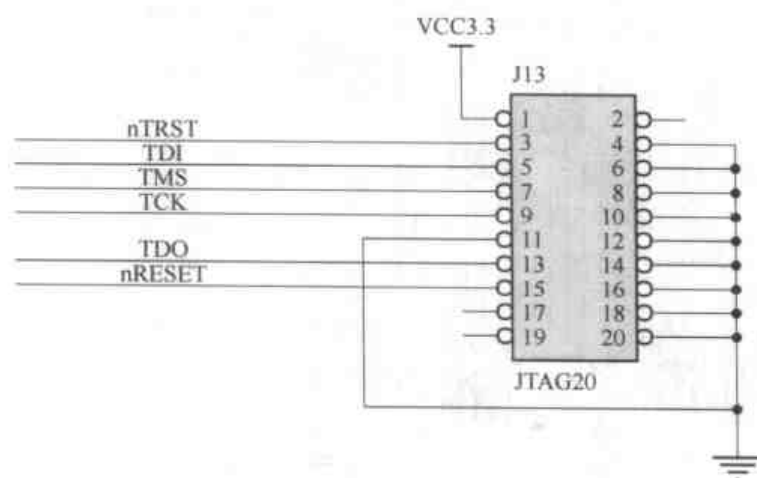
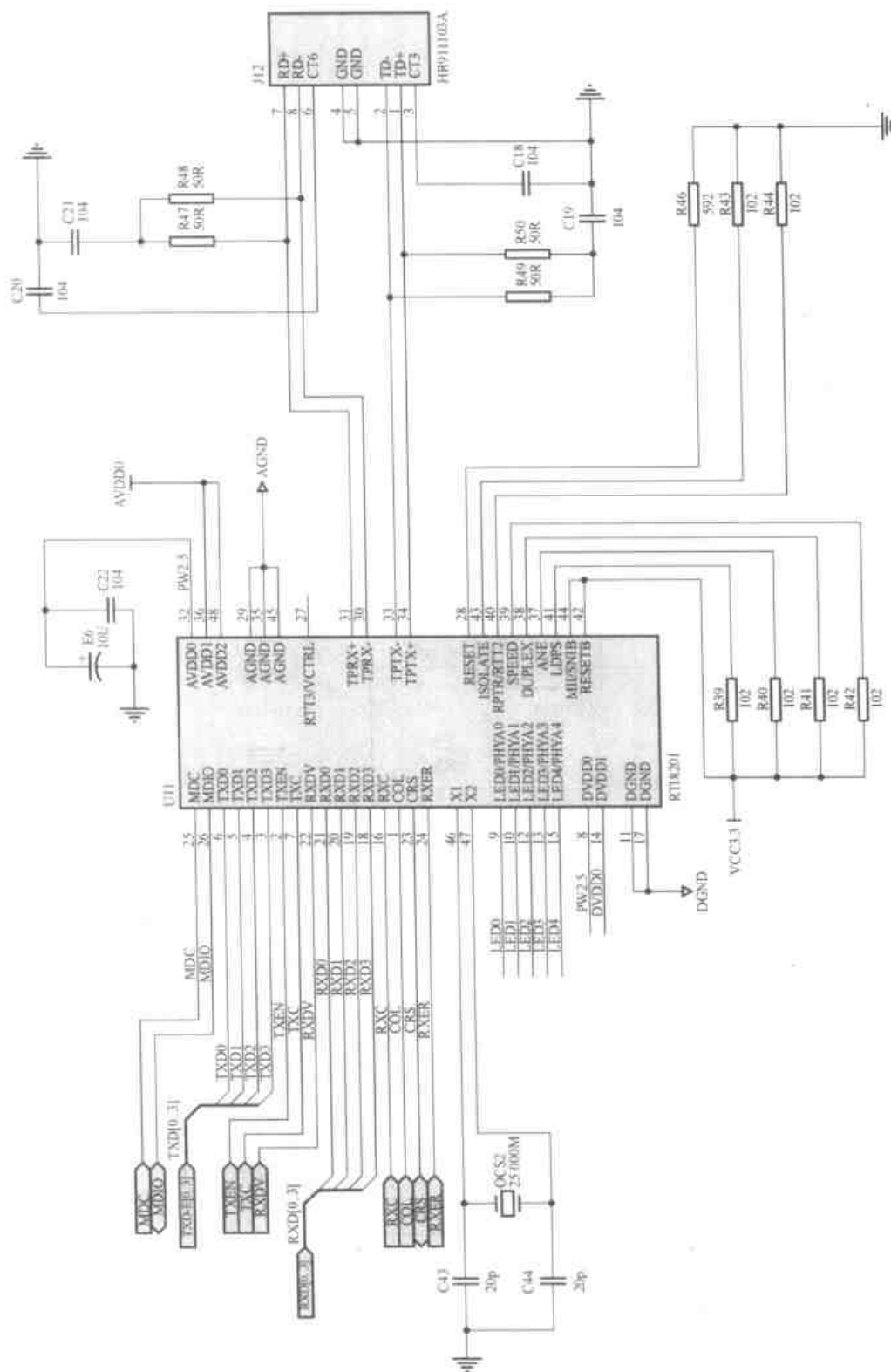
