### [文档末尾](#_末尾)

### C++中内存分配的类型

### 在C++中，内存分成5个区，他们分别是堆、栈、自由存储区、全局/静态存储区和常量存储区。 栈，就是那些由编译器在需要的时候分配，在不需要的时候自动清楚的变量的存储区。里面的变量通常是局部变量、函数参数等。 堆，就是那些由new分配的内存块，他们的释放编译器不去管，由我们的应用程序去控制，一般一个new就要对应一个delete。如果程序员没有释放掉，那么在程序结束后，操作系统会自动回收。 自由存储区，就是那些由malloc等分配的内存块，他和堆是十分相似的，不过它是用free来结束自己的生命的。 全局/静态存储区，全局变量和静态变量被分配到同一块内存中，在以前的C语言中，全局变量又分为初始化的和未初始化的，在C++里面没有这个区分了，他们共同占用同一块内存区。 常量存储区，这是一块比较特殊的存储区，他们里面存放的是常量，不允许修改（当然，你要通过非正当手段也可以修改，而且方法很多，在《const的思考》一文中，我给出了6种方法）

-------------------------------------------------------------------

### 程序跑飞故障

检查中断引脚！有可能是接收到意外的中断。而中断服务子程序有没有相应的返回。

### ucOSII中Tick Timer Interrupt enable的问题

Tick timer 不能在OSStart()之前运行，因为在OSStart()运行前，不能确定应该执行的task，如果这时候开始tick timer，则有可能进入一个未知的状态；一般来讲Tick timer interrupt enable在OSStart()之后的第一个(优先级最高??)task前面几行进行。

### ARM读写的时序问题。

ARM的MCK为60MHz，FPGA的时钟为50MHz。根据MCK读周期的时序需要修改ARM\_Interface的状态机。Arm\_Interafce中设置的状态机时钟为200MHz，当采样到nrd信号时，需要加入一些多余状态，等待NRD恢复到无效状态。(需要确定ARM端NRD信号的保持长度，来确保状态机不会在一个NRD周期触发两次动作), 当前状态加入6个sys\_clk延迟周期。

-------------------------------------------------------------

控制逻辑模块’logic\_control’，需要对控制字的解释。

----------------------------------------------------

读信号RD的时序需要确定。RD低电平的持续时间？需要修改Arm\_interface里面状态机的等待时间。一般来说状态机的等待时间应该稍大与RD有效时间，用来避免一次RD触发两次读动作。

---------------------------------------------------------------

Both the ‘rdclk’ and ‘wrclk’ shall be both connected to sys\_clk, which is 200MHz, in order to increase the speed of reading and writing. But now the ‘rdclk’ is 50MHz, connecting with ‘main\_clk’. So the fifo\_data comes out with 1 main\_clk cycle’s latency than the RD signal.

After timing simulation of FPGA in quartus3, the shortest time of rd\_en, which is generated by ‘Arm\_interface.vhd’, is 10 ns. Whereas the ‘rdreq’ for fifo working in the condition of 200MHz needs to be 5ns at least. So the frequency of ‘rdclk’ must less than 100MHz. Here, we got 50MHz.

----------------------------------------------------------------------

### About the command of arm9 UBOOT

1．md 30000000 9

以30000000为地址读9个长字（每个长字是4个字节）

2．Mm.w 30000000

从30000000开始以word为单位修改存储器

3．md 30000000 9

30000000:000100b0 000300b0 000500b0 000700b0

30000010:00b000b0

30000020:00b000b0

md 30000000 1

000100b0

md 30000004 1

000300b0

4．分析：

arm端地址与FPGA地址的对应关系：

|  |  |
| --- | --- |
| Arm端(hex) | FPGA端A(bit7 downto bit 0) |
| 3000 0000 | 00000001 0 |
| 3000 0004 | 00000011 0 |
| 3000 0008 | 00000101 0 |
| 3000 000C | 00000111 0 |
| 3000 0010 | 00001001 0 |
| 3000 0014 | 00001011 0 |
| 3000 0018 | 00001101 0 |
| 3000 001C | 00001111 0 |

----------------------------------------------------------------------

### 关于arm地址与FPGA地址的对应关系的试验

1. 数据D超过3bit，出错。Md 30000000 9读出的都是00b0
2. 写信号地址与fgpa对应关系

|  |  |
| --- | --- |
| Arm端(hex) | FPGA端A(bit7 downto bit 0) |
| 3000 0000 | 00000 0001 |
| 3000 0002 | 00000 0011 |
| 3000 0004 | 00000 0101 |
|  |  |
|  |  |

----------------------------------------------------------------------

### ARM的地址数据线

ARM的地址和数据线必须都有相应的fpga管脚与其对应，不能悬空。即使A0不用，也应该定义个pin\_a0将其输入。否则arm不能boot。

----------------------------------------------------------------------

### SRAM2GPIO总结：

Arm的9根地址线中的A0恒为值1，为无效值。也就是说A8~A1为有效地址线。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Name | A8~A1(fpga) | A8~A0 | FPGA地址 | ARM 地址 | command |
| C\_DMA\_CH3\_ADDR\_H | 01100000 | 01100000 1 | 0x60 | 0xC0 | Mw.w 300000C0 0003 |
| C\_DMA\_CH3\_ADDR\_L | 01100001 | 01100001 1 | 0x61 | 0xC2 | Mw.w 300000c2 0100 |
| C\_DMA\_CH3\_LEN\_H | 01100010 | 01100010 1 | 0x63 | 0xC4 | Mw.w 300000c4 0003 |
| C\_DMA\_CH3\_LEN\_L | 01100011 | 01100011 1 | 0x64 | 0xC6 | Mw.w 300000c6 ffff |
| C\_DMA\_CH3\_START | 01100100 | 01100100 1 | 0x65 | 0xC8 | Nw.w 300000c8 0055 |

----------------------------------------------------------------------

### DMA\_TRANSCEIVER调试记录

1. 写sram2gpio的dma控制字，gpio\_clk有正常输出，周期为48ns。由160Msys\_clk经过8分频得到。
2. 将GPIO的dma长度设置为F，则只有15个gpio\_clk脉冲。结果基本正确。
3. 测试Arm to SramA，由于mw.w 30000180 1111 200 命令自动将地址累加1，所以修改fpga程序，不判断是否为channel0。直接写入sram，寄存器输入基本正常，dma输送指定长度的数据后，可以正常结束。

----------------------------------------------------------------------

2005-1-28

### Uboot下调试dma\_transceiver中sram\_Interface模块的命令

Mw.w 30000060 0003

Mw.w 30000062 0100

Mw.w 30000064 0000

Mw.w 30000066 000f

Mw.w 30000068 0055

Mw.w 30000180 5555 300

调试前需要修改sram\_interface模块中的ch\_select<='1' WHEN A=DMA\_CHANNEL\_ADDR ELSE '0';语句为ch\_select<='1';。因为‘Mw.w 30000180 5555 300’命令自动将地址加1。逻辑分析仪上面应该显示写入15个数据。问题：好像如果需要重新写dma，必须重新写上面的7条指令，而不能只重写最后两条。

----------------------------------------------------------------------

### Uboot下调试dma\_transceiver中的sram\_interface模块和sram2gpio模块

命令：

1．写sram\_A模块

Mw.w 30000060 0003

Mw.w 30000062 0100

Mw.w 30000064 0000

Mw.w 30000066 000f

Mw.w 30000068 0055

Mw.w 30000180 5555 300

需要注意修改sram\_interface中关于的ch\_select语句。

2．写sram2gpio

mw.w 300000c0 0003

mw.w 300000c2 0100

Mw.w 300000c4 0000

Mw.w 300000c6 000f

Mw.w 300000c8 0055

观察逻辑分析仪的值，显示15个gpio\_clk，值均为0x55.

----------------------------------------------------------------------

### Problems and solution in debugging the ARM9200

Every pin of ARM9200 that connect with the FPGA, which including the address bus (A), the data bus (D), the control signals (NWR0, NRD, NCS2) and others (pc6, pb28, pb11) shall be defined in FPGA entity.

PC6 is set to high,

-------------------------------------------------------------------

Delay-locked Loop (DLL) vs. Phase-Locked Loop (PLL)

Both can be used to reduce the on-chip clock-distribution delay to zero. And both can be considered a servo-control system since they use a feedback.

A PLL uses a phase detector to drive a voltage-controlled oscillator (VCO) such that the VCO output has the desired frequency and phase. A PLL can recover a stable clock from a noisy environment, but it is difficult to avoid generating random clock jitter.

A DLL uses a phase detector to increase the delay so much that the subsequent clock edge occurs at the desired moment.

Neither a PLL nor a DLL can be used in PCI-designs that demand proper operation at instantly changing clock rates.

