基于X86平台的简单多任务内核的分析与实现

潘永才，张 鹏，余小娟，陈俊琪，程 鼎
（湖北大学 物理学与电子技术学院，湖北 武汉 430062）

摘 要：描述了一个简单多任务内核的设计和实现方法。分析了该简单内核的基本结构和加载运行的基本原理，然后描述了其被加载进机器RAM中以及两个任务进行切换的运行方法。

关键词：X86平台；内核；多任务；加载

中图分类号：TP311.6　　**文献标识码**：A　　**文章编号**：2095-1302(2013)12-0059-06

Analysis and application of simple multi-task kernel on X86 platform

PAN Yong-cai, ZHANG Peng, YU Xiao-juan, CHEN Jun-qi, CHENG Ding
(College of Physics and Electronic, Hubei University, Wuhan 430062, China)

Abstract: A design and implementation method of simple multi-task kernel is described. The basic structure of the kernel and the basic principles of loading are simply analyzed, and the method of loading into the RAM and switching in two tasks are also described.

Keywords: X86 platform; kernel; multi-task; loading

0 引 言

当提到多任务时，人们便会联想到 Mac OS、Linux、Windows 等操作系统。通常情况下，若在操作系统下运行多任务，是由操作系统负责管理和调度各个任务的。本文通过分析一个简单的多任务内核，能够便于更容易地理解操作系统的任务管理机制，以及可以理解计算机系统是如何启动的。

1 多任务程序的结构

本文实现的简单多任务内核，主要由两个文件构成：一个是使用 as86 语言编制的引导启动程序，主要用于在计算机系统加电时，将内核代码从启动盘加载到内存中；另一个便是使用 GNU as 汇编语言编写的内核程序，其中实现两个运行在特权级 2 上的任务可在时钟中断控制下相互切换运行，并可通过系统调用在屏幕上实现字符显示。

2 多任务内核工作的启动程序原理

计算机系统加电启动后，会把启动程序从启动盘的第一个扇区加载到物理内存 0x7c00 位置开始处，之后把执行权交给 0x7c00 初开始运行启动程序。

启动程序的主要功能是将软盘或者镜像文件中的内核程序加载到内存的某个指定位置，实现这个目的的方法是利用 ROS BIOS 中断 int 0x13，把软盘或者镜像中的内核代码读入到内存，然后再把这段内核代码移动到内存 0 开始处。最后设置控制寄存器 CR0 中的开启保护运行模式标志，并跳转到内存 0 处开始执行内核代码。启动程序在内存中移动内核代码的示意图如图 1 所示。

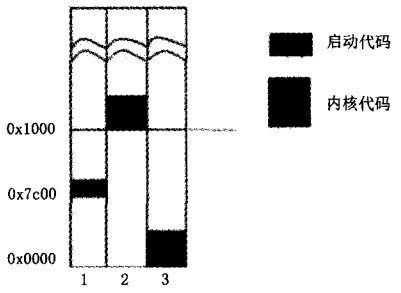

图 1 启动程序在内存中移动内核代码的示意图

将内核代码移动到物理内存 0 开始处的主要原因是这是 GDT 表时可以简单一点。但是，不能让启动程序把内核代码从软盘或映像文件中直接加载到内存 0 处，因为加载操作需要 ROM BIOS 提供中断过程，而 BIOS 使用的中断向量表正处于内存 0 开始处。若直接把内核代码加载到内存 0 处，那么，BIOS 中断过程将不能正常运行。

3 内核程序

3.1 初始化任务

内核程序运行在 32 位保护模式下，初始化阶段主要包括

收稿日期：2013-09-24

重新设置 GDT 表，设置系统定时器芯片，重新设置 IDT 表并且设置时钟和系统调用中断门。内核示例中所有代码和数据段都对应到物理内存同一个区域上，即从物理内存 0 开始的区域。在虚拟地址空间中内核程序的内核代码和任务代码分配图如图 2 所示。

