嵌入式复习资料

1. 嵌入式系统基础

1. 日常生活所广泛使用的嵌入式设备有哪些？

手机、平板计算机、电视机顶盒、车载 GPS、智能家电、机器人。

2. CISC 与 RISC 之间的区别？

CISC：复杂指令集计算机。RISC：精简指令集计算机。

CISC 中各种指令使用频率相差悬殊，大约有 20%的指令被反复使用，占整个程序代码的 80%，而余下的 80%指令却不经常使用。

RISC 结构优先选取使用频率最高的简单指令，避免复杂指令；将指令固定，指令格式和寻址方式种类减少；以逻辑为主。

3. 哈佛结构与冯诺依曼结构区别？

二者的区别就是程序存储空间和数据存储空间是否是一体的。

在哈佛总线体系结构的芯片内部，数据存储空间和程序存储空间是分开的，所以哈佛总线结构在指令执行时，可以同时取指令(来自程序空间)和取操作数(来自数据空间)，因此具有更高的执行效率。

在冯诺依曼结构中，存储器内部的数据存储空间和程序存储空间是合在一起的，它们共享存储器总线，即数据和值令在同一条总线上通过时分复用的方式进行传输。这种结构在高速运行时，不能达到同时取指令和取操作数的目的，从而形成了传输过程上的瓶颈。

4. 写出下列英文缩写的中文含义。USB RTOS RISC TCP IPC GPIO （简答题）

USB：通用串行总线　　RTOS：实时操作系统

RISC:精简指令集计算机　　TCP：传输控制协议

IPC:进程间通信　　GPIO:通用输入输出

5. 比较嵌入式系统与通用计算机的区别。（简述题）

嵌入式系统是以应用为中心，以计算机技术为基础，并且软硬件可裁剪，适用于应用系统对功能、可靠性、成本、体积、功耗有严格要求的专用计算机系统。它一般由嵌入式微处理器、外围硬件设备、嵌入式操作系统以及用户的应用程序等四个部分组成，用于实现对其他设备的控制、监视或管理等功能。

6. 中断处理经过了哪几个阶段。（简述题）

（1）禁止其他中断　　（2）保存上下文

（3）中断处理程序　　（4）中断服务程序

（5）恢复上下文　　（6）允许新的中断

2. 嵌入式硬件体系结构

1. 嵌入式微处理器（ARM）三级流水线、五级流水线？

ARM7 是三级流水线，分别为：取指、译码、执行。

ARM9 是五级流水线，分别为：取指、译码、执行、访存、写入。

通过增加流水线级数，从而简化各级逻辑，提高处理器的性能。

取指——从存储器装载一条指令

译码——识别将要被执行的指令

执行——处理指令并将结果写回寄存器

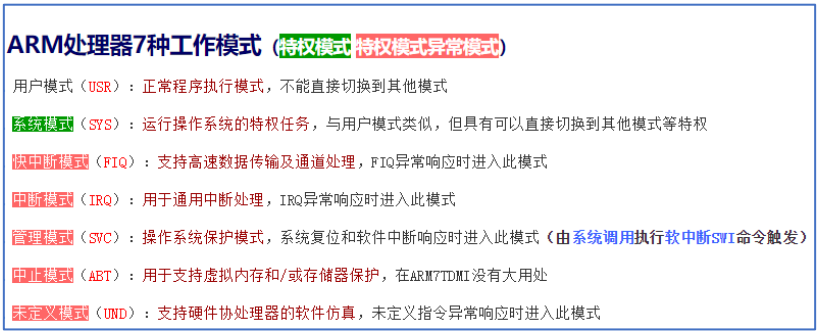
2. ARM 处理器的工作状态？

ARM 状态，此时处理器执行 32 位的字对齐的 ARM 指令；

Thumb 状态，此时处理器执行 16 位的、半字对齐的 Thumb 指令。

3. ARM 处理器的工作模式？

用户 系统 快速中断 中断 管理 数据访问终止 未定义指令终止



4. ARM 处理的中断分别是哪两种模式？

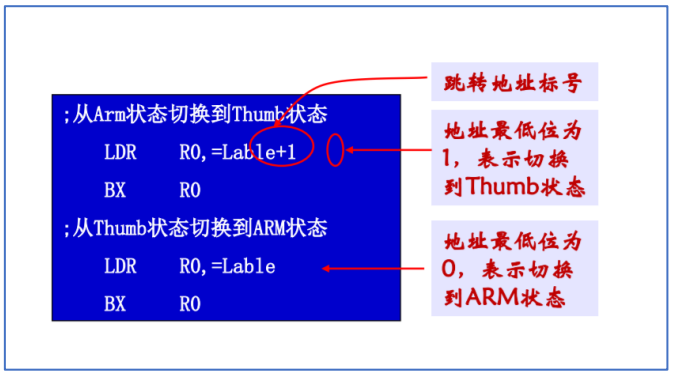
IRQ 是指中断模式，FIQ 是指快速中断模式。对于 FIQ 你必须尽快处理你的事情并离开这个模式。IRQ 可以被 FIQ 所中断，但 IRQ 不能中断 FIQ。