### ADS集成开发环境介绍

ARM ADS=ARM Developer Suite。由命令行开发工具、ARM实时库、GUI开发环境(Code Warrior 和AXD)、实用程序和支持软件组成。

命令行开发工具（具有将源代码编译、链接成可执行代码的功能）。

-------------------------------------------------------------------

### RealView Debug安装

**修改mac地址为00-09-6b-42-4e-49。**

1. 首先找到关键词：“设备管理”－“网卡”－“Realtek RTL8029(AS)-based PCI Ethernet Adapter”－“属性”－“高级”－“属性”里面已经有的属性作为关键词，比如我的是“Line speed”，那么我就将Line speed作为关键词（注意大小写和空格）；在注册表里面查找Line speed，一直找到HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\Control\Class下面的键，比如我的是：HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\Control\Class\{4D36E972-E325-11CE-BFC1-08002BE10318}\0000\Ndi\params\MediaMode下面的ParamDesc有值Line Speed
2. 在上面找到的地方，params下新建一个“项”，名称叫“NetworkAddress”；
3. 在它下面建立这些字符串（关键！）：
4. optional = 1
5. limittext = 12
6. ParamDesc = NetworkAddress
7. type = 1
8. Okey，现在打开“设备管理”－“网卡”里面的“属性”－“高级”，点选“NetworkAddress”属性，在“值”里面直接输入网卡的mac地址，按照下面的格式：00404510EA38确定之后就可以了。

**开始－>程序->ARM->License Wizard v3.0。点击获取lisence。点next，然后在弹出的网页里面填入user和pw。然后点击view lisence。在弹出的页面里面选择save lisence.**

-------------------------------------------------------------------

### Arm板在线仿真问题及解决方案

软件所需要的mac地址为：00-09-6B-42-4E-49 00096B424E49

### 几种芯片的型号和功能

HY57V641620HG T-H 8MB, 64Mbits Synchronous DRAM 4bank\*1M\*16bits, LVTTL, 133MHz 一般来说，最快 存取速度为125M～166M。但是能够达到80M就不错了。

HY57V641620HG(L) T-5 166MHz SDRAM

IS61LV51216 512K\*16bit Asyn SRAM, access time 10~12 ns.

AM29LV320MT/B 32Mbits (2M×16bit/ 4M×8bit)，FLASH RAM，3.0V Boot Sector Flash Memory, AMD

一般读写速度为10M

M58LW064D 64Mbit(8Mb×8，4Mb×16) FLASH MEMORY, 速度在10M左右。

AT29LV1024 1M bits FLASH ROM

ICS8442 700MHz Differential Clock

ICS525 PLL

PDIUSBD12 USB接口芯片。可与89c51等接口

CH375,CH372 USB接口芯片。容易上手。

SLB811 带控制器得USB接口芯片

CP2101 USB转UART。据说很好用。

C8051F005 工控采样，8ADC，单片机

LT1761-3.3 3.3v regulator

RN5RL33AA 3.3V regulator

TPS73133 3.3v REGULATOR

ISP1761 Philips USB2.0 芯片

CY7C68013 CYPRESS USB2.0 芯片

ISP1520 USB HUB

### 基于PDIUSBD12，单片机软件设计主要注意以下几点：

* PDIUSBD12的中断输出引脚只要中断寄存器不为0就保持低电平，所以单片机的对应中断（INT0)应设置成电平触发；中断处理后要用读上次传输状态寄存器清除中断寄存器中对应位(D0－D5）。
* PDIUSBD12靠软件控制USB端口的连接，程序在系统初始化处理完后软件设置连接到USB端口，然后开中断。
* PDIUSBD12对内部寄存器的读写没有边界限制，程序设计中一定不要读写超过端点深度的数据。特别对于描述符请求，由于其长度大于Control IN 深度（16 Bytes），要分几个数据周期传输。
* 描述符一定要设置正确，并且注意USB协议中所有字数据均定义为低字节在前传输（LSB），例如Phlips的ID为471H，应在iDVendor中定义成71H、04H。
* 在接收到Setup包后，一定要用ACK Setup指令来重新使能Control IN 和Control OUT端点。向IN端点写数据后，要用Validate Buffer指令使数据可以在下一个IN数据周期发送。从OUT端点读数据后，要用Clear Buffer指令来清空缓冲区，否则后面OUT周期传输的数据将被丢弃（返回NAK）。
* 协议的处理一定要按USB规范要求进行，对无效请求，用Set Endpoint Status指令将Control IN和Control OUT端点Stall即可。

-------------------------------------------------------------------

### PDIUSBD12丢包问题

说句实在话，ZLG团队做的软件包真的非常好，非常专业！基本上不用怎么修改就可以直接使用。不过在使用过程中，还是有些地方需要我们去注意。以下是我们公司在使用过程中出现的情况，请ZLG团队及D12的用户们多多交流！

1． 当MCU通过D12发数据给主机时不能简简单单的调用写函数，如下：D12\_WriteEndpoint(5,64,filebuffer);

如果您用的是51等慢速的单片机可能没什么事情，当用ARM等快速的处理器作大量数据传输时可能会出现丢包的现象。在MCU连续的给主机发包的过程中，主机还没有将上一个包的数据从D12读走（就是D12的缓冲区处于满的情况），MCU又将另一个包写进去时会覆盖掉以前的。因此在每写入一个包时必须先判断D12有没有空的缓冲区，如下：

while (1) //如果端点的两个缓冲区都为满,则等待

{

outportb(D12\_COMMAND,0x85);

EndpState = inportb(D12\_DATA);

EndpState &= 0x60;

if (EndpState != 0x60) //有空缓冲区,则发送数据

{

break;

}

}

D12\_WriteEndpoint(5,64,filebuffer);

2． 同样，在主机给MCU发送大量数据时也可能会产生丢包的情况。起初我们的MCU是直接查询bEPPflags.bits.ep2\_rxdone来判断D12是否收到主机数据，然后读取。下面的代码在主程序中执行，如下：

if (bEPPflags.bits.ep2\_rxdone == 1) //收到主机下载的数据

{

for(i=0;i<64;i++) //从缓冲区中读走一包数据

{

CheckSum +=EpBuf[i]; //累加校验和

\*pUsbByte= EpBuf[i]; //保存数据

pUsbByte ++;

}

RecDataLong += 64; //累加收到的数据长度

ARMDisableInt(); //关中断，我觉得没有必要，因为都没有判断该标志

bEPPflags.bits.ep2\_rxdone = 0; //清空端点2收到数据标志

ARMEnableInt();

}

有一种很奇怪的现象：当PC机打开的应用程序越多且频繁的操作这些程序时，丢包的现象越严重！所以当我要从PC机发送一个几十MByte的文件到MCU时，就不能让PC机做其它事情了，否则就出现丢包。起初我们在PC机软件做了很多努力，但都失败。后来发现不是D12里的缓冲区被覆盖了，而是软件包中的EpBuf缓冲区被覆盖。因为D12接收到主机发来的一个包后会产生中断，然后将数据放入EpBuf，而此时如果我们还没有从EpBuf读走数据时，D12又从主机收到一个包产生中断，此时读取包数据时就会将EpBuf原来的数据覆盖掉，因为端点接收数据处理函数便没有去判断bEPPflags.bits.ep2\_rxdone是否等于0，如下：

void ep2\_rxdone(void)

{

INT8U len;

D12\_ReadLastTransactionStatus(4); //清中断标志位

len = D12\_ReadEndpoint(4,EP2\_PACKET\_SIZE,EpBuf); //读取缓冲区中的数据

if (len != 0)

bEPPflags.bits.ep2\_rxdone = 1;

}

此时由于已经调用了D12\_ReadEndpoint(4,EP2\_PACKET\_SIZE,EpBuf)函数，而在该函数中写了清缓冲区的命令，所以主机就会继续往D12发送数据，而使EpBuf内容被覆盖。我觉得最简单最保险的方法就是从D12中读出的数据直接存入自己的缓冲区中，就可以保证在给D12写清缓冲区命令前数据都被存放好了，而且也提高了响应速度！修改ep2\_rxdone函数如下：

extern unsigned char \*pUsbByte; //指向自己定义的缓冲区

void ep2\_rxdone(void)

{

INT8U len;

D12\_ReadLastTransactionStatus(4); //清中断标志位

if (pUsbByte && to\_pctrans.trans\_type)

{

len = D12\_ReadEndpoint(4,EP2\_PACKET\_SIZE,pUsbByte);

pUsbByte += len;

RecDataLong += len; //累加实际收到的数据长度

}

if (len != 0)

{

bEPPflags.bits.ep2\_rxdone = 1;

USBTestBack += 1;

}

}

为什么PC机运行的程序越多且在频繁的操作这些程序时，丢包的情况越严重呢？我是这样理解：多任务操作系统(windows也不例外)的调度是由定时中断触发的，所以两次调度的时间间隔是个定值。在第一次调度时，由于很多优先级更高的任务将USB发送任务推迟了。而在第二次调度时，第一次优先级高的任务又被挂起，而使第二次USB发送任务没有被推迟。在最坏的情况下两次USB发送的时间间隔远比正常情况下的时间间隔短，就使MCU出现了EpBuf包被上一次覆盖的情况。

-------------------------------------------------------------------

### The Host's Duties

The host PC is in charge of the bus. The host has to know what devices are on the bus and the capabilities of each. The host must also do its best to ensure that all devices on the bus can send and receive data as needed. A bus may have many devices, each with different requirements, and all wanting to transfer data at the same time.

