“和欣”操作系统内核性能测试与软件开发

计算机科学与技术 沈金　　指导老师 陈榕

摘要

本文通过对“和欣”嵌入式操作系统开发工程中的内核性能测试的规划、编程、实测等的总结，学习了“和欣”技术体系结构以及“和欣”操作系统方面的知识，总结了“和欣”嵌入式操作系统的性能测试技术和方法，设计和实现了用于性能测试的内存统计和时间统计等测量工具，编写了性能测试的代码，完成了针对 ARM Release 版本“和欣”2.0 嵌入式操作系统的性能测试的工作，取得了多方面的性能数据，为“和欣”2.0 嵌入式操作系统的进一步开发和完善提供了重要的参考依据，同时证明了“和欣”2.0 操作系统的 ARM 版更加成熟。这为 2.0 系统的进一步产品化打下了基础，并为下一步将 2.0 系统应用于产品项目做好准备。

关键词

操作系统； 嵌入式； 和欣； 性能测试

装

订

线

Abstract

This article deals with kernel performance test items in the Elastos embedded OS development project. Based on these test items, some knowledge on the Elastos technical architecture and Elastos OS is introduced; a summarization of the performance test skills and methods for the Elastos embedded OS is given; some measurement tools used in memory and time statistics etc. of the performance test are designed and implemented; the related performance test codes are written; and the performance test aimed at the Elastos 2.0 embedded OS (ARM Release version) is performed. As a result of the test, extensive performance data have been obtained, which provide a very important basis of reference for further development and perfection of the Elastos 2.0 embedded OS. Moreover, they also prove that Elastos 2.0 embedded OS (ARM version) is more mature than its previous version. All these have laid a good foundation for further productization of the 2.0 system and made it ready to apply the 2.0 system in related product projects.

Keywords

Operating System; Embedded; Elastos; Performance Testing

装

订

线

目录

装 订 线

1 绪论

1.1 操作系统概述

1.1.1 什么是操作系统

操作系统（Operating System）是用户与计算机之间的界面。它是计算机系统中负责支撑应用程序运行环境以及用户操作环境的系统软件，是计算机上运行的最重要的程序。一方面操作系统管理着所有计算机系统资源，另一方面操作系统为用户提供了一个抽象概念上的计算机。在操作系统的帮助下，用户使用计算机时，避免了对计算机系统硬件的直接操作，对计算机系统而言，操作系统是对所有系统资源进行管理的程序的集合；对用户而言，操作系统提供了对系统资源进行有效利用的简单抽象的方法。事实上，操作系统执行的都是诸如从键盘接收输入、向显示器输出、在磁盘上存储文件和目录、控制磁盘、打印机等设备这些最基本的操作。操作系统通常可以按多用户、多处理、多任务、多线程以及实时性等来划分。

操作系统的历史在某种意义上来说也是计算机的历史。操作系统的理论是计算机科学中一个古老而又活跃的分支，而操作系统的设计与实现则是软件工业的基础与核心。

1.1.2 什么是嵌入式操作系统

嵌入式操作系统（Embedded Operating System）是嵌入在某一专用系统设备之内，完成一种或多种特定功能的计算机系统，是软件和硬件的紧密结合体。它具有集成度高、体积小、反应速度快、智能化、稳定及可靠性强等特点。在过去，由于计算机硬件发展水平的限制，针对嵌入式应用的体积小，功耗低的要求和产品成本的考虑，可供嵌入式系统使用的存储容量通常很小，处理器的运行频率也较低。有限的操作系统功能被集成在嵌入式应用软件中，嵌入式应用系统与处理器硬件紧密相关。

在通用计算机系统中，操作系统软件与应用软件之间的界线分明，应用软件通常是独立设计、独立运行的，并以一组可执行程序文件存在于硬盘中。而在嵌入式系统中，操作系统与应用软件是一体化设计的，也就是说应用软件与操作系统是为特定的应用而设计的。这种一体化的设计满足了嵌入式系统的高集成度，高稳定性的要求，也限制了嵌入式应用的进一步发展。

随着硬件系统的高速发展，和网络化的发展趋势，嵌入式软件系统也相应变得越来越复杂。而同时，嵌入式产品的功能密度在增长，产品的升级换代速度也加快。这些变化使得嵌入式软件的开发难度比过去大大提高。过去一体化的设计思想已被逐渐摒弃，取而代之的是在嵌入式操作系统之上开发应用系统。

嵌入式操作系统指的是运行在各种嵌入式处理器之上，可为各种嵌入式应用系统提供基本的操作系统功能，具有统一的系统调用接口的操作系统。

嵌入式操作系统类似于通用的操作系统，一般都提供各种高级语言编译器、丰富的驱动程序支持和应用软件开发包，优秀的商业嵌入式操作系统还提供集成的应用开发环境。不同的是，由于嵌入式微处理器的多样性，为了更多地占领市场，嵌入式操作系统通常要支持多种处理器。因此，一个嵌入式操作系统会有多种处理器版本。由于嵌入式应用的各种功能需求有巨大的差异，而嵌入式的系统资源毕竟有限，加上体积、功耗、效率上的限制，嵌入式操作系统还要求具有功能可裁减的特性。以下是一个好的通用嵌入式操作系统应该具有的特征：

⑴ 支持多任务。
⑵ 任务具有优先级和支持抢占式调度规则。
⑶ 支持多任务间的通信机制（消息，信号量，管道等）。
⑷ 支持虚拟内存管理和存储器优化管理。
⑸ 支持各种通信接口和标准的网络通信协议，如 TCP/IP。

⑹ 支持多种嵌入式微处理器。

⑺ 操作系统功能具有可裁减性。

从 80 年代起，国际上就有一些 IT 组织、公司开始进行嵌入式操作系统的研发。至今为止，已出现了数百种嵌入式操作系统。其中有一些著名的商业嵌入式操作系统，例如 VxWorks，Windows CE，pSOS，QNX，PalmOS 等。也有一些开放源代码的嵌入式操作系统，如μCOS，Embedded Linux，ECOS 等。这些操作系统从实时性能、可裁减性、网络支持能力、图形界面和对不同微处理器的支持等特性考虑，有着不同的应用市场。

在工业界，嵌入式操作系统已经广泛应用于数控机床，机载设备，网络交换机，路由器等专用系统中。在消费电子产品中，嵌入式操作系统也可以广泛应用于 PDA、掌上电脑、手机、信息家电（网络冰箱、机顶盒）等嵌入式应用。

1.1.3 操作系统相关概念

从概念上讲，操作系统并没有十分明确而被广泛认可的定义，同时，随着操作系统本身的发展，它的概念也在不断地发生变化。归纳地讲，可以列举如下几种表述。第一，操作系统是硬件与应用程序的中间界面。即操作系统是直接操控硬件并支撑应用程序运行的软件。从逻辑上来说，也就是应用程序与硬件打交道的中介。这种定义是最常用的，但是不足之处在于，很多嵌入式系统中的应用程序是直接与硬件交互的。第二，操作系统是应用程序的交集。即操作系统是实现所有应用程序都需要的基本功能的软件的集合。第三，操作系统是应用程序的补集。即操作系统是所有实现必要但是不被任何应用程序所提供的功能的软件的集合。这种定义是最能够体现现代操作系统特性的定义。

从结构上讲，人们常把操作系统分成四个部分。第一是驱动程序，它是在底层的、直接控制和监视各类硬件的部分，它们的职责是隐藏硬件的具体细节，并向其他部分提供一个抽象的、通用的接口。第二是内核，它是操作系统的核心部分，通常运行在最高特权级，负责提供基础性、结构性的功能。第三是接口库，它是一系列特殊的程序库，它们的作用在于把系统所提供的基本服务包装成应用程序所能够使用的编程接口（Application Programming Interface，API），是最靠近应用程序的部分。例如，GNU C 运行时库就属于此类，它把各种操作系统的内部编程接口包装成 ANSI C 和 POSIX 编程接口的形式。第四是外围控制，它是指操作系统中除以上三类以外的所有其他部分，通常是用于提供特定高级服务的部件。例如，在微内核结构中，大部分系统服务，以及 UNIX/Linux 中各种守护进程都通常被划归此列。

从内核结构上讲，因为内核是操作系统中处于核心和基础地位的构件，因而，内核结构往往对操作系统的外部特性以及应用领域有着很大程序上的影响。 尽管随着理论和实践的不断演进，操作系统高层特性与内核结构之间的耦合度越来越低，但习惯上，内核结构仍然是操作系统分类的常用标准。内核的结构可以分为单内核（Monolithic Kernel）、微内核（Microkernel）、超微内核（Nanokernel）以及外核（Exokernel）等。

在应用领域之中，以单内核结构为基础的操作系统一直占据着主导地位。在众多常用操作系统之中，除了 QNX 和基于 Mach 的 UNIX 等个别系统外，几乎全部采用单内核结构，例如 Linux，大部分的 Unix，以及 Windows。 微内核和超微内核结构主要用于研究性操作系统，还有一些嵌入式系统使用外核。

1.2 操作系统内核性能测试概述

操作系统的内核性能测试也就是测试操作系统软件处理事务的速度和操作系统的开销，测试的目的一是为了检验操作系统的性能是否符合需求，是否与期望的性能接近。二是为了得到全面的性能数据供人们参考，这些数据会影响到操作系统的后续开发工作，而且对系统内核的稳定也至关重要。有的时候，人们会关心测试的“绝对值”，如数据传输速率是每秒多少比特，执行某个任务花了多少时间，或者是有多少内存开销。而有的时候人们关心测试的“相对值”，如某个软件比另一个软件快多少倍，某种算法比另一种算法要节省多少内存等等。在获取测试的“绝对值”时，我们要充分考虑并记录运行环境对测试的影响。例如网络环境、计算机主频，总线结构和外部设备都可能影响软件的运行速度。

环境因素对于操作系统性能测试的数据结果有很大的影响。我们可以从这些方面考虑：

⑴ 现代的处理器通常采用大规模的多级流水线和指令预取机制，这极大地提高了处理器的工作速度，但同时这对于系统所造成的延迟却是不可估算的。

⑵ 多级缓存的出现使得系统减少了对磁盘的大规模访问，但在缓存不命中时，速度反而降下来了。缓存里查找失败而去磁盘访问，再写缓存。如果这种操作频繁出现，那么系统延迟将比预料的要大得多。

⑶ 从硬件角度来讲，像使用以太网物理协议的网卡，它的反应速度对于实时性要求高的测试肯定是有问题的。带有冲突检测的载波侦听多路存取协议的随机指数算法的响应速度不是事先可以确定的。

我们所做的性能测试将分成两步。首先是基于理论评估的定性分析，我们主要关注操作系统的系统结构，第二步则是定量地对系统进行完整的测试和评估。

1.2.1 理论评估的定性分析

上面已经对操作系统的体系结构进行了大概的描述。操作系统的体系结构决定了底层的线程管理机制、中断管理机制、存储器管理机制以及其他各方面的内容，同时，操作系统也提供了各种非常有效的同步机制、互斥操作等用户抽象的功能。为完成这些功能，操作系统将消耗处理器时间、内存和其他资源，而这些系统开销对于评价操作系统性能的直接表现。

在操作系统具体实现之前，它的各个方面的内容必然经过了非常详细的完整的设计，这些对于性能测试是一份非常重要的参考。而且至少可以确认的是，它所提供的应用程序接口已经固定了。操作系统所提供的应用程序接口极大地丰富了操作系统自身的完整性，这些高层的应用程序接口将使应用程序的大规模开发提供可能。

另外还会提供各种有效的系统工具，用于开发人员在应用程序开发过程中进行测试。包括测试错误的代码和各种逻辑错误等。所谓错误的代码，可以是非法指针访问、内存有泄漏等，而逻辑错误则可能包括无限的死循环、错误的条件判断等等。

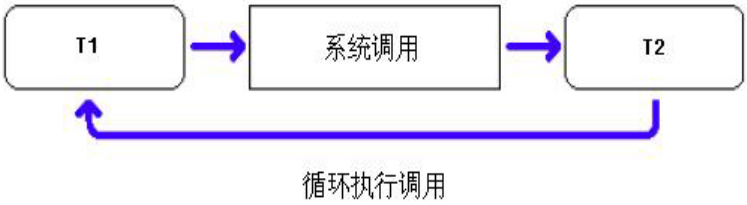
1.2.2 实际验证的定量分析

定量分析就是通过实际的测试来对操作系统的性能进行评估。主要的工作就是获得系统操作的空间和时间的开销，其中可能会使用各种不同的测试手段，它的它的目的很明确，就是去验证操作系统是否适用于实际的应用，或者说是操作系统是否满足实际应用的要求。比如说，对于实时操作系统，所有的系统调用所引起的时间和其他的开销都将是事先可以判断的，也就是说执行时间都是有限的而且它并不会因为系统负载重而减慢。性能测试的任务不仅是测量系统的反应和吞吐量，而且是系统的各种特性和行为。

⑴ 测试方法

为了满足时间开销的测试，一个可用的计时器是非常必需的。大多数操作系统都有软件实现或者说是软件模拟的定时器，而这些定时器也确实可以被用于测量的工具。但操作系统所提供定时器的解决方案因各自不同的实现而有不同的表现，同时，对于精度要求十分高的测试用例来说，它们通常并不能胜任。在进行测试时，操作系统一边要运行测试程序，一边要维护定时器，另外，软件实现的定时器与系统时钟是同步的，这些原因都会使测试的结果产生不必要的偏差。

另外一种做法就是采取硬件提供的定时器，在进行测试代码之前打开定时器，设置定时器初值，在测试代码之后读取计时器（有时需要关闭定时器），具体的数据分析可以在测试后期执行。这样，测试时所有的开销只在于打开和关闭定时器，同时，硬件提供的计时器的精度可以非常高，与系统时钟异步。



图表 1 一般的性能测试方法

上图描述了一般的性能测试方法，T1 时打开计时器，处理中断响应等，然后执行系统调用，在结束时的 T2 时刻，关闭计时器，记录时间间隔，或者就是记录起始时间和终止时间，在测试后期处理数据，计算时间间隔。如果循环执行，测试足够多次对数据进行一系列科学统计和处理，从而可以得到平均值、最大值、最小值、标准差等数据。

⑵ 系统负载

在运行测试程序之前，首先要做的验证测试程序的准确性。操作系统的性能测试要有相当多的考虑。一个是在不同的操作系统上运行测试程序，然后再比较测试结果，另一个是测试结果相当于测试程序或者特定的测试参数而言，它的合理性是可以被很好的解释的。

操作系统的性能在很大程度上取决于系统里运行线程的数目。比如说，一个线程申请使用某一资源，而这一资源当前又不可用，那么这个线程就会在重新调度后陷入等待状态中。至于这个线程什么会被唤醒，则完全取决于等待队列中有多少线程在争用资源，如果多，可能要长时间等待下去，当然如果少，那么它会非常幸运，在短时间等待后就会得到自己所需要的资源。

操作系统的性能也取决于线程切换时上下文需要处理的数据量的多少。通常，在操作系统里，进程用于管理资源，各种对象等，而具体的操作通常由线程来完成。从操作系统的角度来讲，切换的对象通常是线程，因为线程控制块比进程控制块要小，所以线程切换要比进程切换快得多，这可以很好地避免进程切换所带来的延迟。线程控制块里的数据量取决于当前的内存模式，线程运行在不同的内存地址空间里，系统必须要记录下线程与地址映射相关的信息。

操作系统里内存的模式可以通过下面四种保护模式来进行分类：

① 无保护模式：操作系统里线程之间没有内存保护，比如说不需要内存管理单元的支持；

② 系统/用户保护模式：系统地址空间和用户地址空间之间独立开，它们运行在同一个公共虚拟地址空间。这个模式需要内存管理单元的支持；

③ 用户/用户保护模式：在系统/用户保护模式的基础上增加了用户进程之间的内存保护；

④ 私有保护模式：这种模式指用户进程都被分配了各自独立的虚拟地址空间。需要有内存管理单元的支持。

⑶ 性能测试

下面列举的是操作系统内核性能测试所包含的测试项目：

① 线程切换时间。就是操作系统抢占当前线程运行任务，而将当前线程转为等待状态的时间。当前系统里处于竞争状态的线程的数目是测试的主要参数。结合线程的数目和测试的结果，可以显示操作系统在线程切换处理操作效率是否迅速和合理。不同的内存保护模式则可以比较出系统所执行的内存管理所带来的开销。

② 中断响应时间。也就是操作系统抢占当前线程而去执行中断服务程序所需要的时间。测试时不同的内存保护模式可以比较出用于内存管理所带来的开销，同时也可以知道中断处理程序是否处于当前线程的上下文、当前进程的上下文或者是不同进程。

③ 中断分派时间。就是操作系统从中断处理程序上下文回到被中断线程或者是线程等待队列中下一个线程所需要的时间。前者将不需要进行线程调度，测试时将使用不同的内存保护模式。这可以确认中断处理程序运行在哪一个上下文中。后者需要进行线程调度，在退出中断处理程序而进入线程调度时会设置一个同步对象。根据在同步对象上进行等待线程的数目，可以非常容易地知道释放同步对象的时间取决于线程的数目。

④ 线程的创建与释放。就是用来测试线程创建和释放所需要的时间。

⑤ 同步对象。就是用来测试创建和释放同步对象所需要的时间。同步对象将用于其他一些

性能测试项目。

⑥ 互斥对象。就是用来测试创建和释放同步对象所需要的时间。

⑦ 睡眠精度。就是用来测试系统调用 Sleep 的睡眠的精度，Sleep 调用会放弃 CPU，引起线程重新调度。

⑧ 内存开销与泄漏。就是用来测试线程创建、进程创建等情况下的内存开销，同时需要测试在分配内存后不及时释放是否会引起内存的泄漏。

上面这些测试项目将在第四章中详细叙述。这时所列举的测试框架并不是一成不变的，为了更加完整地测试出操作系统的性能，以及针对特定的操作系统某一方面的性能，一些额外的测试会增加。对于操作系统性能测试的完整性和客观性，这是非常必要的。