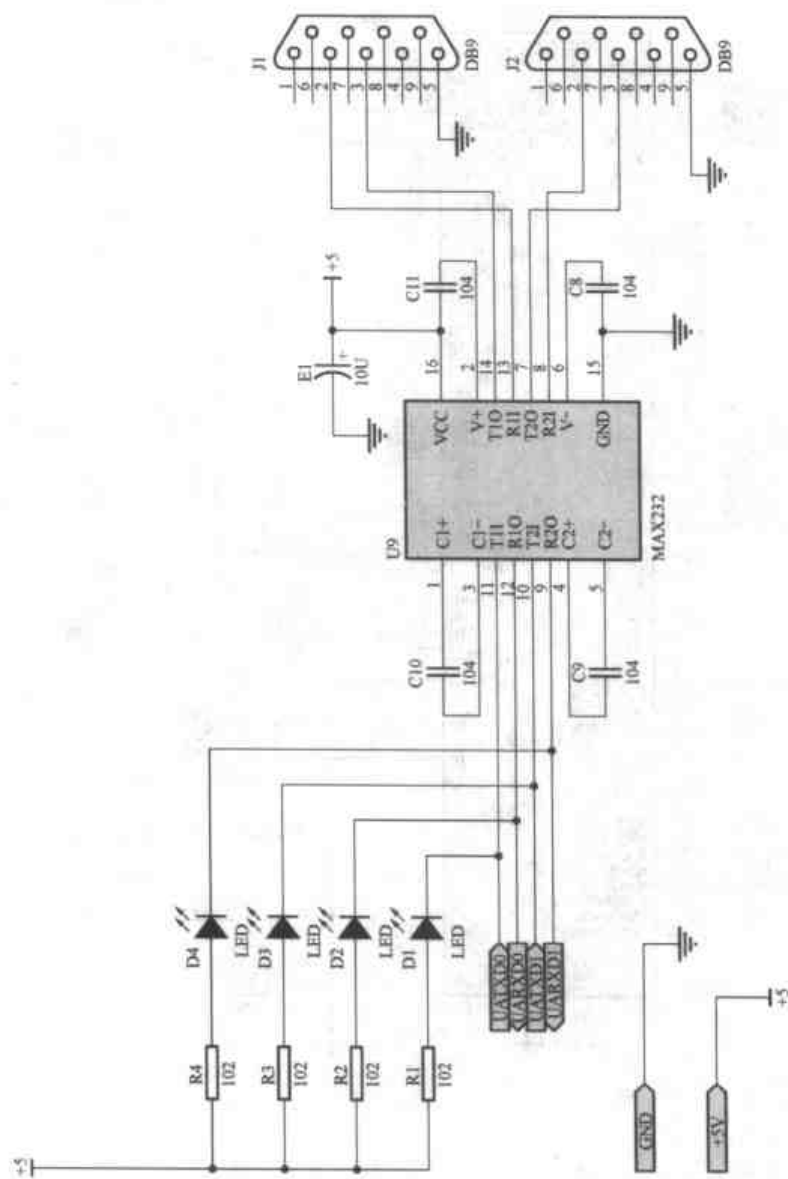


