嵌入式 Linux 知识培训

主要包括以下四部分内容：

一、嵌入式 **Linux** 开发的基本知识

二、 **Linux** 下使用 **C** 语言进行系统开发

三、面向嵌入式 **Linux** 的 **GUI** 系统的体系结构及二次开发

四、基于 **Linux OS Smart Phone** 的体系结构及开发内容

李玉东

第一部分

基础知识

嵌入式 Linux 软件系统的构成

1.BootLoader

2. Kernel

3.FileSystem

4.GUI

5.Application

嵌入式 Linux 系统的构成 (BootLoader)

1.PC 机的系统引导过程

系统加电　跳转到固定地址　**BIOS→MBR→Kernel**

2. 嵌入式系统中的 **bootloader** 与 **PC** 机的引导程序是类似的

系统加电　跳转到固定的地址　读入 **bootloader** 程序

初始化系统　加载 **kernel**，将系统的控制权转到 **kernel.**

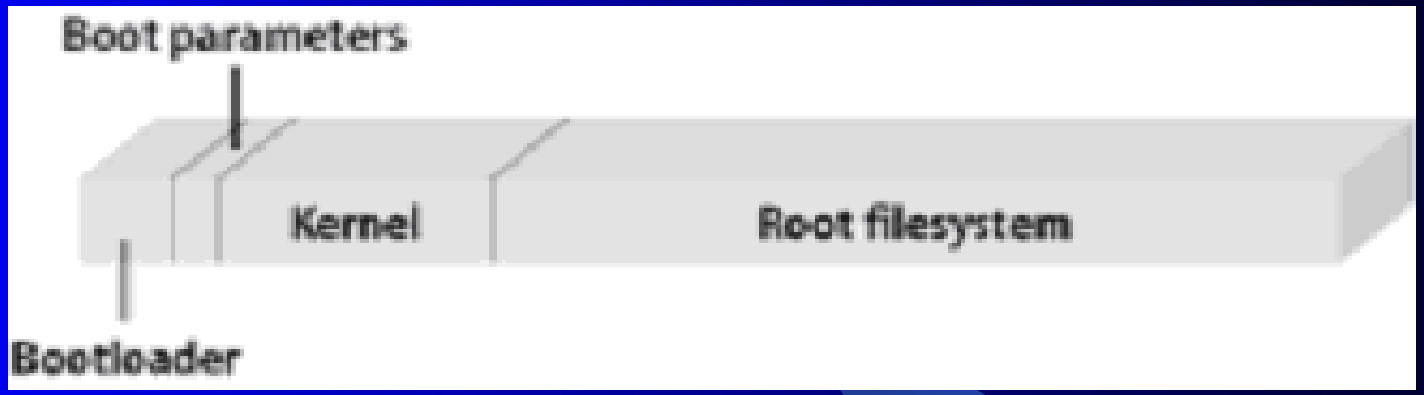
3.bootloader 与机器硬件紧密相关。

4. 有一定通用性的 **bootloader** 程序 **uboot**

http://sourceforge.net/projects/uboot

5.uboot 需要根据机器的情况进行修改

典型的空间分配结构



Creating 4 MTD partitions on "PXA Cerf Flash":
0x00000000-0x00040000 : "Bootloader"
0x00040000-0x000c0000 : "Partition Tables"
0x000c0000-0x001c0000 : "Kernel"
0x001c0000-0x02000000 : "Filesystem"

Multi Stage BootLoader

由于 Boot Loader 的实现依赖于 CPU 的体系结构，因此大多数 Boot Loader 都分为 stage1 和 stage2 两大部分。

依赖于 CPU 体系结构的代码，比如设备初始化代码等，通常都放在 stage1 中，而且通常都用汇编语言来实现，以达到短小精悍的目的。

而 stage2 则通常用 C 语言来实现，这样可以实现给复杂的功能，而且代码会具有更好的可读性和可移植性。

BootLoader Stage1

Boot Loader 的 stage1 通常包括以下步骤 (以执行的先后顺序) :