### 什么是PCMCIA卡

PERSONAL COMPUTER MEMORY CARD INTERNATIONAL ASSOCIATION（PCMCIA）是一个电脑制造商的公会。它们制定了使用在笔记型电脑上信用卡状的通用转接卡的型式。

PCMCIA定义了三种不同型式的卡，它们的长宽都是85.6×54mm，只是在厚度方面有所不同。Type I是最早的PC卡，厚3.3mm主要用于RAM和ROM；Type II将厚度增至5.5mm适用范围也大大扩展包括了大多数的modem和faxmodem，LAN适配器和其它电气设备；Type III则进一步增大厚度到10.5mm这种PC卡主要用于旋转式的存储设备（例如硬盘）。而按照其位宽分可以为16位和32位，而应用了32位新技术通常叫CardBus它主要是用于要求高带宽的传输界面（例如USB2.0、IEEE1394、IDE等）。

高版本的PCMCIA卡接口一般都可以很好地兼容低版本的卡。也就是说一个Type I插槽只可以插入一张Type I 卡；一个Type II插槽可以插入一张Type II卡或一张Type I的卡；一个TypeIII的插槽则除了可以插入一张Type III卡外还可以插入两张兼容的Type I或Type II卡。（第三媒体整理 2003－10－21）

### SDRAM简介

SDRAM器件的管脚分为控制信号、地址和数据三类。通常一个SDRAM中包含几个BANK，每个BANK的存储单元是按行和列寻址的。由于这种特殊的存储结构，SDRAM有以下几个工作特性。

● SDRAM的初始化

SDRAM在上电100～200μs后，必须由一个初始化进程来配置SDRAM的模式寄存器，模式寄存器的值决定着SDRAM的工作模式。

● 访问存储单元

为减少I/O引脚数量，SDRAM复用地址线，所以在读写SDRAM时，先由ACTIVE命令激活要读写的BANK， 并锁存行地址，然后在读写指令有效时锁存列地址。一旦BANK被激活后只有执行一次预充命令后才能再次激活同一BANK。

● 刷新和预充

--- SDRAM的存储单元可以理解为一个电容，总是倾向于放电，因此必须有定时的刷新周期以避免数据丢失。刷新周期可由(最小刷新周期÷时钟周期)计算获得。对BANK预充电或者关闭已激活的BANK，可预充特定BANK也可同时作用于所有BANK，A10、BA0和BA1用于选择BANK。

● 操作控制

--- SDRAM的具体控制命令由一些专用控制引脚和地址线辅助完成。CS、RAS、CAS和WR在时钟上升沿的状态决定具体操作动作，地址线和BANK选择控制线在部分操作动作中作为辅助参数输入。由于特殊的存储结构，SDRAM操作指令比较多，不像SRAM一样只有简单的读写。

-------------------------------------------------------------------

### 同步串口介绍

在路由器的广域网连接中，应用最多的端口还要算"高速同步串口"（SERIAL）了，这种端口主要是用于连接目前应用非常广泛的DDN、帧中继（Frame Relay）、X.25、PSTN（模拟电话线路）等网络连接模式。在企业网之间有时也通过DDN或X.25等广域网连接技术进行专线连接。这种同步端口一般要求速率非常高，因为一般来说通过这种端口所连接的网络的两端都要求实时同步。如图所示为高速同步串口。

-------------------------------------------------------------------

### What does ‘BSC’ stand for on component datasheet?

BASIC

-------------------------------------------------------------------

### 关于MSP430F13、F14系列晶振问题

哎，我不知道怎么回事，15P，30P；两个电容、一个电容、没有电容都试验了，5M电阻，2M电阻，接XIN、XOUT、并在晶体两端都试验了，晶体也换了。 长时间掉电后再上电，不工作的时候少，但也有十之一二（您知道的，百里挑一都不行啊），掉电1秒内再上电，十之八九不工作。

用示波器10MOhm探头测XOUT，不振是0V，有振在电源中间叠加高频，测XIN，不振是任意值，有振的振幅大于XOUT。

以前我用的时候，也是这个芯片，不用晶体用DCO就从来没有不工作的情况。可我现在要时钟很准呀。

精华区版主介绍如下：

为了确保振荡器操作稳定，MSP430x1xx 与MSP430x3xx 系列均需要ESR < 50kOhm的32kHz晶振。

可是我不知道ESR 是多少啊。而且我是4MHz的。

我抄的上电程序：

RESET mov.w #300h,SP ;Initialize stackpointer

SetupWDT mov.w #WDTPW+WDTHOLD,&WDTCTL ; Stop Watchdog Timer

SetupBC bis.b #XTS,&BCSCTL1 ; LFXT1 = HF XTAL

SetupOsc bic.b #OFIFG,&IFG1 ; Clear OSC fault flag

mov.w #0FFh,R15 ; R15 = Delay

SetupOsc1 dec.w R15 ; Additional delay to ensure start

jnz SetupOsc1 ;

bit.b #OFIFG,&IFG1 ; OSC fault flag set?

jnz SetupBC ; OSC Fault, clear flag again

我不懂为什么WDT要停止，就改到让它工作。没有好转。

又把最后的 jnz SetupOsc 改为： jnz SetupBC 没有好转。

用C-SPY调试，全速或单步，程序就在这几句里转，不往下面走。所以证明是晶体不起振。

呵呵。 :(

精华区版主 还 介绍如下：

MSP430F13,14系列没有BOR模块，有时会出现这种情况，可加一片R3111电压监测芯片，R3111可与利尔达联系。也许是复位问题？可我的低成本应用，真是不想加东西了。

负载电容（请参阅数据表中的具体说明）

注：有效负载电容

晶振制造商通常会在晶振的数据表中定义有效负载电容。从电子学角度来说，电容器以串行方式连接到引脚XIN 与XOUT上，这时有效负载电容为：

C(eff) = {C(XIN) ? C(XOUT)}/{C(XIN) + C(XOUT)}

因此，晶振的数据表中规定12pF的有效负载电容要求在每个引脚XIN 与 XOUT上具有22pF（2 \* 12pF = 24pF = 22pF + 2pF 寄生电容）。 MSP430x1xx 与 MSP430x3xx 系列为32kHz振荡器提供了约12pF的固定集成负载电容器，并且无需任何其它外部负载电容器即可支持需要6pF有效负载电容的晶振。高频率 XTAL 振荡器无内置负载电容器。 MSP430x4xx 系列为低频率与高频率模式下的LFXT1 振荡器提供了软件可选的集成负载电容器。该器件数据表中提供了可选值。XT2 振荡器没有任何内置负载电容器。

ESR

为了确保振荡器操作稳定，MSP430x1xx 与MSP430x3xx 系列均需要ESR < 50kOhm的32kHz晶振。MSP430x4xx 系列的低功耗振荡器需要 ESR < 100kOhm的 32kHz 晶振。 高频率晶振的建议 ESR 值是 <= 40Ohms（频率为8MHz时）。 与建议的最大值相比，ESR的值越低，振荡器启动性能与稳定性也越好。

设计考虑事项：

使晶振、外部电容器（如果有）与 MSP430 之间的信号线尽可能保持最短。当非常低的电流通过MSP430晶振振荡器时，如果线路太长，会使它对 EMC、ESD 与串扰产生非常敏感的影响。而且长线路还会给振荡器增加寄生电容。

如果MSP430在插座中：请注意插座会给振荡器增加寄生电容。

尽可能将其它时钟线路与频繁切换的信号线路布置在远离晶振连接的位置。

当心晶振和地的走线

将晶振外壳接地

当 VCC < 2.5 V 时，MSP430x1xx 的 LFXT1 振荡器要求在LF模式下使用从XOUT 到 VSS 的 5.1MOhm 电阻器。

-------------------------------------------------------------------

### MSP430上电复位问题

有时MSP430上电，或者复位时芯片不能正常工作，可能是因为RST和低，或者电源和地之间有大电容相连，造成电平变化缓慢。产生问题。

-------------------------------------------------------------------

### MSP430F149 JTAG 电缆调试问题

1. 其中JTAG的接口J1的pin2和pin4，我都接了3v3，准备和F149的板子接在一起，可否？

回答：这要看仿真器（FET）是什么型号？如果是隔离型的FET，JTAG的P4一定要接电源（VCC）；如果不是隔离FET，且目标板（用户板）有外部电源可以供电，那JTAG的P2、P4可以不接电源VCC，也不必和F149连接。

2. 我的TEST管脚悬空了，不知道接到F149的哪根管脚上，请各位指教

3. 我的XOUT管脚悬空了，是不是 应该接到F149的32.768K晶振上？如果不接，就不能使用？

TEST、XOUT都不必连接，因为，F149的JTAG口是独立的，没有和其他I/O口复用连接

-------------------------------------------------------------------

### MSP430F149调试记录

1. 按键开关弄错了，开关两条长边在内部是连在一起的。板子上面的弄反了。导致开关全部不能使用。补救方法是采用贴片的开关（转90度）。
2. JTAG下载线的R10管脚虚焊。
3. JTAG下载线的R5去掉！！！！！！！！！（这个原来的图 太坑人了！！！）
4. 将/RST管脚的二极管，电容都去掉了，将/rst直接用100K电阻上拉到3V3。（不知道是否对调通有影响）
5. 74HC244的电源没画。以后画这种多part的元件需要注意，因为它的电源和地是隐藏的。

-------------------------------------------------------------------

### MSP430上电复位问题

有时MSP430上电，或者复位时芯片不能正常工作，可能是因为RST和低，或者电源和地之间有大电容相连，造成电平变化缓慢。产生问题。

-------------------------------------------------------------------

### MSPF149中timerA1中断用法

用timerA1实现10MS一次中断与TIMA0没什么区别，但是注意CCIGF1要手复位！

TIMERA0如下：

void Timer0unit(void)

{

TACTL=TASSEL1+TACLR;

CCTL0=CCIE;

CCR0=8000;

TACTL|=MC0;

}

#pragma vector=TIMERA0\_VECTOR

\_\_interrupt void Timer\_A(void)

{

中断处理程序略；

}

-------------------------------------------------------------------

### IAR中MSP430 unsigned int类型问题

for (pwm\_Delay=0;pwm\_Delay<=0xfff0;pwm\_Delay++); 其中pwm\_Delay为unsigned int类型。当红色数值为0xFFFE时，可以正确跳出循环；当红色数值为0xFFFF时，进入死循环。在编程时，需要注意这一点。

