The Linux 2.4 Parallel Port Subsystem

Tim Waugh

twaugh@redhat.com

The Linux 2.4 Parallel Port Subsystem

by Tim Waugh

Table of Contents

Chapter 1. Design goals

1.1. The problems

The first parallel port support for Linux came with the line printer driver, `lp`. The printer driver is a character special device, and (in Linux 2.0) had support for writing, via `write`, and configuration and statistics reporting via `ioctl`.

The printer driver could be used on any computer that had an IBM PC-compatible parallel port. Because some architectures have parallel ports that aren't really the same as PC-style ports, other variants of the printer driver were written in order to support Amiga and Atari parallel ports.

When the Iomega Zip drive was released, and a driver written for it, a problem became apparent. The Zip drive is a parallel port device that provides a parallel port of its own—it is designed to sit between a computer and an attached printer, with the printer plugged into the Zip drive, and the Zip drive plugged into the computer.

The problem was that, although printers and Zip drives were both supported, for any given port only one could be used at a time. Only one of the two drivers could be present in the kernel at once. This was because of the fact that both drivers wanted to drive the same hardware—the parallel port. When the printer driver initialised, it would call the `check_region` function to make sure that the IO region associated with the parallel port was free, and then it would call `request_region` to allocate it. The Zip drive used the same mechanism. Whichever driver initialised first would gain exclusive control of the parallel port.

The only way around this problem at the time was to make sure that both drivers were available as loadable kernel modules. To use the printer, load the printer driver module; then for the Zip drive, unload the printer driver module and load the Zip driver module.

The net effect was that printing a document that was stored on a Zip drive was a bit of an ordeal, at least if the Zip drive and printer shared a parallel port. A better solution was needed.

Zip drives are not the only devices that presented problems for Linux. There are other devices with pass-through ports, for example parallel port CD-ROM drives. There are also printers that report their status textually rather than using simple error pins: sending a command to the printer can cause it to report the number of pages that it has ever printed, or how much free memory it has, or whether it is running out of toner, and so on. The printer driver didn't originally offer any facility for reading back this information (although Carsten Gross added nibble mode readback support for kernel 2.2).

The IEEE has issued a standards document called IEEE 1284, which documents existing practice for parallel port communications in a variety of modes. Those modes are: "compatibility", reverse nibble, reverse byte, ECP and EPP. Newer devices often use the more advanced modes of transfer (ECP and EPP). In Linux 2.0, the printer driver only supported "compatibility mode" (i.e. normal printer protocol) and reverse nibble mode.

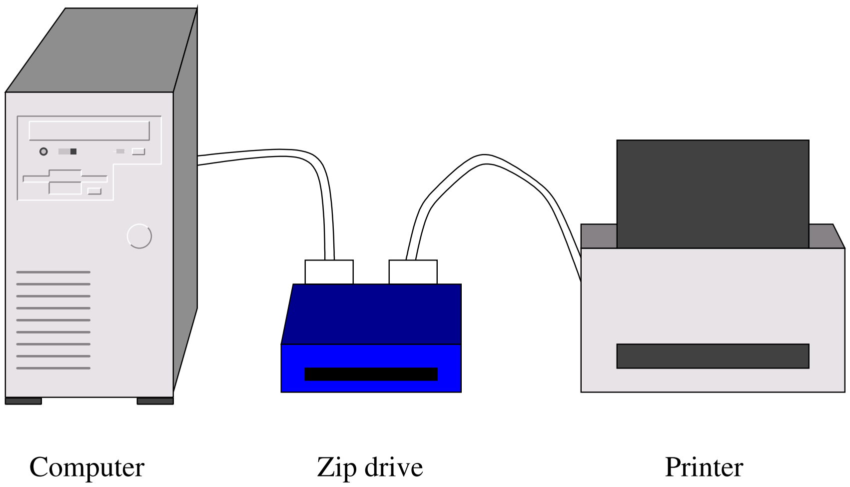
1.2. The solutions

The `parport` code in Linux 2.2 was designed to meet these problems of architectural differences in parallel ports, of port-sharing between devices with pass-through ports, and of lack of support for IEEE 1284 transfer modes.

There are two layers to the `parport` subsystem, only one of which deals directly with the hardware. The other layer deals with sharing and IEEE 1284 transfer modes. In this way, parallel support for a particular architecture comes in the form of a module which registers itself with the generic sharing layer.

The sharing model provided by the `parport` subsystem is one of exclusive access. A device driver, such as the printer driver, must ask the `parport` layer for access to the port, and can only use the port once access has been granted. When it has finished a "transaction", it can tell the `parport` layer that it may release the port for other device drivers to use.

Devices with pass-through ports all manage to share a parallel port with other devices in generally the same way. The device has a latch for each of the pins on its pass-through port. The normal state of affairs is pass-through mode, with the device copying the signal lines between its host port and its pass-through port. When the device sees a special signal from the host port, it latches the pass-through port so that devices further downstream don't get confused by the pass-through device's conversation with the host parallel port: the device connected to the pass-through port (and any devices connected in turn to it) are effectively cut off from the computer. When the pass-through device has completed its transaction with the computer, it enables the pass-through port again.



This technique relies on certain "special signals" being invisible to devices that aren't watching for them. This tends to mean only changing the data signals and leaving the control signals alone. IEEE 1284.3 documents a standard protocol for daisy-chaining devices together with parallel ports.