5. 寄存器 CPSR，SPSR 的功能各是什么？

CPSR 包含条件码标志、中断禁止位、当前处理器模式以及其它状态和控制信息。所有处理器模式下都可以访问当前的程序状态寄存器 CPSR。

SPSR：在每种异常模式下都有一个对应的物理寄存器——程序状态保存寄存器 SPSR。当异常出现时，SPSR 用于保存 CPSR 的状态，以便异常返回后恢复异常发生时的工作状态。

6. 使用 BX 指令将 ARM7TDMI 内核的操作状态在 ARM 状态和 Thumb 状态之间进行切换。



3. ARM 指令系统

1. ARM 指令寻址方式

寄存器寻址；（掌握）

MOV R1,R2 ——将 R2 寄存器中的值存入 R1 寄存器

立即寻址；（掌握）

MOV R1,#1 ——将立即数 1 装入 R1 寄存器

寄存器移位寻址；（掌握）

MOV R1,R2,LSL #3 ——R2 的值左移 3 位，结果放入 R1

MOV R1,R2,LSL R3 ——R2 的值左移与 R3 寄存器内值相等的位数，结果放入 R1

寄存器间接寻址；（掌握）

LDR R1,[R2] ——将 R2 寄存器指向存储单元数据读出，保存在 R1 中

基址寻址；多寄存器寻址；堆栈寻址；块拷贝寻址；相对寻址。

2.单寄存器 Load/Store 指令 STR LDR （简答题）

数据从存储器到寄存器的传送叫加载(Load)，数据从寄存器到存储器的传送叫存储(Store)。

LDR 指令是字加载指令，用于从存储器中将一个 32 位的字数据传送到目的寄存器中。

格式为：LDR{条件} 目的寄存器，<存储器地址>

LDR R3,[R4];将 R4 指向地址的字数据存入 R3

LDR R3，[R1，＃8] 将存储器地址为 R1+8 的字数据读入寄存器 R3。

LDR R3，[R1，R2]！将存储器地址为 R1+R2 的字数据读入寄存器 R3，并将新地址 R1＋R2 写入 R1。

STR 指令是字存储指令，用于从源寄存器中将一个 32 位的字数据传送到存储器中。

格式为：STR{条件} 源寄存器，<存储器地址>

STR R3, [R1], ＃8 将 R3 中的字数据写入以 R1 为地址的存储器中，并将新地址 R1＋8 写入 R1。

STR R3, [R1，＃8] 将 R3 中的字数据写入以 R1＋8 为地址的存储器中。

3.下列 ARM 指令将做什么?

a) LDR r0,[r1,#6] 将存储器地址为 r1+6 的字数据读入寄存器 r0。

b) LDR r0,=0x999 将 0x999 读入寄存器 r0。

4.Linux 基础

1.Linux 经常使用的命令比如：ls、cp、mv、chmod、chown、cat、rm、touch

ls：列出目录下的文件清单（ls -a ls -l）

cp：拷贝文件或者目录（cp （参数） 源文件 目标文件）(-f:若存在，删除再拷贝，不提示；-i：若存在，提示是否覆盖；-r：递归拷贝）

mv 命令：移动文件或目录

stat 命令：显示文件或目录的各种信息

rm：删除文件/目录

touch:用来修改文件时间戳，或者新建一个不存在的文件

mkdir:新建文件夹/目录

chmod：更改文件的权限

chown：更改文件的用户及用户组

cat：显示文件的内容

useradd：添加用户（groupadd：添加用户组）

2.怎么使用 chmod 命令?

chmod 是一条在 Linux 系统中用于控制用户对文件的权限的命令。(权限分别是：文件所有者（user)、用户组（group）其他用户组（other)：读、写。执行）

使用语法：sudo chmod 权限 文件名

chmod [option] … MODE[,MODE]… FILE…

3.777 666 444 642 都是怎么意思?

777——所有用户均有读、写、执行文件权限

666——所有用户均有读、写文件权限

444——所有用户均只有读文件权限

642——所有者具有读、写文件权限，同组用户只有读文件权限，其他用户只有写文件权限

4.ubuntu 关闭防火墙命令？

service iptables stop（关闭）(Linux)

