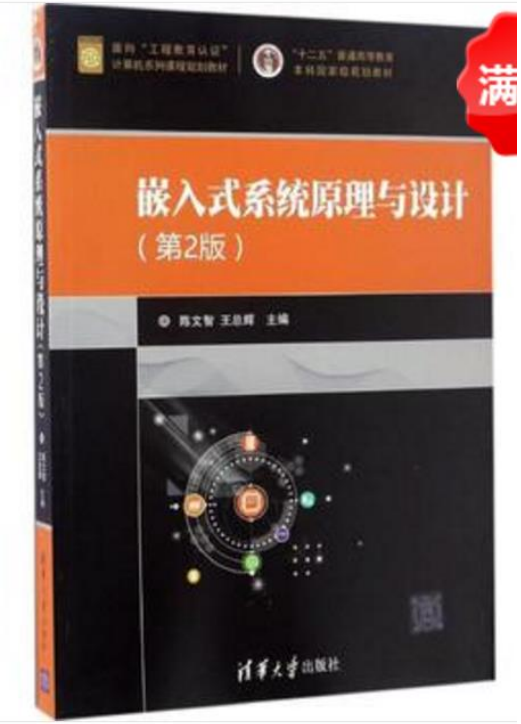
《嵌入式系统》

（总复习——理论部分）

厦门大学信息学院软件工程系 曾文华

2024年12月24日

- 第1章：嵌入式系统概述
- 第2章：ARM处理器和指令集
- 第3章：嵌入式Linux操作系统
- 第4章：嵌入式软件编程技术
- 第5章：开发环境和调试技术
- 第6章：Boot Loader技术
- 第7章：ARM Linux内核
- 第8章：文件系统
- 第9章：设备驱动程序设计基础
- 第10章：字符设备和驱动程序设计
- 第11章：Android操作系统（增加的）
- 第12章：块设备和驱动程序设计
- 第13章：网络设备驱动程序开发
- 第14章：嵌入式GUI及应用程序设计



第1章　嵌入式系统概述

- 1.1　嵌入式系统简介
- 1.2　嵌入式处理器
- 1.3　嵌入式操作系统
- 1.4　嵌入式系统设计

嵌入式处理器的分类

① 嵌入式微处理器：Embedded Microprocessor Unit，EMPU，如ARM系列嵌入式微处理器

② 嵌入式微控制器（单片机）：Microcontroller Unit，MCU，如8051单片机

③ 嵌入式DSP处理器（DSP）：Embedded Digital Signal Processor Unit，EDSP，数字信号处理器（DSP）

④ 嵌入式片上系统（SoC）：System on Chip，SoC，将整个嵌入式系统硬件（嵌入式处理器 + 嵌入式外围设备）集成在一个芯片上

ARM

- ARM： Advanced RISC Machines
- ARM公司：1991年成立于英国剑桥，32位嵌入式RISC微处理器业界的领先IP核供应商，ARM公司本身不生产芯片，而是通过转让设计方案由合作伙伴生产各具特色的芯片。
- ARM商品模式的强大之处在于它在世界范围有超过100个的合作伙伴(Partners)。ARM 是设计公司，本身不生产芯片。采用转让许可证制度，由合作伙伴生产芯片。
- IP（Intellectual Property）核就是知识产权核或知识产权模块的意思。

ARM处理器的典型产品

- **ARM Cortex-A**
  - ARM v8.2-A架构
    - Cortex-A9
    - Cortex-A53
    - Cortex-A72：RK3399（实验箱的MPU）
    - Cortex-A77（最新）
- **ARM Cortex-R**
  - ARM v8-R架构
    - Cortex-R52（最新）
- **ARM Cortex-M**
  - ARM v8-M架构
    - Cortex-M4：STM32F407（实验箱的MCU）
    - Cortex-M35P（最新）

ARM Cortex-A系列

- Cortex-A系列又称“高性能处理器”（Highest Performance），它是面向移动计算如智能手机、平板电脑和服务器市场定制的高端处理器内核，支持了包括Linux、Android、Windows和iOS等系统必须的内存管理单元（MMU），而且也是与我们平时接触最为密切的存在。

- Cortex-A系列面向尖端的基于虚拟内存的操作系统和用户应用。

- Cortex-A5、……、Cortex-A53、……、Cortex-A57、……、Cortex-A77

ARM Cortex-R系列

- ARM Cortex-R系列实时处理器（Real-Time Processing）为要求可靠性、高可用性、容错功能、可维护性和实时响应的嵌入式系统提供高性能计算解决方案。Cortex-R 系列处理器通过已经在数以亿计的产品中得到验证的成熟技术提供极快的上市速度，并利用广泛的 ARM 生态系统、全球和本地语言以及全天候的支持服务，保证快速、低风险的产品开发。

- ARM Cortex-R系列针对实时系统。

- Cortex-R4、Cortex-R5、……、Cortex-R52

ARM Cortex-M系列

- Cortex-M 系列针对成本和功耗敏感（Lowest Power, Lower Cost）的MCU和终端应用（如智能测量、人机接口设备、汽车和工业控制系统、大型家用电器、消费性产品和医疗器械）的混合信号设备进行过优化。

- Cortex-M系列针对微控制器。

- Cortex-M0、 Cortex-M0+、……、 Cortex-M35P

常见的嵌入式操作系统

① 嵌入式Linux
② VxWorks
③ μC/OS-II
④ Windows CE
⑤ Sysbian
⑥ Android
⑦ iOS
⑧ 其它：QNX，Palm OS，LynxOS，NucleusPLUS，ThreadX，eCos

Linux的发行版

- Linux 主要作为Linux发行版（通常被称为"distro"）的一部分而使用。这些发行版由个人，松散组织的团队，以及商业机构和志愿者组织编写。它们通常包括了其他的系统软件和应用软件，以及一个用来简化系统初始安装的安装工具，和让软件安装升级的集成管理器。大多数系统还包括了像提供GUI界面的XFree86之类的曾经运行于BSD的程序。 一个典型的Linux发行版包括：Linux内核，一些GNU程序库和工具，命令行shell，图形界面的X Window系统和相应的桌面环境，如KDE或GNOME，并包含数千种从办公套件，编译器，文本编辑器到科学工具的应用软件。

- 最受欢迎的10个Linux发行版：
  ① Ubuntu
  ② Fedora
  ③ OpenSUSE
  ④ Debian
  ⑤ Mandriva
  ⑥ Mint
  ⑦ PCLinuxOS
  ⑧ Slackware
  ⑨ Gentoo
  ⑩ CentOS

嵌入式应用软件开发

- 交叉开发
  - 宿主机/目标机模式：
    ① 宿主机：PC机（x86环境）
    ② 目标机：可以是实际的运行环境，也可以用仿真系统替代实际的运行环境（ARM环境）
  - 交叉开发环境包括：
    ① 交叉编译器
    ② 交叉调试器
    ③ 系统仿真器
  - 交叉开发环境的类型：
    ① 开放的：如GNU工具链
    ② 商业的：如Microsoft Visual Studio等

第2章　ARM处理器和指令集

- 2.1　ARM处理器简介
- 2.2　ARM指令集简介
- 2.3　ARM指令的寻址方式
- 2.4　ARM指令简介
- 2.5　Thumb指令简介

ARM处理器的两种状态

- ARM状态（32位），Thumb状态（16位）
- ARM指令集（32位），Thumb指令集（16位），Thumb指令集是ARM指令集的子集
- ARM指令必须在ARM状态下执行， Thumb指令必须在Thumb状态下执行
- ARM处理器可以在两种状态（ARM状态、Thumb状态）下进行切换；ARM子程序（ARM指令集编写的程序）和Thumb子程序（Thumb指令集编写的程序）之间可以进行相互调用

ARM处理器的寄存器

- ARM处理器的寄存器（37个32位的寄存器）
  - 31个通用寄存器
    - R0、R1、R2、R3、R4、R5、R6、R7　　8个
    - R8、R8_fiq　　2个
    - R9、R9_fiq　　2个
    - R10、R10_fiq　　2个
    - R11、R11_fiq　　2个
    - R12、R12_fiq　　2个
    - R13、R13_svc、R13_abt、R13_und、R13_irq、R13_fiq　　6个
    - R14、R14_svc、R14_abt、R14_und、R14_irq、R14_fiq　　6个
    - R15（PC）　　1个

    - R13：SP，堆栈指针，Stack Pointer
    - R14：LR，链接寄存器，Link Register
    - R15：PC，程序计数器，Program Counter

  - 6个专用状态寄存器
    - CPSR　　1个
    - SPSR_svc、SPSR_abt、SPSR_und、SPSR_irq、SPSR_fiq　　5个

    - CPSR：Current Program Status Register，当前程序状态寄存器
    - SPSR：Saved Program Status Register，程序状态保存寄存器

ARM处理器的运行模式

- ARM的处理器运行模式：
  1. 系统模式（SYS）
  2. 用户模式（USR）
  3. 快速中断模式（FIQ）
  4. 管理模式（SVC）
  5. 数据访问终止模式（ABT）
  6. 外部中断模式（IRQ）
  7. 未定义指令终止模式（UND，未定义模式）

- 特权模式（除用户模式外）：
  1. 系统模式（SYS）
  2. 快速中断模式（FIQ）
  3. 管理模式（SVC）
  4. 数据访问终止模式（ABT）
  5. 外部中断模式（IRQ）
  6. 未定义指令终止模式（UND，未定义模式）

- 异常模式（除系统模式、用户模式外）：
  1. 快速中断模式（FIQ）
  2. 管理模式（SVC）
  3. 数据访问终止模式（ABT）
  4. 外部中断模式（IRQ）
  5. 未定义指令终止模式（UND，未定义模式）

ARM指令寻址方式

- 9种寻址方式：
  ① 立即寻址
  ② 寄存器寻址
  ③ 寄存器偏移寻址
  ④ 寄存器间接寻址
  ⑤ 基址变址寻址
  ⑥ 多寄存器寻址
  ⑦ 堆栈寻址
  ⑧ 相对寻址
  ⑨ 块复制寻址

ARM指令类型

- 指令类型（8类）：
  ① 跳转指令
  ② 通用数据处理指令
  ③ 乘法指令
  ④ Load/Store内存访问指令
  ⑤ ARM协处理器指令
  ⑥ 杂项指令
  ⑦ 饱和算术指令
  ⑧ ARM伪指令

ARM指令格式

- 指令格式：<opcode> {<cond>} {S} <Rd>, <Rn> {, <shift_op2>}
  - < >内的项是必须的，{ }内的项是可选的
  - opcode：指令助记符（操作码），如LDR，STR 等
  - cond：执行条件（条件码），如EQ，NE 等
  - S：可选后缀，加S时影响CPSR中的条件码标志位，不加S时则不影响
  - Rd：目标寄存器
  - Rn：第1个源操作数的寄存器
  - op2：第2个源操作数
  - shift：位移操作

条件码

- 条件码（（<cond>）：CPSR[31:28]　N Z C V）的16种组合：

AMR指令条件码	助记符	描述	CPSR条件码标志位的值
0000	EQ	相等，运行结果为0	Z置位
0001	NE	不相等，运行结果不为0	Z清零
0010	CS/HS	无符号数大于等于	C置位
0011	CC/LO	无符号数小于	C清零
0100	MI	负数	N置位
0101	PL	非负数	N清零
0110	VS	上溢出	V置位
0111	VC	没有上溢出	V清零
1000	HI	无符号数大于	C置位且Z清零
1001	LS	无符号数小于等于	C清零且Z置位
1010	GE	带符号数大于等于	N=V
1011	LT	带符号数小于	N!=V
1100	GT	带符号数大于	Z清零且N=V
1101	LE	带符号数小于等于	Z置位且N!=V
1110	AL	无条件执行	
1111	系统保留		

位移操作

- 位移操作<shift_op2>的11种形式：

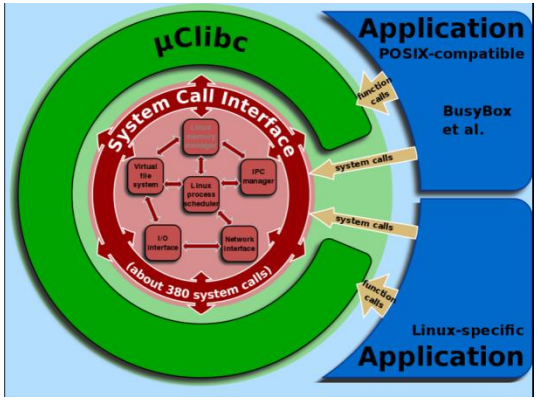
语法	含义
#<immediate>	立即数寻址
<Rm>	寄存器寻址
<Rm>, LSL #<shift_imm>	立即数逻辑左移
<Rm>, LSL <Rs>	寄存器逻辑左移
<Rm>, LSR #<shift_imm>	立即数逻辑右移
<Rm>, LSR <Rs>	寄存器逻辑右移
<Rm>, ASR #<shift_imm>	立即数算术右移
<Rm>, ASR <Rs>	寄存器算术右移
<Rm>, ROR #<shift_imm>	立即数循环右移
<Rm>, ROR <Rs>	寄存器循环右移
<Rm>, RRX	寄存器扩展循环右移

第3章　嵌入式Linux操作系统

- 3.1　嵌入式Linux简介
- 3.2　内存管理
- 3.3　进程管理
- 3.4　文件系统

µCLinux

- µCLinux
  - µ：Micro，微
  - C：Control，控制



  - µCLinux的特点：
    ① 针对微控制领域设计的Linux系统。
    ② 编译后的目标文件可控制在几百KB大小。
    ③ 专门针对没有MMU（存储管理单元）的处理器设计的。
    ④ 无法使用虚拟内存管理技术，采用实存储器管理策略，所有程序访问的地址都是实际的物理地址。

MMU：Memory Management Unit，内存管理单元

RT-Linux

- **RT-Linux（硬实时操作系统）**
  - RT-Linux（Real-time Linux）：具有硬实时特性的多任务操作系统，是美国新墨西哥州立大学计算机科学系Victor Yodaiken和Micae Brannanov开发的。
  - 通过在Linux内核与硬件中断之间增加一个精巧的可抢先的实时内核，把标准的Linux内核作为实时内核的一个进程与用户进程一起调度，标准的Linux内核的优先级最低，可以被实时进程抢断。
  - 正常的Linux进程仍可以在Linux内核上运行，这样既可以使用标准分时操作系统（即Linux的各种服务），又能提供低延时的实时环境。

红旗嵌入式Linux

- 红旗嵌入式Linux 

  – 北京红旗软件技术有限公司推出的，国内做得较好的嵌入式Linux操作系统。

  – 红旗嵌入式Linux的特点：
    ① 精简内核，适用于多种常见的嵌入式处理器。
    ② 提供完善的嵌入式GUI和嵌入式X-Windows。
    ③ 提供嵌入式浏览器、邮件程序和多媒体播放程序。
    ④ 提供完善的开发工具和平台。

内存管理和MMU

- 内存管理和MMU：
  - 存储管理（内存管理）包括：
    ① 地址映射
    ② 内存空间的分配
    ③ 地址访问的限制（即保护机制）
    ④ I/O地址的映射
    ⑤ 代码段、数据段、堆栈段空间的分配
  - MMU（Memory Management Unit）的主要作用：
    ① 地址映射
    ② 对地址访问的保护和限制（访问保护）
  - MMU就是提供一组寄存器，依靠这组寄存器来实现地址映射和访问保护；MMU可以做在CPU芯片中，也可以作为协处理器（以协处理器的形式实现）。

标准Linux的内存管理

- 标准Linux的内存管理：
  - 标准Linux使用虚拟存储器技术
  - 虚拟存储器：由存储器管理机制（MMU）及一个大容量的快速硬盘存储器支持
  - 分页：将实际的存储器分割为相同大小的段，例如每个段为1024KB，即将1024KB大小的段称为一个页面
  - 虚拟地址
  - 物理地址
  - MMU：将虚拟地址映射为物理地址
  - ld文件：链接脚本文件

μCLinux的内存管理

- μCLinux的内存管理：
  - μCLinux没有MMU，而是采用实存储器管理（Real Memory Management）。
  - μCLinux不能使用虚拟内存管理技术（虚拟存储器技术），但仍然采用存储器的分页管理。
  - μCLinux将整个物理内存划分为若干个4KB大小的页面。

进程和进程管理

- 进程和进程管理：
  - 进程：进程是一个运行程序并为其提供执行环境的实体，它包括一个地址空间和至少一个控制点，进程在这个地址空间上执行单一指令序列。进程地址空间包括可以访问或引用的内存单元的集合，进程控制点通过程序计数器（PC）控制和跟踪进程指令序列。
  - 任务调度：主要是协调任务对计算机系统内资源（如内存、I/O设备）的争夺使用。
  - 进程调度：又称为CPU调度，其根本任务是按照某种原则为处于就绪状态的进程分配CPU。

进程调度策略

- 进程调度策略（两种）：

① 抢占式调度：一旦就绪状态中出现优先权更高的进程，或者运行的进程已用满了规定的时间片时，便立即抢占当前进程的运行（将其放回就绪状态），把CPU分配给其他进程。

② 非抢占式调度：一旦某个进程被调度执行，该进程一直执行下去，直至该进程结束，或者由于某种原因自行放弃CPU进入等待状态，才将CPU重新分配给其他进程。

Linux进程的调度

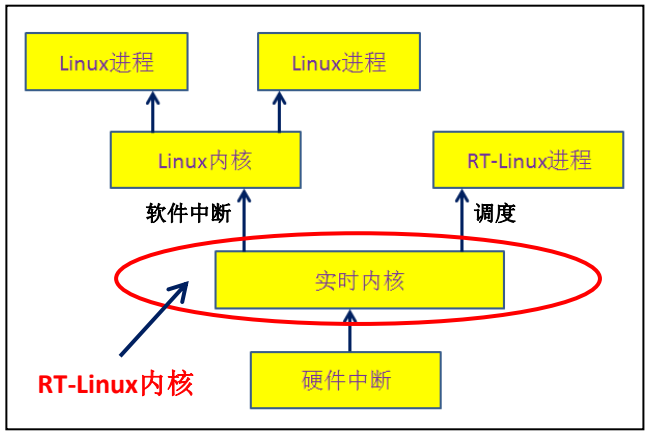
- Linux进程的调度：
  - Linux操作系统进程调度策略（三种）：
    ① 分时调度策略：SCHED_OTHER
    ② 先到先服务的实时调度策略：SCHED_FIFO
    ③ 时间片轮转的实时调度策略：SCHED_RR
  - 实时进程：将得到优先调用，并根据实时优先级决定调度权限。
  - 分时进程：通过nice和counter值决定权值，nice越小、counter越大，被调用的概率越大，即曾经使用CPU最少的进程将会得到优先调度。

RT-Linux的进程管理

- **RT-Linux的进程管理：**

  - **RT-Linux有两种中断：**
    ① **硬中断：是实现实时Linux内核的前提。**
    ② **软中断：是常规Linux内核中断。**

  - **RT-Linux的系统结构：**
    - **在Linux内核和硬件中断之间增加了一个实时内核（RT-Linux内核），该实时内核是一个高效的、可抢占的实时调度核心。**



RT-Linux的系统结构

μCLinux的进程管理

- μCLinux的进程管理：
  - 由于没有MMU，在实现多个进程时需要实现数据保护。
  - 由于没有MMU，系统虽然支持fork系统调用，但实际上是vfork系统调用。
  - 启动新的应用程序时，系统必须为应用程序分配存储空间，并立即将应用程序加载到内存。
  - 必须在可执行文件加载阶段对可执行文件Relocation处理，使得程序执行时能够直接使用物理内存。
  - 操作系统对内存空间没有保护，各个进程实际上共享一个运行空间，会导致用户程序使用的空间可能占用到系统内核空间，容易导致系统崩溃。

