

**学生实验报告册**

|  |  |
| --- | --- |
| 课程名称： | 控制系统开发技术(4)(嵌入式系统) |
| 学 院： | 自动化学院 |
| 专业班级： | 08052001 |
| 姓 名： | 张聿璁 |
| 学 号： | 2020212636 |
| 指导教师： | 翁明江 |
| 成 绩： |  |
| 学年学期： | 2022-2023 学年 🞏春■秋学期 |

**重庆邮电大学教务处制**

|  |  |  |  |
| --- | --- | --- | --- |
| **实验项目名称** | **嵌入式Linux程序开发综合实验** | | |
| **实验地点** | **仙桃314** | **实验时间** | **12.15** |
| **实验指导教师** | 翁明江 | **成绩** |  |
| 一、实验目的  1、掌握嵌入式Linux开发工具使用方法；  2、熟悉嵌入式Linux应用程序开发方法及技术；  3、熟悉嵌入式Linux驱动程序开发方法；  4、掌握嵌入式软硬件联合调试方法及技巧。 | | | |
| 二、实验内容  基于实验中心提供的NXP imx6ull开发板基础软件代码包，开发led字符设备驱动与led控制应用程序，实现对led灯的开关控制，要求：  1、led设备驱动实现read()、write()接口，应用层通过read()接口可获得led当前亮灭状态，通过write()接口可控制led亮灭。  2、led控制应用程序内创建线程A与B、信号量A’与B’，线程A负责点亮led，线程B负责关闭led；A与B线程运行后分别等待信号量A’与B’到达；当线程A接收到信号量A’时点亮led，等待10秒后向线程B发送信号量B’，随即线程A又继续等待信号量A’；当线程B接收到信号量B’后关闭led，等待10秒后向线程A发送信号量A’，随即线程B又继续等待信号量B’。如此交替往复10次，应用程序安全退出。  3、完成软硬件联合调试。 | | | |
| 三、实验仪器设备、材料  硬件：嵌入式Linux开发板、PC机、USB连接线。  软件：PC 机Linux操作系统、嵌入式Linux交叉编译工具链、超级终端等工具。 | | | |
| 四、设计方案   1. 驱动开发   Linux驱动分为三大类：字符设备驱动、块设备驱动、网络设备驱动。字符设备是Linux驱动中最基本的一类设备驱动，字符设备就是按照字节流进行读写操作的设备。本实验驱动LED，本质上就是对一个IO口进行操作，对相关寄存器进行读写操作，应该归类为字符设备驱动。  Linux将空间分为内核空间和用户空间，Linux内核和驱动程序运行在内核空间，驱动成功加载后会在“/dev”下生成对应文件，位于用户空间的应用程序通过具体的驱动程序来实现对硬件的控制。    应用程序可以通过系统调用的方法从应用空间“陷入”到内核空间，这样才能访问底层驱动的资源并进行操作。在Linux系统中，系统调用为C库的一部分，当应用程序调用open、write等函数时，本质上通过调用C库中的open、write函数来进行内核的系统调用，最终实现驱动的open、write函数调用。    因此，驱动中应该有与应用程序对应的函数，驱动函数的定义已经在内核文件中定义，我们只需要引用头文件并调用即可。  Linux内核启动的时候会初始化MMU。MMU实现了虚拟空间到物理空间的映射，也就是地址映射。对于实验中使用的512MB的DDR3物理内存，经过MMU的映射就可以映射到4GB的空间。这就导致了我们并不能像单片机开发时那样，直接对一个物理地址进行读写操作，需要实现物理内存到虚拟内存的转换。这里我们可以通过调用ioremap和iounmap来实现。  对于控制LED来说，只需要拉高或者拉低IO口，所以我们只需要重点编写write函数和init函数。参考正点原子提供的资料，编写了以下函数。    Led初始化函数，通过ioremap获取地址映射，得到相关寄存器的地址。类似于单片机开发，IO口的配置的流程，顺序为使能时钟，设置为复用功能，并且注册为字符驱动。    LED\_write函数主要通过另外一个封装好的led\_switch函数进行对寄存器的操作。    Read函数在后续应用开发并没有用到，其效果就是读取寄存器GPIO1\_DR的值。当为0时，代表LED熄灭，为1时，代表LED点亮。  在Ubuntu 20.04.2环境下编译，产生了后缀为.ko的模块文件。    通过MobaXterm的Z-Modem方式将文件传送到开发板上，并进行模块挂载。通过lsmod和cat /proc/devices命令来查看，发现模块已经成功挂载。至此我们完成了驱动的开发。     1. 应用程序设计   在Linux中，线程是一个进程内部的控制序列，线程在进程内部运行，本质是在进程地址空间内运行。因为描述线程的控制块和表述进程的控制块在Linux是类似的，因此Linux并没有为线程重新设计数据，依然使用task\_struct来模拟线程。在用户层，linux提供了pthread线程库，通过在用户层模拟实现了一套线程相关的接口。  线程中使用的信号量是一种特殊的变量，他可以被增加或者减少，但对其的关键访问是通过原子操作来实现的。只有0和1两种取值的信号量叫做二值信号量，信号量的函数都以sem\_开头，常用的有以下四个：   1. int sem\_init(sem\_t \*sem, int pshared， unsigned int value);   这个函数是用来创建信号量的，因为实验要求两个信号量应该互不干扰，所以应该申请为局部信号量，因此pshared应该为NULL，否则信号量将会在两个进程间共享，value为初始值。   