嵌入式实时多任务操作系统培训教材

（第四版）

北京麦克泰软件技术有限公司
电子科技大学嵌入软件中心
www.taikebj.com.cn
1999 年 11 月

应用基础篇之一：VRTX 实时操作系统

第一章　嵌入式实时系统基础

1.1 实时系统

随着计算机的发展和应用的普及，实时计算机系统(以下简称实时系统)已经在工业、交通、能源、科学研究和科学试验、国防等各个领域发挥极其重要的作用。

实时，表示“立即”、“及时”。关于实时性，人们往往有不尽相同的理解和解释。一般将联机系统视作实时系统，也有人把人－机交互性的系统称为实时系统。当然，它们都是计算机发展到一定阶段的产物。

实时系统是对外来事件在限定时间内能做出反应的系统。限定时间的范围很广可以从微秒级(如信号处理)到分级(如联机查询系统)。

实时控制系统和实时信息处理系统统称为实时系统。在实时控制系统中计算机通过特定的外围设备与被控对象发生联系，被控对象的信息经加工后，通过显示屏幕向控制人员显示或通过外设向被控对象发出指示，实现对被控对象的控制；在实时信息处理系统中，用户通过终端设备向系统提出服务请求，系统完成服务后通过终端回答给用户。

在实时系统中主要有三个指标来衡量系统的实时性，响应时间(Response Time)、生存时间(Survival Time)、吞吐量(Throughput)。

响应时间(Response Time)：是计算机识别一个外部事件到作出响应的时间，在控制应用中它是最重要的指标，如果事件不能及时的处理，系统可能就会崩溃。对于不同的过程，有不同的响应时间要求。对于有些慢变化过程，具有几分钟甚至更长的响应时间都可以认为是实时的。对于快速过程，其响应时间可能要求达到毫秒、微秒、毫微秒级甚至更短。因此，实时性不能单纯从绝对的响应时间长短上来衡量，应当根据不同的对象，在相对意义上进行评价。

生存时间(Survival Time)：是数据有效等待时间，在这段时间里数据是有效

的。

吞吐量(Throughput): 是在一给定时间内，系统可以处理的事件总数。例如通讯控制器用每秒钟处理的字符数来表示吞吐量，吞吐量可能是平均响应时间的倒数，但它通常要小一些，因为在每次响应后可能需要一段时间进行清理(clean up)，这段时间就称为恢复时间(recovery time)。

实时系统强调的是实时性和可靠性，这两方面除了与计算机硬件有关 (如 CPU 的速度，访问存贮器的速度等)外，还与实时系统的软件密切相关。硬件是实时的，而软件往往不一定是实时的。

如何实现实时的应用系统呢？ 可以通过以下的途径：

（1） 使用硬件的功能；

（2） 微处理器的中断机制；

（3） 简单的单线程循环程序；

（4） 基于实时操作系统的复杂多线程程序。

这种实时系统的软件是实时应用软件和实时操作系统 RTOS (Real-Time Operating System)两部分的有机结合，其中 RTOS 起着核心作用，由它来管理和协调各项工作，为应用软件提供良好的运行软件环境及开发环境。

1.2 实时系统的典型应用及特点

实时应用的范围很广，主要有两种应用：嵌入式应用和一般应用

1.2.1 嵌入式应用

大多数实时系统都是嵌入式应用(Embedded Applications)系统，

嵌入式计算机是一种智能部件内装于专用设备/系统的高速计算机。它的主要功能是作为一个大型工程系统中的信息处理部件，来控制专门的硬件设备的。

这种嵌入式系统自动化程度高、威力大、反应速度快。 用户不需知道装置内计算机的存在，一般不能被用户编程，它有一些专用的 I/O 设备，对用户的接口是应用专用的。

嵌入式计算机系统广泛地用于办公自动化、消费、通信、汽车、工业和军事领域，其中，通信，办公自动化、消费电子领域占整的份额最大，约 90%以上。嵌入式的典型应用有：

1.**过程控制**(process control)

即对生产过程中各种动作流程的控制，这种控制是在对被控对象和环境进行不断观测的基础上作出及时的、恰当的反应。在控制过程中，计算机扮演着中心的角色。它通过传感器从外部接收有关过程的信息，对这些信息进行加工处理，然后对执行机构发出控制指令。

2.**通讯设备**(Telecommunication):

如程控交换机，路由器，BB **机，大哥大，桥接器，集线器，**Modem **等。**

3. **智能仪器**(Intelligent Instrument):**如示波器，医疗仪器等。**

4. **消费产品**(Consumer Products):**如洗衣机，微波炉，电视机，游戏机等。**

5. **机器人**(Robots)

6. **计算机外设设备**(Computer Peripherals): **打印机，终端，磁盘驱动器。**

7. **军事电子设备和现代武器**: **如雷达、电子对抗，坦克、战机、战舰等。**

1.2.2 一般应用(General-Purpose Application)

一般应用的计算机系统对用户来说是可见的，如 PC **机、工程工作站等。它具有标准的计算机外设，如键盘、显示器、磁盘等，它的软件具有通用的人机接口，其应用程序可按用户需要随时改变，即重新编制。它通常用于开发，数据处理等，其典型应用如下：**

1. **测控计算机**:

在大型控制系统中，它通常与几个嵌入式计算机相连，作为它们的上位机，进行系统的总控，协调，数据存储等工作。

2. **交互式系统**

实时信息查询系统，如飞机定票系统、银行交易和股票交易系统等，这些系统的响应时间要求不高，只要人可以忍受就行了。

1.3 嵌入式实时系统软件的基本特征

不难看出，与一般的计算机应用相比，嵌入式实时应用系统是具有高速处理、配置专一、结构紧凑和坚固可靠等特点的实时系统，相应的软件系统应是一种别有特色、要求更高的实时软件。对这种实时软件的主要要求是：

1．**实时性**

实时软件对外部事件作出反应的时间必须要快，在某些情况下还需要是确定的、可重复实现的,不管当时系统内部状态如何，都是可预测的(predictable)。

2．有处理异步并发事件的能力

实际环境中，嵌入式实时系统处理的外部事件往往不是单一的，这些事件往往同时出现，而且发生的时刻也是随机的，即异步的。实时软件应有能力对这类外部事件组有效地进行处理。

3．快速启动、并有出错处理和自动复位功能

这一要求对机动性强、环境复杂的智能系统显得特别重要，快速机动的环境，不允许控制软件临时从盘上装入，因此嵌入式实时软件需事先固化到只读存贮器，开机自行启动，并在运行出错死机时能自动恢复先前运行状态。因此嵌入式实时软件应采用特殊的容错、出错处理措施。

4. 嵌入式实时软件是应用程序和操作系统两种软件的一体化程序。

对于通用计算机系统，例如 PC 机、工作站，操作系统等系统软件和应用软件之间界限分明。换句话说，统一配置的操作系统环境下，应用程序是独立的运行软件，可以分别装入执行。但是，在嵌入式实时系统中，这一界限并不明显。这是因为，应用系统配置差别较大，所需操作系统繁简不一，IO 操作也不标准，这部分驱动软件常常由应用程序提供。这就要求采用不同配置的操作系统和应用程序，链接装配成统一的运行软件系统。也就是说，在系统总设计目标指导下将它们综合加以考虑、设计与实现。

5. 嵌入式实时软件的开发需要独立的开发平台

由于嵌入式实时应用系统的软件开发受到时间、空间开销的限制，常常需要在专门的开发平台上进行软件的交叉开发，其交叉开发环境如图 1-1 所示。

图 1-1 交叉开发环境

开发平台称为宿主机，应用系统称作目标机。宿主机可以是与目标机相同或不相同的机型。这种不同机型的开发平台又称作交叉式开发系统。显然，在这种独立的实时软件开发系统上，应配备完整的实时软件开发的工具，如高级语

言、在线调试器和在线仿真器等。因此，嵌入式实时软件开发过程较为复杂。

1.4 嵌入式实时系统的分类

可按速度: 系统响应时间(Response time)或吞吐量(Throughput)、确定性及软件结构分类:

1.4.1 速度分类

按实时性的强弱(即根据系统响应时间的长短)可将嵌入式实时系统大致分为以下几种:

1.强实时系统，其系统响应时间在毫秒或微秒级;

2.一般实时系统，其系统响应时间在几秒的数量级上,其实时性的要求比强实时系统要差一些。

3.弱实时系统，其系统响应时间约为数十秒或更长。这种系统的响应时间可能随系统负载的轻重而变化，即负载轻时系统响应时间可能较短，实时性好一些，反之系统响应时间可能加长。

1.4.2 确定性

按确定性来分，可分为硬实时和软实时：

1.硬实时

系统对系统响应时间有严格的要求，如果系统响应时间不能满足，就要引起系统崩溃或致命的错误。

2.软实时

系统对系统响应时间有要求，但是如果系统响应时间不能满足，不会导致系统出现致命的错误或崩溃。

1.4.3 软件结构分类

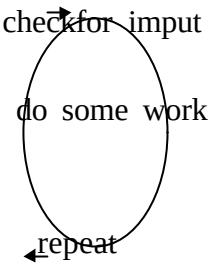
1 单线程程序（Single-threaded program）

也称为顺序程序（Sequential program），它分为两种：

（1）循环轮询系统:(Polling Loop)

最简单的软件结构是循环轮询，程序依次检查系统的每一个输入条件,一旦条件成立就进行相应的处理。其通常的软件结构如下：

```
initialize()
  while(true)  {
if (condition_1) action_1();
if (condition_2) action_2();
                .....
if (condition_n) acition_n();
    }
```



优点：

- **对于简单的系统而言，便于编程和理解**
- **没有中断的机制，程序运行良好，不会出现随机的问题**

缺点：

- **有限的应用领域（由于不可确定性）**
- **对于大量的** I/O **服务的应用，不容易实现**
- **大的程序不便于调试**

因此，它适合于慢速和非常快速的简单系统。

2. **事件驱动系统**:(Event-Driven system)

事件驱动系统是能对外部事件直接响应的系统。它包括前后台、实时多任务、多处理器三个系统。是嵌入式实时系统的主要形式。

(1).**前后台系统**(Foreground/Background)**又叫中断驱动系统。**

后台是一个循环轮询系统一直在运行，前台是由一些中断处理过程组成的。当有一前台事件(外部事件)发生时，引起中断，中断后台运行进行前台处理，处理完成后又回到后台(通常又称主程序)。

这种系统的一个极端情况是，后台只是一个简单的循环不做任何事情，所有其它工作都是由中断处理程序完成的。但大多数情况是中断只处理那些需要快速响应的事件，并且把 I/O **设备的数据放到内存的缓冲区中，再向后台发信号，其它的工作由后台来完成，如对这些数据进行处理，存储，显示，打印等。**

这种系统的软件运行的方式如图 1-2 **所示**:

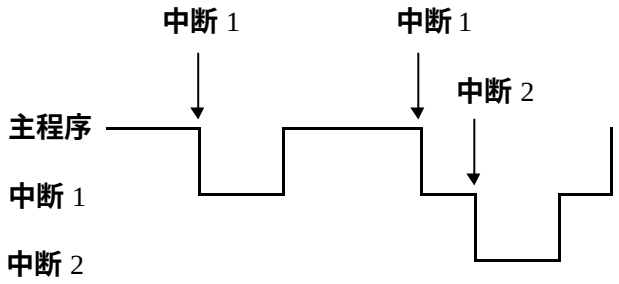


图 1-2　前后台系统运行方式

这种系统需要考虑的是中断的现场保护和恢复，中断嵌套，中断处理过程与主程序的协调(共享资源)问题。系统的性能主要由中断延迟时间(Interrupt latency time)，**响应时间**(response time)**和恢复时间**(recovery time)**来刻画，如图** 1-2 **所示：**

中断延迟时间：**是指从中断发生到系统获知中断，并且开始执行中断服务程序所需要的最大滞后时间。**

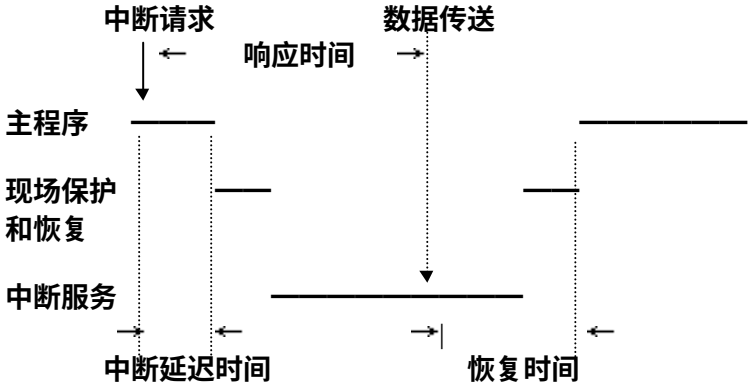
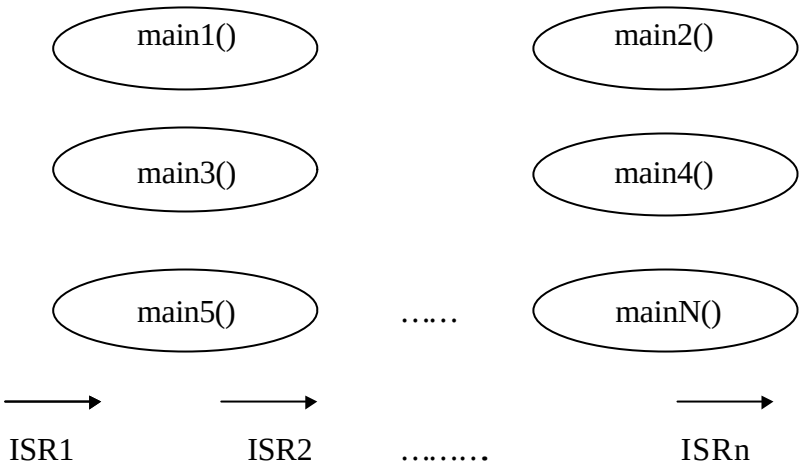


图 1-3　中断响应的时间刻画

(2). **实时多任务系统**(Multitasking **或** Multi-thread Program Model-**简称** RTOS)

对于一个复杂的嵌入式实时系统来说，当采用中断处理程序加一个后台主程序这种软件结构难以实时的、准确的、可靠的完成时，或存在一些互不相关的过程需要在一个计算机中同时处理时，就需要采用实时多任务系统。



其主要特点：

- **多个顺序执行的程序并行运行**
- **宏观上看，所有的程序同时运行，每个程序运行在自己独立的** CPU **上**
- **实际上，不同的程序是共享同一个** CPU **和其它硬件。因此，需要** RTOS **来对这些共享的设备和数据进行管理**
- **每个程序都被编制成无限循环的程序，等待特定的输入，执行相应的任务等。**
- **这种程序模型将系统分成相对简单的，相互合作的模块。**

其主要优点：

- **将复杂的系统分解为相对独立的多个线程， 达到“分而制之”的目的，从而降低系统的复杂性。**
- **保证系统的实时性**
- **系统的模块化好，提高系统的可维护性。**

缺点：

- **需要采用一些新的软件软件设计方法**
- **需要增加功能：线程间的协调，同步和通信功能**
- **需要对每一个共享资源互斥**
- **导致线程间的竞争**
- **需要使用** RTOS，RTOS **要增加系统的开销**

实时多任务系统实际上是由多个任务，多个中断处理过程，实时操作系统组

成的有机的整体。每个任务是顺序执行的，并行性通过操作系统来完成，任务间的相互通信和同步也需要操作系统的支持。

RTOS 的需求：

- 足够的快（上下文切换和系统调用等）
- 可确定的性能
- 任务调度机制是基于优先级的
- 最小的中断延迟
- 可伸缩、可配置的体系结构
- 可靠、健壮

实时多任务系统的实现必须有实时多任务操作系统的支持，操作系统主要完成任务切换，任务调度，任务间通信、同步、互斥，实时时钟管理，中断管理。

(3) 多处理机系统

当有些工作用单台计算机来处理是难以完成时，就需要增加另外的计算机，这就是多处理器系统的由来。在单处理机系统中，多个任务在宏观上看是并发的，但在微观上看实际是顺序执行的；在多处理机系统中，多个任务可以分别在不同的处理机上执行，宏观上看是并发的，微观上看也是并发的。前者称为伪并发性，后者称为真并发性。

按照多处理机的结构，多处理机系统分为紧耦合系统(tightly-coupled system)和松耦合系统(loosely-coupled system)两种。紧耦合系统是多个处理器通过共享内存空间来交换信息的系统(如：SMP)，而松耦合系统多个处理器多是通过通讯线路来连接的，通过它来交换信息。

1.5 嵌入式实时操作系统及发展

1.5.1 嵌入式实时操作系统

在计算机技术发展的初期阶段，计算机系统中没有操作系统这个概念。为了给用户提供一个与计算机之间的接口，同时提高计算机的资源利用率，便出现了计算机监控程序(Monitor),使用户能通过监控程序来使用计算机。随着计算机技术的发展，计算机系统的硬件、软件资源也愈来愈丰富，监控程序已不能适应计

算机应用的要求。于是在六十年代中期，监控程序又进一步发展，形成了操作系统(Operating System)。 发展到现在，广泛使用的有三种操作系统即多道批处理操作系统、分时操作系统以及实时操作系统。

多道批量处理系统一般用于计算中心较大的计算机系统中。由于它的硬件设备比较全，价格较高，所以此类系统十分注意 CPU 及其它设备的充分利用，追求高的吞吐量，不具备实时性。

分时系统的主要目的是让多个计算机用户能共享系统的资源，能及时地响应和服务于联机用户，只具有很弱的实时功能，但与真正的实时操作系统仍然有明显的区别。

那么什么样的操作系统才能称为实时操作系统呢? IEEE 的实时 UNIX 分委会认为实时操作系统应具备以下的几点:

1.异步的事件响应

实时系统为能在系统要求的时间内响应异步的外部事件，要求有异步 I/O 和中断处理能力。I/O 响应时间常受内存访问、盘访问和处理机总线速度所限制。

2.切换时间和中断延迟时间确定

3.优先级中断和调度

必须允许用户定义中断优先级和被调度的任务优先级并指定如何服务中断。

4.抢占式调度

为保证响应时间，实时操作系统必须允许高优先级任务一旦准备好运行，马上抢占低优先级任务的执行。

5.内存锁定

必须具有将程序或部分程序锁定在内存的能力，锁定在内存的程序减少了为获取该程序而访问盘的时间，从而保证了快速的响应时间

6.连续文件

应提供存取盘上数据的优化方法，使得存取数据时查找时间最少。通常要求把数据存储在连续文件上。

7.同步

提供同步和协调共享数据使用和时间执行的手段。

总的来说实时操作系统是事件驱动的(event_driven),能对来自外界的作用和信号在限定的时间范围内作出响应。它强调的是实时性、可靠性和灵活性，与实时应用软件相结合成为有机的整体，起着核心作用，由它来管理和协调各项工作，

为应用软件提供良好的运行软件环境及开发环境。

从实时系统的应用特点来看，实时操作系统可以分为两种：一般实时操作系统和嵌埋式实时操作系统。

一般实时操作系统与嵌入式实时操作系统都是具有实时性的操作系统，它们的主要区别在于应用场合和开发过程。

一般实时操作系统应用于实时处理系统的上位机和实时查询系统等实时性较弱的实时系统，并且提供了开发、调试、运用一致的环境。

嵌入式实时操作系统应用于实时性要求高的实时控制系统，而且应用程序的开发过程是通过交叉开发来完成的，即开发环境与运行环境是不一致的。嵌入式实时操作系统具有规模小(一般在几 K-几十 K 内)、可固化使用、实时性强(在毫秒或微秒数量级上)的特点。

1.5.2 嵌入式实时操作系统的发展

近十年来，嵌入式实时操作系统得到飞速的发展，从支持 8 位微处理器到 16 位、32 位甚至 64 位，从支持单一品种的微处理器芯片到支持多品种微处理器芯片，从只有实时内核到除了内核外还提供其他功能模块如:文件系统，TCP/IP 网络系统，GUI 图型系统等。

据嵌入式系统杂志(Embedded Systems Programming)的最新报告，世界各国有四十多家公司，已成功的推出了百余种可供嵌入式应用的实时操作系统。其中几个著名的操作系统是：Microtec（现 Mentor）公司的 VRTX，Integrated System 公司(ISI)的 pSOS， Wind River 公司的 VxWorks 等 ，这些操作系统适用于实时、多任务应用环境，而且还具有相应的功能齐全的交叉开发环境如 Microtec 的 XRAY。微软的 WINCE 也是嵌入式实时操作系统.

目前，嵌入式实时操作系统及其应用开发环境的发展动向是：

1. 嵌入式实时操作系统正向实时超微内核(Nanokernel)、开放发展

八十年代后期，国外提出了微内核(Microkernel)的思想， 即将传统操作系统中的许多共性的东西抽象出来，构成操作系统的公共基础，即微内核，真正具体的操作系统功能则由构造在微内核之外的服务器实现。这是一种机制与策略分离的开放式设计思路。

近几年，国外发展了一种基于微内核思想设计的精巧的嵌入式微内核，即实时超微内核(Nanokernel)。超微内核是一种非常紧凑的基本内核代码层，为嵌入

式应用提供了可抢占，快而确定的实时服务，在它的基础上可以灵活地构造各种类型的、与现成系统兼容的、可伸缩的嵌入式实时操作系统。因此能满足应用代码的可重用和可伸缩性(scalability)的需求。Microtec 已首先推出了基于实时超微内核的嵌入式实时操作系统 VRTXsa，它与 VRTX32 兼容，并具有更强的功能，实时性和可靠性有了很大的改进。

2 开发环境向开放的、集成化的方向发展

由于嵌入式应用软件的特殊性，往往要求应用程序设计者具有一定的实时操作系统的专门知识，能合理地划分任务，合理的配置系统以及目标联机的调试。因此，要设计实现一个高性能的实时应用软件，需要强有力的交叉开发工具系统的支持。国外十分重视发展与实时操作系统配合的嵌入式应用的集成开发环境，现已发展到第三代，它以客户－服务器的系统结构为基础，具有运行系统的无关性、连接的无关性、开放的软件接口(与嵌入式实时操作系统，与开发工具，与目标环境的接口)、环境的一致性、宿主机上的目标仿真的特点。

1993 年，MICROTEC 推出了世界上最先进的第三代嵌入式集成交叉开发系统 Spectra(现称为 VRTX 开发系统)。该系统可在 UNIX 及 WINDOWS NT 上建立起开放的、网络环境的交叉开发平台，能将多来源的开发工具有机地结成一体，对复杂的嵌入式应用开发提供全过程支持。

3．完整的解决方案

RTOS 厂家本身或与其他软件公司和半导体公司配合为典型应用提纲解决方案 如：PDA，机顶盒，路由器等

综上所述，嵌入式实时操作系统及其应用开发环境正向开放、集成发展。

第二章　实时多任务软件的设计方法

2.1 设计方法及步骤

1.需求分析(Requirement specification)

分析用户需求，定义系统的各功能块、输入/输出及系统的性能指标。

2.数据流分析(Data flow analysis)

进行数据流的分析，设计出各功能块间的数据流图。

3.分解任务(Decompostion into tasks)

在识别出并行性后，进行任务分解，将系统分解成主要的任务。

4. 定义任务间接口(Definiton of task interfaces)

定义任务间，任务与中断处理程序间的同步、通信和互斥关系。

5.任务级的设计

按模块方式设计每个任务，并定义出模块间接口；

6.模块构筑

完成每个模块的详细设计、编码和单元测试；

7.任务与系统集成

逐个模块连接、测试以构成任务，逐个任务连接和测试形成最终系统；

8.系统测试

测试整个系统或主要子系统，以验证功能指标的实现，为具有更强的客观性，系统测试最好由一个独立的测试小组执行。

2.2 如何划分任务

1．任务划分的原则

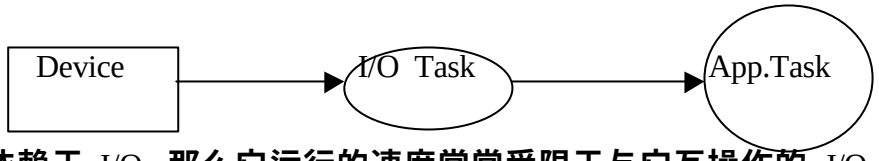
在将一个软件系统分解成并行任务时，主要需考虑的是：系统内功能的异步性。

分析数据流图中的变换，确定哪些变换可以并行，而哪些变换在本质上是顺序的,通过这种方法，划分出任务：一个变换对应一个任务,或者一个任务包括几个变换。

一个变换是应该成为一个独立的任务，还是应该和其它变换一起组成一个任务，决定的原则如下：

- I/O 依赖性 (Dependency on Input/Output Device)
- 时间关键性的功能 (Time-critical functions-Hard Deadline)
- 计算量大的功能（Heavy Computation function）
- 功能内聚（Functional relations）
- 时间内聚（Temporal relations）
- 周期执行的功能（Cyclic executing function）

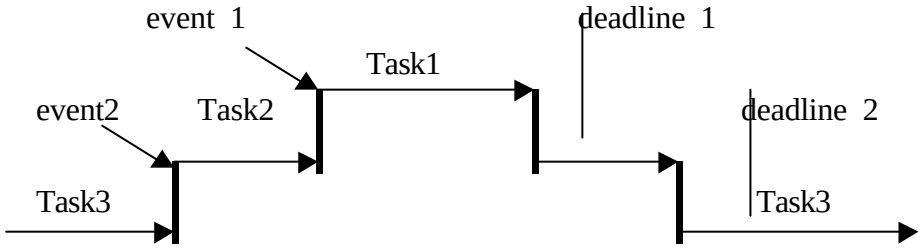
(1). I/O 依赖性



如果换依赖于 I/O，那么它运行的速度常常受限于与它互操作的 I/O 设备的速度。在这种情况下，变换应成为一个独立的任务。

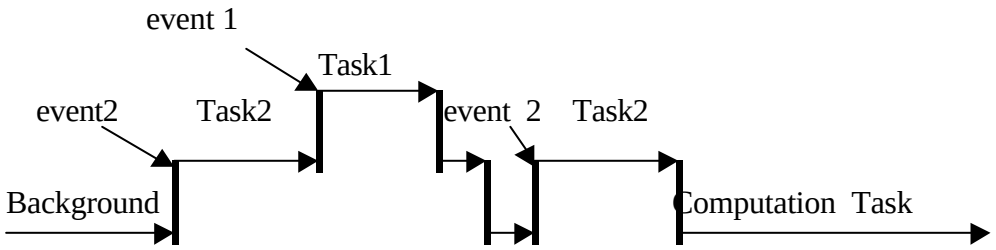
- 在系统中创建多个与 I/O 设备相当数目的 I/O 任务
- I/O 任务只实现与设备相关的代码
- I/O 任务的执行只受限于 I/O 设备的速度，而不是处理器
- 在任务中分离设备相关性

(2). 时间关键性的功能



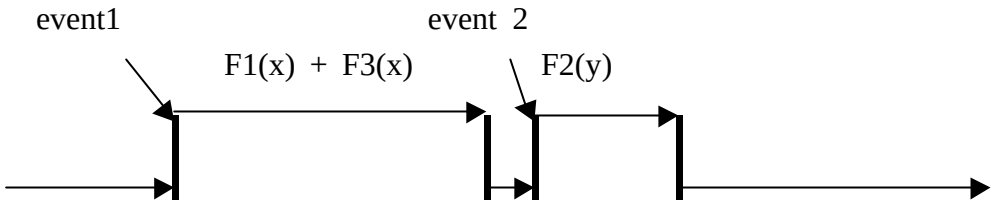
- 将有时间关键性（deadline）的功能分离出来，组成独立运行的任务
- 赋予这些任务高的优先级，以满足对时间的需要

(3).计算功能



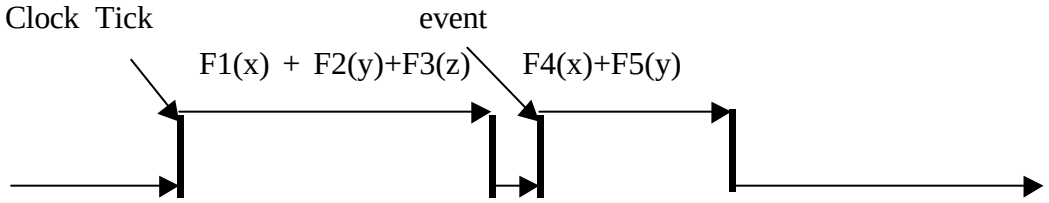
- 计算功能占用 CPU 的时间多
- 捆绑计算功能成任务，赋予它们较低优先级运行，能被高优先级的任务抢占，消耗 CPU 的剩余时间。
- 保持高优先级的任务是轻量级的
- 多个计算任务可安排成同优先级，按时间片循环轮转

(4).**功能内聚**



- **各紧密相关的功能，不能分别对应不同的任务**
- **将这些紧密相关的功能组，组成一个任务，使各功能共享资源或相同事件的驱动。**
- **组成一个任务会减少通信的开销，而且不仅保证了模块级的功能内聚，也保证了任务级的功能内聚。**

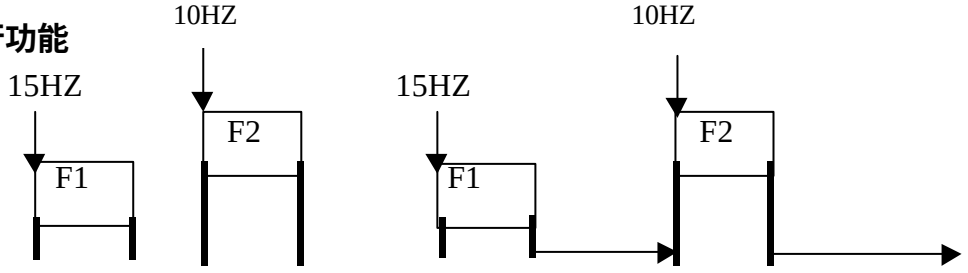
(5).**时间内聚**



- **将在同一时间内完成的各功能，即使这些功能是不相关的，组成功能组，形成一个任务。**
- **功能组的各功能是由相同的外部事件驱动的（如：时钟等），这样每次任务接收到一个事件，它们都可以同时执行。**
- **组成一个任务，减少了系统的开销**

虽然时间内聚在结构化设计中并不被认为是一个好的模块分解原则，但在任务级是可以被接受。每个功能都作为一个独立的模块来实现，从而达到了模块级的功能内聚，这些模块组合在一起，又达到了任务级的时间内聚。

(6). **周期执行功能**



- 将在相同周期内执行的各功能组成一个任务
- 频率高的赋予高优先级

2．如何评价

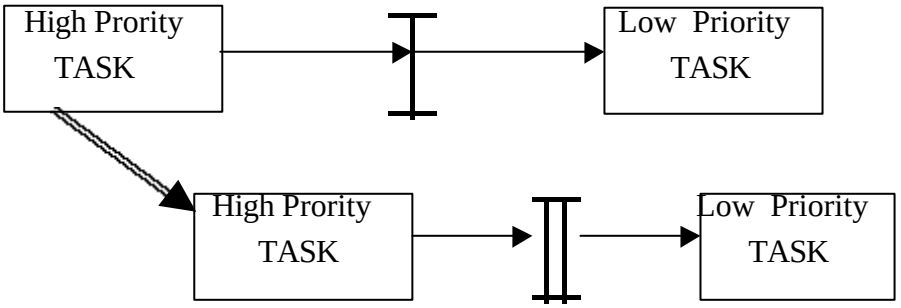
(1). 错误的任务划分

- 任务使用 SUPSPEND/RESUME 太频繁
  是由于任务划分过细，任务的当成功能使用。
  改进的方法是：将任务变成子程序使用。
- 当事件发生时调用子程序
  任务划分的太粗，将子程序划分为任务
- 得到消息后，又立即检查另外的信息
  不要使用轮循的方式，直接使用事件驱动方式

(2) 优先级倒置之一

当高优先级的任务向低优先级的任务发送消息时，如果使用信箱机制，就可能出现高优先级的任务要等待低优先级的任务接收消息以后，才能发送消息。

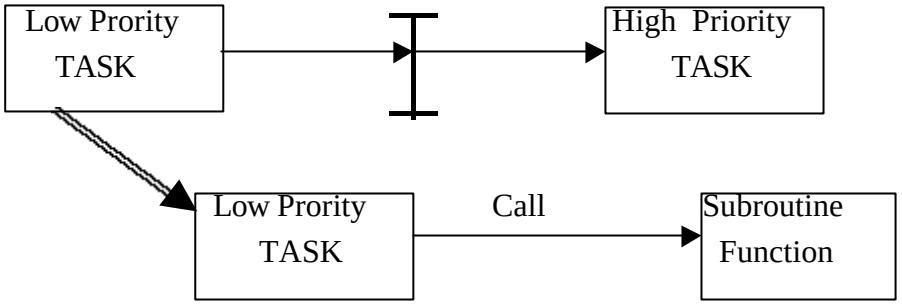
使用队列机制就可以避免这个问题。



(3) 优先级倒置之二

当低优先级的任务向高优先级的任务发送消息时，高优先级的任务不能运行，直到低优先级的任务发送消息后才能运行。

这种情况下，就没有必要分为两个任务，只需要使低优先级的任务调用子程序即可。



(4) **死琐和锁住**

死琐：

- **两个任务同时相互等待对方的信号**
- **导致它们永远不能运行**
- **为了避免死琐，将共享资源统一排序，所有的任务按序来访问多个资源。**

锁住：任务没有机会运行，可能是因为

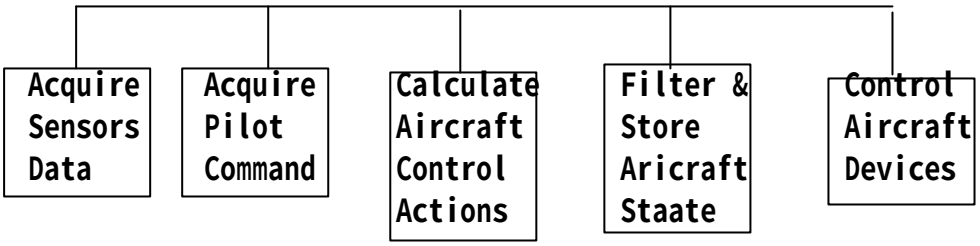
- **它等待的事件没有发生过**
- **它具有太低的优先级**

2.3 设计例子之一 ---- 数字飞控系统“Fly-by-Wire”

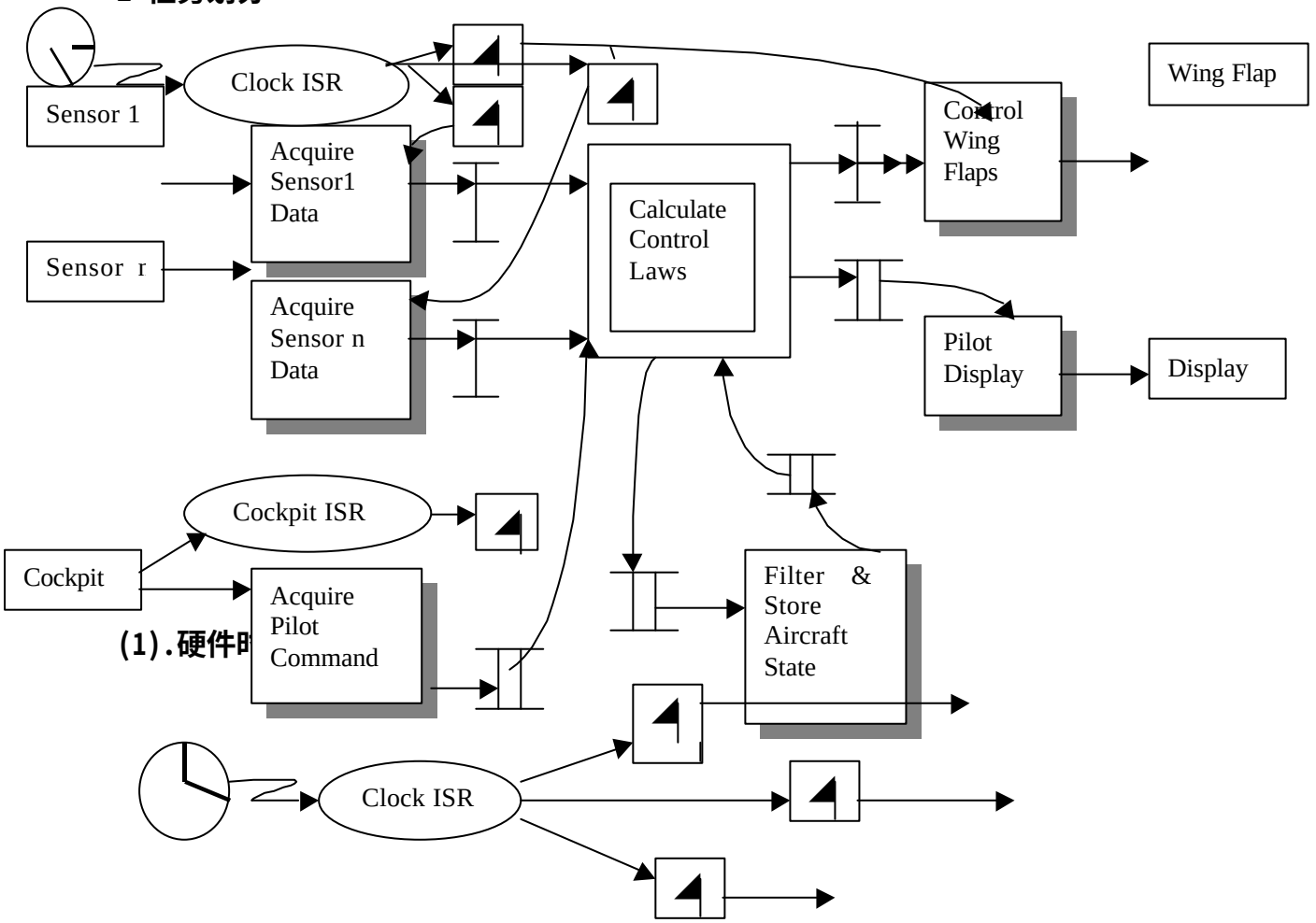
1 **需求分析**

- **飞机的传感器数据必须每 20ms 采样一次**
- **飞机的副翼根据当前周期内获得的数据，每 20ms 调整一次**
- **飞行员命令的发生是基于中断事件，但低于上面两个事件的优先级**
- **飞行员显示的延迟不能大于 100ms**

Input	Process	Output
Gyro measurements Airspeed measurement Pilot commands	Control Flight of Aircraft Fly_by_Wire	Flaps Control Command Pilot Displays



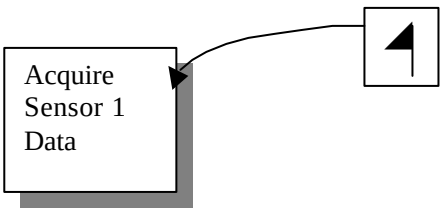
2 任务划分

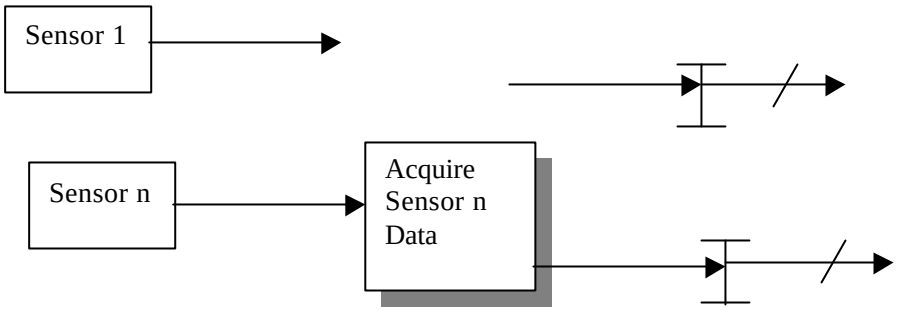


(1).硬件时

- **中断服务程序** ISR post **三个信号量，即使信号量加一。以激活** 20ms **的周期任务：** Acquire Sensor Data **及** Control Wing Flaps。
- **任务的执行严格依赖硬件时钟的，要通过中断+ISR+任务，或中断+ISR。**

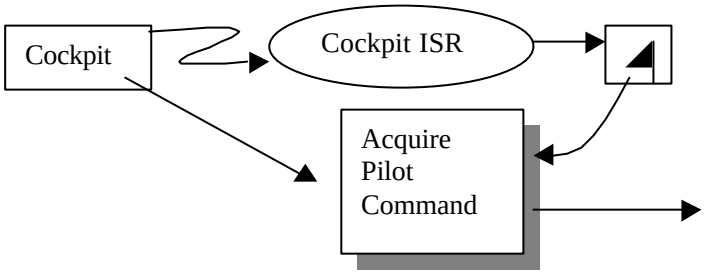
(2). **高优先级的设备服务**





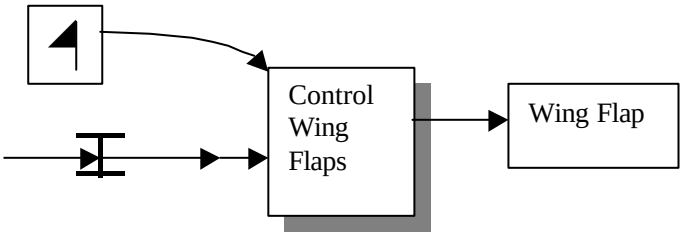
- 设备服务任务每 20ms 被信号量触发
- 当它从传感器上读取数据后，它向信箱发送消息
- 同步采样和确定的操作

(3)低优先级设备服务



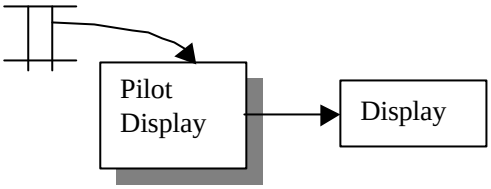
- 中断服务程序 ISR post 信号量，即使信号量加一，以激活设备服务任务 Acquire Pilot Command 完成以后的工作。
- 当高优先级的任务不运行时，设备服务器任务（低优先级的任务）从 驾驶员坐舱中读取数据并且处理它们。

(4) 高优先级输出到设备



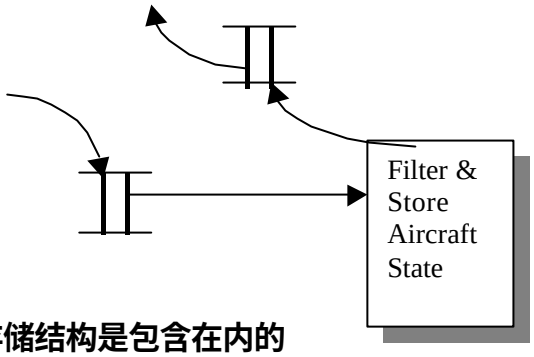
- 这是通过软件处理的最关键路径的结尾部分
- 它是通过信号量，由硬件时钟每 20ms 激发一次
- 采用非阻塞方式获得输入数据

(5)低优先级输出到设备



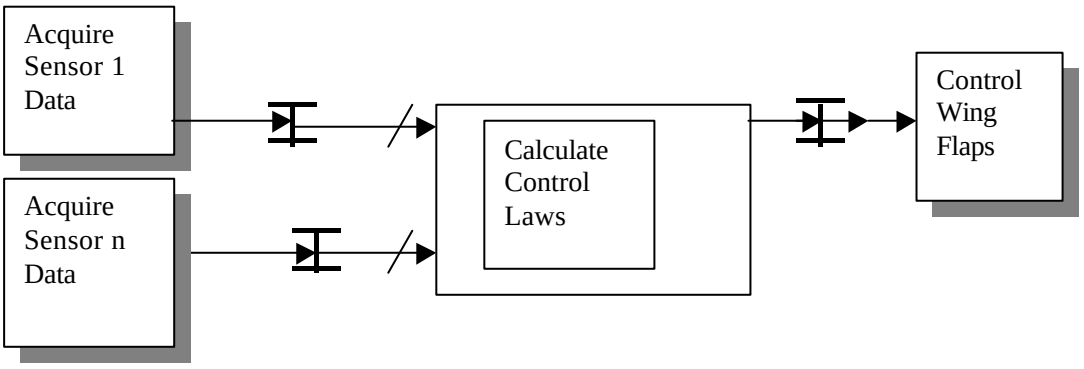
- 显示任务读取队列上的数据，将其送到显示设备上
- 显示数据可放在 BUFFER 中，队列上的数据实际上是 BUFFER 的指针

(6)计算任务



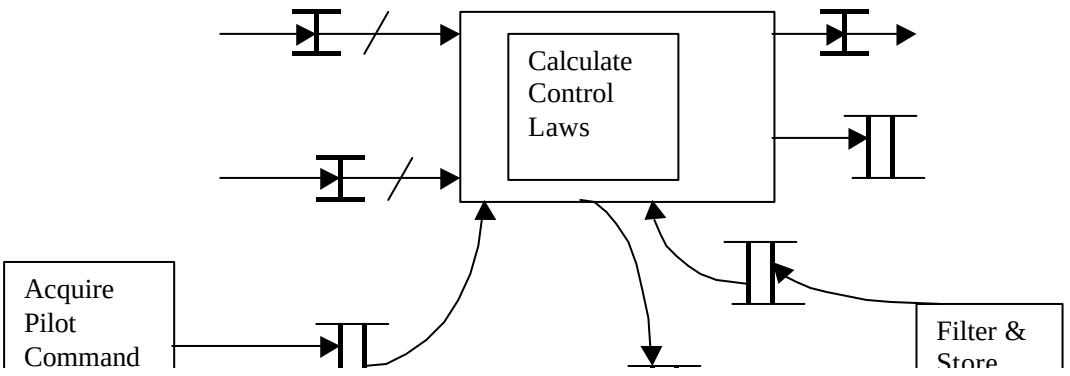
- 过滤算法和存储结构是包含在内的
- 这个任务的优先级不高
- 当高优先级的任务不运行时，它重复的执行计算。

(7)关键响应路径



- 在这条路径上，需要快而确定的响应
- 同步数据的输入和输出
- 非阻塞获取数据：通过 timeout 或 accept 系统调用

(8)计算路径



．从关键路径上快速的获取数据，然后获取驾驶员的命令
．剩余的时间做过滤和存储，且循环回到关键路径

2.4 模块构筑

系统和任务设计完成后，进行每个模块的详细设计；详细设计完成后才可开始编码。但是在单元测试前不必编完模块全部程序，可以分阶段编码和测试。虽然如此，该模块的详细设计应该先完成。这样保证设计工作一气呵成，避免系统以非结构化方式形成。

2.5 任务与系统集成

模块逐个连接、测试以构成任务，任务被逐个连接和测试形成最终系统；可分两步集成，在宿主机上模拟集成(软集成)，在目标机上的集成.

第三章　嵌入式实时内核 VRTXsa

VRTXsa 是针对嵌入式应用的、通用的实时执行程序（内核）。它是实时多任务系统的基础，负责管理、调度任务(安排 CPU 的时间)和任务间的通信、同步，提供时钟和中断管理机制。VRTXsa 与 VRTX32,VRTXmc/VRTXoc,VRTX5 兼容

3.1. VRTXsa 的特征

VRTXsa 具有支持实时多任务执行和可剪裁的体系结构的特点。

3.1.1 支持实时多任务执行的特点

1. 多任务支持
2. 事件驱动，基于优先级的调度
3. 任务间的通信与同步
4. 动态存储分配
5. 实时时钟控制，带有可选的时间片
6. 字符 I/O 支持
7. 完全可抢占内核，硬实时响应

3.1.2 体系结构的特点

VRTXsa 可以用于开发不同类型的嵌入式应用，它具有：

1.目标环境独立性：VRTXsa 仅需要一个小存储容量的 CPU，提供了真正的芯片级的支持。

2.可扩充性：可以很容易地将应用专用的、系统级的软件和 VRTXsa 结合起来。扩充的软件可以是独立运行自己的系统调用处理程序和例程，也可以是由 VRTXsa 来统一管理，调用执行。

3.位置无关性：尽管 VRTXsa 在运行的时候不是位置无关的，但在连接时却是位置无关的，因为它具有一个可重定位的目标库。可以被定位到微处

理器的任意地址空间。

此外，还可很容易地将附加的软件组件(如:IFX,SNX,应用自己的组件)集成到系统中来。

3.2 VRTXsa 的配置

VRTXsa 配置表和简单的、设备专用的中断处理程序在 VRTXsa 和系统环境之间提供了一个接口。配置表为 VRTXsa 与它的周围环境提供了一个重要的连接。它是 VRTXsa 面向周围环境的窗口，使 VRTXsa 的存在不是孤立的，而是有机地和应用环境联系在一起。

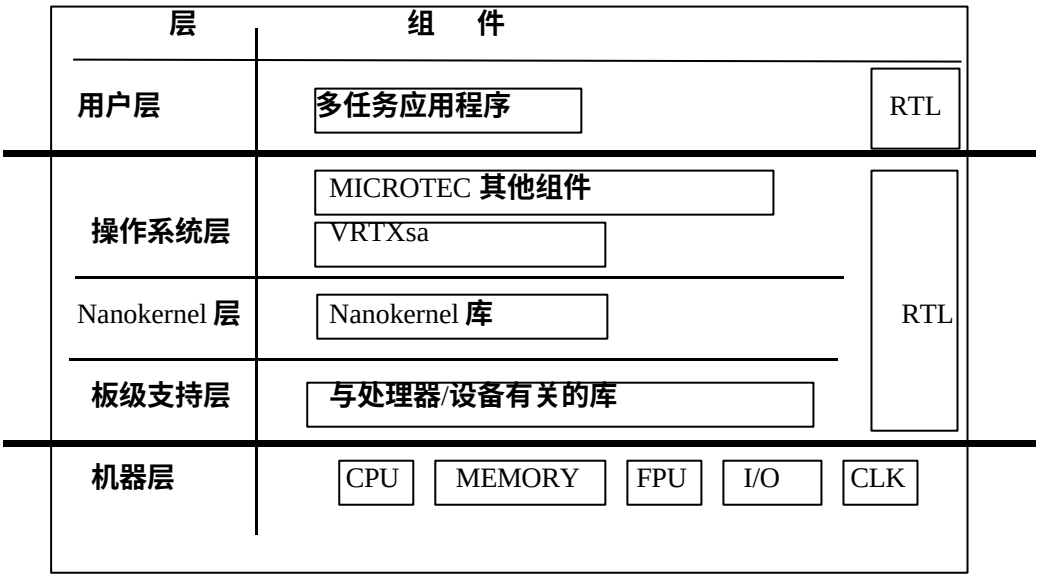


图 3-1 基于 VRTXsa 应用系统的体系结构

配置表指明了 VRTXsa 在一个特定的系统环境当中需要的所有参数。例如系统可管理内存的起点和范围，系统堆栈的大小，多任务参数，与其他软件组件的连接，以及 VRTXsa 扩展代码的指针等。其中，VRTXsa 扩展代码是用户提供的对 VRTXsa 机制的扩充程序，即对 VRTX 某一特定的系统调用或功能(如：SC-TCREATE，SC-TDELETE，任务切换等)的扩展。当 VRTXsa 在执行这个系统调用或功能时，同时检查并执行这个扩展代码。

可使用 Xconfig 配置应用程序来自动地处理大部分的配置任务。

3.3 VRTXsa 体系结构

建立在 VRTXsa 基础上的系统根据功能来分层，每一层都使用下一层提

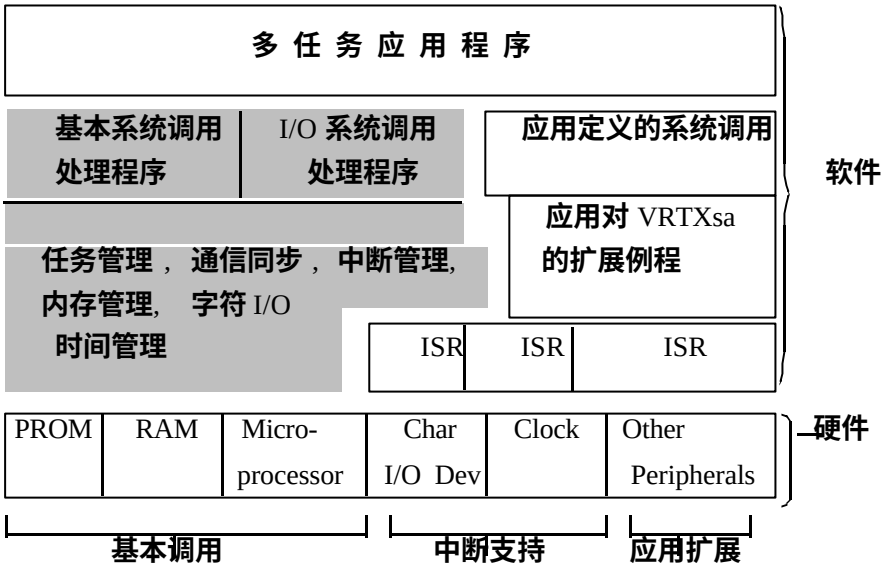


图 3-2 VRTXsa 基本体系结构

供的功能。系统硬件构成了系统的最底层。紧接着一层包括了最简单的，大多是硬件相关的操作系统功能，最上层是应用程序。如图 3-1 所示，从技术的角度来看，每一层都为其上一层定义了一个虚拟机。在更高的层上，是不能分辨出由软件提供的功能与由硬件提供的功能，每一层都增加了一些功能。

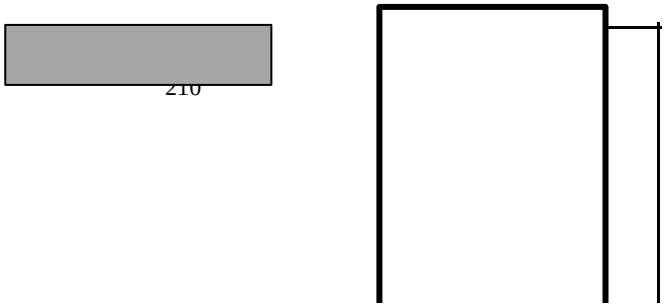
图 3-2 是 VRTXsa 基本体系结构，可以看出，VRTXsa 对硬件的基本要求即 PROM，RAM，CPU，CLOCK，字符 I/O 设备；它与多任务应用程序的接口：系统调用。

图 3-3 展示了一个完整的 VRTXsa 应用系统软件结构

3.4 VRTXsa 关键术语的定义

1. 上下文切换（Context Switch）

在多任务系统中，上下文切换指的是当处理器的控制权由运行任务转移到另



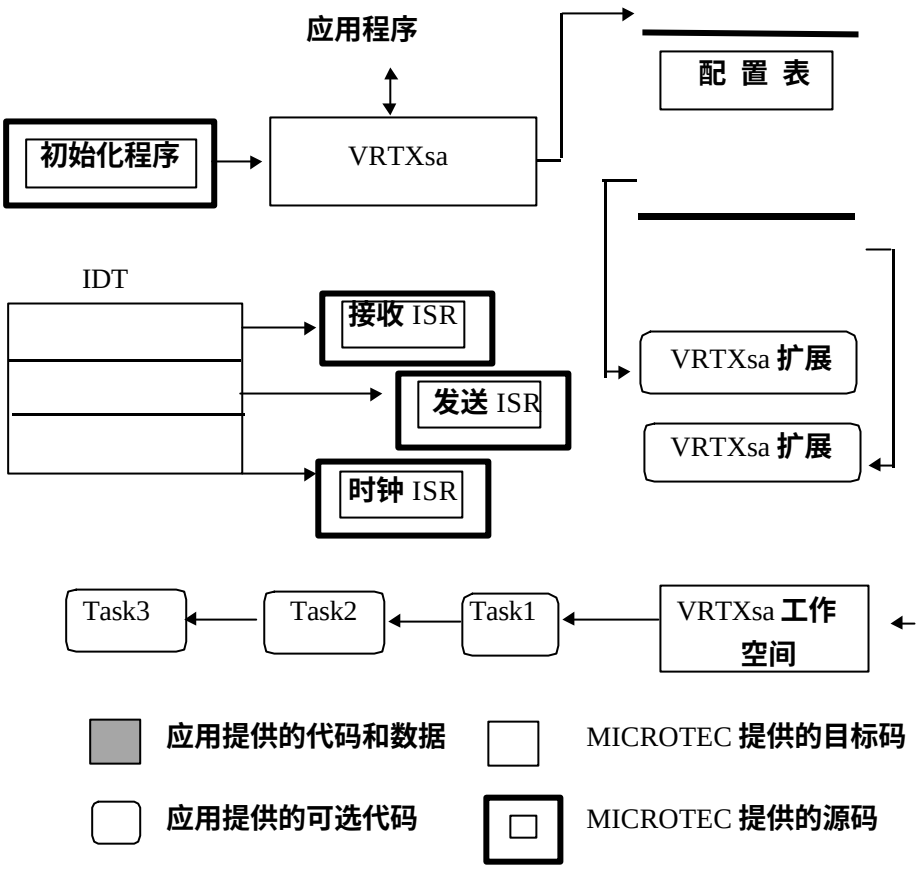


图 3-3　一个完整的 VRTXsa 应用系统软件结构

外一个就绪任务时所发生的事件序列。当前运行的任务转为就绪，挂起，或删除时，另外一个被选定的就绪任务就成为当前任务。上下文切换包括保存当前任务的状态，决定哪个任务运行，恢复将要运行的那个任务的状态。保存和恢复上下文是依赖于相关的处理器的。

2.**确定性**（Deterministic）

在实时操作系统中，在一定的条件下，系统调用运行的时间可以预测。这并不意味着所有的系统调用都总是执行一个固定长度的时间，而是指不管系统的负载如何，系统调用最大的执行时间可以确定。

3.**直接调用与陷井接口**（Direct Call vs. Trap Interface）

VRTXsa **主要是由与应用相连的一套库构成的。这就意味着应用可以通过一个** C **语言调用接口或汇编语言直接调用来使用** VRTXsa **提供的功能。**VRTX32 **中只使用的是一个陷井接口，在** VRTXsa **当中也支持这种接口。**

4.**动态存储分配**(Dynamic Memory Allocation)

任何 VRTXsa **的对象，如队列，分区等，都可以动态地创建和删除 。当创建对象时，系统从该对象的空闲结构池中分配一个对象；删除一个对象并不会破坏结构，而只是将对象归还到该对象的空闲结构池中。系统配置和初始化时，创建所有对象的空闲结构池。**

5.**硬实时**(Hard Real-Time)

一个硬实时系统是一个确定的，优先级驱动的，并且可以抢占的操作系统。

6.**中断延迟**(Interrupt Latency)

中断延迟是指从中断发生到开始执行中断处理程序代码的这一段时间。

7.**模块**(Modular)

一个模块化的系统是由独立操作的并且在使用上提供灵活性和可变性的单元组成的系统。VRTXsa **是模块化的，因为它允许一个应用直接与内核库连接并且只连入应用使用的系统调用。**

8.**互斥**(Mutex)

互斥是用来控制多任务（或线程）对共享数据进行串行访问的同步机制。在多任务应用中，当两个或多个任务同时访问共享数据时，就会造成数据破坏。互斥使它们串行地访问数据，从而达到保护数据的目的。

9.**抢占**(Preemptable)

抢占是指当系统处于核心态的内核运行时，允许任务的重新调度。换句话说就是指一个正在执行的任务可以被打断而让另外一个任务运行。这就提高了应用对外部事件的响应性。VRTXsa **是可以抢占的。但这并不是说调度在任何时候都可以发生。例如当一个** VRTXsa **任务正在通过一个系统调用访问共享数据时，重新调度和中断都是不允许的。**

10.**优先级驱动**(Priority-Driven)

优先级驱动是指在一个多任务系统中，正在运行的任务总是最高优先级的任务。在任何给定的时间，总是把处理器分配给最高优先级的任务。

11.**优先级反转**(Priority Inversion)

当一个任务等待比它优先级低的任务释放资源而被阻塞时，优先级反转发生了。优先级低的任务阻塞了优先级高的任务。优先级继承技术可以解决优先级反转问题。

12.**优先级继承**(Priority Inheritance)

优先级继承是用来解决优先级反转的技术。当优先级反转发生时，较低优先级的任务的优先级被暂时地提高，以匹配较高优先级任务的优先级。这样就可以使较低优先级任务尽快地执行并且释放较高优先级任务所需要的资源。

13.**实时执行体**(Real-Time Executive)

实时执行程序包括一套支持实时系统所必需的机制。如多任务支持，CPU 调度，通信和存储分配。在嵌入式应用中，这一套机制被称为实时操作系统（RTOS）或实时执行体或实时内核。VRTXsa 就是一个实时执行体。编程者以实时执行体为基础来构造自己应用。

14.**重调度过程**(Rescheduling Procedure)

重调度过程是判定任务优先级和执行状态的过程。VRTXsa 在系统调用引起任务状态变化时，就会执行这个过程。这时 VRTXsa 就会确定就绪任务中优先级最高的任务，如果不是当前正在运行的任务，就会执行一个任务切换。