图 B-5 RTL8201 PHY



图B-6 UART



图B-7 RT8019AS

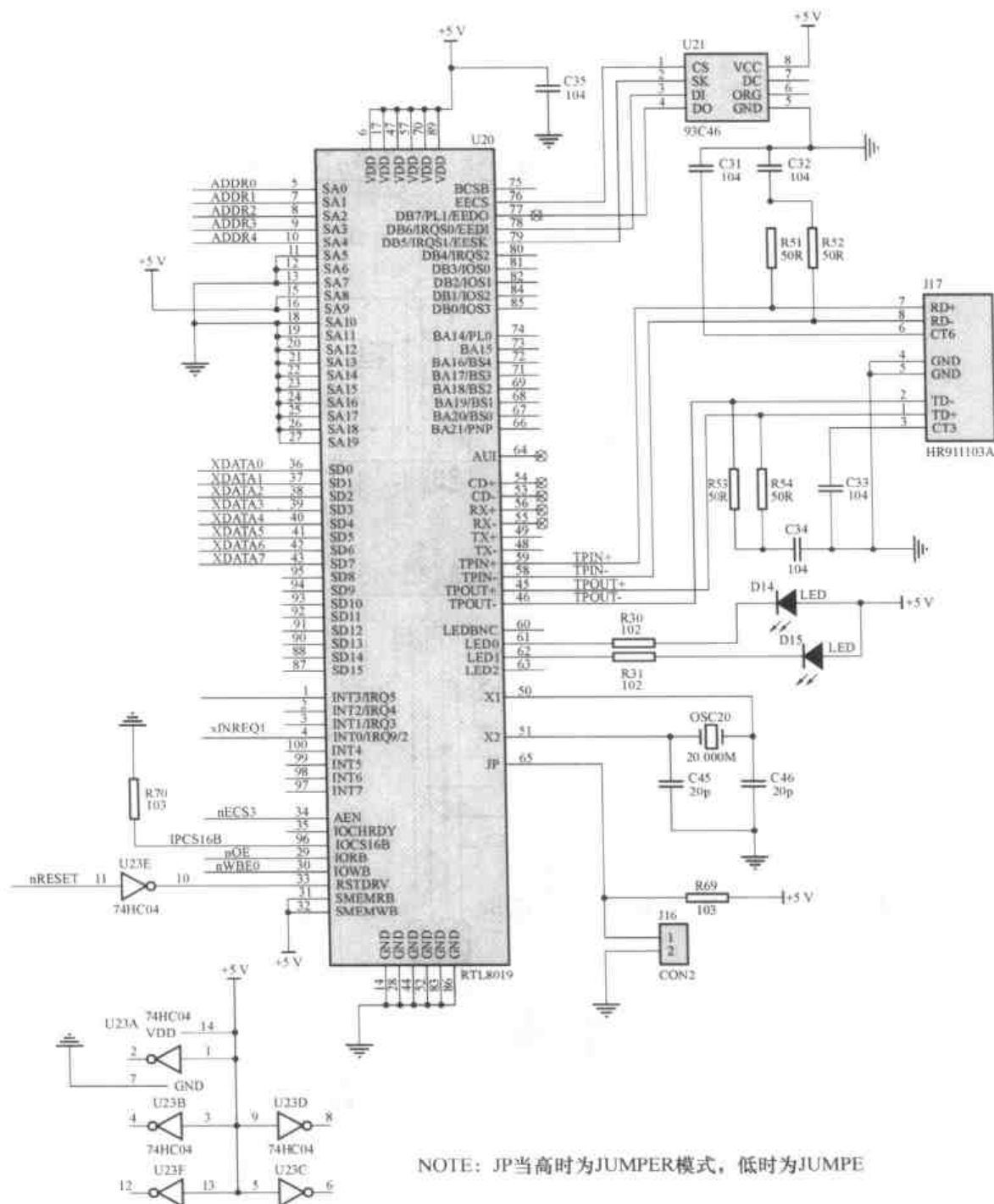


图 B-8 SL811HST



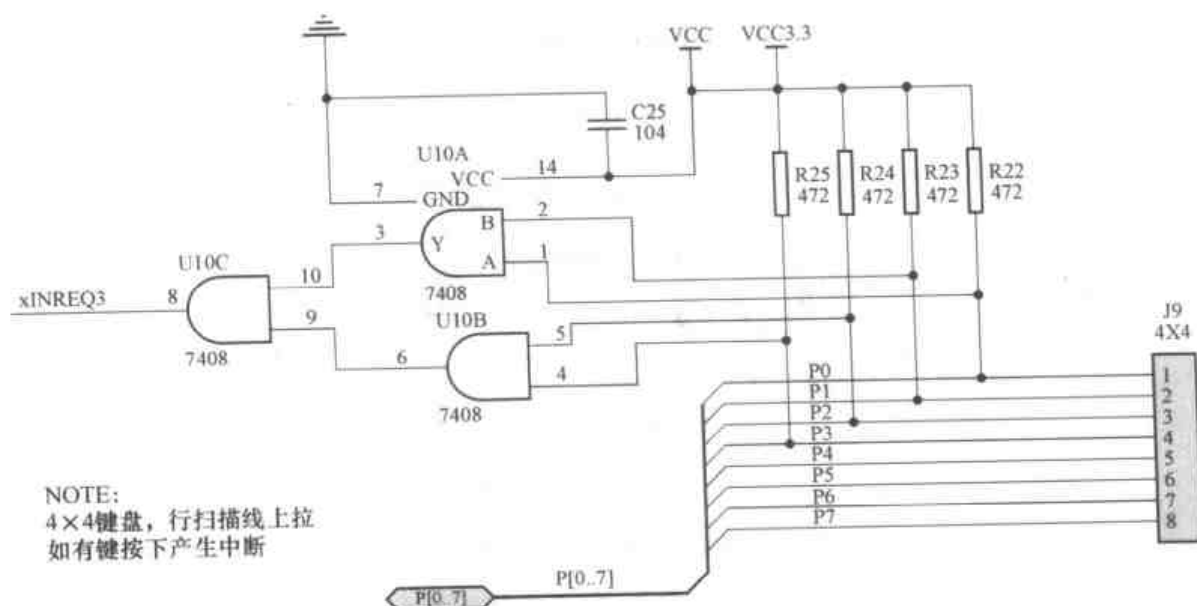


图 B-10 LCD 模块接口

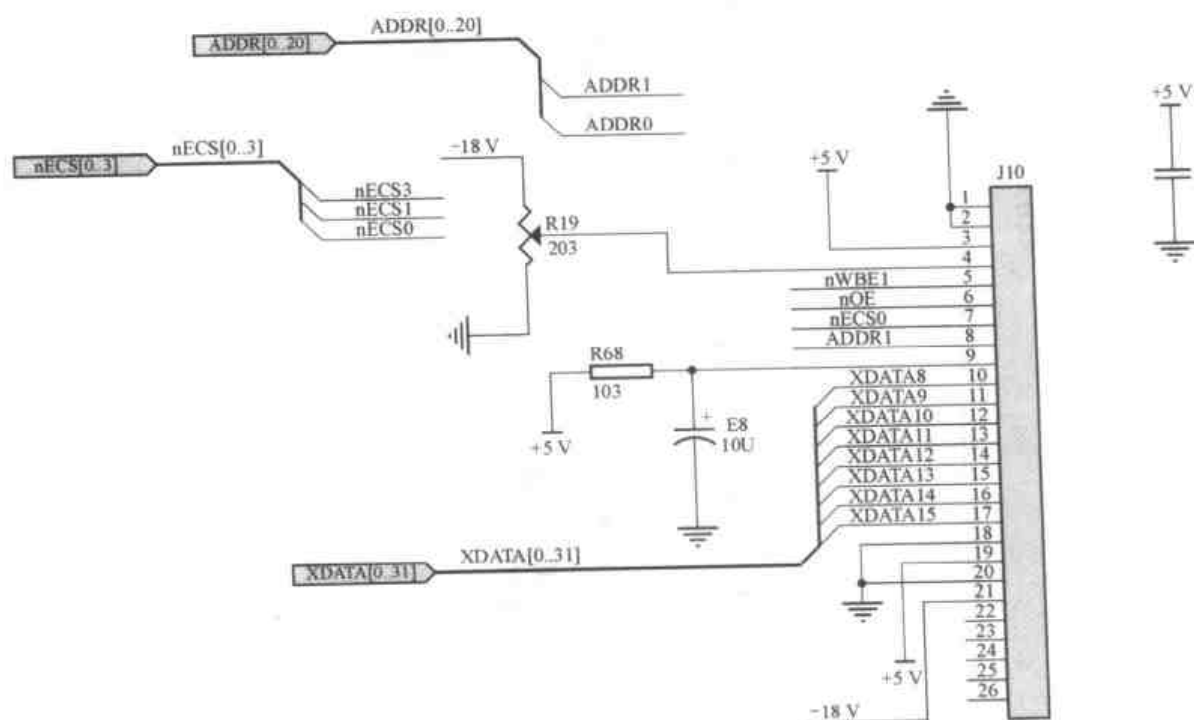


图 B-11 BOOT\_ROM 电路



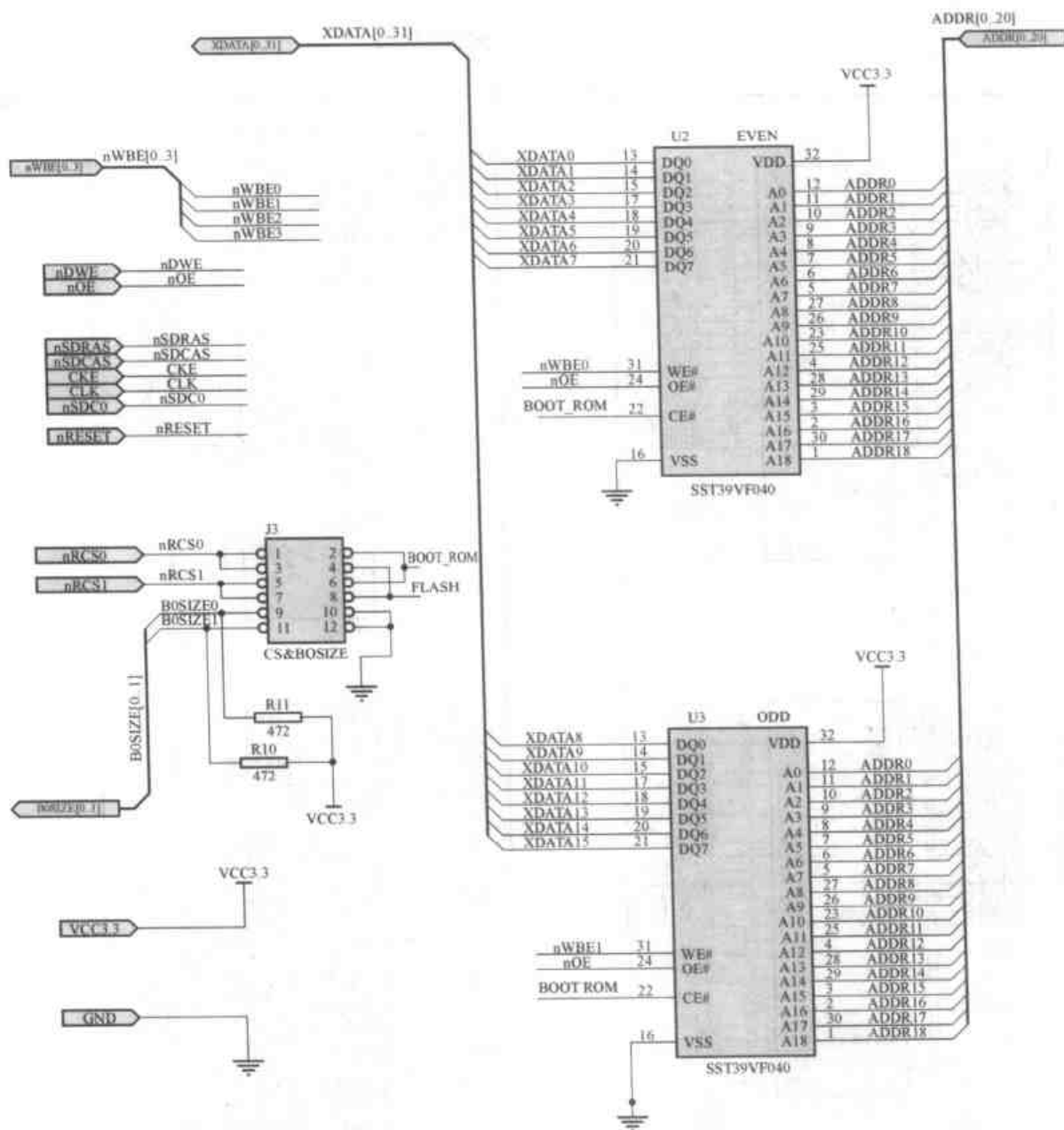


图 B-12 FLASH 与 SDRAM 电路

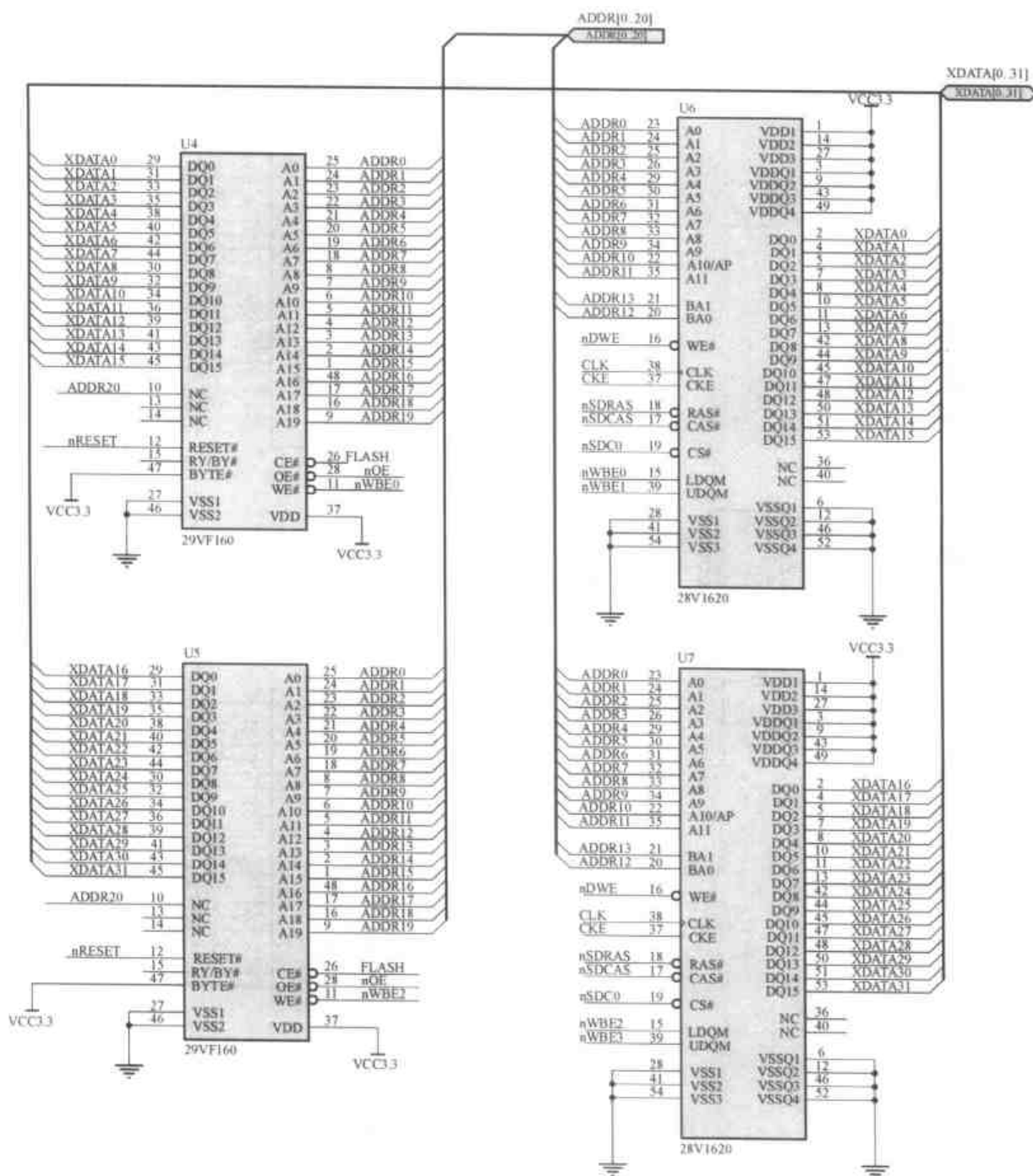


图 B-13 FLASH 与 SDRAM 电路

## 附录 C 标准 C 库函数

表 C-1 数学类库函数

| 函数名   | 函数类型                                                                                      | 功能                                                                                                                   | 返回值                                |
|-------|-------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------|------------------------------------|
| acos  | double acos( <i>x</i> );<br>double <i>x</i> ;                                             | 计算 $\arcsin x$ 的值                                                                                                    | 计算结果                               |
| asin  | double asin( <i>x</i> );<br>double <i>x</i> ;                                             | 计算 $\arccos x$ 的值                                                                                                    | 计算结果                               |
| atan  | double atan( <i>x</i> );<br>double <i>x</i> ;                                             | 计算 $\arctan x$ 的值                                                                                                    | 计算结果                               |
| cos   | double cos( <i>x</i> );<br>double <i>x</i> ;                                              | 计算 $\cos x$ 的值                                                                                                       | 计算结果                               |
| cosh  | double cosh( <i>x</i> );<br>double <i>x</i> ;                                             | 计算 <i>x</i> 的双曲余弦 $\cosh x$ 的值                                                                                       | 计算结果                               |