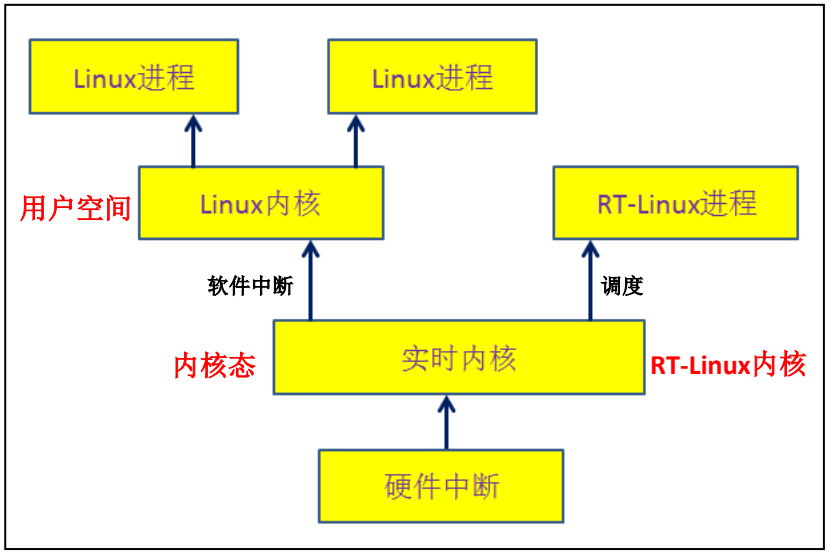
μCLinux进程管理的特点

用户空间和内核态

- RT-Linux程序运行于用户空间和内核态两个空间：

① 实时处理部分编写成内核模块，并装载到RT-Linux内核中，运行于RT-Linux的内核态。

② 非实时部分的应用程序则在Linux下的用户空间中执行。



Linux文件系统

- **Linux文件系统：**

① ext（Extended File System）：扩展文件系统

② ext2（Extended File System 2）：扩展文件系统（版本2）

③ ext3（Extended File System 3）：扩展文件系统（版本3）

④ JFS2（Journal File System 2）：日志文件系统（版本2）

⑤ XFS：一种高性能的日志文件系统， 是 Silicon Graphics公司于 90 年代初开发的文件系统

⑥ ReiserFS：一种新型的文件系统，它支持海量磁盘和磁盘阵列，并能在上面继续保持很快的搜索速度和很高的效率

⑦ CFS：加密文件系统

⑧ VFS：虚拟文件系统

嵌入式Linux文件系统

- 嵌入式Linux文件系统：
  - 嵌入式系统很少使用IDE硬盘，而是选用Flash Memory（闪存）代替硬盘。
  - 1、Flash Memory（闪存）简介
    - 属于Non-Volatile内存（非挥发性内存，即断电信息不丢失）。
  - 2、Flash Memory上的两种技术NOR和NAND

① NOR Flash：Intel于1988年首先开发出NOR Flash技术（采用或非门），彻底改变了原先由EPROM和EEPROM一统天下的局面。

② NAND Flash：1989年，东芝公司发表了NAND Flash结构（采用与非门），强调降低每比特的成本，更高的性能，并且象磁盘一样可以通过接口轻松升级。

NAND Flash和NOR Flash的比较

	1989年东芝公司	1988年Intel公司
	NAND Flash	NOR Flash
芯片容量	<32GBit	<1GBit
访问方式	顺序读写	随机读写
接口方式	任意I/O口	特定完整存储器接口
读写性能	读取快（顺序读） 写入快 擦除快（可按块擦除）	读取快（RAM方式） 写入慢 写入慢
使用寿命	百万次	十万次
价格	低廉	高昂

容量大
价格低
顺序读写

容量小
价格高
随机读写

嵌入式文件系统分类

- 嵌入式文件系统分类：
  - 嵌入式系统主要的存储设备为：
    ① **Flash Memory**：闪存

    ② **SDRAM**：Synchronous Dynamic Random Access Memory，同步动态RAM
  - 嵌入式系统主要的文件系统类型：
    ① JFFS2
    ② YAFFS2
    ③ Cramfs
    ④ Romfs
    ⑤ RamDisk
    ⑥ Ramfs/Tmpfs

基于Flash Memory的文件系统

- 基于Flash Memory的文件系统（NAND Flash）：

容量大
价格低
顺序读写

① JFFS2文件系统
- ✓ Journalling Flash File System 2（闪存日志型文件系统第2版）
- ✓ 主要用于NOR Flash

② YAFFS2文件系统
- ✓ Yet Another Flash File System 2，是一种和JFFS类似的闪存文件系统
- ✓ 主要用于NAND Flash

③ Cramfs文件系统
- ✓ Compressed ROM File System，一种只读的压缩文件系统

④ Romfs文件系统
- ✓ ROM File System，一种简单的只读文件系统

⑤ 其他文件系统
- ✓ FAT/FTA32：主要是为了与Windows系统兼容
- ✓ ext2：是GNU/Linux系统中的标准文件系统，特点是存取文件的性能好，具有存取文件的性能极好，但是用于Flash上会有很多弊端

基于RAM的文件系统

- 基于RAM的文件系统（SDRAM）：

① RamDisk

✓ RamDisk（虚拟内存盘），是通过软件将一部分内存(RAM)模拟为硬盘来使用的一种技术。相对于直接的硬盘文件访问来说，这种技术可以极大的提高在其上进行的文件访问的速度。但是RAM的易失性也意味着当关闭电源后这部分数据将会丢失。但是在一般情况下，传递到RAM盘上的数据都是在硬盘或别处永久贮存的文件的一个拷贝。经由适当的配置，可以实现当系统重启后重新建立虚拟盘。

② Ramfs/Tmpfs

✓ Ramfs（基于RAM的文件系统）是Linus Torvalds开发的一种基于内存的文件系统，工作于虚拟文件系统(VFS)层，不能格式化，可以创建多个，在创建时可以指定其最大能使用的内存大小。

✓ Tmpfs（临时文件系统），是一种基于内存的文件系统，它和虚拟磁盘RamDisk比较类似像，但不完全相同，和RamDisk一样，Tmpfs可以使用RAM，但它也可以使用swap分区来存储，而且传统的RamDisk是个块设备，要用mkfs来格式化它，才能真正地使用它；而Tmpfs是一个文件系统，并不是块设备，只是安装它，就可以使用了。Tmpfs是最好的基于RAM的文件系统。

网络文件系统（NFS）

- 网络文件系统（NFS）：
  - NFS：Network File System（网络文件系统），是FreeBSD支持的文件系统中的一种，它允许网络中的计算机之间通过TCP/IP网络共享资源。在NFS的应用中，本地NFS的客户端应用可以透明地读写位于远端NFS服务器上的文件，就像访问本地文件一样。
  - NFS至少包括两个主要部分：
    ① 1台服务器
    ② 1台（或多台）客户机

第4章　嵌入式软件编程技术

- 4.1　嵌入式编程基础
- 4.2　嵌入式汇编编程技术
- 4.3　嵌入式高级编程技术
- 4.4　高级语言与汇编语言混合编程

可（不可）重入函数

- 可（不可）重入函数：
  - 可重入函数：可以由多个任务并发使用，而不必担心数据出错
  - 不可重入函数：不能由多个任务所共享，除非能确保函数的互斥

可重入函数（Reentrant）主要用于多任务环境中，一个可重入的函数简单来说就是可以被中断的函数，也就是说，可以在这个函数执行的任何时刻中断它，转入OS调度下去执行另外一段代码，而返回控制时不会出现什么错误。

而不可重入的函数（Non-reentrant）由于使用了一些系统资源，比如全局变量区，中断向量表等，所以它如果被中断的话，可能会出现问题，这类函数是不能运行在多任务环境下的。

可（不可）重入函数例子

例子1：

```
static int tmp;

void swap(int *x, int *y){

        tmp = *x;

        *x = *y;

        *y = tmp;
}
```

例子2：

```
void swap2(int *x, int *y){

        int tmp;

        tmp = *x;

        *x = *y;

        *y = tmp;
}
```

swap是不可重入的

有静态变量：tmp

swap2是可重入的

没有静态变量

中断

- 中断：
  - 硬中断：计算机外设引起的事件
  - 软中断：SWI指令（软中断指令）引起的中断
  - 异常：CPU在运行过程中由其本身引起的事件（如被0除、遇到未定义的指令等）

中断处理过程

- 中断处理过程包括硬件和软件两部分：
  - 由硬件完成的工作：
    - ① 复制CPSR到SPSR_<MODE>（如复制CPSR到SPSR_irq）
    - ② 设置正确的CPSR位
    - ③ 切换到<MODE>
    - ④ 保存返回地址到LR_<MODE>（如LR_irq，即R14_irq）
    - ⑤ 设置PC跳转到相应的异常向量表入口

MODE：ARM处理器的运行模式

  - 由软件（中断服务程序）完成的工作：
    - ① 把SPSR和LR压栈
    - ② 把中断服务程序的寄存器压栈
    - ③ 开中断，允许嵌套中断
    - ④ 中断服务程序执行完毕后，恢复寄存器
    - ⑤ 弹出SPSR和PC，恢复执行

Makefile文件

- GNU make：编译工具
- Makefile文件：make工具读取的文件
- make工作过程
  - Makefile文件的基本规则：

```
target : prereg1 prereg2 prereg3
        command
```

    - target：规则的目标
    - prereg1、prereg2、prereg3：规则的依赖
    - command：规则执行的命令，该命令行必须通过Tab键进行缩进

GNU汇编语言语句格式

- **GNU汇编语言语句格式：**
  - **[<label>:][<instruction or directive or pseudo-instruction>] @comment**
    - ① **<label>:** 标号
    - ② **instruction** 指令
    - ③ **directive** 伪操作
    - ④ **pseudo-instruction** 伪指令
    - ⑤ **@comment** 语句的注释

汇编语言程序设计举例（1）

- 例1：20的阶乘
  - 20 X 19 X 18 X ……X 2 X 1 = 2,432,902,008,176,640,000
  - 用ARM汇编语言设计程序实现求 20！（20的阶乘），并将其64位的结果放在[R9:R8]中（R9中放置高32位，R8中放置低32位）

    $2^{64}-1$ = 18,446,744,073,709,551,615
  - 程序设计思路：64位结果的乘法指令通过两个32位的寄存器相乘，可以得到64位的结果，在每次循环相乘中，我们可以将存放64位结果两个32位寄存器分别与递增量相乘，最后将得到的高32位结果相加。

- 程序代码如下：

伪操作

```
.global _start                  @声明全局变量“_start”
.text                           @代码段
_start:
    MOV R8, #20                 @低32位初始化为20
    MOV R9, #0                  @高32位初始化为0     [R9:R8]=20
    SUB R0, R8, #1              @初始化计数器     R0=19
Loop:
    MOV R1, R9                  @暂存高位值
    UMULL R8, R9, R0, R8        @[R9：R8]=R0*R8
    MLA R9, R1, R0, R9          @R9=R1*R0+R9
    SUBS R0, R0, #1             @计数器递减
    BNE Loop                    @计数器不为0，继续循环
Stop:
    B Stop
.end                            @源文件结束（程序结束）
```

伪操作

汇编语言程序设计举例（2）

- 例2：设计一段程序完成数据块的复制，数据从源数据区复制到目标数据区，要求以4个字为单位进行复制，最后所剩不到4个字的数据，以字为单位进行复制。

  - 程序代码如下：

NUM = 18

```
.global _start          @声明全局变量“_start”
equ NUM, 18             @设置要拷贝的字数
.text                   @代码段
.arm                    @ARM程序
_start:
```

```
LDR   R0,   =SRC              @R0指向源数据区起始地址
LDR   R1,   =DST              @R1指向目的数据区起始地址
MOV   R2,   #NUM              @ R2存放待复制数据量大小，
                                以字为单位
MOV   SP, #0X9000             @堆栈指针指向0X9000，
                                堆栈增长模式由装载指令的
                                类型域确定
MOV   R3,   R2,   LSR   #2    @ 将R2中值除以4后的结果存放在R3，
                                R3中值表示NUM中有多少个4字单元
BEQ   COPY_WORDS              @ 若Z=1(R3=0，数据少于1个4字单元)，
                                则跳转到COPY_WORDS处，
                                运行少于4字单元数据处理程序
STMFD   SP!,   {R5-R8}        @保存R5-R8的内容到堆栈，并更新栈指针，
                                FD:满递减堆栈，由此可知堆栈长向
```

```
COPY_4WORD:
    LDMIA   R0!,  {R5-R8}      @从R0所指的源数据区装载4个字数据到R5-R8中，
                                  每次装载1个字后R0中地址加1，
                                  最后更新R0中地址
    STMIA   R1!,  {R5-R8}      @将R5-R8的4个字数据存入R1所指的目的数据区，
                                 每次装载1个字后R1中地址加1，
                                 最后更新R1中地址
    SUBS    R3,   R3,   #1     @每复制一次，则R3=R3-1，
                                 表示已经复制了1个4字单元，结果影响CPSR
    BNE COPY_4WORD             @ 若CPSR的Z=0(即运算结果R3不等于0)，
                                  跳转到COPY_4WORD，
                                  继续复制下一个4字单元数据
    LDMFD   SP!,  {R5-R8}      @ 将堆栈内容恢复到R5-R8中，并更新堆栈指针，
                                  此时整4字单元数据已经复制完成，
                                  且出栈模式应和入栈模式一样
```

```
COPY_WORDS:
    ANDS   R2,R2,   #3          @得到NUM除以4后余数，
                                 即未满4字单元数据的字数(1个字=4个字节)
    BEQ    STOP                 @若R2=0(NUM有整数个4字单元)，
                                  则停止复制
COPY_WORD:
    LDR    R3,   [R0],   #4     @将R0所指源数据区的4个字节(1个字)
                                  数据装载至R3，然后R0=R0+4
    STR    R3,   [R1],   #4     @将R3中4个字节(1个字)数据存到
                                  R1所指目的数据区，然后R1=R1+4
    SUBS   R2,   R2,   #1       @数据传输控制计数器减1(其总是小于4)，
                                  成功复制一个字数据
    BNE    COPY_WORD            @ 若R2不等于0，则转到WORDCOPY，
                                  继续复制下一个字数据
```

第一次复制：1,2,3,4　第二次复制：5,6,7,8　第三次复制：9, 0xa,0xb,0xc　第四次复制：0xd,0xe,0xf,0x10

第五次复制：0x11　第六次复制：0x12

伪操作

```
STOP:
    B   STOP

.ltorg                          @声明一个数据缓冲池的开始，
                                  一般在代码的最后面
SRC:
        .long   1,2,3,4,5,6,7,8,9,0xa,0xb,0xc,
                0xd,0xe,0xf,0x10,0x11,0x12      @18个源数据
DST:
        .long   0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0     @18个目的数据

.end                            @程序结束
```

函数可（不可）重入（1）

- 针对函数可重入问题，有以下几种优化解决方案：
  - 1、将全局变量或静态变量改成局部变量
    - 优化前：

```
static int tmp;
void swap(int *x, int *y){
        tmp = *x;
        *x = *y;
        *y = tmp;
}
```

不可重入

    - 优化后：

```
void swap(int *x, int *y){
        int tmp;
        tmp = *x;
        *x = *y;
        *y = tmp;
}
```

可重入

函数可（不可）重入（2）

- 2、采用信号量进行临界资源保护
  - 优化前：

```
static int tmp;
void swap(int *x, int *y){
        tmp = *x;
        *x = *y;
        *y = tmp;
}
```

不可重入

  - 优化后：

```
static int tmp;
void swap(int *x, int *y){
        [申请信号量操作]
        tmp = *x;
        *x = *y;
        *y = tmp;
        [释放信号量操作]
}
```

可重入

函数可（不可）重入（3）

- 3、禁止中断
  - 优化前：

不可重入

```
static int tmp;
void swap(int *x, int *y){
        tmp = *x;
        *x = *y;
        *y = tmp;
}
```

  - 优化后：

可重入

```
static int tmp;
void swap(int *x, int *y){
        Disalble_IRQ();
        tmp = *x;
        *x = *y;
        *y = tmp;
        Enable_IRQ();
}
```

禁止中断

允许中断

中断处理过程

- 中断处理过程：
  - __irq：中断处理函数声明关键字
    - ✓ irq：外部中断模式
  - 中断处理函数的C语言代码

```
__irq void IRQHandler(void)
{
        volatile unsigned int *source = (unsigned int*) 0x80000000;
        if(*source == 1)
                int_hander_1();                    //中断服务程序
        *source = 0;
}
```

中断处理函数

- 中断处理函数的汇编代码：

```
STMFD sp!, {r0-r4, r12, lr}
MOV r4, #0x80000000
LDR r0, [r4,#0]
CMP r0,#1
BLEQ int_hander_1
MOV r0, #0
STR r0, [r4, #0]
LDMFD sp!, {r0-r4, r12, lr}
SUBS pc, lr, #4
```

r0 = 1 则转中断服务程序

@转中断服务程序入口

高级语言与汇编语言混合编程

- 大部分程序采用嵌入式C语言编写
- 对硬件相关的操作、中断处理、对性能要求比较高的模块，需要用汇编语言编写程序
- C语言调用汇编语言
- 汇编语言调用C语言
- **ATPCS**：过程调用标准（ARM-THUMB Procedure Call Standard）
- **AAPCS**：新的过程调用标准（ARM Architecture Produce Call Standard）

汇编语言调用C语言

- 汇编语言调用C语言：
  - 汇编程序的设计要遵守ATPCS（过程调用标准），保证程序调用时参数的正确传递。在汇编程序中使用IMPORT伪操作声明将要调用的C程序，然后通过BL指令调用C函数。
  - IMPORT伪操作告诉编译器当前的符号不是在本源文件中定义的，而是在其他源文件中定义的，在本源文件中可能引用该符号，而且不论本源文件是否实际引用该符号，该符号都将被加入到本源文件的符号表中。

– 例子：

C语言程序

```
int add(int x, int y)
{
        return(x+y);
}
```

IMPORT伪操作告诉编译器当前的符号不是在本源文件中定义的，而是在其他源文件中定义的

汇编语言调用C语言程序

```
IMPORT add          @声明要调用的C函数
MOV r0, 1
MOV r1,2            @通过r0、r1传递参数（参数传递规则）
BL add              @调用C函数add；返回结果由r0带回
                      （子程序返回结果规则）
```

程序执行完后，r0 = 1+2 = 3

C语言调用汇编语言

- C程序调用汇编程序：
  - 两种形式：
    ① 嵌入式汇编
    ② 内联汇编

– 1、嵌入式汇编

- 在汇编程序中使用EXPORT伪指令声明被调用的子程序，表明该子程序将在其他文件中被调用
- 在C程序中使用extern关键字声明要调用的汇编子程序为外部函数

- EXPORT伪指令用于程序中声明一个全局的标号，该标号可在其他的文件中引用
- extern关键字可置于变量或者函数前，以表示变量或者函数的定义在别的文件中

嵌入式汇编

- 例子：

汇编语言程序

– 汇编语言：

```
EXPORT add        @声明add子程序将被外部函数调用
add:
        ADD r0,r0,r1
        MOV pc,lr
```

C语言调用汇编语言程序

– C语言：

```
extern int add(int x, int y);              //声明add为外部函数
void main()
{
        int a=1, b=2, c;
        c = add(a, b);
}
```

– 2、内联汇编

- 内联汇编：即将汇编语句直接写在C程序中
- 内联汇编的形式1：

C语言调用汇编语言程序

内联汇编形式1

```
void enable_IRQ(void)
{
        int tmp;

        __asm                          //声明内联汇编代码
        {
            MRS tmp, CPSR
            BIC    tmp, tmp, #0x80
            MSR CPSR_c, tmp
        }

}
```

- 内联汇编的形式2：
  - 格式：

```
__asm(
    code
    : 输出操作数列表
    : 输入操作数列表
    : clobber列表（破坏列表）
    );
```

内联汇编形式2

  - 例子：

输入操作数列表

```
__asm( "mov %0, %1, ror #1" : "=r" (result) : "r" (value) );
```

code（代码）

输出操作数列表

– 调用例子：

C语言调用汇编语言程序

内联汇编形式2

```
#include <stdio.h>
int main(void)
{
        int result,value;

        value = 1;

        printf("old value is %x",value);

        __asm( "mov %0, %1, ror #1" : "=r" (result) : "r" (value) );

        printf("new value is %x\n",result);

        return 1;
}
```

result

value

mov result, value, ror #1

00000001H循环右移1位，结果为：10000000H

第5章　开发环境和调试技术

- 5.1　交叉开发模式概述
- 5.2　宿主机环境
- 5.3　目标板环境
- 5.4　交叉编译工具链
- 5.5　本地调试（gdb调试器）
- 5.6　远程调试（gdb+gdbserver调试器）

