东南大学

硕士学位论文

一种嵌入式实时操作系统微内核模型的设计和实现

姓名：孙柯

申请学位级别：硕士

专业：软件工程

指导教师：吴介一;罗莹

20050610

内容摘要

题名：一种嵌入式实时操作系统微内核模型的设计和实现

硕士研究生姓名：孙柯

导师姓名：吴介一（教授） 罗莹（高工）

学校名称:东南大学

论文摘要:

随着电子计算机技术的飞速发展，嵌入式系统及基于其上的实时操作系统已经得到了非常广泛的发展和应用。本论文设计和实现了基于并发数据流驱动模式的嵌入式实时操作系统微内核模型（micro-DDRTOS），主要应用于通信设备、自动控制设备、测量电子设备和医疗设备等需要大量重复数据处理的应用环境，其实现基础为μC/OS-II 源码开放的实时操作系统，对其任务调度和管理、内存管理、中断管理和时间管理等方面做出重新设计和移植，采用了抢占式动态优先级和固定优先级相结合的调度算法，并在此基础之上加入 TCP/IP 网络协议栈、GUI 图形用户界面、文件系统。该微内核基于 ARM 硬件体系结构，利用 ARM 仿真集成开发环境和仿真平台，设计出以稳定性、实时性和并行多任务性为特点的嵌入式实时操作系统模型，形成一个比较完整的最小应用实时嵌入式系统。

关键词:嵌入式系统、实时操作系统、并发数据流模型、GUI 图形用户界面、嵌入式 TCP/IP 协议栈、嵌入式文件系统、ARM

Title: DESIGN AND IMPLEMENTATION OF AN EMBEDED REAL-TIME DATA DRIFT MICRO-KERNEL MODULE

Author: Sun Ke

Supervised by Professor Wu Jie-yi and Senior engineer Luo Ying

Abstract:

A multi-data drift embedded real-time micro operating system (micro-DDRTOS) has been designed and evaluated in this thesis, and TCP/IP stack, GUI user interface and a file system also have been added into this micro real-time OS. Micro-DDRTOS is mainly used in controlling system, telecom devices, and electronic measure devices, which need to handle large of data signal. Micro-DDROTS bases on μC/OS-II, and the tasks management, time management, interrupt management and memory management have been re-designed for data-draft module. This embedded system bases on ARM architecture hardware and design and evaluation by ADS and Armulator tools.

Keywords:

Embedded system, real-time operating system, multi-data draft module, GUI user interface, TCP/IP stack, File system, ARM

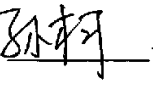
东南大学学位论文独创性声明

本人声明所呈交的学位论文是我个人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得东南大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示了谢意。

研究生签名： 日　期：2005/06/10

东南大学学位论文使用授权声明

东南大学、中国科学技术信息研究所、国家图书馆有权保留本人所送交学位论文的复印件和电子文档，可以采用影印、缩印或其他复制手段保存论文。本人电子文档的内容和纸质论文的内容相一致。除在保密期内的保密论文外，允许论文被查阅和借阅，可以公布（包括刊登）论文的全部或部分内容。论文的公布（包括刊登）授权东南大学研究生院办理。

研究生签名：  导师签名：_____日　期：2005/06/10

第一章 前言

随着电子计算机技术的飞速发展，嵌入式系统及基于其上的实时操作系统已经得到了非常广泛的发展和应用。嵌入式系统一般用于实时性能要求很高的环境，针对于不同的应用，选择合适的系统构架或者说内核结构，能够提高嵌入式系统的质量和降低系统的成本。这里针对于自动控制系统、数据交换设备、电子测量设备和医疗设备等应用提出一种基于并发数据流(multi-data-drift)架构的实时操作系统模型，即面向数据驱动的可重用的并行处理的实时系统微内核模型(data drift micro real-time operating system kernel,简称 micro-DDRTOS)。并发数据流驱动模型的最大特点是，系统需要同时处理大量的实时数据，这些数据有着相同的处理方法，进入系统的待处理数据，是所有任务处理的启动和运行触发点，如果没有数据需要处理，这些数据处理的任务节点在后台处于休眠状态，各个应用任务模块之间以数据信号流作为唯一的接口界面。Micro-DDRTOS 的数据流由数据通道(data channels)统一管理，从某方面来看，数据通道管理相当于实现了实时内核的数据处理任务调度管理的功能。数据通道通过不同优先级别的数据信号队列实现了多任务数据处理的抢占式动态优先级调度算法，Micro-DDRTOS 主要应用于输入数据量大，数据处理模式比较单一，要求速度高的应用环境中，例如控制系统的数据采集模块和回路控制模块、用于数据通信的数据交换设备和信号分析设备、逻辑分析仪等测量设备。基于并发数据流驱动的内核构架，实现了对于数据处理的并行流水线构架，数据处理并行线之间互不干扰，其在任务调度和管理、内存管理、和时间管理等各个方面都和一般的实时内核有所不同。

Micro-DDRTOS 并发数据流驱动模型如图 1.1 所示。

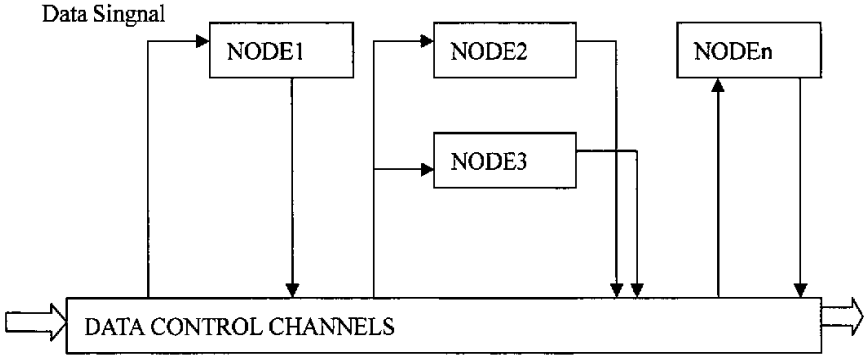


图 1.1micro-DDROTS 数据流驱动模型

这里 NODE 为数据处理任务节点，每个数据处理任务节点(NODE)在没有数据信号(Data Signal)进入的时候，在后台休眠不运行，当有待处理的数据信号进入时，任务调入前台处理，每个进入系统的数据在数据通道（Data Control Channel）的管理和调度下，通过若干数据处理任务节点完成处理，可见每个任务节点之间由数据信号流将它们连接起来，它们之间没有其它的接口界面。这样的最大优点是每个处理过程之间是互相独立的，非常便于应用程序的模块化划分，而且模块之间的耦合性非常小。这种架构方式具有结构简单、系统性能高、运行稳定的特点，而且非常便于组件化，和组件的重用。而且基于这种构架的处理模块完全可以组合使用，用于集散或者分布式的应用之中。

一般系统内核的多线程和多任务的实现是通过时间片的划分和管理来实现的，micro-DDRTOS

的多任务并行处理的实现是通过数据通道的数据流管理来实现的。待处理的数据流进入数据通道，通过数据通道分配到下一个节点 NODE 进行处理，每个从外部进入数据通道的数据就是一个单独的待处理的任务。这种处理方式比起一般的多任务调度模式，并行处理的吞吐量大，平均响应率高，可预测性强，但是这种方式引入了更多的上下文切换和重复任务调度，而且应用范围比较局限。

这种数据驱动的实时内核的实现方法，是建立在 MicroC/OS-II(μC/OS-II)基础之上的，这里采用μC/OS-II 作为实时内核的基本框架结构，对于任务管理和时间管理等方面做出重新设计和改造，加入数据通道管理模块，从而 micro-DDROTS 实时内核得以实现。因此本论文的主要工作是：改造μC/OS-II 任务调度算法和μC/OS-II 上下文切换算法，为并发数据驱动的调度算法模式。并在 micro-DDRTOS 内核基础之上加入 GUI 图形用户界面、TCP/IP 协议栈和文件系统，这些作为内核的扩展部分，可以通过配置选项加入系统，或者不加入系统内核，GUI 图形用户界面、TCP/IP 协议栈和文件系统的加入使微内核更加完整，应用范围更加广泛。

本论文的组织结构为：第二章对于嵌入式系统、实时系统和实时内核进行了基本总结和叙述。讲述了实时内核的基本实现策略，介绍了当前的几种流行的实时系统内核。第三章主要讲述了基于并发数据流驱动的模型，具体从任务调度、上下文切换、中断管理等方面说明了 micro-DDRTOS 的内核架构设计方法。然后讲述了硬件+micro-DDRTOS+系统应用（GUI、TCP/IP 和文件系统）整体系统的架构，以及系统的硬件基础结构。第四章详细叙述了 micro-DDRTOS 的具体实现方法。从程序设计的具体方法和系统 API 方面给出了详细的介绍。第五章到第七章主要讲述了，建立在 micro-DDRTOS 上的 GUI 图形用户界面、TCP/IP 协议栈和文件系统的具体实现方法。第八章主要讲述了系统测试环境和测试结果，详细叙述了对于整个系统仿真测试的实现方法。

第二章 研究背景

2.1 嵌入式系统及其发展趋势

嵌入式系统是指将应用程序和操作系统与计算机硬件集成在一起的系统。简单的说就是系统的应用软件与系统的硬件一体化。这种系统具有软件代码小,高度自动化,响应速度快等特点,特别适合于要求实时的和多任务的应用环境。和其它的应用相比，在设计嵌入式系统过程中，需要考虑很多特殊的问题。

- 低成本：大量的嵌入式产品，包括手机、智能家电、和其它电子产品，设计者首先要考虑的是性能价格比的最大化，而提供给设计者的硬件资源是有限的，并且尽可能的充分利用软硬件资源。
- 低功耗：功耗是嵌入式系统另外一个需要着重考虑的问题，应为许多嵌入式系统都是利用电池供电，或者是工作在无人值守的地方。
- 可预测性：嵌入式系统上面大部分运行的是实时系统，对于实时系统来说，对于系统运行情况的精确预测是非常重要的，这意味着在满负荷运行的情况下，系统的运行情况是可以明确确定的，系统并不因为负荷的增加出现问题，系统的设计一般都考虑最坏情况。
- 快速响应：嵌入式系统，例如控制系统，一般都是事件驱动的，需要系统快速反应和处理事件，这些系统一般依赖于中断，并且拥有很短的中断响应时间，和处理过程。
- 时间的精确性：很多分布式的嵌入式系统，管理许多远端网络设备和远端数据。这些设备需要很高的时间精确性，通常要精确到毫秒或微秒级。

一般嵌入式系统和具体的应用联系非常紧密，一个典型的嵌入式系统的开发过程为：需求设计，软件/硬件划分，软件/硬件协同开发，系统集成和测试。以前的每个开发过程严重依赖于设计者的经验，比如资源管理、设备驱动、某些地方需要使用汇编语言，开发和具体应用有关的任务管理和硬件模块，内存空间分配。所以系统的重用性和兼容信并不高。新的产品也许需要全部的重新设计，并且开发成功率不是很高。现在嵌入式产品需要快速推向市场，开发周期越来越短，所以开发者目前越来越倾向于选择通用的硬件平台，采用更高级别的开发语言和采用第三方的工具用于开发过程。目前市场上出现了很多转门用于嵌入式系统的微处理器，基于ARM核的MMU是目前使用最广泛的一种。而在嵌入式系统的软件方面，第三方的嵌入式开发工具越来越被广泛采用。这些系统给开发者提供了高级语言的开发工具和环境和套件，这样可以有效的减少开发过程中的错误，增大了开发的重用性。这些工具主要有：

- 片上系统设计和软硬件协同设计自动化工具，如 Synopsys' System Studio, ARM® Integrator/CP (TM) development platform
- 软件模块化工具，例如：Artisan.
- 集成开发环境，例如：Code Warrior, Multi 2000,.
- 实时操作系统，例如：vxWorks, embedded Linux

这些第三方的开发工具近年来得到了飞速的发展，现在越来越多的嵌入式系统都采用了通用的实时操作系统。这些通用的第三方软件成为嵌入式系统的一个重要发展方向。

2.2 实时系统

实时系统在工业、商业和军事等领域都有非常广泛的用途，并且已经有很多实际的应用。一般来说，实时系统通常是比较复杂的。在实时计算中，系统的正确性不仅仅依赖于计算的逻辑结果而且依赖于结果产生的时间。实时计算任务与常见的只要求逻辑正确性的计算任务之间的最大不同之处就是要满足处理与时间的关系。它经常要处理很多并发事件的输入数据流，这些事件的到来次序和几率通常是不可预测的，而且还要求系统必须在事先设定好的时限内做出相应的响应。因此，实时系统的设计时面临着与我们常见的分时系统（如基 Windows, Unix 和 Linux 等的系统）相比，更复杂的和不同的考虑因素。

Donald Gilles 对于实时实时系统的定义如下[1]："A real-time system is one in which the correctness of the computations not only depends upon the logical correctness of the computation but also upon the time at which the result is produced. If the timing constraints of the system are not met, system failure is said to have occurred."这表示一个实时系统必须在可预测的时间内及时响应内部的事件和服务任务，实时系统必须充分利用各种软硬件资源，达到最高的时间利用率。例如一个用于传送带运输的机器人手臂，这个机器人手臂必须非常精确的抓取传送带上的物品。这种情况下，时间是最主要的问题，抓的操作不能过早或者过晚。另外一个更为复杂的例子是一个航空交通管制系统。这些实时系统需要实时监测和指挥飞机的运行情况，还要考虑天气的情况，经过复杂的运算提供给飞行员的起飞和着陆等信息。这些系统必须及时处理各种复杂的任务，包括复杂的数据甚至图像处理。而每个任务的响应级别是不同的，就像上面的系统，也许对于天气等环境的变化需要每2秒响应，而显示系统的刷新需要每0.1秒响应。

实时软件的主要要求实时性、有处理异步并发事件的能力、快速启动、出错处理和自动复位功能，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，并且嵌入式实时软件的开发需要独立的开发平台。对于实时系统，有以下几个方面的术语和指标：

任务单元：指一个基本的工作单元，这一工作可以被系统计划安排和执行。例如一个读取外部事件数据然后存入内存的某个区域的功能。

任务：指一组相关工作的集合，可以完成某个特定的功能。例如从外部传感器读入数据存入内存，然后加工和处理数据，输出数据等一系列过程。一个任务一般是周期化的。

响应时间：指一个外部事件发生到相应的任务处理的这段引发时间，这主要是针对于多任务的情况。

限定时间：指一个任务从开始执行到执行完毕所规定的时间限制。

执行时间：指从响应时间到任务执行完毕所消耗的时间，或者指数据有效等待时间。

延迟时间：指一个任务的执行时间和其截止时间的差值。

生存时间：是数据有效等待时间，在这段时间里数据是有效的。

吞吐量：是在给定时间内，系统可以处理的事件总数。

这些参数的定义基本上用来衡量时系统的实时性能。根据对实时性能的要求程度不同，实时系统又可以分为“硬实时系统”和“软实时系统”。二者的区别在于：前者如果在不满足响应时限、响应不及时或反应过早的情况下都会导致灾难性的后果；而后者则在不满足响应时限时，只是系统性能退化，但并不会导致灾难性的后果。由于实时系统很多应用于一些关键性的场合，所以系统的稳定性和容错性也非常重要。另外实时系统的特性决定了传统的性能衡量标准对其是不适用的。对传统的分时系统来说，软件的执行在时间上的要求并不严格，要求的是尽可能大的系统吞吐量，合理的响应速度。然而在实时系统中，实时的数据吞吐放到首先必须确保的位置，而不是追求任务的系统吞吐量；根据实时需求对任务的优先响应取代了分时系统对每个任务的公平恰当的响应速度。

一般来说，实时系统的软硬件结构需要满足以下要求：

（1）可满足系统中实时任务负载的运算速度；

（2）延迟时间可预测并可满足响应时间要求的中断处理机制；

（3）具有时间指标保障能力的I/O处理；
（4）合理的处理器和I/O设备的拓扑连接；
（5）高速可靠的和有时间约束的通信；
（6）实时的出错处理；
（7）实时任务调度和并发处理，主要要求是满足时间指标的正确性要求，然后提供满足运行时间需求的，适应范围尽可能广的实时调度。
（8） 实时多任务操作系统，这是现代实时系统所主要依赖的基础平台，提供实时应用的开发环境。在现代的实时系统当中一般都有实时多任务操作系统的存在。因为操作系统使系统的设计更加的简便，保证系统的质量以及能够提供其他分时操作系统所提供的服务。这样实时的操作系统就面临着更高的设计要求和更大应用的范围。

在一个较为完善的实时系统环境中，除了硬件方面的支持外，在软件方面，最重要的莫过于需要一个支持实时多任务的操作系统（real-time operating system, RTOS）内核，因为广泛应用于现实世界中的计算设备必须具有与外部环境实时交互的能力。另外，RTOS是和实时应用复杂化直接相关的。当前的实时应用不仅仅局限于原来的只是面对系统级的需求，而是需要面向用户级的应用，满足在各个层次上尤其是消费电子产品的需求。由此，实时应用系统的开发和定制变得越来越重要，而传统的使用主循环控制的实时系统远不能满足需求。如果实时应用软件开发还是在基于处理器直接编写，没有基于一个完善的RTOS，那么无法将系统软件和应用软件分离，每次都要特别定制系统软件和应用软件，开发的代价太大，成本过高。RTOS 是实时应用软件的开发从“小生产方式”进入“大生产方式”的必然产物。

2.3 实时操作系统

Posix 标准 1003.1 定义实时操作系统为："Real-time in operating systems: the ability of the operating system to provide a required level of service in a bounded response time."实时操作系统(RTOS)提供了任务进程管理、时间管理、进程间通信等操作系统基本功能。实时操作系统更加强调其实时性,是能支持实时系统工作的操作系统。其首要任务是调度一切可利用的资源完成实时控制任务,其次才着眼于提高计算机系统的使用效率,这种系统的重要特点是要满意对时间的限制和要求。

实时操作系统主要有以下特点：

1．确定性（Determinism）：

实时软件对于外部事件的响应时间必须是实时的、确定的和可以重复实现的，不管当时系统内部状态如何，都是可预测的。一个测量操作系统确定性能力的指标是从一个高优先级设备中断到达到开始服务的最大延迟。在非实时操作系统中，这个延迟可能是几十到数百毫秒；而在 RTOS 中有一个明确的上界，从几个微秒到一毫秒不等。

2．响应性（Responsiveness）：

这是与确定性相关但是不同的特征。确定性考虑在应答一个中断前，操作系统延迟时间；而响应性是在应答中断后，操作系统服务中断时间。响应性特点包含以下几个方面：

1) 初始化中断处理和开始执行中断服务程序（ISR）需要的时间。如果是要求一个进程切换的 ISR 执行，那么比在当前进程上下文中的 ISR 执行延迟更长时间。

2) 执行 ISR 需要的时间，通常依赖于硬件平台。

3) 中断嵌套的作用。如果一个 ISR 可能被另一个中断的到达而中断，那么它的服务将被延迟。

确定性和响应性共同构成了对于外部事件的响应时间。响应时间对于 RTOS 是至关重要的，由于系统必须满足系统外部个人、设备或者数据流的定时要求。

3．用户控制（User control）：

这一点在 RTOS 中比在一般操作系统中更加宽广。在一个典型的非实时操作系统中，用户或者

不能控制操作系统的调度功能，或者只能提供粗略的指导，如将用户编组进入多于一个优先级的类别。但是，在 RTOS 中，允许用户精细控制任务优先级是基本功能。用户应该能够区分硬实时任务和软实时任务，并在每个类别中指定相对的优先级。一个 RTOS 也能够指定这类特征，如进程切换，哪些进程必须总是驻留在主存中，使用哪个磁盘传输算法，在各种优先级类别中的进程有哪些权利，等等。

4．可靠性（Reliability）：

通常它在实时系统中比在非实时系统中更重要。在非实时系统中一个瞬时错误可以简单地通过重新引导系统来解决，在多处理机非实时系统中一个处理机的失败可能在修复或代替该处理机之前造成服务级别的降低。但是，实时系统需要实时响应和控制事件，性能的丧失或降低可能造成灾难性的后果。

5．软失败操作（Fail-soft operation）：

与实时系统和非实时系统其它方面的差别一样，这里存在一个程度上的差别。软失败操作是指这样一个特征，系统能够以这样一种方式失败，它能够尽可能地保留能力和数据。例如，一个典型的传统 UNIX 系统，当它检测到内核数据的误用，在系统控制台上发出失败消息，将内存内容倒到磁盘用于以后的失败分析，同时终止系统的执行。与此相反，RTOS 将试图或者纠正这个问题或者最小化它的影响，同时继续运行。典型情况下，系统通告用户或者用户进程，它将试图进行纠正动作，然后继续操作，或许此时降低了服务级别。

最常用的操作系统是通用操作系统，通用操作系统是由分时操作系统发展而来，大部分都支持多用户和多进程，负责管理众多的进程并为它们分配系统资源。分时操作系统的基本设计原则是：尽量缩短系统的平均响应时间并提高系统的吞吐率，在单位时间内为尽可能多的用户请求提供服务。由此可以看出，分时操作系统注重平均表现性能，不注重个体表现性能。如对于整个系统来说，注重所有任务的平均响应时间而不关心单个任务的响应时间，对于单个任务来说，注重每次执行的平均响应时间而不关心某次特定执行的响应时间。通用操作系统中采用的很多策略和技巧都体现出了这种设计原则，如虚存管理机制中由于采用 LRU 等页替换算法,使得大部分的访存需求能够快速地通过物理内存完成,只有很小一部分的访存需求需要通过调页完成，但从总体上来看，平均访存时间与不采用虚存技术相比没有很大的提高，同时又获得了虚空间可以远大于物理内存容量等好处，因此虚存技术在通用操作系统中得到了十分广泛的应用。

对于实时操作系统，它除了要满足应用的功能需求以外，更重要的是还要满足应用提出的实时性要求，而组成一个应用的众多实时任务对于实时性的要求是各不相同的，此外实时任务之间可能还会有一些复杂的关联和同步关系，如执行顺序限制、共享资源的互斥访问要求等，这就为系统实时性的保证带来了很大的困难。因此，实时操作系统所遵循的最重要的设计原则是：采用各种算法和策略，始终保证系统行为的可预测性。可预测性是指在系统运行的任何时刻,在任何情况下，实时操作系统的资源调配策略都能为争夺资源(包括CPU、内存、网络带宽等)的多个实时任务合理地分配资源，使每个实时任务的实时性要求都能得到满足。与通用操作系统不同，实时操作系统注重的不是系统的平均表现，而是要求每个实时任务在最坏情况下都要满足其实时性要求，也就是说，实时操作系统注重的是个体表现，更准确地讲是个体最坏情况表现。下面就从任务调度策略、中断管理等几个方面来说明实时操作系统的特点。

1．任务调度策略：[20][21]

实时操作系统中的任务调度策略目前使用最广泛的主要可分为两种,一种是静态表驱动方式，另一种是固定优先级抢先式调度方式。静态表驱动方式是指在系统运行前工程师根据各任务的实时要求用手工的方式或在辅助工具的帮助下生成一张任务的运行时间表，这张时间表与列车的运行时刻表类似，指明了各任务的起始运行时间以及运行长度，运行时间表一旦生成就不再变化了，在运行时调度器只需根据这张表在指定的时刻启动相应的任务理即可。静态表驱动方式的主要优点是：

- 运行时间表是在系统运行前生成的,因此可以采用较复杂的搜索算法找到较优的调度方案。

- 运行时调度器开销较小。
- 系统具有非常好的可预测性，实时性验证也比较方便。

这种方式主要缺点是不灵活，需求一旦发生变化，就要重新生成整个运行时间表。由于具有非常好的可预测性，这种方式主要用于航空航天、军事等对系统的实时性要求十分严格的领域。

固定优先级抢先式调度方式则与通用操作系统中采用的基于优先级的调度方式基本类似，但在固定优先级抢先式调度方方式中，进程的优先级是固定不变的，并且该优先级是在运行前通过某种优先级分配策略来指定的。这种方式的优缺点与静态表驱动方式的优缺点正好完全相反，它主要应用于一些较简单、较独立的嵌入式系统，但随着调度理论的不断成熟和完善，这种方式也会逐渐在一些对实时性要求十分严格的领域中得到应用。目前市场上大部分的实时操作系统采用的都是这种调度方式。

2．内存管理:

关于虚存管理机制在上面已经进行了一些分析。为解决虚拟内存给系统带来的不可预测性，实时操作系统一般采用如下两种方式：

•在原有虚存管理机制的基础上增加页面锁功能，用户可将关键页面锁定在内存中，从而不会被 swap 程序将该页面交换出内存。这种方式的优点是既得到了虚存管理机制为软件开发带来的好处，又提高了系统的可预测性。缺点是由于 TLB 等机制的也是按照注重平均表现的原则进行的，因此系统的可预测性并不能完全得到保障。

• 采用静态内存划分的方式,为每个实时任务划分固定的内存区域。这种方式的优点是系统具有较好的可预测性，缺点是灵活性不够好，任务对存储器的需求一旦有变化就需要重新对内存进行划分，此外虚存管理机制所带来的好处也丧失了。

目前市场上的实时操作系统一般都采用第一种管理方式。

3．中断处理:

在通用操作系统中，大部分外部中断都是开启的，中断处理一般由设备驱动程序来完成。由于通用操作系统中的用户进程一般都没有实时性要求，而中断处理程序直接跟硬件设备交互，可能有实时性要求，因此中断处理程序的优先级被设定为高于任何用户进程。

但对于实时操作系统采用上述的中断处理机制是不合适的。首先，外部中断是环境向实时操作系统进行的输入，它的频度是与环境变化的速率相关的，而与实时操作系统无关。如果外部中断产生的频度不可预测，则一个实时任务在运行时被中断处理程序阻塞的时间开销也是不可预测的，从而使任务的实时性得不到保证；如果外部中断产生的频度中可预测的，一旦某外部中断产生的频度超出其预测值(如硬件故障产生的虚假中断信号或预测值本身有误)就可能会破坏整个系统的可预测性；其次，实时控制系统中的各用户进程一般都有实时性要求，因此中断处理程序优先级高于所有用户进程的优先级分配方式是不合适的。

一种较适合实时操作系统的中断处理方式为：除时钟中断外，屏蔽所有其它中断，中断处理程序变为周期性的轮询操作。采用这种方式的主要好处是充分保证了系统的可预测性，主要缺点是对环境变化的响应可能不如上述中断处理方式快。另外轮询操作在一定程度上降低了 CPU 的有效利用率。另一种可行的方式是：对于采用轮询方式无法满足需求的外部事件，采用中断方式，其它时间仍然采用轮询方式。但此时中断处理程序与所有其它任务一样拥有优先级，调度器根据优先级对处于就绪态的任务和中断处理程序统一进行处理器调度。这种方式使外部事件的响应速度加快避免了上述中断方式带来第二个问题，但第一个问题依然存在。此外为提高时钟中断响应时间的可预测性，实时操作系统应尽可能少地屏蔽中断。

4．共享资源的互斥访问:

通用操作系统一般采用信号量机制来解决共享资源的互斥访问问题。　　对于实时操作系统，如果任务调度采用静态表驱动方式，共享资源的互斥访问问题在生成运行时间表时已经考虑到了，在运行时无需再考虑。如果任务调度采用基于优先级的方式，则传统的信号量机制在系统运行时很

容易造成优先级倒置问题，即当一个高优先级任务通过信号量机制访问共享资源时，该信号量已被一个低优先级任务占有，而这个低优先级任务在访问共享资源时可能又被其它一些中等优先级的任务抢先，因此造成高优先级任务被许多具有较低优先级的任务阻塞，实时性难以得到保证。因此在实时操作系统中,往往对传统的信号量机制进行了一些扩展，引入了如优先级继承协议、优先级倒置协议等机制，较好地解决了优先级倒置的问题。

5．系统调用以及系统内部操作的时间开销:

进程通过系统调用得到操作系统提供的服务，操作系统通过内部操作(加上下文切换等)来完成一些内部管理工作。为保证系统的可预测性，实时操作系统中的所有系统调用以及系统内部操作的时间开销都应该是有界的，并且该界限是一个具体的量化数值。而在通用操作系统中对这些时间开销则未做如此限制。

6．系统的可重入性:

在通用操作系统中，核心态系统调用往往是不可重入的，当一低优先级任务调用核心态系统调用时，在该时间段内到达的高优先级任务必须等到低优先级的系统调用完成才能获得 CPU，这就降低了系统的可预测性。因此，实时操作系统型中的核心态系统调用往往设计为可重入的。

7．辅助工具:

实时操作系统额外提供了一些辅助工具，如实时任务在最坏情况下的执行时间估算工具、系统的实时性验证工具等，可帮助工程师进行系统的实时性验证工作。

此外，实时操作系统对系统硬件设计也提出了一些要求，如: DMA 是一种数据交换协议，主要作用是在无需 CPU 参与的情况下将数据在内存与其它外部设备间进行交换。DMA 最常用的一种实现方式被称为周期窃取(Cycle Stealing)方式，即首先遇过总线仲裁协议与 CPU 竞争总线控制权，在获得控制权后再根据用户预设的操作指令进行数据交换。由于这种周期窃取方式会给用户任务带来不可预测的额外阻塞开销，所以实时操作系统往往要求系统设计时不采用 DMA 或采取一些可预测性更好 DMA 实现方式,如 Time-slice method 等。

Cache 的主要作用是采用容量相对较小的快速存储部件来弥补高性能 CPU 与相对来说性能较低的存储器之间的性能差异，由于它可以使系统的平均表现性能得到大幅提高，因此在硬件设计中得到了极为广泛的应用。

但实时操作系统注重的不是平均表现性能，而是个体最坏情况表现，因此在对系统进行实时性验证时必须考虑实时任务运行的最坏情况,即每次访问内存都没有命中Cache情况下的运行时间，所以在利用辅助工具估算实时任务在最坏情况下的执行时间，应将系统中所有的Cache功能暂时关闭，在系统实际运行时再将Cache功能激活。除此以外，另一种较极端的做法则是在硬件设计中完全不采用Cache技术。

2.4 实时系统内核

2.4.1 实时内核发展和特点

八十年代后期，国外提出了微内核(micro kernel) 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，其它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。在理论上，这种方法提供了高度的灵活性、模块性和可移植性。

微内核(micro kernel)的思想是仅仅抽象必须在内核模式中执行的基本的核心操作系统功能，非基本的服务和应用构筑在微内核之上，并且在用户模式中执行。尽管微内核里面和外面的分界随着设计不同而不同，但是共同的特征是，许多传统上是操作系统一部分的服务现在出现在内核模式外

部，这些包括设备驱动程序、文件系统、虚拟内存管理程序、窗口系统和安全系统。

微内核结构用水平结构代替了传统的垂直的分层结构，操作系统组件放在微内核外部，实现成服务器进程，它们的相互作用变成了对等的，典型地通过微内核传递消息，微内核功能成了消息交换：它验证消息，在组件之间传递它们，并且允许访问硬件。

微内核组织的优点：

1．接口一致(Uniform interface)：

所有进程请求使用统一的接口，进程不需要区分内核模式和用户模式服务，由于这些服务全部通过消息传递提供。

2．扩展性(Extensibility)：

由于新型硬件设备和新型软件技术不断发展，任何操作系统不可避免地需要修改，微内核结构允许加入新的服务，支持在同一功能区内提供多重服务。使用微内核结构，加入一个新特点仅仅需要修改或者加入被选择的服务，新的或者修改的服务影响限制在系统的一个子集，进一步地，修改不需要建立一个新的内核。

3．移植性(Portability)：

在微内核结构中，所有与特定处理机的代码仅仅存在微内核中，因此，需要将这个系统移植到新的处理机需要作的变化可以尽可能地少。

4．可靠性(Reliability)：

软件规模越大，确保可靠性越难。尽管模块化设计帮助我们增强了可靠性，但是微内核结构带来了更多的收益。一个小型微内核可以严格地进行测试，它使用少量的应用程序接口有助于内核模式之外的操作系统服务代码提高质量。系统只需要掌握一个有限数量的（API），程序员之间的相互交互也有限，因而对于系统其它组件的影响最小。

5．分布系统支持(Distributed system support)：

微内核适合于由分布操作系统控制的群集系统。当一个消息从客户发到服务器进程，该消息必须包含请求服务的标识符。当一个分布系统（如群集系统）被配置为所有的进程和服务有唯一的标识符，那么似乎在内核模式下存在单一的系统映象，一个进程能够在不知道目标服务驻留在哪台机器情况下发送消息。