15.**调度延迟**(Scheduling Latency)

调度延迟是指当一个事件引起更高优先级的任务就绪到这个任务开始运行之间的时间。即一个任务被触发后，由就绪到运行的时间。

16.**可剪裁的体系结构**(Scalable Architecture)

可剪裁的体系结构是指一个软件系统能够支持多种应用而无需在接口上做很大的变动。因此，开发出成本低、品种多的产品。

17.**任务/线程**(Tasks/Threads)

一个任务是 VRTXsa 中可独立运行的一个过程。一个线程是指在纳核级上的一个可单独运行的过程。“任务”这里特指 VRTXsa 的任务。一个任务可以包含多个线程。线程是一个更新的，更为广泛的术语。

19.**任务上下文**(Task Context)

任务上下文是指一个未运行的任务的状态，如程序计数器，堆栈指针，和通用寄存器的内容。

3.5 如何使用 VRTXsa

3.5.1. 与 C 语言的接口

应用任务可通过 C 语言调用接口函数来调用 VRTXsa 提供的系统功能。

VRTXsa 的库与应用程序相连，且只有应用使用到的功能才被装入。

所有 VRTXsa 的 C 函数都遵循 MICROTEC 的 MCC C 编译器的调用协定。在库里面的每一个函数都与一个 VRTXsa 系统调用相对应。其调用格式如下:

函数返回值(或 void) VRTXsa 系统调用名（参数 1，…，参数 n，&err）

VRTXsa 的错误代码返回到应用任务定义的整形变量 err 中。当调用成功时，err 返回为 0，否则，返回一个错误代码。一些 VRTXsa 系统调用如 sc_accept，sc_qaccept，sc_qinquiry，sc_pend，sc_qpend，sc_tinquiry，sc_screate，sc_minquiry 等，除了返回错误代码外，还要有函数返回值。当错误发生时，系统调用的返回值是不确定的。因此，应用应当在使用返回值之前检查错误代码。

vrtxil.h 头文件说明了 VRTXsa 的 C 函数。当在 C 语言的应用程序里包括了 vrtxil.h 头文件时，就不用再显式地说明每一个 VRTXsa 函数。

VRTXsa 支持 C++ 编程接口

3.5.2 与汇编语言的接口

VRTXsa 的手册中未提供与汇编语言的接口，如果要使用汇编语言，可以参照 C 语言函数的参数压栈顺序，按照由左到右的顺序压栈，然后在调用该函数，返回参数放到相应的寄存器中。以 VRTXsa x86/fpm 为例，返回的参数与寄存器的对应关系如下：

C 语言的函数返回值	汇编寄存器
char	AL
short	AX
int, long, near 指针	EAX
far 指针	EAX/DX
FPU：float	EAX
double	EAX/EDX
long double	EAX/ECX/EDX

如果调用 void ui_rxchr(char,int *errp)

```
mov ebx,offset err     ; int *errp
push ebx
push ax                ;char
call ui_rxchr
pop ax
```

```
pop ebx
```

VRTX32 提供了如何用汇编语言调用系统调用的方法。

3.5.3 函数的可重入性（Reentrancy）

在一个多任务环境中，函数的可重入性是很重要的。可重入函数是一个可以被多个任务调用的过程，任务在调用时不必担心数据是否会出错。在写函数时只要考虑到尽量用局部变量(例如寄存器，在堆栈中的变量等)，对于要使用的全局变量需要加以保护(如采用关中断，信号量等)，这样构成的函数就一定是可重入的函数。

编译器是否有可重入函数的库，与它所服务的操作系统有关，例如 DOS 下的 Borland C 和 Microsoft C/C++等就没有可重入函数库，这是因为 DOS 是一个单用户单任务的操作系统。

为了确保每一个任务控制自己私有的变量，在一个可重入的 C 函数中，将这样的变量声明为局部变量。C 编译器将这样的变量存放在调用栈上或寄存器里。

在 VRTXsa 编写可重入性的函数，需遵循以下的规则：

1. 将所有的局部变量声明为 auto(缺省态）或寄存器型。
2. 尽量不要使用 static 或 extern 变量。
3. 用 sc_gblock 分配大的数据结构。

第四章　任务管理

如图 4-1 可见 VRTXsa 中的任务管理

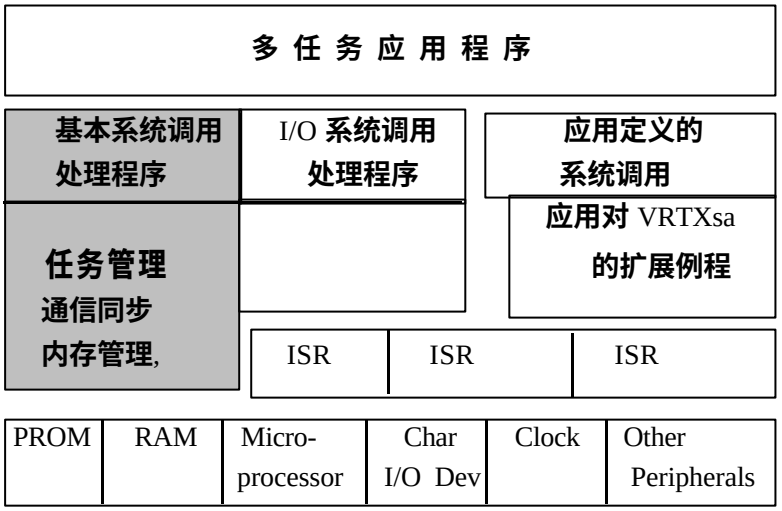


图 4-1　VRTXsa 中的任务管理

4.1 任务管理简介

4.1.1 任务

任务就是一个具有独立功能的无限循环的程序段的一次运行活动。具有动态性、并行性、异步独立性的特点。

动态性：任务的状态是不断变化的，一般分为:休眠态(dormant)，就绪态(ready)，运行态(running)，挂起态（suspended），睡眠态（sleep）等。

并行性：系统中同时存在多个任务，它们宏观上是同时运行的

异步独立性：任务是系统中独立运行的基本单元，也是内核分配和调度的基本单元，每个任务各自按相互独立的不可预知的速度运行，走走停停。

每个任务都要按排一个决定其重要性的优先级，都有一个无限循环的程序段规定其功能（如一个 C 语言过程）,并相应有一个数据段、堆栈段及一个任务控

制块(保存 CPU 的现场，状态等)。

4.1.2 VRTXsa 的任务

VRTXsa 的任务任务有激活和非激活两种。非激活的任务是休眠态(dormant)的任务，它不竞争 CPU。激活的任务具有运行，挂起和就绪三种状态。每个激活的任务都需按排一优先级(0-255)，具有唯一的任务标识号(1-最大任务数)，应用的最大激活任务数（CFUTSKCT）需要在配置表中配置。VRTXsa 根据优先级和引起重调度的系统调用将任务由一个状态变为另一个状态。

VRTXsa 为每个激活的任务分配一任务控制块（TCB）和任务堆栈，以保存任务在非运行状态时的任务状态信息即上下文。

任务可创建其它的任务。它们也可以删除、挂起、解挂任务，查询任务的状态，改变它们自身或其任务的优先级。任务还可锁住调度使其他任务抢占它，以运行其关键的临界代码区。任务可在管态，也可在用户态两种权限下运行，在管态下任务可以使用微处理器的全部指令集。

4.2 任务管理的系统调用

表 4-1 列出了 VRTXsa 提供的任务管理的系统调用

1.创建任务

函数原型:

```
int sc_tcreate (void(*task)(void*), int tid, int pri, int *errp)
int sc_tecreate (void(*task)(void*),int tid, int pri, int mode,
                 unsigned long user, unsigned long sys,
                 char *paddr, unsigned long psize, int *errp)
```

sc_tcreate 调用动态地创建一个具有优先级 pri，任务号 tid 和权限(管态或用户态)，和任务代码的起始地址的任务。创建的任务的初始状态是就绪态，开中断，TCB 的其它内容和堆栈的值由创建者的任务的环境决定。

sc_tecreate 调用是 sc_tcreate 的扩展，它除了指明创建任务的优先级 pri，任务号 tid 和任务代码的起始地址外，还可指明用户栈和系统栈的大小，任务的权限（用户态或管态），任务初始态是否被挂起，是否不可抢占，任务是否使用一个浮点协处理器，时间片是否使能，是否开中断。同时，还可将一个参数块从调用者传递到所创建的任务。VRTXsa 拷贝这个参数块，以便创建任务可使用这块

区域。

任务的标识号 TID 可以在创建时指定（范围是：1-max（255，CFUTSKCT））；也可由 VRTXsa 动态分配 TID（tid=-1）返回给任务；或指定为 tid=0，这是一种特例，即创建的任务标识号，这样其他的任务就无法使用一些需要指明 TID 的系统调用来对它进行操作如调用 sc_tdelete，sc_tsuspend，sc_tpriority，和 sc_tinquiry 时。

表 4-1 VRTXsa 任务管理的系统调用

调用名	功能描述
sc- tcreate	创建一个具有优先级，任务号，权限和任务代码起始地址的任务
sc-tecreate	动态创建任务。除指明优先级，任务号，任务代码起始地址外，还可指明用户栈和系统栈的大小，任务的权限（用户态或管态），任务初始态是否被挂起，是否不可抢占，任务是否使用浮点协处理器，时间片是否使能，是否开中断。同时，还可将一参数块从调用者传递到所创建的任务中。
sc-tdelete	删除用任务 ID 和优先级指定的一个或多个任务
sc-tsuspend	挂起用任务 ID 和优先级指定的一个或多个任务
sc-tresume	解挂用任务 ID 和优先级指定的一个或多个任务
sc-tpiority	改变指定任务的优先级
sc-tinquiry	查询指定任务的状态
sc-lock	关调度，使调度琐加一
sc-unlock	开调度，使调度琐减一

2. 任务删除

函数原型:

void sc_tdelete(int tid/pri, int code, int *errp)

任务可用 sc_tdelete 调用删除一个或多个任务（包括它自己）。通过 tid/pri 和 code 的组合，可以删除一组具有相同优先级的任务，或指定 TID 号的任务，或调用者自己。每个被删除了的任务又回到休眠状态，占用的 TCB 可重用。

VRTXsa 任务当执行完它的代码后就自动地删除它自己，因此不必在任务代码的最后调用 sc_tdelete。在任务创建时，VRTXsa 就将 sc_tdelete 嵌入到任务代码体的最后。如下面的任务所示。

```
void    task( )
{     int  err,j,k;
      for (j=1,j<100,j++);
}
```

VRTXsa 自动插入

3. 挂起任务

函数原型:

void sc_tsuspend(int tid/pri, int code, int *errp)

任务可用 sc_tsuspend 调用挂起一个或多个任务（包括它自己）。通过 tid/pri 和 code 的组合，可以挂起一组具有相同优先级的任务，或指定 TID 号的任务，或调用者自己。

当一个任务被挂起时，在 TCB 的 TBSTAT 指示挂起的原因。挂起的任务只有用 sc_tresume 才能解挂。

4. 解挂任务

函数原型:

void sc_tresume(int tid/pri, int code, int *errp)

与 sc_tsuspend 相对应, sc_tresume 用来解挂一个或多个任务的。

5. 改变任务的优先级

函数原型:

void sc_tpriority(int tid, int pri, int *errp)

任务可用 sc_tpriority 来改变另一个任务或自己的优先级。

6. 查询任务

函数原型:

TCB *sc_tinquiry(int info[3], int tid, int *errp)

sc_tinquiry 调用可获取指定任务的任务号，优先级和状态信息。

7. 关调度和开调度

函数原型:

void sc_lock(void) ；关调度

void sc_unlock(void)；开调度

当任务要运行一段临界区的代码时，可用 sc_lock 调用来使 VRTXsa 不调度，即不允许其他任务包括优先级更高的任务抢占，调用 sc_lock 的任务一直保持运行，直到调用 sc_unlock 开调度为止。lock/unlock 最大的嵌套数是 65535。

在调用了 sc_lock 以后，当遇到当前任务自挂时，系统将 lock/unlock 的嵌套数存放在当前任务的 TCB 中，并且重新调度一个新的任务。当解挂该任务时，lock/unlock 的嵌套数从 TCB 中恢复。

4.3 任务状态和状态变迁

在 VRTXsa 多任务环境中，任务总是处于下列四种状态之一：运行，就绪，挂起和休眠。当某一事件发生时，它们就会从一种状态过渡到另一种状态。其任务状态变迁图如图 4-2 所示:

4.3.1 运行态

处于运行态的任务拥有 CPU 控制权并正在执行。任何时刻都只有一个任务处于运行态。

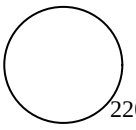
4.3.2 挂起状态

挂起的任务等待系统调用或事件将它唤醒(解挂)。下列原因会引起任务的挂起：

1. delay 调用

- 使用 sc_tsuspend 调用，挂起一个或多个任务;
- 使用 sc_delay 调用，将指定的任务挂起一个确定的时间间隔;
- 使用 sc_adelay 系统调用，将指定任务挂起到 VRTXsa 的时钟的某一时刻。

sc-tdelete



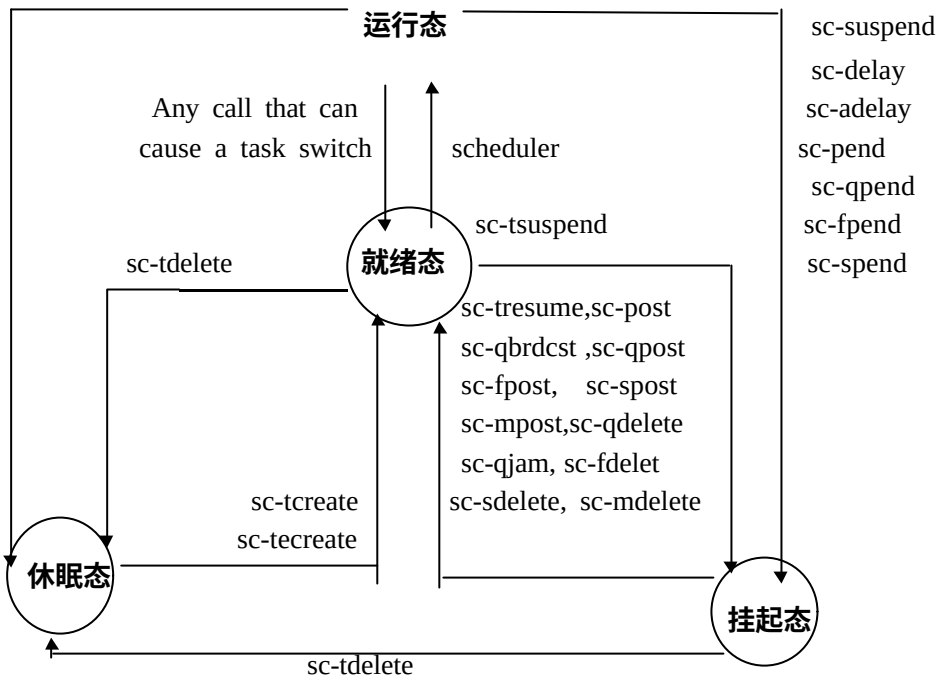


图 4-2 **任务状态变迁图**

2. Pend **调用**

- **任务调用了** sc_pend **或** sc_qpend，**但邮箱或队列中没有消息时**;
- **任务调用了** sc_fpend，**但指明的事件标志却并未设置**；
- **任务调用了** sc_spend, **但信号量等于** 0；
- **任务调用了** sc_mpend, **但是互斥量是锁住的**。

3. **字符调用**

- **任务调用了** sc_waitc **并等待来自一个** I/O **设备某一特定的字符**；
- **任务调用了** sc_getc, **但** VRTXsa **的输入缓冲区为空**；
- **任务调用了** sc_putc,**但** VRTXsa **的输出缓冲区已满**；

要知道任务被挂起的原因，可以使用 sc_tinquiry **调用，去查看** TCB **中的** TBSTAT **域。需注意的是：挂起是独立的且可迭加的。例如，当一个任务由于等待消息而被挂起，且又被另一个任务显式地（调用** sc-suspend）**挂起，那么只有当所有的挂起条件都被解除了，任务才能处于就绪状态。**

此外，当任务调用一些 VRTXsa **的系统调用时，在纳核线程级有时也要发生内部状态变迁。例如当线程想访问已被另一个线程锁住了的数据时，就会发生状**

态变迁，这个线程会被挂起，直到那个线程释放了资源为止。而这种变迁对于 VRTXsa 的使用者来说是不可见的。

这些系统调用是：

sc_adelay，sc_delay，sc_fpend(带超时)，sc_halloc(超过堆中页的大小的数据块)，sc_hdelete，sc_hfree(超过堆中页的大小的数据块)，sc_hinquiry，sc_mpend(带超时)，sc_pcreate，sc_pdelete，sc_pend(带超时)，sc_pextend，sc_qcreate，sc_qecreate，sc_qpend(带超时)，sc_spend(带超时)，sc_tcreate，sc_tdelete，sc_tecreate

4.3.3 就绪状态

就绪状态的任务是指运行的一切条件都准备好马上就能运行的任务。例如，刚被创建的任务就处于就绪状态。但就绪态的任务要成为运行态，就必须比所有处于就绪态的任务的优先级高。任务从挂起态转到就绪态可能因为以下原因：

1. resume 调用

- sc_tresume 调用将一个用 sc_tsuspend 挂起的任务恢复为就绪态;
- 当延迟时间到时，被 sc_delay 或 sc_adelay 调用挂起的任务可从挂起态转到就绪态。一个等待消息的任务超时时，也会从挂起转到就绪态。

2. Post 调用

- sc_post，sc_qpost 或 sc_qbrdcst 调用会唤醒在邮箱或队列中等待的任务，将消息发送给这些任务；
- sc_mpost 解锁互斥量，并且将锁交给等待互斥量的任务；
- sc_fpost 调用发送事件，唤醒等待事件的任务；
- sc_spost 调用指明资源可用。

3. Delete 调用

- 当有任务在等待指定的互斥量、信号量和队列时，强制删除互斥量、信号量和队列（如调用 sc_mdelete,sc_qdelete,sc_sdelete)，就会使这些任务成为就绪态。

4. 字符调用

- 中断处理程序（ISRs）用 ui_rxchr 调用发送一个特定的字符给 VRTXsa 输入缓冲区时 。VRTXsa 将唤醒调用 sc_waitc 等待这个字符的任务。
- ISRs 用 ui_rxchr 调用发送一个字符到输入缓冲区时，将唤醒因缓冲区空而被

挂起的任务。

- ISRs 用 ui_txrdy 调用从输出缓冲区获得一个字符时，将唤醒因输出缓冲区满而被挂起的任务。

此外，当一个优先级更高的任务处于就绪态时，处于执行态的任务就会交出 CPU 控制权而回到就绪态，由更高优先级的任务占有 CPU。直到所有优先级更高的任务都处于挂起态或休眠态为止，这个任务才会重新获得 CPU 的控制权。

4.3.4 休眠状态

一个休眠的任务是指没有被初始化的未被创建的任务，或任务的执行被终止的任务（任务删除）。系统没有为处于休眠状态的任务分配 TCB。

任务在它们被创建之前处于休眠状态。当它们被 sc_tdelete 调用删除后，又重新回到休眠状态。当所有的任务都被删除或挂起时，系统就切换到空闲任务（IDLE TASK），由它占用 CPU 直到外部事件发生，有其它任务就绪为止。

4.4 任务调度

4.4.1 优先级调度

任务调度就是从就绪状态的任务中，挑选一个任务到处理器上运行。负责任务调度功能的内核程序称为任务调度程序或任务调度器。它是内核的一个重要组成部分。在设计任务调度器时，首先要决定选择何种调度算法，然后根据此算法来编制相应的调度程序。而调度算法实际上就是系统所采取的调度策略，选择时所要考虑的因素很多。如系统各类资源的均衡使用；对用户公平并使用户满意等。大多数实时内核都是采用优先级(priority)的调度算法。

优先级调度算法是按照任务的优先级大小来调度，使高优先级任务得到优先处理的调度策略。任务的优先级可以由系统自动地按一定的原则赋给它，也可以由系统外部来按排，甚至可由用户支付高费用来购买优先级。在实时操作系统中，任务的优先级是应用程序设计者按照任务的重要程度来按排的，并且任务在运行中其优先级可以动态改变的。

优先级调度按是否被抢占可分为：不可抢占(non-preemptive)的优先级调度法和可抢占(preemptive)的优先级调度法

1. 不可抢占的优先级调度法:

即一旦某个高优先级的任务占有了处理器，就一直运行下去，直到任务由于自身的原因自愿放弃处理器时(如任务等待事件)才按优先级进行调度让另一高优先级任务运行。任务在运行过程中可以被中断，中断处理程序在运行过程中即使唤醒了一个更高优先级的任务，在 ISR 完成后还是返回到被中断的任务,只有这个任务放弃了处理器时，更高优先级的任务才能运行。

2.可抢占的优先级调度法:

任何时刻都严格按照高优先级任务在处理器上运行的原则进行任务的调度，或者说，在处理器上运行的任务永远是就绪任务中优先级最高的任务。当优先级高的任务能运行时，保证其 CPU 的时间，让其尽快运行完。如果优先级高的任务因故(如等待事件)暂停运行，则就让 CPU 运行优先级次高的任务，一旦优先级高的任务又就绪(因事件的到来而成为就绪)，任务调度器就迫使原运行任务马上让出处理器给优先级最高的这个任务使用或称被抢占了处理器。

此外，为了使优先级相同的任务具有平等的运行权利，优先级调度法还可附加时间片循环轮转法，称为基于优先级的时间片循环轮转法，即:当有两个或多个就绪任务具有相同的优先级且它们是就绪任务中优先级最高的任务时，调度程序就选择这组任务中的第一个就绪任务，让它仅运行一段时间，运行的这段时间就称为时间片(time slicing)，在运行完一个时间片后，该任务即使还没有停止运行，它也必须释放处理器让下一个与它相同优先级的任务运行(假设这时没有比更高优先级的任务就绪)，而释放处理器的任务就排到同级优先级最后任务的后面，等待再次运行。

时间片大小的选择对系统的有效操作是有影响的，时间片太大，时间片轮转调度法就没有意义，时间片太小，任务切换过于频繁，处理器开销大，真正用于运行应用程序的时间将会减小。

4.4.2 VRTXsa 的任务调度

VRTXsa 采用可抢占的优先级调度算法，附加可选择时间片循环轮转法来调度任务。在任务创建时，须为它指定一个优先级(0—255)，系统共有 256 个优先级，优先级数值越大，优先级越低：相反，数值越小，优先级越高，0 为最高优先级。处于同一优先级的任务可以有任意个。

任务可通过 sc_lock 禁止 VRTXsa 进行任务调度。任务可通过 sc_tslice 打开/

关闭时间片循环轮转调度以及重新设置时间片。

4.5 使用任务管理调用

4.5.1 VRTXsa 中的任务

在用 C 编制的 VRTXsa 多任务程序中，任务是用函数来表示的。这就意味着 VRTXsa 任务能够用 C 语言提供给函数的所有机制。比如分块编译，和将多个函数集成为一个库的能力。

一个用 C 语言写的 VRTXsa 多任务系统包括了一些作为任务并行运行的函数和一些作为子程序顺序运行的函数。

C 程序能够隐含地传递一个函数的地址（不需要&号）。当使用函数名作为参数时，编译器实际上传递的是这个函数的地址。例如，下面的 C 语句隐式地传递了一个函数任务的地址。

```
sc_tcreate((void*)task,tid,pri,&err)
```

这里，task 参数是任务的函数名。

任务函数具有以下特性：

- 它们必须被声明为 void 函数，因为它不能返回一个函数值。
- 它们不能够返回到调用者。

VRTXsa 的任务有两种实现方式：

- 任务是一个无限循环

```
void task()
{
   initial ;
  while(1){
    ……
  }
}
```

- 任务当完成时删除它自身

```
void task()
{   .
      . /*task activities*/
      sc_tdelete(0,0,&err);
}
```

一个 C 函数可以表示多个任务。这时创建多个任务的 sc_tcreate 调用指向一个相同的函数地址。如果使用这种方法，一定要确保共享的函数是可重入的。

4.5.2 TCB 和状态地址

sc_tinquiry 调用返回 tcb 和&info 参数。sc_tinquiry 具有下列的调用格式：

```
tcb=sc_tinquiry(info,tid,&err)
```

当这些调用返回后，变量 tcb 的值是任务的 TCB 的地址。程序必须将 tcb 声明为 TCB 的结构指针。

sc_tinquiry 调用同时将任务的状态信息放在一个叫做 info 的三元数组返回。

4.6 任务管理实验

```
#include "vrtxil.h"
#include <stdio.h>

void user_main()
{
    extern void keyboard_interrupt_enable();
    void task1(),task2();
    int opt,err;
    long int i;
    printf("\n\r ===>Type any key to continue.\n");
    opt=sc_getc();
    printf("\n\r            Program: ");
    printf("VRTXDM1---Task Management");
    printf("\n");
    printf("\n");

    err=0;
    sc_tslice(50);
    sc_tcreate(task1,1,4,&err);
    if (err!=0) printf("tcreate task1 error.\n");
    sc_tcreate(task2,2,4,&err);
    if (err!=0) printf("tcreate task2 error.\n");
    sc_tsuspend(1,0,&err);
    if (err!=0) printf("tsuspend task1 error.\n");

    sc_tresume(1,0,&err);
    if (err!=0) printf("tresume task1 error.\n");
    sc_tdelete(0,0,&err);
}
void task1()
{
   int j,err;
   while(1)
    {
      sc_putc('1');
      for (j=20000;j;--j);
    }
```

```
}

void task2()
{
    int j;
    while(1)
    { sc_putc('2');
      for (j=20000;j;--j);
    }
}

void main()
{
  int err;
  sc_tcreate(user_main,25,1,&err);
}
```

第五章 VRTXsa 的内存布局与分配管理

嵌入式软件是操作系统与应用软件一体化的软件，其内存管理比较简单。本章描述了 VRTXsa 的存储布局和 VRTXsa 提供的两种内存分配管理：分区(Partition)和堆(Heap)。如图 5-1 可见 VRTXsa 中的内存管理

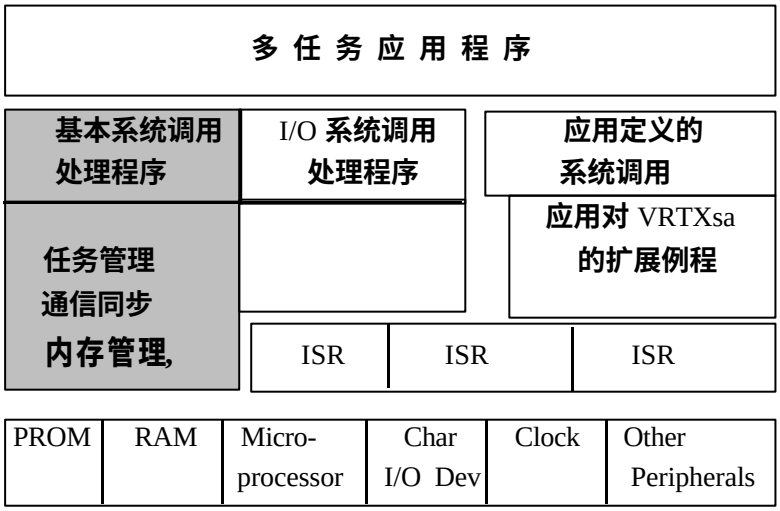


图 5-1 VRTXsa 中的内存管理

5.1 存储布局

基于 VRTXsa 的应用系统的存储映象如图 5-2 所示，它包括以下一些模块：

1. VRTXsa 代码

   VRTXsa 代码可以被放在随机存储器或只读存储器中。.

2. 可装入的应用模块

包括应用代码，应用自定义的系统级代码，以及与这些代码相关的静态变量。

3. VRTXsa 工作区

包括 VRTXsa 的系统变量，中断栈，所有对象的控制结构，空闲任务栈，和为系

统里每个任务分配的栈空间。VRTXsa 负责建立和管理栈及初始化和管理 TCB。

4. 堆管理

VRTXsa 自动分配一个 ID 为 0 的堆来作为其工作区。

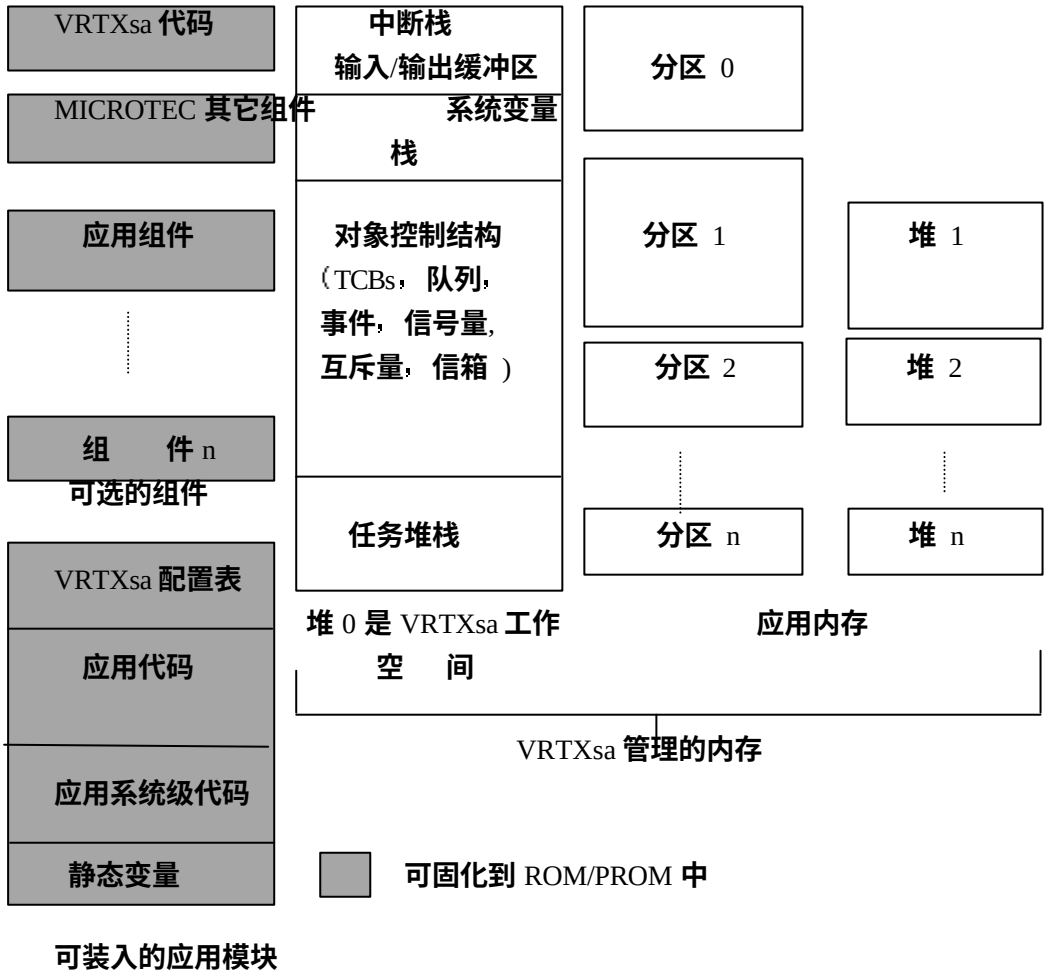


图 5-2 VRTXsa 应用系统的存储映象

5. VRTXsa 管理的应用内存

这部分内存是由应用提交给 VRTXsa 来统一管理的。任务和 ISRs 通过调用 VRTXsa 提供的系统调用，动态地获得和释放的一个或多个分区或堆。

6. 可选的 MICROTEC 组件

如:用于调试，文件 I/O 和支持多处理器的组件。

自己提供的可选组件。

5.2 VRTXsa 中的内存分配

任务在运行过程中对内存的需求是不断变化的，不同的任务有不同的需要。OS 将内存当作一种资源来看，并且在竞争的任务之间分配这种资源，就如同在竞争的任务间分配 CPU 控制权一样。

嵌入式实时操作系统内核通常使用下述两种方法进行内存分配：固定尺寸存储块的静态分配和可变尺寸存储块的动态分配。VRTXsa 对这两种内存分配方法都提供了支持，它提供了分区和堆两种内存分配管理机制。

5.3 分区管理

分区管理采用的是静态的内存分配方法，系统分配和回收固定大小的存储块。其优点是存储块的分配和回收时间是确定的，因为不会出现存储碎片，因而也不需要做回收存储碎片，进行合并等工作。

此外，分区存储分配系统还提供了很大的灵活性，不但具有时间确定、开销小的优点，而且具有可变存储块系统的大部分优点。系统中，可定义多个分区，每个分区中存储块的大小可不同，且可在分区中定义分区。例如，一个分区可以是另一个分区中的一块。这就意味着存储块可以很容易地被分为子存储块。另外，还可以为同一块存储区域定义两个分区，这样就可以从同一块存储区中分配两种不同尺寸的存储块。这样做要求在分配一种尺寸的存储块时，必须释放所有的另一种尺寸的存储块。

VRTXsa 的一个分区如图 5-3 所示

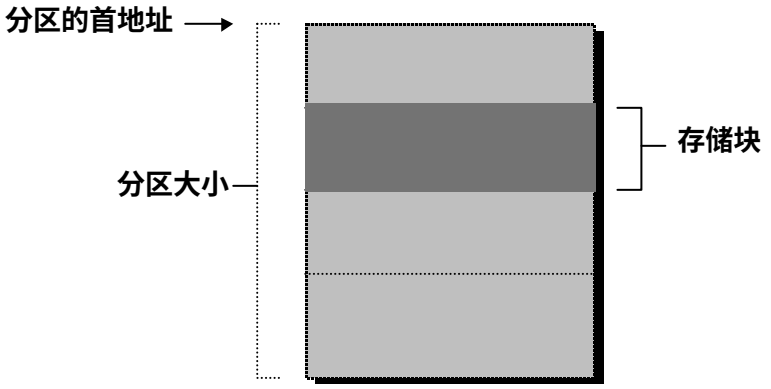


图 5-3　VRTXsa 的分区

表 5-1 列出了 VRTXsa 分区管理的系统调用。

1. 从分区中获得存储块

函数原型:

char *sc_gblock(int pid, int *errp)

任务调用 sc_gblock 可从一个指定分区号为 PID 的分区中，获得一个存储块，返回值是存储块的地址。

2. 释放存储块回分区

函数原型:

void sc_rblock(int pid, char *blockp, int *errp)

sc_rblock 将已申请到的存储块释放回指定的分区中。当一个任务被删除时，任务的存储块并不会自动地释放，因为存储块可能已传递给其他的任务用做数据交换了。因此，在删除任务之前，应当使用 sc_rblock 释放所有的存储块。

表 5-1 VRTXsa 分区管理的系统调用。

调用名	功能描述
sc_gblock	从指定的分区中获得一个存储块
sc_rblock	将一个存储块释放回指定的分区
sc_pcreate	创建一个指定地址、大小，ID 号，存储块大小的分区
sc_pextend	将指定的分区扩展一指定的长度
sc_pinquiry	查询分区的状态
sc_pdelete	删除一指定的分区。

3. 创建分区

函数原型:

int sc_pcreate(int pid, char *paddr, unsigned long psize,
unsigned long bsize, int *errp)

sc_pcreate 将应用的一块连续的存储区定义为分区。调用中的参数指明了分区的起始地址，分区的大小，分区的 ID 号，和块的大小。最初用 sc_pcreate 调用定义的分区中的数据块最多有 32K 个。但你可以使用 sc_pextend 调用对分区进行扩展。数据块的尺寸应大于 0，小于或等于分区的大小。为了避免浪费空间，

分区的大小应当是存储块尺寸的整数倍。

4. 扩展分区

函数原型:

```
void sc_pextend(int pid, char *paddr, unsigned long psize,
                int *errp)
```

sc_pextend 将一个已创建的指定分区扩展一个指定的长度。.扩展的部分可与该分区在物理地址上不一定是相互邻接的，因此，须指定扩展的首地址。在扩展的存储区中最多能有 32K 个存储块。可使用多个 sc_pextend 调用来获得更多的存储块。

5. 查询分区状态

函数原型:

```
void sc_pinquriy(unsinged long info[3], int pid, int *errp)
```

sc_pinquiry 返回分区的状态，如当前已分配的存储块数，未分配的存储块数，和存储块的大小等。如果分区有扩展，那么计数值就是原有的分区和所有扩展区的总和。

6. 删除分区

函数原型:

```
void sc_pdelete(int pid, unsigned int opt, int *errp)
```

sc_pdelete 调用删除指定的分区，使分区的 ID 号和该分区的所有存储区可以重新分配。

5.5 堆管理

VRTXsa 的堆管理可动态分配变长的存储块，并且能够有效的处理存储碎片。与在非实时系统中广泛应用的一种动态存储分配法伙伴系统相比，克服了时间的不可确定性的缺点。

表 5-2 列出了 VRTXsa 堆管理提供的系统调用

表 5-2 VRTXsa 堆管理提供的系统调用

调用名	功能描述
sc_hcreate	创建一个连续存储的堆
sc_halloc	从指定的堆中申请一存储块
sc_hfree	将已申请的存储块释放回堆中

sc_hinquiry **查询堆的状态**
sc_hdelete **删除一指定的堆**

1. **创建堆**

函数原型:

```
int sc_hcreate(char *haddr,unsigned long hsize,
                unsigned log2_psize, int *errp)
```

sc_hcreate **调用创建一个具有连续存储空间的堆，堆的首址为** haddr, **大小为** hsize, **页的大小为** 2 **的** log2_psize **幂次。并且返回堆的** ID **号。**

2. **从堆中申请存储块**

函数原型:

```
char *sc_halloc(int hid, unsigned long bsize, int *errp)
```

sc_halloc **调用从系统管理的一指定堆中分配一存储块。**sc_hcreate **调用在创建堆的同时，指明了页的大小。**sc_halloc **调用指明想要的存储块的尺寸。可以重复** sc_halloc **调用直到堆中的所有存储块都分配完。**

3. **释放存储块回堆中**

函数原型:

```
void sc_hfree(int hid, char *blockp, int *errp)
```

sc_hfree **调用将一个已经申请到的存储块返回到指定的堆中去。在使用** sc_tdelete **删除任务之前，应使用** sc_free **调用将所有的存储块释放回堆，因为** VRTXsa **在删除任务时，并不会自动地释放存储块。**sc_hfree **调用会检查给出的存储块地址是否在指定堆的地址范围内，给出的地址是否是一个已分配的存储块的地址。**

4. **查询堆的状况**

函数原型:

```
void sc_hinquriy(int info[3], int hid, int *errp)
```

sc_hinquiry **调用得到当前已分配的存储块数目，空闲的存储块数目，和** log2 **堆的页尺寸。**

5. **删除堆**

函数原型:

void sc_hdelete(int hid, int opt, int *errp)

sc_hdelete 调用删除一指定的堆，使堆的控制块和所有用于它的系统存储可用。

如果想要申请的存储块尺寸小于或等于堆的页尺寸，那么 sc_halloc 和 sc_hfree 调用不会引起重调度。反之,如果想要申请的存储块尺寸大于页的尺寸，且另一个任务又正在访问堆，就会引起重调度。如果调用 sc_hinquiry 时，有另外的任务正在访问堆结构，也会引起重调度。

5.6 使用 VRTXsa 的存储分配系统调用

5.6.1 块地址

VRTXsa 的 sc_gblock 和 sc_halloc 调用返回一个块地址给调用者。调用程序要为这些调用的输出分配一个指针变量(如 block)：

```
block=sc_gblock(pid,&err)
block=sc_halloc(hid,bsize,&err)
```

当执行 sc_rblock 和 sc_hfree 调用的时候，调用程序将这个地址传回给系统。例如：

```
sc_rblock(pid,block,&err);
sc_hfree(hid,block,&err);
```

5.6.2 分区地址

VRTXsa 动态地从分区中为任务分配存储块。程序可以动态地创建和扩展分区。在系统调用 sc_pcreate 和 sc_pextend 中的 paddr 参数是一个分区或扩展区的绝对起始地址。

C 程序将这个参数定义为指针型，然后使用类型转换为它分配一个绝对地址。例如：

```
char * paddr ;          /*declare paddr to be a character pointer */
paddr=(char*)0XAE00 ; /*assign paddr the value AE00 */
                        /*converting that integer to a pointer to a character */
sc_pcreate(pid,paddr,psize,bsize,&err);  /*make the call*/
```

5.6.3 分区和块的尺寸

系统调用 sc_pcreate 和 sc_pextend 使用 psize 参数来指明分区的长度。sc_pcreate 调用用 bsize 参数来指明块的长度。这些参数必须被声明为长整型。

```
long psize ,bsize;        /* declare psize and bsize as long */
psize=0X1000L;            /* give psize the value 0X1000L */
bsize=0X800L ;            /*  give bsize the value 0X800L */
sc_pcreate (pid,paddr,psize,bsize &err);    /*call VRTXsa*/
```

5.6.4 堆地址

sc_hcreate 调用使用 haddr 参数来作为堆的起始地址。为了避免浪费空间，地址与页的边界对齐。

5.7 内存管理实验

```
/* 360/QM –68360，4M RMA 目标板 */
#include <vrtxil.h>
#include <stdio.h>
#define HEAP_START 0X700000

void main()
{
  char *paddr,*block,*buf,*haddr;
  int pid,err,hid;
  int infoh[3];
  unsigned long psize,bsize;
  unsigned long infop[3];

  pid=1;
  psize=0x1000L;
  bsize=64L;
  paddr=(char *)0x780000;

  printf("\n\r ===>Type any key to continue.\n");
  sc_getc();
  printf("\n\n");
  printf("             Program:");
  printf("YSH2---Memory Layout and Allocation\n");
  printf("\n");
```

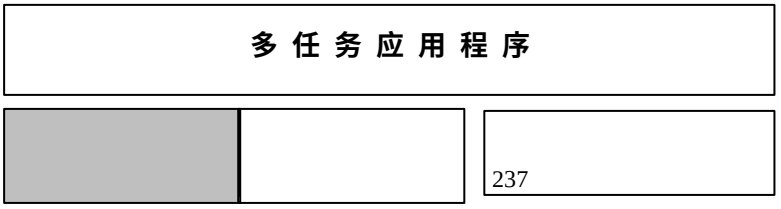
```
sc_pcreate(pid,paddr,psize,bsize,&err);/* create partiton */

paddr=(char *)0x785000;
sc_pextend(pid,paddr,psize,&err); /* extend partition */
block = sc_gblock(pid,&err); /* get a block */
if (err!=0) printf("gblock error\n");
printf("                Gblock succeed.\n\n");
printf("                Block is %x.\n",block);
sc_pinquiry(infop,pid,&err);
printf("                Infop[0] is %x.\n",infop[0]);
printf("                Infop[1] is %x.\n",infop[1]);
printf("                Infop[2] is %x.\n\n",infop[2]);
sc_rblock(pid,block,&err); /* return the block */
if (err!=0)
printf("                rblock error");
haddr=(char *)HEAP_START;
hid=sc_hcreate(haddr,9000,0,&err);
printf("                The heap's ID number is %r.\n",hid);
buf=sc_halloc(hid,512,&err);
printf("                The buffer is %x.\n",buf);
sc_hinquiry(infoh,hid,&err);
printf("                Infoh[0] is %x.\n",infoh[0]);
printf("                Infoh[1] is %x.\n",infoh[1]);
printf("                Infoh[2] is %x.\n",infoh[2]);
buf=sc_halloc(hid,512,&err);
printf("                The buffer is %x.\n",buf);
sc_hinquiry(infoh,hid,&err);
printf("                Infoh[0] is %x.\n",infoh[0]);
printf("                Infoh[1] is %x.\n",infoh[1]);
printf("                Infoh[2] is %x.\n",infoh[2]);
buf=sc_halloc(hid,128,&err);
printf("                The buffer is %x.\n",buf);
sc_hinquiry(infoh,hid,&err);
printf("                Infoh[0] is %x.\n",infoh[0]);
printf("                Infoh[1] is %x.\n",infoh[1]);
printf("                Infoh[2] is %x.\n",infoh[2]);
buf=sc_halloc(hid,128,&err);
printf("                The buffer is %x.\n",buf);
sc_hinquiry(infoh,hid,&err);
printf("                Infoh[0] is %x.\n",infoh[0]);
printf("                Infoh[1] is %x.\n",infoh[1]);
printf("                Infoh[2] is %x.\n",infoh[2]);
sc_hfree(hid,buf,&err);
printf("\n");
```

```
  sc_hinquiry(infoh,hid,&err);
  printf("                Infoh[0] is %x.\n",infoh[0]);
  printf("                Infoh[1] is %x.\n",infoh[1]);
  printf("                Infoh[2] is %x.\n",infoh[2]);
  sc_hdelete(hid,0,&err);
  sc_pdelete(pid,0,&err);
  sc_tdelete(0,0,&err); /* delete self  */
}
void spawn_main()
{ int err;
  sc_tcreate(main,25,1,&err);
}
```

第六章　任务间的通信与同步

如图 6-1 **可见** VRTXsa **中的任务间的通信与同步机制**



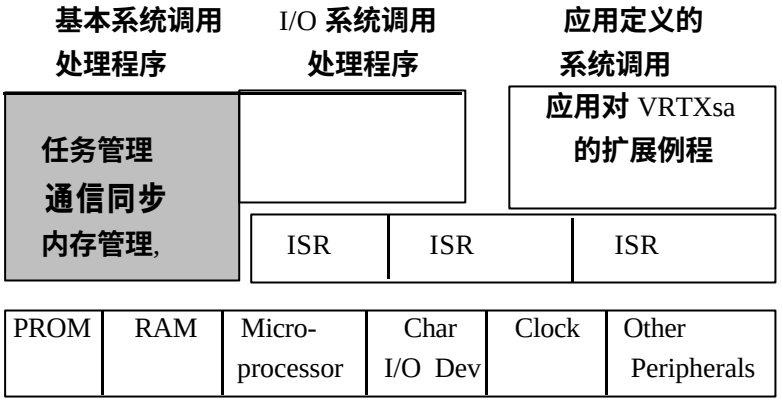


图 6-1　VRTXsa 中的任务间通信与同步机制

6.1 任务间的通信与同步简介

在多任务的实时系统中，一项工作的完成往往要通过多个任务或多个任务与多个中断处理过程(ISRs)共同完成。它们之间必须协调动作，互相配合，甚至需要交换信息－进行通信。这些通信和同步的需要是：

1. 任务能和其他任务及 ISRs 交换数据。
2. 任务能以以下方式与其他任务进行同步。
   - 单向同步：一个任务与另一个任务或一个 ISR 同步
   - 双向同步：两个任务相互同步
   - 与同步：　一个任务与几个事件同时同步
   - 或同步：　一个任务与几个事件中的任何第一个到达事件同步
3. 任务必须能对共享资源进行互斥访问。

为了满足任务间通信、同步和互斥的需要，VRTXsa 提供了邮箱、队列、事件组、信号量和互斥量这五种机制。任务和 ISRSs 可以通过邮箱和队列来传递 32 位的消息，对于较大的消息可以传递指针。任务和 ISRSs 也可以使用事件组来进行同步，使用信号量进行互斥。一个互斥对象（MUTEX）支持优先级继承以确保任务在对共享资源进行互斥访问的同时，不会破坏基于优先级的调度。

6.2 邮箱(Mailbox)

一个邮箱是应用在其内存定义的一个长字变量。允许任务传递 32 位的非 0

消息。VRTXsa 不创建邮箱，而是由应用在其内存中建立邮箱。应用应将邮箱初始化为一个恰当的值：为 0 时，邮箱可用，当邮箱用于互斥时为非 0。

6.2.1 邮箱的系统调用

表 6-1 例出了 VRTXsa 的邮箱系统调用。

表 6-1 VRTXsa 的邮箱系统调用

调用名	功能描述
sc_post	发送消息到一指定的邮箱
sc_pend	接收一指定邮箱来的消息，可永久等待或有限等待
sc_accept	接收一指定邮箱来的消息，当邮箱为空时，不挂起调用者

1. 发送消息到一指定的邮箱

函数原型:

```
void sc_post( char **mboxp, char *msg, int *errp)
```

任务或 ISRs 用 sc_post 调用发送一条消息到指定的邮箱。如果邮箱中已有一条消息（邮箱值非 0），VRTXsa 返回一个错误代码。如果有任务在等待消息就唤醒优先级最高的等待消息的任务，使它获得消息，从挂起态转为就绪态。

2. 从邮箱中接收消息

函数原型:

```
char *sc_pend(char **mboxp,long timeout, int *errp)
char *sc_accept(char **mboxp, int *errp)
```

任务发出 sc_pend 调用来接收消息。如果邮箱中有消息（邮箱值非 0），任务接收消息并且继续执行。消息接收后，VRTXsa 将邮箱清空。如果邮箱中无消息（邮箱值为 0），任务就会被永久挂起(timeout=0)直到有消息到达，或有限时间挂起(timeout≠0)等待消息的到达，如果在这个时间间隔之内没有消息到达，就返回出错信息任务继续执行。

当任务用 sc_accept 从邮箱中接收消息而邮箱为空时，任务不会被挂起。相反 VRTXsa 返回一个错误代码，任务继续运行。为了避免挂起，ISRs 在接收消息时，必须使用 sc_accep 而不是 sc_pend。

当一个任务由于等待消息而被挂起时，又被 sc_tsuspend 调用挂起，如果接

收到消息了不会使任务解挂，必须调用 sc_tresum 后才能将它解挂。

可使用 VRTXsa 的邮箱调用来进行数据传送，同步和互斥。例如，为了同步任务 A 和 B，可以这样做：任务 A 发送一条消息到邮箱甲，然后立刻等待邮箱乙。任务 B 等待邮箱甲，并且向邮箱乙发送消息。

为了实现对共享资源的互斥访问，初始化邮箱为一个非 0 的值来锁住资源。需要使用这个资源的其他任务都会等待这个邮箱的钥匙。当任务完成了对资源的访问，它就归还钥匙，使能下一个任务。

6.2.2 邮箱系统的例子

C 程序将邮箱定义为指针变量。VRTXsa 调用将邮箱的地址作为输入参数。即一个指向指针的指针：

sc_post(&mbox,msg,&err);

mbox 是一个指针大小的邮箱，&mbox 是一个指向邮箱的指针。

6.2.3 邮箱系统实验

```
#include <vrtxil.h>
#include <stdio.h>

char *mbox[2]={0,0};

void user_main()
{
  extern void keyboard_interrupt_enable();
  void task1(),task2();
  int err;

  err=0;
  printf("\r\n ===>Type any key to continue.\n");
  sc_getc();
  printf("\n\r");
  printf("               \nProgram:");
  printf("vrtxdm3---mailboxes");
  printf("\n\n");

  err=0;
  sc_tcreate(task1,1,4,&err);
  if (err!=0) printf(" Tcreate Task1 error.\n");
```

```
  sc_tcreate(task2,2,4,&err);
  if (err!=0) printf("Tcreate Task2 error.\n");
  sc_tdelete(0,0,&err);
  }

 void task1()
{
  char ch,*msg;
  long timeout;
  int  i,j,err;

  ch='A';
  msg=&ch;
  timeout=0L;
  for (i=200000;i>0;i--);
  printf(" \n\r task1 post message 'A' to mbox[0].\n");
  sc_post(&mbox[0],msg,&err);
  for (i=50;i>0;i--)
  {
     sc_putc('1');
     for (j=20000;j>0;j--);
  }

  msg=sc_pend(&mbox[1],timeout,&err);   /*pend for msg*/
  for (i=200000;i>0;i--);
  printf("\n\r task1 accept message %c from mbox[1].\n",*msg);
  for (i=50;i>0;i--)
  { sc_putc(*msg);
    for (j=20000;j>0;j--);
  }
  sc_tdelete(0,0,&err); /* delete self*/
 }

void task2()
{
    char ch,*msg;
    int err,i,j;

    ch='B';
    msg=&ch;
    err=0;

    for (i=200000;i>0;i--);
    printf("\n\r task2 post message 'B' to mbox[1].\n");
```

```
    sc_post(&mbox[1],msg,&err);    /* post a message*/
    for (i=50;i>0;i--)
    {
        sc_putc('2');
        for (j=20000;j>0;j--);
     }
    msg=sc_pend(&mbox[0],0,&err);
    for (i=200000;i>0;i--);
    printf("\n\r task2 accept message %c from mbox[0].\n",*msg);
    for (i=50;i>0;i--)
    {
       sc_putc(*msg);
       for (j=20000;j>0;j--);
     }

    sc_tdelete(0,0,&err);  /*  delete self*/
}

void main()
{
  int err;
  sc_tcreate(user_main,25,1,&err);
  if (err!=0) printf("Tcreate error.\n");
}
```

6.3 队列(Queues)

消息队列是应用动态创建的具有固定长度的缓冲区。队列是系统管理的结构，用队列的 ID 号来引用它。队列允许任务传递长字（32 位）消息。（长度为 1 的队列从逻辑上来说就是一个邮箱）。

可使用队列进行几个同种资源的互斥访问。队列的长度相当于该种资源的数目，它决定了有多少个任务能够同时使用该种资源。

例如，假设在系统中有 5 台行式打印机。定义一个队列长度为 5，任务按优先级的顺序来获得使用这些打印机的权利。用打印机的 ID 号来初始化这个行式打印机队列。所有要使用行式打印机的任务都等待在这个队列上。当有一台打印机可用时，唤醒等待的最高优先级的任务接收打印机的 ID 号，并且使用打印机。当任务使用完毕时，它将打印机的 ID 号送回打印机队列。这样另一个任务可以使用该打印机了。

6.3.1 队列的系统调用

表 6-2 例出了 VRTXsa 提供的队列系统调用。

1. 创建队列

函数原型:

int sc_qcreate(int qid, int qsize, int *errp)

int sc_qecreate(int qid, int qsize, int opt, int *errp)

使用 sc_qcreate 和 sc_qecreate 调用，可在 VRTXsa 的工作区中创建一个具有指定 ID 号和指定长度的队列。用 sc_qcreate 调用创建的队列是任务的等待的顺序是基于优先级的。用 sc_qecreate 调用创建的队列，可指明它们是基于优先级还是基于 FIFO 的。

2. 发送消息到队列

函数原型:

void sc_qpost(int qid, char *msg, int *errp)

void sc_qbrdcst(int qid, char *msg, int *errp)

void sc_qjam(int qid, char *msg, int *errp)

任务用 sc_qpost,sc_qbrdcst 和 sc_qjam 调用来发送消息到队列中去。sc_qpost 调用将消息放在队尾。消息按照先进现出（FIFO）的方式来处理。如果队列已满，则系统将返回一个错误代码。

sc_qbrdcst 调用将广播一条消息到指定的队列，唤醒正在等待消息的所有任务。

sc_qjam 调用将一条消息放在队首。当用 sc_qcreate 和 sc_qecreate 调用创建一个任务时，系统将在指定长度的队列前面添加一个表项。这个添加的表项是为用 sc_qjam 发送的消息保留的。如果队列已满，还可用 sc_qjam 发送一消息到这个保留的表项中。这时如在执行 sc_qjam 操作，系统就会返回一个错误代码。可以用 sc_qjam 发送所有的消息到队列中。这样队列中所有的消息就只能按照 LIFO 的原则来处理。

表 6-2 VRTXsa 的队列系统调用

调用名	功能描述
sc_qcreate	创建一具有指定 ID 号和指定长度的队列

sc_qecreate	创建一具有指定 ID 号和指定长度的队列，并可确定任务等待是基于优先级还是按照 FIFO
sc_qpost	发送一条消息到指定的队列。
sc_qbrdcst	广播一条消息到指定的队列
sc_qjam	发送一条消息，将它塞到队列的起始端。
sc_qpend	接收一指定队列来的消息，可永久等待或有限等待
sc_qaccept	接收一指定队列来的消息，当队列为空时，不挂起调用者
sc_qinquiry	查询指定队列信息
sc_qdelete	删除指定的队列

3. **从队列中接收消息**

函数原型:

```
char *sc_qpend(int qid, long timeout, int *errp)
char *sc_qaccept(int qid, int *errp)
```

任务用 sc_qpend **和** sc_qaccept **调用来接收消息。如果任务用** sc_qpend **来接收消息，如果队列中有消息，任务接收消息并且继续执行。如果队列中无消息，任务就会被永久挂起**(**当** timeout=0)**直到有消息到达，或有限时间挂起**(timeout≠0)**等待消息的到达，如果在这个时间间隔之内没有消息到达，就返回出错信息任务继续执行。**

如果任务用 sc_qaccept **调用来接收消息，如果队列为空，任务不会被挂起。相反** VRTXsa **返回一个错误代码，任务继续运行。为了避免挂起，** ISRs **在接收消息时，必须使用** sc_qaccep **而不是** sc_qpend。

4. **查询队列状态**

函数原型:

```
char *sc_qinquiry(int qid, int *countp, int *errp)
```

sc_qinquiry **调用获得关于一个队列的信息。调用返回队列中消息的数目和在队首消息的内容。队首的消息返回给调用者，但并不会从队列中除去。**

5. **删除队列**

函数原型:

```
void sc_qdelete(int qid, int opt, int *errp)
```

sc_qdelete **调用删除指定的队列，使队列所占用的系统资源可用。**

6.3.2 使用队列系统调用应注意的问题

在队列中的消息可以按 FIFO/LIFO 顺序或者二者混合的顺序处理。对于 FIFO 顺序的消息使用 sc_qpost 发送。对于 LIFO 的消息，使用 sc_qjam.如果同时使用了 sc_qpost 和 sc_qjam，用 sc_qjam 调用传递的消息呈 LIFO 顺序，位于队列的前端，用 sc_qpost 传递的消息呈 FIFO 顺序在它们的后面。不要将消息的 LIFO/FIFO 与基于优先级或 FIFO 的任务等待搞混淆。

动态地创建和删除队列会产生存储碎片。VRTXsa 的队列完全存在于系统的工作区中。如果系统被配置成为多队列的，且动态地创建，删除和重创建不同长度的队列，就有可能在系统工作区中出现存储碎片。

6.3.3 队列实验

```
#include "vrtxil.h"
#include "stdio.h"

void user_main()
{
    void task1(),task2(),task3(),task4(),task5();
    int qid,qsize,err,opt,count;
    char *msg;
    char ch;

        ch='A';
    msg=&ch;

        printf("===>Type any key to continue\n");
    opt=sc_getc();
    printf("\n");
    printf("         \nProgram:");
    printf("VRTXDM4---QUEUE");
    printf("\n\n");

    err=0;
        qid=1;
    qsize=2;
    sc_qcreate(qid,qsize,&err);           /*create queue 1*/
    if (err!=0) printf("qcreate error.\n",err);
```

```
    printf("QUEUE1: ID=%d  SIZE=%d\n",qid,qsize);
        sc_qinquiry(qid,&count,&err);
    if (err!=0)printf("Inquiry queue1 error.\n");
    else
    printf("The number of message:%d\n",count );

        for (;qsize>0;qsize--)
         sc_qpost(qid,msg,&err);
        qsize=2;
        sc_qinquiry(qid,&count,&err);
    if (err!=0)printf("Inquiry queue1 error.\n");
    else
    printf("The number of message:%d\n",count );

        err=0;
        sc_tslice(50);
    sc_tcreate(task1,1,4,&err);
    if (err!=0) printf("crteate task1 error.\n");
        sc_tcreate(task2,2,4,&err);
    if (err!=0) printf("create task2 error.\n");
        sc_tcreate(task3,3,4,&err);
    if (err!=0) printf("create task3 error.\n");
        sc_tcreate(task4,4,4,&err);
    if (err!=0) printf("create task4 error.\n");
        sc_tcreate(task5,5,4,&err);
    if (err!=0) printf("create task5 error.\n");

    sc_tdelete(0,0,&err);       /*delete self */
}


void task1()
{
    char *msg;
    long timeout;
    int qid,count,err,i,j;
    char ch1;

    ch1='A';
    msg=&ch1;

        qid=1;
        timeout=0L;
    sc_qpend(qid,timeout,&err);
```

```
        if (err!=0) printf("qpend error.\n");
        for (i=50;i>0;i--)
          { sc_putc('1');
            for (j=20000;j>0;j--);
           }
    sc_qpost(qid,msg,&err);
    if (err!=0) printf("Post message error.\n");
    sc_qinquiry(qid,&count,&err);     /*queue 1 inquiry */
    if (err!=0) printf("Inquiry queue1 error.\n");
    else
    printf("\n message number after task1 :%d\n",count);

       sc_tdelete(0,0,&err);         /*delete self */
}
void task2()
{
    char *msg;
    long timeout;
    int qid,count,err,i,j;
    char ch;

    ch='A';
    msg=&ch;

        qid=1;
        timeout=0L;
    sc_qpend(qid,timeout,&err);
        if (err!=0) printf("qpend error.\n");
        for (i=30;i>0;i--)
          { sc_putc('2');
            for (j=20000;j>0;j--);
           }
    sc_qpost(qid,msg,&err);
    if (err!=0) printf("Post message error.\n");
    sc_qinquiry(qid,&count,&err);     /*queue 1 inquiry */
    if (err!=0) printf(" Inquiry queue1 error.\n");
    else
    printf("\n message number after task2:%d\n",count);

       sc_tdelete(0,0,&err);         /*delete self */
}
void task3()
{
    char *msg;
```

```
    long timeout;
    int qid,count,err,i,j;
    char ch;

    ch='A';
    msg=&ch;

        qid=1;
        timeout=0L;
    sc_qpend(qid,timeout,&err);
        if (err!=0) printf("qpend error.\n");
        for (i=20;i>0;i--)
          { sc_putc('3');
            for (j=20000;j>0;j--);
           }
    sc_qpost(qid,msg,&err);
    if (err!=0) printf("Post message error.\n");
    sc_qinquiry(qid,&count,&err);     /*queue 1 inquiry */
    if (err!=0) printf("Inquiry queue1 error.\n");
    else
    printf("\n message number after task3:%d\n",count);

       sc_tdelete(0,0,&err);         /*delete self */
}
void task4()
{
    char *msg;
    long timeout;
    int qid,count,err,i,j;
    char ch;

    ch='A';
    msg=&ch;

        qid=1;
        timeout=0L;
    sc_qpend(qid,timeout,&err);
        if (err!=0) printf("qpend error.\n");
        for (i=50;i>0;i--)
          { sc_putc('4');
            for (j=20000;j>0;j--);
           }
    sc_qpost(qid,msg,&err);
    if (err!=0) printf("Post message error.\n");
```

```
    sc_qinquiry(qid,&count,&err);     /*queue 1 inquiry */
    if (err!=0) printf("Inquiry queue1 error.\n");
    else
    printf("\n message number after task4:%d\n",count);

       sc_tdelete(0,0,&err);         /*delete self */
}
void task5()
{
    char *msg;
    long timeout;
    int qid,count,err,i,j;
    char ch;

    ch='A';
    msg=&ch;

    qid=1;
    timeout=0L;
    sc_qpend(qid,timeout,&err);
    if (err!=0) printf("qpend error.\n");
        for (i=50;i>0;i--)
          { sc_putc('5');
            for (j=20000;j>0;j--);
           }
    sc_qpost(qid,msg,&err);
    if (err!=0) printf("Post message error.\n");
    sc_qinquiry(qid,&count,&err);     /*queue 1 inquiry */
    if (err!=0) printf("Inquiry queue1 error.\n");
    else
    printf("\n message number after task5:%d\n",count);
    sc_tdelete(0,0,&err);         /*delete self */
}

void main()
{
  int err;
  err=0;
  sc_tcreate(user_main,253,1,&err);
}
```

