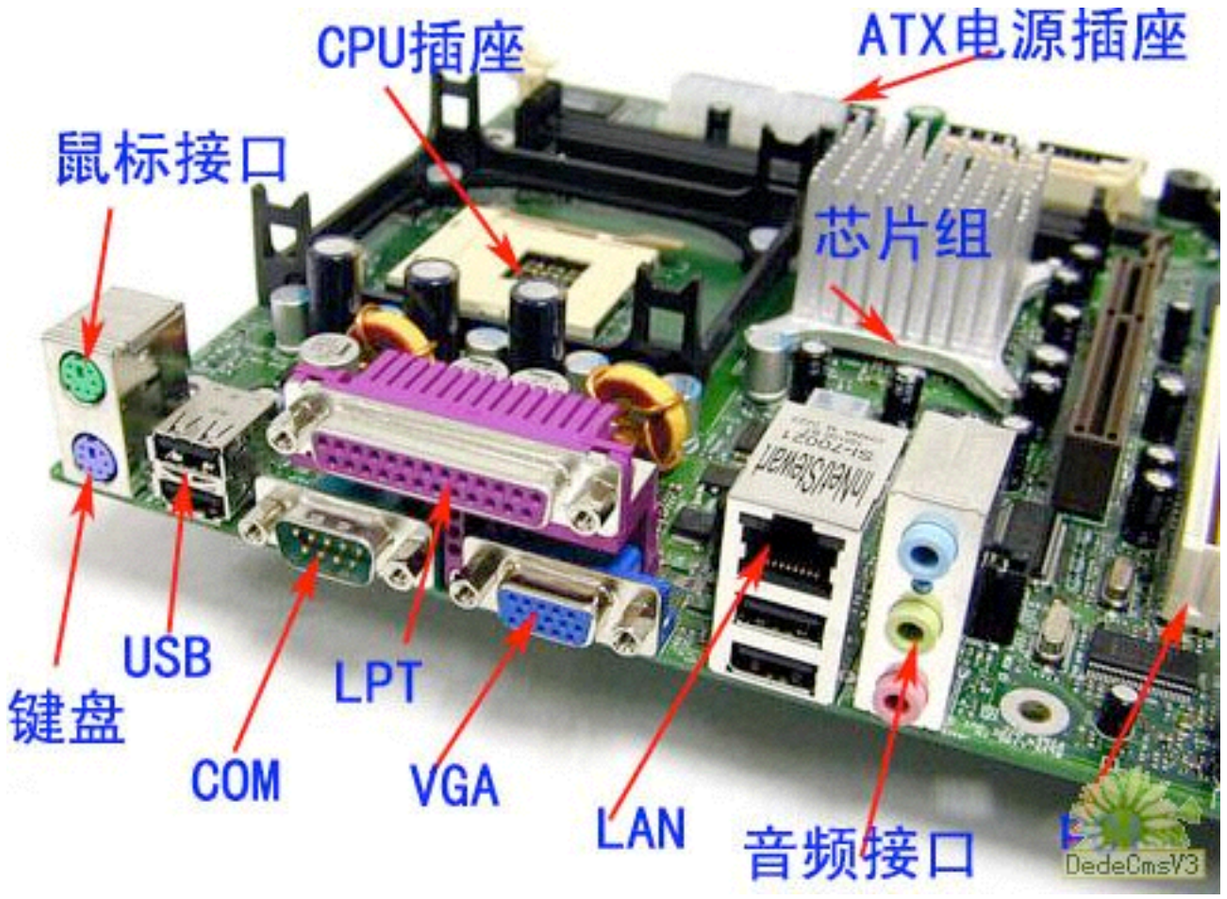
设备驱动程序

http://fm.zju.edu.cn

什么是外部设备?