Support for standard transfer modes are provided as operations that can be performed on a port, along with operations for setting the data lines, or the control lines, or reading the status lines. These operations appear to the device driver as function pointers; more later.

Chapter 2. Standard transfer modes

The “standard” transfer modes in use over the parallel port are “defined” by a document called IEEE 1284. It really just codifies existing practice and documents protocols (and variations on protocols) that have been in common use for quite some time.

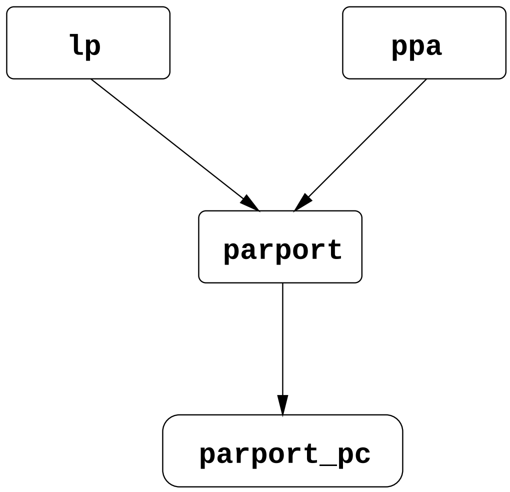
The original definitions of which pin did what were set out by Centronics Data Computer Corporation, but only the printer-side interface signals were specified.

By the early 1980s, IBM’s host-side implementation had become the most widely used. New printers emerged that claimed Centronics compatibility, but although compatible with Centronics they differed from one another in a number of ways.

As a result of this, when IEEE 1284 was published in 1994, all that it could really do was document the various protocols that are used for printers (there are about six variations on a theme).

In addition to the protocol used to talk to Centronics-compatible printers, IEEE 1284 defined other protocols that are used for unidirectional peripheral-to-host transfers (reverse nibble and reverse byte) and for fast bidirectional transfers (ECP and EPP).

Chapter 3. Structure



3.1. Sharing core

At the core of the `parport` subsystem is the sharing mechanism (see `drivers/parport/share.c`). This module, `parport`, is responsible for keeping track of which ports there are in the system, which device drivers might be interested in new ports, and whether or not each port is available for use (or if not, which driver is currently using it).

3.2. Parports and their overrides

The generic `parport` sharing code doesn't directly handle the parallel port hardware. That is done instead by "low-level" `parport` drivers. The function of a low-level `parport` driver is to detect parallel ports, register them with the sharing code, and provide a list of access functions for each port.

The most basic access functions that must be provided are ones for examining the status lines, for setting the control lines, and for setting the data lines. There are also access functions for setting the direction of the data lines; normally they are in the "forward" direction (that is, the computer drives them), but some ports allow switching to "reverse" mode (driven by the peripheral). There is an access function for examining the data lines once in reverse mode.

3.3. IEEE 1284 transfer modes

Stacked on top of the sharing mechanism, but still in the `parport` module, are functions for transferring data. They are provided for the device drivers to use, and are very much like library routines. Since these transfer functions are provided by the generic `parport` core they must use the "lowest common denominator" set of access functions: they can set the control lines, examine the status lines, and use the data lines. With some parallel ports the data lines can only be set and not examined, and with other ports accessing the data register causes control line activity; with these types of situations, the IEEE 1284 transfer functions make a best effort attempt to do the right thing. In some cases, it is not physically possible to use particular IEEE 1284 transfer modes.

The low-level `parport` drivers also provide IEEE 1284 transfer functions, as names in the access function list. The low-level driver can just name the generic IEEE 1284 transfer functions for this. Some parallel ports can do IEEE 1284 transfers in hardware; for those ports, the low-level driver can provide functions to utilise that feature.

3.4. Pardevices and parport_drivers

When a parallel port device driver (such as `lp`) initialises it tells the sharing layer about itself using `parport_register_driver`. The information is put into a struct parport_driver, which is put into a linked list. The information in a struct parport_driver really just amounts to some function pointers to callbacks in the parallel port device driver.

During its initialisation, a low-level port driver tells the sharing layer about all the ports that it has found (using `parport_register_port`), and the sharing layer creates a struct parport for each of them. Each struct parport contains (among other things) a pointer to a struct parport_operations, which is a list of function pointers for the various operations that can be performed on a port. You can think of a struct parport as a parallel port "object", if "object-orientated" programming is your thing. The parport structures are chained in a linked list, whose head is `portlist` (in `drivers/parport/share.c`).

Once the port has been registered, the low-level port driver announces it. The `parport_announce_port` function walks down the list of parallel port device drivers (struct parport_drivers) calling the `attach` function of each.

Similarly, a low-level port driver can undo the effect of registering a port with the `parport_unregister_port` function, and device drivers are notified using the `detach` callback.

Device drivers can undo the effect of registering themselves with the `parport_unregister_driver` function.

3.5. The IEEE 1284.3 API

The ability to daisy-chain devices is very useful, but if every device does it in a different way it could lead to lots of complications for device driver writers. Fortunately, the IEEE are standardising it in IEEE 1284.3, which covers daisy-chain devices and port multiplexors.