6.4 事件标志(Event Flag)

一个事件标志组是 VRTXsa 工作区中的一个 32 位的变量。32 位中的每一位都是表示一个事件标志。事件标志有两种状态：设置（1）和清除（0）。当一个标志处于设置状态时，表示相关的事件已经发生了。任务和 ISRs 可以使用事件标志来向其他任务发送信号，表示事件已发生。

6.4.1 事件标志提供的同步特征

1. 任务可以“或”的方式等待事件的发生。换句话说，任务指明一组要等待的事件。只要有一个事件发生了，任务就处于就绪态。
2. 任务可以“与”的方式等待事件的发生。即任务指明一组要等待的事件，直到所有的事件都发生了，任务才处于就绪态。
3. 多个任务可以等待同一个事件。

6.4.2 事件提供的系统调用

表 6-3 列出了 VRTXsa 提供的事件标志系统调用。

表 6-3 VRTXsa 的事件标志系统调用

调用名	功能描述
sc_fcreate	创建一个事件标志组，将事件标志组 ID 号返回给调用者。
sc_fdelete	删除一个指定的事件标志组。
sc_fpost	发送一个或多个事件到指定的事件组中去。
sc_fpend	以“与”/“或”方式等待一指定事件组中一个或多个事件。可以有限等待或永久等待。
sc_fclear	清除一指定的事件组中的一个或多个事件标志。
sc_finquiry	查询指定事件标志组的状态。

1. **创建事件组**

函数原型:

```
int sc_fcreate(int *errp)
```

sc_fcreate **调用在** VRTXsa **工作区中创建一个** 32 **位的事件标志组，并且返回事件标志组** ID **号。**

2. **删除事件组**

函数原型:

```
void sc_fdelete(int group_id, int opt, int *errp)
```

sc_fdelete **调用删除一个事件标志组，使它的控制块可用。删除时有可能仍有任务等待这个事件标志组。可指明只在没有任务等待时删除，或者是强制删除。在后一种情况中，所有等待的任务都回到就绪态。**

3. **等待事件**

函数原型:

```
void sc_fpend(int group_id, long timeout, int mask,
              int opt, int *errp)
```

任务用 sc_fpend **调用等待一个或多个事件。任务需要指明是** AND **等待还是** OR **等待。如果指定的事件标志已设定，任务就继续执行（**sc_fpend **调用并不清除事件标志）。如果指定的事件标志没有被设定，任务被挂起。可指明是有限等待还是永久等待。如果是有限等待，在这个时间间隔内事件仍未发生，**VRTXsa **返回一个错误代码** TIMEOUT **给任务，任务就继续运行。如果等待的事件标志组被删除，任务就会回到就绪态，且** VRTXsa **返回一个错误代码给任务。**

4. **发送事件**

函数原型:

```
void sc_fpost(int group_id, int mask, int *errp)
```

任务和 ISRs **用** sc_fpost **调用发送一个或多个事件。如果** sc_fpost **调用满足了任务的** AND **或** OR **等待，就唤醒等待任务。如果一个事件标志已经为** 1**（被设置），而** sc_fpost **想再一次设定它，系统就会返回一个错误代码。当** sc_fpost **指明了一组事件标志，并且它们中的一些已经被设置，**VRTXsa **返回一个错误代码，并且设置先前没有设置的事件标志。**

5. **清除事件标志**

函数原型:

int sc_fclear(int group_id, int mask, int *errp)

sc_fclear **调用清除事件标志。再次发送事件标志之前，应当先清除该标志。**

6. **查询事件组**

函数原型:

int sc_finquiry(int group_id, int *errp)

任务可以用 sc_finquiry **调用来检查事件标志组的状态，它返回一个** 32 **位的事件标志组给调用者。**

6.4.3 事件组实验

```
#include <vrtxil.h>
#include <stdio.h>

int group_id,ev_group;

void user_main()
{
  void task1(),task2();
  int err;

  printf("\n   ==>Type any key to continue.\n\n");
  sc_getc();
  printf("\n\r        Program :");
  printf("VRTXDM5---Event Flag\n\n");

  err=0;
  group_id=sc_fcreate(&err);        /*create event flag group */
  if (err!=0) printf("create event flag error.\n");
  else
    printf("\n\r group_id=%d\n",group_id);  /* print event flag id */

  sc_tcreate(task1,1,4,&err);
  if (err!=0) printf("create task1 error.\n");
  sc_tcreate(task2,2,4,&err);
  if (err!=0) printf("create task2 error.\n");

   sc_tdelete(0,0,&err);

}
```

```
void task1()
{
    int i,j,err;
    long timeout;

    err=0;
    timeout=0L;

    sc_fpend(group_id,timeout,5,0,&err); /* OR pend on event flag */
    printf(" \n\r Task1:I have got event flag!\n");

    ev_group=sc_finquiry(group_id,&err); /* return event flag state */
    if (err!=0)
        printf("inquiry event flag error.\n");
    for (j=200000;j>0;j--);
      printf("\n\r current event flag state :ev_group1=%d\n",ev_group);
    for (i=30;i>0;i--)
    { for (j=50000;j>0;j--);
      sc_putc('A');
    }
      sc_putc('\n');
      sc_putc('\r');

     sc_fclear(group_id,1,&err);    /* clear event flag bit 1 */
     if (err!=0)
        printf("clear event flag error.\n");
     for (j=200000;j>0;j--);
     printf("\n\r Task1:I have cleared event flag!\n");

  sc_fdelete(group_id,0,&err);
  sc_tdelete(0,0,&err);
}

void task2()
{
  int i,j,err;

  err=0;
  sc_fpost(group_id,5,&err);             /*signal bit 1 and bit 3 */
  if (err!=0) printf("post event flag error.%x\n",err);
  else
    printf("\n\r Task2:I have posted event flag!\n");
  ev_group=sc_finquiry(group_id,&err); /* return event flag state */
```

```
  if (err!=0) printf("inquiry event flag error.\n");
  for (j=200000;j>0;j--);
  printf("\n\r current event flag state :ev_group2=%d\n",ev_group);

  for (i=30;i>0;i--)
  {  for (j=50000;j>0;j--);
     sc_putc('B');
  }
     sc_putc('\n');
     sc_putc('\r');

  sc_tdelete(0,0,&err); /*delete self*/
}


void main()
{   int err;
    sc_tcreate(user_main,253,1,&err);
    if (err!=0) printf("create user_main error.\n");
}
```

6.5 信号量(Semaphore)

VRTXsa 为互斥提供计数信号量。一个计数信号量是 VRTXsa 工作区中的一个 16 位的变量。初始值可以是 0~65535。初始值为 0 表示资源开始处于锁住状态。一个非 0 的值表示有多个资源，供多个任务访问。

表 6-4 列出了 VRTXsa 提供的信号量系统调用。

6.5.1 系统调用

1.创建信号量

函数原型:

int sc_screate(int sem_init, int opt, int *errp)

sc_screate 调用是在 VRTXsa 工作区中创建一个信号量，并且返回一个信号量的 ID 号。在调用中指明信号量的初始值和任务是按优先级还是按 FIFO 顺序等待。

每一个信号量、互斥量、堆和事件标志组都与一个控制块相关。在 VRTXsa 的配置表中指明控制块的最大值。如果试图创建多于指明的控制块，系统就会返回一个错误代码。

表 6-4　VRTXsa 的信号量系统调用。

调用名	功能描述
sc_screate	创建一个信号量，并且返回一个信号量的 ID 号
sc_sdelete	删除一个信号量，使它的控制块可用。
sc_spend	等待信号量，可以永久等待或有限等待
sc_saccept	等待信号量，且不等待
sc_spost	发送信号量
sc_sinquiry	查询信号量的状态

2. 删除信号量

函数原型:

```
void sc_sdelete(int sem_id, int opt, int *errp)
```

sc_sdelete 调用删除一个信号量，使它占用的控制块可用。可以选择是否要强制删除，如果是强制删除，就要唤醒所有正在等待该信号量的任务。如果不选择强制删除，就要返回错误信息。

3. 等待信号量

函数原型:

```
void sc_spend(int sem_id, long timeout, int *errp)
void sc_saccept(int sem_id, int *errp)
```

为使用有限的资源，任务发出 sc_spend 调用。如果信号量的值不为 0，信号量就会减一，任务继续执行。如果信号量为零，任务就会被挂起。可以有限等待或永久等待。如果在有限等待时间到时，信号量仍然为 0，系统就返回一错误代码，任务就会继续执行。如果任务正在等待的信号量被删除了，任务就会回到就绪态，并且系统会返回一个错误代码给调用者。sc_saccept 调用，使信号量减一。如果信号量为零，不等待，因而 ISRs 可调用它。

4. 发送信号量

函数原型:

void sc_spost(int sem_id, int *errp)

任务或 ISRs **用** sc_spost **调用发发送信号量，使信号量加一。如果正有任务等待该信号量，就要唤醒该任务，使它回到就绪态，信号量的值保持不变。当信号量已经处于最大值** 65535 **时，再调用** sc_spost**，就会发生溢出，这时系统返回一个错误代码。**

当两个或更多的任务按照优先级方式等待一个信号量时，最高优先级的任务最先就绪。当两个或多个任务按照 FIFO **的方式等待一个信号量时，最早被挂起的任务最早就绪。**

5. **查询信号量**

函数原型:

int sc_sinquiry(int sem_id, int *errp)

任务和 ISRs **用** sc_sinquiry **调用查询一指定信号量的值。**

6.5.2 信号量实验

```
#include "vrtxil.h"
#include "stdio.h"
void putstr();
int sem_id1,sem_id2,sem_id3,sem_id4,sem_id5;
void user_main()
{
  extern void keyboard_interrupt_enable();
  void task1(),task2(),task3(),task4(),task5(),task6();
  int err,opt,sem_init;
  err=0;
  putstr("\n\n ==>Type any key to continue\n");
  opt=sc_getc();
  putstr("\n     Program:");
  putstr("VRTXDM6---Semaphore\n\n");
  err=0;
  sem_init=1;       /*inital  value of semaphore is 1 */
  sem_id1=sc_screate(sem_init,1,&err);
                   /* create semaphore;tasks pend in FIFO order*/
  if (err!=0) putstr("screate error.\n");
    /* return semaphore ID number and inital semaphore value */
  printf("sem_id1=%d\n",sem_id1);
  sem_id2=sc_screate(sem_init,1,&err);
  if (err!=0) putstr("screate error.\n");
```

```
  printf("sem_id2=%d\n",sem_id2);
  sem_id3=sc_screate(sem_init,1,&err);
  if (err!=0) putstr("screate error.\n");
  printf("sem_id3=%d\n",sem_id3);
  sem_id4=sc_screate(sem_init,1,&err);
  if (err!=0) putstr("screate error.\n");
  printf("sem_id4=%d\n",sem_id4);
  sem_id5=sc_screate(sem_init,1,&err);
  if (err!=0) putstr("screate error.\n");
  printf("sem_id5=%d\n",sem_id5);
  err=0;
  sc_tslice(50);
  sc_tcreate(task1,1,2,&err);
  if (err!=0) putstr("create task1 error.\n");
  sc_tcreate(task2,2,2,&err);
  if (err!=0) putstr("create task2 error.\n");
  sc_tcreate(task3,3,2,&err);
  if (err!=0) putstr("create task3 error.\n");
  sc_tcreate(task4,4,2,&err);
  if (err!=0) putstr("create task4 error.\n");
  sc_tcreate(task5,5,2,&err);
  if (err!=0) putstr("create task5 error.\n");
  sc_tcreate(task6,6,4,&err);
  if (err!=0) putstr("create task6 error.\n");
  sc_tdelete(0,0,&err);
}

void task1()
{    long timeout;
     int i,err;
     err=0;
     timeout=0L;
     sc_spend(sem_id1,timeout,&err);
     for (i=100000;i>0;i--);
     putstr("\n              NO.1:I have got left chopstick.\n");
     sc_spend(sem_id5,timeout,&err);
     sc_putc('\n');
      for (i=100000;i>0;i--);

     putstr("\n              NO.1:I have got right chopstick.\n");
     sc_putc('\n');
     for (i=100000;i>0;i--);
     putstr("\n              NO.1:I begin to eat...\n");
     sc_putc('\n');
```

```
     for (i=100000;i>0;i--);
     putstr("\n              NO.1: I have eaten,I begin to think!\n\n");
     sc_spost(sem_id1,&err);
     sc_spost(sem_id5,&err);
    sc_tdelete(0,0,&err);              /*delete self */
}

void task2()
{    long timeout;
     int  i,err;

     err=0;
     timeout=0L;
     sc_spend(sem_id1,timeout,&err);
     sc_putc('\n');
     for (i=100000;i>0;i--);
     putstr("\n              NO.2:I have got right chopstick.\n");
     sc_spend(sem_id2,timeout,&err);
     sc_putc('\n');
     for (i=100000;i>0;i--);
     putstr("\n              NO.2:I have got left chopstick.\n");
     sc_putc('\n');
     for (i=100000;i>0;i--);
     putstr("\n              NO.2:I begin to eat...\n");
     sc_putc('\n');
     for (i=200000;i>0;i--);
     putstr("\n              NO.2: I have eaten,I begin to think!\n\n");
     sc_spost(sem_id1,&err);
     sc_spost(sem_id2,&err);
     sc_tdelete(0,0,&err);              /*delete self */
}

void task3()
{    long timeout;
     int  i,err;

     err=0;
     timeout=0L;
     sc_spend(sem_id3,timeout,&err);
     sc_putc('\n');
     for (i=100000;i>0;i--);
     putstr("\n               NO.3:I have got left chopstick.\n");
     sc_spend(sem_id2,timeout,&err);
     sc_putc('\n');
```

```
    for (i=100000;i>0;i--);
    putstr("\n              NO.3:I have got right chopstick.\n");
    sc_putc('\n');
    for (i=100000;i>0;i--);
    putstr("\n              NO.3:I begin to eat...\n");
    sc_putc('\n');
    for (i=200000;i>0;i--);
    putstr("\n              NO.3: I have eaten,I begin to think!\n\n");
    sc_spost(sem_id3,&err);
    sc_spost(sem_id2,&err);
    sc_tdelete(0,0,&err);              /*delete self */
}

void task4()
{   long timeout;
    int  i,err;

    err=0;
    timeout=0L;
    sc_spend(sem_id3,timeout,&err);
    sc_putc('\n');
    for (i=100000;i>0;i--);
    putstr("\n              NO.4:I have got right chopstick.\n");
    sc_spend(sem_id4,timeout,&err);
    sc_putc('\n');
    for (i=100000;i>0;i--);
    putstr("\n              NO.4:I have got left chopstick.\n");
    sc_putc('\n');
    for (i=100000;i>0;i--);
    putstr("\n              NO.4:I begin to eat...\n");
    sc_putc('\n');
    for (i=100000;i>0;i--);
    putstr("\n              NO.4: I have eaten,I begin to think!\n\n");
    sc_spost(sem_id3,&err);
    sc_spost(sem_id4,&err);
    sc_tdelete(0,0,&err);              /*delete self */
}

void task5()
{   long timeout;
    int i, err;

    err=0;
    timeout=0L;
```

```
    sc_spend(sem_id5,timeout,&err);
    sc_putc('\n');
    for (i=100000;i>0;i--);
    putstr("\n               NO.5:I have got left chopstick.\n");
    sc_spend(sem_id4,timeout,&err);
    sc_putc('\n');
    for (i=100000;i>0;i--);
    putstr("\n               NO.5:I have got right chopstick.\n");
    sc_putc('\n');
    for (i=100000;i>0;i--);
    putstr("\n               NO.5:I begin to eat...\n");
    sc_putc('\n');
    for (i=100000;i>0;i--);
    putstr("\n               NO.5: I have eaten,I begin to think!\n\n");
    sc_spost(sem_id5,&err);
    sc_spost(sem_id4,&err);
    sc_tdelete(0,0,&err);              /*delete self */
}

void task6()
{
   int i,err;

   err=0;
   sc_sdelete(sem_id1,0,&err);
   sc_sdelete(sem_id2,0,&err);
   sc_sdelete(sem_id3,0,&err);
   sc_sdelete(sem_id4,0,&err);
   sc_sdelete(sem_id5,0,&err);
   for(i=200000;i>0;i--);
   putstr("\n                     delete all semaphores.\n");
   sc_tdelete(0,0,&err);
}

void putstr(str)
char *str;
{
  while (*str != '\0')
   {     if (*str == '\n')
            sc_putc('\r');
         sc_putc(*str);
         str++;
   }
}
```

```
  void main()
{
    int err;
    sc_tcreate(user_main,253,1,&err);
}
```

6.6 互斥量(Mutexes)

在多任务应用中，有可能出现两个或多个任务试图同时访问相同数据的情况，这将会导致数据破坏。为了保护共享数据，必须串行地对它进行访问。用于实现这种访问的机制称为互斥量。一个互斥量就是一个同步对象。用于多个任务（线程）串行访问共享数据。

VRTXsa 允许将互斥量创建成具有优先级继承的互斥量以避免优先级反转的情况。在任何时候，只有一个任务可以获得和拥有一个互斥量，也只有这个任务随后才能够释放这个互斥量。

6.6.1 系统调用

表 6-5 列出了 VRTXsa 提供的互斥量系统调用。

表 6-5　VRTXsa 中的互斥量系统调用。

调用名	功能描述
sc_mcreate	创建一个互斥量并且返回互斥量的 ID 号。
sc_maccept	申请锁住互斥量。如已锁住，不等待
sc_mpost	打开互斥量；
sc_mpend	申请锁住互斥量。如已锁住，有限等待或永久等待。
sc_minquiry	获得指明的互斥量的状态（锁住或打开）。
sc_mdelete	删除一指定的互斥量

互斥量有两种状态，锁住和打开。当用 sc_mcreate 调用成功地创建了一个互斥量时，系统返回该互斥量的 ID 号，该互斥量被初始化为打开状态。任何想要

访问与该互斥量相关的数据的任务都必须在调用中指明互斥量的 ID 号以获得该互斥量。

1. 创建互斥量

函数原型:

```
int sc_mcreate(unsigned int opt, int *errp)
```

sc_mcreate 调用创建一个互斥量并且返回该互斥量的 ID 号。互斥量在被创建时被初始化为打开状态。在创建时需指明等待该互斥量的任务是按照 FIFO 的顺序，还是优先级的顺序，还是按照带有优先级继承的优先级顺序。优先级顺序即具有最高优先级的任务最早获得该互斥量。FIFO 即任务按照它们被挂起的顺序获得该互斥量。带有优先级继承的优先级顺序与优先级顺序一样，只是采用了优先级继承技术，解决优先级反转问题。

2. 申请锁住互斥量

函数原型:

```
void sc_mpend(int mid, unsigned long timeout, int *errp)
void sc_maccept(int mid, int *errp)
```

sc_mpend 调用申请锁住一指定的互斥量。如果互斥量是打开的，那么该任务就锁住互斥量，继续执行。如果指定的互斥量已被锁住，任务会被挂起，这时候就会发生任务切换。只有在用 sc_mpost 调用打开互斥量，释放资源后，被挂起的任务才会处于就绪态。sc_mpend 调用可让任务选择是有限时间等待还是永久等待，如果等待超时，系统就返回一错误信息给任务，任务可以继续运行。

sc_accept 调用也是申请锁住一指定的互斥量，与 sc_mpend 调用不同的是，这个调用在互斥量已是锁住状态时，并不会挂起调用者。而是 VRTXsa 立刻返回错误代码给调用者，调用者可继续执行。中断处理程序就只能使用这个系统调用来申请锁住互斥量。

3. 打开互斥量

函数原型:

```
void sc_mpost(int mid, int *errp)
```

sc_mpost 调用打开一指定的互斥量。在调用 sc_mpost 解锁互斥量时，必须先用 sc_mpend 或 sc_maccept 调用锁住互斥量。如果有另一个任务正在等待该互斥量，该任务就会立刻得到互斥量，成为就绪状态。这时候，互斥量并没有被解锁，而只是被重新分配了。这种情况发生时，可能会引起重调度。

4. **查询互斥量状态**

函数原型:

int sc_minquiry(int mid, int *errp)

sc_minquiry **调用获得指定互斥量的当前状态（锁住或打开）。**

5. **删除互斥量**

函数原型:

void sc_mpend(int mid, int opt, int *errp)

sc_mdelete **调用删除指定的互斥量，可以选择是否强制删除，使该互斥量占用的控制块可用。**

6.6.2 实验

```
#include "vrtxil.h"
#include "stdio.h"

unsigned int mid;

void main()
{
   void task1();
   int err,opt;

   printf("\n\n ==>Type any key to continue \n");
   opt=sc_getc();
   printf("\n           Program:");
   printf("vrtxdm7---mutex\n");

   mid=sc_mcreate(2,&err);
         /* create a mutex,pend tasks in priority inheritance*/
   if (err!=0) printf("mcreate error.\n");
   else
   printf("mutex ID=%d\n",mid);

   sc_tcreate(task1,1,6,&err);      /* create task1 id=1 pri=6 */

   sc_tdelete(0,0,&err);
 }

void task1()
{
```

```
  void task2();
  int i,j,err;



  err=0;
  sc_mpend(mid,0,&err);        /* pend on restricted resource */
  if (err!=0) printf("pend mutex error.\n");

  for (i=50;i>0;i--)
  {sc_putc('c');
   for (j=2000;j>0;j--);
  }
  sc_putc('\n');
  sc_putc('\r');

  err=0;
  sc_tcreate(task2,2,5,&err);

  for (i=50;i>0;i--)
  {sc_putc('c');
   for (j=2000;j>0;j--);
  }
  sc_putc('\n');
  sc_putc('\r');

  sc_mpost(mid,&err);              /* unlock the mutex */
  for (i=200000;i>0;i--);
  for (i=50;i>0;i--)
  {sc_putc('c');
   for (j=2000;j>0;j--);
  }
  sc_putc('\n');
  sc_putc('\r');
  sc_mdelete(mid,1,&err);
  sc_tdelete(0,0,&err);

}

void task2()
{ void task3();
  int i,j,err;

  err=0;
```

```
   for (i=50;i>0;i--)
  {sc_putc('b');
   for (j=2000;j>0;j--);
  }
  sc_putc('\n');
  sc_putc('\r');

  err=0;
  sc_tcreate(task3,3,4,&err);

  for (i=50;i>0;i--)
  {sc_putc('b');
   for (j=2000;j>0;j--);
  }
  sc_putc('\n');
  sc_putc('\r');

  for (i=200000;i>0;i--);
  for (i=50;i>0;i--)
  {sc_putc('b');
   for (j=2000;j>0;j--);
  }
  sc_putc('\n');
  sc_putc('\r');
  sc_tdelete(0,0,&err);

}


void task3()
{
  int i,j,err;

  err=0;
  sc_mpend(mid,0,&err);        /* pend on restricted resource */
  if (err!=0) printf("pend mutex error.\n");
  for (i=200000;i>0;i--);
  for (i=50;i>0;i--)
  {sc_putc('a');
   for (j=2000;j>0;j--);
  }
  sc_putc('\n');
  sc_putc('\r');
  sc_mpost(mid,&err);
```

```
  sc_tdelete(0,0,&err);

}
void spawn_main()
{
    int err;
    sc_tcreate(main,253,4,&err);
}
```

第七章　中断处理

7.1 中断概述

7.1.1 VRTX 的中断

中断是硬件机制，它向 CPU 发信号，表示外部异步事件发生。异步事件是指无一定时序关系的随机发生的事件。如外部设备完成数据传输，实时控制设备出现异常情况等。

与应用任务相比，任务是由 VRTX 同步调度的；而中断处理程序是异步地执行的不由 VRTX 调度。当中断被触发时，中断处理程序就开始运行。

中断处理和 VRTX 管理的多任务环境是相分离的。无需 VRTX 的介入而直接进入 ISRs。当硬件检测到了一个中断，所有多任务活动都停止并且将 CPU 控制权交给指定的 ISRs。这个 CPU 的控制权由任务转到 ISRs 的过程完全由硬件来完成，VRTX 不会造成任何开销。

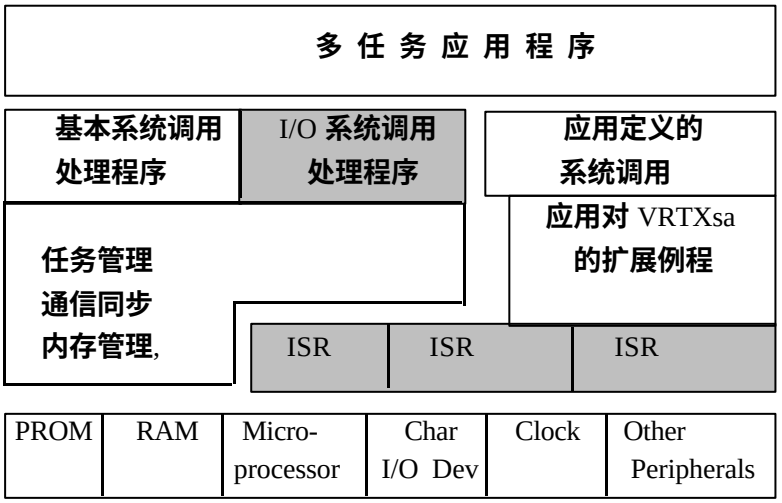


图 7-1　VRTXsa 中的中断管理机制

实时系统必须能快速地响应外部产生的中断，以成功地与外部环境进行交

互。每个 ISR 都必须保存和恢复在它执行时要使用的寄存器。以保证不会影响被中断代码的环境。

ISRs 和任务都可以使用系统调用。VRTX 必须能区分出是 ISRs 发出的系统调用还是任务发出的系统调用。这很重要，因为一个任务可以抢占其他的任务，但任务不能够抢占 ISRs。

VRTX 为应用的中断处理程序（ISRs）提供了一些系统调用，以便 ISRs 能够与任务进行通信，这样对中断的处理就可以由 ISR 和任务共同来完成，ISR 只做一些必要的操作如：输入数据，输出数据或将控制信息传递给任务，由任务来进行进一步的处理。如图 7-1 可见 VRTXsa 提供的中断管理机制。

7.1.2 VRTX32 对中断的处理：进入和退出中断

每个 ISR 都必须保存和恢复在它执行时要使用的寄存器。以保证不会影响被中断代码的环境。

ISRs 和任务都可以使用系统调用。VRTX 必须能区分出是 ISRs 发出的系统调用还是任务发出的系统调用。这很重要，因为一个任务可以抢占其他的任务，但任务不能够抢占 ISRs。为此, VRTX32 提供了中断进入 UI_ENTER 和中断退出 UI_EXIT 两个系统调用如表 7-1 所示。

表 7-1　VRTX32 为 ISR 提供的系统调用

调用名	功能描述
ui_enter	进入一个 ISR。
ui_exit	退出一个 ISR。当 ISR 没有嵌套时，就重调度，返回到优先级最高的就绪任务。

可以采用两种策略来使用 ui_enter 和 ui_exit。第一种也是最简单的一种是在每个 ISR 中使用它们。采用这种方法就不必在每一个中断程序中都考虑是否需要这个系统调用。它具有代码的可移植性，不会因漏掉 ui_enter 和 ui_exit 而导致不可预测的系统行为。缺点是在不需要 ui_enter 和 ui_exit 时,调用它们而降低系统性能。

第二种策略是只在需要的时候使用 ui_enter 和 ui_exit。这样做的一个很大的优点是具有好的系统性能。采用这个方法需要对这些调用如何工作有一个更透彻

的理解。

7.1.3 VRTXsa 的中断处理

VRTXsa 在中断发生时，不会自动地被调用。因此，VRTXsa 在中断发生的时候，不能保存任何寄存器。这就意味着 ISRs 必须包括保存和恢复它要修改的寄存器的代码。为了方便应用，VRTXsa 提供了一些方便的机制，以避免 ISR 在进入和退出中断时没有调用 VRTX 的系统调用，不能保护和恢复现场，通知 VRTX。

因此，它没有提供 ui_enter 及 ui_exit，而是用自己的小程序来代替，应用的 ISR 只需以函数的形式提供给它，由它统一的调用。

中断处理程序一般包括以下五部分：

- 一个 VRTXsa 专用的前导部分。主要保存必要的状态信息，为中断处理程序余下部分的执行做准备。
- 一个板和总线专用的前导部分，这一部分主要负责识别出中断向量。这部分可由硬件完成，如 80X86。
- 一个设备模块专用的前导部分，这一部分主要负责识别出一个或多个与给定的中断向量相关的中断源。如果不能确定中断源，那么这就是一个假中断。在多个中断源以相同的中断向量同时请求中断时，用软件来识别其中一个中断源，然后再开中断，第二个中断发生，这通常是比较容易的。但是，一些设备模块并不支持这样做，当识别了一个中断源时，所有其他具有相同中断向量的中断源也被识别。在这种情况下，软件必须一次处理所有的中断源。
- 一个中断源专用的程序体。它要从中断源接受状态信息和数据并且向中断源发送控制信息和数据。如果这个中断表示着重要的事件已经发生，它可能会调用 VRTXsa 的系统调用来唤醒一个任务。
- VRTXsa 专用的后续部分。这一部分要重新调度，恢复状态，并且返回到优先级最高的任务。

7.2 在中断中允许使用及不能使用的系统调用

7.2.1 允许使用的系统调用

一般来说，ISRs **中使用的系统调用的作用与任务是一样的。如:**ISR **使用** sc_post。sc_post **调用允许** ISR **就绪任务并且向它们传送消息。**sc_post

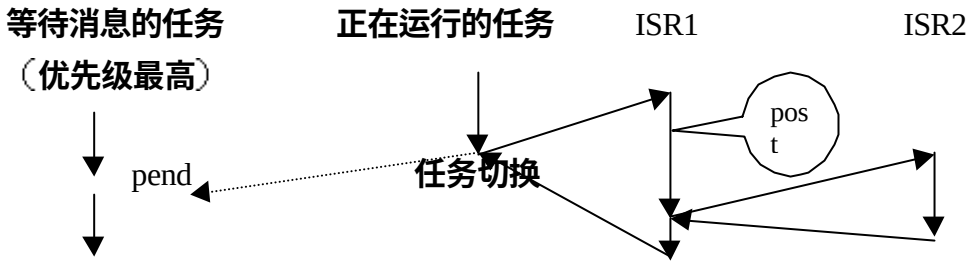


表 7-2 VRTXsa **允许** ISRs **使用的系统调用**

- **任务管理**
  sc_lock sc_tinquiry sc_tresume sc_unlock
- **存储分配**
  sc_gblock sc_rblock
- **通信和同步**
  sc_accept sc_post sc_qinquiry sc_saccept sc_fclear
  sc_qaccept sc_qjam sc_sinquiry sc_finquiry sc_qbrdcst
  sc_post sc_spost sc_fpost
- **中断支持**
  ui_enter ui_exit
- **实时时钟**
  sc_gtime sc_gclock sc_stime ui_timer
- **字符** I/O
  sc_accept ui_rxchr ui_txrdy
- **系统**
  sc_gversion

调用存放一个长字消息到指定的邮箱中去。在进行 sc_post **调用时，邮箱**

的内容应当为 0，否则系统会认为邮箱已用。当有任务在这个邮箱等待消息时，任务状态就会由挂起变为就绪。任务状态的改变是 sc_post 执行的结果，而不是退出中断时 ui_exit 的结果。如果要发生任务切换，切换只在最后一个 ui_exit 时发生。

当有多个任务在邮箱等待消息时，只有最高优先级的任务接收消息并且成为就绪态。如果没有任务在邮箱等待消息，sc_post 只是将消息放到邮箱中。

7.2.2 在 ISRs 中不允许使用的系统调用

ISRs 中不允许使用会影响任务、队列、堆或分区控制链的调用。ISRs 也不允许使用可能会导致当前运行环境挂起的调用。如表 7-3 所示：

表 7-3 ISRs 中不允许使用的系统调用

任务管理
sc_tcreate sc_tdelete sc_tpriority sc_tsuspend
sc_tecreate
存储分配
sc_pcreate sc_pinuiry sc_hcreate sc_hfree sc_pextend
sc_pdelete sc_halloc sc_hdelete sc_hinquiry
通信和同步
sc_fcreate sc_mcreate sc_pend sc_qpend
sc_fdelete sc_mdelete sc_qcreate sc_screate
sc_fpend sc_mpost sc_qecreate sc_sdelete sc_mpend
sc_minquiry sc_qdelete sc_spend sc_maccept
初始化
vrtx_go vrtx_init
实时时钟
sc_adelay sc_delay sc_sclock sc_tslice
字符 I/O
sc_getc sc_putc sc_waitc

7.3 中断处理程序和 8086 处理器

80x86 体系结构使用中断描述表（IDT）来控制对软中断、硬中断、异常、出错和陷阱的服务例程的访问。检查 IDTR 寄存器的内容或者全局描述表 GDT 中的选择子二，就会找到 IDT。

7.3.1 中断描述子表（IDT）

表 7-4　80X86 中断描述表

向量号	描述
0	除法出错
1	调试异常
2	NMI 中断
3	断点
4	溢出中断异常
5	越界异常
6	操作码无效异常
7	数字协处理器不能用异常
8	双异常
9	协处理器操作量段超限异常
10	任务状态段无效异常
11	段不存在异常
12	堆栈出错
13	一般保护异常
14	页面出错
15	（Intel 保留，没有使用）
16	数字协处理器异常
17	对齐检查
18-31	（Intel 保留，没有使用）
32-255	可屏蔽中断

在初始化过程中，寄存器 IDTR 被初始化指向一个 2K 存储区的起始。这个 2K 存储区称为中断描述表 IDT，它被分为 256 个单元，每个单元称为描述子。这 256 个描述子与所有可能的异常、错误、陷阱和中断相对应。描述子可以是任务门、中断门或自陷门。表 7-4 列出了 80X86 中断描述表的内容。

7.3.2 安装中断向量及初始化

要使用中断，首先就要安装中断处理程序，并为中断的产生准备一切工作：初始化设备，中断控制芯片等

在 VRTXsa 中使用 vsyslib 中的函数 sys_load_vrtx_isr 来安装一个中断处理程序。不应当直接修改 IDT 中的任何内容。当使用 sys_load_vrtx_isr 函数时，在 IDT 中安装了一个缺省的 VRTXsa 中断处理程序的中断门。缺省的中断处理程序在调用你的中断处理程序之前，要调用 ui_enter 调用。ISRs 的定位标志保存在 VRTXsa 内部的一张虚拟中断描述表中。如图 7-2 所示

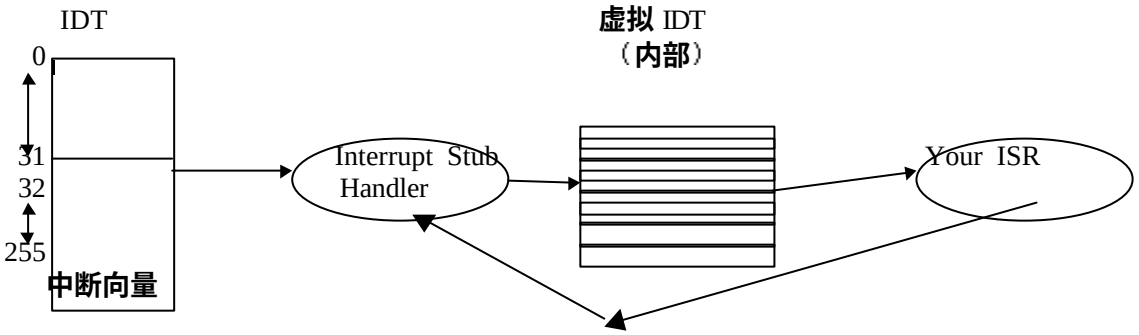


图 7-2　　IDT 和 ISRs

在安装中断向量的时候，应初始化所有可编程的中断控制器，如在嵌入 PC 和其他嵌入 80x86 硬件环境中常见的 8259。如果在 VRTXsa 的配置文件(即在 VRTXsa86\SRC 目录中的 devcnfxx.c 文件)中定义了 8259 这样的设备，VRTXsa 就会在初始化过程中初始化这个设备。对 8259 的定义应当根据不同硬件环境的参数作出相应的修改，包括主/从 PICs 的地址。

在初始化过程中，VRTXsa 保存了复位时 PIC 的状态。因此，如果在复位时，所有的中断都被屏蔽，它们就保持屏蔽直到被中断处理程序安装代码的初始化序列打开屏蔽为止。

应用的初始化程序，要在系统初始化的过程中执行，可根据实际的情况放在不同的位置上。

在 VRTX x86/spm 下应用程序可使用函数 create_idt_desc 来安装一个中断处理程序。 create_idt_desc 用来创建一个中断描述符，即设置 IDT 中某一项的内容。函数 create_idt_desc 的原型为 create_idt_desc(desc_sel,

isr_handler, type)。desc_sel 为中断选择子，isr_handler 为 ISR 的入口地址，type 指明中断类型等其他特性。

7.4 中断的实验

该实验是在 ACE360/QM 评估板上进行的，实现了实现 RISC 定时器 0 的中断处理程序

```
#include <stdio.h>
#include <vrtxvisi.h>
#include <vrtxil.h>
#include <stdlib.h>
#include <errno.h>
#include <syskind.h>
#include <compiler.h> /* turns on M68K flag for 68K targets */
#include <mriext.h>
#include "68360.h"


void report(int err)
{
 /* if you get an error, execute XRAY command UP to see from where */
   asm(" illegal");
}

struct CPM_REGISTER *CpmRegister ;
struct SIM_REGISTER *SimRegister ;
struct  RISC_Timer     *r_timer ;


LONG GetMbar(void) ;
LONG GetVbr(void) ;
void Setup68360(void) ;
void setcmpvect(LONG lIntMask, BYTE lIntNum, void (* func)() ) ;
void risc_tm0(void) ;
void risc_tm0_body(void) ;


/* timer ISR */
void risc_tm0(void)
{
    asm(" XREF _v90k_interrupt_enter ");
    asm(" jsr _v90k_interrupt_enter ");

     asm(" movem.l d0/d1/d2/a0/a1/a2,-(SP) ");

    asm(" jsr _risc_tm0_body ");
```

```
     asm(" movem.l (SP)+,d0/d1/d2/a0/a1/a2 ");

    asm(" XREF _v90k_interrupt_exit ");
    asm(" jmp _v90k_interrupt_exit ");
}
void risc_tm0_body(void)
{

    printf(" I have into the interrupt\n");
    CpmRegister->CISR |= 0x00020000 ;
    CpmRegister->RTER |= 0x0001 ;
}


LONG GetMbar(void)
{
     asm(" move.w #7,d0");      /* CPU space func code to d0    */
     asm(" movec.l d0,sfc");   /* load SFC for CPU space        */
     asm(" lea.l $3ff00,a0"); /* A0 points to MBAR              */
      asm(" moves.l (a0),d0"); /* get MBAR                          */
     asm(" andi.l #$ffffe000,d0") ;
}
LONG GetVbr(void)
{
     asm(" movec.l vbr,d0 ");
}

void Setup68360(void)
{
   CpmRegister = ( struct CPM_REGISTER * )( GetMbar() + CPM_BASE );
   SimRegister = ( struct SIM_REGISTER * )( GetMbar() + SIM_BASE );
   r_timer = (struct RISC_Timer *)(GetMbar() + Timer_BASE) ;
}

void setcmpvect( LONG lIntMask, BYTE lIntNum, void (* func)() )
{
      LONG  *pVec;

/* directly set the interrupt vector by modifying the EVT */
     pVec = ( LONG * )( GetVbr() + ((CpmRegister->CICR & 0x000000e0) + lIntNum ) * 4 );
      *pVec = ( LONG )func;

      CpmRegister->CIPR |= lIntMask;
      CpmRegister->CISR |= lIntMask;
      CpmRegister->CIMR |= lIntMask;
}

void main( void )
{
```

```
   int     err, i,j;
  struct CICR *cicr_p ;
  struct RTMR *rtmr_p ;
   printf("\n\rmain initializing.");

/* get the base address of CPM and SIM */
    Setup68360() ;


/*   set the vector of RISC_Timer0 , every a period of time ,
      it send a message to the mailbox which pend the disp task    */
   setcmpvect(0x00020000 , 0x11 , risc_tm0) ;

/*   setup the necessary registers of RISC_Timer0 and enable the interrupt */
/*
    SimRegister->PEPAR   |= 0x0080 ;
*/
    CpmRegister->RCCR     |= 0x3F00 ;
    r_timer->TM_BASE        = 0x0000 ;
    r_timer->TM_cnt         = 0x0000 ;
    CpmRegister->RTER       = 0xFFFF ;
   rtmr_p = (struct RTMR *)&CpmRegister->RTMR ;
    rtmr_p->T01 = 1 ;/*    = 0x0001 ;*/
    r_timer->TM_cmd         = 0xc00000ff ;
    CpmRegister->RCCR     |= 0x8000 ;
    CpmRegister->CR         = 0x0851 ;
}
```

该实验是在 INTEL386EX **评估板上进行的，实现了串口** 1 **的中断处理程序。该实验包括三个程序：**Initial.s , Isrio.c **及** Isr_io.s。

1．Initial.s **初始化程序**

```
name 'initial'
_TEXT segment er public
        public  initial
initial  proc near
          cli
             push bp
             push ax
             push dx
             mov  bp,sp
             mov  dx,21h
             in   al,dx
             and  al,0efh
             mov  dx,21h
             out  dx,al
             mov  dx,3fbh
             mov  al,80h
```

```
            out  dx,al
            mov  dx,3f8h
            mov  al,0ch
            out  dx,al
            mov  dx,3f9h
            mov  al,0
            out  dx,al       ;以上九条设置波特率为 9600 BIT/S
            mov  dx,3fbh
            mov  al,03h
            out  dx,al       ;8 位数据位，1 位停止位，无检验
            mov  dx,3fch
            mov  al,0bh
            out  dx,al       ;置 DTR,RTS 和 OUT2 有效，允许请求中断
            mov  dx,3f9h
            mov  al,01
            out  dx,al       ;只允许接收中断
            sti              ;布置完开中断
            pop  dx
            pop  ax
            pop  bp
            ret
initial  endp
_TEXT ends
      end
```

2．Isrio.c **任务的程序**

```
#include <vrtxil.h>

extern void initial();
extern void io_isr();

void user_main()
{
 void task1();
 int c=0,err=0;
 int tid1;
 initial();
 sys_load_vrtx_isr(0x34,io_isr);
 while(1) {
 }
}

void main()
{ int err;
```

```
  sc_tcreate(user_main,254,1,&err);
}
```

3. Isr_io.s **中断处理程序**

```
name 'io_isr'
_DATA         segment rw public
      err    dd 0
_DATA         ends
        assume ds:_DATA
_TEXT segment er public

public io_isr

  io_isr    proc near
                cli
                pushad
                mov  dx,3fdh
                in   al,dx                ;取线路状态
                test al,1eh               ;接收错吗?
                jnz  lerror               ;错,转错误处理
                mov  dx,3f8h
                in   al,dx                ;输入串口字符
                out  dx,al
                mov eax,0
  over:         mov al,20h
                mov dx,20h
                out dx,al
                popad
                sti
                ret
  lerror:       mov dx,3f8h
                in  al,dx
                mov eax,1
                mov err,eax
                jmp over
io_isr          endp
_TEXT  ends
            end
```

4. Isrio.def **文件**

```
@dnl ************************************************************
@dnl      Master .def file for VRTXsa x86/fpm
@dnl ************************************************************
```

```
make.root.name:     isrio
sys.verbose.major:  yes
sys.verbose.minor:  yes
sys.verbose.panic:  yes
make.target.debug:    yes
@dnl ************************************************************
@dnl      Include necessary VRTXsa x86/fpm Components
@dnl ************************************************************
ifx.enabled:        no
esh.enabled:        no
snx.enabled:        no
rtl.enabled:        yes
@dnl ************************************************************
@dnl      Include OS, target and tool definitions
@dnl ************************************************************
@include(system.def)
@include(ev386ex.def)
@include(microtec.def)
@dnl ************************************************************
@dnl            Set SYSTEM ENTRY POINTS, Application Modules     *
@dnl ************************************************************
sys.entry_point2:    main
sys.objs.usr:        isrio.obj,initial.obj,io_isr.obj,
@dnl ************************************************************
@dnl define Actual Devices for PC Target                          *
@dnl ************************************************************
@dnl  Remove serial_x if used for XRAY interface
@dnl  Add ether_1 if SNX enabled
board.devices:   board, timer_1
@dnl ************************************************************
@dnl x86/fpm Logical Devices                                      *
@dnl ************************************************************
@dnl vrtxos.console:       DEV_SERIAL_1
sys.env.devices:      board,timer
#default console for startup messages
@dnl sys.env.dev.console.value:       serial_1
@dnl ************************************************************
@dnl           Application Memory Requirements                    *
@dnl ************************************************************
xdm.size:       5000
code.size:      4f000
@dnl remove the following for 386ex systems
@dnl remove the following if PC RAM is 1MB or less
@dnl remove the following if total code+debugger size is < 640K
```

```
@dnl @include(pcmemhi.def)
@dnl ************************************************************
@dnl      RunTime Library Configuration
@dnl ************************************************************
@dnl rtl.hardware.fp_support:     no
```

第八章　计数器—定时器和字符 I/O

MICROTEC 提供了与 8254 兼容的定时器驱动程序的源代码。还提供了与 16450/550 兼容的 UARTS，PC/AT 键盘，和文本方式视频显示的源代码驱动程序。

8.1 计数器—定时器、字符 I/O 概述

尽管 VRTXsa 自身的操作并不需要计数器-定时器、字符 I/O 这些设备，但许多 VRTXsa 的应用需要计数器—定时器和字符 I/O 设备。VRTXsa 支持这些设备的集成。为此，VRTXsa 提供了用于实时时钟管理和字符 I/O 管理的系统调用。如图 8-1 可见 VRTXsa 提供的时钟和字符 I/O 管理。

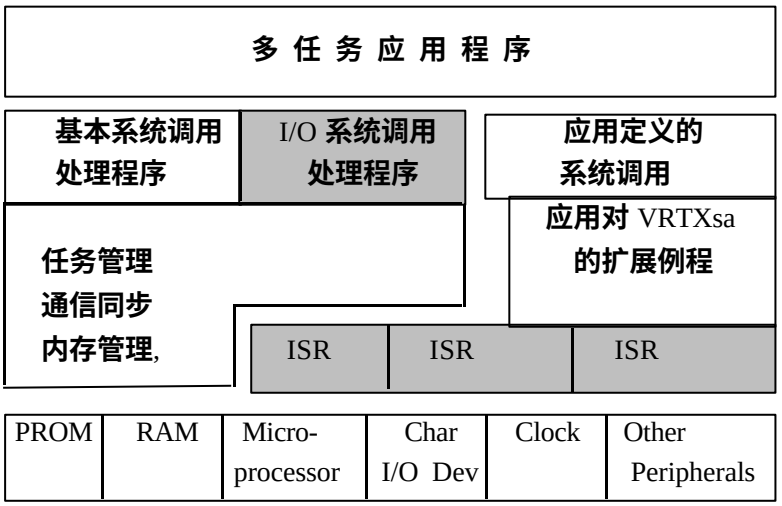


图 8-1　VRTXsa 中的时钟和字符 I/O 管理

VRTXsa 提供的实时时钟和字符 I/O 系统调用分为两种类型，一种是提供给使用这些设备的任务使用的，另一类是提供给管理这些设备的 ISR 使用的。表 8-1 列出了这两类调用。

表 8-1　VRTXsa 为任务和 ISR 提供的实时时钟

和字符 I/O 调用

实时时钟	
任务使用的调用：	**ISR 使用的调用：**
sc_gtime	sc_gtime
sc_stime	sc_stime
sc_sclock	sc_gclock
sc_gclock	ui_timer
sc_delay	
sc_adelay	
ui_timer	
sc_tslice	
字符 I/O	
任务使用的调用：	**ISR 使用的调用：**
sc_getc	ui_rxchr
sc_putc	ui_txrdy
sc_acceptc	
sc_waitc	

8.2 实时时钟管理

8.2.1 实时时钟

在实时系统中，一般不能缺少实时时钟，它是实时软件运行的必不可少的硬件设施。实时时钟单纯地提供一个规则的脉冲序列，脉冲之间的间隔可以作为系统的时间基准称为时基，时基的大小代表了实时时钟的精度，这个精度取决于系统的要求。

为了计准时间间隔，一个很重要的问题是 CPU 与时钟应同步工作。同步的方法可以用硬件，也可以用软件。软件方法是使 CPU 能用程序启动、停止时钟工作，设置时基的大小，并在启动后，利用实时时钟中断信号的方法来对准系统的时钟。每当实时时钟的时基到时，它就引起中断，中断响应后实时时钟又开始工作，时基到时又引起中断，这样达到与 CPU 的同步。显然，软件方法具有简单、灵活、易实现和低成本的优点，可以很方便修改实时时钟的设置和系统时间

的表示，且可以在不增加硬件的基础上非常灵活地用软件模拟多个“软时钟”；因此，在实时系统中广泛采用此种方法。但由于中断的延迟，对系统的时钟可能会造成一定的误差，因此在设计中通常将实时时钟中断的优先级设置的很高，一般仅次于掉电中断。系统的时间精度要求的越高，时钟中断的频度就越高，这样执行时钟 ISR 的时间就会增多，系统的开销就会增大，就会影响系统的其他的工作，因此，应使时钟 ISR 程序竟尽可能的短，同时要考虑时间精度。

由于嵌入式实时系统的硬件设备的多样化，实时内核提供的系统时钟服务就要适应这种灵活性的要求，它通常并不是以 ISR 的身份出现，而只是提供供应用的 ISR 调用的系统调用(在 VRTX 中是 ui_timer)，系统的时间精度完全由应用决定，其大小是调用 ui_timer 的时间间隔的大小即系统时基(tick)。ui_timer 完成系统计时，唤醒睡眠时间到和等待时间到的任务，时间片循环轮转调度等工作。除此以外，实时时钟管理还提供任务睡眠，设置和获取系统时间等系统调用。

8.2.2 VRTX 的时钟管理

VRTXsa 在系统中保存了一个 32 位的系统时钟，通过 VRTXsa 提供的系统调用 ui_timer 来计时。系统时钟从 0 开始计数，或者从应用设置的一个起始值开始计数。

为了提供年月日时分秒的系统时间，VRTXsa 提供了一些系统调用，可以设置起始系统时间，获得系统时间。系统时间都以 1970 年 1 月 1 日午夜为起始可把这个时间看成 0 秒，设置和获得的系统时间都是以秒为单位的值。

VRTXsa 的实时时钟管理提供如表 8-2 所示的系统调用。此外，VRTXsa 的实时时钟管理还支持 sc_pend，sc_qpend，sc_fpend，sc_spend，sc_mpend 和 sc_delay 调用中的超时处理。

1. 系统时钟加一

函数原型

```
void ui_timer(void)
```

在调用 vrtx_init 对 VRTX 初始化时，将 VRTXsa 的系统时钟设置成 0。每当发出一个 ui_timer 调用就将系统时钟加 1。ui_timer 调用通常都是由时钟中断程序发出的。32 位的 VRTXsa 系统时钟的取值范围是从 0 到 0XFFFFFFFF。在

vrtx_init 调用完成以后，只有 ui_timer 和 sc_stime 调用能够修改 VRTXsa 系统时钟的值。可以用 sc_sclock 调用来设置时钟。

表 8-2　VRTXsa 的实时时钟管理

调用名	功能描述
sc_stime	设置系统时钟。 即以时钟 TICK 数的形式设置
sc_gtime	获得系统时钟
sc_sclock	设置系统时间。以秒的形式设置
sc_gclock	获得系统时间
sc_delay	将调用任务的执行延迟指定的 TICK 数，（如果指定值为 0，那么就抢占该任务。）
sc_adelay	将调用任务挂起直到 VRTXsa 的系统时间
sc_tslice	打开/关闭同优先级按时间片循环轮转调度
ui_timer	系统时钟加一

2. 设置系统时钟

函数原型:

```
void sc_stime(long time)
```

将系统时钟设置成以系统时基 tick 为单位。

3. 获得系统时钟

函数原型

```
long sc_gtime()
```

获得系统时钟，其值为 tick 的倍数。

4. 设置系统时间

函数原型

```
void sc_sclock(struct timespec time, unsigned long ns,
               int *errp)
```

该调用将当前系统时间设置成 time 的值(以秒为单位)，系统的时基(tick)设置成 ns 的值(以纳秒为单位)。

5. 获得系统时间

函数原型

void sc_gclock(struct timespec *timep,unsigned long *nsp,
int *errp)

获得系统时间，该时间是以 1970 年 1 月 1 日午夜为起点的，以秒为单位。 还获得系统的时基。

6. 任务睡眠

函数原型

void sc_adelay(struct timespec time, int *errp)
void sc_delay(long timeout)

其中 sc_adelay 是使任务睡眠到一个绝对的系统时间(以秒为单位)值才唤醒它，而 sc_delay 则是使任务睡眠一段时间(以 tick 为单位)。

7. 打开/关闭同优先级按时间片循环轮转调度

函数原型:

void sc_tslice(unsigned short ticks)

当输入参数 ticks 不为 0 使，sc_tslice 调用打开同优先级的任务轮流执行调度，每个任务执行的时间片为设置的 ticks 参数值。如果 ticks 为 0，就关闭同优先级按时间片循环轮转调度

8.2.3 实时时钟实验

```
#include <vrtxil.h>
#include <stdio.h>

struct timespec
{  unsigned long seconds;
    unsigned long nanoseconds;
};
void main()
{ void task1(),task2();
  unsigned long ns;/* ns per tick for VRTXsa clock */
  long time;
  int err=0;
  char c='\0';
```

```
  struct timespec time_s;
    time_s.seconds=423000000;
    time_s.nanoseconds=0;
  ns=10000000;

  sc_stime(0L);
  time=sc_gtime();/* 'time' should be zero */
  printf("now time is %ld.\n\n",time);
  time=100L;
  sc_getc();
  sc_stime(time);
  sc_delay(time);
  time=sc_gtime();/* 'time'should be 200 */
  printf("now time is %ld.\n\n",time);
  sc_sclock(time_s,ns,&err);
  if (err!=0) printf("sc_sclock error %d.",err);

  sc_tslice(10);  /* enable time-slicing,slice=10 ticks */
  sc_tcreate(task1,1,4,&err);
  if (err!=0) printf("create task1 error %d.",err);
  sc_tcreate(task2,2,4,&err);
  if (err!=0) printf("create task2 error %d.",err);
  c=sc_getc();
  time_s.seconds=423000500;
  sc_adelay(time_s,&err);
  sc_gclock(&time_s,&ns,&err);
  if(err!=0) printf("Get clock error.%d\n",err);
  sc_tdelete(0,0,&err);/* delete self */
}
/*************************************************************************/
/* task1        - Prints '1' forever.                                    */
/*************************************************************************/
void task1()
{
    int   j,i,err;
    struct timespec time_s;
    unsigned long ns;
    i=10;
    while(i)
    {   sc_gclock(&time_s,&ns,&err);
         printf("\n Now time is %lud :%lud\n",time_s.seconds,time_s.nanoseconds);
         for(j=600000;j;--j);
         i--;
    }
```

```
}


/*************************************************************************/
/* task2        - Prints '2' forever.                                    */
/*************************************************************************/
void task2()
{
    int   j,i;
    i=600;

    while(i)
    {   sc_putc('2') ;
         for (j=20000;j;--j);
         i--;
    }
}
void spawn_main()
{ int err;
  sc_tcreate(main,10,1,&err);
}
```

8.3 字符 I/O 管理

8.3.1 系统调用

表 8-3 是 VRTXsa 提供的字符 I/O 管理的系统调用

VRTXsa 为 I/O 端口的输入和输出分别设置了 64 字符 FIFO 的输入缓冲区和 64 字符 FIFO 的输出缓冲区。并提供了用于任务和 ISR 的系统调用

1. 输出一个字符到输出缓冲区

函数原型:

void sc_putc(int chr)

sc_putc 调用将一个单独的字符放进输出缓冲区中。当缓冲区已满，调用任务就在等待，直到缓冲区只有 1/4 满为止。这是 VRTXsa 系统的缺省方式。为了与 VRTX32 兼容，系统还支持另一种方式，即只要输出缓冲区不满，任务就不等待。第一种方式可以减少任务切换的次数。

表 8-3　VRTXsa 字符 I/O 管理的系统调用

调用名	功能描述
sc_getc	从输入缓冲区中获得一个字符，当缓冲区为 空时，挂起调用者。
sc_acceptc	从输入缓冲区中获得一个字符，当缓冲区为空时，不挂起调用者。
sc_putc	输出一个字符到输出缓冲区
sc_waitc	从输入缓冲区中获得一指定的字符
ui_rxchr	ISR 将接收到的一个字符传送到系统的输入缓冲区中
ui_txrdy	ISR 发送一个传送准备好信号给系统，如果当系统输出缓冲区有字符时，就获得它。

2. **从输入缓冲区中获得一个字符**

函数原型

```
int  sc_getc()
char  sc_acceptc(int  *errp)
```

sc_getc **调用从输入缓冲区中获得一个字符。当缓冲区为空时，调用任务被挂起，直到在缓冲区中有一个字符为止。**

sc_acceptc **调用同样从输入缓冲区中获得一字符。但是，与** sc_getc **不同，当输入缓冲区为空时，这个调用并不挂起调用者。相反，系统立刻返回一个错误信息，调用任务继续执行。**