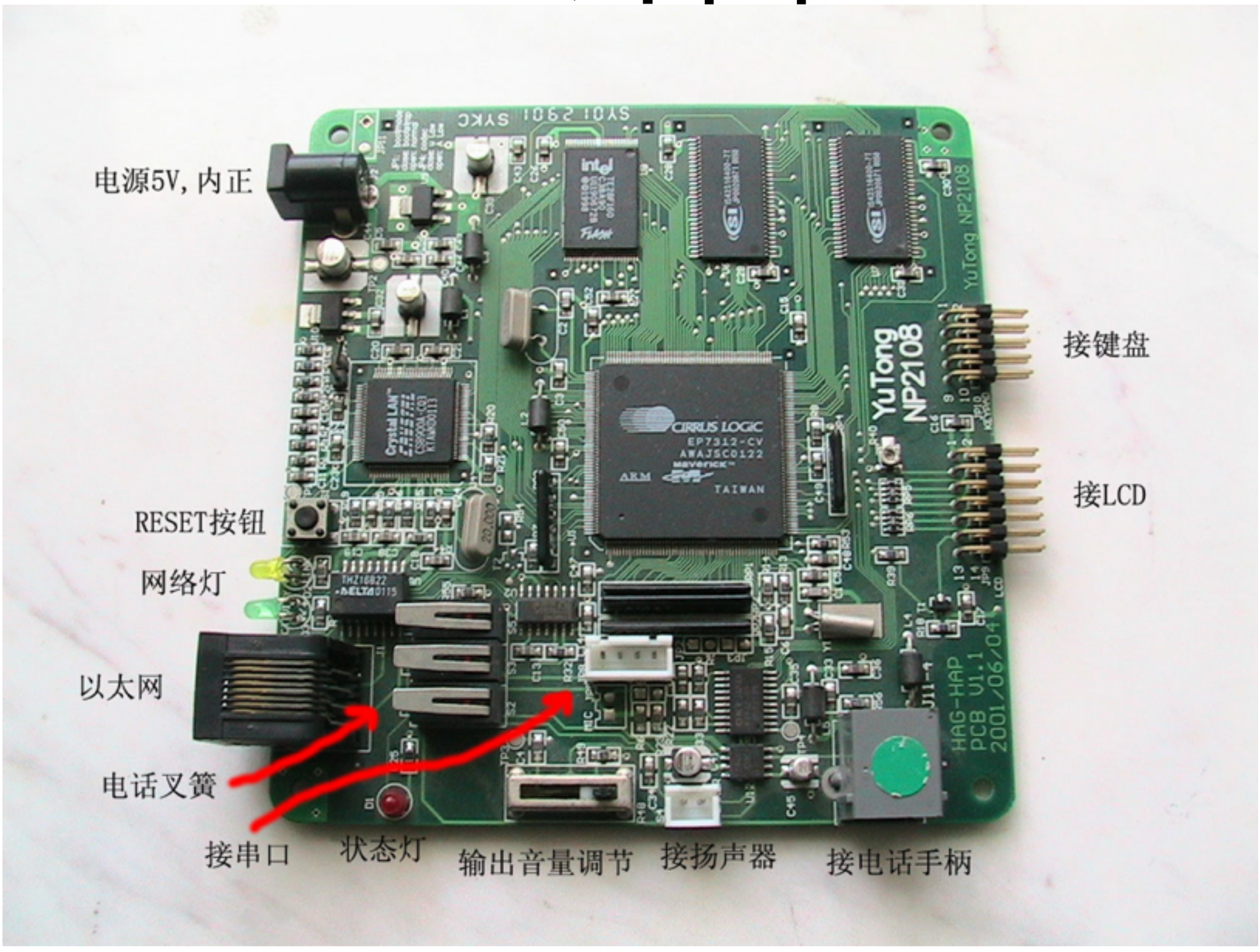


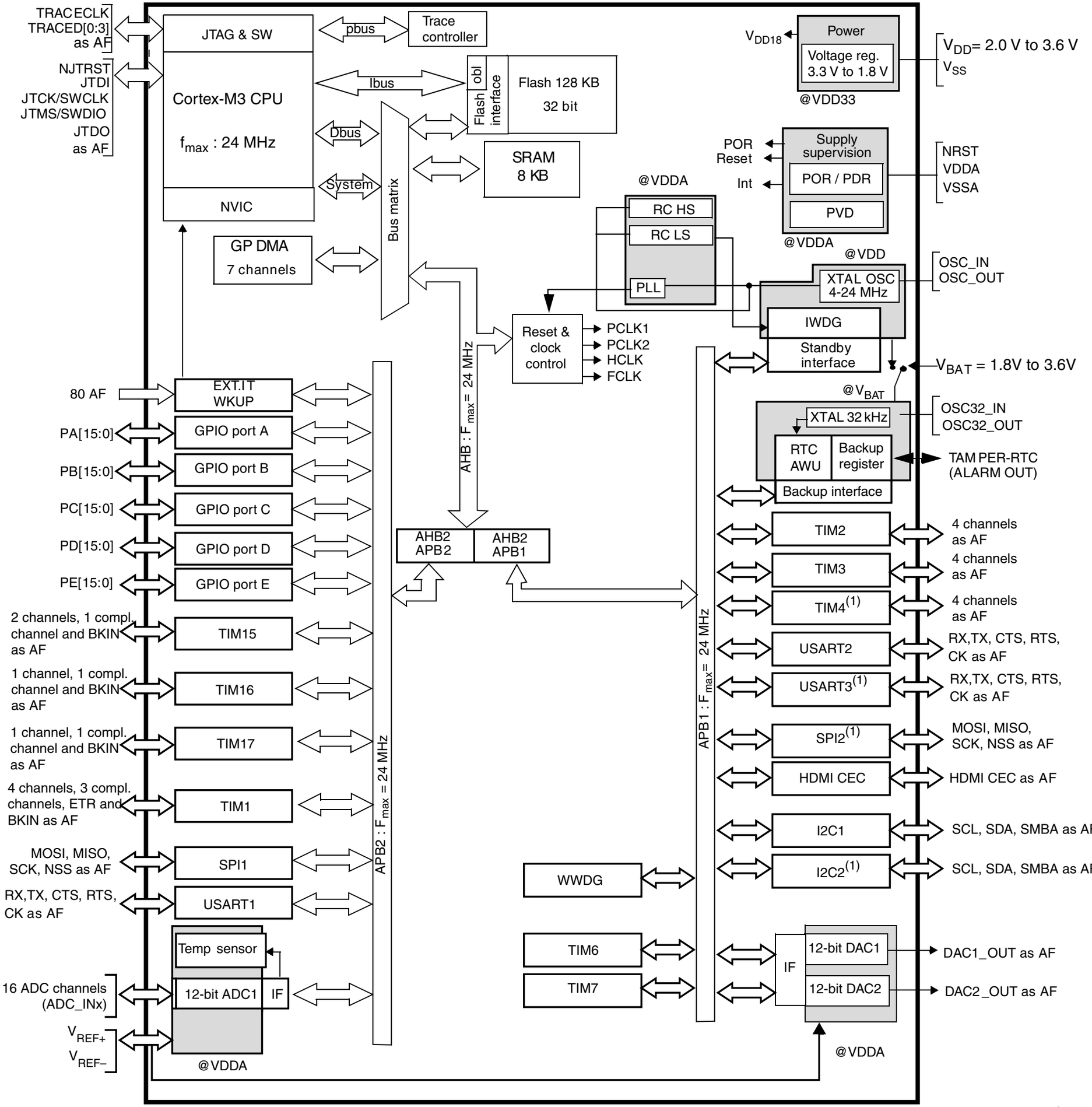
- 键盘? 鼠标? 显示器?

主机箱外面?

- PC用户的习惯认识

CPU外面?


电源5V, 内正
接键盘
接LCD
RESET按钮
网络灯
以太网
电话叉簧
接串口
状态灯
输出音量调节
接扬声器
接电话手柄
YuTong NP2108
CIRRUS LOGIC
EP7312-CV
AWAJSC0122
ARM
TAIWAN
HAG-HAP
PCB V1.1
2001/06/04


TRACECLK
TRACED[0:3]
as AF
NJTRST
JTDI
JTCK/SWCLK
JTMS/SWDIO
JTDO
as AF
JTAG & SW
Cortex-M3 CPU
f_{max} : 24 MHz
NVIC
pbus
Trace controller
Ibus
Dbus
System
obl
Flash interface
Flash 128 KB
32 bit
SRAM
8 KB
Bus matrix
GP DMA
7 channels
V_{DD18}
Power
Voltage reg.
3.3 V to 1.8 V
@VDD33
V_{DD}= 2.0 V to 3.6 V
V_{SS}
POR
Reset
Int
Supply supervision
POR / PDR
PVD
@VDDA
NRST
VDDA
VSSA
@VDDA
RC HS
RC LS
PLL
@VDD
XTAL OSC
4-24 MHz
IWDG
Standby interface
OSC_IN
OSC_OUT
V_{BAT} = 1.8V to 3.6V
@V_{BAT}
XTAL 32 kHz
RTC
AWU
Backup register
Backup interface
OSC32_IN
OSC32_OUT
TAM PER-RTC
(ALARM OUT)
Reset & clock control
PCLK1
PCLK2
HCLK
FCLK
AHB : F_{max}= 24 MHz
AHB2
APB2
AHB2
APB1
APB1 : F_{max}= 24 MHz
APB2 : F_{max} = 24 MHz
80 AF
EXT.IT
WKUP
PA[15:0]
GPIO port A
PB[15:0]
GPIO port B
PC[15:0]
GPIO port C
PD[15:0]
GPIO port D
PE[15:0]
GPIO port E
2 channels, 1 compl. channel and BKIN as AF
TIM15
1 channel, 1 compl. channel and BKIN as AF
TIM16
1 channel, 1 compl. channel and BKIN as AF
TIM17
4 channels, 3 compl. channels, ETR and BKIN as AF
TIM1
MOSI, MISO, SCK, NSS as AF
SPI1
RX,TX, CTS, RTS, CK as AF
USART1
Temp sensor
16 ADC channels
(ADC_INx)
12-bit ADC1
IF
V_{REF+}
V_{REF-}
@VDDA
WWDG
TIM6
TIM7
TIM2
TIM3
TIM4(1)
USART2
USART3(1)
SPI2(1)
HDMI CEC
I2C1
I2C2(1)
IF
12-bit DAC1
12-bit DAC2
@VDDA
4 channels as AF
4 channels as AF
4 channels as AF
RX,TX, CTS, RTS, CK as AF
RX,TX, CTS, RTS, CK as AF
MOSI, MISO, SCK, NSS as AF
HDMI CEC as AF
SCL, SDA, SMBA as AF
SCL, SDA, SMBA as AF
DAC1_OUT as AF
DAC2_OUT as AF

对于嵌入式设备

- 无论片内片外，CPU核以外的设备都可称外部设备，因为都需要写程序去操纵

对于嵌入式设备

硬盘是不是外设?

