如何构造嵌入式 Linux 系统

黄敦

编者备注：根据众多零散资料整理。

特别鸣谢广州博利思软件公司提供 POCKET IX 系统及参考资料。

E_mail:dunn@163.net

二〇〇一年二月十三日

目录

什么是嵌入式系统

嵌入式系统被定义为：以应用为中心、以计算机技术为基础、软件硬件可裁剪、适应应用系统对功能、可靠性、成本、体积、功耗严格要求的专用计算机系统。

嵌入式计算机在应用数量上远远超过了各种通用计算机，一台通用计算机的外部设备中就包含了 5-10 个嵌入式微处理器，键盘、鼠标、软驱、硬盘、显示卡、显示器、网卡、Modem、声卡、打印机、扫描仪、数字相机、USB 集线器等均是由嵌入式处理器控制的。在制造工业、过程控制、通讯、仪器、仪表、汽车、船舶、航空、航天、军事装备、消费类产品等方面均是嵌入式计算机的应用领域。

嵌入式系统是将先进的计算机技术、半导体技术和电子技术和各个行业的具体应用相结合后的产物，这一点就决定了它必然是一个技术密集、资金密集、高度分散、不断创新的知识集成系统。

Linux 提供了完成嵌入功能的基本的内核和你所需要的所有用户界面，它是多面的。它能处理嵌入式任务和用户界面。将 Linux 看作是连续的统一体，从一个具有内存管理、任务切换和时间服务及其他的分拆的、微内核到完整的服务器，支持所有的文件系统和网络服务。

一个小型的嵌入式 Linux 系统只需要下面三个基本元素：

- 引导工具
- Linux 微内核，由内存管理、进程管理和事务处理构成
- 初始化进程

如果要让它能干点什么且继续保持小型化，还得加上：

- 硬件驱动程序
- 提供所需功能的一个或更多应用程序。

再增加功能，或许需要这些

- 一个文件系统（也许在 ROM 或 RAM）中
- TCP / IP 网络堆栈

存储半过渡数据和交换用的磁盘。

嵌入式系统的特征

1. 硬件

嵌入式系统是面向用户、面向产品、面向应用的，如果独立于应用自行发展，则会失去市场。嵌入式处理器的功耗、体积、成本、可靠性、速度、处理能力、电磁兼容性等方面均受到应用要求的制约，这些也是各个半导体厂商之间竞争的热点。 和通用计算机不同，嵌入式系统的硬件和软件都必须高效率地设计，量体裁衣、去除冗余，力争在同样的硅片面积上实现更高的性能，这样才能在具体应用对处理器的选择面前更具有竞争力。嵌入式处理器要针对用户的具体需求，对

芯片配置进行裁剪和添加才能达到理想的性能；但同时还受用户订货量的制约。因此不同的处理器面向的用户是不一样的，可能是一般用户，行业用户或单一用户。 嵌入式系统和具体应用有机地结合在一起，它的升级换代也是和具体产品同步进行，因此嵌入式系统产品一旦进入市场，具有较长的生命周期。嵌入式系统中的软件，一般都固化在只读存储器中，而不是以磁盘为载体，可以随意更换，所以嵌入式系统的应用软件生命周期也和嵌入式产品一样长。另外，各个行业的应用系统和产品，和通用计算机软件不同，很少发生突然性的跳跃，嵌入式系统中的软件也因此更强调可继承性和技术衔接性，发展比较稳定。嵌入式处理器的发展也体现出稳定性，一个体系一般要存在 8-10 年的时间。一个体系结构及其相关的片上外设、开发工具、库函数、嵌入式应用产品是一套复杂的知识系统，用户和半导体厂商都不会轻易地放弃一种处理器。

2. 系统软件和应用软件

嵌入式处理器的应用软件是实现嵌入式系统功能的关键，对嵌入式处理器系统软件和应用软件的要求也和通用计算机有所不同。

(1) 软件要求固态化存储：为了提高执行速度和系统可靠性，嵌入式系统中的软件一般都固化在存储器芯片或单片机本身中，而不是存贮于磁盘等载体中。

(2) 软件代码高质量、高可靠性：尽管半导体技术的发展使处理器速度不断提高、片上存储器容量不断增加，但在大多数应用中，存储空间仍然是宝贵的，还存在实时性的要求。为此要求程序编写和编译工具的质量要高，以减少程序二进制代码长度、提高执行速度。

(3) 系统软件(OS)的高实时性是基本要求：在多任务嵌入式系统中，对重要性各不相同的任务进行统筹兼顾的合理调度是保证每个任务及时执行的关键，单纯通过提高处理器速度是无法完成和没有效率的，这种任务调度只能由优化编写的系统软件来完成，因此系统软件的高实时性是基本要求。

(4) 多任务操作系统是知识集成的平台和走向工业标准化道路的基础。

Linux 作为嵌入式操作系统的优势

1. 免许可证费用

大多数的商业操作系统，例如 Windows、Windows CE 对每套操作系统收取一定的许可证费用。相对地，Linux 是一个免费软件，并且公开源代码。只要你不违反 GPL 协议，你可以自由应用和发布 Linux。

2. 有很高的稳定性

在 PC 硬件上运行时，Linux 是非常可靠和稳定的，特别是和现在流行的一些操作系统相比。嵌入式内核本身有多稳定呢？对大多数微处理器来说，Linux 非常好。移植到新微处理器家族的 Linux 内核运行起来与本来的微处理器一样稳定。它经常被移植到一个或多个特定的主板上。这些板包括特定的外围设备和 CPU。

幸运的是，许多不同处理器的指令代码是相通的，所以移植集中在差异上。其中大多数是在内存管理和中断控制领域。一旦成功移植，它们就非常稳定。

根据大部分国内外使用者的经验，Linux 至少和许多著名的商业性操作系统一样稳定。总之，这些操作系统和 Linux 的问题在于对工作过程微秒之处的误解，而不在于代码的难度或基本的设计错误。任何操作系统都有很多争论不休的故事，这里不需要重复。Linux 的优势在于源代码是公开、注释清晰和文档齐全的。这样，你就可以控制和处理所出现的任何问题。

不过仍然有两个因素会影响稳定性，一是使用了混乱的驱动程序。驱动程序的选择很有限，有些稳定有些不稳定。一旦你离开了通用的 PC 平台，你需要自己编写。幸运的是，周围有许多驱动程序，你可能可以找到一个与你的需求相近的修改一下。这种驱动程序界面已定义好。许多类的驱动程序都非常相近，所以把磁盘、网络或一系列的端口驱动程序从一个设备移植到另一个设备上通常并不难。现在许多驱动程序都写得很好，很容易理解，但你还是要准备一本关于内核结构的书在手头。二是使用了硬盘。文件系统的可靠性就成问题。我们有用磁盘进行 Linux 系统设计超过两年的经验。这些系统几乎从未正常关闭过。电源随时都可能被中断。感觉非常好，使用的是标准（EXT2）文件系统。标准 Linux 初始化脚本运行 fsck 程序，它在检查和清除不稳定的 inodes 方面非常有效。将默认的每隔 30 秒运行更新程序改为每隔 5 或 10 秒运行是比较明智的。这样缩短了数据在进入磁盘之前，待在高速缓冲存储器内的时间，降低了丢失数据的可能性。

3. 强大的网络功能

Linux 天生就是一个网络操作系统，几乎所有的网络协议和网络接口都已经被定制在 Linux 中。Linux 内核在处理网络协议方面比标准的 Unix 更具执行效率，在每一个端口上有更高的吞吐量。

4. 丰富的开发工具

Linux 提供 C、C++、JAVA 以及其他很多的开发工具。更重要的是，爱好者可以免费获得。并且这些开发工具设计时已经考虑到支持各种不同的微处理器结构和调试环境。Linux 基于 GNU 的工具包，此工具包提供了完整与无缝交叉平台开发工具，从编辑器到底层调试。其 C 编译器产生更有效率的执行代码。

5. 大量的文档

对新手来说，有很多用户界面友好的参考文档，这些资料很容易从网上获得。对处于黄金时期的 Linux 来说，许多书店都愿意在书架上放上这方面的书籍。

规划一个好的硬件平台

1. 选择一个好的嵌入式微处理器

嵌入式微处理器的基础是通用计算机中的 CPU。在应用中，将微处理器装配在专门设计的电路板上，只保留和嵌入式应用有关的母板功能，这样可以大幅度减小系统体积和功耗。为了满足嵌入式应用的特殊要求，嵌入式微处理器虽然