查看防火墙状态

sudo ufw status

开启/关闭防火墙（默认设置是’disable’）

sudo ufw enable|disable）

5.Mount 命令使用

将宿主机 NFS 服务所共享的目录/mnt/nfs 挂载到开发板的/mnt/yidao 目录下（假设宿主机的 IP 地址为 192.168.0.253），使用的命令是_____。

mount -t nfs 192.168.0.253:/mnt/nfs /mnt/yidao -o nolock

6.Vi 的工作模式？

命令模式、插入模式、编辑模式

7.Vi 命令行模式-----各种命令的意思！

1、指令模式(Command Mode）

指令模式主要使用方向键移动光标位置进行文字的编辑，下面列出了常用的操作命令及含义。

0 ——光标移动至行首　　h ——光标左移一格

l ——光标右移一格　　j ——光标下移一行

k ——光标上移一行　　$+A——将光标移动到该行最后

PageDn ——向下移动一页　　PageUp ——向上移动一页

d+方向键 ——删除文字　　dd ——删除整行

pp ——整行复制　　r ——修改光标所在的字符

S ——删除光标所在的列，并进入输入模式

2、文本输入模式（Input Mode）

在指令模式下（Command Mode）按 a / A 键、i/I 键、o / O 键进入文本模式，文本输入模式的命令及其含义如下所示。

a ——在光标后开始插入

A ——在行尾开始插入

i ——从光标所在位置前面开始插入

I ——从光标所在列的第一个非空白字元前面开始插入

o ——在光标所在列下新增一列并进入输入模式

O ——在光标所在列上方新增一列并进入输入模式

ESC —返回命令行模式

3、末行模式（Last line Mode）

末行模式主要进行一些文字编辑辅助功能，比如字串搜索、替代、保存文件等操作。主要命令如下

q ——结束 Vi 程序，如果文件有过修改，先保存文件

q! —强制退出 Vi 程序

wq —保存修改并退出程序

set nu —设置行号

8.什么是 GCC？它的执行过程包括哪四个阶段？

GCC 是 Linux 平台下最常用的编译程序，是 Linux 平台编译器的实施标准。

四个阶段：预处理（预编译）、编译、汇编、链接

9. Shell 脚本文件中变量都有哪些变量？

系统变量：是指 Bash Shell 内部定义的保留变量，

环境变量：可以看成是在整个 Shell 的各个程序都能访问的全局变量

用户自定义变量：由用户自行定义.定义格式为 变量名=值，等号两边不能有空格，引用时用$

10. Shell 中单引号与双引号作用？

在 Linux Shell 脚本里，字符串 ‘’ 和”” 表示不同的含义，最大区别就是在于对变量引用的处理.

单引号‘’中包括字符串，如果字符串包含保留字符，则保留字符失效，按原样输出

x=5;echo ‘x=$x’ 输出 x=$x

双引号“”中包括字符串，如果字符串包含保留字符，则保留字符生效，如变量要显示变量值

x=5;echo “x=$x” 输出 x=5

11. Shell 编程—流程控制语句。

重点考察 shell 程序执行结果：

```
#!/bin/sh
for var in abc xyz
do
echo "$var 123456"
done
```

abc 123456

xyz 123456

12. 在机房做 gdb 调试实验时；为了使用 gdb 调试，我们是如何编译程序的?

GDB 程序调试的对象是可执行文件，如果要让产生的可执行文件可以用来调试，需要在执行 GCC 指令编译程序时，加上-g 参数，这样才可用 gdb 来调试该可执行文件。

例： $gcc -g -o progb.1_1 progb.1.c

要利用 GDB 调试程序，请在执行 GDB 指令时，加上要调试的可执行文件名。

$gdb progb.1

(gdb)