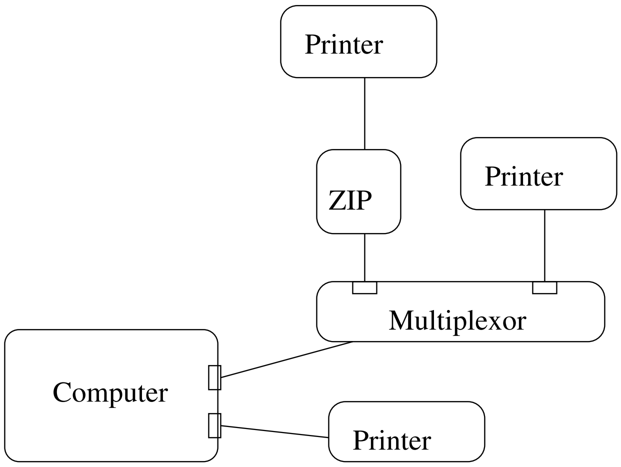
At the time of writing, IEEE 1284.3 has not been published, but the draft specifies the on-the-wire protocol for daisy-chaining and multiplexing, and also suggests a programming interface for using it. That interface (or most of it) has been implemented in the `parport` code in Linux.

At initialisation of the parallel port "bus", daisy-chained devices are assigned addresses starting from zero. There can only be four devices with daisy-chain addresses, plus one device on the end that doesn't know about daisy-chaining and thinks it's connected directly to a computer.

Another way of connecting more parallel port devices is to use a multiplexor. The idea is to have a device that is connected directly to a parallel port on a computer, but has a number of parallel ports on the other side for other peripherals to connect to (two or four ports are allowed). The multiplexor switches control to different ports under software control—it is, in effect, a programmable printer switch.

Combining the ability of daisy-chaining five devices together with the ability to multiplex one parallel port between four gives the potential to have twenty peripherals connected to the same parallel port!

In addition, of course, a single computer can have multiple parallel ports. So, each parallel port peripheral in the system can be identified with three numbers, or co-ordinates: the parallel port, the multiplexed port, and the daisy-chain address.



Each device in the system is numbered at initialisation (by `parport_daisy_init`). You can convert between this device number and its co-ordinates with `parport_device_num` and `parport_device_coords`.

```
int parport_device_num(int parport, int mux, int daisy);
```

`int` **`parport_device_coords`**`(int` *`devnum`*`, int *`*`parport`*`, int *`*`mux`*`, int *`*`daisy`*`);`

Any parallel port peripheral will be connected directly or indirectly to a parallel port on the system, but it won't have a daisy-chain address if it does not know about daisy-chaining, and it won't be connected through a multiplexor port if there is no multiplexor. The special co-ordinate value `-1` is used to indicate these cases.

Two functions are provided for finding devices based on their IEEE 1284 Device ID: `parport_find_device` and `parport_find_class`.

`int` **`parport_find_device`**`(const char *`*`mfg`*`, const char *`*`mdl`*`, int` *`from`*`);`

`int` **`parport_find_class`**`(parport_device_class` *`cls`*`, int` *`from`*`);`

These functions take a device number (in addition to some other things), and return another device number. They walk through the list of detected devices until they find one that matches the requirements, and then return that device number (or `-1` if there are no more such devices). They start their search at the device after the one in the list with the number given (at *`from`*+1, in other words).

Chapter 4. Device driver's view

This section is written from the point of view of the device driver programmer, who might be writing a driver for a printer or a scanner or else anything that plugs into the parallel port. It explains how to use the `parport` interface to find parallel ports, use them, and share them with other device drivers.

We'll start out with a description of the various functions that can be called, and then look at a reasonably simple example of their use: the printer driver.

The interactions between the device driver and the `parport` layer are as follows. First, the device driver registers its existence with `parport`, in order to get told about any parallel ports that have been (or will be) detected. When it gets told about a parallel port, it then tells `parport` that it wants to drive a device on that port. Thereafter it can claim exclusive access to the port in order to talk to its device.

So, the first thing for the device driver to do is tell `parport` that it wants to know what parallel ports are on the system. To do this, it uses the `parport_register_device` function:

```
struct parport_driver {
        const char *name;
        void (*attach) (struct parport *);
        void (*detach) (struct parport *);
        struct parport_driver *next;
};
```

```
int parport_register_driver(struct parport_driver *driver);
```

In other words, the device driver passes pointers to a couple of functions to `parport`, and `parport` calls `attach` for each port that's detected (and `detach` for each port that disappears – yes, this can happen).

The next thing that happens is that the device driver tells `parport` that it thinks there's a device on the port that it can drive. This typically will happen in the driver's `attach` function, and is done with `parport_register_device`:

```
struct pardevice *parport_register_device(struct parport *port,
const char *name, int (*pf) (void *), void (*kf) (void *), void
(*irq_func) (int, void *, struct pt_regs *), int flags, void
*handle);
```

The *port* comes from the parameter supplied to the `attach` function when it is called, or alternatively can be found from the list of detected parallel ports directly with the (now deprecated) `parport_enumerate` function.

The next three parameters, *pf*, *kf*, and *irq_func*, are more function pointers. These callback functions get called under various circumstances, and are always given the *handle* as one of their parameters.

The preemption callback, *pf*, is called when the driver has claimed access to the port but another device driver wants access. If the driver is willing to let the port go, it should return zero and the port will be released on its behalf. There is no need to call `parport_release`. If *pf* gets called at a bad time for letting the port go, it should return non-zero and no action will be taken. It is good manners for the driver to try to release the port at the earliest opportunity after its preemption callback is called.