在功能上和标准微处理器基本是一样的，但在工作温度、成本、功耗、可靠性、健壮性等方面和工业控制计算机相比，嵌入式微处理器具有体积小、重量轻、成本低、可靠性高的优点，但是在电路板上必须包括 ROM、RAM、总线接口、各种外设等器件，从而降低了系统的可靠性，技术保密性也较差。嵌入式微处理器及其存储器、总线、外设等安装在一块电路板上，称为单板计算机。如 STD-BUS、PC104 等。嵌入式处理器目前主要有国半 X86、Dragon Ball、ARM、Strong ARM、Power PC、68000、MIPS 系列等。

2. 规划通讯技术

PCI 方案：在高速通信中一个重要的因素是嵌入装置如何快速地传输数据而不涉及 CPU。在很多低功率手持产品中，基本的 I/O 设备是与主处理器集成在一起的，不需要主 CPU 总线扩展。但大多数新的设计不仅需要基本的 I/O 设备，而且很多都采用广泛应用 PC 机标准以便主 CPU 总线扩展，即 PCI（外设部件互连）总线。PCI 总线工作频率为 33MHz(rev2.1 支持 66MHz)，对于连接到它上面的器件是具有即插即用能力。Compact PCI(PCI 总线的一种)正在进入工业和通信市场。PC104/104+基本上分别为 PC ISA 和 PCI 总线的改进型。PC104 总线与 ISA 总线完全兼容的。这些总线的出现有助于 PCI 技术进入嵌入领域。

IrDA/FastIrDA:红外数据联盟（IrDA）是一个由 150 多个公司组成的联合体。IrDA 提供一种价廉的无线、点到点、双向红外通信技术。它旨在用于小于 1 米的极短距离通信。IrDA 有两个速度：低速运行于 9.6～115kits/s（简称 IrDA）；高速运行于 1～4Mbits/s（即 Fast IrDA）。高达 16M bits/s 的更高速度的正在开发。IrDA 用于笔记本计算机、PDA、打印机、照相机等产品中。其他产品如复印机、投影机和游戏控制等也正在考虑采用。

USB: 通用串行总线（USB）是由 IBM、Compaq、Nortel、NEC、Intel 和 Microsoft 公司开发的一种外设总线标准。它为所有 USB 外设提供一种通用的连接，其数据率为 12Mbits/s。USB 缆线是为适用于短距离（最长 5 米）而设计的。连接遵从树拓扑结构，在任何时间可连接 127 个器件而外设可以是带电交换的。USB 缆线也把功率（+5V）分配给低功率外设。它为不能处理瞬间传输、又需要保证带宽和有限执行时间的应用提供同步通信。同步工作量可以是 USB 总线带宽的一部分或全部。USB 特别适合于需要高数据率和易于即插即用的应用，如调制解调器、游戏控制、打印机、扫描仪和数字相机。需要保证带宽和有限执行时间的应用包括 PC 电话和其他语音及视频通信应用。除了这些新的多媒体设备外，USB 也用于传统的 I/O 设备，如键盘和鼠标，其处理速度为低速（1.5Mbits/s）。Windows CE 为 USB 提供支持。USB 的系统软件由两部分组成：USBD（通用串行总线驱动器）和 HCD（主控制器驱动器）。USBD 由 Microsoft 提供而用 USB 器件驱动器实现高级功能。HCD 模件提供到实际硬件（OHCD 开路主控制器驱动器或 UHCD 通用主控制器驱动器）的接口。

Ethernet/Fast Ethernet: Ethernet(以太网)和 Fast Ethernet(IEEE 802.3 和 802.3n)是最广泛应用的局域网络技术，旨在小区域（即一个办公室）范围连接计算机。Ethernet 工作在 10Mbits/s 而 Fast Ethernet 工作在 100M bits/s。两个协议的差别限于物理层和通信媒体。媒体存取规则是 CSMA/CD（载波检测多路存取/冲突检测）。Windows CE 通过其 NDIS 4.0 实现支持 IEEE802.3 小口驱动器。Ethernet 卡可以在平台上或通过一个 PCMCIA 槽进行热插拔。

IEEE1394:IEEE1394 是高速串行总线，其数据率为 25～400Mbits/s。它起源于 Apple Computer 的 FireWire 总线，是作为通用外设串行总线而设计的，但它的应用重点转为所有类型的消费类设备如数字相机和扫描仪。缆线型 1394 总线可支持 63 个器件。器件之一变成总线管理者，与其他器件协调之后管理总线执行。缆线越长它所能够处理的数据率就越低。一般长度为几米。IEEE1394 和 USB 都是串行协议，然而 USB 和 IEEE1394 比其竞争技术有更大的互补性，USB 属于低到中带宽，而 IEEE1394 属于中到高带宽。

3. 适合嵌入适Linux开发调试工具

开发嵌入式系统的关键的是可用的工具包。像任何工作一样，好的工具使得工作更快更好。开发的不同阶段需要不同的工具。

传统上，首先用于开发嵌入式系统工具是内部电路仿真器（ICE），它是一个相对昂贵的部件，用于植入微处理器与总线之间的电路中，允许使用者监视和控制微处理器所有信号的进出。这有点难做，因为它是异体，可能会引起不稳定。但是它提供了总线工作的清晰状况，免了许多对硬件软件底层工作状况的猜测。

过去，一些工作依赖 ICE 为主要调试工具，用于整个开发过程。但是，一旦初始化软件对串口支持良好的话，多数的调试可以不用 ICE 而用其他方法进行。较新的嵌入式系统利用非常清晰的微处理器设计。有时，相应工作初始码已经有了能够快速获得串口工作。这意味着没有 ICE 人们也能够方便地工作。省去 ICE 降低了开发的成本。一旦串口开始工作，它可以支持各种专业开发工具。

Linux 是基于 GNU 的 C 编译器，作为 GNU 工具链的一部分，与 gdb 源调试器一起工作。它提供了开发嵌入式 Linux 系统的所有软件工具。这有些典型的、用于在新硬件上开发嵌入式 Linux 系统的调试工具。

1. 写入或植入引导码
2. 向串口打印字符串的编码，如“Hello World”（事实上我更喜欢“Watson，Come hre I need you”，电话上常用的第一个词。）
3. 将 gdb 目标码植入工作串口，这可与另一台运行 gdb 程序的 Linux 主机系统对话。只要简单地告诉 gdb 通过串口调试程序。它通过串口与测试机的 gdb 目标码对话，你可以进行 C 源代码调试，也可以用这个功能将更多的码载入 RAM 或 Flash Memory 中。
4. 利用 gdb 让硬件和软件初始化码在 Linux 内核启动时工作。
5. 一旦 Linux 内核启动，串口成为 Linux 控制口并可用于后续开发。利用 kgdb，内核调试版的 gdb，这步常常不作要求，如果你与网络联接，如 10BaseT,下一步你可能要启动它。

如果在你的目标硬件上运行了完整的 Linux 内核，你可以调试你的应用进程。利用其他的 gdb 或覆盖 gdb 的图形如 xgdb。

构造嵌入式 Linux 前先要了解的几个关键问题

1. 如何引导?

当一个微处理器第一次启动的时候，它开始在预先设置的地址上执行指令。通常在那里有一些只读内存，包括初始化或引导代码。在 PC 上，这是 BIOS。

它执行了一些低水平的 CPU 初始化和其它硬件的配置。BIOS 继续辨认哪个磁盘里有操作系统，把操作系统复制到 RAM 并且转向它。实际上，这非常复杂，但对我们的目标来说也非常重要。在 PC 上运行的 Linux 依靠 PC 的 BIOS 来提供这些配置和 OS 加载功能。

在一个嵌入式系统里经常没有这种 BIOS。这样你就要提供同等的启动代码。幸运的是，嵌入式系统并不需要 PC BIOS 引导程序那样的灵活性，因为它通常只需要处理一个硬件的配置。这个代码更简单也更枯燥。它只是一指令清单，将固定的数字塞到硬件寄存器中去。然而，这是关键的代码，因为这些数值要与你的硬件相符而且要按照特定的顺序进行。所以在大多数情况下，一个最小的通电自检模块，可以检查内存的正常运行、让 LED 闪烁，并且驱动其它必须的硬件以使主 Linux OS 启动和运行。这些启动代码完全根据硬件决定，不可随意移动。

幸运的是，许多系统都有为核心微处理器和内存所定制的菜单式硬件设计。典型的是，芯片制造商有一个样本主板，可以用来作为设计的参考——或多或少与新设计相同。通常这些菜单式设计的启动代码是可以获得的，它可以根据你的需要轻易的修改。在少数情况下，启动代码需要重新编写。

为了测试这些代码，你可以使用一个包含‘模拟内存’的电路内置模拟器，它可以代替目标内存。你把代码装到模拟器上并通过模拟器调试。如果这样不行，你可以跳过这一步，但这样就要一个更长的调试周期。

这个代码最终要在较为稳定的内存上运行，通常是 Flash 或 EPROM 芯片。你需要使用一些方法将代码放在芯片上。怎么做，要根据“目标”硬件和工具来定。

一种流行的方法是把 Flash 或 EPROM 芯片插入 EPROM 或 Flash 烧制器。这将把你的程序“烧”（存）入芯片。然后，把芯片插入你的目标板的插座，打开电源。这个方法需要板上配有插座，但有些设备是不能配插座的。

另一个方法是通过一个 JTAG 界面。一些芯片有 JTAG 界面可以用来对芯片进行编程。这是最方便的方法。芯片可以永远被焊在主板上，一个小电缆从板上的 JTAG 连接器，通常是一个 PC 卡，联到 JTAG 界面。下面是 PC 运行 JTAG 界面所需的一些惯用程序。这个设备还可以用来小量生产。

2. 需要虚拟内存吗?