- 无论片内片外，CPU核以外的设备都可称外部设备，因为都需要写程序去操纵

对于嵌入式设备

硬盘是不是外设?

内存是不是外设?

- 无论片内片外，CPU核以外的设备都可称外部设备，因为都需要写程序去操纵

对于嵌入式设备

硬盘是不是外设?

内存是不是外设?

网卡是不是外设?

- 无论片内片外，CPU核以外的设备都可称外部设备，因为都需要与程序去操纵

关于外部设备控制

- 嵌入式系统的几乎每种操作，最后都要映射到实际的物理设备上。
- 除处理器、内存和少数其他实体外，几乎所有设备的控制操作都由设备相关的设备驱动程序来实现。
- 嵌入式系统开发平台功能成熟度的重要指标：是否具有大量、丰富的设备驱动及BSP（板级支持包）支持。

设备驱动程序的重要性

- 设备驱动程序往往工作于系统内核状态，因而，其运行性能、可靠性制约着应用系统的性能和可靠性。

- 由于设备驱动错误，导致操作系统崩溃约占30－40%。

设备驱动程序的重要性

- 串口设备驱动程序的缓冲区溢出bug：

```
#define MaxFrameLen 1500
BYTE recBuf[MaxFrameLen];
WORD pos = 0;

VOID
RS232_12_Interrupt1_HandlerRoutine(
    RS232_12_HANDLE hRS232_12,
    RS232_12_Interrupt1_RESULT *intResult ) {
    recBuf[pos]=RS232_12_ReadByte(hRS232_12,(RS232_12_ADDR)0,0);
    if(++pos>=1000) {
        if (m_pEventToSignal) m_pEventToSignal->Set();
    }
}
```

读取串口数据

通知应用程序读取串口数据

- 为串口数据存储设置的缓冲区为1500字节，RS232的传输速率在9600bps-115200bps左右，如果应用程序在0.104秒-1.25秒之内不读取数据的话，1500字节的缓冲区就会发生溢出，从而导致操作系统内核数据区的数据损坏。

对于嵌入式Linux

- 内核发行包一般支持内存、网卡、显卡、硬盘等常规设备
- 本课的研究范围，是指内核发行包一般不支持的外设
  - 移植
  - 新设备

CPU如何连接外设

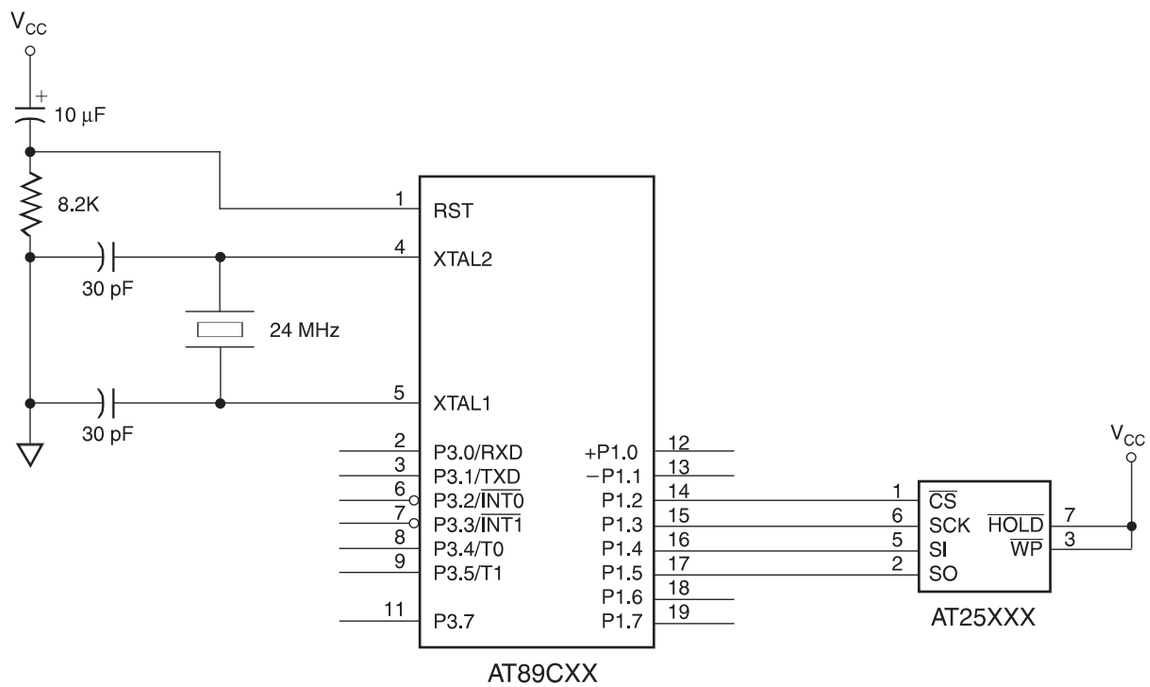
- 专门的指令（x86）： PC串口、并口
- 地址/数据总线： flash、网卡、GPIO、VGA
  - 其他总线： PCI
  - DMA
- SFR操作

SFR

- GPIO/INT
- I2C
- SPI
- UART
- AD/DA
- PWM
- RTC
- USB
- MAC
- CAN

二层接口外设

- 外设通过某种接口接入CPU
- 设备驱动程序要实现外设操作，首先要实现基础的接口操作



程序操纵外设?