3. **从输入缓冲区中获得一指定的字符**

函数原型

```
int  sc_waitc(int  chr,  int  *errp)
```

sc_waitc **调用挂起任务，直到接收到一指定字符位置。系统一次只能管理一个** sc_waitc **调用。**

4. ISR **将接收到的一个字符传送到系统的输入缓冲区中**

函数原型

```
void  ui_rxchr(int  chr,  int  *errp)
```

ISR（通常是由应用的 BSP 提供）ISR 使用 ui_rxchr 调用将从 I/O 设备接收到的字符传递到 VRTXsa 的输入缓冲区中。

5. ISR 发出输出准备好信号给系统，并从系统的输出缓冲区中获得一字符

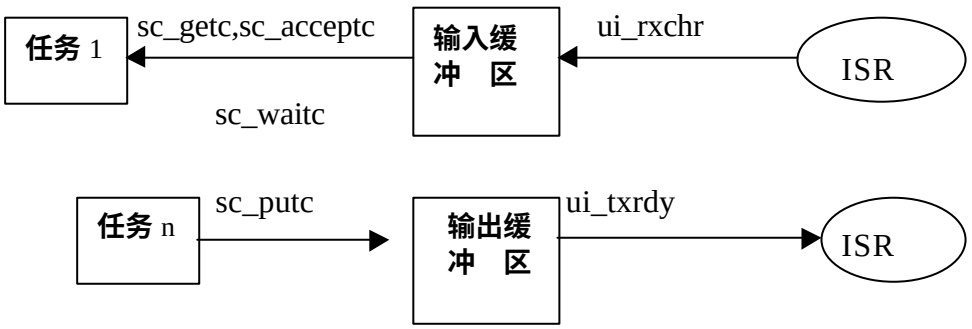
函数原型

char ui_txrdy(int *errp)

当输出设备发出一个传送准备好中断信号时，设备的 ISR 被触发。ISR 使用 ui_txrdy 调用告诉 VRTXsa 设备已经准备好。如果输出缓冲区中有字符，ui_txrdy 调用就将一个字符返回到 ISR。ISR 可以直接输出字符，也可以调用 TXRDY 驱动例程来输出字符。（ISR 没有必要一定使用 TXRDY 驱动例程来输出字符，但这样做可以使程序具有更好的结构）。TXRDY 驱动例程传送字符到输出设备，并且返回。

如果输出缓冲区为空，ui_txrdy 返回 ER-NCP 错误代码到 ISR。VRTXsa 知道了输出设备的准备好状态。这时，ISR 不会调用 TXRDY 例程，而只是退出。

尽管 VRTXsa 只有单端口的 I/O 调用，但可以创建自己的系统调用或者使用 IFX 实现多端口 I/O。



8.3.2 字符 I/O 实验

```
#include "vrtxil.h"
#include "stdio.h"

void user_main()
{
  int getch();
```

```
  int err,opt,c;
  err=0;
  printf("\n\n ==>Type any key to continue\n");
  opt=sc_getc();
  printf("\n     Program:");
  printf("VRTXsa demo---Character I/O\n\n");

  err=0;
  printf("please type 'go' to continue\n\n\n");
  sc_waitc('g',&err);/* wait for 'g' */
  if (err!=0) printf("waitc error.%d\n",err);
  sc_waitc('o',&err);/* wait for 'o' */
  if (err!=0) printf("waitc error.%d\n",err);

  printf("please type some characters,then you can see them input\n");
  printf("if you want to over,press ESC.\n\n");
  c=sc_acceptc(&err);/* get a char without pending the current task */
  if (err!=0x000B) sc_putc(c);
  while (c!=27)
    {   c=getch();
       sc_putc('\n');
        if (c=='\r') sc_putc('\n');
    }
    sc_putc('\r');
     sc_putc('\n');
  printf("This program is over.\n");
  sc_tdelete(0,0,&err);

}

int getch()
{
  int c;
  c=sc_getc();/* get the character */
  sc_putc(c);/* put the character */
  return(c);
}

void main()
{
    int err;
    sc_tcreate(user_main,253,1,&err);
}
```

第九章　配置和初始化

在大多数环境下，采用 XCONFIG **配置工具就可完成下面描述的配置和初始化工作。如果用** XCONFIG **来处理配置和初始化，这一章的内容具有很重要的参考价值。如果不使用** XCONFIG，**就必须提供初始化代码来完成下面描述的配置和初始化工作。**

$00	CFWSADDR	VRTX 工作空间地址
$04	CFWSSIZE	VRTX 工作空间大小
$08	CFSSTKSZ	系统堆栈大小
$0A	CFISTKSZ	中断栈大小
$0C	CFCBCOUNT	控制块数目
$0E	CFPARTCOUNT	分区数
$10	CFIDLE	Idle 任务堆栈大小
$12	CFUECOUNT	队列数
$14	CFDISLEV	组件屏蔽级
$16	CFUSTKSZ	用户堆栈大小
$18	CFMAXTID	最大任务标识号
$1A	CFTARGET	目标结构标识号
$1C	CFUTSKCT	应用任务数
$1E	CFOPTIONS	配置选项
$20	CFTXRDY	TXRDY 驱动程序地址
$24	CFTCREATE	TCREATE 扩展程序地址
$28	CFTDELETE	TDELETE 扩展程序地址
$2C	CFTSWITCH	TSWITCH 扩展程序地址
$30	CFCVTADDR	组件向量表 CVT 地址
$34	CFARSRVD	保留，必须=0
$36	CFENTCT	保留，必须=0
$38	CFETBCT	保留，必须=0
$3A	CFMFRCT	保留，必须=0

图 9-1　VRTXsa **配置表**

9.1 VRTXsa 的配置和初始化

系统初始化依赖于目标板的总体环境。它包括所有的预先要做的动作，这些动作使系统在应用执行前处在一个确定的状态。系统初始化主要包括设备初始化，初始化任务的创建和软件静态变量的初始化。VRTX/OS **初始化工作包括安**

装缺省的设备中断处理程序，使能中断，创建一个用做 OS 工作区的堆，初始化 VRTXsa 和应用。

VRTXsa 的初始化完成微处理器和系统工作区的初始化工作。板级支持包（BSP）完成剩余的初始化过程。VRTXsa 配置表是 BSP 的一部分，配置表中描述了系统环境。

9.2 VRTXsa 配置表(Configuration Table)

VRTXsa 配置表中的参数定义了特定的系统配置。图 9-1 列出了 VRTXsa 配置表的格式：

9.3 配置表参数

在系统初始化时，配置表中的参数向 VRTXsa 提供了以下一些信息。

- VRTXsa 工作区的位置和大小
- 任务栈的尺寸
- 事件标志组、堆、互斥量、信号量的控制块的数目
- 系统中的最大任务数
- 可选的 TXRDY 驱动程序的地址
- 可选的组件向量表的地址（CVT）

VRTXsa 配置表中的每一个参数的助记符在 vrtxvisi.ini 文件中都有定义

1. VRTX 工作空间地址(CFWSADDR)

指定了 VRTXsa 工作区的起始地址。工作区它包括 VRTXsa 的系统变量，系统中每个任务的 TCB 和栈，空闲任务栈，中断栈和所有队列，分区，事件标志组，堆，互斥量，信号量的控制结构。将其定位在一个长字的边界可以获得最佳的系统性能。

2.VRTX 工作空间大小(CFWSSIZE)

以字节为单位指定了 VRTXsa 可用的总的存储区的大小。

3. 系统堆栈大小(CFSSTKSZ)

这个参数再加上 CFUSTKSZ 用户堆栈大小就指明了 VRTXsa 为每个任务所分配的总的堆栈空间的大小。

当 VRTXsa 创建一个管态的任务时，系统为它分配一个栈，这个栈的总长等于上述两个参数的总和。当创建一个用户态的任务时，系统为它分配两个分离的栈系统栈和用户栈，两个栈的大小分别为 CFSSTKSZ 和 CFUSTKSZ。

CFSSTKSZ 参数指明了当任务（用户态或管态）调用 VRTX 系统调用所需的栈空间的最大尺寸。栈的尺寸依赖于任务使用的系统调用、任何扩展程序、应用提供的系统调用对栈的需要及调用其他组件需用的额外栈空间。其中 VRTXsa 需要 256 字节的栈空间。

4. 中断栈的大小(CFISTKSZ)

这个值必须足够大以容纳所有的 ISR 活动。在 vrtx_init 中，VRTXsa 在系统工作区中分配这个栈。对于 80x86 处理器来说，这个值为 0，因为 VRTXsa 不支持中断栈切换。

5. 控制块数(CFCBCOUNT)

这是一个可选的参数，它表示同时存在于系统中的事件标志组、堆、互斥量、和信号量的最大数目。每个事件标志组、堆、互斥量和信号量都与一个控制块相联系。如果在应用中不使用事件标志组、堆、互斥量和信号量，将这个参数设置为 0。

6. 分区数（CFPARTCOUNT）

它表示系统中的最大分区数。分区的 ID 号范围为 0 到 CFPARTCOUNT 减 1。

7. 空闲任务栈大小（CFIDLE）

它指明空闲任务栈的尺寸。在 vrtx_init 中，VRTXsa 从系统工作区中分配空闲任务栈。最好设置为 256 个字节。

8. 队列数目（CFQUECOUNT）

是个可选的参数，它表示系统中队列的最大值。队列的 ID 号范围为 0 到 CFQUECOUNT-1。

9. 组件屏蔽级(CFDISLEV)

在 VRTXsa/fpm 中不考虑 CFDISLEV。

10. **用户堆栈大小**（CFUSTKSZ）

这个参数再加上 CFSSTKSZ，**就表示每个** VRTXsa **任务堆栈的总容量。**

当创建一个处于用户态的任务时，系统为它分配两个栈，分别具有 CFUSTKSZ **和** CFSSTKSZ **的大小。**CFUSTKSZ **参数指明用来处理任务本地调用和临时变量存储区的大小。**

可使用 sc_tecreate **调用显式地分配栈，在调用中指明的栈的大小而不用配置表中的值。**

11. **任务最大标识号**(CFMAXTID)

定义了在系统中任务标识号的数目。在 VRTX32 **中，有效的任务标识号号** ID **从** 0~255。**为了与** VRTX32 **兼容，应将这个值设置为** 0。**当系统中的任务数**（CFUTSKCT）**小于** 255 **时，为了节省空间，应将这个值设置在** 1 **到** CFUTSKCT **之间。任务标识号** ID **的取值范围是从** 0 **到** MAX（**大于** 255，CFUTSKCT），**上界由这个参数的设置来确定。**

12. **目标结构标识号**(CFTARGET)

这个值指明了目标体系结构。Xconfig **配置工具自动地设置这个值。如果应用使用自己的初始化程序，必须在这里指明目标结构。查询**/include/target.h **可以获得** VRTX **所支持的所有目标体系结构的清单。**

13. **应用任务数**（CFUTSKCT）

表示系统中能够同时处于激活状态的任务的最大数目。VRTXsa **使用这个值来分配** TCB **和栈空间，并且决定有效的任务** ID **号的范围。**

14. **配置选项**(CFOPTIONS)

是个用位表示的屏蔽值，如表 9-1 **所示**

表 9-1　**配置选项位**

配置选项	**位**	**描述**
V32_OPTION_PUTC	**位** 0	=0 **推荐值。在这种情况下，**sc_putc **使用低/高水平线减少因输出缓冲区满而发生任务切换的次数。当输出**

		缓冲区已满，调用任务就等待，直到缓冲区只有 1/4 满为止。
		=1sc_putc 在这种情况下与 VRTX32 兼容。
V32_OPTION_	位 1	=0 vrtx_init 和 vrtx_go 被正常地使用
		=1 vrtx_init 只执行部分初始化。在这种情况下不支持 vrtx_go。这是为将来的多内核环境做准备的。
V32_OPTION_C_TXRDY	位 2	保留
		=1
V32_OPTION_C_HOOKS	位 3	保留

15. TXRDY 设备驱动程序地址(CFTXRDY)

是个任选的参数。这个参数指明了 VRTXsa 调用的 TSRDY 处理程序的地址。当输出设备准备好，并且 sc_putc 将一个字符放在了输出缓冲区中时，VRTXsa 就调用这个处理程序。可以参考 ui_txrdy 调用的说明部分。如果应用程序不使用 VRTXsa 的 I/O，就不必提供 TXRDY 驱动程序。在这种情况下，设置这个参数为空指针（0）。

16. TCREATE 扩展程序地址(CFTCREATE)

TDELETE 扩展程序地址(CFTDELETE)

都是可选的参数。这两个参数指明了当创建或删除一个任务时，应用提供的进行特别处理的扩展程序的地址。如果在应用中当创建或删除任务时，无需进行特别的处理，将这两个参数的值设置为空指针（0）。

17. TSWITCH 扩展程序地址(CFTSWITCH)

是个可选的参数。它表示当任务切换发生时，要调用的应用提供的扩展程序的地址。如果在任务切换时，无需进行特别的处理，将这个值设置为空指针（0）。

18. 可选的组件向量表 CVT 地址(CFCVTADDR)

是一个可选的参数。这个参数指明了组件向量表（CVT）的地址。CVT 将执行控制传给组件而不是 VRTXsa 。如果没有使用其他组件，将这个值设置为空指针（0）。

19. CFARSRVD，CFENTCT，CFETBCT 和 CFMFRCT

是为将来的扩展而准备的参数。应当将这些值都设置为 0。

9.4 如何决定 VRTXsa 工作区的大小

配置表中的参数 CFWSSIZE **指明了** VRTXsa **可用的总的存储空间的大小。工作区的大小是由栈需求、控制结构需求和内部变量需求决定的，如图** 9-2 **所示。**

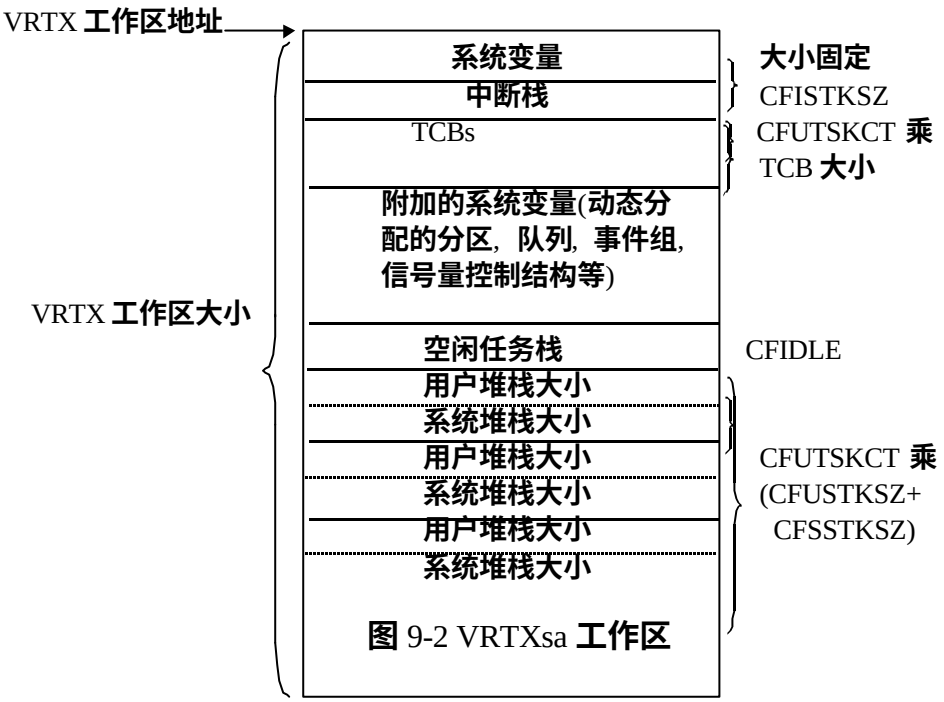


图 9-2 VRTXsa **工作区**

为 VRTXsa **分配的工作区越大，就可以使用越多的** VRTXsa **对象，如任务，队列，信号量，事件标志组，分区，互斥量，堆等等。有两种方法可以用于决定工作区的大小：估算和计算**

建议使用估算的方法来决定工作区的大小，因为这样做更为简单。系统同时提供了精确计算工作空间大小的公式。Xconfig **工具可以从配置表设置的参数中计算出工作区的大小。**

9.4.1 估算工作区大小

根据应用的特征要精确地计算出工作区的精确长度是可能的。但建议使用以下方法：

(1). 开始时先分配一个较大的工作区（最初的估算完全建立在应用之上）。
(2). 让应用运行一段时间，试着运行代码的所有路径。
(3). 重复第 2 步，减少工作区，直到出现出错代码 ER_MEM(存储空间不够)。
(4). 记下这时的工作区的尺寸，再加上百分之二十的当前尺寸，将这个值作为工作区大小的值。

9.4.2 计算工作区的大小

这一部分提供了如何计算 VRTXsa 工作区大小的公式。

这里给出的公式是近似公式。应当在这个基础上加上至少百分之二十的余量。对工作区中一些元素的存储需求的取值也是近似的，如栈的长度，当小于 512 字节时，近似为一个 2 的幂，当大于 512 字节时，近似为 512 字节的整数倍。

1. VRTXsa 工作区尺寸的计算公式

VRTXsa 用控制结构和内部变量来管理多任务。计算公式如下:

VRTX 工作空间大小

=((task+tcb+mbox+s+us)*t)+64p+2b+32e+64q+(qesize*qe)+
64cb+istk+getc+putc+hash+tid+tptr+pb+d

符号	定义
s	系统栈大小。缺省值在配置表中的 CFSSTKSZ 域中，其大小的确定参见表 9-3。
us	用户栈大小。缺省值在配置表中的 CFUSTKSZ 域中，其大小由任务子程序和变量的存储需求决定。
task	任务控制块的长度。这个值是依赖于处理器的。
tcb	线程控制块的长度。这个值是依赖于处理器的。
mbox	邮箱控制块的长度。这个值是依赖于处理器的。
t	最大任务数。VRTXsa 为每个任务分配任务控制块，线程控制块和邮箱控制块。
p	最大存储分区数。由配置表中的 CFPARTCOUNT 域来指明。每一个分区，不管是否被创建，都需要 64 字节的控制块。
b	存储块的最大数目。用每个分区或扩展区的长度除以存储块的长度就可以得到分区或扩展区的存储块的数目。

每个存储块都需要 2 个字节。每一个分区或扩展区需要的工作区的长度是这个分区或工作区中的存储块数目的两倍。如果分区中的存储块数目小于或等于 256，这个值就近似为一个最接近的 2 的幂的值，如果大于 256，这个值就近似为一个 512 的整数倍。

e 分区扩展的最大数目。由 sc_pextend 调用的次数决定。每一个扩展区都需要 20 字节，在分配的时候，近似为 32 字节。

q 系统中最大的队列 ID 号。由配置表中的 CFEQUECOUNT 参数决定。每一个队列，不管是否被创建，都需要 64 字节的控制块。

qesize 每一个队列元素的长度。除了 i386 系列的处理器需要 6 字节外，其余的处理器都需要 4 字节。

符号 定义

qe 队列元素的最大数目。每一个队列元素都需要 qesize 个字节。每个队列中的实际元素数目比在 sc_qcreate 和 sc_qecreate 调用中指明的数目大 1。因为在队列中要为 sc_qjam 调用保存一个额外的元素。

cb 事件标志组、信号量、互斥量和堆的控制块的最大数目。由配置表中的 CFCBCOUNT 参数指明。每一个控制块需要 64 字节，不管所对应的对象是否已经创建。

istk 中断栈大小，由配置表中的 CFISTKSZ 定义

getc sc_getc 输入缓冲区的大小，目前固定为 64 字节。

putc sc_putc 输出缓冲区的大小，目前固定为 64 字节。

hash 邮箱散列表中的每个表项都需要 4 个字节。表中的表项数目当前固定为 64，因此用于邮箱散列表的总的存储空间为 256 字节。

tid 两张任务 ID 表中的每一个表项都需要 2 个字节。任务 ID 数目是可变的，由配置表中的 CFMAXTID 参数决定。如果为了与 VRTX32 兼容将 CFMAXTID 设置为 0，任务 ID 表所需要的存储空间就为 512 字节，否则为 2*2*（CFMAXTID+1）字节。

tptr 任务指针表中的每一项都需要 4 字节；任务指针的数目是可变的，由配置表中的 CFMAXTID 参数决定。如果 CFMAXTID 为了与 VRTX32 兼容设置为 0，任务 ID 表所需要的存储空间为

1024 字节，否则为 4*（CFMAXTID+1）字节。

pb　拷贝任务参数块所需要的总空间。

d　系统线程所使用的存储空间。其大小如表 9-2 所示

表 9-2 系统线程的存储需求

描述	控制结构大小	栈的长度（i386/486）
Clock	tcb	512
Delay	tcb	512
Idle	tcb	CFIDLE
Delete	tcb	CFSSTKSZ
Initial task	task	——

2. 决定任务栈的大小

系统的任务栈的需求会影响系统配置表中的 CFUSTKSZ 和 CFSSTKSZ 参数。这些参数又会影响 CFWSSIZE 参数。

用户堆栈大小 CFUSTKSZ 是由两个因素决定的：任务中子程序调用次数和任务所需要的变量存储的大小。确定系统堆栈大小 CFSSTKSZ，必须考虑以下的因素：VRTXsa 的栈需求，应用是否定义了 VRTXsa 的扩充，应用是否定义了自己的系统调用处理程序，和是否使用其他组件。表 9-3 列出了系统堆栈大小的计算公式。

表 9-3　系统堆栈大小计算公式

公式	
系统堆栈大小=256+UX+USC+ISR+C	
符号	
___	256 字节表示 VRTXsa 的栈需求
UX	自定义的 VRTXsa 扩充需求
USC	自定义的系统调用处理程序需求
ISR	中断栈需求

C 总的组件调用栈需求，以字节为单位指明。

3. 计算工作区大小的例子

假设系统具有如下的配置：

10 个任务

一个 2048 字节的分区，存储块的大小为 64 字节=32 个存储块

一个 4096 字节的分区，存储块的大小为 512 字节=8 个存储块

对该分区的一个扩展， 也具有 4096 字节=8 个存储块

8 个队列，每一个队列都具有 10+1 的长度=88 个队列元素

2 个队列，每一个队列都具有 20+1 的长度=42 个队列元素

1 个事件标志组

1 个信号量

无自定义的扩展

无自定义的系统调用

无任务参数块

没有使用其他的组件

第一步是确定栈需求 us,istk 和 s。假设系统中所有任务的子程序调用和变量存储需求都小于等于 128 字节，即 us=128。假设中断需求为 428 字节，即 istk=428。

以下是对 s 的计算：

s=256+UX+USC+C=100+0+0+0=256 字节（十进制）

以下是对工作区大小的计算：

=2700+（(328+s+us）*t）+64p+2b+20e+64q+4qe+64cb+istk

=2700+((328+100+128)*10)+64*2+2*(32+8+8)+20*1+64*10+4*(88+42)+64*2+428

=2700+5560+128+96+20+640+520+128+428

=10220 字节（十进制）

9.5 系统初始化

注意: 以下的内容是为不使用 Xconfig 进行初始化的用户提供的。需要这些用户提供自己的初始化代码。

初始化包括对硬件设备和应用程序的初始化。当系统复位时就进行初始化。

在 VRTXsa 系统中，在 VRTXsa 初始化开始之前首先对 VRTXsa 的指针进行初始化。在 vrtx_init 之前，不应当进行中断。

一般都使用 Xconfig 配置工具来对系统进行配置和初始化，它会自动地执行以下的初始化步骤。

9.5.1 系统复位

大部分目标机都为系统复位提供一个特殊的信号，当这个信号出现时，CPU 就进行一些特定的动作。

9.5.2 设备初始化

设备初始化依赖于板级环境。通常在系统初始化的时候，要对定时器，字符 I/O 设备以及其他一些设备进行初始化。

设备初始化代码可以在 vrtx_init 调用之前或之后执行，但必须在 vrtx_go 调用之前执行。应当在对 VRTXsa 初始化之前先初始化一些设备，因为这些设备的成功运行是 VRTXsa 正常初始化的先决条件。例如，系统中有一个存储管理单元（MMU），在对 VRTXsa 初始化之前首先初始化它，因为系统的存储分配依赖于 MMU。

9.5.3 应用初始化

应用程序的初始化通常包括建立软件控制结构和与应用代码相关的变量，如邮箱、队列和布尔变量。同时，应用初始化代码通常要创建系统在 vrtx_go 调用之后要运行的任务。

必须在 vrtx_init 之后初始化需用 VRTXsa 服务的结构，如队列和存储分区。

9.5.4 VRTXsa 初始化

表 9-4 列出了 VRTXsa 提供的初始化系统调用。

表 9-4 VRTXsa 提供的初始化系统调用。

调用名	功能描述
vrtx_init	初始化 VRTXsa
vrtx_go	进入多任务状态，切换到应用任务运行

vrtx_init 系统调用执行以下的功能:

1. 定界 VRTXsa 工作区，并将其初始化为 0。
2. 为 sc_tcreate 调用建立和保存 TCB。
3. 建立每个任务的栈。
4. 建立中断栈。
5. 初始化其他 VRTXsa 内部变量。
6. 返回控制权到调用者。

返回值指出了 vrtx_init 在运行过程中是否遇到错误。vrtx_init 调用需要一个临时栈。在调用 vrtx_init 之前，建立一个小的（256 字节）的系统管态栈。

在 vrtx_init 调用之后调用 vrtx_go。这个调用切换到在初始化代码中创建的最高优先级的任务，使它运行，并且不会返回调用者。系统进入多任务状态。

9.5.5 在初始化过程中使用系统调用

初始化代码可以在 vrtx_init 之后 vrtx_go 之前使用以下的系统调用。在 vrtx_init 之前，应当避免开中断。建议在 vrtx_go 之前将中断关闭。初始化代码大多使用 sc_tcreate, sc_tecreate, sc_mcreate, sc_hcreate, sc_fcreate, sc_screate, sc_pcreate, sc_qcreae, sc_qecreate 系统调用。注意在 vrtx_init 之前不能使用 VRTXsa 系统调用，否则会造成难以预料的后果。

对 MICROTEC 其他组件的初始化应当在 vrtx_init 之后进行。在 vrtx_init 和 vrtx_go 之间允许使用的系统调用如表 9-5 所示

表 9-5 在 vrtx_init 和 vrtx_go 之间允许使用的系统调用

任务管理
sc_tcreate sc_tinquiry sc_tsuspend sc_tresume sc_tecreate sc_tpriority
存储分配
sc_gblock sc_hdelete sc_pcreate sc_pinquiry sc_halloc sc_hfree
sc_pdelete sc_rblock sc_hcreate sc_hinquiry sc_pextend
通信与同步
sc_accept sc_post sc_qinquiry sc_spost sc_fclear sc_qaccept

sc_qjam sc_sinquiry sc_fcreate sc_qbrdcst sc_qpost sc_maccept
sc_fdelete sc_qcreate sc_saccept sc_mcreate sc_finquiry sc_qdelete
sc_screate sc_mdelete sc_fpost sc_qecreate sc_sdelete sc_minquiry

实时时钟

sc_gblock sc_stime sc_tslice ui_timer sc_gtime

字符 I/O

sc_acceptc sc_putc(**最多只能使用** 64 **次，参见注意**)

系统

sc_gversion

注意:在初始化过程中使用 sc_putc **时要注意。这个调用可以用做显示开始提示字符。但在** vrtx_go **之前，在缓冲区中放入了多于** 64 **个字符时，会带来难以预料的后果。**

应用基础篇之二：网络协议

第一章 TCP/IP

TCP/IP 的网络技术是一种非常有效力的网络技术，它使我们能够处理多种复杂的基础通信技术。它把网络硬件的细节隐藏起来，提供一种高级通信环境，很好的解决了异种网的互连问题。也就是解决不同物理网络（异种通信子网，比如以太网与令牌环网）的互连问题。

1．1 网络的数据交换方式和服务的类型

交换方式其实质上是在交换设备内部将数据从输入线切换到输出线方式。它分为静态分配线路的线路交换方式和动态分配线路的存储转发方式（包括报文交换方式和分组交换方式）。其中分组交换方式根据是否采用连接可分为两类：A. 在有连接的子网中，连接称为虚电路。它需要一个建立连接的过程，即将从信源到信宿路径上所有主机表的相应表目串起来，便构成一条虚电路。B. 在无连接子网中的独立分组称为数据报，它携带了信宿地址，传输时子网对各数据报单独寻径，勿需建立连接。

在计算机网络协议层次结构中，层与层之间是完全单向依赖的，相邻层之间通过一组服务原语建立相互作用：下层是服务提供者，提供服务原语操作，上层是服务调用者，调用服务原语操作。下层向上层提供的服务分为两大类：面向连接的服务和无连接的服务。

面向连接服务是每次完整的数据传输都必须经过建立连接、使用连接、终止连接三个过程。在数据传输过程中，各数据分组不携带信宿地址，而使用连接号。本质上，服务类型中的连接是一个管道，发送者在一端放入数据，接受者从另一端取出数据。其特点是收发数据不但顺序一致且内容相同。

无连接服务是每个分组都携带完整的信宿地址，各分组在系统中独立传送。无连接的服务不能保证分组的先后顺序，由于先后发送的分组可能经不同路径去往信宿，所以先发的不一定先到。无连接服务甚至不进行损失分组的恢复和重传，不保证分组一定被收到或一定被正确收到。无连接不保证传输的可靠性。

1. 2 TCP/IP 协议概述

各种不同的网络技术和网络标准层出不穷。主要的网络标准有开放式系统互

连（OSI）模型、TCP/IP 标准、X.25 协议等。其中 TCP/IP 采用开放策略，适应了社会的需要，同时它与流行操作系统 UNIX 结合，已经成为事实上的网络协议标准。

对 TCP/IP 协议来说，TCP 提供传输层服务，IP 提供网络层服务。

TCP 提供给用户进程的一个可靠的全双工字节流的面向连接的协议。大多数 Internet 应用程序使用 TCP。

UDP 为用户数据报协议。这是一个用户进程的非连接协议，它不象 TCP 协议，它不能保证 UDP 数据报一定能到达其要求的目的地。

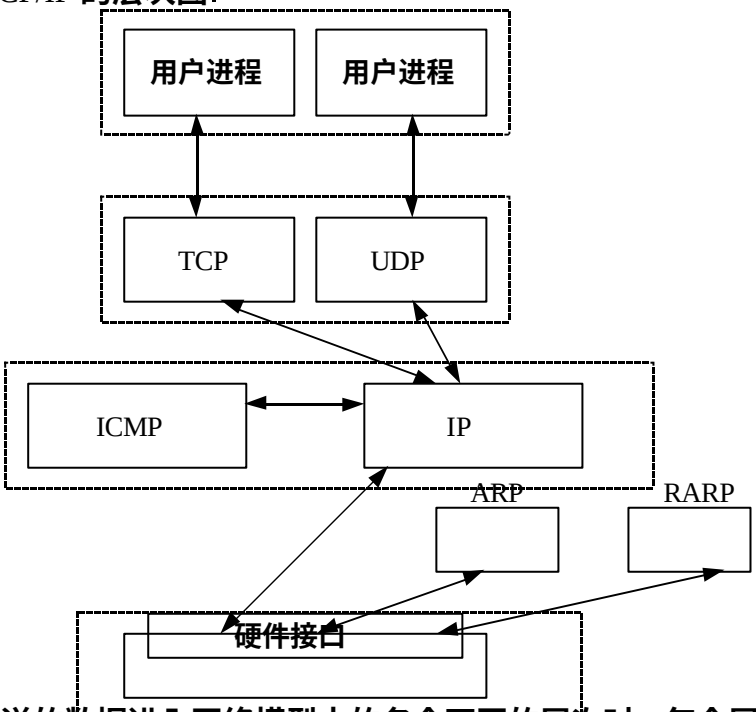
ICMP 为网间报文控制协议。它处理信关和主机间的差错与控制信息。

IP，网间协议。IP 协议是为 TCP、UDP 和 ICMP 提供分组发送服务的协议，用户进程通常不涉及到 IP 层。

ARP，地址转换协议。它将逻辑地址转化成物理地址。

RARP 反向地址转换协议。它将硬件地址转化为逻辑地址。

下图为 TCP/IP 的层次图。



当用户发送的数据进入网络模型中的各个不同的层次时，每个层次都要附加有关的控制信息，从而形成一个打包的过程，而在用户接受数据时，网络模型中的各层都要去掉有关的控制信息，最后仅将用户所需的数据交给用户，这叫拆包。如 TCP/IP 协议族，以太帧有 14 字节的帧头和 4 字节的帧尾，IP 层有 20 字节的 IP 头，TCP 层有 20 字节，UDP 层有 8 个字节的附加信息。

1.3 TCP/IP 各层的功能

IP 层提供一个非连接的、不可靠的数据报传送系统。每个 IP 数据报相互独立，均含有源地址和目的地址，独立进行发送和路由选择。IP 也负责分段，将一个大的数据报划分为几个能在下一个网络中传输的小的分段。IP 层还提供了基本形式的流控制。

TCP 层提供虚电路服务和面向连接的传输服务，对用户数据进行有效、可靠的传送。TCP 在一对传输端点之间提供数据连接。连接管理可分为三个阶段：建立连接、维护连接和终止连接。为实现可靠性，TCP 采用确认与超时重传机制；为实现顺序的报文流，TCP 采用滑动窗口协议，为保证数据的正确性，TCP 采用若干差错检验、报告和纠正措施。

UDP 的服务是不可靠的。它不确认报文是否到达，不对报文排序，也不进行流控。它的优点是传输延迟小，吞吐量大。

第二章流机制介绍

SNX 提供了一个实时的，与 UNIX SVR 3/4 标准兼容的嵌入式的流式机制，及一个完整的基于流的 TCP/IP 协议栈。流模块包括了一整套标准的模块间的队列服务，以提供快速的和灵活的协议模块间的流控和消息传递。基于"零拷贝"的基础，数据从流的顶端通过协议模块传向一个设备驱动程序。因为这个有效的体系结构，流提供很高的吞吐量，模块性和开放性。

另外，流可以在不影响其他模块的情况下压入和弹出协议模块。将一个模块压栈是由应用任务完成的简单操作。流同时也标准化了应用接口，允许以统一的形式来访问 OSI 模型的不同层协议。

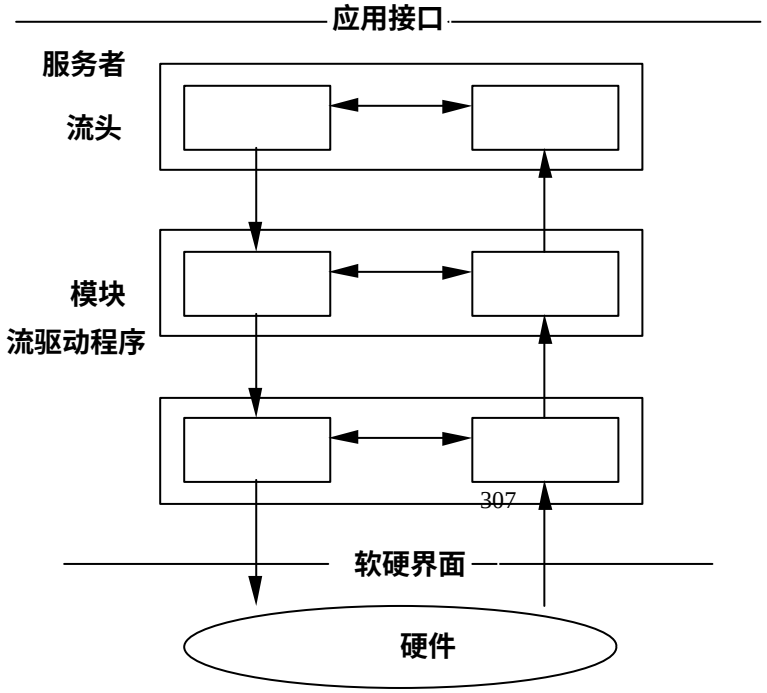
2.1 关于流机制的一些概念

流是一种为网络协议及驱动程序而设计的 I/O 子系统。它是提供友好界面，模块化和内建缓冲机制的队列系统。流机制独立于通信协议，因为流机制本身并不是协议。流机制仅仅是管理这些协议模块的操作。只有基于流的协议模块和驱动程序才能被流机制管理。

在流机制下，各驱动程序和协议模块使用流报文来交换信息，流报文由两部分组成： 一部分是流报文本身的控制信息，另一部分为该报文的数据区。

流至少包括两个组成部分：流头和通信驱动程序。流头提供并实现了流的系统调用的接口。能实现协议层的模块由用户自由安置在流中的流头和驱动程序之间。流机制下的各驱动程序及协议模块利用流机制提供的工具来相互传递报文。一个是“上行流”工具，它起于通信驱动程序，止于应用程序；另一个是“下行流”，它始于应用程序与 SNX 的接口，止于通信驱动程序。用另一种方式理解，“上行流”报文可以看作为读信息，而“下行流”则为写信息。

下图解释流的结构：



在流机制中，用流队列这种数据结构来联系各个协议模块和驱动程序。流队列总是成队分配，其中一个为读队列，一个为写队列。驱动程序可以有多对“流队列”，而协议模块一般只有单对“流队列”。这是“driver”与“module”重要区别。

在读信息过程中，驱动程序将收到的帧拆包，组成流的消息，向上传递，每个模块处理一个消息，并将其传送至上面的队列；应用进程在调用 read/getmsg 时，如果流头的读队列未收到一个消息，则阻塞该应用进程，受到消息后，流头将数据报中的数据部分复制到用户缓冲区，并唤醒阻塞的应用进程。

在写信息过程中，流头解释应用程序中的系统调用，将用户缓冲区复制到相应的流缓冲区，并根据系统调用的参数将要发送的数据打包成流报文，向下传递。类似，每个模块队列处理一个消息，并将之传至下一个队列。

2.2 SNX 流的模块和驱动程序

在流机制中，有两种流驱动程序。一种是设备驱动程序，也叫硬件驱动程序。它主要
是作为控制硬件设备的接口。设备驱动程序可以有多个上行流。另一种流驱动程序是协议驱动程序，也叫软件驱动程序。这种驱动程序虽然不控制硬件设备，但是却提纲了通信进程所需的各层协议。协议驱动程序可以有多个上行流或（和）下行流。

模块与驱动程序不同，它不提供协议处理，而仅提供一些函数支持。例如，BSD 的 SOCKETS API 的实现就是在用户库与内核流机制的 sockmod 模块之间的接口。

SNX 提供了完备的网络模块及驱动程序，满足了工程技术人员开发各种不同目的、不同功能的网络应用软件。这体现了 SNX 的强大的技术支持。

其中包括：（部分）

- TCP 驱动程序(/dev/tcp)
- UDP 驱动程序(/dev/udp)
- IP 驱动程序(/dev/ip)
- ICMP 驱动程序（/dev/icmp）
- ARP 驱动程序（/dev/arp）和模块（/dev/arpproc）
- Loop-back 驱动程序（/dev/lo）
- Tirdwr 模块（/dev/tirdwr）
- logio 以太网驱动程序（/dev/lg）
- 管道驱动程序（/dev/pipe）

具体的模块介绍和配置在下一章详细说明。

第三章 VRTX 的基于流的网络组件 SNX

本章列举了如何够造一个包含 snx 的内核，然后把该内核下载到目标机上，然后测试 snx 是否工作正常的步骤。同时，也为用户能顺利使用 snx 而介绍了 snx 的重要选项及其特点。

3.1 SNX 的硬件要求和软件配置

SNX，即 Microtec 基于流的核心机制及 TCP/IP 栈的网络开发组件。

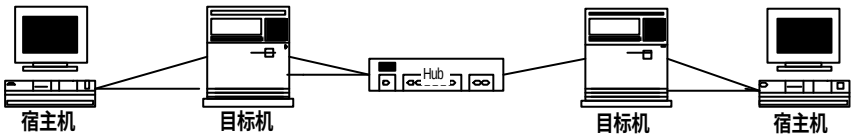
3.1.1 Hardware Configuration 硬件设置

因为 SNX 主要是为嵌入式开发而设计的，所以其网络的开发环境也与一般的网络开发环境不同。我们主要针对的是嵌入式 PC 和嵌入式 360,860。

用户的开发环境可能包括如下内容：

- 开发的主机
- 目标机
- 串行连接
- 网上的其它主机

下图为典型开发环境示意图：



通常，目标机的 PROM 中有 Microtec 提供的或是定做的 BSP。

3.1.2 SNX 的软件组成

1. SNX 提供了丰富的编程接口:
   UNIX SYSTEM V TLI 库
   Berkeley 4.3 BSD 套接字接口库
   流及 I/O 系统调用

地址解析库（DNS 客户）
RPC 库（客户和服务者）

2．地址解析库

域名服务地址解析函数库 libresolv.a 中提供了一组例程。名字和地址查询的过程如下所述：首先，在当地主机名表中查找，如未找到，产生一个查询，并将之发送给系统启动时指定的域名服务器。与 UNIX 不同的是，SNX 每个库函数调用返回的缓冲区必须由 h_free 释放。提供了以下的库函数：

```
struct hostent *gethostbyname(char *name);
struct hostent *gethostbyaddr(char *addr,int len,int type);
void h_free(struct hostent *hp);
```

这个函数释放有 gethostbyname 或 gethostbyaddr 返回的 hostent 结构。

3．应用程序

SNX 支持一些网络命令,这些命令可在命令行下运行 。

如：ping 、route 、netstat、ttcp 等

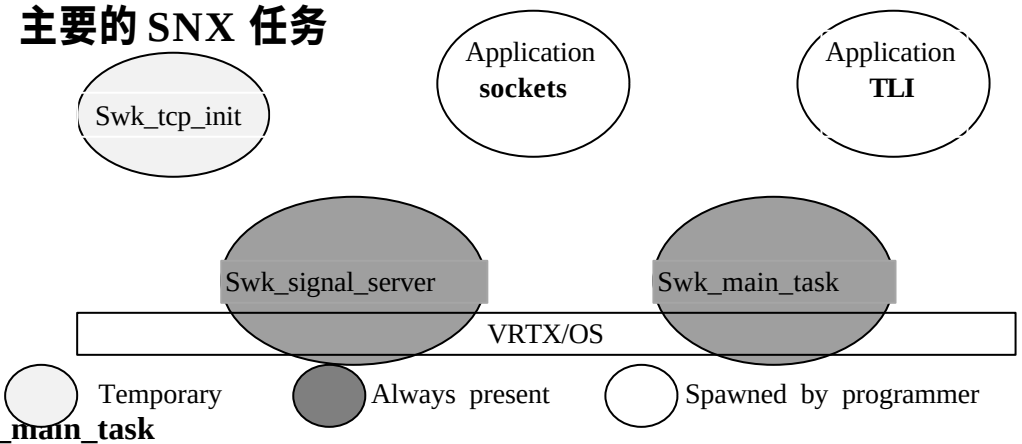
ttcp 是用来测试传送层、套接字接口、TCP，UDP 驱动程序等的性能的网络命令，如可以统计发送或接受的总字节数，及花费的时间，计算出网络吞吐量。

4. RPC/XDR 库

RPC 和 XDR 提供了系统之间过程调用。XDR 提供了一组网络中标准的数据和用户数据表示之间转换的例程，确保了应用程序的可移植性。RPC 则提供了远程调用的一组例程。

3. 2 SNX 体系结构

3.2.1 主要的 SNX 任务



1. **swk_main_task**

提供流框架和 TCP/IP 的 SNX 主任务。为客户应用程序提供服务器任务。在 SNX 初始化时，被 snxcnfg. c 中函数 sys_initialize_snx 调用激活。

2. **swk_signal_server**

当收到信号就调用信号处理器的信号服务器任务。当 snx 初始化时,被 snxcnfg.c 中的函数 sys_initialize_snx 调用激活。

2. **swk_tcp_init**

TCP/IP 初始化例程。 当 VRTX 初始化时，被 snxcnfg.c 中的函数 sys_initialize_snx 调用。

3.2.2 SNX 的应用任务优先级

网络应用程序作为 VRTX/OS 的任务在运行。应用程序任务的优先级是可以由用户定义的。其中，swk_main_task 的缺省优先级在系统中是最低的。但是，也可以由用户确定 swk_main_task 的优先级。

应用程序任务可以使用 SOCKETS 或 TLI 编程接口与 TCP/IP 协议栈通讯。这些编程接口是作为 SNX 的产品的一部分，并且与用户的应用程序任务联在一起。相反，这些编程接口并没有与 swk_main_task 直接相连，而是被转换相应的系统调用如 Open,Close,putmsg,getmsg 等等。在 UNIX SVR4 系统中，系统调用将中断应用程序的处理，同时将系统切换到内核态。而在 VRTX/OS SNX 的实现中，系统调用被转换为请求报文发给 swk_main_task，然后发送该请求报文的应用任务就挂起，直到 swk_main_task 接收到请求报文并处理完该请求并发回应答后。

即使发送请求报文的应用程序任务优先级比 swk_main_task 高，在它发出请求报文之后，就被阻塞。其他任务继续运行，仍然可以发送各自的请求报文给 swk_main_task。swk_main_task 在 它成为就绪任务中优先级最高的任务后才会运行，处理请求。每一个应用程序任务在它再次运行前，只能向 swk_main_task 发送一个请求。但是必须注意的是：在 swk_main_task 有机会处理任何请求以前，可能有多个任务向 swk_main_task 发送请求。

在 swk_main_task 响应以前阻塞任务看似不利，但是这类似其他操作系统处理系统调用的方式。如在 UNIX 系统 V 中，一个进程进行一个系统调用，在内核的 TCP/IP 协议栈处理请求并返回到用户态前，这个进程被剥夺运行权。

1. swk_main_task 设计

swk_main_task 负责处理多个应用程序以及驱动程序产生的各种事件。应用程序产生的事件通常是流的系统调用请求。而硬件设备驱动程序产生的事件通常是告诉 swk_main_task 已收到从网上来的数据。

swk_main_task 所做的事分为两大类：内部事情、外部事情。通常， 内部事情是指各种模块例程的周期调度。以及各种定时器的处理等事情。外部事情是处理由应用程序任务发出的系统调用和设备驱动程序发出数据到达指示等内部事件。如果没有外部事情产生，swk_main_task 会周期被 VRTX/OS 唤醒以处理各

种内部事情。一旦所有的内部事情做完后，又没有外部事情产生，则重新阻塞自己直到 VRTX 唤醒自己。

2. swk_signal_server 信号服务器

信号是处理异步事件典型的方法。什么时候信号会产生是无法预测的。SNX 为了处理这种异步事件引入了 swk_signal_server 信号服务器概念。

swk_signal_server 被用来当产生信号时，唤醒信号处理器。其中，信号处理器通过 signal 系统调用被安装。

以下是可处理的 4 种信号，如果相应的信号处理器没有被安装，则执行缺省处理程序。

SIGPOLL，SIGIO，SIGURG，SIGPIPE。

另外，还可以通过使用 poll 函数或 SOCKETS 的 select 系统调用来使用信号。swk_signal_server 是在 sys_initialize_snx 函数的最后生成的。swk_signal_server 负责处理由流头产生的信号并为用户提供安装信号处理程序的机制。

swk_signal_server 可以接收两种输入：一是用户应用程序执行 signal 系统调用以安装信号处理程序；二是由 swk_main_task 的流头产生的四种信号之一。每一种输入都有单独的输入队列。swk_signal_server 在队列处等待输入产生，没有超时限制。在没有输入产生前，swk_signal_server 一直出于休眠状态。

用于接收用户应用程序的 signal 系统调用请求报文的输入队列大小，缺省值为 3，这就意味着在 swk_signal_server 开始处理前，最多允许三个应用程序调用 signal；用于接收流头产生的信号的输入队列大小，缺省值为 30。swk_signal_server 的缺省优先级为 1，比 swk_main_task 高。也即是说，一旦 swk_signal_server 准备执行，将抢占 swk_main_task。

3. 初始化工作

共有四种 SNX 初始化工作：sys_initialize_snx 初始化，TCP/IP 协议栈初始化，用户的“hook”初始化，私有协议栈的初始化。

sys_initialize_snx 初始化流以及任务间通信的资源，然后生成 swk_main_task 和 swk_signal_server。然后，生成 swk_tcp_init 来初始化 TCP/IP 协议栈以及 SOCKETS 或是 TLI 库（如果需要的话）。在 swk_tcp_init 的最后，调用 SNX 的初始化钩子例程（由 Xconfig 变量 snx,hook_init 指定）。snx.hook_init 被用作在 TCP/IP 栈初始化后产生 TCP/IP 应用程序。如果要加入自己的流代码，就不使用 snx.hook_init，把代码加到 sys_initialize_snx 后。如果 IP 地址必须但未配置，RARP 就在网络接口初始化时被调用。

4. 打开和关闭请求（Open and Close Requests）

在打开以及关闭操作期间，各流模块或流驱动程序均可以睡眠，这使得执行打开和关闭操作的任务睡眠。执行打开和关闭操作的 OC_thread 临时任务不被分配任务号。它们只存在很短的一段时间。OC_thread 任务在有打开和关闭请求

时产生，并被赋予比 swk_main_task 高一级的优先级。

5. 文件描述符（File Descriptors）

在执行 SNX 的 open 系统调用后，将返回文件描述符。文件描述符的最大个数由 Xconfig 变量 snx.fdtab_size 决定。文件描述符的缺省个数为 256 个。

特别应当注意的是：SNX 的文件描述符是全局的，而不象 UNIX 中，文件描述符是由进程指定的。这也是以后设计并发服务器的一个基础。

以下是有关 SNX 文件描述符的一些特点：

- 文件描述符由 0 到 255 标识。
- 文件描述符号 0，1，2 并没有特别意思，而不象 UNIX 中，分别与 stdin，stdout，stderr 对应。
- SNX 的文件描述符与 stdin/stdout/stderr 描述符和 IFX 文件描述符完全分开的。
- 文件描述符由低号向高号依次分配。如果有文件描述符释放，则下一次分配将是释放的文件描述符中号数最低的。
- 当任务终止后，该任务打开的文件描述符不会自动关闭。

6. Sockets

在执行 SNX 的 Socket 函数后，将返回 socket 号。

以下是 SNX 的 Sockets 的特点：

- 最大 Socket 数号只能为 256，不能被修改。
- Socket 的描述符被限制只能使用文件描述符的前 256 个。
- 其余有关文件描述符的一些关闭。共享特点，Socket 都具备。

7. 消息传递（Message Passing）

消息传递接口使用 VRTX 的机制把系统调用以及中断请求传递给 swk_main_task。这一切对用户是透明的。

8. SNX 网络接口（SNX Network Interfaces）

SNX 使用提供的模块和驱动程序来支持以太网及串行接口和 VME 通信通道三种网络接口。同时，SNX 既可以通过 logio 来使用网络接口，也可以直接控制网络接口的驱动程序。另外，用户还可以开发自己的网络接口驱动程序。

3. 3 SNX 的配置

3.3.1 Basic Setup 基本设置

如果要把 SNX 设置加入 VRTX 的映象中，就要在用户自己的配置文件(假设为 user.def) 进行配置.

例如，使用 ACE360/QM 目标板的 VRTXsa 映象，并加入 SNX，则 user.def

文件将包含以下内容：

@ include (system.def)
@ include (cp.def)
@ include (microtec.def)
snx.enabled: ：yes

设置变量 snx.enabled 为 yes, 则使得将 SNX 包含入 VRTXsa 的映象。然后当执行完命令 Xconfig user.def 后，将在用户的工作目录中生成 snxcnfg.c 文件及其它文件。

1. snxcnfg.c

SNX 将检查设置文件 snxcnfg.c 中包含的自己要用的关键变量及数据结构的名字及内容。 但是，snxcnfg.c 文件不能被直接修改，其内容由文件 snx.def、snx.tpl 和 user.def 控制。这些在<<Microtec Configuration Tool (Xconfig) User's Guide and Reference>> 中有详细描述。这些文件的数据结构不能被用户代码引用。它们被看成 SNX 的内部数据结构。

snxcnfg.c 文件使用 SNX 的 Xconfig 变量的值设置各种 SNX 的配置参数，从而配置 SNX ,这些变量例如：

- 报文传递库参数
- 流机制参数：定时器， 主任务， 内存
- 接口库
- 驱动程序/模块的声明和配置
- 网络接口表
- SNX 设备
- 主机表
- 本地主机信息
- 名字服务器

. 网关

. RARP 参数

. RPC 参数

2. Local Host Name and Host Table

在主机表中， 通常在 user.def 文件中定义本地主机名十分有用， 但不是必需的。本地主机名是分配给一个特定 Internet 主机（例如运行 SNX 的目标板）的符号名。

SNX 的主机表与 UNIX 中的/etc/hosts 文件很相似. 主机表提供了主机符号名与其 IP 地址之间的映射关系。

多个符号名（又叫别名）可以对应同一个 IP 地址。

Xconfig 变量 snx.local_host，snx.hosts.name 和 snx.hosts. list 被用来定义本地主机名和主机表（可参见 snxcnfg.c 中的数据结构 snx_host_table）。

例；如果要生成一个有 SNX，并且有本地主机名和主机表的 VRTXsa 映

象，文件 user.def 则应如下所示：

```
@ include (system.def)
@ include (cp.def)
@ include (microtec.def)
 snx.enabled:  yes
 snx.local_host:  local_147
snx. hosts. Local_147:  149.147.5.119
snx. hosts. Other_host_2:  149.147.5.324
snx. hosts. Dup_host_2:  149.147.5.325
snx. hosts. list:  local_147, other_host_2, dup_host_2
```

3.3.2 Configuration Options 配置选项

可以通过配置 SNX 的一些特性来使得 SNX 适合于特定的环境。 这些特性的定义及改变可以通过向文件 user.def 中加入 Xconfig 变量来实现。

内存分配变量

1. snx.streams.heap_size

定义由使用的内存堆大小，所有流报文、定时器、流模块和驱动程序使用的各种私有资源以及流框架都从所定义的堆中分配。

缺省值为 512K（524,288 bytes）。

2. 用于数据缓冲区预分配的变量

snx. streams. str16bufs	预分配 16 字节数据缓冲区的数目及相应的数据结构
snx. streams. str32bufs	预分配 32 字节数据缓冲区的数目及相应的数据结构
snx. streams. str64bufs	预分配 64 字节数据缓冲区的数目及相应的数据结构
snx. streams. str128bufs	预分配 64 字节数据缓冲区的数目及相应的数据结构
snx. streams. str256bufs	预分配 64 字节数据缓冲区的数目及相应的数据结构
snx. streams. str512bufs	预分配 64 字节数据缓冲区的数目及相应的数据结构
snx. streams. str1024bufs	预分配 1024 字节数据缓冲区的数目及相应的数据结构
snx. streams. str2048bufs	预分配 1024 字节数据缓冲区的数目及相应的数据结构
snx. streams. str4560bufs	预分配 1024 字节数据缓冲区的数目及相应的数据结构

3. 用于别的流资源的预分配变量

snx. streams. max_mblks 可以预分配的流报文块的总数。

snx. streams. max_times 预分配给使用的定时器数据结构的总数。定时器被驱动程序和模块用来确定是否超时。

snx. streams. max_msgs 内部调用请求数据结构的个数。

snx. streams. max_stacks swk_main_task 内部报文堆栈数据结构的个数。

3.3.3 驱动程序和模块变量

1. SNX 的驱动程序和模块配置变量

. snx. drvmod. component. enable　　缺省值为 yes。

模块使能。设置为“yes**”时，将模块/驱动程序加入到模块/驱动程序表中。**

snx. drvmod. component. streamtab　　缺省值没有。

定义与 component 相联系的 streamtab 数据结构。如果 component 已被配置入模块/驱动程序表，则该变量必须定义。如果所给的 streamtab 数据结构不存在，SNX **的初始化将失败。**

. snx. drvmod. component. initf　　缺省值没有。

为 component 定义由 sys_init_snx 任务所调用的初始化函数，如果没初始化函数，则该变量为 0。

. snx. drvmod. list　缺省值为所有的模块或驱动程序。

用一组由逗号分离的名字来定义所有配置入 SNX **的流模块或驱动程序。各名字的顺序无所谓先后。**

. snx. drvmod. component. flags　　缺省值没有。

定义 component 是驱动程序还是模块，是 SVR3 版本还是 SVR4 版本。其中 M **表示模块，**D **表示驱动程序，**3 **表示 SVR3 版本，**4 **表示 SVR4 版本。 因此，合法的值只有** 4 **个：**M3，M4，D3，D4。

swk_main_task 被设置为可以使用不同模块及驱动程序的组合。各个模块及驱动程序通过 Xconfig 变量被配置入 swk_main_task。例如 当使用标准以太网 logio **驱动程序构造 TCP/IP 协议栈，应用程序接口为** sockets，snx.drvmod.list **配置如下：**

. snx.drvmod.list:　lo, ip,tcp,udp,icmp,arpproc,arp,lg,sockmod

2. SNX 驱动程序的配置变量

SNX 支持的的网络接口有 le、lg。

le：　**是** MVME147 LANCE **芯片流驱动程序。**

lg：　**在** logio **驱动程序上的基于流的驱动程序。**

可通过以下变量设置接口变量：

. snx.interface.driver.devname

定义驱动程序名，缺省值没有。

. snx.interface.driver.othermodule

定义当生成流时，需要放在 driver 之上的模块。缺省值没有。

. snx.interface.driver.broadcast

确定是否支持广播，缺省值没有。

. snx.interface.driver.netmask

定义用于子网寻径的网络掩码，缺省值没有。

. snx.interface.driver.arpsupport

确定是否支持 ARP 协议，缺省值没有。snx.interface.driver.addr 定义接口的 IP 地址，并以小数点相隔。缺省值没有。

. snx.interface.list

定义将被配置入数据结构 snx_interface_table 中的接口链表。其中 snx_interface_table 在文件 snxcnfg.c 中。缺省值没有。

. snx.logic.devname.driver

把 driver 与特定的要用的逻辑 I/O 驱动程序相联系。 其中 driver 为驱动程序名加上一个实例，并从 0 开始，即 lg0。缺省值没有。

3.3.4 TCP/IP 配置变量

1. 基本 TCP/IP 配置

snx.local.host 定义本地主机的名字。

snx.hosts.name 定义 name 确定的目标板的 IP 地址。

snx.hosts.list 定义所有出现在主机表中的主机名，以逗号相隔开。

2. 一般 TCP/IP 配置

snx.hook_init 使能 SNX 的 hook 例程，在 SNX 和 TCP/IP 初始化后被调用。用以产生网络应用或命令（如 routed）缺省值为 0，即不使能。

snx.tcp.def_rcvbuf TCP 的窗口大小，最大为 32K bytes, 缺省值为 32K。

3. RARP 配置

snx.rarp.enable 使能 RARP 协议模块。即在系统初始化时获得正确的 IP 地址。缺省值为 yes。

snx.rarp.retries 定义 RARP 模块重发 RARP 请求的次数，最小值为-1，最大值为 255，缺省值为-1。

snx.rarp.timeout 定义 RARP 模块在发出 RARP 请求后，等待 RARP 回应的

时间长度，以毫秒为单位。最小值为 1，最大值 0xFFFFFFFF，缺省值为 5000。

4. DNS 配置

snx.domain.name 定义本地主机所属的域名。缺省值为 NULL。

snx.nameserver.list 定义本地主机所知道的全部域名服务器名，以逗号相隔开。当 DNS 查询产生后，按本变量所定义的顺序，依次与其联系，直到收到回答。

5. 应用程序接口配置

snx.interface.socket 使能 Socket 接口，为 1 则使能 Socket 接口，为 0 则不使能 Socket 接口，缺省值为 1。

snx.interface.tli 使能 TLI 接口。为 1 则使能 TLI 接口，为 0 则不使能 TLI 接口，缺省值为 1。

3.3.5 各种其它变量

1. 各种其它 SNX 变量

snx.swk.priority 定义 swk_main_task 的优先级。最小值为 0，最大值为 255，缺省值为 255。推荐为比任何进行系统调用的任务的优先级都低。

snx.swk.tid 定义 swk_main_task 的任务号。缺省值 1，取值范围 0~255。

snx.fdtab_size 定义 SNX 的全局文件描述符表的大小。缺省值为 256。

snx.verbosity 定义流报告错误的级别大小

0：可以报告所有错误

1：不能报告 CE-CONT 错误

2：不能报告 CE-NOTE 错误

3：不能报告 CE-WARN 错误

缺省值为 0。.

snx.sig.tid 定义分配给 SNX 服务器任务 swk_signal_server 的任务号。缺省值为 0。范围值为 0~255。

snx.sig.priority 定义 signal handlers 执行的优先级。缺省值为 1。

rpc.enabled 确定是否配置 RPC，缺省值为 no。

3. 4 使用 SNX 的流机制

3.4.1 配置自己的协议栈 （Configuring Your Own Stack）

要构造自己的私有协议栈，需按下列步骤进行：

1. 去掉缺省的 TCP/IP 驱动程序和模块。
2. 声明新的驱动程序和模块。
3. 自己写在 swk_tcp_init 任务中要调用的初始化函数。也可以在 sys_initialize_snx 函数之后调用。

3.4.2 把模块和驱动程序配置入 SNX （Configuring Modules and Drivers Into SNX）

配置程序 user.def **详见** STREAMS and TCP/IP Networking Executive and SNMP Programmer's Guide and Reference. P5-4

1. 初始化过程 Initialization Process

初始化函数 Swk_tcp_init **详见** P5-4 ~ P5-9。

3.5 使用 SNX 提供的模块和驱动程序

1. 构造一个协议栈 （Building a Stack）

可以使用变量 snx.drvmod.list 来定义协议栈的内容（顺序没有关系）。例如，要人工定义一个最小的 TCP/IP 协议栈，则有：

snx.drvmod.list：lo，ip，tcp，udp，icmp，arpproc，arp，lg，sockmod

对于列表中出现的每一个模块和驱动程序，还需定义以下相应的 Xconfig 变量：

snx.drvmod.module.enable
snx.drvmod.module.streamtab
snx.drvmod.module.flags
snx.drvmod.module.initf