| exp   | double exp( <i>x</i> );<br>double <i>x</i> ;                                              | 求 $e^x$ 的值                                                                                                           | 计算结果                               |
| fabs  | double fabs( <i>x</i> );<br>double <i>x</i> ;                                             | 求 <i>x</i> 的绝对值                                                                                                      | 计算结果                               |
| floor | double floor( <i>x</i> );<br>double <i>x</i> ;                                            | 求出不大于 <i>x</i> 的最大整数                                                                                                 | 该整数的双精度实数                          |
| fmod  | double fmod( <i>x</i> , <i>y</i> );<br>double <i>x</i> , <i>y</i> ;                       | 求整数 <i>x</i> / <i>y</i> 的余数                                                                                          | 返回余数双精度数                           |
| frexp | double frexp( <i>val</i> , <i>eptr</i> );<br>double <i>val</i> ;<br>int * <i>eptr</i> ;   | 把双精度数 <i>val</i> 分解为数字部分(尾数) <i>x</i> 和以 2 为底的指数 <i>n</i> , 即 $val = x \times 2^n$ , <i>n</i> 存放在 <i>eptr</i> 指向的变量中 | 返回数字部分 <i>x</i> , $0.5 \leq x < 1$ |
| log   | double log( <i>x</i> );<br>double <i>x</i> ;                                              | 求 $\log_e x$ , 即 $\ln x$                                                                                             | 计算结果                               |
| log10 | double log10( <i>x</i> );<br>double <i>x</i> ;                                            | 求 $\lg x$                                                                                                            | 计算结果                               |
