RT Embedded http://www.kontronn.com

基于嵌入式Linux操作系统内核实时性的改进方法研究*

The Study of Improved Methods for Embedded Linux Systems' Real-Time Capabilities

任旭龙,周纬杰
REN Xu-long, ZHOU Wei-jie
(中国科学院软件研究所,北京 100080)
(Institute of Software, Chinese Academy of Sciences, Beijing 100080, China)

摘　要:分析当前Linux内核在实时性方面的不足,从不同侧面讨论几种改进的方法。根据嵌入式系统的不同设计目标,针对Linux内核实时性在某些方面的缺陷,使用相应的改进方法。

Abstract: The limitation of the current Linux kernel's real-time capabilities is analyzed and several improved methods for real-time capabilities from various aspects are discussed. According to the different design targets for embedded systems, the corresponding improved methods can be adopted aiming at some limitations of the Linux kernel's real-time capabilities.

关键词:嵌入式系统;Linux;内核;实时性

Key words: embedded system; Linux; kernel; real-time capabilities

中图分类号:TP316　　**文献标识码**:A

1　引言

如今计算机与信息技术的发展使得实时系统在日常工作和生活中所占的比重越来越大,在国防、交通、能源等领域都能看到实时系统的身影。在我们的周围也随处可见基于实时系统的产品,如个人数字助理(PDA)、掌上电脑(HPC)、家电机顶盒(TOP BOX)等[1]。

嵌入式操作系统是指以应用为中心,以计算机技术为基础,软硬件可剪裁,适应应用系统对功能、可靠性、成本、体积、功耗要求严格的专用计算机系统。嵌入式操作系统主要有Palm OS、Windows CE、VxWorks等,但这些都是商业化的操作系统,其价格昂贵,往往令人难以接受。嵌入式Linux相对于商业化的嵌入式操作系统具有无可替代的优越性。商业化嵌入式操作系统大都没有公开其核心源代码,这种源代码的封闭性大大限制了开发者的积极性。因此,文章将以嵌入式Linux为基础探讨内核实时性问题。

实时,顾名思义就是表示立即、及时的意思[2]。对于什么是实时系统,POSIX 1003.b作了这样的定义:指系统能够在限定的响应时间内提供所需水平的服务。而一个由Donald Gillies提出的更加为大家接受的定义是:一个实时系统是指计算的正确性不仅取决于程序的逻辑正确性,也取决于结果产生的时间,如果系统的时间约束条件得不到满足,将会发生系统出错。

2　Linux内核在实时性方面的不足[3]

(1)Linux的定时器的频率仅为100Hz,远不能满足多种实时应用的要求。

(2)Linux分为用户态和内核态两种模式,进程运行在用户态时,实时进程具有高的优先级,能进行进程抢占,故可以较好地完成任务;运行在内核态时,如系统调用,实时进程不能抢占该进程。因此,从实质上来说,Linux内核是非抢占式的。

(3)Linux进程采用多级轮转调度算法,该调度算法仅能获得秒级响应时间,一个实时进程在一个时间片内未完成,其优先级将降低,从而可能造成到截止时间无法完成。

(4)Linux虽然给实时进程提供了较高的优先级,但并没有加入时间限制,如完成的最后期限、应在多长时间内完成、执行周期等。同时,其它大量的非实时进程也可能对实

Created by Master PDF Editor

RT Embedded http://www.kontronn.com

时进程造成阻塞,无法确保实时进程的响应时间。

由于 Linux 是一个通用的操作系统,主要是作为桌面和服务器操作系统,不能用于专门的嵌入式系统中。因此,在构造嵌入式 Linux 系统时,必须根据具体的需要进行定制和裁减。

3 对 Linux 内核实时性的改进方法

3.1 利用自身系统调用法[4]

由于 Linux 的定时器只提供 10ms 的调度粒度,定时粒度比较大,如果简单地提高时钟频率会引起系统负载的增加,从而降低系统的性能。Linux 内核中有几个时钟精度在微秒级的系统调用,它们分别是:

(1) nanosleep:函数原形为:

int nanosleep (const struct timespec *req, struct timespec *rem);

其中结构 struct timespec

{time_t tv_sec; /*秒数*/

long tv_nsec;}; /*纳秒数,其值在 0 到 999999999 之间变化*/

此系统调用的作用是延迟一个执行程序在 req 中指定的秒数。如果一个信号传递给此程序,函数返回值会比预先的早些。在这种情况下,返回值为 -1,使 EINTR 为 errno,除非 rem 为 NULL;否则把余下的时间写到由 rem 所指向的结构中去。通过再次调用 nanosleep 系统调用,rem 值还可以继续使用以完成特殊的中断。可以利用 nanosleep 进程实现毫秒级的精度。

(2) setitimer:函数原形为:

int setitimer(int which, const struct itimeval *value, struct itimerval *ovalue);

其中结构 struct itimerval

{struct timerval it_interval; /*下一个时间间隔值*/

struct timerval it_value;}; /*当前值*/

struct timerval

{long tv_sec; /*秒数*/

long tv_usec;}; /*微秒数*/

系统为每一个进程提供 ITIMER_REAL、INTIMER_VIRTUAL、ITIMER_PROF 三个时间间隔定时器,它们在各自的时间域内自减,当任何一个定时器时间到达时,一个信号就传递给进程,并且定时器重新开始计时。通过设置 ITIMER_REAL 时间间隔值可以实现微秒级的精度。

(3) select:函数原形为:

int select (int n, fd_set *read_fds, fd_set *write_fds, fd_set *except_fds, struct timeval *timeout);

其中的结构 timeval 同上。

此调用允许用户在 select 调用返回之前指定一个超时时间,可达到微秒级的定时精度。

(4) poll:函数原形为:

int poll (struct pollfd *ufds, unsigned int nfds, int timeout);

与 select 系统调用相类似,允许用户指定一个超时时间值,但 poll 系统调用只能达到毫秒级的定时精度。

为提高 Linux 系统的时钟精度,考虑将实时时钟编程为单次触发状态,然后利用 CPU 的时钟计数寄存器提供高达 CPU 时钟频率的定时精度。即每次给时钟芯片设置一个超时时间,然后到该超时事件发生时,在时钟中断处理程序中再次根据需要给时钟芯片设置一个超时时间。它的基本思想是,一个精确的定时意味着我们所需的时钟中断在一个比较精确的时间发生,但并非一定要系统时钟频率达到此精度,它利用 CPU 的时钟计数器来提供精度可达 CPU 主频的时钟精度。

下面以系统调用 nanosleep 为例介绍其实现方法:

```
void nanosleep_test(int delay)
{
  struct timespec sleepy;
  sleepy.tv_sec = 0;
  sleepy.tv_nsec = delay * 1000;
  nanosleep( &sleepy, 0);
}
```

可以看出,通过对结构成员 tv_nsec 的时间设置能够达到毫秒级精度。

3.2 添加实时内核法[5]

将普通的 Linux 作为操作系统的一个低优先级的任务来处理,在普通 Linux 内核的底层实现一个微小的实时内核,如图 1 所示。此内核通过软件来模拟硬件的中断控制器,当 Linux 要封锁 CPU 中断时,实时内核就会截取这个请求并把它记录下来,判断此中断请求是交给实时内核的中断例程来处理还是交给普通 Linux 内核来出理。消减掉实时内核中与中断有关的 Linux 例程,可以解决 Linux 不能被中断的问题。例如,无论何时 Linux 调用 cli()例程,软件中断标志位都被置 0。实时内核通过标志位的状态位的状态及中断标志来捕捉中断,传送中断到 Linux 内核。当然,即使中断对 Linux 无效,对实时内核也是有效的。