- 硬件设备初始化。
- 为加载 **Boot Loader** 的 **stage2** 准备 **RAM** 空间。
- 拷贝 **Boot Loader** 的 **stage2** 到 **RAM** 空间中。
- 设置好堆栈。
- 跳转到 **stage2** 的 **C** 入口点。

BootLoader Stage2

Boot Loader 的 **stage2** 通常包括以下步骤：

- 初始化本阶段要使用到的硬件设备。
- 检测系统内存映射 **(memory map)** 。
- 将 **kernel** 映像和根文件系统映像从 **flash** 上读到 **RAM** 空间中。
- 为内核设置启动参数。
- 调用内核。

BootLoader 调用内核

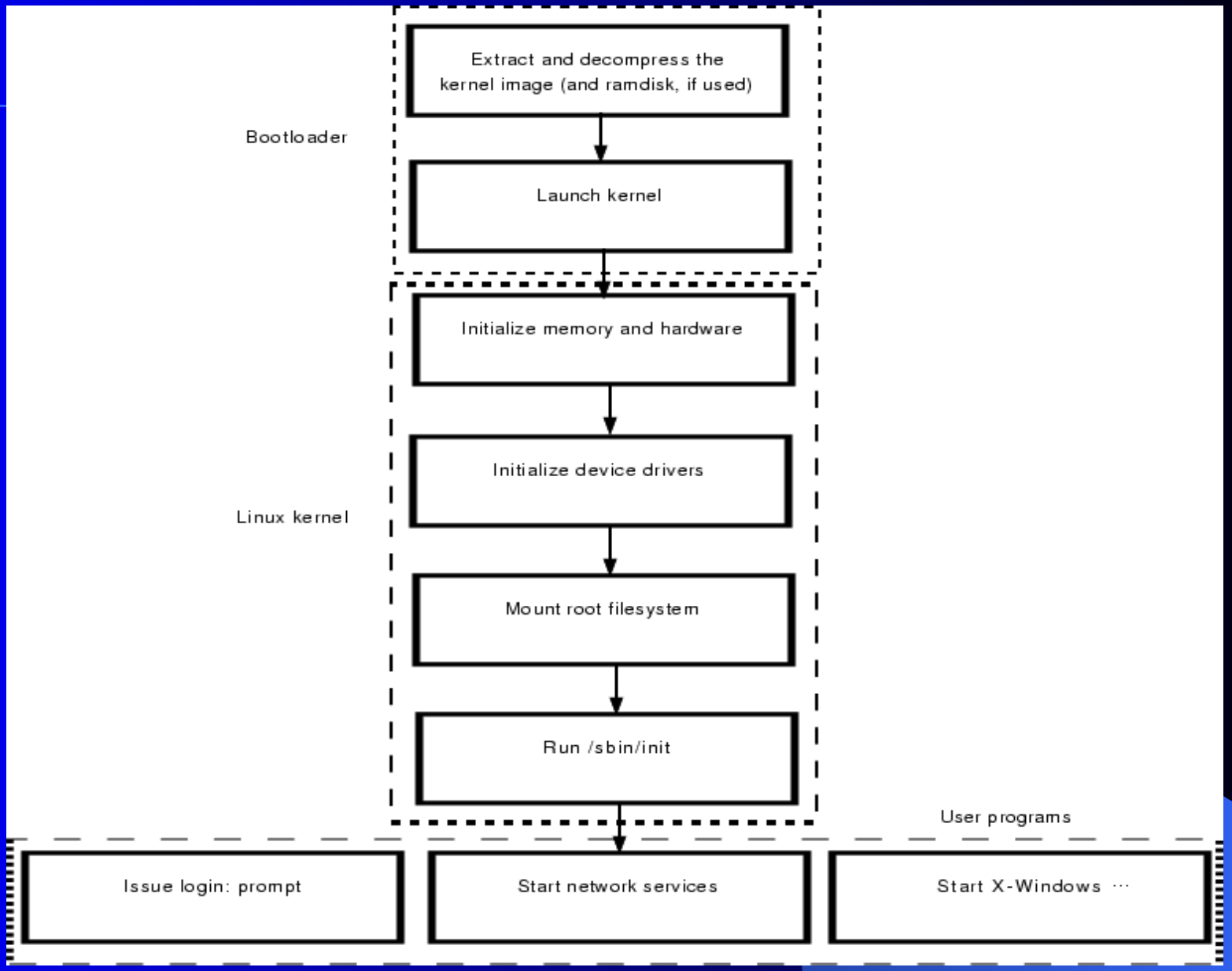
Boot Loader 调用 **Linux** 内核的方法是直接跳转到内核的第一条指令处开始执行

如果系统显示

Uncompressing Linux..... done, booting the kernel.

