The Linux LP (parallel) device driver.

Paul Dwerryhouse 6th December, 1997

This document describes the operation of the Linux parallel device driver verb, found in drivers/char/lp.c, in the Linux kernel tree. It is based on the code found in the 2.1.x series of the Linux kernel.

Contents

1 Introduction

1.1 Initial notes

The following describes the operation of the printer drivers in the Linux 2.1.x kernel. I've taken the code from the 2.1.65, so the code in the current developmental release may have changed somewhat. I've taken up the format of placing my comments within the code, because I feel this is the best method for actually seeing what's going on – to have to skip back and forth from a page of description to a page containing the source code is something I always found annoying.

I put these notes together for a few reasons: firstly, it's one of the best ways that I can make sure that *I* understand what's going on ;), secondly, to get a bit more *content* on the Web, instead of pages of links to even more links, and thirdly, and most importantly, because their is a great need for Linux documentation that pertains to programming, rather than just setup and administration.

I hope that people can find this of some use ;)

1.2 Distribution policy

2 lp_char(char lpchar, int minor, int use_polling);

2.1 General Info

Function: *lp_char* sends a character to the parallel port.

Called from: *lp_write_buf*.

Files: drivers/char/lp.c, include/linux/lp.h

2.2 Code description

The function is called with three parameters: *lpchar*, which is the character that we wish to sent to the parallel device, *minor*, the minor device number for this device, and *use_polling*, which is 1 if we're polling this device, 0 if using an interrupt.

```
static inline int lp_char(char lpchar, int minor, int use_polling)
{
        int status;
        unsigned int wait = 0;
        unsigned long count = 0;
        struct lp_stats *stats;
```

The following do/while loop iterates until the parallel device signals that it is ready, or that it is busy. *r_str* gets the status of the device. A check is made in the middle of the loop to see if scheduling has been requested, and if so, it jumps off to *lp_schedule*. The count of the number of attempts is kept and when this reaches a certain cut off point (*LP_CHAR(minor)*), the loop exits.

```
        do {
                status = r_str(minor);
                count++;
                if (need_resched)
                        lp_schedule (minor);
        } while (((use_polling && !LP_READY(minor, status)) ||
                 (!use_polling && !(status & LP_PBUSY))) && (count < LP_CHAR(minor)));
```

If the number of attempts to get a status of "ready" from the device has reached the cutoff point, or if we're using an interrupt on this device and the device is still registering as ready, then exit *lp_char*.

```
        if (count == LP_CHAR(minor) ||
            (!use_polling && !LP_CAREFUL_READY(minor, status)))
                return 0;
```

Now we're past all the preliminary status checks, we can send the character to the device. *w_dtr* is just a macro that expands, in this case, to *outb(lpchar,LP_BASE(minor))* – ie, output a byte to the device.

```
        w_dtr(minor, lpchar);
```

Now that the character has been sent to the device, we can update the device statistics – here, we just increment the number of characters that have been sent to the device in total:

```
        stats = & LP_STAT(minor);
        stats->chars++;
```

Strobe the parallel port – this involves taking the strobe line high and then bringing it down again, which basically signals to the device on the port that there a character has been sent to the port, which can be read by the device.

```
        /* must wait before taking strobe high, and after taking strobe
           low, according spec.  Some printers need it, others don't. */
        while (wait != LP_WAIT(minor)) /* FIXME: should be a udelay() */
                wait++;
        /* control port takes strobe high */
        w_ctr(minor, LP_PSELECP | LP_PINITP | LP_PSTROBE);
        while (wait)                    /* FIXME: should be a udelay() */
                wait--;
        /* take strobe low */
        w_ctr(minor, LP_PSELECP | LP_PINITP);
```

Now update the statistics, so that we can get an idea of the performance of the device, should we ever need to. After that, exit from this function, returning a successful value.

```
        /* update waittime statistics */
        if (count > stats->maxwait) {
#ifdef LP_DEBUG
                printk(KERN_DEBUG "lp%d success after %d counts.\n", minor, count);
#endif
                stats->maxwait = count;
        }
        count *= 256;
        wait = (count > stats->meanwait) ? count - stats->meanwait :
            stats->meanwait - count;
        stats->meanwait = (255 * stats->meanwait + count + 128) / 256;
        stats->mdev = ((127 * stats->mdev) + wait + 64) / 128;

        return 1;
}
```

3 lp_write_buf(unsigned int minor, const char *buf, int count)

3.1 General info.

Function: *lp_write_buf* copies a buffer of characters to the parallel device.

Called from: *lp_write*.