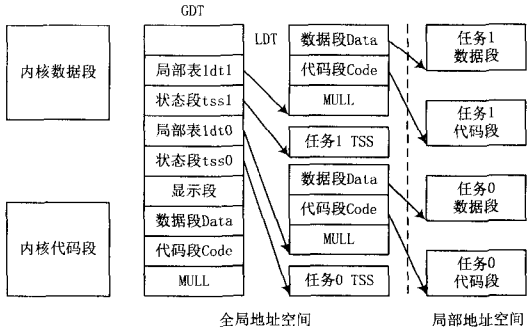

图 2 虚拟地址空间中内核程序的内核代码和任务代码分配图

3.2 启动第一个任务

特权级 0 的代码不能直接把控制权转移到特权级 2 的代码中执行，但可以使用中断返回操作来实现，因此当初始化 GDT、IDT 和定时芯片结束后，就利用中断返回指令 IRET 来启动第一个任务。

具体的实现方法是在初始堆栈 init_stack 中人工设置一个返回环境，即把任务 0 的 TSS 段选择符加载到任务寄存器 TR 中，LDT 段选择符加载到 LDTR 中以后，把任务 0 的用户栈指针和代码指针以及标志寄存器值压入栈中，然后执行中断返回指令 IRET。该指令会弹出堆栈上的堆栈指针作为任务 0 用户栈指针，恢复假设的任务 0 的标志寄存器内容，并且弹出栈中代码指针放入 CS：EIP 寄存器中，从而开始执行任务 0 的代码，以完成从特权级 0 到特权级 3 代码的控制转移。

3.3 两个任务的切换

内核程序将定时器芯片的通道 0 设置成每经过 10 ms 就向中断控制芯片发送一个时钟中断请求信号，这样，每个 10 ms 将会切换运行的任务。PC 的 ROM BIOS 开机时已经在定时器芯片中把时钟中断请求信号设置成中断向量 8，因此需要在中断 8 的处理过程中执行任务切换操作。

每个任务在执行时，会首先把一个字符的 ASCII 码放入寄存器 AL 中，然后调用系统中断 int 0x80，而该系统调用处理过程会调用一个简单的字符写屏子程序。在显示过一个字符后，任务代码会使用循环语句延迟一段时间，然后又跳转到任务代码开始处继续循环执行，直到运行了 10 ms 而发生了定时中断，从而代码会切换到另一个任务去运行。

目前，该内核示例已经在 Bochs 模拟软件中运行测试过，测试结果如图 3 所示。

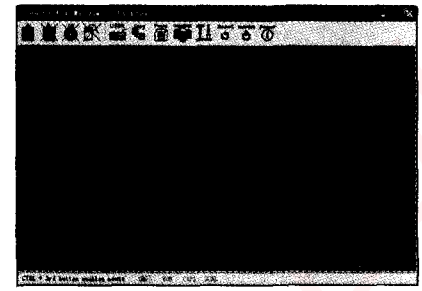

图 3 基于 Bochs 模拟软件的运行测试结果

4 结 语

本文分析了一个基于 X86 平台的简单多任务内核的基本结构和加载运行原理，描述了其被加载进机器 RAM 中的基本思路，同时给出了两个任务进行切换的运行方法。其主要目的是理解操作系统的启动加载过程。