2. 使用 ARPPROC 和 ARP（Using ARPPROC and ARP)

ARP 包含两个方面：地址解析（arp）和逆向地址解析（Rarp）。

arpproc 模块和 arp 驱动程序必须被配置为支持 Rarp。在缺省情况下支持 Rarp。因此，snx.rarp.enable,snx.rarp.timeout 和 snx.rarp.retries 变量也应相应设置。

3. 使用 Sockmod （Using Sockmod）

当 socket 被打开时，sockmod 模块被加入流框架中。如果要使用 SOCKETS 接口，则把 sockmod 模块配置入流的子系统，并把应用程序任务与库 libsockets.a 和库 libswksyscall.a 相链接。

3.6 使用 SNX 的 TCP/IP

3.6.1 TCP/IP API

snx 支持几种网络 API。要在自己应用程序中使用一个 API，就必须定义头文件和指定库。

1. TLI （Transport Layer Interface）

传输层接口是一个允许应用程序使用传输层协议服务的 API。TLI 用于提供 TCP/IP 应用程序接口。在 SNX 中使用 TLI 的优点是应用程序的可移植性。

TLI 的库 tli.lib 使用 TPI(Transport Provider Interface)与 TCP 或 UDP 通信。TPI 由 SNX 模块 tcp 和 udp 提供 TLI 的头文件为 tiuser.h。

2. SOCKETS

SOCKETS 接口由一组库函数和一个流模块（sockmod）实现。当 socket 被打开后，sockmod 被加在协议栈的上层。要使用 SOCKETS 接口，需要把模块

sockmod 配置入流框架并把应用程序任务与库 sockets 和 swksyscall.lib 相链接。SNX SOCKETS API 支持除 readv 和 writev 之外的全部通用 SOCKETS APIs。

SNX SOCKETS 库 sockets.lib 使用自己的内部原语与 SNX 流模块 sockmod 通信。sockmod 使用 TPI 与 tcp 和 udp 模块通信。

3.6.2 网络工具 (Networking Utilities)

网络工具由库 snx.lib 中的一组函数提供。其中，有三个函数是独有的，它们是:recvfrom_tli,setsockopt_tli,sendto_tli。其余的函数中，除 getopt_r 和 inet_ntoa_r 是由 UNIX 网络工具 getopt 和 inet_ntoa 引出之外，均与标准 UNIX 相同函数名。

详见 SNX 程序员参考手册

3.6.3 域名系统地址解析器（Domain Name System Address Resolver）

TCP/IP 应用程序使用 DNS 来知道主机姓名所对应的 IP 地址。在 SNX 中，DNS 地址解析器由库 resolve.lib 中的一组库函数提供。

在地址解析时，首先搜寻本地主机姓名表，如果没有搜寻到，则生成一个查询报文发向 DNS 服务器。DNS 服务器是在系统启动时，由 Xconfig 变量 snx.nameserver.list 所定义的。

DNS 通过使用 Xconfig 变量 snx.domain.name 和 snx.nameserver.list 配置，如下所示:

snx.domain.name ： mri.com

snx.nameserver.list： doors，mri_gw

第四章 SNX 的网络应用开发

在这一章，详细介绍了 SNX 的编程接口，着重介绍了 SOCKET 编程接口。并利用该接口如何开发应用程序。

4.1 SNX 的应用程序接口

SNX 提供给用户两个主要的应用程序接口。一个为与 BSD UNIX 兼容的管套（socket）系统调用，另一个为与 UNIX system V 兼容的传送层接口 TLI 函数库。管套比较流行，在这我主要介绍 SOCKET 的使用方法。

4.1.1 SOCKET 编程简介

目前，随着网络技术的飞速发展，网络对通信效率的要求愈来愈高。特别是客户/服务器应用系统的出现，要求以高效率的网络通信机制为其提供通信技术支持。而 socket 套接字机制正满足了这种要求。socket 套接字由 BSD UNIX 首先提出，目的是解决网络间的进程通信问题。4BSD UNIX 对上述进程通信的全部解答构成了 socket 机制的全部内容。

使用 socket 套接字开发接口实现通信程序，可以采用面向连接的客户/服务器模型和无连接的客户/服务器模型。两者典型的时序图见下页图 4-1 和图 4-2。面向连接的客户/服务器模型采用是 TCP/IP 协议族提供的高可靠性的虚电路服务（sock_STREAM）。而无连接的客户/服务器模型采用的 tcp/ip 提供的高效率的数据报服务（sock_DGRAM）。

1. 端口的概念

传输层与网络层在功能上的最大区别是前者提供进程通信能力，后者不提供进程通信能力。在进程通信的意义上，网络通信的最终地址就不仅仅是主机的地址，还包括可以描述进程的某种标识符。

为此，TCP/UDP 提出协议端口（protocol port）的概念，用于标识通信的进程。端口相当于 OSI 的传输层服务访问点，是一种抽象的软件结构（包括一些数

据结构和 I/O 缓冲区）。应用程序（及进程）通过系统调用与某些端口建立联编（binding）后，传输层传给该端口的数据都被相应进程所接收。

类似于文件描述符，每个端口都拥有一个叫端口号（port number）的整数标识符，用于区分不同端口。由于 TCP 和 UDP 是完全独立的两个软件模块，因此各自的端口号也相互独立。TCP/UDP 允许长达 16 比特的端口值。

两个不同机器上的进程相互通信，如何得到对方的端口号呢？TCP/IP 提出一种端口分配方法。将端口分为两部分：一部分是保留端口，一部分是自由端口。其中保留端口只占很小的数目，将一些常用的应用程序使用的端口确定下来，如端口 20，21 作为 ftp 的端口，
23 为 TELNET 的端口等。每个标准的服务器都拥有一个全局公认的端口。自由端口占全部端口的绝大部分，当进程需要访问传输服务时，向本地操作系统提出动态申请，操作系统返回一个本地唯一的端口号，进程再通过合适的系统调用将自己和相应端口号联系起来。然后根据全局分配的公认端口号与远地服务器建立连接，才能进行数据传输。

这里所提到的客户/服务器模型是指将两种不同的程序分开来进行处理。这两种程序分别是：对用户各自的环境进行管理的客户程序（被称为客户方），以及维持系统整体环境的服务程序。运行时由客户方向服务器方提出服务请求，再由服务器方对请求进行处理后向客户方作出服务应答。从而完成整个客户的任务请求。

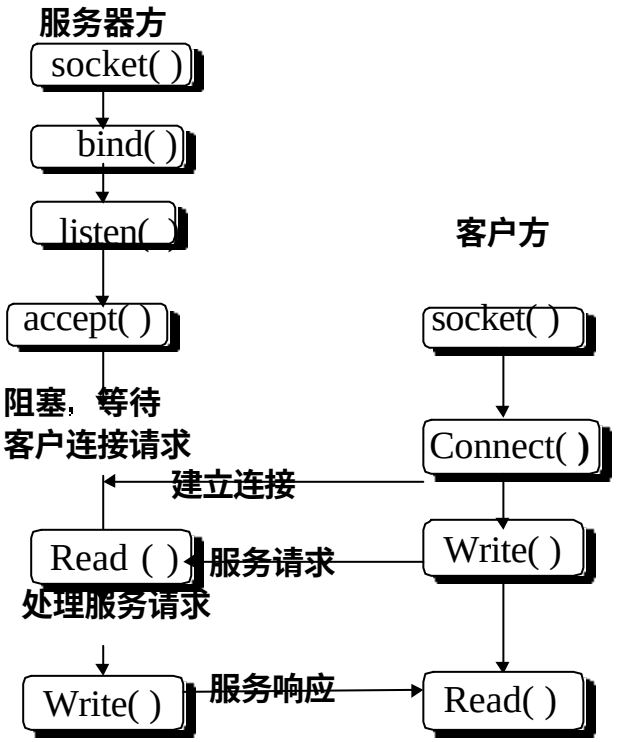


图 4-1 面向连接的客户/服务器模型时序图

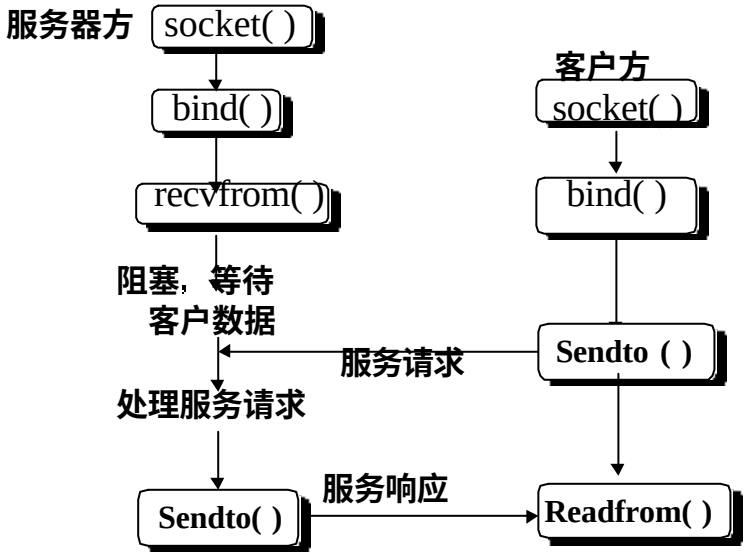


图 4-2 无连接的客户/服务器模型时序图

4.1.2 socket 提供的主要函数

- **创建** socket —— socket (af, type, protocol)

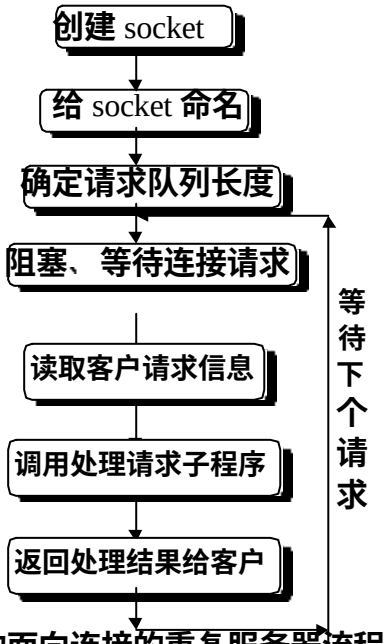
- socket 命名 —— bind (sockid, localaddr, addrlen)
- 建立 socket 连接 ——
  connect (sockid, destaddr, addrlen)
  listen (scokid, qulen)
- 发送数据 ——
  面向连接服务的三个函数
  write (sockid, buff, bufflen)
  writev (sockid, iovector,vectorlen)
  send (sockid,buff,bufflen,flags)
  无连接服务的两个函数
  sendto (sockid,buff,bufflen,flages,destaddr,addrlen)
  sendmsg (sockid, message, flage)
- 接收数据 ——
  read( )、readv( )、rec( )、
  recvfrom( )、recvmsg( )

详细的说明参看附录。

4.2 面向连接的客户/服务器典型模型服务器方的设计

服务器可设计成重复服务器或并发服务器

4.2.1 重复服务器



典型的面向连接的重复服务器流程图

重复服务器是指对于客户方提出的每个服务请求都由服务器亲自去处理，这种处理方式适合于处理在短时间便可处理完的请求。它对客户方提出的服务请求是按顺序进行处理的。其流程图见下：

4. 2. 2 并发服务器

并发服务器是一个守护进程（daemon），随系统启动而启动。无请求到达时，并发服务器处于等待状态。一旦客户请求到达，服务器立即为之产生一个子进程，然后回到等待状态，由子进程响应请求。当下一请求到达，服务器再为之产生一个新的子进程。其中，并发服务器叫主服务器（master），子进程叫从服务器（slave）。这种主从服务器的方式巧妙地解决了并发请求问题。显然并发服务器的性能比重复服务器优越。

4. 3 SNX 上socket 的并发服务器的实现

4 .3.1 实现的设计思想

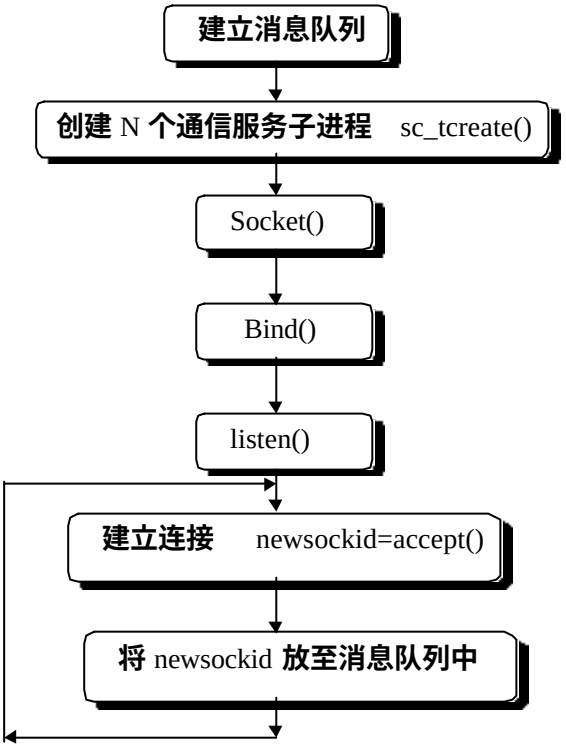
上节所述的并发服务器的实现是建立在一般的目标系统（如 UNIX）基础上的。而 SNX 的内核与其他模块分离运行机制。使得它用 SCOKET 机制不能实现典型的并发服务器。为了在 SNX 上实现 socket 并发服务器，必须对并发服务器模型进行重新设计。

用 VRTX 操作系统提供的系统调用来建立并发服务器模型。

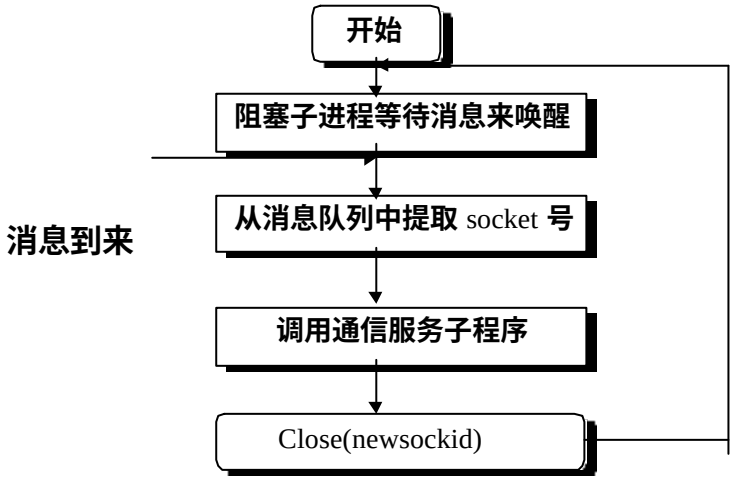
其实现思想为：

先创建一个存放消息的空消息队列,再创建几个优先级相同的提供通信服务的任务（相当于 UNIX 中的子进程，下本文就称为子进程）。创建完子进程后,各子进程就调用实时操作系统 VRTXsa 的系统调用 sc_qpend，将自身悬挂起来等待接收消息队列中的消息来唤醒自己，父进程进行 socket 的初始化，并等待接收连接。当客户方的连接请求到来时就建立连接。并将 socket 号作为消息发送到消息队列中，这时消息将唤醒事先悬挂的通信服务子进程来提取 socket 号进行通信服务。当子进程的通信服务完成后，通信服务子进程将撤消连接，并自行悬挂起，等待接收下一条消息的到来，准备为下一个连接提供通信服务。父进程可不断地建立连接，将连接的 socket 号存入消息队列，供空闲的通信服务子进程提取。这样，多个通信服务子进程同时为多个连接提供通信服务，从而完成了并发服务器的基本功能。

4. 3. 2　snx 上 socket 并发服务器实现的程序流程图



服务器方程序流程图



通信服务子进程流程图

4.3.3 操作系统的系统调用

SNX 上 SOCKET 并发服务器的实现中所调用的操作系统系统调用如下：

- 创建任务

  int sc_tcreate(viod(*task) (void*), int tid, int pri, int *errp)

- 发送消息到指定队列

  int sc_gpost(int qid, char *msg, int *errp)

- 接收一指定队列来的消息，可能等待

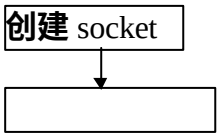
  char *sc_qaccept (int qid, int *errp)

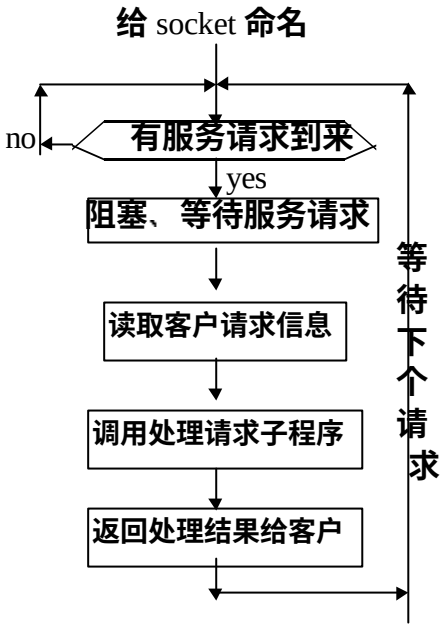
4.4 面向连接的客户方程序设计框架

```
int initsockid;
if ((initsockid=socket(AF_INET,SOCK_STREAM,0))<0)
{ perror("socket open failed");
  exit(1); }
if (connect(initsockid,···)<0)
{    perror("socket connect failed ");
  exit(1); }
if (write (initsockid,···)<0)
{    perror(" writeing on stream socket  failed ");
  exit(1); }
if (read(initsockid,···)<0)
{    perror(" reading on stream socket  failed ")
     exit(1); }
```

4.5 无连接的客户/服务器的程序流程图

服务器方：　　客户方：（略）





4.6 TLI 网络的开发简介

传送层接口 TLI 是一函数集合，它为用户提供传送层接口，建立通信、管理连接以及传送数据。应用程序利用该接口例程构成网络应用，来处理低层的网络活动。

TLI 是基于 ISO 传输层功能要求而设计的，利用 TLI 可以实现 ISO 模型的传输层协议的接口服务；还可以实现同 TCP、UDP 的接口。TLI 提供的例程支持无连接和有连接的通信服务。TLI 称相互通信的两个进程为传输端点，把在主计算机上的用户进程提供通信支持的一组例程称为传输提供者。

附录 A　SOCKET 系统调用简介

4.3　BSD UNIX 系统网络的 I/O 基础集中一种称为套接字(SOCKET)的概念，它是 UNIX 系统的文件访问机制产生者，提供通讯端点。套接字与文件描述符的主要区别是应用程序在调用“打开”时，操作系统将文件描述符与一个特定的文件或设备绑在一起，但套接字的产生则不必将它绑在特定的目标地址上。应用程序既可以每次提供一个使用套接字目标地址，传输数据报时也可将目标地址绑在套接字上，从而避免了每次重复指定目标地址。无论那种情况，套接字的功能如同 UNIX 的文件或设备，可用传统的“读写”操作对它进行访问。例如，一旦两个应用进程建立了套接字并在它们之间形成了 TCP 连接，一个进程可以用“写”操作发送数据而另一个进程可以用“读”操作接收该数据。

建立套接字

套接字系统调用要求创建套接字。要接收三个整型参数并返回一个整型结果：

result = socket（af，type，protocol）

参数 AF 表示该操作使用的协议族，即决定套接字按整样的方式翻译提供的地址。现

行

协议族包括 INTERNET 网际协议（AF-INET），施乐公司 PUP 网际协议（AF-PUP），苹果计算机公司 APPLETALK 网络（AF-APPLETALK）和 UNIX 的文件系统（AF-UNIX）以及许多其它协议族。

参数 TYPE 表示要求的通讯类型。包括可靠的数据流传送服务（SOCK–STREAM）和无连接的数据报传送服务（SOCK_DGRAM），以及一个原始类型（SOCK_RAW）用以允许特权程序对低层协议或网络接口进行访问。

第三个参数是使用的协议族。一般为 0。

返回值是建立的套接字描述符，可由它来进行套接字访问。

套接字的关闭

当一个进程完成对某个套接字的使用时，它调用关闭：

CLOSE（socket）

参数 socket 表示被关闭的套接字描述符。

确定本地地址

初始化时，套接字被创建时没有与任何本地或目的地址相联系。一旦建立了一个套接字，

服务进程调用捆扎（bind）系统调用为自己建立一个本地地址，如下所示：

bind（socket，localaddr，addrlen）

参数 socket 是被捆扎的套接字的整数描述符，

参数 localaddr 确定套接字被绑定的本地地址的一种结构，通常被称为 sockaddr（套接字地址）的固定结构以一个 2 字节的段开始，用于确定协议族，其后 14 字节专门用于该家族的信息。Internet 网络协议族的地址以 sockaddr_in 的形式出现，用 2 字节表示协

议族的值 AF_INET，接着是 2 字节协议端口号和 4 字节的 IP 地址，剩余的 8 字节未用。

参数 addrlen 是指以字节计量的该地址长度。

与目标地址相连

最初，套接字不与任何目标相连接。连接（connect）这一系统调用将一个常用目标同一个套接字捆扎在一起，并将套接字置为连接状态。应用程序在通过可靠性数据流套接

字传送数据前必须用“连接”来建立一种连接。套接字与无连接数据报服务一起使用时

不必事先建立连接。调用格式如下

connect（socket，desaddr，addrlen）

参数 socket 是被连接的套接字的整型描述符，

参数 desaddr 是用于确定该套接字被绑定的目标地址的一种结构，

参数 addrlen 是以字节计量的目标地址的长度。

通过套接字发送数据

一旦应用程序建立了一个套接字，就可以用它发送数据。有五种系统调用：send，write，writev，sendto，sendmsg，前三个只允许与处于连接状态的套接字一起工作，因为他们不允许调用者再确定目标地址。

write（dcsriptor，buffer，length）。参数 dcsriptor 是套接字的整型描述符，参数 buffer 是存放要发送数据的缓冲区地址，参数 length 是这些数据的字节数目。

writev 除了采用“集中写”外，其他与 write 功能相似，他使应用程序可以写一个报文而

不必将其拷贝成连续的字节形式。

Send 系统调用格式如下： send （ socket，message，length，flags）

参数 socket 表示要使用的套接字描述符，参数 message 表示被发送的字节序列的地址，参数 length 表示被发送序列的字节长度，参数 flags 控制发送。Flags 有一个值可以使发送者指出，报文应在套接字频带之外发送，另一个值允许调用者不使用局部路由表发送报文，使调用者可以控制路由选择，并可以编写网络调试软件。

Sendto 和 sendmsg 这两个系统调用允许调用者用非连接状态的套接字发送报文。因为两者均要求调用者指明一个目标。Sendto 系统调用有一个参数是目标地址。

Sendto （socket，message，length，flags，destaddr，addrlen）

其中后两个参数分别确定了目标地址和地址的长度。

当 sendto 系统调用要求的参数表过长而引起程序效率降低或难读时，可以选用 sendmsg 系统调用，形式如下：

sendmsg （socket，messagestruct，flags）

参数 messagestruct 包括将要发送报文的一种结构，还包括发送报文的长度，目标地址和目标地址的长度。

通过套接字接收数据

与五种输出操作相似，4.3 BSD UNIX 提供了五种系统调用使进程可以用它们从套接字上接收数据：read，readv，recv，recvfrom 和 recvmsg。

传统的 UNIX 的输入操作 read 只能由套接字处于联接状态时工作，格式如下：

read（descriptor，buffer，length）。参数 descriptor 是一个整型套接字描述符或文件描述符，用来指示要读取的数据的来源，参数 buffer 是数据的存储地址，参数 length 是要读取的数据的最大字节数。

另一种供选择的形式 readv 允许调用者使用一种“分散读”形式的接口，把到来的数据

放到非连续的单元中，形式如下：

readv（descriptor，iovector，vectorien）

参数 iovector 给出一个 iovec 型结构的地址，该结构包含一系列指向存放到来数据的存储

块的指针，参数 vectorien 表示 iovector 中入口的编号。

系统提供了三种系统调用来用于网络报文输入。进程可以调用 recv 从一个处于连接状态

的套接字上接收数据。其形式如下：

recv（socket，buffer，length，flags）

参数 socket 是一个套接字描述符，buffer 参数表示将要接收报文的存储器地址，参数 length 指出缓冲区长度，参数 flags 使调用者能控制接收。

Recvfrom 系统调用允许调用者从非连接状态的套接字上接收数据，它有附加参数指出接受数据的来源，形式如下：

Recvfrom（socket，buffer，length，flags，fromaddr，addrlen）

两个附加的参数 fromaddr 和 addrlen 分别指向套接字地址结构的指针和一个整数指针，前者记录报文发送者的地址，后者记录发送者的地址长度。

系统调用 Recvmsg（socket，messagestruct，flags）中的参数 messagestruct 表示一个结构地址，包含要到达的报文地址，也包括发送者的地址。

确定服务器队列长度

listen 系统调用允许服务器为到来的连接预备一个套接字，即将套接字放在一种被动状态

中准备接收连接。服务器调用 listen 时，还要通知操作系统：同时到达套接字的多个请求应由协议软件把它们排入队列。调用格式如下：

listen（socket，qlength）

参数 socket 是服务器使用的套接字的描述符，参数 qlength 表示上述套接字上请求队列的长度。这个系统调用只能用在选择可靠数据流传送服务的套接字上。

服务器接收连接

一旦建立一个套接字，服务器就需要等待一个连接，使套接字与某个外部目标连接。通过系统调用 accept 直到一个连接请求到达时才能实现。调用形式如下：

newsock = accept（socket，addr，addrlen）

参数 socket 确定处于等待状态的套接字的描述符；参数 addr 是指向一个 sockaddr 型的结构指针。当一个请求到达时，系统在 addr 中存放发出请求的目标客户的地址，addrlen 存放地址长度。系统创建一个连接发出请求的目标客户的新套接字，并将他的描述符返回给客户调用者。

附录 B　User.def 的举例

```
@dnl*************************************************
@dnl      Project Configuration File for SNX Demo
@dnl*************************************************
make.root.name:        myclient        /*   应用程序名   */

sys.verbose.major:   yes
sys.verbose.minor:    yes

sys.verbose.panic:    yes
make.target.debug:      yes            /*   调試工具   */

@dnl*************************************************
@dnl       Include necessary VRTXsa Components
@dnl*************************************************
rpc.enabled:            no                    远程调用
nfs.enabled:            no
ifx.enabled:             no
esh.enabled:            no
snx.enabled:            yes              /*SNX 网络开发工具*/
rtl.enabled:            yes

@dnl*************************************************
@dnl       Include OS, target and tool definitions
@dnl*************************************************

@include(system.def)
@include(pcat.def)
@include(microtec.def)


@dnl*************************************************
@dnl              Set SYSTEM ENTRY POINTS, Application Modules      *
@dnl*************************************************
sys.entry_point2:      main                    /*主调函数名*/
sys.objs.usr:        ne2000.obj,myclient.obj,

@dnl*************************************************
@dnl             Define Actual Devices for PC Target
@dnl*************************************************

@dnl    Remove serial_x if used for XRAY interface
@dnl   Add ether_1 if SNX enabled
       board.devices:    board, screen, timer_1, serial_2, ether_1

@dnl*************************************************
@dnl             Define VRTX Logical Devices
```

```
@dnl************************************************
vrtxos.console:            DEV_SCREEN
sys.env.devices:           board,console,timer,network

sys.env.dev.network.value:             ether_1

#default console for startup messages
sys.env.dev.console.value:             screen

@dnl************************************************
@dnl                  Application  Memory  Requirements
@dnl************************************************
  xdm.size:             20000
  code.size:            200000

@dnl remove the following for 386ex systems
@dnl remove the following if PC RAM is 1MB or less
@dnl remove the following if total code+debugger size is < 640K

@include(memdefs\pcmemhi.def)

@dnl************************************************
@dnl          RunTime  Library  Configuration
@dnl************************************************
rtl.hardware.fp_support:          no
rtl.IFX_support:                    no
#rpc.pm_priority:               20
#nfs.userid: 588

#nfs.groupid: 100

@dnl ====================================
@dnl                  SNX  HOST  table
@dnl ====================================
snx.swk.priority:           1               /*SNX 初始化程序优先级*/
snx.hosts.serv6:            202.115.4.17
snx.hosts.mripc1:        202.115.4.18
snx.hosts.serv5:            202.115.4.10
snx.hosts.mripc2:           202.115.4.11
snx.hosts.list:             serv6,mripc1,serv5,mripc2
snx.local_host:               mripc1
snx.rarp.enable: no
snx.interface.lg0.addr:        202.115.4.18
#snx.interface.lg0.netmask: 0xffff0000
```

第五章 InterNiche 的 TCP/IP 协议简介

概述

一．InterNiche 的 TCP/IP 协议栈软件包含了大多数 INTERNET 网络协议栈，它被组织成若干个模块，基本上按照目录划分，下面我们简单描述一下这些目录结构及所包含的程序。

1．以下目录中的程序是与目标系统相对独立的程序：

- **inet ——IP 及 UDP 的有关源文件。**
- **TCP ——TCP 及 Socket 的有关源文件。**
- **Misclib ——与用户接口有关的程序和提供 TCP/IP 扩展功能的相关程序。**
- **modem——与 modem 有关的程序。**
- **ppp ——PPP (Point to Point Protocol) 网络接口程序。**
- **drivers ——与网卡有关的程序。**

2．以下目录包含的程序是依赖于目标系统的，只能在特定的目标系统上运行：

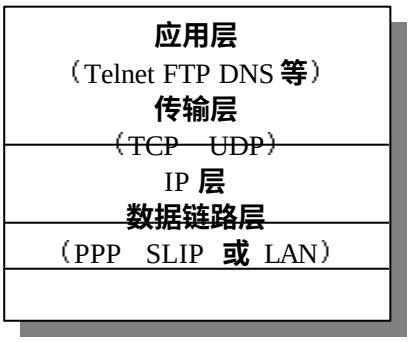
- **ace360 ——支持 ACE360 目标板。**
- **dostsr ——支持在 DOS TSR(终止及驻留程序)运行。**
- **dosmain ——支持 Intel x86 实模式，在嵌入式 DOS 应用程序中运行。**
- **dos4gw ——支持 Intel x86 保护模式，在 DOS 扩展应用程序中运行。**
- **net186 ——支持 AMD 的 net186 目标板。**
- **vrtxsa86 ——支持在 VRTX sa 实时操作系统（VRTX86/fpm）下运行的 Intel x86 处理器。**
- **vrtx68k/PPC ——支持在 VRTX sa 实时操作系统下运行的 Motorola**

68k/PPC 处理器。

3. 以下目录包含与应用有关的一些程序：

- **http** - HTTP (Hypertext Transport Protocol)服务器程序。
- **ftp** - FTP (File Transfer Protocol)客户和服务器程序。
- **snmp** - SNMP (Simple Network Management Protocol) 程序。
- **dhcpsrv** - DHCP (Dynamic Host Configuration Protocol) 服务器程序。
- **natrt** - NAT (Network Address Translation) 路由器程序。
- **telnet** - Telnet 服务器程序。
- **rip** - RIP (Routing Information Protocol)服务器程序。.
- **browser** –流览器协议程序

二．InterNiche 的 TCP/IP 协议栈软件的设计符合标准 TCP/IP 协议模型：



标准 TCP/IP 协议模型

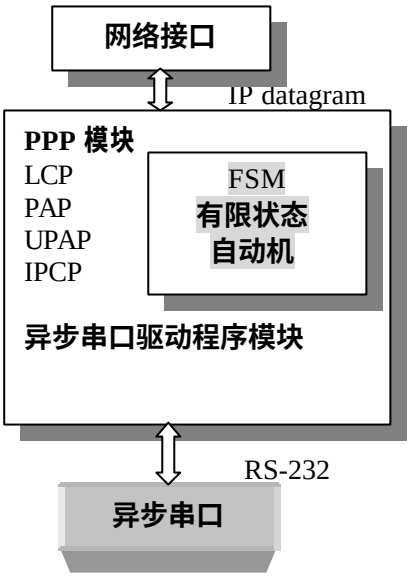
下面几章我们将分别就各层协议的详细设计作相应的分析说明。

5.1 数据链路层协议设计

第一节 PPP（point to point protocol）协议

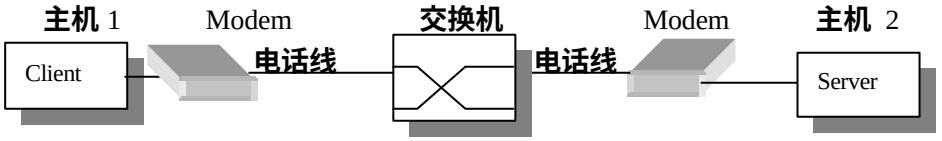
PPP 是属于数据链路层的点到点协议，实现链路控制、身份认证及上层

协议控制等功能，是处在网络接口层与异步串口之间的一个协议模块。



PPP 协议模块层次图

PPP 定义了在点对点链路上传送多种协议数据的协议。底层物理线路可以是直接相连的专线，也可以是电话线（我们这里主要介绍的是通过电话线相连的方式，即 Modem 方式）。通信方式可以是同步方式，也可以是异步方式。异步方式下通信速率为 1200bps 到 38400bps，同步方式下速率可达 6.4 Kbps，通过 PPP 链路可以实现网络与网络之间，路由器与路由器之间的互联。



主机间通过 PPP 实现通讯的实际线路图

由于 PPP 协议的建立连接、接收及发送是与 Modem 的建立连接、接收命令及发送命令联系在一起的，因此我们将这两个过程综合在一起分析。

1．初始化过程

1） **函数描述：**

prep_ppp( )

语法描述： int prep_ppp(int firstnet)

功能描述：将网络接口初始化为支持 PPP 协议的串型接口。

Init_ppp( )

语法描述： int init_ppp(int iface)

功能描述：在没有建立连接之前完成 PPP 初始化的大部分工作。

Ppp_port_init( )

语法描述：int ppp_port_init(int unit)

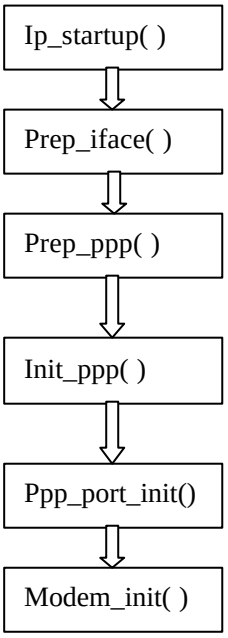
功能描述：完成对接口的初始化，包括与接口有关的一些数据结构及表的初始设置，并调用 modem_init()，**实现对硬件** modem **的初始化。**

Modem_init( )

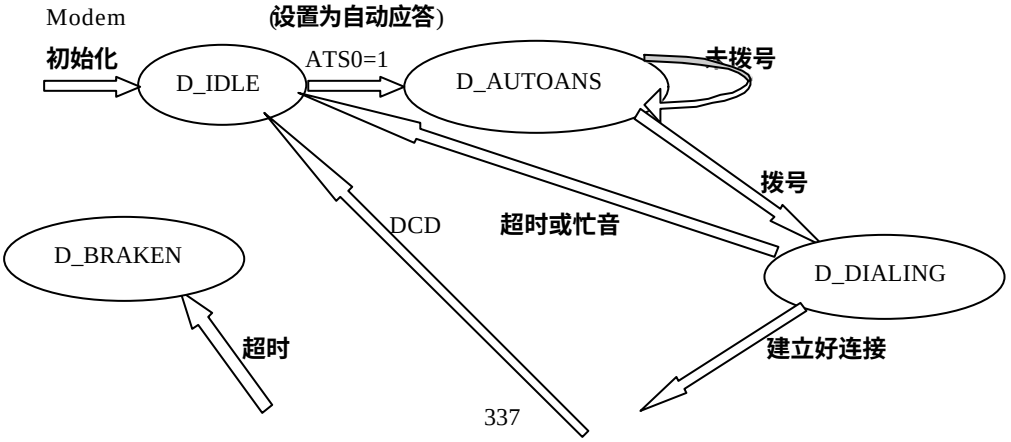
语法描述：init modem_init(int unit)

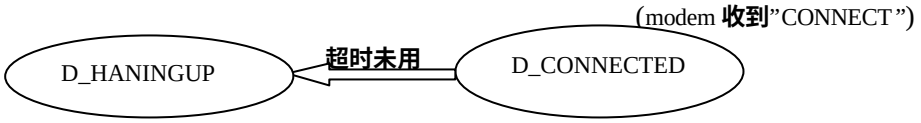
功能描述：完成对硬件 modem **的初始化，包括对串口的驱动及** modem **初始化串的设置，并向串口发送初始化命令。**

2） **流程图 (函数调用关系图)**



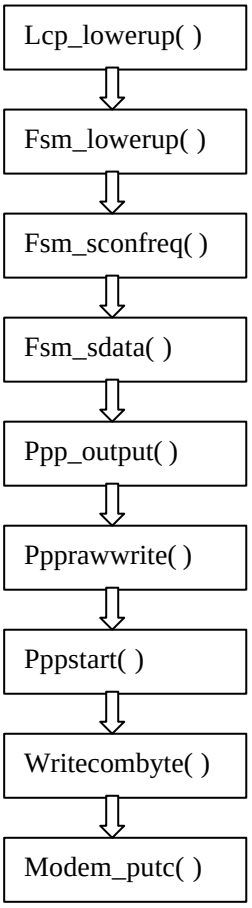
2. Modem 的状态转换(client 的状态转换图)





3．发送过程

1) **在硬件连接建立以后**，client **与** server **之间进行交互式的登录过程，登录正确后切换到** PPP **过程。对于** client **有下列执行流程：**



2）**函数描述**

lcp_lowerup

语法描述：void lcp_lowerup(int unit)

功能描述：配置网络接口的最大发送传输率和最大接收传输率，并调用 fsm_lowerup **函数转动有限状态机直到** Lcp **协商成功，即** Lcp **为** OPENED **状态。**

Fsm_lowerup

语法描述：void fsm_lowerup(fsm *f)

功能描述：使 ppp 层的有限状态机处于 STARTING 状态，并调用 fsm_sconfreq()函数发送初始配置请求。

Fsm_sconfreq

语法描述：static void fsm_sconfreq(fsm *f,int retransmit)

功能描述：构造一个请求包，并调用 fsm_sdata()函数，发送一个配置请求。

Fsm_sdata

语法描述：void fsm_sdata(fsm *f,u_char code,u_char id,u_char *data,unsigned datalen)

功能描述：发送数据到对等主机的对等实体。

Ppp_output

语法描述：void ppp_output(int unit, u_char data, int len)

功能描述：检查接口，并调用 ppprawwrite()函数发送原始的 ppp 协议包。

Ppprawwrite

语法描述：int ppprawwrite(int unit, char *buf, int nlen)

功能描述：在 ppp 链路上发送 ppp 协议包。

Pppstart

语法描述：void pppstart(struct ppp_softc *sc)

功能描述：开始发送一个完整的包到接口上。

Writecommbyte

语法描述：int writecommbyte(int unit, u_char bByte)

功能描述：写一个字节的数据到串口上。

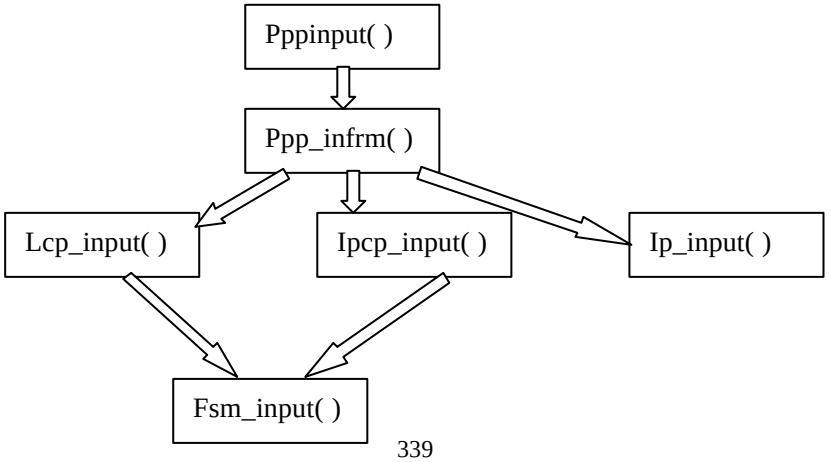
Modem_putc

语法描述：int modem_putc(int unit, u_char bByte)

功能描述：发送一个字节的数据到 modem 连接上。

4．接收过程

1) **接收流程图**



2）函数描述

pppinput

语法描述：void pppinput(int unit, int c)

功能描述：接收 PPP **链路上的一个字符（已被中断接收到** freebuf **中），在** modem\dialer.c **文件中被** dial_check()**函数周期调用，主循环** tk_yield()**每循环一次调用一次** dial_check()**，接收一个字符，对程序的接收效率有一定的影响，是值得改进的地方。**

Ppp_infrm

语法描述：void ppp_infrm(int unit)

功能描述：由 pppinput( )**调用，接收一帧数据，并根据帧的类型分别调用相应的接收函数。**

Lcp_input

语法描述：void lcp_input(int unit, u_char *p, int len)

功能描述：接收各种 LCP **包，并调用函数** fsm_input( )**根据收到的** LCP **包的类型进行状态迁移。**

Ipcp_input

语法描述：void ipcp_input(int unit, u_char *p, int len)

功能描述：接收各种 IPCP **包，并调用函数** fsm_input( )**根据收到的** IPCP **包的类型进行状态迁移。**

Fsm_input

语法描述：void fsm_input(fsm *f, u_char inpacket, int l)

功能描述：接收 PPP **包，并根据包的类型调用有关函数，进行状态迁移。**

在 fsm_input( )调用了一些与状态迁移有关的函数，它们是：

fsm_rconfreq( ) 处理接收到的配置请求包

fsm_rconfack( ) 处理接收到的配置应答包

fsm_rconfnakrej( ) 处理接收到的配置拒绝包

fsm_rtermreq( ) 处理接收到的终止请求包

fsm_rtermack( ) 处理接收到的终止应答包

ip_input

语法描述：void ip_input(int unit, u_char *data, int length)

功能描述：在 ppp **链路建立好之后，通过此链路接收** IP **包。**

5．主要数据结构

typedef struct fsm_callbacks{ ｡｡｡｡｡｡ }fsm_callbacks;

/*安装与 FSM **有关的例程*/**

typedef struct fsm{ ｡｡｡｡｡｡ }fsm; /*与 ppp **有限状态机有关的数据**

结构*/
typedef struct ppp_softc{ 。。。。。。}PPP_SOFTC;
/*与接口控制有关的数据结构*/
struct ppp_header{ 。。。。。。} /*PPP 头结构*/
struct ip { 。。。。。。}; /*IP 头结构*/
struct tcphdr{ 。。。。。。}; /*TCP 头结构*/

6．与接口 driver 有关的函数列表

与接口有关的函数主要安排在 sys_np.c 和 ifppp.c 程序中，列表如下：

ifppp.c 中

pppopen()：为建立 PPP 连接构造一个 ppp_softc 结构。
pppwrite()：在 PPP 链路上发送 IP 包。
ppprawwrite()：在 PPP 链路上发送 PPP 包。
pppstart()：发送一个完整的包到接口上。
Ppp_setinbuf()：根据当前的 MRU 分配 PPP 连接输入缓冲区。
Ppp_infrm()：处理从 PPP 链路上接收到的一个完整的帧。
Pppinput()：处理从 PPP 链路上接收到的一个字节数据。

Sys_np.c 中

Establish_ppp()：在 modem 端口建立 PPP 连接。
Link_terminated()：终止 PPP 连接。
Link_established()：当 LCP 有限状态自动机到达 OPENED 状态时被调用。
Link_required()：当 PPP 核心想在已知的接口上建立连接时被调用。
Prep_ppp()：这是第一个被调用的 PPP 例程，用于初始化与接口有关的一些数据结构。
Shutdown_ppp()：关闭整个 PPP 链路。
Ppp_quit()：完成结束 PPP 的一些善后处理，如复位一些数据结构。
Ppp_output()：调用 ppprawwrite()发送 PPP 包。
Ppp_pkt_output()：通过 PPP 链路发送一个包（PPP 包或上层协议包）。
Ppp_send_config()：配置 PPP 接口的发送参数。
Ppp_recv_config() ：配置 PPP 接口的接收参数。
Init_ppp()：PPP 初始化。
Ppp_timeout()：PPP 超时处理。
Ppp_timeisup()：这个例程应该被系统周期性的调用，驱动 PPP 内部时钟。
Writecommbyte()：串口驱动程序，调用发送例程向串口发送一字节数据。
Ip_input()：在 PPP 链路上接收 IP 分组。
Ppp_check()：检查 PPP 链路是否准备好。

第二节 其它链路层协议

Interniche TCP/IP 协议软件还提供了简单的数据链路层协议（SLIP）和地址

解析协议（ARP）作为可选项，例如：在使用网卡而不是通过 MODEM 进行通讯时，链路层应使用 ARP 协议将 IP 地址映射为物理地址，关于 SLIP 协议和 ARP 协议的实现这里不再赘述。

5.2 IP 协议设计

第一节　　IP 协议简介

IP 协议的主要功能包括无连接数据报传送、数据报寻径以及差错处理三部分。IP 协议是点到点的，且非常简单，不能保证传输的可靠性。

下面我们将分成几部分，对 InterNiche TCP/IP 软件进行讨论：

1） ICMP 提供的差错报告和控制功能
2） 分片与重组
3） IP 寻径

第二节　　差错与控制报文（ICMP）

这部分程序主要位于 icmp.c、ping，app_ping.c 中

1．Icmp.c 中的程序

Icmprcv

语法描述：int icmprcv(PACKET p)

功能描述：被 ip_demux.c 中的 ipdemux()函数调用（若接收到 ICMP 报文），将接收到的报文再根据其类型（如差错报文或请求/应答报文等），分别调用相应的函数进行处理。

Icmp_du

语法描述：void icmp_du(PACKET p,struct destun *pdp)

功能描述：处理接收到的一个目标不可到达包。

2．Ping.c 中的程序

IcmpEcho

语法描述：int icmpEcho(ip_addr host, char *data, unsigned length, unshort pingseq)

功能描述：向目标主机发送一个“ping”包。

3．App_ping.c 中的程序

Ping_init

语法描述：int ping_init(void)

功能描述：初始化“ping”操作，调用函数 install_menu()将 ping 的菜单 settings 安装到主菜单上。

Ping_new

语法描述：PING_INFO ping_new(void)

功能描述：为 PingInfo 结构分配相应内存，将各字段设置为缺省值，并调用函数 ping_addq()将该结构加到一个全局队列中，为函数 ping_check()提供查询。

Ping_delete

语法描述：int ping_delete(PING_INFO p)

功能描述：调用函数 ping_delq()从全局队列中取下指定的一个 PingInfo 结构，并释放已分配的缓冲区。

Ping_addq

语法描述：int ping_addq(PING_INFO p)

功能描述：向全局队列中加入相应的 PingInfo 结构。

Ping_delq

语法描述：int ping_delq(PING_INFO p)

功能描述：从全局队列中取下相应的 PingInfo 结构。

Ping_search

语法描述：PING_INFO ping_search(GEN_IO pio)

功能描述：查询 PingInfo 结构列表，找到一个匹配。

Ping_start

语法描述：int ping_start(void *pio)

功能描述：是由键盘输入“ping xx”命令后执行的第一个程序，由菜单调用，处理命令行参数，并调用函数 ping_send()发送“ping”命令。

Ping_send

语法描述：int ping_send(void *pio)

功能描述：调用函数 icmpEcho()发送”ping”包。

Ping_end

语法描述：int ping_end(void *pio)

功能描述：结束一个正在进行的“ping”过程。

Ping_check

语法描述：void ping_check()

功能描述：被周期性调用，用来检查当前“ping”过程的状态，确定是否有响应，若有则显示正确 reply 信息，并在必要时发送更多的 ping 包。

Ping_demux 略

Ping_setdelay 略

Ping_setlength 略

Ping_sethost 略

Pingupcall 略

4．用 ping 命令简单测试 IP 层的工作情况

在许多 TCP/IP 实现中，用户命令“ping ”利用 ICMP 回应请求/应答报文测试 IP 及 IP 下层协议的连接、工作情况。我们这里也用同样的方法测试我们的 IP 软件。其测试方法为：

在键盘上或直接在数组中输入命令行：“ping xxx（目的 IP 地址）”，由菜单操作直接调用函数 ping_start( ), 解析地址并最后调用函数 ping_send( ) 发送第一个 ping 包。另一方面，周期性调用函数 ping_check( )检查是否收到应答，若正确收到一个 ICMP 回应应答报文，显示 ping reply …..。测试成功。

第三节 IP 分片与重组

1．要实现 IP 分片与重组，首先在 ipport.h 中定义#define IP_FRAGMENT 1。实现分片的函数为 **ip_fragment()**，当待发数据报的长度大于该网络的最大传输单元 MTU 时调用该函数。该函数主要完成一些计算，若有必要，将大包分成小包并分别加上报头，形成新的发送分组。

IP 报文的重组主要在 ip_reasm.c 中实现，其中包括的主要函数有：

Ip_reassm

语法描述：int ip_reassm(PACKET newp)

功能描述：重组 IP 分片。若该分片不是最后一个分片，则只是存包并记录下当时的时戳；若该分片已经是最后一个分片，即已经收到一个完整的包，则向上调用函数 ip_demux()去分析该包。该函数在必要时被函数 ip_rcv()调用。

Ip_frag_check

语法描述：void ip_frag_check()

功能描述：检查重组是否超时，若超时则放弃所有已经重组好的分组。

Frag_punt

语法描述：void frag_punt(struct frag_waiting *p)

功能描述：放弃已重组好的分组，释放相应的空间。

2．与 IP 分片、重组有关的数据结构是：

```
struct   frag_waiting {
        PACKET     p;                 /*分片重组的报文*/
        HOLE       hole_list;         /*指向第一个“HOLE ”结构的指针*/
        Unsigned   totalrx;           /*迄今收到的总字节数*/
        Ulong      timestamp;         /*最近分片到达的时间*/
        }
```

并定义了 **struct frag_waiting fragbufs[MAX_FRAG_PKTS];** 用于存放各个未完全到达的分片重组报文。

第四节 IP 寻径（路由）

1．实现 IP 寻径，首先#define IP_ROUTING 1 或#define MULTI_HOMED 1。
IP 寻径由函数 iproute()实现，该函数被 ip.c 中的 ip_write()发送函数调用，在发送 IP 包之前查路由表确定下一跳的地址。另外的函数，如 add_route()和 del_route()为路由表增加新的表项或删除无用的表项。（这三个函数定义都在 ip.c 中）

2．与 IP 寻径有关的数据结构定义如下：

```
struct ipRouteEntry_mib{ …..};
define RtMib ipRouteEntry_mib;
typedef struct RtMib *RTMIB;
```

这些数据结构主要用于对路由表的管理，也用于 RIP 协议中。

3．RIP 协议

寻径信息协议（RIP，Routing Information Protocol）是最广泛使用的 IGP（内部网关协议）之一。RIP 将协议的参与者分为主动机（active machine）和被动机（passive machine）两种，主动机主动地参与听取其他网关传来的 RIP 信息，将新路由添加到选路表中，并发送包含有更改后的选路表表项的消息。被动机被动地听取网关发送的 RIP 消息，从中提取选路信息，并更改自己的选路表，但不传播本地选路表中的信息。一般情况下，网关作主动机，它主机作被动机。RIP 使用矢量距离(vector-distanse)算法传播路由，并利用本地网广播交付报文。每个网关定期地将它现有的 IP 选路表中路由广播发送给其上的所有网络接口。

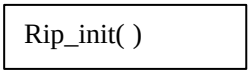
RIP 提供两种基本的报文类型。它允许一个客户机发送 request 报文来询问特定路由，并提供 response 报文用以应答 request 报文或是周期性地通告路由。总的来说，很少有客户机来轮询更新。事实上，大多数的实现方案依赖网关来产生一个周期性地更新报文。周期性广播报文的术语是”无偿响应”,因为这是在没有 request 报文使它产生的情况下，仍然使用 response 报文类型。

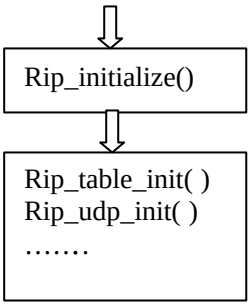
RIP 的具体实现有两种形式：在第一种实现形式下，RIP 本身的选路数据库与 IP 用来转发数据报的选路表是分开的;在第二种形式下，RIP 与 IP 共同使用一张选路表。在 InterNiche 提供的 RIP 实现中采用的是第二种形式。

以下我们通过几个设计流程简单分析在 InterNiche 中对 RIP 协议的实现。

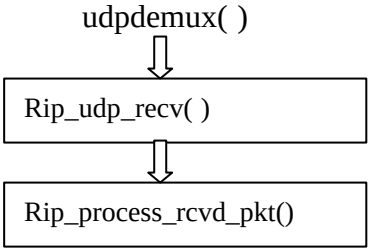
1） 初始化

实现对有关数据结构的初始化、打开 UDP 相应端口及对菜单的安装等功能。





2）输入处理

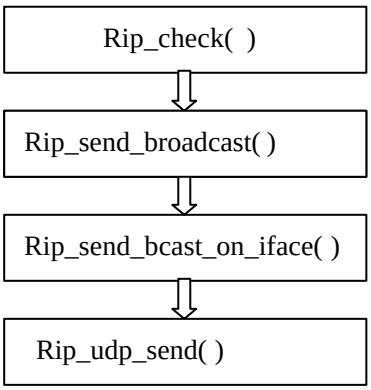


rip_process_rcvd_pkt( )**首先检查版本号，然后检查报文首部相应字段确定是请求还是响应报文。若为请求报文调用函数** rip_porcess_rcvd_req_pkt( )**对其进行处理；若为响应报文，如果有必要，更新选路表。**

3）输出处理

网关定期发送 RIP **响应。函数** rip_check()**用来实现这个功能，它应被应用程序周期性的调用（一般间隔是** 30 **秒钟）。**

基本流程如下：



5.3 传输层协议设计

第一节　传输层概述

传输层是计算机网络（包括网间网）体系结构中至关重要的一层。传输层的作用是向信源机提供端到端数据传输，而传输层以下各层只提供相邻机器的点到点传输。在 TCP/IP 协议栈的传输层中同时提供无连接的 UDP 和面向连接的 TCP，其中 UDP 适用于可靠性较高的局域网，一次传输交换少量报文，而 TCP 适用于可靠性较差的广域网，一次传输要交换大量报文的情形。

为了包容通信子网的各种不可靠因素，TCP 协议要做大量的工作，解决我们各种可靠性问题，因此 TCP 协议非常复杂。而 UDP 几乎直接建立在 IP 协议之上，相对简单得多，效率较高。

在 InterNiche TCP/IP 协议栈的传输层既提供了面向连接的可靠的 TCP，又提供了无连接的 UDP。下面章节中我们将分别进行讨论。

第二节　TCP 协议

传输控制协议 TCP 是传输层的一个重要协议，它是一个完整的传输协议的典范，除提供和 UDP 一样的进程通信能力外，其主要特点是可靠性高，几乎可以解决所有的可靠性问题。

TCP 的可靠性特点可以用一句话来概括：TCP 提供面向连接的流传输。面向连接对可靠性的保证首先是它在进行实际数据传输前，必须在信源端与信宿端建立一条连接，假如由于种种原因，连接建立不成功，则信源端不会贸然向信宿端发送数据。其次，面向连接传输的每一个报文都需接收端确认，未确认报文被认为是出错报文。InterNiche TCP/IP 通过三次握手协议、滑动窗口协议，超时重传等协议机制实现为上层用户提供一定的可靠性传输，并实现带外数据通道提供对紧急数据的处理，提高了效率。

InterNiche 的 TCP 协议全部在 TCP 目录下实现。

1．InterNiche 的 TCP 协议实现中涉及到的主要数据结构是：

Struct　inpcb{ ….};　　/*in_pcb.h 中*/

Struct　tcpcb{ ….};　　/*tcp control block 结构，tcp_var.h*/

2．实现初始化的过程

tcpinit：提供对与 TCP 有关的数据结构及部分应用的初始化。(tcpport.c 中)

nptcp_init：被 tcpinit()函数调用，分配有用的缓冲区。(nptcp.c 中)

tcp_init：被 nptcp_init()函数调用，实现链表初始化。(tcp_subr.c 中)

3．输入处理

当 IP 层判断已收到一个有效的 TCP 包，将向上调用函数 tcp_rcv()(nptcp.c 中)进行 TCP 层的接收。Tcp_rcv()继续调用函数 tcp_input()(tcp_in.c 中)，在该函数中实现了 TCP 的有限状态机，当一切资源就绪，调用函数 tcp_wakeup()唤醒应用层接收任务接收。

4．输出处理

InterNiche TCP 对普通数据和带外数据分别进行处理，但最终都调用函数 tcp_output()(tcp_out.c 中)向下层发送数据，在该函数中还实现了对滑动窗口的控制。

5．时间控制

应用程序中的 tk_yield()函数周期调用函数 tcp_tick()(nptcp.c 中)实现对 TCP 的时间控制，该函数继续调用 tcp_slowtimo()(tcp_timr.c 中)，最后由 tcp_times()(tcp_timr.c 中)完成对 TCP 超时重传等有关的时间处理。

6．其余与 TCP 有关的函数分别完成 TCP 的几大功能，这里不再赘述。

第二节　UDP 协议

用户数据报协议 UDP(User Datagram Protocol)建立在 IP 协议之上的另一种传输协议，同 IP 协议一样提供无连接数据报传输。相对于 IP 协议，它唯一增加的能力是提供协议端口，以保证进程通信。

许多基于 UDP 的应用在高可靠性、低延迟的局域网上运行很好，而一旦到了通信子网 QOS 很低的网间网环境下，可能根本不能运行。原因在于 UDP 不可靠，而这些程序本身又没有做可靠性处理。因此，基于 UDP 的应用程序在不可靠子网上必须自己解决可靠性(诸如报文丢失、重复、失序和流控等问题)。

既然 UDP 如此不可靠，为何 TCP/IP 还要采纳它？最主要的原因在于 UDP 的高效率。在实践中，UDP 往往面向交易型应用，一次交易往往只有一来一回两次报文交换，假如为此而建立连接和撤除连接，开销是相当大的。这种情况下使用 UDP 就有效了，即使因报文损失而重传依次，其开销也比面向连接的传输要小。(例如简单网络管理协议 SNMP 就是采用 UDP 作为传输协议的一个应用实例)

在 InterNiche 中 UDP 的实现也非常简单，主要有三个程序：

udpsock.c：实现与 socket 的接口。

udp.c：实现 UDP 报文解析及发送等功能，其中包括 udpdemux()、udp_send() 及 udp_socket()等函数。

udp_open.c：建立和关闭一个 UDP 连接。

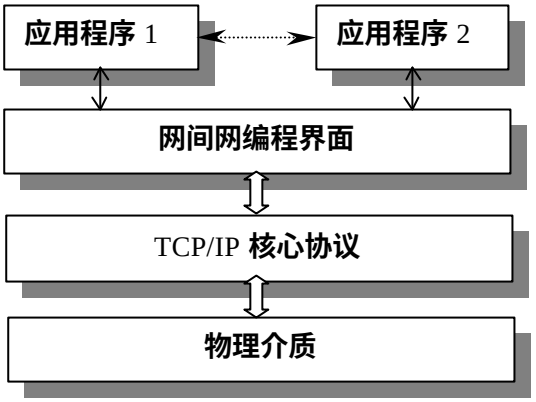
5.4　Socket 与应用基础

在 InterNiche TCP/IP 协议栈之上提供各种各样的应用，它们通过编程接口 Socket，在下层协议的支持下为用户提供各种服务。其中已实现并测试过的应用有：HTTP-GET、Web-Server 及 Telnet-Server 等。

第一节　Socket 的实现

建立在 TCP/IP 协议栈之上的应用程序不是直接与 TCP/IP 核心协议打交道的，它与之打交道的是核心协议提供的编程界面（Socket）。编程界面构成了核心协议的应用程序视图。

TCP/IP 核心与应用程序的关系如下图：



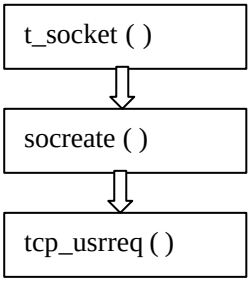
应用程序与 TCP/IP 核心协议

Socket 编程界面由 BSD UNIX 首先提出，目的是解决网间网进程通信问题。不管 socket 的内部机制如何，它提供给应用程序员的最终界面。是一组系统调用。其中常用的有：

socket ( ) ——创建 socket
bind ( ) ——指定本地地址
connect ( )与 accept ( ) ——建立 socket **连接**
listen ( ) ——用于面向连接服务器，表明它愿意接收连接
send ( )与 sendto ( ) ——发送数据
receive ( )与 receivefrom ( ) ——接收数据