⑷ 压力测试

其实除了基本的性能测试以外，测试框架还包括了一些压力测试的项目。

① 两个同时发生的中断。系统产生了两个相同的中断，测试这两个中断都得到中断服务的时间延迟。

② 中断嵌套。发生了两个相邻中断级别的中断，首先产生一个低优先级的中断，然后立即又产生了一个高优先级的中断。前后两个中断之间的时间间隔肯定要比完成前一个中断所需要的时间要短。

③ 最大可承受的中断频率。它是指系统可以满足的对中断进行响应而不丢失中断源的最快的中断发生频率。

④ 对象的最大数目。这是用于测试大量的内核对象（比如信号量、线程对象等等）是否会对操作系统的性能有明显的影响。

1.3 课题研究背景以及我的工作

本文所讨论的操作系统性能测试是上海科泰世纪科技有限公司“和欣”操作系统项目的一部分。所做的工作也是在科泰公司完成的。

科泰公司是 2000 年 6 月由几位曾在美国微软公司、网景公司从事系统软件研发的高级技术专家发起成立的高新技术公司。公司与清华大学联建操作系统与中间件技术研究中心、深圳研究生院软件工程中心，与同济大学联建基础软件工程中心，形成了产学研用的完整体系。

公司致力于新型嵌入式网络操作系统的研发与产业化，其产品“和欣”通过自主创新的 SOA（Service-Oriented Architecture）技术，即基于目标代码封装、继承、多态与面向方面（AOP）的软件工厂技术及支持软件工厂的基于元数据运行时软件零件自动透明拼装技术的运行平台，为面向下一代网络应用的嵌入式设备提供信息交换平台、设备接入平台和软件工厂平台，在嵌入式设备上实现了随需应变、操作简易、与国际主流软件技术同步并且兼容的软件运行环境。

“和欣”可广泛应用于信息家电、汽车电子、手持设备、工业控制、国防装备等嵌入式领域，目前已有智能手机、数字电视、工控终端、数控机床、医疗仪器等产品。“和欣”的诞生，表明中国已经拥有自主知识产权的高端操作系统，具备了基础核心软件研发能力，从根本上解决信息技术安全问题，可以自主打造中国软件应用产业链。

公司的核心产品是“和欣”构件技术以及“和欣”嵌入式操作系统。操作系统的性能测试是在操作系统开发过程中非常重要的一个环节。笔者有幸在实习这一段时间内参与了这个项目的工作，并主要负责性能测试工具的设计与实现和性能测试代码的编写工作。在这一历时三个月的项目中，主要完成了以下工作：

⑴ 学习和研究 VxWorks、Linux 等操作系统下的性能测试方法；

⑵ 学习和研究“和欣”（Elastos 2.0）操作系统以及开发环境 Daily Build；

⑶ 制定“和欣”（Elastos 2.0）操作系统内核性能测试计划；

⑷ 实现内存统计和定时器操作等驱动程序，作为性能测量的工具，改进和完善测量工具、测试方法，提供性能测试数据结果的精度；

⑸ 完成各项性能测试代码的编写工作，运行内核性能测试代码，得到内核性能数据，整合进 2.0 的 Daily Build 等。

1.4 本文组织结构

本篇论文的组织结构如下：

第一章　绪论　主要介绍操作系统、嵌入式操作系统、操作系统性能测试的各个方面以及本课题研究背景和具体工作。

第二章　“和欣”操作系统概述　主要介绍了“和欣”技术的总体架构、“和欣”操作系统内核体系结构概述、“和欣”的灵活内核以及“和欣”的开发环境。

第三章　性能测试工具的研究与实现　主要是学习和研究性能测试所需要的测试工具，研究“和欣”环境下构件化设备驱动程序的方法、设计和实现性能测试所需要的性能测试工具并进行改进和完善，确保测试工具的高精度的要求。

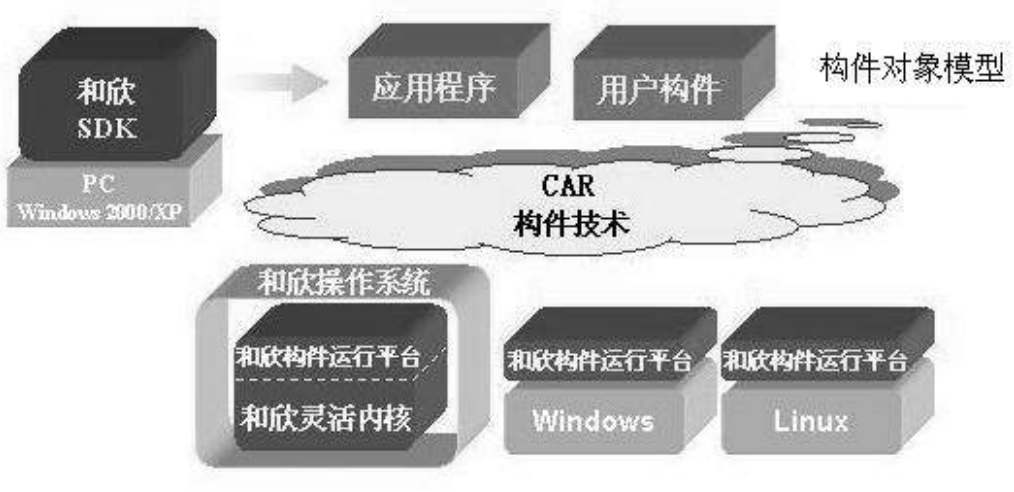
第四章　内核性能测试的实现　对性能测试项目进行分类和编号，设计和运行性能测试编码。

2 “和欣”操作系统概述

2.1 “和欣”技术的总体架构

随着网络技术的发展以及因特网应用的普及，网络服务（Web Service）的概念正在成为新一代因特网应用的重要特点。支持这一概念的基础软件技术，是争夺新一轮软件发展主导权的制高点。实现网络服务的关键技术是面向构件、中间件的编程技术，以及一整套的运行环境、开发环境等平台技术。“和欣”技术体系中，包括了以下部分：

⑴ “和欣”CAR 构件技术：面向构件的编程模型及编程规范；
⑵ “和欣”构件运行平台：CAR 构件的运行环境，支持应用软件跨平台；
⑶ “和欣”操作系统：基于构件技术，支持构件化应用的嵌入式操作系统；
⑷ “和欣”灵活内核：构件化操作系统中"灵活"的内核体系结构；
⑸ “和欣”SDK：面向构件化编程的应用软件集成开发环境。
下面的示意图大致图示了这些技术之间以及与用户程序之间的关系。



图表 2 “和欣”技术体系

2.1.1 “和欣”CAR 构件技术

CAR 构件技术是面向构件编程的编程模型，它规定了一组构件间相互调用的标准，使得二进制构件能够自描述，能够在运行时动态链接。CAR 兼容微软的 COM。但是和微软 COM 相比，CAR 删除了 COM 中过时的约定，禁止用户定义 COM 的非自描述接口；完备了构件及其接口的自描述功能，实现了对 COM 的扩展；对 COM 的用户界面进行了简化包装，易学易用。CAR 技术是在总结面向对象编程、面向构件编程技术的发展历史和经验，为更好地支持面向 Web Service（WEB 服务）的下一代网络应用软件开发而开发的。CAR 的编程思想是“和欣”技术的精髓，它贯穿于整个技术体系的实现中。

为了在资源有限的嵌入式系统中实现面向中间件编程技术，同时又能得到 C/C++的运行效率，CAR 采用了用 C++编程，用“和欣”SDK 提供的工具直接生成运行于“和欣”构件运行平台的二进制代码的机制。用 C++编程实现构件技术，使得更多的程序员能够充分运用自己熟悉的编程语言知识和开发经验，很容易掌握面向构件、中间件编程的技术。在不同操作系统上实现的“和欣”构件运行平台，使得 CAR 构件的二进制代码可以实现跨操作系统平台兼容。

2.1.2 “和欣”构件运行平台

⑴ “和欣”构件运行平台简介

“和欣”构件运行平台提供了一套符合 CAR 规范的系统服务构件及支持构件相关编程的 API 函数，实现并支持系统构件及用户构件相互调用的机制，为 CAR 构件提供了编程运行环境。“和欣”运行平台在不同操作系统上有不同的实现，符合 CAR 编程规范的应用程序通过该平台实现二进制一级跨操作系统平台的兼容。

在“和欣”操作系统中，“和欣”构件运行平台与“和欣”灵活内核共同构成了完整的操作系统。在 Windows 2000、WinCE、Linux 等其他操作系统上，“和欣”构件运行平台屏蔽了底层传统操作系统的具体特征，实现了一个构件化的虚拟操作系统。在“和欣”构件运行平台上开发的应用程序，可以不经修改、不损失太多效率、以相同的二进制代码形式，运行于传统操作系统之上。

⑵ “和欣”构件运行平台的功能

“和欣”操作系统提供的其它构件库也是通过这些系统服务构件及系统 API 实现的。系统提供的这些构件库为应用编程开发提供了方便：图形系统构件库、设备驱动构件库、文件系统构件库、网络系统构件库等。从“和欣”构件运行平台来看，这些构件和应用程序的构件是处于同样的地位。用户可以开发性能更好或者更符合需求的文件系统、网络系统等构件库，替换这些构件库，也可以开发并建立自己的应用程序构件库。

从支持 CAR 构件的运行环境的角度看，“和欣”构件运行平台提供了以下功能：根据二进制构件的自描述信息自动生成构件的运行环境，动态加载构件；提供构件之间的自动通信机制，构件间通信可以跨进程甚至跨网络；构件的运行状态监控，错误报告等；提供可干预构件运行状态的机制，如负载均衡、线程同步、访问顺序控制、安全（容错）性控制、软件使用权的控制；构件的生命周期管理，比如进程延续控制、事务控制等。

总之，构件运行平台为 CAR 构件提供了对程序员完全透明的运行环境，构件可以运行在不同地址空间，不同环境，甚至跨网络。构件运行平台自动为构件运行提供支持，配置必要的网络协议、针对不同的输入输出设备的协议。程序员不必过多地去关心诸如网络协议转换及构件运行控制等与其它构件互操作时的协调问题，只需专注于自己需要解决的程序算法的实现。从而可以从繁杂庞大的应用环境体系中解放出来，大大提高编程的效率。

⑶ “和欣”构件运行平台的技术优势

“和欣”构件运行平台的技术优势包括开发跨操作系统平台的应用软件；对程序员透明的 CAR 构件运行环境，提高编程的效率；直接运行二进制构件代码，实现软件运行的高效率；构件可替换，用户可建立自己的构件库等。

2.1.3 “和欣”操作系统

⑴ “和欣”操作系统简介

“和欣”（Elastos 2.0）操作系统首先是 32 位嵌入式操作系统。具有基于微内核，具有多进程、多线程、抢占式、基于线程的多优先级任务调度等特性。提供 FAT 兼容的文件系统，可以从软盘、硬盘、FLASH ROM 启动，也可以通过网络启动。“和欣”操作系统体积小，速度快，适合网络时代的绝大部分嵌入式信息设备。其次是完全面向构件技术的操作系统，提供的功能模块全部基于 CAR 构件技术，因此是可拆卸的构件，应用系统可以按照需要剪裁组装，或在运行时动态加载必要的构件。

从传统的操作系统体系结构的角度可以看出，“和欣”操作系统是由微内核、构件支持模块、系统服务器组成的。首先，微内核包括了硬件抽象层（对硬件的抽象描述，为该层之上的软件模块提供统一的接口）、内存管理（规范化的内存管理接口，虚拟内存管理）、任务管理（进程管理的基本支持，支持多进程，多线程）和进程间通信（实现进程间通信的机制，是构件技术的基础设施）在内的四大部分。 其次，构件支持模块则提供了对 CAR 构件的支持，实现了构件运行环境。构件支持模块并不是独立于微内核单独存在的，微内核中的进程间通讯部分为其提供了必要的支持功能。 第三，系统服务器是指在微内核体系结构的操作系统中，文件系统、设备驱动、

网络支持等系统服务是由系统服务器提供的。在“和欣”操作系统中，系统服务器都是以动态链接库的形式存在。

⑵ “和欣”操作系统提供的功能

从应用编程的角度看，“和欣”操作系统提供了一套完整的、符合 CAR 规范的系统服务构件及系统 API，为在各种嵌入式设备的硬件平台上运行 CAR 二进制构件提供了统一的编程环境。

“和欣”操作系统还提供了一组动态链接构件库。系统提供的构件库，以及用户开发的应用程序构件都是通过系统接口与内核交互，从这个意义上说，他们处于同样的地位。用户可以开发性能更好或者更符合需求的文件系统、网络系统等构件库，替换这些构件库，也可以开发并建立自己的应用程序构件库。这就是基于构件技术操作系统的优势之一。

此外，为了方便用户编程，在“和欣”SDK 中还提供了与微软 Win32 API 兼容的应用程序编程接口、标准 C 运行库、“和欣”提供的工具类函数等等函数库。

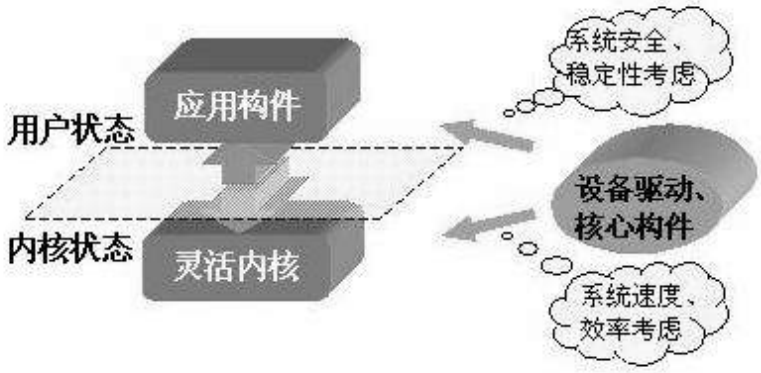
“和欣”操作系统实现并支持系统构件及用户构件相互调用的机制，为 CAR 构件提供了运行环境。关于 CAR 构件的运行环境，其描述与“和欣”构件运行平台是一样的。

2.1.4 “和欣”灵活内核

⑴ “和欣”灵活内核介绍

“和欣”操作系统的实现全面贯穿了 CAR 思想，CAR 构件可以运行于不同地址空间或不同的运行环境。我们可以把操作系统的内核地址区看成是一段特殊的地址空间，用户可以根据运行时的需求，自主选择将操作系统的某些系统服务构件、文件系统、图形系统、设备驱动构件等运行于内核地址空间或用户地址空间。与传统的操作系统的“大内核”、“微内核”体系结构相比，“和欣”操作系统内核里提供的系统服务，完全可以由用户依据系统自身的需求动态决定。因此我们称“和欣”操作系统内核为"灵活内核"（Agile Kernel）。

“和欣”灵活内核的体系结构，利用构件和中间件技术解决了长期以来困扰操作系统体系结构设计者的大内核和微内核在性能、效率与稳定性、安全性之间不能两全其美的矛盾。下图就是“和欣”灵活内核的示意图：



图表 3 “和欣”灵活内核

⑵ “和欣”灵活内核技术的实现

构件技术已经广泛应用于应用软件的开发过程中，在系统软件设计中运用构件技术已经成为研究的热点，特别是分布式系统的发展，为构件技术提供了广阔的施展空间。

构件技术可以做到构件运行环境对用户和构件编写者透明，如构件可以不加修改运行于不同地址空间，“和欣”操作系统正是利用构件技术的优势在系统层面实现了“三层结构”(Client/Middleware/Server)计算模型，完全把构件技术的使用融入到系统内核功能中，“和欣”操作系统的灵活内核技术完全基于这个模型。

“和欣”操作系统把部分系统功能用构件技术封装成独立的二进制模块（如文件系统和图形系统），用户可以根据需要动态地配置这些模块运行状态，也就是说可以根据用户的需要，将一

些来源值得信赖或对运行效率要求高的构件模块配置于内核态，而另一些不太稳定的模块置于用户态运行，在一个系统中同时满足稳定性、安全性与实时性的特殊要求。当然用户也可以开发自己的功能构件，并动态决定其运行状态，而且这种开发工作和开发一个普通的 CAR 构件没有什么分别。

⑶ “ 和欣 ” 灵活内核技术的优点

“ 和欣 ” 的灵活内核是构件技术和系统软件充分结合的产物，它利用了构件技术这一先进的软件组织技术，使得操作系统的体系结构发生了根本性的变化。和通常研究的可扩展操作系统比较，它在许多系统性能方面都得到了极大的改善。第一，增强系统的灵活性。 在 “ 和欣 ” 的灵活内核系统中，用户可以动态配置扩展构件的运行状态，让经过测试的可靠的扩展构件运行于内核态，从而获得较高的效率；把不够可靠的扩展构件置于用户空间。灵活内核不同于前面讨论的可扩展系统研究中的两种方法，其灵活性显然得到了很大提高。第二，提高系统扩展性能。采用灵活内核技术不但使得系统功能可以扩展，而且扩展构件的接口设计也具有相当大的灵活性，只要接口形式符合一定的标准即可，这里主要是指接口方法的参数应能够被系统 “ 理解 ”，从而为其生成正确的中间件。一般研究中的可扩展系统基本上要求扩展模块具有特定的接口形式。第三，更容易实现跨平台。构件技术的出现本身就是为了实现异种平台（包括软件平台和硬件平台）之间实现软件之间的互操作问题，“ 和欣 ” 系统在系统级支持构件技术，为实现跨平台操作奠定了良好的基础。第四。提高开发工作效率。可扩展系统中扩展模块的开发也是非常重要的问题，大部分可扩展系统为扩展模块设立单独的开发平台，这样开发系统扩展模块就需要学习新的开发平台。在 “ 和欣 ” 系统中，开发一个所谓的扩展构件和开发普通构件没有区别，这意味着在 “ 和欣 ” 系统上只需学习一套开发平台，大大高了开发效率，同时开发出了的构件维护工作也会简单的多。