| modf  | double modf( <i>val</i> , <i>iptr</i> );<br>double <i>val</i> ;<br>double * <i>iptr</i> ; | 把双精度数 <i>val</i> 分解为整数部分和小数部分, 把整数部分存到 <i>iptr</i> 指向的单元                                                             | <i>val</i> 的小数部分                   |

续表

| 函数名  | 函数类型                                                               | 功能                                   | 返回值  |
|------|--------------------------------------------------------------------|--------------------------------------|------|
| pow  | double pow( <i>x</i> , <i>y</i> );<br>double <i>x</i> , <i>y</i> ; | 计算 $x^y$ 的值                          | 计算结果 |
| sin  | double sin( <i>x</i> );<br>double <i>x</i> ;                       | 计算 sin <i>x</i> 的值                   | 计算结果 |
| sinh | double sinh( <i>x</i> );<br>double <i>x</i> ;                      | 计算 <i>x</i> 的双曲正弦函数 sinh <i>x</i> 的值 | 计算结果 |
| sqrt | double sqrt( <i>x</i> );<br>double <i>x</i> ;                      | 计算根号下 <i>x</i> 的值                    | 计算结果 |
| tan  | double tan( <i>x</i> );<br>double <i>x</i> ;                       | 计算 tan <i>x</i> 的值                   | 计算结果 |
| tanh | double tanh( <i>x</i> );<br>double <i>x</i> ;                      | 计算 <i>x</i> 的双曲正切函数 tanh <i>x</i> 的值 | 计算结果 |

表 C-2 字符函数和字符串函数

| 函数名     | 函数类型                                                                                | 功能                                                           | 返回值                                                                                                  |
|---------|-------------------------------------------------------------------------------------|--------------------------------------------------------------|------------------------------------------------------------------------------------------------------|
| isdigit | int isdigit( <i>ch</i> ); int <i>ch</i> ;                                           | 检查 <i>ch</i> 是否是数字 0~9                                       | 是, 返回 1; 不是, 返回 0                                                                                    |