13. 我们都敲过哪些命令，这些命令的含义是什么？

列出可执行文件的程序代码：（gdb）list

执行程序：(gdb) run（中断程序继续执行，则按 Ctrl+C 键）

暂时回到 LINUX 提示符：(gdb)shell。exit 又进入 gdb 调试

显示操作命令的在线帮助：help

查看当前源程序的信息：info source

查看断点的信息和状态：info break（br）

当程序执行到一半因为中断点而停止时，若想让程序继续向下执行，输入 continue 操作命令

使中断点失效：disable +断点号

使中断点生效：enable +断点号

14. 怎么设置断点？

设置调试程序的中断点：break 6（中断点设置在第 6 行），也可以是 br swap（断点设置在 swap 函数处，br 是 break 的简写）

15. 怎么查看变量值？

print 操作命令可列出程序执行到此时的某项变量的值：print i（变量名）

5.1 嵌入式系统开发环境

1. 什么是宿主机？

安装一台装有指令操作系统的 PC 机称作宿主开发机

2. 什么是目标机？

除宿主机虚拟机外的网络上的第三方机器（如嵌入式开发板）称为目标机

3. NFS 是什么？

网络文件系统服务。是 linux 系统中经常使用的一种数据文件共享服务

4. 什么是交叉编译？为什么要采用交叉编译？（简答题）

（1）交叉编译是指在一个平台上生成可以在另一个平台上执行的代码。

（2）因为对于嵌入式系统来说，在本机没有足够的资源来运行开发工具和调试工具。

5.2 嵌入式 Linux 系统的构建

1. Bootloader 的两种操作模式？

启动加载模式：

在这种模式下，BootLoader 从目标机的某个固态存储设备上将操作系统加载到 RAM 中运行。

下载模式：

目标机上的 BootLoader 通过串口或网络连接等通信手段从宿主机上下载文件。

2. 向无任何程序的目标机中写入 Bootloader 程序，一般使用哪种接口？

JTAG 接口

3. Linux 有两种工作界面：字符界面和图形界面！

4. 内核裁剪的流程？

1、将新内核拷贝到/usr/src 下，

2、解压缩内核：

tar jxvf linux.2.6.29.tar.bz2

3、cd /usr/src/linux

4、清除不必要的文件：make disclean

5、裁剪内核：系统运行的必要配置+动态添加其他配置，make menuconfig 生成一个.config 文件，文件位置位于 arch/$cpu/configs

6、menuconfig 配置菜单，选择相应配置时，有三种选择方式：Y--将该功能编译进内核、N--不将该功能编译进内核、M--将该功能编译成可以在需要的时候动态插入到内核的模块，通过空格键进行选取，可以将其重命名，以便下次直接使用 cp mini2451_config .config

7、将配置好的内核进行交叉编译：make ARCH=arm CROSS_COMPILE=arm-linux-

8、生成 uImage：内核镜像在 arch/arm/boot 中的 zImage

将 mkimage 工具拷贝/bin：cp mkimage /bin

生成内核镜像 uImage：make uImage

9、编译内核模块
命令：make modules
10、安装内核模块
命令：make modules_install
将编译好的内核模块从内核源代码目录拷贝至/lib/modules 下
11、通过 modprobe filename 或 insmod hello.ko(路径：/sbin/insmod)把模块加载到内核中，modprobe -r filename 和 rmmod hello.ko 从内核中卸载模块
简述为：下载内核，解压内核，裁剪配置菜单，交叉编译内核，编译内核模块，安装内核模块。

5.请根据你对嵌入式系统中Bootloader的理解,简要设计一下stage1和 stage2需要完成的功能。(简述题)

Stage1：完成初始化硬件，为 stage2 准备内存空间，并将 stage2 复制到内存中，设置堆栈，然后跳转到 stage2。

Stage2：初始化本阶段要使用到的硬件设备；将内核映像和根文件系统映像从 flash 上读到 RAM 中；调用内核。

5.3 嵌入式 Linux 系统编程

1.通过什么函数可以获得父、子进程的 ID 号？

子进程：getpid()、父进程：getppid()

2.进程与程序的区别？(选择题)

程序是一组有序的静态指令，进程是一次程序的执行过程
程序可以长期保存，进程是暂时的
程序没有状态，而进程是有状态的

3.wait 函数与 waitpid 函数的区别？(选择题)

wait 函数用于使父进程（即调用 wait 的进程）阻塞，直到一个子进程结束或者该进程接到了一个指定的信号为止

wait 函数调用时，如果该父进程没有子进程或者他的子进程已经结束，则 wait 就会立即返回

waitpid 函数用于使父进程（即调用 wait 的进程）阻塞，并可提供一个非阻塞版本的 wait 功能

4.什么是进程？怎样区别子进程和父进程？（简答题）

进程是一个具有独立功能的程序的一次动态执行过程。简言之，进程就正在执行的程序。在子进程中返回 0 值，而在父进程中返回子进程的进程号 PID。

5.什么是僵尸进程？什么是孤儿进程？ （简答题）

僵尸进程：僵尸进程是指它的父进程已经退出(父进程没有等待(调用 wait/waitpid)它)，而该进程 dead 之后没有进程接受，就成为僵尸进程，也就是(zombie)进程。

孤儿进程：一个父进程退出，而它的一个或多个子进程还在运行，那么那些子进程将成为孤儿进程。

6.什么是进程描述符？怎样获得进程描述符？ （简答题）

进程描述符是一个非零的正整数。通过调用函数 getpid( )可以获得当前进程的 PID。

7.设已有C语言的源代码hello.c,现编写一个程序,在程序中自动编译hello.c,并运行编译后的执行文件 hello。(程序题)

注意：必须通过 fork 函数产生一个新进程，然后调用 exec 函数来执行 hello。

考察方式：根据要求写出完成的程序

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
int main()
{
    system("gcc -o hello hello.c");
    if (fork() == 0)
    {
        if (execlp("./hello", "hello", NULL)<0)
        {
            printf("execlp error\n");
        }
    }
}
```

8.以下程序在父进程和子进程之间创建了一个管道，然后建立它们之间的通信，实现父进程向子进程写数据的功能。说明标号所在行代码的功能。(程序题)

考察方式：根据语句写出注释