2.1.5 “ 和欣 ” 图形系统

在 “ 和欣 ” 系统上的图形系统，是基于多窗口界面设计的图形系统，是面向控件编程的图形系统。“ 和欣 ”图形系统采用模块化 CAR 构件技术，和用来处理输入输出事件的独特的事件机制，把应用程序的编程工作量减到最低。整个图形系统是由各个控件模块搭建而成，这便于构件的独立升级与跨平台应用。

在 “ 和欣 ” 图形系统中，采用时间触发的机制来执行某个操作，系统从消息队列中获取消息，解码后传递给所属窗口的相应控件，触发相应动作。此外、用户自定义事件机制，在图形系统中得到良好的应用。

“ 和欣 ” 图形系统上编程环境是面向构件的编程环境，每个控件有其各自的事件、方法与属性，用户可以通过其接口智能指针或类智能指针访问其属性，或调用其方法；用户也可以自定义事件。“ 和欣 ” 图形系统中提供的常见控件的接口方法，类似于.net 上的控件的 API，便于理解，易于掌握。

2.1.6 工程管理系统

工程管理系统是 “ 和欣 ” 操作系统这种大型系统软件开发的辅助系统，是 “ 和欣 ” 项目开发的质量保障体系。工程管理系统是一个软件工程管理体系及一系列的配套管理工具构成的。它包括任务管理、源程序管理、缺陷管理、测试程序数据库、自动测试与报告、文档管理及自动化文档生成等几个部分。

2.2 “ 和欣 ” 操作系统内核体系结构概述

“ 和欣 ” 操作系统基于灵活内核体系结构，与微内核体系结构类似，“ 和欣 ” 内核仅提供最小化的功能集，而将其余大部分功能交给核外构件完成。下面是 “ 和欣 ” 内核内部的体系结构及功能实现。

2.2.1 体系结构

“ 和欣 ” 内核作为单独的进程运行，拥有独立的地址空间，内核中的大部分模块以内核服务线程的方式运行。大多数系统调用将挂起正在执行系统调用的当前线程（这些线程属于核外的进

程，核外进程可能运行在用户态或内核态），并且唤醒内核中相应的服务线程，执行完毕后再恢复当前线程的执行。这种设计不仅大大减小了核内各模块的耦合关系，并且可将不可抢占时间降至最低，提高了中断响应时间和并行执行的程度。

内存管理模块管理内核及核外程序的内存分配和释放，以及地址空间的映射。任务调度模块以线程为单位实现任务调度。模块管理模块负责应用程序和构件模块的装载和卸载等操作，以及相关内存的管理。设备管理模块管理所有设备驱动构件，是构件化驱动模型的重要组成部分。进程间通讯模块是 CAR 的基础设施，实现了从核外进程到内核，以及核外进程之间的基本通讯机制。“ 和欣 ”系统调用全部基于 CAR，即通过进程间通讯模块实现。“ 和欣 ”构件运行平台是内核提供给核外程序的一组接口。绝大多数接口通过系统调用实现，因此这一模块在 “ 和欣 ” 内核中基本是逻辑上的划分，它的绝大部分功能由内核其它模块完成。

2.2.2 内存管理

“ 和欣 ” 内存管理涉及两个概念：地址空间和堆的概念。地址空间即一段或多段可被程序访问的内存地址，是内核管理的首要内存资源。地址空间对象负责管理地址空间的分配，以及地址空间之间的映射关系。内核中存在三个地址空间对象，分别管理物理内存地址空间、内核虚拟地址空间与共享虚拟地址空间。每个用户进程拥有一个独立的地址空间对象，负责管理本进程的虚拟地址空间。

“ 和欣 ” 内核以页为单位管理地址空间。每个地址空间对象维护一个树状数据结构，以记录地址空间的分配和映射情况。树状数据结构的每个节点对应一段地址，记录着这段地址的起始和结束位置，以及映射的目标地址空间对象指针和映射起始位置。

对地址空间的管理通过节点的增加、删除、合并、映射等操作完成。从虚拟地址空间到物理内存地址空间映射的改变会导致硬件页表的修改。“ 和欣 ” 内核专门有一个模块负责管理页表。这一模块记录着所有硬件页表的正、反向映射关系。

对其它操作系统而言，堆的概念仅适用于应用程序；而 “ 和欣 ” 操作系统将堆的概念延伸到内核。同应用程序一样，“ 和欣 ” 内核采用堆算法管理内存的分配和释放。对于地址空间和堆的关系，可以理解为：当内核需要一块内存时，程序首先从相应的地址空间上分配以页为单位的内存，然后将这段内存交付给堆算法，由后者对它进行细粒度的管理，以减小由于分配单位过大（1 页）造成的内部碎片。

堆算法依赖于地址空间对象分配的内存，而地址空间节点又必须从内核堆上分配。为解决这种递归的依赖关系，除了常规的内核堆，在内核中还设有一个固定尺寸的常驻堆，该常驻堆直接操作物理内存而不依赖任何地址空间对象，物理内存地址空间对象在该堆上分配节点。

2.2.3 任务调度

“ 和欣 ” 内核使用带优先级的轮转（round robin）与先进先出（FIFO）混合调度算法，以线程为单位进行调度。调度算法支持 16 级线程和进程优先级。对于一般优先级的线程，“ 和欣 ” 内核使用轮转算法，即分配给每个就绪线程一定长度的时间片（通常为 10 毫秒），当前线程的时间片用尽后，内核将选择运行另一个优先级最高的线程；对于优先级为 0（即实时优先级）的线程，内核使用先进先出算法，即当前线程持续运行，直到阻塞或主动放弃 CPU。在默认情况下，每个线程继承其父进程的优先级。调度算法不区分内核进程与用户进程。

“ 和欣 ” 内核通过三个链表管理线程的运行，分别是：就绪队列、阻塞队列和挂起队列。就绪队列中的线程按优先级顺序排队，调度算法总选择队首线程作为当前线程，刚刚被抢占的线程则被放在队尾。

2.2.4 进程间通讯

进程间通讯机制比较复杂。进程间通讯模块实现了跨内存地址空间的 CAR 方法调用。每个接口有一个接口描述表，该表记录着所有接口方法的详细信息，包括方法的参数类型和长度等。内核将通过某些途径得到这一描述表，并根据该表对接口调用进行 marshal/unmarshal。

2.2.5 模块管理

模块以文件的形式保存在存储介质上，即通常所说的可执行文件。模块可以是动态链接库或者 EXE 文件。CAR 构件必须以动态链接库的形式编译链接。为执行一个程序，内核必须将相应的模块从存储介质装载到内存，连接函数导出表、导入表，重定位全局指针，初始化数据段、堆、栈等，以及模块运行必须的数据结构，对于动态链接库，需要装载到共享地址空间，对于 EXE 文件，需要初始化进程自身的地址空间。模块的生命周期也需要有严格的控制。这些工作都由模块管理模块完成。

“和欣”内核使用延迟装载的方式加载模块，即文件以页为单位划分，每一页延迟到内核读取该页时才被装载到内存。这种方式有效降低了模块初始化的时间。

2.2.6 代码映射机制

内核的全部代码段及部分数据段被映射到用户空间，使得可以在用户态执行内核代码，并以只读方式访问内核的部分数据。由于不需要在用户态和内核态之间切换，这种执行方式加快了关键性代码的运行，同时保证了足够的安全性。例如对于互斥锁、信号量等同步对象，它们的大部分代码都通过代码映射运行在用户态。由于常用的代码可以通过映射机制集中到内核，这种方法虽然在表面上增加了内核的尺寸，但实际上有效避免了由于应用程序静态链接运行库而导致整个系统尺寸的增加。

装

2.3 “和欣”的开发环境

2.3.1 “和欣”SDK 介绍

“和欣”SDK 在 Windows 2000/XP 上为软件开发技术人员提供了一个面向构件化编程的应用软件集成开发环境。“和欣”SDK 帮助技术人员充分运用“和欣”技术体系所包括的 CAR 构件技术、“和欣”构件运行平台技术，开发“和欣”构件运行平台上的应用软件。

CAR 构件技术的思想，充分体现在利用“和欣”SDK 所提供的开发工具，利用“和欣”构件运行平台提供的接口、构件库等进行应用开发的过程中。利用“和欣”SDK，技术人员在不用了解很多关于构件技术细节的情况下，就可以开发出面向构件的应用软件。

订

2.3.2 “和欣”2.0 DailyBuild

线

DailyBuild 系统为一套自动的测试系统。之所有说是自动的测试系统，是因为，此系统可以在没有人员干预的情况下，能够自动完成“和欣”操作系统的各个功能模块的测试。产生测试报告，定位运行出错的测试程序，帮助开发人员查找 CheckIn 代码中的错误。

DailyBuild 的含义就是每天编译一次。测试组负责 DailyBuild 的工作，将测试结果反馈给开发人员。DailyBuild 每天都运行，这样的好处，就可以通过当天的测试报告与前一天的测试报告进行对比来确定当天代码 CheckIn 的错误，以便于及时进行更正。使开发工作能步步为营，保证开发质量。所以 DailyBuild 是“和欣”操作系统的稳定开发和管理提供基本的保证。

3 性能测试工具的设计与实现

进行性能测试的是“和欣”（Elastos 2.0）操作系统的 ARM Release 版本。这里首先将介绍 ARM 的硬件配置，“和欣”操作系统开发环境的搭建过程，开发树的编译，通过网络和串口向开发机传输操作系统镜像文件等等，这些都是操作系统的基础，操作系统的性能测试也是建立在这些工作之上的。性能测试需要有完整的测试工具支持，这里主要包括用于空间测量的内存统计工具和用于时间测量的定时计时工具，下面的论述里将会有测试工具的研究过程和详细的实现。这些工具都将以 DDK 驱动的形式提供给用户程序调用，由于涉及到了驱动程序的开发，所以专门有一部分用于介绍 “和欣”操作系统里面构件化设备驱动程序相关的内容。

3.1 “和欣”操作系统开发环境的搭建

3.1.1 开发环境

“和欣”2.0 是“和欣”操作系统的第二个正式发布版本。“和欣”的开发环境一般是在宿主机上开发操作系统和应用程序，在目标机上运行操作系统和应用程序。对于前者，一般是一台普通的 PC 机，只要具备“和欣”安装条件而且已经有“和欣”软件开发平台，就可以用作开发主机。而对于后者，则是 ARM 板，用作运行“和欣”2.0 的目标机。具体地讲，宿主机需要启动 TFTPD 服务，宿主机和开发机都需要有串口和网络等硬件支持。

性能测试所用的开发机是 SHARP 公司的 LH7A400ARM 板。它属于 LH7A4XX 系列，该系列是用于多媒体和移动应用的 32 位的 SoC（System-on-Chip）解决方案。LH7A400 基于 ARM922T 的 SoC，用于支持因特网和多媒体的整合应用的完整的解决方案。它包括了 32 位高性能的 ARM9TDMI RISC 的芯片，16KB 的缓存，存储器管理单元（MMU），彩色液晶触摸屏和静态存储器。

3.1.2 操作系统的下载

“和欣”操作系统在开发过程中使用了 CVS（Concurrent Versions System）作为协作开发和版本控制的工具。CVS 是一个将一组文件放在层次目录树中以保持同步的系统。开发人员可以从 CVS 服务器上更新他们的本地层次树副本，并将修改的结果或新文件发回；或者删除旧文件。CVS 基于客户端/服务器的行为使得其可容纳多用户，构成网络也很方便。很多知名的软件，如 Gnome、KDE、The GIMP、Wine 等都是在 CVS 的管理之下进行开发的。

服务器端的 CVS 配置、管理都是系统管理员的工作。作为开发人员，需要一个 CVS 客户端，现在一般使用的是 WinCVS，它是一个简单易用的 CVS 前端工具的图形界面。WinCVS 的安装的同时一定要安装 Python、TCL 等模块。另外，也可以给 WinCVS 配置一个外部的比较器，比较器是代码编辑的一把利器，因为对比新老版本的差异是开发人员在向服务器提交新版本代码时核对和检查的重要步骤，而一个高效的比较器则是非常必要的。

从 CVS 的仓库中将 “和欣”（Elastos 2.0）操作系统对应的模块下载到本地，从大的方面可以分为 DDK 和 SDK 两大模块，下载到本地后，它们也在两个独立的根目录下，前者文件夹下面包括了编译器在内的各种开发工具、操作系统内核代码、BSP 代码、驱动程序代码等各种核心模块，而后者包括了文件系统、图形系统、网络、运行时库等各种代码。

由于“和欣”（Elastos 2.0）操作系统文件比较多，WinCVS 更新时需要较长时间从服务器的 CVS 仓库里下载。

3.1.3 开发树的编译、打包

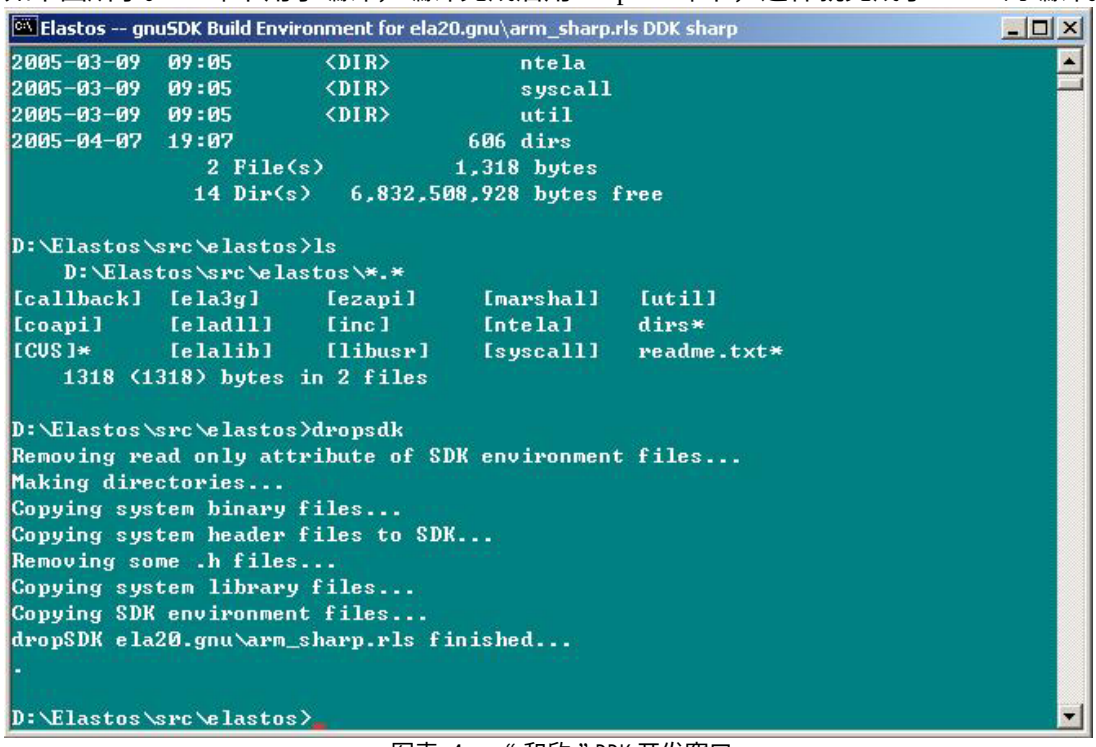
从服务器上下载的“和欣”（Elastos 2.0）操作系统的源代码包括了多个平台的版本。使用的是同一份代码，与 X86、ARM、MIPS 等体系结构相关的代码则分布在各自的子目录下。

在启动开发环境时，会进行环境变量、路径等配置的初始设定。这些配置对于开发是必须的，系统根据 makefile 编译，而 makefile 里的宏的打开或者关闭往往是由环境变量控制的，用于为编

译指定路径的 dirs 和 sources 文件里往往会根据环境变量进行设置。开发人员使用同一份源代码，但编译环境却因体系结构而有差异，开发 X86 下的代码时，编译的环境是 X86 的，而不会是 ARM 或者是 MIPS 的。

从服务器上下载的是源代码，这些源代码需要进行编译。操作系统和应用程序的编译在“和欣”操作系统下非常简单，根据对应的 dirs 和 sources，环境就可以调用编译器进行编译和链接，这里的 dirs 和 sources 类似于 makefile 文件，但要简单和直接。emake 命令用于编译，看名字就知道，它的功能和 make、nmake 是一样的。系统还为 emake 提供了一个别名，叫 z，它和 emake 的效果是一样的。emake 加上命令参数/all 就可以完整地编译整个开发环境，当然，这也可通过别名 za 来完成。

“和欣”（Elastos 2.0）操作系统的编译需要以下几步。首先是编译 DDK。打开 DDK 开发窗口，如下图所示。Za 命令用于编译，编译完成后用 dropsdk 命令，这样就完成了 DDK 的编译。



图表 4 “和欣”DDK 开发窗口

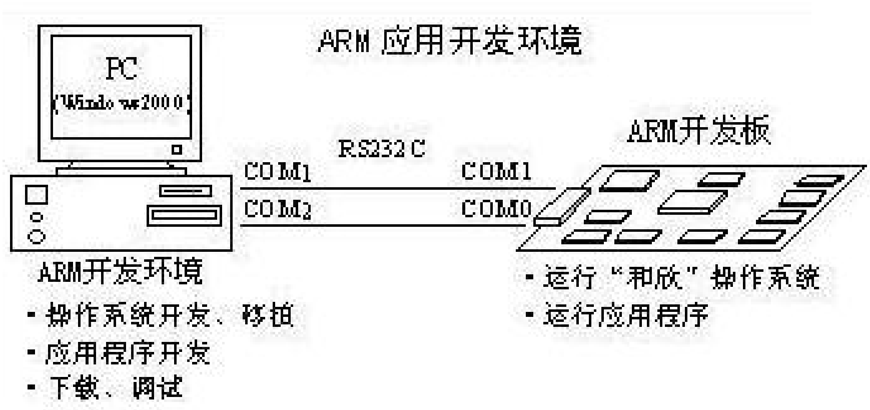
其次，打开 SDK 开发窗口，用 za 命令编译 SDK。系统里有 mkpg.cfg 文件，用于打包配置，开发人员可以发动配置文件，把需要的文件添加进去。mkpkg.exe 命令将根据这个配置文件生成操作系统镜像文件 elastos.img。