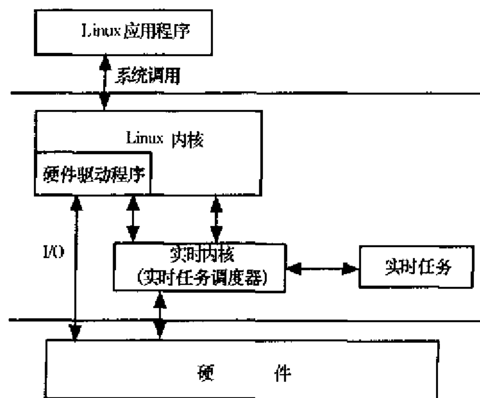

图 1 添加实时内核法

Linux 内核例程调用 cli()清除软中断标志位。当中断发生时,实时执行程序将捕捉它并决定如何做。如果中断导致实时任务,实时执行程序将保存 Linux 状态并立即执行实时任务。如果中断仅仅需要被传送到 Linux,实时执行程序将设置一标志位表明这是一个等待中断,然后恢复 Linux 而不执行中断处理程序。当 Linux 恢复中断时,实时执行程序将执行所有的等待中断并导致相应的中断处理程序执行。实时内核自身并无优先权,但由于其例程非常快而且简短,因此并不会导致大的延迟。

RT Embedded http://www.kontronn.com

下面的示例程序是测出一种添加实时内核方法在进行实时任务调度过程中调度需要花费的时间多少。

此程序将一个任务分为实时部分和非实时部分。在实时部分完成的是实时任务,在非实时部分主要完成显示等不需要实时的功能。实时部分通过使用一个模块,在将该实时模块插入之后运行实时任务。非实时任务的功能是实现对 FIFO 设备的读取,完成和实时任务的通信。程序中用到的参数是每个循环的周期 1 毫秒,并使用周期的模式。实时部分和非实时部分的程序流程以及它们之间的通信关系如图 2 所示。

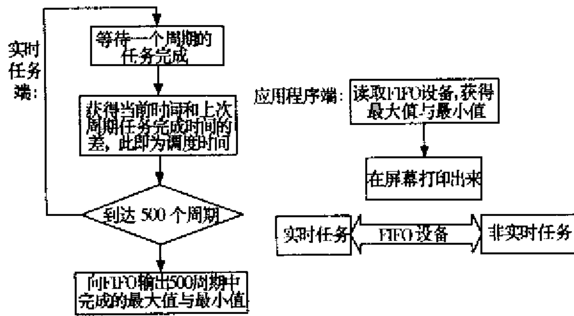

图 2　实时部分与非实时部分程序流程图

程序的运行结果如下:

MIN　　MAX

5976　　5978

……(单位是纳秒)

3. 3　三种调度策略综合算法[5]

即同时利用优先级调度、比例共享调度和时间驱动调度,通过给每个任务增加如下几项调度任务属性,并将它们作为进程调度的依据,实现一个可以支持各种调度算法的通用调度框架,如图 3 所示。其中,Priority 为作业优先级;Start-time 为作业开始时间;Finish-time 为作业结束时间;Budget 为作业在运行期间所使用的资源的多少。

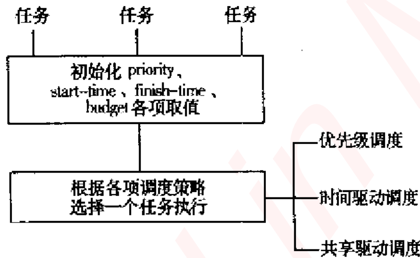

图 3　三种调度策略综合法

从图 3 中可以看出,通过调整这些属性的取值以及调度程序按照什么样的优先顺序来使用这些属性值,可以发挥这几种调度算法各自的优点,将这三种调度算法可以无缝地整合在一起。整个过程分为两个阶段进行:首先初始化 Priority、Start-time、Finish-time 以及 Budget 各项取值;然后调整这几个属性值的前后优先顺序,即这几个调度策略的权重,从而选择一个任务执行。由于此种改进方法是一种综合通用调度框架,可以由许多方式来实现,在此就不给出相应的示例。