附：本文的启动代码及内核代码如下：

```
###############################################
# 名称：引导程序 boot.s                        #
# 说明：把镜像文件中的 head 内核代码加载到内存某个指定位置。  #
#                                              #
###############################################
BOOTSEG=0x07c0
SYSSEG=0x1000
SYSLEN=17

entry start
start：
    jmpi go，#BOOTSEG
go：
    mov ax，cs
    mov ds，ax
    mov es，ax
    mov ss，ax
    mov sp，#0x400

    mov ax，#0x0600
    mov cx，#0x0000
    mov dx，#0xFFFF
    int 0x10

    mov cx，#10
    mov dx，#0x0000
    mov bx，#0x000c
```

```
    mov bp, #msg
    mov ax, #0x1301
    int 0x10


load_system :
    mov dx, #0x0000
    mov cx, #0x0002
    mov ax, #SYSSEG
    mov es, ax
    xor bx, bx
    mov ax, #0x200+SYSLEN
    int 0x13
    jnc ok_load
    mov dx, #0x0000
    mov ax, #0x0000
    int 0x13
    jmp load_system

ok_load :
    cli
    mov ax, #SYSSEG
    mov ds, ax
    xor ax, ax
    mov es, ax
    mov cx, #0x1000
    sub si, si
    sub di, di
    rep
    movw

    mov ax, cs
    mov ds, ax

    lidt idt_48
    lgdt gdt_48

    mov ax, #0x0001
    lmsw ax
    jmpi 0, 8

msg : .ascii "Loading..."
    .byte 13, 10

gdt :
    .word 0, 0, 0, 0

    .word 0x07FF
    .word 0x0000
    .word 0x9A00
    .word 0x00C0

    .word 0x07FF
    .word 0x0000
    .word 0x9200
    .word 0x00C0

idt_48 :
    .word 0
    .word 0, 0

gdt_48 :
    .word 0x7FF
    .word 0x7c00+gdt, 0

.org 510
    .word 0xAA55
    ##############################################
    # 名称：内核程序 head.s
    # 说明：包含 32 位保护模式初始化设置代码，时钟中断代码，系统调用中断代码和两个任务的代码。在初始化完成之后程序移动到任务 0 开始执行，并在时钟中断控制下进行任务 0 和 1 之间的切换操作。
    ##############################################
    LATCH = 11930    # 定时器初始计数值，即每隔 10ms 发送一次中断请求。
    SCRN_SEL = 0x18 # 屏幕显示内存段选择符
    TSS0_SEL = 0x20 # 任务 0 的 TSS 段选择符
    LDT0_SEL = 0x28 # 任务 0 的 LDT 段选择符
    TSS1_SEL = 0x30 # 任务 1 的 TSS 段选择符
    LDT1_SEL = 0x38 # 任务 1 的 LDT 段选择符

    .globl startup_32
    .text
```

```
startup_32：