生成镜像文件之前，系统已经编译完成了。在这个过程当中，系统可能会提示需要从 CVS 服务器上下载本地没有的模块，比如说 Atlas 编译时需要 elafont 而本地没有，编译的过程就会中止并给出提示。编译系统将花费较长的时间，需要耐心等待。编译的方法是非常简单的，因为系统已经为开发人员提供了最简单易用的开发工具。

系统编译后，开发环境目录下会出现一个 obj 目录，里面放置编译过程中产生的中间文件和二进制文件，统一管理。传统上的开发环境都是把 obj 目录放下开发目录下面，一个项目有一个 obj 目录，这样不容易管理。

3.1.4 串口传输和网络传输

宿主机上开发完之后，要用 mkpkg.exe 打包 elastos.img 镜像文件，打包时可以选择需要的文件，当然操作系统运行所需要的核心模块、配置文件是必须要添加的。镜像文件需要拿到开发机上才能运行。宿主机与开发机之间的联系可以从下面的示意图中看出来。



图表 5　“和欣”宿主机与开发机

向传输镜像文件一般可以通过串口传输和网络传输。

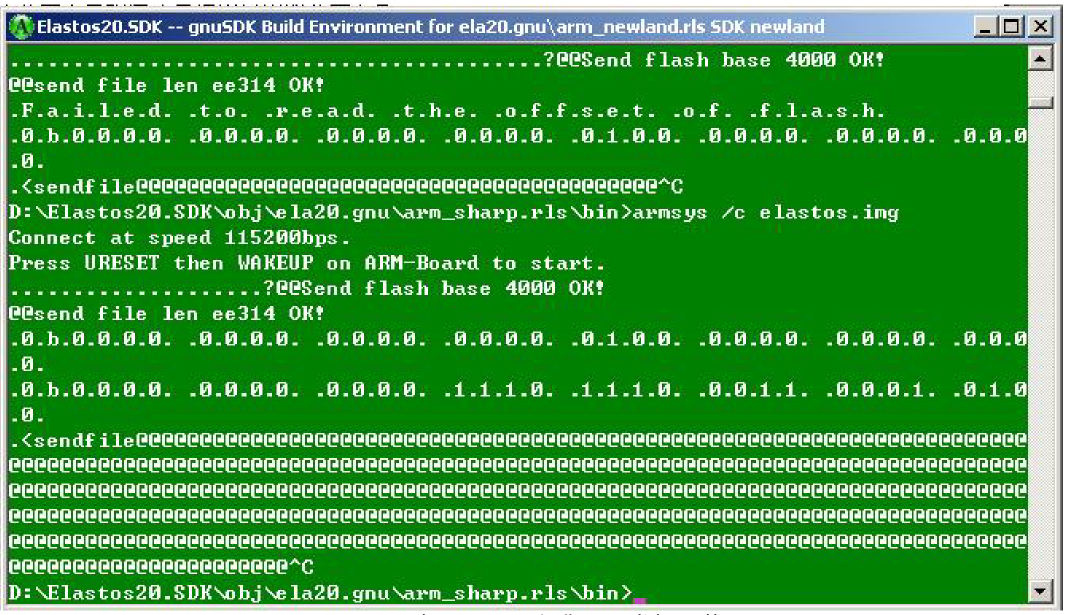
⑴ 串口传输

也就是通过串口将打包后的镜像文件传输到开发机上，这里当然是针对 ARM 开发板而言的。“和欣”环境下提供了用于通过串口传输的下载工具 armsys.exe，它的命令格式是这样的：

armsys　[/? | /Hn | /L | /C | /W | /S] [loadfile] userfile

其中各个参数的详细含义可以参考“和欣”相关资料。通常，可以输入这样的命令：

Armsys /c Elastos.img，其中前一个参数表示传输结后保持连接，而后面的则是镜像文件的文件名。对于像 newland 这样的开发板，由于没有网络条件，所以只能通过串口来工作，一般来说，串口的下载速度相对要慢一些。下面开发机下载时宿主机上控制台的截图，其中有进度的显示：



图表 6　“和欣”开发机下载

⑵ 网络传输

也就是通过网络将打包后的镜像文件传输到开发机上。首先宿主机上要运行 TFTP 服务，由于 Windows 下默认是关闭的,所以必须手动打开。当在宿主机上通过串口控制台工具程序 comcons 时，开发机在重启后，系统就进入了传输，在执行之前会进行宿主机 IP、开发机 IP、镜像文件目录路径、是否存储到 Flash 等设置，一般而言，直接按回车选择默认的配置就可转以了。完成这些工作之后就开始传输，传输时宿主机上控制台窗口显示的内容与上图类似。网络传输的速度要

装

订

线

比串口传输快得多。Sharp 版支持上述两种传输方式，但出于开发效率的考虑，通常采用后者。

3.2 测试工具和测试方法

3.2.1 测试需要的工具

操作系统的性能测试主要用于测试操作系统在特定的情况下的系统消耗、反应时间等各项指标。为了准确地获取操作系统性能指标，我们需要系统为测试提供一些基本的工具。比如内存的统计，也就是了解当前系统里已经占用的内存、可用的内存等非常重要的参数。再比如说系统应该提供用于计时的方法，从而可以测试出一些系统调用和系统操作的时间开销。因为系统操作等操作系统内部的操作往往很短，这些计时工具的要求也就相对比较高。当然对于一些特定的测试，可能还需要有其他测试的工具。

如果系统已经提供了这些工具，那么对于测试工具是非常好的消息。但如果系统里没有，或者说系统时没有直接给出，那么这时候只能自己实现测试工具。事实上，这次性能测试中这些工具都是重新实现的。测试的工具都在 DDK 里以驱动的形式实现，提供给用户程序使用。所以稍后将先介绍“和欣”构件化设备驱动程序，然后介绍所用到的测试程序的设计和实现。

3.2.2 测试方法

装

有了测试工程的支持，问题已经解决了好多，至少可以进入测试这一个环节而不再需要为测量的手段担忧了。对于内存的开销，通常可以选择需要测试的代码的前后作为测试点，代码运行到前一个测试点时进行内存统计，代码运行到后一个测试点时再做一个内存统计。将数据做一下对比，那么就非常简单报。

对于时间上的开销，测试的方法也是类似的。选取两个测试点，代码运行到前一个测试点时打开计时器，代码运行到后一个测试点时关闭计时器，这样就可以计算出时间间隔，而这通常就

订

是测试所需要的。或者也可以通常全部的计数器，在代码运行到前一个测试点时读取计数器，当代码运行到后一个测试点时再读取计数器，这样将两个数据相减，也可以非常容易地得到时间间隔。当然，对于测试而言，一次或者几次测试往往是不够的，一般都要循环地进行多次运行，得到足够多的数据，然后通过这些数据进行科学分析，比如说得出平均值、最大值、最小值、标准差等各项数据。

从统计上来讲，多次测试可以获取稳定的数据结果，提供测试数据的精度，但其他方面的误

线

差也要尽量减少。首先要考虑的就是测试点上进行统计的开销，这些开销有时很少，有时则相对来说比较大。但无论怎么说，都会影响到测试数据结果的精度。对于这些额外的开销，可以通过测试工具的空转来计算，最后再减去这部分开销。计时工具在实现时就考虑到了这一点。

3.3 构件化设备驱动程序

测试工具的实现是在 DDK 里以（伪）驱动程序的形式提供给用户的，所以这里先介绍一下“和欣”构件化设备驱动程序的概念、设计与和工作过程。

与传统 UNIX、Windows 操作系统不同的是，“和欣”操作系统是以 CAR 构件技术为基础搭建起来的，是真正的构件化操作系统，这也决定了“和欣”操作系统下的设备驱动模型的设计必然不同于传统的 Unix 或 Windows 系列操作系统。

正因为“和欣”操作系统的设备驱动模型是以 CAR 技术为基础技术设计的，一方面，编写设备驱动程序需要按照编写 CAR 构件的步骤来完成。另一方面，由于编写的是一个驱动程序构件，必然有不同于编写普通的 CAR 构件的地方，这种差异性主要表现在驱动程序与操作系统的交互接口上。

3.3.1 基本的设计思想

“和欣”操作系统通过内核对象设备管理器（DeviceManager）来管理系统中的所有设备与驱动对象。“和欣”设备驱动模型正是以 DeviceManager 为核心，并基于以下几个设计原则实现的。第一，由 DeviceManager 收集、管理系统中的所有设备信息，为每一个设备建立一个设备节

点，并使用设备标识与设备号来唯一标识这些设备。第二，把驱动构件对象看作设备的一个属性，当且仅当设备节点的驱动对象被创建出来后，这个设备节点才被称为激活了的设备或活动设备（Actived Device）。第三，由 DeviceManager 负责创建驱动构件对象，并使用设备标识、设备号来匹配设备节点与驱动构件对象。第四，驱动构件所实现的接口分为两大类：一是系统接口，是系统定义的标准接口，由操作系统调用；二是上层用户接口，由驱动构件的开发者定义，并由驱动构件的客户程序调用。第五，DeviceManager 管理驱动构件对象的创建及消亡过程，但不参与驱动构件对象的使用过程。

3.3.2 驱动程序接口的实现与配置

“和欣”（Elastos 2.0）操作系统的内核以一个设备名（宽字符串型）加上一个设备号（无符号整型）来唯一标识一个特定的设备实例。这里所说的“设备”，可以是实际存在的硬件设备实体，也可以是虚拟的（伪）设备。通常情况下，设备名标识设备的类型，而设备号标识此类设备的一个实例。例如：系统中存在有两个串口设备，设备名可以为“com”，而设备号为 1 和 2。那么设备名加设备号“1”就唯一表示了第一个串口设备；而设备名“com”加设备号“2”就唯一表示了第二个串口设备。

“和欣”（Elastos 2.0）操作系统设备驱动程序以静态方式被链接到系统内核映像文件成为内核的一部分。与内核的其他部分一样，设备驱动程序代码在处理器特权态下执行。除了这些特征之外，Elastos 2.0 设备驱动程序本质上就是一类存在于内核之中的特殊 CAR 的构件。它们都必须通过暴露出一个统一的 CAR 接口 IDriver 来提供服务。

所有的内核驱动程序都提供了一致的 IDriver 接口，但是却以各自特定于设备的方式来解释 IDriver 接口的精确语义。通常，IDriver 接口的 Read()方法用于从设备中读入数据；Write()方法用于向设备输出数据；而 Control()方法则用于向设备发送命令、获取状态信息等等。

对系统中已经存在的设备，它们的信息可以静态地放入一张全局设备配置表中。内核为这张全局设备配置表维护一个以 struct DeviceConfig 为元素类型的数组 g_deviceConfigs 加上一个 uint_t 类型的全局变量 g_uNumberOfDeviceConfigs 来指明该数组的实际长度。

通常情况下，g_deviceConfigs 和 g_uNumberOfDeviceConfigs 的实现位于

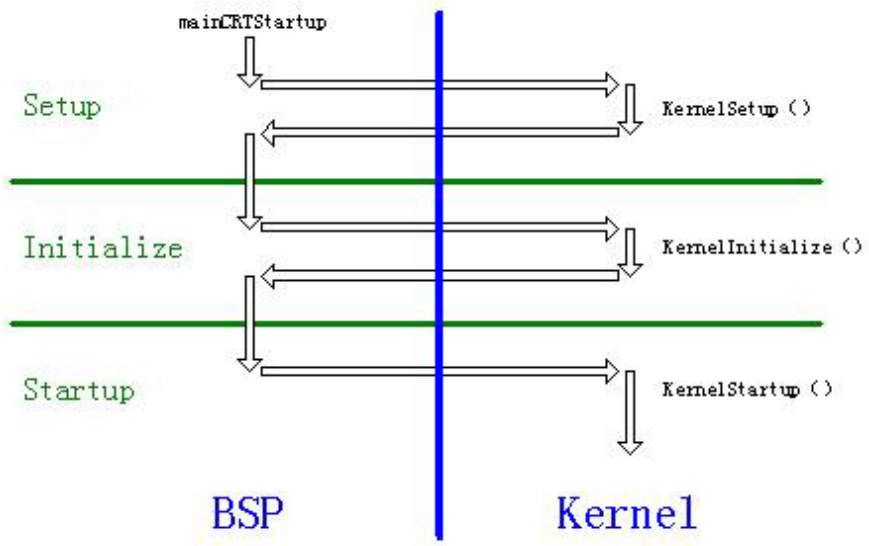
$Elastos/src/kernel/drivers/%arch%/%board%/config/config.cpp 中。其中 arch 是体系平台，包括 X86、ARM 和 MIPS，而 board 更细的划分，就 ARM 而言，它包括了 Newland 和 Sharp 两种。可以通过标准内核功能接口 RegisterDevice()函数动态地向系统注册设备信息。注销设备信息则是通过标准内核功能接口 UnregisterDevice()。

3.4 内存统计工具的设计与实现

简单地说，内存统计工具的作用是为了查看当前系统内存的分配情况。对于系统执行了一个操作或调用，它所引起的内存开销通常是被广泛关注的东西。系统提供了完整的内存管理策略，但系统并没有直接将内存统计的功能提供给用户。这里首先将介绍一下“和欣”（Elastos 2.0）操作系统物理内存管理的相关策略，然后将介绍内存统计工具的实现。

3.4.1 操作系统启动

从操作系统的角度来看，整个计算机系统从用户打开电源到操作系统启动运行的整个过程依次划分成三个抽象阶段：Setup 阶段、Initialize 阶段和 Startup 阶段。首先，系统启动的第一阶段被称为 Setup 阶段，该阶段是最基本的计算机系统准备阶段。经过该阶段后，计算机系统处于一种“基本可以运行内核代码”的状态。接下来进入 Initialize 阶段，该阶段着重进行内核各部分软硬件的初始化工作，继续为最终内核启动做准备工作。最后内核进入 Startup 阶段，最终启动内核并运行之。这三个阶段由 BSP 模块主导，在 BSP 模块和 Kernel 模块（除 BSP 模块外）中交替进行。大致流程如下图所述：



图表 7 操作系统启动

通过上述操作过程之后，系统已经完成了整个引导启动过短，系统就正式运行起来了。

3.4.2 “和欣”操作系统物理内存管理

上面描述了操作系统引导启动的过程，在这个过程中，包括了内存的初始化操作。详细地说，物理内存的初始化是在 KernelStartup 过程中调用了 InitPhysicalMemory 操作。在系统启动时，BSP 模块会以 memory zone 和 memory hole 的形式向 Kernel 模块提供关于物理内存的信息。其中 memory zone 是可分配的物理内存，而 memory hole 是不可分配的物理内存，这可以从它们各自的名字里看出来。它们有相同的数据结构，里面包括了区域的基础的偏移。系统里的内存相关的操作都是对数据结构进行访问和维护。

系统内核里提供了访问这些数据信息的一个非常有用的数据结构：

```
typedef struct {
    // cache
    UINT uAllocedBytesOfCache;
    UINT uTotalPagesOfCache;

    // buddysys
    UINT uAllocedKernelPages;
    UINT uAllocedPhysicsPages;

    // total Pages
    UINT uTotalPages;

    // free pages
    UINT uFreePages;
} StatInfo, *PStatInfo;
```

数据结构里包括了缓存、物理页面的大小、可分配数量等。另外，系统还提供了 StatInit、GetStatInfo 等几个函数，用于访问当前系统内存相关的信息。其中 GetStatInfo 是最重要的测量工具，它的原型是这样：

```
EXTERN void GetStatInfo(StatInfo *pStatInfo);
```

在需要检查内存配置的地方声明一个 StatInfo 类型的变量，调用 GetStatInfo 函数后，就可以了。因为 GetStatInfo 函数没有返回值，所以这个函数肯定不会执行失败。

3.4.3 内存统计工具的实现

如果上面能够解决所有问题，内存统计的测试工具就有了，但事实上问题还没有解决。GetStatInfo 函数是内核里的函数，对于用户程序而言是不可见的。性能测试里许多的情况是在用户程序里测量某一系统调用和系统操作的开销，对于这样的需要，必须采用另外的办法，让用户程序可以进行 GetStatInfo 调用。为此，现在采取的办法就将这个调用写成驱动，配置到 DDK 里去，用户程序可就按驱动的方法去访问和调用。当然由于这并不是真正意义上的设备驱动，所以准确地说应该是伪驱动。

驱动的编写主要就是从驱动的基础 Driver 继承接口并实现特定的功能。Driver 是从 IDriver 接口继承下来的 C++类，接口 IDriver 里定义了驱动操作所需要的最基本和最抽象的方法：

```
interface IDriver : IUnknown {
    HRESULT Read(
         [in] UINT64 u64Offset,
         [in] UINT uNumberOfBytesToRead,
         [in, out] SAFEARRAY(byte) ebbData,
         [out] IEvent ** ppCompletionEvent);

    HRESULT Write(
         [in] UINT64 u64Offset,
         [in] SAFEARRAY(byte) ebbData,
         [out] UINT * puNumberOfBytesWritten,
         [out] IEvent ** ppCompletionEvent);

    HRESULT Control(
         [in] INT nControlCode,
         [in] SAFEARRAY(byte) ebbInData,
         [in, out] SAFEARRAY(byte) ebbOutData,
         [out] IEvent ** ppCompletionEvent);
}
```

其中的读操作一般是从设备读入相关数据，读操作则是向设备输出相关数据，控制操作则可以是与设备交互的控制和交流。这些都是抽象意义上的方法，具体的意义与实现设备有关。

从需求上考虑，用户程序只需要进行 GetStatInfo 调用，具体实现时是采用通过 Control 方法来传递这个操作，在实现先从 Driver 继承一个 C++类，在它的 Control 方法的内部进行 GetStatInfo 调用，将 StatInfo 结构体按内存拷贝到 Control 方法的第三个参数 ebbOutData，用户操作时只需要按 StatInfo 的结构去访问那块内存区域就可以得到所需要的数据了。