3. 4　长任务分割算法

长任务指的是整个任务执行时间较长,超出了其它某一个或某几个任务的实时要求容限,对整个操作系统构成威胁的那些任务。在可被强占的实时内核中,长任务由于执行时间较长,因而更容易被高优先级的任务打断。一旦高优先级的任务进入了就绪态,当前任务的 CPU 使用权就被剥夺了,或者说任务被挂起了,那个高优先级的任务立刻得到 CPU 控制权。这样会出现两个问题:一是长任务可能在一次执行的过程中被频繁打断,长时间得不到一次完整的执行;二是长任务被打断时,可能要保存大量的现场信息,其目的是为了保证在高优先级的任务执行完返回后,长任务能得以继续执行。然而,这样要占用一定的系统资源;同时,保存现场本身也是要占用 CPU 时间。因此,实时性也会下降。在不可强占的实时内核中,任务级响应取决于最长的任务执行时间,长任务的存在使得 CPU 长时间停留于此,其它的任务得不到实时的响应,甚至根本得不到执行,系统的实时性必然会下降。分割长任务可以用如下两种方法来实现:

(1) 将长任务按功能分为若干个小模块,每一个模块构成一个小任务,每一个小任务执行一个相对独立的功能,且保证执行时间 $t < \min(1/f1, 1/f2, 1/f3, \cdots, 1/fn)$。其中,$fn$ 代表各个任务所要求的最小执行频率。各个任务被内核顺序调用,合起来完成整个任务的功能。

(2) 有的长任务比较特殊,例如键盘任务和动态 LED 显示任务,很难按照方法(1)那样分成若干个功能相对独立的小模块。这时,可以按照方便保存现场信息的原则,强制将其分割成若干个小任务,每个任务在 $\min(1/f1, 1/f2, 1/f3, \cdots, 1/fn)$ 时间内主动保存现场信息,并放弃 CPU 的控制权,等到再次被内核调度时继续执行。

下面以键盘接受用户键击为例进行说明。键盘主要用于输入数据、代码和命令,因此系统必须不停地扫描。一旦有键按下,CPU 就作出响应。键盘任务子程序就是根据这个要求设计的,从有键按下时开始到按键释放并转到相应的子程序为止。它一般有两种处理方式:一是中断方式,二种是查询方式。如图 4 中所示,Δt 是人手按键的时间,一般为 100ms～300ms,为说明问题,这里取 100ms。令 $t1' - t1 = \Delta t1$, $t2' - t2 = \Delta t2$,则一次键盘任务的执行时间:

$$\Delta t' = t2' - t1' = t2 + (t2' - t2) - [t1 + (t1' - t1)] = t2 + \Delta t2 - t1 - \Delta t1 = t2 - t1 + (\Delta t2 + \Delta t1) = \Delta t + (\Delta t2 - \Delta t1)$$

由图 4 可知,$\Delta t1 \leq 5\text{ms}$, $\Delta t2 \leq 5\text{ms}$。实际上,由于两次任务执行的情况不完全相同,$\Delta t1$ 与 $\Delta t2$ 可能会有相当于几个指令周期的时间差异,但也只有 μs 级的误差。而且,这里是以时间片的形式来计算的,所以依然可以认为 $\Delta t2 - \Delta t1 = 0$,因而有 $\Delta t' = \Delta t + (\Delta t2 - \Delta t1) = \Delta t = 100\text{ms}$,即 20 个时间片。在这 20 个时间片内只执行一次键盘任务,大部分时间都在空等待。在此期间,系统中会有 19 次其它任务的执行被错过。显然,这在大多数多任务系统中是不允许的。因此,在这个多任务实时系统中,键盘任务是个典型的长任务,要使其它 19 次任务中断都有被应用的机会,就必须对该键盘任务进行分割。

键盘分割的方法是将一次键盘任务分成多个小任务,通过多次任务中断来完成。我们将其放在任务 4 中(80ms 执行一次),这样有 $t_{task4} \leq t_{task} \leq 2\, t_{task4}$,这样既可以保