交叉开发模式

- 交叉开发模式：宿主机（PC机：VMware下的Ubuntu）-目标板（IMX6实验箱）

- GNU软件：
  ① Shell：Shell基本上是一个命令解释器，类似于DOS下的command
  ② glibc：glibc是GNU发布的libc库，即c运行库
  ③ GCC：GCC（GNU Compiler Collection，GNU编译器套件）是由 GNU 开发的编程语言编译器
  ④ gdb： UNIX及UNIX-like下的调试工具
  ⑤ vim：vim是一个类似于Vi的著名的功能强大、高度可定制的文本编辑器，在vi的基础上改进和增加了很多特性
  ⑥ Emacs： Emacs是著名的集成开发环境和文本编辑器

vim hello.c　　vi hello.c

- 宿主机与目标板的连接方式：
  ① 串口（COM1 TO USB）
  ② 以太网接口（RJ45）
  ③ USB接口
  ④ JTAG接口（Joint Test Action Group）

串口终端

- 串口终端：
  - Windows下的超级终端
    - 超级终端是Windows自带的一个串口调试工具，其使用较为简单，被广泛使用在串口设备的初级调试上，如Xshell 4。
  - Linux下的minicom
    - minicom是一个串口通信工具，就像Windows下的超级终端。可用来与串口设备通信，如调试交换机和Modem等。它的Debian软件包的名称就叫minicom，用apt-get install minicom命令即可下载安装。

BOOTP

- **BOOTP：**
  - BOOTP（Bootstrap Protocol，引导程序协议）是一种引导协议，基于IP/UDP协议，也称自举协议，是DHCP协议的前身。BOOTP用于无盘工作站的局域网中，可以让无盘工作站从一个中心服务器上获得IP地址。通过BOOTP协议可以为局域网中的无盘工作站分配动态IP地址，这样就不需要管理员去为每个用户去设置静态IP地址。
  - BOOTP的一般工作流程就是BOOTP客户端（目标板，实验箱）和BOOTP服务器（宿主机，Ubuntu）之间的交互，其流程如下：
    ① 由BOOTP启动代码来启动BOOTP客户端，这个时候BOOTP客户端还没有IP地址。
    ② BOOTP客户端使用广播形式的IP地址255.255.255.255向网络中发出IP地址查询要求。
    ③ 运行BOOTP协议的服务器接收到这个请求，会根据请求中提供的MAC地址找到BOOTP客户端，并发送一个含有IP地址、服务器IP地址、网关等信息的回应帧。
    ④ BOOTP客户端会根据该回应帧来获得自己的IP地址并通过专用文件服务器（如TFTP服务器）下载启动镜像文件，模拟成磁盘来完成启动。

TFTP

- **TFTP：**
  - TFTP（Trivial File Transfer Protocol，简单文件传输协议）是TCP/IP协议族中的一个用来在客户机与服务器之间进行简单文件传输的协议，提供不复杂、开销不大的文件传输服务。
  - TFTP是简化了的FTP，TFTP没有用户权限管理的功能。

交叉编译

- 交叉编译：
  - 交叉编译：在x86架构的宿主机（PC机：VMware下的Ubuntu）上编译生成适用于ARM架构（IMX6实验箱）的ELF格式的可执行代码。
  - ELF：Executable and Linkable Format，可执行与可链接格式，是一种用于二进制文件、可执行文件、目标代码、共享库和核心转储格式文件。

JTAG接口

- JTAG接口简介：
  - JTAG（Joint Test Action Group，联合测试工作组）是一种国际标准测试协议（IEEE 1149.1兼容），主要用于芯片内部测试。现在多数的高级器件都支持JTAG协议，如DSP、FPGA器件等。标准的JTAG接口是5线：TMS、TCK、TDI、TDO、nTRST，分别为模式选择、时钟、数据输入、数据输出线、系统复位。

Boot Loader

- **Boot Loader简介：**
  - 嵌入式Linux系统启动后，先执行Bootloader，进行硬件和内存的初始化工作，然后加载Linux内核和根文件系统映像，完成Linux系统的启动
  - **Bootloader：** 引导加载程序，是嵌入式目标板（实验箱）加电后运行的第一段软件代码；是在操作系统内核运行之前用来初始化硬件设备、建立内存空间的映射图的小程序

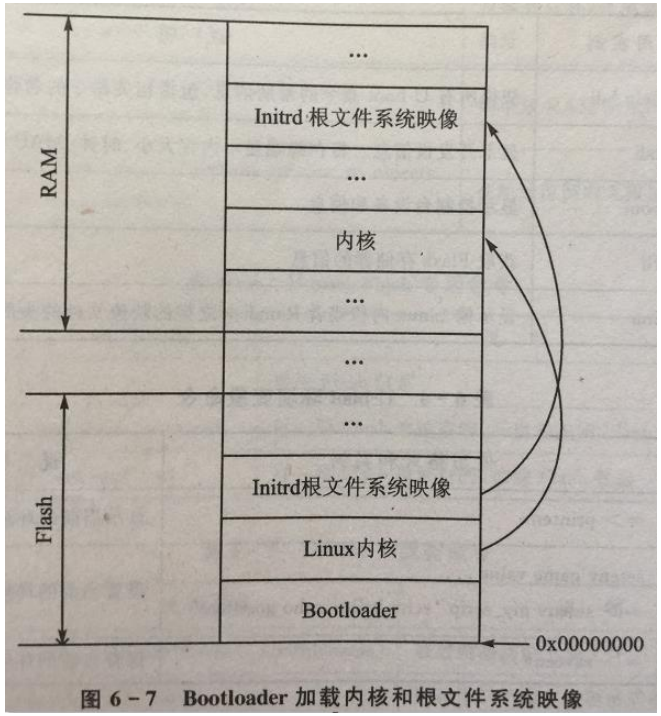


图 6－7　Bootloader 加载内核和根文件系统映像

Bootloader加载内核和根文件系统映像

常见的Bootloader

- 常见的Bootloader：
  - u-boot：Universal Boot Loader，是遵循GPL条款的开放源码项目，u-boot的作用是系统引导
  - vivi：韩国Mizi公司开发的Bootloader引导程序

交叉编译工具链

- 本地编译：在PC机上，编译生成PC平台运行的程序
- 交叉编译：在PC机上，编译生成目标机（ARM，实验箱）平台运行的程序
- 交叉编译工具链：是一个由编译器、链接器、解释器组成的集成开发环境

交叉编译的构建

- 交叉编译的构建：
  - ARM平台的交叉编译工具：aarch64-linux-gnu-gcc
  - 制作交叉编译工具链（是一个由标准库、编译器、链接器、汇编器、调试器组成的集成开发环境）的方法：
    ① 从头编译
    ② 脚本编译
    ③ 下载使用

本地调试（gdb）

- gdb：GNU Debugger

- 本地调试：
  - 调试ARM可执行文件（实验箱上运行的可执行文件）
    - 在“串口超级终端Xshell 4”上运行：gdb
  - 调试x86可执行文件（Ubuntu上运行的可执行文件）
    - 在Ubuntu的“终端”上运行：gdb

远程调试（gdb+ gdbserver）

- 远程调试（用于调试ARM程序）：
  - 在实验箱的“Xshell 4超级终端”上运行：gdbserver
    - 方法一：挂载方式。ARM可执行文件通过挂载（mount）方式共享到实验箱。
    - 方法二：下载方法。ARM可执行文件需要预先下载到实验箱。
  - 在Ubuntu的“终端”上运行：arm-linux-gdb（gdb）

第6章　Boot Loader技术

- 6.1　Boot Loader基本概念
- 6.2　Boot Loader典型结构
- 6.3　U-Boot简介
- 6.4　vivi简介

Boot Loader基本概念

- **Boot Loader**（引导程序）是在操作系统内核运行之前运行的一段小程序，**Boot Loader**初始化硬件设备和建立内存空间的映射图，从而将系统的软硬件环境带到一个合适的状态，以便为最终调用操作系统内核准备好正确的环境。

- 嵌入式**Linux**系统启动后，先执行**Boot Loader**，进行硬件和内存的初始化工作，然后加载**Linux**内核和根文件系统，完成**Linux**系统的启动。

Bootloader加载内核和根文件系统映像

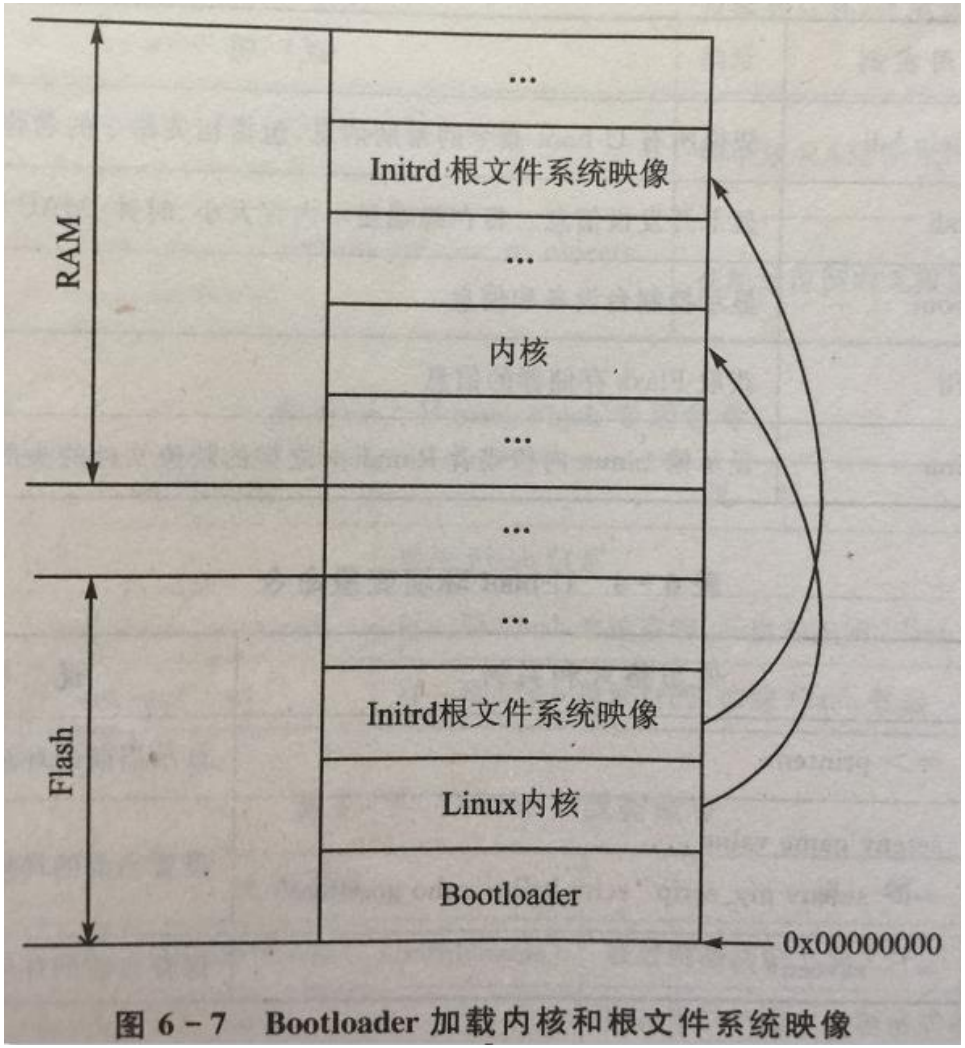


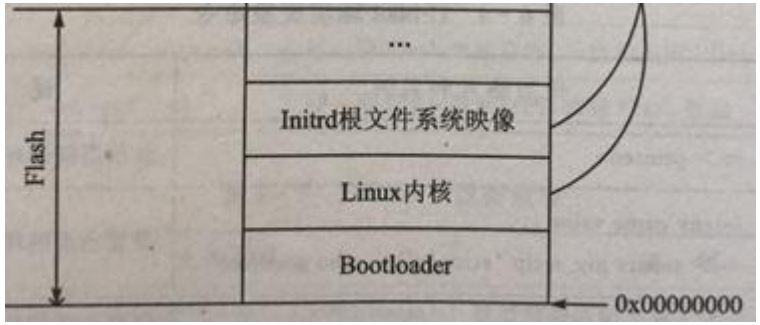
图 6－7　Bootloader 加载内核和根文件系统映像

固态存储器的典型空间分配结构

- **Boot Loader**的安装地址（即**ARM**处理器的复位启动地址）：**0x00000000**

- 固态存储器（**Flash**存储器）的典型空间分配结构：
  - ① **Boot Loader**
  - ② 内核的启动参数（**Boot parameters**）
  - ③ 内核映像
  - ④ 根文件系统映像

Linux内核



Boot Loader的操作模式

- Boot Loader的两种模式：
  - 启动加载模式：也称为自主模式，也即Boot Loader从目标机（实验箱）上的某个固态存储设备（通常为Flash存储器）上将操作系统加载到RAM（通常为SDRAM）中运行，整个过程并没有用户的介入。这种模式是Boot Loader的正常工作模式。
  - 下载模式：在这种模式下目标机（实验箱）上的Boot Loader将通过串口连接或网络连接等通信手段从主机（宿主机，Ubuntu）下载文件（内核映像、根文件系统映像），从主机下载的文件通常首先被Boot Loader保存到目标机的RAM（SDRAM）中，然后再被Boot Loader写（烧写）到目标机上的固态存储设备（Flash存储器）中。

常用的Boot Loader

- 常用的Boot Loader：

① **U-Boot**：全称 Universal Boot Loader，是由开源项目PPCBoot发展起来的，ARMboot并入了PPCBoot，和其他一些arch的Loader合称U-Boot。

② **vivi**：是韩国mizi 公司开发的Boot Loader，适用于ARM9处理器。vivi有两种工作模式：启动加载模式和下载模式。启动加载模式可以在一段时间后（这个时间可更改）自行启动Linux内核，这是vivi的默认模式。在下载模式下，vivi为用户提供一个命令行接口，通过接口可以使用vivi提供的一些命令。

③ **Blob**：全称Boot Loader Object，是由Jan-Derk Bakker和Erik Mouw发布的，是专门为StrongARM 构架下的LART设计的Boot Loader。

Boot Loader与主机之间的通信设备及协议

- **Boot Loader与主机之间的通信设备及协议：**
  - 串口：目标机使用串口与主机相连，这时的传输协议通常是xmodem/ymodem/zmodem中的一种。
  - 网口：目标机使用网口与主机相连，用网络连接的方式传输文件，这时使用的协议多为TFTP。

Boot Loader典型结构

- Boot Loader的阶段1：主要包含依赖于CPU的体系结构硬件初始化的代码，通常都用汇编语言来实现。这个阶段的任务有（5个任务）：
  ① 基本的硬件设备初始化（屏蔽所有的中断、关闭处理器内部指令/数据Cache等）
  ② 为加载Boot Loader的阶段2准备RAM空间
  ③ 复制Boot Loader的阶段2代码到RAM
  ④ 设置堆栈
  ⑤ 跳转到阶段2的C程序入口点

- Boot Loader的阶段2：通常用C语言完成，以便实现更复杂的功能，也使程序有更好的可读性和可移植性。这个阶段的任务有（5个任务）：
  ① 初始化本阶段要使用到的硬件设备
  ② 检测系统内存映射
  ③ 将内核映像和根文件系统映像从Flash读到RAM
  ④ 为内核设置启动参数
  ⑤ 调用内核

U-Boot简介

- 认识U-Boot：
  - U-Boot，全称 Universal Boot Loader（通用的引导程序），是遵循GPL条款的开放源码项目。U-Boot的作用是系统引导。U-Boot从FADSROM、8xxROM、PPCBOOT逐步发展演化而来。其源码目录、编译形式与Linux内核很相似，事实上，不少U-Boot源码就是根据相应的Linux内核源程序进行简化而形成的，尤其是一些设备的驱动程序，这从U-Boot源码的注释中能体现这一点。
  - http://ftp.denx.de/pub/u-boot/（下载地址）
  - u-boot-1.3.4.tar.bz2 2008-08-12 16:23 7.4M（压缩文件：1.3.4版本）
  - u-boot-latest.tar.bz2 2017-09-11 20:11 11M（压缩文件：最新版本）
  - u-boot-latest.tar.bz2.sig 2017-09-11 20:12 543（签名验证文件：最新版本）

U-Boot代码结构分析

- **U-Boot代码结构分析：**
  - **U-boot 2017.03的目标结构：**
    - ① **api：存放u-boot提供的接口函数**
    - ② **arch：与体系结构相关的代码**
    - ③ **board：根据不同开发板所定制的代码**
    - ④ **common：通用的代码，涵盖各个方面，已对命令行的处理为主**
    - ⑤ **disk：磁盘分区相关代码**
    - ⑥ **doc：文档，readme**
    - ⑦ **drivers：驱动相关代码，每种类型的设备驱动占用一个子目录**
    - ⑧ **dts：设备树文件**
    - ⑨ **examples：示例程序**
    - ⑩ **fs：文件系统，支持嵌入式开发板常见的文件系统**
    - ⑪ **include：头文件，以通用的头文件为主**
    - ⑫ **lib：通用库文件（14个）**
    - ⑬ **Licenses：许可证**
    - ⑭ **nand_spl：nand存储器相关的代码**
    - ⑮ **net：网络相关的代码，小型的协议栈**
    - ⑯ **post：上电自检程序**
    - ⑰ **scripts：脚本文件**
    - ⑱ **spl：secondary program loader的简称，第二阶段程序加载器**
    - ⑲ **test：测试文件**
    - ⑳ **tools：辅助功能程序，用于制作u-boot镜像等**

U-boot两阶段代码

- ARM926 EJ-S系列处理器的U-boot两阶段代码：
  - 第一阶段（用汇编语言编写）：start.S
  - 第二阶段（用C语言编写）：board.c

vivi简介

- 认识vivi：
  - vivi是韩国mizi公司设计的一款主要针S3C2410平台的Boot Loader，其特点是体积小，功能强大，运行效率高和使用方便。vivi代码虽然比较小巧，但麻雀虽小，五脏俱全，用来学习bootloader还是不错的。
  - 代码在http://download.csdn.net/detail/yu4700/4388601可以下载到。

vivi的两阶段代码

- vivi的两阶段代码：
  - 第一阶段（用汇编语言编写）：head.S
    - vivi/arch/s3c2410/head.S
  - 第二阶段（用C语言编写）：main.c
    - vivi/init/main.c

第7章　ARM-Linux内核

- 7.1　ARM-Linux内核简介
- 7.2　ARM-Linux内存管理
- 7.3　ARM-Linux进程管理和调度
- 7.4　ARM-Linux模块机制
- 7.5　ARM-Linux系统启动和初始化

ARM-Linux内核简介

- 内核：是一个操作系统的核心。是基于硬件的第一层软件扩充，提供操作系统的最基本的功能，是操作系统工作的基础，它负责管理系统的进程、内存、设备驱动程序、文件和网络系统，决定着系统的性能和稳定性。现代操作系统设计中，为减少系统本身的开销，往往将一些与硬件紧密相关的（如中断处理程序、设备驱动程序等）、基本的、公共的、运行频率较高的模块（如时钟管理、进程调度等）以及关键性数据结构独立开来，使之常驻内存，并对他们进行保护。通常把这一部分称之为操作系统的内核。

- Linux内核： Linux内核的主要模块（或组件）分以下几个部分：存储管理、CPU和进程管理、文件系统、设备管理和驱动、网络通信，以及系统的初始化（引导）、系统调用等。