至于其他两个方法，则可以直接返回 E_NOTIMPL，表示方法未实现。这样对用户没有任何影响。

然后是实现一个厂方法，用于创建功能类，实现很简单，就是开辟内存新建对象，并返回接口 IDriver 指针。驱动实现后要在驱动表里注册，内容包括添加驱动的名字，添加前面已经实现的厂方法，注册对应关系等。之所以要添加驱动的名字，主要是因为驱动是“和欣”的命令服务机制，即通过名字访问服务。内存统计的驱动名字叫“device: mstatserver”。厂方法已经实现，作为一个回调，写到驱动注册表里就可以了。

3.5 计时工具的设计与实现

时间上的开销是性能测试中最关注的内容。很大程度上，系统执行操作和系统响应的时间是一个非常重要的指标，它是操作系统性能的最直接的表现。计时工具可以使用系统内置的时钟，但这有许多局限性，特别是对于内核性能的测试，精度要求也无法满足。使用硬件提供的定时器是最好的选择，下面首先将研究 LH7A400 的定时器，接着将介绍定时计时器的实现，然后是用户计时库的封装，主要处理测试数据结果的输出、统计等。

3.5.1 Sharp 板定时器研究

⑴ Sharp 定时器介绍

Sharp 板提供了三个可以通过软件读写的 16 位定时器，这三个定时器在启动将按各自固有的频率递减，当递减到 0 后，会触发一个定时器中断。

定时器 1 和定时器 2 是两个完全一样的定时器，定时器 3 则有一些不一样的地方。区别主要在于前两个定时器有两种固有频率，开发人员可以在 508.469KHz 和 1.9994KHz 中选择一个频率，而定时器 3 只有 3.6864MHz 一个频率。其他的读写、控制操作都是一样的。相对而言，定时器 3 的频率要大得多，而且最主要的是，系统里并没有用到这个定时器，所以它完全可以用来计时。定时器 3 是 16 位，频率是 3.6864MHz，折算下来每个节拍是 271 纳秒，一个周期下来是 17.7 毫秒。

⑵ 定时器寄存器

下面的表格里是定时器寄存器相对于基地址 0x8000.0C00 的偏移地址：

表格 1 定时器寄存器内存表

偏移地址	定时器寄存器	描述
0x00	LOAD1	用于读取和写入定时器 1 的初始值
0x04	VALUE1	用于读取当前定时器 1 的值
0x08	CONTROL1	用于读取和写入定时器 1 的配置
0x0C	TCEOI1	清定时器 1 中断
0x10	///	系统保留。读取时将返回不可预告的数据，同时禁止写入。
0x20	LOAD2	用于读取和写入定时器 2 的初始值
0x24	VALUE2	用于读取当前定时器 2 的值
0x28	CONTROL2	用于读取和写入定时器 2 的配置
0x2C	TCEOI2	清定时器 2 中断
0x40	BZCON	用于读取和写入蜂鸣器输出的配置
0x80	LOAD3	用于读取和写入定时器 3 的初始值
0x84	VALUE3	用于读取当前定时器 3 的值
0x88	CONTROL3	用于读取和写入定时器 3 的配置
0x8C	TCEOI3	清定时器 3 中断
0x30 – 0x90	///	系统保留。读取时将返回不可预告的数据，同时禁止写入。

对于定时器 3 来说，LOAD3 用于设定定时器的初始值，VALUE3 则可以读取当前定时器的值，CONTROL3 用于控制定时器，包括开启和关闭和模式选择，模式选择即选择在定时器溢出后是直接从 0xffff 还是从上次设定的初始值重新开始，TCEOI3 用于清定时器中断。寄存器都是 32 位的，具体使用时可以直接对指定内存进行操作。

⑶ 定时器中断

一般来说，计时工具在没有中断的情况下更简单，启动和结束两个操作就足够了。但有些情况需要有时钟中断。下面的表格里是 LH7A400 中断控制器里寄存器相对于基地址 0x8000.0500 的偏移地址：

表格 2 中断控制器寄存器内存表

偏移地址	定时器寄存器	描述
0x00	INTSR	中断状态寄存器（Interrupt Status Register） 1 = 相应位所对应的中断请求被允许 0 = 相应位所对应的中断请求被禁止
0x04	INTRSR	中断原始状态寄存器（Interrupt Raw Status Register） 1 = 相应位所对应的中断请求允许 0 = 相应位所对应的中断请求禁止
0x08	INTENS	中断使能开启寄存器（Interrupt Enable Set Register） 1 = 允许相应位所对应的中断请求

		0 = 无效
0x0C	INTENC	中断使能关闭寄存器（Interrupt Enable Clear Register） 1 = 禁止相应位所对应的中断请求 0 = 无效
0x10	///	系统保留。读取时将返回不可预知的数据。

从表中可以知道，通过 INTENS 寄存器开中断，通过 INTENC 寄存器关中断，将 INTENS 相应位清空将不会有任何效果，同样地，将 INTENC 相应位清空也不会有任何效果。四个寄存器的 32 位数据中，三个定时器分别在 9、10 和 22 位。所以，如果想把定时器 3 的中断源打开，那么将 INTENS 寄存器里第 22 位置成 1，如果想把定时器 3 的中断源关闭，那么将 INTENC 寄存器里第 22 位置成 1，将这些位清空是没有用的。对于其他情况，操作是一样的。

3.5.2 定时器驱动的实现

前面一节中仔细学习和研究了定时器寄存器以及中断控制器寄存器的内存地址、功能等内容。对于计时，在前一个测试点，设定定时器初始值（LOAD3），打开定时器（CONTROL3），在后一个测试点，读取定时器的值（VALUE3），重新开始计时。数据的统计可以在后期完成。对于中断，也是类似的操作，开中断源，有中断时清中断并进行响应，最后关中断。定时器里的寄存器以及中断控制器里的寄存器在内存分布上都是连续的，出于编码的简洁和日后的维护，先定义下面两个结构体：

```
// Timer Module Register Structure
typedef struct {
    volatile unsigned int load;      // RW
    volatile unsigned int value;     // RO
    volatile unsigned int control;    // RW
    volatile unsigned int clear;     // WO
} TIMERREGS;

// Interrupt Controller Module Register Structure
typedef struct {
    volatile unsigned int status;
    volatile unsigned int rawstatus;
    volatile unsigned int enableset;
    volatile unsigned int enableclear;
} INTCREGS;
```

这样，就可以将结构体指针直接指向定时器和中断控制器的基地址。相对于操作物理内存，对一个结构体的成员变量进行读写要显得更加直观。比如说：

```
#define TIMER3_BASE = (0x80000000 + 0x0C80)        // Timer3 的物理地址
TIMERREGS* pTimer = ((TIMERREGS *)(TIMER3_BASE));  // 直接指向物理内存
pTimer->load = 0xffff;                             // 定时器初值为 0xffff
……
```

前面实现内存统计的伪驱动时，由于功能单一，使用一个返回参数就可以了。这里定时器的操作要复杂一些，设置初始值、打开/关闭定时器，读取定时器当前的值以及清定时器中断等。这些将通过 Control 方法的第一个参数 nControlCode 来实现，这个参数是一个 32 位无符号的整数，下面规定了 32 位的具体含义：

```
// Timer Server Helper Macro
//  +------------+-------------+------------+---------------+
//  |31          |23           |15          |7            0 |
//  +------------+-------------+------------+---------------+
//  |<---------------- ControlCode (32b)------------------->|
//
```

```
//  [31:27] : RESERVED
//  [26]    : TIMER3
//  [25]    : TIMER2
//  [24]    : TIMER1
//
//  [23]    : ENABLE
//  [22]    : MODE
//  [21:20] : RESERVED
//  [19]    : CLKSEL
//  [18:16] :   000 LOAD
//              001 VALUE
//              010 CONTROL
//              011 TCEOI
//              100 T#INT_ON
//
//  [15:0]  : LOADx
```

这是从源代码里截取出来的对 32 位命令参数的含义的说明。比如需要控制定时器 3 时，将 26 位置位，将 23 位置位时将开启定时器，等等。具体的源代码可以参考本文附录。

输出的参数将通过 Control 方法的第三个参数 ebbOutData 输出，与前面一样，它仅仅是一块内存块，不考虑具体的数据类型。驱动实现里的其他两个方法，则可以直接返回 E_NOTIMPL，表示方法未实现。这样对用户没有任何影响。

接下来和前面一样，实现一个厂方法，用于创建功能类，并返回接口 IDriver 指针。驱动实现后在驱动表里注册，计时的驱动名字叫“ device: timerserver ”。厂方法已经实现，作为一个回调，写到驱动注册表里就可以了。

3.6 用户计时库的封装

3.6.1 计时驱动直接使用的问题

通过上述驱动程序，用户程序里可以通过完成对定时器的各种读写和控制操作，比如下面的代码：

```
IDriver* pDriver = NULL;
HRESULT hr = EzFindService(EZCSTR(“device:timerserver”),
                           (IUnknown **)&pDriver);
if (FAILED(hr)) {
   printf(“TimerServer initialize Failed…\n”);
   exit(1);
}
hr = pDirver->Control(…);
if (FAILED(hr)) {
   printf(“TimerServer control failed…\n”);
}
pDriver->Release();
```

这段代码并不复杂，但并不是性能测试所需要的代码。首先前后两个测试点都需要进行开关定时器或者是读取操作，其次这样的操作需要在大量的性能测试代码中出现，重复这样的代码并不是一件让人感到愉快的事。事实上人们需要一个性能良好、使用简便的计时工具，从而可以有更多的精力投入性能测试代码的开发工作中去，而不再是将精力分散在这些方面。

3.6.2 封装用户计时库

相同的代码不应该重复，重复的代码应该进行封装，这是最容易想到的。比如在第一个测试点时执行一个函数 Start，在第二个测试点时执行另一个函数 Stop，其中 Start 方法开始计时，Stop 方法结束计时，或者计算出时间间隔，或者将数据存储起来到后期进行统计。这样，将上面的代码写到函数 Start 和 Stop 里去，在具体的性能测试代码里，只需要在测试点上进行简单的函数调

用。虽然进行了简单的函数封装，但对于性能测试代码的实现来说，意义却不寻常。从某种意义上说，测试代码甚至不用考虑具体时通过什么工具进行时间统计的。这样的话，这些工作完全可以独立出来分别解决。

当然上面的方法只是迈出的第一步而已，接下来要做更多的事情。最好的办法就是将这些 Start、Stop 或者更多函数作为一个函数工具库提供给性能测试程序。函数工具库和性能测试程序之间进行一个约定，之后两边的任务就完全可以独立进行。

作为一个函数库，在对于接口的调用有了统一的约定之后，具体的实现完全是函数库的事情了。在“ 和欣 ”开发环境下，首先要修改用于编译的 sources 文件。

```
TARGET_NAME=timing
TARGET_TYPE=lib

EXPORT_HEADERS =   \
    timing.h       \

SOURCES= \
    timing.cpp \
```

其中 timing.h 是测试程序里需要引用的函数库的头文件，里面声明了双方约定的调用函数。在 timing.cpp 文件里是对这些调用函数的实现。至于最前面的两句，则表示编译器将（向特定的路径里）输出以 timing.lib 为文件名的库文件。

Start 函数用于在测试点 1 调用，主要是开启定时器计时，记录定时器初始值；Stop 函数则用于在测试点 2 调用，此时关闭定时器，记录当前定时器的值；最后可以在函数 PrntResult 中进行计算，比如说计算平均值、最大值、最小值或者标准方差等数据结果，然后再输出，这里输出可能是文件，也可能是标准输出。这就是函数库需要实现的东西，用户程序在使用时也完全是按照这个过程进行调用。

```
#ifndef __TIMING_H__
#define __TIMING_H__

#define TSTTIMES 1000

// insert the function at the front of the tested function
EXTERN_C void Start();

// insert the function after the tested function
EXTERN_C UINT64 Stop();

//print result after testing
EXTERN_C void PrntResult(wchar_t * funcName);

#endif // __TIMING_H__
```

这段 C 语言的代码就是刚才的设计的结果。其中宏__TIMING_H__用于防止文件被多重引用，这种方法在 C/C++程序开发过程中非常普遍。宏 TSTTIMES 规定了循环的次数，因为测试次数足够多才可以从统计意义上保证数据的正确性和有效性。在这样设计后，测试用例里就不再需要关心伪驱动、命名服务等这些额外的问题了。毕竟，在需要的地方加上一对 Start/Stop，最后通过 PrntResult 打印结果已经是所能做到的最简洁的封装了。

3.6.3 改进用户计时库

将代码模块分离，独立地封装成函数库提供给用户程序调用是一个非常好的改进，这部分改进并没有考虑时间开销的问题，下面将继续对用户计时函数库进行改进。

Start 和 Stop 函数里调用 EzFindService 获取接口指针，通过接口指针进行相关调用，最后释放接口指针，这样的开销如何呢？很显然，开销相当可观。这个库的目的就是用于测试时间开销，

而库本身却又在进行时间消耗，准确地说是库的计时将自己的时间消耗也计算在内，这就是一个需要改进的地方。

每一次调用都需要通过 EzFindService 获取接口指针，用完之后又马上释放指针，这样做是非常浪费时间的。改进的策略就是维护一个全局的、对用户不可见的指针，初始化一次，最后释放，在这段时间里可以无限制地使用。这可以解决额外开销的问题，但又引入了一个问题，什么时候进行接口指针的初始化，什么时候释放指针？一个办法是放在 Start 里面，每次都检测，如果指针是空的，那么创建，否则直接使用。那释放呢？可以给 Stop 一个缺少参数，默认是关闭的，但在需要释放指针时，可以完成指针释放的调用。

这的确是一个可行的解决方法，但仔细想想并不好，在 Start/Stop 里面执行判断，而且每一次都会执行，又会引入额外的时间开销。毕竟一个在 10000 里只有一个成功的判断并不理想，所以现在的办法就是将这两个操作独立出来，TimerServerUp 和 TimerServerDown 的引入就是出于这样的考虑。前者将全局接口指针通过命名服务初始化，检测有效性，如果访问失败了，则可以直接退出应用程序；后者用于显式地释放接口指针。这样设计的结果是用户性能测试程序里需要在开头和结尾处添加两个函数调用，不如原来那样简练，但它将额外的时间挡在了外面，可能初始化接口指针需要很长的时间，但这没有任何问题，因为它没有被计时。

这时要考虑另外一个问题。命名服务的访问所起的开销已经排除，但 Start/Stop 里每次通过驱动访问定时器的操作所引起的时间却无法避免。要想访问定时器，一定要经过驱动，经过驱动就一定会有时间开销，这个问题现在是这样解决的。保持 Start/Stop 现在有设计，在开始计时之前先让 Start/Stop 空转，即调用 Start 开始计时，之后马上调用 Stop 停止计时，记录下所经历的时间间隔，如此循环 TSTTIMES 次。性能测试时的 Start/Stop 调用结束时的时间间隔减去空转的时间间隔，从理论上讲，就应该是被测试代码的真实开销。

这段空转的代码放在哪里呢？显然，TimerServerUp 是非常好的选择，也就是在接口指针初始化后马上进行计时器的空转，用于测量出计时器本身的开销。至于将这部分开销去除，则完全可以在 PrntResult 里实现，反正那时已经不在计时了，增加计算量并不会有什么问题。下面的代码段就是修改后的函数库声明文件：

```
#ifndef __TIMING_H__
#define __TIMING_H__

#define TSTTIMES 1000

// timer server up and down
EXTERN_C void TimerServerUp(int nLoopCount = 0);
EXTERN_C void TimerServerDown();

// insert the function at the front of the tested function
EXTERN_C void Start();

// insert the function after the tested function
EXTERN_C UINT64 Stop();

//print result after testing
EXTERN_C void PrntResult(wchar_t * funcName);

#endif // __TIMING_H__
```

尽管做了这么多改进，其中必须还有其他原因造成的误差，在具体进行性能测试时需要再回来做相应的改进。但至少这样设计之后可以尽可能地逼近真实的数据，保证测试数据结果的精度和可靠性。

对数据结果进行统计则是另一个需要考虑的问题，按现在的设计，数据结果就以如下形式出现：

```
// header
value[0000]=215
```

```
value[0001]=217
value[0002]=285
value[0003]=238
value[0004]=208
// ……
value[0997]=321
value[0998]=298
value[0999]=289
// ……
```

这段格式化的数据结果将会输出到文件，也有可能到标准输出，无论怎样，对于数据结果统计而言没有区别。采用重定向功能就可以把操作统一起来。具体的实现代码包括用 C 语言实现和 Perl 语言的实现两种，具体可以参考后面的附录。

3.7 测试工具模块的配置和编译

“和欣”（Elastos 2.0）操作系统 SDK 的 src 开发目录下有一个 testing 目录，这个目录下包括了 2.0 测试的所有内容。这个目录的代码不会影响操作系统的开发和编译。在 testing 目录下有一个 perform 目录，里面都是和性能测试相关的全部内容，包括了用户计时函数库和详细的性能测试用例。

内存统计和定时器两个伪驱动在 DDK 的驱动开发目录下，并且已经在驱动配置表里注册，同时还有一些与这两个驱动相关的代码分散在其他开发目录。每次 DDK 的编译都会将这两个驱动编译到 kernel.exe 里面去。作为只提供给性能测试使用的驱动程序，这样的做法并不合适。对于大多数情况，操作系统并不需要提供这样的驱动，只有在进行性能测试时，才需要进行编译。

为此，在驱动配置文件、用于编译的 dirs、sources 文件等与这两个驱动程序相关的代码使用 C 语言的宏包装起来，默认情况下把宏关闭，也就是说默认情况下这两个驱动程序不参加操作系统的编译，这样也就不会对操作系统有任何影响。在需要的时候，可以把宏打开，重新编译系统后就可以向用户程序提供驱动服务了。

在系统里用于编译的 makefile.gnu 文件里添加宏的定义，这个定义将对所有文件都有效。在不需要的时候，关闭这个宏就可以了。但这样做也麻烦，makefile.gnu 用于整个的系统编译，反复修改 makefile.gnu 并不是一件好事。