# 首先加载数据段寄存器 DS、堆栈段寄存器 SS 和堆栈指针 ESP。所有段的线性基地址都是 0.
    movl $0x10, %eax  # 0x10 是 GDT 中数据段选择符。
    mov %ax，%ds
    lss init_stack，%esp # 把 init_stack 地址处的内容既 init_stack 有效地址给 esp，同时把 0x10 给 ss 段寄存器

# 在新的位置重新设置 IDT 和 GDT 表。
    call setup_idt   # 设置 IDT，先把 256 个中断门都填默认处理过程的描述符
    call setup_gdt

    movl $0x10，%eax  # 在改变了 GDT 之后 重新加载所有段寄存器 .
    mov %ax，%ds
    mov %ax，%es
    mov %ax，%fs
    mov %ax，%gs
    lss init_stack，%esp

# 设置 8253 定时芯片。把计数器通道 0 设置成每个 10ms 向中断控制器发送一个中断请求信号。
    movb $0x36,%al # 控制字：设置通道 0 工作在方式 3、计数处置采用二进制
    movl $0x43，%edx #8253 芯片控制字寄存器写端口
    outb %al，%dx
    movl $LATCH，%eax # 初始计数值设置为 LATCH（1193180/100），即频率 100HZ
    movl $0x40，%edx # 通道 0 的端口
    outb %al，%dx   # 分两次把初始计数值写入通道 0
    movb %ah，%al
    outb %al，%dx
# 在 IDT 表第 8 和第 128（0x80）项处分别设置定时中断门描述符和系统调用陷阱门描述符
    movl $0x00080000，%eax # 中断处理属内核，即 EAX 高字节是内核代码段选择符 0x0008.
    movw $timer_interrupt，%ax # 设置定时中断门描述符。取定时中断处理程序地址。
    movw $0x8E00,%dx # 中断门类型是 14（屏蔽中断），特权级 0 或硬件使用。
    movl $0x08，%ecx # 开机时 BIOS 设置的时钟中断向量号 8. 这里直接使用它。
    lea idt(，%ecx，8)，%esi # 把 IDT 描述符 0x08 地址放入 ESI 中，然后设置该描述符
    movl %eax，(%esi)
    movl %edx，4(%esi)
    movw $system_interrupt，%ax # 设置系统调用陷阱门描述符。取系统调用处理程序地址。
    movw $0xef00，%dx # 陷阱门类型是 15，特权级 3 的程序可执行。
    movl $0x80，%ecx # 系统调用向量号是 0x80.
    lea idt(，%ecx，8)，%esi # 把 IDT 描述符项 0x80 地址放入 ESI 中，然后设置该描述符 .
    movl %eax，(%esi)
    movl %edx，4(%esi)
# 现在，为移动到任务 0 中执行来操作堆栈内容，在堆栈中人工建立中断返回时的场景
    pushfl
    andl $0xffffbfff，(%esp)
    popfl
    movl $TSS0_SEL，%eax # 把任务 0 的 TSS 段选择符加载到任务寄存器 TR
    ltr %ax
    movl $LDT0_SEL，%eax # 把任务 0 的 LDT 段选择符加载到局部描述符表寄存器 LDTR
    lldt %ax
    movl $0，current
    sti

    pushl $0x17 # 任务 0 当前局部空间数据段选择符入栈
    pushl $init_stack # 堆栈指针入栈
    pushfl # 标志寄存器值入栈
    pushl $0x0f # 任务 0 局部空间代码段选择符入栈
    pushl $task0 # 把代码指针入栈

    iret # 执行中断返回指令，从而切换到特权级 3 的任务 0 中执行 .

# 一下是设置 GDT 和 IDT 中描述符项的子程序
setup_gdt：
    lgdt lgdt_opcode
    ret
```