InterNiche TCP/IP 中对 socket 的实现主要在 tcp 目录下，以下我们将对几个重要的系统调用的实现流程做简单的分析。

1．**socket ( )**



t_socket( )

语法描述：long t_socket(int family, int type, int proto)

功能描述：调用函数 socreate( )**创建** socket，**并返回一个唯一的** socket **号。**

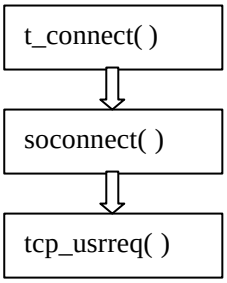
（接口函数，被应用程序调用）

socreate( )

语法描述： struct socket *socreate(int dom, int type, int proto)

功能描述：初始化与 socket **有关的数据结构，调用函数** tcp_usrreq( ),tcp_usrreq() **函数根据相应类型为** PRU_ATTACH，**则调用函数** tcp_attach( ),**确定** socket **的协议为** TCP,**并欲留** TCP **协议控制块。**

2．**connect ( )**



t_connect( )

语法描述： int t_connect(long s, struct sockaddr *addr)

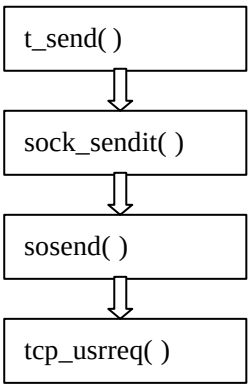
功能描述：调用 soconnect( )**请求建立连接。（接口函数，被应用程序调用）**

soconnect( )

语法描述： int soconnect(struct socket *so, struct mbuf *nam)

功能描述：检查连接状态，调用函数 tcp_usrreq( ),tcp_usrreq( )**函数根据相应类型为** PRU_CONNECT，**则初始化对等层连接，标识** socket **为** connecting **状态，并在该连接上发送初始段，使其状态变为** SYN_SEND。

3．**send ( )**

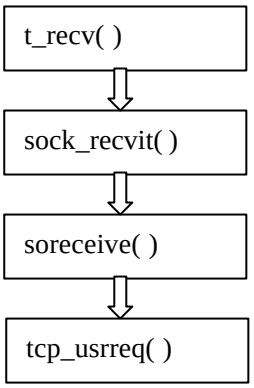


设置相应的数据结构，在 tcp_usrreq()**函数中根据相应类型为** PRU_SEND **或** PRU_SENDOOB **调用函数** tcp_output( )**分别进行带内和带外数据的发送。**

Recv( )

设置相应的数据结构，在 tcp_usrreq()函数中根据相应类型为 PRU_RCVD 或 PRU_RCVDOOB 分别进行带内和带外数据的处理。

与发送存在相同的问题，当不能接收时判断 socket 是否能等待（阻塞），若能阻塞，调用函数 sbwait( )，否则返回错误，并退出接收例程。



5.5 应用实例

InterNiche TCP/IP 不仅提供了多种应用，而且提供了一定的可移植性。我们就曾在 DOS 下、VRTX86/rm 下（基于 net186 目标板）,VRTXsa(x86,68k,PPC)下均实现了部分应用。以下我们将分别进行简要介绍：

1．**Dos 环境下**

InterNiche TCP/IP 提供了 dos 环境下的 demo 程序，实现了基于 ppp 和网卡的通信。能与标准 TCP/IP 正常通信。（其中基于网卡的程序还需有 ODI 驱动程序的支持）实现方法为：

在本机运行 InterNiche TCP/IP 程序，通过 Modem 或网卡与一正常的局域网相连，利用 ping 命令测试网络是否连通，并利用 HTTP 协议的 GET 命令测试 TCP/IP 是否能正常工作。

因为 dos 为单任务操作系统，所以在 dos 下实现的 InterNiche TCP/IP 应用程序只有一个任务，而 tk_yield()为主循环，被周期性调用，实现网络操作。

2．**VRTX86/rm 环境下（基于 net186 目标板）**

我们将 InterNiche TCP/IP 移植到 vrtxsa86/rm 操作系统下，并实现了简单的多任务。实现方法为：

将 net186 目标板（目标机）通过串口与主机（宿主机）相连，在宿主机上开发并生成*.abs 文件在目标板上运行。当通过 PPP 进行通信时，由于目标板只有两个串口（一个与宿主机相连用于下载、调试，一个与 modem 相连），没有多余的串口作为回显，因此给调试工作带来一定的困难，我们

的方法是将键盘输入该为在程序中直接写入，并将要回显的内容写入内存，调试过程中通过查看内存跟踪程序的执行；当通过网卡通信时，可用 terminal（超级终端）作为回显，由键盘作为输入。Vrtxsa86/rm 支持多任务，我们为了实现多任务机制，将 tk_yield()程序简化，增加 network_loop()任务，并把 ping 和 http 也作为应用任务实现。其中 network_loop()为网络主任务，优先级最高，ping_task 和 http_task 的任务优先级比它低一点，这些任务都在系统主任务 main_loop 中被创建，且 ping_task 与 http_task 被创建时为“挂起”状态，当初始化完成，并且有条件发送时，被唤醒执行。

3．**VRTXsa 环境下**（pc **机作为目标机**）

InterNiche TCP/IP **提供了** VRTXsa **环境下的应用实例，实现了** ping **和** Web Server **的功能，方法也很简单：将两台** pc **机通过串口相连，一台作宿主机，一台作目标机。在宿主机上编译、链接** VRTXsa **目录**（InterNiche TCP/IP **提供的**）**下的程序，生成***.abs **文件在目标机上运行。**

TCP Server Example

```
/* tcp_echo.c
 * Copyright by CESD(Centre of Embeded SoftWare Design) of UESTC.
 * All rights reserved.
 */

#include "tcpport.h"    /* embedded system includes */
#include "mbuf.h"        /* BSD-ish Sockets includes */
#include "socket.h"
#include "sockvar.h"
#include "sockcall.h"

/* define NP sockets to standard calls */
#define SOCKTYPE long
#define socket(x,y,z) t_socket(x,y,z)
#define bind(s,a) t_bind(s,a)
#define connect(s,a) t_connect(s,a)
#define listen(s,i) t_listen(s,i)
#define accept(s,a) t_accept(s,a)
#define send(s,b,l,f) t_send(s,b,l,f)
#define recv(s,b,l,f) t_recv(s,b,l,f)
#define socketclose(s) t_socketclose(s)
#define setsockopt(s, l, o, d) t_setsockopt(s, o, d)

#define ECHO_PORT    5000      /* standard UDP/TCP echo port */
```

```
SOCKTYPE elisten_sock = INVALID_SOCKET;   /* echo server socket */
SOCKTYPE esrv_sock = INVALID_SOCKET;      /* echo server active socket */

int    tcp_echo_close(void);    /* internal */

static     u_long e_replies;
char *cp;
char *src_ip = "192.9.200.3";
/*char *src_ip = "202.115.17.128";*/
static     ip_addr    my_addr;

extern void putstr(char *);

int
tcp_echos_init()
{
int e;        /* error holder */
struct sockaddr_in me;   /* my IP info, for bind() */
unsigned snbits;

        putstr("tcp server starting.\n");
        cp = parse_ipad(&my_addr, &snbits, src_ip);

        /* open TCP socket */
        elisten_sock = socket(AF_INET, SOCK_STREAM, 0);
         if(elisten_sock == INVALID_SOCKET)
          {
              putstr("bad socket.\n");
     return -1;
          }

         me.sin_family = AF_INET;
         me.sin_addr.s_addr = my_addr;
         me.sin_port = htons(ECHO_PORT);

        e = bind(elisten_sock, (struct sockaddr*)&me);
         if(e != 0)
          {
```

```
    e = t_errno(elisten_sock);
          putstr("bad socket bind.\n");
    return e;
       }
      e = listen(elisten_sock, 3);
      if(e != 0)
       {
    e = t_errno(elisten_sock);
          putstr("bad socket listen.\n");
    return e;
       }
     /* for listen socket into Non-blocking mode so we can poll accept */
      setsockopt(elisten_sock, SOL_SOCKET, SO_NBIO, NULL);

      return 0;
}

int
tcp_echo_close()
{
int e = 0;          /* scratch error holder */
int retval = 0;   /* return last non-zero error */

   putstr("tcp server closing.\n");

   if(esrv_sock != INVALID_SOCKET)
    {
      e = socketclose(esrv_sock);
       if(e)
       {
    retval = e = t_errno(esrv_sock);
          putstr("close error.\n");
       }
      esrv_sock = INVALID_SOCKET;
    }

   if(elisten_sock == INVALID_SOCKET)
       return e;
```

```
    e = socketclose(elisten_sock);
     if(e)
     {
        retval = e = t_errno(elisten_sock);
         putstr("server close error.\n");
     }
    elisten_sock = INVALID_SOCKET;

    return retval;
}

static char inbuf[TCP_MSS];
static int in_echopoll = 0;    /* re-entry flag */

void
tcp_echo_poll()
{
int len;                /* length of recv data */
int e;                   /* error holder */
int i = 0;                    /* generic index */
int count = 0;
struct sockaddr_in client;
SOCKTYPE tmpsock;    /* scratch socket */

    if(elisten_sock == INVALID_SOCKET)
        return;           /* Echo not set up, don't bother */

     in_echopoll++;       /* don't re-enter from net_loop() */
    if(in_echopoll != 1)
     {
         in_echopoll--;
         return;
     }

    while(esrv_sock != INVALID_SOCKET)
     {
        len = recv(esrv_sock, inbuf, TCP_MSS, 0);
         if(len < 0)
          {
```

```
      e = t_errno(esrv_sock);
      if(e != EWOULDBLOCK)
                putstr("TCP receive error.\n");
      in_echopoll--;
      return;
          }
        else if(len == 0)
          {
      in_echopoll--;
      return;
          }
       else /* if(len > 0) - got some echo data */
          {
       /* we must be server, send echo reply */
       e = send(esrv_sock, inbuf, len, 0);
       if(e < 0)
          {
            /* Print the error to console */
             e = t_errno(esrv_sock);
                 putstr("TCP echo server, error sending reply.\n");
           }
        continue;
           }        /* e>0*/
   } /* end of for */

   /* check for received echo connection on server */
   tmpsock = accept(elisten_sock, (struct sockaddr*)&client);
    if(tmpsock != INVALID_SOCKET)
     esrv_sock = tmpsock;
     in_echopoll--;
}
```

TCP Client Example

```
* tcpcln.c
 * Copyright by CESD(Centre of Embeded SoftWare Design) of UESTC.
 * All rights reserved.
  */

#include "tcpport.h"    /* embedded system includes */
```

```
#include "menu.h"
#include "mbuf.h"        /* BSD-ish Sockets includes */
#include "socket.h"
#include "sockvar.h"
#include "sockcall.h"

/* define NP sockets to standard calls */
#define SOCKTYPE long
#define socket(x,y,z) t_socket(x,y,z)
#define bind(s,a) t_bind(s,a)
#define connect(s,a) t_connect(s,a)
#define listen(s,i) t_listen(s,i)
#define accept(s,a) t_accept(s,a)
#define send(s,b,l,f) t_send(s,b,l,f)
#define recv(s,b,l,f) t_recv(s,b,l,f)
#define socketclose(s) t_socketclose(s)
#define setsockopt(s, l, o, d) t_setsockopt(s, o, d)

#define ECHO_PORT    5000     /* standard UDP/TCP echo port */

extern int kbhit(void);  /* from Microsquash|Borland library */

int     tcp_echo_close(void);     /* internal */

SOCKTYPE ecl_sock = INVALID_SOCKET;
static     char *     echodata;
static     u_long     e_replies;
static     ip_addr    serv_addr;
char *cp;
char *dest_cp = "192.9.200.33";
/*char *dest_cp = "202.115.17.222";*/
int    times = 1;
extern void putstr(char *);

int
tcp_sendecho()
{
long i;
```

```
struct sockaddr_in sa;
int e;
SOCKTYPE tmp;
char * arg2;
unsigned snbits;
int    len;

        if(ecl_sock != INVALID_SOCKET)
          {
      socketclose(ecl_sock);
      ecl_sock = INVALID_SOCKET;
            putstr("Close socket,Please try again.\n");
      return -1;
     }
        arg2 = nextarg(cbuf);                              /* get 1 arg from command
     line */
       cp = parse_ipad(&serv_addr, &snbits, dest_cp);
       tmp = socket(AF_INET, SOCK_STREAM, 0);
        if(tmp == INVALID_SOCKET)
         {
           putstr("tcp echo: can't open socket\n");
      return -1;
         }

        sa.sin_family = AF_INET;
       /* host is already in network endian */
        sa.sin_addr.s_addr = serv_addr;
        sa.sin_port = htons(ECHO_PORT);

       e = connect(tmp, (struct sockaddr*)&sa);
        if(e != 0)
         {
      e = t_errno(tmp);
           putstr("tcp_echo: bad socket connect.\n");
      return e;
         }
     /* Put in non-block mode */
        putstr("connect established.\n\r");
       setsockopt(tmp, SOL_SOCKET, SO_NBIO, NULL);
```

```
        ecl_sock = tmp;

    e_replies = 0;
    for(i = 0; i < times; i++)
     {
        putstr("sending TCP echo.\n");
        len = sizeof(arg2);
         sprintf(echodata,arg2, len);

        echodata[len] = '\0';
       e = send(ecl_sock, echodata, len, 0);
         putstr("send\n\r");
        if(e != len)
         {
      if(e < 0)
      {
          e = t_errno(ecl_sock);
                putstr("error sending TCP echo.\n");
          dtrap();
          return -1;
      }
      else
                  putstr("tcp_echo: could only send x bytes\n");
         }

     }
     return 0;
}
```

应用开发篇一:XRAY、SPECTRA 及工具

第一章 应用开发工具

- VRTXsa X86/fpm 应用开发工具
- VRTXsa for 68k 的应用开发工具

1.1 VRTXsa X86/fpm 应用开发工具

VRTXsa X86/fpm 有一套完整的、兼容的开发工具，包括:

- MCC x86/fpm 编译器

命令行语法:

MCC386 [-option| source_filename]

option 为控制选项。常用的有以下几项:

控制选项	解 释
-c	产生非执行的目标文件
-D*name*	定义预处理宏
-d*option_file*	指定存放控制选项的文件
-g	产生调试信息
-I*dir*	指定非标准头文件的查找路径
-I@	改变 include 文件的查找路径
-J*dir*	指定标准头文件的查找路径
-l[*filename*]	产生带出错信息的列表文件
-o*filename*	指定输出文件名

Source_filename 为编译器接收文件。这些文件的扩展名为下表所示:

文件	扩展名
C 语言级中间预处理文件	.i
C 语言级文件	.c
汇编级文件	.src / .asm

ELF 可重定位目标文件	.o
OMF386 可重定位目标文件	.obj
目标文件库	.lib
连接命令	.cmd

MCC x86/fpm 编译器根据输入文件的扩展名自动激活相应的工具。

例如：

mcc386 –c –g -lvrtxdemo.lst –ovrtxdemo.o vrtxdemo.c

● ASM x86/fpm 宏汇编器

命令行语法：

ASM386 source_file control_list [%macro_string]

Source_file 必须为第一个参数，且每次只能有一个文件。

Control_list 为控制选项。常用的有以下几项：

控制选项	解 释
[no]ca	[关闭]大小写敏感
[no]db	[关闭]调试信息
[no]oj	[关闭]输出目标文件
[no]pr	[关闭]输出列表文件

%macro_string 最多允许 212 个字符的宏串。

例如：

asm386 myfile.s noca db oj(myfile.o) pr "%set(a,1)"

● LINK x86/fpm 连接器/定位器

命令行语法：

lnk386 [-c*command_file*] [-i] [-m] [-o[**output_file**]] input_object_file [,input_object_file] …

解释：

-c *command* 指明控制文件

-I 指明输出文件是可重定位的，而不是绝对文件。

-m 输出 map 文件。

-o[output_file]指明输出文件名

input_object_file 输入文件名

例如：

lnk386 –c sample.cmd –m mod1.o mod2.o mod3.o >sample.map

其中 sample.cmd **内容为：**

LOAD test1.o
LOAD test2.o
LOAD lib1.lib

● BND x86/fpm **联编器**

命令行语法：

bnd386 input_list control_list

解释：

input_list **可连接的目标文件、库文件列表**

control_list **控制选项列表。常用的控制见下表：**

控制选项	解 释
[no]ca	[关闭]大小写敏感
[no]cf	[关闭]控制文件
[no]db	[关闭]调试信息
[no]lo	生成可装载文件（可连接文件）
[no]oj	[关闭]输出目标文件
[no]pr(filename)	[关闭]输出 mp1 文件

例如：

bnd386 sample.obj, cf(in.ctl) noca db nolo

其中 in.ctl **的内容为** util.lib,system.lib oj(lbt)

● BLD x86/fpm **系统构建器**

命令行语法：

bld386 input_list control_list

解释：

input_list **可连接的目标文件、库、库模板**

control_list **控制选项列表。常用的控制见下表：**

控制选项	解 释
[no]case	[关闭]大小写敏感
[no]cf(filename[,filename])	[关闭]指明控制文件
[no]db	[关闭]调试信息
[no]oj	[关闭]指明输出文件名
[no]pr	[关闭]输出 mp2 文件

例如:

bld386 sample.obj, cf(cntl1.dat) cf(cntl2.dat)

其中 cntl1.dat 内容为 util.lib 和 system.lib。cntl2.dat 内容为 pr(sample.map)和 db。

- LIB x86/fpm 目标模块库

命令行语法：

```
lib386  [-a  filename[,filename] ..]
        [-d  module_name[,module] ..]
        [-e  module_name[,module] ..] [-f {l/s}]
        [-r  filename[,filename] ..] library_name
        [-V]  library_filename
```

解释：

-a filename[,filename] … 指明加入库的模板名。

-d module_name[,module] ..指明要从库中删除的模板名。

-f l 列出库中整个符号表和模块名。

-f s 仅列出模块名。

-r filename[,filename] 取代同名模块。

Library_name 指明生成的库名。

例如

```
lib386 –r ”sym1.o,sym2.o” –a “mod1.o,mod2.o,mod3” –l abc.lib
```

- XRAY x86/fpm Debugger for Windows

详见第二章 XRAY Debugger for Windows。

1.2 VRTXsa for 68k 的应用开发工具

VRTXsa for 68k 有一套完整的、兼容的开发工具，包括：

- MCC68K ANSI 编译器

命令行语法：

```
MCC68K  [-option| source_filename]...
```

option 为控制选项。常用的有以下几项：

控制选项	解　　释
-c	产生非执行的目标文件
-D*name*	定义预处理宏
-d*option_file*	指定存放控制选项的文件
-g	产生调试信息
-H	保存汇编级文件
-I*dir*	指定非标准头文件的查找路径

-I@	改变 include 文件的查找路径
-J*dir*	指定标准头文件的查找路径
-l[*filename*]	产生带出错信息的列表文件
-o*filename*	指定输出文件名
-pprocessor	指定处理器类型
-S	产生汇编级代码
-U	取消预定义的宏

Source_filename 为编译器接收文件。这些文件可以是：

文件	扩展名
C 语言级中间预处理文件	.i
C 语言级文件	.c
汇编级文件	.s .asm .src
可重定位目标文件	.o
目标文件库	.lib
连接命令	.cmd

MCC 68K 编译器根据输入文件的扩展名自动激活相应的工具。

例如：

```
mcc68k  –c –g -lvrtxdemo.lst –ovrtxdemo.o  vrtxdemo.c
```

● ASM68K 宏编译器

命令行语法：

```
ASM68K  control_list  source_file
```

Control_list 为控制选项。常用的有以下几项：

控制选项	解　　释
[no]ca	[关闭]大小写敏感
[no]g	[关闭]调试信息
[no]object_name	[关闭]输出的目标文件
-L	指明列表文件
p=type/cotype	指明目标处理器类型

Source_file 汇编级源文件。必须为命令行最后一个参数，且每次只能有一个文件。一般的扩展名为.S(UNIX)、.SRC(DOS)。

例如：

asm68k –o temp.o -g divd.s

● LINK 68K 连接器/定位器

命令行语法：

lnk68k [-c*command_file*] [-C command][-M] [-m] [-o[**output_file**]][-p]
input_object_file [,input_object_file] …

解释：

-c command_file **指明一个控制文件。**
-C command_file **指明一个连接控制命令。该命令必须放在命令行选项之首。**
-m **输出一个** map **文件。**
-M **指定** map **文件名。**
-o[output_object_file] **指定输出目标文件名。**
-p nnnn **指明处理器类型。**
Intput_object_file **输入目标模板。缺省扩展名为**.O（UNIX）、.OBJ（DOS）。

例如：

lnk68K –c sample.cmd –M –l lib1.lib mod1,mod2,mod3

其中 sample.cmd **内容为：**

LOAD test1.o
LOAD test2.o

● LIB 68K 目标模块库

命令行语法：

lib68k [-a filename[,filename] ..]
[-d module_name[,module] ..]
[-e module_name[,module] ..] [-l}]
[-r filename[,filename] ..] library_name
[-V] library_filename

解释：

-a filename[,filename] … **指明加入库的模板名。**
-d module_name[,module] ..**指明要从库中删除的模板名。**
-l **列出库中整个符号表和模块名。**
-r filename[,filename] **取代同名模块。**
Library_name **指明生成的库名。它必须是命令行的最后一项。**

例如

lib68k –r ”sym1.o,sym2.o” –a “mod1.o,mod2.o,mod3” –l abc.lib

- XRAY Debugger for SPECTRA

详见第二章 XRAY Debugger for Windows。

第二章 XRAY Debugger for Windows

VRTX 应用程序开发工具包提供了包括编译和调试器在内的一整套开发包。XRAY Debugger for Windows 就是 VRTX 提供的源码级调试器。应用程序开发的每一阶段，XRAY Debugger 提供的强大功能，能帮助用户缩短开发时间，加快产品开发的进度。MICROTEC 提供的 XRAY Debugger 有两种类型：

XRAY x86/fpm Debugger 和 XRAY Debugger for SPECTRA。这两种调试器有许多相似之处，但是由于它们所处的系统有较大的差异，所以，各有其特点。总地来说，XRAY Debugger for SPECTRA 功能更强大，因此，相对要复杂一些。
以下说明，XRAY Debugger for Windows 指代两者，对于不同点分别说明。

2.1 XRAY Debugger for Windows 的特性。

- 图形化界面。
- 提供高级语言级、汇编语言级的调试。
- 编辑器

允许在调试阶段对源文件的编辑工作。

- 有多种方式操作调试器 。可以从命令行键入命令，或者通过点击控件，还可以选择菜单。
- 断点

XRAY Debugger for Windows 提供对程序执行的完全控制。通过设置断点 可以控制程序的执行。断点有设置、读/写、访问四种操作。断点还可以和宏结合使用，提供更加强有力的调试手段。

- 宏

XRAY 提供一个强大的宏工具。允许创建宏、向宏传参、执行宏。宏可以单独调用，也可以和断点、用户定制的窗口结合使用，提供有力的调试手段。

- 具有查看复杂数据结构的值和自动转换动态数据结构值的功能。
- 支持多任务
- 头文件

头文件包含有若干调试命令。调试器能自动执头文件。用户可以通过使用头文件重建一个调试部分，也可以减少重复键入相同命令的操作。

- 目标监控器

使用目标监控器不需要对应用程序做任何修改。它是一个独立的模块，可以放在目标地址范围内的任意位置。

- 扩展的调试命令集。
- 在线帮助

2.1.1 XRAY x86/fpm Debugger for Windows

XRAY x86/fpm Debugger for Windows 与 VRTX x86/fpm 应用开发工具 C 编译器、汇编器、binder/builder、linker 配合使用。x86/fpm 环境下的 XRAY 图见图 2-1：

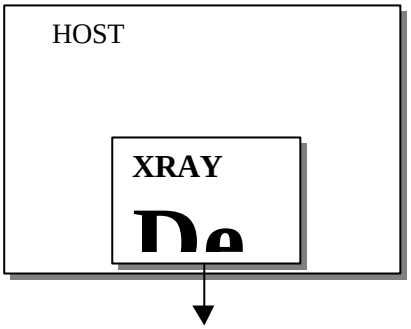


图 2-1 x86/fpm 环境下的 XRAY

■ 启动

启动 XRAY x86/fpm Debugger for Window 有两种方法：

- 双击 Windows 下 XRAY x86/fpm Debugger for Window 的图标。
- 在 DOS 窗口用下列命令行启动：

drive:\vrtxsa86\xhm386\xhm386 [-e string][-I include_filename]
[-j journal_filename][-llog_filename][-ni]
[-s startup_filename][absolute_filename] –Base = <defaultdir>\master

最简单的命令为不带任何参数的 xhm386。

■ 调试命令。

XRAY x86/fpm Debugger for Windows 的调试命令分为四类：

- 程序执行控制命令。如：GO、STEP。
- 调试文件命令。包括工具命令、宏命令、显示命令、在线帮助等。
- 内存控制命令。如：SETMEM（改变内存的值）。
- 符号管理命令。如：ADD（创建一个符号）。

2.1.2 XRAY Debugger for SPECTRA

XRAY Debugger for SPECTRA 与 C 编译器、汇编器、linker、SPECTRA 配合使用。它是一个多窗口的调试器，允许在调试时，针对不同线程开相应的窗口，便于隔离错误，方便了应用程序的调试。在启动 XRAY Debugger for SPECTRA 之前要确保 SPECTRA 已经正确安装；采用以太网连接时，网络服务进程 vserver 已经启动。由于 SPECTRA 允许多个基于主机的工具对同一目标机的调试，所以 XRAY for SPECTRA 能与其它主机上的工具一起共享目标信息。SPECTRA 环境下的 XRAY 图见图二：

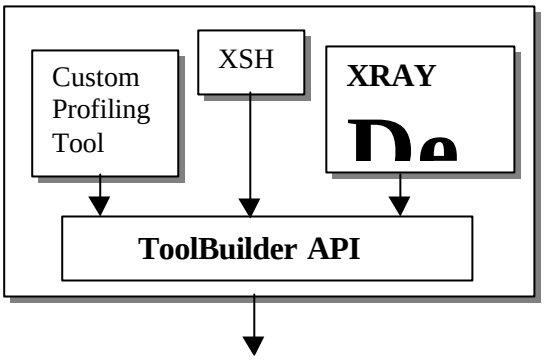


图 2-2 SPECTRA 环境下的 XRAY

■ 启动

启动 XRAY Debugger for SPECTRA 有两种方法：

- 点击 Windows NT 下 Pragrams->SpectracClient->XRAY Debugger 。
- 在 DOS 窗口用下列命令行启动：

drive:\microtec\master\bin\xray [-b][-cmd][-VABS=env][-TABS=tar][-INIT=string]
[-i include_filename][-j journal_filename][-llog_filename]
[-s startup_filename][absolute_filename] –Base = <defaultdir>\master

最简单的命令为不带任何参数的 xray。

■ 调试命令

XRAY Debugger for SPECTRA 支持与 XRAY x86/fpm Debugger for Windows 类似的调试命令。

2.2. XRAY Debugger for Windows 的调试宏。

支持调试宏是 XRAY Debugger for Windows 的一大特色。使用调试宏能使调试器发挥出强大的调试能力。

2.2.1 宏定义

XRAY 的宏是一个类 C 语言的函数。它由一系列表达式、语句和调试命令组成。需要时，宏还可以带参数。用户可以在调试的任何阶段定义和使用宏。

- 宏定义的语法：

DEFINE [macro_type] macro_name ([parameter_list])

[param_definitions]
{
macro_body
}

其中 macro_type **为宏的返回类型，却省为** int。

parameter_list **为参数列表，** param_definitions **为参数类型定义，却省为** int。

macro_body **的语法为**

[local_definitions]
macro_statement;[macro_statement;] …
macro_body **中可以使用调试命令，规则为**$**调试命令**[**调试命令** ..]$;

例如： $printf "demo %d\n",i$;

宏定义的一个例子：

```
define my_macro()
{
 int i;
 for (i=0;i<5;i++)
 $printf "hello,word!"$;
}
```

- **宏定义的注释：**
  使用/**/，**注释可以不止一行，但是不能嵌套。**

- **宏不能使用的调试命令**
  宏定义中能使用大部分的调试命令，除以下命令：
  符号表修改命令 add、define、delete
  执行控制命令 go、gostep、step、stepover、next
  调试部分的命令 host、include、quit

- **预定义的宏**
  调试器识别几个预定义的宏。比如， strcmp、memset **等。用户可以在命令行的表达式中直接使用这些宏或者在用户定义的宏中调用这些宏。**

2.2.2 XRAY 调试宏的生成

XRAY **调试宏有两中生成法：**

- **在** XRAY **外，用** editor **编辑器生成。**
- **在** XRAY **环境下，以** x86/fpm **为例，选择** NoteBook **下** Symbol Management **。从菜单中选择** edit a debugger macro。

2.3 宏的使用

宏的基本使用有：

- **交互式调用**
  这时，宏作为一个命令使用。在 XRAY **的命令行键入宏名** ()。
  注意：宏名要区分大小写。
  使用宏的命令有：break、go、gostep、inport、outport、show。
  如： show my_macro
  显示 my_macro **的内容。**

- **和断点结合使用。**
  宏和断点结合使用，每当程序执行到该断点处时，XRAY **自动执行宏。通过这种方式，可实现复杂的条件断点，在程序继续执行前可以测试指定变量、寄存器的值等等。**

例如定义一个宏 ss：

```
 define ss()
{
 $SETMEM 1000h $;
}
```

在命令行键入 breakinstruction #18;ss()
当程序执行到第 18 **行语句时,自动执行宏** ss()，**允许你对内存地址** 1000h **单元的内容修改。**

- **和硬件模拟器结合使用。**
  在 I/O **硬件不可用时，模拟嵌入式** I/O。

例如定义两个宏进行 I/O **模拟：**

```
 /*simulate hardware sensors.*/
/*open window 50 to trace correct data.*/
vopen 50,1,2,5,12,17
fprintf 50,"Sensor Data\n"
fprintf 50," ……\n"

define int SensorData(Byte)
char Byte;
{
 $ fprintf 50,"%c\n",Byte$;
 return(1); /*silent breakpoint*/
}
```

```
 /* Open viewport 51 to trace incorrect data*/
vopen 51,1,16,5,21,17
fprintf 51,"Sensor Error\n"
fprintf 50," ….……\n"

define int SensorError(Byte1,Byte2)
char Byte1,Byte2;
{
 $fprintf 51,"%c %c\n",Byte1,Byte2$;
 $fprintf 50,"<ERROR>\n";$;
 return(1);/*silent breakpoint*/
}
```

XRAY Debugger for Windows **是一个强大的工具。一旦掌握了基本的调试命令，很容易就能掌握整个命令集、宏和自定义窗口的使用。这些特性使你在调试阶段事半功倍。**

第三章 SPECTRA

3.1 SPECTRA 的概述

SPECTRA **是新一代的、开放式的交叉开发环境，具有真正意义的客户机/服务器结构，为嵌入式实时应用的开发提供了强有力的工具。**（SPECTRA **客户机/服务器的调试模式见图** 3-1。）

调试特点有：

- **采用客户机/服务器的模式。**
- **主机和目标机可通过以太网、串口或者用户提供的方式连接。**
- **支持多用户通过网络对同一目标机的调试。**
- **提供主机上多种工具的协同工作。**

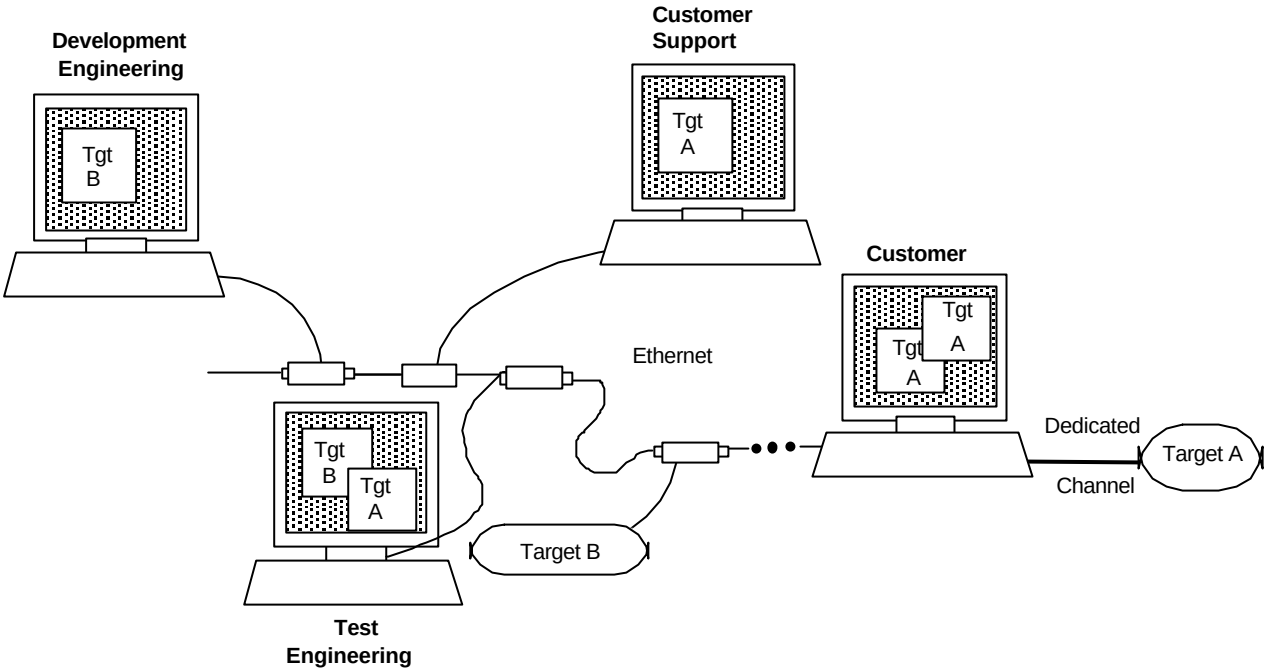


图 3-1 客户机/服务器调试模式

3.2 SPECTRA 的结构

它包括：

- ☐SPECTRA Backplane
- ☐SPECTRA Development Tools
- ☐VRTX Operating System

（SPECTRA 的结构见图 3-2）

■ SPECTRA Backplane

SPECTRA Backplane 实现基于主机上的工具与基于目标机的操作系统间的通信。操作系统可以使用 VRTX 提供的 RTOS，也可以是其它的实时多任务操作系统。Backplane 独立于目标机上使用的操作系统。即使在操作系统停止或者失效时，仍能维持主机和目标机间的连接。SPECTRA Backplane 交叉开发环境包括：

☐ Host Side

Target Manager --- 目标管理器

ToolBuilder（UNIX only）---工具生成器

☐ In Between

Xtrace Protocol --- Xtrace 协议

☐ Target Side

Xtrace Monitor --- Xtrace 监控器

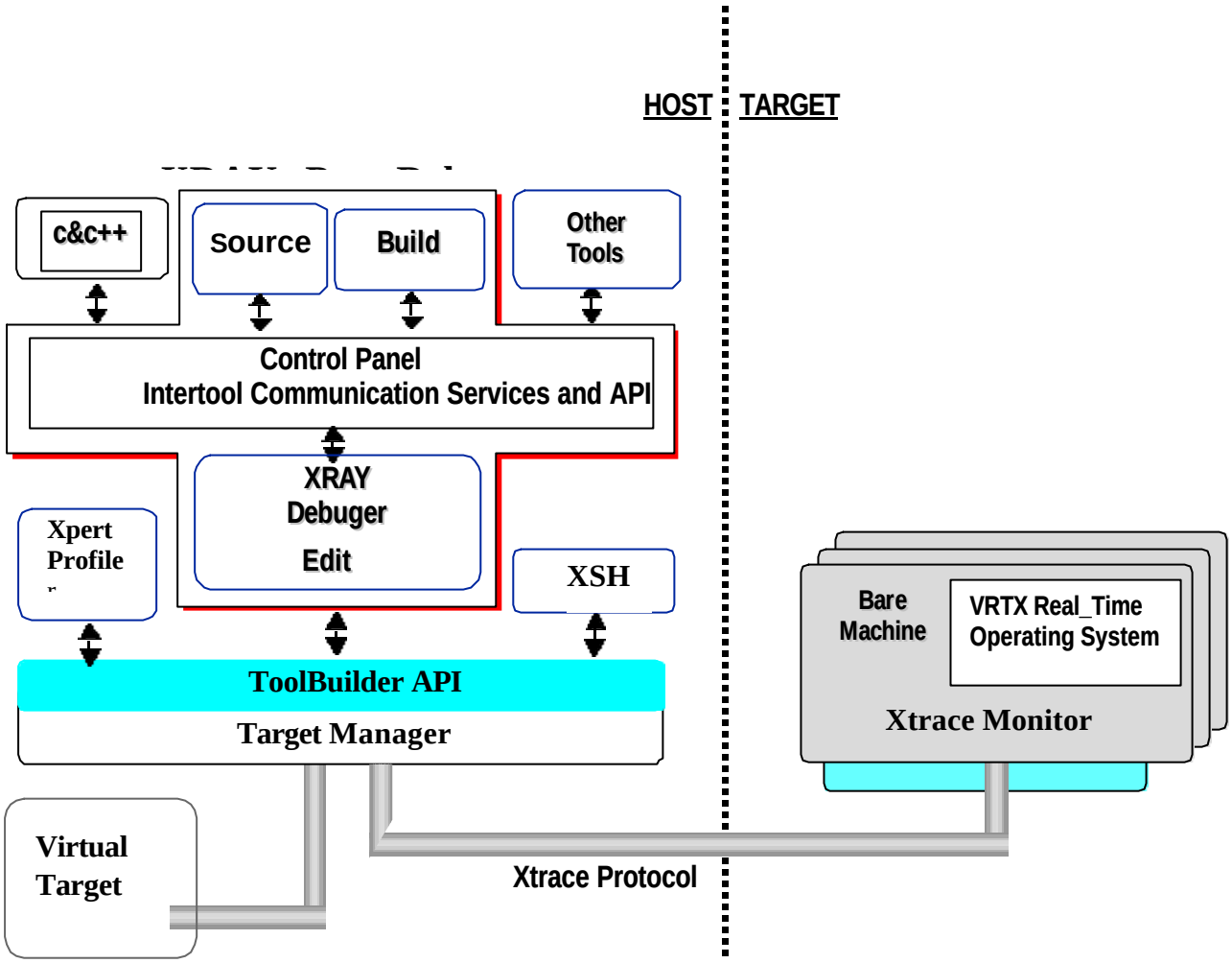


图 3-2 SPECTRA 的结构

■ Xtrace Protocol

Xtrace 协议用于把主机上工具的请求信息送到目标系统。它独立于操作系统、主机和目标机之间的互联方式。如果主机与目标机之间不支持以太网的连接，Xtrace 使用一个驻留在主机上的服务进程提供物理连接的访问。SPECTRA 提供一个基于包的串口通信服务进程 serial_server 提供主机与目标机之间的串口连接。

■ Xtrace Monitor

Xtrace Monitor 是一个驻留在目标机上的模块，完成目标机内的寻径，提供简单的服务支持 Xtrace 协议。Xtrace Monitor 有三种工作模式：

☐ Polled mode：在这种模式下中断被禁止，没有应用程序运行。Xtrace 可直接访问端口，而不需通过中断。

□ Non OS mode：在这种模式下 Xtrace 使用中断与目标管理器进行通信。这时，Xtrace 运行在后台，应用程序运行在前台。Xtrace 不能访问多线程、也无意识内核的存在。
□ OS mode：在这种模式下 Xtrace 作为 OS 的一个线程运行在后台。调试器可以进行指定线程或任务的调试。所以，如果要进行多线程的调试，Xtrace 必须工作在 OS mode 下。
I/O 和中断在不同模式下的情况见图 3-3 ：

	Single-Threaded(Non-OS) Stopped Running	Multi-Threaded(OS) OS OS Stopped Running
I/O Interrupt	Polled Interrupt Disabled Enabled (depends upon application)	Polled Interrupt Disabled Enabled

图 3-3 I/O 和中断在不同模式下的情况

当 Xtrace 工作在 OS mode 时，Xtrace Monitor 的一个拷贝 Xtrace Daemon 用来提供多线程的调试；当 Xtrace 工作 Non OS mode 时，Xtrace Monitor 作为单任务的一部分运行，这时，不支持多线程的调试。

■ Target Manager

Target Manager 是一个驻留在主机上的模块。对于主机上的若干工具来说，它是调试服务器，工具则作为客户端。Target Manager 与工具的关系见图 3-4。Target Manager 是 SPECTRA 客户机/服务器调试模式的核心所在。Target Manager 与工具可以在不同主机上，但 Target Manager 与 Xtrace Monitor 必须在同一网络上，SPECTRA 不支持网关。

Target Manager 负责与目标的所有连接，并且接管了以前由目标机完成的调试工作，给目标机中的应用提供更充足的空间。Target Manager 在第一个工具请求和目标机建立连接时产生，在最后一个工具与目标机断开连接时撤消。

Target Manager 提供的服务有：
□ 维护供主机工具使用的目标机可用内存印象
□ 维护
□ 维护
□ 向主机上
□ 管理断点。

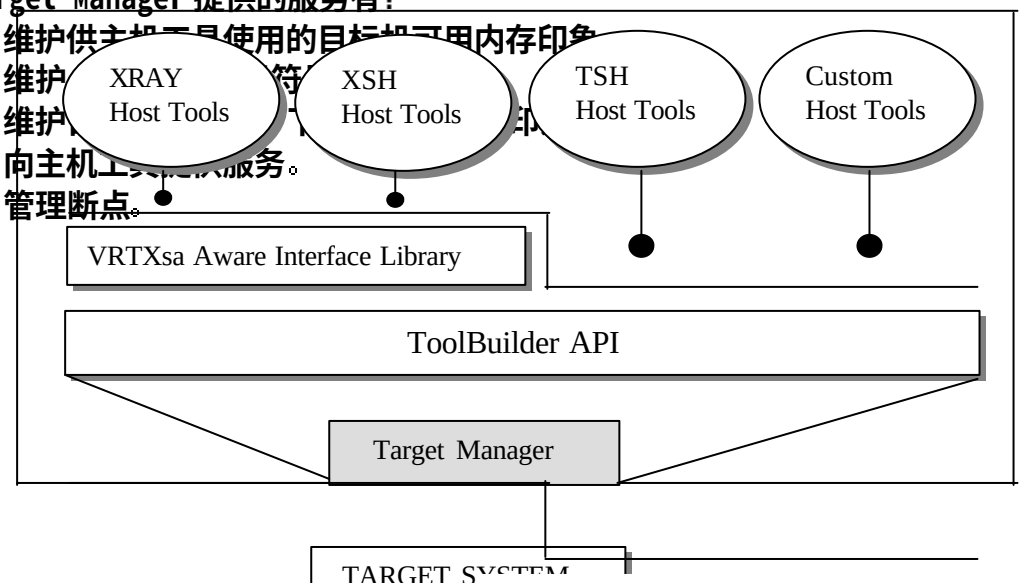


图 3-4 Target Manager

■ ToolBuilder:应用程序的接口（API）

ToolBuilder 是一个开放的、可编程的函数调用接口，允许多个工具访问 Target Manager 和 Xtrace 服务。它由多个库文件组成。用户可以通过使用这些库文件，开发出自己的针对特定环境的开发工具。

■ Communication in the Spectra Backplane

SPECTRA Backplane 通信组键有：

- ☐ sdemon(UNIX only)
- ☐ vserver
- ☐ Target Manager
- ☐ Connection Server
- ☐ Xtrace Protocol
- ☐ Xtrace Monitor/Daemon

对于 Xtrace Protocol、Xtrace Monitor/Daemon、Target Manager 前面已经说明，下面仅对 sdemon、vserver、Connection Server 加以说明。

■ sdemon(UNIX only)

sdemon 仅针对 UNIX，它是运行 SPECTRA Backplane 的一个基本进程。每一个指定的目标机上，只能有一个 sdemon 进程。

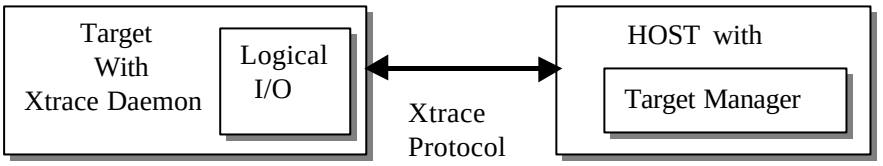
■ vserver

vserver 用于维护主机工具与目标机之间的连接信息。Targert Manager 由 Vserver 管理。在 UNIX 下，vserver 由 sdemon 创建，在 NT 下，vserver 只能由用户手工创建。

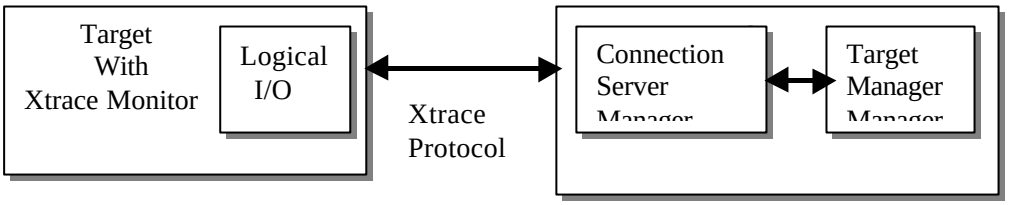
■ Connection Server

Connection Server 是一个用于把包类型转换成以太网包类型的进程。一个特殊的 Connection Server 是 serial_server，用于把串口包转换成以太网包。SPECTRA 只提供了这一种特殊的 Connection Server。如果用户要使用其它方式，需要自己编写 Connection Server 代码。

图 3-5 说明了通过以太网和串口两种通信方式的示意图：



以太网通信方式



串口通信方式

图 3-5 以太网和串口通信方式

■ SPECTRA Development Tools

Microtec 提供了下列基于主机的工具：

- ☐ XSH
- ☐ XRAY debugger
- ☐ Virtul Target(SunOS only)
- ☐ Xconfig
- ☐ Xpert Profiler
- ☐ C++ and ANSI C compilers

■ XSH

XSH 是驻留在主机上的命令解释器，包含一个嵌入式命令集。这些命令提供了主机与目标机之间一个交叉开发接口。

XSH 提供下列类型的命令：

- ☐ 主机与目标机之间的通信
- ☐ 目标代码的下载
- ☐ 符号表的管理
- ☐ 系统级的调试
- ☐ 目标机的监控
- ☐ 目标机内存的管理

XSH 命令集具有可扩充性，允许用户添加自己的 XSH 命令。

XSH 的命令行语法：

xsh [-f] [-t<target>] [-s][-c<command>|<script>]

其中：

-f ：快速启动。
-t<target> ： 与指定目标系统建立连接。
-s ：执行标准输入中的命令。
-c<command>：指定的命令串。
script：执行指定文件中的命令。

XSH 命令解释器的操作图见图 3-6：

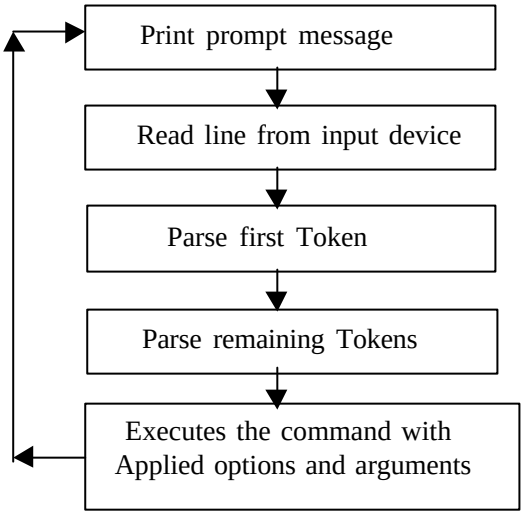


图 3-6 XSH 操作流程图

XSH 有两种工作模式：Non OS mode 和 OS mode。XSH 处于哪种工作模式由 Xtrace 决定。

■ XRAY Debugger for SPECTRA

XRAY Debugger for SPECTRA 提供了一个实时的、C、C++和汇编级的调试工具。由于 XRAY 具有对程序执行流程的完全控制权，所以用户可以方便地隔离错误，加快开发进度。XRAY 的一个突出的特色是支持调试宏。用户可以把自定义的宏与断点结合使用，形成条件断点，提供更有力、更复杂的调试手段。

■ Virtual Target(SunOS only)

Virtual Target 是在主机上提供的一个物理目标的抽象。一般，它包括有 Xtrace Monitor 以及目标环境应有的其它关键组键。主机可以是 SPECTRA 支持的任何基于 SunOS 的计算机。（Virtual Target 见图 3-7。）

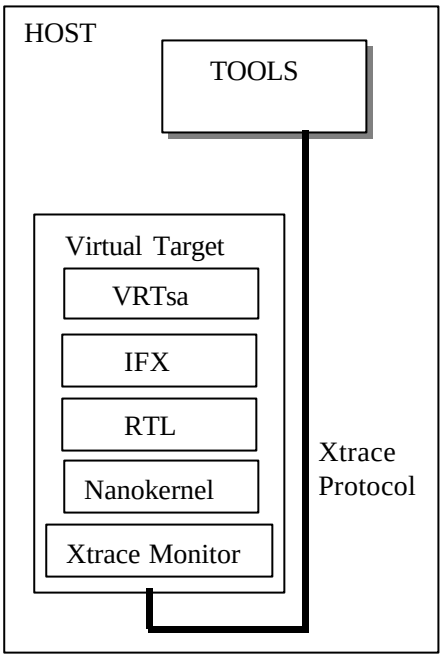


图 3-7 Virtual Target

■ XCONFIG

XCONFIG 是一个用于对目标机上的软件进行配置的工具。它需要一套开发环境所需的配置信息。使用 Microtec 提供的却省配置文件可以配置：

- ☐ VRTX 实时操作系统
- ☐ 目标板的 boot image
- ☐ Virtual Target

■ Xpert Profiler

对于嵌入式软件开发者来说，在各个阶段都应把握软件的性能特点。

Xpert Profiler 是一个面向嵌入式软件开发工程师的性能测量和评估的工具。在软件开发周期的任何阶段，都可以用 Xpert Profiler 检测软件的性能。

■ C++ and ANSI C Compilers

Microtec 提供了符合 ANSI 标准的 C 编译器和 C++编译器。

■ VRTX Real_Time Operating System

VRTX Real_Time Operatine System 为应用提供多任务服务、网络服务、I/O 文件管理服务，以及 ANSI C 的子运行库 RTL 和面向对象的实时运行库 OORTL。VRTX 操作系统包括以下的几个组键。VRTX 具有可裁剪性，可根据应用环境的实际需要来配置。

- ☐ VRTX32（M68XXX），VRTXsa,or VRTXmc kernel

☐ ESH
☐ IFX
☐ SNX
☐ RTL
☐ OORTL

VRTX 的接口见图 3-8：

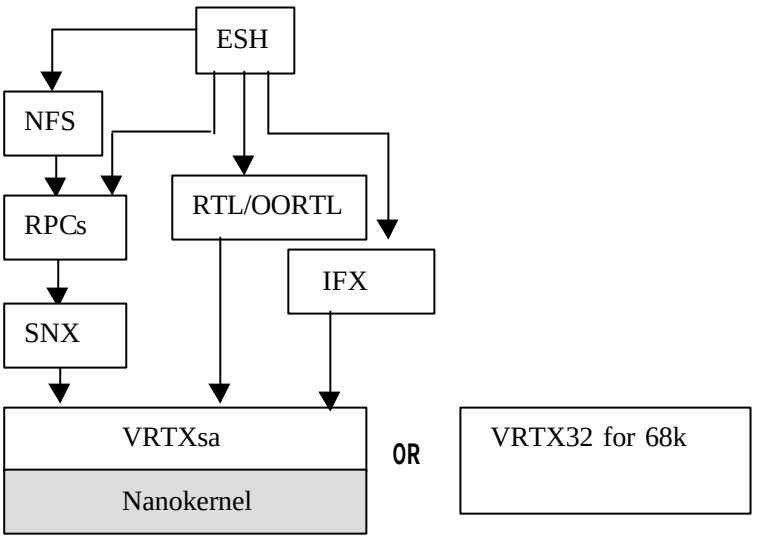


图 3-8 VRTX 接口

■ VRTX Kernels

VRTX 操作系统提供三种实时内核：
VRTX32
VRTXsa
VRTXmc

■ ESH:Run_Time Embedded Shell

ESH 提供了用户与 VRTX 操作系统之间的接口。ESH 包括若干个嵌入式命令。用户可以在 ESH 中加入自己定义的命令。

■ IFX：I/O and File Management

IFX 支持：

☐ 实时多任务环境下，与设备无关的 I/O 和文件的管理。
☐ 设备驱动程序的动态安装。
☐ 支持随机、串行访问设备、文件、物理 I/O 和 I/O 控制函数。
☐ 支持同步、异步串口。

■ SNX Networking：STREAM,TCP/IP,and SNMP

SNX 提供与 UNIX SVR3/4 完全兼容的流机制。流环境为用户编写自己的通信设备驱动程序提供标准的框架。SNX 支持 SNMP，允许用户在以太网的连接方式下使用 TCP/IP、BSD 4.3 socket、TLI 接口或者串行口连接方式下的 SLIP、RS-232。

■ RTL：ANSI-Compliant Run-Time Library

RTL 提供可重入的 C 语言库函数。这些函数包括字符和文件 I/O、字符串操作、存储分配、文件管理和数学函数。

■ OORTL:Object-Oriented Run-Time Library

OORTL 是一个 C++语言类库，提供一个面向对象的编程接口。

应用开发篇二：BSP、monitor 及固化

第一章 BSP

1.1 BSP 在系统体系结构中的位置

BSP（Board Support Package）即“板级支持软件包”，它是操作系统和目标硬件环境的中间接口，如图 1-1 所示。

BSP 通常包括以下部分：
【1】 描述操作系统所需的系统环境；
【2】 初始化代码；
【3】 设备中断服务例程（ISRs）；
【4】 用户提供的与设备有关的操作例程。

1.2 Spectra BSP

1.2.1 概述

Spectra 是 Microtect 提供的一种独特的调试体系结构，它改变了传统的交叉开发环境下主机和目标机一一对应的调试体系结构，允许多个开发者在不同的调试主机上使用不同的基于 Spectra 的调试工具对同一物理目标同时进行分布调试。为了支持 Spectra 这种多对一的调试体系结构，Spectra 包括关键的五个部件：
a * Xtrace 协议
它是驻留在主机上的目标管理器（Target Manager）和驻留在目标上的 Xtrace

进程通信协议。

b • Xtrace 监控器

它驻留在目标上，完成对物理目标的具体操作。

c • 目标管理器（Target Manager）

它驻留在主机上，处理在目标上进行的调试活动。对于主机上的多个调试工具（client tools）来说它是调试服务器（debug server）。

d • logio（λογιχαλ ινπυτ/ουτπυτ）接口

logio 接口层屏蔽了底层的物理特性，使上层的 Xtrace 协议和底层的物理连接无关，同时也支持操作系统的可移植性。

基于 Spectra 的系统体系结构如图 1-2 所示：

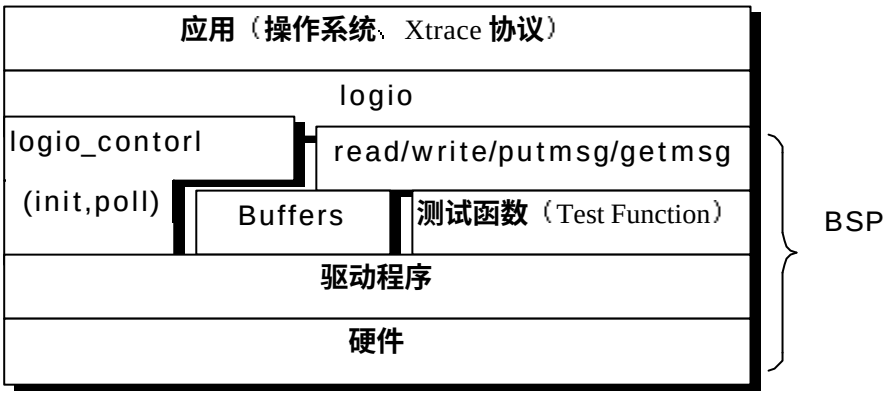


图 1-2　系统结构图

1.2.2 BSP 中的 logio 实现

基于 Spectra 的 logio 是一个接口标准，它提供了存取 I/O 设备的抽象接口。

1．logio 接口

logio 使用 fops(function operations)表实现对设备间接操作的调用接口。fops 表的数据结构如下：

```
typedef   struct {
                logio_status_t    (*init)  ();
               logio_status_t    (*read)  (logio_device_id_t,char  *,  int  *  );
               logio_status_t    (*write)  (logio_device_id_t ,char  *,  int  *);
               logio_status_t    (*control)  (logio_device_id_t,  void  *  );
               logio_status_t    (*getmsg)  (logio_device_id_t  ,  logio_buff_t  **);
               logio_status_t    (*putmsg)  (logio_device_id_t,  void*  );
               logio_status_t    (*poll)  (logio_device_id_t,  void*  );
               }  logio_fops_t;
```

从上面的数据结构可以看出，logio 提供给应用七个标准的函数接口来对设备进行操作。对不同的设备，这七个函数的实现是不同的。由于采用 logio 接口实现了 Xtrace 协议和操作系统的设备无关性。logio 和系统之间的调用关系如图 1-3 所示：

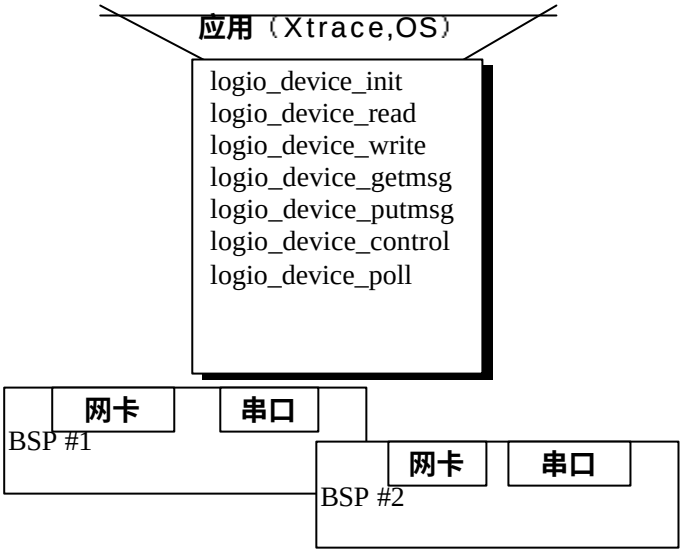


图 1-3 logio 和系统之间的调用关系

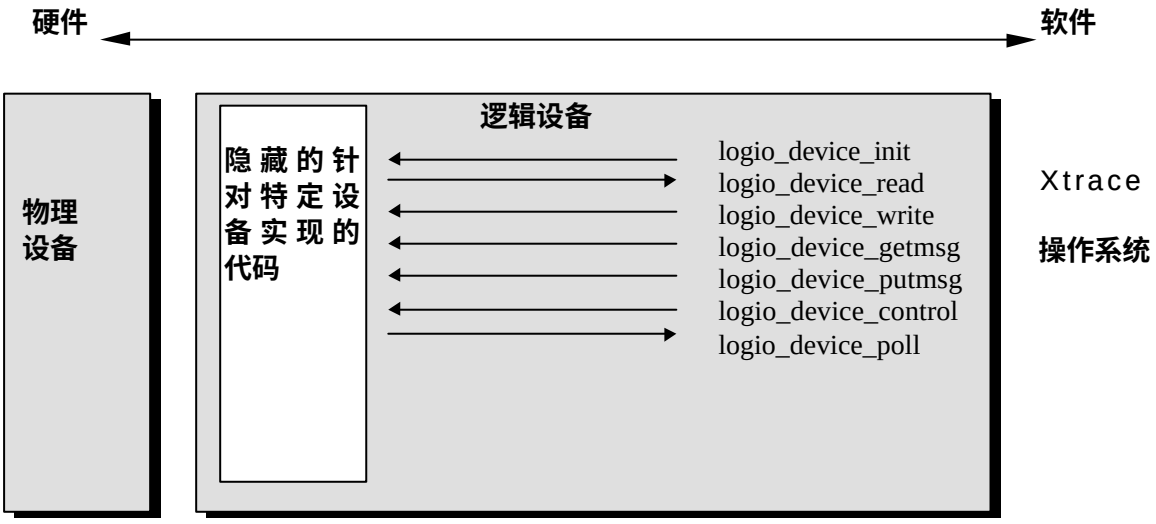
上图中，每一个目标板的网卡和串口类型可能是不同的，但应用对它们的操作接口是相同的。