-------------------------------------------------------------------

### MSP430F149的TIMER\_A和LCD调试报告

1. 电路板A和电路板B的timer\_A功能均正常
2. 选择内部timer\_A的语句如下： TACTL = TASSEL\_2 + MC\_1; // DCO SMCLK, up mode
3. 两块电路板的6M晶体均工作不正常。怀疑是最近湿度过大，影响晶体。下次争取选择晶振！而不要选择晶体。MSP430的内部起振电路有问题。
4. 另外，在测试HF 6M晶体时，程序将XTAL2选择位ACLK，而不是SMCLK。
5. LCD程序运行不成功。推测原因：没有加wait\_free()程序，造成写数据过快。新版本的程序在Write\_command()和Write\_data()之前都加入了Wait\_free()程序。
6. 内部DCO的频率为200K左右，周期为5us左右。

-------------------------------------------------------------------

### MSP430F149中程序中没有用到的变量

MSP430F149中程序中没有用到的变量需要声明成全局变量static unsigned int。否则，会被编译器去掉。

-------------------------------------------------------------------

### F149中的io口

F149的每个IO口都有四个寄存器：

1. pioIN
2. PioOUT
3. PioDIR
4. PioSEL

-------------------------------------------------------------------

### 北京青云LCD使用总结

1. /RST可以不接。
2. PSB（并行/串行选择）最好接。如果只用并行方式（8bit or 4bit）可以悬空。
3. 背光用5V加一个1N4148即可。
4. 上电初始化(lcm\_init())过程如下：



1. 汉字按照以下方式定义: const unsigned char hanzi[]=”你好”
2. 汉字输出:



-------------------------------------------------------------------

### MSP430捕获/比较定时器的作用

捕获比较模块用于捕获应用事件的发生时间，或产生定时间隔。如果相应的中断允许，那么完成一个时间捕获或一次定时间隔，捕获/比较模块都将产生中断。每一个捕获比较模块都可以对应一组硬件引脚。图1是捕获比较模块的结构框图。

捕获功能可以捕捉选定输入引脚的状态的变化，它可以选择捕捉上升沿、下降沿、前后沿。如果捕捉到了相应的变化，则定时器计数值将被复制到捕获比较寄存器CCR中，并会产生相应的中断。在串行通信中，正是利用捕获功能的特点来捕捉起始位的信息。

图1：捕获比较模块结构框图

比较功能是借助比较器不断地将CCR中的设定值与定时器中的计数值相比较，当二者相等时，就产生中断，并产生设定的输出。利用比较功能，可以获得精确的时间间隔，利用该特性可以构造一个精确的波特率发生器，为串行通信提供时间基准。

-------------------------------------------------------------------

### 什么是电磁兼容问题EMC

那么电磁兼容到底是一个什么东西呢？其实很简单，不知是否你留意这样一个现象，当您拨打手机时，信号会突然变得嘈杂不清；当您打开电视机时，会无意中收到邻居家播放的游戏机信号；当您爱作电脑时，会因为一次小小的静电事故造成死机，这些就是EMC现象。它的标准定义是设备或系统在其电磁环境中能正常工作且不对该环境中任何事物构成不能承受的电磁骚扰的能力。抛开这繁复的定义，对一般人来说，电磁兼容（EMC）就是干扰（EMI）与抗干扰（EMS）一对矛盾而又孪生的兄弟。

-------------------------------------------------------------------

### 示波器不能显示频率和周期

示波器不能显示测量的频率和周期

measure->参考电平->选择%，不要选择‘单元’即可。

### 设置SouceInSight新建项目的默认路径方法

In regedit, find the key: HKEY\_CURRENT\_USER\Software\Source Dynamics\Source Insight\3.0\Paths

Add a new string value named “UserDataDir”. Set the string value to the full path of the folder you want to user for per-user data.

-------------------------------------------------------------------

### FLASH的读取方法

* Nor的flash是可以随机存取的。
* Nand的flash只能一次擦除或写入16K，所以一般有两种方案：1. 用另外16K的RAM做缓存。2. 用flash的另外的一个block做缓存（速度比1要慢很多）。

-------------------------------------------------------------------

2005-10-31

### 5000pF的电容特性与封装

5000pF的电容如果要有极性的，只有独石或者是瓷片的。插针式封装，间距大概100mil。

-------------------------------------------------------------------

2005-11-3

### SourceInSight 配置方法

Option->Document Options:

Screen Fonts: Verdana 9

Printer Fonts: Verdana 9

Tab width:2

Margin Width:80

Choose:

Allow auto-complete, Allow Smart Paste, Enter key->new Line, Show line numbers

Auto indent…->Simple

### MSP430,IAR软件中断问题

Options/code/optimizitions/size这里选择什么了，low和none这两个选项换一下，可能就能用了

-------------------------------------------------------------------

2005-11-21

### MSP430存储大量数据

将大量数据定义成: const int a[100]={…}

### 用什么电池为单片机供电

用干电池或者一次性锂电池。

锂电池太差了,一般都有滞后的问题,很不可靠,而且受温度的影响很大,温度高时放电特快.一般能用一, 两年就谢天谢地了

### MIDDLEWARE定义

这个名词使用的情况很多。从根本上讲，中间软件是软件"胶水"，它使得在不同计算机上的程序和数据库可以在一起工作。GartnerGroup给中间软件下了一个更为正式的定义：在分布式的计算环境中能够使应用层的程序直接相互连通的实时系统软件。中间软件的最基本功能是使应用程序间或单一应用系统的或跨多个应用系统的DBMS能够相互交流。

-------------------------------------------------------------------

2005-12-21

### DVB-CI PC CARD 工作流程

配置PCMCIA 卡：

1. STPCCRD\_ModCheck(Handle);检测PC card是否插入插槽（检测CD1#和CD2#管脚是否为低）。如未插入，则等待并提示‘请插卡’；如插槽内有卡，进入第二步。
2. STPCM\_ResetCIchip();通过PIO对PC card进行reset。
3. STPCCRD\_CheckCIS(Handle);通过读取tuple来检测CIS，并根据CIS判断卡是否符合协议要求。
4. 写入配置寄存器。
5. 在COMMAND寄存器里面写入PCCRD\_SETRS（软reset）。

-------------------------------------------------------------------

### USB Massage Storage Initialization (USB 海量存储初始过程)

HOST->PC; DEVICE->U disk, MCU,etc…

1. USB Power On (USB Device Insert)🡪
2. Device Upload Device Descriptor🡪
3. Device Upload Configuration Descriptor🡪
4. Device Upload Interface Descriptor🡪
5. Device Upload Endpoint Descriptor(OFTEN Not need, use the default endpoint descriptor) 🡪
6. After all Descriptor received correctly, USB host identify the device as the mass storage device, and enter into BULK-ONLY mode. 🡪
7. HOST downloads CBW(Command Block Wrapper) to DEVICE, then DEVICE executes the CBW, then upload CSW( Command Status Wrapper) to HOST. According to the received CSW, HOST decide whether to send another CBW or to execute other commands.(其中CBW的格式遵从bInterfaceSubClass所指定的格式； 另外CBW包中的命令，需要遵从SCSI-2,12字节长度指令的格式). 🡪
8. HOST will download SCSI command (in CBW)：INQUIRY、Read Capacity、UFI Mode Sense… in sequence. 🡪
9. According to the SCSI command sent by HOST, the DEVICE will reply them (in CSW) one by one. 🡪
10. If step 8 and 9 are all done, the HOST will sent READ command in CBW to read MBR (0 cluster 0 sector). After all process is done, it enters the ‘File system recognition stage’.

-------------------------------------------------------------------

### FAT16 File system 文件系统

A FAT16 system consists of:

* Reserved region (保留分区): The first sector should be BPB(BIOS Parameter Block).
* FAT region: 操作系统分配磁盘空间按簇来分配的。因此，文件占用磁盘空间时，基本单位不是字节而是簇，即使某个文件只有一个字节，操作系统也会给他分配一个最小单元——即一个簇。为了可以将磁盘空间有序地分配给相应的文件，而读取文件的时候又可以从相应的地址读出文件，我们把整个磁盘空间分成32K 字节长的簇来管理，每个簇在FAT 表中占据着一个16位的位置，称为一个表项。对于大文件，需要分配多个簇。同一个文件的数据并不一定完整地存放在磁盘的一个连续的区域内，而往往会分成若干段，像一条链子一样存放。这种存储方式称为文件的链式存储。为了实现文件的链式存储，硬盘上必须准确地记录哪些簇已经被文件占用，还必须为每个已经占用的簇指明存储后继内容的下一个簇的簇号，对一个文件的最后一簇，则要指明本簇无后继簇。这些都是由FAT 表来保存的，FAT 表的对应表项中记录着它所代表的簇的有关信息：诸如是否空，是否是坏簇，是否已经是某个文件的尾簇等。
* Root Directory Region: record the information of folder.
* File Data Region: Record the file data

-------------------------------------------------------------------

### HARD Disk Architecture (硬盘结构)

1. MBR (Main Boot Record): 0 track, 0 column, 1 sector. Total 512 bytes.



1. DBR (Dos Boot Record): includes boot software and BPB (BIOS Parameters Block) 引导程序的主要任务是当MBR 将系统控制权交给它时，判断本分区根目录前两个文件是不是操作系统的引导文件（以DOS 为例，即是Io.sys 和Msdos.sys）。如果确定存在，就把其读入内存，并把控制权交给该文件。BPB 参数块记录着本分区的起始扇区、结束扇区、文件存储格式、硬盘介质描述符、根目录大小、FAT 个数，分配单元的大小等重要参数，如在FAT 结构中所介绍的那样。
2. FAT (File Allocation Table): It allocates spaces for file and data.
3. DIR (Directory): DIR（Directory）是根目录区，紧接着第二FAT 表（即备份的FAT 表）之后，记录着根目录下每个文件（目录）的起始单元，文件的属性等。定位文件位置时，操作系统根据DIR中的起始单元，结合FAT 表就可以知道文件在硬盘中的具体位置和大小了。
4. DATA: 数据区。