标准 Linux 的具备虚拟内存的能力。正是这种神奇的特征使应用程序员可以狂热的编写代码而不计后果，不管程序有多大。程序溢出到了磁盘交换区。在没有磁盘的嵌入式系统里，通常不能这么做。

在嵌入式系统里不需要这种强大的功能。实际上，你可能不希望它在实时的关键系统里，因为它会带来无法控制的时间因素。这个软件必须设计得更加精悍，以适合市面上物理内存，就象其它嵌入式系统一样。

注意由于CPU的原因，我认为在嵌入式Linux中保存虚拟内存代码是明智的，因为将它清除很费事。而且还有另外一个原因——它支持共享文本，这样就可以使许多程序共享一个软件。没有这个，每一个程序都要有它自己的库，就象 printf 一样。

虚拟内存的调入功能可以被关掉，只要将交换空间的大小设置为零。然后，如果你写的程序比实际的内存大，系统就会当作你的运行用尽了交换空间来处理；这个程序将不会运行，或者 malloc 将会失灵。

在许多 CPU 上，虚拟内存提供的内存管理可以将不同程序分开，防止它们

写到其它地址的空间上。这在嵌入式系统上通常不可能，因为它只支持一个简单、扁平的地址空间。Linux 的这种功能有助于其发展。它减少了胡乱的编写程序造成系统崩溃的可能性。许多嵌入式系统基于效率方面的原因有意识使用程序间可以共享的“全局”数据。这也可以通过 Linux 共享内存功能来支持，共享的只是指定的内存部分。

3. 选用什么样的文件系统?

许多嵌入式系统没有磁盘或者文件系统。Linux 不需要它们也能运行。在这种情况下，应用程序任务可以和内核一起编写，并且在引导时作为一个影像加载。对于简单的系统来说，这就够了。然而，它缺乏灵活性。

实际上，许多商业性嵌入式系统，提供文件系统作为选项。许多或者是专用的文件系统或者是 MS-DOS-Compatible 文件系统。Linux 提供 MS-DOS-Compatible 文件系统，同时还有其它多种选择。之所以提供其它选择是因为它们更加强大而且具有容错功能。Linux 还具有检查和维护的功能，商业性供应商往往不提供这些。这对于 Flash 系统来说尤其重要，因为它是通过网络更新的。如果系统在升级过程中失去了能力，那它就没有用了。维护的功能通常可以解决这类问题。

文件系统可以被放在传统的磁盘驱动器、Flash Memory 或其它这类的介质上。而且，用于暂时保存文件，一个小 RAM 盘就足够了。

Flash Memories 被分割成块。这些块中也许包括一个含有当 CPU 启动时运行的最初的软件的引导块。这可能包括 Linux 引导代码。剩余的 Flash 可以用作文件系统。Linux 的内核可以通过引导代码从 Flash 复制到 RAM，或者还有一个选择，内核可以被存储在 Flash 的一个独立部分，并且直接从那里执行。

另外对于一些系统来说还有一个有趣的选择，那就是将一个便宜的 CD-ROM 包含在内。这比 Flash Memory 便宜，而且通过交换 CD-ROM 支持简单的升级。有了这个，Linux 只要从 CD-ROM 上引导，并且象从硬盘上一样从 CD-ROM 上获得所有的程序。

最后，对于联网的嵌入式系统来说，Linux 支持 NFS(Network File System)。这为实现联网系统的许多增值功能打开了大门。第一，它允许通过网络上加载应用程序。这是控制软件修改的基础，因为每一个嵌入式系统的软件都可以在一个普通的服务器上加载。它在运行的时候也可以用来输入或输出大量的数据、配置和状态信息。这对用户监督和控制来说是一个非常强大的功能。举例来说，嵌入式系统可以建立一个小的 RAM 磁盘，包含的文件中有与当前状态信息同步的内容。其它系统可以简单的把这个 RAM 磁盘设置为基于网络的远程磁盘，并且空中存取状态文件。这就允许另一个机器上的 Web 服务器通过简单的 CGI Script 存取状态信息。在其它电脑上运行的其它应用程序包可以很容易的存取数据。对更复杂的监控，应用程序包如 Matlab(http://www.mathworks.com/products/matlab/)，可以用来在操作员的 PC 或工作站的提供系统运行的图形展示。

4. 如何消除嵌入式Linux系统对磁盘的依赖

对于 Linux 一个共同的认识是它用于嵌入式系统简直是神奇极了。这可能不大对，典型的 PC 上的 Linux 对 PC 用户来说功能有多。

对初学者而言，可以将内核与任务分开，标准的 Linux 内核通常驻留在内存

中，每一个应用程序程序都是从磁盘运到内存上执行。当程序结束后，它所占用的内存就被释放，程序就被下载了。

在一个嵌入式系统里，可能没有磁盘。有两种途径可以消除对磁盘的依赖，这要看系统的复杂性和硬件的设计。

在一个简单的系统里，当系统启动后，内核和所有的应用程序都在内存里。这就是大多数传统的嵌入式系统工作模式，它同样可以被 Linux 支持。

有了 Linux，就有了第二种可能性。因为 Linux 已经有能力“加载”和“卸载”程序，一个嵌入式系统就可以利用它来节省内存。试想一个典型的包括一个大概 8MB 到 16MB 的 Flash Memory 和 8MB 内存的系统。Flash Memory 可以作为一个文件系统。Flash Memory 驱动程序用来连接 Flash Memory 和文件系统。作为替代，可使用 Flash Disk。这 Flash 部件用软件仿真磁盘。其中一个例是 M-Systems 的 DiskOnChip，可以达到 160MB。（http://www.m-systems.com）。所有的程序都以文件形式存储在 Flash 文件中，需要时可以装入内存。这种动态的、“根据需要加载”的能力是支持其它一系列功能的重要特征：

- 它使初始化代码在系统引导后被释放。Linux 同样有很多内核外运行的公用程序。这些通常程序在初始化时运行一次，以后就不再运行。而且，这些公用程序可以用它们相互共有的方式，一个接一个按顺序运行。这样，相同内存空间可以被反复使用以“召入”每一个程序，就象系统引导一样。这的确可以节省内存，特别是那些配置一次以后就不再更改的网络堆栈
- 如果 Linux 可加载模块的功能包括在内核里，驱动程序和应用程序就都可以被加载。它可以检查硬件环境并且为硬件装上相应的软件。这就消除了用一个程序占用许多 Flash Memory 来处理多种硬件的复杂性。
- 软件的升级更模块化。你可以在系统运行的时候在 Flash 上升级应用程序和可加载驱动程序。
- 配置信息和运行时间参数可以作为数据文件储存在 Flash 上。

5. 嵌入式Linux达到怎样的实时性？

实时的含义是指在规定的时限内能够传递正确的结果，迟到的结果就是错误。实时系统并非是指“快速”的系统，实时系统有限定的响应时间，从而使系统具有可预测性。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（如航空航天系统）；而后者则在不满足响应时限时，系统性能退化，但并不会导致灾难性的后果（如交换系统）。

在嵌入式领域中，实时并非是最重要的。嵌入式系统常常被错误地分为实时系统，尽管多数系统一般并不要求实时功能。如上文所述，实时是一个相对的词，常常被严格地定义实时为对一事件以预定的方式在极短的时间如微秒作出响应，渐渐地，在如此短暂时间间隔内的严格实时功能在专用 DSP 芯片或 ASIC 上实现了。只有在设计低层硬件 FIFO、分散 / 聚集 DMA 引擎和定制硬件时才会有这样的要求。