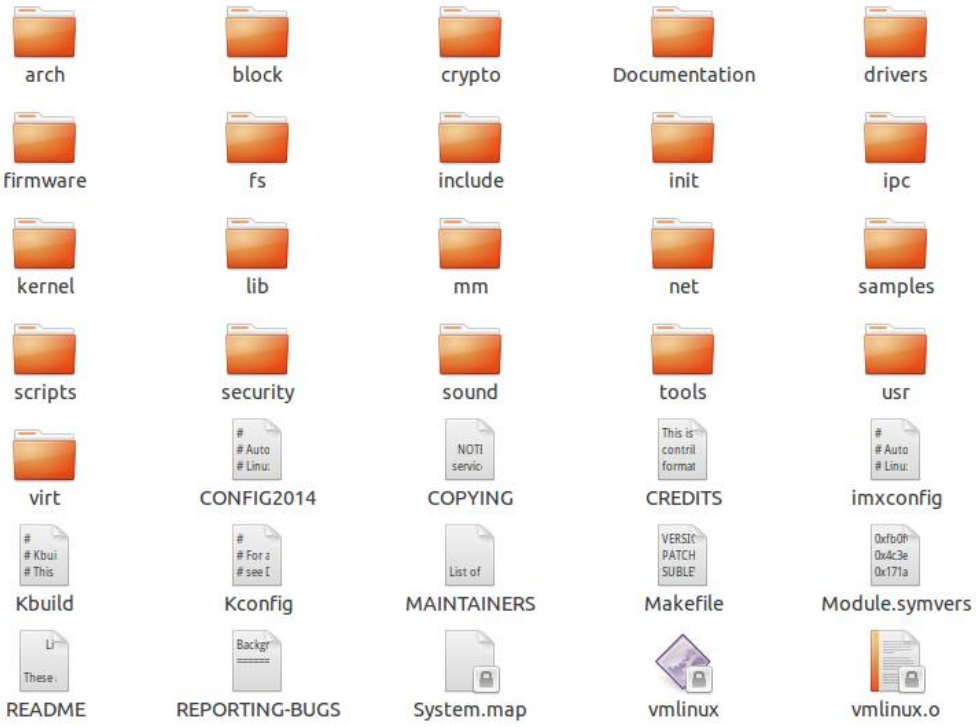
- ARM-Linux内核：基于ARM处理器的Linux内核。

ARM-Linux内核和普通Linux内核的区别

- ARM-Linux内核和普通Linux内核的区别：
  - 相对于ARM Linux，我们说的普通Linux指的是X86 Linux，他们都是Linux系统，但是由于ARM和X86是不同的CPU架构，他们的指令集不同，所以软件编译环境不同，软件代码一般不能互用，一般需要进行兼容性移植。
  - X86是经典的CISC指令集，指令集复杂，功能多，串行执行，但是也意味着执行效率低下，但性价比突出，所以称为民用终端的主流处理器内置指令集。Intel和AMD的家用处理器都是X86指令集。以X86为代表的CISC，理论并发线程1-2条。
  - ARM是Advanced RISC Machine 的缩写。它的指令集比RISC还要精简。通常使用ARM架构处理器的机型，多为嵌入式或者便携机。主频通常不高，现在高通公司的ARM架构处理器有1.0GHz的，已经算相当高了。另外，ARM 7沿用冯·诺依曼结构；而从ARM 9以后，就都采用了哈佛结构。ARM的并发线程，理论上有4条左右，处理效率较X86高不少。

ARM-Linux的代码结构

- **ARM-Linux的代码结构：**
  - 位于：**/home/linux/workdir/fs3399/system/kernel**目录下



ARM-Linux内存管理

- 影响内存管理的两个方面：
  - Linux内核对内存的管理（Linux操作系统的内存管理）
    - 内存管理是操作系统必不可少也是非常重要的一部分，包括：
      ① 地址映射
      ② 内存空间的分配
      ③ 地址访问的限制（即保护机制）
      ④ I/O地址的映射（I/O编址与内存编址相同）
  - ARM体系结构对内存的管理
    - MMU（存储器管理单元），主要作用有两个方面：
      ① 地址映射
      ② 对地址访问进行保护和限制
    - MMU可以做在CPU芯片中，也可以作为一个协处理器（用协处理器实现）

ARM-Linux的存储机制

- **ARM-Linux的存储机制：**
  - 基于x86体系结构的Linux内核的存储空间
    - 32位地址形成4GB的虚拟地址空间，被分为两部分：
      ① 内核空间（系统空间）：位于高端的1GB，属于Linux操作系统
      ② 用户空间：位于低端的3GB，属于应用程序
  - ARM-Linux内核的存储空间
    - 32位地址形成4GB的虚拟地址空间，也被分为两部分，但是内核空间（系统空间）和用户空间的具体划分，可以因CPU芯片和开发板（实验箱）而有所不同
    - 另外，ARM将I/O也放在内存地址空间中

虚拟内存

- 虚拟内存：
  - 虚拟内存（虚拟存储器）：是计算机系统内存管理的一种技术，它使得应用程序认为它拥有连续的可用的内存（一个连续完整的地址空间），而实际上，它通常是被分隔成多个物理内存碎片，还有部分暂时存储在外部磁盘存储器上，在需要时进行数据交换
  - Linux虚拟内存的实现需要6种机制的支持：
    ① 地址映射机制
    ② 请求页机制
    ③ 内存分配回收机制
    ④ 缓存和刷新机制
    ⑤ 交换机制
    ⑥ 内存共享机制

ARM-Linux进程管理和调度

- 进程：也称为任务，是一个动态的执行过程，是处于执行期的程序，进程是系统资源分配的最小单位。
- 狭义定义：进程是正在运行的程序的实例。
- 广义定义：进程是一个具有一定独立功能的程序关于某个数据集合的一次运行活动。它是操作系统动态执行的基本单元，在传统的操作系统中，进程既是基本的分配单元，也是基本的执行单元。
- 进程的概念主要有两点：
  ① 进程是一个实体。每一个进程都有它自己的地址空间，一般情况下，包括文本区域（text region）、数据区域（data region）和堆栈（stack region）。文本区域存储处理器执行的代码；数据区域存储变量和进程执行期间使用的动态分配的内存；堆栈区域存储着活动过程调用的指令和本地变量。

  ② 进程是一个“执行中的程序”。程序是一个没有生命的实体，只有处理器赋予程序生命时（操作系统执行之），它才能成为一个活动的实体，我们称其为进程。

ARM-Linux模块机制

- Linux是单内核的，单内核最大的优点是效率高，因为所有的内容都集中在一起，单内核也有可扩展性以及可维护性差的缺点。
  - 单内核，是个很大的进程。它的内部又能够被分为若干模块（或是层次或其他）。但是在运行的时候，它是个单独的二进制大映象。其模块间的通讯是通过直接调用其他模块中的函数实现的，而不是消息传递。单内核结构的例子：传统的UNIX内核----例如伯克利大学发行的版本，Linux内核。
  - 微内核结构由一个非常简单的硬件抽象层和一组比较关键的原语或系统调用组成，这些原语仅仅包括了建立一个系统必需的几个部分，如线程管理，地址空间和进程间通信等。微内核的例子：AIX，BeOS，L4微内核系列，.Mach中用于GNU Hurd和Mac OS X，Minix，MorphOS，QNX，RadiOS，VSTa。
  - 混合内核它很像微内核结构，只不过它的的组件更多的在核心态中运行，以获得更快的执行速度。混合内核实质上是微内核，只不过它让一些微核结构运行在用户空间的代码运行在内核空间，这样让内核的运行效率更高些。混合内核的例子： BeOS 内核 ，DragonFly BSD，ReactOS 内核，Windows NT、Windows 2000、Windows XP、Windows Server 2003以及Windows Vista等基于NT技术的操作系统。
  - 外内核系统，也被称为纵向结构操作系统，是一种比较极端的设计方法。 外内核这种内核不提供任何硬件抽象操作，但是允许为内核增加额外的运行库，通过这些运行库应用程序可以直接地或者接近直接地对硬件进行操作。外核设计还停留在研究阶段，没有任何一个商业系统采用了这种设计。几种概念上的操作系统正在被开发，如剑桥大学的Nemesis，格拉斯哥大学的Citrix系统和瑞士计算机科学院的一套系统。麻省理工学院也在进行着这类研究。
- 模块机制的引入就是为了弥补这一缺点。模块（内核模块，动态可加载内核模块，Loadable Kernel Module，LKM）：是Linux内核向外部提供的一个插口。

Linux模块概述

- **Linux模块概述：**
  - Linux内核支持动态可加载模块，模块通常是设备驱动程序。
  - 与模块相关的命令：
    ① **insmod**：加载模块
    ② **rmmod**：卸载模块
    ③ **lsmod**：列出已经安装的模块
    ④ **depmod**：产生模块依赖的映射文件
    ⑤ **modprob**：根据depmod命令所产生的相依关系，决定要载入哪些模块

内核映像载入

- 使用Boot Loader将内核映像载入：
  - Boot Loader将Linux的内核加载到内存（SDRAM）后，将跳到函数start_kernel()进入初始化过程
  - start_kernel()函数：位于/home/linux/workdir/fs3399/system/kernel/init/main.c中

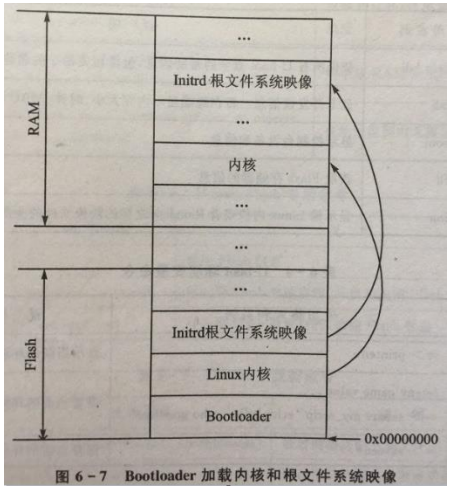
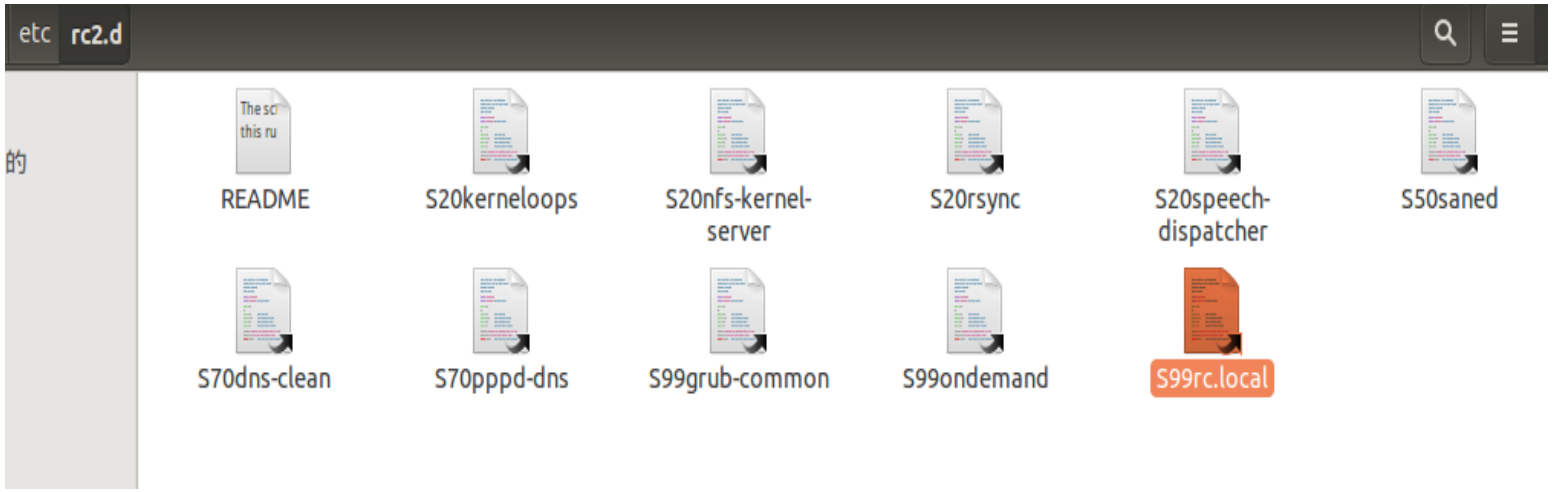


图 6－7　Bootloader 加载内核和根文件系统映像

rc启动脚本

- **rc启动脚本：**
  - **Linux系统运行后将启动rc脚本（S99rc.local）**
  - **rc启动脚本（S99rc.local）位于/etc/rc2.d/目录下**



实验箱的rc启动脚本

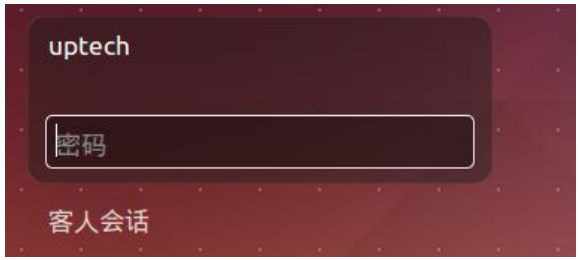
S99rc.local

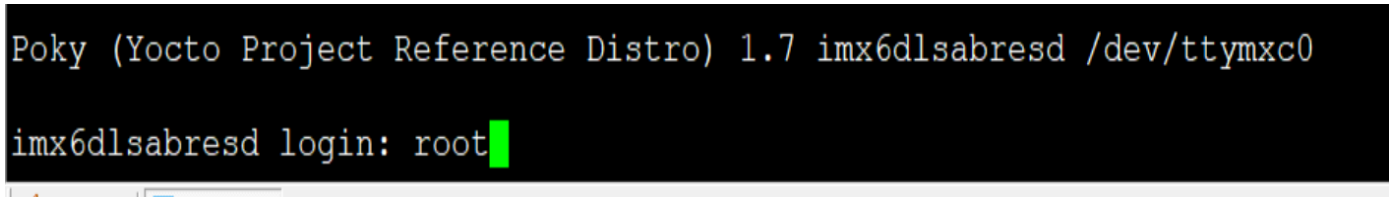
```
#! /bin/sh
### BEGIN INIT INFO
# Provides:        rc.local
# Required-Start:    $all
# Required-Stop:
# Default-Start:     2 3 4 5
# Default-Stop:
# Short-Description: Run /etc/rc.local if it exist
### END INIT INFO

PATH=/sbin:/usr/sbin:/bin:/usr/bin

do_start() {
                        if [ -x /etc/rc.local ]; then
                                        echo -n "Running local boot scripts (/etc/rc.local)"
                                        /etc/rc.local
                                        [ $? = 0 ] && echo "." || echo "error"
                                        return $ES
                        fi
}

case "$1" in
  start)
                        do_start
    ;;
  restart|reload|force-reload)
    echo "Error: argument '$1' not supported" >&2
    exit 3
    ;;
  stop)
    ;;
  *)
    echo "Usage: $0 start|stop" >&2
    exit 3
    ;;
esac
/home/root/imx6_V1_0 -platform eglfs -plugin evdevtouch:/dev/input/event0 evdevinput:/dev/input/event4&
```

/home/root/userver.sh

Shell的启动

- **Shell的启动：**
  - Login用户将启动一个用户指定的Shell，这个指定的Shell就是/bin/bash
    - Shell：俗称壳（用来区别于核），是指“为使用者提供操作界面”的软件（命令解析器）。
  - bash：是一个为GNU计划编写的Unix shell。它的名字是一系列缩写：Bourne-Again SHell — 这是关于Bourne shell（sh）的一个双关语（Bourne again / born again）。Bourne shell是一个早期的重要shell，由史蒂夫·伯恩在1978年前后编写，并同Version 7 Unix一起发布。bash则在1987年由布莱恩·福克斯创造。在1990年，Chet Ramey成为了主要的维护者。





第8章　文件系统

- 8.1　嵌入式文件系统简介
- 8.2　嵌入式Linux文件系统框架
- 8.3　JFFS2嵌入式文件系统
- 8.4　根文件系统

Linux文件系统简介

- Linux文件系统简介：
  - 各操作系统使用的文件系统并不相同，例如，Windows 98 以前的微软操作系统使用 FAT（FAT16）文件系统，Windows 2000 以后的版本使用 NTFS 文件系统，而 Linux 的正统文件系统是 Ext2
    - FAT：File Allocation Table，文件配置表
      - FAT16使用了16位的空间来表示每个扇区(Sector)配置文件的情形，故称之为FAT16
    - NTFS： New Technology File System，新技术文件系统
    - Ext2：second Extended file system，第二代扩展文件系统

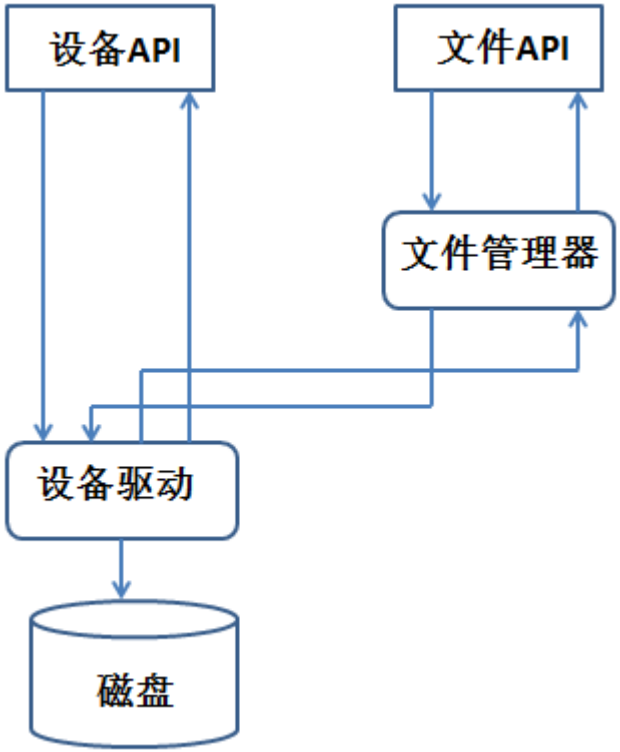
Linux支持的常见文件系统

文件系统	描述
Ext	Linux 中最早的文件系统，由于在性能和兼容性上具有很多缺陷，现在已经很少使用
Ext2	是 Ext 文件系统的升级版本，Red Hat Linux 7.2 版本以前的系统默认都是 Ext2 文件系统。于 1993 年发布，支持最大 16TB 的分区和最大 2TB 的文件（1TB=1024GB=1024x1024KB）
Ext3	是 Ext2 文件系统的升级版本，最大的区别就是带日志功能，以便在系统突然停止时提高文件系统的可靠性。支持最大 16TB 的分区和最大 2TB 的文件
Ext4	是 Ext3 文件系统的升级版。Ext4 在性能、伸缩性和可靠性方面进行了大量改进。Ext4 的变化可以说是翻天覆地的，比如向下兼容 Ext3、最大 1EB 文件系统和 16TB 文件、无限数量子目录、Extents 连续数据块 概念、多块分配、延迟分配、持久预分配、快速 FSCK、日志校验、无日志模式、在线碎片整理、inode 增强、默认启用 barrier 等。它是 CentOS 6.3 的默认文件系统
xfs	被业界称为最先进、最具有可升级性的文件系统技术，由 SGI 公司设计，目前最新的 CentOS 7 版本默认使用的就是此文件系统。
swap	swap 是 Linux 中用于交换分区的文件系统（类似于 Windows 中的虚拟内存），当内存不够用时，使用交换分区暂时替代内存。一般大小为内存的 2 倍，但是不要超过 2GB。它是 Linux 的必需分区
NFS	NFS 是网络文件系统（Network File System）的缩写，是用来实现不同主机之间文件共享的一种网络服务，本地主机可以通过挂载的方式使用远程共享的资源
iso9660	光盘的标准文件系统。Linux 要想使用光盘，必须支持 iso9660 文件系统
fat	就是 Windows 下的 fatl6 文件系统，在 Linux 中识别为 fat
vfat	就是 Windows 下的 fat32 文件系统，在 Linux 中识别为 vfat。支持最大 32GB 的分区和最大 4GB 的文件
NTFS	就是 Windows 下的 NTFS 文件系统，不过 Linux 默认是不能识别 NTFS 文件系统的，如果需要识别，则需要重新编译内核才能支持。它比 fat32 文件系统更加安全，速度更快，支持最大 2TB 的分区和最大 64GB 的文件
ufs	Sun 公司的操作系统 Solaris 和 SunOS 所采用的文件系统
proc	Linux 中基于内存的虚拟文件系统，用来管理内存存储目录 /proc
sysfs	和 proc 一样，也是基于内存的虚拟文件系统，用来管理内存存储目录 /sysfs
tmpfs	也是一种基于内存的虚拟文件系统，不过也可以使用 swap 交换分区

嵌入式Linux文件系统框架（1）

- 传统文件系统：
  - 采用设备API，经过设备驱动，直接去访问磁盘。
  - 或者，采用文件API，经过文件管理器，以及设备驱动，去访问磁盘。

设备API函数：
① open()
② close()
③ read()
④ write()
⑤ lseek()：移动文件读/写指针
⑥ ioctl()

