PARPORT interface documentation

Time-stamp: <2000-02-24 13:30:20 twaugh>

Described here are the following functions:

Global functions::
parport_register_driver parport_unregister_driver parport_enumerate parport_register_device parport_unregister_device parport_claim parport_claim_or_block parport_release parport_yield parport_yield_blocking parport_wait_peripheral parport_poll_peripheral parport_wait_event parport_negotiate parport_read parport_write parport_open parport_close parport_device_id parport_device_coords parport_find_class parport_find_device parport_set_timeout

Port functions (can be overridden by low-level drivers):

SPP::
port->ops->read_data port->ops->write_data port->ops->read_status port->ops->read_control port->ops->write_control port->ops->frob_control port->ops->enable_irq port->ops->disable_irq port->ops->data_forward port->ops->data_reverse

EPP::
port->ops->epp_write_data port->ops->epp_read_data port->ops->epp_write_addr port->ops->epp_read_addr

ECP::
port->ops->ecp_write_data port->ops->ecp_read_data port->ops->ecp_write_addr

Other::
port->ops->nibble_read_data port->ops->byte_read_data port->ops->compat_write_data

The parport subsystem comprises `parport` (the core port-sharing code), and a variety of low-level drivers that actually do the port accesses. Each low-level driver handles a particular style of port (PC, Amiga, and so on).

The parport interface to the device driver author can be broken down into global functions and port functions.

The global functions are mostly for communicating between the device driver and the parport subsystem: acquiring a list of available ports, claiming a port for exclusive use, and so on. They also include `generic` functions for doing standard things that will work on any IEEE 1284-capable architecture.

The port functions are provided by the low-level drivers, although the core parport module provides generic `defaults` for some routines. The port functions can be split into three groups: SPP, EPP, and ECP.

SPP (Standard Parallel Port) functions modify so-called `SPP` registers: data, status, and control. The hardware may not actually have registers exactly like that, but the PC does and this interface is modelled after common PC implementations. Other low-level drivers may be able to emulate most of the functionality.

EPP (Enhanced Parallel Port) functions are provided for reading and writing in IEEE 1284 EPP mode, and ECP (Extended Capabilities Port) functions are used for IEEE 1284 ECP mode. (What about BECP? Does anyone care?)

Hardware assistance for EPP and/or ECP transfers may or may not be available, and if it is available it may or may not be used. If hardware is not used, the transfer will be software-driven. In order to cope with peripherals that only tenuously support IEEE 1284, a low-level driver specific function is provided, for altering 'fudge factors'.

Global functions

parport_register_driver - register a device driver with parport

SYNOPSIS

```
#include <linux/parport.h>

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);
```

DESCRIPTION

In order to be notified about parallel ports when they are detected, parport_register_driver should be called. Your driver will immediately be notified of all ports that have already been detected, and of each new port as low-level drivers are loaded.

A `struct parport_driver` contains the textual name of your driver, a pointer to a function to handle new ports, and a pointer to a

function to handle ports going away due to a low-level driver unloading. Ports will only be detached if they are not being used (i.e. there are no devices registered on them).

The visible parts of the `struct parport *` argument given to attach/detach are:

```
struct parport
{
        struct parport *next; /* next parport in list */
        const char *name;     /* port's name */
        unsigned int modes;   /* bitfield of hardware modes */
        struct parport_device_info probe_info;
                        /* IEEE1284 info */
        int number;           /* parport index */
        struct parport_operations *ops;
        ...
};
```

There are other members of the structure, but they should not be touched.

The `modes` member summarises the capabilities of the underlying hardware. It consists of flags which may be bitwise-ored together:

PARPORT_MODE_PCSPP	IBM PC registers are available, i.e. functions that act on data, control and status registers are probably writing directly to the hardware.
PARPORT_MODE_TRISTATE	The data drivers may be turned off. This allows the data lines to be used for reverse (peripheral to host) transfers.
PARPORT_MODE_COMPAT	The hardware can assist with compatibility-mode (printer) transfers, i.e. compat_write_block.
PARPORT_MODE_EPP	The hardware can assist with EPP transfers.
PARPORT_MODE_ECP	The hardware can assist with ECP transfers.
PARPORT_MODE_DMA	The hardware can use DMA, so you might want to pass ISA DMA-able memory (i.e. memory allocated using the GFP_DMA flag with kmalloc) to the low-level driver in order to take advantage of it.