- 裸机写程序很自由:

```
char *p = (char*)0x00100011;
*p = 0x01;
SBUF = ‘H’;
```

Linux的设备驱动

- 直接访问外设使得OS无法管理资源
- MMU使得外设所使用的地址发生变化
- Linux的内核态和应用程序的地址空间不同

Linux的设备驱动

- 需要设备驱动程序做为应用程序和实际硬件之间的桥梁，应用程序通过某种手段向OS发出请求，由OS的设备驱动程序进行实际的操作

Linux的设备驱动

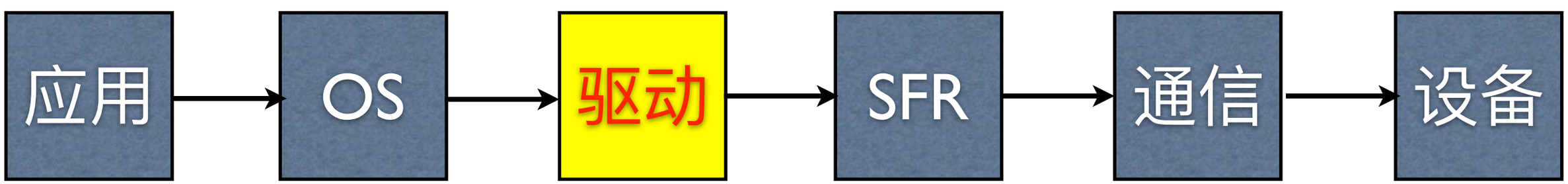
- 需要设备驱动程序做为应用程序和实际硬件之间的桥梁，应用程序通过某种手段向OS发出请求，由OS的设备驱动程序进行实际的操作

Linux的设备驱动

- 需要设备驱动程序做为应用程序和实际硬件之间的桥梁，应用程序通过某种手段向OS发出请求，由OS的设备驱动程序进行实际的操作
  - 直接访问外设使得OS无法管理资源
  - MMU使得外设所使用的地址发生变化

写驱动必须的知识

- OS如何提供接口给应用程序
- 如何驱动CPU实现与外设的通信
- 如何通过与外设的通信操纵外设



外部设备的分类

- 外部设备主要分为三类
  - 字符设备
  - 块设备
  - 网络设备

字符设备

- 能够像字节流一样被访问的设备，提供数据通道，不允许来回读写，在数据传输中以字符为单位进行传输。例如：串口、鼠标、终端、打印机
- 字符设备特点：
  - 在数据传输中，可以传输任意大小的字符数据；
  - 可以通过文件方式（/tyCo/0）访问，但只能顺序访问数据通道；
  - 字符设备中的缓存可有可无。

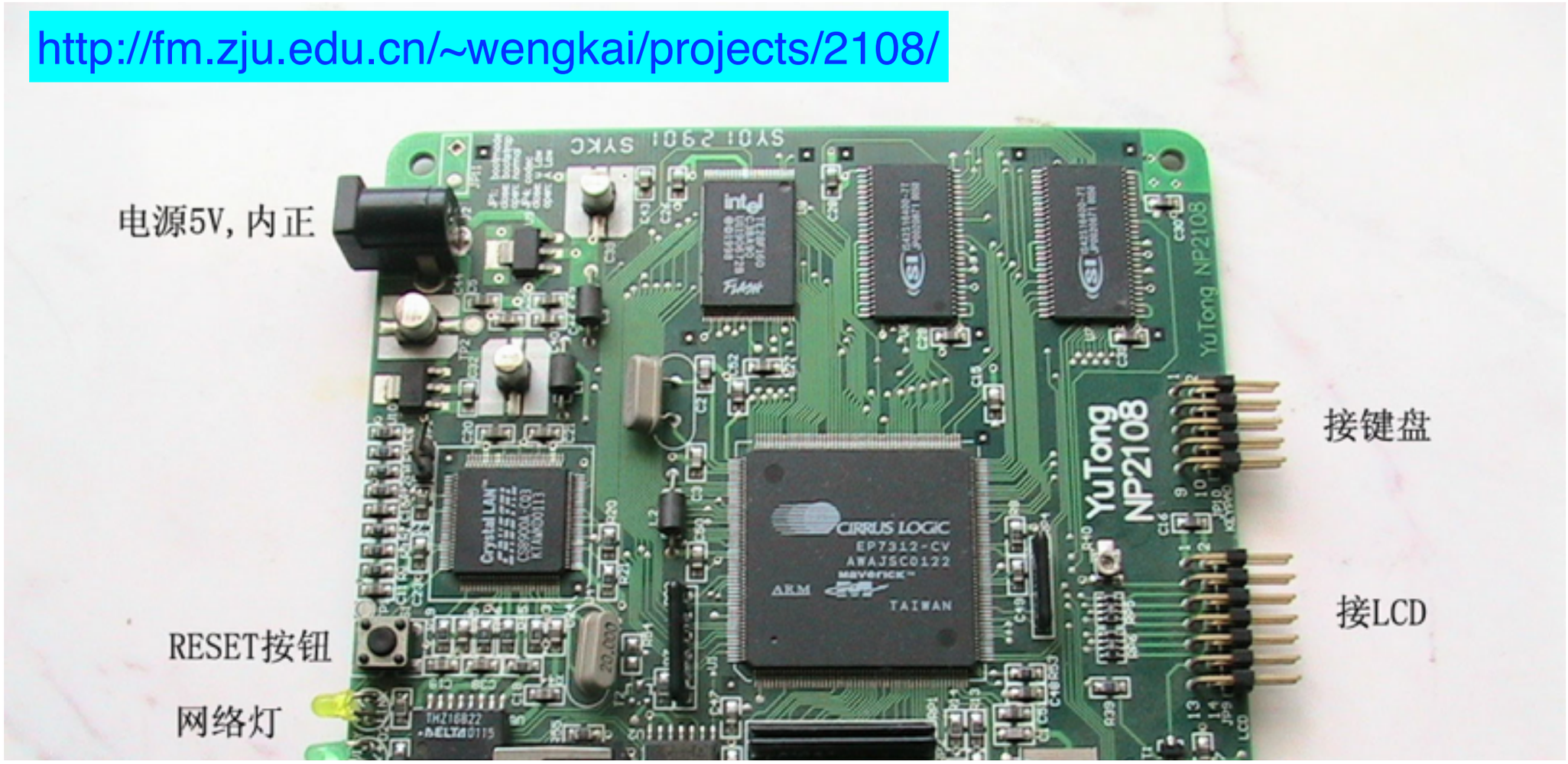
块设备

- 以数据“块”为单位，对数据进行来回读写/存取的设备。1个数据块可包括512B、1KB数据。块设备可以容纳文件系统。
- 块设备包括：硬盘、光驱、软驱等。
- 块设备特点：
  - 在数据传输中，传输单位是固定大小的数据块。
  - 块设备的存取是通过buffer、cache来进行。
  - 可以随机访问块设备中存放的数据块。
  - 块设备不直接与I/O系统连接，而是通过与文件系统关联，提供接口。

网络设备

- 能够与其他主机进行网络通讯的设备。
- 与普通I/O设备不同，网络设备没有对应的设备文件，数据通信不是基于标准的I/O系统接口，而是基于BSD套接口访问，例如：socket、bind、listen、accept、send等系统调用。