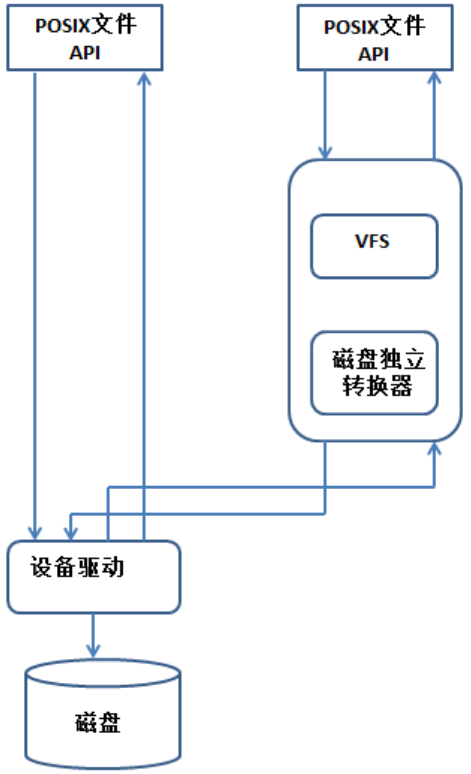


嵌入式Linux文件系统框架（2）

- Linux文件系统：
  - 采用POSIX文件API，经过设备驱动，直接去访问磁盘。
  - 或者，采用POSIX文件API，经过VFS和磁盘独立转换器，以及设备驱动，去访问磁盘。

POSIX：Portable Operating System Interface of UNIX，可移植操作系统接口

VFS：Virtual File Systems，虚拟文件系统



JFFS2嵌入式文件系统

- JFFS：Journalling Flash File System，闪存设备日志型文件系统，最初是由瑞典的 Axis Communication AB 开发，其目的是作为嵌入式系统免受宕机和断电危害的文件系统。

- JFFS2：Journalling Flash File System Version2，闪存日志型文件系统第2版，其功能就是管理在MTD设备上实现的日志型文件系统。除了提供具有断电可靠性的日志结构文件系统，JFFS2还会在它管理的MTD设备上实现“损耗平衡”和“数据压缩”等特性。

MTD：Memory Technology Device，内存技术设备。是用于访问memory设备（ROM、flash）的Linux的子系统。MTD的主要目的是为了使新的memory设备的驱动更加简单，为此它在硬件和上层之间提供了一个抽象的接口。MTD的所有源代码在/drivers/mtd子目录下。CFI接口的MTD设备分为四层（从设备节点直到底层硬件驱动），这四层从上到下依次是：设备节点、MTD设备层、MTD原始设备层和硬件驱动层。

JFFS3嵌入式文件系统

- JFFS3简介
  - JFFS3：Journalling Flash File System Version3，闪存日志型文件系统第3版
  - JFFS3支持大容量Flash：大于1TB
  - JFFS3是将索引信息存放在Flash上，而JFFS2则将索引信息保存在内存中
  - JFFS3的基本结构借鉴了ReiserFS4的设计思想，整个文件系统就是一个B+树

ReiserFS4：是一种新型的文件系统，它通过一种与众不同的方式——完全平衡树结构来容纳数据，包括文件数据，文件名以及日志支持。ReiserFS4还以支持海量磁盘和磁盘阵列，并能在上面继续保持很快的搜索速度和很高的效率。

B+树：是一种树数据结构，通常用于数据库和操作系统的文件系统中。B+树的特点是能够保持数据稳定有序，其插入与修改拥有较稳定的对数时间复杂度。B+树元素自底向上插入，这与二叉树恰好相反。

根文件系统（1）

- 什么是根文件系统：
  - 系统挂载的第一个文件系统就是根文件系统
  - 根文件系统顶层目录：
    ① bin：用户命令所在目录
    ② sbin：系统管理员命令目录
    ③ usr：共享的文件
    ④ proc：虚拟文件系统，用来显示内核及进程信息
    ⑤ dev：硬件设备文件及其它特殊文件
    ⑥ etc：系统配置文件，包括启动文件等
    ⑦ lib：链接库文件目录
    ⑧ boot：引导加载程序使用的静态文件
    ⑨ home：多用户主目录
    ⑩ mnt：装配点,用于装配临时文件系统或其他的文件系统
    ⑪ opt：附加的软件套件目录
    ⑫ root：用户主目录
    ⑬ tmp：临时文件目录
    ⑭ var：监控程序和工具程序所存放的可变数据

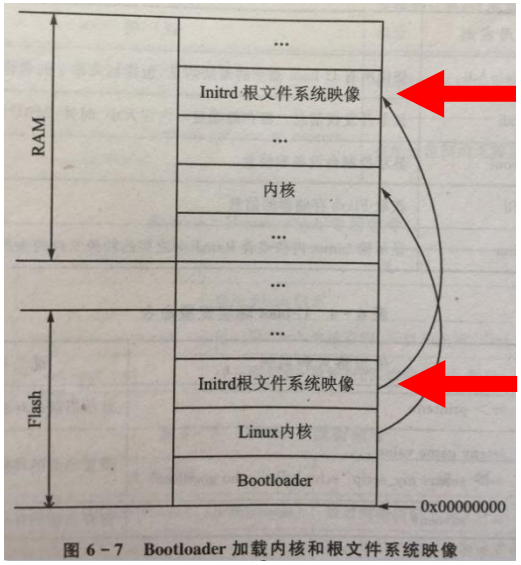
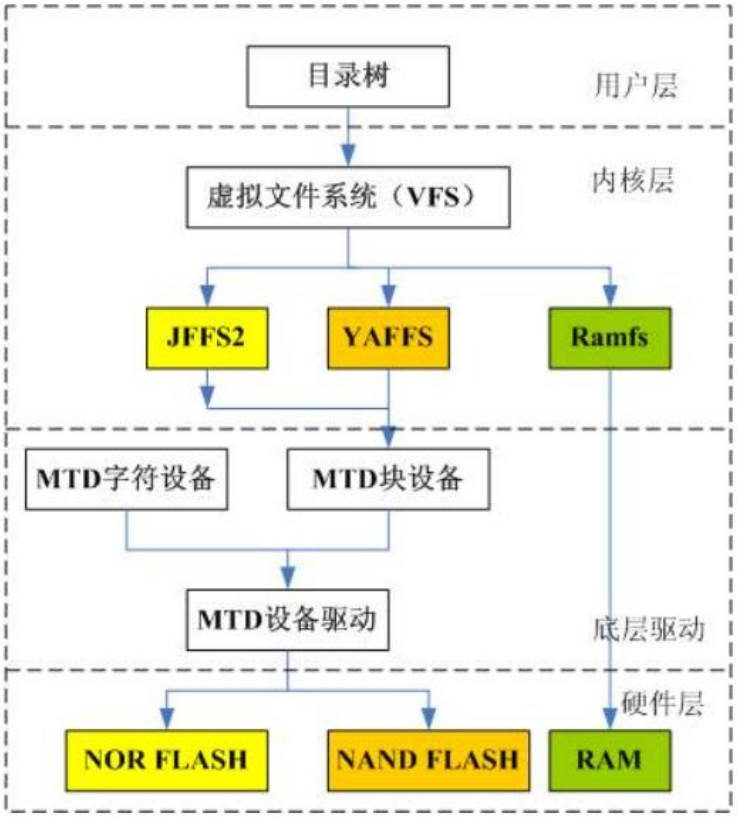


图 6－7 Bootloader 加载内核和根文件系统映像

根文件系统（2）

- **Linux下常用文件系统结构：**
  - **采用JFFS2作为根文件系统**



Linux下常用文件系统结构

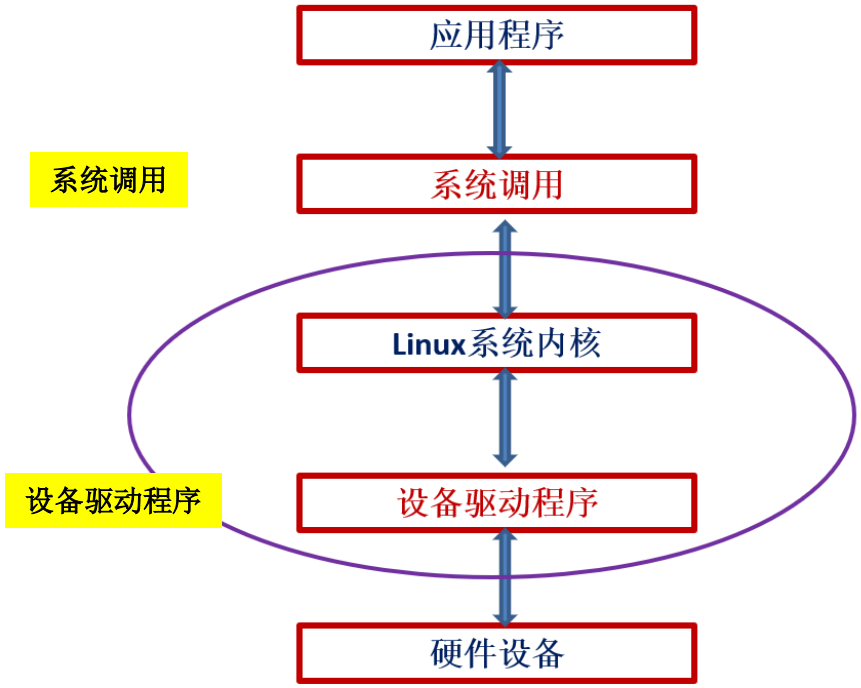
- **JFFS2：Journalling Flash File System Version2，闪存日志型文件系统第2版，其功能就是管理在MTD设备上实现的日志型文件系统。除了提供具有断电可靠性的日志结构文件系统，JFFS2还会在它管理的MTD设备上实现“损耗平衡”和“数据压缩”等特性。**
- **YAFFS：Yet Another Flash File System，是专为嵌入式系统使用NAND型闪存而设计的一种日志型文件系统。**
- **Ramfs：基于RAM的文件系统。**
- **Cramfs：Compressed ROM File System，只读压缩的文件系统。**

第9章　设备驱动程序设计基础

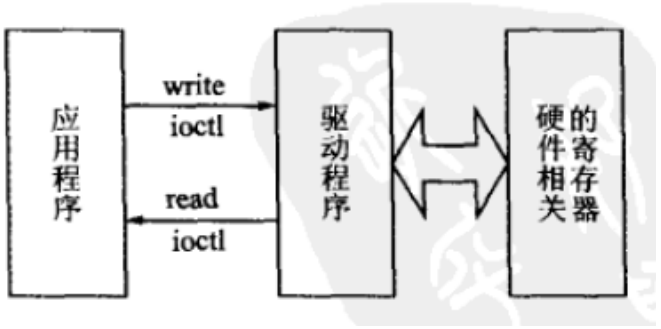
- 9.1　Linux设备驱动程序简介
- 9.2　设备驱动程序结构
- 9.3　Linux内核设备模型
- 9.4　内存映射和管理

设备驱动程序与系统调用（1）

- 设备驱动程序与系统调用：



设备驱动程序与系统调用



设备驱动程序

设备驱动程序与系统调用（2）

- 系统调用：是操作系统内核（Linux系统内核）和应用程序之间的接口。

- 设备驱动程序：是操作系统内核（ Linux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，在应用程序看来，硬件设备只是一个设备文件，应用程序可以向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，完成以下功能：
  ① 对设备的初始化和释放；
  ② 把数据从内核传送到硬件，和从硬件读取数据到内核；
  ③ 读取应用程序传送给设备文件的数据，和回送应用程序请求的数据；这需要在用户空间、内核空间、总线以及外设之间传输数据；
  ④ 检测和处理设备出现的错误。

设备驱动程序与应用程序的区别

- 设备驱动程序与应用程序的区别：

① 应用程序一般有一个main 函数，从头到尾执行一个任务。

② 设备驱动程序却不同，它没有main函数，通过使用宏module_init()，将初始化函数加入内核全局初始化函数列表中，在内核初始化时执行驱动的初始化函数，从而完成驱动的初始化和注册，之后驱动便停止等待被应用软件调用；驱动程序中有一个宏module_exit()注册退出处理函数，它在驱动退出时被调用。

③ 应用程序可以和GLIBC 库连接，因此可以包含标准的头文件，比如<stdio.h>、<stdlib.h>。

④ 在设备驱动程序中是不能使用标准C 库的，因此不能调用所有的C 库函数，比如输出打印函数只能使用内核的printk 函数，包含的头文件只能是内核的头文件，比如<linux/module.h>。

Linux 设备驱动程序开发调试的两种方法

- Linux 的设备驱动程序开发调试有两种方法：
  ① 一种是直接编译到内核，再运行新的内核来测试；

  ② 二是编译为模块的形式，单独加载运行调试。

- 第一种方法（直接编译到内核）效率较低，但在某些场合是唯一的方法。

- 第二种方式（编译为模块——模块方式）调试效率很高，它使用 insmod 命令将编译的模块直接插入内核；如果出现故障，可以使用 rmmod命令 从内核中卸载模块；不需要重新启动内核，这使驱动调试效率大大提高。lsmod命令为查看模块。

设备的分类

- 设备的分类：
  - 字符设备：无须缓冲直接读写的设备，如串口等设备。
  - 块设备：通过缓冲区进行（缓冲区通常为系统内存），只能以块为单位进行读写，块大小可以是512B或1024B，如硬盘等设备。
  - 网络设备：可以通过BSD套接口访问。
    - BSD（Berkeley Software Distribution，伯克利软件套件）是Unix的衍生系统，在1977至1995年间由加州大学伯克利分校开发和发布的。

设备文件

- 设备文件：
  - Linux抽象了对硬件的处理，所有的硬件设备都可以作为普通文件一样对待，可以使用标准的系统调用接口来完成对设备的打开（open）、关闭（close）、读写（read、write）和I/O控制操作（ioctl），驱动程序的主要任务是实现这些系统调用函数。
  - Linux系统中所有的硬件设备都使用一个特殊的设备文件来表示，如：
    - 系统中的第一个硬盘：/dev/had
    - 串口0：/dev/ttyS0
    - CAN总线：/dev/can/0
  - 对用户来说，设备文件和普通文件并无区别
  - 查看设备文件命令：ls -l /dev

主设备号和次设备号

- 主设备号和次设备号：
  - 主设备号：标识该设备的种类，也标识了该设备所使用的驱动程序，主设备号在/proc/devices文件中查看
    - 查看主设备号命令：cat /proc/devices
  - 次设备号：标识使用同一设备驱动程序的不同硬件设备
  - 创建设备文件的命令：mknod /dev/lp0 c 6 0
    - /dev/lp0：设备名
    - c：表示字符设备（b：表示块设备）
    - 6：主设备号
    - 0：次设备号

实验箱的主设备号： cat /proc/devices

Character devices: 字符设备

```
  1 mem
  4 /dev/vc/0
  4 tty
  4 ttyS
  5 /dev/tty
  5 /dev/console
  5 /dev/ptmx
  7 vcs
 10 misc
 13 input
 29 fb
 30 UART485
 81 video4linux
 89 i2c
 90 mtd
116 alsa
128 ptm
136 pts
180 usb
188 ttyUSB
189 usb_device
199 galcore
207 ttymxc
216 rfcomm
226 drm
247 mxc_vpu
248 ttyLP
249 mxc_hdmi
250 iio
251 mxc_ipu
252 ptp
253 pps
254 rtc
```

Block devices: 块设备

```
  1 ramdisk
259 blkext
  7 loop
  8 sd
 31 mtdblock
 65 sd
 66 sd
 67 sd
 68 sd
 69 sd
 70 sd
 71 sd
128 sd
129 sd
130 sd
131 sd
132 sd
133 sd
134 sd
135 sd
179 mmc
```

虚拟机的主设备号： cat /proc/devices

Character devices: 字符设备

1 mem
4 /dev/vc/0
4 tty
4 ttyS
5 /dev/tty
5 /dev/console
5 /dev/ptmx
5 ttyprintk
6 lp
7 vcs
10 misc
13 input
14 sound/midi
14 sound/dmmidi
21 sg
29 fb
89 i2c
99 ppdev
108 ppp
116 alsa
128 ptm
136 pts
180 usb
189 usb_device
216 rfcomm
226 drm
251 hidraw
252 bsg
253 watchdog
254 rtc

Block devices: 块设备

1 ramdisk
2 fd
259 blkext
7 loop
8 sd
9 md
11 sr
65 sd
66 sd
67 sd
68 sd
69 sd
70 sd
71 sd
128 sd
129 sd
130 sd
131 sd
132 sd
133 sd
134 sd
135 sd
252 device-mapper
253 virtblk
254 mdp

Linux设备驱动代码的分布

- **Linux设备驱动代码的分布：**
  - 所有驱动位于： **/home/linux/workdir/fs3399/system/kernel/drivers/**目录下
    - **char**：字符设备驱动
    - **block**：块设备驱动
    - **cdrom**：CDROM驱动
    - **pci**：PCI驱动
    - **scsi**：SCSI驱动
    - **net**：网络驱动

设备驱动程序结构

- Linux设备驱动程序与外界的接口分为以下三部分：
  ① 驱动程序与Linux操作系统内核的接口
  ② 驱动程序与系统引导的接口
  ③ 驱动程序与设备的接口

- Linux设备驱动程序的代码结构包括：
  ① 驱动程序的注册与注销
  ② 设备的打开与释放
  ③ 设备的读写操作
  ④ 设备的控制操作
  ⑤ 设备的中断和轮询处理

驱动程序的注册与注销

- 驱动程序的注册与注销：
  - 注册：赋予设备一个主设备号
    - 字符设备（chr）：register_chrdev()函数
      - 例如：ret = register_chrdev(0, DEVICE_NAME, &s3c2410_dcm_fops);
    - 块设备（blk）：register_blkdev()函数
  - 注销：释放占用的主设备号
    - 字符设备：unregister_chrdev()函数
      - 例如：unregister_chrdev(dcmMajor, DEVICE_NAME);
    - 块设备：unregister_blkdev()函数

设备的打开与释放

- 设备的打开与释放：
  - file_operations结构体：设备文件操作结构体

```
static const struct file_operations uart485_fops = {
        .owner = THIS_MODULE,
        .write = Uart485PowerWrite,
        .read = Uart485PowerRead,
        .open = Uart485PowerOpen,
        .unlocked_ioctl = Uart485PowerIoctl,
};
```

  - 设备的打开：通过调用file_operations结构体中的open()函数完成
    - 例如： com485 = open("/dev/UART485", O_RDWR);　　//打开RS-485

  - 设备的释放（关闭）：通过调用file_operations结构体中的release()函数完成（有时也称为close()函数）
    - 例如： close(com485);　　//关闭RS-485

设备的读写操作

- 设备的读写操作：
  - 设备的读操作：通过调用file_operations结构体中的read()函数完成
    - 字符设备：read()函数
      - 例如：re = read(COMDevice,buf,sizeof(buf));　　//从RS-485读（接收）数据
    - 块设备：block_read()
  - 设备的写操作：通过调用file_operations结构体中的write()函数完成
    - 字符设备：write()函数
      - 例如：re = write(COMDevice,buf,strlen(buf));　　//向RS-485写（发送）数据
    - 块设备：block_write()函数

设备的控制操作

- 设备的控制操作：
  - 设备的控制操作：通过调用file_operations结构体中的ioctl()函数完成
  - 例如，使RS-485处于发送模式或接收模式：
    - ioctl(com485,UART485_TX);　　//设置RS-485为发送方式（TX）
    - ioctl(com485,UART485_RX);　　//设置RS-485为接收方式（RX）

设备的轮询和中断处理

- 设备的轮询和中断处理：
  - 轮询方式（查询方式）：对于不支持中断的硬件设备，读写时需要轮流查询设备状态，以便决定是否继续进行数据传输
    - 轮询设备驱动可以通过使用系统定时器，使内核周期性的调用设备驱动中的某个例程来检查设备状态
  - 中断方式：内核负责把硬件产生的中断传递给相应的设备驱动
    - 在/proc/interrupts文件中可以看到设备驱动所对应的中断号及类型
    - 查询中断号的命令：cat /proc/interrupts