| islower | int islower( <i>ch</i> ); int <i>ch</i> ;                                           | 检查 <i>ch</i> 是否小写字母 a~z                                      | 是, 返回 1; 不是, 返回 0                                                                                    |
| isupper | int isupper( <i>ch</i> ); int <i>ch</i> ;                                           | 检查 <i>ch</i> 是否大写字母 A~Z                                      | 是, 返回 1; 不是, 返回 0                                                                                    |
| strcat  | char * strcat( <i>str1</i> , <i>str2</i> );<br>char * <i>str1</i> , * <i>str2</i> ; | 把字符串 <i>str2</i> 接到 <i>str1</i> 的后面, <i>str1</i> 最后的 '\0' 去掉 | <i>str1</i>                                                                                          |
| strcmp  | int strcmp( <i>str1</i> , <i>str2</i> ); char<br>* <i>str1</i> , * <i>str2</i> ;    | 比较两个字符串                                                      | <i>str1</i> < <i>str2</i> 返回负数;<br><i>str1</i> = <i>str2</i> 返回 0;<br><i>str1</i> > <i>str2</i> 返回正数 |
| strcpy  | char * strcpy( <i>str1</i> , <i>str2</i> );<br>char * <i>str1</i> , * <i>str2</i> ; | 拷贝 <i>str2</i> 串到 <i>str1</i> 中                              | <i>str1</i>                                                                                          |
| tolower | int tolower( <i>ch</i> );                                                           | 将 <i>ch</i> 字符转换为小写字符                                        | 返回小写字母                                                                                               |
| toupper | int toupper( <i>ch</i> );                                                           | 将 <i>ch</i> 字符转换为大写字符                                        | 返回大写字符                                                                                               |