```
/*pipe.c*/
int main()
{
    int fd[2];
    pid_t pid;
    char buf[100];
    char data[] = "hello world!";
    int num;
    memset(buf, 0, sizeof(buf)); // 将数据缓冲区清零
    //STEP 1. create pipe
    if (pipe(fd)<0)
    {
        printf("pipe create error!\n");
        return -1;
    }
    if ((pid = fork()) == 0)
    {
        printf("\n");
        close(fd[1]); //关闭父进程写描述符
        sleep(2);
        if ((num = read(fd[0], buf, 100))>0) //子进程读取管道内容
        {
            printf("%d bytes read from the pipe!\n", num);
            printf("That is \"%s\"\n", buf);
```

```
        }
        close(fd[0]); //关闭子进程读描述符
        exit(0);
    }
    else if (pid>0)
    {
        close(fd[0]); //关闭子进程读描述符
        if (write(fd[1], data, strlen(data)) == -1);  //父进程将数据写入缓冲区
        {
            printf("parent write data success!\n");
        }
        close(fd[1]); //关闭父进程写描述符
        sleep(3);
        waitpid(pid, NULL, 0); //等待子进程
        exit(0);
    }
```

9.通过命名管道实现从一个进程(发送端)读取当前目录下指定文件(Data.txt)的大小，将该大小发送至另外一个进程(接收端)，并打印。(程序题)

考察方式：根据注释完成程序中相应的语句

比如：

(1) lseek(data_fd, 0, SEEK_END); //通过 lseek 获取文件大小

(2) write(pipe_fd, &data_size, sizeof(float)); //将文件大小发送至管道

```
/*fifo_write_block*/
int main()
{
    const char *fifo_name = "/tmp/my_fifo";
    int pipe_fd = -1;
    int data_fd = -1;
    int res = 0;
    int bytes_sent = 0;
    char buffer[PIPE_BUF + 1];    //where is PIPE_BUF? Do you known?
    if (access(fifo_name, F_OK) == -1) //F_OK is used to determine whether the FIFO exists?
    {
        //step 1. Create the FIFO if it does not exist!
        res = mkfifo(fifo_name, 0777);
        if (res != 0)
        {
            fprintf(stderr, "Could not create fifo %s\n", fifo_name);
            exit(1);
        }
    }
```

```
    //step 2. Open the FIFO with O_WRONLY
    pipe_fd = open(fifo_name, O_WRONLY);
    printf("Process %d opening FIFO %d with O_WRONLY\n", getpid(), pipe_fd);
    data_fd = open("Data.txt", O_RDONLY);
    float data_size = lseek(data_fd, 0, SEEK_END); //通过lseek获取文件大小
    lseek(data_fd, 0, SEEK_SET);
    printf("total data size : %.2f \n", data_size);
    if (pipe_fd != -1)
    {
        int bytes_read = 0;
        write(pipe_fd, &data_size, sizeof(float)); //将文件大小发送至管道
        //read data from "Data.txt", and save in buffer
        bytes_read = read(data_fd, buffer, PIPE_BUF);
        buffer[bytes_read] = '\0';
        while (bytes_read > 0)
        {
            //step 3. Write data form the buffer to FIFO
            res = write(pipe_fd, buffer, bytes_read);
            if (res == -1)
            {
                fprintf(stderr, "Write error on pipe\n");
                exit(1);
            }
            bytes_sent += res;
            bytes_read = read(data_fd, buffer, PIPE_BUF);
            buffer[bytes_read] = '\0';  //why assigning it with '\0'.
            //transfer, please wait for a moment
            //Completion Rate: complete = bytes_sent/data_size
            float bytes_sent_temp = bytes_sent;
            float rate = 100 * (bytes_sent_temp / data_size); //计算传输完成率
            printf("send complete %.2f%% \n", rate);
            sleep(1);
        }
        //step 4. colse the FIFO
        close(pipe_fd);
        close(data_fd);
    }
    else
        exit(1);
    printf("Process %d finished, %d bytes send\n", getpid(), bytes_sent);
    exit(0);
}
/*fifo_block_r.c*/
int main()
```