Files: drivers/char/lp.c, include/linux/lp.h

3.2 Code description

lp_write_buf has three arguments: *minor*, the minor device number of this parallel port, *buf*, the buffer in which the data to be sent to the device is held and *count*, the number of characters to be sent.

```
static inline int lp_write_buf(unsigned int minor, const char *buf, int count)
{
        unsigned long copy_size;
        unsigned long total_bytes_written = 0;
        unsigned long bytes_written;
        struct lp_struct *lp = & lp_table[minor];
        unsigned char status;
```

A couple of reality checks, to begin with. Ensure that the minor number being used isn't higher than the number of parallel devices that the kernel has actually been configured to cope with (*LP_NO*). In the code that this was taken from, *LP_NO* was set to 3, although this is, of course, arbitrary. Then make sure that the device that we have asked for has actually been detected by the kernel at all.

```
        if (minor >= LP_NO)
                return -ENXIO;
        if (lp_table[minor].dev == NULL)
                return -ENXIO;
```

Begin the loop that copies the data from user space to the device. Determine the amount of characters that will be copied from user space in one go - either use *LP_BUFFER_SIZE*, or copy all the characters in one shot, if the number is less than *LP_BUFFER_SIZE*.

```
        do {
                bytes_written = 0;
                copy_size = (count <= LP_BUFFER_SIZE ? count : LP_BUFFER_SIZE);
                copy_from_user(lp->lp_buffer, buf, copy_size);
```

Now send the characters one by one into *lp_char*, checking the return value each time.

```
		while (copy_size) {
			if (lp_char(lp->lp_buffer[bytes_written], minor, must_use_polling(minor))) {
				--copy_size;
				++bytes_written;
				lp_table[minor].runchars++;
```

If *lp_char* returned with an error, get the status from the parallel port, and log an appropriate message.

```
			} else {
				int rc = total_bytes_written + bytes_written;
				if (lp_table[minor].runchars > LP_STAT(minor).maxrun)
					LP_STAT(minor).maxrun = lp_table[minor].runchars;
				status = r_str(minor);
```

Out of paper:

```
				if ((status & LP_POUTPA)) {
					printk(KERN_INFO "lp%d out of paper\n", minor);
					if (LP_F(minor) & LP_ABORT)
						return rc ? rc : -ENOSPC;
					lp_error(minor);
```

Printer offline:

```
				} else if (!(status & LP_PSELECD)) {
					printk(KERN_INFO "lp%d off-line\n", minor);
					if (LP_F(minor) & LP_ABORT)
						return rc ? rc : -EIO;
					lp_error(minor);
```

Some other error:

```
				} else if (!(status & LP_PERRORP)) {
					printk(KERN_ERR "lp%d printer error\n", minor);
					if (LP_F(minor) & LP_ABORT)
						return rc ? rc : -EIO;
					lp_error(minor);
				}
```

Now we sleep for a short while (I'm not entirely sure why here – perhaps someone would kindly let me know). The first case is for where we're using polling. I really don't know what's going on here. Russell Coker (rjc@virtual.net.au) suggested that this might just be for scheduling - allowing the kernel to go do something else at this point.

```
                                LP_STAT(minor).sleeps++;

                                if (must_use_polling(minor)) {
#ifdef LP_DEBUG
                                        printk(KERN_DEBUG "lp%d sleeping at %d characters for %d jiffies\n", minor, lp_table[minor].runchars, LP_TIME(minor));
#endif
                                        lp_table[minor].runchars = 0;
                                        current->state = TASK_INTERRUPTIBLE;
                                        current->timeout = jiffies + LP_TIME(minor);
                                        lp_schedule (minor);
```

And the second is for where we're using interrupts. Again, I'm not too sure what's happening here.

```
                                } else {
                                        cli();
                                        enable_irq(lp->dev->port->irq);
                                        w_ctr(minor, LP_PSELECP|LP_PINITP|LP_PINTEN);
                                        status = r_str(minor);
                                        if ((!(status & LP_PACK) || (status & LP_PBUSY))
                                            && LP_CAREFUL_READY(minor, status)) {
                                                w_ctr(minor, LP_PSELECP | LP_PINITP);
                                                sti();
                                                continue;
                                        }
                                        lp_table[minor].runchars = 0;
                                        current->timeout = jiffies + LP_TIMEOUT_INTERRUPT;
                                        interruptible_sleep_on(& lp->lp_wait_q);

                                        w_ctr(minor, LP_PSELECP | LP_PINITP);
                                        sti();
                                }
```

If a signal is pending, then clean up and leave.

```
                                if (signal_pending(current)) {
                                        if (total_bytes_written + bytes_written)
                                                return total_bytes_written + bytes_written;
                                        else
                                                return -EINTR;
                                }
                        }
                }
```

Update the counters and go back through the loop again, until we've sent every character.

```
                total_bytes_written += bytes_written;
                buf += bytes_written;
                count -= bytes_written;

        } while (count > 0);

        return total_bytes_written;
}
```

4 lp_write(struct file * file, const char * buf, size_t count, loff_t *ppos)

4.1 General info.

Function: *lp_write* interfaces the lp driver to *write()*

Called from: *write* (Base driver operation).