1. int sem\_wait(sem\_t \*sem);   此函数实现了以原子操作的方式对信号量减1。信号量为1代表后续线程可以通过，信号量为0时，将线程阻塞。   1. int sem\_post(sem\_t \*sem);   此函数实现了以原子操作的方式对信号量加1。   1. int sem\_destroy(sem\_t \*sem);   此函数实现了对sem\_init创建的信号量的回收。  借助信号量，我们可以实现进程的同步。但是我们需要依靠pthread库来实现多线程的基本操作。在Linux下，与本实验多线程相关的API函数接口有：   1. int pthread\_create(pthread\_t \*thread, const pthread\_attr\_t \*attr, void \*(\*start\_routine)(void\*), void \*arg); 2. int pthread\_join(pthread\_t thread, void \*\*retval); 3. void pthread\_exit（void \*retval）;   pthread\_create函数用于创建线程，pthread\_join函数用于在主线程中结束线程并且确认线程退出的状态，pthread\_exit函数用于在子线程中结束本进程。  了解了以上的函数，就可以进行App函数的编写，但是在实际操作中发现，还需要注意两点问题：   1. 开发板的出厂固件中将led设置为心跳灯，我们应该通过以下命令将心跳灯关闭：echo none > /sys/class/leds/sys-led/trigger 2. 只在App.c中加入<pthread.h>函数并不能通过gcc编译，我们需要将其链接libpthread.so这个库。同时，需要将gcc的编译标准改为c99。以上两点均需要在编译时注意。最终的编译命令应该为：     分析实验要求，利用信号量和pthread库，我们可以画出以下的流程图：    在初始化信号量时，我将A’设为1，B’为0，这样就可以使线程开始运行时，A线程先工作，B线程阻塞。随后A线程阻塞，B线程工作，使用简单的for循环来实现10个来回。往返10个来回后，线程通过pthread\_exit退出。在主线程main中通过pthread\_join中确保线程已经安全退出，然后使用sem\_destroy将信号量回收。在主线程中，需要申请2个pthread\_t结构体，为pthread\_create函数提供入口参数。pthread\_t 是一种用于表示线程的数据类型，每一个 pthread\_t 类型的变量都可以表示一个线程。同时，信号量的申请也需要在主函数中进行操作，通过sem\_init函数，对两个信号量进行赋值。    将代码编译生成测试App，通过串口发送给IMX6ULL，通过chmod 777 myledTest命令给代码执行命令，在终端上输入执行命令，运行通过。 | | | |
| 五、实验结果及分析（或设计总结）      经过10个来回，代码打印出信息并退出。 | | | |

附录(关键源程序)

|  |
| --- |
| //以下为App应用程序  #include "stdio.h"  #include "unistd.h"  #include "sys/types.h"  #include "sys/stat.h"  #include "fcntl.h"  #include "stdlib.h"  #include "string.h"  #include "pthread.h"  #include "semaphore.h"  #define LEDOFF 0  #define LEDON 1  sem\_t number\_A,number\_B;  int Led\_Status\_Switch(char status)  {  int fd, retvalue;  char \*filename;  unsigned char databuf[1];    filename = "/dev/led";  /\* 打开led驱动 \*/  fd = open(filename, O\_RDWR);  if(fd < 0){  printf("file %s open failed!\r\n",filename);  return -1;  }  databuf[0] = status; /\* 要执行的操作：打开或关闭 \*/  /\* 向/dev/led文件写入数据 \*/  retvalue = write(fd, databuf, sizeof(databuf));  if(retvalue < 0){  printf("LED Control Failed!\r\n");  close(fd);  return -1;  }  if(status)  printf("LED ON\n");  else  printf("LED OFF\n");  retvalue = close(fd); /\* 关闭文件 \*/  if(retvalue < 0){  printf("file %s close failed!