6．面向对象操作系统(Object-oriented operating system)：

微内核结构非常适合面向对象操作系统的要求，一个面向对象的方法有助于一个微内核的设计和操作系统的模块化扩充。将微内核结构和面向对象操作系统原理结合起来的例子是组件的使用，组件是清楚定义了接口的对象，它们能够互连起来以预制建筑块的方式形成软件，使用组件之间的接口实现组件之间的全部相互作用。

2.4.2 实时内核种类

对于实时系统内核有很多种的定义方式。下面给出了从简单到复杂的几种类型的内核定义。

- 超小内核：只是提供了一种简单的线程切换和管理，提供了有限的任务切换、管理和任务间通信的简单功能模块。
- 微内核：提供了完整的任务调度和任务管理及通过信号量、邮箱等机制实现进程间同步和通信进程间通信的管理功能。
- 内核：在微内核的基础之上，包括了内存管理、I/O 管理、和其它复杂的功能模块。
- 操作系统：在内核的基础之上，具有了通用用户界面，开发工具，安全管理和文件管理系统能扩展功能。

下面讨论一些在内核设计上常用了一些从简单到复杂的策略。

1．循环轮询系统(Polling Loop)

这样的系统里面只有一个简单的代码循环，也就是循环查询系统。在循环过程中不断检测某些标志位，这些标志位表示某些事件是否发生。如果事件没有发生，则继续循环查询。如果事件发生，则执行相应处理程序。这样的系统没有任务切换和进程间通信，它只有一个单循环进程。这类系统一般在主循环之外还可以处理一些简单的中断事件。这种方式在一些简单的嵌入式系统中被广泛采用，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，而且响应时间可以非常简单的得到预测。但这些系统不能够处理更加复杂的情况，不能够充分利用 CPU 的资源，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。

2．有限状态机（Finit State Machine）

有限状态机非常类似于 IF-ELSE 或者 CASE 语句，对于小的系统而言，便于编程和理解，可以快速的执行，只是通过改变输出功能来改变机器的响应。但是其应用领域有限，不能保证确定性，对于大的应用系统，难于调试。

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

应用程序是一个无限的循环，循环中调用相应的函数完成相应的操作，这部分可以看成后台行为(background)。中断服务程序处理异步事件，这部分可以看成前台行为（foreground）。后台也可以叫做任务级。前台也叫中断级。时间相关性很强的关键操作(Critical operation)一定是靠中断服务来保证的。因为中断服务提供的信息一直要等到后台程序走到该处理这个信息这一步时才能得到处理，这种系统在处理信息的及时性上，比实际可以做到的要差。这个指标称作任务级响应时间。最坏情况下的任务级响应时间取决于整个循环的执行时间。因为循环的执行时间不是常数，程序经过某一特定部分的准确时间也是不能确定的。进而，如果程序修改了，循环的时序也会受到影响。

4．实时多任务系统(Multitasking 或 Multi-thread Program Model)

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。

需要采用一些新的软件设计方法，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。

2.4.3 常见的实时内核

这里简短的介绍一下比较著名的实时操作系统内核。这里主要有两种类型的实时内核。一种是用于商业化的实时内核，主要有 VxWorks、QNX、pSOS、WindowsCE、eCOS 和 RTLinux，基本上这些商业化的实时内核符合 POSIX 标准，并且都是可裁减配置的，以用于不同的应用环境。另外一种是学术研究的实时内核，YAROTS 和 EMERALDS。

1． MicroC/OS-II[3][19]

μC/OS-II 的前身是μC/OS，最早出自于 1992 年美国嵌入式系统专家 Jean J.Labrosse 在《嵌入式系统编程》杂志的 5 月和 6 月刊上刊登的文章连载，并把μC/OS 的源码发布在该杂志的 BBS 上。当时就有 500 多人下载了这份源码。世界上数以千计的工程技术人员将μC/OS 应用到了各个领域，如照相机业、发动机控制、网络接入设备、高速公路电话系统、ATM 机和工业机器人等。μC/OS 和μC/OS-II 是专门为计算机的嵌入式应用设计的，绝大部分代码是用 C 语言编写的。CPU 硬件相关部分是用汇编语言编写的、总量约 200 行的汇编语言部分被压缩到最低限度，为的是便于移植到任何一种其它的 CPU 上。许多移植的范例可以从网站上得到。用户只要有标准的 ANSI 的 C 交叉编译器，有汇编器、连接器等软件工具，就可以将μC/OS 嵌入到开发的产品中。

μC/OS 具有执行效率高、占用空间小、实时性能优良和可扩展性能等特点，最小内核可编译至 2KB。μC/OS-II 已经移植到了几乎所有知名的 CPU 上。并且是公开源代码的。其主要特点是：

1). 可移植性（Portable）

绝大部分μC/OS-II 的源码是用移植性很强的 ANSI C 写的。和微处理器硬件相关的那部分是用

汇编语言写的。汇编语言写的部分已经压到最低限度，使得μC/OS-II便于移植到其他微处理器上。如同μC/OS 一样，μC/OS-II 可以移植到许许多多微处理器上。条件是，只要该微处理器有堆栈指针，由 CPU 内部寄存器入栈、出栈指令。另外，使用的 C 编译器必须支持内嵌汇编（inline assembly）或者该 C 语言可扩展、可连接汇编模块，使得关中断、开中断能在 C 语言程序中实现。μC/OS-II 可以在绝大多数 8 位、16 位、32 位以至 64 位微处理器、微控制器、数字信号处理器（DSP）上运行。 从移植了的μC/OS 升级到μC/OS-Ⅱ，全部工作一个小时左右就可完成。因为μC/OS-II 和μC/OS 是向下兼容的，应用程序从μC/OS 升级到μC/OS-II 几乎不需要改动或根本不需要改动。移植的范例可以从互联网上找到，网址是 www.μC/OS-II.com。

2) 可固化（ROMable）

μC/OS-II 是为嵌入式应用而设计的，这就意味着，只要读者有固化手段（C 编译、连接、下载和固化），μC/OS-II 可以嵌入到读者的产品中成为产品的一部分。

3) 可裁剪（Scalable）

可以只使用μC/OS-II 中应用程序需要的那些系统服务。也就是说某产品可以只使用很少几个μC/OS-II 调用，而另一个产品则使用了几乎所有μC/OS-II 的功能。这样可以减少产品中的] μC/OS-II 所需的存储空间（RAM 和 ROM），这种可裁剪性是靠条件编译实现的。只要在用户的应用程序中（用 #define constants 语句）定义那些μC/OS-II 中的功能是应用程序需要的就可以了。程序和数据两部分的存储用量已被最大努力的压低了。

4) 占先式（Preemptive）

μC/OS-II 完全是占先式的实时内核。这意味着μC/OS-II 总是运行就绪条件下优先级最高的任务。大多数商业内核也是占先式的，μC/OS-II 在性能上和它们类似。

5) 多任务(Multi-task)

μC/OS-II 可以管理 64 个任务，然而，目前这一版本保留 8 个给系统。应用程序最多可以有 56 个任务。赋予每个任务的优先级必须是不同的，这意味着μC/OS-II 不支持时间片轮转调度法（Round-robin Scheduling）。该调度法适用于调度优先级平等的任务。

6) 可确定性

全部μC/OS-II 的函数调用与服务的执行时间具有其可确定性。也就是说，全部μC/OS-II 的函数调用与服务的执行时间是可知的。进而言之，μC/OS 系统服务的执行时间不依赖于应用程序任务的多少。

7) 任务栈

每个任务有自己单独的栈，μC/OS-II 允许每个任务有不同的栈空间。以便压低应用程序对 RAM 的需求。使用μC/OS-II 的栈空间校验函数，可以确定每个任务到底需要多少栈空间。

8) 系统服务

μC/OS-II 提供很多系统服务，例如邮箱、消息队列、信号量、块大小固定的内存的申请与释放、时间相关函数等。

9) 中断管理

中断使正在执行的任务暂时挂起。如果优先级更高的任务被该中断唤醒，则高优先级的任务在中断嵌套全部退出后立即执行，中断嵌套层数可达 255 层。

2．QNX

QNX 是一个分布式、嵌入式、可规模扩展的实时操作系统。它遵循 POSIX.1、(程序接口)和 POSIX.2(Shell 和工具)、部分遵循 POSIX.1b(实时扩展)。它最早开发于 1980 年，到现在已相当成熟。QNX 是一个微内核实时操作系统，其核心仅提供 4 种服务：进程调度、进程间通信、底层网络通信和中断处理，其进程在独立的地址空间运行。

1) 任务调度策略

QNX 提供 POSIX.1b 标准进程调度；

32 个进程优先级；

抢占式的、基于优先级的正文切换；

可选调度策略：FIFO、轮转策略、适应性策略。

LynxOS 其调度策略为：

LynxOS 支持线程概念，提供 256 个全局用户线程优先级；

硬实时优先级调度：在每个优先级上实现了轮转调度、定量调度和 FIFO 调度策略；

快速正文切换和阻塞时间短；

抢占式的 RTOS 核心。

2) QNX 的系统服务：

多种资源管理器，包括各种文件系统和设备管理，支持多个文件系统同时运行，包括提供完全 POSIX.1 及 UNIX 语法的 POSIX 文件系统，支持多种闪存设备的嵌入式文件系统，支持对多种文件服务器(如 Windows NT/95、LAN Manager 等)的透明访问的 SMB 文件系统、DOS 文件系统、CDROM 文件系统等。

设备管理：在进程和终端设备间提供大吞吐量、低开销接口服务。

图形 / 窗口支持：包括 QNX Windows、X Window System for QNX、对 MS Windows NT/95 和 X Window 系统的远程图形连接。

TCP/IP for QNX。

高性能、容错型 QNX 网络--FLEET，使得所有连入网络的计算机变成一个逻辑上的超级计算机。

透明的分布式处理。FLEET 网络处理与消息传递和进程管理原语的集成，将本地和网络 IPC 统一起来，使得网络对 IPC 而言是透明的。

3．VxWorks

VxWorks 操作系统是美国 WindRiver 公司于 1983 年设计开发的一种嵌入式实时操作系统（RTOS），具有良好的持续发展能力、高性能的内核以及友好的用户开发环境，在嵌入式实时操作系统领域占据一席之地。它以其良好的可靠性和卓越的实时性被广泛地应用在通信、军事、航空、航天等高精尖技术及实时性要求极高的领域中。VxWorks 是 Windriver 公司开发专门为实时嵌入式系统设计开发的操作系统软件，为程序员提供了高效的实时任务调度、中断管理，实时的系统资源以及实时的任务间通信。应用程序员可以将尽可能多的精力放在应用程序本身，而不必再去关心系统资源的管理。该系统主要应用在单板机、数据网络（以太网交换机、路由器）和通信方面等多方面。目前 VxWorks 操作系统的版本为 VxWorks 5.4，其核心功能主要有：

- 微内核 wind
- 任务间通信机制
- 网络支持
- 文件系统和 I/O 管理
- POSIX 标准实时扩展，C++以及其他标准支持

4．RTLinux

RT－Linux 是一个嵌入式硬实时操作系统，它部分支持 POSIX.1b 标准，　RT－Linux 实现了一个小的实时核心，仅支持底层任务创建、中断服务例程的装入、底层任务通信队列、中断服务例程(ISR)和 Linux 进程。原来的非实时 Linux 核心作为一个可抢先的任务运行于这个小的核心之上，所有的任务都在核心地址空间运行。它不同于其它微内核和大型内核，属于实时 EXE(realtime executive)体系结构。RT－Linux 在操作系统之下实现了一个简单的实时核心，Linux 本身作为一个可抢占的任务在核内运行，优先级最低，随时会被高优先级任务抢占。用户可自行编写调度程序，它

们可实现为可加载的核心模块；已实现的调度程序有：基于优先级的抢占式调度和 EDF 调度；基于优先级的调度使用"单调率算法"，它直接支持周期任务。具有 Linux 的开放性。

第三章 面向并发数据驱动的微内核架构

数据流在计算机的各个方面都有相关的概念，比如在程序编写方面，它指程序处理的数据流向。这里的数据流指的是嵌入式实时系统内核的处理数据的一种模型结构。如图 3.1 所示。

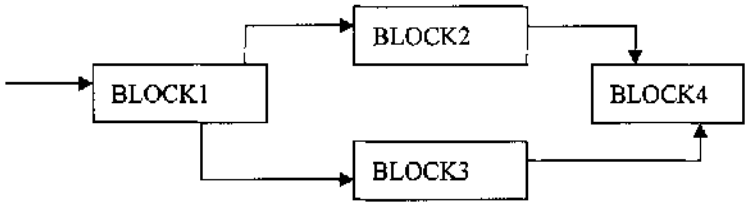


图 3.1 事件并发数据流模型

图中的并发数据流处理模型，箭头指明了数据的流向，每一个模块等待数据的流入和触发，每个模块承担了系统处理数据的某个功能，例如这个系统可以是一个用于电信的信号分析仪，它可以接收标准 7 号信令，可以分析在 7 号信令之上传送的各种标准信号，例如 ISUP、TUP、IS41 等。这里面 BLOCK1 负责接收底层信号（MTP1-MTP3）,然后将其转变为系统内部数据格式，交给不同的 BLOCK 进行处理分析，最后分析的结果交给 BLOCK4 输出显示。数据流的传送方式有两种：一对一和一对多。每个进入 BLOCK1 的数据就是系统待处理的一个任务。借助于实时系统内核的任务调度可以实现数据流的并行处理，这样就实现了数据流任务的并行处理。图 3.1 的数据流模型完全不同于在软件工程里面定义的数据流程图，数据流程图用于需求分析之后的系统构架设计，它主要用于说明系统整体架构内部各个子系统之间的层次关系和系统各个部分的输入输出关系。这里所讨论的数据流模型表示通过数据通道的管理，将数据流作为处理节点（BLOCK）的统一界面，形成一种嵌入式实时系统任务管理的模式。

3.1 需求和前提

数据流驱动的应用模型框架，和一般的应用系统有很大的区别，所以在系统内核的设计上面也有特殊的需求和设计要求。下面总结了数据流驱动方式和其它应用方式的不同点。

1) 子系统的划分：传统的嵌入式系统中，整个系统被划分为子系统或是组件，每个组件被封装在分割的任务过程或者线程中，数据流驱动的应用模型框架中，对于子系统和组件的划分，与传统嵌入式系统不同，它是按照系统处理数据的流程，在考虑系统或者组件的重用性的前提下，把整个过程划分为可重用的数据处理节点，每个节点视为一个数据处理子过程。
2) 子系统之间的交互：数据流驱动的模式，如果节点或者组件的数量划分的比较多，那么就会增大组建间的通信量和交互频率。每个相邻的节点或者组件之间通过数据通道进行数据传递的效率直接影响到系统的性能。
3) 调度方面：数据流驱动的模式对于系统调度也有特殊的要求，它不同于传统的任务调度策略。每个节点或者组件被从数据通道传递的数据触发唤醒之后，需要花费更多的时间处理状态的改变和等待数据的接收和流入以及处理结果的数据输出。
4) 数据流的管理：数据流驱动方式的核心是对于数据流信号的管理。需要设计高效和可靠的

数据信号管理方式。

除了这些数据流驱动方式的特殊要求之外，作为嵌入式实时操作系统内核，还应该就有以下一些需求，需要在系统设计的时候考虑。

1) 高性能：数据流驱动方式主要应用于处理大数据量的业务当中，所以应用环境要求系统充分发挥 CPU 和硬件资源，增大 CPU 的利用率。
2) 快速上下文切换：数据驱动的构架方式，意味着比传统方式更多的处理线程和任务，这就需要更快的上下文切换速度，以弥补数据驱动方式内核的这一缺点。
3) 高效的内部通信机制：由于数据驱动方式的内核构架中，子组件之间的接口和交互完全依赖于数据信号的传递，所以需要设计更加有效的数据流传递方式。
4) 基于数据流的调度：传统的嵌入式系统实时内核的任务调度一般采用了抢占是优先级策略，而数据流驱动数据调度策略则依赖于数据信号的优先级，数据通道根据数据信号的优先级来管理数据流的任务调度。
5) 组件触发的动态优先级：子组件或者节点的执行是通过数据信号来触发的，而触发的数据信号的优先级是不同的，这样同一个组件或者节点的每一次的执行的优先级就有所不同。

3.2 多线程事件触发数流模型

数据流模型:不仅可以很好地描述数据处理与转换的问题，而且具有很好的可视性与较高的运行效率，因此被广泛应用于面向数据处理的编程领域。近年来对数据流的研究主要是对纯数据驱动的数据流模型的研究，而纯数据流模型不能很好地与事件驱动的操作系统协调，因此必须对单线程动态纯数据流的驱动机制与运行机制加以改进和扩展才能提高系统的响应速度和运行效率，目前对测控系统进行建模和对操作系统的运行机制的分析，多线程事件触发数据流模型已经被提出。[2]

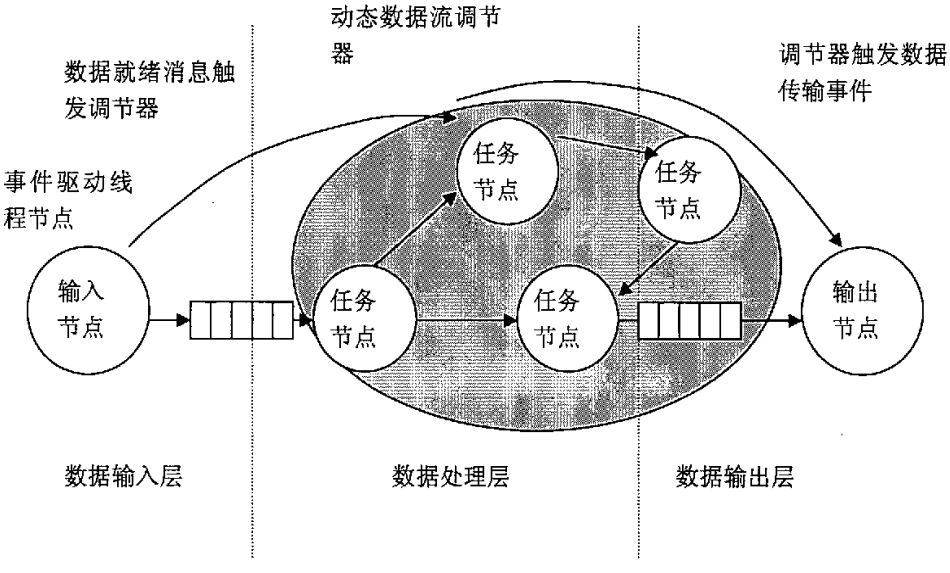


图3.2事件触发并发数据流模型

事件触发并发数据流模型，是在研究测控系统的特性之后，在动态纯数据流的基础上提出来的。一般来讲，用于数据处理的嵌入式系统可以分为三个层次：数据输入层，其主要功能是从物理硬件获取数据或自身产生数据，是数据处理层的数据来源。数据处理层，完成各种数据处理，如滤波、控制算法计算等等。数据输出层，包括从网络输出数据,把数据输出到存储器或输出到显示器等等。

从实时性的角度来考虑，数据输入层的优先级应该高于数据处理层的优先级，而数据处理层的优先级又应该高于数据输出层，在事件触发并发数据流模型中，不同优先级的任务可以用不同优先级的线程节点来表示，不同线程节点之间用带有先入先出队列的边连接，不至于在运行时丢失数据。数据流的调度器由事先定义的消息激活，调度器执行完毕又可以触发数据传输事件，通知线程输出节点可以输出数据。模型如图3.2所示。

一个事件触发并发数据流程序可以表示为一个四元组P=(F,T,A,E)，其中F为数据任务节点集，每个节点包含个输出端口集Qo，一个输入端口集Qi；T是一个线程节点集，也包含一个输入端口集Qi与一个输出端口集Qo；A是有序积((F*Qo)×((F*Qi)∨(T*Qi))的一个子集，其元素称为边，是一个事件集，含有线程之间通信的事件，或消息。A中的边a=(u*qo,v*qi,q)表示由源节点u的输出端口到目标节点v的输入端口qi的连接，其中q是内建的先入先出队列结构，源节点可以将数据通过边内 r=FIFO结构缓冲后传输给目标节点。这样的结构，再加上对边内数据读写的同步，顺利解决了线程间数据传输与同步的问题。

数据处理任务节点F可以用一个七元组来表示，F=(B,Em,P,Qi,Qo,a,S)，其中：B为是否线程节点标志，这里为FALSE表示为非线程节点；Em为数据处理任务节点的可激活函数，返回一个布尔量，p为节点的执行函数，这里用p来替代纯数据流模型中的执行函数，以避免与事件集E的命名冲突；Qi为输入端口集合；Qo为输出端口集合；a为节点的属性集，依节点的用途而定；S为节点的状态集，线程节点可以用一个九元组来表示T=(B,Pl,El,l,Es,Qi,Qo,a,S)，其中：B为是否线程节点标志，这里为TRUE表示为线程节点；Pl为线程的优先级；El为监听事件集，指定线程节点响应哪些事件；l为监听函数，对不同的事件作响应的处理；Es为触发事件集，预先定义的用于触发调度器运行的消息集；Qi为输入端口集合；Qo为输出端口集合；a为节点的属性集，依节点的用途而定；S为节点的状态集。

事件触发并发数据流运行调度算法在引入带有FIFO结构的边之后，目标节点和源节点通过读写FIFO直接实现数据传输，因此在程序编辑完成后，不需要对边进行操作。程序完成之后是一个节点列表Ln，算法开始：

1) 遍历节点列表Ln，对每个节点的B进行判断，若B为TRUE，把线程节点放入一个线程节点列表Lt并且在节点列表中删除该节点。
2) 遍历线程节点列表Lt，执行监听函数l，启动线程并对事件进行监听。
3) 若相应的事件发生，则通过监听函数读取数据，若数据满足一定数量则发出数据就绪消息，触发纯数据流调度器。
4) 若数据流调度器被触发，则往下执行，否则等待消息。
5) 遍历节点列表Ln，若满足激活条件则将该节点放入激活节点列表La。
6) 查询La，若调La=Φ，度器关闭，发送数据发送消息.
7) 执行La中的所有节点的执行函数p，并清空La回到算法5。

3.3 面向并发数据流实时系统微内核架构

面向并发数据流驱动的嵌入式实时内核(Micro Data-draft Embeded Real-time Operating System, micro-DDRTOS)，采用了在3.2节的事件触发并发数据流模型的基本框架结构，根据嵌入式实时系统的应用和3.1节的需求特点，提出的一种微内核模型，它通过任务调度和任务切换和特殊的数据通道，对于数据信号进行处理，从而提供了并行流水结构和并行多任务处理，以及动态优先级切换的功能特点。 Micro-DDRTOS是一个动态优先级多任务系统内核，最高优先级的任务总是优先执行，系统具有远程配置功能，无需改变内核代码，就可以对于系统的各种参数配置、动态调度和优先级切换进行重新配置，从而满足了系统的各种应用环境，发挥系统最大应用性能。

Micro-DDROTS由标准C语言实现，少数部分采用汇编语言实现（上下文切换，寄存器操作和

中断管理），具体硬件基于 ARM 体系结构。下面从介绍其主要内核组件：任务调度策略，上下文切换，中断管理，数据流管理，线程管理等的实现方法。

1) 任务节点（NODE）

如图 1.1，面向数据流的模型主要有两部分组成，一个是处理各个数据信号的节点 NODE（数据处理任务节点）,和管理数据信号的数据通道（Data Control channel），这里数据信号是指数据的载体，是待处理数据在系统内部接点传送的方式，任务节点是包装在系统任务来实现的，每个系统任务相互独立，拥有独立数据存储区，和标志目前任务状态的标志区域，用于上下文切换时保存 CPU 寄存器组的内容区域，独立的堆栈用于存放临时数据（例如子程序参数，返回的地址，临时变量）图 3.3 给出了节点的结构。

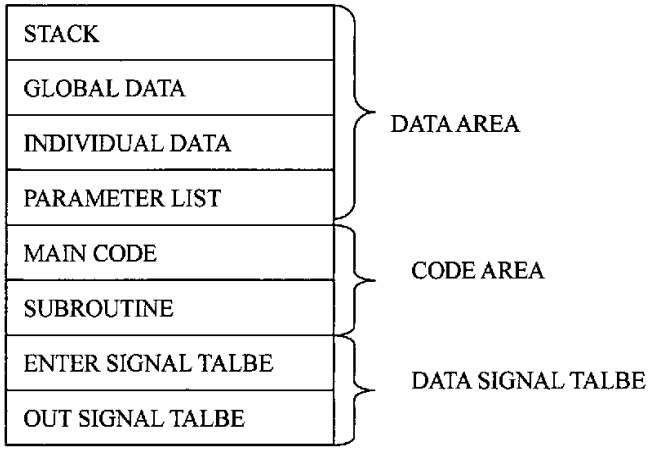


图 3.3 任务节点数据结构

数据处理任务节点主要有三部分组成：数据存储区、代码区和数据信号表。数据存储区主要有任务节点的堆栈、全局数据、局部数据、和参数列表。代码存储区用于存放节点的代码，主要有主程序存储区和子程序存储区两部分组成。数据信号表主要用于存放流入和流出节点的数据信号表结构，用于节点处理数据信号时使用。对应于每个节点在系统内以任务的形式实现，这样每个任务具有独立的地址空间，独立的堆栈和数据存储区域，这样的构架对于实现任务之间的切换和上下文切换都是非常方便的。对于每一个节点任务，在初始化的过程中，给出所占堆栈的大小，任务的初始优先级别，和数据流触发方式。数据流的触发方式和任务节点的调度有很大的关系，对于任务的调度将在后面讨论。对于每一个系统任务节点，在运行的时候有一个任务状态块的数据结构和其相对应，任务状态块主要记录了每个任务节点的相关信息。用于任务节点的管理。其数据结构为：

```
typedef struct{
    Stack       *task_stack;
    Int          stack_size;
    Data        *task_data;
    Individual   *task_individual;
    Parameter   *task_parameter;
    Code         *task_code;
    Context      task_node_env;
    Excute_rule task_excute_run;
    Int          wakeup_time;
    Int          deadline;
    Table        *task_table;
```

```
Attribute    *task_attribute;
State        task_state;
}TCB;
```

每一个任务节点的的任务控制块在系统启动时被初始化，指针*task_stack 指向任务节点的堆栈起始地址；stack_size 记录堆栈大小；指针*task_data 指向任务节点的全局变量；指针*task_individual 指向任务节点的局部数据；指针 *task_parameter 指向任务节点参数列表地址；指针*task_code 指向任务节点的起始代码地址；变量 task_excute_run 记录了任务节点被数据流触发执行的条件规则；wakeup_time 用于任务节点的时间管理；指针*task_table 用于指向任务节点的数据信号表结构；指针*task_attribute 指向存放任务节点的属性结构，本结构存放了任务节点的基本属性；task_state 用于任务的调度状态切换。

2) 任务节点（NODE）的管理

图 3.4 显示了任务节点的运行状态和运行时结点之间的关系。

图中显示了任务节点的运行方式，图中同时运行了 2 个数据处理过程，分别由数据信号 1 和数据信号 2 触发执行，他们具有不同的优先级别，每个信号进入任务节点时，首先查询任务节点的数据信号表，得到相应处理代码段的入口地址 data signal code address，同时在局部数据区为此数据信号申请相应的局部数据 data area，这样不同的数据信号可以触发同一任务节点的不同的处理代码段的运行。当任务节点处理完毕之后将处理的结果数据，以数据信号的方式返回给数据通道管理模块，数据通道管理模块，根据返回的数据信号，接着执行下面的操作。可见同一个任务节点的运行级别完全是根据触发其运行的数据信号的级别决定的。从数据的信号流入，到任务节点代码的执行，再到数据信号的流出的整个过程是通过任务调度实现的。这样的任务节点的运行模式优点是任务管理方式非常灵活，任务节点之间以数据信号为唯一的交互方式，节点之间的耦合度非常小，非常便于系统模块化设计，任务接点平时处于休眠静止状态，不占用 CPU 资源，而传统的嵌入式系统内核的每个任务基本上都要占用 CPU 的资源，这样大大地提高的系统的 CPU 利用率； 缺点是如果任务节点划分过多，数据信号流动过于频繁，势必会导致系统任务调度的频繁操作，这样也降低系统的运行效率。

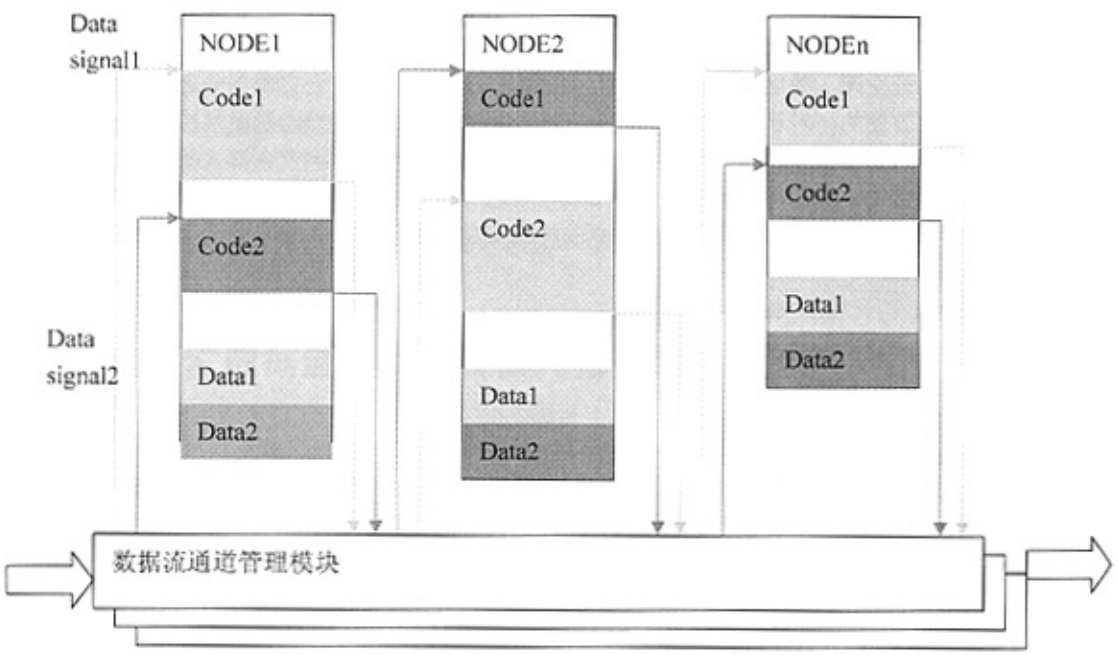


图 3.4 任务节点的运行方式和任务节点之间的关系

3）数据通道的管理

数据通道主要负责管理系统数据流的调度管理。是数据流驱动的核心，这里通过 5 个先进先出的数据队列组成。每个队列对应了同一个优先级别的数据信号。其结构如图 3.5 所示。数据通道管理模块维护了一个全局数据信号表结构，其中主要存放了，系统所有使用的数据信号相关信息，保存了每个数据信号对应的优先级别和每个数据信号所对应的任务节点。数据信号的缓冲队列有 4 个优级别，数据通道管理模块总是处理优先级别最高的数据信号缓冲队列。只有高优先级别的数据信号处理完毕，才会处理低优先级别的数据信号。对于同一级别的数据信号按照先进先出的方式处理。这四个等级分别是：1 故障处理级，2 功能级，3 基本级，4 维护级。故障处理级主要处理实时内核的一些系统级故障，当系统产生了比较严重的错误时，会优先处理。功能级主要负责处理和内核功能相关的一些任务的处理。基本级主要完成用户应用程序的数据处理。维护级主要负责和系统维护相关的任务处理。时间队列主要完成和应用程序相关的时间计数和时间延时功能。

数据通道管理队列的每一项成为数据队列控制块，其定义如下：

```
typedef struct{
    int     Data_s_id;
    Type    Data_type;
    Int     Data_size;
    Int     Element_type;
    Data    *Data_buffer;
    Task    *Node_task;
    State   Data_s_state;
    }QCB;
```

Data_s_id 存储本数据信号的 id 号，Data_type 说明了数据信号所带数的类型，Data_size 存放数据信号所带数据长度，指针*Data_buffer 指向数据信号所带数据的地址，指针*Node_task 指向数据信号待触发的任务节点的地址，Data_s_state 存储当前数据信号的状态。