```
{
    const char *fifo_name = "/tmp/my_fifo";
    int pipe_fd = -1;
    int data_fd = -1;
    int res = 0;
    char buffer[PIPE_BUF + 1];    //where is PIPE_BUF? Do you known?
    memset(buffer, '\0', sizeof(buffer));
    int bytes_read = 0;
    int bytes_write = 0;
    float data_size;
    //step 1. Open the FIFO with O_RDONLY
    pipe_fd = open(fifo_name, O_RDONLY);
    printf("Process %d opening FIFO %d with O_RDONLY\n", getpid(), pipe_fd);
    data_fd = open("DataFormFIFO.txt", O_WRONLY | O_CREAT | O_TRUNC, 0666);
    if (pipe_fd != -1)
    {
        read(pipe_fd, &data_size, sizeof(float)); //从管道读取文件大小
        printf("total data size %.2f \n", data_size);
        do
        {
            //step 2. read the data from FIFO
            res = read(pipe_fd, buffer, PIPE_BUF);
            //save the data on "DataFormFIFO.txt"
            bytes_write = write(data_fd, buffer, res);
            bytes_read += res;
            float bytes_read_temp = bytes_read;
            float rate = 100 * (bytes_read_temp / data_size); //计算传输完成率
            printf("receiving now ... receive complete %.2f%% \n ", rate);
        } while (res > 0);
        //step 3. colse the FIFO
        close(pipe_fd);
        close(data_fd);
    }
    else exit(1);
    printf("Process %d finished, %d bytes read\n", getpid(), bytes_read);
    exit(0);
}
```

10. fork 与 vfork 的本质区别？(程序题)

考察方式：正确写出程序的运行结果

fork 的父子进程是同级别的，没有前后限制，而 vfork 保证子进程先运行，在调用 exec 或 exit 之前与父进程数据是共享的，在它调用 exec 或 exit 之后父进程才可能被调度运行。

vfork 函数是通过允许父子进程可访问相同物理内存从而伪装了对进程地址空间的真实拷贝，当子进程需要改变内存中数据时才复制父进程。

```
/*fork.c*/
int main()
{
    pid_t result;
    int var = 50;
    printf("before fork,var=%d\n", var);
    result = fork();
    if (result<0)
        printf("fork fail\n");
    else if (result>0)
    {
        var++;
        printf("This is parent!\n");
    }
    else
    {
        var--;
        printf("This is child!\n");
    }
    printf("after fork var=%d\n", var);
    return 0;
}
运行结果:
before fork,var=50

This is parent!
after fork var=51

This is child!
after fork var=49
/*vfork.c*/
int main()
{
    pid_t pid;
    int a = 50;
    printf("Fork program starting!\n");
    if ((pid = vfork())<0)
    {
        printf("vfork error!\n");
        exit(1);
    }
    else if (pid == 0)
    {
        printf("This is the child process return!\n");
```

```
            sleep(5);
            a++;
            printf("The child process a is %d\n", a);
            exit(0);
        }
        else
        {
            printf("This is the parent process return!\n");
            a++;
            printf("The parent process a is %d\n", a);
        }
        printf("BYE BYE!\n");
        exit(0);
    }
```

运行结果：

```
Fork program starting!
This is the child process return!
The child process a is 51
This is the parent process return!
The parent process a is 52
BYE BYE!
```

6. 嵌入式 Linux 网络应用开发

1. 在嵌入式 Linux 网络编程应用中，端口号的取值范围？

0-65535

2. socket 类型有哪三类，并进行简单描述。（简答题）

流式 Socket(Stream Socket)：基于 TCP，提供可靠的字节流传输

```
int s = socket (PF_INET, SOCK_STREAM, 0);
```

数据报 Socket(Datagram Socket)：基于 UDP，提供不可靠的报文传输