2. 逻辑设备

在 spectra 中，物理设备被抽象为两级逻辑设备（logio 设备）。低级的逻辑设备如 ETHER_N（N 代表板上网卡编号），SERIAL_N,TIMER_N 等。高级的逻辑设备如 BRIDGE，VT_TIMER,VCONSOLE 等。之所以被抽象为两级设备的原因就是某种高级设备可能被映射到板上的任意某个设备。低级设备依次又被映射为某种物理设备。比如，作为主机上的目标管理器（task manager）和目标上的 Xtrace 监控器之间的连接 BRIDGE，它可能为串行连接，也可能为网络连接。如果使用串行连接，串口可以配置为板上串口中的任意一个。所以，物理设备的两级抽象方便了设备的配置。物理设备两级抽象的调用关系如下图 1-4 所示：

图 1-4 物理设备的两级抽象

由于物理设备被抽象为逻辑设备，应用不直接存取物理设备，而是存取一个抽象的 I/O 对象。因此，使用设备描述符和逻辑设备相对应。对于低级的物理设备，设备描述符包含了关于某个设备的相关数据，包括配置和对这个设备特定的操作。通过设备描述符唯一标示和某个逻辑设备相对应的物理设备。物理设备和逻辑设备的关系如图 1-5 所示：



通过定义 3 种数据结构类型可以完整地描述一个逻辑设备。这 3 种数据结构类型为：

◆配置

它保存设备的特定数据。

```
typedef    struct{
             /*特定数据*/
           }  device_config_t;
```

◆设备函数表（dev_fops）

它保存针对特定设备实现的函数的指针。这里的函数表不同于前面所说的函数表（logio 函数表）。

```
typedef    struct{
           /*函数指针*/
           }device_fops_t;
```

◆设备描述符

它包括配置和函数表。

```
 typedef    struct{
            device_config_t    config;
            device_fops_t       fops;
            /*其他数据*/
              }dev_desc_t;
```

对同一类设备，其配置、函数表和设备描述符的数据结构相同。其他数据包括了外部 I/O 函数等数据结构。这些数据结构不同种类的设备各不相同。对于不能被 logio_fops 映射执行的功能函数（即 dev_fops 函数表以外的函数），被放入外部 I/O（External I/O）函数表中。如读写端口寄存器。

3．logio 接口和逻辑设备的关联

spectra 使用了一种称之为 logio_method 的数据结构，它实现了抽象的 logio 层和特定设备代码（即设备描述符）之间的联系。其数据结构如下：

```
typedef    struct    logio_method{
                      logio_fops_t *    logio_fops;
                      (void*)    &dev_desc;
                     };
dev_desc_t      dev_desc;
```

采用 logio_method 可以允许不同的设备使用相同的接口代码，即使用相同的 logio_fops 函数表。使用方式如图 1-6 所示：

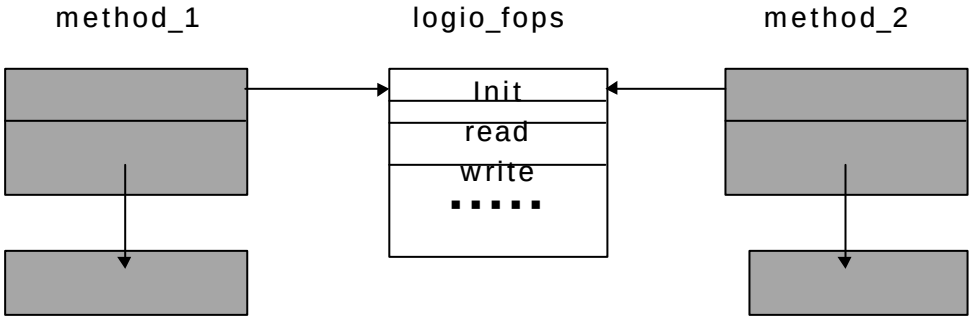


图 1-6 logio 接口使用方式

逻辑设备通过设备 ID 标示。当通过 logio_fops 函数表存取逻辑设备时，设备 ID 决定了实际被存取的物理设备。设备 ID 实际上是指向 logio_method 的指针。

```
logio_method *    logio_device_id ;
```

当通过 logio_method 中的 logio_fops 函数表调用接口函数时，实际将设备 ID 传递给了 logio_method。然后，设备 ID 通过 logio_method 中的设备描述符调用针对特定设备实现的函数，达到存取设备的目的。

4．logio 中断处理

物理设备产生的中断被抽象为逻辑 I/O 中断，它对应于虚拟中断向量表（Virtual Interrupt Table）。

在许多情况下，每一个中断向量对应不同的中断源,及对应不同的事件。比如，对应串口所产生的中断，它可能的中断源包括接收字符、发送字符及出错。这种情况下，需要为逻辑设备的每一个中断源写相应的中断处理程序。中断处理程序的指针放在中断向量事件表（Interrupt Vector Event Table）的表项中。每一个中断向量对应一个中断向量事件表。

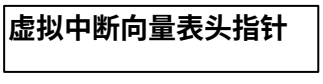
虚拟中断向量表数据结构如下：

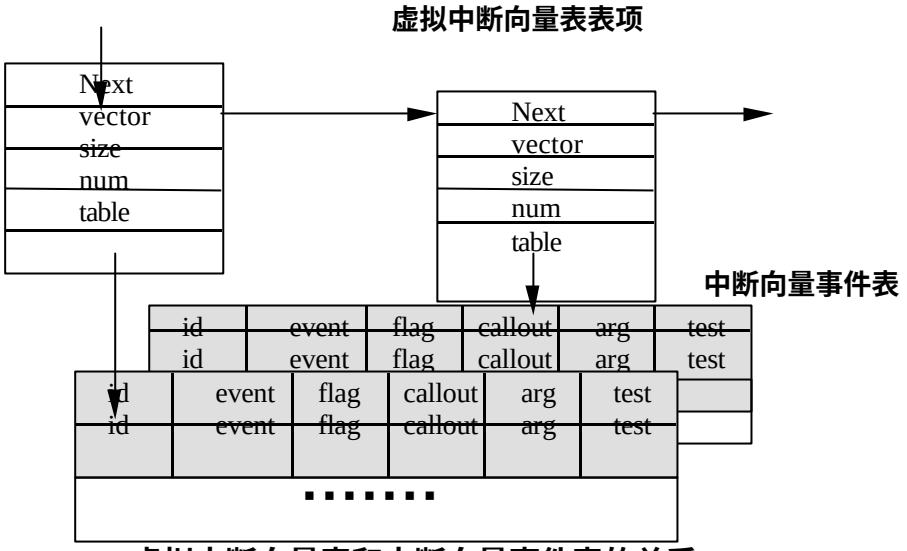
```
typedef    struct    logio_int_table_t    {
              struct    logio_int_table_t  *      next;
               int        vector;      /*中断向量*/
               int        size;        /*整个虚拟中断向量表的表项数*/
               int        num;         /*中断向量事件表的表项数*/
               logio_interrupt_entry_t    *    table;      /*指向事件表的指针*/
        }  logio_int_table_t;
```

中断向量事件表的数据结构如下:

```
typedef    struct    logio_interrupt_entry  {
               logio_device_id_t         id;            /*设备 ID*/
               logio_int_event_t         event;         /*标示中断源*/
               int                         flags;    /*保留*/
               int      (*callout)  ();                 /*中断处理程序 ISR*/
               void    *  arg;                         /*作为参数传递给 ISR 的指针*/
               int      (*test)  ();                   /*测试事件是否发生的函数的指针*/
          }  logio_interrupt_entry_t;
```

虚拟中断向量表和中断向量事件表的关系如下图 1-7 所示：





1-7 虚拟中断向量表和中断向量事件表的关系

在调试环境下，Xtrace 接管所有的中断。当某个硬件引发中断后，首先执行 Xtrace 的中断预处理程序，由它调用缺省的 logio 中断处理程序，引发逻辑 I/O 中断。logio 中断处理程序找到对应于这个硬件中断向量的虚拟中断向量表表项，执行由 table 指向的中断向量事件表中的 test 测试函数。通过测试函数的返回值判断该事件是否发生。如果测试函数返回值为假则执行下一个事件的测试函数。如果函数返回值为真，则执行和该 test 函数对应的 callout 函数，即真正的 ISR。在应用环境下，当中断发生后直接执行缺省的 logio 中断处理程序。

1.2.3.基于 Spectra 的 BSP 分析

不难看出，基于 Spectra 的 BSP 开发采用了面向对象的思想。所谓对象，最广泛的解释是将某一数据和使用该数据的一组基本操作或过程封装在一起，而将此封装体看作一个实体。“面向对象”的思想克服了软件的复杂性，同时将现实世界模型在计算机中自然地表示出来。基于 Spectra 的 BSP 采用了类似于类的数据结构 method，它封装设备的“私有数据”，同时向上提供一组通用调用接口来对设备进行存取操作。正是由于采用面向对象的思想，使基于 Spectra 的 BSP 具有了以下一些特点：

1．可靠的二进制 logio 接口代码文件，可适用于不同目标扳的 BSP；

2．可配置的设备驱动程序，不依赖于地址、I/O 端口和内存映射；

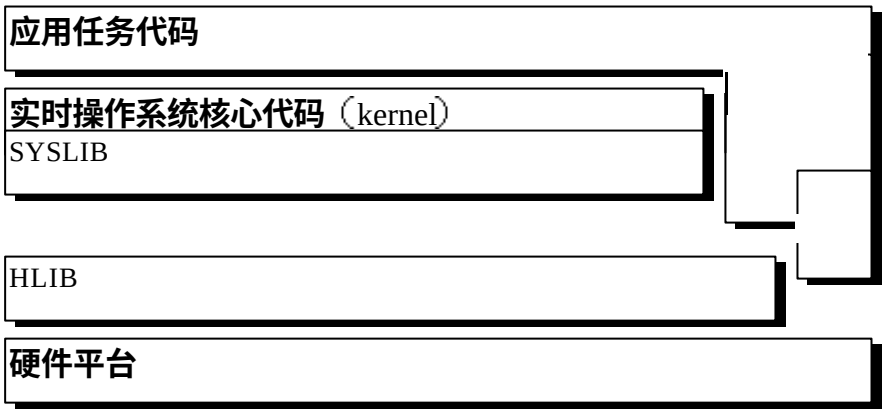
3．支持使用具有外部接口的设备驱动程序；

4． 同一类设备的设备驱动程序可以使用相同数据结构和流程，具有同样的调用接口（包括外部接口）。这样开发者根据摸板文件和测试代码可以使新设备驱动程序的

开发更容易和更快。

1.3 VRTX x86/pm BSP

VRTX x86/pm(下简称 spm)的 BSP 结构较 SPECTRA 简单，它主要由 HLIB 和 SYSLIB 组成，使用 namke 将 HLIB 和 SYSLIB 同 appliction 链接在一起，形成可执行程序。整个体系结构如图 1-8 所示：



1.3.1 HLIB

HLIB 是依赖于硬件的库，随硬件的不同而不同。随 spm 软件包提供了对嵌入式 PC、i386EX 和 NS486SXF 系统的 BSP 支持。HLIB 由下列部分组成：

- HLIB 例程
- 对 SYSLIB 的调用
- 和目标板和设备相关的中断服务例程

HLIB 支持 spm 所需的基本外围设备，包括：

- 时钟
- 串行 I/O
- 板初始化和启动代码

HLIB 提供了一套和板及设备相关的例程被 SYSLIB 调用。如初始化板、输出一个字符到端口等。在 HLIB 的中断处理程序中，也可以调用 SYSLIB 例程。HLIB 提供的例程如表 1-1 所示：

目标相关例程：

hwl_initialize_board	初始化目标板
hwl_initialize_timer	初始化系统 Tick 时钟
hwl_reset_board	目标板 RESET 例程

端口相关例程：

hwl_com1_init	初始化串口 1
hwl_com1_pgets	轮询从串口 1 获得一字符

表 1-1 HLIB 例程

1.3.2 SYSLIB

SYSLIB 是独立于外围硬件的库，它通过 HLIB 提供系统部件、设备初始化、端口和各种通用功能支持。大多数对 SYSLIB 的调用对于应用来说是透明的，它们是否被调用由配置所决定。SYSLIB 提供的例程如表 1-2 所示：

表 1-2

部件支持：	
sys_vrtx_init	初始化 VRTX X86/spm
sys_ifx_init	初始化 IFX
sys_rtl_init	初始化 RTL Hooks
sys_malloc_init	初始化内存分区
sys_tnx_init	初始化 TNX
sys_wix_init	初始化 WIX
sys_init_small	初始化小内存模式 Hooks
设备初始化：	
sys_timer_init	初始化 VRTX x86/spm 系统时钟
sys_init_vrtxio	初始化 VRTX x86/spm 字符 I/O
sys_init_wixio	初始化 WIX I/O
端口支持：	
sys_com1_isr	面向 SYSLIB 的串口 1ISR（接收或发送）
sys_com2_isr	面向 SYSLIB 的串口 2ISR（接收或发送）
通用功能：	

SYSLIB 规定了初始化例程的调用顺序，错误的调用顺序可能引起目标机挂起。调用顺序如下：

sys_rtl_init
sys_vrtx_init
sys_init_vrxio
sys_malloc_init
sys_wix_init
sys_timer_init
sys_ifx_init

1.4 VRTX x86/rm BSP

VRTX x86/rm（以下简称 RM）对设备驱动程序的调用方式分为通过 IFX（I/O and File Exective） 和在应用程序中直接调用两种方式。

通过 IFX 的方式使用设备驱动程序之前需要使用以下三个 IFX 函数为使用做准备：

- ifx_driver( )
- ifx_install( )
- ifx_mount( )

对 IFX 的逻辑结构可以参见《IFX/X86 用户手册》和《IFX x86 设备驱动程序开发》。

1.5 BSP 创建

创建 BSP 一般包括几种方式：

- 利用现成的 BSP 即随软件包安装的 BSP
- 在现成的驱动程序基础上修改使之满足调用格式，从而形成新的 BSP
- 开发新的驱动程序形成新的 BSP

利用现成的 BSP 一般和系统的配置有比较紧密的关系，有关配置的情况将

在有关的章节详述。而修改驱动程序相当于开发新的驱动程序的一部分工作，因此，这一部分主要介绍开发新的驱动程序，即增加现成的 BSP 所不支持的新的设备驱动程序的过程。

1.5.1 VRTXsa x86/fpm BSP 开发

VRTXsa x86/fpm 和 spectra 一样，也有高级逻辑设备到低级逻辑设备再到物理设备的映射关系，只不过它称之为系统设备映射到逻辑设备再映射到物理设备。为了增加对新设备的支持，需要按下列步骤进行：

1.在文件 vrtxcnfg.def 中定义系统设备名以及映射到的逻辑设备名：

sys.env.dev.*device*.envname xxxxx # 定义的系统设备名

sy.env.dev.*device*.value xxxxx #定义的逻辑设备名，*device* 是系统设备名

2.在文件<board>.def 中定义物理设备名并且定义和新设备所对应的 logio method 结构名：

dev.*device*.name xxxxx # 定义的物理设备名

dev.*device*.value logio_xxxxx_method # 定义 logio method 结构名

3.在文件<project>.def 中将新设备增加到系统设备列表和逻辑设备列表：

sys.env.devices xxxx, yyyy, zzzz

board.devices xxxx, yyyy, zzzz

4.在文件 devcnfcxx.c 中建立新设备的 logio method 结构

5.按调用格式为新设备编写相应的驱动程序

6.编辑 makefile 文件，使用 dmake 生成和 BSP 对应的库文件<board>.lib

1.5.2 VRTX x86/spm BSP 开发

如前面所述，VRTX x86/spm 的 BSP 由库文件 SYSLIB 和 HLIB 组成。从前面 VRTXx86/spm 的体系结构图可以看出，应用可以直接调用 HLIB 中的驱动程序。因此，可以将新设备的驱动程序并入 HLIB 中。这样带来的缺点的是应用程序的可移植性差，当硬件平台改变时，不得不重写应用程序。SYSLIB 可以通过配置工具 config 调整参数，所以 SYSLIB 几乎不需要修改。

生成新的 HLIB 库的过程主要包括两个方面：编写驱动程序和编辑 makefile 文件。然后，使用命令:

nmake -f <board>.mak

在使用 config.bat 配置生成*.abs 的过程中，会调用上述命令，因此，可以不单独使用上面的命令。

1.5.3 Spectra BSP 开发

Spectra 由于独特的开发调式环境，它的 BSP 开发采用了增量式的开发方式，

即开发一个驱动程序后随即测试其功能，然后再开发另一个驱动程序。并且，开发的驱动程序有顺序要求，即必须先开发目标和主机之间的 BRIDGE 驱动程序，以便使用主机的开发工具诸如 XSH、XRAY 等，然后才能开发其他设备的驱动程序。由于 Spectra 随软件包提供了许多常用的目标板和设备的驱动程序模板，使开发过程大为简单。例如，在目录$SPECTRA/target/xsp/下就有许多关于目标板的配置模板，在$SPECTRA/target/device/下有许多关于常用外围设备的驱动程序。这样，开发 BSP 仅仅需要做的就是修改配置参数，使之满足目标应用环境的需要。

需要说明的是，从狭义上讲，BSP 仅仅包含了外围设备的驱动程序，即 BSP 库。从广义上讲，BSP 的功能是对目标环境进行初始化并向上层提供对外围设备的的存取，所以 BSP 还包括了和处理器相关的代码以及各种和驱动程序相关的数据结构。因此，本小节还将涉及到将 BSP 生成可引导的.HEX 文件的过程。

下面给出 BSP 库的开发步骤，并且以目标板使用 Motorola CPU 和 Z8530 串口设备为例列出需要的文件，其他设备依次类推添加。

1.创建一个工作目录，拷贝如下表所示的文件到当前工作目录；

文件	说明
$SPECTRA/target/device/zi8530/common/zi8530.c	设备驱动程序文件
$SPECTRA/target/device/zi8530/doc/zi8530.txt	帮助文件
$SPECTRA/target/device/zi8530/include/zi8530.h	头文件
$SPECTRA/target/xsp/forms/68xxx/makefile	建立 BSP 库的 makefile
$SPECTRA/target/xsp/forms/board.c	配置板设备的文件
$SPECTRA/target/xsp/forms/68xxx/lnk.cmd	Link 命令文件
$SPECTRA/target/xsp/forms/board/devconfg.c	配置串口、网卡和时钟等的模板文件
$SPECTRA/target/xsp/forms/68xxx/crt0.s	板初始化的开工文件

2. 件样文件；
3.按照目标环境的需要修改 makefile 文件；
4.按照配置参数修改 devcnfg.c，其内容可以参考文件 $SPECTRA/target/xsp/*borad*/common/deccnfg.c；
5.在当前目录下使用命令 nmake finallib 创建<board>.lib;这里需要说明的是，当需要对所开发的串口驱动程序进行功能测试时，使用命令 nmake pkt_poll.hex 生成 pkt_poll.hex，另外需要将测试程序如 $SPECTRA/target/xsp/forms/serial/pkt_poll 拷贝到当前目录。具体的测试过程随目标硬件环境的不同而不同，详细情况可以参考有关手册。
6.为了最终生成一个可引导的.HEX 文件，将

$SPECTRA/target/xsp/forms/68xxx/board.def 拷贝到当前目录下，将 board.def 改名为<board>.def，并修改其中的内容。<board>.def 中有关固化的内容将在有关 Spectra 固化的章节介绍。

7.使用命令 xconfig <board>.def 生成可引导的.HEX 文件。

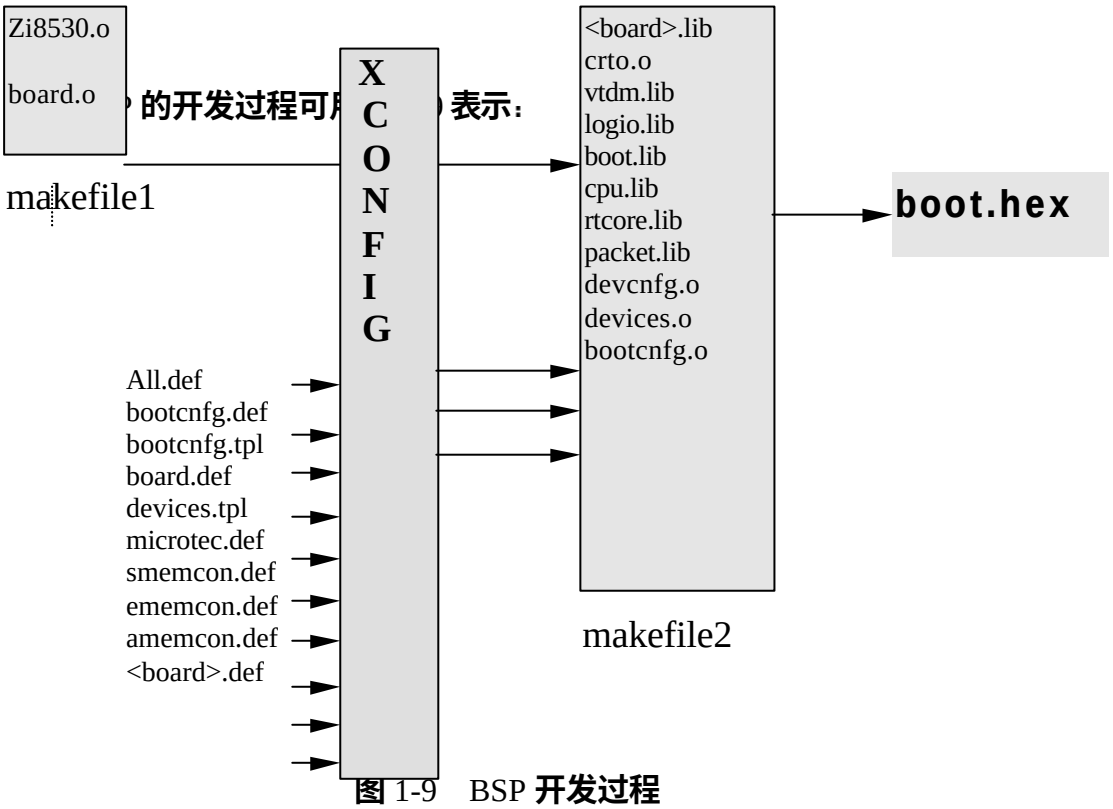


图 1-9 BSP 开发过程

1.5.4 VRTX x86/rm BSP 开发

在 PC 的环境下开发自己的设备驱动程序应该使用 BIOS 提供的服务来存取硬件，这样可以使硬件驱动程序具有可移植性。一般来说，RM 的设备驱动程序包括以下三种类型：

a.同步方式工作的设备驱动程序，它们通过应用直接调用，或通过内核，或通过

IFX。例如 PC 的磁盘驱动程序 dsp_disk_device( )；

b.由 ISR 构成的驱动程序。例如 PC 下的时钟和键盘的驱动程序；

c.异步方式工作的设备驱动程序，由 ISR 和 I/O 初始化例程构成的驱动程序。ISR 和 I/O 初始化例程通过邮箱来同步。例如 PC 下的 COM1/COM2 的驱动程序和异步方式的磁盘读写。

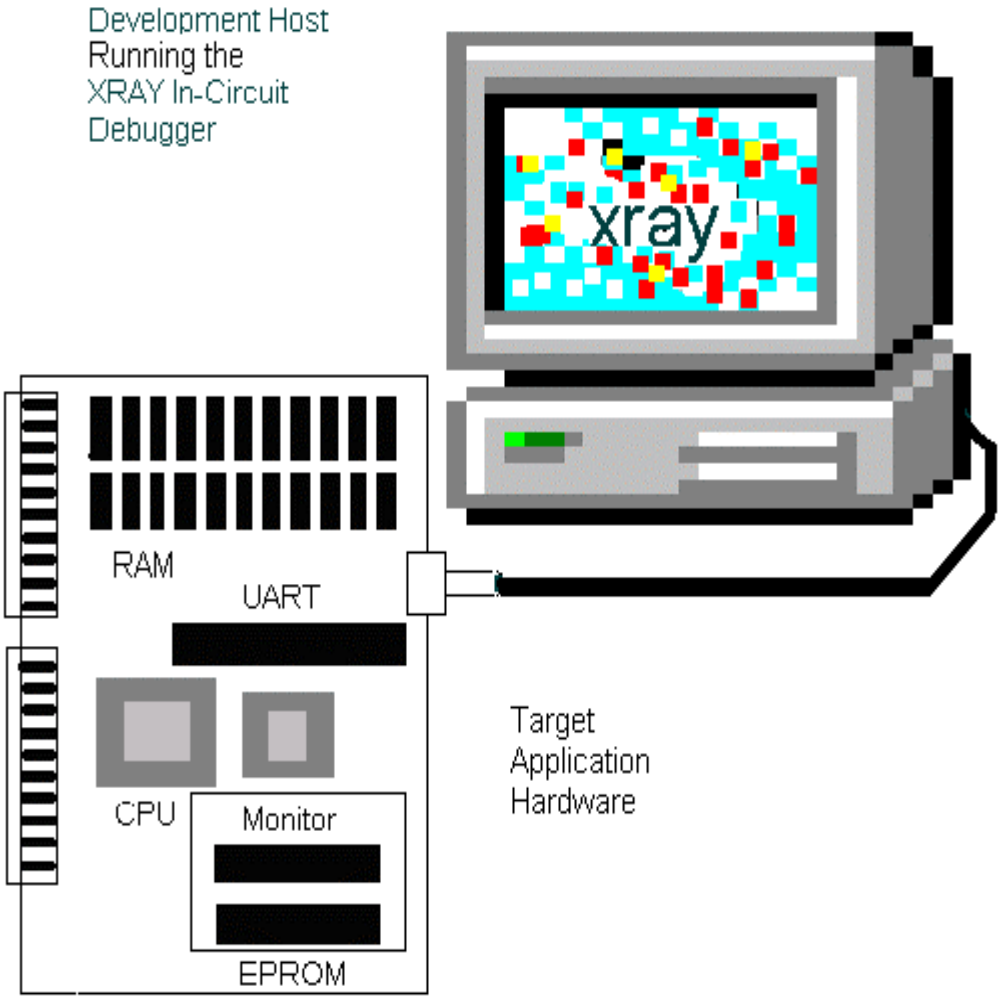
以上三种例程的源程序可以在 SRC 和 INCLUDE 目录下看见。

第二章　MONITOR

2.1 前言

XDM86 的 Monitor 与 XHM86 XRAY 在线调试器相结合，可以为用户提供一个实时、在线的高级调试环境。它有许多在线仿真器的功能，但是却花费甚少。驻留在目标机上的 Monitor 通过宿主机和目标机之间的通讯机制大大便利了集成的实时微处理器系统上的各种开发和调试工作。

Monitor 根据不同的硬件要求可以工作在多种工业标准的 CPU 板上、多种厂商提供的汇编语言调试监控、多种实时核心以及不同的串行接口设备。您可以通过不同的硬件配置来建立 Monitor。


Development Host
Running the
XRAY In-Circuit
Debugger
xray
RAM
UART
CPU
Monitor
EPROM
Target
Application
Hardware

2.2 宿主机和目标机的调试环境

使用 XHM86 XRAY 的在线调试器包括两部分环境：开发所用的宿主机和目标应用的硬件部分。XHM86 构件总是运行在您的开发系统上。您可以把 XHM86 看作一个用户调试的界面，它可以使您看到高级语言或汇编级语言的代码、设置的断点、显示的数据结构等等。XDM86 Monitor 驻留在目标硬件上，通过它才使得 XHM86 有能力控制您的应用程序的执行。如图所示：

• 目标板硬件需求

装载 Monitor 的硬件目标板必须达到以下几个要求:

(1). 与宿主机上的 XRAY 在线调试器进行通讯的专用 I/O 设备
(2). 容纳 Monitor 代码的可用 ROM 或 RAM 空间
(3). 用于 Monitor 工作空间的读写内存

• 建立 Monitor 的步骤

第一步，是否能用 Prebuild Monitor。

目前已有一些现成的处理器及其相应的 I/O， Monitor 已作好了配置。如果您的目标环境能够与它们相匹配，这当然最好不过了。在文件 BOARDS.XDM 中列出了所有的 Prebuild Monitor，这都是为一些标准板做好了配置的 Monitor。比如：使用 80186 CPU 的 R.L.C.SBC-186 目标板，其相应的 Monitor 已建立完毕，可直接使用。

第二步，使用 MCT86(Monitor Configuration Tool)。

如果您不能使用上述的 Prebuild Monitor，那么您可以使用 MCT86 这一工具来建立您的 Monitor。这一工具针对各种目标板支持大量的器件选项，各种器件种类十分丰富。MCT86 使用菜单驱动的方式，让您针对不同的目标板、不同的工作器件进行选择，比如对于 I/O 通讯设备，它有 4、5 种不同的器件供您选择，这基本上就包括了大部分现在流行的串口设备，如果您的目标板正好就使用了这些器件中的一种，那么您将不用再为通讯这一部分而劳神了，标准配置将为您的器件按照您对串口的所有要求一一建立对应的例程。建立了标准的 MCT86 的配置，在大多数情况下，它都能满足您的要求。

第三步，修改。

针对您的目标板，以 MCT86 生成的源文件为样板进行修改和匹配，这当然是因为 Monitor 与目标板的硬件及驻留的软件的接口需要较为特殊的环境所造成的。您应该针对您的目标板建立适应您的要求的 BSP(Board Support Package)。比如您的 I/O 口不在 MCT86 的配置之列，那么您就只好参考生成的模板针对您的通讯器件进行初始化及通讯例程的编写工作了。通常在这一步中，我们只需加入针对我们的目标板的一些较为特殊的硬件初始化部分就行了。

将上述自动产生及作过修改的源文件经过编译、链接之后，将生成一个 16 进制文件，这就是我们所要建立的 Monitor，将这个 16 进制文件写入目标系统(如比 EPROM 中)，

然后从宿主机上进入调试环境，下载应用程序，即可方便地进行调试和跟踪。

• Monitor 的文件(针对 PC)

☞ XMON86.OBJ

包括 Monitor 的核心功能如命令处理、断点控制、执行控制等等，它只是以目标码的形式(.OBJ)提供给用户的。

☞ MONBASE.MAC

汇编语言的源文件，包括 Monitor 所需的各种目标环境信息。它所包含的文件 BOARDSET.INC 含有各种源文件中的控制信息。这个文件控制了所有环境的起始，并定义了配置表和工作空间。这个文件可以说是 Monitor 的主程序，因为其它文件中都以子程序的形式提供各种服务例程，而它则是整个程序的入口，并负责各子程序的调用工作。程序从 COLD_PC 这个子例程入口，直到最后将控制权交给 Monitor 核心。

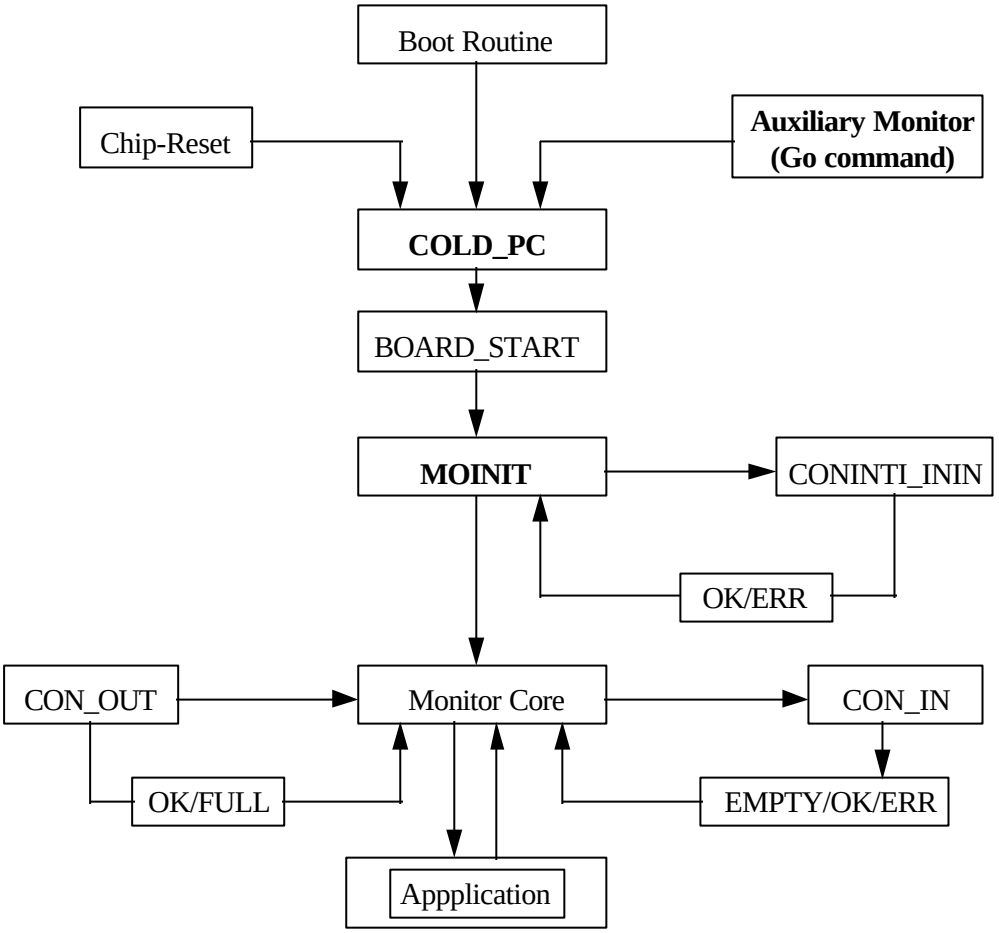
☞ BOARD.MAC

汇编语言的源文件，包括用户和 MCT86 所提供的各种功能例程(目标板初始化和 I/O 驱动。它的控制是由包含文件 BOARDSET.INC 及 WEQU.INC、HWDRV.INC 所提供的。

☞ BUILDMON.BAT

文件是最后的命令批处理文件，它负责源文件的编译、链接工作，生成 Monitor 监控的十六进制码，在文件 XDM86.ABS 中，将此文件写入 EPROM 内，并注意在缓冲区最高地址处加一条跳转指令到监控程序开始处即可。
以上文件是整个 Monitor 中最关键的文件。

• Monitor 流程



☞ COLD-PC

程序的入口点，首先要进行与目标板密切相关的状态寄存器配置、存储分配、堆栈分配;它在文件 MONBASE.MAC **中。** ☞ BOARD_START

目标板初始化，板级初始化，可将其看作 BSP **的预初始化，它位于文件** BOARD.MAC **中，这一文件保存了用户提供的** I/O **例程、辅助** MonitorI/O **服务以及用户应用程序向量。它也可以改变** Monitor **的配置表;执行特别的初始化;它还可以将表从** ROM **拷贝到** RAM **等操作。**

☞ MOINIT

Monitor **初始化，**MOINIT **例程根据** Monitor **配置表执行所有** Monitor **内部的初始化。完成之后** MOINIT **调用** CON_INIT **以初始化目标板与在宿主机上的** XRAY **在线调试器之间的通讯。**MOINIT **在** XMON86.OBJ **文件中。它只以目标文件的形式提供**(.OBJ)。MOINIT **的入口点在** monvise.inc **中定义。**

☞ CON_INIT

I/O **初始化。初始化与在宿主机上的** XRAY **在线调试器通讯。根据所使用的配置情况，可将其初始化为串行** I/O **设备。若目标板上有** ABORT **按钮，还可初始化** ABORT **向量。**

☞ CON_IN

字符输入。与宿主机上的 XRAY 在线调试器通讯。检查字符，并返回一个字符或相应的状态码。根据所使用的配置，我们将这一通道就作为一个串行 I/O 设备来完成。

☞ CON_OUT

字符输出。Monitor 核心代码与宿主机上 XRAY 在线调试器通讯。CON_OUT 例程为 Monitor 核心代码执行一个字符输出功能。CON_OUT 试图将字符写入 DEBUG 输出设备，并返回相应状态码。根据所使用的配置，我们将这一通道就作为一个串行 I/O 设备来完成。

• 应用实例简述

我们曾经为某单位的一块硬件目标板配置了相应的 Monitor，应该说 Monitor 的配置还是比较容易的。只要您对自己的硬件目标板比较熟悉，一切都将十分顺利。这块目标板的 CPU 采用的是嵌入式微处理器 Intel 的 386EX，目标板上还包括计时/计数器、实时钟以及大量的并行口等等，当然 386EX 内部的外围如同步串口、异步串口、计时/计数器、看门狗等都得到了充分的利用。除了大量的 I/O 口之外，板上当然还带有自己的 ROM 及 RAM 区。由总线控制器完成总线信号的转换，配合 CPU 与外围的工作时序。大多数情况下，您都可以使用 MCT86 工具为您的目标板建立一个模板配置，无论它是否与您的具体应用相一致。当然您应该选择最为接近的配置。关于 MCT86 的菜单选择是十分容易的，您只要运行一下这个工具，便会一目了然。比如 CPU 是选择 186、286 还是 386，串口是选择 8250，还是 8251，波特率是选择 2400，还是 19200……这些只要您对目标板有一个大致的了解，所有的选择都将不费吹灰之力。选择完毕，MCT86 将会自动生成相应的源文件，经过一个“BUILDMON.BAT”文件可编译、链接生成所需的 16 进制文件。由 MCT86 生成的源文件当然不能不加修改就完全符合用户目标板的要求。针对 386EX 的特性，在 Monitor 进入程序入口点 COLD_PC 之后，应该立即将一些关键的电源管理特性、状态寄存器配置及内存、椎栈的分配初始化好。之后的板级初始化工作 BOARD_START 应根据用户目标板的特性及配备的操作系统要求加入一些关键的外围初始化工作，如中断向量的分配、中断工作方式等等，注意在这里并不要求将所有的外围都初始化完毕，因为用户将来要根据不同的应用程序需求，将外围配置成不同的工作方式，而这些工作完全都可以放在下载到 RAM 区的应用程序来完成，所以这里的板级初始化工作不必过于复杂和完备。最后还要重点注意一下有关串口的初始化和驱动例程。比如在文件 HWEQU.INC 文件中列出的不同波特率情况下的对应的除数因子，这是 Monitor 按标准的串口时钟生成的，它究竟与您的设计是否一致呢？这样的细节应加以注意。仔细阅读一下 MONBASE.MAC 文件及 BOARD.MAC 文件，它们都有详细的注释，根据您的硬件目标板设计情况作一些小小的改动，您会轻轻松松的完成 Monitor 的建立工作。

2.3 基于 Spectra 的 MONITOR

驻留在目标上的 DEBUG MONITOR 和 BSP 一起存在于可引导的 BOOT.HEX 中，它在引导过程作为一个程序（method）被启动。在 Spectra 的调式环境下，目标上可以启动的 DEBUG MONITOR 包括 Xtrace 和 XDM。由于使用的通信协议不同，Xtrace 只

能和 TARGET MANAGER 配套使用，XDM 只能和 XHM 配套使用。由于 Xtrace 的功能较 XDM 强大，所以一般在 Spectra 的调试环境下使用 Xtrace。

为了使 Xtrace 正常运行，下面以主机和目标板之间用网络连接为例说明需要在 <board>.def 文件中添加的内容：

```
boot.env.dev.bridge.envname: BRIDGE
boot.env.dev.bridge.value:     ether_1
```

这两项说明主机和目标板之间使用连接方式；

```
boot.env.dev.timer.envname:    TIMER
boot.env.dev.timer.value:      timer_1
```

Xtrace 使用 TIMER 实现非阻塞（nonblocking）特性；

```
boot.env.variables: ip_addr,ether_addr
boot.env.ether_addr.envname:    ETHER_ADDR
boot.env.ether_addr.value:   08:00:14:21:45:80
boot.env.ip_addr.envname:  IP_ADDR
boot.env.ip_addr.value: 202.115.4.152
boot.methods:  rarp,xtrace
boot.env.boot_order:     rarp,xtrace
```

指定启动的程序，包括 rarp 和 xtrace。rarp 即实现 RARP 协议，解决物理地址到 IP 地址的映射。

第三章　　固化

3.1 保护模式下程序固化的原理与实现

VRTXsa x86/fpm 操作系统由一系列库模块组成，它们最终和应用部分程序连接在一起形成一体的，绝对定位的代码，载入到目标系统中。通过 VRTXsa x86/fpm 软件包中提供的 DMAKE 工具生成的代码有两种方式载入目标系统：

1. 通过 XRAY 调试器加载到 RAM 中。
2. 固化到只读存储器（PROM，ROM 或 FLASH）中。

方式 1 在调试阶段非常有用，当调试过程结束，代码中的错误被排除后，就可以用方式 2 把代码固化到 EPROM/FLASH 中，程序就可以脱离调试环境在单板上独立运行。

固化的代码和 RAM 中的代码有两个差别：

1. 代码定位不同。
2. 初始化部分不同。

3.1.1 代码定位

1. 如果代码在 RAM 中运行，那么全部代码和数据都定位在 RAM 中，如下图 3-1 中所示的 board.memory.code.address 开始的地方，此时系统的只读存储器中存放的是 X-RAY DEBUGGER 的 MONITOR 代码。

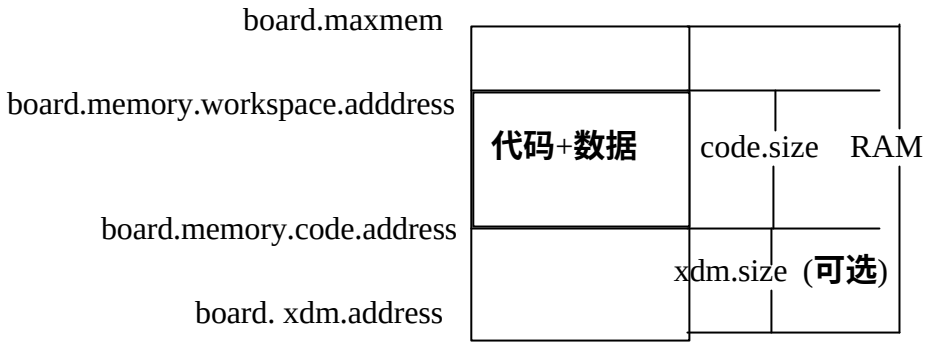


图 3-1 RAM 运行方式下代码在存储器中的定位

2. 如果程序是在只读存储器中运行，那么应用部分的代码存放在只读存储器，即图 3-2 中 board.memory.code.address 开始的地方，这部分存储器只要不和 RAM 的地址发生冲突，在 CPU 可管理的存储区的范围，可以位于任何地方，但是其结束地址一定是在 1MB 的边界，即 XXXFFFFFH。

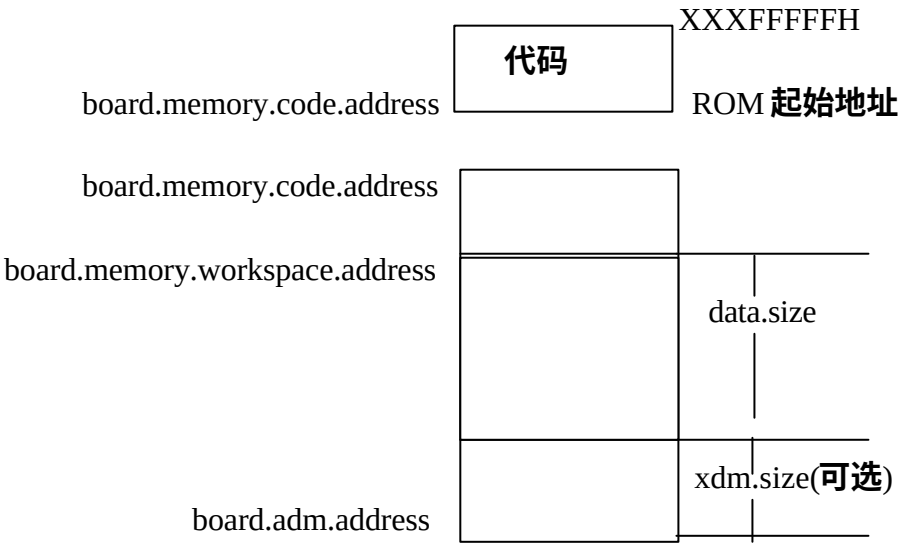


图 3-2 ROM 运行方式代码存储器中的定位

明确了上述定位关系，根据系统的资源使用情况修改<project>.def 中相应的变量。

3.1.2 初始化过程

RAM 运行方式下，涉及系统的基本配置是由 MONITOR 起动时完成的，而在 ROM 方式下工作时，系统基本配置必须包括在代码中，而且这部分代码是和硬件环境相关的，应对系统提供的模板文件 CRT0EXF.S（该文件针对 intel EV386 板）进行修改。

3.1.3 实模式到保护模式的转换

Intel386EX 有四个控制寄存器（CR0、CR1、CR2 和 CR3，CR1 为 Intel 保留），它们用来存放全局特性的机器状态。CR0 寄存器中有一位保护模式位 PE。在对 CR0 的 PE 位操作之前，应该用指令 lgdt 和 lidt 把 GDT 表和 IDT 表的 32 位线性基地址和 16 位极限值装入相对应的系统寄存器 GDTR 和 IDTR 中。从实地址模式进入保护模式是将控制寄存器 CR0 的 PE 位置 1 来实现的。程序用指令—— MOV CR0，寄存器 ， 来完成将 PE 位置 1 的操作。这样， 在这条指令的前后，CPU 处于两种不同的工作方式，对指令有不同的解释。因此，应紧接着用一条 JMP 短跳转指令来清除已预取到 CPU 队列的指令或已译码但尚未执行的指令。

在短跳转指令之后接着进行一次长跳转，则将长跳转目的段的选择符（即指向描述符表的索引）装入段寄存器。如果 GDTR 的内容（GDT 表的基址和段限）正确，程序代码将按照分段管理保护方式执行：根据段寄存器的内容（选择符），从描述符表中取出相应的段描述符（含段基址，界限和存取权限属性）；段描述符中的 32 位基地址与 32 位有效地址相加，就形成所要访问的 32 位的物理地址。

3.1.4 GDT 表和 IDT 表的定位

程序固化过程中，涉及到 GDT 表和 IDT 表在 FLASH/EPROM 中的定位。要使 GDT 表和 IDT 表正确定位，首先，应了解目标板的内存映象，清楚 FLASH/EPROM 在内存中的地址映射范围。

对于 FPM 方式，CRT0EXF.S 文件中已经考虑了 GDT 和 IDT 表上电时，在 RAM 中的重新建立，因此，用户不需考虑。

而对于 SPM 方式，在生成应用时就需要在 ROM/FLASH 中定位 GDT 表和 IDT 表，上电后，要将 GDT 表和 IDT 表搬到 RAM 中。可按下列方法处理。

在 386EX 中，如果需要固化的程序代码段指定在地址 3f80000H 之上。一般，GDT 表和 IDT 表在 FLASH 或 EPROM 中的位置可用以下两种方式确定。现用 GDT 表为例说明定位技术。

方式之一，通过在 Build 控制文件中指定 LOCATION = gdt_desc，其中符号 gdt_desc 为初始化模块的 6 字节区域。Build 过程中把 GDT 表的 2 字节表限和 4 字节基址放在此区域中。然后用如下指令将 gdt_desc 区域的值装入 GDTR。

```
mov eax,offset gdt_desc
lgdt  cs:[eax]
```

方式之二，根据 intel 386EX/EV 板的内存映象，应将 GDT 表定位在 FLASH 内存块内。通过在 Build 控制文件中直接指定 BASE 值,BASE = 3f80000H，则将 GDT 表指定在绝对物理地址 3f80000H 处。

对于第二种方式来说，由于已事先确定 GDT 表放在某个绝对地址处，这样可以在初始化代码中把 GDT 的基址和表限放在某一个 DRAM 临时内存空间，然后通过 lgdt 指令装入 GDTR。但某些时候临时内存空间不可用，比如对于 Intel386EX，在复位的情况下，仅仅 UCS#通道使能整个 64M 的地址空间。而此时 UCS#通道用来选定 FLASH，在

连接 DRAM 的片选通道未初始化之前，DRAM 临时内存空间不可用。在这种情况下，可在初始化代码中按如下方式装载 GDTR：

```
GDT_BASE    EQU    3F80000H                ；GDT 表基址
GDT_DESC_COUNT    EQU    512               ；GDT 表项数
GDTR_VALUE    dw    GDT_DESC_COUNT*8       ；GDT 表表限
                    dd    GDT_BASE
lgdt    CS：GDTR_VALUE                     ；装入基址和表限
```

特别值得注意，IDT 表在初始化进入保护模式时不会马上用到，所以可以将 IDT 表在 Build 控制文件中用指定 BASE 的方法定位到 DRAM 的某一位置。但如果将 IDT 表指定到 00H 处，对于目标板为 FLASH 的情况不会出现什么问题，装入工具软件不会把 *.hex 文件中的 IDT 表装入 FLASH 中。对于目标板为 EPROM 的情况，如通过编程器装入的是*.bin 文件，编程器把本该放在 DRAM 的 00H 处的 IDT 表放在 EPROM 中，使 GDT 表没有放到指定的位置，引起程序代码在保护模式下不能正常运行。

3.1.5 内存的映射

下面以 i386EX/EV 板为例，说明 FLASH 在内存中的分布及其初始化。

intel 386EX/EV 板使用片选通道 UCS#使能 FLASH，使用片选通道 CS4#使能

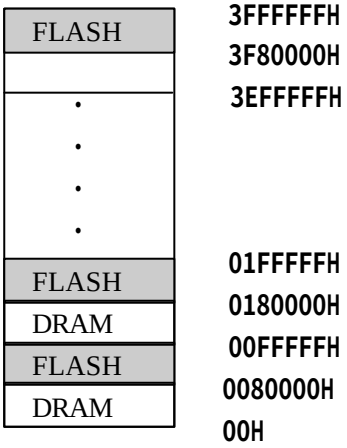


图 3-3 系统复位内存映象

DRAM。i386EX/EV 板在系统复位后经板上的辅助监控器初始化后的内存映象如图九所示。

由上图可见，FLASH 被映射在地址 XX80000H--XXFFFFFH 之间（XX 代表任意值），即每 1M 地址空间的高端 512K 重复。为了使 80000H--FFFFFH 范围的 DRAM 可用必须重新初始化内存映象。

i386EX/EV 板在进入保护模式后被初始化成的最终内存映象如图 3-4 所示。

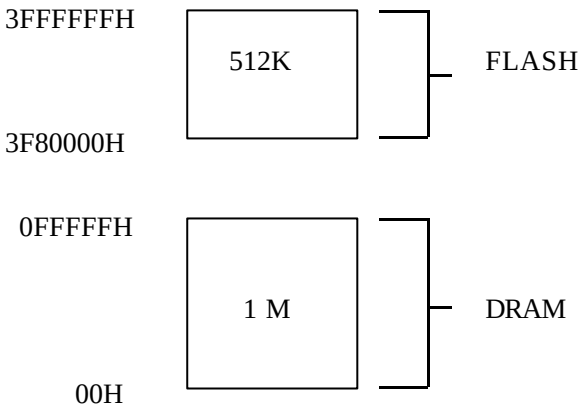


图 3-4 Intel 386EX/EV **适合保护模式的内存映象**

3.1.6 固化实例

1．**建立正确的工程配置文件**

以 EV386EV **板为例，利用** MICROTEC **提供的** dmake **工具，生成基于** ROM **的代码时，必须在**<project>.def **中“**@include(ev386ex.def)**”后加入以下声明：**

```
make.vrtxsa.type: rom
tool.format:ihex
@include(memdefs/3exflash.def)
```

特别地注意 3exflash.def **文件中对以下变量的声明必须和** ROM **的最终映射地址空间相一致。**

```
board.application.code:            3F80000
board.application.code.size:       80000
board.memory.code.address:         0x3F80000
board.memory.code.size:            0x80000
board.memory.code.type:            BOOTOS_MEMORY_SHARED|BOOTOS_MEMORY_COPYBACK
```

2．**生成可加载的** HEX **文件**

利用 Xconfig <project>.def，**将生成应用的**.abs **文件转换成** HEX **文件，就可以利用** intel **的编程工具** FLASHLDR **将** Intdemon **下载到** FLASH **中。**

3．**编写命令加载文件**

在使用 FLASHLDR **写入前应先编写命令加载文件（***.FLC**），它相似于** DOS **不传递参数的批处理文件。**

典型的 FLC **文件内容为：**

```
init  com2                  //初始化宿主机和目标机的连接
baud=9600                   //设置下载波特率为 9600bps
rt  exeval                  //重新初始化 FLASH 表
program  .\intdemon.hex     //指明下载的文件为当前目录下的 Intdemon.hex
vector=9000:063C            //指定实模式下执行的第一条指令地址
```

```
setboot                    //设置程序在启动时执行
shutdown                   //关闭宿主机和目标机的连接
exit                       //退出
```

其中 vector 的值可以在随 Intdemon.abs 文件同时生成的 Intdemon.mp2 中查到。在 Intdemon.mp2 文件中 LDT1 的 crt0._START 和主任务 TSS 内容 CS:IP=GDT(3):3F9063C 表明起始地址为实模式下的 9000：063C。

4．跳线

在用 FLASHLDR 加载前，将 EV386EX 板的电源关闭，合上跳线 JP1，再开启电源。

5．执行加载命令

FLASHLDR 命令行语法为：

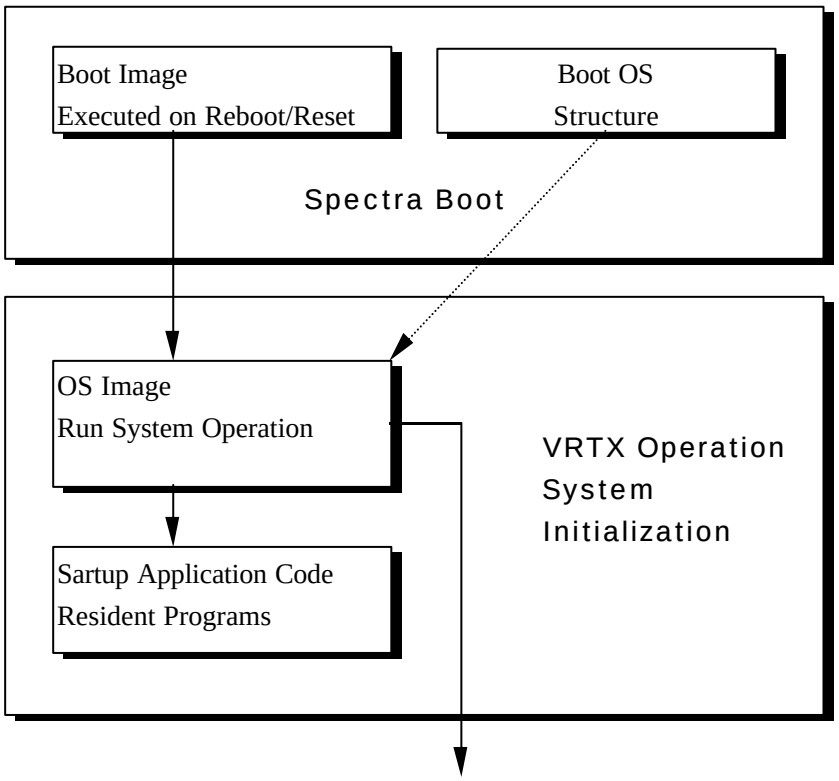
```
FLASHLDR    Intdemon.flc
```

下载后，关闭 EV386EX 板的电源，移去 JP1 跳线，按 RESET 键即可启动 Intdemon。

3.2 Spectra 的固化

Spectra 环境下的固化和 VRTXsa x86/fpm 的固化有相似之处，都是通过设置 xconfig 变量来实现。但是，Spectra 环境下的固化有它的特殊性，这和它的调试环境有关。

在调试阶段，目标上驻留的是分别链接生成的映像（image），它们包括启动映像（boot image）、操作系统映像（os image）、应用程序映像（application image）。这些映像通过函数调用、数据结构传递和自陷（trap）相互作用。它们之间的相互关系如图 3-5 所示：



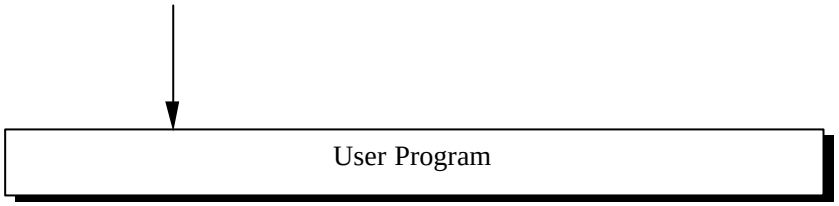


图 3-5 映像关系图

联系本书的 BSP 部分，可以看出在整个应用程序的开发阶段，对程序进行了两次固化，一次是将启动映像（boot image）即前面所述生成的 BOOT.HEX 固化到 PROM 中，另一次是在应用经过调试后，把启动、操作系统和应用映像链接在一起，形成一个单独的映像，然后将它编程到 PROM 中去。

3.2.1 启动映像的固化

如前所述，首先应编写一个<board>.def 文件，文件内容包括下面四个部分：

◆目标板名、处理器和工具定义

board.name: <board>

board.target: mc68010

这是 Microtec 专门的对板上处理器的描述，它的合法值参见有关参考书；

board.proc: 68010

指定处理器类型，合法值包括 sun4,68000,68010,68020,680303,68040,68020,68020,680303,68040,sparc,sparclite,80386。它的作用在于决定库和 OBJ 文件的路径、配置基于处理器类型的库等；

board.tool.type: m68k

它的作用在于选择将被使用的编译器工具包。合法的值包括 m68k，sparc，i386；

board.cc_flag: -DMC68010 -DM68K

用于设置编译时使用的选项。

◆设备情况

board.devices: board,serial_1,timer_1,ether_1

dev.board.name: DEV_BOARD

dev.board.method: logio_board_method

dev.serial_1.name: DEV_SERIAL_1

dev.serial_1.method: logio_serial_1_method

dev.timer_1.name: DEV_TIMER_1

dev.timer_1.method: logio_timer_1_method

dev.ether_1.name: DEV_ETHER_1

dev.ether_1.method: logio_ether_1_method

在板上的设备能够被使用来作为控制台（consol）和桥（bridge）之前，一定要在 xconfig

变量 board.devices 中列出，并且要使物理设备名（DEV_BOARD）和一个 method（logio_board_method）结构相关。

◆内存布置

对于一个目标板的内存，至少应该定义四个分区：

●Boot Code（启动程序代码使用的 ROM 区）

●Boot Data（启动程序程序使用的 RAM 区）

●BOOTOS_MEMEORY_UNUSED_HOST（下载应用程序使用的区间，仅仅定义一个）

●BOOTOS_MEMEORY_UNUSED_TARGET（下载操作系统使用的区间，仅仅定义一个）

board.boot.code: 000000

Boot Code 起始地址；

board.boot.data: 400000

Boot Data 起始地址；

board.memory :rom,data,host,target

定义内存的分区，每一个分区下面进一步定义其起始地址、大小和类型；

board.memory.rom.address : 0x000000
board.memory.rom.size : 0x20000
board.memory.rom.type : BOOTOS_MEMORY_READONLY

board.memory.data.address : 0x400000
board.memory.data.size : 0x40000
board.memory.data,type : BOOTOS_MEMORY_SHARED

board.memory.host.address : 0x480000
board.memory.host.size : 0x100000
board.memory.host.type : BOOTOS_MEMORY_UNUSED_HOST

board.memory.target.address : 0x580000
board.memory.target.size : 0x200000
board.memory.target.type : BOOTOS_MEMORY_UNUSED_TARGET

分区类型除上面四种以外，还包括 BOOTOS_MEMORY_BOOTED 和 BOOTOS_MEMORY_I/O。

◆启动配置

定义桥（bridge）和控制台（consol）的配置。它们的可能配置如下表所示：

桥	控制台
Ethernet	VCONSOL
Ethernet	serial
serial	VCONSOL

当桥为 Ethernet、控制台为 VCOSOL 时的配置为：

boot.env.devices: board,bridge,timer
boot.env.dev.board.envname: BOARD
boot.env.dev.board.value: board
boot.env.dev.bridge.envname: BRIDGE
boot.env.dev.bridge.value: ether_1
boot.env.dev.timer.envname: TIMER
boot.env.dev.timer.value: timer_1
boot.objs.rs.curdir: bootcnfg,devices,devcnfg

指定将要被增加到启动映像中去的文件；

boot.env.variables: ip_addr,ether_addr
boot.env.ether_addr.envname: ETHER_ADDR
boot.env.ether_addr.value:08:00:14:21:45:80

boot.env.ip_addr.envname: IP_ADDR
boot.env.ip_addr.value: 202.115.4.152
boot.methods: rarp,xtrace
boot.env.boot_order: rarp,xtrace
board.devcnfgdir: /xxx/cpbsp