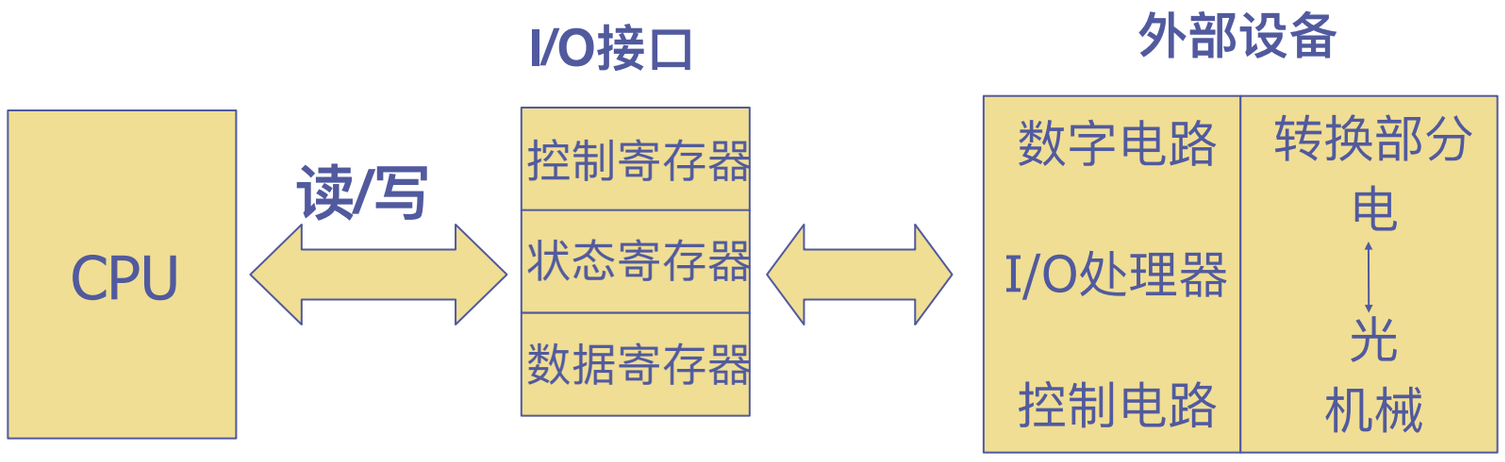
http://fm.zju.edu.cn/~wengkai/projects/2108/



设备	设备文件	驱动程序	说明
LCD	/dev/lcd	/lib/modules/lcd.o	直接写文本信息到这个文件,就可以在LCD上输出文字。回车会换车，能自动卷屏，支持以下esc字符： 01：清屏，02：光标回到行首 使用magic number为0x21400599的ioctl来开关背光
LCD	/dev/lcd	/lib/modules/lcdalt.o	这是一个alternative的LCD驱动，它只能用来点亮LCD的背光灯，而不能输出文字
键盘	/dev/keypad	/lib/modules/keypad.o	直接读这个文件,就可以读到用户在键盘上按的键
叉簧开关	/dev/hook	/lib/modules/hook.o	读（read）这个文件，返回非0表示摘机
音频编解码器	/dev/codec	/lib/modules/codec.o	从这个文件读，就可以读到音频输入口编码产生的G.711的码流；向这个文件写G.711的码流，就可以在扬声器听到声音

设备与CPU的信息交换

- 外部设备与CPU的通讯信息，主要包括：
  - 控制信息：告诉要进行什么处理
  - 状态信息：当前设备的状态
  - 数据信息：不同的设备，传输数据类型及编码各异。



输入、输出方式控制

- 与外部设备的输入/输出的方式，主要包括：
  - 直接控制I/O方式：通过指令直接对端口进行输入、输出控制。如x86的in、out指令。
  - 内存映射方式：可以访问较大的地址空间，实现快速数据交换，如：ISA总线可以映射的空间为0xC8000～0xEFFF。
  - 中断方式：采用中断实现数据的输入/输出。
  - DMA方式：采用DMA控制器，实现数据传输。
- 讨论：
  - 数据量不大的情况下，如字符设备，常采用中断方式；
  - 大量数据传输的I/O设备，如磁盘、网卡等设备，常采用DMA方式，以减少CPU资源占用。

设备驱动程序概述

- 设备驱动程序，是直接控制设备操作的那部分程序，是设备上层的一个软件接口。

- 实际上从软件角度来说，设备驱动程序就是负责完成对I/O端口地址进行读、写操作。

- 设备驱动程序的功能是对I/O进行操作。

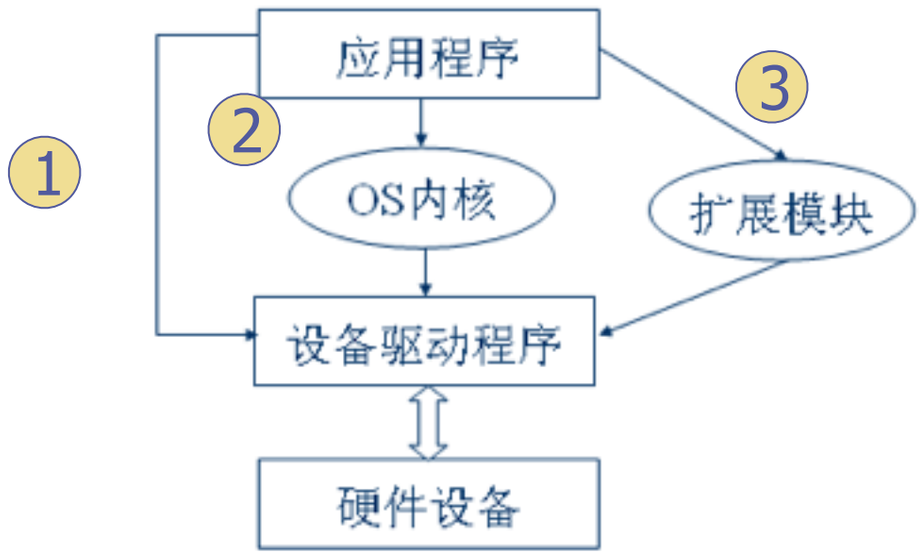
- 设备驱动程序不能自动执行，只能被操作系统，或应用程序调用。

驱动程序 vs BSP

- BSP（Board Support Package）－板级支持包
  - 针对具体目标机平台，所编写的与硬件结构相关的代码部分。
  - 主要完成系统上加电后CPU初始化、各类控制芯片初始化等工作。
    - BSP与bootloader、设备驱动程序关系密切。
- 设备驱动程序
  - 与硬件结构相关的部分，包含在BSP中
  - 通用部分，可以在其他部分中实现

调用设备驱动的方式

- 嵌入式系统中调用设备驱动程序的三种方式：
  - 应用程序直接调用
  - 应用程序通过操作系统内核调用
  - 应用程序通过操作系统扩展模块进行调用



驱动程序的主要功能

- 设备驱动程序的主要功能包括以下6项：
  - 对设备进行初始化
  - 打开设备操作
  - 关闭设备操作
  - 从设备上接收数据，并提交给系统－读操作
  - 把数据从主机上发送给设备－写操作
  - 对设备进行控制操作