2005-12-23

### 泰克仪器培训

* 上升沿时间和测量带宽：上升沿时间和测量带宽满足以下公式：。上升沿时间公式满足，其中Ro是示波器上升时间，Rs是信号上升时间，Rp是探针上升时间。
* 示波器使用调试流程：DPO捕获🡪触发捕获🡪测量🡪分析🡪记录
* 探头分类：探头分为有源探头和无源探头。无源探头阻抗在1Mohm，测量频率为500MHz以下，测量电压400以下；有源探头阻抗50ohm，分为电源探头、差分探头、高频探头灯，测量电压<20V，一般在8V~10V左右。

-------------------------------------------------------------------

2006-1-6

### VxD,WDM和DDk三者有什么样的联系

vxd与wdm是windows的两种驱动模式,vxd是早期的驱动模式,wdm是新的驱动模式.ddk是widows 的Driver develope kit 还有类似于WinDriver,DriverStudio等可以做驱动,但核心的还是DDK啦,最终得用DDK提供的API和编译器吧! WDM是驱动的一种模式,DDK是工具,两者不混为一谈!DDK是用来开发WDM驱动程序的。

WinDriver,DriverStudio是在DDK之上提供了一种向导形式的开发界面来开发驱动程序的工具。

-------------------------------------------------------------------

### 如何调用wdm驱动程序

要看你的驱动程序用的符号链接还是使用GUID

各位老大，我有一个设备驱动的程序 可是我如何在我的应用程序里调用呢？

如果我用CreateFile("filename",,,,,,,

那么在filename处改使用什么？是驱动的名字，还是设备的名字，还是别的什么呢？

-------------------------------------------------------------------

### 如何学习开发USB2.0系统

1. Read USB Spec  
2. Buy a USB evaluation board  
3. Read WDM programing books  
4. Write your own driver for the evaluation board

-------------------------------------------------------------------

2006-01-13

### Static 的作用

1. 静态变量（函数体内）
2. 模块内（文件内）全局变量；模块外不能使用。（函数体外，模块内）
3. 声明函数时，该函数只能在模块内有效；模块外无效。（声明函数时）

-------------------------------------------------------------------

### Const的作用是什么？

Const意味着‘只读’。

-------------------------------------------------------------------

### Volatile作用

一个定义为volatile的变量是说这变量可能会被意想不到地改变，这样，编译器就不会去假设这个变量的值了。精确地说就是，优化器在用到这个变量时必须每次都小心地重新读取这个变量的值，而不是使用保存在寄存器里的备份。下面是volatile变量的几个例子：  
• 并行设备的硬件寄存器（如：状态寄存器）  
• 一个中断服务子程序中会访问到的非自动变量(Non-automatic variables)  
• 多线程应用中被几个任务共享的变量

-------------------------------------------------------------------

2005-01-16

### DriverStudio套件包的安装与运行环境设置

**硬件和软件需求**  
在安装该产品前最好将你的系统进行备分。下列是安装和运行DriverStudio 所需的要求：  
· PC-compatible Intel x86 系统  
· Microsoft Windows NT 4.0 (Service Pack 3 or later), Windows 2000, Whistler Build 2296. (Optional: 其中一些工具也支持 Windows 98 和 Windows 95.)  
· 内存: 最少32 MB, 推荐使用64 MB   
· 硬盘:完全安装需要 72 MB   
· 针对 SoftICE的远程调试: NE2000-compatible 网卡或 3Com 网卡  
· 针对 DriverWorks: Microsoft DDK,MS Visual C++  
DriverStudio 也支持其它的编译器。当前支持的编译器的详细列表参见<http://www.numega.com/drivercentral/components/compatibility.shtml>  
  
**安装DriverWorks**  
1. 在安装DriverWorks之前，首先要保证你的计算机上安装了Microsoft Visual C++以及相应针对Windows NT 或WDM的DDK。  
2. 在安装完成后，启动MSVC 5.0或6.0。启动MSVC最好的方法是使用SetDDKGo工具，位于开始|程序|Numega DriverStudio|Tools菜单中（Setup DDk and Start MSVC）。SetDDKGo执行一系列的命令行参数来定义BASEDIR环境变量。  
3. 在FILE|Open Workspace中打开DriverStudio\DriverWorks\Source\vdwlibs.dsw将VDWLIBS工程载入。  
4. 选择Build|Batch Build，去掉IA64相关的选项。  
5. 点击Build编译所选择的库文件。  
注意：库文件只需在安装完成后第一次使用前编译一次即可。以后要使用DriverWorks，只需通过SetDDKGo进入MSVC即可。

**安装VtoolsD**  
安装程序主要执行以下两个基本的功能：  
1． 将VtoolsD的组成文件复制到你指定的目录下。  
2． 创建配制文件USER.MAK，用来定义VtoolsD在系统中使用的符号。  
USER.MAK是用来定位VtoolsD在系统中生成VxDs时使用的工具的配置文件。它由一系列的符号定义列表组成，每一个都告知 NMAKE 各个特定的程序在系统中驻留的位置。SETUP提示你输入工具所需变量的路径。  
正确的USER.MAK是安装成功的关键。在安装完成后要检查该文件。  
一些在USER.MAK中定义的符号是文件，一些是用来控制生成过程的符号。下列表格说明了在USER.MAK中定义符号的含义 Symbol Meaning  
COMPILER 选择编译器的符号: MS5 或 BCB3  
DEBUGGER 选择调试器的符号: SOFTICE3, WDEB386, 或NONE  
C32 32-bit 命令行编译器的完全路径  
每个符号定义语句的格式为 SYMBOL = .  
例如:  
C32 = C:\BCC32\BIN\BCC32.EXE  
LINKMSVC2 MS可执行连接器的完全路径名  
TLINK32 Borland可执行连接器的完全路径名  
ASM6 Microsoft MASM 6可执行的完全路径名  
TASM32 Borland assembler的完全路径名  
NMSYM NuMega 符号文件生成器的完全路径名  
EDITBIN Microsoft EDITBIN 工具的完全路径名  
VTOOLSD 环境变量  
VTOOLSD 生成系统时需要定义环境变量VTOOLSD. 该变量的值必须是VTOOLSD 安装的根目录。  
例如:  
set VTOOLSD=C:\VTOOLSD  
SETUP 自动将该语句加入到你的AUTOEXEC.BAT文件中。如果你没有让SETUP这么做，那么你就必须手动的添加或每次用VTOOLSD在系统中生成VxD时设置变量。

**安装SoftICE**  
**安装之前：**  
1. 如果要将SoftICE安装在Windows NT 或Windows 2000上，你必须要有系统管理员的权限。  
2. 如果要将SoftICE安装在Windows Me上，在安装之前必须下载并安装"Microsoft Tools for Windows Me"  
3. 下列是各个显卡选项的需求列表，根据你自己的配置选择相应的驱动。  
l 一台显示器和显卡：  
根据显卡的生产厂家和型号来决定  
l 第二块单色显卡  
由主要的（第一台）显卡的生产厂家和型号来决定  
l 第二台计算机  
由串口特性或网络特性（如果使用网络）来决定  
l 第二块VGA显卡  
找到能与另一块显卡共存的显卡的驱动。  
4. 选择使用的鼠标的类型，串口或PS/2，如果使用了串口鼠标，搞清楚其连接的端口是COM1还是COM2  
5. 退出所有的Windows应用程序  
**安装**  
1. 选择好安装目录。  
2. 在显示适配器选择窗口，选择你将要使用的显示选项  
l 若要使用通用显卡驱动，选中"Universal Video Driver"复选框。  
l 若在只有一块显卡的计算机上使用SoftICE，选择显卡的生产厂家和型号。如果显卡未列出，在COMPATIBILIBY 框中选择相同的图形处理芯片。如果相同的芯片也没有，选择STANDARD VGA（640X480 pixels）.  
l 若在计算机的第二块单显上使用SoftICE，选择DISPLAY SOFTICE ON ATTACHED MONOCHROME MONITOR，然后选择主显卡的生产厂家和型号。如果显卡未列出，在COMPATIBILIBY 框中选择相同的图形处理芯片。如果相同的芯片也没有，选择STANDARD VGA（640X480 pixels）.  
l 若在另外一台计算机上使用SoftICE，选择主显卡的生产厂家和型号。如果显卡未列出，在COMPATIBILIBY 框中选择相同的图形处理芯片。如果相同的芯片也没有，选择STANDARD VGA（640X480 pixels）.  
l 若在计算机的第二块VGA卡上使用SoftICE，选择STANDARD VGA（640X480 pixels）。  
3. 如果你选好了相配的显卡设置，点击TEST来测试显卡的设置。如果测试不成功，参见步骤7。  
4. 如果采用了通用显卡驱动并且测试成功，系统会弹出一个对话框告诉你测试已通过。如果你测试其它任何显卡，将会有一个彩色的文本模式的屏幕出现大约5秒钟。  
如果SoftICE与你的显卡不兼容，测试失败，你看到的是一个黑屏。  
5. 如果在Windows NT或Windows 2000上安装，按照下列选项决定SoftICE加载的方式：  
BOOT  
SYSTEM  
AUTOMATIC  
MANUAL  
6. 选择鼠标的安装方式  
7. 如果在Windows 95或Windows 98下安装，选择系统配置窗口选项  
  