The "kick" callback, *kf*, is called when the port can be claimed for exclusive access; that is, `parport_claim` is guaranteed to succeed inside the "kick" callback. If the driver wants to claim the port it should do so; otherwise, it need not take any action.

The *irq_func* callback is called, predictably, when a parallel port interrupt is generated. But it is not the only code that hooks on the interrupt. The sequence is this: the lowlevel driver is the one that has done `request_irq`; it then does whatever hardware-specific things it needs to do to the parallel port hardware (for PC-style ports, there is nothing special to do); it then tells the IEEE 1284 code about the interrupt, which may involve reacting to an IEEE 1284 event, depending on the current IEEE 1284 phase; and finally the *irq_func* function is called.

None of the callback functions are allowed to block.

The *flags* are for telling `parport` any requirements or hints that are useful. The only useful value here (other than `0`, which is the usual value) is `PARPORT_DEV_EXCL`. The point of that flag is to request exclusive access at all times—once a driver has successfully called `parport_register_device` with that flag, no other device drivers will be able to register devices on that port (until the successful driver deregisters its device, of course).

The `PARPORT_DEV_EXCL` flag is for preventing port sharing, and so should only be used when sharing the port with other device drivers is impossible and would lead to incorrect behaviour. Use it sparingly!

Devices can also be registered by device drivers based on their device numbers (the same device numbers as in the previous section).

The `parport_open` function is similar to `parport_register_device`, and `parport_close` is the equivalent of `parport_unregister_device`. The difference is that `parport_open` takes a device number rather than a pointer to a struct parport.

```
struct pardevice *parport_open(int devnum, int (*pf) (void *),
int (*kf) (void *), int (*irqf) (int, void *, struct pt_regs *),
int flags, void *handle);
```

```
void parport_close(struct pardevice *dev);
```

```
struct pardevice *parport_register_device(struct parport *port,
const char *name, int (*pf) (void *), int (*kf) (void *), int
(*irqf) (int, void *, struct pt_regs *), int flags, void
*handle);
```

```
void parport_unregister_device(struct pardevice *dev);
```

The intended use of these functions is during driver initialisation while the driver looks for devices that it supports, as demonstrated by the following code fragment:

```
int devnum = -1;
while ((devnum = parport_find_class (PARPORT_CLASS_DIGCAM,
```

```
                                          devnum)) != -1) {
    struct pardevice *dev = parport_open (devnum, ...);
    ...
}
```

Once your device driver has registered its device and been handed a pointer to a struct pardevice, the next thing you are likely to want to do is communicate with the device you think is there. To do that you'll need to claim access to the port.

`int` **`parport_claim`**`(struct pardevice *`*`dev`*`);`

`int` **`parport_claim_or_block`**`(struct pardevice *`*`dev`*`);`

`void` **`parport_release`**`(struct pardevice *`*`dev`*`);`

To claim access to the port, use `parport_claim` or `parport_claim_or_block`. The first of these will not block, and so can be used from interrupt context. If `parport_claim` succeeds it will return zero and the port is available to use. It may fail (returning non-zero) if the port is in use by another driver and that driver is not willing to relinquish control of the port.

The other function, `parport_claim_or_block`, will block if necessary to wait for the port to be free. If it slept, it returns `1`; if it succeeded without needing to sleep it returns `0`. If it fails it will return a negative error code.

When you have finished communicating with the device, you can give up access to the port so that other drivers can communicate with their devices. The `parport_release` function cannot fail, but it should not be called without the port claimed. Similarly, you should not try to claim the port if you already have it claimed.

You may find that although there are convenient points for your driver to relinquish the parallel port and allow other drivers to talk to their devices, it would be preferable to keep hold of the port. The printer driver only needs the port when there is data to print, for example, but a network driver (such as PLIP) could be sent a remote packet at any time. With PLIP, it is no huge catastrophe if a network packet is dropped, since it will likely be sent again, so it is possible for that kind of driver to share the port with other (pass-through) devices.

The `parport_yield` and `parport_yield_blocking` functions are for marking points in the driver at which other drivers may claim the port and use their devices. Yielding the port is similar to releasing it and reclaiming it, but is more efficient because nothing is done if there are no other devices needing the port. In fact, nothing is done even if there are other devices waiting but the current device is still within its "timeslice". The default timeslice is half a second, but it can be adjusted via a `/proc` entry.

```
int parport_yield(struct pardevice *dev);
```

```
int parport_yield_blocking(struct pardevice *dev);
```

The first of these, `parport_yield`, will not block but as a result may fail. The return value for `parport_yield` is the same as for `parport_claim`. The blocking version, `parport_yield_blocking`, has the same return code as `parport_claim_or_block`.

Once the port has been claimed, the device driver can use the functions in the struct parport_operations pointer in the struct parport it has a pointer to. For example:

```
port->ops->write_data (port, d);
```

Some of these operations have "shortcuts". For instance, `parport_write_data` is equivalent to the above, but may be a little bit faster (it's a macro that in some cases can avoid needing to indirect through `port` and `ops`).