设备驱动程序表

- 应用程序对设备进行操作是通过操作系统的I/O管理子系统来进行的。

- I/O管理子系统对设备的管理，是通过“设备驱动程序表”进行的。

- “设备驱动程序表”描述了系统中所有设备驱动程序，以及设备驱动程序所支持操作（open/read/write/ioctl/close）的入口地址。

设备的标识

- 主、从设备号（系统标识法）
  - 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
  - 从设备号：标识使用同一设备驱动程序的不同硬件设备
- 设备名称（外部标识法）
  - I/O管理子系统提供对设备进行命名的接口，以提高设备管理的直观性。

文件形式接口

- Unix将所有设备统一以文件的形式提供接口
- 接口简单、统一
  - open/close, read/write, seek/tell
- 不能照顾千奇百怪的设备操作
  - ioctl

设备操作

- open/close, read/write, seek/tell, ioctl
- 当应用程序打开/dev下的文件，并进行各种操作时，OS将应用程序的调用及参数转给注册过的对应的设备驱动程序的对应函数

/dev

- OS提供给应用的接口

/dev

- OS提供给应用的接口

```
crw-rw-rw-   1 root      wheel         4, 127 Oct 27 13:12 ttywf
crw-rw-rw-   1 root      wheel         8,   1 Oct 27 13:12 urandom
brw-------   1 root      operator      1,   0 Oct 27 13:12 vn0
brw-------   1 root      operator      1,   1 Oct 27 13:12 vn1
```

/dev

字符设

- OS提供给应用的接口

```
crw-rw-rw-   1 root     wheel        4, 127 Oct 27 13:12 ttywf
crw-rw-rw-   1 root     wheel        8,   1 Oct 27 13:12 urandom
brw-------   1 root     operator     1,   0 Oct 27 13:12 vn0
brw-------   1 root     operator     1,   1 Oct 27 13:12 vn1
```

/dev

- OS提供给应用的接口

字符设

```
crw-rw-rw-   1 root     wheel        4, 127 Oct 27 13:12 ttywf
crw-rw-rw-   1 root     wheel        8,   1 Oct 27 13:12 urandom
brw-------   1 root     operator     1,   0 Oct 27 13:12 vn0
brw-------   1 root     operator     1,   1 Oct 27 13:12 vn1
```

块设

/dev

- OS提供给应用的接口

字符设

```
crw-rw-rw-   1 root     wheel        4, 127 Oct 27 13:12 ttywf
crw-rw-rw-   1 root     wheel        8,   1 Oct 27 13:12 urandom
brw-------   1 root     operator     1,   0 Oct 27 13:12 vn0
brw-------   1 root     operator     1,   1 Oct 27 13:12 vn1
```

块设

主设备

/dev

- OS提供给应用的接口

字符设

```
crw-rw-rw-   1 root      wheel        4, 127 Oct 27 13:12 ttywf
crw-rw-rw-   1 root      wheel        8,   1 Oct 27 13:12 urandom
brw-------   1 root      operator     1,   0 Oct 27 13:12 vn0
brw-------   1 root      operator     1,   1 Oct 27 13:12 vn1
```

块设

主设备

子设备

设备号

- 主设备号，标识设备种类；从设备号，标识同一设备不同实例
- 采用mknod命令创建指定类型的设备文件，分配主、从设备号

```
[root@xsbase root]# mknod  /dev/lp0  c  6  0
```

- 主设备号分配规则见：Documentation/Devices.txt
- 已安装设备驱动程序的主设备号，可从/proc/devices中获取

mknod

```
mknod [-F format] name [c | b] major minor
```

```
mknod [-F format] name [c | b] major unit subunit
```

```
mknod name [c | b] number
```

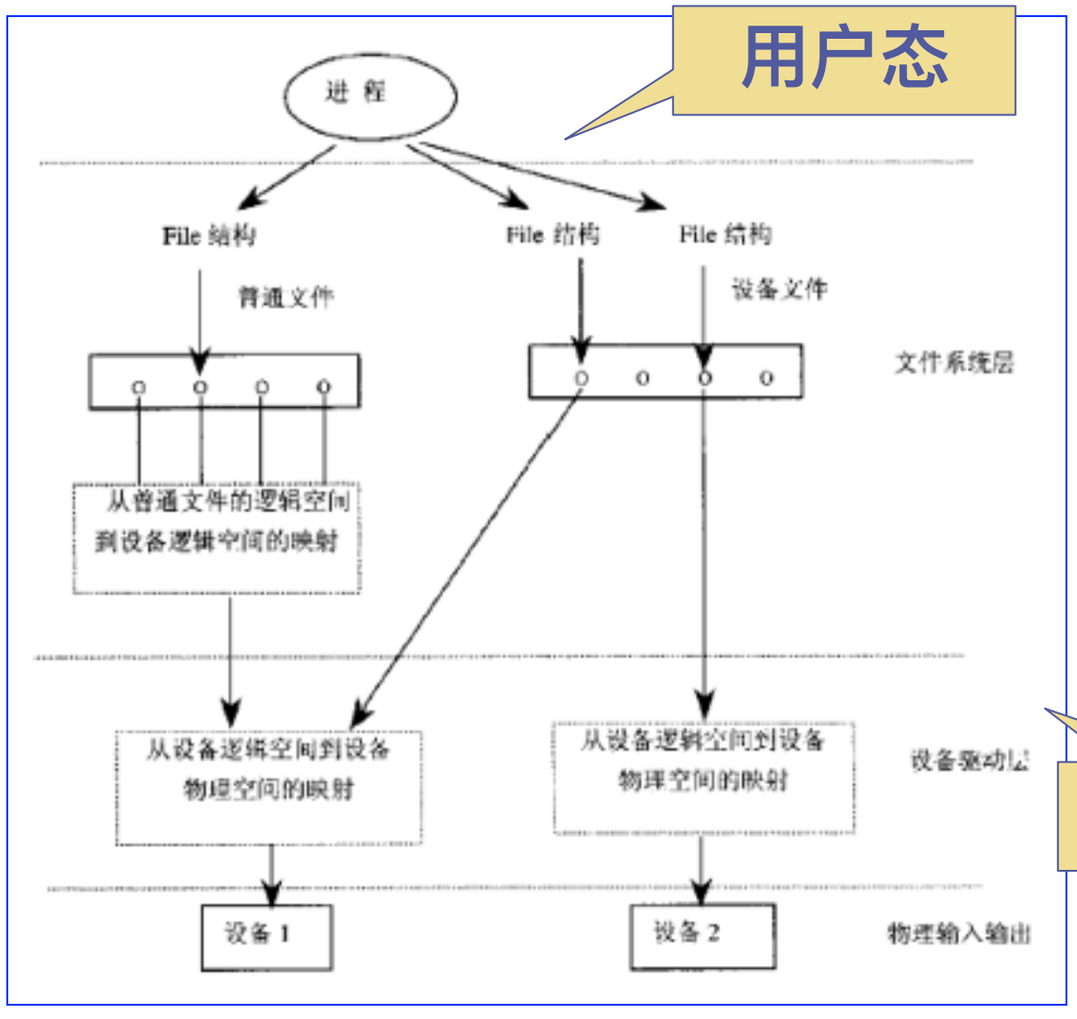
设备文件系统

- devfs从Linux内核的测试版本2.3.46开始引入
- devfs类似于/proc，是一种虚拟文件系统，内存空间占用小（devfs：72b / inode：256b）
- devfs是一种基于内核的动态设备文件系统，设备文件不需要手工采用mknod命令创建，在驱动程序安装时自动完成