设置 IDT 表中所有 256 个中断们描述符都为统一个默认值，均使用默认的中断处理过程 ignore_int。

```
setup_idt：
    lea ignore_int，%edx
    movl $0x0008000，%eax
    movw %dx，%ax
    movw $0x8E00，%dx
    lea idt，%edi
    mov $256，%ecx
rp_idt：movl %eax，(%edi)
    movl %edx，4(%edi)
    addl $8，%edi
    dec %ecx
    jne rp_idt
    lidt lidt_opcode
    ret

# 显示字符子程序
write_char：
  push %gs
  pushl %ebx

  mov $SCRN_SEL，%ebx
  mov %bx，%gs

  mov src_loc，%bx
  shl $1，%ebx
  mov %ax，%gs：(%ebx)
  shr $1，%ebx
  incl %ebx
  cmpl $2000，%ebx
  jb 1f
  movl $0，%ebx
1：
  movl %ebx，src_loc
  popl %ebx
  pop %gs
  ret
```

以下是 3 个中断处理程序：默认中断、定时中断和系统调用中断

#ignore_int 是默认的中断处理程序，若系统产生了其他中断，会在屏幕上显示一个字符“C”

```
.align 2
ignore_int：
  push %ds
  pushl %eax
  movl $0x10，%eax
  mov %ax，%ds
  mov $0x0c98，%ax        /* print 'C' */
  call write_char
  popl %eax
  pop %ds
  iret

# 这是定时中断处理程序，主要执行任务切换操作.
.align 2
timer_interrupt：
  push %ds
  pushl %eax
  movl $0x10，%eax
  mov %ax，%ds
  movb $0x20，%al
  outb %al，$0x20

    movl $1，%eax
    cmpl %eax，current
    je 1f
    movl %eax，current
    ljmp $TSS1_SEL，$0
    jmp 2f
1：  movl $0，current
    ljmp $TSS0_SEL，$0
2：  popl %eax
  pop %ds
  iret

# 系统调用中断 int 0x80 处理程序.显示字符功能
.align 2
system_interrupt：
    push %ds
  pushl %edx
  pushl %ecx
  pushl %ebx
```

```
    pushl %eax
    movl $0x10, %edx
    mov %dx, %ds

    call write_char

    popl %eax
    popl %ebx
    popl %ecx
    popl %edx
    pop %ds
    iret

/************************************************/
current: .long 0
src_loc: .long 0

.align 2
lidt_opcode:
    .word 256*8-1
    .long idt
lgdt_opcode:
    .word (end_gdt-gdt)-1
    .long gdt

.align 2
idt: .fill 256, 8, 0

gdt:        .quad 0x0000000000000000 #GDT 表，第一个描述符不用
    .quad 0x00c09a00000007ff # 第二个是内核代码段描述符，其选择符是 0x08.
    .quad 0x00c09200000007ff # 第三个是内核数据段描述符，其选择符是 0x10.
    .quad 0x00c0920b80000002 # 第 4 个是显示内存段描述符，其选择符是 0x18.
    .word 0x68, tss0, 0xe900, 0x0 # 第 5 个是 TSS0 段的描述符，其选择符是 0x20.
    .word 0x40, ldt0, 0xe200, 0x0 # 第 6 个是 LDT0 段的描述符，其选择符是 0x28
    .word 0x68, tss1, 0xe900, 0x0 # 第 7 个是 TSS1 段的描述符，其选择符是 0x30
    .word 0x40, ldt1, 0xe200, 0x0 # 第 8 个是 LDT1 段的描述符，其选择符是 0x38
end_gdt:
    .fill 128, 4, 0         # 初始内核堆栈空间
init_stack:                 # 刚进入保护模式时用于加载 SS:ESP 堆栈指针值
    .long init_stack
    .word 0x10

# 下面是任务 0 的 LDT 表段中的局部段描述符
.align 2
ldt0:       .quad 0x0000000000000000
    .quad 0x00c0fa00000003ff    # 第 2 个是局部代码段描述符，对应选择符是 0x0f.
    .quad 0x00c0f200000003ff    # 第 3 个是局部数度段描述符，对应选择符是 0x17.
# 下面是任务 0 的 TSS 段的内容
tss0:       .long 0
    .long krn_stk0, 0x10  /*esp0, ss0*/
    .long 0, 0, 0, 0, 0         /*esp1, ss1, esp2, ss2, cr3*/
    .long 0, 0, 0, 0, 0         /*eip, eflags, eax, ecx, edx*/
    .long 0, 0, 0, 0, 0         /*ebx, esp, ebp, esi, edi*/
    .long 0, 0, 0, 0, 0, 0      /*es, cs, ss, ds, fs, gs*/
    .long LDT0_SEL, 0x8000000       /*ldt, trace bitmap*/

    .fill 128, 4, 0             # 这是任务 0 的内核栈空间
krn_stk0:

# 下面是任务 1 的 LDT 表段内容和 TSS 段内容
.align 2
ldt1:       .quad 0x0000000000000000# 第 1 个描述符，不用
    .quad 0x00c0fa00000003ff  # 选择符是 0x0f，基地址 = 0x00000
    .quad 0x00c0f200000003ff  # 选择符是 0x17，基地
```

（下转第 67 页）

表 1 GPRS 网络 TCP 连接的 AT 命令

AT命令	命令解释
AT	检查模块与串口是否正常连接
AT+CIPMUX=0	GPRS网络链路为单路通信
AT+CIPQRCLOSE=1	加速远端断开连接
AT+CIPMODE=0	选择TCPIP为非透明应用模式
AT+CIPSTART=TCP，122.233.174.908，1029	选择TCP连接方式，其中122.233.174.908为PC公网地址，1029为GPPRS网络监听端口
AT+CIPSEND	发送数据

3 监控服务中心器

监控中心与视频数据图像采集终端建立客户 / 服务器（C/S）模式。监控中心服务器使用普通 PC，有静态 IP 和开放监听端口。监控中心服务器具有监听 TCP 端口，返回数据包，描述接收的数据和大小，保存图像文件等功能。监控中心服务器工作流程如图 5 所示。