许多设计人员因为对真实的要求设有清晰的理解而对实时的要求焦虑不安。对于大多数的系统，在一至五微秒的近似实时响应已经足够。同样的软件需求也是可以接受的。如 Windows 98 已经崩溃的中断必须在 4 毫秒内（±98％）内、或 20 毫秒（±0）内进行处理。这种软要求是比较容易满足的，包括环境转换时

间、中断等待时间、任务优先级和排序。环境转换时间曾是操作系统的一个热门话题。总之，多数 CPU 这些要求处理得很好，而且 CPU 的速度现在已经快了很多，这个问题也就不重要了。

严格的实时要求通常由中断例程或其他内核环境驱动程序功能处理，以确保稳定的表现，等待时间，一旦请求出现要求服务的时间很大程度上取决于中断的优先及其他能暂时掩盖中断的软件。

中断必须进行处理和管理以确保时间要求能符合，如同许多其他的操作系统。在 Intel X86 处理器中，这工作很容易由 Linux 实时扩展处理。实时扩展处理提供了一个以后台任务方式运行 Linux 的中断处理调度。关键的中断响应不必通知 Linux。因此可以得到许多对于关键时钟的控制。在实时控制级和时间限制宽松的基本 Linux 级之间提供接口，这提供了与其他嵌入式操作系统相似的实时框架。因此，实时关键代码是隔开的、并“设计”成满足要求的。代码处理的结果是以更一般的方法也许只在应用任务级。

开发嵌入式 Linux 的关键步骤

1. 精简内核

构造内核常用命令包括：make config, dep, clean, mrproper, zImage, bzImage, modules, modules_install

(1) make config

核心配置，调用./scripts/Configure 按照 arch/i386/config.in 来进行配置。命令执行完后产生文件.config，其中保存着配置信息。下一次再做 make config 将产生新的.config 文件，原.config 被改名为.config.old

(2)make dep

寻找依存关系。产生两个文件.depend 和.hdepend 其中.hdepend 表示每个.h 文件都包含其它哪些嵌入文件。而.depend 文件有多个，在每个会产生目标文件(.o)文件的目录下均有，它表示每个目标文件都依赖哪些嵌入文件(.h)。

(3)make clean

清出以前构核所产生的所有目标文件、模块文件、核心以及一些临时文件等，不产生任何文件

(4)make rmproper

删除所有因构核过程中产生的所有文件，及除了做 make clean 外，还要删除.config，.depend 等文件，把核心源码恢复到最原始的状态。下次构核时就必须重新配置了。

(5)make, make zImage, make bzImage

make:构核。通过各目录的 Makefile 文件进行。会在各个目录下产生一大堆目标文件，若核心代码没有错误，将产生文件 vmlinux，这就是所构的核心。并产生映射文件 System.map 通过各目录的 Makefile 文件进行。.version 文件中的数加 1，表示版本号（又产生一个新的版本了），让你明白，你已经对核心改动过多少次了。

Make zImage：在 make 的基础上产生压缩的核心映象文件./arch/$(ARCH)/boot/zImage 以及在./arch/$(ARCH)/boot/compresed/目录

下产生一些临时文件。

Make bzImage:在 make 的基础上产生压缩比例更大的核心映象文件./arch/$(ARCH)/boot/bzImage 以及在./arch/$(ARCH)/boot/compresed/目录下产生一些临时文件。在核心太大时进行。

(6)make modules

编译模块文件，你在 make config 时所配置的所有模块将在这时编译，形成模块目标文件，并把这些目标文件存放在 modules 目录中。使用如下命令看一看。Ls modules

(7)make modules_install

把上面编译好的模块目标文件目录/lib/modules/$KERNEL_VERSION/ 中。

比如我的版本是 2.0.36，做完这个操作后可使用下面的命令看看：ls /lib/modules/2.0.36/

相关的命令还有很多，有兴趣可看相关资料和 Makefile 文件。另外注意，这儿我们产生了一些隐含文件

.config

.oldconfig

.depend

.hdepend

.version

它们的意义应该很清楚了。

废话说了一堆，现在还是举个例子说明一下：

我使用的是 Mandrake 内付的 2.2.15，我没有修改任何一行程序码，完全只靠修改组态档得到这些数据。

首先，使用 make xconfig 把所有可以拿掉的选项都拿得。

不要 floppy；不要 SMP,MTRR；不要 networking,SCSI；把所有的 block device 移除，只留下 old IDE device；把所有的 character device 移除；把所有的 filesystem 移除，只留下 minix；不要 sound 支援。相信我，我已经把所有的选项都移除了。这样做之后，我得到了一个 188K 的核心。

还不够小吗?OK，再加上一招，请把下列二个档案中的 -O3,-O2 用 -Os 取代。

./Makefile

./arch/i386/kernel/Makefile

这样一来，整个核心水小了 9K，成为 179K。

不过这个核心恐怕很难发挥 Linux 的功能，因此我决定把网络加回去。把 General 中的 network support 加回去，重新编译，核心变成 189 K。10K 换个 TCP/IP stack，似乎是很上算的生意。

不过有 stack 没有 driver 也是惘然，所以我把 embedded board 常用的 RTL8139 的 driver 加回去，195K。

如果你需要 DOS 档案系统，那大小成为 213K。如果 minix 用 ext2 换代，则大小成长至 222K。

不过大家要注意，那里的大小指的是核心档的大小。那和所需要的随取记忆体是二回事。这个数字代表的意义是你需要多小的 ROM 来存放你的核心。

Linux 所需的记忆体大约在 600~800 K 之间。1MB 可能可以开机了，但可能不太有用。因为可能连载入 C 程序库都有困难。2MB 应该就可以做点事了，

但可能要到 4MB 以上才可以执行一个比较完整的系统。

因为 Linux 的 filesystem 相当的大。大约在 230K 左右，占了 1/3 的体积。记忆体管理占了 80K，和核心其它部份的总合差不多。TCP/IP stack 占了 65K，驱动程序占了 120K。SysV IPC 占了 21K，必要的话可以拿掉，核心档应该可以再小个 10K 左右。

所以如果要减核心大小，应该动那里呢? 答案应该很明显，当然是文件系统。Linux 的 VFS 减化了档案系统的设计，buffer cache, directory cache 增加了系统的效率。但这些对整个系统都在 flash 上的 embedded 系统而言根本就用处不大。如果可以把它们对拿掉，核心可以马上缩小 20K 左右。如果跳过整个 VFS，直接将文件系统写成一个 driver 的型式，应该可以将 230K 缩减至 50K 左右。整个核心缩到 100K 左右。

2. 系统启动

系统的启动顺序及相关文件仍在核心源码目录下，看以下几个文件

./arch/$ARCH/boot/bootsect.s

./arch/$ARCH/boot/setup.s

./init/main.c

bootsect.S 及 setup.S

这个程序是 linux kernel 的第一个程序，包括了 linux 自己的 bootstrap 程序，但是在说明这个程序前，必须先说明一般 IBM PC 开机时的动作(此处的开机是指"打开 PC 的电源"):

一般 PC 在电源一开时，是由内存中地址 FFFF:0000 开始执行(这个地址一定在 ROM BIOS 中，ROM BIOS 一般是在 FE000h 到 FFFFFh 中)，而此处的内容则是一个 jump 指令，jump 到另一个位於 ROM BIOS 中的位置，开始执行一系列的动作，包括了检查 RAM，keyboard，显示器，软硬磁盘等等，这些动作是由系统测试代码(system test code)来执行的，随着制作 BIOS 厂商的不同而会有些许差异，但都是大同小异，读者可自行观察自家机器开机时，萤幕上所显示的检查讯息。

紧接着系统测试码之后，控制权会转移给 ROM 中的启动程序(ROM bootstrap routine)，这个程序会将磁盘上的第零轨第零扇区读入内存中(这就是一般所谓的 boot sector，如果你曾接触过电脑病毒，就大概听过它的大名)，至於被读到内存的哪里呢? --绝对位置 07C0:0000(即 07C00h 处)，这是 IBM 系列 PC 的特性。而位在 linux 开机磁盘的 boot sector 上的正是 linux 的 bootsect 程序，也就是说，bootsect 是第一个被读入内存中并执行的程序。现在，我们可以开始来看看到底 bootsect 做了什么。

第一步

首先，bootsect 将它"自己"从被 ROM BIOS 载入的绝对地址 0x7C00 处搬到 0x90000 处，然后利用一个 jmpi(jump indirectly)的指令，跳到新位置的 jmpi 的下一行去执行，