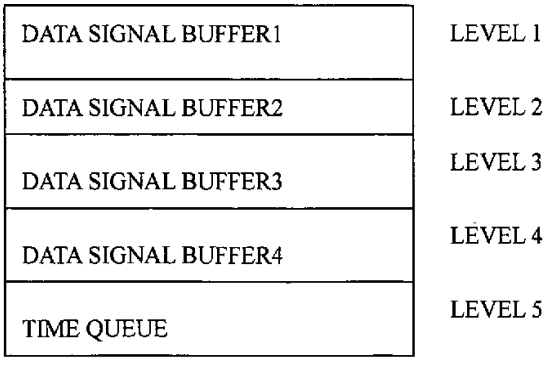


图 3.5 数据通道管理队列

4)调度策略

当系统内部有多个任务需要同时运行的时候，就需要调度功能来决定哪个任务运行。调度是指为任务分配资源和时间，使系统满足特定的性能要求。调度算法的目的是在正常情况下，尽可能满足所有任务的时限；在峰值负载条件下，保证实时任务满足时限。因为时限是区分实时系统和非实时系统的关键因素，因此调度算法是实时系统的基本问题。实时操作系统所具有的运行性能，如吞吐量的

大小、周转时间的长短、相应的及时性和可预测性等在很大程度上都取决于实时调度。在数据流驱动的实时内核micro-DDRTOS中，一个任务的开始运行是依靠数据信号来触发。一般的嵌入式系统的调度策略一般采用静态或者动态优先级策略，高优先级的任务可以首先得到系统的运行。也有一些实时操作系统，支持事件触发的消息队列和邮箱。与这些系统不同，micro-DDRTOS的任务调度策略由数据通道管理来完成，不同优先级的数据信号在缓冲队列中排队，按照先进先出的原则，优先级最高的数据信号送入对应的任务节点，通过调度程序，引发相应的处理任务的执行，此时相应的的任务处在激活的状态，对于CPU拥有控制权限，当数据处理完毕之后，任务节点将保存相应上下文数据，让出控制权限返回给数据通道管理模块，实际上我们是把一个数据任务的处理整个过程分割到若干任务节点中去完成，如图3.6所示，所有的数据输入的数据信号就有相同的优先级别，当其在相应的任务节点处理完毕之后，就会以数据信号的方式返回给数据通道管理这个时候此数据处理任务就会排在数据通道缓冲队列的末尾，从而其优先级排在同优先级的最后，这种策略实际上是将系统资源平均分配给相同优先级的任务。对于不同优先级的数据处理任务，则采用固定优先级的策略。因此micro-DDRTOS实际上采用的时固定优先级加动态优先级混合的策略。

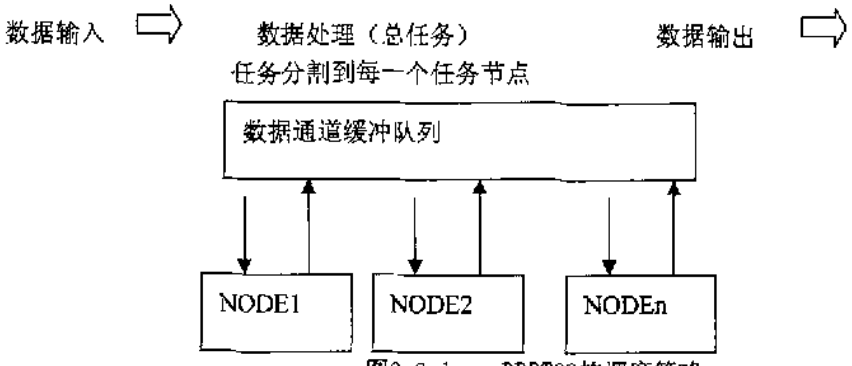


图3.6micro-DDRTOS的调度策略

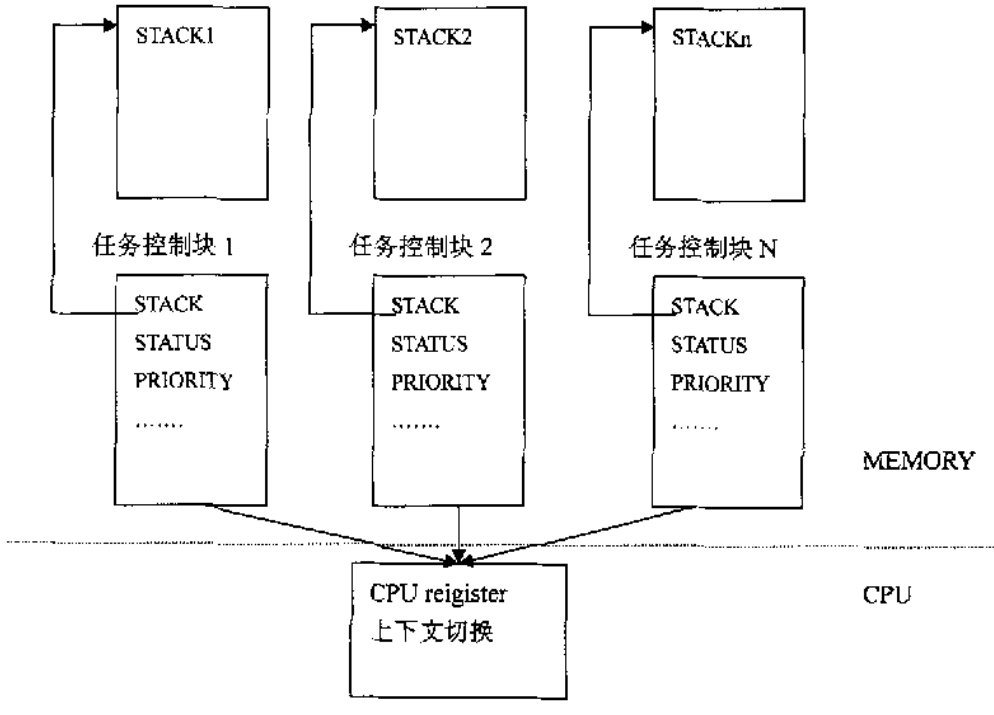


图3.7μC/OS-II多任务结构

micro-DDRTOS的任务调度策略是在μC/OS-II的基础上改造完成的。下面介绍μC/OS-II的任务调度方法。如图3.7。

μC/OS-II的每个人任务拥有独立的堆栈空间。每个任务都处在以下5种状态之一的状态下，这5种状态是休眠态，就绪态、运行态、挂起态(等待某一事件发生)和被中断态（参见图3.8）休眠态相当于该任务驻留在内存中，但并不被多任务内核所调度。就绪意味着该任务已经准备好，可以运行了，但由于该任务的优先级比正在运行的任务的优先级低，还暂时不能运行。运行态的任务是指该任务掌握了CPU的控制权，正在运行中。挂起状态也可以叫做等待事件态WAITING，指该任务在等待，等待某一事件的发生，（例如等待某外设的I/O操作，等待某共享资源由暂不能使用变成能使用状态，等待定时脉冲的到来或等待超时信号的到来以结束目前的等待，等等）。最后，发生中断时，CPU提供相应的中断服务，原来正在运行的任务暂不能运行，就进入了被中断状态。

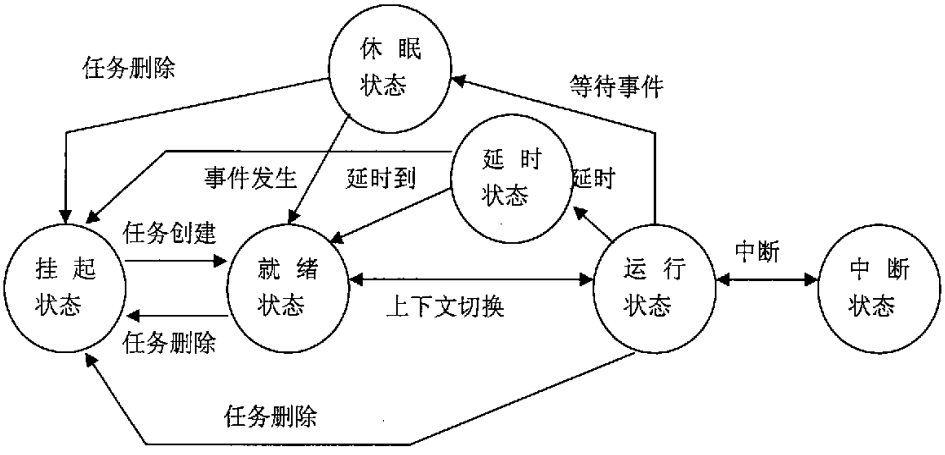


图3.8 μC/OS-II任务状态

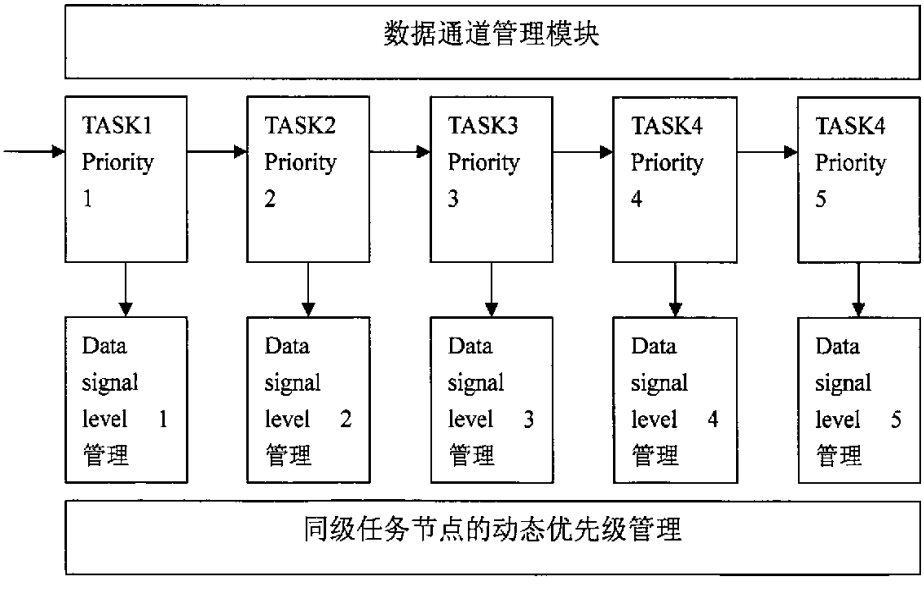


图 3.9 数据通道的实现方法

μC/OS-II 是一个占先式的内核，即已经准备就绪的高优先级任务可以剥夺正在运行的低优先级任务的 CPU 使用权。这个特点使得它的实时性比非占先式的内核要好，μC/OS-II 不支持时间片轮

转法。它是一个基于优先级的实时操作系统。每一个任务的优先级必须不同，μC/OS-II 把任务的优先级当作任务的标识来使用，如果优先级相同，任务将无法区分。进入就绪态的优先级最高的任务首先得到 CPU 的使用权，只有等它交出 CPU 的使用权后，其他任务才可以被执行。所以它是多任务，不能说是多进程，μC/OS-II 的这种特性是好是坏，主要看从什么角度来判断。如果只考虑实时性，比分时系统好，它可以保证重要任务总是优先占有 CPU。在此基础上，对于μC/OS-II 的固定优先级的调度策略进行改造使其适用于数据流驱动的内核模式。

如图 3.9 所示，数据通道管理分别由四个不同优先级别μC/OS-II 任务实现。每个任务负责管理不同级别的数据信号缓冲队列。通过μC/OS-II 固定优先级的调度方式，实现了数据信号的优先级管理，对于同一优先级的数据信号，通过 FIFO 的缓冲队列和任务分割实现了同一级别的动态优先级管理。

3.4 微内核扩展架构

这里将面向数据流驱动的微内核 micro-DDRTOS 在 ARM 硬件平台上实现，并在此基础之上，加入 TCP/IP 协议栈、GUI 图形用户界面和文件系统，使之成为一个比较完整的应用微内核系统。针对与不同的应用，只需要编制和添加相应的数据处理节点模块，就可以直接应用于实际产品和设备之中。其整体系统架构如图 3.10 所示。这里 TCP/IP 协议栈、GUI 图形用户界面和文件系统是是微内核的一部分，作为可选扩展部分，可以通过设置配置选项选择。

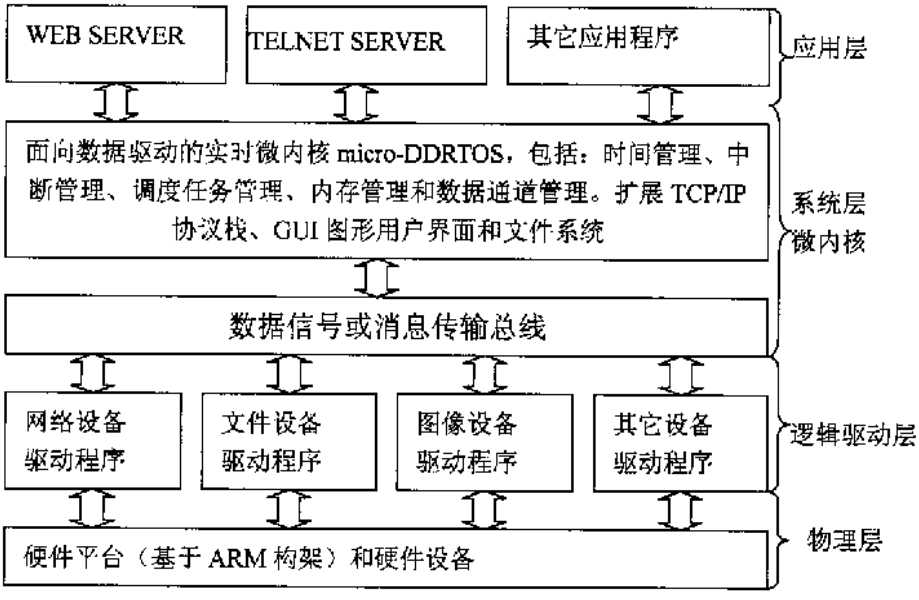


图 3.10 嵌入式系统整体系统架构逻辑框图

下面分别从运行视图（方面）和代码视图两个方面给出微内核系统的设计方法。

系统运行架构视图如图 3.11 所示，图中有两个比较重要的参数需要根据具体情况进行设计，系统 time slot,这个时间相当于内核运行的心跳，其时间的设置范围主要是根据硬件定时器/计数器的定时范围。一般从 0.1ms-10s。处理时限 limit time，主要是设置数据处理节点处理一个数据信号的最大处理时限，这主要是为了系统实时性能的要求决定的。从这里可以看出系统的整体运行调度策略采用了静态优先级的方法，其优先级从左到右依次递减。而在最高优先级的数据通道处理任务中，通过面向数据驱动的任务管理方式，实现了动态优先级的抢占式管理模式。这种方式不同于 Linux 这样的通用操作系统，在对线程和进程的 CPU 占用上采用了“公平”调度策略。通用操作系统的策略能够提供良好的整体表现，但是不能保证实时性的要求，其结果可能造成对时间要求严格的线程很

容易地错过它们的最终期限，甚至在一个高速的高端处理器上运行时也会出现这种情况。 而在micro-DDROTS 中，线程按照其优先级顺序执行。如果一个高优先级的线程准备运行时，它将在一个短的、有限时间间隔内从任何可能正在运行的低优先级进程接管 CPU。另外，高优先级的线程能够不被中断地运行，直到它已经完成了需要做的事情－当然是在不被更高优先级进程抢占的前提下。这种方法就是抢占式调度，保证了高优先级线程始终满足其最终期限，而不管有多少其它线程正在竞争 CPU 时间。 通过合理地控制任务优先级，能显著地提高很多对用户非常重要的应用响应速度。然而，控制优先级可能是一把双刃剑，当使用不当时它可能会潜在地导致低优先级的进程不能得到 CPU 时间。通过固定静态优先级调度+动态抢占式优先级调度策略的方法保证高优先级任务的同时确保不会使其它任务处于“饥饿”状态，通过对 time slot 和 limit time 的调整，能够限制这些任务消耗的 CPU 时间比例，既保证了系统实时性能和可预测的性能，又保证的低优先级任务的有效运行。Task state 和 task idle 分别为系统的统计和系统空闲任务，其中 taks state 统计了内核的运行效率，分别对于 CPU 的利用率、任务个数、每个任务占用系统时间、任务切换速率等方面给出了系统的统计数据，便于对于系统的运行状态进行监测。

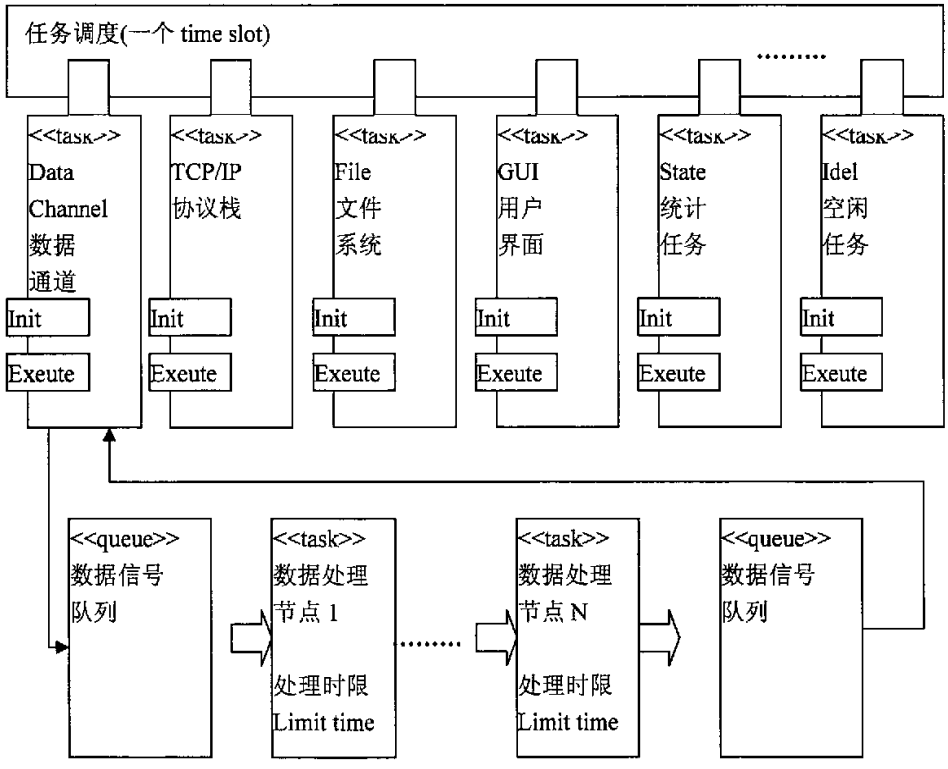


图 3.11 微内核运行视图

micro-DDRTOS 的代码管理如图 3.12 所示，管理模式以目录树形结构为主。每个子系统都包括了文档说明、测试文档、版本控制、源代码文档、编译链接相关文档和源代码库函数等。这样的优点是代码管理结构明确简单，子系统可以单独管理和处理，当某一子系统改变或者升级时不会影响整体。[6][7][8]

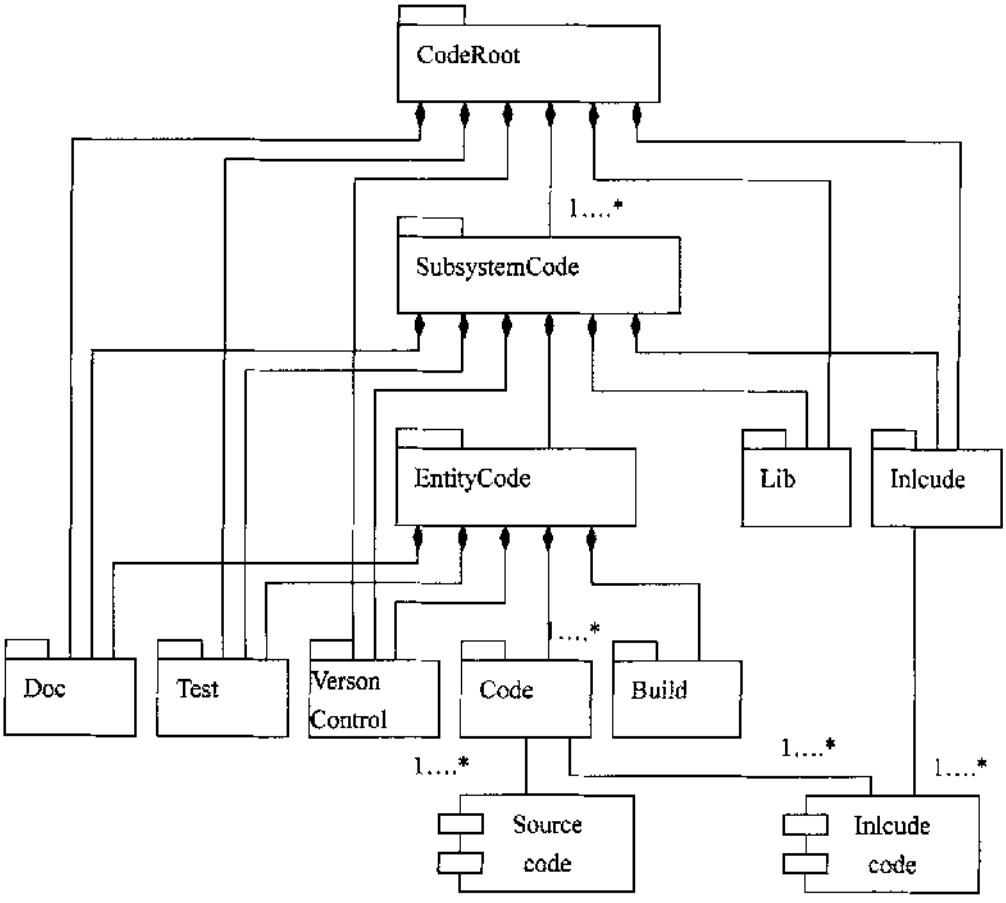


图 3.12 代码管理架构

3.5 系统的硬件基础

这个嵌入式实时操作微内核及其扩展应用是建立在ARM的硬件体系结构之上的。1991年ARM公司成立于英国剑桥，主要出售芯片设计技术的授权。目前，采用ARM技术知识产权（IP）核的微处理器，即我们通常所说的ARM微处理器，已遍及工业控制、消费类电子产品、通信系统、网络系统、无线系统等各类产品市场，基于ARM技术的微处理器应用约占据了32位RISC微处理器75%以上的市场份额，ARM技术正在逐步渗入到我们生活的各个方面。[4][10]

ARM公司是专门从事基于RISC技术芯片设计开发的公司，作为知识产权供应商，本身不直接从事芯片生产，靠转让设计许可由合作公司生产各具特色的芯片，世界各大半导体生产商从ARM公司购买其设计的ARM微处理器核，根据各自不同的应用领域，加入适当的外围电路，从而形成自己的ARM微处理器芯片进入市场。目前，全世界有几十家大的半导体公司都使用ARM公司的授权，因此既使得ARM技术获得更多的第三方工具、制造、软件的支持，又使整个系统成本降低，使产品更容易进入市场被消费者所接受，更具有竞争力。到目前为止，ARM微处理器及技术的应用几乎已经深入到各个领域：

- 工业控制领域：作为32的RISC架构，基于ARM核的微控制器芯片不但占据了高端微控制器市场的大部分市场份额，同时也逐渐向低端微控制器应用领域扩展，ARM微控制器的低功耗、高性价比，向传统的8位/16位微控制器提出了挑战。
- 无线通讯领域：目前已有超过85%的无线通讯设备采用了ARM技术，ARM以其高性能和低

成本，在该领域的地位日益巩固。

- 网络应用：随着宽带技术的推广，采用 ARM 技术的 ADSL 芯片正逐步获得竞争优势。此外，ARM 在语音及视频处理上行了优化，并获得广泛支持，也对 DSP 的应用领域提出了挑战。
- 消费类电子产品：ARM 技术在目前流行的数字音频播放器、数字机顶盒和游戏机中得到广泛采用。
- 成像和安全产品：现在流行的数码相机和打印机中绝大部分采用 ARM 技术。手机中的 32 位 SIM 智能卡也采用了 ARM 技术。

ARM 微处理器的特点：

1. 采用 RISC 架构的 ARM 微处理器特点：

1) 体积小、低功耗、低成本、高性能；
2) 支持 Thumb（16 位）/ARM（32 位）双指令集，能很好的兼容 8 位/16 位器件；
3) 大量使用寄存器，指令执行速度更快；
4) 大多数数据操作都在寄存器中完成；
5) 寻址方式灵活简单，执行效率高；
6) 指令长度固定；

ARM 微处理器目前包括下面几个系列，以及其它厂商基于 ARM 体系结构的处理器，除了具有 ARM 体系结构的共同特点以外，每一个系列的 ARM 微处理器都有各自的特点和应用领域。

- ARM7 系列
- ARM9 系列
- ARM9E 系列
- ARM10E 系列
- SecurCore 系列
- Inter 的 Xscale
- Inter 的 StrongARM

其中，ARM7、ARM9、ARM9E 和 ARM10 为 4 个通用处理器系列，每一个系列提供一套相对独特的性能来满足不同应用领域的需求。这里的实现硬件基础是 ARM7 通用处理器结构。

ARM7 系列微处理器为低功耗的 32 位 RISC 处理器，最适合用于对价位和功耗要求较高的消费类应用。ARM7 微处理器系列具有如下特点：

- 具有嵌入式 ICE－RT 逻辑，调试开发方便。
- 极低的功耗，适合对功耗要求较高的应用，如便携式产品。
- 能够提供 0.9MIPS/MHz 的三级流水线结构。
- 代码密度高并兼容 16 位的 Thumb 指令集。
- 对操作系统的支持广泛，包括 Windows CE、Linux、Palm OS 等。
- 指令系统与 ARM9 系列、ARM9E 系列和 ARM10E 系列兼容，便于用户的产品升级换代。
- 主频最高可达 130MIPS，高速的运算处理能力能胜任绝大多数的复杂应用。

ARM7 系列微处理器的主要应用领域为：工业控制、Internet 设备、网络和调制解调器设备、移动电话等多种多媒体和嵌入式应用。

到目前为止，RISC 体系结构也还没有严格的定义，一般认为，RISC 体系结构应具有如下特点：

- 采用固定长度的指令格式，指令归整、简单、基本寻址方式有 2～3 种。
- 使用单周期指令，便于流水线操作执行。
- 大量使用寄存器，数据处理指令只对寄存器进行操作，只有加载/ 存储指令可以访问存储器，以提高指令的执行效率。
- 除此以外，ARM 体系结构还采用了一些特别的技术，在保证高性能的前提下尽量缩小芯片的面积，并降低功耗：

- 所有的指令都可根据前面的执行结果决定是否被执行，从而提高指令的执行效率。
- 可用加载/存储指令批量传输数据，以提高数据的传输效率。
- 可在一条数据处理指令中同时完成逻辑处理和移位处理。
- 在循环处理中使用地址的自动增减来提高运行效率。

ARM7 系列微处理器包括如下几种类型的核：ARM7TDMI、ARM7TDMI-S、ARM720T、ARM7EJ。其中，ARM7TMDI 是目前使用最广泛的 32 位嵌入式 RISC 处理器，属低端 ARM 处理器核。TDMI 的基本含义为：

T：支持 16 为压缩指令集 Thumb；

D：支持片上 Debug；

M：内嵌硬件乘法器（Multiplier）

I：嵌入式 ICE，支持片上断点和调试点；

2. ARM处理器模式

ARM微处理器支持7种运行模式，分别为：

- 用户模式（usr）：ARM处理器正常的程序执行状态
- 快速中断模式（fiq）：用于高速数据传输或通道处理
- 外部中断模式（irq）：用于通用的中断处理
- 管理模式（svc）：操作系统使用的保护模式
- 数据访问终止模式(abt)：当数据或指令预取终止时进入该模式，可用于虚拟存储及存储保护。
- 系统模式（sys）：运行具有特权的操作系统任务。
- 未定义指令中止模式（und）：当未定义的指令执行时进入该模式，可用于支持硬件协处理器的软件仿真。

ARM微处理器的运行模式可以通过软件改变，也可以通过外部中断或异常处理改变。

大多数的应用程序运行在用户模式下，当处理器运行在用户模式下时，某些被保护的系统资源是不能被访问的。除用户模式以外，其余的所有6种模式称之为非用户模式，或特权模式（Privileged Modes）；其中除去用户模式和系统模式以外的5种又称为异常模式（Exception Modes），常用于处理中断或异常，以及需要访问受保护的系统资源等情况。

3. ARM 寄存器组织

如图 3.13，ARM 处理器共有 37 个寄存器，被分为若干个组（BANK），这些寄存器包括：31 个通用寄存器，包括程序计数器（PC 指针），均为 32 位的寄存器。6 个状态寄存器，用以标识 CPU 的工作状态及程序的运行状态，均为 32 位，目前只使用了其中的一部分。同时，ARM 处理器又有 7 种不同的处理器模式，在每一种处理器模式下均有一组相应的寄存器与之对应。即在任意一种处理器模式下，可访问的寄存器包括 15 个通用寄存器（R0～R14）、一至二个状态寄存器和程序计数器。在所有的寄存器中，有些是在 7 种处理器模式下共用的同一个物理寄存器，而有些寄存器则是在不同的处理器模式下有不同的物理寄存器。

图 3.13ARM 寄存器结构

ARM 微处理器的在较新的体系结构中支持两种指令集：ARM 指令集和 Thumb 指令集。其中，ARM 指令为 32 位的长度，Thumb 指令为 16 位长度。Thumb 指令集为 ARM 指令集的功能子集，但与等价的 ARM 代码相比较，可节省 30%～40%以上的存储空间，同时具备 32 位代码的所有优点。

第四章 微内核的实现

实时多任务操作系统微内核是嵌入式应用的基础平台，实际上，此时的嵌入式操作系统相当于一个通用而复杂的主控程序，为嵌入式应用软件提供更强大的开发平台和运行环境。嵌入式内核系统已经将处理器、中断、定时器、I/O 等资源包装起来，用一系列的 API 提供给用户，应用程序可以不关注底层硬件，直接借用操作系统提供的功能进行开发。本章主要讲述了 Micro-DDROTS 的具体实现方法，详细介绍的系统内核每一部分的 API 函数，这些函数提供了和系统应用层的接口界面。

4.1 中断管理

嵌入式实时系统对外部事件的响应一般都是通过中断来处理的，其对中断的处理方式，直接影响到系统的实时性能。这里为了方便对中断的处理，中断的处理成为 micro-DDRTOS 实时内核的一部分，提供一些系统调用接口来安装用户的中断，提供统一的中断处理接口。根据系统内核的可抢占或者非抢占性，系统内核接管中断又有两种不同处理模式。在非抢占式内核的中断处理模式中，当在中断处理过程中有高优先级任务就绪时，不会立即切换到高优先级的任务，必须等待中断处理完后返回到被中断的任务中，等待被中断的任务执行完后，再切换到高优先级任务。在抢占式内核的中断处理模式中，如果有高优先级任务就绪时，则立刻切换到高优先级的任务。这里采用了后者的处理方法。[23]

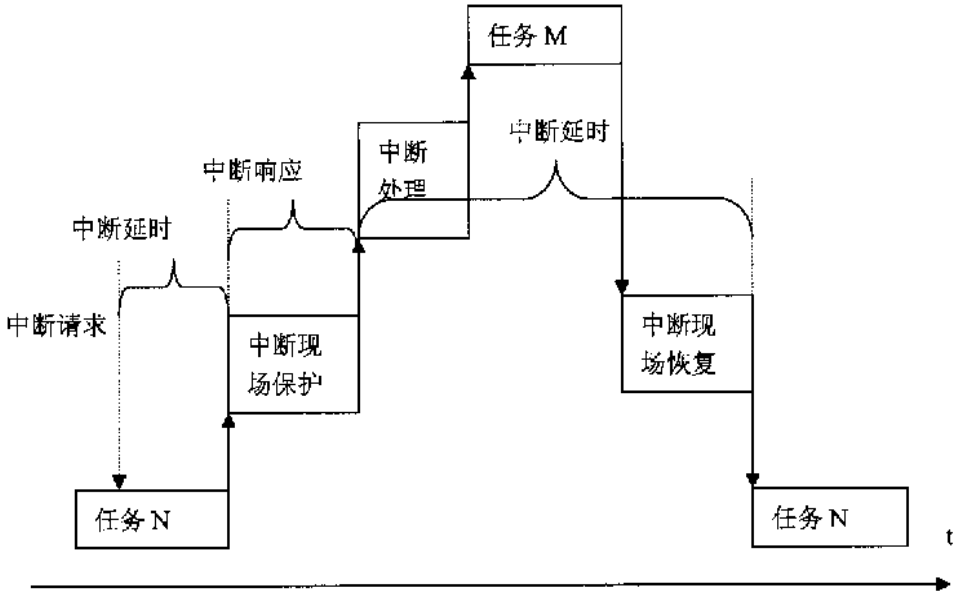


图 4.1 内核中断处理模式

处理模式如图 4.1 所示。在图中，中断的产生停止了任务 N 的运行。在中断处理的时候，更高级别的任务 M 就绪，切换到 M 执行，这种处理模式有利于高优先级任务的处理，但相应地延长了被中断的低优先级任务的执行时间。