There may be other flags in `modes` as well.

The contents of `modes` is advisory only. For example, if the hardware is capable of DMA, and PARPORT_MODE_DMA is in `modes`, it doesn't necessarily mean that DMA will always be used when possible. Similarly, hardware that is capable of assisting ECP transfers won't necessarily be used.

RETURN VALUE

Zero on success, otherwise an error code.

ERRORS

None. (Can it fail? Why return int?)

EXAMPLE

```
static void lp_attach (struct parport *port)
{
        ...
        private = kmalloc (...);
        dev[count++] = parport_register_device (...);
        ...
}

static void lp_detach (struct parport *port)
{
        ...
}

static struct parport_driver lp_driver = {
        "lp",
        lp_attach,
        lp_detach,
        NULL /* always put NULL here */
};

int lp_init (void)
{
        ...
        if (parport_register_driver (&lp_driver)) {
                /* Failed; nothing we can do. */
                return -EIO;
        }
        ...
```

```
}
```

SEE ALSO

parport_unregister_driver, parport_register_device, parport_enumerate

parport_unregister_driver - tell parport to forget about this driver

SYNOPSIS

```
#include <linux/parport.h>

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

DESCRIPTION

This tells parport not to notify the device driver of new ports or of ports going away. Registered devices belonging to that driver are NOT unregistered: parport_unregister_device must be used for each one.

EXAMPLE

```
void cleanup_module (void)
{
        ...
        /* Stop notifications. */
        parport_unregister_driver (&lp_driver);

        /* Unregister devices. */
        for (i = 0; i < NUM_DEVS; i++)
                parport_unregister_device (dev[i]);
        ...
}
```

SEE ALSO

parport_register_driver, parport_enumerate

parport_enumerate - retrieve a list of parallel ports (DEPRECATED)

SYNOPSIS

```
#include <linux/parport.h>

struct parport *parport_enumerate (void);
```

DESCRIPTION

Retrieve the first of a list of valid parallel ports for this machine. Successive parallel ports can be found using the `struct parport *next` element of the `struct parport *` that is returned. If `next` is NULL, there are no more parallel ports in the list. The number of ports in the list will not exceed PARPORT_MAX.

RETURN VALUE

A `struct parport *` describing a valid parallel port for the machine, or NULL if there are none.

ERRORS

This function can return NULL to indicate that there are no parallel ports to use.

EXAMPLE

```
int detect_device (void)
{
        struct parport *port;

        for (port = parport_enumerate ();
        port != NULL;
        port = port->next) {
                /* Try to detect a device on the port... */
                ...
```

```
        }
        }

        ...
}
```

NOTES

parport_enumerate is deprecated; parport_register_driver should be used instead.

SEE ALSO

parport_register_driver, parport_unregister_driver

parport_register_device - register to use a port

SYNOPSIS

```
#include <linux/parport.h>

typedef int (*preempt_func) (void *handle);
typedef void (*wakeup_func) (void *handle);
typedef int (*irq_func) (int irq, void *handle, struct pt_regs *);

struct pardevice *parport_register_device(struct parport *port,
                                          const char *name,
                                          preempt_func preempt,
                                          wakeup_func wakeup,
                                          irq_func irq,
                                          int flags,
                                          void *handle);
```

DESCRIPTION

Use this function to register your device driver on a parallel port (`port`). Once you have done that, you will be able to use parport_claim and parport_release in order to use the port.

The (`name`) argument is the name of the device that appears in /proc filesystem. The string must be valid for the whole lifetime of the device (until parport_unregister_device is called).

This function will register three callbacks into your driver: `preempt`, `wakeup` and `irq`. Each of these may be NULL in order to indicate that you do not want a callback.

When the `preempt` function is called, it is because another driver wishes to use the parallel port. The `preempt` function should return non-zero if the parallel port cannot be released yet -- if zero is returned, the port is lost to another driver and the port must be re-claimed before use.

The `wakeup` function is called once another driver has released the port and no other driver has yet claimed it. You can claim the parallel port from within the `wakeup` function (in which case the claim is guaranteed to succeed), or choose not to if you don't need it now.

If an interrupt occurs on the parallel port your driver has claimed, the `irq` function will be called. (Write something about shared interrupts here.)

The `handle` is a pointer to driver-specific data, and is passed to the callback functions.

`flags` may be a bitwise combination of the following flags:

Flag	Meaning
PARPORT_DEV_EXCL	The device cannot share the parallel port at all. Use this only when absolutely necessary.

The typedefs are not actually defined -- they are only shown in order to make the function prototype more readable.

The visible parts of the returned `struct pardevice` are:

```
struct pardevice {
        struct parport *port;   /* Associated port */
        void *private;          /* Device driver's 'handle' */
        ...
};
```

RETURN VALUE

A `struct pardevice *`: a handle to the registered parallel port device that can be used for parport_claim, parport_release, etc.

ERRORS

A return value of NULL indicates that there was a problem registering a device on that port.

EXAMPLE

```
static int preempt (void *handle)
{
        if (busy_right_now)
                return 1;

        must_reclaim_port = 1;
        return 0;
}

static void wakeup (void *handle)
{
        struct toaster *private = handle;
        struct pardevice *dev = private->dev;
        if (!dev) return; /* avoid races */

        if (want_port)
                parport_claim (dev);
}

static int toaster_detect (struct toaster *private, struct parport *port)
{
        private->dev = parport_register_device (port, "toaster", preempt,
                                                wakeup, NULL, 0,
                                                private);
        if (!private->dev)
                /* Couldn't register with parport. */
                return -EIO;

        must_reclaim_port = 0;
        busy_right_now = 1;
        parport_claim_or_block (private->dev);
        ...
        /* Don't need the port while the toaster warms up. */
        busy_right_now = 0;
        ...
        busy_right_now = 1;
        if (must_reclaim_port) {
                parport_claim_or_block (private->dev);
                must_reclaim_port = 0;
        }
        ...
}
```

SEE ALSO

parport_unregister_device, parport_claim

parport_unregister_device - finish using a port

SYNPOPSIS

```
#include <linux/parport.h>

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

DESCRIPTION

This function is the opposite of parport_register_device. After using parport_unregister_device, `dev` is no longer a valid device handle.

You should not unregister a device that is currently claimed, although if you do it will be released automatically.

EXAMPLE

```
...
kfree (dev->private); /* before we lose the pointer */
parport_unregister_device (dev);
...
```

SEE ALSO

parport_unregister_driver

parport_claim, parport_claim_or_block - claim the parallel port for a device

SYNOPSIS

```
#include <linux/parport.h>

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

DESCRIPTION

These functions attempt to gain control of the parallel port on which `dev` is registered. `parport_claim` does not block, but `parport_claim_or_block` may do. (Put something here about blocking interruptibly or non-interruptibly.)

You should not try to claim a port that you have already claimed.

RETURN VALUE

A return value of zero indicates that the port was successfully claimed, and the caller now has possession of the parallel port.

If `parport_claim_or_block` blocks before returning successfully, the return value is positive.

ERRORS

-EAGAIN	The port is unavailable at the moment, but another attempt to claim it may succeed.

SEE ALSO

parport_release

parport_release - release the parallel port

SYNOPSIS

```
#include <linux/parport.h>

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

DESCRIPTION

Once a parallel port device has been claimed, it can be released using `parport_release`. It cannot fail, but you should not release a device that you do not have possession of.

EXAMPLE

```
static size_t write (struct pardevice *dev, const void *buf,
                     size_t len)
{
        ...
        written = dev->port->ops->write_ecp_data (dev->port, buf,
                                                  len);
        parport_release (dev);
        ...
}
```

SEE ALSO

change_mode, parport_claim, parport_claim_or_block, parport_yield

parport_yield, parport_yield_blocking - temporarily release a parallel port

SYNOPSIS

```
#include <linux/parport.h>

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

DESCRIPTION

When a driver has control of a parallel port, it may allow another driver to temporarily `borrow` it. `parport_yield` does not block; `parport_yield_blocking` may do.

RETURN VALUE

A return value of zero indicates that the caller still owns the port and the call did not block.

A positive return value from `parport_yield_blocking` indicates that the caller still owns the port and the call blocked.

A return value of -EAGAIN indicates that the caller no longer owns the port, and it must be re-claimed before use.

ERRORS

-EAGAIN	Ownership of the parallel port was given away.

SEE ALSO

parport_release

parport_wait_peripheral - wait for status lines, up to 35ms

SYNOPSIS