第二步

接着，将其他 segment registers 包括 DS，ES，SS 都指向 0x9000 这个位置，与 CS 看齐。另外将 SP 及 DX 指向一任意位移地址(offset)，这个地址等一下会用来存放磁盘参数表(disk para- meter table)

第三步

接着利用 BIOS 中断服务 int 13h 的第 0 号功能，重置磁盘控制器，使得刚才的设定发挥功能。

第四步

完成重置磁盘控制器之后，bootsect 就从磁盘上读入紧邻着 bootsect 的 setup 程序，也就是 setup.S，此读入动作是利用 BIOS 中断服务 int 13h 的第 2 号功能。Setup 的 image 将会读入至程序所指定的内存绝对地址 0x90200 处，也就是在内存中紧邻着 bootsect 所在的位置。待 setup 的 image 读入内存后，利用 BIOS 中断服务 int 13h 的第 8 号功能读取目前磁盘的参数。

第五步

再来，就要读入真正 linux 的 kernel 了，也就是你可以在 linux 的根目录下看到的"vmlinuz" 。在读入前，将会先呼叫 BIOS 中断服务 int 10h 的第 3 号功能，读取游标位置，之后再呼叫 BIOS 中断服务 int 10h 的第 13h 号功能，在萤幕上输出字串"Loading"，这个字串在 boot linux 时都会首先被看到，相信大家应该觉得很眼熟吧。

第六步

接下来做的事是检查 root device,之后就仿照一开始的方法,利用 indirect jump 跳至刚刚已读入的 setup 部份比较。

把大家所熟知的 MS DOS 与 linux 的开机部份做个粗浅的比较，MS DOS 由位於磁盘上 boot sector 的 boot 程序负责把 IO.SYS 载入内存中，而 IO.SYS 则负有把 DOS 的 kernel --MSDOS.SYS 载入内存的重责大任。而 linux 则是由位於 boot sector 的 bootsect 程序负责把 setup 及 linux 的 kernel 载入内存中，再将控制权交给 setup。##这几步内容主要参照一个台湾同胞写的文档，setup.s 的内容希望有人补充。

Start_kernel()

当核心被载入后，首先进入的函数就是 start_kernel。

./init/main.c 中函数 start_kernel 包含核心的启动过程及顺序。通过它来看核心整个初始化过程。

首先进行一系列初始化，包括：

```
trap_init(); ##./arch/i386/kernel/traps.c 陷入
init_IRQ(); ##./arch/i386/kernel/irq.c setup IRQ
sched_init(); ##./kernel/sched.c 调度初始化，并初始化 bottom_half
time_init(); ##./arch/i386/kernel/time.c
init_modules(); ##模块初始化
mem_init(memory_start,memory_end);
buffer_init(); ## ./fs/buffer.c 缓冲区
sock_init(); ## ./net/socket.c socket 初始化，并初始化各协议(TCP 等)
ipc_init();
sysctl_init();
```

然后通过调用 kernelthread()产生 init 进程，全权交由 init 进程处理。调用 cpu_idle(NULL)休息。感兴趣又有时间的同志可以写一个 startkernel()函数的详细分析报告。

下面看一看 init 进程的工作：

首先创建进程

bdflush ##./fs/buffer.c 缓冲区管理和 kswapd ##./mm/vmscan.c 虚拟内存管理这两个进程非常重要系统初始化（系统调用 setup）系统初始化包含设备初始化及各文件系统初始化。

```
Sys_setup  (./fs/filesystems.c)
|
|-device_setup
| |
| -- chr_dev_init(); ##字符设备
| blk_dev_init(); ##块设备
| scsi_dev_init(); ##SCSI
| net_dev_init(); ##网络设备
| console_map_init(); ##控制台
|-binfmt_setup();
|-init_nls() ##各文件系统初始化
|-init_ext_fs()
|-init_ext2_fs()
. .
. .
. .
|-init_autofs_fs()
--mount_root() ##mount root fs
```

##从这儿看看设备及文件的初始化顺序，加入我们的设备时就有了大局观。

执行/etc/rc (rc.sysinit, rc.local, rc.#) 和执行/bin/sh

3. 驱动程序

在 LINUX 系统里，设备驱动程序所提供的这组入口点由一个结构来向系统进行说明，此结构定义为：

```
#include <linux/fs.h>
struct file_operations {
        int (*lseek)(struct inode *inode,struct file *filp,
                off_t off,int pos);
        int (*read)(struct inode *inode,struct file *filp,
                char *buf, int count);
        int (*write)(struct inode *inode,struct file *filp,
                char *buf,int count);
        int (*readdir)(struct inode *inode,struct file *filp,
                struct dirent *dirent,int count);
        int (*select)(struct inode *inode,struct file *filp,
                int sel_type,select_table *wait);
        int (*ioctl) (struct inode *inode,struct file *filp,
                unsigned int cmd,unsigned int arg);
        int (*mmap) (void);
int (*open) (struct inode *inode, struct file *filp);
        void (*release) (struct inode *inode, struct file *filp);
```

```
        int (*fsync) (struct inode *inode, struct file *filp);
};
```

其中，struct inode 提供了关于特别设备文件/dev/driver（假设此设备名为 driver）的信息，它的定义为：

```
#include <linux/fs.h>
struct inode {
        dev_t           i_dev;
        unsigned long   i_ino;  /* Inode number */
        umode_t         i_mode; /* Mode of the file */
        nlink_t         i_nlink;
        uid_t           i_uid;
        gid_t           i_gid;
        dev_t           i_rdev;  /* Device major and minor numbers*/
        off_t           i_size;
        time_t          i_atime;
        time_t          i_mtime;
        time_t          i_ctime;
        unsigned long   i_blksize;
        unsigned long   i_blocks;
        struct inode_operations * i_op;
      struct super_block * i_sb;
        struct wait_queue * i_wait;
        struct file_lock * i_flock;
        struct vm_area_struct * i_mmap;
        struct inode * i_next, * i_prev;
        struct inode * i_hash_next, * i_hash_prev;
        struct inode * i_bound_to, * i_bound_by;
        unsigned short i_count;
        unsigned short i_flags;  /* Mount flags (see fs.h) */
        unsigned char i_lock;
        unsigned char i_dirt;
        unsigned char i_pipe;
        unsigned char i_mount;
        unsigned char i_seek;
        unsigned char i_update;
        union {
                struct pipe_inode_info pipe_i;
                struct minix_inode_info minix_i;
                struct ext_inode_info ext_i;
                struct msdos_inode_info msdos_i;
                struct iso_inode_info isofs_i;
                struct nfs_inode_info nfs_i;
        } u;
};
```

struct file 主要用于与文件系统对应的设备驱动程序使用。当然，其它设备驱动程序也可以使用它。它提供关于被打开的文件的信息，定义为：

```
#include <linux/fs.h>
struct file {
        mode_t f_mode;
        dev_t f_rdev;             /* needed for /dev/tty */
        off_t f_pos;              /* Curr. posn in file */
        unsigned short f_flags;   /* The flags arg passed to open */
        unsigned short f_count;   /* Number of opens on this file */
        unsigned short f_reada;
        struct inode *f_inode;    /* pointer to the inode struct */
        struct file_operations *f_op;/* pointer to the fops struct*/
};
```

在结构 file_operations 里，指出了设备驱动程序所提供的入口点位置，分别是：

(1) lseek，移动文件指针的位置，显然只能用于可以随机存取的设备。

(2) read，进行读操作，参数 buf 为存放读取结果的缓冲区，count 为所要读取的数据长度。返回值为负表示读取操作发生错误，否则返回实际读取的字节数。对于字符型，要求读取的字节数和返回的实际读取字节数都必须是 inode->i_blksize 的的倍数。

(3) write，进行写操作，与 read 类似。

(4) readdir，取得下一个目录入口点，只有与文件系统相关的设备驱动程序才使用。

(5) selec，进行选择操作，如果驱动程序没有提供 select 入口，select 操作将会认为设备已经准备好进行任何的 I/O 操作。

(6) ioctl，进行读、写以外的其它操作，参数 cmd 为自定义的的命令。

(7) mmap，用于把设备的内容映射到地址空间，一般只有块设备驱动程序使用。

(8) open，打开设备准备进行 I/O 操作。返回 0 表示打开成功，返回负数表示失败。如果驱动程序没有提供 open 入口，则只要/dev/driver 文件存在就认为打开成功。