嵌入式内核接管中断的处理机制主要包括两个部分：面向应用的编程接口部分和面向底层的处理部分。面向用户应用的编程接口的任务之一是供支持用户安装中断处理例程。面向底层处理部分

可以分为两个部分：中断向量表部分和中断处理部分。中断向量表部分主要指中断向量表的定位和向量表中表项内容的形式，一般在嵌入式内核中都提供一个中断向量表， 其表项的向量号与处理器中所描述的向量对应；向量表表项的内容形式一般有两种形式。最常见的形式就是在具体的向量位置存储的是一些转移程序，转到具体的中断处理部分；另一种形式也就是中断向量位置存放具体的中断处理程序，此仅针对向量号之间彼此有一定的距离，此距离足以存放中断处理程序。面向底层部分的中断处理部分，是整个嵌入式内核中断管理的核心。内核中断处理部分采用了“统一接管”的思想，即内核为所有的外部中断都提供一个统一的入口_ISR_Handler。此入口的主要功能是保护中断现场，执行用户的中断服务程序，判断是否允许可抢占调度，中断现场的恢复等。内核中断处理的流程如图 4.2。从内核中断处理流程图中，可看出嵌入式内核中一些专用的处理方式。

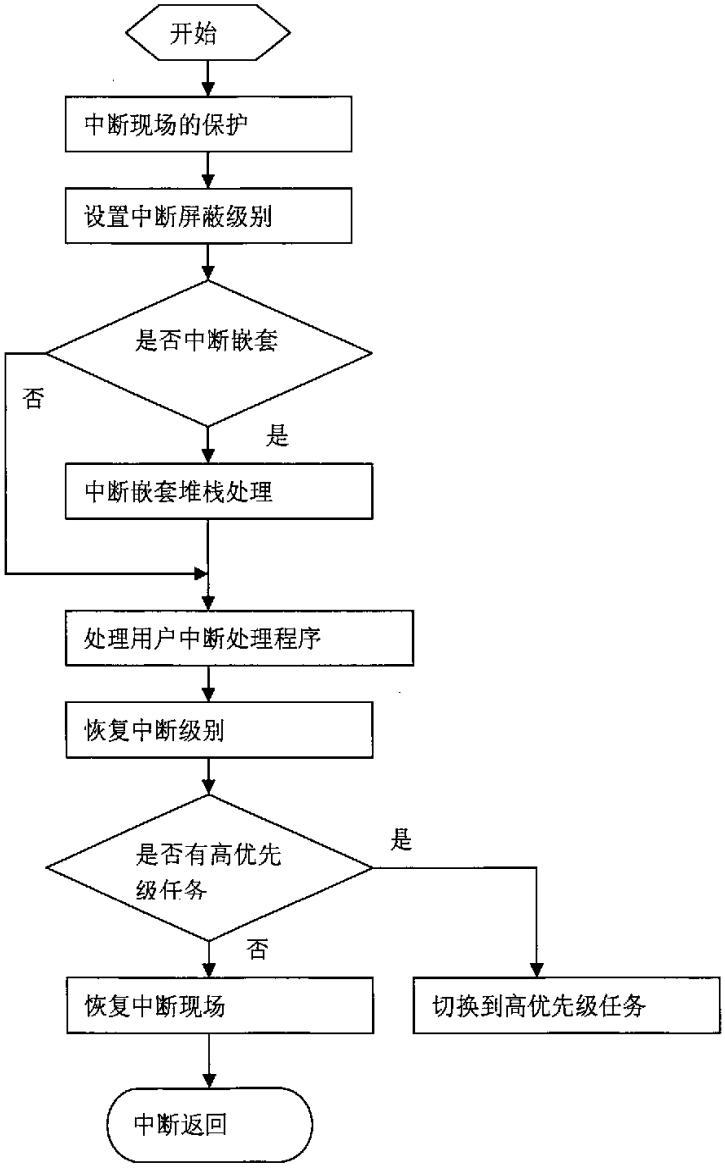


图 4.2 内核中断处理流程

① 在嵌入式内核中一般有两个堆栈：系统栈和任务栈。系统栈是系统为中断上下文处理而预留

的堆栈；任务栈属于任务本身的私有堆栈，用来存储任务执行过程中一些临时变量等信息。因为中断上下文不隶属于任何任务的上下文中，所以嵌入式内核一般都有一个系统栈专门处理中断上下文的。当产生中断且非中断嵌套时，堆栈由被中断任务中的任务栈切换到系统栈；当在中断处理中又发生中断时，堆栈不再切换，仍用系统栈；当退出最外层中断时，堆栈又由系统栈切换到被中断的任务中的任务栈。

② 一般嵌入式内核有两种形式：抢占式和非抢占式。为了更好地支持系统的实时性，实时内核采用了抢占式，内核中断处理流程图可知其支持了抢占式内核，因为在中断处理中，当检测到有高优先级任务就绪时，就会切换到高优先级任务里，而不是等到退出中断后，再进行任务调度。

③ 在嵌入式内核中，中断时机和调度时机直接影响到系统的实时性。关中断的时机一般在执行核心操作之前。核心操作包括对链表的操作，对核心数据项（如指示同步，反应重要信息状态）的修改等场合都须关中断。执行完相应的核心操作后，就可以开中断。开调度时机主要提供重新调度的机会，一般在执行操作系统核心调用前关调度，执行完后开调度。系统中开关中断与开关调度的关系大致如下：开关中断的粒度比开关调度要深，要细。开关中断主要是为实时性提供各种可能的中断时机，允许响应外部中断。中断里也可以执行调度和系统调用，但中断的上下文与任务的上下文是不一样的，因此在中断里只能执行一些特定的系统调用。这些特定系统调用是不会引起调用阻塞的，不要试图在中断里执行获取信号量，执行 I/O 操作等这些很容易引起调用阻塞的系统调用。

这里嵌入式内核一般采用中断统一接管思想，在中断统一接管中调用用户的中断服务程序。中断管理模式中的中断处理部分又可以细化，这里采用了嵌入式 Linux 系统中关于中断管理机制中提出了“前半部”和“后半部”的处理思想。这种中断管理的思想把中断处理部分按照重要性分两部分，将必须要做的中断处理部分归为“前半部”，即这部分在中断处理部分实施；而将中断处理中可以延迟操作且影响不大的部分归为“后半部”，这部分在退出中断服务程序后实施。通过这样的中断管理思想减少的中断服务时间，为其它外部事件的中断响应提供了更多的时机。在实时内核中还有其它的中断处理机制，它们的都是尽量减少中断处理的时间。如在一些 I/O 处理部分，I/O 操作所引起的中断处理部分只做标记功能，即只设一个标志或者发一个消息说明外部中断来了，而具体的 I/O 传输操作放在中断外部实施。根据上面的分析，将前面的中断处理思想归结为：中断“前一后”段处理模型。“中断前部”主要完成外部事件发生中断请求时，系统对其响应所完成的必要功能，如中断现场保护、数据预取和预放等；“置标”部分主要通知某个任务或者线程已有一个中断发生，且中断的前部已完成；“中断后部”并不是在中断服务程序里执行，而是由接收到标记或者通知的任务或者线程来完成的，主要是完成本应在中断服务里完成的后继工作。举个例子，当网络接口卡报告新的数据包到达时，“中断前部”主要将数据包送到协议层；“中断后部”完成对数据包的具体处理，其中应该注意两个方面：

① 如何划分“中断前部”和“中断后部”。基本的划分标准是，应该立即处理的和必要的功能部分放在“中断前部”完成，可以推迟处理或者可以在中断外处理的功能部分放在“中断后部”完成。

②“中断后部”何时执行，取决于用于完成“中断后部”功能的任务或者线程的优先级。如果要让中断的后继部分较快地执行，则可以通过提高获得标记的任务或者线程的优先级。从极限角度思维，当获得标记的任务或者优先级很高时，在“中断前部”完成退出中断后，立即就执行获得标记的任务或者线程，这相当于获得标记的任务或者线程执行部分就在中断里执行。如果中断的后继部分并不要求较快的执行，则可以赋给获得标记的任务或者线程为普通的优先级。

在前面的嵌入式内核中断管理模式图中，中断向量表部分也属于模式图的一部分，不同嵌入式处理器体系中断向量的支持也不同。为了处理这种多个外部中断共用一个向量的情况，系统内核使用了单向量多中断处理方法，此方法的思想如下：当外设中断触发时，首先定位到实向量位置，调用中断统一接口函数，中断统一接口函数对外设中断触发的参数进行测试，寻找到其对应的虚向量，从而触发虚向量处的回调函数，从而实现多个外部中断通过同一的实向量到多个虚向量的映射，解

决了单向量多中断处理的问题。

下面具体讨论micro-DDROTS的中断处理的具体实现方法。由于micro-DDRTOS是建立在ARM硬件平台之上，这里面有两个中断入口地址，IRQ和FRQ，由于系统实时性能的要求，这部分代码由汇编指令完成。

底层硬件中断服务程序：

```
OS_CPU_IRQ_ISR
    STMFD   SP!, {R1-R3}  ; 保存工作寄存器进入系统堆栈
    MOV     R1, SP        ; 保存堆栈指针
    ADD     SP, SP,#12    ; 调节堆栈指针
    SUB     R2, LR,#4     ; 调节中断返回地址指针
    MRS     R3, SPSR      ; 备份 SPSR (i.e. interrupted task's CPSR) 到 R3
    MSR     CPSR_c, #(NO_INT | SVC32_MODE) ; 切换 ARM CPU 工作模式，进入 SVC 模式
    STMFD   SP!, {R2}                      ;保存当前中断的任务的上下文，进入任务堆栈
    STMFD   SP!, {LR}                      ;
    STMFD   SP!, {R4-R12}                  ;
    LDMFD   R1!, {R4-R6}                   ;
    STMFD   SP!, {R4-R6}
    STMFD   SP!, {R0}                      ;
    STMFD   SP!, {R3}                      ;
    LDR     R0, OS_IntNesting              ;处理任务嵌套
    LDRB    R1, [R0]
    ADD     R1, R1,#1
    STRB    R1, [R0]
    CMP     R1, #1
    BNE     OS_CPU_IRQ_ISR_1
    LDR     R4, OS_TCBCur
    LDR     R5, [R4]
    STR     SP, [R5]
OS_CPU_IRQ_ISR_1
     MSR     CPSR_c, #(NO_INT | IRQ32_MODE) ; 切换 ARM CPU 工作模式，进入 IRQ 模式
     LDR     R0, ARMOS_CPU_IRQ_ISR_Handler   ;调用系统中断处理服务程序
     MOV     LR, PC
     BX      R0
     MSR     CPSR_c, #(NO_INT | SVC32_MODE) ; 切换 ARM CPU 工作模式，进入 SVC 模式
     LDR     R0, OS_IntExit                 ;处理中断返回
     MOV     LR, PC
     BX      R0
     LDMFD   SP!, {R4}                      ;恢复返回任务点的上下文
     MSR     SPSR_cxsf, R4
     LDMFD   SP!, {R0-R12,LR,PC}^           ;从任务堆栈弹出任务的上下文，返回任务
```

系统内核中断处理服务程序：

```
void OS_CPU_IRQ_ISR_Handler(void)
{
  unsigned status;
  status = *IRQStatus;  /*获取中断标志寄存器内容*/
```

```
  if (status & IRQTimer1) /*处理计数器定时器中断*/
  {
    *Timer1Clear = 0;
    OSTimeTick();       /*产生系统的节拍*/

  }
  else if (status & IRQTimer2) /*处理其它的中断，和相关操作*/
  {
    ..................
    ..................
}
```

系统中断相关 API 函数：

unsigned Install_Handler (unsigned routine, unsigned *vector)；/*中断向量装载程序*/

4.2 时间管理

在时间管理上面实时内核需要用户提供周期性信号源，用于实现时间延时和其它和时间有关的操作。这里利用 ARM 提供的两个计时器/计数器，提供了系统的基本节拍，节拍率在每秒 10KHZ 次到 20MHZ 次之间，时钟节拍率越高，系统的额外负荷就越重。时钟节拍的实际频率取决于用户应用程序的精度。用户必须在多任务系统启动以后再开启时钟节拍器。对于时间管理，基本采用了 μC/OS-II 的原有处理方法。这里提供了以下一些和时间相关的系统 API 函数调用。

`void OSTimeDly (INT16U ticks)`

这个函数的功能是，任务申请延时一段时间，这段时间的长短是用时钟节拍的数目 ticks 来确定的。调用该函数会使内核进行一次任务调度，并且执行下一个优先级最高的就绪态任务。任务调用 OSTimeDly()后，一旦规定的时间期满或者有其它的任务通过调用 OSTimeDlyResume()取消了延时，它就会马上进入就绪状态。注意，只有当该任务在所有就绪任务中具有最高的优先级时，它才会立即运行。

`INT8U OSTimeDlyHMSM (INT8U hours, INT8U minutes, INT8U seconds, INT16U milli)`

应用程序是通过用小时、分、秒和毫秒指定延时来调用该函数。已达到一个确定的延时时间。

`INT8U OSTimeDlyResume (INT8U prio)`

允许用户结束正处于延时期的任务，延时的任务可以不等待延时期满，而是通过其它任务取消延时来使自己处于就绪态。这可以通过调用 OSTimeDlyResume()和指定要恢复的任务的优先级来完成。实际上，OSTimeDlyResume()也可以唤醒正在等待事件在这种情况下，等待事件发生的任务会考虑是否终止等待事件。

`INT32U OSTimeGet (void)`

`void OSTimeSet (INT32U ticks)`

得到和改变系统时间

4.3 内存管理

在 ANSI C 中可以用 malloc()和 free()两个函数动态地分配内存和释放内存。但是，在嵌入式实时操作系统中，多次这样做会把原来很大的一块连续内存区域，逐渐地分割成许多非常小而且彼

此又不相邻的内存区域，也就是内存碎片。由于这些碎片的大量存在，使得程序到后来连非常小的内存也分配不到。另外，由于内存管理算法的原因，malloc()和 free()函数执行时间是不确定的。这里系统内核把连续的大块内存按分区来管理。每个分区中包含有整数个大小相同的内存块。利用这种机制，对 malloc()和 free()函数进行了改进，使得它们可以分配和释放固定大小的内存块，这样一来，malloc()和 free()函数的执行时间也是固定的了。如图 4.3，在一个系统中可以有多个内存分区。这样，用户的应用程序就可以从不同的内存分区中得到不同大小的内存块。但是，特定的内存块在释放时必须重新放回它以前所属于的内存分区。显然，采用这样的内存管理算法，上面的内存碎片问题就得到了解决。这种内存管理的方式非常类似于堆的分配策略。

这种方式采用固定内存分配块，可以解决内存碎片的问题，另一方面采用创建不同内存分配块大小的内存分区的方法，解决内存利用率的问题。

这里内存控制块的结构定义为：

```
typedef struct {
    void    *Mem_Addr; /*指向内存分区的指针*/
    void    *Mem_FreeList;/*指向下一个内存分区的指针*/
    INT32U  Mem_Block_Size;/*内存分区中可分配块的大小*/
    INT32U  Mem_NBlks;/*内存分区中内存块数量*/
    INT32U  Mem_NFree; /*内存分区中空闲内存块数量*/
    } MEM_Area;
```

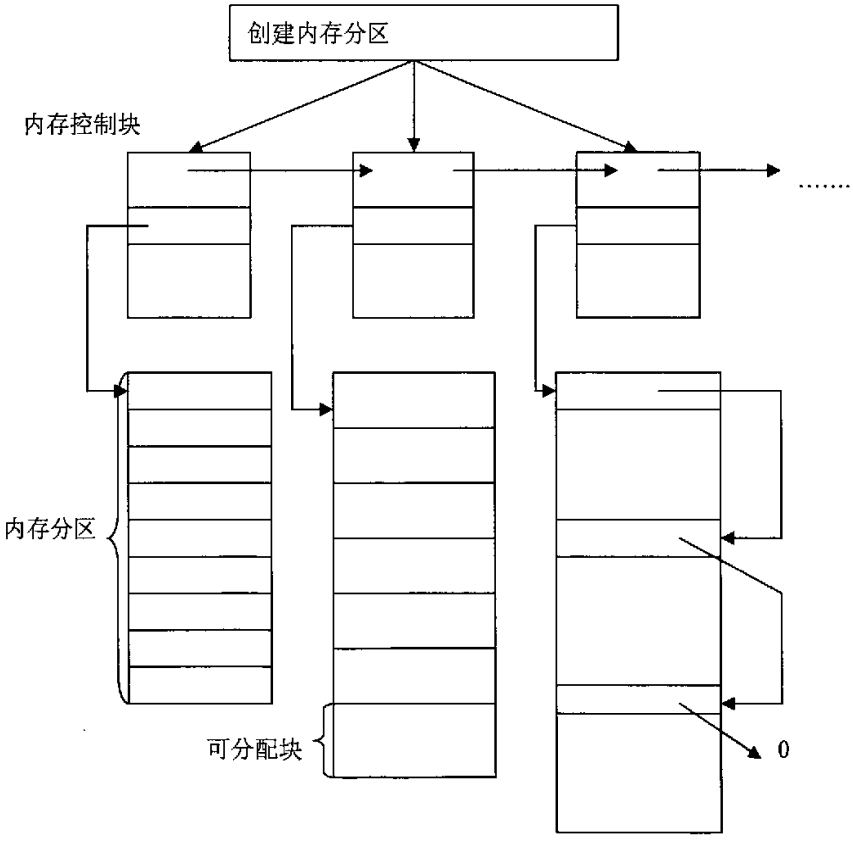


图 4.3 内存管理

相关 AIP 函数主要有：

MEM_Area *MemCreate (void *addr, INT32U nblks, INT32U blksize, INT8U *err)

建立一个内存分区，nblks 分区内部内存块的个数，blksize 每个内存块的大小。

void *MemGet (MEM_Area *pmem, INT8U *err)

在指定的内存分区 pmem 上面，分配一个内存块。应用程序可以调用 MemGet()函数从已经建立的内存分区中申请一个内存块。该函数的唯一参数是指向特定内存分区的指针，该指针在建立内存分区时，由 MemCreate()函数返回。显然，应用程序必须知道内存块的大小，并且在使用时不能超过该容量。例如，如果一个内存分区内的内存块为 32 字节，那么，应用程序最多只能使用该内存块中的 32 字节。

INT8U MemPut (MEM_Area *pmem, void *pblk)

释放一内存块，归还给指定的内存分区 pmem。 当用户应用程序不再使用一个内存块时，必须及时地把它释放并放回到相应的内存分区中。这个操作由 MemPut()函数完成。必须注意的是，MemPut()并不知道一个内存块是属于哪个内存分区的。例如，用户任务从一个包含 32 字节内存块的分区中分配了一个内存块，用完后，把它返还给了一个包含 120 字节内存块的内存分区。当用户应用程序下一次申请 120 字节分区中的一个内存块时，它会只得到 32 字节的可用空间，其它 88 字节属于其它的任务，这就有可能使系统崩溃。

INT8U MemQuery (OS_MEM *pmem, OS_MEM_DATA *pdata)

查询一个内存分区的状态，通过该函数可以知道特定内存分区中内存块的大小、可用内存块数和正在使用的内存块数等信息。

4.4 任务管理

Micro-DDROTS 内核主要有两种类型的任务，如图 4.4 所示，一种是数据流处理任务节点，它主要完成了处理数据信号流的数据，这些任务一般处于休眠状态，当数据通道分配数据信号进入任务节点时，数据处理任务进入运行状态，完成数据的相应处理功能，其原理已在前面的章节中详细叙述。另一种是完成系统其它基本功能的处理任务，例如 TCP/IP 协议栈、GUI 图形用户界面、文件系统等都是以基本任务的形式实现的。这里数据通道管理模块，也是以基本任务的方式实现的。这里对于基本任务的管理基本采用了 μ C/OS-II 的基本方式，然后在数据通道任务模块的基础之上，实现了数据处理任务节点的动态优先级数据处理任务节点的管理。

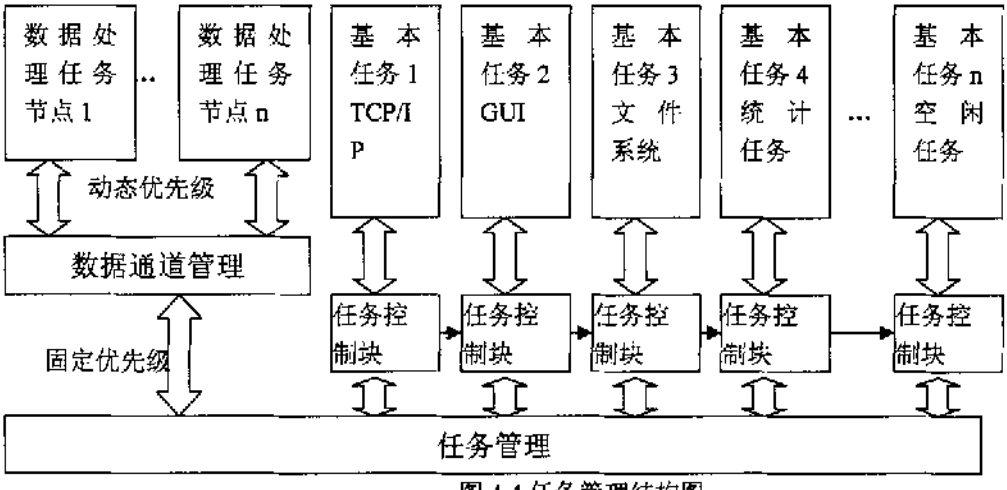


图 4.4 任务管理结构图

4.4.1 基本任务管理

1. 建立任务:

INT8U OSTaskCreate (void (*task)(void *pd), void *pdata, OS_STK *ptos, INT8U prio)

用户可以通过传递任务地址和其它参数到这个函数来建立任务，任务可以在多任务调度开始前建立，也可以在其它任务的执行过程中被建立。在开始多任务调度前，用户必须建立至少一个任务。任务不能由中断服务程序来建立。其中 pd 为待建立任务的入口地址，pdata 是当任务开始执行时传递给任务的参数的指针，ptos 为任务独立的堆栈空间，prio 为建立任务的优先级别。

2. 删除任务:

INT8U OSTaskDel (INT8U prio)

删除任务,是说任务将返回并处于休眠状态，并不是说任务的代码被删除了，只是任务的代码不再被调用。通过调用 OSTaskDel()就可以完成删除任务的功能。OSTaskDel()一开始应确保用户所要删除的任务并非是空闲任务，因为删除空闲任务是不允许的。接着，OSTaskDel()还应确保用户不是在 ISR 例程中去试图删除一个任务，因为这也是不被允许的。

3. 挂起任务:

INT8U OSTaskSuspend (INT8U prio)

任务挂起是很有用的。挂起任务可通过调用 OSTaskSuspend()函数来完成。被挂起的任务只能通过调用 OSTaskResume()函数来恢复。任务挂起是一个附加功能。也就是说，如果任务在被挂起的同时也在等待延时的期满，那么，挂起操作需要被取消，而任务继续等待延时期满，并转入就绪状态。任务可以挂起自己或者其它任务。

4. 恢复任务

INT8U OSTaskResume (INT8U prio)

被挂起的任务只有通过调用 OSTaskResume()才能恢复。 要恢复的任务必须是存在的，因为用户要需要操作它的任务控制块,并且该任务必须是被挂起的。查询 5. 任务信息

INT8U OSTaskQuery (INT8U prio, OS_TCB *pdata)

用户的应用程序可以通过调用 OSTaskQuery()来获得自身或其它应用任务的信息。实际上，OSTaskQuery()获得的是对应任务的任务控制块中内容的拷贝。

4.4.2 数据处理任务节点的管理

1. 建立数据处理任务节点:

INT8U Data_TaskCreate (void (*task)(void *pd), void *pdata, OS_STK *pto, Data_signal_talbe *Ds, Data_recoard *Dr)

数据处理任务可以通过 Data_TaskCreate()来建立。其中 pd 为数据处理任务的入口地址，pto 为独立堆栈空间，Ds 为数据处理任务的数据信号表。该表格记录了所有本数据处理任务处理节点的流入和流出数据信号，以及相应处理代码的入口地址。Dr 为该任务节点处理每一个数据信号所对应功能的单独数据区域。其建立过程如图 4.5 所示。

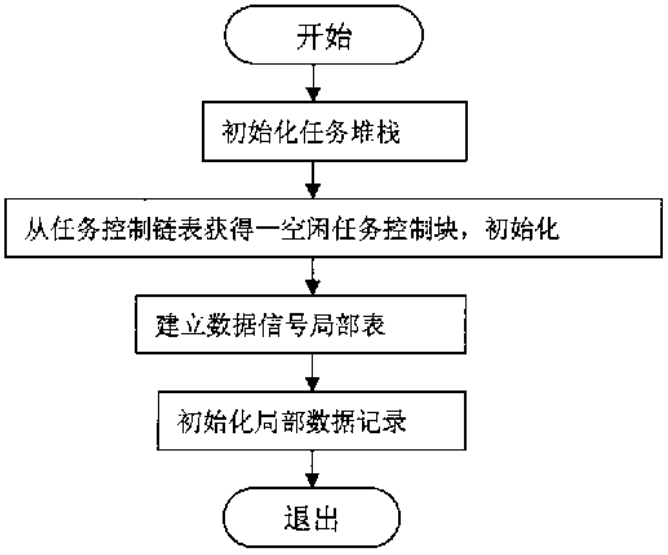


图 4.5 数据处理任务的建立过程

数据处理任务的运行模式如图 4.6 所示。

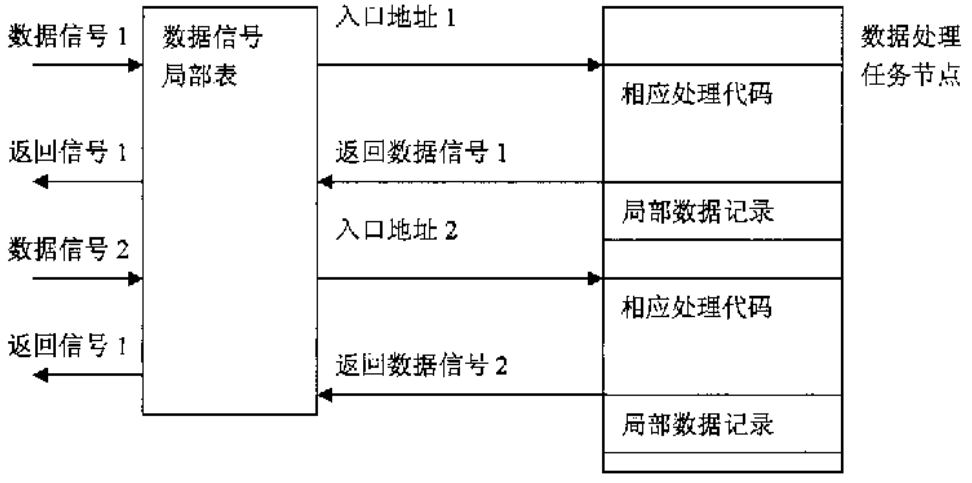


图 4.6 数据处理任务的运行模式

当数据信号通过数据通道管理分配给数据处理任务时，通过查询数据信号局部表，得到任务处理的相应入口地址，然后将数据信号及其所带的数据交给数据任务处理，处理的结果数据放入输出数据信号，再次查询数据局部表，得到相应参数，返回给数据通道，进行下一步处理过程。图中显示了两个数据信号同步处理的过程，每个处理过程都有一个局部数据区（局部数据记录）和其相对应，记录相应的处理中间数据。

2 删除数据处理任务节点

INT8U Data_TaskDel (void (*task)(void *pd), void *pdata)

将数据处理任务设置为休眠状态。

3 设置数据处理任务的期限时间

INT8U Data_Set_deadline(int time)

设置数据处理任务，处理某一数据信号的最大处理时间。当一数据处理任务调用此函数之后，其告诉内核在 time 的时间之内，数据信号将被处理完毕。

4 查询数据处理任务的期限时间
int Data_Check_deadline(void)
查询数据处理任务的处理时间，是否越界。

4.5 数据通道管理

在数据流驱动实时微内核中，数据流触发每个任务进入运行状态，数据流在内核系统中以数据信号作为载体，数据流主要是系统输入设备输入的待处理数据，经过系统 I/O 数据接口包装进入数据信号中，根据代处理数据（任务）的级别，分别进入数据通道管理模块，数据通道模块主要负责管理不同优先级别的 5 个 FIFO 数据信号队列，根据优先级策略，将最高优先级别的数据信号弹出队列发送给目标任务节点，任务节点加工处理数据信号所携带的数据，将处理结果还是以数据信号的方式返回给数据通道，数据通道根据返回的数据信号的优先级，将返回的数据信号推入相应优先级别 FIFO 队列。通过数据通道的管理模式，实现了实时内核的抢占式固定优先级别和同级之间动态优先级的调度策略。
void Create_Globle_Datasignal_Table(Data_singnal Ds[],Task_address Ta[]);
创建全局数据信号表，该全局表格主要记录了数据信号和任务节点之间的对应关系。
void Create_Data_Channel(Data_Control_Block Dcb, Data_Queue Dq,int Priority);
创建数据通道的数据信号队列，其中 priority 标志数据队列的优先级别。
void Data_Enter_Queue(Data_signal Ds，Task_address Ta);
数据信号进入数据通道队列。
void Data_Out_Queue(Data_signal Ds,Task ts, Task_address Ta);
数据信号出数据通道队列，本操作主要是弹出最高优先级别队列中的第一个元素，根据出队列的数据信号查询全局数据信号表，得到对其处理的对应任务节点指针 Ta。数据流进入任务节点之后，系统控制权限交给任务节点，此时触发一次任务调度切换操作。任务节点查询其局部数据信号表，得到对应处理代码的入口地址，这样待处理数据由相应的任务代码进行处理。
int Data_Channel_Size(Data_Queue Dq, int Priority);
返回数据信号队列的空间大小。
int Data_Channel_Element(Data_Queue Dq, int Priority);
返回数据信号队列的元素个数。
Void Data_Channel_Empty(Data_Queue Dq, int Priority);
清空数据信号队列。

4.6 系统复位管理

在系统复位的过程中，主要完成的系统的初始化功能，完成了系统的主要数据结构的建立，主要有：全局数据信号表、任务控制块链表，数据通道的数据信号队列的建立。数据处理任务节点的部署、中断的处理程序的初始化、任务管理模块的初始化和其它所有扩展任务的初始化工作。处理过程如图 4.7 所示。

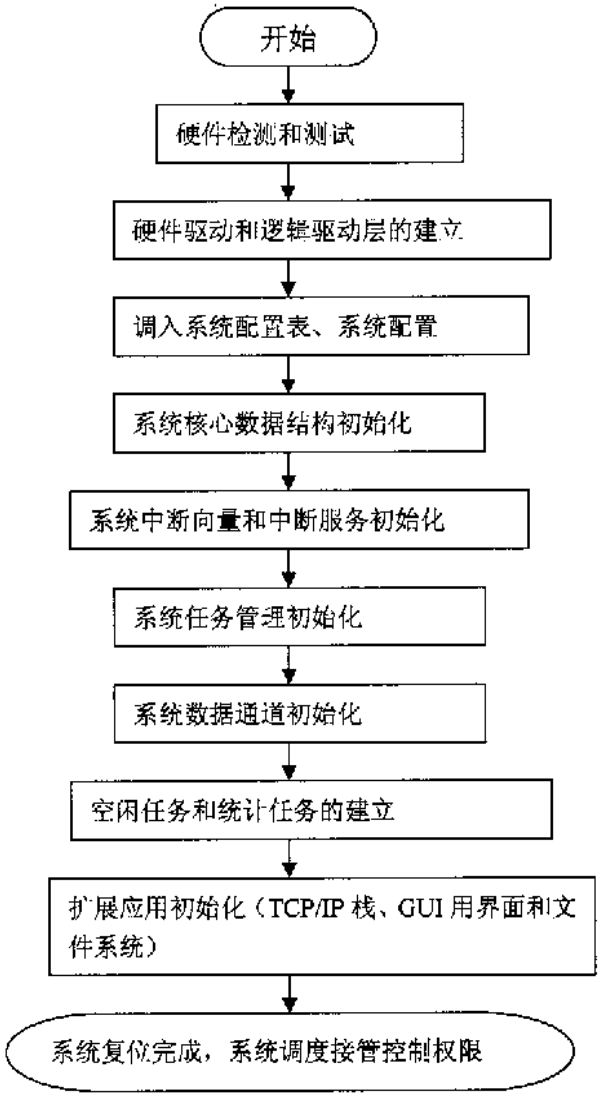

开始
硬件检测和测试
硬件驱动和逻辑驱动层的建立
调入系统配置表、系统配置
系统核心数据结构初始化
系统中断向量和中断服务初始化
系统任务管理初始化
系统数据通道初始化
空闲任务和统计任务的建立
扩展应用初始化（TCP/IP 栈、GUI 用界面和文件系统）
系统复位完成，系统调度接管控制权限