sysfs

- 设备文件系统（devfs）存在的问题
    - 设备管理受主、从设备号（<=255）数量限制
    - 不确定的设备映射问题
    - 修改设备文件名字困难
    - 内核内存消耗多
- sysfs管理模式：
    - sysfs是linux2.6内核引入的文件系统，是一种虚拟文件系统
    - 对设备和总线进行管理
    - 支持设备数量更多
    - 支持设备热插拔
    - 运行在用户空间

Linux设备操作执行过程



- 设备操作功能调用
- 当应用程序对某个设备文件进行系统调用时，将从用户态进入到内核态；
- 内核根据该设备文件的设备类型和主设备号查询相应的设备驱动程序
- 由驱动程序判断该设备的次设备号，最终完成对相应设备的操作。

Linux设备驱动安装模式

- 将设备驱动以静态编译方式加入Linux内核。
- 将设备驱动模块，以动态加载方式加入内核。

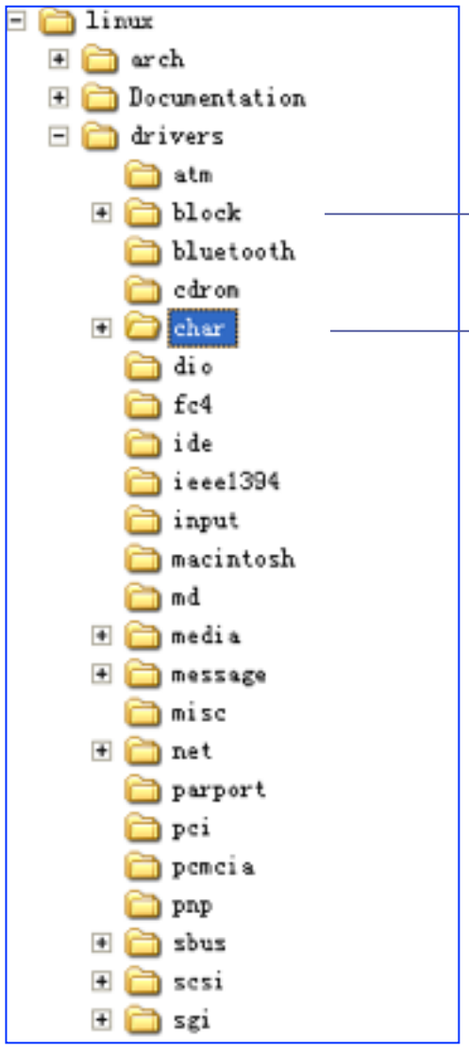
Linux设备驱动特点

- 内核编程模式
  - 设备驱动是内核的一部分，必须使用标准的内核编程模式进行设计，如：内核内存分配、中断处理和等待队列。
- 内核、用户接口
  - 设备驱动必须为内核或者其从属子系统提供标准接口；
  - 设备驱动与应用程序交互接口，需完成用户空间/内核空间转换。
- 动态可加载
  - 设备驱动可以在需要的时候加载到内核，不再使用时被卸载，使内核能更有效地利用系统资源。

驱动程序开发三部曲

- mknod创建设备文件
- 编写处理文件各操作的函数
- 构造file_operation结构，注册到对应的设备文件

Linux设备驱动代码分布



块设备驱动公用部分

字符设备驱动公用部分

各种设备驱动源程序

设备驱动与外界接口

- Linux的设备驱动程序与外界的接口，主要可以分成三个部分：
  - 与操作系统内核的接口：通过include/linux/fs.h中的file_operations数据结构完成。
  - 与系统引导的接口：完成对设备的初始化
    - 字符设备初始化在drivers/char/mem.c中的chr_dev_init()函数
    - 块设备初始化在drivers/block/ll_rw_blk.c中的blk_dev_init()函数
  - 与设备的接口：完成对设备的交互操作功能。

Linux设备驱动主要功能

- Linux设备驱动程序代码结构大致可分为如下几个部分：
  - 驱动程序的注册与注销
  - 设备的打开与释放
  - 设备的读写操作
  - 设备的控制操作
  - 设备的中断和轮询处理

驱动程序的注册、注销

- 字符设备注册
  - int register_chrdev(unsigned int major, const char * name, struct file_operations *fops)
  - major：主设备号，＝0，系统自动分配主设备号
  - name： 设备名称
  - fops ： 设备操作接口
- 字符设备注销
  - unregister_chrdev( )

返回主设备号

```
/* Set up character device for user mode clients. */
i = register_chrdev(0, "pcmcia", &ds_fops);
if (i == -EBUSY)
    printk(KERN_NOTICE "unable to find a free device # for "
           "Driver Services\n");
else
    major_dev = i;
```

```
static struct file_operations ds_fops = {
    owner:      THIS_MODULE,
    open:       ds_open,
    release:    ds_release,
    ioctl:      ds_ioctl,
    read:       ds_read,
    write:      ds_write,
    poll:       ds_poll,
};
```

驱动程序的操作接口

- 设备驱动程序的操作接口与文件系统的接口一致，在file和inode中均有file_operations数据结构。
  - include/linux/fs.h

```
struct file {
        struct list_head        f_list;
        struct dentry           *f_dentry;
        struct vfsmount         *f_vfsmnt;
        struct file_operations  *f_op;
        atomic_t                f_count;
        unsigned int            f_flags;
        mode_t                  f_mode;
        loff_t                  f_pos;
        unsigned long           f_reada, f_ramax, f_raend, f_ralen, f_rawin;
        struct fown_struct      f_owner;
        unsigned int            f_uid, f_gid;
        int                     f_error;

        unsigned long           f_version;

        /* needed for tty driver, and maybe others */
        void                    *private_data;

        /* preallocated helper kiobuf to speedup O_DIRECT */
        struct kiobuf           *f_iobuf;
        long                    f_iobuf_lock;
} ?  end file ?  ;
```