(9) release，即 close 操作。

设备驱动程序所提供的入口点，在设备驱动程序初始化的时候向系统进行登记，以便系统在适当的时候调用。LINUX 系统里，通过调用 register_chrdev 向系统注册字符型设备驱动程序。register_chrdev 定义为：

```
#include <linux/fs.h>
#include <linux/errno.h>
int register_chrdev(unsigned int major, const char *name,
            struct file_operations *fops);
```

其中，major 是为设备驱动程序向系统申请的主设备号，如果为 0 则系统为此驱动程序动态地分配一个主设备号。name 是设备名。fops 就是前面所说的对各个调用的入口点的说明。此函数返回 0 表示成功。返回-EINVAL 表示申请的主设备号非法，一般来说是主设备号大于系统所允许的最大设备号。返回-EBUSY 表示所申请的主设备号正在被其它设备驱动程序使用。如果是动态分配主设备号

成功，此函数将返回所分配的主设备号。如果 register_chrdev 操作成功，设备名就会出现在/proc/devices 文件里。初始化部分一般还负责给设备驱动程序申请系统资源，包括内存、中断、时钟、I/O 端口等，这些资源也可以在 open 子程序或别的地方申请。在这些资源不用的时候，应该释放它们，以利于资源的共享。

在 UNIX 系统里，对中断的处理是属于系统核心的部分，因此如果设备与系统之间以中断方式进行数据交换的话，就必须把该设备的驱动程序作为系统核心的一部分。设备驱动程序通过调用 request_irq 函数来申请中断，通过 free_irq 来释放中断。它们的定义为：

```
#include <linux/sched.h>
int request_irq(unsigned int irq,
                 void  (*handler)(int  irq,void  dev_id,struct  pt_regs
*regs),
                unsigned long flags,
                const char *device,
                void *dev_id);
void free_irq(unsigned int irq, void *dev_id);
```

参数 irq 表示所要申请的硬件中断号。handler 为向系统登记的中断处理子程序，中断产生时由系统来调用，调用时所带参数 irq 为中断号，dev_id 为申请时告诉系统的设备标识，regs 为中断发生时寄存器内容。device 为设备名，将会出现在/proc/interrupts 文件里。flag 是申请时的选项，它决定中断处理程序的一些特性，其中最重要的是中断处理程序是快速处理程序（flag 里设置了 SA_INTERRUPT）还是慢速处理程序（不设置 SA_INTERRUPT），快速处理程序运行时，所有中断都被屏蔽，而慢速处理程序运行时，除了正在处理的中断外，其它中断都没有被屏蔽。在 LINUX 系统中，中断可以被不同的中断处理程序共享，这要求每一个共享此中断的处理程序在申请中断时在 flags 里设置 SA_SHIRQ，这些处理程序之间以 dev_id 来区分。如果中断由某个处理程序独占，则 dev_id 可以为 NULL。request_irq 返回 0 表示成功，返回-INVAL 表示 irq>15 或 handler==NULL，返回-EBUSY 表示中断已经被占用且不能共享。

作为系统核心的一部分，设备驱动程序在申请和释放内存时不是调用 malloc 和 free，而代之以调用 kmalloc 和 kfree，它们被定义为：

```
#include <linux/kernel.h>
void * kmalloc(unsigned int len, int priority);
void kfree(void * obj);
```

参数 len 为希望申请的字节数，obj 为要释放的内存指针。priority 为分配内存操作的优先级，即在没有足够空闲内存时如何操作，一般用 GFP_KERNEL。与中断和内存不同，使用一个没有申请的 I/O 端口不会使 CPU 产生异常，也就不会导致诸如“segmentation fault”一类的错误发生。任何进程都可以访问任何一个 I/O 端口。此时系统无法保证对 I/O 端口的操作不会发生冲突，甚至会因此而使系统崩溃。因此，在使用 I/O 端口前，也应该检查此 I/O 端口是否已有别的程序在使用，若没有，再把此端口标记为正在使用，在使用完以后释放它。

这样需要用到如下几个函数：

```
int check_region(unsigned int from, unsigned int extent);
void request_region(unsigned int from, unsigned int extent,
```

```
                        const char *name);
void release_region(unsigned int from, unsigned int extent);
```

调用这些函数时的参数为：from 表示所申请的 I/O 端口的起始地址；extent 为所要申请的从 from 开始的端口数；name 为设备名，将会出现在/proc/ioports 文件里。check_region 返回 0 表示 I/O 端口空闲，否则为正在被使用。在申请了 I/O 端口之后，就可以如下几个函数来访问 I/O 端口：

```
#include <asm/io.h>
inline unsigned int inb(unsigned short port);
inline unsigned int inb_p(unsigned short port);
inline void outb(char value, unsigned short port);
inline void outb_p(char value, unsigned short port);
```

其中 inb_p 和 outb_p 插入了一定的延时以适应某些慢的 I/O 端口。

在设备驱动程序里，一般都需要用到计时机制。在 LINUX 系统中，时钟是由系统接管，设备驱动程序可以向系统申请时钟。与时钟有关的系统调用有：

```
#include <asm/param.h>
#include <linux/timer.h>
void add_timer(struct timer_list * timer);
int  del_timer(struct timer_list * timer);
inline void init_timer(struct timer_list * timer);
```

struct timer_list 的定义为：

```
struct timer_list {
                struct timer_list *next;
                struct timer_list *prev;
                unsigned long expires;
                unsigned long data;
                void (*function)(unsigned long d);
        };
```

其中 expires 是要执行 function 的时间。系统核心有一个全局变量 JIFFIES 表示当前时间，一般在调用 add_timer 时 jiffies=JIFFIES+num,表示在 num 个系统最小时间间隔后执行 function。系统最小时间间隔与所用的硬件平台有关，在核心里定义了常数 HZ 表示一秒内最小时间间隔的数目，则 num*HZ 表示 num 秒。系统计时到预定时间就调用 function，并把此子程序从定时队列里删除，因此如果想要每隔一定时间间隔执行一次的话，就必须在 function 里再一次调用 add_timer。function 的参数 d 即为 timer 里面的 data 项。在设备驱动程序里，还可能会用到如下的一些系统函数：

```
#include <asm/system.h>
#define cli() __asm__ __volatile__ ("cli"::)
#define sti() __asm__ __volatile__ ("sti"::)
```

这两个函数负责打开和关闭中断允许。

```
#include <asm/segment.h>
void memcpy_fromfs(void * to,const void * from,unsigned long n);
void memcpy_tofs(void * to,const void * from,unsigned long n);
```

在用户程序调用 read 、write 时，因为进程的运行状态由用户态变为核心态，地址空间也变为核心地址空间。而 read、write 中参数 buf 是指向用户程序

的私有地址空间的，所以不能直接访问，必须通过上述两个系统函数来访问用户程序的私有地址空间。memcpy_fromfs 由用户程序地址空间往核心地址空间复制，memcpy_tofs 则反之。参数 to 为复制的目的指针，from 为源指针，n 为要复制的字节数。在设备驱动程序里，可以调用 printk 来打印一些调试信息，用法与 printf 类似。printk 打印的信息不仅出现在屏幕上，同时还记录在文件 syslog 里。

在 LINUX 里，除了直接修改系统核心的源代码，把设备驱动程序加进核心里 以外，还可以把设备驱动程序作为可加载的模块，由系统管理员动态地加载它，使之成为核心地一部分。也可以由系统管理员把已加载地模块动态地卸载下来。LINUX 中，模块可以用 C 语言编写，用 gcc 编译成目标文件（不进行链接，作为*.o 文件存在），为此需要在 gcc 命令行里加上-c 的参数。在编译时，还应该在 gcc 的命令行里加上这样的参数：-D__KERNEL__ -DMODULE。由于在不链接时，gcc 只允许一个输入文件，因此一个模块的所有部分都必须在一个文件里实现。编译好的模块*.o 放在/lib/modules/xxxx/misc 下（xxxx 表示核心版本，如在核心版本为 2.0.30 时应该为/lib/modules/2.0.30/misc），然后用 depmod -a 使此模块成为可加载模块。模块用 insmod 命令加载，用 rmmod 命令来卸载，并可以用 lsmod 命令来查看所有已加载的模块的状态。编写模块程序的时候，必须提供两个函数，一个是 int init_module(void)，供 insmod 在加载此模块的时候自动调用，负责进行设备驱动程序的初始化工作。init_module 返回 0 以表示初始化成功，返回负数表示失败。另一个函数是 void cleanup_module (void)，在模块被卸载时调用，负责进行设备驱动程序的清除工作。在成功的向系统注册了设备驱动程序后（调用 register_chrdev 成功后），就可以用 mknod 命令来把设备映射为一个特别文件，其它程序使用这个设备的时候，只要对此特别文件进行操作就行了。