图 4.7 系统复位管理

第五章 图形用户界面

[13]在一般的嵌入式系统当中，由于硬件条件等的限制，我们看到的用户界面都非常简单，几乎看不到我们在 PC 机看到的华丽美观的 GUI 支持。最近出现的 Palm 等手持式电脑或者在 Windows CE 等，面向嵌入式系统的操作系统上，我们已经看到了完整的图形用户界面支持。随着手持式设备的硬件条件的提高，目前嵌入式系统对轻量级 GUI 的需求会越来越迫切。近来的市场需求显示，越来越多的嵌入式系统，包括 PDA、机顶盒、DVD/VCD 播放机、WAP 手机等等系统均要求提供全功能的 Web 浏览器。这包括 HTML 4.0 的支持、JavaScript 的支持，甚至包括 Java 虚拟机的支持。而这一切均要求有一个高性能、高可靠的 GUI 的支持。另外一个迫切需要轻量级 GUI 的系统是工业实时控制系统。这些系统一般建立在标准 PC 平台上，硬件条件相对嵌入式系统要好，但对实时性的要求非常高，并且比起嵌入式系统来说，对 GUI 的要求也更高。这些系统一般不希望建立在庞大累赘的、非常消耗系统资源的操作系统和 GUI 之上，比如 Windows 或 X Window。目前许多这类系统都建立在 DOS 等系统上，并且采用比较简单的手法实现 GUI。但是，在出现 Linux 系统之后，尤其在 RT-Linux 系统出现之后，许多工业控制系统开始采用 RT-Linux 作为操作系统，但 GUI 仍然是一个问题。关键是 X Window 太过庞大和臃肿。这样，这些系统对轻型 GUI 的需求更加突出。但是嵌入式系统往往是一种定制设备，它们对 GUI 的需求也各不相同。有的系统只要求一些图形功能，而有些系统要求完备的 GUI 支持。因此，GUI 也必须是可定制的。综上所述，GUI 在嵌入式系统或者实时系统中的地位将越来越重要，这些系统对 GUI 的基本要求包括：轻型、占用资源少；高性能；高可靠性；可配置。尽管实时嵌入式系统对 GUI 的需求越来越明显，但目前 GUI 的实现方法各有不同：

1) 某些大型厂商有能力自己开发满足自身需要的 GUI 系统。

2) 某些厂商没有将 GUI 作为一个软件层从应用程序中剥离，GUI 的支持逻辑由应用程序自己负责。

3) 采用某些比较成熟的 GUI 系统，比如 MiniGUI 或者其他 GUI 系统。

在上述手段中，第 2 种方法是一种临时解决方案。利用这种手段编写的程序，无法将显示逻辑和数据处理逻辑划分开来，从而导致程序结构不好，不便于调试，并导致大量的代码重复。GUI 是一种类似于操作系统的基础软件，这种软件系统应该遵循一定的标准。目前看来，在 Linux 之上进行（实时）嵌入式系统开发的厂商，一般选择如下几种 GUI 系统：紧缩的 X Window 系统、MiniGUI、MicroWindows、OpenGUI、QT/Embedded 等，下面对这些系统进行简单介绍，并对它们的优缺点进行比较。

1) 紧缩的 X Window 系统

众所周知，X Window 是 Linux 以及其他类 UNIX 系统的标准 GUI。X Window 系统采用标准的客户/服务器体系结构，具有可扩展性好、可移植性好等优点。但该系统的庞大、累赘和低效率也是大家所共知的。为了获得应用程序的可移植性，许多厂家都试图通过对 X Window 系统的紧缩开发，使之能够在嵌入式系统上运行。国外已经开发出了大小约为 800K 的 X 服务器。这对西方国家来说基本能够满足嵌入式系统的需求了。但该系统的源代码尚不开放，从而很难进行本地化开发。

2) MicroWindows

MicroWindows 是一个开放源码的项目，目前由美国一家公司在主持开发。该项目的开发非常活跃，国内也有人参与了其中的开发，并编写了 GB2312 等字符集的支持。该项目的主要特色在于提供了比较完善的图形功能，包括一些高级的功能，比如 Alpha 混合，三维支持，TrueType 字体

支持等。但作为一个窗口系统，该项目提供的窗口处理功能还需要进一步完善，比如控件或构件的实现还很不完备，键盘和鼠标等的驱动还很不完善。值得一提的是，该项目的许多控件是从 MiniGUI 中移植过去的。该项目已经启动了一个开放源码的浏览器项目，该浏览器在 KDE kfm 提供的 HTML 解释器的基础上开发，目前能够解释一些简单的 HTML 页面。

3) OpenGUI

OpenGUI 在 Linux 系统上存在已经很长时间了。最初的名字叫 FastGL，只支持 256 的线性显存模式。但目前也支持其他显示模式。这个库是用 C++ 编写的，只提供 C++ 接口。

4) Qt/ Embedded

QT/Embedded 是著名的 QT 库开放商正在进行的面向嵌入式系统的 QT 版本。这个版本的主要特点是可移植性较好，许多基于 QT 的 X Window 程序可以非常方便地移植到嵌入式版本。但是该系统不是开放源码的，如果你要使用这个库，可能需要支付昂贵的授权费用。目前，已经有了 QT/Embedded 的演示版，预计明年初将出现 QT/Embedded 的正式发行版。

5) MiniGUI 简介

MiniGUI 是一种面向嵌入式系统或者实时系统的图形用户界面支持系统。它主要运行于 Linux 控制台，实际可以运行在任何一种具有 POSIX 线程支持的 POSIX 兼容系统上。MiniGUI 同时也是国内最早出现的几个自由软件项目之一。MiniGUI 的主要特色有：遵循 LGPL 条款的纯自由软件；提供了完备的多窗口机制；这包括：多个单独线程中运行的多窗口，单个线程中主窗口的附属，对话框和预定义的控件类（按钮、单行和多行编辑框、列表框、进度条、工具栏等）；消息传递机制；多字符集和多字体支持，目前支持 ISO8859-1、GB2312、Big5 等字符集，并且支持各种光栅字体和 TrueType、Type 1 等矢量字体；全拼、五笔等汉字输入法支持；BMP、GIF、JPEG、PCX、TGA 等常见图像文件的支持；Windows 的资源文件支持，如位图、图标、光标等；插入符、定时器、加速键等；其他（Beep、ETC/INI 文件操作等）；很小巧，包含全部功能的库文件大小为 300 K 左右；可配置，可根据项目需求进行定制配置和编译；高稳定性和高性能。MiniGUI 已经在 Linux 发行版安装程序、CNC 系统、蓝点嵌入式系统等关键应用程序中得到了实际的应用；可移植性好。目前，MiniGUI 可以在 X Window 和 Linux 控制台上运行。

为了适合不同的操作系统环境，MiniGUI 可以配置成三种运行模式：

a) MiniGUI-Threads。运行在 MiniGUI-Threads 上的程序可以在不同的线程中建立多个窗口，但所有的窗口在一个进程或者地址空间中运行。这种运行模式非常适合于大多数传统意义上的嵌入式操作系统，比如 uC/OS-II、eCos、VxWorks、pSOS 等等。当然，在 Linux 和 uClinux 上，MiniGUI 也能以 MiniGUI-Threads 的模式运行。

b) MiniGUI-Lite。和 MiniGUI-Threads 相反，MiniGUI-Lite 上的每个程序是单独的进程，每个进程也可以建立多个窗口。MiniGUI-Lite 适合于具有完整 UNIX 特性的嵌入式操作系统，比如嵌入式 Linux。

c) MiniGUI-Standalone。这种运行模式下，MiniGUI 可以以独立进程的方式运行，既不需要多线程也不需要多进程的支持，这种运行模式适合功能单一的应用场合。比如在一些使用 uClinux 的嵌入式产品中，因为各种原因而缺少线程支持，这时，就可以使用 MiniGUI-Standalone 来开发应用软件。一般而言，MiniGUI-Standalone 模式的适应面最广，可以支持几乎所有的操作系统，甚至包括类似 DOS 这样的操作系统；MiniGUI-Threads 模式的适用面次之，可运行在支持多任务的实时嵌入式操作系统，或者具备完整 UNIX 特性的普通操作系统；MiniGUI-Lite 模式的适用面较小，它仅适合于具备完整 UNIX 特性的普通操作系统。但不论采用哪种运行模式，MiniGUI 为上层应用软件提供了最大程度上的一致性；只有少数几个涉及初始化的接口在不同运行模式上有所不同。

比较上述几个面向嵌入式系统的 GUI，我们认为目前比较成熟，同时得到最多开发人员认可的有紧缩的 X Window 系统、MiniGUI、MicroWindows 等系统。尽管紧缩的 X Window 系统其 X 服务器可以降低到 800K 的大小，但因为 X Window 系统的运行还需要其他程序和库的支持，包括 X

窗口管理器、XLib、建立在 XLib 之上的 GTK 和 QT 等函数库，因此，紧缩的 X Window 系统在运行期间所占用的系统资源很多，加上中文显示和中文输入等本地化代码之后，系统的整体尺寸和运行时的资源消耗将进一步变大。因此，嵌入式系统的开发商往往将紧缩的 X Window 系统定位在机顶盒等对资源要求并不苛刻的嵌入式系统上。MiniGUI 和 MicroWindows 均为自由软件，只是前者遵循 LGPL 条款，后者遵循 MPL 条款。这两个系统的技术路线也有所不同。MiniGUI 的策略是首先建立在比较成熟的图形引擎之上，比如 SVGALib 和 LibGGI，开发的重点在于窗口系统、图形接口之上；MicroWindows 目前的开发重点则在底层的图形引擎之上，窗口系统和图形接口方面的功能还比较欠缺。举个例子来说，MiniGUI 有一套用来支持多字符集和多编码的函数接口，可以支持各种常见的字符集，包括 GB、BIG5、UNICODE 等，而 MicroWindows 在多字符集的支持上尚没有统一接口。MiniGUI 是一个非常适合于工业控制实时系统以及嵌入式系统的可定制的、小巧的图形用户界面支持系统。

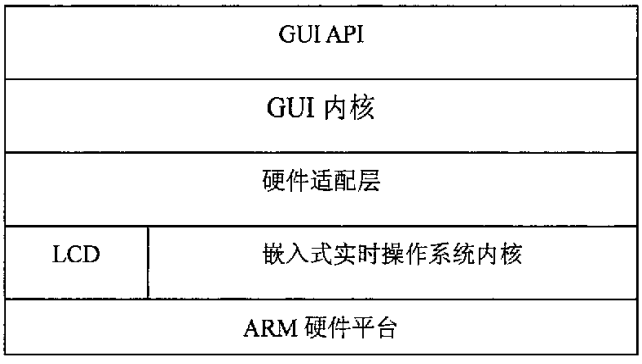


图 5.1GUI 框架结构

这里主要以 MiniGUI 为主要框架结构，在嵌入式实时内核的基础之上建立图形用户界面子系统。其架构如图 5.1 所示。通过硬件适配层将 GUI 上层和底层操作系统隔离开来。基于 GUI 的应用程序一般通过 GUI 自身提供的 API 来实现各种功能。硬件适配层将特定操作系统及底层硬件的细节隐藏起来，而上层应用程序则无需关系底层的硬件平台输出和输入设备。GUI API 则提供了文本字符显示、2D 图形显示和操作和窗口等基本 GUI 功能。

5.1 LCD 驱动

(LCD)英文全称为 Liquid Crystal Display，它一种是采用了液晶控制透光度技术来实现色彩的显示器。和 CRT 显示器相比，LCD 的优点是很明显的。由于通过控制是否透光来控制亮和暗，当色彩不变时，液晶也保持不变，这样就无须考虑刷新率的问题。对于画面稳定、无闪烁感的液晶显示器，刷新率不高但图像也很稳定。LCD 显示器还通过液晶控制透光度的技术原理让底板整体发光，所以它做到了真正的完全平面。体积小、能耗低也是 CRT 显示器无法比拟的。综合来看液晶显示，稳定可靠、成本低、功耗小、控制驱动方便、接口简单易用、模块化结构紧凑，因此在嵌入式系统中作为人机界面获得了广泛的应用。LCD 通常可按照其显示方式分为段式、点字符式、点阵式等。除了黑白显示外，还有多灰度和彩色显示等。一般嵌入式系统的 LCD 驱动原理图如 5.2 所示。

这里的 LCD 主要针对于点阵型 LCD，对于段式和字符型 LCD 不在驱动范围。灰度点阵型 LCD 使用的控制器的显存中每 n 位对应一个 LCD 显示点，整个 LCD 实现的灰度等级就是 2n。彩色 LCD 显示基于红 R、绿 G、蓝 B 三基色叠加原理，每个 LCD 像素点由三个 RGB 子像素点构成，分别由三个 RGB 色彩驱动。彩色 LCD 显示需要更大的显存，每个色彩有 2 n 种颜色，就需占用 n 位显存。

图中可以看出对于 LCD 的驱动，主要是对于 LCD 控制器和显示缓存的使用。具体到实际的控制电路，每个厂家的实现方法略有不同，总结起来主要区别在于地址总线使用得位数，这个决定了显示存储器的读写方式不同，和读写数据宽度的不同。另外 LCD 实现的灰度级别或者是颜色多少，决定了显示存储器和 LCD 的每个点的对应关系有所不同，和段驱动和行驱动的不同，图 5.3 说明了它们的关系。

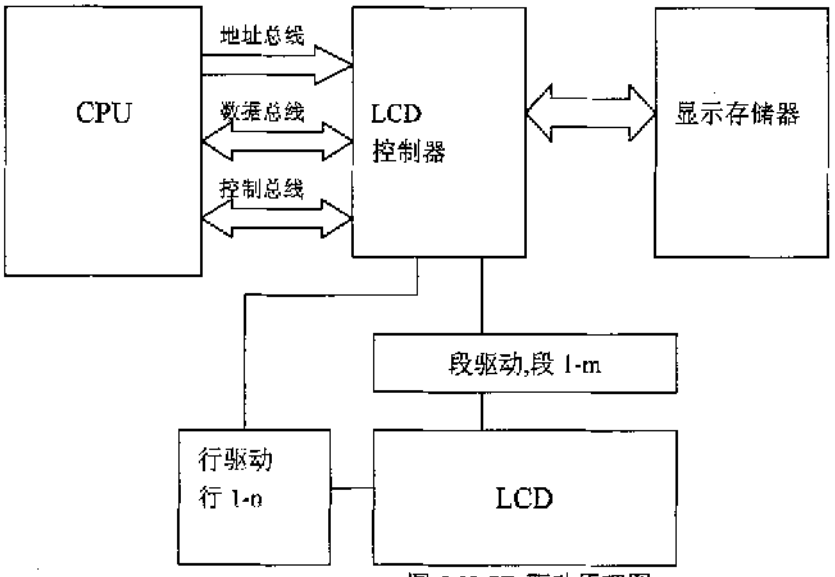


图 5.2LCD 驱动原理图

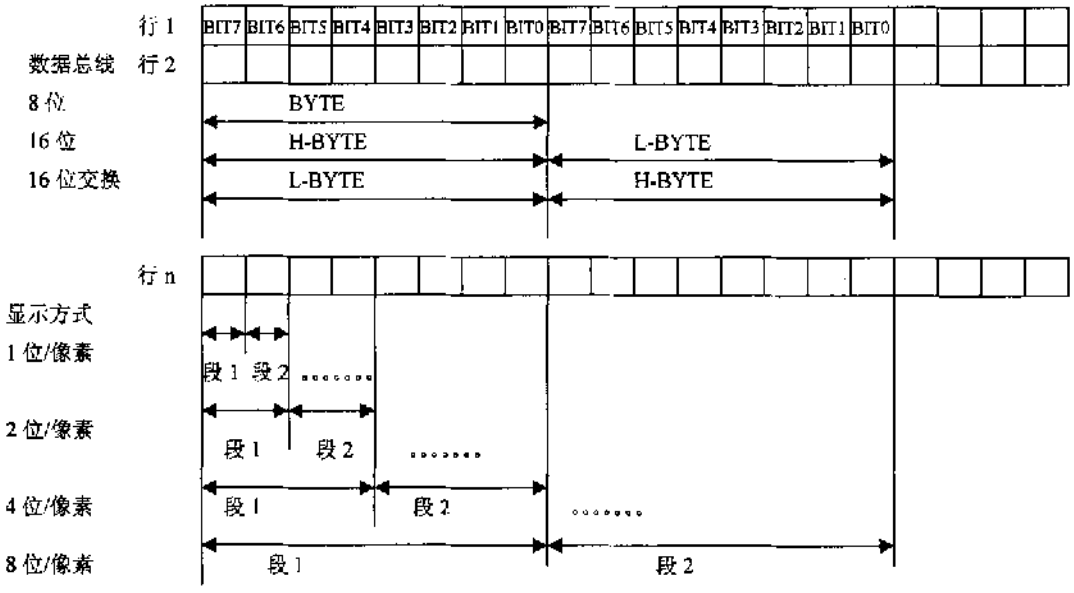


图5.3显示存储器和像素点的关系

目前主要有两种总线接口方式和LCD控制器连接。对于一些小的LCD(240X128 或者 320X240)

通常采用已有一位地址总线的简单的连接方式。这种方式一般使用CPU的一个I/O引脚来驱动，这种方式一般速度比较慢，访问显存时使用单一地址，连续写入。另一种方式采用全地址线连接，这种方式可以单一访问全部现存，根据数据总线的宽度，可以进行显存的字节读写，和字读写。以加快显示速度。

一般的嵌入式系统(ARM体系结构)是按照存储器外设统一编址的，也就是说显示存储器的地址和存储器(RAM,ROM)在同一地址空间，所以对于这一部分显存的读写访问就直接控制了LCD上面，对应像素点的灰度或者颜色的显示，也就可以在LCD显示任意的字符和图像了。下面给出了读写显存的基本程序语句。

```
#define LCD_WRITE_REG(OFF,data) *((void *)(0xc0000+(OFF<<1)))=data
#define LCD_WRITE_MEM(OFF,data) *((void *)(0x80000+(OFF<<1)))=data
```

5.2 颜色管理

嵌入式系统的 LCD 的颜色主要有单色（黑/白），单色灰度级，和彩色。对于颜色管理来说，设置不同的配置参数，驱动各种类型的 LCD，用于用户程序。GUI 定义逻辑颜色作为和应用程序的界面。这里的逻辑颜色即我们经常使用的 RGB,对于 RGB 色彩来说，就是用 8BIT 来分别表示红、绿和蓝，可以用 24BIT 值来表示，0XBBGGRR 。这样白色对应于 0XFFFFFF，黑色对应于 0X000000，亮红色为 0XFF。而物理颜色对应于真正的 LCD 显示颜色。物理颜色也是由 RGB 形成，但是一般位数没有达到 24BIT,颜色级别也没那么多（比如单色灰度显示和 8/16 位彩色 LCD）,这里采用最小平方差查询的方法，将逻辑颜色对应到最接近的 LCD 可以提供物理颜色。

GUI 提供了常见的若干种物理颜色的颜色模板，各它们和逻辑颜色的对应关系。

1) 模式 1，只有黑白两种颜色，每个像素占用 1BIT。
2) 模式 2，4 种颜色灰度级别，每个像素对应 2BIT。
3) 模式 4，每像素对应 4BIT，有 2 x 2 x 2 x 2 = 16 种颜色。
4) 模式 111，每个颜色有两个级别，2 x 2 x 2 = 8 种颜色。这用模式硬件只提供 1bit 每像素，和每个基色。
5) 模式 222，每个颜色有四个级别，4 x 4 x 4 = 64 种颜色。
6) 模式 233，2bits 蓝，3bits 绿，3bits 红。这种模式支持 256 色，这样对于绿色和红色有 8 个灰度级别，蓝色有 4 个灰度级别。颜色序列为 BBGGGRRR。
7) 模式 233，2bits 红，3bits 绿，3bits 蓝，红蓝两色交换。这种模式支持 256 种颜色，这样对于绿色和蓝色有 8 个灰度级别，红色有 4 个灰度级别。颜色序列为 RRGGGBBB。
8) 模式 323，3bits 蓝，2bits 绿，3bits 红，这种模式支持 256 种颜色，这样对于红色和蓝色有 8 个灰度级别，绿色有 4 个灰度级别。颜色序列为 RRRGGBBB。
9) 模式 332，3bits 蓝，3bits 绿，2bits 红，这种模式支持 256 种颜色，这样对于蓝色和绿色有 8 个灰度级别，红色有 4 个灰度级别。颜色序列为 BBBGGGRR。
10) 模式 444，4bits 红，4bits 绿，4bits 蓝，共有 16 x 16 x 16 = 4096 种颜色。
11) 模式 555，5bits 红，5bits 绿，5bits 蓝，共有 32 x 32 x 32 = 32768 种颜色，支持 RGB 15bpp 颜色深度。
12) 模式 565，5bits 红，6bits 绿，5bits 蓝，共有 32 x 64 x 32 = 65535 种颜色。
13) 模式 8666，这种模式,提供每个基色 6 个级别，再加上 16 个灰度级别。这种模式经常被可编程颜色查找表(LUT)所使用，支持 256 种颜色调色板。一用有 6 x 6 x 6 + 16 = 232 种颜色。

每个像素点所存储了颜色信息，既可以是 RGB 方式，每个像素点的颜色是 RGB 颜色的叠加，

也可以是 color-index 方式（每个像素点的颜色对应于一个眼色索引值），每个索引值对应于颜色查找表，或者是颜色模板里的一个 R，G，B 的颜色值。这两种方式主要取决于 LCD 控制器所选用的方式。如果选用 color-index 方式，需要初始化颜色调色板，存储于 ROM 里面。

这里主要提供了一下一些和颜色有关的 API 函数调用。

GUIGetBkColor() 返回当前背景色
GUIGetBkColorIndex()返回当前背景索引色
GUIGetColor()返回当前前景色
GUIGetColorIndex()返回当前前景索引色
GUISetBkColor()设置当前背景色
GUISetBkColorIndex()设置当前背景索引色
GUISetColor()设置当前前景色
GUISetColorIndex()设置当前前景索引色
GUIColorToIndex() 将颜色转换为颜色索引值
GUIIndexToColor() 将颜色索引值转换为颜色
GUIInitLUT() 初始化颜色模板

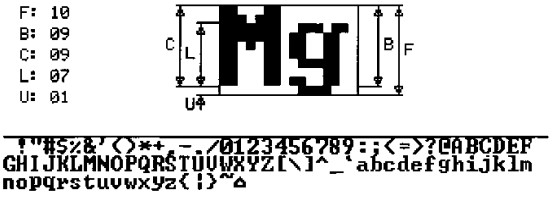
5.3 字库和文本显示

目前字符集标准组要有ASCII, ISO 8859-1 和 Unicode. 最常见的是8BIT的ASCII字符集，这里GUI提供了标准ASCII字符库和汉字库。主要API有：

GUIGetFont() 返回当前所选字库
GUISetFont() 设置当前字库
GUIGetCharDistX()获取当前字库的显示字符宽度
GUIGetFontDistY() 获取当前字库字符的高度
GUIGetFontInfo() 获取当前字库信息
GUIGetFontSizeY() 返回当前字库显示字符的像素高度
GUIGetStringDistX() 返回当前字库显示字符串所需像素宽度
GUIGetYDistOfFont() 返回当前字库的所需高度
GUIIsInFont() 判断所显示字符是否在字库内存在

这里字库的实现方法主要是在C文件内，定义静态点阵字符信息，存储在ROM中，例如宽8像素，高10像素的ASCII字库的字符 'M' 定义如下：

```
const unsigned char Font8x10ASCII_004D[10] = { /* code 004D */
  XX___XX_,
  XXX_XXX_,
  XXXXXXX_,
  XXXXXXX_,
  XX_X_XX_,
  XX___XX_,
  XX___XX_,
  XX___XX_,
  XX___XX_,
  ________};
```



对于UNICODE汉字库也以同样的方法来实现。例如字符‘列’的定义如下：

```
const unsigned char acFontHZ12_c1d0[24] = { // code  0xc1d0
        ________,__X_____,
        XXXXXXX_,__X_____,
        ___X____,X_X_____,
        ___XXX__,X_X_____,
        __X__X__,X_X_____,
        _X_X_X__,X_X_____,
        X__XX___,X_X_____,
        ____X___,X_X_____,
        ___X____,X_X_____,
        __X_____,__X_____,
        _X______,X_X_____,
        ________,_X______
};
```

在此基础之上，提供了关于文本字符显示的 API 函数调用。

GUI_DispChar() 在当前位置显示当前设定字库和颜色的字符

GUI_DispCharAt()在指定位置显示当前设定字库和颜色的字符

GUI_DispString()在当前位置显示当前设定字库和颜色的字符串

GUI_DispStringAt()在指定位置显示当前设定字库和颜色的字符串

GUI_DispStringAtCEOL()在指定位置显示当前设定字库和颜色的字符串并清空到行末尾

GUI_DispStringInRect()在指定矩形框内部显示当前设定字库和颜色的字符串

GUI_GetTextAlign() 返回当前字符对齐方式

GUI_SetTextAlign()设置当前字符对齐方式

GUI_GotoX() 设置当前光标X坐标

GUI_GotoXY()设置当前光标X，Y坐标

GUI_GotoY()设置当前光标Y坐标

GUI_GetDispPosX()返回当前光标X坐标

GUI_GetDispPosY()返回当前光标Y坐标

GUI_Clear()清空屏幕

GUI_DispCEOL() 清空到行末

5.4 2D 图形，窗口管理

对于 2D 图形，是在基本画点函数的基础之上，实现了一些基本的 2D 图形的 API 函数。

GUI_SetDrawMode 设置画图模式，主要指正常模式和异或模式

GUI_FillRect 用当前背景色填充矩形

GUI_DrawPixel 画像素点
GUI_DrawPoint 画点
GUI_FillRect 画矩形
GUI_DrawLine 画线
GUI_DrawPolygon 画多边形.
GUI_DrawCircle 画圆
GUI_FillCircle 填充元
GUI_DrawEllipse 画椭圆
GUI_FillEllipse 填充椭圆
GUI_DrawArc 画弧

GUI 的窗口管理提供了窗口和窗口控件的基本功能，当使用 GUI 窗口管理时，任何能显示在显示终端上的内容都包含在一个窗口里面，这个窗口是 LCD 屏幕上的一个给用户画图或者显示目标的区域。窗口可以是任何尺寸的，可以一次在屏幕上显示多个窗口，也可以在其它窗口当中或之前显示窗口。GUI 对于窗口管理的一些定义：

- 活动窗口：当前正在用来画图或是显示操作的窗口。
- 回调函数：回调函数是由用户定义的，当特定事件发生时，指示图形系统调用特定函数的函数。通常当窗口的内容发生改变时，它们用来自动重绘窗口。例如，窗口内显示一幅位图，当窗口移动时，位图并不会自动移动，此时，就需要调用回调函数对窗口进行重绘，窗口内容也和窗口一起移动。
- 子/父窗口：子窗口是相对于父窗口定义的窗口。无论何时，只要父窗口移动，则子窗口们都要相应移动。子窗口被父窗口完全包含。拥有同一个父窗口的子窗口们互相间称为兄弟窗口。
- 客户区：窗口的客户区就是窗口的可用区。如果窗口包含方框或标题栏，则客户区就是方框或标题栏的内部区域。
- 句柄：当一个新的窗口被创建时，WM 分配一个唯一的标识符，称为句柄。句柄用在后续对该窗口操作的所有函数中，而且可以利用该句柄唯一标识该窗口。
- 有效窗口/无效窗口：有效窗口是一个已经完全更新，不需要重绘的窗口。不管是完全重绘还是局部重绘。当窗口内容发生改变时，WM 则标识窗口无效。下一次重绘之后（调用回调函数），窗口又变为有效。

在 GUI 中，回到机制用来在窗口管理中出控制窗口的重绘操作，这使得窗口管理的有效性成为可能。除了窗体管理以外，GUI 提供了基于窗口的按钮、滚动条、单选/多选按钮和文本框等一些基本窗口控件，以用于图形界面的应用。

第六章 TCP/IP 协议栈

这里将 TCP/IP 协议栈作为系统内核的一部分。在其上建立一个小型的 HTTP WEB 服务和 TELNET 服务，WEB 服务主要用于和系统和外界的数据交换，TELNET 服务和文件系统相互联系，通过 TELNET 登陆到系统，对于系统的文件进行操作。这一章主要讲述了 TCP/IP 中 IP 和 TCP 的在本系统的实现方法。这里采用一种功能比较齐全相对简单的免费 TCP/IP 协议栈 uIP 0.9 作为设计核心。uIP 是由瑞士计算机科学院的 Adam Dunkels 等开发的一种免费公开源代码的小型 TCP / IP 协议栈，它专门为 8 位和 16 位 MCU 编写。uIP 完全是用 C 语言编写，它在保证一个完整的 TCP / IP 栈的前提下，只保留了最必要的一系列特征，使其代码最少、占用的 RAM 最小；它只能处理单一的网络接口。通常的 TCP / IP 栈采用 BSD socket API，需要来自下层的多任务操作系统的支持，而且任务管理、语境切换以及堆栈空间的分配都要占用很大的开销。uIP 采用了一个事件驱动接口，通过调用应用程序响应事件。而相应的应用程序作为 C 函数调用。通常情况下，uIP 的源代码虽然只有几 KB，RAM 占用仅几百字节，但 uIP 提供了网络通信所必须的协议，包括：ARP、SLIP、IP、UDP、ICMP(PING)和 TCP，能够满足系统接入 TCP / IP 网络(如 Internet)的需要。目前 uIP 最新的版本为 V0.9，符合 Internet 标准。[12][14]

6.1 物理层和数据链路层的实现

uIP 直接建立在数据链路层之上，提供了数据连接接口，这样物理层和数据链路层的结构实现模式和 uIP 是没有关系的， 对于实际的系统硬件，物理层和数据链路层可以采用以太网，也可以采用点对点的 modem 连接方式，或者是运用无线连接方式。其典型的以太网连接方式如图 6.1 所示。

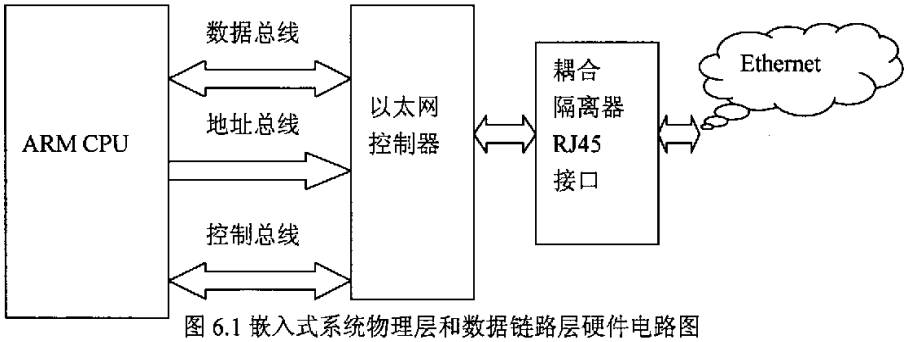


图 6.1 嵌入式系统物理层和数据链路层硬件电路图

在此系统的开发实现过程中，为了简化设计，便于调试和测试，在 armulator 中，利用 pc 机的网卡在 windows 系统中虚拟了以太网控制器。这样只要编写统一的设备驱动接口，在虚拟硬件平台上实现系统的 TCP/IP 协议栈的调试。具体内容在第 8 章测试环境中加以叙述。

6.2 TCP/IP 的实现

这里采用的协议栈系瑞典计算机科学研究所 Adam Dunkels 开发的 uIP0.9 。其有如下的功能特性：

1. 完整的说明文档和公开的源代码(全部用 C 语言编写，并附有详细注释)；
2. 极少的代码占用量和 RAM 资源要求，尤其适用于 8/16 位单片机；
3. 高度可配置性，以适应不同资源条件和应用场合；
4. 支持 ARP、IP、ICMP、TCP、UDP(可选)等必要的功能特性；
5. 支持多个主动连接和被动连接并发，支持连接的动态分配和释放；
6. 简易的应用层接口和设备驱动层接口；
7. 完善的示例程序和应用协议实现范例。

正是由于 uIP 所具有的显著特点，自从 0.6 版本以来就被移植到多种处理器上，包括 MSP430、AVR 和 Z80 等。TCP/IP 协议栈的流程模式一般有几种方式.