**安装 DriverNetworks**  
  
**编译库文件和驱动程序**  
  
随着Windows 2000 DDK的引入，网络驱动开发人员必须在同一台开发机器上对付两种DDK环境和两种或更多的目标平台: Windows NT 4 DDK 和 Windows 2000 DDK.  
对于一个给定的驱动推荐使用下列的DDK：  
NT 4 DDK 针对 NDIS 4 miniport 驱动 (目标平台: Windows NT 4 ,Windows 95)  
Windows 2000 DDK 针对 NDIS 5 miniport 驱动 (目标平台: Windows 2000, Windows 98, Windows Me)  
Windows 2000 DDK 针对 Intermediate Drivers 和 TDI Clients(目标平台: Windows NT 4 , Windows 2000, Windows 98, Windows Me)  
对于用DriverNetworks开发驱动程序来说选择正确的环境是非常关键的  
为驱动程序选择正确的Build环境  
DriverNetworks? 鼓励但不要求一定要使用Microsoft Visual Studio 6.0 IDE 作为驱动的开发环境。  
DriverStudio 提供两种途径在Visual Studio IDE中来选择正确的build 环境：  
1．使用DriverStudio中自带的SetDDKgo.exe工具在指定的DDK环境中启动Microsoft Visual Studio 6.0 IDE  
2. 在Microsoft Visual Studio 6.0 IDE的工具条中使用 Set DDK add-on 按钮设置合理的DDK 环境。  
利用SetDDKGo.exe 方法,一种使用 -K 命令行参数指定DDK的路径，  
例如 SetDDKGo -K c:\ddkw2k.  
如果不使用 -K 参数, SetDDKgo.exe 使用注册表中默认的DDK路径。  
编译库文件  
在使用 DriverNetworks? 编译驱动程序之前，必须先编译DriverNetworks? 库文件. 在IDE下打开DriverNetworks? 工作组文件, dnw.dsw. 其中由两个工程组成: KNdisLib.dsp 和 Tdiclient.dsp. 该工程中，依次包括针对不同的NDIS版本，checked/free 和目标平台的多种配置。  
下列部分简单描述了编译所需的必须的步骤：  
DriverNetworks? libraries under Microsoft Visual Studio IDE 6.0. 假设你的开发用机上安装了NT 4.0 和 Windows 2000 DDK.如果只安装了一种DDK，跳过相关的步骤  
1．启动 Microsoft Visual Studio IDE 6.0 IDE.  
2．单击New Workspace 并在$(DRIVERNETWORKS)\source目录中找到 dnw.dsw，打开该工程.  
3．在IDE工具条中单击DDK 按钮。选择 Windows 2000 DDK 的路径.注意通常会从Set DDK对话框中提示。  
4．从 IDE Build 菜单中选择 Batch Build.你将会看到在DriverNetworks 库配置表中的所有配置列表。选择除了KNdisLib - Win32 NDIS 4 Miniport Free 和 Checked以外的所有配置.  
5．选中Selection Only 检查框并点击Build.   
6．如果也安装了NT 4.0 DDK，回到步骤3指定DDK 4.0目录.在第4步中只选择 KNdisLib - Win32 NDIS 4 Miniport Free 和 Checked配置并编译。

-------------------------------------------------------------------

2006-01-19

### VC6, DDK, DriverStudio安装指南

1. 先安装VC6.0
2. 安装Win2KDDK。
3. 开始/程序/Development Kit/win2000DDK/Checked build environment/输入 build –cZ
4. 安装DriverStudio3.1版本
5. 开始/程序/…/DriverStudio/DriverWorks/Tools/Set…./ 启动VC

打开../ DriverWorks/Sources/vdwlibs.dsw，然后选择build/batch build. 去掉/A64相关选项。Build

-------------------------------------------------------------------

2006-01-27

### NAND和NOR flash的区别

NOR和NAND是现在市场上两种主要的非易失闪存技术。Intel于1988年首先开发出NOR flash技术，彻底改变了原先由EPROM和EEPROM一统天下的局面。紧接着，1989年，东芝公司发表了NAND flash结构，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。但是经过了十多年之后，仍然有相当多的硬件工程师分不清NOR 和NAND闪存。

　　相“flash存储器”经常可以与相“NOR存储器”互换使用。许多业内人士也搞不清楚NAND闪存技术相对于NOR技术的优越之处，因为大多数情况下闪存只是用来存储少量的代码，这时NOR闪存更适合一些。而NAND则是高数据存储密度的理想解决方案。

　　NOR的特点是芯片内执行(XIP, eXecute In Place)，这样应用程序可以直接在flash闪存内运行，不必再把代码读到系统RAM中。NOR的传输效率很高，在1～4MB的小容量时具有很高的成本效益，但是很低的写入和擦除速度大大影响了它的性能。

　　NAND结构能提供极高的单元密度，可以达到高存储密度，并且写入和擦除的速度也很快。应用NAND的困难在于flash的管理和需要特殊的系统接口。

性能比较

　　flash闪存是非易失存储器，可以对称为块的存储器单元块进行擦写和再编程。任何flash器件的写入操作只能在空或已擦除的单元内进行，所以大多数情况下，在进行写入操作之前必须先执行擦除。NAND器件执行擦除操作是十分简单的，而NOR则要求在进行擦除前先要将目标块内所有的位都写为0。

　　由于擦除NOR器件时是以64～128KB的块进行的，执行一个写入/擦除操作的时间为5s，与此相反，擦除NAND器件是以8～32KB的块进行的，执行相同的操作最多只需要4ms。

　　执行擦除时块尺寸的不同进一步拉大了NOR和NADN之间的性能差距，统计表明，对于给定的一套写入操作(尤其是更新小文件时)，更多的擦除操作必须在基于NOR的单元中进行。这样，当选择存储解决方案时，设计师必须权衡以下的各项因素。

　　● NOR的读速度比NAND稍快一些。

　　● NAND的写入速度比NOR快很多。

　　● NAND的4ms擦除速度远比NOR的5s快。

　　● 大多数写入操作需要先进行擦除操作。

　　● NAND的擦除单元更小，相应的擦除电路更少。

接口差别

　　NOR flash带有SRAM接口，有足够的地址引脚来寻址，可以很容易地存取其内部的每一个字节。

　　NAND器件使用复杂的I/O口来串行地存取数据，各个产品或厂商的方法可能各不相同。8个引脚用来传送控制、地址和数据信息。

　　NAND读和写操作采用512字节的块，这一点有点像硬盘管理此类操作，很自然地，基于NAND的存储器就可以取代硬盘或其他块设备。

容量和成本

　　NAND flash的单元尺寸几乎是NOR器件的一半，由于生产过程更为简单，NAND结构可以在给定的模具尺寸内提供更高的容量，也就相应地降低了价格。

　　NOR flash占据了容量为1～16MB闪存市场的大部分，而NAND flash只是用在8～128MB的产品当中，这也说明NOR主要应用在代码存储介质中，NAND适合于数据存储，NAND在CompactFlash、 Secure Digital、PC Cards和MMC存储卡市场上所占份额最大。

可靠性和耐用性

　　采用flahs介质时一个需要重点考虑的问题是可靠性。对于需要扩展MTBF的系统来说，Flash是非常合适的存储方案。可以从寿命(耐用性)、位交换和坏块处理三个方面来比较NOR和NAND的可靠性。

　　寿命(耐用性)

　　在NAND闪存中每个块的最大擦写次数是一百万次，而NOR的擦写次数是十万次。NAND存储器除了具有10比1的块擦除周期优势，典型的NAND块尺寸要比NOR器件小8倍，每个NAND存储器块在给定的时间内的删除次数要少一些。

　　位交换

　　所有flash器件都受位交换现象的困扰。在某些情况下(很少见，NAND发生的次数要比NOR多)，一个比特位会发生反转或被报告反转了。

　　一位的变化可能不很明显，但是如果发生在一个关键文件上，这个小小的故障可能导致系统停机。如果只是报告有问题，多读几次就可能解决了。

　　当然，如果这个位真的改变了，就必须采用错误探测/错误更正(EDC/ECC)算法。位反转的问题更多见于NAND闪存，NAND的供应商建议使用NAND闪存的时候，同时使用EDC/ECC算法。

　　这个问题对于用NAND存储多媒体信息时倒不是致命的。当然，如果用本地存储设备来存储操作系统、配置文件或其他敏感信息时，必须使用EDC/ECC系统以确保可靠性。

　　坏块处理

　　NAND器件中的坏块是随机分布的。以前也曾有过消除坏块的努力，但发现成品率太低，代价太高，根本不划算。

　　NAND器件需要对介质进行初始化扫描以发现坏块，并将坏块标记为不可用。在已制成的器件中，如果通过可靠的方法不能进行这项处理，将导致高故障率。

易于使用

　　可以非常直接地使用基于NOR的闪存，可以像其他存储器那样连接，并可以在上面直接运行代码。

　　由于需要I/O接口，NAND要复杂得多。各种NAND器件的存取方法因厂家而异。

　　在使用NAND器件时，必须先写入驱动程序，才能继续执行其他操作。向NAND器件写入信息需要相当的技巧，因为设计师绝不能向坏块写入，这就意味着在NAND器件上自始至终都必须进行虚拟映射。

软件支持

　　当讨论软件支持的时候，应该区别基本的读/写/擦操作和高一级的用于磁盘仿真和闪存管理算法的软件，包括性能优化。

　　在NOR器件上运行代码不需要任何的软件支持，在NAND器件上进行同样操作时，通常需要驱动程序，也就是内存技术驱动程序(MTD)，NAND和NOR器件在进行写入和擦除操作时都需要MTD。

　　使用NOR器件时所需要的MTD要相对少一些，许多厂商都提供用于NOR器件的更高级软件，这其中包括M-System的TrueFFS驱动，该驱动被 Wind River System、Microsoft、QNX Software System、Symbian和Intel等厂商所采用。