Files: drivers/char/lp.c, include/linux/lp.h

4.2 Code description

lp_write is the main interface between the Linux *write()* operation and the lp code. The arguments are: *file*, the actual file that this routine was called upon (eg, /dev/lp0); *buf*, the data that we wish to write; *count*, the length of the data; and *ppos*(not used, apparently).

```
static ssize_t lp_write(struct file * file, const char * buf, size_t count, loff_t *ppos)
{
```

Work out which inode corresponds to this file, then get the minor device number that this inode is pointing at.

```
        struct inode *inode = file->f_dentry->d_inode;
        unsigned int minor = MINOR(inode->i_rdev);
        int retv;
```

Check how much time the previous call took. Not sure what this is for.

```
        if (jiffies-lp_table[minor].lastcall > LP_TIME(minor))
                lp_table[minor].runchars = 0;

        lp_table[minor].lastcall = jiffies;
```

Claim the parport (as it says ;). The parport code allows one parallel port to be shared by several device drivers (so you can use certain devices all plugged into one port – eg, a ZIP drive with a "through" connector. Claiming a parport prevents any other device drivers from being able to use the device while this driver is.

```
        /* Claim Parport or sleep until it becomes available
         * (see lp_wakeup() for details)
         */
        lp_parport_claim (minor);
```

Write the contents of the buffer to the device, and then release the parport.

```
        retv = lp_write_buf(minor, buf, count);

        lp_parport_release (minor);
        return retv;
}
```

5 lp_open(struct inode * inode, struct file * file)

5.1 General info.

Function: *lp_open* opens a parallel device for data.

Called from: *open* (Base driver operation).

Files: drivers/char/lp.c, include/linux/lp.h

5.2 Code description

The parameters that *lp_open* takes are: *inode*, the inode to which the called device corresponds; and *file*, the file to which the called device belongs.

```
static int lp_open(struct inode * inode, struct file * file)
{
```

Again, the initial reality checks(TM). Get the minor device, check it's not out of range, check it actually exists and check it's not already busy.

```
        unsigned int minor = MINOR(inode->i_rdev);

        if (minor >= LP_NO)
                return -ENXIO;
        if ((LP_F(minor) & LP_EXIST) == 0)
```

```
                return -ENXIO;
        if (LP_F(minor) & LP_BUSY)
                return -EBUSY;

        MOD_INC_USE_COUNT;
```

Just in case the following comments aren't clear enough, the following code will bail out upon out-of-paper, off-line or any other printer error, unless the the file was opened using the O_NONBLOCK flag.

```
        /* If ABORTOPEN is set and the printer is offline or out of paper,
           we may still want to open it to perform ioctl()s.  Therefore we
           have commandeered O_NONBLOCK, even though it is being used in
           a non-standard manner.  This is strictly a Linux hack, and
           should most likely only ever be used by the tunelp application. */
        if ((LP_F(minor) & LP_ABORTOPEN) && !(file->f_flags & O_NONBLOCK)) {
                int status = r_str(minor);
                if (status & LP_POUTPA) {
                        printk(KERN_INFO "lp%d out of paper\n", minor);
                        MOD_DEC_USE_COUNT;
                        return -ENOSPC;
                } else if (!(status & LP_PSELECD)) {
                        printk(KERN_INFO "lp%d off-line\n", minor);
                        MOD_DEC_USE_COUNT;
                        return -EIO;
                } else if (!(status & LP_PERRORP)) {
                        printk(KERN_ERR "lp%d printer error\n", minor);
                        MOD_DEC_USE_COUNT;
                        return -EIO;
                }
        }
```

Allocate some memory for the buffer. If we can't get any memory, then exit with a complaint. Otherwise, set the *LP_BUSY* flag and return cleanly.

```
        lp_table[minor].lp_buffer = (char *) kmalloc(LP_BUFFER_SIZE, GFP_KERNEL);
        if (!lp_table[minor].lp_buffer) {
                MOD_DEC_USE_COUNT;
                return -ENOMEM;
        }
        LP_F(minor) |= LP_BUSY;
        return 0;
}
```

6 lp_release(struct inode * inode, struct file * file)

6.1 General info.

Function: *lp_release* releases a parallel device.

Called from: *release* (Base driver operation).