现在的解决方法是设置环境变量的方法，即 makefile.gnu 里检测 PERFORMANCE_TESTING 这个环境变量是否已经设置，如果设置了，那么定义宏_PERFORMANCE_TESTING，如果没有设置，那么也就不需要定义这个宏：

```
ifneq "$(PERFORMANCE_TESTING_ARM)" ""
C_DEFINES:= -D_PERFORMANCE_TESTING =$(PERFORMANCE_TESTING) $(C_DEFINES)
endif
```

在其他文件里，则根据_PERFORMANCE_TESTING 这个宏来实现是否在编译时加载内存统计驱动和计时器驱动进行编译，对于 dirs 和 sources，则根据环境变量 PERFORMANCE_TESTING 来判断：

```
// sources

ifneq "$(PERFORMANCE_TESTING_ARM)" ""
    LIBRARIES:= \
    $(LIBRARIES) \
    $(TARGET_LIB_PATH)\mstatserver.lib \
    $(TARGET_LIB_PATH)\timerserver.lib
endif


// config.cpp

#ifdef _PERFORMANCE_TESTING_ARM
```

```
// MStatServer
#define DEVICENAME_MSTATSERVER     L"mstatserver"
EXTERN IDriver * CDECL CreateMStatServer(uint_t uDeviceNo, void
*pvParameter);

// TimerServer
#define DEVICENAME_TIMERSERVER     L"timerserver"
EXTERN IDriver * CDECL CreateTimerServer(uint_t uDeviceNo, void
*pvParameter);

#endif // _PERFORMANCE_TESTING_ARM


// timerserver.cpp
#if _PERFORMANCE_TESTING == 2
#define TST_INTERRUPT_RESPOND_LATENCY
#endif // _PERFORMANCE_TESTING
```

这样修改的结果是如果需要进行性能测试，那么手动输入 set _PERFORMANCE_TESTING=1 的方法来设置环境就是，就会将它们编译到 kernel.exe，再执行 dropsdk 命令就可以了。当然，如果不进行性能测试，就不需要设置变量，也就是和原来的操作没有任何区别，这也就不会影响到其他开发人员的正常工作。很显然，这样的设计是大家都可以接受的。

装

订

线

4 内核性能测试的设计与实现

本章首先将给出性能测试项目的定义，包括内存模式、命名规范等等，在这基础上将给出性能测试项目的分类，然后就将对性能测试里各个项目进行详细地说明，最后还包括了数据结果和其他一些信息。

4.1 测试项目的定义

4.1.1 内存保护模式

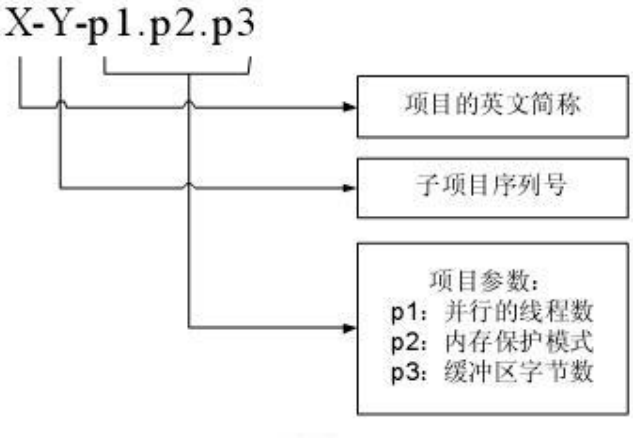
“和欣”Elastos 操作系统通过 MMU 实现内存保护。内核进程与用户进程各自运行在独立的虚拟地址空间，每个进程拥有独立的页表。通过映射，用户进程可以以只读/执行的方式访问部分内核数据及代码。每个测试项目涉及下面四种内存保护模式之一：内核共享模式、内核/用户模式、用户/用户模式以及用户共享模式，分别记做 A、B、C、D。内核共享模式是指测试项目所涉及的所有线程（或中断处理程序）运行在内核进程。这些线程共享内核的虚拟地址空间，线程切换不需要切换页表。内核/用户模式是指测试项目涉及的线程包括内核态线程和用户态线程，它们并不共享虚拟地址空间，线程切换需要切换页表。用户共享模式是指测试项目涉及的线程运行在相同的用户进程，它们共享虚拟地址空间，线程切换不需要切换页表。用户/用户模式是指测试项目涉及的线程运行在不同的用户进程，它们并不共享虚拟地址空间，线程切换需要切换页表。

4.1.2 工作负荷

只有在运行特定任务的情况下，操作系统的运行才有意义。我们需要测试在各种工作负荷下“和欣”操作系统的表现。工作负荷用并发运行的线程数目来衡量，典型的工作负荷为 1、10 和 128 个线程。

4.1.3 测试项目的命名法

每个测试项目用“X-Y-p1.p2.p3”的格式命名。其中 X 代表通过测试项目的英文首字母；Y 为某个项目下属项目的序列号，用小写英文字母表示；“p1.p2.p3”代表三个参数：p1 为并发运行的线程数目，不关心为 X，p2 为内存保护模式，p3 为缓冲区字节数，可选。如下图所示：



图表 8　测试项目的命名规范

4.2 测试项目的分类

性能测试衡量“和欣”（Elastos 2.0）操作系统在低工作负荷的情况下所能达到的最优性能指标。性能测试包括的项目有：最小内核尺寸、用户态运行最大进程数、用户态运行最大线程数、

创建进程线程的最小内存需求、中断延迟、系统调用时间、线程辅助函数、线程切换时间、进程切换时间、实时优先级线程切换、SLEEP 时间精度测试、CRITICAL_SECTION 测试、同步互锁函数、MUTEX 性能测试、用户进程初次加载组件时间等。下面各小节将就这些项目进行详细的说明，附录里有部分源代码。

4.3 系统最小内核尺寸

4.3.1 静态尺寸

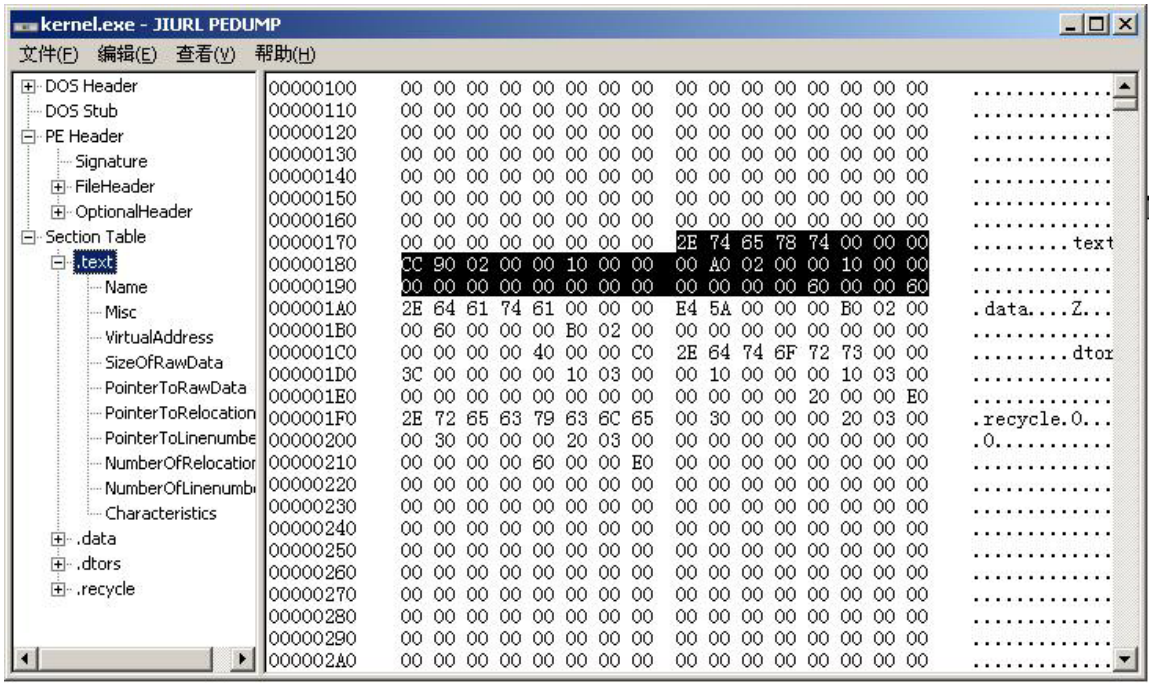
静态信息的测量无需附加硬件环境和测试方法，编译器生成二进制，分析其 text 段，data 段和 bss 段以及编译器为代码中制定的特殊段的大小即可。kernel.exe 是 PE 格式的二进制文件，所做的信息测量必须按 PE 格式的规范进行。

PE 格式，是 Windows 的可执行文件的格式。Windows 中的 exe 文件、dll 文件都是 PE 格式。PE 就是 Portable Executable 的缩写。Portable 是指对于不同的 Windows 版本和不同的 CPU 类型上 PE 文件的格式是一样的，当然 CPU 不一样了，CPU 指令的二进制编码是不一样的。只是文件中各种东西的布局是一样的。PE 文件结构的总体层次分布如下所示：

DOS MZ Header
DOS Stub
PE Header
Section Table
Section 1
Section 2
……
Section n

PE 文件最开始是一个简单的 DOS MZ header，它是一个 IMAGE_DOS_HEADER 结构。有了它，一旦程序在 DOS 下执行，DOS 就能识别出这是有效的执行体，然后运行紧随 MZ Header 之后的 DOS Stub。DOS Stub 是一个有效的 DOS 程序。当程序在 DOS 下运时，输出诸如"This program cannot be run in DOS mode" 这样的提示。这是编译器生成的默认 stub 程序。紧接着 DOS Stub 的是 PE Header。它是一个 IMAGE_NT_HEADERS 结构。其中包含了很多 PE 文件被载入内存时需要用到的重要域。执行体在支持 PE 文件结构的操作系统中执行时，PE 装载器将从 DOS MZ header 中找到 PE header 的起始偏移量。因而跳过了 DOS stub 直接定位到真正的文件头 PE header。PE Header 接下来的数组结构 Section Table (节表)。如果 PE 文件里有 5 个节，那么此 Section Table 结构数组内就有 5 个成员，每个成员包含对应节的属性、文件偏移量、虚拟偏移量等。

有很多分析PE信息的工具可以用来分析kernel.exe中的相关信息，比如说各个段的大小等等。



图表 9 PE 格式解析工具

上面是利用Jiurl编写的PEDUMP.exe[①]工具对kernel.exe解析的结果，界面里非常清楚地罗列了PE结构里的全部内容。Kernel.exe的Section表里包括Text段、Data段、Dtors段和Recycle段。每段都有用于标识段初址的VirtualAddress和段的大小SizeOfRawData，从这些变量中可以分析出性能测试所需要的内核文件的静态尺寸大小。

4.3.2 动态尺寸

动态内核大小是指内核启动后占用的所有内存资源的大小，也就是系统从开始启动到开始等待第一个用户进程创建运行的过程中需要消耗内存资源。测量点只需测试系统初始化结束后准备进入 Idle 态前系统所消耗的内存，输出结果，所消耗的内存的多少即为内存最小内核尺寸。期间，内核必须创建多个系统服务线程及相关数据结构，为后续的服务奠定基础。

编写内存统计的驱动时已经讨论过系统的启动过程。在 BspStartup 函数最后调用 KernelStartup 函数，KernelStartup 不返回，否则打印出错信息并导致系统死机(Halt)。KernelStatup 进行内核初始化并在最后调用 Thread0Routine，Thread0Routine 函数在进行一系列操作后进行 Idle 循环，所以测试点就选在进入 Idle 循环之前。测试调用 GetStatInfo 函数完成。关于 GetStatInfo 这个函数的声明和用途，前面已经有过论述。

4.3.3 内核尺寸测试结论

用于测试的是“和欣”（Elastos 2.0）操作系统ARM的Release[②]版本，kernel.exe的大小为217,808 字节，Text段为 172,032 字节，Data段为 24,574 字节。动态尺寸为 3,063,808 字节。

4.4 用户态下运行的最大进程、线程数

4.4.1 用户态下可以创建的最大进程数

进程通常被定义为正在运行的程序实例，每个进程有自己的地址空间，包含可执行程序的代码和数据，还包含动态分配的内存空间。对于每个进程，在操作系统内核中都有唯一对应的进程对象，用来控制、管理、监视进程。它存放着关于进程的信息。用户必须通过进程对象访问进程。

[①] 这是Jirul开发的PE解析工具。详细内容可以参考他的主页http://jirul.yeah.net。
[②] 这里用的是 2005 年 5 月 11 日更新的版本，以下测试都是基于这个版本。

“和欣”操作系统环境下，可以通过调用 API 中的 EzCreateProcess 函数直接创建并运行一个进程，并返回指向进程对象接口的指针。EzCreateProcess 创建进程成功后，将返回 S_OK，该进程立刻运行，否则将返回一些出错信息。

测试时先构造这样的运行环境：主线程不断循环，通过 EzSleep(0)放弃 CPU 时间。然后是在系统启动后循环调用 EzCreateProcess 创建新进程并开始运行，同时打印进程数目，直至系统提示没有可用内存为止。由于系统已经运行了 Elacmd 以及父进程，所以需要在所得到的结果的基础上再加上这两个进程计数。

测试的结果显示，在当前的硬件配置下最多可以运行 51 个进程。这里有一点要说明：“和欣”（Elastos 2.0）操作系统下的进程数目没有逻辑上的限制，系统会创建进程，直到物理内存耗尽，系统会显示 *ERROR* Not enough physical memory 为止。

4.4.2 用户态下可以创建的最大线程数

线程是程序执行的单位，用来执行进程地址空间中的代码。一个进程包含了一个或多个线程，每个线程有自己的执行上下文环境，包括 CPU 寄存器和栈，因此各个线程之间的执行互不干扰。和进程类似，每个线程都有在操作系统内核中对应的线程对象，用户通过线程对象控制、管理、监视线程。线程对象生命期大于等于它所对应的线程。

通过 API 中的 EzCreateThread 方法可以创建并启动一个新的线程，并返回一个指向线程对象接口的指针。

系统启动后在测试程序的主线程中循环调用 EzCreateThread 方式创建新线程并运行，同时打印线程数目，子线程每次循环调用 EzSleep（0）释放 CPU 控制权，直至系统提示创建线程失败。系统所创建的线程数应该包含 shell 的主线程及测试程序本身的主线程。测试的结果显示，在当前的硬件配置下最多可以运行 511 个线程。

4.5 创建进程、线程的最小内存需求

测试进程、线程创建的内存需求时，第一个测试点放在进程、线程创建的 API 调用之前，第二个测试点则放在新创建的进程、线程里的第一条指令位置。在两个测试点上分别测量出当前系统的内存消耗并输出到屏幕，手动记录并比较两次结果。

内存的消耗可以通过访问先前已经实现的驱动来完成。进程和线程创建的内存需求的采用同样的测试方法。

表格 3 内存开销测试结果

创建进程的最小内存需求	测试结果：122,880 字节
创建线程的最小内存需求	测试结果：8,192 字节

4.6 中断响应延迟时间

中断响应延迟时间就是测量从硬件中断信号开始，到中断服务程序开始执行第一条指令的时间间隔。

对于 ARM 系统，可以采用如下方法进行：设置 TIMER3 的工作方式为 periodic 模式，频率为 3.6864MHz，TIMER3 的中断处理程序就是一个取得 TIMER3 的计数值操作，这样并设置初始的计数值为 0xFFFF, 然后打开 TIMER3 的中断。TIMER3 开始计数并且计数为 0 时，再减 1 时就会产生一个 TIMER3 的中断。在 TIMER3 的中断处理程序中取得当前的 TIMER3 的计数值。这样便计算出由发出 TIMER3 中断开始到进入用户中断服务程序所用的时间。在一定的实际运行环境中，长时间的进行测试，可以看出系统最大的中断延时。

TIMER3 中断服务程序并没有在计时驱动程序里实现，这部分功能需要进行添加。包括打开 TIMER3 中断源，向系统注册 TIMER3 中断服务程序。前者可以通过中断控制器打开 TIMER3 中断，后者则可以通过 API 中的 RegisterIsr 函数完成，这里主要是注册中断优先级和中断服务程序。

每一次定时器中断响应时，清中断，读取并记录当前定时器的值，可以看出，中断服务程序的工作很简单。在测试达到指定次数后禁止定时器中断，并将数据结构输出。这些操作可以完整地在驱动程序里实现，用户程序里需要测试时只需要打开定时器中断即可。定时器驱动的 Control

方法为此将增加这种功能。至于中断服务程序的调用，则可以放在创建定时驱动的厂方法里。由于定时器中断服务在大多数情况下是不需要的，所以可以进行这样的约定，在设置全局的环境变量 set PERFORMANCE_TESTING_ARM=2 时将定时器中断相关的代码编译到 kernel.exe 中去。而对于其他的情况，则没有任何影响。

项目测试时将考虑操作系统恰当的工作负荷。现在的做法是运行 1、10 和 128 个并发的工作线程，下面是测试的结果：

表格 4 中断迟延测试结果

负载描述	取样数	单位	平均值	最小值	最大值	标准方差
无其它负载	500	微秒	3.265	3.252	6.775	0.192
同时起 10 个不结束的线程	500	微秒	3.637	2.981	15.45	0.948
同时起 128 个不结束的线程	500	微秒	4.589	2.981	13.28	1.692

4.7 系统调用时间

“和欣”操作系统采用 CAR 机制通过内核服务构件对象的接口调用实现用户态到内核态的系统调用，对于“和欣”而言，系统调用即从用户态到内核态的 COM 调用。由于系统调用过程较为复杂，且效率与传递的参数紧密相关，必须建立多个测试子项目对不同的参数传递进行测试。系统调用参数共分为输入参数参数、输出参数和输入输出参数共三种类型。每个子项目都必须包含三种测试。在系统中实现了一个专门为性能测试使用的接口 IPerformTest，定义的方法的参数类型即为待测试的参数类型，方法的实现均为空。测试程序中分别调用这些方法测试传递不同参数类型时内核态和用户态的切换延迟时间。