```
int s = socket (PF_INET, SOCK_DGRAM, 0);
```

Raw Socket：基于 IP，允许用户直接对 IP 操作

```
int s = socket (PF_INET, SOCKET_RAW, protocol);
```

3. 简单说明 TCP 和 UDP 协议有什么区别？（简答题）

传输控制协议：TCP 提供的是面向连接、可靠的字节流服务。当客户和服务器彼此交换数据前，必须先在双方之间建立一个 TCP 连接，之后才能传输数据。TCP 提供超时重发，丢弃重复数据，检验数据，流量控制等功能，保证数据能从一端传到另一端。

用户数据报协议：UDP 不提供可靠性，它只是把应用程序传给 IP 层的数据报发送出去，但是并不能保证它们能到达目的地。由于 UDP 在传输数据报前不用在客户和服务器之间建立一个连接，且没有超时重发等机制，故而传输速度很快。

7. 嵌入式设备驱动程序设计

1. 嵌入式系统中，设备类型分为哪三种？

字符设备文件：指不需要缓冲就能直接读写的设备。

块设备文件：指需要缓冲区来进行数据过渡的设备。

网络接口设备文件：指网络设备访问接口，如网卡等。

2.常见的驱动中，有哪些驱动作为内核模块动态加载？哪些不能？

作为内核模块动态加载：声卡驱动和网卡驱动等。

Linux 最基础的驱动：CPU、PCI 总线、TCP/IP 协议、APM（高级电源管理）、VFS 等驱动程序则编译在内核文件中，非动态加载。

3.建立设备接入点的命令？

创建设备进入点

命令格式：mknod　/dev/xxx　type　major　minor

（设备接入点　文件类型　主设备号　次设备号）

查看设备进入点

命令格式：ls　-l　/dev　|grep 设备名

删除设备进入点

root@ubuntu:/# rm /dev/demo_drv

4.卸载、加载、查看驱动程序的命令？

卸载：rmmod <设备驱动程序.0>

加载：insmod <设备驱动程序.0>

查看：lsmod -l

5.Linux 内核的功能由哪几部分组成？（简答题）

Linux 内核主要由五个部分组成：进程管理，内存管理，文件管理，设备控制，网络功能。

8.设备驱动程序的作用是什么？（简答题）

设备驱动可以理解为操作系统的一部分，它的作用就是让操作系统能正确识别和使用设备。

9.简要说明用户应用程序与 Linux 设备驱动程序之间的区别？（简答题）

（1）应用程序一般都有一个 main 函数，从头到尾执行一个任务。设备驱动程序没有 main 函数，它通过用户空间的 insmod 命令将设备驱动程序的初始化函数加入到内核中，在内核空间执行驱动程序的初始化函数，完成驱动程序的初始化和注册，之后驱动便停止下来，一直等待被应用程序调用。

（2）设备驱动程序运行在内核空间，用户应用程序则运行在用户空间。

10. 字符设备驱动程序开发的流程主要是什么？（简述题）

字符设备驱动程序开发流程的步骤如下：

（1）创建设备进入点

（2）编写字符设备驱动程序

（3）编写 Makefile 文件，编译设备驱动程序

（4）编写用户应用程序，并编译用户程序

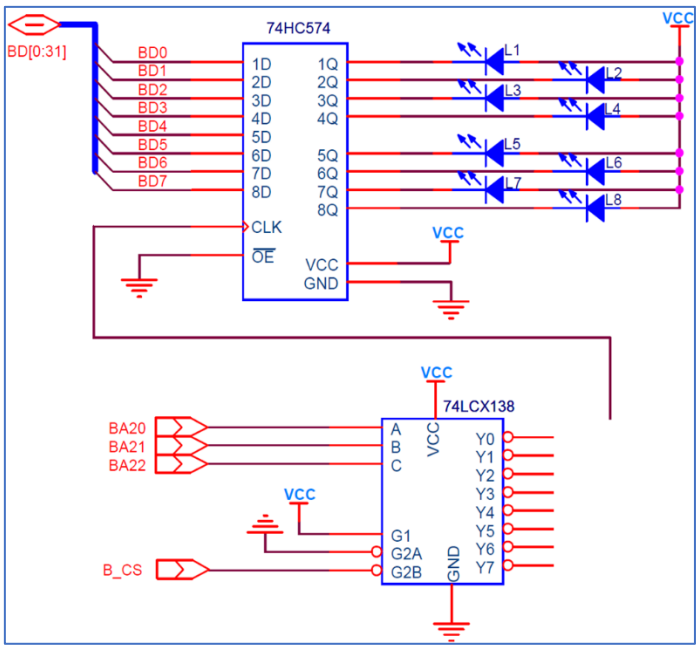
（5）加载设备驱动程序

（6）运行用户应用程序

11.已知：目标板 LED 显示接口电路如图所示，LED 将 74HC574 的时钟信号输入端(CLK)作为片选信号，该片选信号由目标板系统的 PXA270 CPU 的地址信号 BA22 ～ BA20 通过 3-8 译码器 74LCX138 产生。根据 PXA270 芯片手册，得知当 B_CS3 片选信号有效时(低电平有效)，PXA270 芯片控制的内存地址空间范围为 0x0C000000~0x10000000（64Mbyte）。

试卷给出了实现对 LED（8 个发光二极管）控制的驱动程序代码与测试应用程序代码，该驱动模块的设备名为“demo_led”，主设备号为 98。

请根据注释填空。




```
/*led_driver.c*/
#define DEVICE_NAME  "xsb_Led"          //设备名
#define SEG_LED      0x10500000            //内存地址
#define Led_MAJOR    60                    //主设备名
static char Led_Buff = 0x0;
unsigned long * Led_Address;          //申明一个内核空间中的虚拟地址
static void Updateled(void)
{
    writeb(Led_Buff, Led_Address);    //将内核空间数据写入到虚拟地址
    return;
}
static ssize_t Led_write(struct file *file, const char *buffer, size_t count,
loff_t * ppos)
{
    if (count != 1)
    {
        printk(KERN_EMERG "the count of input is not one!!");
        return 0;
    }
    copy_from_user(&Led_Buff, buffer, 1); //将用户空间数据拷贝到内核空间
    Updateled();
    return 1;
}
static int Led_open(struct inode *inode, struct file *filp)
{
    return 0;
}
static int Led_release(struct inode *inode, struct file *filp)
```