Files: drivers/char/lp.c, include/linux/lp.h

6.2 Code description

This is a fairly simple function. Again, the arguments are just *inode* and *file* (same as for *open*). First, it grabs the minor device number, then frees the memory that was assigned for the buffer (and sets the buffer flag to null). Then it marks the device as no longer busy, and exists.

```
static int lp_release(struct inode * inode, struct file * file)
{
        unsigned int minor = MINOR(inode->i_rdev);

        kfree_s(lp_table[minor].lp_buffer, LP_BUFFER_SIZE);
        lp_table[minor].lp_buffer = NULL;
        LP_F(minor) &= ~LP_BUSY;
        MOD_DEC_USE_COUNT;
        return 0;
}
```

7 lp_ioctl(struct inode *inode, struct file *file, unsigned int cmd, unsigned long arg)

7.1 General info.

Function: *lp_ioctl* enables ioctls for parallel devices.

Called from: *ioctl* (Base driver operation).

Files: drivers/char/lp.c, include/linux/lp.h

7.2 Code description

lp_ioctl handles any miscellaneous control functions that are needed for printing devices. The arguments are the standard *inode* and *file* (see above), as well as *cmd*, the command which we wish to send and *arg*, any further arguments.

```
static int lp_ioctl(struct inode *inode, struct file *file,
                    unsigned int cmd, unsigned long arg)
{
        unsigned int minor = MINOR(inode->i_rdev);
        int retval = 0;

#ifdef LP_DEBUG
        printk(KERN_DEBUG "lp%d ioctl, cmd: 0x%x, arg: 0x%x\n", minor, cmd, arg);
#endif
        if (minor >= LP_NO)
                return -ENODEV;
        if ((LP_F(minor) & LP_EXIST) == 0)
                return -ENODEV;
```

Now go through each of the options. LPTIME sets the amount of time that the driver waits for the printer to catch up when the buffer fills up. LPCHAR sets the timeout for each character. Sending an LPABORT with a true argument will cause an abort upon error. Sending LPABORT will cause a retry upon error (default).

```
        switch ( cmd ) {
                case LPTIME:
                        LP_TIME(minor) = arg * HZ/100;
                        break;
                case LPCHAR:
                        LP_CHAR(minor) = arg;
                        break;
                case LPABORT:
                        if (arg)
                                LP_F(minor) |= LP_ABORT;
                        else
                                LP_F(minor) &= ~LP_ABORT;
                        break;
```

LPABORTOPEN with a true argument will abort future open() calls on error. Calling LPABORTOPEN with a false argument will cause errors on open() to be ignored. LPCAREFUL (true) causes the driver to ensure the the out-of-paper flags, etc, are checked on writes. LPCAREFUL (false) will ignore them.

```
                case LPABORTOPEN:
                        if (arg)
                                LP_F(minor) |= LP_ABORTOPEN;
                        else
                                LP_F(minor) &= ~LP_ABORTOPEN;
                        break;
                case LPCAREFUL:
```

```
                if (arg)
                        LP_F(minor) |= LP_CAREFUL;
                else
                        LP_F(minor) &= ~LP_CAREFUL;
                break;
```

LPWAIT sets the amount of time that the driver waits before and after a strobe. LPSETIRQ and LPGETIRQ set and get the IRQ number for the driver respectively – although, from the look of the code below, LPSETIRQ isn't working, or has been deprecated perhaps?

```
        case LPWAIT:
                LP_WAIT(minor) = arg;
                break;
        case LPSETIRQ:
                return -EINVAL;
                break;
        case LPGETIRQ:
                retval = verify_area(VERIFY_WRITE, (void *) arg,
                    sizeof(int));
                if (retval)
                        return retval;
                copy_to_user((int *) arg, & LP_IRQ(minor), sizeof(int));
                break;
```

LPGETSTATUS gets the status of the printer, LPRESET resets the printer, LPGETSTATS gets the current statistics from the driver and LPGETFLAGS gets the status flags. In a couple of these cases, *verify_area()* is called to ensure that it is permitted to write to these areas. The *copy_to_user()* call copies the resulting structure back to user space.

```
        case LPGETSTATUS:
                retval = verify_area(VERIFY_WRITE, (void *) arg,
                    sizeof(int));
                if (retval)
                        return retval;
                else {
                        int status = r_str(minor);
                        copy_to_user((int *) arg, & status, sizeof(int));
                }
                break;
        case LPRESET:
                lp_reset(minor);
                break;
        case LPGETSTATS:
                retval = verify_area(VERIFY_WRITE, (void *) arg,
```

```
                    sizeof(struct lp_stats));
                if (retval)
                        return retval;
                else {
                        copy_to_user((int *) arg, & LP_STAT(minor), sizeof(struct lp_stats));
                        if (suser())
                                memset(& LP_STAT(minor), 0, sizeof(struct lp_stats));
                }
                break;
        case LPGETFLAGS:
                retval = verify_area(VERIFY_WRITE, (void *) arg,
                    sizeof(int));
                if (retval)
                        return retval;
                else {
                        int status = LP_F(minor);
                        copy_to_user((int *) arg, & status, sizeof(int));
                }
                break;
        default:
                retval = -EINVAL;
    }
    return retval;
}
```