　　驱动还用于对DiskOnChip产品进行仿真和NAND闪存的管理，包括纠错、坏块处理和损耗平衡。

-------------------------------------------------------------------

2006-3-7

### C语言编程技巧

1. 对于大部分的编译器，使用指针比使用数组生成的代码更短，执行效率更高。但是在Keil中则相反，使用数组比使用的指针生成的代码更短。
2. 使用尽量小的数据类型
3. 使用自加、自减指令a+=1
4. 求余运算。a=a%8; 可以改为：a=a&7;
5. 平方运算a=pow(a,2.0); 可以改为：a=a\*a;
6. a=pow(a,3.0);更改为：a=a\*a\*a；则效率的改善更明显。
7. 用移位实现乘除法运算a=a\*4;b=b/4;可以改为：a=a<<2;b=b>>2;
8. a=a\*9可以改为：a=(a<<3)+a
9. 因为几乎所有的MCU均有为0转移的指令，采用以下指令做循环会节省指令void delay (void){unsigned int i;for (i=1000;i>0;i--);}。While循环也是如此

-------------------------------------------------------------------

### ARM7与ARM9区别

1. 时钟频率的提高

ARM7处理器采用3级流水线(预取指，译码，执行)，而ARM9采用5级流水线（预取指，译码，执行，存储器，写回）。增加的流水线设计提高了时钟频率和并行处理能力。5级流水线能够将每一个指令处理分配到5个时钟周期内，在每一个时钟周期内同时有5个指令在执行。在同样的加工工艺下，ARM9TDMI处理器的时钟频率是ARM7TDMI的1．8～2．2倍。

1. ARM9结构及特点

以ARM9E-S为例介绍ARM9处理器的主要结构及其特点。ARM9E-S的结构如图4所示。其主要特点如下：

(1)32bit定点RISC处理器，改进型ARM／Thumb代码交织，增强性乘法器设计。支持实时(real-time)调试；

(2)片内指令和数据SRAM，而且指令和数据的存储器容量可调；

(3)片内指令和数据高速缓冲器(cache)容量从4K字节到1M字节；

(4)设置保护单元(protcction unit)，非常适合嵌入式应用中对存储器进行分段和保护；

(5)采用AMBA AHB总线接口，为外设提供统一的地址和数据总线；

(6)支持外部协处理器，指令和数据总线有简单的握手信令支持；

(7)支持标准基本逻辑单元扫描测试方法学，而且支持BIST(built-in-self-test)；

(8)支持嵌入式跟踪宏单元，支持实时跟踪指令和数据。s

1. 指令周期的改进

指令周期的改进对于处理器性能的提高有很大的帮助。性能提高的幅度依赖于代码执行时指令的重叠，这实际上是程序本身的问题。对于采用最高级的语言，一般来说，性能的提高在30％左右。

2．1 loads 指令矛n stores指令

指令周期数的改进最明显的是loads指令和stores指令。从ARM7到ARM9这两条指令的执行时间减少了30％。指令周期的减少是由于ARM7和ARM9两种处理器内的两个基本的微处理结构不同所造成的。

(1)ARM9有独立的指令和数据存储器接口，允许处理器同时进行取指和读写数据。这叫作改进型哈佛结构。而ARM7只有数据存储器接口，它同时用来取指令和数据访问。

(2)5级流水线引入了独立的存储器和写回流水线，分别用来访问存储器和将结果写回寄存器。

以上两点实现了一个周期完成loads指令和stores指令。

2．2 互锁(interlocks)技术

当指令需要的数据因为以前的指令没有执行完而没有准备好就会产生管道互锁。当管道互锁发生时，硬件会停止这个指令的执行，直到数据准备好为止。虽然这种技术会增加代码执行时间，但是为初期的设计者提供了巨大的方便。编译器以及汇编程序员可以通过重新设计代码的顺序或者其他方法来减少管道互锁的数量。

2．3 分枝指令

ARM9和ARM7的分枝指令周期是相同的。而且ARM9TDMI和ARM9E-S并没有对分枝指令进行预测处理。s

-----------------------------------------------------------------------------------------------------------------------------------

### Usage of OSTaskCreatHook()

OSTaskCreateHook() can be declared either in OS\_CPU\_C.C (if OS\_CPU\_HOOKS\_EN is set to 1) or elsewhere. Note that interrupts are disabled when OSTaskCreate() calls OSTaskCreateHook(). Because of this, you should keep the code in this function to a minimum because it can directly impact interrupt latency.

-------------------------------

### BootLoader的概念

[精品]Bootloader的概念 流星 发表于 2005-8-9 19:49:37

一、Bootloader的基本概念

一个嵌入式Linux系统从软件的角度看通常可以分为四个层次：引导加载程序、Linux内核、文件系统、用户应用程序。

引导加载程序是系统加电后运行的第一段代码。我们熟悉的PC中的引导程序一般由BIOS和位于MBR的OS bootloader（例如LILO或者GRUB）一起组成。然而在嵌入式系统中通常没有像BIOS那样的固件程序（有的嵌入式CPU有），因此整个系统的加载启动任务就完全由bootloader来完成。在嵌入式Linux中,引导加载程序即等效为bootloader。

简单地说，bootloader就是在操作系统内核运行前运行地一段小程序。通过这段小程序，我们可以初始化必要的硬件设备，创建内核需要的一些信息并将这些信息通过相关机制传递给内核，从而将系统的软硬件环境带到一个合适的状态，最终调用操作系统内核，真正起到引导和加载内核的作用。

bootloader是依赖于硬件而实现的，特别是在嵌入式系统中。不同的体系结构需求的bootloader是不同的；除了体系结构，bootloader还依赖于具体的嵌入式板级设备的配置。也就是说，对于两块不同的嵌入式板而言，即使它们基于相同的CPU构建，运行在其中一块电路板上的bootloader，未必能够运行在另一块电路开发板上。

Bootloader的启动过程可以是单阶段的，也可以是多阶段的。通常多阶段的bootloader能提供更为复杂的功能，以及更好的可移植性。从固态存储设备上启动的bootloader大多数是二阶段的启动过程，也即启动过程可以分为stage 1和stage 2两部分。

二、Bootloader的操作模式

大多数bootloader都包含两种不同的操作模式：“启动加载”模式和“下载”模式，这种区别对于开发人员才有意义。但从最终用户的角度看，bootloader的作用永远就是用来加载操作系统，而并不存在所谓的启动加载模式与下载工作模式的区别。

启动加载模式：这种模式也称为“自主”模式，即bootloader从目标机上的某个固体存储设备上将操作系统加载到RAM中运行，整个过程没有用户的介入。这种模式是bootloader的正常工作模式，因此当以嵌入式产品发布的时候，bootloader必须工作在这种模式下。

下载模式：在这种模式下，目标机上的bootloader将通过串口或者网络连接或者其它通信手段从主机下载文件，比如：下载内核镜像和根文件系统镜像等。从主机下载的文件通常首先被bootloader保存到目标机的RAM中，然后被bootloader写到目标机上的FLASH类固态存储设备中。Bootloader的这种模式通常在第一次安装内核与根文件系统时使用；此外，以后的系统更新也会使用bootloader的这种工作模式。工作于这种模式下的bootloader通常都会向它的中断用户提供一个简单的命令行接口。

三、Bootloader的概念扩展

Bootloader最主要的功能是引导加载内核镜像。但是随着嵌入式系统的发展，bootloader已经逐渐在基本功能的基础上，进行了扩展，bootloader可以更多地增加对具体系统的板级支持，即增加一些硬件模块功能上的使用支持，以方便开发人员进行开发和调试。从这个层面上看，功能扩展后的bootloader可以虚拟地看成是一个微小的系统级的代码包。

四、 ARM Bootloader的共性

对于一个ARM系统来说，本质上，bootloader作为引导与加载内核镜像的“工具”，在实现上，必须提供以下几个功能，更确切地说，必须做到以下几点：

1. 初始化RAM（必需）：bootloader必须能够初始化RAM，因为将来系统要通过它保存一些Volatile数据，但具体地实现要依赖与具体的CPU以及硬件系统。
2. 初始化串口（可选，推荐）：bootloader应该要初始化以及使能至少一个串口，通过它与控制台联系进行一些debug的工作；甚至与PC通信。
3. 创建内核参数列表（针对linux操作系统，推荐）。
4. 启动内核镜像（必需）：根据内核镜像保存的存储介质不同，可以有两种启动方式：FALSH启动以及RAM启动；但是无论是哪种启动方式，下面的系统状态必须得到满足：

CPU寄存器的设置： R0＝0

R1＝机器类型；

R2＝启动参数标记列表在RAM中的起始地址；

这三个寄存器的设置是在最后启动内核时通过启动参数来传递完成的。

CPU模式： 关闭中断；属于SVC模式；

Bootloader中没有必要支持中断的实现，这属于内核机制以及设备驱动管理的管理范畴；SVC模式是系统的一种保护模式，这样就可以进行一些只能在SVC模式下的操作，例如一些特定寄存器访问操作。

Cache和MMU的设置：MMU必须关闭；数据cache必须关闭；指令cache可以关闭也可以开启；

Bootloader中所有对地址的操作都是使用物理地址，是实在的实地址，不存在虚拟地址，因此MMU必须关闭。Bootloader主要是装载内核镜像，镜像数据必须真实写回SDRAM中，所以数据cache必须关闭；而对于指令cache，不存在强制性的规定，但是一般情况下，推荐关闭指令cache。

Bootloader启动内核镜像的方法是通过跳转语句直接跳转至内核镜像的第一句指令语句。

-------------------------------

### 多进程、多线程、多任务概念

多任务和多线程的基本概念是一样的。只不过多任务的概念是应用在嵌入式操作系统里；而多进程是应用于软件工程。二者共同的特点是：各个线程或者任务之间共享同一块内存资源，资源可以被多个任务或者线程使用。

而多进程的概念是不一样的。每个不同的进程之间的资源是独占的，不能被其他进程所使用。但是一个进程中的多个线程之间的资源是可以相互共享的。

-------------------------------

### SCU说明

SCU， Snoop Control Unit,用来保持双核之间的数据Cache的一致性。 也就是是说，第一个A9

处理器写存储时，只是写在了缓存里，没有进主存，如果第二个A9读操作，涉及到第一个写脏了的数据段，SCU要保证第二个A9的缓存里是最新的数据。 如果第二个A9写同样数据段的数据，需要在第一个中体现出写的内容。SCU的存在，才使得两个核成互相联系的“双核”，才能成为MPsoc。在原先XILINX的双PowerPC的芯片中， 是木有这个东东的。不少学校的老师拿XILINX的双powerpc练手和教学，从头搭一个Snoop协议在裸的双PowerPC中，倒也不错

-------------------------------

### ARM WFI

Wait for interrupt (WFI/WFE) mode

Wait for Interrupt mode disables most of the clocks of a CPU, while keeping its logic powered up. This reduces the power drawn to the static leakage current, plus a tiny clock power overhead required to enable the device to wake up from the WFI state. The transition from the WFI mode to the Run mode is caused by:

an interrupt, masked or unmasked

a debug request, regardless of whether debug is enabled

a reset.