Chapter 5. Port drivers

To recap, then:

- The device driver registers itself with `parport`.
- A low-level driver finds a parallel port and registers it with `parport` (these first two things can happen in either order). This registration creates a struct parport which is linked onto a list of known ports.
- `parport` calls the `attach` function of each registered device driver, passing it the pointer to the new struct parport.
- The device driver gets a handle from `parport`, for use with `parport_claim`/`release`. This handle takes the form of a pointer to a struct pardevice, representing a particular device on the parallel port, and is acquired using `parport_register_device`.
- The device driver claims the port using `parport_claim` (or `function_claim_or_block`).
- Then it goes ahead and uses the port. When finished it releases the port.

The purpose of the low-level drivers, then, is to detect parallel ports and provide methods of accessing them (i.e. implementing the operations in struct parport_operations).

A more complete description of which operation is supposed to do what is available in `Documentation/parport-lowlevel.txt`.

Chapter 6. The printer driver

The printer driver, `lp` is a character special device driver and a `parport` client. As a character special device driver it registers a struct file_operations using `register_chrdev`, with pointers filled in for *write*, *ioctl*, *open* and *release*. As a client of `parport`, it registers a struct parport_driver using `parport_register_driver`, so that `parport` knows to call `lp_attach` when a new parallel port is discovered (and `lp_detach` when it goes away).

The parallel port console functionality is also implemented in `lp.c`, but that won't be covered here (it's quite simple though).

The initialisation of the driver is quite easy to understand (see `lp_init`). The `lp_table` is an array of structures that contain information about a specific device (the struct pardevice associated with it, for example). That array is initialised to sensible values first of all.

Next, the printer driver calls `register_chrdev` passing it a pointer to `lp_fops`, which contains function pointers for the printer driver's implementation of `open`, `write`, and so on. This part is the same as for any character special device driver.

After successfully registering itself as a character special device driver, the printer driver registers itself as a `parport` client using `parport_register_driver`. It passes a pointer to this structure:

```
static struct parport_driver lp_driver = {
        "lp",
        lp_attach,
        lp_detach,
        NULL
};
```

The `lp_detach` function is not very interesting (it does nothing); the interesting bit is `lp_attach`. What goes on here depends on whether the user supplied any parameters. The possibilities are: no parameters supplied, in which case the printer driver uses every port that is detected; the user supplied the parameter "auto", in which case only ports on which the device ID string indicates a printer is present are used; or the user supplied a list of parallel port numbers to try, in which case only those are used.

For each port that the printer driver wants to use (see `lp_register`), it calls `parport_register_device` and stores the resulting struct pardevice pointer in the `lp_table`. If the user told it to do so, it then resets the printer.

The other interesting piece of the printer driver, from the point of view of `parport`, is `lp_write`. In this function, the user space process has data that it wants printed, and the printer driver hands it off to the `parport` code to deal with.

The `parport` functions it uses that we have not seen yet are `parport_negotiate`, `parport_set_timeout`, and `parport_write`. These functions are part of the IEEE 1284 implementation.

The way the IEEE 1284 protocol works is that the host tells the peripheral what transfer mode it would like to use, and the peripheral either accepts that mode or rejects it; if the mode is rejected, the host can try again with a different mode. This is the negotation phase. Once the peripheral has accepted a particular transfer mode, data transfer can begin that mode.

The particular transfer mode that the printer driver wants to use is named in IEEE 1284 as "compatibility" mode, and the function to request a particular mode is called `parport_negotiate`.

```
int parport_negotiate(struct parport *port, int mode);
```

The *`modes`* parameter is a symbolic constant representing an IEEE 1284 mode; in this instance, it is `IEEE1284_MODE_COMPAT`. (Compatibility mode is slightly different to the other modes—rather than being specifically requested, it is the default until another mode is selected.)

Back to `lp_write` then. First, access to the parallel port is secured with `parport_claim_or_block`. At this point the driver might sleep, waiting for another driver (perhaps a Zip drive driver, for instance) to let the port go. Next, it goes to compatibility mode using `parport_negotiate`.

The main work is done in the write-loop. In particular, the line that hands the data over to `parport` reads:

```
        written = parport_write (port, kbuf, copy_size);
```

The `parport_write` function writes data to the peripheral using the currently selected transfer mode (compatibility mode, in this case). It returns the number of bytes successfully written:

`ssize_t` **`parport_write`**`(struct parport *`*`port`*`, const void *`*`buf`*`, size_t` *`len`*`);`

`ssize_t` **`parport_read`**`(struct parport *`*`port`*`, void *`*`buf`*`, size_t` *`len`*`);`

(`parport_read` does what it sounds like, but only works for modes in which reverse transfer is possible. Of course, `parport_write` only works in modes in which forward transfer is possible, too.)

The *`buf`* pointer should be to kernel space memory, and obviously the *`len`* parameter specifies the amount of data to transfer.