\r\n",filename);  return -1;  }  return 0;  }  void \*thread\_A(void \*arg)  {  for(int i = 0; i <10;i++)  {  sem\_wait(&number\_A);  printf("thread\_A work\n");  Led\_Status\_Switch(LEDON);  sleep(2);  sem\_post(&number\_B);  printf("thread\_A stop\n");  }  printf("thread\_A exited\n");  pthread\_exit(NULL);  return NULL;  }  void \*thread\_B(void \*arg)  {  for(int i = 0; i <10;i++)  {  sem\_wait(&number\_B);  printf("thread\_B work\n");  Led\_Status\_Switch(LEDOFF);  sleep(2);  sem\_post(&number\_A);  printf("thread\_B stop\n");  printf("count=%d\n",i);  }  printf("thread\_B exited\n");  pthread\_exit(NULL);  return NULL;  }  int main()  {  pthread\_t pid,cid;  sem\_init(&number\_A,0,1);  sem\_init(&number\_B,0,0);  pthread\_create(&pid,NULL,thread\_A,NULL);  pthread\_create(&cid,NULL,thread\_B,NULL);  pthread\_join(pid,NULL);  pthread\_join(cid,NULL);  sem\_destroy(&number\_A);  sem\_destroy(&number\_B);  pthread\_exit(NULL);  return 0;  }  //以下为LED驱动程序  #include <linux/types.h>  #include <linux/kernel.h>  #include <linux/delay.h>  #include <linux/ide.h>  #include <linux/init.h>  #include <linux/module.h>  #include <linux/errno.h>  #include <linux/gpio.h>  #include <asm/mach/map.h>  #include <asm/uaccess.h>  #include <asm/io.h>  #define LED\_MAJOR       200 */\* 主设备号 \*/*  #define LED\_NAME        "led" */\* 设备名字 \*/*  #define LEDOFF  0 */\* 关灯 \*/*  #define LEDON   1 */\* 开灯 \*/*    */\* 寄存器物理地址 \*/*  #define CCM\_CCGR1\_BASE              (0X020C406C)  #define SW\_MUX\_GPIO1\_IO03\_BASE      (0X020E0068)  #define SW\_PAD\_GPIO1\_IO03\_BASE      (0X020E02F4)  #define GPIO1\_DR\_BASE               (0X0209C000)  #define GPIO1\_GDIR\_BASE             (0X0209C004)  */\* 映射后的寄存器虚拟地址指针 \*/*  *static* *void* \_\_iomem \*IMX6U\_CCM\_CCGR1;  *static* *void* \_\_iomem \*SW\_MUX\_GPIO1\_IO03;  *static* *void* \_\_iomem \*SW\_PAD\_GPIO1\_IO03;  *static* *void* \_\_iomem \*GPIO1\_DR;  *static* *void* \_\_iomem \*GPIO1\_GDIR;  *void* led\_switch(u8 *sta*)  {      u32 val = 0;      if(*sta* == LEDON) {          val = readl(GPIO1\_DR);          val &= ~(1 << 3);          writel(val, GPIO1\_DR);      }else if(*sta* == LEDOFF) {          val = readl(GPIO1\_DR);          val|= (1 << 3);          writel(val, GPIO1\_DR);      }  }  *static* *int* led\_open(*struct* inode \**inode*, *struct* file \**filp*)  {      return 0;  }  *static* *ssize\_t* led\_read(*struct* file \**filp*, *char* *\_\_user* \**buf*, *size\_t* *cnt*, *loff\_t* \**offt*)  {  //因为应用程序并未使用read，所以此处留成空      return 0;  }  *static* *ssize\_t* led\_write(*struct* file \**filp*, *const* *char* *\_\_user* \**buf*, *size\_t* *cnt*, *loff\_t* \**offt*)  {  *int* retvalue;  *unsigned* *char* databuf[1];  *unsigned* *char* ledstat;      retvalue = copy\_from\_user(databuf, buf, cnt);      if(retvalue < 0) {          printk("kernel write failed!\r\n");          return -EFAULT;      }      ledstat = databuf[0]; */\* 获取状态值 \*/*      if(ledstat == LEDON) {          led\_switch(LEDON); */\* 打开LED灯 \*/*      } else if(ledstat == LEDOFF) {          led\_switch(LEDOFF); */\* 关闭LED灯 \*/*      }      return 0;  }  *static* *int* led\_release(*struct* inode \**inode*, *struct* file \**filp*)  {      return 0;  }  */\* 设备操作函数 \*/*  *static* *struct* file\_operations led\_fops = {      .owner = THIS\_MODULE,      .open = led\_open,      .read = led\_read,      .write = led\_write,      .release =  led\_release,  };  *static* *int* \_\_init led\_init(*void*)  {  *int* retvalue = 0;      u32 val = 0;      IMX6U\_CCM\_CCGR1 = ioremap(CCM\_CCGR1\_BASE, 4);      SW\_MUX\_GPIO1\_IO03 = ioremap(SW\_MUX\_GPIO1\_IO03\_BASE, 4);      SW\_PAD\_GPIO1\_IO03 = ioremap(SW\_PAD\_GPIO1\_IO03\_BASE, 4);      GPIO1\_DR = ioremap(GPIO1\_DR\_BASE, 4);      GPIO1\_GDIR = ioremap(GPIO1\_GDIR\_BASE, 4);      val = readl(IMX6U\_CCM\_CCGR1);      val &= ~(3 << 26); */\* 清楚以前的设置 \*/*      val |= (3 << 26); */\* 设置新值 \*/*      writel(val, IMX6U\_CCM\_CCGR1);      writel(5, SW\_MUX\_GPIO1\_IO03);      writel(0x10B0, SW\_PAD\_GPIO1\_IO03);      val = readl(GPIO1\_GDIR);      val &= ~(1 << 3); */\* 清除以前的设置 \*/*      val |= (1 << 3); */\* 设置为输出 \*/*      writel(val, GPIO1\_GDIR);      val = readl(GPIO1\_DR);      val |= (1 << 3);      writel(val, GPIO1\_DR);      retvalue = register\_chrdev(LED\_MAJOR, LED\_NAME, &led\_fops);      if(retvalue < 0){          printk("register chrdev failed!\r\n");          return -EIO;      }      return 0;  }  *static* *void* \_\_exit led\_exit(*void*)  {  */\* 取消映射 \*/*      iounmap(IMX6U\_CCM\_CCGR1);      iounmap(SW\_MUX\_GPIO1\_IO03);      iounmap(SW\_PAD\_GPIO1\_IO03);      iounmap(GPIO1\_DR);      iounmap(GPIO1\_GDIR);      unregister\_chrdev(LED\_MAJOR, LED\_NAME);  }  module\_init(led\_init);  module\_exit(*led\_exit*);  MODULE\_LICENSE("GPL");  MODULE\_AUTHOR("zhangyucong"); |