表 C-3 输入输出函数

| 函数名            | 函数类型                                                                                                             | 功能                                                           | 返回值                          |
|----------------|------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------|------------------------------|
| fclose         | int fclose(fp); FILE * fp;                                                                                       | 关闭 fp 所指文件, 释放文件缓冲区                                          | 有错, 返回非 0; 无错, 返回 0          |
| fgetc          | int fgetc(fp); FILE * fp;                                                                                        | 从 fp 所指文件读取一个字符                                              | 无错, 返回所得字符, 有错, 返回 EOF       |
| fgets          | int fgets(buf, n, fp); FILE * fp;                                                                                | 从 fp 所指文件读取一个长度为 (n - 1) 的字符串, 存入 buf                        | 返回地址 buf, 若遇文件结束或出错, 返回 NULL |
| fopen          | FILE * fopen ( filename, mode); char * filename, * mode;                                                         | 以 mode 方式打开 filename 文件                                      | 成功, 返回一个文件指针                 |
| fprintf        | int fprintf ( fp, format, arg_list); FILE * fp; char * format;                                                   | 把 arg_list 的值以 format 指定的格式输出到文件中                            | 输出字符的个数                      |
| fputc          | int fputc(ch, fp); char ch;                                                                                      | 将字符 ch 输出到 fp 所指的文件                                          | 成功, 返回该字符, 失败, 返回 0          |
| fputs/fread    | int fputs (str, fp); char * str, FILE * fp; int fread(buf, size, n, fp); char * buf; int size; int n; FILE * fp; | 将字符串 str 写到 fp 所指文件<br>从 fp 指向的文件读取 n 个长度为 size 的数据项, 存到 buf | 成功, 返回 0 失败, 返回非 0           |
| fscanf         | int fscanf ( fp, format, arg_list); FILE * fp; char * format;                                                    | 将 fp 所指的文件按 format 给定的格式输入数据到 arg_list 所指的内容                 | 返回实际读取的数据项个数。读到文件结束或出错返回 0   |
| fwrite/getchar | int fwrite ( buf, size, n, fp ); char * buf; int size; int n; FILE * fp; int getchar;                            | 将 buf 所指向的 n 个 size 字节输出到 fp 所指文件, 从键盘输入一个字符                 | 实际写入的数据项的个数                  |

续 表

| 函数名     | 函数类型                                             | 功能                                    | 返回值                                    |
|---------|--------------------------------------------------|---------------------------------------|----------------------------------------|
| printf  | int printf (format, arg_list);<br>char * format; | 将输出项 arg_list 的值输出到标准输出设备上            | 所读字符输出字符的个数                            |
| putchar | int putchar(ch); char ch;                        | 输出字符 ch 到标准输出设备                       | 输出字符 ch                                |
| puts    | int puts(str); char * str;                       | 输出字符串 str 到标准输出设备                     | 成功, 返回换行符; 失败, 返回 EOF                  |
| scanf   | int scanf (format, arg_list);<br>char * format;  | 从标准输入设备按 format 格式输入数据到 arg_list 所指内存 | 读入 arg_list 数据个数。遇文件结束, 返回 EOF, 出错返回 0 |

表 C-4 动态存储分配函数

| 函数名    | 函数类型和形参类型                              | 功能             | 返回值                     |
|--------|----------------------------------------|----------------|-------------------------|
| free   | void free(p);<br>void * p;             | 释放 p 所占的内存区    |                         |
| malloc | void * malloc(size);<br>unsigned size; | 分配 size 字节的存储区 | 被分配的内存区的地址, 如内存不够, 返回 0 |

## 附录 D WindShell 常用命令

表 D-1 WindShell 系统调试命令

| 指令名称               | 指令解释                    |
|--------------------|-------------------------|
| ld( )              | 装载程序模块到目标机中,并且动态链接,开始运行 |
| undl( )            | Ld( )的反过程               |
| m( )               | 调整内存宽度                  |
| mRegs( )           | 调整寄存器值                  |
| b( )               | 设置和显示断点                 |
| bb( )              | 设置一个硬断点                 |
| s( )               | 单步执行指令到下一条              |
| so( )              | 单步                      |
| c( )               | 从断点处继续                  |
| cret( )            | 直到当前的子程序返回,然后继续         |
| bdall( )           | 删除所有断点                  |
| bd( )              | 删除一个断点                  |
| reboot( )          | 重新启动                    |
| bootChange( )      | 调整启动参数值                 |
| sysSuspend( )      | 进入系统模式                  |
| sysResume( )       | 从系统模式返回到任务模式            |
| agentModeShow( )   | 显示代理模式                  |
| sysStatusShow( )   | 显示系统上下文状态               |
| quit( ) or exit( ) | 退出 Shell                |

表 D-2 WindShell 中显示指令

| 指令名称                  | 指令解释                    |
|-----------------------|-------------------------|
| show( )               | 打印相关信息                  |
| browse( )             | 在 Tornado 浏览器中显示一个指定的目标 |
| classShow( )          | 显示一类 VxWorks 内核信息       |
| taskShow( )           | 先是任务 TCB 信息             |
| taskCreateHookShow( ) | 显示任务创建函数列表              |
| taskDeleteHookShow( ) | 显示任务删除函数列表              |

续表

| 指令名称                  | 指令解释            |
|-----------------------|-----------------|
| taskRegsShow( )       | 显示任务寄存器的内容      |
| taskSwitchHookShow( ) | 显示任务状态切换内容列表    |
| taskWaitShow( )       | 显示挂起的任务信息       |
| semShow( )            | 显示信号量信息         |
| semPxShow( )          | 显示 POSIX 信号量信息  |
| wdShow( )             | 显示一个看门狗定时器信息    |
| msgQShow( )           | 显示消息队列信息        |
| mqPxShow( )           | 显示 POSIX 消息队列信息 |
| iosDrvShow( )         | 显示系统驱动列表        |
| iosDevShow( )         | 显示系统设备列表        |