In fact what `parport_write` does is call the appropriate block transfer function from the struct parport_operations:

```
struct parport_operations {
        [...]

        /* Block read/write */
        size_t (*epp_write_data) (struct par-
port *port, const void *buf,
                                  size_t len, int flags);
        size_t (*epp_read_data) (struct par-
port *port, void *buf, size_t len,
                                 int flags);
```

```
        size_t (*epp_write_addr) (struct par-
port *port, const void *buf,
                                  size_t len, int flags);
        size_t (*epp_read_addr) (struct par-
port *port, void *buf, size_t len,
                                 int flags);

        size_t (*ecp_write_data) (struct par-
port *port, const void *buf,
                                  size_t len, int flags);
        size_t (*ecp_read_data) (struct par-
port *port, void *buf, size_t len,
                                 int flags);
        size_t (*ecp_write_addr) (struct par-
port *port, const void *buf,
                                  size_t len, int flags);

        size_t (*compat_write_data) (struct par-
port *port, const void *buf,
                                     size_t len, int flags);
        size_t (*nibble_read_data) (struct par-
port *port, void *buf,
                                    size_t len, int flags);
        size_t (*byte_read_data) (struct par-
port *port, void *buf,
                                  size_t len, int flags);
};
```

The transfer code in `parport` will tolerate a data transfer stall only for so long, and this timeout can be specified with `parport_set_timeout`, which returns the previous timeout:

`long` **`parport_set_timeout`**`(struct pardevice *`*`dev`*`, long` *`inactivity`*`);`

This timeout is specific to the device, and is restored on `parport_claim`.

Chapter 7. User-level device drivers

7.1. Introduction to ppdev

The printer is accessible through `/dev/lp0`; in the same way, the parallel port itself is accessible through `/dev/parport0`. The difference is in the level of control that you have over the wires in the parallel port cable.

With the printer driver, a user-space program (such as the printer spooler) can send bytes in “printer protocol”. Briefly, this means that for each byte, the eight data lines are set up, then a “strobe” line tells the printer to look at the data lines, and the printer sets an “acknowledgement” line to say that it got the byte. The printer driver also allows the user-space program to read bytes in “nibble mode”, which is a way of transferring data from the peripheral to the computer half a byte at a time (and so it’s quite slow).

In contrast, the `ppdev` driver (accessed via `/dev/parport0`) allows you to:

- examine status lines,
- set control lines,
- set/examine data lines (and control the direction of the data lines),
- wait for an interrupt (triggered by one of the status lines),
- find out how many new interrupts have occurred,
- set up a response to an interrupt,
- use IEEE 1284 negotiation (for telling peripheral which transfer mode, to use)
- transfer data using a specified IEEE 1284 mode.

7.2. User-level or kernel-level driver?

The decision of whether to choose to write a kernel-level device driver or a user-level device driver depends on several factors. One of the main ones from a practical point of view is speed: kernel-level device drivers get to run faster because they are not preemptable, unlike user-level applications.

Another factor is ease of development. It is in general easier to write a user-level driver because (a) one wrong move does not result in a crashed machine, (b) you have access to user libraries (such as the C library), and (c) debugging is easier.

7.3. Programming interface

The `ppdev` interface is largely the same as that of other character special devices, in that it supports `open`, `close`, `read`, `write`, and `ioctl`.

7.3.1. Starting and stopping: `open` and `close`

The device node `/dev/parport0` represents any device that is connected to `parport0`, the first parallel port in the system. Each time the device node is opened, it represents (to the process doing the opening) a different device. It can be opened more than once, but only one instance can actually be in control of the parallel port at any time. A process that has opened `/dev/parport0` shares the parallel port in the same way as any other device driver. A user-land driver may be sharing the parallel port with in-kernel device drivers as well as other user-land drivers.

7.3.2. Control: `ioctl`

Most of the control is done, naturally enough, via the `ioctl` call. Using `ioctl`, the user-land driver can control both the `ppdev` driver in the kernel and the physical parallel port itself. The `ioctl` call takes as parameters a file descriptor (the one returned from opening the device node), a command, and optionally (a pointer to) some data.

`PPCLAIM`

Claims access to the port. As a user-land device driver writer, you will need to do this before you are able to actually change the state of the parallel port in any way. Note that some operations only affect the `ppdev` driver and not the port, such as `PPSETMODE`; they can be performed while access to the port is not claimed.

`PPEXCL`

Instructs the kernel driver to forbid any sharing of the port with other drivers, i.e. it requests exclusivity. The `PPEXCL` command is only valid when the port is not already claimed for use, and it may mean that the next `PPCLAIM` `ioctl` will fail: some other driver may already have registered itself on that port.

Most device drivers don't need exclusive access to the port. It's only provided in case it is really needed, for example for devices where access to the port is required for extensive periods of time (many seconds).

Note that the `PPEXCL` `ioctl` doesn't actually claim the port there and then—action is deferred until the `PPCLAIM` `ioctl` is performed.

`PPRELEASE`

Releases the port. Releasing the port undoes the effect of claiming the port. It allows other device drivers to talk to their devices (assuming that there are any).

`PPYIELD`

Yields the port to another driver. This `ioctl` is a kind of short-hand for releasing the port and immediately reclaiming it. It gives other drivers a chance to talk to their devices, but afterwards claims the port back. An example of using this would be in a user-land printer driver: once a few characters have been written we could give the port to another device driver for a while, but if we still have characters to send to the printer we would want the port back as soon as possible.

It is important not to claim the parallel port for too long, as other device drivers will have no time to service their devices. If your device does not allow for parallel port sharing at all, it is better to claim the parallel port exclusively (see `PPEXCL`).

`PPNEGOT`

Performs IEEE 1284 negotiation into a particular mode. Briefly, negotiation is the method by which the host and the peripheral decide on a protocol to use when transferring data.

An IEEE 1284 compliant device will start out in compatibility mode, and then the host can negotiate to another mode (such as ECP).