对打开文件的处理函数

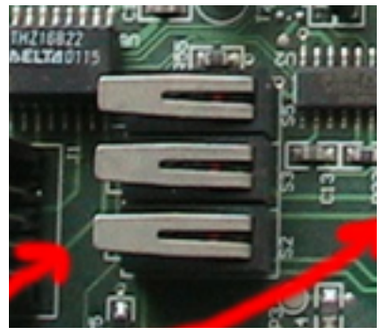
```
struct inode {
        struct list_head        i_hash;
        struct list_head        i_list;
        struct list_head        i_dentry;

        struct list_head        i_dirty_buffers;
        struct list_head        i_dirty_data_buffers;

        unsigned long           i_ino;
        atomic_t                i_count;
        kdev_t                  i_dev;
        umode_t                 i_mode;
        nlink_t                 i_nlink;
        uid_t                   i_uid;
        gid_t                   i_gid;
        kdev_t                  i_rdev;
        loff_t                  i_size;
        time_t                  i_atime;
        time_t                  i_mtime;
        time_t                  i_ctime;
        unsigned long           i_blksize;
        unsigned long           i_blocks;
        unsigned long           i_version;
        struct semaphore        i_sem;
        struct semaphore        i_zombie;
        struct inode_operations *i_op;
        struct file_operations  *i_fop;
        struct super_block      *i_sb;
        wait_queue_head_t       i_wait;
        struct file_lock        *i_flock;
        struct address_space    *i_mapping;
        struct address_space    i_data;
        struct dquot            *i_dquot[MAXQUOTAS];
        /* These three should probably be a union */
        struct list_head        i_devices;
        struct pipe_inode_info  *i_pipe;
        struct block_device     *i_bdev;
        struct char_device      *i_cdev;
```

file_operation

- linux-2.6.22/include/linux/fs.h

```
struct file_operations {
        struct module *owner;
        loff_t        (*llseek) ();
        ssize_t       (*read) ();
        ssize_t       (*write) ();
        ssize_t       (*aio_read) ();
        ssize_t       (*aio_write) ();
        int           (*readdir) ();
        unsigned int (*poll) ();
        int           (*ioctl) ();
        long          (*unlocked_ioctl) ();
        long          (*compat_ioctl) ();
        int           (*mmap) ();
        int           (*open) ();
        int           (*flush) ();
        int           (*release) ();
        int           (*fsync) ();
        int           (*aio_fsync) ();
        int           (*fasync) ();
        int           (*lock) ();
        ssize_t       (*sendfile) ();
        ssize_t       (*sendpage) ();
        unsigned long (*get_unmapped_area) ();
        int           (*check_flags) ();
        int           (*dir_notify) ();
        int           (*flock) ();
        ssize_t       (*splice_write) ();
        ssize_t       (*splice_read) ();
};
```

sample: hook



内核编程模式

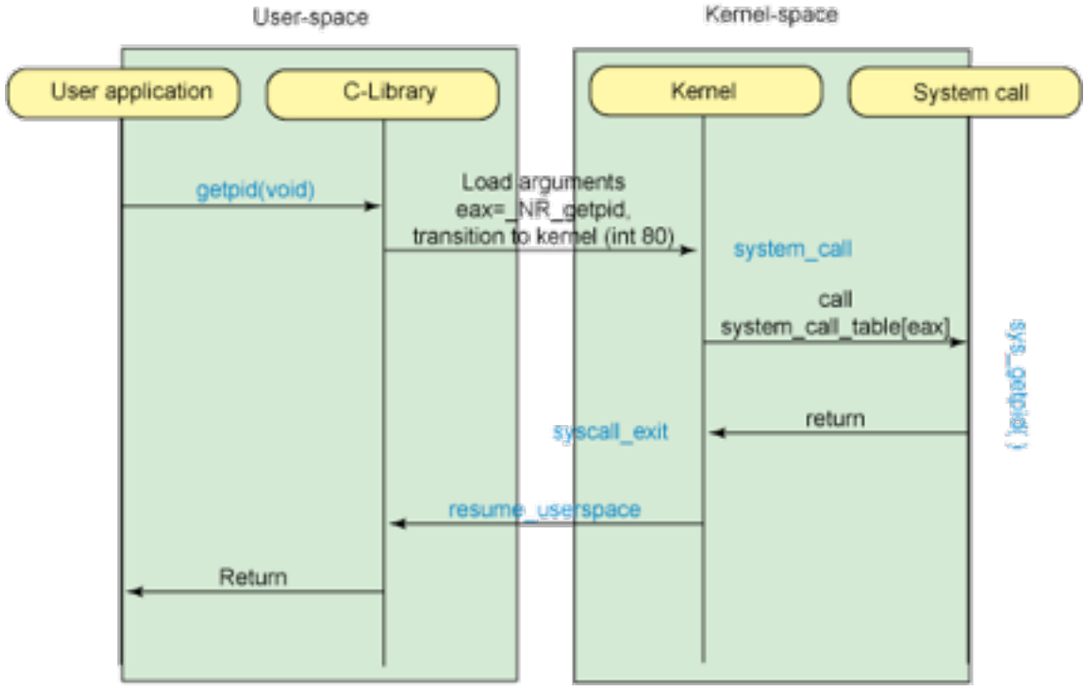
- 可以使用系统调用，但不能使用libc库，如printf。
- 内存申请采用kmalloc
    - 内核内存管理需求：对象分配/回收频繁；需要较小内存碎片
    - slab算法思想：
        - 预定义对象大小
        - 按2的幂次方进行内存大小划分
        - 对象缓存
        - 对象复用

内核编程模式

- 内核编程中的并发处理：
  - 多任务并发调用一个设备驱动程序
  - 一个功能部分可能运行于多个上下文中（可重入）
  - 一个设备驱动程序运行于多处理器上
  - 设备驱动执行可被其他ISR中断

内核空间 vs 用户空间

- 驱动程序运行于内核态，用户程序运行于用户态，如何将用户空间的数据传递入内核空间?



内核空间 vs 用户空间

- 内核空间、用户空间数据传递方式
  - 寄存器方式传递方式
    - 寄存器传参数地址，调用内容放内存参数表中，可用于传递大量数据结构
  - 堆栈传递方式

内核空间 vs 用户空间

- Linux提供的内核空间、用户空间数据传递函数（include/asm/uaccess.h）
  - access_ok(type, address, size)：用户空间指针有效性判断
  - get_user(var, ptr)：移动int/long数据至内核空间
  - put_user(var, ptr)：移动int/long数据至用户空间
  - strncpy_from_user(char *dst, const char __user *src, long count)：将字符串从用户空间移动到内核空间中
  - copy_from_user(void *to, const void __user *from, unsigned long n)：由用户空间拷贝一块数据
  - copy_to_user(void *to, const void __user *from, unsigned long n)：向用户空间复制一块数据

模块动态加载机制

- Linux中的可加载模块(Module)是对内核功能的扩展
  - insmod
  - rmmod
- Linux模块载入内核后，它就成为内核代码的一部分
- 若某个模块空闲，用户便可将它卸载出内核

模块动态加载机制

- 与模块相关的命令有：
  - lsmod：列出内核中已经安装的模块
  - insmod：安装模块到内核中
  - rmmod：将模块从内核中卸载
  - depmod：分析可载入模块的依赖关系