下面是测试的结果：

表格 5 系统调用测试结果

测试描述	取样数	单位	平均值	最小值	最大值	标准方差
不带参数	10,000	纳秒	163.3	149.1	414.6	35.31
[in] EzStr 类型参数	10,000	纳秒	199.6	187.0	455.3	34.16
[in] EzStrBuf 类型参数	10,000	纳秒	160.1	143.6	477.0	44.23
[out] EzStrBuf 类型参数	10,000	纳秒	168.2	151.8	430.9	38.40
[in,out]EzStrBuf 类型参数	10,000	纳秒	164.4	149.1	585.4	53,08
[in] EzArray 类型参数	10,000	纳秒	159.3	149.1	447.2	41.34
[out] EzArray 类型参数	10,000	纳秒	160.0	146.3	414.6	39.15
[in,out] EzArray 类型参数	10,000	纳秒	168.3	151.8	466.1	46.12
[in,out] EzArray 类型参数	10,000	纳秒	193.8	178.9	561.0	48.83
[in] int 类型参数	10,000	纳秒	164.6	149.1	471.5	45.59
[out] int*类型参数	10,000	纳秒	163.7	149.1	531.2	48.40
[in] IUnknown*类型参数	10,000	纳秒	179.5	168.0	468.8	31.91
[out] IUnknown**类型参数	10,000	纳秒	160.3	149.1	447.2	32.72

4.8 线程创建时间

这里主要测试系统创建一个线程需要消耗多少时间。测试将直接使用已经实现的用户计时函

数库。测量从执行启动线程的指令开始，到新的线程开始执行第一条指令的时间间隔。

通过调用 API 中的 EzCreateThread 函数创建新线程，在创建新线程之前通过 Start 开始计时，在线程函数的入口处通过 Stop 停止计时并读取测试值。完成需要的测试次数后打印测试结果。

当然在使用函数库之前需要调用 TimerServerUp 函数，在测试结束之前调用 TimerServerDown 函数。从功能上讲，这样的调用是调用驱动服务程序，计算计时工具本身的开销，最后再释放驱动接口，不过对于用户而言，这仅仅一个调用的约定，至于它究竟完成了什么功能，则完全可以不去关心。下面是用于测量线程创建时间的代码片断：

```
#include <elastos.h>
#include <stdio.h>
#include "timing.h"   // 包含用户计时库头文件

HRESULT __stdcall Start(void * arg)
{
    Stop();           // 测试点 2：在创建的线程里读取时间并记录
    return 0;
}

int __cdecl main(int argc, char **argv)
{
    IThread* pThread;
    TimerServerUp();  // 加载计时服务

    for (int i = 0; i < TSTTIMES; i++) {
        Start();      // 测试点 1：在创建线程之前读取时间并记录
        HRESULT hr = EzCreateThread(Start, NULL, 0, &pThread);
        if (FAILED(hr)) {
            printf("Error......Fail to CreateThread[0x%08x]\n", hr);
            goto EXIT;
        }
        pThread->SetPriority(ThreadPriority_Highest);
        pThread->Join(INFINITE);
        pThread->Release();
    }
    PrntResult(L"TF-Y-1-C.pfm");  // 输出测试数据结果
EXIT:
    TimerServerDown();            // 卸载计时服务
    return 0;
}
```

表格 6　创建线程时间

取样数	单位	平均值	最小值	最大值	标准方差
200	微秒	99.32	89.43	356.91	18.92
500	微秒	99.52	89.16	392.14	14.28
1,000	微秒	99.40	89.97	361.51	9.792

4.9 Sleep 的时间精度测试

“和欣”（Elastos 2.0）操作系统的 API 里提供了 EzSleep 函数。调用此函数的线程会进入睡眠状态，在指定时间达到之后，系统会自动唤醒此线程并继续执行。其它线程可以使用 Interrupt 方法打断线程的睡眠状态。当参数为 0 时，线程放弃时间片所有权，将其让给任意具有同优先级的就绪线程，如果没有其它同优先级就绪线程，函数立即返回，线程继续执行；当参数为 INFINITE 时，调用 EzSleep 函数的线程将一直保持睡眠状态。

Sleep 的时间精度测试就是用于在一定运行环境中，EzSleep 函数中指定的时间和线程实际等

待的时间的误差。具体的测试环境为单个用户进程，10 个子线程，对其中的一个线程测试，其他线程不做具体的任务。

从“和欣”文档可以知道，EzSleep 是以毫秒为单位的，定时器 3 的频率为 3.6864MHz，测试长度为 17ms。因此，对于 1 毫秒、5 毫秒和 10 毫秒的测试用 Timer3 定时器，而 10 毫秒以上的精度测试通过获取当前的系统标准时间（UTC）来实现。测试工具的选择通过宏 USESYSTIME 来控制，需要重新编译生成 timing.lib。

为什么不通过定时器溢出中断进行累计呢？主要还是性能和误差的考虑。测试时完全可以在定时器中断服务程序中记录循环的次数，最后统一处理，但中断服务有延时，为精度测试引入了大量的误差。因此，对于 10 毫秒以上的睡眠时间，采用获取当前的系统标准时间的方法来测试。在用户计时函数库中将添加这部分功能的实现。

睡眠精度测试选取了 1 毫秒、5 毫秒、10 毫秒、15 毫秒、25 毫秒、50 毫秒、75 毫秒和 100 毫秒，下面的表格是测试的结果：

表格 7　Sleep 精度测试结果

测试时间	取样数	单位	平均值	最小值	最大值	标准方差
1	1,000	毫秒	9.90	1.97	9.95	0.25
5	1,000	毫秒	9.95	1.08	9.99	0.28
10	1,000	毫秒	9.95	1.00	10.05	0.28
15	1,000	毫秒	20.01	19.99	39.99	0.63
25	1,000	毫秒	30.25	30.00	130.0	4.456
50	1,000	毫秒	50.11	50.00	140.0	2.914
75	1,000	毫秒	80.12	80.00	140.0	2.681
100	1,000	毫秒	99.02	99.00	119.0	0.632

分析上面的表格，测试时间为 10、30、100 时的数据测试结果与测试时间非常接近，但其他情况下的数据测试结果却有很大的偏差。仔细分析其中的数据，可以得出两个结论：第一，Sleep 睡眠的最少时 10 毫秒。1、5 和 10 毫秒的测试结果均在 10 毫秒左右。第二，Sleep 睡眠精度是 10 毫秒，5、15、25 和 75 毫秒的测试结果分别在 10、20、30 和 80 毫秒左右。事实上，这两个结论可以统一起来，即 Sleep 睡眠的精度是 10 毫秒。

4.10 CRITICAL_SECTION 测试

CRITICAL_SECTION 是一种同步对象， 主要用于进程内多线程互斥地使用临界区。临界区是一种轻量级机制，在某一时间内只允许一个线程执行某个给定代码段。通常在修改全局数据（如集合类）时会使用临界区。事件、多用户终端执行程序和信号量也用于多线程同步，但临界区与它们不同，它并不总是执行向内核模式的控制转换，这一转换成本昂贵。稍后将会看到，要获得一个未占用临界区，事实上只需要对内存做出很少的修改，其速度非常快。只有在尝试获得已占用临界区时，它才会跳至内核模式。这一轻量级特性的缺点在于临界区只能用于对同一进程内的线程进行同步。

在将临界区传递给 EzInitializeCriticalSection 时（或者更准确地说，是在传递其地址时），临界区即开始存在。初始化之后，代码即将临界区传递给 AP 中的 EzEnterCriticalSection 函数和 EzLeaveCriticalSection 函数。一个线程自 EzEnterCriticalSection 中返回后，所有其他调用 EzEnterCriticalSection 的线程都将被阻止，直到第一个线程调用 EzLeaveCriticalSection 函数为止。最后，当不再需要该临界区时，一种良好的编码习惯是将其传递给 EzDeleteCriticalSection。

在临界区未被使用的理想情况中，对 EzEnterCriticalSection 的调用非常快速，因为它只是读取和修改用户模式内存中的内存位置。否则，阻止于临界区的线程有效地完成这一工作，而不需要消耗额外的 CPU 周期。所阻止的线程以内核模式等待，在该临界区的所有者将其释放之前，不能对这些线程进行调度。如果有多个线程被阻止于一个临界区中，当另一线程释放该临界区时，

只有一个线程获得该临界区。

测试的方法是在用户进程创建一个子线程，在子线程中调用相应同步互锁操作函数。主程序开始和结束时分别调用 TimerServerUp 和 TimerServerDown，在每个函数前后调用 Start 和 Stop 方法来测量时间。最后调用 PrntResult 输出结果。

表格 8 CRITICAL_SECTION 性能测试结果

函数描述	取样数	单位	平均值	最小值	最大值	标准方差
EzInitializeCriticalSection	1,000	纳秒	9713	9512	13322	489
EzDeleteCriticalSection	1,000	纳秒	9747	9544	13371	493
EzEnterCriticalSection	1,000	纳秒	917	894	1447	67
EzTryEnterCriticalSection	1,000	纳秒	937	916	1409	69
EzLeaveCriticalSection	1,000	纳秒	379	352	1065	81

4.11 同步互锁函数

Interlocked 系列函数提供一个简单的机制来同步访问被多个线程共享的变量。如果变量处于共享内存中，不同进程的线程可以使用这种机制。在 32 位系统中，参数是 32 位的并且是 32 位对齐的。否则该函数在多处理器的 x86 系统和任何非 x86 系统运行时可能会失败。下面简要地说明互锁函数各自的用途。

InterlockedCompareExchange 函数用来对指定的值进行原子比较，并根据比较后的结果进行数值交换。该函数可以防止多个（一个以上）线程同时对一个变量进行操作。

InterlockedCompareExchangePointer 函数用来对指定的指针进行原子比较，并根据比较后的结果进行指针交换。该函数可以防止多个（一个以上）线程同时对一个变量进行操作。

InterlockedDecrement 函数用来对指定的变量值进行原子减一，并检查结果值。该函数防止多个（一个以上）线程同时对一个变量进行操作。

InterlockedExchange 函数用来完成一对数值的原子交换。该函数可以防止多个（一个以上）线程同时对一个变量进行操作。

InterlockedExchangeAdd 函数用来完成对一个变量的原子加法操作。该函数防止多个（一个以上）线程同时对一个变量进行操作。

InterlockedExchangePointer 函数用来执行一对指针的交换。该函数可以防止多个（一个以上）线程同时对一个变量进行操作。

InterlockedIncrement 函数用来对指定的变量值进行原子加一，并检查结果值，该函数防止多个（一个以上）线程同时对一个变量进行操作。

测试的方法是在用户进程中创建一个子线程，在子线程中调用相应同步互锁操作函数。主程序开始和结束时分别调用 TimerServerUp 和 TimerServerDown，在每个函数前后调用 Start 和 Stop 方法来测量时间。最后调用 PrntResult 输出结果。

表格 9 同步互锁函数性能测试结果

函数描述	取样数	单位	平均值	最小值	最大值	标准方差
InterlockedCompareExchange	1,000	纳秒	384.3	374.0	785.9	28.8
InterlockedCompareExchangePointer	1,000	纳秒	384.7	374.0	1466.1	44.2
InterlockedDecrement	1,000	纳秒	372.8	360.4	748.0	27.8
InterlockedExchange	1,000	纳秒	23.19	13.55	374.0	25.0
InterlockedExchangeAdd	1,000	纳秒	370.8	360.4	777.7	28.0
InterlockedExchangePointer	1,000	纳秒	23.69	13.55	436.3	27.8

InterlockedIncrement	1,000	纳秒	373.7	363.1	775.1	31.7

4.12 MUTEX 性能测试

Mutex 是一个互斥体类的对象，它代表一个“ 互斥体 ”（一种同步对象），通过它可允许一个线程互斥地访问一个资源。互斥体对象在某个时刻仅可以被一个线程所占有，该线程被称为该互斥体对象的属主，其它去获取该互斥体对象的线程将全部进入等待状态，直到互斥体对象的属主释放它，才允许这些等待线程中的一个成功获得到它，其余线程将继续保持等待状态。

为了使用互斥体对象，必须在需要时通过互斥体类构造函数 Mutex::Mutex 构造它。互斥体对象可以被定义为全局或局部变量，或是被创建在内核堆上。

当前线程通过调用 Mutex::Lock 方法或 Mutex::TryLock 方法获取或尝试获取互斥体对象，通过调用 Mutex::Unlock 方法释放已占有的互斥体对象。

互斥体对象是递归的。这意味着一个已经占有互斥体对象的线程可以再次成功地获取此互斥体对象，但是需要成功调用与已经成功调用的 Mutex::Lock 方法和 Mutex::TryLock 方法次数之和相同次数的 Mutex::Unlock 方法才能最终释放此互斥体对象。在互斥体对象析构前必须保证此互斥体对象处于无属主状态。

测试的方法是在用户进程里创建一个子线程，在子线程中分别完成对 Mutex 的创建，无竞争条件获取，无竞争条件下释放和销毁 Mutex 的时间测试。

表格 10 MUTEX 性能测试结果

函数描述	取样数	单位	平均值	最小值	最大值	标准方差
EzCreateMutex	1,000	微秒	243.3	231.1	326.3	8.07
EzCreateNamedMutex	1,000	微秒	292.0	277.5	383.7	10.5
IMutex::Lock	1,000	微秒	16.98	15.45	69.12	3.98
IMutex::Unlock	1,000	微秒	18.62	17.07	63.96	3.59

4.13 用户进程初次加载组件时间

测试用户进程初次加载组件时间。在当前进程中装载映像模块的功能由 API 中 EzLoadModule 函数实现，下面是这个函数的声明：

```
HRESULT EzLoadModule(
    EzStr       esName,
    DWORD       dwFlags,
    IModule **  ppModule
);
```

其中，esName 是模块的映像名，dwFlags 是装载组件时的可选标志，ppModule 则用于指向模块对象接口指针的指针，用以获取装载的模块对象接口指针。

如果参数 esName 为 EZSTR_NULL 或参数 ppModule 为 NULL，返回 E_INVALIDARG。参数 esName 指定的字符串是模块的文件名，必须是全名，并且区分大小写。此函数只能用来装载 DLL 模块，不能装载 EXE 或其它类型的模块。通过调用 EzLoadModule 得到与被装载模块对应的模块对象不是全局或可继承的。一个进程调用 EzLoadModule 得到的模块对象只可以在本进程使用。

前后测试的点在用户态调用 EzLoadModule 的前后。组件加载的性能测试和被加载模块的大小有比较大的关系，所以性能测试前实现了一个标准的 dummy module(tmod.dll)，copy 该模块 10 次并取不同的名字（0tmod.dll～9tmod.dll）进行测试。下面是用于拷贝的批处理：

```
for /l %%i in (0,1,9) do copy tmod.dll %%itmod.dll
pause
```

表格 11 用户进程加载组件时间测试结果

取样数	单位	平均值	最小值	最大值	标准方差
10	微秒	2.67	2.61	2.76	0.05

取样 10 次对于测试程序来说是不够的，但之所以这样是因为通过 mkpkg.exe 工具打包时的文件个数有限，无法实现更多的取样数。

4.14 性能测试小结

在“和欣”（Elastos 2.0）操作系统（ARM-release 版）环境下，实现了用于性能测试的内存统计和定时计时的测量工具。并在此基础上对操作系统的各个方面进行了性能测试，包括系统最小内核尺寸（静态尺寸和动态尺寸）、用户态下运行的最大进程、线程数、创建进程、线程的最小内存需求、中断响应延迟时间、系统调用时间的测量、线程创建时间、Sleep 的时间精度测试、CRITICAL_SECTION 性能测试、同步互锁函数性能测试、 MUTEX 性能测试和用户进程初次加载组件时间等。通过测试程序的运行，获取了大量测试数据，并进行了分析和处理。

由于时间以及其他一些原因，上面所包括的这些性能测试项目并不十分完整。比如说线程切换开销，中断分派时间和同步对象性能。这几项内容中有些是测试不完整，有些则没有进行。

第一，线程切换开销。线程切换开销在于调度线程对上下文的管理以及切换线程栈所消耗的时间，与所属进程无关。测试的计时起点应该是时钟中断发生保存线程上下文时，结束点应该是在恢复线程上下文并准备回到线程运行状态的时刻。

“和欣”（Elastos 2.0）操作系统中的线程调度有两个入口点也有两个出口点，因此把测试语句分别对应嵌在这四个地方。另外，由于在出口点处需要判断当前线程与调度前线程是否相同以及当前线程与调度前线程是否在同一个进程下，从而分别作出统计，所以有三种结果，即同线程、同一进程不同线程、不同进程下的线程。因此需要在调度前纪录线程号，最后在调度后比较当前线程。

“和欣”（Elastos 2.0）操作系统 X86 版本里面线程切换的测试使用了 timepeg。timepeg 是 Linux 系统中用于测试内核性能的标准工具。Pentium 系列 CPU 中支持一个获取开机至当前 CPU 时间戳的指令 RDTSC。Timepegs 的基本工作原理是在代码点处使用 RDTSC 指令获得 CPU 周期数，用两个代码点的 CPU 周期数相减，得到之间经过的 CPU 周期数。

ARM 上并没有类似的计数器，所以最好是利用定时器 Timer3 来模拟实现 timepeg。由于时间原因，同时这样的解决方案也并不一定是可行，所以线程切换开销的性能测试没有实现。

第二，中断分派时间。中断响应延迟时间的测试已经完成，但中断分派时间的测试并没有完成。所谓的中断分派时间，是指从中断处理程序从执行完最后一条指令起到被指定的线程执行第一条指令之间的时间。也就是说，从中断服务程序响应的服务线程所需要的时间。

第三，同步对象性能测试。“和欣”（Elastos 2.0）操作系统里提供的同步对象包括了 CRITICAL_SECTION、Condition、Event、ReaderWriterLock 和 Mutex，其中除了第一个以外，其余的都是以接口的形式出现。