4. 将X-WINDOW换成MICROWINDOWS

Microwindows 是使用分层结构的设计方法。允许改变不同的层来适应实际的应用。在最底一层，提供了屏幕、鼠标/触摸屏和键盘的驱动，使程序能访问实际的硬件设备和其它用户定制设备。在中间一层，有一个轻巧的图形引擎，提供了绘制线条，区域填充、绘制多边形、裁剪和使用颜色模式的方法。在最上一层，提供了不同的 API 给图形应用程序使用。这些 API 可以提供或不提供桌面和窗口外型。目前，Microwindows 支持 Windows Win32/WinCE GDI 和 Nano-X API。这些 API 提供了 Win32 和 X 窗口系统的紧密兼容性，使得别的应用程序可以很容易就能移植到 Microwindows 上。

设备驱动：设备驱动接口在 device.h 中定义。一个 Microwindows 程序通常有屏幕、鼠标和键盘驱动。中间层设备无关图形引擎核心函数将直接调用设备驱动来执行硬件相关操作。这样，我们在 Microwindows 中改变或增加硬件设备时不会影响整个系统的工作。

屏幕驱动：目前，在屏幕驱动方面，有支持 Linux 2.2.x 的 framebuffer 系统的驱动，有 16 为 ELKS 和 MSDOS 系统的 VGA 显卡驱动。可以通过配置纯硬件 bios 驱动（scr_bios.c、vgaplan4.c、mempl4.c、scr_herc.c）来直接初始化 VGA 硬件，也可以利用 PC BIOS 来开始和结束图形模式。Framebuffer 驱动（scr_fb.c、fb.c、fblin?.c）提供了 1、2、4 和 8 位色显示函数，还有 8、15、16、24 和 32 色真彩显示的函数。在 Linux 下，Framebuffer 系统通过打开/dev/fb0

（或 getenv（“FRAMEBUFFER”））和使用 mmap 函数把显示内存变成线性内存。一些显示模式，如在 VGA 4 色环境下，要求 OUT 指令由屏幕驱动程序发出，而像素包装驱动使用读写 framebuffer 来实现。所有的图形模式初始化和还原都是由 Linux 内核来完成。

屏幕驱动是本系统中最复杂的驱动，在设计上很容易就能移植一个新的硬件到 Microwindows 上来。一个屏幕驱动必须提供 ReadPixel、DrawPixel、DrawHorzLine 和 DrawVertLine 等函数功能。这些函数从显存中读写像素，绘制水平线和垂直线。裁剪功能在设备无关层处理。目前，所有的鼠标动作、文本绘制和位图绘制都是基于上面的基本函数实现的。如果显示器使用调色板，必须调用 SetPalette 函数，除非系统连接了一个与系统调色板相配的静态调色板。初始化时，屏幕驱动返回包括屏幕 x、y 值和颜色模式等的值。

Microwindows 目前提供两种字体模式。提供了等比例字体从.bdf 或其它格式转换为 Microwindows 字体的转换工具。

屏幕驱动可以选择实现 bitblitting，具体是对返回标记执行或（OR）操作 PSF_HAVEBLIT。目前，位的 blitting 允许 Microwindows 执行屏幕外绘制。Microwindows 允许任何对物理屏幕的图形操作通过以下方式进行，首先执行到屏幕外的图形操作，然后复制（bit-blitted）到物理屏幕。实现一个 blitting 屏幕驱动相当复杂。实现 blitting 屏幕驱动首先要考虑的是底层显示硬件是否支持传送用于 framebuffer 的硬件地址。如果支持，在物理屏幕上的绘制函数就可以用于在屏幕外缓冲区的绘制。这是使用 framebuffer 驱动的方法。系统使用 malloc 分配的内存地址代替用 mmap 分配的物理 framebuffer 地址。在系统不使用实际物理内存地址的情况下（X 或 MS 窗口），必须写两套函数，一套用于图形系统硬件，一套用于内存地址。另外，需要知道在两种格式之间的复制方法。实际上，所有四种操作，screen-to-memory，memory-to-screen，memory-to-memory 和 screen-to-screen，Microwindows 都支持。当然，位的 blitting 函数必须要快。查看 fblin8.c 和 mempl4.c 文件可以找到支持两种显示硬件类型的例子。

如果你要写自己的屏幕驱动程序，推荐你先从 PC BIOS（scr_bios.c）驱动开始，或者看看 framebuffer 的驱动程序（scr_fb.c），scr_fb.c 被 fblin?.c 函数包装起来以至能读写各种 framebuffer 格式。先不要设置 PSF_HAVEBLIT 标记，开始时不要写 bitblit 函数。

注意：目前所有的 SCREENDEVICE 函数指针必须最少用空函数来实现。由于速度的原因，系统总是认为函数指针是有效的。所以，就算不实现 bitblit，也要提供一个什么事也不做的 bit-blit 过程。

鼠标驱动：目前，Microwindows 支持三种鼠标。mou_gpm.c 提供了 Linux 下鼠标的 GPM 驱动，mou_ser.c 提供了 Linux 和 ELKS 下串口鼠标的驱动，mou_dos.c 提供了 MSDOS 下鼠标的 int33 驱动。鼠标驱动的最基本功能是转换鼠标数据，返回鼠标的相对或绝对位置和按键。另外，Brad LaRonde 写了一个触摸屏的驱动程序 mou_tp.c，在某些场合可以代替鼠标工作。它返回触摸笔在显示屏表面的 x、y 坐标。

在 Linux 下，Microwindows 在主循环执行 select 函数，通常传递鼠标和键盘的文件描述符给 select 函数。如果系统不支持 select 函数或者在文件描述符中没有传递鼠标的数据，可以用 Poll 函数获取鼠标活动信息。

键盘驱动：提供了两种键盘驱动。第一种是 kbd_tty.c，用于 Linux 和 ELKS 系统，通过打开和读取文件描述符的方法来实现。第二种是 kbd_bios.c，用于

MSDOS 系统，通过读取 PC BIOS 的击键来实现。原来的键盘驱动只返回 8 位键盘数据，没有扩展功能键的区分能力。0.89pre7 版本重写了键盘驱动，提供了扫描码、up/down 事件、重发控制等功能。

MicroGUI - 设备无关图形引擎：调用屏幕、鼠标和键盘驱动程序来跟硬件打交道 Microwindows 的核心图形函数位于设备无关的图形引擎层中。用户应用程序从不直接调用核心图形引擎函数，因为 Microwindows 提供了相关的 API（将在下章论述）。核心图形引擎函数独立与应用程序 API 有几个原因。其一，在客户/服务器模式下，核心图形函数常常驻留在服务器上。其二，由于速度上的原因，核心图形函数使用内部文本字体和位图格式，这与标准 API 数据结构有区别。其三，核心图形函数常常使用指针，从不使用标识，等等。

在 Microwindows 中，核心图形函数总是遵守 GdXXX 的命名方式，它们只关心图形输出，不是窗口管理。另外，所有的裁剪和颜色转换都在这一层处理。下面的文件组成了 Microwindows 的核心模块：

devdraw.c 核心图形函数（线、圆、多边形、文本、位图、颜色转换）

devclip.c 核心剪裁函数（devclip2.c 是新的 y-x-banding 运算法则、devclip1.c 是旧的）

devrgn.c 相交/联合/相减/异或生成区域动态分配的函数

devmouse.c 鼠标指针更新和剪裁的核心函数

devkbd.c 核心键盘处理程序

devpalX.c 提供 1、2、4 和 8 位色调色板支持。

应用程序接口：目前，Microwindows 提供两种不同的应用程序接口。窗口管理器实现标题栏和关闭按钮的绘制，并处理程序的图形输出要求。
Microwindows API 最初是初始化屏幕、键盘和鼠标，然后是在 select 循环中等待事件。当有事件发生时，如果发生的是系统事件（如键盘、鼠标行为），则事件会转换为 expose 事件或 paint 等事件传送给应用程序。如果是用户请求图形操作，则将参数转换后调用相应的 GdXXX 引擎函数。注意，窗口与原始图形操作相对的概念在 API 层处理。说得更精确些是，API 定义好窗口、坐标系统等，然后转换成屏幕相关参数传给核心引擎函数（GdXXX），执行实际操作。这一层定义了图形上下文和屏幕设备上下文，并传递包括裁剪的信息给核心引擎函数。
目前，Microwindows API 的源程序是 mwin/win*.c，而 Nano-X API 的源程序是 nanox/srv*.c。

