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 (which may block).

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 (which may not block).

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`.

```
#include parport.h

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`.

```
#include parport.h

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


int parport_find_class (parport_device_class );
```