1. TCP/IP 协议的每一层是一个单独进程。IP 协议是一个进程处理,TCP 也是一个进程.这样的好处是网络协议的每一层都非常清晰,代码的调试和理解都非常容易. 但是这样得多线程环境代价重大，因为，不但在线程管理里涉及增加代码的复杂性，而且保存每线程堆栈需要额外的储存器，还有执行任务切换的时间开销也摊派在这里。对于嵌入式系统不会有足够的资源去实现一个多线程环境，另外对于一般的嵌入式系统不会应用过于复杂的 TCP/IP 应用。
2. 另外一种方式是 TCP/IP 协议栈在操作系统内核当中.应用程序通过操作系统的系统调用和协议栈来进行通信，这样 TCP/IP 的协议栈就限定于特定的操作系统内核了.如 windows 就是这种方式。
3. 这里 uIP 所采用得方式是，TCP/IP 协议栈都在一个进程当中,包括了应用服务，这样 TCP/IP 协议栈作为操作系统内核的一个分离得应用进程，而应用层程序驻留在 TCP/IP 进程中，也可以把应用层进程作为单独的进程，通过操作系统的邮箱,消息队列等和 TCP/IP 进程进行通信。如果应用层程序驻留 TCP/IP 进程中,那应用层程序就利用内部回调函数和 TCP/IP 协议栈通信。

6.2.1 uIP 的体系结构

uIP0.9 是一个适用于 8/16 位机上的小型嵌入式 TCP/IP 协议栈，简单易用，资源占用少是它的设计特点。它去掉了许多全功能协议栈中不常用的功能，而保留网络通信所必要的协议机制。 其设计重点放在 IP、ICMP 和 TCP 协议的实现上，将这三个模块合为一个有机的整体，而将 UDP 和 ARP 协议实现作为可选模块。uIP0.9 的体系结构如图 6.2 所示。

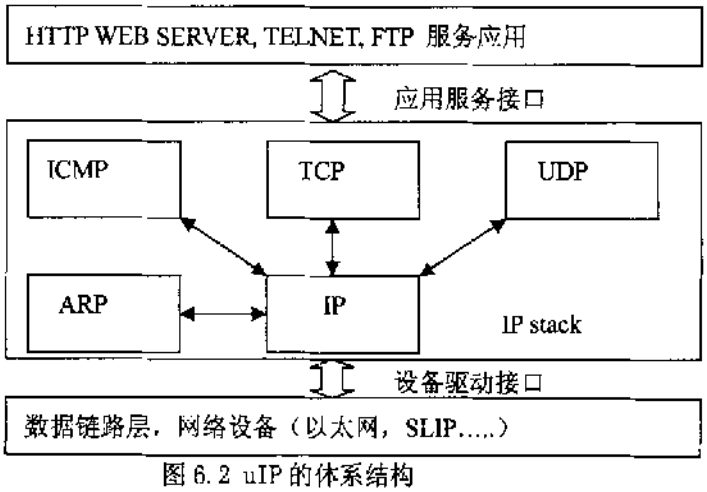


图 6.2 uIP 的体系结构

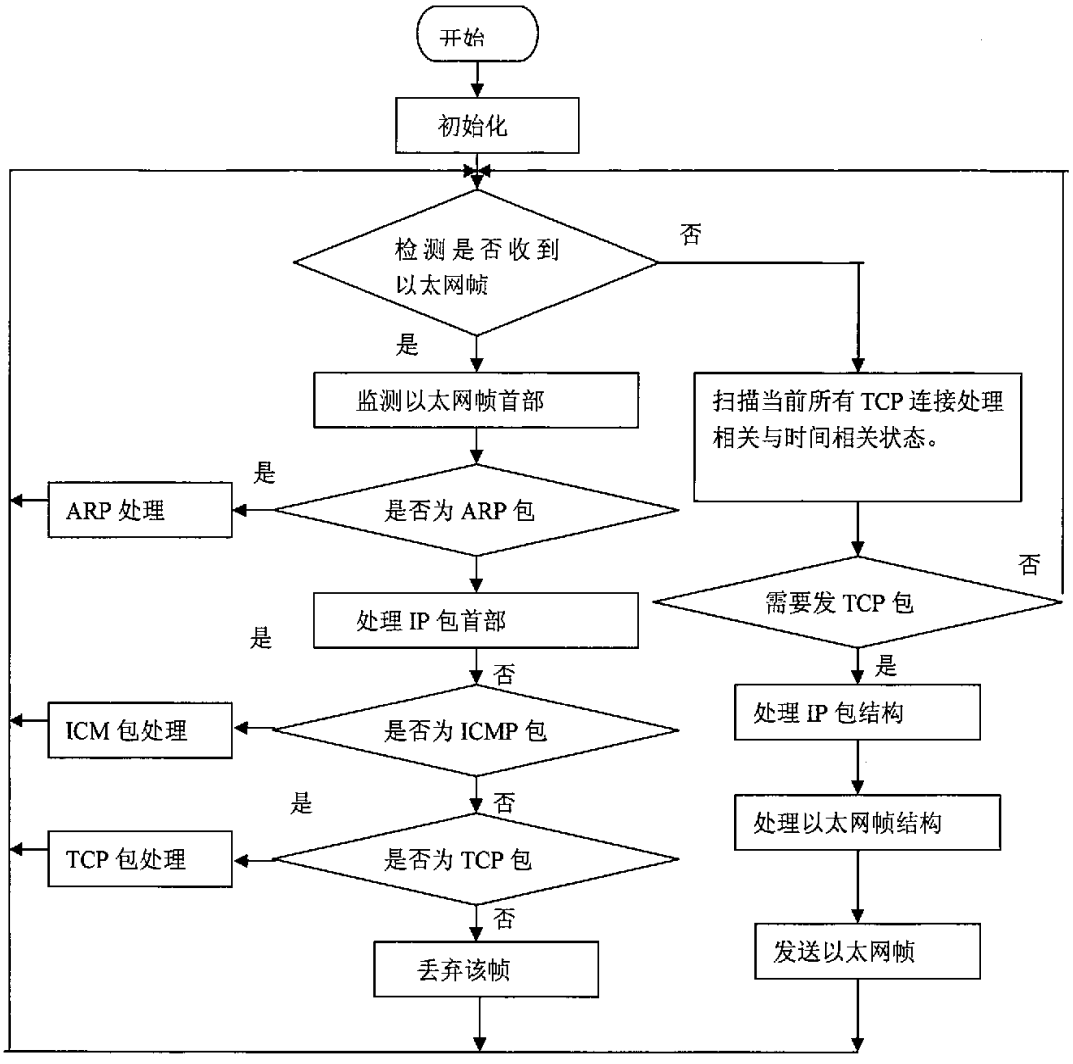


图 6.3TCP/IP 协议栈处理流程

uIP0.9 处于网络通信的中间层，其上层协议在这里被称之为应用程序，而下层硬件或固件被称之为网络设备驱动。显然，uIP0.9 并不是仅仅针对以太网设计的，它具有媒体无关性。为了节省资源占用， 简化应用接口， uIP0.9 在内部实现上作了特殊的处理。

a) 注意各模块的融合，减少处理函数的个数和调用次数，提高代码复用率，以减少 ROM 占用。

b) 基于单一全局数组的收发数据缓冲区，不支持内存动态分配， 由应用负责处理收发的数据。

c) 基于事件驱动的应用程序接口，各并发连接采用轮循处理，仅当网络事件发生时，由 uIP 内核唤起应用程序处理。这样，uIP 用户只须关注特定应用就可以了。传统的 TCP/IP 实现一般要基于多任务处理环境，而大多数 8 位机系统不具备这个条件。

应用程序主动参与部分协议栈功能的实现(如 TCP 的重发机制，数据包分段和流量控制)，由 uIP 内核设置重发事件，应用程序重新生成数据提交发送，免去了大量内部缓存的占用。基于事件驱动的应用接口使得这些实现较为简单。整个 TCP/IP 线程的处理流程如图 6.3 所示。

6.2.2 uIP 链路设备驱动程序接口

uIP 内核中有两个函数直接需要底层设备驱动程序的支持。一是 uip_input()。当设备驱动程序从网络层收到一个数据包时要调用这个函数，设备驱动程序必须事先将数据包存放到 uip_buf[]中，包长放到 uip_len，然后交由 uip_input()处理。当函数返回时，如果 uip_len 不为 0，则表明有带外数据(如 SYN，ACK 等)要发送。当需要 ARP 支持时，还需要考虑更新 ARP 表或发出 ARP 请求和回应，示例如下。

```
#define BUF ((struct uip_eth_hdr *)&uip_buf[0])
uip_len = ethernet_devicedriver_poll(); //接收以太网数据包
  //(设备驱动程序)
  if(uip_len>0){ //收到数据
    if(BUF->type = = HTONS(UIP_ETHTYPE_IP)) { //是 IP 包吗?
      uip_arp_ipin(); //去除以太网头结
      //构，更新 ARP 表
      uip_input(); //IP 包处理
      if(uip_len>0){ //有带回应数据
        uip_arp_out(); //加以太网头结构，在主动连接时可能要
        //构造 ARP 请求
        ethernet_devicedriver_send(); //发送数据到以太网
        //(设备驱动程序)
      }
  }else if (BUF->type = = HTONS(UIP_ETHTYPE_ARP)) {
    //是 ARP 请求包
    uip_arp_arpin(); //如是 ARP 回应，更新 ARP 表；如果是
    //请求，构造回应数据包
    if(uip_len>0) { //是 ARP 请求，要发送回应
      ethernet_devicedriver_send(); //发 ARP 回应到以太网上
    }
  }
```

另一个需要驱动程序支持的函数是 uip_periodic(conn)。这个函数用于 uIP 内核对各连接的定时轮循，因此需要一个硬件支持的定时程序周期性地用它轮循各连接，一般用于检查主机是否有数据要发送，如有，则构造 IP 包。使用示例如下。

```
for(i=0 ; i<UIP_CONNS; i++){
    uip_periodic(i);
    if(uip_len > 0){
      uip_arp_out();
      ethernet_devicedriver_send();
    }
}
```

从本质上来说， uip_input()和 uip_periodic()在内部是一个函数，即 uip_process (u8t flag)， UIP 的设计者将 uip_process(UIP_DATA)定义成 uip_input()，而将 uip_process(UIP_TIMER)定义成 uip_periodic()，因此从代码实现上来说是完全复用的。

6.2.3 uIP 的应用服务接口

为了将用户的应用程序挂接到uIP中，必须将宏UIP_APPCALL()定义成实际的应用程序函数名，这样每当某个uIP事件发生时，内核就会调用该应用程序进行处理。如果要加入应用程序状态的话，必须将宏 UIP_APPSTATE_SIZE 定义成应用程序状态结构体的长度。在应用程序函数中，依靠 uIP 事件检测函数来决定处理的方法，另外可以通过判断当前连接的端口号来区分处理不同的连接。下面的示例程序是实现的一个 Web 服务器应用的框架。

```
#define UIP_APPCALL uip51_appcall
#define UIP_APPSTATE_SIZE sizeof(struct uip51app_state)
struct uip51app_state{
    unsigned char *dataptr;
    unsigned int dataleft;
};
void uip51_initapp{ //设置主机地址
    u16_t ipaddr[2];
    uip_ipaddr(ipaddr， 202 ，120，127，192 );
    uip_sethostaddr(ipaddr);
    uip_listen(HTTP_PORT); //HTTP WEB PORT(80);
}
void uip51_appcall(void){
    struct uip51app_state *s;
    s = (struct uip51app_state *)uip_conn->appstate;
    //获取当前连接状态指针
    if(uip_connected()) {
        … //有一个客户机连上
    }
    if(uip_newdata()||uip_rexmit()) { //收到新数据或需要重发
      if(uip_datalen()>0){
        if(uip_conn->lport = = 80) { //收到 GET HTTP 请求
            update_table_data(); //根据电平状态数据表动态
            //生成网页
            s->dataptr=newpage;
            s->dataleft=2653;
            uip_send(s->dataptr，s->dataleft);
            //发送长度为 2653 B 的网页
        }
    }
  }
  if(uip_acked()) { //收到客户机的 ACK
      if(s->dataleft>uip_mss()&&uip_conn->lport = = 80){
      //发送长度>最大段长时
      s->dataptr+=uip_conn->len; //继续发送剩下的数据
      s->dataleft-=uip_conn->len;
      uip_send(s->dataptr，s->dataleft);
```

```
    }
    return;
  }
if(uip_poll())
{ … //将串口缓存的数据复制到
    //电平状态数据表
    return;
}
if(uip_timedout()|| //重发确认超时
    uip_closed()|| //客户机关闭了连接
    uip_aborted()){ //客户机中断连接
    return; }
}
```

6.3HTTP WEB 服务的实现

HTTP WEB 服务是在应用服务接口之上实现的。首先实现 HTTP 协议。而 HTTP 协议是基于 TCP 协议实现传输的。这里简化了 HTTP WEB 应用服务，支持静态页面和简单的 CGI 操作，CGI 主要实现动态页面的生成，在具体应用中可以通过 CGI 动态页面输出处理的数据和系统当前的运行状态。HTTP WEB 服务程序的框架如图 6.4 所示。下面分别介绍和 TCP/IP 应用服务有关的函数操作。

1．接收数据

如果 uIP 测试函数 uip_newdata()值为 1,远程连接的主机有发送新数据，uip_appdata 指针指向实际数据。数据的大小通过 uIP 函数 uip_datalen()获得。在数据不是被缓冲后，应用程序必须立刻启动。

2．发送数据

应用程序通过使用 uIP 函数 uip_send()发送数据。uip_send()函数采用两个参数：一个指针指向发送数据和数据的长度。如果应用程序为了产生要发送的实际数据需要 RAM 空间，包缓存(通过 uip_appdata 指针指向)可以用于这方面。在一个时间里应用程序只能在连接中发送一块数据。因此不可以在每个应用程序启用中调用 uip_send()超过一次；只有上一次调用的数据将会发出后才可以。注意，调用 uip_send()以后会改变某些全局变量，在应用函数返回前它不能被调用。

3．重发数据

如果数据在网络中丢失，应用程序必须重发数据。无论数据收到或没有收到，uIP 保持跟踪，和通知应用程序什么时候察觉出数据是丢失了。如果测试函数 uip_rexmit()为真，应用程序要重发上一次发出的数据。重发就好像原来那样发送，也就是通过 uip_send()。

4．关闭连接

应用程序通过调用 uip_close()关闭当前连接。这会导致连接干净地关闭。为了指出致命的错误，应用程序可以中止连接和调用 uip_abort()函数完成这个工作。如果连接已经被远端关闭，测试函数 uip_closed()为真。应用程序接着可以做一些必要的清理工作。

5．报告错误

有两个致命的错误可以发生在连接中，不是连接由远程主机中止，就是连接多次重发上一数据和被中止。uIP 通过调用函数报告这些问题。应用程序使用两个测试函数 uip_aborted()和 uip _timedout()去测试那些错误情况。

6．轮询

当连接空闲时，uIP 在每一个时候周期性地轮询应用程序。应用程序使用测试函数 uip_poll()去检查它是否被轮询过。

7．监听端口

uIP 维持一个监听 TCP 端口列表。通过 uip_listen()函数，一个新的监听端口打开。当一个连接请求在一个监听端口到达，uIP 产生一个新的连接和调用应用程序函数。如果一个新连接产生，应用程序被调用，测试函数 uip_connected()为真。

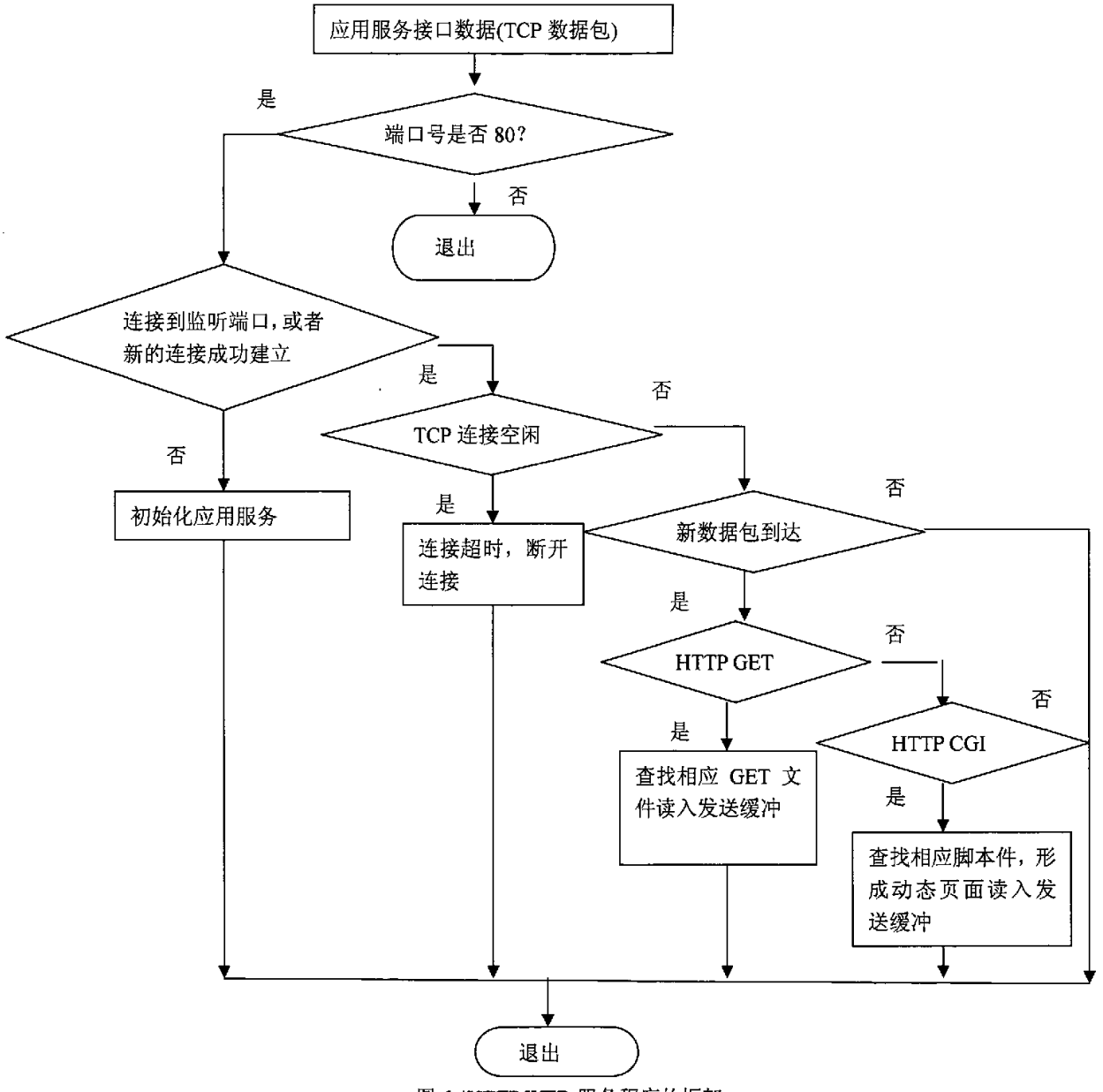


图 6.4HTTP WEB 服务程序的框架

8．打开连接

作为 uIP 的 0.6 版，在 uIP 里面通过使用 uip_connect()函数打开一个新连接。这个函数打开一个新连接到指定的 IP 地址和端口，返回一个新连接的指针到 uip_conn 结构。如果没有空余的连接槽，函数返回空值。为了方便，函数 uip_ipaddr()可以用于将 IP 地址打包进两个单元 16 位数组里，通过

uIP 去代表 IP 地址。程序 1 展示了怎样打开一个连接去远端 TCP 端口 8080。如果没有足够的 TCP 连接插槽去允许一个新连接打开，uip_connect()函数返回 NULL 和通过 uip_abort()中止当前连接。程序 2 展示怎样打开一个新连接去指定的 IP 地址。这里没有错误检查。

```
void connect_example1_app(void) {
  if(uip_connect(uip_conn->ripaddr, 8080) == NULL) {
    uip_abort();
  }
}
```

程序 1：打开一个连接去当前连接的远端的端口 8080

```
void connect_example2(void) {
  u16_t  ipaddr[2];
  uip_ipaddr(ipaddr, 192,168,0,1);
  uip_connect(ipaddr, 8080);
}
```

程序 2：打开一个到主机 192.168.0.1 上端口 8080 的连接

9．数据流控制

通过函数 uip_stop()和 uip_restart()，uIP 提供存取 TCP 数据流的控制途径。设想一个应用程序下载数据到一个慢速设备，例如磁盘驱动器。如果磁盘驱动器的作业队列满了，应用程序不会准备从服务器接收更多的数据，直到队列排出空位。函数 uip_stop()可以用于维护流控制和停止远程主机发送数据。当应用程序准备好接收更多数据，函数 uip_restart()用于告知远程终端再次发送数据。函数 uip_stopped()可以用于检查当前连接是否停止。

10． uIP/系统接口

从系统的立场看， uIP 由 3 个 C 函数 uip_init(),uip_input(), 和 uip_periodic()。uip_init()函数用于初始化 uIP 堆栈和在系统启动期间调用。当网络设备驱动器读一个 IP 包到包缓存时，调用函数 uip_input()。周期性运行是调用 uip_periodic()，代表的是一秒一次。调用 uIP 函数是系统的职责。

11． uIP/设备驱动接口

当设备驱动放一个输入包在包缓存里(uip_buf)，系统应该调用 uip_input()函数。函数将会处理这个包和需要时调用应用程序。当 uip_input()返回，一个输出包放在包缓存里。包的大小由全局变量 uip_len 约束。如果 uip_len 是 0，没有包要发送。

12． uIP/周期计时接口

周期计时是用于驱动所有 uIP 内部时钟事件，例如包重发。当周期计时激发，每一个 TCP 连接应该调用 uIP 函数 uip_periodic()。连接编号传递是作为自变量给 uip_periodic()函数的。类似于 uip_input()函数，当 uip_periodic()函数返回，输出的 IP 包要放在包缓存里。下面的程序展示了调用 uip_periodic()函数和监视输出包的一小段代码。在这个特别的例子，函数 netdev_send()是网络驱动的部分，将 uip_buf 数组的目录发出到网上。

```
for(i = 0; i < UIP_CONNS; ++i) {
    uip_periodic(i);
      if(uip_len > 0)
        netdev_send();
}
```

6.4TELNET 服务实现

TELNET 协议提供了双向的、面向字符(以 8bit 为数据单位) 的通信方式。最初它被用作终端与面向终端的进程之间通信的标准方法,到后来它也用于终端间的点对点通信以及在分布式环境下进程间的通信。RFC854 对 TELNET 协议进行了总体描述,RFC855 规定了制定协商选项所应遵循的标准,各种各样的选项的定义则在此后的 RFC 中被单独分别说明。TELNET 协议在 TCP/ IP 协议栈中位于应用层(application layer) , 直接工作在 TCP 层之上;TELNET 服务器程序工作在 TCP 的 23 号端口上。TELNET 协议有三个基本概念:网络虚拟终端、选项协商和对称性。

1. 网络虚拟终端

网络虚拟终端(NVT), 包括键盘(keyboard) 和打印机(printer) ,分别对应于普通终端的键盘和显示器。TELNET 程序的工作是:在发送数据前先把要发送的字符转换为 NVT 的字符,送给 NVT 键盘;把从网络上来的数据(已是 NVT 字符) 送给 NVT 打印机,经过 NVT 打印机过滤的字符交给上层软件处理。这样就能实现两端的正确通信。NVT 打印机没有规定回车的宽度(carriagewidth) 和页长(page length) 。它提供了 USASCII 的 95 个图形码(graphics , 从 32 到 126) 。对于 US2ASCII 的 33 个控制码(cont rol code ,从 0 到 31 和 127) 和 128 个 USASCII 所没有覆盖的码(从 128 到 255) , 它只使用了以下部分(剩余部分对打印机的行为不产生任何作用) :

NULL (NUL) 0 空操作
Line Feed(L F) 10 换行
Carriage Return(CR) 13 回车
BELL (BEL) 7 响铃
Back Space (BS) 8 退格
Horizontal Tab(HT) 9 光标移至下一水平制表符处
Vertical Tab(VT) 11 光标移至下一垂直制表符处
Form Feed(FF) 12 换页

NVT 键盘可以通过键、键组合和键序列产生所有 128 个 USASCII 码。除此之外,键盘还应该能够产生这些键值:

Synch 同步
Break(BRK) (为某些系统提供)BREAK 键
Interrupt Process(IP) 挂起或中断与 NVT 相连的进程
Abort Output (AO) 清除输出
Are You There(AYT) “你还在吗”
Erase Line(EL) 删除一行
Erase Character (EC) 删除字符

2. 选择协商

当用户需要使用 NVT 所不能提供的额外服务时,必须有个机制使通信双方对通信中的“语法”进行预先商定,这一过程就是选项的协商。目前 RFC 已经定义了很多的选项,最常用的如 RFC856 定义了二进制传输(binary t ransmission) 的选项、RFC857 定义了回显(echo) 的选项、RFC858 定义了抑制 GA 信号(suppress go ahead signal) 的选项等等。协商的方法是一端发出使某一选项生效的请求(request) 命令给另一端,而另一端则可以通过发送响应(acknowledge) 命令来接受(accept) 或拒绝(reject) 这一请求。该选项若被接受,在连接的两端立即同时生效; 如若被拒绝,两端仍都保持原来的约定。TELNET 的命令是至少包括两个字符的字节序列: IAC (interpreter as command) 转义符和命令码。选项协商的命令则有三个字节,第三个字节指示协商的选项。以下是 TELENT 的所有命令码:

SE 240 子协商结束

NOP 241 无操作
Data Mark 242 Data Mark
Break 243 NVT 字符 BRK.
Interrupt Process 244 IP 函数
Abort Output 245 AO 函数
Are You There 246 AYT 函数
Erase Character 247 EC 函数
Erase Line 248 EL 函数
Go ahead 249 GA 信号
SB 250 子协商开始
WILL (option code) 251 (协商码) 本方将响应
WONT(option code) 252 (协商码) 本方将不响应
DO(option code) 253 (协商码) 对方应答
DONT(option code) 254 (协商码) 对方不应答
IAC 255 数据 255

3．对称性

由于进行选项协商的两端是完全对称的,任意一端都有可能将对方的确认命令当成是请求命令而再次发出响应命令,这样就会造成协商过程的无限循环(loop) 。因此为了避免循环的发生,协商过程应遵循下面三个规则:

(1) 只为选项的变化发出请求;
(2) 接收到的请求如果要求自己进入已经具有的状态,那么此请求将不被响应;
(3) 当一端向另一端发送一个协商命令时(不管这个命令是请求还是响应命令) ,如果该选项的使用(接受) 将影响到对方对接收数据的处理,那么这个命令应该被插入到该选项开始起作用的数据流的首部。建立这一规则是因为从请求命令的发出到接收到响应命令将会有一定的延时。

TELNET 服务器运行起来后,监听 TCP 的 23 号端口,等待客户机的连接请求;连接建立起来之后,服务器和客户机就能够平等的交流了。虽然实现 TELNET 的应用程序有服务器和客户机之分,但如前所述,运用 TELNET 协议进行通信的双方是完全对称的,就此而言是不分服务器和客户机的。双方是完全相同的。

4．TELNET 的实现

TELNET 软件可由三大部分组成:NVT 键盘、NVT 打印机和命令解释执行体。NVT 键盘处理用户的输出流,输出 NVT 键盘所允许的字符和命令。如果输出的是命令字符,调用命令执行体完成相应功能; 如果是普通字符,则可以将之发送到网络上。另外,NVT 键盘还维护一个字符缓冲区,缺省情况下当该缓冲区存满一行数据时,才将数据真正发送到网络上;当然也可以让NVT 键盘每收到一个字符时就立即发送到网络。

NVT 打印机的工作很简单:网络上输入的数据如果在所支持的 95 个字符内,直接交给上层软件;如果输入的数据为前面所述的八个特殊字符(NUL 、L F、CR、BEL 、BS、HT、VT 和 FF) ,则把它们转换为本机所使用的相应字符再上传;对于其他字符,一律作抛弃处理。

命令解释执行体将混杂在输入流中的 TELNET 命令分离出来,解释之后执行相应的操作。它通过直接写 SOCKET 发送 TELNET 命令。此外,NVT 键盘和打印机的工作方式(如回显、二进制方式收发等) 都要受到命令解释执行体的选项的协商结果的影响,因此命令解释执行体还要通过选项来影响它们的工作方式。命令解释执行体要接收识别 TELNET 命令执行,具有较多的状态,是实现的难点。用有限状态机(finite state machine) 来记录状态间的变化转移,并执行相应的操作,可以大为简化程序

的设计、提高编程效率,更能增强源代码的可读性,是实现命令解释执行体的理想方法。状态宏定义如下：

```
#define STATE_NORMAL 0
#define STATE_IAC      1
#define STATE_WILL     2
#define STATE_WONT     3
#define STATE_DO       4
#define STATE_DONT     5
#define STATE_CLOSE    6
```

TELENT 的所有命令码的宏定义：

```
#define TELNET_IAC     255
#define TELNET_WILL    251
#define TELNET_WONT    252
#define TELNET_DO      253
#define TELNET_DONT    254
```

命令解释器的有限状态机的实现子程序：

```
static void newdata(struct telnetd_state *s)   /*端口 23 收到新的 TCP 数据*/
{
  u16_t len;
  u8_t c;
  len = uip_datalen();        /*接收数据长度*/
   while(len > 0 && s->bufptr < sizeof(s->buf)) { /*处理接受数据*/
    c = *uip_appdata;
    ++uip_appdata;
    --len;
    switch(s->state) {
    case STATE_IAC:
      if(c == TELNET_IAC) {
    getchar(s, c);
    s->state = STATE_NORMAL;
      } else {
    switch(c) {
    case TELNET_WILL:
      s->state = STATE_WILL;
      break;
    case TELNET_WONT:
      s->state = STATE_WONT;
      break;
    case TELNET_DO:
      s->state = STATE_DO;
      break;
    case TELNET_DONT:
      s->state = STATE_DONT;
      break;
```

```
    default:
      s->state = STATE_NORMAL;
      break;
    }
      }
      break;
    case STATE_WILL:
      /* Reply with a DONT */
      sendopt(s, TELNET_DONT, c);
      s->state = STATE_NORMAL;
      break;

    case STATE_WONT:
      /* Reply with a DONT */
      sendopt(s, TELNET_DONT, c);
      s->state = STATE_NORMAL;
      break;
    case STATE_DO:
      /* Reply with a WONT */
      sendopt(s, TELNET_WONT, c);
      s->state = STATE_NORMAL;
      break;
    case STATE_DONT:
      /* Reply with a WONT */
      sendopt(s, TELNET_WONT, c);
      s->state = STATE_NORMAL;
      break;
    case STATE_NORMAL:
      if(c == TELNET_IAC) {
    s->state = STATE_IAC;
      } else {
    getchar(s, c);
      }
      break;
    }
  }
}
```

TELNET 服务也是建立在应用服务接口之上,利用 TCP 协议实现传输。在 TELNET 连接之上，实现了一个文件操作的命令 SHELL 界面。提供了文件列表、文件读、文件写、文件删除等基本文件操作。这样在具体应用中，可以利用 TELNET 终端直接登陆到系统上来，对于文件进行维护和操作。这样极大的扩展了系统的灵活性。方面了系统的设置操作。TELNET 服务应用程序如图 6.5 所示。