实验箱的中断号

```
root@imx6dlsabresd:~# cat   /proc/interrupts
           CPU0       CPU1
 29:     166850      14973         GIC  29  twd
 34:          0          0         GIC  34  sdma
 35:          0          0         GIC  35  VPU_JPG_IRQ
 37:          0          0         GIC  37  2400000.ipu
 38:          7          0         GIC  38  2400000.ipu
 41:        707          0         GIC  41
 44:          0          0         GIC  44  VPU_CODEC_IRQ
 50:          0          0         GIC  50  vdoa
 51:          0          0         GIC  51  rtc alarm
 52:          0          0         GIC  52  snvs-secvio
 56:          0          0         GIC  56  mmc2
 57:       2468          0         GIC  57  mmc3
 58:        521          0         GIC  58  2020000.serial
 68:        132          0         GIC  68  21a0000.i2c
 69:       1491          0         GIC  69  21a4000.i2c
 70:      76809          0         GIC  70  21a8000.i2c
 72:       1647          0         GIC  72  2184200.usb
 75:          0          0         GIC  75  2184000.usb
```

虚拟机的中断号

```
root@uptech-virtual-machine:/imx6/whzeng/nfc-3# cat   /proc/interrupts
           CPU0
  0:         50   IO-APIC-edge      timer
  1:       2272   IO-APIC-edge      i8042
  6:          2   IO-APIC-edge      floppy
  7:          0   IO-APIC-edge      parport0
  8:          1   IO-APIC-edge      rtc0
  9:          0   IO-APIC-fasteoi   acpi
 12:      96144   IO-APIC-edge      i8042
 14:          0   IO-APIC-edge      ata_piix
 15:          0   IO-APIC-edge      ata_piix
 16:       5445   IO-APIC  16-fasteoi   vmwgfx, snd_ens1371, eth0
 17:      74235   IO-APIC  17-fasteoi   ehci_hcd:usb1, ioc0
 18:        165   IO-APIC  18-fasteoi   uhci_hcd:usb2
 24:          0   PCI-MSI-edge      PCIe PME, pciehp
 25:          0   PCI-MSI-edge      PCIe PME, pciehp
 26:          0   PCI-MSI-edge      PCIe PME, pciehp
 27:          0   PCI-MSI-edge      PCIe PME, pciehp
 28:          0   PCI-MSI-edge      PCIe PME, pciehp
 29:          0   PCI-MSI-edge      PCIe PME, pciehp
 30:          0   PCI-MSI-edge      PCIe PME, pciehp
 31:          0   PCI-MSI-edge      PCIe PME, pciehp
 32:          0   PCI-MSI-edge      PCIe PME, pciehp
 33:          0   PCI-MSI-edge      PCIe PME, pciehp
 34:          0   PCI-MSI-edge      PCIe PME, pciehp
 35:          0   PCI-MSI-edge      PCIe PME, pciehp
 36:          0   PCI-MSI-edge      PCIe PME, pciehp
 37:          0   PCI-MSI-edge      PCIe PME, pciehp
 38:          0   PCI-MSI-edge      PCIe PME, pciehp
 39:          0   PCI-MSI-edge      PCIe PME, pciehp
 40:          0   PCI-MSI-edge      PCIe PME, pciehp
 41:          0   PCI-MSI-edge      PCIe PME, pciehp
 42:          0   PCI-MSI-edge      PCIe PME, pciehp
 43:          0   PCI-MSI-edge      PCIe PME, pciehp
 44:          0   PCI-MSI-edge      PCIe PME, pciehp
 45:          0   PCI-MSI-edge      PCIe PME, pciehp
 46:          0   PCI-MSI-edge      PCIe PME, pciehp
 47:          0   PCI-MSI-edge      PCIe PME, pciehp
 48:          0   PCI-MSI-edge      PCIe PME, pciehp
 49:          0   PCI-MSI-edge      PCIe PME, pciehp
 50:          0   PCI-MSI-edge      PCIe PME, pciehp
 51:          0   PCI-MSI-edge      PCIe PME, pciehp
 52:          0   PCI-MSI-edge      PCIe PME, pciehp
 53:          0   PCI-MSI-edge      PCIe PME, pciehp
```

内存映射和管理

- 物理地址映射到虚拟地址：
  - 在内核中访问I/O内存（I/O与内存统一编制，访问I/O就像访问内存一样）之前，我们只有I/O内存的物理地址，这样是无法通过软件直接访问的，需要首先用ioremap()函数将设备所处的物理地址映射到内核虚拟地址空间（3GB~4GB），然后，才能根据映射所得到的内核虚拟地址范围，通过访问指令访问这些I/O内存资源。
    - void * ioremap(unsigned long phys_addr, unsigned long size, unsigned long flags)
      - phys_addr：要映射的起始的I/O地址
      - size：要映射的空间的大小
      - flags：要映射的I/O空间的和权限有关的标志

内存映射和管理（1）

- 内核空间映射到用户空间：
  - 使用mmap系统调用，可以将内核空间的地址映射到用户空间
    - void* mmap(void* start, size_t length, int prot, int flags, int fd, off_t offset);
    - int (* mmap)(struct file*filp, struct vm_area_struct *vma);
  - 查看设备内存是如何映射的： cat /proc/iomem

```
root@imx6dlsabresd:/sys/bus/platform# cat /proc/iomem
00110000-00111fff : /soc/dma-apbh@00110000
00120000-00128fff : 120000.hdmi_core
00130000-00133fff : galcore register region
00134000-00137fff : galcore register region
00905000-0091ffff : /soc/sram@00905000
02020000-02023fff : /soc/aips-bus@02000000/spba-bus@02000000/serial@02020000
02034000-02037fff : /soc/aips-bus@02000000/spba-bus@02000000/asrc@02034000
02080000-02083fff : /soc/aips-bus@02000000/pwm@02080000
02084000-02087fff : /soc/aips-bus@02000000/pwm@02084000
02088000-0208bfff : /soc/aips-bus@02000000/pwm@02088000
0208c000-0208ffff : /soc/aips-bus@02000000/pwm@0208c000
02090000-02093fff : /soc/aips-bus@02000000/can@02090000
0209c000-0209ffff : /soc/aips-bus@02000000/gpio@0209c000
020a0000-020a3fff : /soc/aips-bus@02000000/gpio@020a0000
020a4000-020a7fff : /soc/aips-bus@02000000/gpio@020a4000
020a8000-020abfff : /soc/aips-bus@02000000/gpio@020a8000
020ac000-020affff : /soc/aips-bus@02000000/gpio@020ac000
```

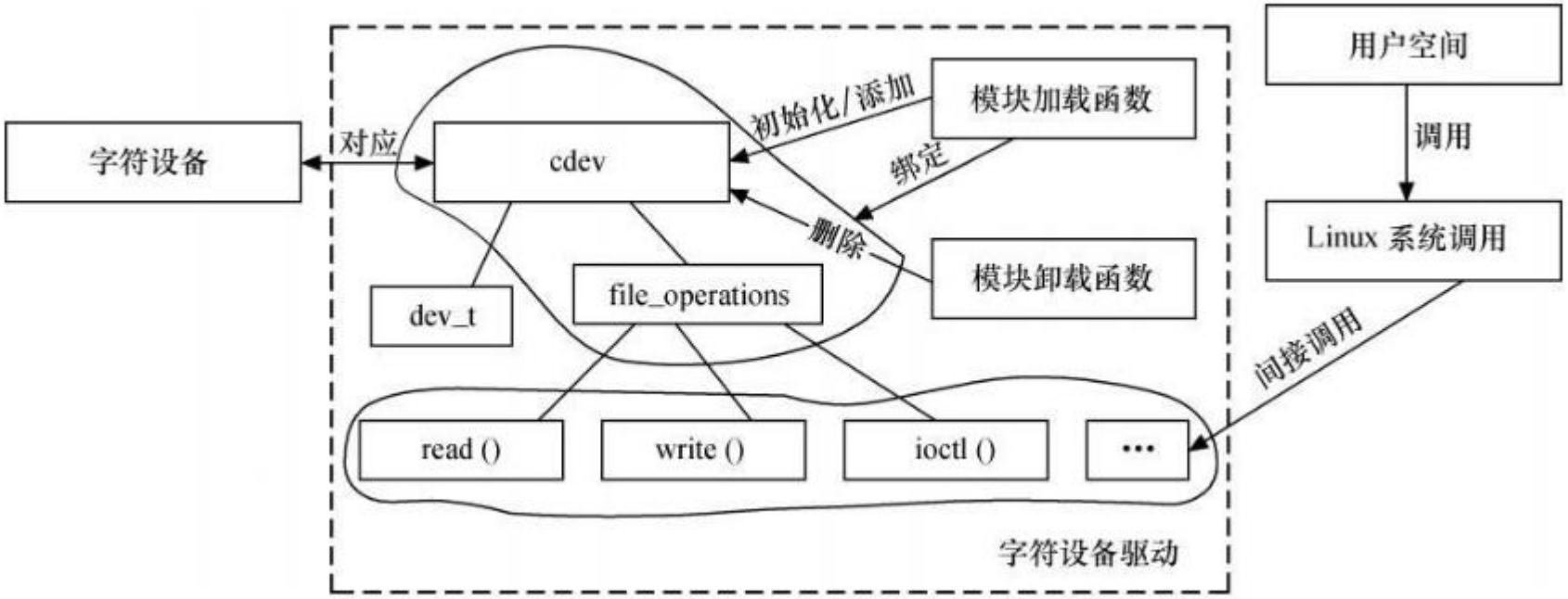
第10章　字符设备和驱动程序设计

- 11.1　字符设备驱动框架
- 11.2　字符设备驱动开发
- 11.3　GPIO驱动概述
- 11.4　串口总线概述
- 11.5　字符设备驱动程序示例

Linux字符设备

- 字符设备是Linux三大设备之一（另外两种是块设备，网络设备）。
- 字符设备就是采用字节流形式通讯的I/O设备，绝大部分设备都是字符设备。
- 常见的字符设备包括鼠标、键盘、显示器、串口等等。

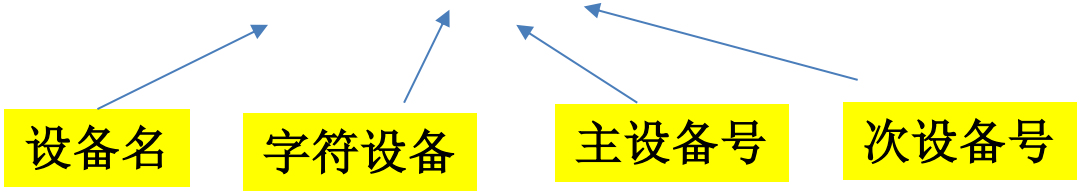
字符设备驱动框架



字符设备驱动

设备号

- 设备号：
  - 查看主设备号和次设备号：
    - ls -l /dev
  - 查看已经加载了驱动程序的主设备号：
    - cat /proc/devices
  - 创建指定类型的设备文件：
    - mknod /dev/lp0 c 6 0



查看实验箱的主设备号和次设备号（ls -l /dev）

```
root@imx6dlsabresd:~# ls  -l  /dev
total 0
crw------- 1 root root     30,    0 Jan  1  1970 UART485
crw------- 1 root root     10,  235 Jan  1  1970 autofs
drwxr-xr-x 2 root root          640 Jan  1  1970 block
drwxr-xr-x 3 root root           60 Jan  1  1970 bus
drwxr-xr-x 2 root root         2700 Sep 29 08:03 char
crw------- 1 root root      5,    1 Sep 29 08:03 console
crw------- 1 root root     10,   61 Jan  1  1970 cpu_dma_latency
drwxr-xr-x 5 root root          100 Jan  1  1970 disk
drwxr-xr-x 2 root root           60 Jan  1  1970 dri
crw-rw---- 1 root video    29,    0 Jan  1  1970 fb0
crw-rw---- 1 root video    29,    1 Jan  1  1970 fb1
lrwxrwxrwx 1 root root           13 Jan  1  1970 fd -> /proc/self/fd
crw-rw-rw- 1 root root      1,    7 Jan  1  1970 full
crw-rw-rw- 1 root root     10,  229 Jan  1  1970 fuse
crw-rw---- 1 root video   199,    0 Jan  1  1970 galcore
crw------- 1 root root     10,  183 Jan  1  1970 hwrng
crw------- 1 root root     89,    0 Jan  1  1970 i2c-0
crw------- 1 root root     89,    1 Jan  1  1970 i2c-1
crw------- 1 root root     89,    2 Jan  1  1970 i2c-2
prw------- 1 root root            0 Jan  1  1970 initctl
```

实验箱字符设备的主设备号（cat /proc/devices）

```
root@imx6dlsabresd:~# cat  /proc/devices
Character devices:
  1 mem
  4 /dev/vc/0
  4 tty
  4 ttyS
  5 /dev/tty
  5 /dev/console
  5 /dev/ptmx
  7 vcs
 10 misc
 13 input
 29 fb
 30 UART485
 81 video4linux
 89 i2c
 90 mtd
116 alsa
128 ptm
136 pts
180 usb
188 ttyUSB
```

关键数据结构

- 关键数据结构：
  - 1、file_operations（文件操作数据结构/结构体）
    - 改变文件中的读写位置
      - loff_t (*llseek) (struct file *, loff_t, int);
    - 从设备中读取数据
      - ssize_t (*read) (struct file *, char *, size_t, loff_t *);
    - 向设备写数据
      - ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
    - 对设备进行控制
      - int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    - 将设备内存映射到进程的地址空间
      - int (*mmap) (struct file *, struct vm_area_struct *);
    - 打开设备和初始化
      - int (*open) (struct inode *, struct file *);
    - 释放设备占用的内存并关闭设备
      - int (*release) (struct inode *, struct file *);

```
struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
        ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
        ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
        int (*readdir) (struct file *, void *, filldir_t);
        unsigned int (*poll) (struct file *, struct poll_table_struct *);
        int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
        long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
        long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
        int (*mmap) (struct file *, struct vm_area_struct *);
        int (*open) (struct inode *, struct file *);
        int (*flush) (struct file *, fl_owner_t id);
        int (*release) (struct inode *, struct file *);
        int (*fsync) (struct file *, struct dentry *, int datasync);
        int (*aio_fsync) (struct kiocb *, int datasync);
        int (*fasync) (int, struct file *, int);
        int (*lock) (struct file *, int, struct file_lock *);
        ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
        unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned long, unsigned long);
        int (*check_flags)(int);
        int (*flock) (struct file *, int, struct file_lock *);
        ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_t, unsigned int);
        ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t, unsigned int);
        int (*setlease)(struct file *, long, struct file_lock **);
};
```

file_operations结构体

GPIO驱动概述

- GPIO：General Purpose Input/Output，通用输入输出，可以对GPIO进行编程，将GPIO的每一个引脚设为输入或输出，因此GPIO也称为通用可编程接口。

- GPIO接口至少要有两个寄存器：
  - 控制寄存器
  - 数据寄存器

- GPIO的寄存器可以使用内存映射（将I/O当作内存看待，I/O与存储器统一编址，访问I/O与访问存储器一样），或者端口映射（I/O单独编址）。

- 如果使用内存映射，要向GPIO的寄存器A写入数据0xff，设寄存器A的地址为0x36000000，则使用以下代码：
  - #define A (*(volatile unsigned long *)0x36000000)
  - A = 0xff

串行总线概述（1）

- SPI总线：
  - SPI是串行外设接口（Serial Peripheral Interface）的缩写。是Motorola 公司推出的一 种同步串行接口技术，是一种高速的，全双工，同步的通信总线。主要应用于EEPROM、Flash、实时时钟、A/D转换以及数字信号处理器和数字信号解码器。SPI的传输速率可达3Mb/s。
  - SPI有两种工作模式：
    - 主模式
    - 从模式
  - SPI有4条接口线：
    - SDI（MISO）： Serial Data In，串行数据输入；
    - SDO（MOSI）： Serial Data Out，串行数据输出；
    - SCLK（SCK）： Serial Clock，时钟信号，由主设备产生；
    - CS（SS）： Chip Select，从设备使能信号，由主设备控制。

串行总线概述（2）

- **I²C总线：**
  - **I²C（Inter Integrated-Circuit，IIC，I2C，内部集成电路）总线，是由PHILIPS公司在上世纪80年代发明的一种电路板级串行总线标准，最初应用于音频和视频领域的设备开发。**
  - **I²C总线有两根接口线：**
    - **数据线：SDA**
    - **时钟线：SCK，或SCL**
  - **I²C总线在传输过程中有三种不同类型的信号：**
    - **开始信号**
    - **结束信号**
    - **应答信号**
  - **I²C总线在标准模式下传输速率可达100kb/s，在快速模式下传输速率可达400kb/s，在高速模式下传输速率可达3.4Mb/s。**

串行总线概述（3）

- **SMBus：**
  - SMBus（System Management Bus，系统管理总线）是1995年由Intel提出的，应用于移动PC和桌面PC系统中的低速率通讯。希望通过一条廉价并且功能强大的总线（由两条线组成），来控制主板上的设备并收集相应的信息。
  - SMBus有两根接口线：
    - 数据线：SMBDAT
    - 时钟线：SMBCLK
  - SMBus的传输率只有100kb/s，SMBus总线的特点是结构简单、造价低。

第11章　Android 操作系统

- 10.1　Android 操作系统介绍
- 10.2　Android 软件架构介绍
- 10.3　Android 内核
- 10.4　Android 子系统介绍
- 10.5　Android 应用程序开发过程
- 10.6　Android 源码目录结构

Android 操作系统介绍

- Android 是 Google 公司于 2007 年 11 月发布的一款非常优秀的智能移动平台操作系统。到 2011 年第一季度 Android 在全球的市场份额首次超过 Nokia 的 Symbian 系统，跃居全球第一。

- Android 系统最初由 Andy Rubin 等人于 2003 年 10 月创建。Google 于 2005 年 8 月 17 日收购 Android 并组建 OHA（Open Handset Alliance，开放手机联盟）开发改良 Android，之后逐渐扩展到平板电脑及其他移动平台领域上。

- Android 系统是一个基于 Apache License（Apache 许可证）、GPL（GNU General Public License，GNU 通用公共许可证）软件许可的开源手机操作系统，底层由 Linux 操作系统作为内核，我们可以直接从 Android 的官方网站上下载最新的 Android 源码和相关开发工具包。
  - Android 官方首页： http://www.android.com/
  - Android 官方开发者首页： http://developer.android.com/index.html
  - Android 官方开源项目 AOSP 首页：http://source.android.com/

- 这些网站可能打不开，需要设置代理或者翻墙。

Android 软件架构介绍（1）

- Android 的软件架构采用了分层结构，如图所示，由上至下分别为：Application 应用层、Application Framework 应用框架层、Android Runtime & Libraries 运行时库和本地库层、Linux Kernel 内核层。

应用层

应用框架层

运行时库和本地库层

内核层

Android 软件架构介绍（2）

① Application 应用层：用户安装应用程序及系统自带应用程序层，主要用来与用户进行交互，如 Home 指 Android 手机的桌面，Phone 指电话应用，用来拨打电话等。

② Application Framework 应用框架层：系统框架层，封装了大量应用程序所使用的类，从而达到组件重用的目的，它主要向上层应用层提供 API，如：ActivityManager 主要用于管理所有的 Activity、画面导航、 回退等生命周期相关的操作，PackageManager 主要用来管理程序安装包的安装、更新、删除等操作。

③ Android Runtime & Libraries 运行时库和本地库层：Runtime 是 Android 的运行环境，在该层有 DalvikVirtualMachine（Android 的虚拟机简称 DVM）的实现，在 DVM 中运行着 Java 的核心语言库代码和 Java 程序。同时，在 DVM 运行期间要调用系统库代码，如：负责显示的 SurfaceManager 本地代码，负责 多媒体处理相关的 MediaFrameworks 代码及 C 库 libc 等。

④ Linux Kernel 内核层：Android 系统是基于 Linux 系统的，所以 Android 底层系统相关的框架和标准的 Linux 内核没有什么很大的区别，只不过添加了几个 Android 系统运行必备的驱动，如：BinderIPC 进程 间通信驱动、PowerManager 电源管理驱动等。

- 总结：Android 的软件架构是我们学习 Android 开发必须要掌握的知识点，它对我们将来编写 Android 应用程序、理解 Android 框架代码、编写本地代码、修改底层驱动都有重要的指导意义，可谓是学习 Android 的灵魂。