RT Embedded http://www.kontronn.com

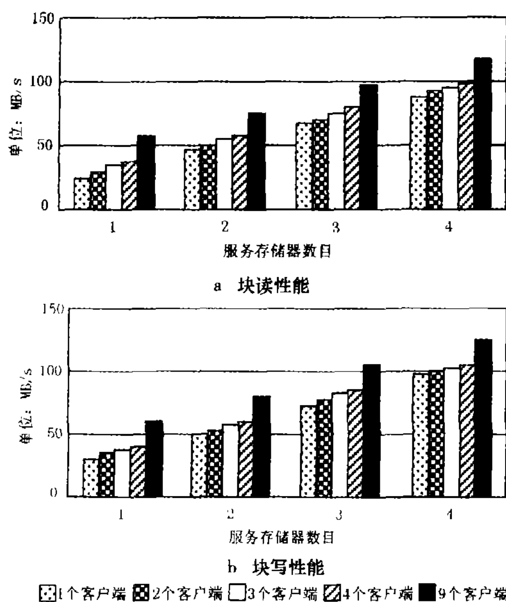

图 4 SDSP604 系统可扩展能测试结果

高性能并行计算机 I/O 系统设计还涉及很多关键技术,如:I/O 相关的 Cache 一致性维护机制、I/O 同步技术、RAS 技术等等,限于篇幅,在此不可能一一深入研讨。系统设计者不能孤立地研究和设计并行 I/O 系统,应从硬件体系结构、文件系统并行 I/O 优化、编译器 I/O 支持、并行编程环境,以及应用层应用程序等各个方面,全面研究并行计算机 I/O 系统。

参考文献:

[1] Kai Hwang. Computer Architecture and Parallel Processing [M]. The McGraw-Hill Companies Inc,1998.

[2] Kai Hwang. Scalable Parallel Computing[M]. The McGraw-Hill Companies Inc,1993.

[3] Li Qiong, Liu Guang-ming, Guo Yu-feng, et al. A New High-Performance Distributed Shared I/O System[A]. 5th Int'l Workshop on Advanced Parallel Processing Technologies [C]. 2003. 41-49.

(上接第 127 页)

证键盘任务完全可靠地执行,又提高了 CPU 的效率。图 5 是分割后的键盘任务的执行过程。从图 5 中可以看出,一次完整的键盘任务是由 2～3 次任务中断完成的。情况 a 中,第一次任务中断发生在 *ta*1 时刻,在接下来的一个时间片(5ms)内,检测到有键按下,调用查询值任务;第二次任务中断发生在 *ta*2 时刻,在接下来的一个时间片(5ms)内,检测到按键已释放,跳转并执行相应键处理程序。情况 b 中,第二次任务中断 *tb*2 时刻,键还没有松开,因此实际上什么也不做;第三次中断 *tb*3 时刻才跳到与键对应的子程序。

通过这样的分割,每个子任务都可以在 5ms 内完成。如果系统中还有大于 5ms 的长任务的话,可按此方法继续进行分割;如果没有的话,系统任务级响应时间一定小于 5ms。这样,系统的实时性大大提高,由 100ms 提高到 5ms。