```
#include <linux/parport.h>

int parport_wait_peripheral (struct parport *port,
                             unsigned char mask,
                             unsigned char val);
```

DESCRIPTION

Wait for the status lines in mask to match the values in val.

RETURN VALUE

-EINTR	a signal is pending
0	the status lines in mask have values in val
1	timed out while waiting (35ms elapsed)

SEE ALSO

parport_poll_peripheral

parport_poll_peripheral - wait for status lines, in usec

SYNOPSIS

```
#include <linux/parport.h>

int parport_poll_peripheral (struct parport *port,
                             unsigned char mask,
                             unsigned char val,
                             int usec);
```

DESCRIPTION

Wait for the status lines in mask to match the values in val.

RETURN VALUE

-EINTR	a signal is pending
0	the status lines in mask have values in val
1	timed out while waiting (usec microseconds have elapsed)

SEE ALSO

parport_wait_peripheral

parport_wait_event - wait for an event on a port

SYNOPSIS

```
#include <linux/parport.h>

int parport_wait_event (struct parport *port, signed long timeout)
```

DESCRIPTION

Wait for an event (e.g. interrupt) on a port. The timeout is in jiffies.

RETURN VALUE

0	success
<0	error (exit as soon as possible)
>0	timed out

parport_negotiate - perform IEEE 1284 negotiation

SYNOPSIS

```
#include <linux/parport.h>

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

DESCRIPTION

Perform IEEE 1284 negotiation.

RETURN VALUE

0	handshake OK; IEEE 1284 peripheral and mode available
-1	handshake failed; peripheral not compliant (or none present)
1	handshake OK; IEEE 1284 peripheral present but mode not available

SEE ALSO

parport_read, parport_write

parport_read - read data from device

SYNOPSIS

```
#include <linux/parport.h>

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

DESCRIPTION

Read data from device in current IEEE 1284 transfer mode. This only works for modes that support reverse data transfer.

RETURN VALUE

If negative, an error code; otherwise the number of bytes transferred.

SEE ALSO

parport_write, parport_negotiate

parport_write - write data to device

SYNOPSIS

```
#include <linux/parport.h>

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

DESCRIPTION

Write data to device in current IEEE 1284 transfer mode. This only works for modes that support forward data transfer.

RETURN VALUE

If negative, an error code; otherwise the number of bytes transferred.

SEE ALSO

parport_read, parport_negotiate

parport_open - register device for particular device number

SYNOPSIS

```
#include <linux/parport.h>

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

DESCRIPTION

This is like parport_register_device but takes a device number instead of a pointer to a struct parport.

RETURN VALUE

See parport_register_device. If no device is associated with devnum, NULL is returned.

SEE ALSO

parport_register_device

parport_close - unregister device for particular device number

SYNOPSIS

```
#include <linux/parport.h>

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

DESCRIPTION

This is the equivalent of parport_unregister_device for parport_open.

SEE ALSO

parport_unregister_device, parport_open

parport_device_id - obtain IEEE 1284 Device ID

SYNOPSIS

```
#include <linux/parport.h>

ssize_t parport_device_id (int devnum, char *buffer, size_t len);
```

DESCRIPTION

Obtains the IEEE 1284 Device ID associated with a given device.

RETURN VALUE

If negative, an error code; otherwise, the number of bytes of buffer that contain the device ID. The format of the device ID is as follows:

```
[length][ID]
```

The first two bytes indicate the inclusive length of the entire Device ID, and are in big-endian order. The ID is a sequence of pairs of the form:

```
key:value;
```

NOTES

Many devices have ill-formed IEEE 1284 Device IDs.

SEE ALSO

parport_find_class, parport_find_device

parport_device_coords - convert device number to device coordinates

SYNOPSIS

```
#include <linux/parport.h>
```

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

DESCRIPTION

Convert between device number (zero-based) and device coordinates (port, multiplexor, daisy chain address).

RETURN VALUE

Zero on success, in which case the coordinates are (`*parport`, `*mux`, `*daisy`).