说明 **Bootloader** 已成功加载了 **kernel,Kernel** 开始启动

嵌入式 Linux 有关知识培训（基础部分）



Kernel 启动过程

检查 root device mount root file system.
如果 root file system 中， /linuxrc 存在的话，就会被执行。

/linuxrc 就是一般所说的 linux 的启动脚本

```
#!/bin/sh
export PATH=/bin:/sbin:/usr/bin
echo "Setting up RAMFS, please wait... "
mount tmpfs /mnt/ramfs -t tmpfs
tar zxvf /.ramfs.tar.gz -C /mnt/ramfs > /dev/null 2>&1
mount -n /proc /proc -t proc
echo -en "show the pictures "
/usr/local/bin/fbctl 2&            //setup background light
echo "done"
echo "Executing /sbin/init..."
exec /sbin/init
```

Kernel 启动过程

exec /sbin/init
Init 根据 **/etc/inittab** 中的内容启动相应的程序

这里有一个重要参数： **RunLevel**
例： **id:2:initdefault:**
则 **RunLevel** 为 **2**
执行的内容为： **/etc/rc2.d** 下面的内容

实际上 **rc*.d** 下的内容都是符号链接，都指向 **/etc/init.d** 下面的内容

Rc*.d 下的程序是按字母顺序执行的

开发环境的建立

服务器的安装

客户端需安装的程序

开发环境的建立

交叉编译器

为什么要安装交叉编译器

ARM-LINUX 交叉编译器的下载与安装

下载地址：

ftp://ftp.arm.linux.org.uk/pub/armlinux/toolchain/

2.6.6 内核编译器

http://www.scratchbox.org/index.html?id=4

内核的下载

1、内核的版本号
2、内核的下载地址
http://www.kernel.org/pub/linux/kernel/
3、 ARM Patch 的下载地址
ftp://ftp.arm.linux.org.uk/pub/armlinux/kernel/
4、 OMAP Patch 的下载地址
http://www.muru.com/linux/omap/
5、最新的 OMAP Kerenl 的下载方法
BitKeeper 工具
以下文档中有详细介绍
http://focus.ti.com/docs/apps/catalog/resources/appnoteabstract.jhtml?abstra ctName=swpa011

内核的解压及 Patch 的使用方法

以 **Innovator Development Kit for OMAP Platform** 为例：

1、解压内核包：

```
tar -xzvf  linux-2.4.19.tar.gz
```

2、使用 Patch

```
patch -p1 </root/patch-2.4.19-rmk7
patch -p1 </root/patch-2.4.19-rmk7-omap1
```

注：使用 Patch 以后将修改内核源代码中的部分内容

内核的编译

不同内核版本的编译方法有一些不同

1、以 2.4.18 的内核的编译为例：

```
make menuconfig
make dep
make clean
make zImage
make modules
make modules_install
```

2、P2 sample (OMAP 730) 内核的编译方法

```
cp
$(LinuxPath)/arch/arm/configs/omap_perseus2_730_defconfig
        $(LinuxPath)/.config
make old_config
make zImage
make modules
make modules_install INSTALL_MOD_PATH=<root fs>
```

内核的编译

内核生成后，位于以下的路径中：

$(linux_path)/arch/boot/zImage

可将生成的内核 **image** 文件直接下载到嵌入式开发板中

Linux 根文件系统 (Rootfs)

Linux 支持的文件系统包括：

ext2 、 ext3 、 vfat 、 iso9660 、 proc….