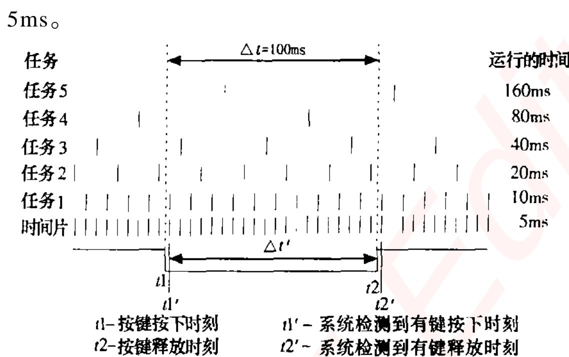

图 4 键盘任务在未分割任务前的执行过程

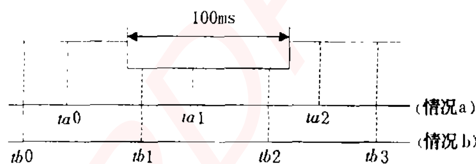

图 5 分割后键盘任务的执行过程

4 结束语

针对嵌入式系统的不同设计目标,可以采用不同的实现方法。针对 Linux 系统定时粒度过大问题,可以将实时时钟编程为单次触发状态,利用 CPU 时钟记数寄存器提供高达 CPU 时钟频率的定时精度;针对 Linux 系统运行在内核态进程不能被抢占的问题,实现一个微小的实时内核,用软件模拟硬件的中断控制器,解决内核态时中断实时进程不能运行的问题;针对 Linux 中缺乏实时调度机制和调度算法的问题,可以将目前采用的几种调度算法结合起来,实现一个通用的实时调度框架;针对长任务的存在使系统的实时性下降的问题,可以采用两种长任务分割方法来实现。目前采用的这几种提高 Linux 实时性的方法还有许多需要改进的地方,相信 Linux 作为一种开放源码的操作系统,与其它商用的嵌入式操作系统相比具有一定的优势,一定会有更新颖的调度方法和策略出现。

参考文献:

[1] 唐寅. 实时操作系统应用开发指南[M]. 北京:中国电力出版社,2002.

[2] 邹思轶. 嵌入式 Linux 设计与应用[M]. 北京:清华大学出版社,2001

[3] 毛德操,胡希明. Linux 内核源代码情景分析[M]. 杭州:浙江大学出版社,2001.

[4] http://www.ittc.ku.edu/kurt/papers/user_manual_DRAFT.pdf,2002-03.

[5] http://www.fsmlab.com/articles/archive/rtmanifesto.pdf,2003-09.

[6] http://www.linuxdevices.com/links/LK7432493143.html,2000-01.

Created in Master PDF Editor

嵌入式资源免费下载

总线协议：

1. 基于 PCIe 驱动程序的数据传输卡 DMA 传输
2. 基于 PCIe 总线协议的设备驱动开发
3. CANopen 协议介绍
4. 基于 PXI 总线 RS422 数据通信卡 WDM 驱动程序设计
5. FPGA 实现 PCIe 总线 DMA 设计
6. PCI Express 协议实现与验证
7. VPX 总线技术及其实现
8. 基于 Xilinx FPGA 的 PCIE 接口实现
9. 基于 PCI 总线的 GPS 授时卡设计
10. 基于 CPCI 标准的 6U 信号处理平台的设计
11. USB30 电路保护
12. USB30 协议分析与框架设计
13. USB 30 中的 CRC 校验原理及实现
14. 基于 CPLD 的 UART 设计
15. IPMI 在 VPX 系统中的应用与设计
16. 基于 CPCI 总线的 PMC 载板设计
17. 基于 VPX 总线的工件台运动控制系统研究与开发
18. PCI Express 流控机制的研究与实现
19. UART16C554 的设计
20. 基于 VPX 的高性能计算机设计

VxWorks：

1. 基于 VxWorks 的多任务程序设计
2. 基于 VxWorks 的数据采集存储装置设计
3. Flash 文件系统分析及其在 VxWorks 中的实现
4. VxWorks 多任务编程中的异常研究
5. VxWorks 应用技巧两例
6. 一种基于 VxWorks 的飞行仿真实时管理系统
7. 在 VxWorks 系统中使用 TrueType 字库
8. 基于 FreeType 的 VxWorks 中文显示方案

Created in Master PDF Editor

9. 基于 Tilcon 的 VxWorks 简单动画开发
10. 基于 Tilcon 的某武器显控系统界面设计
11. 基于 Tilcon 的综合导航信息处理装置界面设计
12. VxWorks 的内存配置和管理
13. 基于 VxWorks 系统的 PCI 配置与应用
14. 基于 MPC8270 的 VxWorks BSP 的移植
15. Bootrom 功能改进经验谈
16. 基于 VxWorks 嵌入式系统的中文平台研究与实现
17. VxBus 的 A429 接口驱动
18. 基于 VxBus 和 MPC8569E 千兆网驱动开发和实现
19. 一种基于 vxBus 的 PPC 与 FPGA 高速互联的驱动设计方法
20. 基于 VxBus 的设备驱动开发