Android 内核

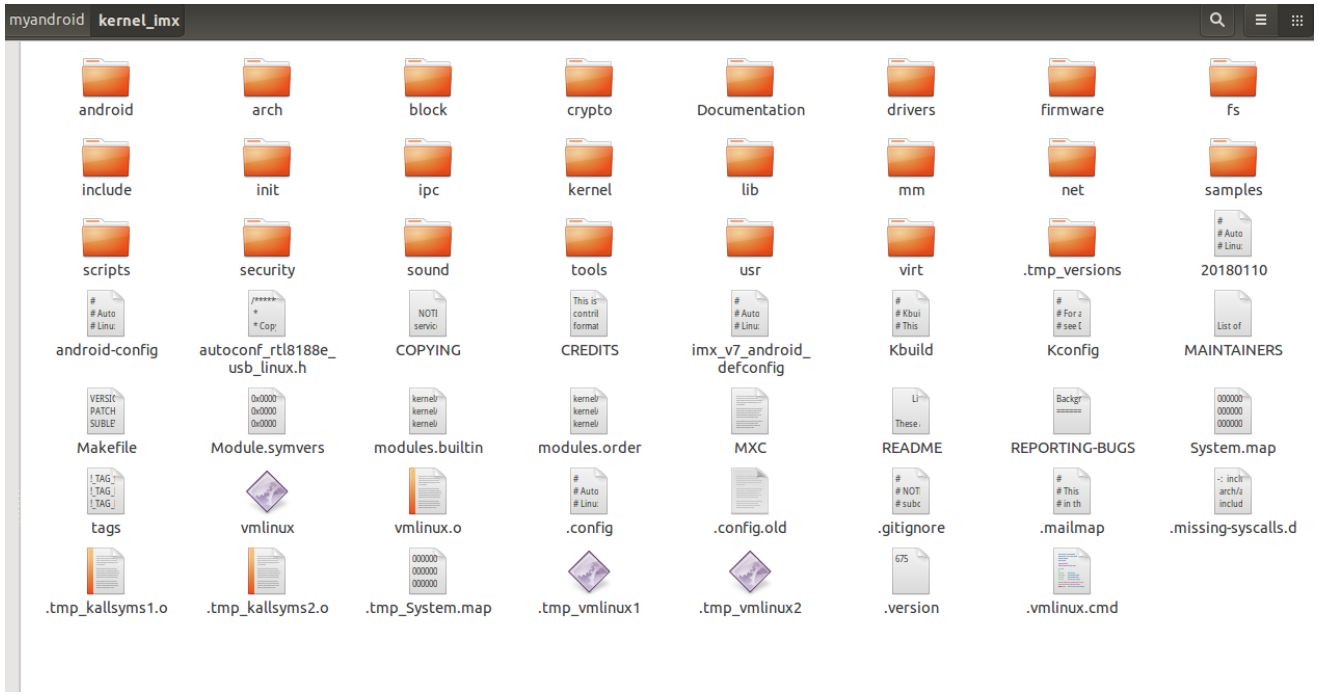
- 内核（Kernel）是Android系统最核心的部分，其主要作用在于与计算机硬件进行交互，实现对硬件的编程控制和接口操作。主要功能包括：内存管理、进程间通信、进程调度、虚拟文件系统、网络等功能。和标准的 Linux 内核一样，Android 内核主要实现内存管理、进程间通信及进程调度等功能。 Android 对 Linux 内核的更改较少，但增加了一些没有加入标准内核的内容，例如：yaffs 文件系统。

- Android 内核结构和标准 Linux 4.1.15 内核基本上相同，为了适应嵌入式硬件环境和移动应用程序的开 发，Android 在其基础上，增加了私有内容。

- Android 在标准 Linux 内核中增加的内容是一些驱动程序，这些驱动主要分为两种类型：
  ① Android 专用驱动
  ② Android 使用的设备驱动

Android 内核目录结构

- **Android 内核目录结构：**
  - **Android 内核结构和标准 Linux 4.1.15 内核基本上相同，IMX6 综合嵌入式教学科 研平台（实验箱）运行的 Android 内核版本为 /Android/myandroid/kernel_imx，其源码目录结构如图：**



Android 子系统介绍

- Android 是一个庞大的手机的系统，它不仅仅实现了手机的基本的打电话，发信息的功能，还实现了更复杂的多媒体处理、2D 和 3D 游戏处理、信息感知处理等。

- Android 的子系统主要包含：

① Android RIL 子系统： Radio Interface Layer（简称：RIL子系统，即：无线电接口层子系统）用于管理用户的电话、短信、数据通信等相关功能，它是每个移动通信设备必备的系统。

② Android Input 子系统： Input 输入子系统用来处理所有来自己用户的输入数据，如：触摸屏，声音控制物理按键等。

③ Android GUI 子系统： GUI 即：图形用户接口子系统，也就是所谓的图形界面，它用来负责显示系统图形化界面，形象让用户和系统操作及信息进行交互。Android 的 GUI 系统和其它各子系统关系密切相关，是 Android 中最重要的子系 统之一，如：绘制一个 2D 图形、通过 OpenGL 库处理 3D 游戏、通过 SurfaceFlinger（Android 中图形混合器，用于将屏幕上显示的多个图形进行混合显示）来重叠几个图形界面。

④ Android Audio 子系统： Android 的音频处理子系统，主要用于音频方面的数据流传输和控制功能，也负责音频设备的管理。 Android 的 Audio 系统和多媒体处理紧密相连，如：视频的音频处理和播放、电话通信及录音等。

⑤ Android Media 子系统： Android 的多媒体子系统，它是 Android 系统中最庞大的子系统，与硬件编解码、OpenCore 多媒体框架、 Android 多媒体框架等相关，如：音频播放器，视频播放器，Camera 摄像预览等。

⑥ Android Connectivity 子系统： Android 连接子系统是智能设备的重要组成部分，它除了一般所谓的网络连接，如：以太网、WIFI 外， 还包含：蓝牙连接、GPS 定位连接、NFC 等。

⑦ Android Sensor 子系统： Android 的传感器子系统为当前智能设备大大提高了交互性，它在创新应用程序和应用体验里发挥了重 要作用，传感器子系统和手机的硬件设备紧密相关，如： gyroscope 陀螺仪、 accelerometer 加速度计、proximity 距离感应器、magnetic 磁力传感器等。

Android 应用程序开发过程

- Android 应用程序开发是基于 Android 架构提供的 API 和类库编写程序，这些应用程序是完全的 Java 代码程序，它们构建在 Android 系统提供的 API 之上。

- Android 开发方式：开发 Android 应用程序可以基于 Google 提供的 Android SDK 开发工具包，也可以直接在 Android 源码中进行编写。

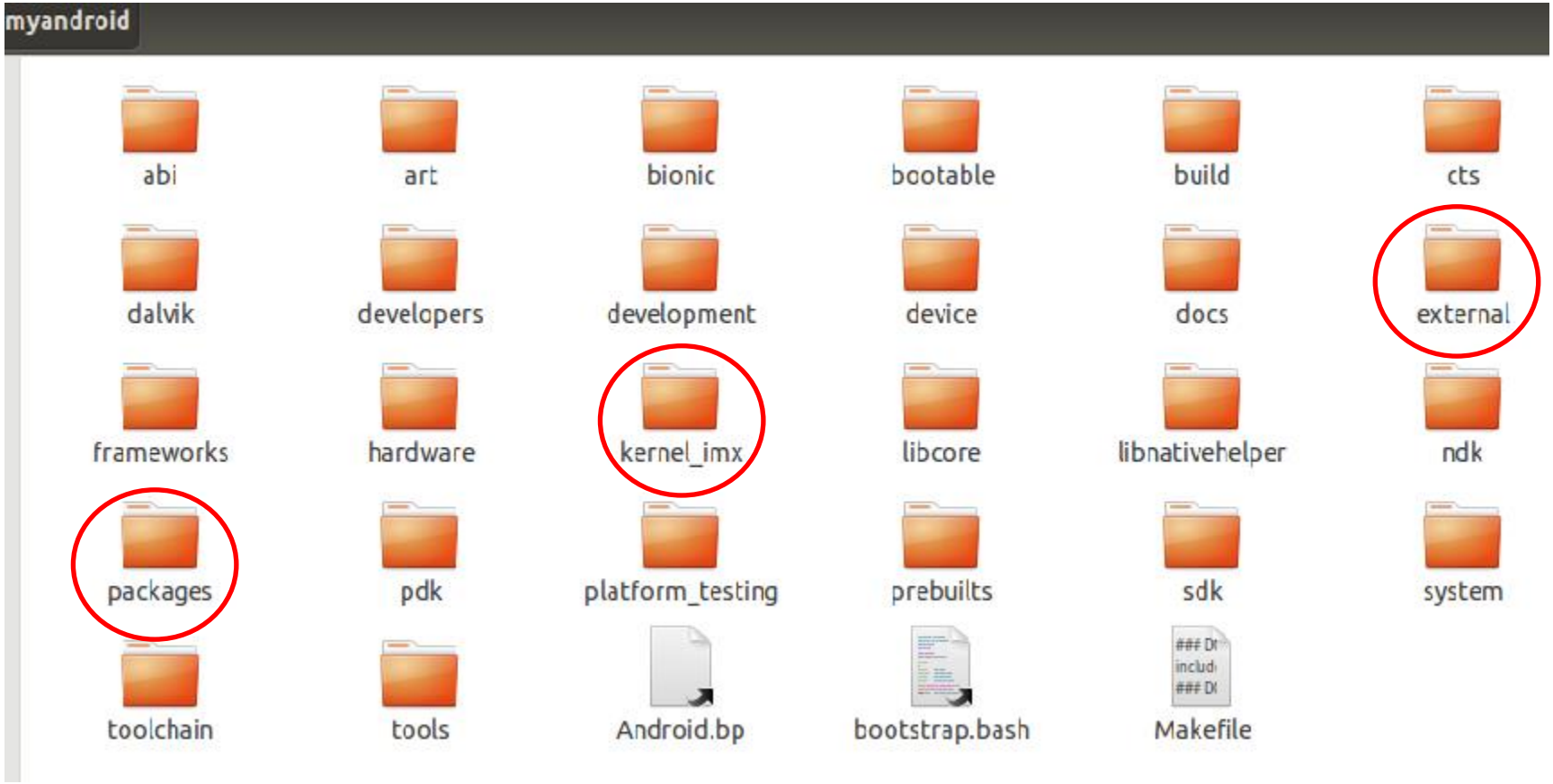
  ① Android SDK 开发：它提供给程序员一种最快捷的开发方式，基于 IDE 开发环境（Android Studio）和 SDK 套件（Android SDK），快速开发出标准的 Android 应用程序，但是，对于一些要修改框架代码或基于自定义 API 的高级开发，这种方式难以胜任。

  ② Android 源码开发：基于 Android 提供的源码进行开发，可以最大体现出开源的优势，让用户自定义个性的 Android 系统，开发出更高效、更与众不同的应用程序，这种方式更适合于系统级开发，对程序员要求比较高。

API：Application Programming Interface，应用程序接口

SDK：Software Development Kit，软件开发工具包

Android 源码目录结构



myandroid.tar.xz

/Android/myandroid/

第12章　块设备和驱动程序设计

- 12.1　块设备驱动程序设计概要
- 12.2　Linux块设备驱动相关数据结构与函数
- 12.3　块设备的注册与注销
- 12.4　块设备初始化与卸载
- 12.5　块设备操作
- 12.6　请求处理
- 12.7　MMC卡驱动

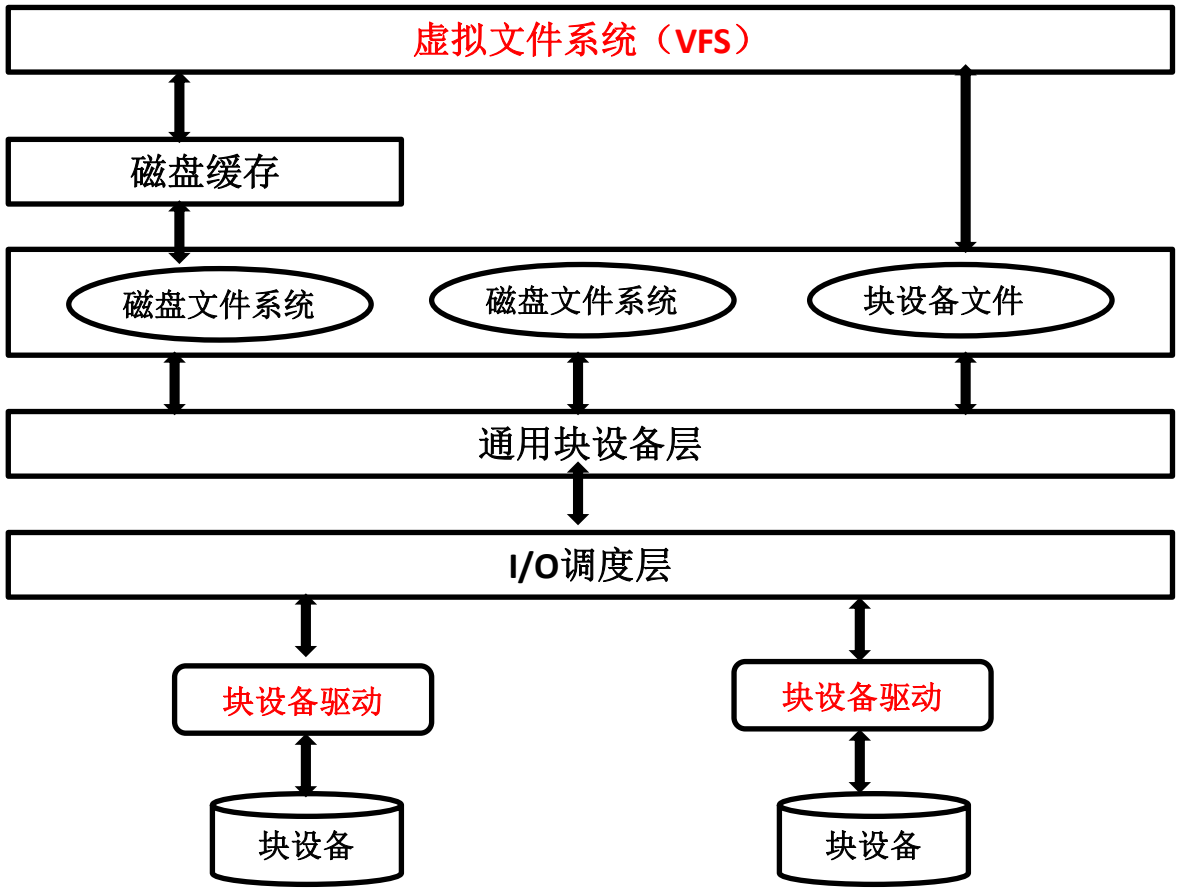
Linux块设备

- 块设备是Linux三大设备之一（另外两种是字符设备，网络设备），块设备也是通过/dev下的文件系统节点访问。
- 块设备的数据存储单位是块，块的大小通常为512B至32KB不等。
- 块设备每次能传输一个或多个块，支持随机访问，并且采用了缓存技术。
- 常见的块设备包括IDE硬盘、SCSI硬盘、CD-ROM等等。
  - IDE：Integrated Device Electronics，集成磁盘电子接口
  - SCSI：Small Computer System Interface，小型计算机系统接口

块设备驱动程序设计概要

- 块设备驱动在虚拟文件系统（VFS）中的位置：

VFS（Virtual File System，虚拟文件系统）的作用就是采用标准的Unix系统调用读写位于不同物理介质上的不同文件系统，即为各类文件系统提供了一个统一的操作界面和应用编程接口。VFS是一个可以让open()、read()、write()等系统调用不用关心底层的存储介质和文件系统类型就可以工作的粘合层。



块设备的数据交换方式

- 块设备的数据交换方式：
  - 块设备以块（512B至32KB）为单位进行读写；字符设备以字节为单位进行读写。
  - 块设备有对应的缓冲区，并使用了请求队列对I/O请求进行管理，块设备支持随机访问；字符设备只能顺序访问。

块设备读写请求

- 块设备读写请求：
  - 对块设备的读写都是通过请求实现的。
  - Linux中每一个块设备都有一个I/O请求队列，每个请求队列都有调度器的插口，调度器可以实现对请求队列里请求的合理组织，如合并临近请求，调整请求完成顺序等。
  - Linux 2.6内核有4个I/O调度器（Scheduler）：
    ① No-op I/O scheduler：实现了一个简单的FIFO队列；
    ② Anticipatory I/O scheduler：是目前内核中默认的I/O调度器；
    ③ Deadline I/O scheduler：是针对Anticipatory I/O scheduler的缺点进行改善而来的；
    ④ CFQ I/O schedule：为系统内的所有任务分配相同的带宽，提供一个公平的工作环境，它比较适合桌面环境。

块设备的注册和注销

- 块设备的注册：
  - int register_blkdev(unsigned int major, const char *name);
    - major：主设备号
    - name：设备名

- 块设备的注销
  - int unregister_blkdev(unsigned int major, const char* name);
    - major：主设备号
    - name：设备名

块设备的初始化和卸载

- 块设备的初始化过程主要完成以下的工作：
  ① 注册块设备及块设备驱动程序；
  ② 分配、初始化、绑定请求队列（如果使用请求队列的话）；
  ③ 分配、初始化gendisk，为相应的成员赋值并添加gendisk；
  ④ 其他初始化工作，如申请缓存区，设置硬件尺寸（不同设备，有不同的处理）。

- 块设备的卸载过程刚好与初始化过程相反：
  ① 删除请求队列；
  ② 撤销gendisk的引用，并删除gendisk；
  ③ 释放缓冲区，撤销对块设备的应用，注销块设备驱动。

块设备操作数据结构

- 块设备操作数据结构：**struct block_device_operations**
  - 字符设备文件操作数据结构： **struct file_operations**

struct block_device_operations
块设备操作结构体

```
struct block_device_operations {
        int (*open) (struct block_device *, fmode_t);
        int (*release) (struct gendisk *, fmode_t);
        int (*ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);
        int (*locked_ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);
        int (*compat_ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);
        int (*direct_access) (struct block_device *, sector_t,  void **, unsigned long *);
        int (*media_changed) (struct gendisk *);
        int (*revalidate_disk) (struct gendisk *);
        int (*getgeo)(struct block_device *, struct hd_geometry *);
        struct module *owner;
};
```

块设备操作

① 打开和释放

- int (*open) (struct block_device *, fmode_t);
- int (*release) (struct gendisk *, fmode_t);

② I/O操作

- int (*ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);
- int (*locked_ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);
- int (*compat_ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);

③ 介质改变

- int (*media_changed) (struct gendisk *);

④ 使介质有效

- int (*revalidate_disk) (struct gendisk *);

⑤ 获得驱动器信息

- int (*getgeo)(struct block_device *, struct hd_geometry *);

⑥ 模块指针

- struct module *owner;

请求处理

- 块设备没有read和write操作函数。
- 对块设备的读写是通过请求函数完成的。
- 请求处理分为两种情况：
  1. 使用请求队列
     ① 请求函数
     ② 通告内核
     ③ 屏障请求和不可重试请求
  2. 不使用请求队列

MMC卡/SD卡

- MMC卡/SD卡介绍：
  - MMC卡（Multi-Media Card，多媒体卡）：1997年由西门子公司和SanDisk公司共同开发，基于东芝公司的NAND Flash技术。
  - SD卡（Secure Digital Memory Card，安全数码卡）：SD卡是由松下电器、东芝和SanDisk联合推出，1999年8月发布。
  - SD卡的数据传送和物理规范由MMC卡发展而来，大小和MMC卡（32mm×24mm×1.4mm）差不多，尺寸为32mm x 24mm x 2.1mm，长宽和MMC卡一样，只是比MMC卡厚了0.7mm，以容纳更大容量的存贮单元。