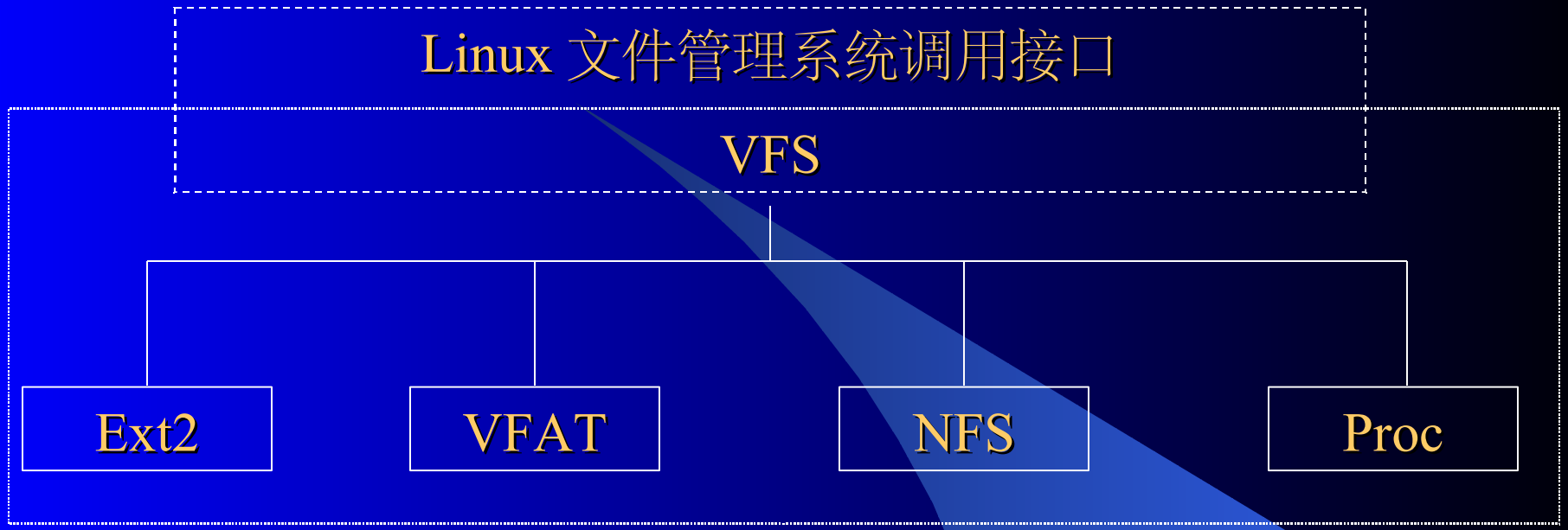
嵌入式 Linux 中常用的文件系统包括：

Cramfs　RamDisk

Jffs2　The Journalling Flash File System, version 2

http://sources.redhat.com/jffs2/

虚拟文件系统 VFS



系统启动过程中可以看到： **VFS: Mounted root (jffs2 filesystem)**

将 **jffs2** 文件系统 **mount** 到 **VFS**

虚拟文件系统 VFS

重新 Mount Root fs

在 **/etc/fstab** 文件中，列出了系统启动时自动 **mount** 的文件系统。

例如：

# <device>	<mountpoint>	<filesystemtype>	<options>	<dump>	<fsckorder>
/dev/mtdblock3	/	jffs2	defaults	1	1
ramdisk	mnt/ramdisk	tmpfs	size=10m	0	0
proc	/proc	proc	defaults	0	0

注：其中将 **jffs2** 文件系统 **mount** 到 **VFS** 的 “ /” 上，而对应的设备是：**/dev/mtdblock3**

设备驱动程序

设备驱动程序是操作系统内核和机器硬件之间的接口．设备驱动程序为应用程序屏蔽了硬件的细节，这样在应用程序看来，硬件设备只是一个设备文件，应用程序可以象操作普通文件一样对硬件设备进行操作