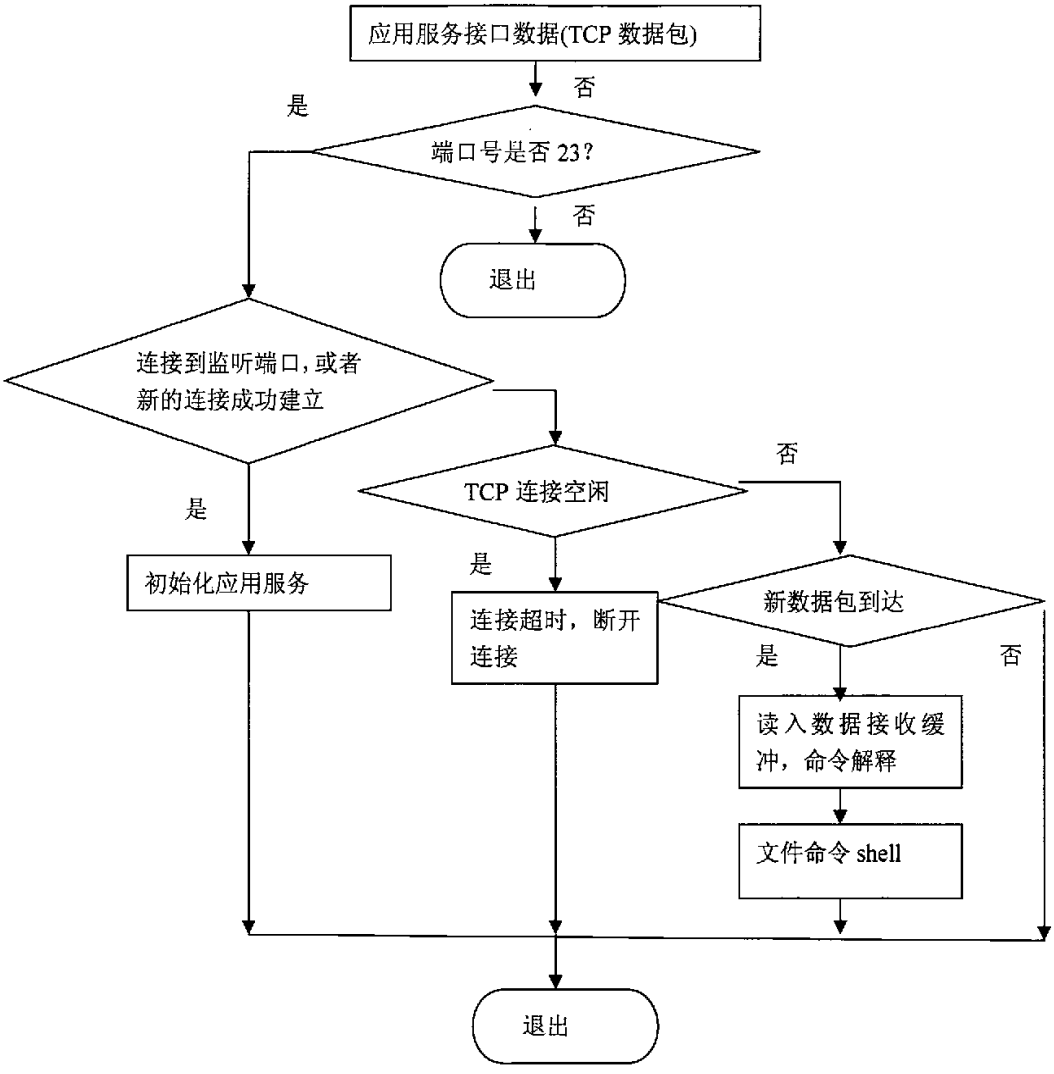


图 6.5 TELNET 服务应用程序框架

第七章 文件系统

[31]文件系统是建立在存储设备的分割区域之上，用于存放系统的数据资料，一个文件系统在使用之前，必须先挂接在系统树形目录结构的某个接点上（例如 UNIX 系统的 MOUNT），同理也可以通过 UMOUNT，将其系统目录结构上除去。其中属性目录是组成文件系统的主要数据结构，和组织形式。目前常见的文件系统主要有：

1. Ext2:

Ext2 是 GNU/Linux 文件系统，其特点是文件存取性能较好，适用于中小型文件存取，其单一文件大小与文件系统数据区块的大小有关，在 PC 机中，数据区块最大为 4KB,所以单个文件上限为 2048GB,文件系统容量上限为 16384GB。

2. BSD 操作平台文件系统:

BSD 是除 LINUX 以外的另一个开源的 UNIX 操作系统，主要包括：FreeBSD、NetBSD、OpenBSD 等，其中 FreeBSD 最为常见。它的文件系统为 UFS,这个文件系统也应用于 SunOS、NextStep。

3. WINDOWS 操作系统:

微软平台的操作系统包括 MSDOS(使用于 MS-DOS),VFAT(用于 WIN95,WIN98,WINME),NTFS(用于 WINNT)。其中 VFAT 和 MSDOS 基本类似，差别只是 VFAT 支持长文件名。

4. 光盘文件系统:

这种文件系统主要包括 ISO9660，以及最近新开发的 UDF,前者用于一般的 CDROM,后者用于 DVD,这两个都是只读文件系统。

5. MINIX:

此文件系统是由阿姆斯特丹的 Andrew S. Tanenbaum 教授为了教学用开发的类似于 UNIX 操作系统。其功能比较简单，是 LINUX 早期发展使用的文件系统。

6. hpfs: OS/2 平台的文件系统。

7. qnx4fs: 用于 QNX 4 文件系统。

8. 日志式文件系统:

当文件操作正在进行时，如果中断（断电），有可能造成文件系统的错误，造成文件系统的数据不一致性，当这种情况出现时，需要进行文件重整工作，这一工作往往相当耗时，特别是容量大的文件系统。为了克服这个问题，出现了所谓的日志式文件系统。(Journal File System)，这类文件系统最大的特点是，他将针对于磁盘的写入操作记录在磁盘的某的区域上面，以便于需要时可以回溯操作。所以在日志式文件系统中，由于详细记录的文件操作的每一个细节，当某个操作被中断时，系统可以根据这些记录直接回溯到操作以前的状态。目前在 GNU/Linux 上 ext3 属于此类文件系统。它就是在保持 ext2 的格式之下再加上日志功能。

9. 嵌入式文件系统:

这一类文件系统一般拥有的存储介质容量很小，一般需要特别的设计。嵌入式文件系统主要有:

cramfs: 其是 Compressed ROM File System，只读文件系统，其容量上限只有 256MB。

romfs: 也是一个很小的只读文件系统。

jffs: 此为日志式快速(flash)文件系统，用于嵌入式系统。

Ramfs,tmpfs: 此文件系统可以用来将数据暂时保存在内存 RAM 中，而且其容量可以随着保存数据的量而增减。

一般的嵌入式系统都配置很少的外部存储空间甚至不带外部磁盘。但随着用户需求和功能复杂度的增加，越来越多的嵌入式系统需要处理大容量的数据，或者在运行过程中会产生大量的临时数

据。一方面这些数据处理完后不能立即删除；另一方面这些临时文件不需要长期保存。例如，用来上网冲浪的机顶盒设备在用户浏览过程中不断从互联网上接收数据，因此用户访问后的页面很可能再次浏览，所不能将浏览后的网页立即清除，当然，系统不需要也不可能将所有浏览过的页面保存于硬盘中。所以，处理数据量的增大给嵌入式系统的设计提供了新的要求。

这里提出一种用于嵌入式系统的文件系统，它使用单级目录结构。它既可以应用于 FLASH 线性系统中，也可以用于 ROM 只读文件系统，也可以作为存储大容量临时数据的内存文件系统。对于内存文件系统来说，其具有存取速度快、可动态改变文件系统大小的优点，但是数据掉电即丢失，Linux 下的 Tmpfs、Proc 文件系统以及 Freebsd 下的 MFS 都是一种内存文件系统。但是，这些通用操作系统上的内存文件系统不能够直接运用于到嵌入式系统中：其一，它们都是为资源丰富的通用 PC 平台设计的，不适用于资源有限的嵌入式系统；其二，这些通用内存文件系统的设计方案一般是利用内存来模拟磁盘文件系统，在内存中会建立文件系统缓冲区。这就是说除了文件系统本身占据了内存之外，磁盘缓冲区又会占所一些内存，这些就会导致内存的浪费和利用率的下降。

7.1 文件系统构架

此嵌入式文件系统构建于层次结构基础之上。其结构如图 7.1 所示。

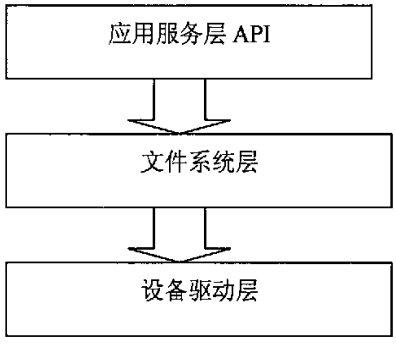


图 7.1 文件系统架构图

1．应用服务 API 层。

API 层是文件系统与用户应用程序之间的接口，包含一个与文件函数相关的函数库，如 FS_FOpen、FS_Fwrite 等，也相当于 High Level 层。这一层相当于文件系统和用户之间的界面和接口，其中包括了面向于 ANSI C 文件操作库函数，主要有文件创建 Fnew，文件打开 FOpen,文件读 Fopen,文件写 Fwrite，文件删除 FDel 等基本操作，API 层将文件的操作传递给文件系统层，此文件系统目前不兼容 FAT 等文件系统，但是这一层的文件操作的功能调用可以处理不同的文件系统层，这样可以处理其他的文件系统。

2．文件系统层。

文件系统层，即 Low Level 层。该层读取结构化描述文件系统，描述具体文件系统的树形目录结构，该层处理文件是否存在，打开，关闭和为文件分配相应的缓存等。该层调用低层驱动。

3．设备驱动层。

Device Level 层，就是设备驱动层。对于存储设备的实际读写操作就是在该层进行的，特定的存储器对应特定的读写程序。结构化的设备驱动层可以简化硬件存储设备的集成。

这里我们将设备驱动层用 RAM 来模拟，这样相当于一个内存文件系统。可以加入其他设备的驱动程序，同样可以支持 FLAG , SMART MEDIA CAR,MULTIMEDIA CARD 等文件系统。

7.2 线性文件系统的设计

存储设备的文件组织结构如图 7.2 所示。基本存储结构为 512 字节，一个扇区。

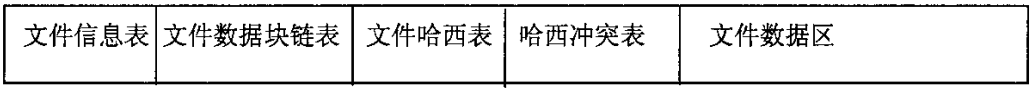


图 7.2 文件系统的组织结构

文件系统中的每一个文件由文件头和文件数据区两个部分组成，每个文件头(文件哈西表的查询结果)存放在文件信息表中，文件信息表以单向链表的结构存储。文件头，包含文件的属性、指向下一个文件头的指针、文件头和文件数据区的 32 位循环冗余校验和（CRC32）等。文件头用一个 16 位的字来表示文件属性，每位表示一种属性，如数据文件或者是可执行文件，是否已删除的文件等，具体可以根据应用的需要来定义文件的属性。为了加快文件操作的速度，通过文件的名称形成全局哈西表，对应于文件信息表的入口项，通过文件信息表，得到文件头信息，通过文件信息头，查找文件数据块链表，就可以对存放在文件数据区的文件数据进行操作。

文件头的结构如下：

```
struct file_head{
unsigned short hdrsize; /*文件头字节数*/
long filsize; /*文件头版本*/
long filsize; /*文件大小*/
long flags; /*描述文件的标识*/
unsigned long filcrc;  /*文件数据的 CRC32 的值*/
unsigned long hdrcec; /*文件的最后修改时间*/
unsigned long hashindex; /*文件对应哈西表项*/
struct hdr *next; /*指向下一个文件头的指针*/
char name[NAMESIZE]; /*文件名*/
char info[INFOSIZE]; /*文件描述信息*/
};
```

文件句柄结构体：

```
struct file{
unsigned int Buffer[CLUSTER_BLOCK_SIZE];//文件缓冲区
unsigned int fileblock;//文件当前簇的位置
unsigned int filemode;//打开支持的模式
unsigned int filebufnum;//文件缓冲区中已被/写的字节数
unsigned int fileCurpos;//文件读写的当前位置
unsigned int filesize;//文件的大小
};
```

文件应用层函数。

```
Void FS_Exit(void); /*卸载文件系统*/
Void FS_Init(void); /*加载文件系统*/
File *  FOpen(const char * name,const char * mode); /*打开文件*/
Void Fclose(file fp); /*关闭文件*/
file_size FRead(void ptr, file fp); /*读打开文件*/
file_size FWrite(void ptr, file fp);/*写打开文件*/
int FSeek(FS_FILE *fp, unsigned long offset, int mode);/*文件定位*/
```

```
file_ptr FTell(FS_FILE *fp); /*返回打开文件定位指针*/
int FError(file *fp); /*返回文件操作错误信息*/
int FRemove(const char * name); /*删除文件*/
```

设备驱动层基本函数

```
int Init_device(void); /*初始化存储设备*/
int dev_open(char *path,unsigned flags,int fno); /*打开存储设备*/
int dev_close(int fno); /*关闭存储设备*/
int dev_read(int fno,char *buffer,unsigned count); /*读存储设备*/
int dev_write(int fno,char buffer,unsigned count);/*写存储设备*/
```

1. 读文件流程

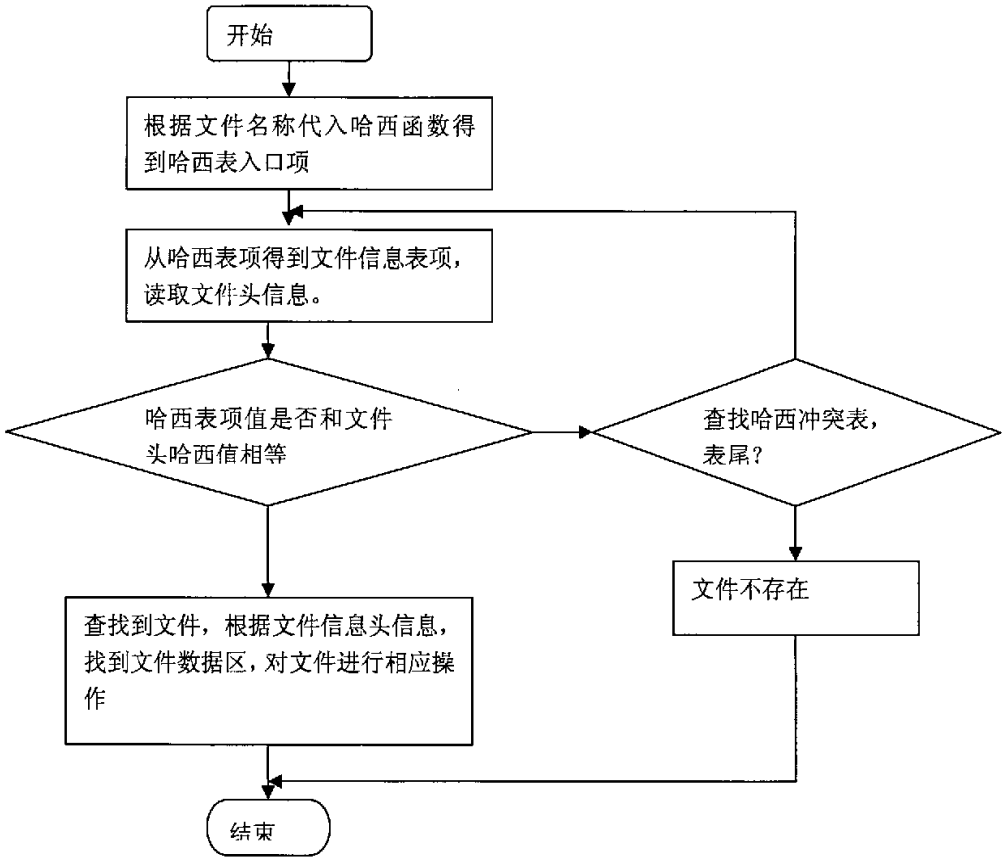


图 7.3 读文件流程

2. 创建文件流程

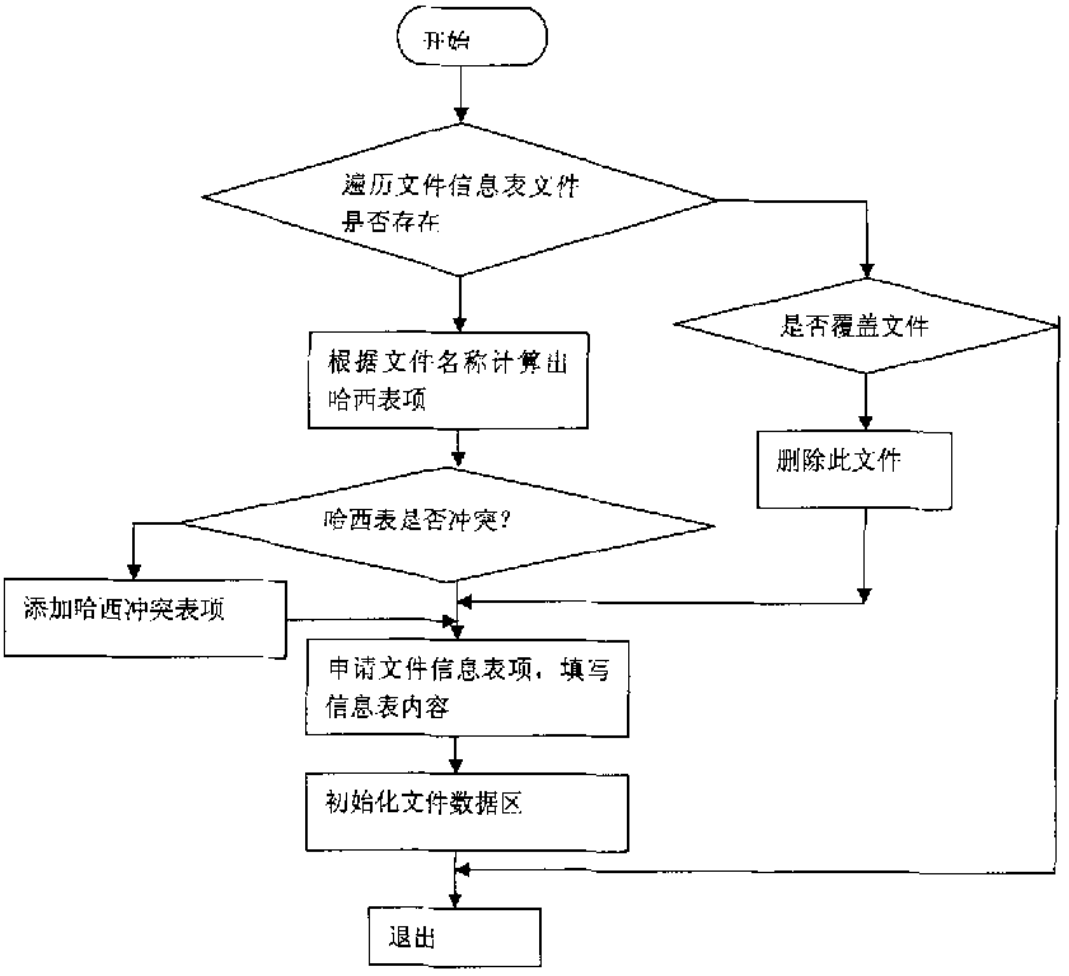


图 7.4 建立文件流程

3．碎片整理

创建新文件需要占用文件系统空间；但是删除一个文件的时候，暂时没有把整个文件所占的空间删除，仅仅是在文件头的标识里作一个删除标识，并保留在存储设备中。这样的好处对于 falsh 存储设备来说时显而易见得，这样可以减少 flash 删除操作的频率，加快删除操作，但是这样，被删除文件积累到一定的数量时，就会占用相当大的空间。因此，需要整理文件系统空间，使被删除文件占用的空间重新使用。

进行碎片整理的方法可以有多种。对于嵌入式系统来说，选择哪种方法，衡量的依据是复杂性和功能之间的平衡。下面讨论两种不同的方法：第一种方法相当简单，但是有缺陷；第二种方法功能强大得多。当然，存在更加复杂的解决办法，但通常的情况是，所添加的复杂性会使整个文件系统的实现更加复杂。目标是保持文件存储的简单和线性，保证所有的文件都是以连续的空间存储在存储设备中。

最简单的方法是将活动的文件备份在 RAM 中，删除分配给文件系统的存储空间，然后将 RAM 中备份的所有文件拷贝回存储设备。这种方法很简单，并且不需要分配一个扇区作为空闲区。但问题是，需要有一整块和分配给文件系统的空间一样大的 RAM 来完成这项工作。更糟的是，如果此时系统被复位，或者在删除扇区内容却还没有将文件拷贝回存储设备的时候被断电，文件系统将会崩溃。因为 RAM 中的内容会随之选择，文件内容会被破坏掉。

我们在文件系统实现设计了一种碎片整理方法，可以防止在碎片整理过程中系统复位导致文件

崩溃的情况。采用这种方法，不需要大块的 RAM，但是需要预选先分配给碎片整理过程一个 Flash 扇区作为备份区。这个扇区的字节数不小于任何分配给文件系统的扇区。在整个文件系统中，这个扇区位于分配给文件系统最后一个扇区的下一个扇区。因为扇区可能比需要分配给非删除文件的备份的空间要小，所以它必须逐个扇区进行处理，而不是一下就把所有的碎片整理完。采用备份扇区的好处是，在碎片整理过程中，无论断电或者复位都不会破坏文件系统。当下次系统重新恢复时，会根据在碎片整理前记录的每个扇区碎片整理前后 CRC 值，来判断当前的文件碎片整理状态。如果上次文件整理没有完成，就会继续上次的整理。这种技术的一个缺陷是空闲扇区的擦写次数会较多。这样空闲扇区就可能因为达到擦写寿命而失败。达到这一点的关键依赖于使用的 Flash、所分配给文件系统的扇区数、文件删除和重建的频率。一个可行的解决办法采用电池备份的 RAM 来替换空闲扇区，可以增加 Flash 的整体寿命，但是对那些预算紧张的应用来说太过奢移。

具体的碎片整理过程是，首先建立碎片整理区。①为每个扇区建立 2 个 CRC32 表项：第一个 CRC32 是这个扇区在碎片整理前的 CRC 值；第二个 CRC32 值是计算出来的碎片整理后的 CRC32 值。这些 CRC 是当碎片整理过程被打断时，用来重新恢复整理用的。②创建碎片整理文件头信息表，每个活动的文件占用一个表项。③计算①和②的 CRC 值，并保存。①～③的数据保存在碎片整理记录区。第二步是文件重定位；遍历文件系统的每个扇区，处理重新定位后存储空间和该扇区相覆盖的文件。在每个扇区被重写之前，扇区原来的信息被保存在空闲扇区里。第三步，擦除 Flash；遍历未使用的扇区，确认所有的扇区被删除。第四步，完整性检测：对新的文件进行检测，保证所有重定位的文件都是完整的。

7.3 文件系统和 TELNET 服务的结合

这里我们基于 TELNET 服务提供了一个文件操作的命令 SHELL，这样可以通过 TELNET 登陆到系统的 23 号端口，通过 FILE SHELL 提供的命令对于文件系统进行维护和操作。这里提供了 mount,umount,dir,write,read 等一些基本的文件操作命令。

命令解释器的核心数据结构如下：

```
struct command_function {
  char c[10];
  void (* pfunc)(struct telnetd_state *s, char *str);
};

static struct command_function config_para[] =
  {{"stats", stats},    /*显示当前连接状态*/
   {"exit", exitt},     /*退出 telnet*/
   {'?', help},         /*命令帮助*/
   {"write", write}, /*写文件操作*/
   {"read", read},    /*读文件操作*/
   {"dir", dir},       /*文件列表*/
   {"delete", delete}, /*文件删除*/
   {"copy",copy}, /*文件拷贝*/
   ……..
   /* Default action */
   {0, none}};     /*输入命令错误*/
```

这样的结构的优点是，命令的增加和修改和真个体系结构无关，添加一条命令，只需要在

config_para 添加一个新的数据项（命令名称，执行命令的函数）即可。这样系统的扩展非常灵活。

第八章 测试环境和验证

本嵌入式系统的主要基于 ARM7 MCU 硬件平台的基础上开发和测试。主要的开发集成环境采用 ADS1.2(ARM Developer suite V1.2),测试和调试环境基于 ADS 集成开发环境中的 AXD DEBUGGER。 AXD 可以基于 ARMULATOR 仿真平台，对于 ARM7 结构的代码进行调试和测试。其中 ARMULATOR 是一个非常灵活和开放的模拟开发平台，ADS 的 ARMULATOR 提供了 ARM7 MCU、基本内存和内存映射、WATCH DOG、SMIHOST 等用于调试的一个最小系统环境。本论文的所有代码都是基于这个测试平台进行的。ADS 的 AXD ARMULATOR 非常灵活，在这个基本框架上面可以增加自己的外设和硬件的仿真，ADS ARMULATOR 仿真环境和实际的硬件电路构架基本类似，因为 ARM 也是内存统一地址分配的体系结构,也就是中断向量，程序存储器，数据存储器和外设控制寄存器、数据寄存器和地址寄存器是统一地址分配的，所以对于 ARMULATOR 的扩展非常方便，在 ARMULATOR 上调试成功的程序，基本上不需要修改就可以平滑移植到实际硬件平台上，这样整个开发精力可以集中在系统的逻辑驱动层以上。这里我们在 ARMULATOR 已有的仿真平台上，添加了两个时钟计数器、一个键盘、一个 480X320 的 8BIT 彩色 LCD、一个以太网驱动、和 16M 的 FLASH 存储器的仿真外设模块。从而利用这个仿真平台，可以测试实时操作系统内核、图形用户界面、TCP/IP 协议栈和文件系统，基本能够满足调试和测试整个系统的需要。这一章首先介绍了 ADS 集成开发环境，然后集中讲述了 AXD ARMULATOR 的仿真环境，和仿真外设的具体扩展方法和调试测试方法，最后给出了整个系统的测试方法和结果。

8.1 ADS 集成开发环境

ADS(ARM Developer Suit)是基于 ARM 系列 RISC 的集成开发工具。可以利用其开发、建立和调试基于 C、C++和 ARM 汇编的程序。

ADS 主要包括以下一些主要的构件：

•基于命令行方式的开发工具

主要提供了编译链接等命令工具。例如 armcc ARM C 编译工具； armcpp ARM C++编译工具；tcpp Thumb 指令集的 C++ 编译工具；armasm ARM 指令和 Thumb 指令的汇编编译工具；armlink ARM 链接工具，产生基于 ELF 格式的执行映像文件；armsd ARM 调试工具。

• GUI 集成开发工具

AXD 是基于 ARM 的调试工具，有 Windows 和 UNIX 版本，这个工具提供了一个比较全面的集成 C,C++和汇编语言的调试功能。

CodeWarrior IDE 是一个程序开发的主开发界面。提供了代码管理和建立的开发集成环境。

• 公共工具集

主要提供了一些便于开发的工具集，例如：fromELF 提供了 ELF 格式到 ARM image 格式转换工具，可以转换为 Motorola 32-bit S-record 格式, Intel Hex 32 格式,和 Verilog-like hex 格式;Flash downloader 可以下载二进制文件到 ARM 开发办的 FALSH 存储器中。

• 其它协助工具.

ARMulator ARM 内核仿真工具，它提供了精确的机器代码级别的仿真功能，它集成于 ADX debugger 工具中。

8.2 AXD DEBUGGING 和 ARMULATOR

AXD 提供了非常强大的调试手段,它的调试目标可以是软件虚拟的 ARMulator 仿真器，也可以是各种实际的开发板。其主要架构如图 8.1 所示。

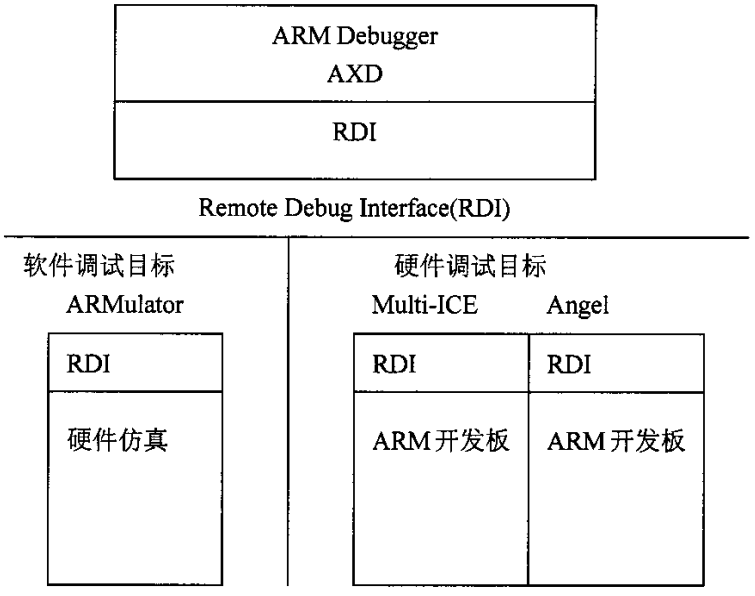


图 8.1AXD 调试接口

AXD的调试目标可以是软件仿真也可以是通过Multi-ICE和Angel为调试接口的实际硬件电路。其中ARMulator提供了ARM指令集，ARM CPU内核和外围扩展电路的软件仿真。Multi-ICE和EmbededICE是基于JTAG的ARM调试工具。它提供了调试工具和实际嵌入式ARM内核的调试界面接口。

ARMulator 是一个在 ARM 公司推出的集成开发环境 ADS （ARM Developer Suite）中提供的指令集模拟器。它与运行在通用计算机（通常是x86体系结构）上的调试器相连接，模拟 ARM 微处理器体系结构和指令集，提供了开发和调试 ARM 程序的软件仿真环境。ARMulator 不仅可以仿真 ARM 处理器的体系结构和指令集，还可以仿真存储器和处理器外围设备，例如中断控制器和定时器等，这样就模拟了一个进行嵌入式开发的最小子系统，另外使用者还可以扩展添加自己的外设。ARMulator 同时支持全部的标准C库函数，这样所有的C程序都可以在仿真器上运行。ARMulator 通过 Semihosting 方式来实现程序中输入/输出功能。Semihosting是指在 ARM 程序开发调试的过程中，目标系统还不具备输入/输出的功能，通过调试代理向调试器请求 I/O。ARMulator 就是这里所说的调试代理之一，其他还有 Angel、Multi-ICE也通过类似的方式来实现 I/O 功能。ARMulator 在 Windows 平台上实现为动态链接库文件（.dll），在 Linux 或者 Solaris 系统上实现为共享库文件（.so）。前者的使用一般是通过 ARM 公司提供的图形界面调试器 AXD 中，选择目标系统为 ARMulate.dll，这样就设置了 ARMulator 作为调试代理。用户在进行调试开发时，对于底层使用的调试代理是透明的。

ARMulator 提供了软件仿真调试的平台，它仿真了 ARM 指令集、ARM 内核架构、内存系统和其他一些外围扩展电路。我们可以在这个基础之上加入其他的外围硬件电路仿真，可以利用 ARMulator 开发和评测基于 ARM 体系机构的各种应用。ARMulator 的主要架构如图 8.2 所示。

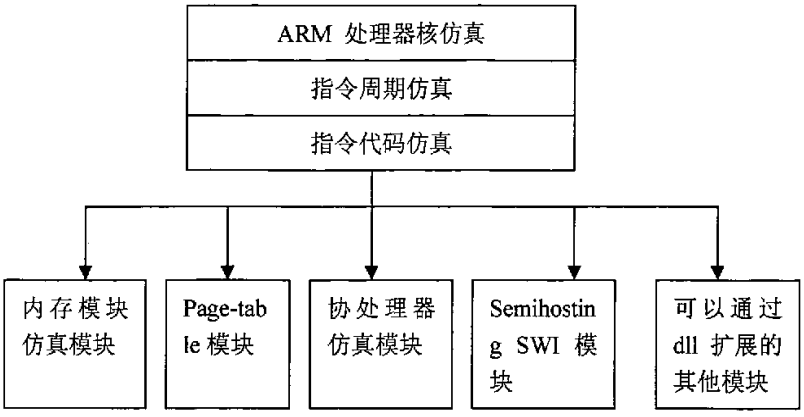


图8.2ARMulator仿真架构

ARMulator 提供了统一的接口界面扩展系统外围设备，按照这种接口界面可以很容易的增加需要的外设，而不会影响已有的外设模块。外部模块的扩展是通过 windows dll 来实现的，ARMulator 提供了丰富的 ARMulator 开发库，可以围绕内存模块开发各种应用模块。ARMulator 本身是由一系列的模块（module）组成的，其中最主要的模块包括： ARM 处理器核心模块，用来仿真处理器的体系结构。 处理器使用的存储器模块，用来仿真存储系统。缺省的内存模型是 flatmem ，内存的大小不限制，理论上 4G 的地址空间都可以使用。 同时，ARMulator 也包含了几种外设模块，用来仿真如下的外设模型：

1) Interrupt controller（中断控制器） 用于中断控制，包含状态寄存器、使能寄存器、状态位清除寄存器等。寄存器基址：Base=0x0a000000

2) Timer（定时器）

ARMulator 提供了两个定时器。包含控制寄存器、计数值寄存器、中断清除寄存器等。定时器支持 free-running 和 periodic 两种工作方式，前者计数器从 0xFFFF 递减，到“0”后计数器溢出发生中断。后者计数器从计数值寄存器中读取计数值，然后递减至“0”后溢出发生中断。 寄存器基址：Base=0x0a800000

3) Watchdog（看门狗）

看门狗是为了防止用户程序出错导致系统死锁而采用的一种保护手段。一旦到了预定时间没有设置看门狗，看门狗就会停止 ARMulator 的运行，返回调试器中。看门狗使用 2 个定时器工作，第一个定时器在复位时启动，或者通过程序写入 KeyValue 寄存器预先设定好的值来启动，当过了一定时钟周期（WatchPeriod）后，第一个定时器产生 IRQ 中断，并启动第二个定时器；如果在规定时间（IRQPeriod）内程序没有写 KeyValue 寄存器，则看门狗将停止 ARMulator ，返回调试器中。寄存器基址：Base=0xb0000000 KeyValue=0x12345678 ，WatchPeriod=0x80000 IRQPeriod=3000 IntNumber=16 .