正面

反面

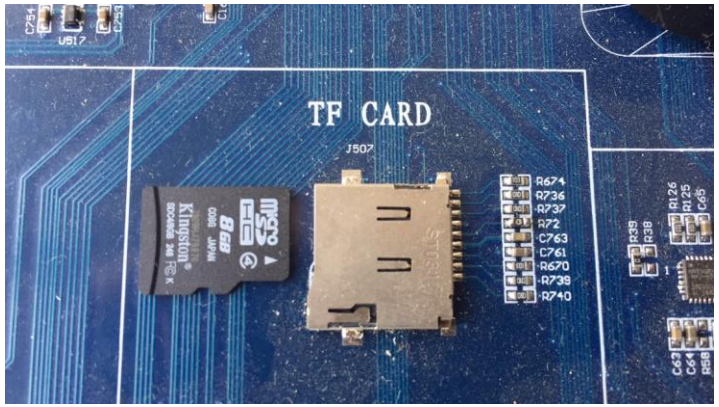
MMC卡

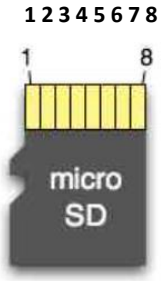
反面

SD卡

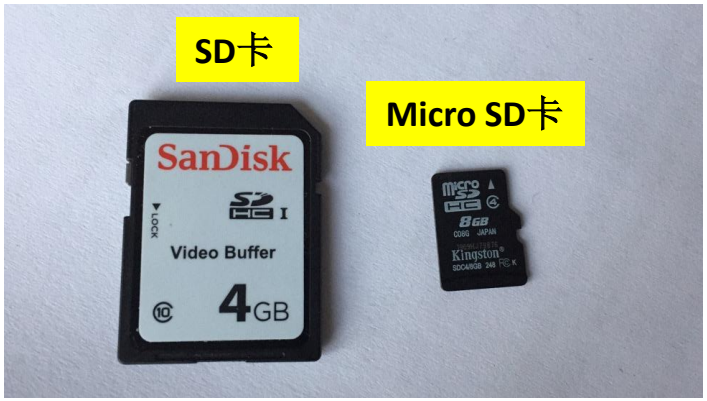
Micro SD卡（TF卡）

- **Micro SD卡（TF卡）：**
  - **Micro SD Card，原名Trans-flash Card（TF卡），2004年正式更名为Micro SD Card，由SanDisk（闪迪）公司发明，主要用于移动电话。**





Pin	SD	SPI
1	DAT2	X
2	CD/DAT3	CS
3	CMD	DI
4	VDD	VDD
5	CLK	SCLK
6	VSS	VSS
7	DAT0	DO
8	DAT1	X



第13章　网络设备驱动程序开发

- 13.1　以太网基础知识
- 13.2　嵌入式网络设备驱动开发概述
- 13.3　网络设备驱动基本数据结构
- 13.4　网络设备初始化
- 13.5　打开和关闭接口
- 13.6　数据接收与发送
- 13.7　查询状态与参数设置
- 13.8　AT91SAM9G45网卡驱动

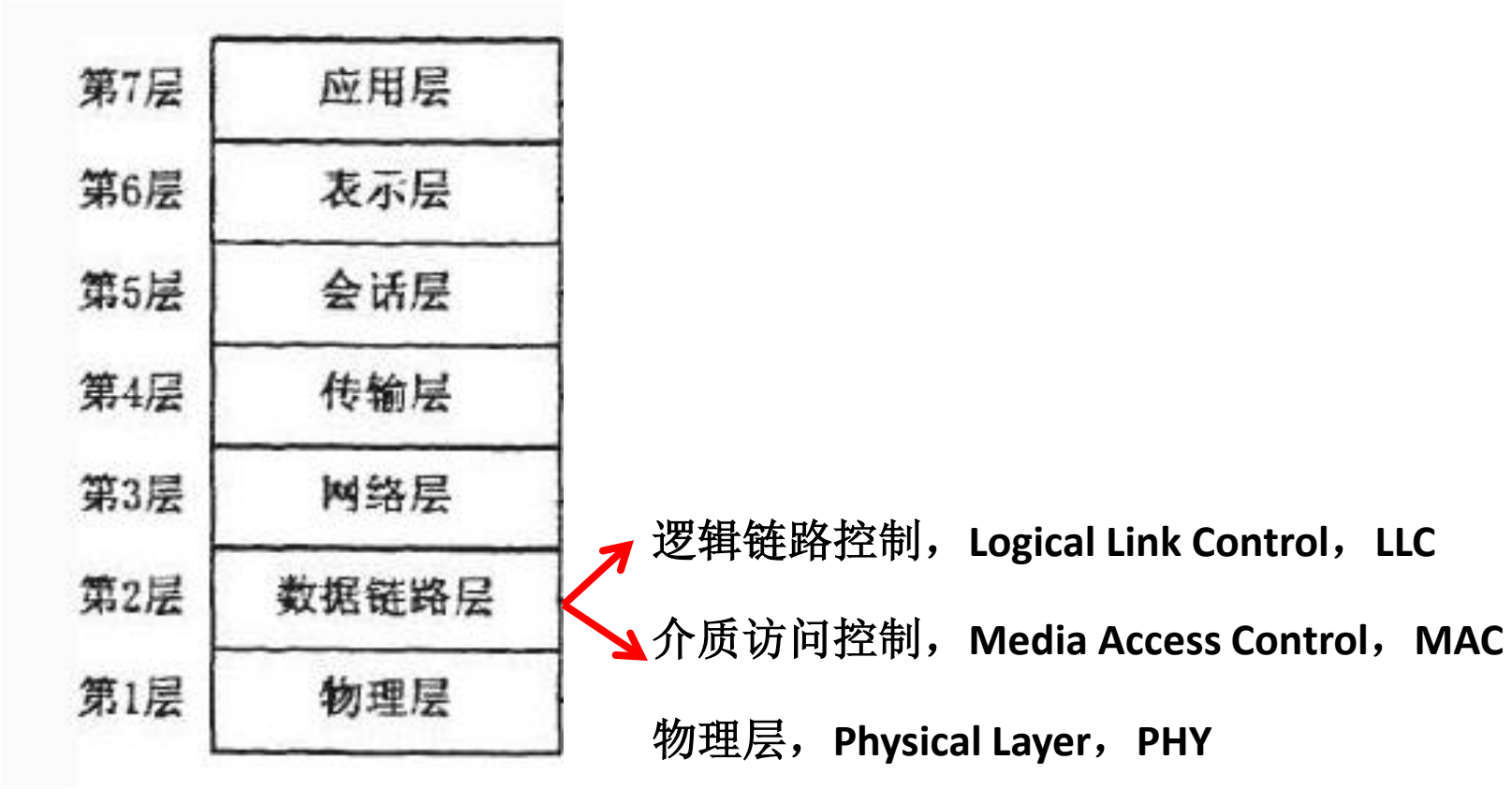
Linux网络设备

- 网络设备是Linux三大设备之一（另外两类是字符设备，块设备）。

- 由于网络设备的特殊工作方式，网络驱动程序的开发与字符设备、块设备驱动的开发有很大的不同。

嵌入式网络设备驱动开发概述

- 硬件描述：
  - 以太网对应于ISO网络分层中的数据链路层和物理层

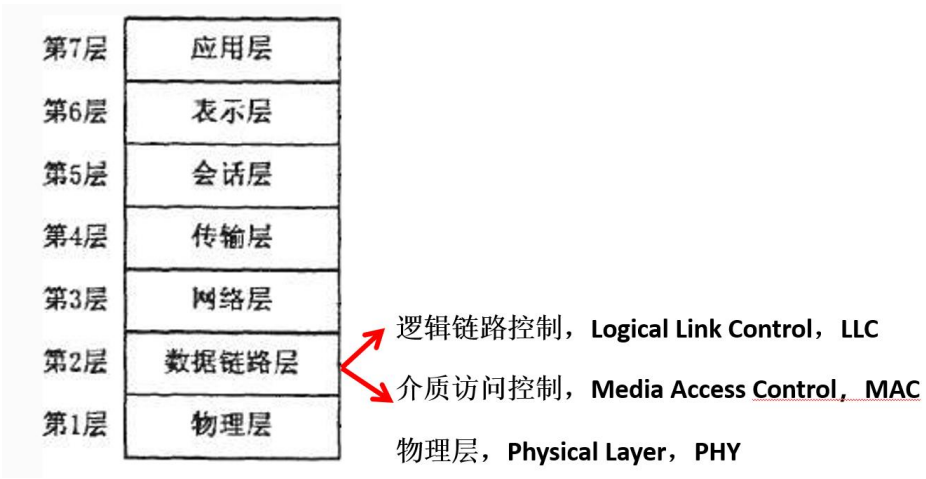


使用嵌入式以太网接口的两种方式

- 使用嵌入式以太网接口的两种方式：

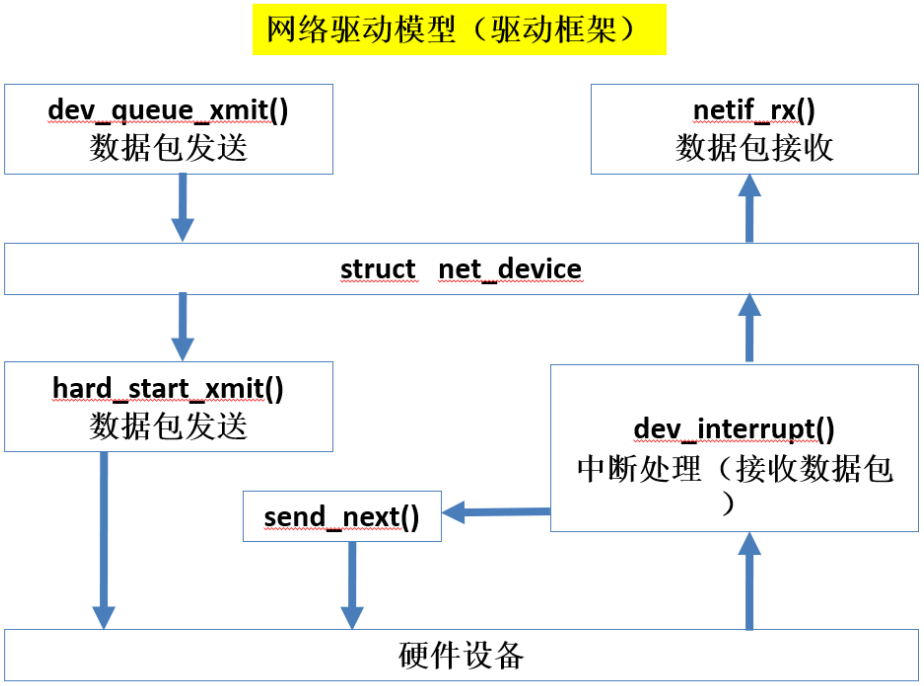
① 嵌入式处理器集成MAC控制器，但是不集成物理层接收器（PHY），此时需要外接PHY芯片，如RTL8201BL、VT6103等芯片。

② 嵌入式处理器既不集成MAC控制器，又不集成物理层接收器（PHY），此时需要外接同时具有MAC控制器和PHY接收器的芯片，如DM9000、CS8900、SIS900等芯片。



驱动框架

- 驱动框架：
  - 在/dev目录下没有对应的设备文件。
  - 对网络设备的访问必须使用套接字（Socket），而非读写设备文件。



网络设备驱动基本数据结构

- **struct net_device**（网络设备结构体）：包含具体网络设备的各种信息和操作接口。

- **struct sk_buff**（套接字缓冲结构体）：是Linux网络子系统的核心，在Linux网络子系统各层协议中的数据传递实际上传递的是struct sk_buff结构，它为各层之间的数据交换单元提供了统一的定义。

网络设备初始化

- 网络设备初始化由struct net_device数据结构的init函数指针指向的函数完成：
  - int (*init)(struct net_device *dev);

- 初始化工作包括以下几个方面的任务：
  ① 检测网络设备的硬件特征，检查物理设备是否存在；

  ② 如果检测到网络设备存在，则进行资源配置；

  ③ 对struct net_device成员变量进行赋值。

打开和关闭接口

- 打开接口的工作由struct net_device数据结构的open函数指针指向的函数完成：
  - int (*open)(struct net_device *dev);
- 该函数负责的工作包括请求系统资源，如申请I/O区域、DMA通道及中断等资源，并告知接口开始工作，调用netif_start_queue函数激活网络设备发送队列：
  - void netif_start_queue(struct net_device *dev);

打开队列

- 关闭接口的工作由struct net_device数据结构的stop函数指针指向的函数完成：
  - int (*stop)(struct net_device *dev);
- 该函数需要调用netif_stop_queue函数停止数据包传送：
  - void netif_stop_queue(struct net_device *dev);

停止队列

数据接收与发送

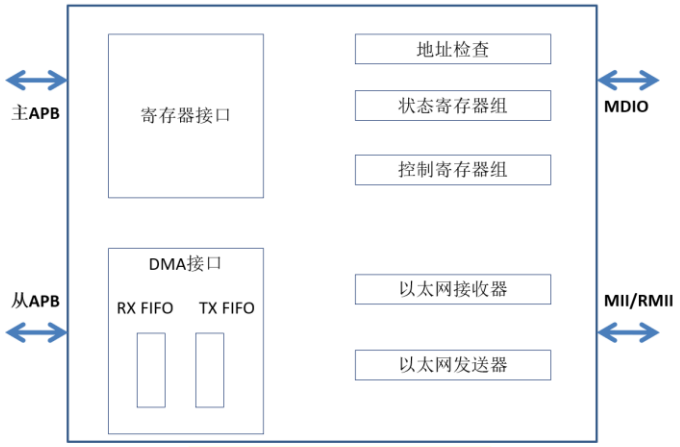
- 1、数据发送
  - 数据在实际发送的时候会调用struct net_device数据结构的hard_start_transmit函数指针指向的函数，该函数会将要发送的数据放入外发队列，并启动数据包发送。
- 2、并发控制
  - 发送函数可利用struct net_device数据结构的xmit_lock自旋锁来保护临界区资源。
- 3、传输超时
  - 驱动程序需要处理超时带来的问题，调用struct net_device数据结构的tx_timeout函数，并调用netif_wake_queue函数重启设备发送队列。
- 4、数据接收
  - 在Linux中有两种方式实现数据接收：
    - 第一种是中断方式：调用netif_rx函数实现数据接收；
    - 第二种是轮询方式：调用netif_receive_skb函数实现数据接收。

查询状态与参数设置

- 1、链路状态
  - 驱动程序需要掌握当前链路的状态，当链路状态改变时，驱动程序需要通知内核：
    - void netif_carrier_off(struct net_device *dev);
    - void netif_carrier_on(struct net_device *dev);
    - int netif_carrier_ok(const struct net_device *dev);
- 2、设备状态
  - 驱动程序的get_stats()函数用于向用户返回设备的状态和统计信息，这些信息保存在struct net_device_stats结构体中。
- 3、设置MAC地址
  - 调用set_mac_address函数，设置新的MAC地址。
- 4、接口参数设置
  - 调用set_config函数，设置I/O地址、中断等信息。

AT91SAM9G45网卡驱动

- EMAC模块简介：
  - AT91SAM9G45的EMAC模块是一个完全兼容IEEE 802.3标准的10/100M的以太网控制器，它包含一个地址检查模块、统计和控制寄存器组、接收和发送模块，以及一个DMA接口。
  - AT91SAM9G45：嵌入式处理器
- 模块图：



AT91SAM9G45的EMAC模块

第14章 嵌入式GUI及应用程序设计

- 14.1 嵌入式GUI设计概述
- 14.2 嵌入式GUI体系结构设计
- 14.3 基于主流GUI的应用程序设计

嵌入式GUI简介

- GUI：Graphical User Interface，图形用户界面。

- 嵌入式GUI设计包括以下3个方面的内容：
  ① 硬件设计：通过LCD控制器，将LCD显示器与开发板连接起来。
  ② 驱动程序设计：为LCD设计驱动程序，并移植嵌入式GUI系统，为上层应用程序设计提供图形函数库。
  ③ 用户界面程序设计：使用嵌入式系统提供的函数库，进行图形化应用程序设计。

- 嵌入式GUI分为以下3大类：
  ① 与操作系统结合的GUI：这些GUI一般由有操作系统开发实力的大公司开发，如微软的Windows Phone（其前身是Windows CE和Windows Mobile）、苹果公司的iOS等。
  ② 外挂GUI：这些GUI通常基于操作系统运行，向应用层提供开发接口，如Android、Qt/E、MiniGUI、Microwindows等。
  ③ 简单GUI：这些GUI通常与应用程序结合在一起，可重用性较差。

主流嵌入式GUI简介（1）

- 1、Qt/E。Qt/Embedded是面向嵌入式系统的Qt版本。 Qt/Embedded是一个C++函数库。是一个多平台的C++图形用户界面应用程序框架，能给用户提供精美的图形用户界面所需要的所有元素。

- 2、MiniGUI。MiniGUI是一个自由软件项目，其目标是为基于Linux的实时嵌入式系统提供一个轻量级的图形用户界面支持系统，比较适合工控领域的应用。MiniGUI具有：方便的编程接口、使用了图形抽象层和输入抽象层、多字体和多字符集支持、多线程机制的特点。

- 3、MicroWindows。Qt/Embedded是一个开放源码的项目。是一个基于典型客户/服务器体系结构的GUI系统。 Qt/Embedded有三层：最底层是面向图形输出和键盘、鼠标或触摸屏的驱动程序；中间层提供底层硬件的抽象接口，并进行窗口管理；最高层分别提供兼容于X Window和Windows CE的API。

- 4、Tiny-X。Tiny-X是标准X Window系统的简化版，去掉了许多对设备的检测过程，无须设置显示卡驱动，很容易对各种不同硬件进行移植。其设计目的是为了在小容量内存的环境下运行，非常适合用作嵌入式Linux的GUI系统。

主流嵌入式GUI简介（2）

- 5、Android。Android的应用程序都是使用Java来编写的，因此很容易移植到新的硬件平台上，用户可以使用Google提供的SDK平台来设计与开发Android周边应用，此外，Android平台还包括3D图形加速引擎、SQLite支持、Webkit支持等特性。

- 6、Windows CE。是Microsoft针对嵌入式产品的一套模块化设计的操作系统。为用户提供良好的GUI。 Windows CE的基本GUI模块包括：窗口管理模块、COM组件、窗口控制组件。

- 7、Palm。Palm OS是Palm公司研制的专门用于其掌上电脑产品Palm的操作系统。Palm操作系统的特点是简单易用，运行需要的内存与处理器资源较小，速度也很快，Palm操作系统不支持多线程。

- 8、iOS。iOS是以Darwin为基础的，属于类UNIX的商业操作系统。iOS用户界面的创新设计是多点触控。控制方法包括滑动、轻触开关和按键，交互方式包括滑动、轻按、挤压和旋转等。

嵌入式GUI体系结构（1）

- 抽象层
  - 包括操作系统抽象层、硬件输入抽象层、图形输出抽象层。
  - 1、操作系统抽象层：主要用来隔离具体的操作系统。
  - 2、硬件输入抽象层：主要用来实现硬件输入功能。
  - 3、图形输出抽象层：主要用来实现图形输出功能。

嵌入式GUI体系结构（2）

- 核心层
  - 1、消息管理。其主要任务是保证消息能够正常地发送、传递、捕获和处理。大部分GUI系统采用事件和消息驱动机制作为系统的基本通信机制。
  - 2、内存管理。在GUI初始化之初就申请一块连续的共享内存，用链表把此块内存管理起来，避免在应用程序中频繁动态地申请和释放内存时造成大量的内存碎片。
  - 3、窗口管理。负责窗口的分类、窗口树和Z序的管理、窗口剪切域的管理，以及窗口绘制和跟GDI模块的交互等。
  - 4、资源管理。资源是指GUI中所使用到的图片、字体库等。GUI把所有需要用到的图片数据进行预处理，调用时可大大提高效率。
  - 5、定时器管理。根据操作系统时钟，为GUI提供定时服务。
  - 6、图形设备接口。完成点、线、矩形、椭圆、多边形等绘制的基本操作。

嵌入式GUI体系结构（3）

- 接口层
  - 接口层提供各种GUI对象（窗口、控件）的数据结构、应用编程接口以及绘制接口。
  - 数据结构包括各种图形设备接口对象的数据结构，如画笔、画刷、背景、位图、字体等。
  - 接口（应用编程接口、绘制接口）包括设备上下文的操作、图形设备接口对象的操作、坐标系统转换、图形绘制单元的操作等。
  - 所有的接口一般以封装的方式提供，不仅可以提高代码的可重用性，还便于开发人员对已有的窗口或控件对象进行扩展。

关于期末考试

- 考试时间：2024年12月30日
- 试卷题型：
  - 填空题：30空，30分
  - 简答题：8小题，35分
  - 综合分析题：7小题，35分

祝同学们期末考试取得好成绩！

Thanks