设备驱动程序是内核的一部分
设备驱动程序是在核心态执行的

驱动程序或者在编译内容时直接编译到内核中，或者在生成内核以后将驱动程序编译成模块，然后通过 **insmod** 加入到内核中

设备驱动程序一般结构

```
struct file_operations {
int (*seek) (struct inode * ,struct file *, off_t ,int);
int (*read) (struct inode * ,struct file *, char ,int);
int (*write) (struct inode * ,struct file *, off_t ,int);
int (*readdir) (struct inode * ,struct file *, struct dirent * ,int);
int (*select) (struct inode * ,struct file *, int ,select_table *);
int (*ioctl) (struct inode * ,struct file *, unsined int ,unsigned long
int (*mmap) (struct inode * ,struct file *, struct vm_area_struct *);
int (*open) (struct inode * ,struct file *);
int (*release) (struct inode * ,struct file *);
int (*fsync) (struct inode * ,struct file *);
int (*fasync) (struct inode * ,struct file *,int);
int (*check_media_change) (struct inode * ,struct file *);
int (*revalidate) (dev_t dev);
}
```

驱动程序的主要任务就是填充这个结构

设备驱动程序中两个重要函数

int init_module(void)

就是向系统的字符设备表登记了一个设备

当使用 **insmod** 加入模块时，该函数被调用

void cleanup_module(void)

它释放字符设备在系统设备表中占有的表项

在用 **rmmod** 卸载模块时， **cleanup_module** 函数被调用

设备驱动程序

编译设备驱动程序

```
gcc -DMODULE -D__KERNEL__ -c test.c
```

得到的 **test.o** 就是设备驱动程序

设备文件与设备号

设备被加入到系统中后，系统会为其分配设备号

在 **/proc/devices** 可以看到设备的设备号

通过 **mknod** 命令建立设备节点。

通过打开设备节点就可以操作对应的设备了

第二部分

C 语言开发

Linux 下 C 语言开发基本知识

编辑工具　**vi　kate**

编译、链接工具　**gcc**

Linux 下的器（　　C　言）是 **cc**，器是 **as**，接器是　**ld**

调试工具　**gdb**

举例

嵌入式 **Linux** 知识培训（ **C** 语言的开发）

Linux 下的高级开发技术

Makefile 的使用（举例）

动态链接库的生成及使用方法（举例）

多线程编程

线程的同步技术

嵌入式 **Linux** 知识培训（ **C** 语言的开发）

Linux 下的高级开发技术

线程的同步技术

互斥量（ **mutex)**
条件变量
守候条件变量
向条件变量发出信号
信号量（ **P,V** 操作）

Linux 下的高级开发技术

IPC（进程间通信）

消息队列
共享内存
管道
……

GUI 中通常使用的是 **domain_socket**

Linux 下的高级开发技术

关键设备的访问方法—— **framebuffer**

Framebuffer 简介

Linux 下的 **framebuffer** 是一个抽象的图形设备，它可以使应用程序直接访问图形硬件设备。

设备节点是 **/dev/fb***

面向嵌入式 **Linux** 的 **GUI** 系统基本上都是基于 **Framebuffer**

嵌入式 Linux 知识培训（ C 语言的开发）

```
BOOL InitFrameBuffer()
{
   _lGUI_iFrameBuffer = open ("/dev/fb0", O_RDWR);
   ioctl (_lGUI_iFrameBuffer, FBIOGET_VSCREENINFO,&_lGUI_vInfo);
   _lGUI_iFrameHeight=_lGUI_vInfo.yres;
   _lGUI_iFrameWidth=_lGUI_vInfo.xres;
   _lGUI_iLineSize = _lGUI_vInfo.xres * _lGUI_vInfo.bits_per_pixel / 8;
   _lGUI_iBufferSize = _lGUI_iLineSize * _lGUI_vInfo.yres;

   ioctl (_lGUI_iFrameBuffer, FBIOPAN_DISPLAY, &_lGUI_vInfo);
   _lGUI_pFrameBuffer =mmap (NULL, _lGUI_iBufferSize, PROT_READ |
PROT_WRITE,    MAP_SHARED, _lGUI_iFrameBuffer, 0);
   if(!_lGUI_pFrameBuffer){
      printerror("mmap return error.");
      return FALSE;
   }
   return TRUE;
}
```

嵌入式 **Linux** 知识培训（ **C** 语言的开发）

Linux 下的高级开发技术

关键设备的访问方法——串口操作

示例