SEE ALSO

parport_open, parport_device_id

parport_find_class - find a device by its class

SYNOPSIS

```
#include <linux/parport.h>

typedef enum {
        PARPORT_CLASS_LEGACY = 0,       /* Non-IEEE1284 device */
        PARPORT_CLASS_PRINTER,
        PARPORT_CLASS_MODEM,
        PARPORT_CLASS_NET,
        PARPORT_CLASS_HDC,              /* Hard disk controller */
        PARPORT_CLASS_PCMCIA,
        PARPORT_CLASS_MEDIA,            /* Multimedia device */
        PARPORT_CLASS_FDC,              /* Floppy disk controller */
        PARPORT_CLASS_PORTS,
        PARPORT_CLASS_SCANNER,
        PARPORT_CLASS_DIGCAM,
        PARPORT_CLASS_OTHER,            /* Anything else */
        PARPORT_CLASS_UNSPEC,           /* No CLS field in ID */
        PARPORT_CLASS_SCSIADAPTER
} parport_device_class;

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

DESCRIPTION

Find a device by class. The search starts from device number from+1.

RETURN VALUE

The device number of the next device in that class, or -1 if no such device exists.

NOTES

Example usage:

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

SEE ALSO

parport_find_device, parport_open, parport_device_id

parport_find_device - find a device by its class

SYNOPSIS

```
#include <linux/parport.h>

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

DESCRIPTION

Find a device by vendor and model. The search starts from device number from+1.

RETURN VALUE

The device number of the next device matching the specifications, or -1 if no such device exists.

NOTES

Example usage:

```
int devnum = -1;
while ((devnum = parport_find_device ("IOMEGA", "ZIP+", devnum)) != -1) {
        struct pardevice *dev = parport_open (devnum, ...);
        ...
}
```

SEE ALSO

parport_find_class, parport_open, parport_device_id

parport_set_timeout - set the inactivity timeout

SYNOPSIS

```
#include <linux/parport.h>

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

DESCRIPTION

Set the inactivity timeout, in jiffies, for a registered device. The previous timeout is returned.

RETURN VALUE

The previous timeout, in jiffies.

NOTES

Some of the port->ops functions for a parport may take time, owing to delays at the peripheral. After the peripheral has not responded for `inactivity` jiffies, a timeout will occur and the blocking function will return.

A timeout of 0 jiffies is a special case: the function must do as much as it can without blocking or leaving the hardware in an unknown state. If port operations are performed from within an interrupt handler, for instance, a timeout of 0 jiffies should be used.

Once set for a registered device, the timeout will remain at the set value until set again.

SEE ALSO

port->ops->xxx_read/write_yyy

PORT FUNCTIONS

The functions in the port->ops structure (struct parport_operations) are provided by the low-level driver responsible for that port.

port->ops->read_data - read the data register

SYNOPSIS

```
#include <linux/parport.h>

struct parport_operations {
        ...
        unsigned char (*read_data) (struct parport *port);
        ...
};
```

DESCRIPTION

If port->modes contains the PARPORT_MODE_TRISTATE flag and the PARPORT_CONTROL_DIRECTION bit in the control register is set, this returns the value on the data pins. If port->modes contains the PARPORT_MODE_TRISTATE flag and the PARPORT_CONTROL_DIRECTION bit is not set, the return value _may_ be the last value written to the data register. Otherwise the return value is undefined.

SEE ALSO

write_data, read_status, write_control

port->ops->write_data - write the data register

SYNOPSIS

```
#include <linux/parport.h>

struct parport_operations {
        ...
        void (*write_data) (struct parport *port, unsigned char d);
        ...
};
```

DESCRIPTION

Writes to the data register. May have side-effects (a STROBE pulse, for instance).

SEE ALSO

read_data, read_status, write_control

port->ops->read_status - read the status register

SYNOPSIS

```
#include <linux/parport.h>

struct parport_operations {
        ...
        unsigned char (*read_status) (struct parport *port);
        ...
};
```

DESCRIPTION

Reads from the status register. This is a bitmask:

- PARPORT_STATUS_ERROR (printer fault, "nFault")
- PARPORT_STATUS_SELECT (on-line, "Select")
- PARPORT_STATUS_PAPEROUT (no paper, "PError")
- PARPORT_STATUS_ACK (handshake, "nAck")
- PARPORT_STATUS_BUSY (busy, "Busy")

There may be other bits set.

SEE ALSO

read_data, write_data, write_control

port->ops->read_control - read the control register

SYNOPSIS