表 D-3 WindShell 中网络状态显示指令

| 指令名称             | 指令解释                                             |
|------------------|--------------------------------------------------|
| hostShow( )      | 显示主机表                                            |
| icmpstatShow( )  | 显示 ICMP (Internet Control Message Protocol) 统计信息 |
| ifShow( )        | 显示网络接口信息                                         |
| inetstatShow( )  | 显示所有活动的 Internet 协议套接字连接信息                       |
| ipstatShow( )    | 显示 ip 统计信息                                       |
| routeStatShow( ) | 显示路由统计信息                                         |
| tcpstatShow( )   | 显示 TCP 协议的统计信息                                   |
| tftpInfoShow( )  | 得到 tftp 状态信息                                     |
| udpstatShow( )   | 显示 UDP 协议的统计信息                                   |



## 附录 E 串口调试助手使用说明

使用平台: WIN9X/NT/2000/XP

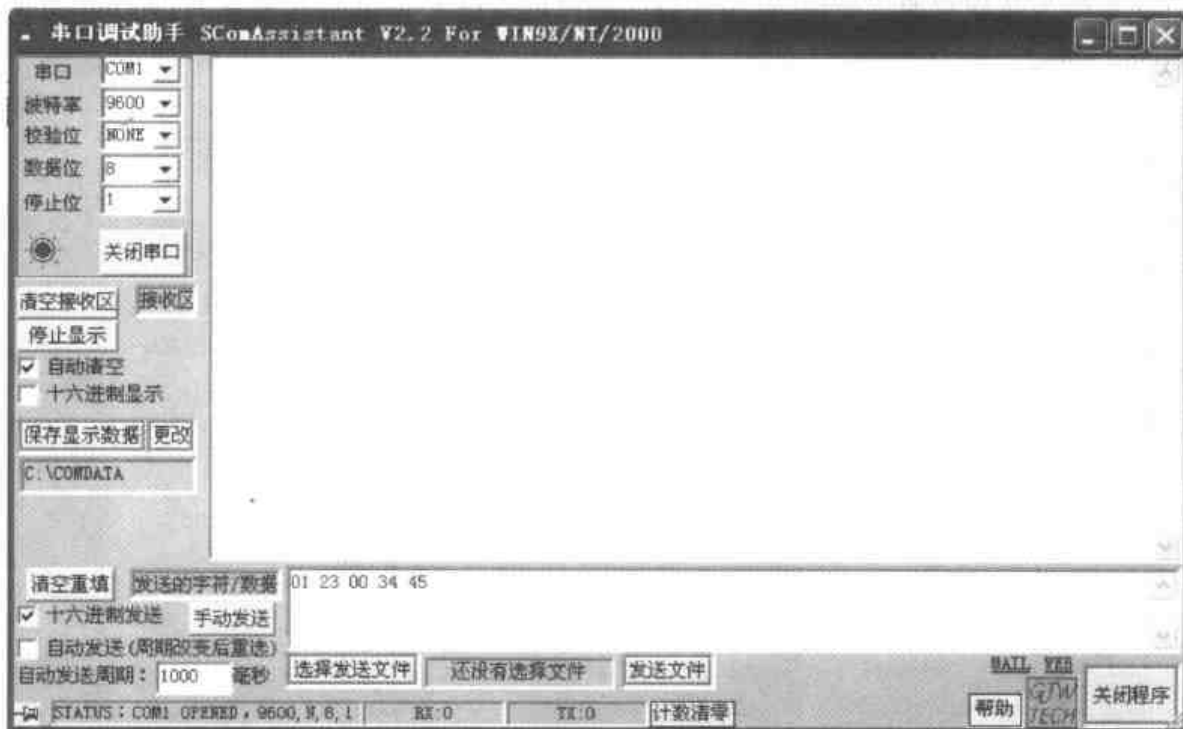


图 E-1 串口调试助手

使用说明:

如图 E-1 所示,软件左上角为串口参数设置,分别为串口号、波特率、校验位、数据位、停止位。通信必须正确设置这些参数,否则没有显示或者显示乱码。

显示的时候主要设置“十六进制显示”和“自动清空”选项,如图 E-1 所示,一般都将其选上。显示的结果可以通过点击“保存显示数据”按钮进行保存。

右上角大片的文本框为显示接收数据的区域,下方的文本框为填写发送数据的区域,如图 E-1 所示,将要发送的 01 23 00 34 45 数据填写到发送文本框中。发送时可以选择以十六进制形式发送,还可以选择“自动发送”方式。选择“自动发送”方式需要设置“自动发送周期”,如图中设置为 1 000 ms。发送的数据可以手填,也可以通过点击“发送文件”选择相应的数据文本进行发送。

另外,可以使用窗口悬浮功能:点击程序左下角的针状按钮可以使程序置于最上层,保持可见;使用放大至全屏功能:当需要扩大接收窗口以方便观看数据时,可以点击右上角最大化按钮。

注意事项:

串口调试助手仅供三线制串口调试使用,不可调试 Modem 等多线制通信。

另外还要注意的是调试串口时,插拨串口接头应尽量关闭计算机,至少保证一端是关闭的。

此软件为龚建伟博士所编,其他特殊问题可以通过 E-mail 与作者联系:webmaster@gjwtech.com。