The `ioctl` parameter should be a pointer to an int; values for this are in `parport.h` and include:

- `IEEE1284_MODE_COMPAT`
- `IEEE1284_MODE_NIBBLE`
- `IEEE1284_MODE_BYTE`
- `IEEE1284_MODE_EPP`
- `IEEE1284_MODE_ECP`

The `PPNEGOT` `ioctl` actually does two things: it performs the on-the-wire negotiation, and it sets the behaviour of subsequent `read`/`write` calls so that they use that mode (but see `PPSETMODE`).

`PPSETMODE`

Sets which IEEE 1284 protocol to use for the `read` and `write` calls.

The `ioctl` parameter should be a pointer to an int.

`PPGETTIME`

Retrieves the time-out value. The `read` and `write` calls will time out if the peripheral doesn't respond quickly enough. The `PPGETTIME` `ioctl` retrieves the length of time that the peripheral is allowed to have before giving up.

The `ioctl` parameter should be a pointer to a struct timeval.

`PPSETTIME`

Sets the time-out. The `ioctl` parameter should be a pointer to a struct timeval.

`PPWCONTROL`

Sets the control lines. The `ioctl` parameter is a pointer to an unsigned char, the bitwise OR of the control line values in `parport.h`.

`PPRCONTROL`

Returns the last value written to the control register, in the form of an unsigned char: each bit corresponds to a control line (although some are unused). The `ioctl` parameter should be a pointer to an unsigned char.

This doesn't actually touch the hardware; the last value written is remembered in software. This is because some parallel port hardware does not offer read access to the control register.

The control lines bits are defined in `parport.h`:

- `PARPORT_CONTROL_STROBE`
- `PARPORT_CONTROL_AUTOFD`
- `PARPORT_CONTROL_SELECT`
- `PARPORT_CONTROL_INIT`

`PPFCONTROL`

Frobs the control lines. Since a common operation is to change one of the control signals while leaving the others alone, it would be quite inefficient for the user-land driver to have to use `PPRCONTROL`, make the change, and then use `PPWCONTROL`. Of course, each driver could remember what state the control lines are supposed to be in (they are never changed by anything else), but in order to provide `PPRCONTROL`, `ppdev` must remember the state of the control lines anyway.

The `PPFCONTROL` `ioctl` is for "frobbing" control lines, and is like `PPWCONTROL` but acts on a restricted set of control lines. The `ioctl` parameter is a pointer to a struct ppdev_frob_struct:

```
struct ppdev_frob_struct {
        unsigned char mask;
```

```
        unsigned char val;
};
```

The *mask* and *val* fields are bitwise ORs of control line names (such as in PPWCONTROL). The operation performed by PPFCONTROL is:

```
new_ctr = (old_ctr & ~mask) | val;
```

In other words, the signals named in *mask* are set to the values in *val*.

PPRSTATUS

Returns an unsigned char containing bits set for each status line that is set (for instance, PARPORT_STATUS_BUSY). The ioctl parameter should be a pointer to an unsigned char.

PPDATADIR

Controls the data line drivers. Normally the computer's parallel port will drive the data lines, but for byte-wide transfers from the peripheral to the host it is useful to turn off those drivers and let the peripheral drive the signals. (If the drivers on the computer's parallel port are left on when this happens, the port might be damaged.)

This is only needed in conjunction with PPWDATA or PPRDATA.

The ioctl parameter is a pointer to an int. If the int is zero, the drivers are turned on (forward direction); if non-zero, the drivers are turned off (reverse direction).

PPWDATA

Sets the data lines (if in forward mode). The ioctl parameter is a pointer to an unsigned char.

PPRDATA

Reads the data lines (if in reverse mode). The ioctl parameter is a pointer to an unsigned char.

`PPCLRIRQ`

Clears the interrupt count. The `ppdev` driver keeps a count of interrupts as they are triggered. `PPCLRIRQ` stores this count in an int, a pointer to which is passed in as the `ioctl` parameter.

In addition, the interrupt count is reset to zero.

`PPWCTLONIRQ`

Set a trigger response. Afterwards when an interrupt is triggered, the interrupt handler will set the control lines as requested. The `ioctl` parameter is a pointer to an unsigned char, which is interpreted in the same way as for `PPWCONTROL`.

The reason for this `ioctl` is simply speed. Without this `ioctl`, responding to an interrupt would start in the interrupt handler, switch context to the user-land driver via `poll` or `select`, and then switch context back to the kernel in order to handle `PPWCONTROL`. Doing the whole lot in the interrupt handler is a lot faster.

7.3.3. Transferring data: `read` and `write`

Transferring data using `read` and `write` is straightforward. The data is transferring using the current IEEE 1284 mode (see the `PPSETMODE` `ioctl`). For modes which can only transfer data in one direction, only the appropriate function will work, of course.

7.3.4. Waiting for events: `poll` and `select`

The `ppdev` driver provides user-land device drivers with the ability to wait for interrupts, and this is done using `poll` (and `select`, which is implemented in terms of `poll`).

When a user-land device driver wants to wait for an interrupt, it sleeps with `poll`. When the interrupt arrives, `ppdev` wakes it up (with a “read” event, although strictly speaking there is nothing to actually `read`).

7.4. Examples

Presented here are two demonstrations of how to write a simple printer driver for `ppdev`. Firstly we will use the `write` function, and after that we will drive the control and data lines directly.