Linux:

1. Linux 程序设计第三版及源代码
2. NAND FLASH 文件系统的设计与实现
3. 多通道串行通信设备的 Linux 驱动程序实现
4. Zsh 开发指南-数组
5. 常用 GDB 命令中文速览
6. 嵌入式 C 进阶之道
7. Linux 串口编程实例
8. 基于 Yocto Project 的嵌入式应用设计
9. Android 应用的反编译
10. 基于 Android 行为的加密应用系统研究
11. 嵌入式 Linux 系统移植步步通
12. 嵌入式 CC++语言精华文章集锦
13. 基于 Linux 的高性能服务器端的设计与研究
14. S3C6410 移植 Android 内核
15. Android 开发指南中文版
16. 图解 Linux 操作系统架构设计与实现原理（第二版）
17. 如何在 Ubuntu 和 Linux Mint 下轻松升级 Linux 内核
18. Android 简单 mp3 播放器源码
19. 嵌入式 Linux 系统实时性的研究
20. Android 嵌入式系统架构及内核浅析

Windows CE:

Created with Master PDF Editor

1. Windows CE.NET 下 YAFFS 文件系统 NAND Flash 驱动程序设计
2. Windows CE 的 CAN 总线驱动程序设计
3. 基于 Windows CE.NET 的 ADC 驱动程序实现与应用的研究
4. 基于 Windows CE.NET 平台的串行通信实现
5. 基于 Windows CE.NET 下的 GPRS 模块的研究与开发
6. win2k 下 NTFS 分区用 ntldr 加载进 dos 源代码
7. Windows 下的 USB 设备驱动程序开发
8. WinCE 的大容量程控数据传输解决方案设计
9. WinCE6.0 安装开发详解
10. DOS 下仿 Windows 的自带计算器程序 C 源码
11. G726 局域网语音通话程序和源代码
12. WinCE 主板加载第三方驱动程序的方法
13. WinCE 下的注册表编辑程序和源代码
14. WinCE 串口通信源代码
15. WINCE 的 SD 卡程序[可实现读写的源码]
16. 基于 WinCE 的 BootLoader 研究

PowerPC：

1. Freescale MPC8536 开发板原理图
2. 基于 MPC8548E 的固件设计
3. 基于 MPC8548E 的嵌入式数据处理系统设计
4. 基于 PowerPC 嵌入式网络通信平台的实现
5. PowerPC 在车辆显控系统中的应用
6. 基于 PowerPC 的单板计算机的设计
7. 用 PowerPC860 实现 FPGA 配置

ARM：

1. 基于 DiskOnChip 2000 的驱动程序设计及应用
2. 基于 ARM 体系的 PC-104 总线设计
3. 基于 ARM 的嵌入式系统中断处理机制研究
4. 设计 ARM 的中断处理
5. 基于 ARM 的数据采集系统并行总线的驱动设计

Created in Master PDF Editor

6. S3C2410 下的 TFT LCD 驱动源码
7. STM32 SD 卡移植 FATFS 文件系统源码
8. STM32 ADC 多通道源码
9. ARM Linux 在 EP7312 上的移植
10. ARM 经典 300 问
11. 基于 S5PV210 的频谱监测设备嵌入式系统设计与实现
12. Uboot 中 start.S 源码的指令级的详尽解析
13. 基于 ARM9 的嵌入式 Zigbee 网关设计与实现

Hardware：

1. DSP 电源的典型设计
2. 高频脉冲电源设计
3. 电源的综合保护设计
4. 任意波形电源的设计
5. 高速 PCB 信号完整性分析及应用
6. DM642 高速图像采集系统的电磁干扰设计
7. 使用 COMExpress Nano 工控板实现 IP 调度设备
8. 基于 COM Express 架构的数据记录仪的设计与实现
9. 基于 COM Express 的信号系统逻辑运算单元设计

Created in Master PDF Editor