Linux操作系统内核和设备文件对话 PDF转换可能丢失图片或格式，建议阅读原文

https://www.100test.com/kao_ti2020/144/2021_2022_Linux_E6_93_8D_E4_BD_c103_144794.htm 设备文件是用来代表物理设备的。多数物理设备是用来进行输出或输入的，所以必须由某种机制使得内核中的设备驱动从进程中得到输出送给设备。这可以通过打开输出设备文件并且写入做到，就想写入一个普通文件。在下面的例子里，这由device_write实现。 这不是总能奏效的。设想你与一个连向modem的串口（技是你有一个内猫，从CPU看来它也是作为一个串口实现，所以你不需要认为这个设想太困难）。最自然要做的事情就是使用设备文件把内容写到modem上（无论用modem命令还是电话线）或者从modem读信息（同样可以从modem命令回答或者通过电话线）。但是这留下的问题是当你需要和串口本身对话的时候需要怎样做？比如发送数据发送和接收的速率。 回答是Unix使用一个叫做ioctl(input output control的简写)的特殊函数。每个设备都有自己的ioctl命令，这个命令可以是ioctl读的，也可以是写的，也可以是两者都是或都不是。Ioctl函数由三个参数调用：适当设备的描述子，ioctl数，和一个长整型参数，可以赋予一个角色用来传递任何东西。 Ioctl数对设备主码、ioctl类型、编码、和参数的类型进行编码。Ioctl数通常在头文件由一个宏调用（_IO，_IOR，_IOW或_IOWR决定于类型）。这个头文件必须包含在使用ioctl（所以它们可以产生正确的ioctls）程序和内核模块（所以它可以理解）中。在下面的例子里，这个头文件是chardev.h，使用它的程序是ioctl.c。 如果你希望在你自己的内核模块中使用ioctls，最好

去接受一分正式的ioctl职位，这样你就可以得到别人的ioctls，或者他们得到你，你就可以知道哪里出了错误。如果想得到更多的信息，到documentation/ioctl-number.txt中查看内核源文件树。 ex chardev.c /* chardev.c * * Create an input/output character device */ /* Copyright (C) 1998-99 by Ori Pomerantz */ /* The necessary header files */ /* Standard in kernel modules */ #include /* Were doing kernel work */ #include /* Specifically, a module */ /* Deal with CONFIG_MODVERSIONS */ #if CONFIG_MODVERSIONS==1 #define MODVERSIONS #include #endif /* For character devices */ /* The character device definitions are here */ #include /* A wrapper which does next to nothing at * at present, but may help for compatibility * with future versions of Linux */ #include /* Our own ioctl numbers */ #include "chardev.h" /* In 2.2.3 /usr/include/linux/version.h includes a * macro for this, but 2.0.35 doesnt - so I add it * here if necessary. */ #ifndef KERNEL_VERSION #define KERNEL_VERSION(a,b,c) ((a)*65536 (b)*256 (c)) #endif #if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0) #include /* for get_user and put_user */ #endif #define SUCCESS 0 /* Device Declarations ******************************** */ /* The name for our device, as it will appear in * /proc/devices */ #define DEVICE_NAME "char_dev" /* The maximum length of the message for the device */ #define BUF_LEN 80 /* Is the device open right now? Used to prevent * concurent access into the same device */ static int Device_Open = 0. /* The message the device will give when asked */ static char Message[BUF_LEN]. /* How far did the process reading the message get? * Useful if the message is larger than the size of the * buffer we get to fill in device_read. */ static char *Message_Ptr. /* This function is called whenever a process attempts * to open the device file */ static int device_open(struct inode *inode, struct file *file) { #ifdef DEBUG printk ("device_open(%p)\n", file). #endif /* We dont want to talk to two processes at the * same time */ if (Device_Open) return -EBUSY. /* If this was a process, we would have had to be * more careful here, because one process might have * checked Device_Open right before the other one * tried to increment it. However, were in the * kernel, so were protected against context switches. * * This is NOT the right attitude to take, because we * might be running on an SMP box, but well deal with * SMP in a later chapter. */ Device_Open . /* Initialize the message */ Message_Ptr = Message. MOD_INC_USE_COUNT. return SUCCESS. } /* This function is called when a process closes the * device file. It doesnt have a return value because * it cannot fail. Regardless of what else happens, you * should always be able to close a device (in 2.0, a 2.2 * device file could be impossible to close). */ #if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0) static int device_release(struct inode *inode, struct file *file) #else static void device_release(struct inode *inode, struct file *file) #endif { #ifdef DEBUG printk ("device_release(%p,%p)\n", inode, file). #endif /* Were now ready for our next caller */ Device_Open --. MOD_DEC_USE_COUNT. #if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0) return 0. #endif }/* This function is called whenever a process which * has already opened the device file attempts to * read from it. */ #if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0) static ssize_t device_read(struct file *file, char *buffer, /* The buffer to fill with the data */ size_t length, /* The length of the buffer */ loff_t *offset) /* offset to the file */ #else static int device_read(struct inode *inode, struct file *file, char *buffer, /* The buffer to fill with the data */ int length) /* The length of the buffer * (mustnt write beyond that!) */ #endif { /* Number of bytes actually written to the buffer */ int bytes_read = 0. #ifdef DEBUG printk("device_read(%p,%p,%d)\n", file, buffer, length). #endif /* If were at the end of the message, return 0 * (which signifies end of file) */ if (*Message_Ptr == 0) return 0. /* Actually put the data into the buffer */ while (length amp. *Message_Ptr) { /* Because the buffer is in the user data segment, * not the kernel data segment, assignment wouldnt * work. Instead, we have to use put_user which * copies data from the kernel data segment to the * user data segment. */ put_user(*(Message_Ptr), buffer). length --. bytes_read . } #ifdef DEBUG printk ("Read %d bytes, %d left\n", bytes_read, length). #endif /* Read functions are supposed to return the number * of bytes actually inserted into the buffer */ return bytes_read. }/* This function is called when somebody tries to * write into our device file. */ #if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0) static ssize_t device_write(struct file *file, const char *buffer, size_t length, loff_t *offset) #else static int device_write(struct inode *inode, struct file *file, const char *buffer, int length) #endif { int i. #ifdef DEBUG printk ("device_write(%p,%s,%d)", file, buffer, length). #endif for(i=0. i #if

LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0) get_user(Message, buffer i). #else Message = get_user(buffer i). #endif Message_Ptr = Message. /* Again, return the number of input characters used */ return i. }/* This function is called whenever a process tries to * do an ioctl on our device file. We get two extra * parameters (additional to the inode and file * structures, which all device functions get): the number * of the ioctl called and the parameter given to the * ioctl function. * * If the ioctl is write or read/write (meaning output * is returned to the calling process), the ioctl call * returns the output of this function. */ int device_ioctl(struct inode *inode, struct file *file, unsigned int ioctl_num,/* The number of the ioctl */ unsigned long ioctl_param) /* The parameter to it */ { int i. char *temp. #if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0) char ch. #endif /* Switch according to the ioctl called */ switch (ioctl_num) { case IOCTL_SET_MSG: /* Receive a pointer to a message (in user space) * and set that to be the devices message. */ /* Get the parameter given to ioctl by the process */ temp = (char *) ioctl_param. /* Find the length of the message */ #if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0) get_user(ch, temp). for (i=0. ch amp. ibr temp) i ,> get_user(ch, temp). #else for (i=0. get_user(temp) amp. ibr temp) i ,> . #endif /* Dont reinvent the wheel - call device_write */ #if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0) device_write(file, (char *) ioctl_param, i, 0). #else device_write(inode, file, (char *) ioctl_param, i). #endif break. case IOCTL_GET_MSG: /* Give the current message to the calling * process - the parameter we got is a pointer, * fill it. */ #if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0) i = device_read(file, (char *) ioctl_param, 99, 0). #else i = device_read(inode, file, (char *) ioctl_param, 99). #endif /* Warning - we assume here the buffer length is * 100. If its less than that we might overflow * the buffer, causing the process to core dump. * * The reason we only allow up to 99 characters is * that the NULL which terminates the string also * needs room. */ /* Put a zero at the end of the buffer, so it * will be properly terminated */ put_user(\, (char *) ioctl_param i). break. case IOCTL_GET_NTH_BYTE: /* This ioctl is both input (ioctl_param) and * output (the return value of this function) */ return Message[ioctl_param]. break. } return SUCCESS. }/* Module Declarations *************************** */ /* This structure will hold the functions to be called * when a process does something to the device we * created. Since a pointer to this structure is kept in * the devices table, it cant be local to * init_module. NULL is for unimplemented functions. */ struct file_operations Fops = { NULL, /* seek */ device_read, device_write, NULL, /* readdir */ NULL, /* 0select */ device_ioctl, /* ioctl */ NULL, /* mmap */ device_open, #if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0) NULL, /* flush */ #endif device_release /* a.k.a. close */ }. /* Initialize the module - Register the character device */ int init_module() { int ret_val. /* Register the character device (atleast try) */ ret_val = module_register_chrdev(MAJOR_NUM, DEVICE_NAME, &amp.Fops). /* Negative values signify an error */ if (ret_val 100Test