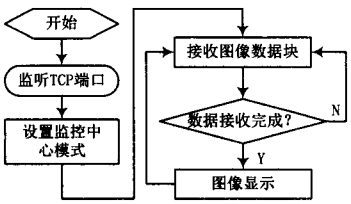

图 5 监控中心服务器工作流程图

4 结 语

本系统充分发挥了永远在线的 GPRS 无线传输技术和高性能的嵌入式监控系统的优点，实现了远程图像监控。系统测试结果较满意，使用中国移动的 GPRS 网络服务，本系统的数据传输率大于 10 Kb/s，经压缩后分辨率为 320×240，图像大小在 50 Kb 左右并且能够在 4 s 左右传输到监控中心服务器上进行实时显示。经过以上测试，本系统具有较高的视频采集和传输速率，实现了基于 GPRS 无线传输技术的远程图像监控，具有广泛的应用前景。

参 考 文 献

[1] ZHANG Xing-ming, LIN Run-jie. Design and implementation of a mobile video surveillance system [C]// Proceedings of 2011 International Conference on Internet Technology and Applications. Wuhan, China: iTAP, 2011: 1-5.

[2] 梁笃国 . 网络视频监控技术与智能应用 [M]. 北京：人民邮电出版社，2013.

[3] 孟庆洪，侯宝稳 .ARM 嵌入式系统开发与编程 [M]. 北京：清华大学出版社，2011.

[4] 胡晓，高鹰，陈柏成 . 基于 GPRS 的远程图像监控系统设计 [J]. 广州大学学报：自然科学版，2009，8（1）：24-27.

[5] 张洁，何晓燕，凌志浩 . 基于 ARM 技术的远程图像监控系统设计 [J]. 自动化仪表，2006，27（11）：8-12.

作者简介：王哲梁 男，1989年出生，浙江乐清人，在读研究生。研究方向为微电子学与固体学。

（上接第 64 页）

值 = 0x00000

```
tss1：          .long 0
        .long krn_stk1，0x10
        .long 0，0，0，0，0
        .long task1，0x200
        .long 0，0，0，0
        .long usr_stk1，0，0，0
        .long 0x17，0x0f，0x17，0x17，0x17，0x17
        .long LDT1_SEL，0x8000000

        .fill 128，4，0              # 这是任务 1 的内核栈空间，其用户栈直接使用初始栈空间
    krn_stk1：

    # 下面是任务 0 和任务 1 的程序，它们分别循环显示字符“A”和“B”
    task0：
        movw $0x0c61，%ax            # 把需要显示的字符 A 放入 AL 寄存器中
        int $0x80
        movl $0xfff，%ecx            # 执行循环，起延时作用
    1：loop 1b
        jmp task0

    task1：
        movw $0x0d62，%ax
        int $0x80
        movl $0xfff，%ecx
    1：loop 1b
        jmp task1

        .fill 128，4，0              # 这是任务 1 的用户栈空间
    usr_stk1：
        .long 0
```

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
21. 基于 CAN 总线技术的嵌入式网关设计
22. Visual C 串行通讯控件使用方法与技巧的研究
23. IEEE1588 精密时钟同步关键技术研究
24. GPS 信号发生器射频模块的一种实现方案
25. 基于 CPCI 接口的视频采集卡的设计
26. 基于 VPX 的 3U 信号处理平台的设计
27. 基于 PCI Express 总线 1394b 网络传输系统 WDM 驱动设计

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
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
21. 基于 VxBus 的驱动程序架构分析

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
21. 基于嵌入式 Linux 操作系统内核实时性的改进方法研究
22. Linux TCP IP 协议详解

Windows CE:

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
8. 基于 MPC8247 嵌入式电力交换系统的设计与实现

9. 基于设备树的 MPC8247 嵌入式 Linux 系统开发
10. 基于 MPC8313E 嵌入式系统 UBoot 的移植

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
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
14. 基于 S3C6410 处理器的嵌入式 Linux 系统移植

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
10. 基于 COM Express 的回波预处理模块设计