Microwindows API：Microwindows API 在设计上尽量顺从 Microsoft Win32 和 WinCE GDI 标准。目前，支持大部分的图形绘制和裁剪工作、支持自动绘制窗口标题栏、支持窗口的拖动。Microwindows API 使用消息机制，允许编写应用程序时无需考虑系统最终是否使用窗口管理器。目前，Microwindows API 不是客户/服务器模式，在第 4 章有详细讨论。

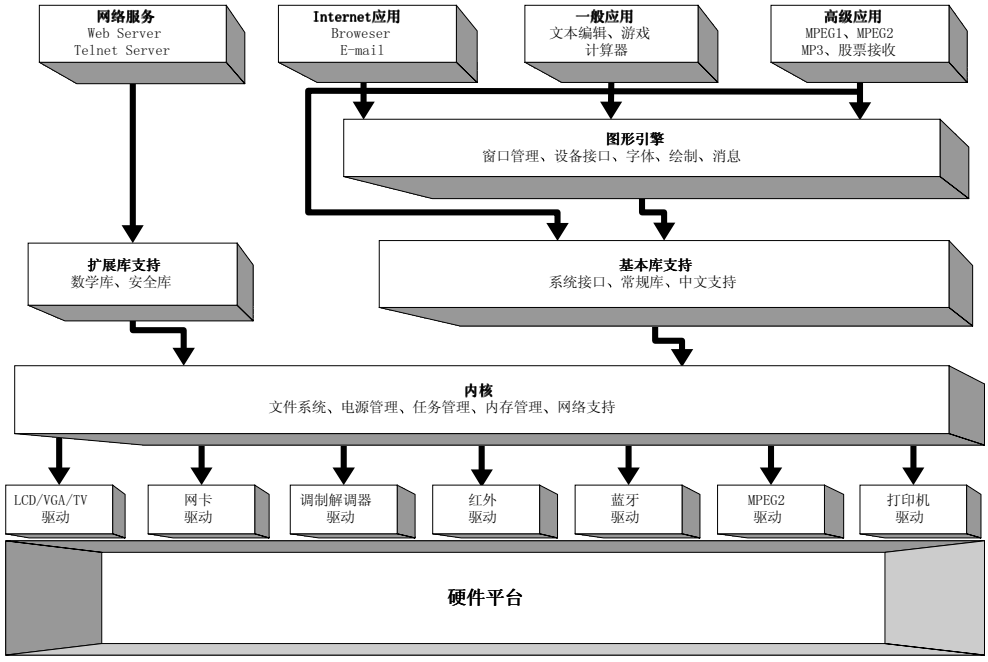
Nano-X API：Nano-X API 是模仿 David Bell 写的 mini-x 服务器开发出来的。它的 API 宽松地仿照 X 窗口系统的 Xlib API，但函数以 GrXXX 的方式命名，而不是 X...。目前，Nano-X API 使用客户/服务器模式，但不提供窗口的自动装饰、标题栏和用户窗口移动。

典型的嵌入式 Linux 操作系统—POCKET IX

1. POCKET IX概况

POCKET IX 是广州博利思软件公司为嵌入系统设计的可扩展的多任务 LINUX 操作系统，其 95%的代码用 C 语言编写，因此它能很方便移植。POCKT IX 包括 Web 支持，网络，图形包，文件系统等模块。POCKET IX 最大的特点是价格低廉并且全部提供源代码，非常符合中国的国情。自 POCKET IX 问世以来，因为其出色的稳定性和广泛的平台支持，成为了中国市场增长最快的嵌入式 Linux 操作系统，现在在各个行业中得到广泛应用。

2. POCKET IX系统结构



Pocket IX OS结构图

POCKET IX 主要由以下核心部件组成：

OS LOADER：负责将 LINUX 内核加载到内存的某处，并将控制权交给内核初始化程序。具体工作包括：寻找或将指定的内核映像解压，解压文件系统，初始化系统 CONSOLE，中断系统、时钟、计时器的管理。

KERNEL：稳定的 LINUX 2. 2 内核，支持多任务调度，可以通过外部指令动态的改变各进程的优先级，合理安排各进程的优先级可以达到最佳的系统响应。系统为每个进程提供了必备的系统资源，进程间相互隔离。系统提供信号量作为任务间同步和互斥的机制。进程间通信机制使得任务的行为同步、协调。半双工 UNIX 管道，FIFO SYSTEM V IPC，共享内存，BSD SOCKET，STREAMS。在内核中由几种类型的信号量、它们分别对不同的应用需求：二进制信号量、计数器信号量和 POSIX 信号量。所有的这些信号量是快速和高效的，它们除了被应用在开发过程中外，还广泛地应用在 LINUX 高层应用系统中。对于进程间通信，LINUX 内核也提供了诸如消息队列、管道、套接字和信号等机制。

设备驱动程序：设备驱动程序位于操作系统与硬件之间，操作系统或用户程序是通过驱动程序来控制硬件工作的。POCKET IX OS 支持的设备类型有：字

符设备、块设备、网络设备

文件系统：文件系统将存储在物理介质中的文件组织成一个树状的目录结构。POCKET IX OS 的文件系统可以支持多种的文件系统单独或同时地使用。现支持：EXT，EXT2，XIA，MINIX，UMSDOS，MSDOS，VFAT，PROC，SMB，NCP，ISO9660，SYSV，HPFS，AFFS 和 UFS。其核心为 VFS 虚拟文件系统，提供了严密的机制与独特的缓存技术，确保了数据在创建、读写、删除时的安全和性能。作为纯正的 UNIX 系统，文件系统还担当着设备驱动程序接口的作用。外部设备统一作为文件来处理，不仅提供了系统保护，而且，简化了系统设计，便于用户的使用。

图形用户接口：POCKET IX OS 提供两类图形用户接口，Tiny X window 和 Microwindows。Tiny X window 体积较大，但是功能较强，程序的通用性很好，绝大部分的 Linux 程序均可以平滑移植，有很多资源可用。Microwindows 体积较小，功能较弱，现有的资源不多，但提供类似 Win32 接口，可以移植 Win32 程序。

3. POCKET IX的技术特点

实现广泛的硬件平台支持：POCKET IX 支持 CPU 的类型多种多样，所以为不同平台定制相应的系统。在很多的在嵌入式系统中，往往不支持运行于典型的 PC 平台上的键盘、鼠标器、显示卡、显示器、声卡、ISA/PCI/AGP/USB 等设备接口，而必须支持嵌入式系统特有的接口和设备，如 PCMCIA 接口、LCD 显示屏、触摸屏等。在 POCKET IX 中，对上述设备统统支持。

采用裁减过的嵌入式实时网络操作系统内核（Linux kernel 2.2）：在标准的 Linux 中，文件系统、驱动程序、网络支持等很多功能是在内核中实现的，所以其内核相当复杂，为此 POCKET IX 保留了必要的功能，将无关的模块从内核中剔除。这样做的好处是节省了开机需要的大量时间。（例如装载系统、检测、初始化设备和文件系统、启动驱动程序以及检查硬盘等工作）。嵌入式应用要求能够瞬时开机，所以其 OS 还采用了存储映像和一系列相关技术，使系统可以瞬时加载。

基于 Tiny X 或 Microwindows 技术的图形用户接口(GUI)：采用 Tiny X GUI，考虑到 Linux 提供丰富的开放源程序资源，系统必须提供与标准 Linux 一致的 API（这里的 API 包括语法上的，如函数和系统调用的格式。也有语义上的，如相同的设备名有相同的功能。），使得标准 Linux 程序无须修改或少量修改即可使用，做到平滑移植；采用 MicroWindows GUI，考虑到对于嵌入式应用而言，标准的 X window 太庞大了。MicroWindows GUI 作为专门的 Embedded Linux GUI 系统，其大小只有 300KB（未压缩），提供了和 Win32 相似的接口和功能,同时支持中文输入和显示。

完整的网络通讯协议支持:包括 TCP/IP、PPP、POP3、SMTP 等。

作为一个软件系统平台，POCKET IX OS 能够满足大多数信息终端应用的开发，对于一些传统家电行业把一般家电产品升级到信息产品并不困难，可以在增加有限成本的基础上实现具有网络特色的新一代产品。依靠 POCKET IX OS 的强大支持，用户的应用软件可以很快地实现，同时由于 OS 起到一个相当于中间件的作用，所开发的应用有很强的硬件无关性和可移植性。因此，当用户的平台进行升级时，并不需要过多关心自己开发的应用程序，无须修改（与硬件无关的应用程序）或者只须少量修改（主要修改与硬件接口驱动部分，可以由博利思公司提供服务）即可。