4) Stack tracker（栈跟踪器）

栈跟踪器在每条指令执行后，检查栈指针（r13）寄存器的值。它记录下栈顶的最小值，从而可以计算出所用堆栈的最大长度。 栈底地址：StackBase=0x80000000 栈空间限制：StackLimit=0x70000000.

5) Tube（显示管）

Tube 实际上是一个寄存器，它用来显示输出。一旦写入这个寄存器某个可以显示的字符，则调试器将会把它显示在控制台 Console 下。这样用来显示对某个指定地址的写操作记录。 寄存器基址：Base=0x0d800020

使用者也可以根据一定的接口规范编写自己的外设模块，或者通过修改已有模块的方式来适合自己的需求。比如定制自己的存储系统等。

8.3 基于 ARMULATOR 的仿真系统的扩展

图 8.3 内存分配

这里为了调试和验证嵌入式实时操作系统内核、用户图形用户界、TCP/IP 协议栈和文件系统，在 ARMulator 的基础之上扩展了计数器/定时器从而为操作系统内核提供实时中断；键盘和 LCD 提供了图形用户界面的硬件基础；以太网驱动主要提供了调试 TCP/IP 协议栈的硬件平台；仿真 FLASH 存储设备，在用来调试文件系统。内存分配和系统架构如图 8.3 所示。

8.3.1 计数器/定时器仿真模块的实现

这里定义了两个计数器/定时器，其控制寄存器的定义如下：

```
/*********************************************************/
/*Counter/timer registers                                */
/*********************************************************/
```

```
#define TimerBase 0x0A800000 /*Counter/Timer Base              */
#define Timer1Load          ((volatile unsigned *)TimerBase)
#define Timer1Value         ((volatile unsigned *)(TimerBase + 0x04))
#define Timer1Control       ((volatile unsigned *)(TimerBase + 0x08))
#define Timer1Clear         ((volatile unsigned *)(TimerBase + 0x0C))

#define Timer2Load          ((volatile unsigned *)(TimerBase + 0x020))
#define Timer2Value         ((volatile unsigned *)(TimerBase + 0x024))
#define Timer2Control       ((volatile unsigned *)(TimerBase + 0x028))
#define Timer2Clear         ((volatile unsigned *)(TimerBase + 0x02C))
```

其工作模式设置成为周期计数方式，计数满产生中断 IRQ(0x0000_0018)，中断标志寄存器的第 4 位和第 5 位。

```
#define IRQTimer1           0x0010
#define IRQTimer2           0x0020
```

其计数方式为每周期自减计数，范围 0x0000-0xffff，所以频率为系统主频的 2-65535 分频，这个周期时钟中断为系统内核提供实时时钟，作为系统任务切换和时间片轮询的依据。

8.3.2 键盘和彩色 LCD 的仿真模块的实现

键盘和 LCD 定义和地址分配如下：

```
/* This constant is used by both model and viewer */
#define BITS_PER_PIXEL 8
/* These are defaults used if not found in peripherals.ami */
#define LCD_WIDTH    (unsigned)480
#define LCD_HEIGHT   (unsigned)360
/* Display Registers */
#define DISP_BASE   0x0C000000
#define DISP_CSR    (DISP_BASE+0x0004)    /* Display Control Reg    */
#define DISP_ISR    (DISP_BASE+0x0000)    /* Display interrupt Reg */
#define DISP_XSIZE  (DISP_BASE+0x0008)    /* Display Width Reg      */
#define DISP_YSIZE  (DISP_BASE+0x000C)    /* Display Height Reg     */
#define DISPLAY_PTR (DISP_BASE + 0x10)
// Reserve enough for 1024x768x8 bit display
#define REG_BASE (DISPLAY_PTR + 1024*768)
/* Main interrupt registers */
#define CPU_BASE     REG_BASE
#define CPU_ISR      (CPU_BASE+0x0)    /* cpu interrupt status */
#define CPU_MR       (CPU_BASE+0x4)    /* cpu mask register      */
/* Keyboard registers */
#define KB_BASE                        (CPU_MR + 0x08)
#define KB_ISR   (KB_BASE + 0x0004)   /* offset from keyboard base read */
#define KB_CSR (KB_BASE + 0x0000)   /* offset from keyboard base read */
#define KB_INTR_MASK                    0x0001
#define KB_RDRF                         0x0001
```

```
#define KB_PARITY              0x0100
#define KB_DATA_RD             0x0200
#define KB_CLK_RD              0x0400
#define KB_DATA_DR             0x0800
#define KB_CLK_DR              0x1000
#define KB_DATA_DR_EN          0x2000
#define KB_CLK_DR_EN           0x4000
#define KB_CLK_EN              0x8000
#define KEYB_INTR              0x00000040
```

其实现原理图如图 8.4 所示。

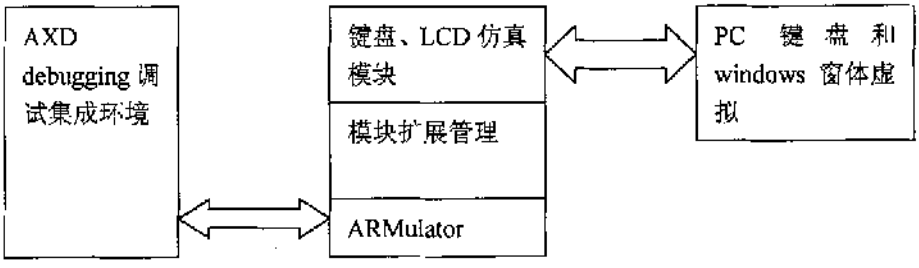


图 8.4 键盘和 LCD 仿真

通过 VC6.0 编写基于 ARMulator 扩展模块的接口界面的键盘和 LCD 仿真模块，利用 PC 机键盘和 windows 窗口来虚拟仿真嵌入式系统键盘和 LCD 显示器。

8.3.3 以太网驱动模块的仿真实现

以太网驱动的存储器分配如下。

```
#define EthernetBase     0xa2000000
#define EthernetPRlen    0xa2000004
#define EthernetPSlen    0xa2000008
#define EthernetPaket    0xa2000010
```

实现原理如图 8.5 所示。

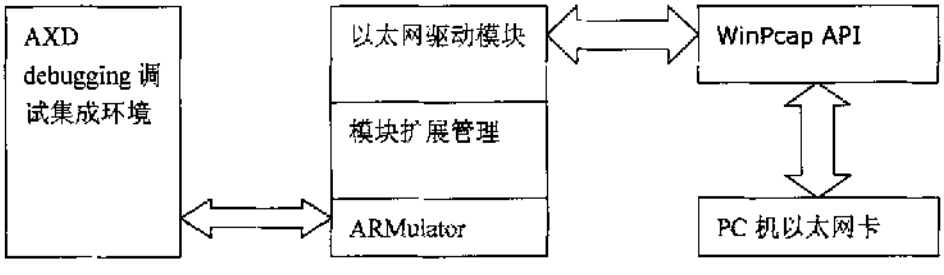


图 8.5 以太网驱动模块仿真原理

这里利用 PC 机的以太网卡来仿真嵌入式系统的以太网驱动模块。其中 WinPcap 是开源的网络数据包驱动软件，它提供了一组 windows 动态连接库，实现了以太网数据包的抓取和发送功能。它

可以直接驱动 windows 系统的网络适配器，通过这一组 API 函数，可以实现了以太网数据包和嵌入式系统的交互，从而和系统的 TCP/IP 协议栈连接起来，实现了基于其上的 HTTP 和 TELNET 的测试。

8.4 系统的测试结果

系统调试界面如图 8.6 所示，图中显示了，利用 AXD 的 Arumulator 模拟器，调试 micro-DDROTS 的工具界面。通过 armulator 平台，虚拟了，ARM7 内核和 LCD、TIMER、KEY、NET 等外围设备。

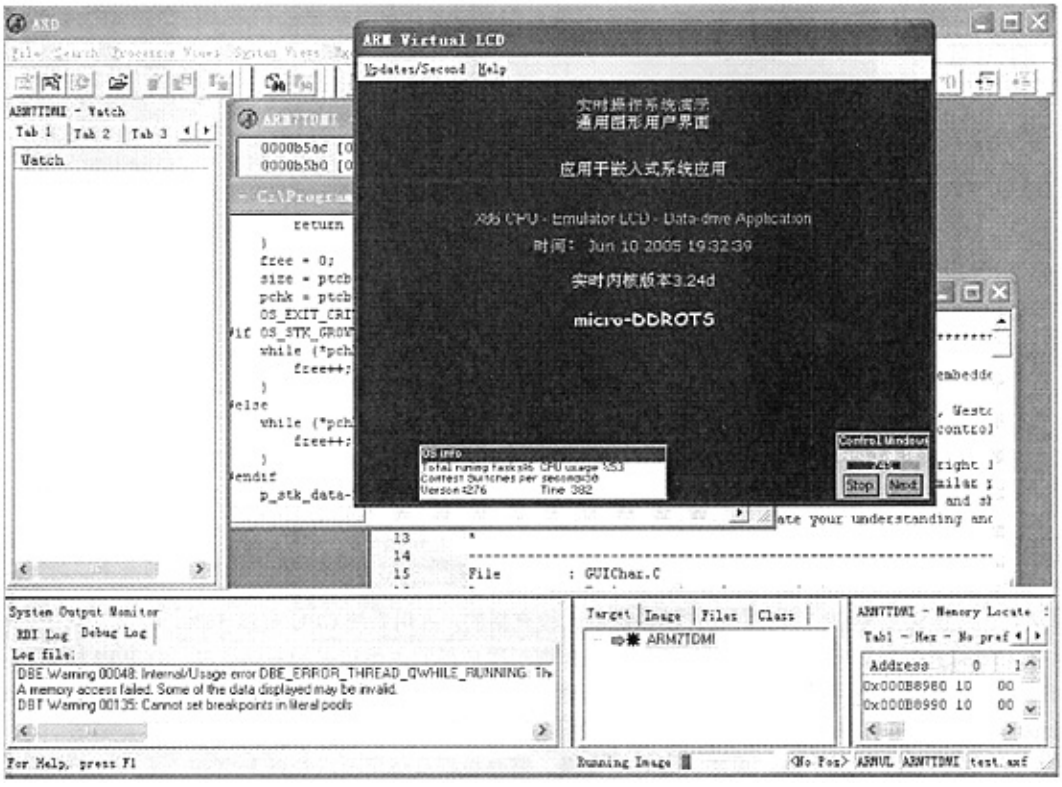


图 8.6 系统内核调试界面

1. 实时性能指标

衡量计算机系统的实时性，有一些专门的测试方法，其中 Rhealstone 方法[22] 定义了以下 6 项指标来衡量实时系统的性能:

1) 任务切换(Task Sw itch) 时间，即系统在两个独立的、处在激活态并具有相同优先级的任务之间切换所需要的平均时间。切换时间取决于在存入和恢复上下文时处理数据结构的效率以及计算机的 CPU 的结构和指令集。
2) 抢先(P reemp tion) 时间，即系统将控制从低优先级的任务转移到高优先级的任务所花费的平均时间。抢先时间类似于任务切换时间,只是时间较长。系统为了抢先，必须首先识别引起高优先级任务激活的事件，再对两个任务(正在执行的任务和提出请求的任务) 的相对优先级进行存和取，然后才进行任务切换。
3) 中断等待时间(L atency) 即从 CPU 收到中断请求到执行中断服务程序的第一条指令所用的时间。它既取决于操作系统的效率，也取决于处理外部中断的处理器的效率。
4) 信号灯(Semaphore) 交替延迟时间是指从一个任务释放信号灯到另一个等待信号灯的任务被激

活间的时间延迟。在实时系统中，通常有许多任务同时竞争同样的资源，使用信号灯的互斥是保证任一时刻只有一个任务存取公共资源的有效方法。信号灯交替时间反映了与互斥有关的时间开销，因此也是衡量系统实时性能的重要指标。

5) 死锁解脱时间是指系统解开死锁所需的平均时间。死锁解脱时间反映了操作系统解决死锁的算法的效率。

6) 数据报(DA TA GRAM) 通过率是当一个任务，调用实时操作系统的原语，把数据传送到另一个任务去时，每秒传送的字节数。数据报通过率度量了任务间通讯的平均速率。

这里设定系统主频为：20MHZ,设置定时器/计数器值为：0x1FF, Time slot=100ms,测试了 4 项性能指标。

任务切换时间μs	任务抢先时间μs	消息传输延迟μs	信号灯交替时间μs
11	12	15	18

2. 负载测试

测试负载：数据通道数据每周期个数：20

TCP/IP 栈及应用：ping 测试数据包（800ms）

GUI 图形用户界面: 正弦函数作图

其他辅助任务：统计任务，延时任务，空闲任务

	数据通道			时间	TCP/IP 栈及应用	GUI 图形用户界面	辅助任务
	数据接收	数据处理	数据发送				
占用时间	3	50	2	1	11	121	36
总任务比例	0.5%	9.7%	0.39%	0.1%	2.15%	23.6%	7%
堆栈使用	521B	1021B	300B	231B	2048B	4000B	580B

CPU 利用率：54%， 上下文切换：50 次/time_slot.[72]

从测试数据可以看出，系统完成有任务还是比较充裕的，占用系统 CPU 资源 54%，也就是系统可以有 46%的资源用于应用程序负载的增加。GUI 图形用户界面占用较多的系统资源，TCP/IP 栈及其引用和辅助任务都占用了较少的系统资源。这里对于 Time_slot 的选择主要考虑两个方面的问题。Time_slot 选取的周期越长则系统 CPU 的时间利用资源越多。而所有任务的 CPU 利用率则降低。但是系统的实时性能有所下降，系统响应时间变慢。需要根据具体用要求加以考虑。

3. 扩展应用的测试

图 8.7 显示了在 LCD 上的 GUI 图形用户界面，利用作图函数作图的测试画面。

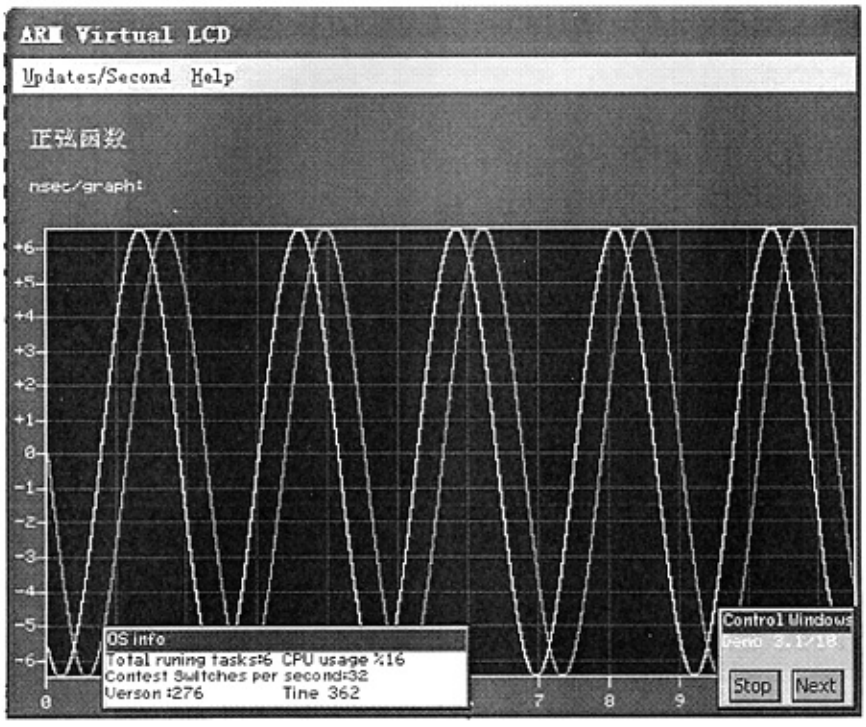


图 8.7 图形用户界面作图函数应用

图 8.8 显示了 TCP/IP 协议栈的测试结果。图中显示了，基于 TCP/IP 的 WEB SERVER 应用的调试结果。通过 IE 浏览器访问嵌入式节点的 IP 地址，通过 HTTP 协议访问嵌入式实时系统的静态页面和动态页面。

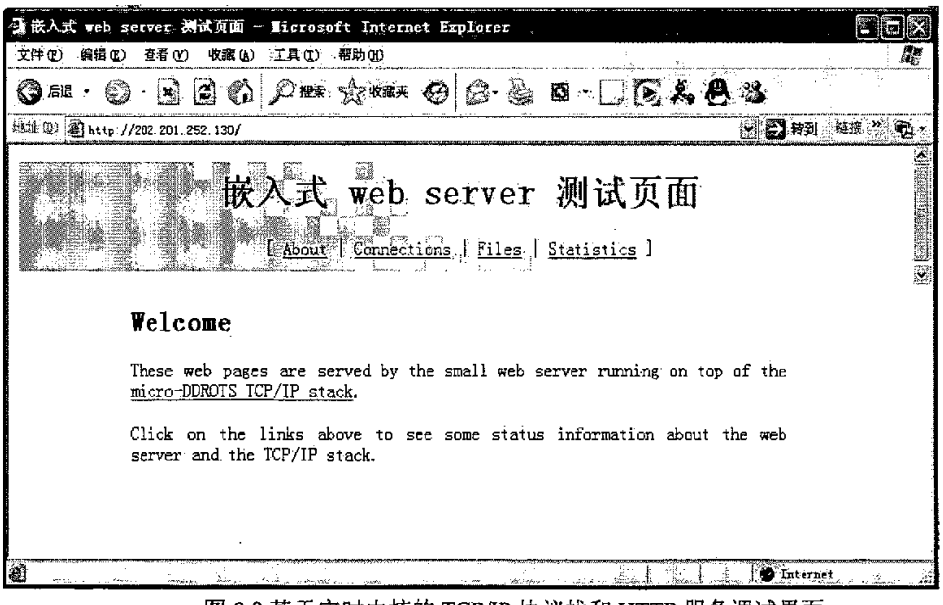


图 8.8 基于实时内核的 TCP/IP 协议栈和 HTTP 服务调试界面

图 8.9 显示了基于 TCP/IP 协议栈的 FTP 服务的调试结果。通过 TELNET 终端登陆到实时系统的 IP 节点。图中显示了登陆到节点后，系统提供了一个简单的 shell command。这些命令提供了对于系统的文件系统的操作界面。提供了对于系统文件系统的简单操作。

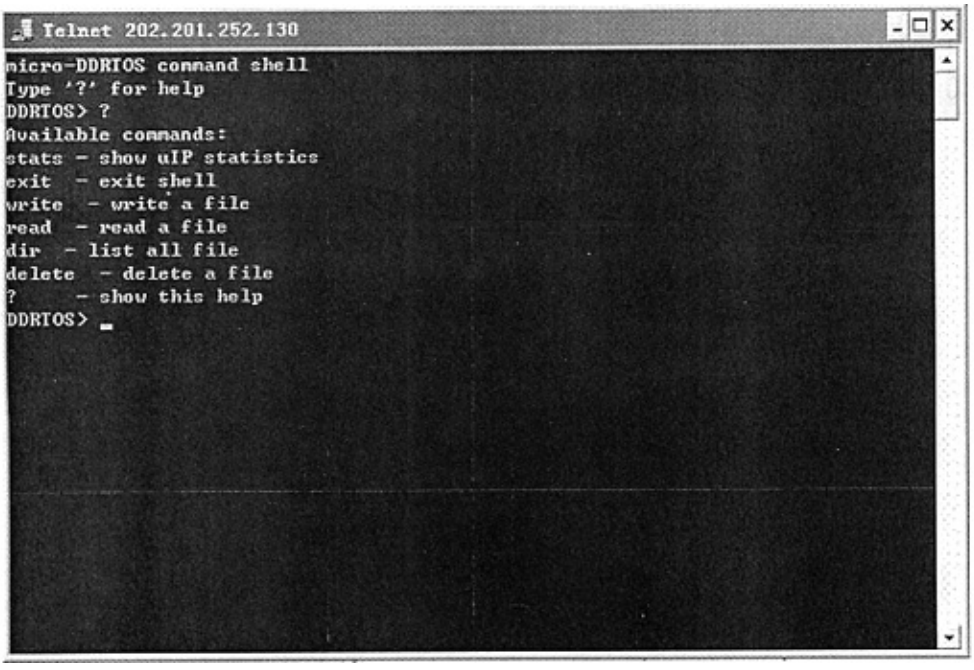


图 8.9 基于实时内核的 TCP/IP 协议栈和 FTP 服务调试界面

图 8.10 显示了通过 telnet 终端，对于系统的文件系统操作的结果。通过 read,write,dir,delete 等操作命令可是实现文件的读、写、显示和删除操作。

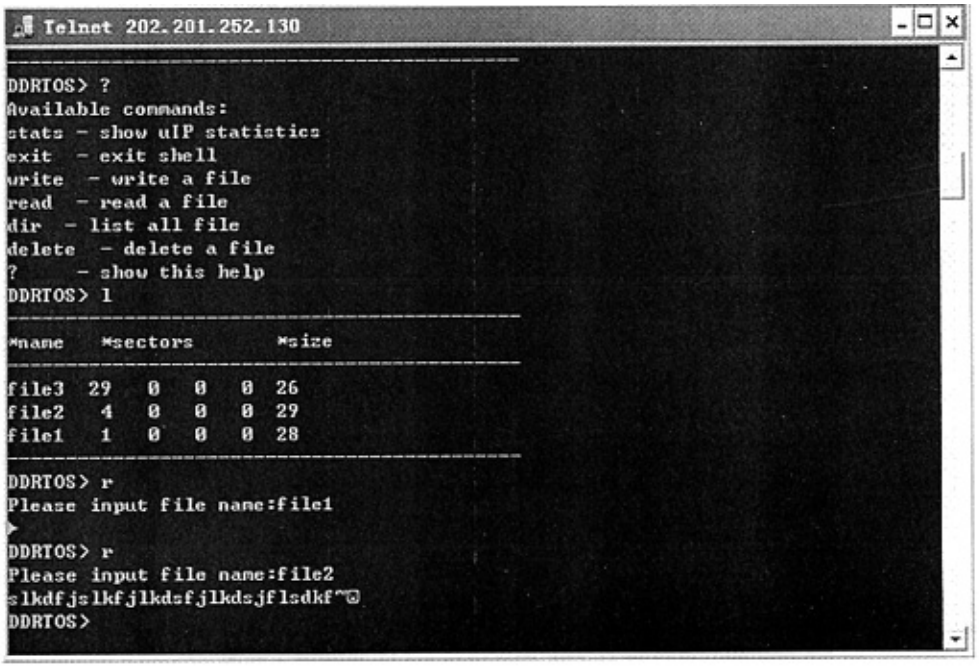


图 8.10 文件系统的调试结果

8.5 总结

本论文提出了一种基于数据流驱动的实时操作系统为内核模型，非常适合于面向于大规模重复数据处理的嵌入式应用环境。通过提供配置参数，可以对于内核的任务优先级和系统扩展应用模块进行配置选择。这里主要提供和完成了以下工作：

1. 基于数据流驱动的实时操作系统微内核模型的提出。
2. 在μC/OS-II 的基础之上对于其进行改造和移植，实现了 micro-DDROTS 内核。
3. 扩展了 micro-DDROTS，在μIP0.9 基础之上实现了 TCP/IP 协议栈，并在此基础之上，实现了 HTTP SERVER 和 TELNET SERVER 服务应用程序。
4. 扩展了 micro-DDROTS，在 MINIGUI 基础之，加入 GUI 通用图形用户界面。
5. 扩展了 micro-DDROTS，加入文件系统。
6. 在 ARM 硬件体系结构之上，利用 ADS 开发环境，在 ADS Armulate 建立仿真环境，添加了 LCD、以太网和 FLASH 等硬件仿真测试环境。
7. 通过建立统计任务和相关测试函数，完成了对于 micro-DDROTS 的整体测试。

本论文在 ARM 开发集成环境和仿真测试平台上完成了 micro-DDROTS 微内核，进一步还有很多的工作去做，对于系统的实时性能和可预测性能需要进一步的研究和测试。现在 micro-DDROTS 只是处于模型阶段，需要进一步的完善，需要将其应用于具体的设备和产品中去，进一步验证系统的可靠性能和稳定性能。

致谢

这里首先感谢我的校内指导老师吴介一教授和校外实习单位的指导老师罗莹老师，感谢他们对于我学位论文的指导和帮助。感谢吴介一老师对于整个论文的研究方法、研究思路和整体框架的指导。感觉罗莹老师在实习期间给予的指导和帮助，使我在熟悉和掌握了移动通讯领域的嵌入式系统的设计方法和测试方法，对于软件工程在这一领域的具体应用有了进一步的认识。

感谢方宁生院长、张家浩老师和实习单位的冯化箱老师对于我这篇学位论文的预评审工作，提出了很多非常有益的帮助和建议。感谢东南大学软件学院的所有老师对于我的培养，使我在软件工程领域有了扎实的基础知识和最新的专业知识。感谢方宁生院长和张飒兵老师对于我们论文开题、论文评审、论文答辩的所有支持和帮助工作，是他们在学习和实习期间给予了我们非常大的帮助。

没有这些老师的帮助和支持，这篇学位论文时不可能完成的，这里再次感谢他们。

参考文献

[1] http://www.dedicated-systems.com/encyc/publications/

[2] 王瑞荣，汪乐宇　面向图形化编程的事件触发并发数据流模型 浙江大学学报（工学版）2003，36（5）535-539

[3] Jean J. Labrosse <<MicroC/OS-II The Real-Time Kernel, Second Edition>> 北京航空航天大学出版社

[4] ARM7TDMI(REV 4) Technical Reference Manual
http:// www. arm. com

[5] 毛德操, 胡希明 <<Linux 内核源代码情景分析>> 浙江大学出版社 2001/9

[6] Christine Hofmeister <<Applied Software Architecture>>
Pearson Education Corporate Sales Division ISBN 0-201-32571-3 2000

[7] Jeffrey L. Whitten <<System analysis and design methods(fifth edition)>> 高等教育出版社 McGraw-Hill 出版社 2003

[8] Shari Lawrence Pfleeger <<Software Engineering Theory and Practice(second edition)>>
Pearson Education 出版社 2003

[9] 陈渝 李明 <<源码开放的嵌入式系统软件分析与实践>>北京航天航空出版社 2005

[10] 马忠梅 马广云 <<ARM 嵌入式处理器结构与应用基础>> 北京航天航空出版社 2005

[11] 黄天成 刘 洋 一种嵌入式 WEB 服务器的设计与实现 电子技术应用 2005/3

[12] 张懿慧 陈泉林 源码公开的 TCP/IP 协议栈在远程监测中的应用 单片机与嵌入式系统应用 2004/12

[13] 潘明泉 张侃谕 嵌入式 Linux 下基于 MiniGUI 的信息终端软件开发 单片机及嵌入式系统应用 2004/12

[14] 胡友水 李汉强 基于ARM的嵌入式TCP/IP协议的实现 电子技术应用 2004/12

[15] 楚育军 刘守印 利用实时内核开发嵌入式多任务程序 单片机与嵌入式系统应用 2004/5

[16] 倪 敏 周治廷贞 杨继堂 uC/OS-II的任务切换机理及中断调度优化单片机及嵌入式系统应用 2004/8

[17] 郑玉全 微型抢占式多任务实时内核设计 单片机及嵌入式系统应用 2004/12

[18] Brinda Ganesh ARCHITECTURAL SUPPORT FOR EMBEDDED OPERATING SYSTEMS
Thesis submitted to the Faculty of the Graduate School of the University of Maryland, College Park in partial fulfillment of the requirements for the degree of Master of Science 2002

[19] 陈强 王建华 μC/OS-II 任务调度机制的改进 单片机与嵌入式系统应用 2004/1

[20] 曾庆伟 韩波 嵌入式实时操作系统核心的设计与实现 计算机工程与应用 2002/09

[21] 郭江鸿 董祥和 嵌入式实时操作系统中的调度问题 太原理工大学学报 2004/01

[22] 沈国保 刘松强 实时系统的性能测试 核电子学与探测技术 2002/09

[23] 雷 红卫 桑 楠 熊光泽 嵌入式实时系统中断管理技术研究 单片机及嵌入式系统应用 2004/12

作者简介

本人姓名孙柯，男，籍贯江苏省丰县，1993-1997 年就读于新疆大学电子信息科学系，所学专业电子学与信息系统，在校期间，学习成绩优异，被评为新疆大学优秀毕业生,1997 年毕业后直接留校任教，先后承担了本科班，专科班的基础课和专业课的教学工作和基础实验室建设和管理的工作。本人工作积极认真，授课效果良好，教学评估分数平均在 4.6 分以上。在新疆大学工作期间，在完成教学任务的同时，本人相继参加和独立完成了如下工程项目：

1、1998 年与自治区无线局合作指导学生完成短波天线的改造工作的毕业设计。主要完成将天线的手动操作改为计算机自动操作。通过在 pc 总线上扩展 8255 并口，实现开关矩阵控制，控制软件采用 c 语言完成。

2、1999 年参与政府办公自动化软件的设计开发。主要应用于政府部门的办公自动化，进行无纸办公。主要采用 Domino lotus notes 实现。

3、2000-2001 年完成智能 IC 卡燃气表的研制和全市银行联网预收费系统的开发。通过国家建设部认证，被自治区鉴定为高新技术产品，现在乌鲁木齐市正常投入应用。

4、2001-2002 工业锅炉自动化控制设计、安装和调试。包括工程师站和操作员站的组态软件设计、现场控制站的数据采集和控制回路的程序设计、工业仪器仪表的调试安装。前后完成了若干热水锅炉（乌市蓝珠热力公司、西山热力公司等）、蒸汽锅炉（库尔勒泰昌棉纺厂自备电厂）、硫化床锅炉（矿务局煤矸石电厂）和远程无人职守供热站（奎屯市热力公司）的控制工程。

5、2002 年焊角压力强度测定仪的主控电路的设计和软件设计。主要功能：塑料钢窗的强度测定，通过自治区计量局的鉴定，投入使用。

2003 年考入东南大学，攻读硕士研究生，专业方向软件工程。

学位课程和专业课程：

学位课程：	自然辩证法及科学社会主义理论和实践	82			
	高级英语	89	软件数学	89	
	软件体系结构	92	软件过程管理	89	
	计算机通信与网络	91	软件项目管理	80	
	专业英语	74			
专业课程：	软件系统分析与设计	90	软件测试	91	
	面向对象的设计与分析	90	模糊数学及应用	95	
	实时操作系统	81	经济管理基础	88	

总完成学分：32

发表论文情况：

论文题目：智能 IC 卡燃气表及燃气预收费管理系统的设计和实现

发表刊物：金卡工程

发表时间：2004 Vol.8 No.5

参加学术活动：

本人报告内容：循环硫化床锅炉模糊控制系统的设计和实现

组织单位：东南大学软件学院校庆报告会

实习情况：

实习内容：在实习单位从事 CDMA 移动通信交换子系统(MSC)的通讯软件开发和系统测试。

实习单位：上海爱立信通讯软件研究开发中心

实习时间：2004/7-2005/7，一年