使用文件 devcnfg.c 来修改缺省配置。devcnfg.c 的缺省路径是：

$SPECTRA/target/xsp/*board*/common

为了改变缺省路径，使用 xconfig 变量 board.boarddir 所设置的值；

board.boarddir: /xxx/cpbsp

被 Xconfig 产生的 makefile 文件搜寻目标板 BSP 库文件和 crt0.o 的缺省路径是：

$SPECTRA/target/xsp/*board*/object_format

为了改变缺省路径，使用 xconfig 变量 board.boarddir 所设置的值；

vrtxos.consol: DEV_VCONSOL

上面所述的 Xconfig 变量和 MONITOR 相关的部分在 MONITOR 部分说明。

编写一个 user.def，内容如下：

@include(boot.def)
@include(cp.def)
@include(microtec.def)

使用命令 xconfig user.def 生成 BOOT. HEX。

3.2.2 应用程序固化

应用程序的固化映像包括 boot、os 和 application。为了精确定位代码所需要的地址空间，一方面需要了解目标板的内存分布，另一方面需要查看在调试阶段生成的 boot.map、os.map 和<应用>.map 文件，知道各种代码段以及初始化数据段的大小，并准确的在 user.def 文件中反映出来。这一部分将以目标板 mo133 为例，讲述固化的过程。mo163 在调试阶段的内存分布如图 3-6 所示：

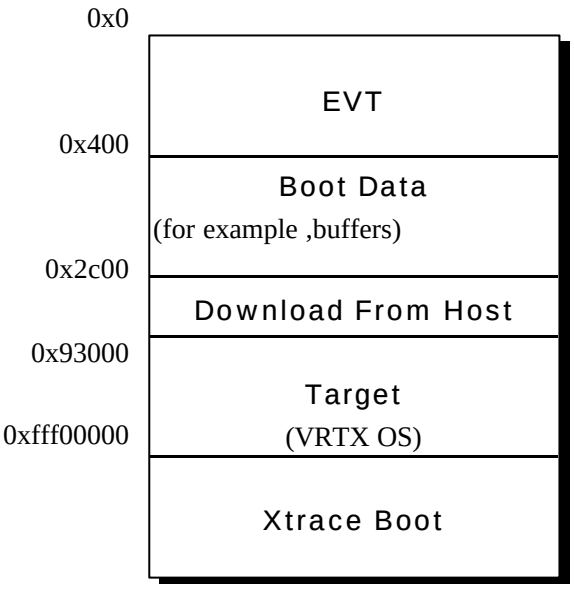


图 3-6 **内存映像**

在进行固化之前，假设应用程序 demo.o **已经生成。固化过程按下列步骤进行：**

1 • **创建** user.def **文件，它包含以下内容：**

```
@include(vrtxsa.def)
@include(mo133.def)
@include(microtec.def)
@include(boot.def)
vrtxos.obj.usr: ./demo.o
sys.entry_point2: start_main
make.what:    final
make.rule:    sutil
make.*.final.name:    final
make.*.final.target:    demo.hex
make.*.final.deps:    hextmp
make.*.final.code:    fff00000
make.*.final.hextmp.rule:    cathex
make.*.hextmp.name:    hextmp
make.*.hextmp.target:    hextmp.hex
make.*.hextmp.deps:    boot, vrtxos
make.*.hextmp.boot.rule:    link_hex
make.*.boot.code:    fff00000
make.*.boot.data:    40000
make.*.boot.target:    boottmp.hex
```

```
make.*.hextmp.vrtxos.rule:      link_hex
make.*.vrtxos.code:     fff08000
make.*.vrtxos.data:     80000
make.*.vrxos.target:     ostmp.hex
boot.methods:    go
boot.env.boot_order:     go
boot.method.go.orderparam:     fff08000
```

在系统发出 vrtx_go **调用之前将调用** start_main **过程，它将创建** main **任务：**

```
star_main( )
 {
      int    err;
      int    main;
      sc_tcreate(main, 11, 2, &err);
      if(err)
      sys_bsp_abort( );
 }
```

在上面 user.def **文件中，** make **规则被广泛的使用以产生** makefile **文件。在** makefile **文件中，最终的目标文件是** demo.hex，**它使用工具** sutil **生成。在** makefiel **文件中的形式是：**

```
demo.hex : hextmp.hex
            sutil -S3   -r   fff00000 hextmp.hex > demo.hex
```

使用 xconfig user.def **生成** demo.hex。

附录：logio 的定义

在 logio.h **中：**

```
/*
 * Device id definitions
 */
typedef struct logio_method   * logio_device_id_t;

typedef struct {
   logio_status_t (*init)(_ANSIPROT1(logio_device_id_t));
   logio_status_t (*read)(_ANSIPROT3(logio_device_id_t, char *, int *));
  logio_status_t (*write)(_ANSIPROT3(logio_device_id_t, char *, int *));
  logio_status_t (*control)(_ANSIPROT3(logio_device_id_t, logio_control_t, void *));
  logio_status_t (*getmsg)(_ANSIPROT2(logio_device_id_t, logio_buff_t **));
  logio_status_t (*putmsg)(_ANSIPROT2(logio_device_id_t, logio_buff_t *));
  logio_status_t (*poll)(_ANSIPROT3(logio_device_id_t, logio_control_t, void *));
} logio_fops_t;

typedef struct logio_method {
  logio_fops_t       *logio_fops;
```

```
   void     *data; /* user defined structure */
} logio_method_t;
```

在 devcnfg.c **中：**

```
const logio_method_t      logio_serial_1_method = {

        (logio_fops_t  *)&logio_serial_fops,
        (void  *)&serial_1_dev_desc
};
serial_dev_desc_t   serial_1_dev_desc;
const logio_device_id_t  logio_serial_1_id = (logio_device_id_t)&logio_serial_1_method;
```

在 serial.h **中**

```
extern const logio_fops_t logio_serial_fops;
typedef struct serial_dev_desc_t {
    void  *map;
    serial_config_t    config;
    serial_fops_t    fops;
    extern_fops_t   externio;          /* external register read/write and interrupt functions */
    union  {
        serial_tty_buff_t  tty;
        serial_pkt_buff_t pkt;
    }  buf;
    void  *xtra;
} serial_dev_desc_t;

typedef struct {
    int (*tx_tst)(_ANSIPROT1(logio_interrupt_entry_t *));
    int (*rx_tst)(_ANSIPROT1(logio_interrupt_entry_t *));
    int (*err_tst)(_ANSIPROT1(logio_interrupt_entry_t *));
    vbsp_return_t (*init)(_ANSIPROT1(struct serial_dev_desc_t *));
    vbsp_return_t (*transmitter_off)(_ANSIPROT1(struct serial_dev_desc_t *));
    vbsp_return_t (*start_transmitter)(_ANSIPROT1(struct serial_dev_desc_t *));
    vbsp_return_t (*int_info_set)(_ANSIPROT2(struct serial_dev_desc_t *,
                logio_int_entry_t  *));
    vbsp_return_t (*int_info_get)(_ANSIPROT2(struct serial_dev_desc_t *,
                logio_int_entry_t  *));
    vbsp_return_t (*pputc)(_ANSIPROT2(struct serial_dev_desc_t *,
                unsigned  char  ));
    vbsp_return_t (*pgetc)(_ANSIPROT2(struct serial_dev_desc_t *,
                unsigned  char  *));

   /* the following pointer should be used if more device specific
    * functions must be called through the function table
```

```
      */
    void *specific;
} serial_fops_t;
typedef struct {
    serial_flavor_t flavor;

   /* the counter value to write to the baud rate generator register */
    unsigned long baud_counter_val;
     serial_parity_t     parity;
     serial_bits_t       bits;
    serial_stop_bit_t stop;

   /* interrupt vectors must be known even if not programmable */
    unsigned char rx_int_vector;
    unsigned char tx_int_vector;
    unsigned char err_int_vector;

   /* the following pointer should be used to add extra device
   ** specific information needed that all devices may not need
    */
    void *specific;
} serial_config_t;
```

在 extio_1.h **中：**

```
typedef struct {
     void (*reg_write_b)(_ANSIPROT2(BYTE *, BYTE ));
     /* and initialized PER DEVICE*/

     void (*reg_write_w)(_ANSIPROT2(WORD *, WORD));
     /* these function pointers */

     void (*reg_write_l)(_ANSIPROT2(LONG *, LONG));
     /* are used to call functions*/

     BYTE (*reg_read_b)( _ANSIPROT1(BYTE *)); /* that may contain board */
     WORD (*reg_read_w)( _ANSIPROT1(WORD *));  /* specific code */
     LONG (*reg_read_l)( _ANSIPROT1(LONG *));
     vbsp_return_t (*interrupt_start)(_ANSIPROT2(void *, unsigned char));
     vbsp_return_t (*interrupt_end)(_ANSIPROT2(void *,unsigned char));
     vbsp_return_t (*brd_spec_int_disable)(_ANSIPROT2
                              (void *,logio_int_entry_t *));
     vbsp_return_t (*brd_spec_int_enable)(_ANSIPROT2
                             (void *,logio_int_entry_t *));
     /*  the following pointer should be used to add extra device
       *  specific information needed that all devices may not need
```

```
        */
       void *specific;
} extern_fops_t;
```

应用开发篇三：VRTXsa 的配置

第一章 配置过程概述

- 配置过程
- 使用 Xconfig 进行系统配置
- 建立应用

1.1 配置过程

VRTXsa 操作系统和它的选件 IFX，SNX，RTL 与应用软件相连接，产生一个单一的，绝对的，可引导的，可下载到目标机存储区中的存储映象，然后执行。应用映象可以以三种方式装载：

- XRAY 调试器
- 写入 PROM，ROM 或 FLASH，由处理器复位

1.2 使用 Xconfig 进行系统配置

Xconfig 工具是一个自动配置工具，帮助定义系统参数。在这一过程中，系统参数的设置决定于：

- 目标机硬件特性
- 希望使用的指定操作系统特性
- 编译过程的控制

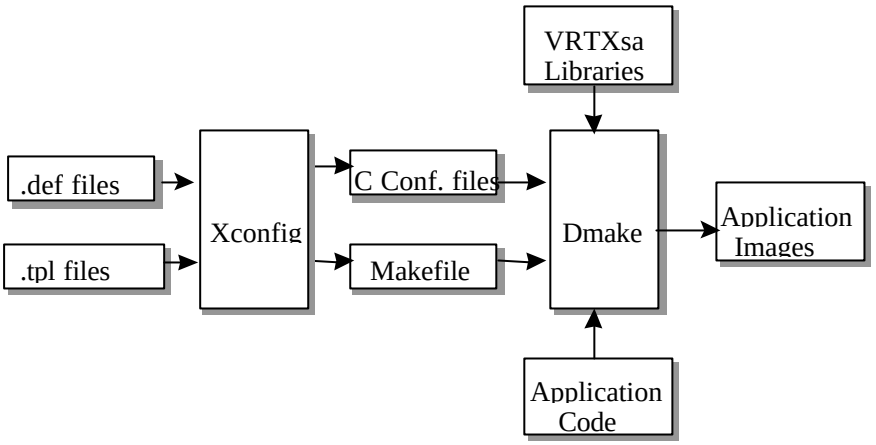
Xconfig 工具从定义文件中读取这些系统参数，并且使用这些参数转换样板文件为 C 语言文件，make 文件，及命令文件(.CMD)，这些都是对特殊目标机环境和系统配置的制作。

从 Xconfig 输出文件用于 make 过程建立绝对系统映象。C 语言文件被编译，与应用代码包括所需系统库连接，并定位到指定绝对内存位置。

提供给 Xconfig 系统参数是变量形式，它指定应用中的所需各种系统模块信息。这些系统参数设置为缺省值在产品中的定义文件。这些缺省值可能匹配目标机和系统配置，或需要修改这些值适应需求。

其配置过程简单来说是这样的：XCONFIG 从一组*.DEF 文件中读取系统参数的设置，然后用这些参数将一组摸板文件(*.TPL)转换为一组 C 配置文件，一个 MAKEFILE 文件，(或.CMD 的控制文件)。

XCONFIG 在完成处理后会自动地调用 MAKE，MAKE 完成编译，链接，最后生成可执行文件。其工作过程可以用下图所示。



Xconfig **配置过程**

配置文件

Xconfig **使用的文件类型**:

- **缺省配置文件**
- **样板文件**
- **工程配置文件**
- Xconfig **和** make **创建的输出文件**

在 microtec/spectra/target/config/default/**目录下提供了一组** DEF **文件，定义了系统变量的缺省设置。为了使系统配置符合用户应用的实际情况，用户需要对缺省的一些系统变量进行修改。但是不要直接修改** microtec/spectra/target/config/default/**下的缺省值，用户应当自己建立一个** DEF **文件，在该文件中定义的值会自动覆盖系统变量的缺省值。**

VRTXsa **提供的这套配置工具是非常智能化的，用户开发自己的应用程序所需要做的工作就是编制一个源程序（***.C**）和一个应用的配置文件（***.DEF**）。**

1.3 建立应用

在应用开发时，要根据实际的情况建立自己的配置文件如： user.def**，该文件中应包含所需的各种缺省的配置文件。同时，还要定义不同与缺省配置的配置参数，如：各模块的名称，内存的大小等。**

然后，编制应用程序。编制完成后，再使用

Xconfig user.def

生成 makefile，一些源码之后。再经过编译、链接生成可重定位，或直接固化的代码。

如：生成 ELF 格式的绝对文件，可用于 XRAY 调试器加载，或使用 Powerpc-eabi-objcopy 工具转换到 Motorola S 格式用于只读存储器。

第二章　基本配置参数

基本配置参数通常建立在 user.def 中。如果使用标准硬件（mo860fads 评估板）作为目标机，这些基本参数可以作为需要改变的配置参数并且它们是缺省设置。在大多数情况下，缺省设置对于一般应用是适用的。可以根据应用和目标环境需要进行设置。

2.1 .def 文件基础

缺省或配置文件包含使用 Xconfig 变量建立系统参数的配置信息

1 • 包含其它配置文件

Xconfig 允许定义配置引入已经建立的定义文件

@include(another_configuration file)

2 • 指定配置变量值

除了@include 语句，配置文件每行指定配置变量名称和值，配置变量可以是布尔值，字串或数字值。

vrtx.user_task_count:　32

3 • 配置变量范围

配置变量可以不同值在文件中被定义多次，以最新定义值为准。config\default 中的文件提供配置变量的单一缺省值，user.def 重定义这些变量。

4 • 注释

配置文件中使用注释两种方法：”#”，”@dnl”。如果行中包含”@”，则必须使用”@dnl”。

5· 空格

Xconfig 具有严格的空格字符语法，注意使用 tabs，spaces 在文件中。

6· 列表

对具有一系列值的配置变量，它的值必须用逗号隔开。

make.include.dirs.usr：..\include,..\..\include

7· 文件名称

不能使用 Microtec 提供的文件名。

2.2 设置应用入口点

在 user.def 文件中使用 Xconfig 变量 sys.entry_point2 建立应用过程，它被创建为一

个任务，这个任务在 vrtx_go 系统调用之后被创建。
变量 sys.entry_point2 值可以是应用代码中的作为公共符号的任何可用过程名，它的缺省值是 main。

2.3 包含应用目标模块

在 user.def 文件中使用 Xconfig 变量 sys.obj.usr 列出包含在应用程序中的所有目标模块。列出的模块名用逗号隔开，列表最后跟随逗号。

sys.obj.usr: myfile1.obj,myfile2.obj,c:\mydir\myfile3.obj,myfile4.obj,

2.4 结合其它Microtec 组件

提供六种变量形式<component>.enabled 引入其他软件模块，IFX，SNX。

2.5 定义系统控制台

使用系统控制台设备作为 sc_putc，sc_getc 系统调用及 printf 的输入输出设备。系统控制台被映射到 user.def 文件中的一个物理设备。Xconfig 变量 vrtxos.console 用于定义控制台物理设备。
CONSOLE 使用 COM1 或 COM2 作为系统控制台
可以建立虚拟系统控制台，DEV_VCONSOLE 为虚拟系统控制台

2.6 设置缺省目标机硬件环境

提供两种目标机硬件环境定义文件在 config\default。如果建立自己硬件，可以选择一种定义文件作为模板，根据需要进行修改，并在 user.def 文件中引入。

@include(mo860fads.def)

2.7 设置目标机内存配置

VRTXsa 需要了解目标机硬件的内存配置以便定位 VRTX 堆，存储分区和 VRTX 工作空间。系统需要了解目标机内存配置避免不可使用的存储区域及分离的基于 ROM，flash 代码区，基于 RAM 数据区。根据需要建立内存映象定义文件。软件包提供了样板文件，如 mo860fads.def。

2.8 目标机设备配置

Xconfig 变量 board.devices 包含用于目标机的所有设备。

2.9 目标机类型

Xconfig 变量 board.target 用于定义使用在目标机系统中的处理器。适当的定义可以告知 VRTXsa 是否和如何处理 CPU cache，MMU，浮点和其他功能。可接收的目标机类型在文件 target.h 中描述。

常用可修改配置变量

如前所述，虽然大量的 Xconfig 变量仅使用缺省设置，但仍有一些变量为用户定义，这些定义通常放在 user.def

vrtx.control_block_count：用户应用中的事件标志组，堆，互斥量及信号量的最大数量

vrtx.cvt_user_max：Microtec OS 组件的最大数量

vrtx.idle_task.stack_size：idle 任务的堆栈大小

vrtx.partition_block_count：所有分区中块的数量

vrtx.partition_extension_count：扩展分区的数量

vrtx.user_task_count：用户定义任务的最大数量

vrtxos.console：定义系统控制台

vrtxos.libs.usr：与 VRTX OS 连接库的绝对路径列表

vrtxos.objs.usr：与 VRTX OS 连接目标文件的绝对路径列表

rtl.heap_size：定义使用 Xconfig 生成 os.o 时对 RTL 需求的堆(heap)的大小

sys.entry_point1：VRTX 内核初始化前调用应用入口

sys.entry_point2：VRTX 内核初始化后但在 vrtx_go 前调用应用入口

user.entry_point：VRTX 内核初始化后但在 vrtx_go 和前 sys.entry_point2 调用应用入口

第三章 高级配置参数

这种类型的配置参数用于自己制作的目标机系统，或很少遇见的应用条件。

- 设置内核配置参数
- 定义 VRTXsa 内核设备
  - 系统设备
  - 逻辑设备
  - 物理设备
  - 物理设备中断
- 存储器配置

3.1 设置 VRTXsa 配置参数

使用配置参数建立内核的工作空间大小，堆栈大小及最大任务和内存分区数。这些参数通过 VRTXsa 配置表传递到内核。

VRTXsa 配置表自动由 Xconfig 工具使用样板文件 vrtxcftb.tpl 建立，并且根据定义文件 vrtxcftb.def 中的指定 Xconfig 变量参数值修改。要修改 VRTXsa 配置表参数，根据配置需要编辑文件 vrtxcftb.def。Xconfig 自动计算工作空间大小。

3.2 定义VRTXsa 内核设备

VRTXsa 设备是由 VRTXsa 内核引用的设备如系统控制台或时钟。对这些设备的处理不同于通过 IFX I/O 和文件管理模块定义和存取的设备。然而通过 SNX 模块存取的网络设备设备是内核设备。

所有不能由内核存取的 I/O 设备，特别是 IFX 设备如软盘或硬盘设备，直接由应用程序定义和安装。

内核设备通过一组逻辑 I/O 函数存取。称为 logio 函数，它们由内核和应用使用与外部发送和获取信息。应用可以调用 logio 函数，或直接存取驱动程序函数。read/write 和 put/get msg 函数通过 logio 或直接与硬件通讯。

一个内核设备的定义通过三步映象过程完成，使得操作系统感知的设备名与由 logio 方式识别的设备驱动程序相关。

所有 VRTXsa 内核设备定义为三层，系统设备、逻辑设备、物理设备。

1. 系统设备

系统设备是由内核感知的设备名。它被定义在文件 vrtxcnfg.def 中。每一设备由 VRTXsa 在环境变量中识别，Xconfig 一对变量映射到逻辑设备。

boot.env.dev.device.envname 内核使用名寻找逻辑设备名

boot.env.dev.device.value 赋值与环境变量一致的逻辑设备名

例如有环境变量 BOARD

boot.env.dev.board.envname: BOARD

boot.env.dev.board.value: board

所有映射到环境变量的系统设备需要列入 Xconfig 变量 boot.env.devices

2. 逻辑设备

一旦环境变量被定义并映射到逻辑设备，逻辑设备需要被定义并映射到物理设备。逻辑设备通常包含所有从目标机环境存取的设备。逻辑设备定义在定义目标机环境的文件<board>.def。

类似系统设备，每一逻辑设备由一对 Xconfig 变量定义。这些变量映射逻辑设备到一致的物理设备及物理设备驱动程序。

dev.device.name 赋值物理设备名

dev.device.method 定义 logio 方式。logio 方式是一个 C 结构，它定义设备的驱动程序（通过包含存取 logio 设备函数的调用结构）并建立设备驱动程序代码和数据之间的联系。

例如标准 mo860fads 系统的逻辑设备定义

dev.ether_1.name DEV_ETHER_1
dev.ether_1.method: logio_ether_1_method
dev.serial_1.name DEV_SERIAL_1
dev.serial_1.method: logio_serial_1_me thod
dev.timer_1.name: DEV_TIMER_1
dev.timer_1.method: logio_timer_1.method

所有映射到物理设备的逻辑设备需要列入 Xconfig 变量 board.devices

3. 物理设备

每一物理设备的 logio 结构定义在文件 devcnfg.c 中提供。devcnfg.c 由 Xconfig 在配置过程期间从 devcnfg.tpl 样板文件建立，它并不包含驱动程序本身，驱动程序存在于目标机的 BSP 库中。

物理设备中断

devcnfg.c 文件定义内核物理设备的 logio 结构。通常 logio 设备驱动程序方法定义处理设备中断的特定过程。

3.4 存储器配置

存储器映象的作用是定义存储器的区域，包括只读存储器如 flash，EPROM，ROM，包括读写存储器如 DRAM，SRAM。代码放置在只读存储器中，初始化数据从只读存储器拷贝到读写存储器中。数据区定位在读写存储器，堆、存储分区放置在数据区。

通常对于目标机系统定义四个区域

Code－定义代码的起始地址和存储器块长度
Workspace－定义 OS 工作空间的起始地址和存储器块长度
Boot Code－启动程序代码使用的 ROM 区
Boot Data－启动程序程序使用的 RAM 区

三个 Xconfig 变量确定每一个区域：

board.memory.name.address，board.memory.name.size，board.memory.name.type

这里 name 为 host 和 target，其中 host 为代码区，target 为 OS 工作空间

board.memory.name.address 以十六进制表示存储器块起始地址
board.memory.name.size 以十六进制表示存储器块大小
board.memory.name.type 提供五种常用的值 SHARED， COPYBACK，NONSERIALIZED, UNUSED_HOST, UNUSED_TARGET

配置实例

1. 建立 BOOT 映象

user.def
@include(boot.def)
@include(mo860fads.def)
@include(microtec.def)

xconfig user.def
可以获得 boot.o 和 boot.hex

2. 建立 OS 映象

user.def
@include(vrtxsa.def)
@include(mo860fads.def)
@include(microtec.def)

vrtxos.console: DEV_VCONSOLE
vrtx.user_task_count: 24

xconfig user.def
可以获得 os.o

3. 建立 OS 和应用单一映象

user.def
@include(vrtxsa.def)
@include(mo860fads.def)
@include(microtec.def)

vrtxos.console: DEV_VCONSOEL

vrtxos.objs.usr: vrtxdemo.o
sys.entry_point2: spawn_main

xconfig user.def
可以获得 os.o

4. 建立 BOOT，OS 和应用单一映象

user.def
@include(boot.def)
@include(vrtxsa.def)
@include(ace860qm.def)
@include(microtec.def)

vrtx.user_task_count: 40
vrtx.queue_count: 60

vrtxos.objs.usr: vrtxdemo.o
sys.entry_point2: spawn_main

make.what: final
make.rule: sutil
make.*.final.name: final
make.*.final.target: vrtxdemo.hex
make.*.final.deps: hextmp
make.*.final.code: 2000000
make.*.final.hextmp.rule: cathex

make.*.hextmp.name: hextmp
make.*.hextmp.target: hextmp.hex
make.*.hextmp.deps: boot,vrtxos
make.*.hextmp.boot.rule: link_hex

make.*.boot.code: 2000000
make.*.boot.data: 10000
make.*.boot.target: boottmp.hex
make.*.hextmp.vrtxos.rule: link_hex
make.*.vrtxos.code: 2020000
make.*.vrtxos.data: 20000
make.*.vrtxos.target: ostmp.hex

boot.methods: go
boot.env.boot_order: go
boot.method.go.orderparam: 2020000

xconfig user.def
可以获得 boot.hex

应用开发篇之四：开发实例

第一章　设计实现

1.1 需求分析

- 有 10 个人（开始都在房间外）和 2 个房间（每个房间最多容纳 2 人）
- 每个人自创建起 20ms 时以 30%的机率选择是否进入房间。
- 进入房间的人需呆够 1 秒钟。此后，每 300ms 人员以 20%的机率出房间。
- 出了房间的人再次以每 20ms 决定是否进入房间；留在房内的人重复以 300ms 决定是否出房间。
- 当人员在房内呆到 2.5s 时，发警告信息。
- 当人员在房内呆到 4s 时强制他出房间，再次以每 20ms 决定是否进入房间。
- 每隔 500ms 显示一次人员/状态信息。
- 系统提供一个硬件定时器，每 20ms 触发一次中断事件。

每个人的状态变迁图如下：

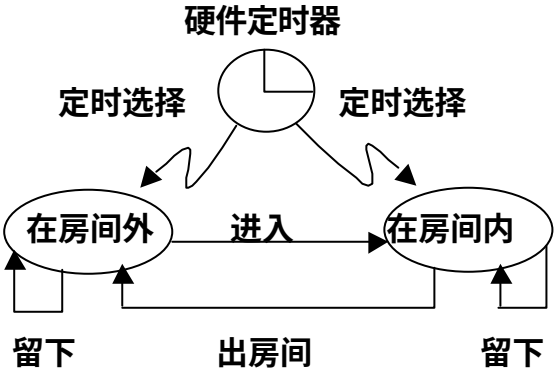
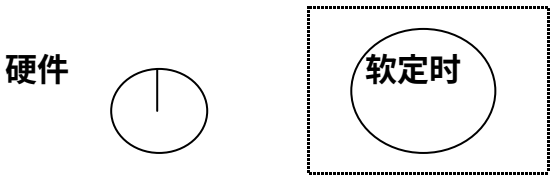


图 4-1

1.2 系统设计

1.2.1 数据流分析



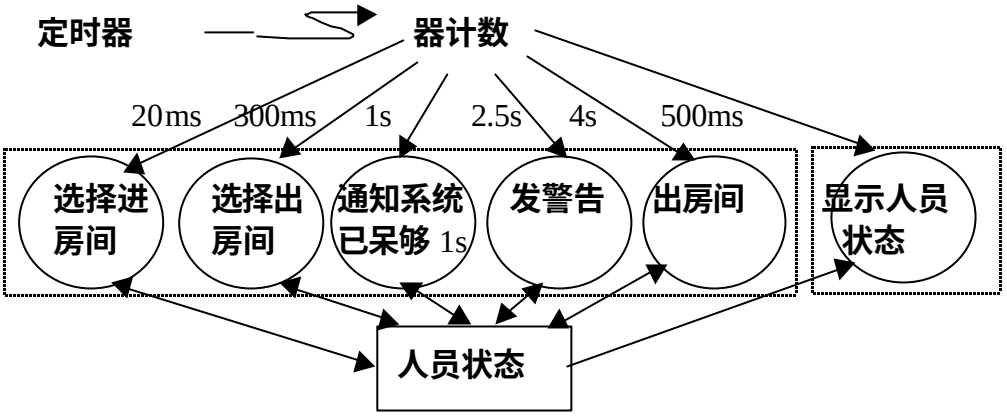


图 4-2 人员进出房间系统的数据流图

系统由定时器定时，在不同时间段根据相应人员的状态作出不同的状态变迁，定时显示人员状态。

1.2.2. 任务划分

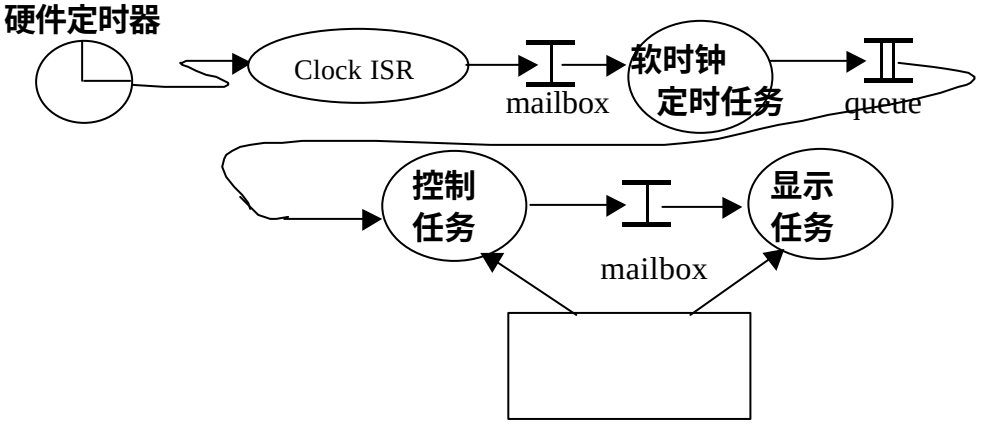
由于系统只提供一个硬件定时器，要完成对 6 个不同时间段的控制，故采用软定时器的方法。软定时器的计数基于由硬件定时器触发的中断事件。考虑到实时系统需及时响应中断，中断处理程序应尽量简单。因此，定时器中断处理程序不直接对软定时器计数，而是通过与任务同步，由任务完成。按照任务划分的原则，将完成软定时器计数的功能模块作为一个低优先级的独立任务。完成显示功能的模块也作为一个独立的任务。为了能保证定时显示而不被其它任务强占，使显示任务具有最高优先级。对人员动作的控制由功能内聚性作为一个控制任务。图二中每一个虚框图对应一个独立的任务。

1.2.3. 任务间的接口

软时钟计数任务通过队列与控制任务通信；控制任务通过邮箱和显示任务的同步；软时钟计数任务通过邮箱与中断处理程序的同步。

1.2.4 系统结构图

在经过数据流分析、任务划分和确定任务间的接口以后，系统结构图为：



人员状态

图 4-3 任务的转换图

1.3 设计实现

1．人员出入房间系统共划分为 5 个任务，它们是 main，master，task1，task3，task4。一个定时器中断处理程序 io_isr，由硬件定时器每 20ms 触发一次。 任务间的同步、通信分别通过邮箱 dis_box 和 队列 out_queue 实现；定时器中断处理程序调用 mytimer，通过邮箱 mailbox_timer 实现与任务的同步。系统由 6 个软定时器完成对不同阶段的控制。6 个定时器的作用如表：

定时器	定时值	作　　用
Timer0	20ms	用于人员每 20ms 决定是否进入房间的定时器
Timer1	300ms	用于从进入房间时起在房内呆够 1s 但不到 4s 的人员以每 300ms 决定是否出房间的定时器
Timer2	1s	用于从进入房间时起人员在房内呆够 1s 的定时器
Timer3	2.5s	用于记录人员从进入房间时起在房内呆够 2.5s 的定时器
Timer4	4s	用于记录人员从进入房间时起在房内呆够 4s 的定时器
Timer5	500ms	用于定时激活显示人员/状态的任务

表 4-1

在这个系统中软定时器由 set_timer(timeout，person_id，timer_id)激活。由于软定时器是通过计数任务实现，在计数值减为 0 后将软定时器占用的内存空间释放，所以要实现重复定时的方式需要再次调用 set_timer 激活软定时器。

下面介绍一下 5 个任务的主要功能：

- main 任务是一个初始化任务，主要功能为：
  - 在应用内存中分配一个 8096 字节的堆；
  - 为时间链头、链尾及时间链当前指针分配空间；
  - 初始化时间链和人员/状态的数据结构；
  - 创建一个大小为 100 字节的队列 out_queue；
  - 创建优先级不等的四个任务；
  - 启动软定时器 5；
  - 删除自身。

- master 任务是一个显示人员/状态信息的任务，具有最高优先级。它的主要功能为：
  - 等待邮箱 dis_box 的消息；
  - 等到消息后做：
    显示在房间外的人员/状态信息；
    显示在 1 号房间的人员/状态信息；

显示在 2 号房间的人员/状态信息；
启动软定时器 5。
备注：dis_box 用于 master task 与 task1 的同步

- task1 任务是系统的关键任务，根据人员所处的不同状态（有不同的定时器在工作）控制人员的动作，主要功能为：
  - 等待队列 out_queue 的消息;
  - 等到消息后根据不同的软定时器号做：
    软定时器 0：
以一定的机率和房间数决定人员是否可以进入房间，进入房间的人员状态改变为 1 或 3，并设置软定时器 2、3、4；否则重设软定时器 0。
软定时器 2：
当人员状态处于 1 或 3 时，即是在 1 号房间或 2 号房间内时，状态相应转换为 2 或 4；启动定时器 1。
软定时器 1：
当人员状态不处于门外时（即非 0 状态时）以一定的机率出房间。对并出房间的人员关闭软定时器 3、4，重设软定时器 0；对没出房间的人员重设软定时器 1。
软定时器 3：
当人员状态不处于门外时，显示警告信息。
软定时器 4：
当人员状态处于 2 或 4 时，令人员必出房间，关闭软定时器 1，重设软定时器 0。
软定时器 5：
向邮箱 dis_box 发消息。
备注：out_queue 用于 task1 与 task3 的同步

- task3 任务是一个软定时器的计数任务，具有最低优先级。它的主要功能为：
  - 等待邮箱 mailbox_timer 的消息；
  - 等到消息后做：
    对时间链上各结构的超时值减 1；
    对超时值为 0 的结构释放其内存空间，并向队列 out_queue 发送相应的人员/定时器号信息。
  备注：maibox_timer 用于 task3 与定时器中断处理程序 ISR 的同步。

- task4 任务是一个创建人员/状态数据结构并启动软定时器的任务，主要功能为：
  - 创建 10 个人员/状态数据结构，初始化状态为 0；
  - 启动每个人的软定时器 0；
  - 删除自己。

2. **人员状态转换的时序图：**

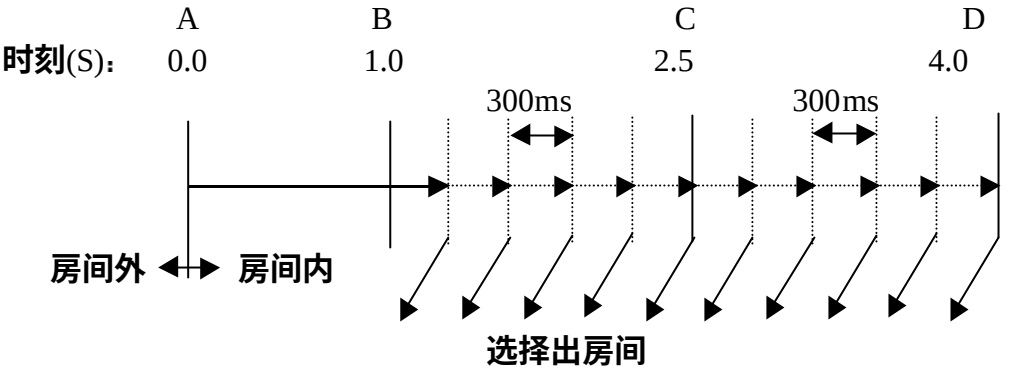


图 4-4

说明：图中“——►”表示人员必经阶段；“ ┈┈►”表示人员可能经过的阶段；
“↙”表示人员在该时刻选择出房间。

3．**人员状态转换图：**

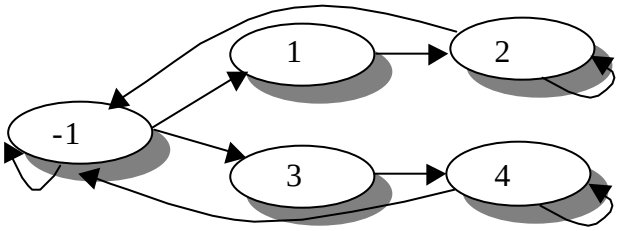


图 4-5

说明： person_state = -1 **人员在房间外**
person_state = 1 **人员进入** 1 **号房间，但还未呆够** 1s。
person_state = 3 **人员进入** 2 **号房间，但还未呆够** 1s。
person_state = 2 **人员在** 1 **号房间内已呆够** 1s。
person_state = 4 **人员在** 2 **号房间内已呆够** 1s。

4．**任务流程图：**

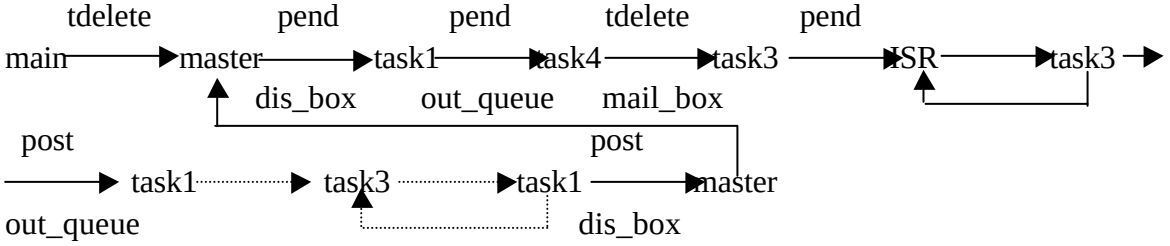


图 4-6

说明：“——►”上方表示引起任务切换的原因，“ ——►”下方表示所等待的资源。

1.4 编程

1. **应用程序** Intdemon.c

```
/*
 * Room Demo
 *
 */


#include <stdio.h>
#include <vrtxvisi.h>
#include <vrtxil.h>
#include <stdlib.h>
#include <errno.h>
#include <syskind.h>
#include <compiler.h> /* turns on M68K flag for 68K targets */
#include <mriext.h>
#include "68360.h"

#define HEAP_START 0x500000

int out_queue;
int group_id;
int _randx = 0x87654321;                /* random seed */

char *mailbox_timer;
char *dis_box;
typedef struct timer_que{
struct timer_que *next;
int timeout;
int pid_tid;
}T_QUE;

T_QUE *timer_que_boot,*timer_que_current,*block;
int pid=1,hid;
unsigned long bsize=(sizeof(T_QUE));

typedef struct {
    int person_id;
    int person_state;
  } PERSON;
PERSON person[10];
```

```
int  rand_100(int i)
{
  if(rand()%100<i)
  return (1);
  else return (0);
 }

/*
 * report error function
 *
 */

void report(int err)
{
 /* if you get an error, execute XRAY command UP to see from where */
  asm(" illegal");
}

struct CPM_REGISTER *CpmRegister ;
struct SIM_REGISTER *SimRegister ;
struct RISC_Timer   *r_timer ;

LONG GetMbar(void) ;
LONG GetVbr(void) ;
void Setup68360(void) ;
void setcmpvect(LONG lIntMask, BYTE lIntNum, void (* func)() ) ;
void risc_tm0(void) ;
void risc_tm0_body(void) ;

void task1(void) ;
void task3(void) ;
void master(void) ;
void task4(void) ;

/* timer ISR */
void risc_tm0(void)
{
   asm(" XREF _v90k_interrupt_enter ");
   asm(" jsr _v90k_interrupt_enter ");

   asm(" movem.l d0/d1/d2/a0/a1/a2,-(SP) ");

   asm(" jsr _risc_tm0_body ");
```

```
   asm(" movem.l (SP)+,d0/d1/d2/a0/a1/a2 ");

   asm(" XREF _v90k_interrupt_exit ");
   asm(" jmp _v90k_interrupt_exit ");
}
void risc_tm0_body(void)
{
   int err ;

   sc_post((VRTX_MSG VRTX_FAR *)&mailbox_timer,(VRTX_MSG)1,&err);
   if(err) printf("\n\rpost mailbox error. err=%d",err);

   CpmRegister->CISR |= 0x00020000 ;
   CpmRegister->RTER |= 0x0001 ;
}

LONG GetMbar(void)
{
    asm(" move.w #7,d0");    /* CPU space func code to d0   */
    asm(" movec.l d0,sfc");  /* load SFC for CPU space      */
    asm(" lea.l $3ff00,a0"); /* A0 points to MBAR           */
    asm(" moves.l (a0),d0"); /* get MBAR                    */
    asm(" andi.l #$ffffe000,d0") ;
}
LONG GetVbr(void)
{
    asm(" movec.l vbr,d0 ");
}

void Setup68360(void)
{
   CpmRegister = ( struct CPM_REGISTER * )( GetMbar() + CPM_BASE );
   SimRegister = ( struct SIM_REGISTER * )( GetMbar() + SIM_BASE );
   r_timer = (struct RISC_Timer *)(GetMbar() + Timer_BASE) ;
}

void setcmpvect( LONG lIntMask, BYTE lIntNum, void (* func)() )
{
    LONG *pVec;

/* directly set the interrupt vector by modifying the EVT */
    pVec = ( LONG * )( GetVbr() + ((CpmRegister->CICR & 0x000000e0) + lIntNum ) * 4 );
    *pVec = ( LONG )func;
```

```
    CpmRegister->CIPR |= lIntMask;
    CpmRegister->CISR |= lIntMask;
    CpmRegister->CIMR |= lIntMask;
}

/*
*  send timer message to queue
*  This message consists of p_id,t_id and Timeout value
*/

void set_timer(int timeout1,int p_id,int t_id)
{
 int err;
 int pid_tid;
 T_QUE * tmp;
     pid_tid=p_id*10+t_id;
     sc_lock();

     tmp=(T_QUE *)sc_halloc(hid,bsize,&err);
     if(err) printf("\n\rset_timer err=%d",err);

     tmp->timeout=timeout1;
     tmp->pid_tid=pid_tid;
     tmp->next=block;
     timer_que_current->next=tmp;
     timer_que_current=tmp;

     sc_unlock();
  }

void close_timer(int p_id,int t_id)
{
  int err;
  T_QUE * tmp,*tmp1;
  int pid_tid;
  sc_lock();

  pid_tid=10*p_id+t_id;
  tmp1=timer_que_boot;
  tmp=tmp1->next;

 while(tmp!=block&&tmp->pid_tid!=pid_tid)
  {
   tmp1=tmp;tmp=tmp->next;
```

```
   }

if(tmp==block)
    printf("\n\r...timer pid_tid=%d is closed",pid_tid);
else
{
 if(tmp->pid_tid==pid_tid&&tmp->next!=block) {
    tmp1->next=tmp->next;
    printf("\n\r...del pid_tid=%d",pid_tid);
    }
 if(tmp->pid_tid==pid_tid&&tmp->next==block)
   {  timer_que_current=tmp1;
     timer_que_current->next=block;    }
 sc_hfree(hid,(char *)tmp,&err);
  }
  sc_unlock();
}


void task1(void)
 {
   int err, i=0;
   int room1_num=0,room2_num=0;
   char * msg;
   unsigned pid_tid;
   int pid,tid;

   printf("\n\rtask1 (Inserter) active.");
   msg=sc_qaccept(out_queue,&err);
   i=0;
   for (;;)
      {
    pid_tid = (unsigned)sc_qpend(out_queue, 0, &err);
    if (err) report(err);
    tid=pid_tid%10;
    pid=pid_tid/10;
    printf("\n\r(Pid,Tid)=(%d , %d) timer is over!",pid,tid);
    switch(tid)
    {
     case 0:
     if(rand_100(30))
     {
     if(rand_100(50)&&room1_num<3)
      {
```

```
room1_num++;
person[pid].person_state=1;
printf("\n\r person_id=%d --->room1",person[pid].person_id);
set_timer(20,pid,2);
set_timer(45,pid,3);
set_timer(60,pid,4);
     }
else if(room2_num<3)
     {
     room2_num++;
     person[pid].person_state=3;
     printf("\n\r person_id=%d --->room2",person[pid].person_id);
     set_timer(20,pid,2);
     set_timer(45,pid,3);
     set_timer(60,pid,4);
     }
    else
    set_timer(4,pid,0);
   }
else
set_timer(4,pid,0);
break;
case 3:
if(person[pid].person_state!=0)
{
printf("\n\r----------------------------------------");
printf("\n\r*****!!!! warning person=%d  !!!",pid);
printf("\n\r----------------------------------------");
}
break;
case 2:
if(person[pid].person_state%2==1)
 {
  person[pid].person_state++;
  set_timer(8,pid,1);
  }
break;
case 1:
if(rand_100(20)&&person[pid].person_state!=0)
{
printf("\n\r person_id=%d ---->out",person[pid].person_id);
if(person[pid].person_state<3)
room1_num--;
else room2_num--;
```

```
      person[pid].person_state=0;
      sc_lock();
      close_timer(pid,3);
      close_timer(pid,4);
      set_timer(4,pid,0);
      sc_unlock();
      }
      else
      set_timer(8,pid,1);
      break;
      case 4:
      if(person[pid].person_state==2||
         person[pid].person_state==4)
      {
      if(person[pid].person_state==2)
      room1_num--;
      else room2_num--;
      person[pid].person_state=0;
      sc_lock();
      printf("\n\r person_id=%d ---->out",person[pid].person_id);
      close_timer(pid,1);
      set_timer(4,pid,0);
          }
      sc_unlock();
      break;
      case 5:
      msg=(char *)1l;
      sc_post(&dis_box,msg,&err);
      break;

      }

       }
  }


void master(void)
{ int err, i;
  T_QUE *tmp;

  printf("\n\rmaster task (Inserter) active.");

  for (;;)
      {
```

```
        i=(int)sc_pend((VRTX_MSG VRTX_FAR *)&dis_box,0l,&err);
        printf("\n\r------------------------------------------------>");

        printf("\n\r OUtDoor:");
        for(i=0;i<10;i++)
        if(person[i].person_state==0)
        printf("(%d,%d),",person[i].person_id,person[i].person_state);

        printf("\n\r Room1:");
        for(i=0;i<10;i++)
        if((person[i].person_state==1)||(person[i].person_state==2))
        printf("(%d,%d),",person[i].person_id,person[i].person_state);

        printf("\n\r Room2:");
        for(i=0;i<10;i++)
        if((person[i].person_state==3)||(person[i].person_state==4))
        printf("(%d,%d),",person[i].person_id,person[i].person_state);

        printf("\n\r------------------------------------------------>");
        set_timer(10,0,5);
    }
}


/*
 * task3 - This task generates random numbers
 *         It runs on medium priority and sends random
 *         numbers to a queue
 */
void task3(void)
{
  int err, i,j;
  T_QUE *tmp,*tmp1;
  char * msg_timer;
  VRTX_MSG msg ;

  printf("\n\rtask3 (Generator) active.");

  for(;;)
    {
   msg = sc_pend((VRTX_MSG VRTX_FAR *)&mailbox_timer , 0 , &err) ;

   tmp1=timer_que_boot;
   tmp=tmp1->next;
```

```
   while(tmp!=block) {
       (tmp->timeout)--;
       if((tmp->timeout)==0)
        {
          msg_timer=(char *)(tmp->pid_tid);
          tmp1->next=tmp->next;
          if(tmp==timer_que_current) timer_que_current=tmp1;
          if(timer_que_boot->next==tmp) timer_que_boot->next=tmp->next;
          sc_hfree(hid,(char *)tmp,&err);
          if(err) printf("\n\r free buf error. err=%d",err);
          tmp=tmp1->next;
          sc_qpost(out_queue,msg_timer,&err);

          }
         else { tmp1=tmp;
            tmp=tmp->next;
              }
        }
     }
}


/*This task  generate 10 persons and then kill itself */

void task4()
{
 int err,i;
 unsigned pid_tid;

 printf("\n\r task4 active.....");

 for(i=0;i<10;i++)
 {
  sc_delay(3);
  person[i].person_id=i;
  person[i].person_state=0;
  pid_tid=10*person[i].person_id+0;
  printf("\n\r-------------------------------");
  printf("\n\r(%d,%d) is created ",person[i].person_id,
              person[i].person_state);
  set_timer(4,person[i].person_id,0);
  }
 sc_tdelete(0,0,&err);
}
```

```
/*
 * Initial task (main)
 *
 */
void main( void )
{

  int   err, i,j;
  struct CICR *cicr_p ;
  struct RTMR *rtmr_p ;

  printf("\n\rmain initializing.");

/* get the base address of CPM and SIM */
   Setup68360() ;

/*create timer queue boot pointer */
  hid=sc_hcreate((char *)HEAP_START,8096,4,&err);
  if(err) printf("\n\rheap err=%d",err);

  timer_que_boot=(T_QUE *)sc_halloc(hid,bsize,&err);
  if(err) printf("\n\rgetting a block failure. err=%d",err);

  block=(T_QUE *)sc_halloc(hid,bsize,&err);
  if(err) printf("\n\rgetting a block failuer. err=%d",err);

  timer_que_boot->timeout=-1;
  timer_que_boot->next=block;
  timer_que_boot->pid_tid=-1;
  timer_que_current=timer_que_boot;

for (i=0; i <10; i++)
   {
    person[i].person_id=-1;
    person[i].person_state=-1;
    }
for (i=0; i <10; i++)
    { printf("\n\r person_id=%d person_state=%d ",
      person[i].person_id,person[i].person_state);
    }
```

```
  /* create system objects */
  out_queue = sc_qcreate(4, 100, &err);  /* create queue */
  if (err) report(err);

  /* create tasks */
  sc_tcreate(task1, 1, 100, &err);     /* receive msg from outqueue and determinate what
                       state will changed to,which type
                       of timer(s) are to be opened */
  if (err) report(err);
  sc_tcreate(master, 2, 60, &err);      /*display persons state*/
  if (err) report(err);
  sc_tcreate(task3, 3, 120, &err);      /*handler msg of timer queue ,*/
  if (err) report(err);
  sc_tcreate(task4, 4, 100, &err);       /*create 10 persons and open their 5ms timer
immediately*/
  if (err) report(err);

/*   set the vector of RISC_Timer0 , every a period of time ,
     it send a message to the mailbox which pend the disp task   */
   setcmpvect(0x00020000 , 0x11 , risc_tm0) ;

/*   setup the necessary registers of RISC_Timer0 and enable the interrupt */
/*
   SimRegister->PEPAR  |= 0x0080 ;
*/
   CpmRegister->RCCR   |= 0x3F00 ;
   r_timer->TM_BASE     = 0x0000 ;
   r_timer->TM_cnt      = 0x0000 ;
   CpmRegister->RTER    = 0xFFFF ;
   rtmr_p = (struct RTMR *)&CpmRegister->RTMR ;
   rtmr_p->T01 = 1 ;/*   = 0x0001 ;*/
   r_timer->TM_cmd      = 0xc00000ff ;
   CpmRegister->RCCR   |= 0x8000 ;
   CpmRegister->CR      = 0x0851 ;


  /* delete self to start system */
  set_timer(10,0,5);
  sc_accept(&dis_box,&err);
  sc_tdelete(0,0,&err);
  if (err) report(err);
}
```

```
/*
 * create the main task
 * called from VRTXsa during init
 */
void spawn_main()
{
  int err;
  sc_tcreate(main, 8, 1, &err);
}
```

1.5 配置文件 makefile.demo 和 do_make 文件

1. **执行文件** do_make

```
    usage()
    {
    echo "Usage:   do_make <target>"
    echo "<target>: 68000, 68010, 68020, 68030, 68040, 68060, cpu32"
    echo "          ppc505, ppc603, ppc603e, ppc604, ppc604e, ppc821, ppc860"
    echo "          sun4"
    echo "Example: do_make 68040"
    exit 1
    }

    TARGET=$1
    INCLUDES=$SPECTRA/target/include

    # tools to build the application program
    if [ "$TARGET" = 68000 ]; then
         PREFIX=68k
         FAMILY=68000
         FORMAT=microtec
          CFLAGS="-c -p68000 -g -nOR -Za2 -U_SIZE_T -DM68K -D__READY_EXTENSIONS__ -
J$INCLUDES"

    elif [ "$TARGET" = 68010 ]; then
         PREFIX=68k
         FAMILY=68010
         FORMAT=microtec
          CFLAGS="-c -p68010 -g -nOR -Za2 -U_SIZE_T -DM68K -D__READY_EXTENSIONS__ -
J$INCLUDES"
        # EXTRA_LIBS=c:/ramadata/tims/libvista.lib
```

```
    elif [ "$TARGET" = 68020 ]; then
         PREFIX=68k
         FAMILY=68020
         FORMAT=microtec
         CFLAGS="-c -p68020 -g -nOR -Za2 -c -U_SIZE_T -DM68K -D__READY_EXTENSIONS__ -
J$INCLUDES"

    elif [ "$TARGET" = 68030 ]; then
         PREFIX=68k
         FAMILY=68030
         FORMAT=microtec
          CFLAGS="-c -p68030 -g -nOR -Za2 -U_SIZE_T -DM68K -D__READY_EXTENSIONS__ -
J$INCLUDES"

    elif [ "$TARGET" = 68040 ]; then
         PREFIX=68k
         FAMILY=68040
         FORMAT=microtec
          CFLAGS="-c -p68040 -g -nOR -Za2 -U_SIZE_T -DM68K -D__READY_EXTENSIONS__ -
J$INCLUDES"

    elif [ "$TARGET" = 68060 ]; then
         PREFIX=68k
         FAMILY=68060
         FORMAT=microtec
          CFLAGS="-c -p68060 -g -nOR -Za2 -U_SIZE_T -DM68K -D__READY_EXTENSIONS__ -
J$INCLUDES"

    elif [ "$TARGET" = cpu32 ]; then
         PREFIX=68k
         FAMILY=cpu32
         FORMAT=microtec
          CFLAGS="-c -pcpu32 -g -nOR -Za2 -U_SIZE_T -DM68K -D__READY_EXTENSIONS__ -
J$INCLUDES"

    elif [ "$TARGET" = ppc505 ]; then
         PREFIX=ppc
         FAMILY=ppc5xx
         FORMAT=elf
         CFLAGS="-c -p505 -g -Xp -U_SIZE_T -DPPC -D__READY_EXTENSIONS__ -J$INCLUDES"

    elif [ "$TARGET" = ppc603 -o "$TARGET" = ppc603e ]; then
         PREFIX=ppc
```

```
        FAMILY=ppc603
        FORMAT=elf
        EXTRA_LIBS=$SPECTRA/target/lib/ppc603/elf/elf.lib
        CFLAGS="-c -p603 -g -Xp -U_SIZE_T -DPPC -D__READY_EXTENSIONS__ -J$INCLUDES"

    elif [ "$TARGET" = ppc604 -o "$TARGET" = ppc604e ]; then
        PREFIX=ppc
        FAMILY=ppc603
        FORMAT=elf
        EXTRA_LIBS=$SPECTRA/target/lib/ppc603/elf/elf.lib
        CFLAGS="-c -p604 -g -Xp -U_SIZE_T -DPPC -D__READY_EXTENSIONS__ -J$INCLUDES"

    elif [ "$TARGET" = ppc860 ]; then
        PREFIX=ppc
        FAMILY=ppc860
        FORMAT=elf
        EXTRA_LIBS=$SPECTRA/target/lib/ppc860/elf/elf.lib
        CFLAGS="-c -p860 -g -Xp -U_SIZE_T -DPPC -D__READY_EXTENSIONS__ -J$INCLUDES"

    elif [ "$TARGET" = ppc821 ]; then
        PREFIX=ppc
        FAMILY=ppc860
        FORMAT=elf
        EXTRA_LIBS=$SPECTRA/target/lib/ppc860/elf/elf.lib
        CFLAGS="-c -p821 -g -Xp -U_SIZE_T -DPPC -D__READY_EXTENSIONS__ -J$INCLUDES"

    elif [ "$TARGET" = sun4 ]; then
        PREFIX=sp
        FAMILY=sun4
        FORMAT=coff
        CFLAGS="-g -c -U_SIZE_T -DSUN4 -D__READY_EXTENSIONS__ -J$INCLUDES"

    elif [ "$TARGET" = clean ]; then
        make -f makefile.demo clean
        exit 0

    else
        usage
    fi

    make -f makefile.demo PREFIX=$PREFIX FAMILY=$FAMILY FORMAT=$FORMAT CFLAGS="$CFLAGS"
EXTRA_LIBS=$EXTRA_LIBS
```

2.**配置文件** makefile.demo

```
PROG = intdemon
SRC  = .
OBJS =  $(PROG).o

CC=mcc$(PREFIX)
AS=asm$(PREFIX)
LD=lnk$(PREFIX)
SPECTRA_LIBS=$(SPECTRA)/target/lib/$(FAMILY)/$(FORMAT)

# To generate $(PROG) to run in the user mode add following library
#        $(SPECTRA_LIBS)/vrtxil.lib
# to the  LIBS list of libraries below:

LIBS =  $(SPECTRA_LIBS)/rtnofp.lib $(SPECTRA_LIBS)/rt.lib \
    $(SPECTRA_LIBS)/cpu.lib \
    $(EXTRA_LIBS)

# The following definition lists which object modules make up
# the relocatable application modules.

#-----------------------------------------------------------------------#
#
# Create relocatable $(PROG) module and symbol table from objects
#
# ----------------------------------------------------------------------#

all:    $(PROG)

$(PROG) : $(OBJS)
    @echo "for i in $(OBJS) $(LIBS); do" > temp.xxx
    @echo 'echo LOAD $$i; done' >> temp.xxx
    @sh temp.xxx > temp.xxxx
    @if [ $(LD) = "lnk68k" ] ; then \
        echo "$(LD) -o  $@ -r -m  > $(PROG).map $(OBJS) $(LIBS)" ;\
        $(LD) -o  $@.tmp -r -m  > $(PROG).map -c temp.xxxx ;\
        mv  $@.tmp $@.x ;\
    elif [ $(LD) = "lnkppc" -o $(LD) = "lnksp" ] ; then \
        echo "$(LD) -o  $@ -i -m  > $(PROG).map $(OBJS) $(LIBS)" ;\
        $(LD) -o  $@.tmp -i -m  > $(PROG).map -c temp.xxxx ;\
        mv  $@.tmp $@.x ;\
    fi
    @rm -f temp.xxx temp.xxxx
```

```
#
# Compile/assemble standard source files
# ------------------------------------

%.o: %.c
    @echo "$(CC) $(CFLAGS) -o $@ $(SRC)/$<"
    @echo "$(CFLAGS) -o $@ $(SRC)/$<" > temp.xxx
    @$(CC) -dtemp.xxx
    @rm temp.xxx

clean:
    rm -f $(OBJS) $(PROG) $(PROG).map

# ---- End of makefile ----
```

1. 6 如何生成可调试的文件 intdemo.x

当有了文件 do_make **和** makefile.demo **以后. 用户进入** spectra **的** xconfig(Ksh) **窗口，然后用以下命令生成可加载的文件** intdemo.x

sh do_make 68010

VRTX 培训的实验说明

一.实验目录说明

所有的实验都在 :\360vrtx 中. 其中具体的目录说明如下:

:\360vrtx\task_m	任务管理实验
:\360vrtx\memory	内存管理实验
:\360vrtx\mbox	邮箱系统实验
:\360vrtx\queues	队列实验
:\360vrtx\event	事件组实验
:\360vrtx\sem	信号量实验
:\360vrtx\mutexes	互斥量实验
:\360vrtx\interrupt	中断处理实验
:\360vrtx\timer	实时时钟实验
:\360vrtx\char	字符 I/O 实验
:\360vrtx\os	操作系统

二. Spectra 的操作使用

1. 启动 vserver

进入 NT 的 Spectra 用户.选择 Start->Program->Spectra Server->Vserver

2. 启动 Xray Dubugger

选择 Start->Program->Spectra Client ->XRAY Debug

3. 从 Manage->connect->Available Connection 中选择一个目标板(cp151)
4. 双击板名(CP151). 当 Code 窗口左上角黑框变为淡兰,则目标机连到 Xray Debug
5. 下装 OS

(a). 在 Manage 窗口下, 按 Files 按钮, 在 Files 下,选择 Load 中的 Load File To Target .

(b). 选择:\360vrtx\os\os.o , 并按 OK.

6. 用同样的方法下装应用
7. 启动 VCONSOLE

选择 Start->Program->Spectra Client ->Vconsole. 在 Vconsole 窗口中输入 cp151 回车

8. 启动 VRTX 多任务

在 Files 下,选择 Control ->Start Process Running(GO).

9. 创建多任务

在 Files 下,选择 Control ->Spawn Thread from main()

三. 怎样 Build 一个应用

1. 启动 Xconfig

选择 Start->Program->Spectra Client ->Xconfig(Ksh)

2. 在 Xconfig(Ksh)窗口中, 进入应用程序所在目录. 然后输入如下命令:

sh do_make 68010

即可生成可下载的文件 *.x