-------------------------------

### Standby, suspend, hibernate

Standby

Standby is the action where the CPU is held in a low power state but no data is saved to RAM or hard disk. It normally takes a split second to standby and resume. You cannot remove the battery when on laptop power or work will be lost.

Recommended Nomenclature

Forward action: STANDBY

Reverse action: CONTINUE (or RESUME for user-documentation or translations)

Suspend

Suspend is the action where the computer freezes all activity, and copies working data to RAM. It then turns off the screen, and goes into a low-power mode. It normally takes a few seconds to suspend and a few seconds to resume. You cannot remove the battery when on laptop power or work will be lost.

Recommended Nomenclature

Forward action: SUSPEND

Reverse action: RESUME

Suspend is a good word as the computer is not powered down but in a temporary sleep. A user can understand that suspend'ing the work is not the same as hibernating. You can boot other OS's, or change batteries when hibernated.

Resume is a good word as the user un-suspends the computer, and resumes all work. It is also previously used by pm-tools, HAL and the kernel and Microsoft Windows.

Bad Nomenclature

Sleep

Standby

Suspend-to-RAM

Sleep is a BAD word as it does not convey time, and the name sleep has been used to mean standby and suspend and hibernate over the years.

Standby is a name that does not convey what is actually about to happen - "standby for what?" and is used by the kernel in ACPI for a power saving state that is rarely used. See above...

Suspend-to-RAM is a bad description as it forces the user to know about hardware when it should be abstract. s-t-r may be great for programmers, but is not intuitive for new users.

Hibernate

Hibernate is the action where the computer freezes all activity and swaps data to disk, turns off the screen and powers down. This takes over one minute for me to swap my ram to disk, and nearly 40 seconds to thaw.

Recommended Nomenclature

Forward action: HIBERNATE

Reverse action: THAW (or RESUME for user-documentation or translations)

Hibernate is used by pm-tools, HAL, the kernel and Microsoft Windows already, and convey nicely that hibernating is a long term thing, and takes longer than suspending.

The backwards action name isn't as important for end users IMO. The user will only ever click a "Suspend" button, not a "Resume" button. This, and the difficulty translating thaw into non-english languages, I think it is permissable to add to user-visible parts of the stack hibernate and resume as a valid phrase. The logic - a user doesn't care - but we must use thaw as the programmatic name as the actions from return from standby and return from hibernate may be very different.

Bad Nomenclature

Wake

Suspend-to-disk

Wake is bad as it is the opposite of sleep, not hibernate. Wake has been used in the past for wakeup from standby, wakeup from hibernate and wakeup from suspend.

Suspend-to-disk is a bad description as it also forces the user to know about hardware. Why should a new user be expected to know that writing to RAM is quicker than writing to disk?

Conclusion

I really do think getting these 6 words correct will make a big difference to the bigger picture. When people are using the correct words to describe problems, and users know that suspend does the same thing on GNOME and on KDE.

-------------------------------

### JLINK设置

1. 安装“Setup\_JLinkARM\_V402.exe”驱动
2. 运行RVDS，Target/Connect to Target/File/Manage RDI Components/Add Dll.
3. Choose “D:\SEGGER\JLinkARM\_V402\JLinkRDI.dll”
4. Target/Connect to Target/Configure 降低Jtag时钟

-------------------------------

2018-07-11

### C/C++头文件和变量定义和声明

1. 不要在头文件里定义什么东西，比如全局变量。头文件中变量如果没有初始化就是声明；初始化了就是定义，此时被包含之后会发生重复定义的错误。全局变量的定义宜放在某个 .c 文件中，而 .h 文件里写的是变量的声明。头文件应用于声明而不是用于定义。
2. 不是所有的变量都能用extern声明，只有全局变量并且没有被static 声明的变量才能声明为extern. 所以如果你不想自己源文件中全局的变量被其他文件引用，你就可以给变量加上static声明
3. 如果所要定义的变量为局部变量，并且其值在编译时就已经可以确定，就可以用const关键词在头文件中进行定义：非const变量默认为extern const变量，默认为文件的局部变量。而const变量如果想要在其他文件里被访问，必须显示的指定它为extern。如果const对象在编译时可以确定其值，可以将其放在头文件中定义。这么做是有原因的——const对象默认情况下是定义该变量的文件的局部变量，这样即使此头文件被多个包含，都不会在包含这些文件中重复定义此变量，所有对于这个const对象的引用全都指向在头文件中定义的对象。如果在const对象定义的时候声明extern，const对象就变得和普通非const对象一样，从而失去此特性（非const对象的默认属性为extern）
4. .h文件中一般只能包含全局变量的声明，函数声明，宏定义一类的，在.h文件中定义变量是不被推荐的

-------------------------------

### c语言 中 为什么不将全局变量的定义放在头文件中

原因有二：

一、跟踪难度大。如果工程小，跟踪其变化没有什么难度，如果工程很大，包含这个头文件的文件都有可能修改其值，出了问题不好排查。

二、c主要还是用于嵌入式，与硬件有关。许多嵌入式系统的内存不想电脑那么大，如果在头文件中声明全局变量，那么所有引用该头文件的文件都将为此变量非配内存，这样降低了内存的利用率，有时几K就是致命的。

-------------------------------

### 在C语言头文件里“定义”全局变量的方法

有时候我们想在多个C程序中使用某些全局变量，我们会想到在头文件里声明变量，再在C文件里#include之。这样的话我们就必须得把变量的定义放在某一个C文件中，但是我们又不想这么做因为会使程序看起来臃肿。让我们来看看ucos ii是如何巧妙地解决这个问题的

ucos ii在其头文件ucos\_ii.h中有这么一段代码：

#ifdef OS\_GLOBALS

#define OS\_EXT

#else

#define OS\_EXT extern

#endif

然后用OS\_EXT宏来声明变量，如：OS\_EXT INT16U OSRdyGrp;

再在其中一个包含ucos\_ii.h的C文件中#define OS\_GLOBALS（见ucos ii 的os\_core.c）。注意这句话必须放在#include "ucos\_ii.h"之前，原因你懂的（如果不懂，且看下文）。

只要明白#include "xxx.h"语句的本质就明白了这一技巧的原理。编译过程中编译器对#include"xxx.h"语句的处理方法是用xxx.h中的全部内容替换这一语句。上面的例子中只有os\_core.c中定义了#define OS\_GLOBALS ，故只有在编译该文件时OS\_EXT宏会被处理为什么都没有，而在其他C文件中处理为extern关键字。

-------------------------------

### C/C++全局变量定义的一般方法

一般来说，不会将全局变量的定义写在头文件中，因为如果多个c源文件都添加了头文件，那很容易引起重定义的问题，这时候一般编译器都会提示。正确的作法是在c源文件中定义一个全局变量。在头文件中加入全局变量的声明。比如

//var.c 一个主要用于定义全局变量的c源文件

int g\_a;

char g\_x;

//var.h 用于声明定义的全局变量

extern int g\_a;

extern char g\_x;

//test.c 一个用于测试的头文件

#include "var.h"

void f(){

g\_a = 1;

g\_x = 'x';

}

-------------------------------

### 一般规则，头文件(.h)应包含:

a.宏定义（预处理#define）；

b.结构、联合和枚举声明；

c.typedef声明；

d.外部函数声明；

e.全局变量声明；

当声明或定义需要在多个文件中共享时，把他们放入一个头文件中尤其重要。不要在两个或多个源文件的顶部重复声明或定义宏。应该把它们放入一个头文件中，然后在需要的时候用#include包含进来。这样做的原因并不仅仅是减少打字输入——这样可以保证在声明或定义变化的时候，只需要修改一处即可将结果一致地传播到各个源文件。（特别是，永远不要把外部函数原型放到.c文件中。参见2.）

另一方面，如果定义或声明为一个源文件私有则最好留在该文件中。（作用域限于单文件的私有函数和变量应该声明为static。）

最后，不能把实际的代码（如函数体）或全局变量定义（即定义和初始化实例）放入头文件中。而且，当用多个源文件创建一个项目的时候，应该单独编译每个文件（使用特定的编译选项，只进行编译），然后用链接器将生成的目标文件链接起来。（如果是集成开发环境，这些事情可能已经不需要你自己操心了。）不要试图用#include把你的源文件“链接”成一个整体。#include是用来引入头文件而不是.c文件的

-------------------------------

-------------------------------

-------------------------------

-------------------------------

-------------------------------

-------------------------------

### 末尾