The first thing to do is to actually open the device.

```
int drive_printer (const char *name)
{
    int fd;
    int mode; /* We'll need this later. */

    fd = open (name, O_RDWR);
    if (fd == -1) {
        perror ("open");
        return 1;
    }
```

Here `name` should be something along the lines of `"/dev/parport0"`. (If you don't have any `/dev/parport` files, you can make them with **mknod**; they are character special device nodes with major 99.)

In order to do anything with the port we need to claim access to it.

```
    if (ioctl (fd, PPCLAIM)) {
        perror ("PPCLAIM");
        close (fd);
        return 1;
    }
```

Our printer driver will copy its input (from `stdin`) to the printer, and it can do that it one of two ways. The first way is to hand it all off to the kernel driver, with the knowledge that the protocol that the printer speaks is IEEE 1284's "compatibility" mode.

```
    /* Switch to compatibility mode.  (In fact we don't need
     * to do this, since we start off in compatibility mode
```

```
 * anyway, but this demonstrates PPNEGOT.)
mode = IEEE1284_MODE_COMPAT;
if (ioctl (fd, PPNEGOT, &mode)) {
    perror ("PPNEGOT");
    close (fd);
    return 1;
}

for (;;) {
    char buffer[1000];
    char *ptr = buffer;
    size_t got;

    got = read (0 /* stdin */, buffer, 1000);
    if (got < 0) {
        perror ("read");
        close (fd);
        return 1;
    }

    if (got == 0)
        /* End of input */
        break;

    while (got > 0) {
        int written = write_printer (fd, ptr, got);

        if (written < 0) {
            perror ("write");
            close (fd);
            return 1;
        }

        ptr += written;
        got -= written;
    }
}
```

The `write_printer` function is not pictured above. This is because the main loop that is shown can be used for both methods of driving the printer. Here is one implementation of `write_printer`:

```
ssize_t write_printer (int fd, const void *ptr, size_t count)
{
    return write (fd, ptr, count);
}
```

We hand the data to the kernel-level driver (using `write`) and it handles the printer protocol.

Now let's do it the hard way! In this particular example there is no practical reason to do anything other than just call `write`, because we know that the printer talks an IEEE 1284 protocol. On the other hand, this particular example does not even need a user-land driver since there is already a kernel-level one; for the purpose of this discussion, try to imagine that the printer speaks a protocol that is not already implemented under Linux.

So, here is the alternative implementation of `write_printer` (for brevity, error checking has been omitted):

```
ssize_t write_printer (int fd, const void *ptr, size_t count)
{
    ssize_t wrote = 0;

    while (wrote < count) {
        unsigned char status, control, data;
        unsigned char mask = (PARPORT_STATUS_ERROR
                              | PARPORT_STATUS_BUSY);
        unsigned char val = (PARPORT_STATUS_ERROR
                              | PARPORT_STATUS_BUSY);
        struct parport_frob_struct frob;
        struct timespec ts;

        /* Wait for printer to be ready */
        for (;;) {
            ioctl (fd, PPRSTATUS, &status);
```

```
            if ((status & mask) == val)
                break;

            ioctl (fd, PPRELEASE);
            sleep (1);
            ioctl (fd, PPCLAIM);
        }

        /* Set the data lines */
        data = * ((char *) ptr)++;
        ioctl (fd, PPWDATA, &data);

        /* Delay for a bit */
        ts.tv_sec = 0;
        ts.tv_nsec = 1000;
        nanosleep (&ts, NULL);

        /* Pulse strobe */
        frob.mask = PARPORT_CONTROL_STROBE;
        frob.val = PARPORT_CONTROL_STROBE;
        ioctl (fd, PPFCONTROL, &frob);
        nanosleep (&ts, NULL);

        /* End the pulse */
        frob.val = 0;
        ioctl (fd, PPFCONTROL, &frob);
        nanosleep (&ts, NULL);

        wrote++;
    }

    return wrote;
}
```

To show a bit more of the `ppdev` interface, here is a small piece of code that is intended to mimic the printer's side of printer protocol.

```
for (;;)
  {
    int irqc;
    int busy = nAck | nFault;
    int acking = nFault;
    int ready = Busy | nAck | nFault;
    char ch;

    /* Set up the control lines when an interrupt happens. */
    ioctl (fd, PPWCTLONIRQ, &busy);

    /* Now we're ready. */
    ioctl (fd, PPWCONTROL, &ready);

    /* Wait for an interrupt. */
    {
      fd_set rfds;
      FD_ZERO (&rfds);
      FD_SET (fd, &rfds);
      if (!select (fd + 1, &rfds, NULL, NULL, NULL))
        /* Caught a signal? */
        continue;
    }

    /* We are now marked as busy. */

    /* Fetch the data. */
    ioctl (fd, PPRDATA, &ch);

    /* Clear the interrupt. */
    ioctl (fd, PPCLRIRQ, &irqc);
    if (irqc > 1)
      fprintf (stderr, "Arghh! Missed %d interrupt%s!\n",
       irqc - 1, irqc == 2 ? "s" : "");

    /* Ack it. */
    ioctl (fd, PPWCONTROL, &acking);
```

```
  usleep (2);
  ioctl (fd, PPWCONTROL, &busy);

  putchar (ch);
}
```