CRITICAL_SECTION 主要用于进程内多线程互斥地使用临界区。Condition 是一个条件变量类的对象，它代表一个“条件变量”，它提供了一种与某些共享数据的谓词（其结果为布尔逻辑表达式）相关的机制。Event 是一个事件类的对象，它代表一个“事件”，一个线程可以通过它来通知其他线程一个事件已经发生了。ReaderWriterLock 是一个读写锁类的对象，它代表一个“读写锁”，通过它可允许一个线程独占访问一个资源，或是多个线程共享访问此资源。而 Mutex 则是一个互斥体类的对象，它代表一个“互斥体”，通过它可允许一个线程互斥地访问一个资源。

目前已经完成了 CRITICAL_SECTION 和 MUTEX 两个同步对象的性能测试，Condition、Event 和 ReaderWriterLock 这三个同步对象的性能测试工作将在下一阶段进行。

结束语

“和欣”（Elastos）操作系统是上海科泰世纪科技有限公司在 863 计划“十五”重大软件专项、信息产业部产业发展基金、上海科教兴市项目的大力支持下开发的网络操作系统，这实现了“和欣”移动软件平台。“和欣”属于继 DOS 及 Windows 之后的第三代操作系统技术，其主要目的是为了迎接继电子邮件和浏览器之后的第三次因特网浪潮：Web 服务（Web Service）。

在科泰公司实习工作期间，作者参与了“和欣”（Elastos 2.0）操作系统性能测试的工作，接触到了“和欣”技术的架构，学习“和欣”操作系统内核体系结构以及“和欣”的开发环境，研究和设计了用于性能测试的工具，编写了性能测试用例。在上述这些工作的基础上，完成了本论文。

在本文中首先对操作系统及操作系统性能测试的内容进行概要介绍，同时对“和欣”技术的总体架构、“和欣”操作系统的体系结构以及“和欣”开发环境进行了介绍。接着在“和欣”环境下研究和设计了用于性能测试的工具，以伪驱动的形式提供给用户程序调用，并且实现了用户计时函数库，解决了驱动访问和调用、额外调用引起的开销、性能测试用例模式、统一数据输出格式等问题。然后利用这些已经实现的工具，编写了性能测试用例并进行测试程序的运行和数据采集工作。由于时间以及其他一些原因，性能测试中关于线程切换、中断分派时间和部分同步对象性能的测试没有完成，但文章里对这些方面的内容进行了主要的介绍。

“和欣”（Elastos 2.0）操作系统的 ARM 版本的开发工作正在进行当中，这些性能测试的数据结果将在实际开发中发挥作用。当然，这次性能测试有其不足之处，包括性能测试的覆盖不完整，测试工具的不完善以及性能测试代码本身的质量等各方面问题。

在科泰实习工作期间，深切地体会到作为一家国内公司在做操作系统核心软件开发方面所面临的诸多困难。真心希望有越来越多的技术工程师能够投身其中，为国产核心软件的大发展贡献自己的力量。

装 订 线

谢辞

在完成此文时，我由衷地感谢我的辅导老师陈榕教授、顾伟楠教授，正是由于他们无私的关心和悉心的指导使我受益匪浅，更帮助我解决了许多问题，使得这次设计得以顺利完成。感谢裴喜龙老师在毕业设计中所提供的热情指导和帮助。

毕业设计的课题以及这篇论文在科泰公司实习期间完成，感谢给过我关心、帮助和指导的许多老师、同事和同学。正是由于他们的热情和耐心，帮助我解决了许多方面的问题，使得我的工作能够顺利进行。

感谢培育我四年的同济大学、所有教授我知识的老师和朝夕相处的同学。

最后要感谢我的父母和亲朋好友的关怀与鼓励，正是由于他们的支持，才取得了今天的成绩。

装

订

线

参考文献

[1] 郑人杰 . 计算机软件测试技术 . 清华大学出版社，1992
[2] 赵炯 . Linux 内核完全注释 . 机械工业出版社，2004
[3] 潘爱民 . COM 原理及应用 . 清华大学出版社，1999
[4] Don Box 著，潘爱民 译. COM 本质论 . 中国电力出版社，2001
[5] 科泰世纪 .《和欣 2.0》大全 . 2004
[6] Patterson David A & Hennessy John L. Computer Organization and Design: The Hardware/Software Interface. 2nd Edition, San Francisco: Morgan Kaufmann, 1994
[7] Patterson David A & Hennessy John L. Computer Architecture: A Quantitative Approach. 3rd Edition, Morgan Kaufmann, 2002
[8] Advanced Risc Machine Limited. ARM Architecture Reference Manual. June 2000
[9] SHARP Microelectronics of the Americas. LH7A400 Universal SOC Preliminary User's Guide.
[10] Dedicated Systems Experts NV. Evaluation Report Definition. June 2001

装 订 线

附录

这里是性能测试过程中实现的代码，包括性能测试工具和部分的性能测试的代码。因为性能测试代码十分类似，所以只给出了其中的一部分。下面首先是性能测试工具中内存统计伪驱动的实现代码。

```
// mstatserver.cpp

#include <ddk.h>

class CMStatServer : public Driver {
public:
    STDMETHODIMP Read(
        /* [in] */ UINT64 u64Offset,
        /* [in] */ UINT uNumberOfBytesToRead,
        /* [out] */ EzByteBuf ebbData,
        /* [out] */ IEvent * * ppCompletionEvent)
    {
        return E_NOTIMPL;
    }

    STDMETHODIMP Write(
        /* [in] */ UINT64 u64Offset,
        /* [in] */ EzByteBuf ebbData,
        /* [out] */ UINT * puNumberOfBytesWritten,
        /* [out] */ IEvent * * ppCompletionEvent)
    {
        return E_NOTIMPL;
    }

   STDMETHODIMP Control(
        /* [in] */ INT nControlCode,
        /* [in] */ EzByteBuf ebbInData,
        /* [out] */ EzByteBuf ebbOutData,
        /* [out] */ IEvent * * ppCompletionEvent);

    virtual void Dispose() {}
};

EXTERN IDriver * CDECL CreateMStatServer(uint_t uDeviceNo, void
*pvParameter)
{
    CMStatServer *pStatServer = new CMStatServer;
    if (NULL == pStatServer) {
        kprintf("ERROR: Not enough memory!\n");
        return NULL;
    }
    pStatServer->AddRef();
    return pStatServer;
}

HRESULT CMStatServer::Control(
        /* [in] */ INT nControlCode,
        /* [in] */ EzByteBuf ebbInData,
        /* [out] */ EzByteBuf ebbOutData,
        /* [out] */ IEvent * * ppCompletionEvent)
{
```

```
#ifdef KCONFIG_MSTAT
    StatInfo statInfo;
    GetStatInfo(&statInfo);
    memcpy((unsigned char *)(char *)ebbOutData, \
           (unsigned char *)&statInfo, sizeof(statInfo));
    ebbOutData.SetUsed(sizeof(statInfo));
#endif //KCONFIG_MSTAT
    return S_OK;
}
```

下面是性能测试工具中用于定时器控制的代码。文件 timerint.h，定义了定时器寄存器、中断控制寄存器等相关的宏，定义定时器中断服务程序函数的原型。

```
#ifndef __TIMERINT_H__
#define __TIMERINT_H__

#include <ddk.h>

typedef unsigned int    UNS_32;
typedef signed int      INT_32;

// Timer Module Register Structure
typedef struct {
    volatile UNS_32    load;      /* RW */
    volatile UNS_32    value;     /* RO */
    volatile UNS_32    control;    /* RW */
    volatile UNS_32    clear;     /* WO */
} TIMERREGS;

// Interrupt Controller Module Register Structure
typedef struct {
        volatile UNS_32             status;
        volatile UNS_32             rawstatus;
        volatile UNS_32             enableset;
        volatile UNS_32             enableclear;
} INTCREGS;

#define _BIT(n)            (((UNS_32)(1)) << (n))

#define TIMER_CTRL_ENABLE      _BIT(7)
#define TIMER_CTRL_DISABLE     (0)
#define TIMER_CTRL_PERIODIC    _BIT(6)
#define TIMER_CTRL_FREERUN     (0)

#define APB_BASE     (0x80000000)
#define TIMER1_BASE   (APB_BASE + 0x0C00)
#define TIMER2_BASE   (APB_BASE + 0x0C20)
#define TIMER3_BASE   (APB_BASE + 0x0C80)
#define INTC_BASE    (APB_BASE + 0x0500)
#define TIMER1       ((TIMERREGS *)(TIMER1_BASE))
#define TIMER2       ((TIMERREGS *)(TIMER2_BASE))
#define TIMER3       ((TIMERREGS *)(TIMER3_BASE))

// Interrupt Controller
#define INTC         ((INTCREGS *)(INTC_BASE))
#define INTC_TC3OINTR_BIT      22

#define TSTTIMES 500
```

```
EXTERN_C INT_32 TstIndex;
EXTERN_C INT_32 EndValue[TSTTIMES];

void TimerIsr(irq_t irq, void *pvDevice, InterruptContext *pContext);

#endif // __TIMERINT_H__
```

文件 timerint.cpp，实现了定时器中断服务，以及将数据结果打印。

```
#include <ddk.h>
#include "timerint.h"

INT_32 TstIndex = 0;
INT_32 EndValue[TSTTIMES];

static void KPrntResult()
{
    int i, j;

    INTC->enableclear |= _BIT(INTC_TC3OINTR_BIT); //disable time3 int
    kprintf("%%\n"); //symbol of starting log to output file
    for (i = 0; i < TSTTIMES; i++) {
        kprintf("value[%4d] = %d\n", i, 65535 - EndValue[i]);
        for(j = 0; j < 200000; j++);
    }
    kprintf("\nlogging result to stdio OK. System reboot...\n\n");
    kprintf("$\n"); //symbol of finishing log to output file
}

void TimerIsr(irq_t irq, void *pvDevice, InterruptContext *pContext)
{
    EndValue[TstIndex] = *(volatile UNS_32 *)(TIMER3_BASE + 0x0004);

    // Clears the TIMER3 int
    *(volatile unsigned int *)(TIMER3_BASE + 0x000C) = 0;

    // TIMER3 reload
    *(volatile UNS_32 *)TIMER3_BASE = 0xFFFF;

    TstIndex++;
    if (TstIndex == TSTTIMES) {
        TstIndex = 0;
        KPrntResult();
    }
}
```

文件 timerserver.cpp，实现了定时器访问的伪驱动。

```
#include <ddk.h>
#include "timerint.h"

// 中断响应延迟测试
#if _PERFORMANCE_TESTING_ARM == 2
#define TST_INTERRUPT_RESPOND_LATENCY
#endif // _PERFORMANCE_TESTING_ARM

enum {
    TC3OI_Irq          = 22,
};
```

```
// TIMER3: 3.6864MHz  271ns/tick  range：271ns ~ 17777506ns
#define TSHM_TIMER1              __32BIT(24)
#define TSHM_TIMER2              __32BIT(25)
#define TSHM_TIMER3              __32BIT(26)

#define IS_TSHM_TIMER1(x)         ((x) & TSHM_TIMER1)
#define IS_TSHM_TIMER2(x)         ((x) & TSHM_TIMER2)
#define IS_TSHM_TIMER3(x)         ((x) & TSHM_TIMER3)

//#define TSHM_ENABLE              __32BIT(23)
//#define TSHM_MODE                __32BIT(22)
//#define TSHM_CLKSEL              __32BIT(19)

#define TSHM_MASK           (__32BIT(16) | __32BIT(17) | __32BIT(18))
#define TSHM_LOAD           0                          // 000
#define TSHM_VALUE          __32BIT(16)                // 001
#define TSHM_CONTROL        __32BIT(17)                // 010
#define TSHM_TCEOI         (__32BIT(16) | __32BIT(17)) // 011
#define TSHM_TCEOI_ON       __32BIT(18)                // 100

#define TSHM_OPERATE_NULL         0x00
#define TSHM_OPERATE_LOAD         0x01
#define TSHM_OPERATE_VALUE        0x02
#define TSHM_OPERATE_CONTROL      0x04
#define TSHM_OPERATE_TCEOI        0x08
#define TSHM_OPERATE_TCEOI_ON     0x10

#define IS_TSHM_OPERATE_LOAD(x)    (((x) & TSHM_MASK) == TSHM_LOAD)
#define IS_TSHM_OPERATE_VALUE(x)   (((x) & TSHM_MASK) == TSHM_VALUE)
#define IS_TSHM_OPERATE_CONTROL(x) (((x) & TSHM_MASK) == TSHM_CONTROL)
#define IS_TSHM_OPERATE_TCEOI(x)   (((x) & TSHM_MASK) == TSHM_TCEOI)
#define IS_TSHM_OPERATE_TCEOI_ON(x) (((x) & TSHM_MASK) == TSHM_TCEOI_ON)

class CTimerServer : public Driver {
public:
    STDMETHODIMP Read(
        /* [in] */ UINT64 u64Offset,
        /* [in] */ UINT uNumberOfBytesToRead,
        /* [out] */ EzByteBuf ebbData,
        /* [out] */ IEvent * * ppCompletionEvent)
    {
        return E_NOTIMPL;
    };

    STDMETHODIMP Write(
        /* [in] */ UINT64 u64Offset,
        /* [in] */ EzByteBuf ebbData,
        /* [out] */ UINT * puNumberOfBytesWritten,
        /* [out] */ IEvent * * ppCompletionEvent);
    {
        return E_NOTIMPL;
    };

   STDMETHODIMP Control(
        /* [in] */ INT nControlCode,
        /* [in] */ EzByteBuf ebbInData,
        /* [out] */ EzByteBuf ebbOutData,
        /* [out] */ IEvent * * ppCompletionEvent);
```

装 订 线

```
    virtual void Dispose()  {}
private:
    TIMERREGS*  m_pTimer;
};

EXTERN IDriver * CDECL CreateTimerServer(uint_t uDeviceNo, void
*pvParameter)
{
    CTimerServer* pTimerServer = new CTimerServer;
    if (pTimerServer == NULL) {
        kprintf("ERROR: Not enough memory!\n");
        return NULL;
    }
    pTimerServer->AddRef();
#ifdef TST_INTERRUPT_RESPOND_LATENCY
    RegisterIsr(TC3OI_Irq, IPL10, TimerIsr, NULL);
#endif // TST_INTERRUPT_RESPOND_LATENCY
    return pTimerServer;
}

HRESULT CTimerServer::Control(
        /* [in] */ INT nControlCode,
        /* [in] */ EzByteBuf ebbInData,
        /* [out] */ EzByteBuf ebbOutData,
        /* [out] */ IEvent * * ppCompletionEvent)
{
    m_pTimer = NULL;
    if (IS_TSHM_TIMER1(nControlCode)) {   m_pTimer = TIMER1;  }
    if (IS_TSHM_TIMER2(nControlCode)) {   m_pTimer = TIMER2;  }
    if (IS_TSHM_TIMER3(nControlCode)) {   m_pTimer = TIMER3;  }
    if (m_pTimer == NULL)             {  return E_INVALIDARG;}

    if (IS_TSHM_OPERATE_LOAD(nControlCode)) {
        // set initial timer value
        m_pTimer->load = (nControlCode & 0xffff);
        return S_OK;
    }
    if (IS_TSHM_OPERATE_VALUE(nControlCode)) {
        // get current timer value
        if (ebbOutData.IsNull()) {
            return E_INVALIDARG;
        }
        unsigned int value = m_pTimer->value;
        value &= 0xffff;
        memcpy((unsigned char *)(char *)ebbOutData, \
               (unsigned char *)&value, sizeof(value));
        ebbOutData.SetUsed(sizeof(value));
        return S_OK;
    }
    if (IS_TSHM_OPERATE_CONTROL(nControlCode)) {
        m_pTimer->control = (nControlCode >> 16) & 0xc8;
        return S_OK;
    }
    if (IS_TSHM_OPERATE_TCEOI(nControlCode)) {
        m_pTimer->clear = 0;
        return S_OK;
    }

#ifdef TST_INTERRUPT_RESPOND_LATENCY
```

装 订 线

```
    if (IS_TSHM_OPERATE_TCEOI_ON(nControlCode)) {
        TIMERREGS * timer = TIMER3;
        timer->control = 0;
        timer->load = 0;
        INTC->enableclear |= _BIT(INTC_TC3OINTR_BIT); //disable time3 int
        timer->clear = 0;
        timer->control |= TIMER_CTRL_PERIODIC; //set periodic mode
        INTC->enableset |= _BIT(INTC_TC3OINTR_BIT); //enable time3 int
        timer->load = 0xffff;
        timer->control |= TIMER_CTRL_ENABLE; //start timer
        return S_OK;
    }
#endif // TST_INTERRUPT_RESPOND_LATENCY
    return E_INVALIDARG;
}
```

性能测试的代码在测试的方式上都是按统一的模式进行的，因为代码量很大同时这些代码结构非常相似，限于篇幅，下面只给出互锁函数 InterlockedIncrement 的性能测试代码。

```
#include <stdio.h>
#include <elastos.h>
#include "timing.h"

static int  nLoopCount = 100;
static long base = 0;

HRESULT __stdcall dowork(void *arg)
{
    for (int i = 0; i < TSTTIMES; i++) {
        Start();                    // 开始计时
        for (int j = 0; j < nLoopCount; j++) {
            InterlockedIncrement((PLONG)(ULONG)&arg);
        }
        Stop();                     // 停止计时
    }
    PrntResult(L"SI-g-1-C.pfm");  // 输出测试数据
    return 0;
}

int __cdecl main()
{
    IThread* pThread;
    TimerServerUp(nLoopCount);     // 启动计时服务
    HRESULT hr = EzCreateThread(dowork, (PVOID)base, 0, &pThread);
    if (FAILED(hr)) {
        printf("createThread failed\n");
        goto EXIT;
    }
    pThread->Join(INFINITE);
    pThread->Release();
EXIT:
    TimerServerDown();           // 停止计时服务
    return 0;
}
```