```
{
    return 0;
}
static struct file_operations Led_fops = {
    open:    Led_open,
    write : Led_write,
    release : Led_release,
    owner : THIS_MODULE,
};//向系统注册操作功能，申明接口函数
static int __init Led_init(void)
{
    int ret;
    Led_Address = ioremap(SEG_LED, 4); //为I/O内存区域分配虚拟地址

    ret = register_chrdev(Led_MAJOR, DEVICE_NAME, &Led_fops); //注册
    if (ret < 0) {
        printk(DEVICE_NAME " can't get major number\n");
        return ret;
    }
    else
        return 0;
}
static void __exit Led_exit(void)
{
    iounmap(Led_Address);
    unregister_chrdev(Led_MAJOR, DEVICE_NAME);
}
module_init(Led_init); //注册
module_exit(Led_exit); //注销
MODULE_LICENSE("GPL");
MODULE_AUTHOR("dj.z");
MODULE_DESCRIPTION("This is a Led driver demo");
/*Makefile */
$(warning KERNELRELEASE = $(KERNELRELEASE))
ifeq($(KERNELRELEASE), )
KERNELDIR ? = /usr/src/linux - 2.6.22.10//设置内核源代码路径
PWD : = $(shell pwd)
  modules :
          $(MAKE) - C $(KERNELDIR) M = $(PWD) modules
      modules_install :
$(MAKE) - C $(KERNELDIR) M = $(PWD) modules_install
clean :
rm - rf *.o *~core.depend .*.cmd *.ko *.mod.c.tmp_versions Module* modules*
.PHONY : modules modules_install clean
```

```
else
obj - m : = led.o//指定驱动模块名
endif
/*test.c*/
#define LED_DEV      "/dev/xsb_Led"//应用程序需要打开的设备文件名
bool bstop = false;
int fd;
void *Led_Right_Shift(void *data)
{
    unsigned char ledvalue = 0x01;
    unsigned char tmp;
    int i = 0;
    while (1)
    {
        for (i = 0; i < 8; i++)
        {
            if (bstop)
            {
                bstop = false;
                return;
            }
            tmp = ~ledvalue;
            write(fd, &tmp, 1); //将tmp写入设备文件
            sleep(1);
            if (i < 7)
                ledvalue = ledvalue << 1; //实现右移
            else
                ledvalue = 0x01;
        }
    }
}
void *Led_Left_Shift(void *data)
{
    unsigned char ledvalue = 0x80;
    unsigned char tmp;
    int i = 0;
    while (1)
    {
        for (i = 0; i < 8; i++)
        {
            if (bstop)
            {
                bstop = false;
                return;
```

```
            }
            tmp = ~ledvalue;
            write(fd, &tmp, 1);
            sleep(1);
            if (i < 7)
                ledvalue = ledvalue >> 1; //实现左移
            else
                ledvalue = 0x80;
        }
    }
}

int main(int argc, char **argv)
{
    fd = -1;
    pthread_t th_shift = 0;
    char ch = 0x00;
    display_menu();
    void *retval;
    while (1)
    {
        ch = getchar();
        switch (ch)
        {
        case '0':
            if (fd > 0)
                printf("##Led Device has been open##%d \n", fd);
            else
            {
                fd = open(LED_DEV, O_RDWR); //打开设备文件
                if (fd < 0)
                    printf("####LED Device open Fail####\n");
                else
                    printf("####LED Device open Success####%d \n", fd);
            }
            display_menu();
            break;
        case '1':
            if (fd)
            {
                if (th_shift)
                {
                    bstop = true;
                    pthread_join(th_shift, &retval); //等待线程结束
```

```
                }
                bstop = false;
                pthread_create(&th_shift, NULL, Led_Right_Shift, NULL); //右移显
示
            }
            else
                printf("The device is not open!!");
            display_menu();
            break;
        case '2':
            if (fd)
            {
                if (th_shift)
                {
                    bstop = true;
                    pthread_join(th_shift, &retval); //等待线程结束
                }
                bstop = false;
                pthread_create(&th_shift, NULL, Led_Left_Shift, NULL); //左移显
示
            }
            else
                printf("The device is not open!!");
            display_menu();
            break;
        case 't':
        case 'T':
            bstop = true; //停止显示
            display_menu();
            break;
        case 'c':
        case 'C':
            if (fd)
                close(fd); //关闭设备文件
            display_menu();
            bstop = false;
            break;
        case 'x':
        case 'X':
            exit(1);
        default:
            break;
        }
    }
```

```
    return(0);
}
```

8. 课后作业考察题

1. 通过虚拟机的方式安装 linux 系统。 （简述题）

一、准备工作

1、下载安装 VMware 并安装。

2、下载 Linux 系统镜像。

二、新建虚拟机

1、打开 VMware，新建一个虚拟机。

2、选择“典型（推荐）(T)”，点“下一步”。

3、选择“稍后安装操作系统”，点“下一步”。

4、选择系统版本，点“下一步”。

5、设置“虚拟机名称”跟虚拟机的存放“位置”，点“下一步”。

6、设置“最大磁盘大小”，选择“将虚拟磁盘存储为单个文件”，点“下一步”。

三、安装

1、在 CD/DVD 处加载第一步下载的 Linux 系统镜像文件。

2、点击虚拟机的电源键，然后按照步骤提示进行安装，完成后即可使用虚拟机了。