```
#include <linux/parport.h>

struct parport_operations {
        ...
        unsigned char (*read_control) (struct parport *port);
        ...
};
```

DESCRIPTION

Returns the last value written to the control register (either from write_control or frob_control). No port access is performed.

SEE ALSO

read_data, write_data, read_status, write_control

port->ops->write_control - write the control register

SYNOPSIS

```
#include <linux/parport.h>

struct parport_operations {
        ...
        void (*write_control) (struct parport *port, unsigned char s);
        ...
};
```

DESCRIPTION

Writes to the control register. This is a bitmask:

- PARPORT_CONTROL_STROBE (nStrobe)
- PARPORT_CONTROL_AUTOFD (nAutoFd)
- PARPORT_CONTROL_INIT (nInit)
- PARPORT_CONTROL_SELECT (nSelectIn)

SEE ALSO

read_data, write_data, read_status, frob_control

port->ops->frob_control - write control register bits

SYNOPSIS

```
#include <linux/parport.h>

struct parport_operations {
        ...
        unsigned char (*frob_control) (struct parport *port,
                                       unsigned char mask,
                                       unsigned char val);
        ...
};
```

DESCRIPTION

This is equivalent to reading from the control register, masking out the bits in mask, exclusive-or'ing with the bits in val, and writing the result to the control register.

As some ports don't allow reads from the control port, a software copy of its contents is maintained, so frob_control is in fact only one port access.

SEE ALSO

read_data, write_data, read_status, write_control

port->ops->enable_irq - enable interrupt generation

SYNOPSIS

```
#include <linux/parport.h>

struct parport_operations {
        ...
        void (*enable_irq) (struct parport *port);
        ...
};
```

DESCRIPTION

The parallel port hardware is instructed to generate interrupts at appropriate moments, although those moments are architecture-specific. For the PC architecture, interrupts are commonly generated on the rising edge of nAck.

SEE ALSO

disable_irq

port->ops->disable_irq - disable interrupt generation

SYNOPSIS

```
#include <linux/parport.h>

struct parport_operations {
        ...
        void (*disable_irq) (struct parport *port);
        ...
};
```

DESCRIPTION

The parallel port hardware is instructed not to generate interrupts. The interrupt itself is not masked.

SEE ALSO

enable_irq

port->ops->data_forward - enable data drivers

SYNOPSIS

```
#include <linux/parport.h>

struct parport_operations {
        ...
        void (*data_forward) (struct parport *port);
        ...
};
```

DESCRIPTION

Enables the data line drivers, for 8-bit host-to-peripheral communications.

SEE ALSO

data_reverse

port->ops->data_reverse - tristate the buffer

SYNOPSIS

```
#include <linux/parport.h>

struct parport_operations {
        ...
        void (*data_reverse) (struct parport *port);
        ...
};
```

DESCRIPTION

Places the data bus in a high impedance state, if port->modes has the PARPORT_MODE_TRISTATE bit set.

SEE ALSO

data_forward

port->ops->epp_write_data - write EPP data

SYNOPSIS

```
#include <linux/parport.h>

struct parport_operations {
        ...
        size_t (*epp_write_data) (struct parport *port, const void *buf,
                                  size_t len, int flags);
        ...
};
```

DESCRIPTION

Writes data in EPP mode, and returns the number of bytes written.

The `flags` parameter may be one or more of the following, bitwise-or'ed together:

PARPORT_EPP_FAST	Use fast transfers. Some chips provide 16-bit and 32-bit registers. However, if a transfer times out, the return value may be unreliable.

SEE ALSO

epp_read_data, epp_write_addr, epp_read_addr

port->ops->epp_read_data - read EPP data

SYNOPSIS

```
#include <linux/parport.h>

struct parport_operations {
        ...
        size_t (*epp_read_data) (struct parport *port, void *buf,
                                 size_t len, int flags);
        ...
};
```

DESCRIPTION

Reads data in EPP mode, and returns the number of bytes read.

The `flags` parameter may be one or more of the following, bitwise-or'ed together:

PARPORT_EPP_FAST	Use fast transfers. Some chips provide 16-bit and 32-bit registers. However, if a transfer times out, the return value may be unreliable.

SEE ALSO

epp_write_data, epp_write_addr, epp_read_addr

port->ops->epp_write_addr - write EPP address

SYNOPSIS

```
#include <linux/parport.h>

struct parport_operations {
        ...
        size_t (*epp_write_addr) (struct parport *port,
                                  const void *buf, size_t len, int flags);
        ...
};
```

DESCRIPTION

Writes EPP addresses (8 bits each), and returns the number written.

The `flags` parameter may be one or more of the following, bitwise-or'ed together:

PARPORT_EPP_FAST	Use fast transfers. Some chips provide 16-bit and 32-bit registers. However, if a transfer times out, the return value may be unreliable.

(Does PARPORT_EPP_FAST make sense for this function?)

SEE ALSO

epp_write_data, epp_read_data, epp_read_addr

port->ops->epp_read_addr - read EPP address

SYNOPSIS

```
#include <linux/parport.h>

struct parport_operations {
        ...
        size_t (*epp_read_addr) (struct parport *port, void *buf,
                                 size_t len, int flags);
        ...
};
```

DESCRIPTION

Reads EPP addresses (8 bits each), and returns the number read.

The `flags` parameter may be one or more of the following, bitwise-or'ed together:

PARPORT_EPP_FAST	Use fast transfers. Some chips provide 16-bit and 32-bit registers. However, if a transfer times out, the return value may be unreliable.

(Does PARPORT_EPP_FAST make sense for this function?)

SEE ALSO

epp_write_data, epp_read_data, epp_write_addr

port->ops->ecp_write_data - write a block of ECP data

SYNOPSIS

```
#include <linux/parport.h>

struct parport_operations {
        ...
        size_t (*ecp_write_data) (struct parport *port,
                                  const void *buf, size_t len, int flags);
        ...
};
```

DESCRIPTION

Writes a block of ECP data. The `flags` parameter is ignored.

RETURN VALUE

The number of bytes written.

SEE ALSO

ecp_read_data, ecp_write_addr

port->ops->ecp_read_data - read a block of ECP data

SYNOPSIS

```
#include <linux/parport.h>

struct parport_operations {
        ...
        size_t (*ecp_read_data) (struct parport *port,
                                 void *buf, size_t len, int flags);
        ...
};
```

DESCRIPTION

Reads a block of ECP data. The `flags` parameter is ignored.

RETURN VALUE

The number of bytes read. NB. There may be more unread data in a FIFO. Is there a way of stunning the FIFO to prevent this?

SEE ALSO

ecp_write_block, ecp_write_addr

port->ops->ecp_write_addr - write a block of ECP addresses

SYNOPSIS

```
#include <linux/parport.h>

struct parport_operations {
        ...
        size_t (*ecp_write_addr) (struct parport *port,
                                  const void *buf, size_t len, int flags);
        ...
};
```

DESCRIPTION

Writes a block of ECP addresses. The `flags` parameter is ignored.

RETURN VALUE

The number of bytes written.

NOTES

This may use a FIFO, and if so shall not return until the FIFO is empty.

SEE ALSO

ecp_read_data, ecp_write_data

port->ops->nibble_read_data - read a block of data in nibble mode

SYNOPSIS

```
#include <linux/parport.h>

struct parport_operations {
        ...
        size_t (*nibble_read_data) (struct parport *port,
                                    void *buf, size_t len, int flags);
        ...
};
```

DESCRIPTION

Reads a block of data in nibble mode. The `flags` parameter is ignored.

RETURN VALUE

The number of whole bytes read.

SEE ALSO

byte_read_data, compat_write_data

port->ops->byte_read_data - read a block of data in byte mode

SYNOPSIS

```
#include <linux/parport.h>

struct parport_operations {
        ...
        size_t (*byte_read_data) (struct parport *port,
                                  void *buf, size_t len, int flags);
        ...
};
```

DESCRIPTION

Reads a block of data in byte mode. The `flags` parameter is ignored.

RETURN VALUE

The number of bytes read.

SEE ALSO

nibble_read_data, compat_write_data

port->ops->compat_write_data - write a block of data in compatibility mode

SYNOPSIS

```
#include <linux/parport.h>

struct parport_operations {
        ...
        size_t (*compat_write_data) (struct parport *port,
                                     const void *buf, size_t len, int flags);
        ...
};
```

DESCRIPTION

Writes a block of data in compatibility mode. The `flags` parameter is ignored.

RETURN VALUE

The number of bytes written.

SEE ALSO

nibble_read_data, byte_read_data