

# **Public interface of libieee1284**

## **API 3.2**

**Tim Waugh**

---

## **Public interface of libieee1284: API 3.2**

Tim Waugh

Copyright © 2001-2003 Tim Waugh

---

# Table of Contents

Introduction .....	iv
libieee1284 .....	v
I. Structures .....	1
parport .....	2
parport_list .....	3
II. Functions .....	4
ieee1284_find_ports .....	5
ieee1284_free_ports .....	6
ieee1284_get_deviceid .....	7
ieee1284_open .....	9
ieee1284_close .....	11
ieee1284_ref .....	12
ieee1284_claim .....	13
ieee1284_release .....	14
ieee1284_data .....	15
ieee1284_status .....	17
ieee1284_control .....	19
ieee1284_negotiation .....	21
ieee1284_ecp_fwd_to_rev .....	23
ieee1284_transfer .....	24
ieee1284_get_irq_fd .....	27
ieee1284_set_timeout .....	28

---

# Introduction

# Name

libieee1284 — IEEE1284 communications library

## Synopsis

```
#include <ieee1284.h>
cc files... -libieee1284
```

## Overview

The libieee1284 library is a library for accessing parallel port devices.

The model presented to the user is fairly abstract: a list of parallel ports with arbitrary names, with functions to access them in various ways ranging from bit operations to block data transfer in one of the IEEE 1284 sanctioned protocols.

Although the library resides in user space the speed penalty may not be as bad as you initially think, since the operating system may well provide assistance with block data transfer operations; in fact, the operating system may even use hardware assistance to get the job done. So, using libieee1284, ECP transfers using DMA are possible.

The normal sequence of events will be that the application

1. calls `ieee1284_find_ports` to get a list of available ports
2. then `ieee1284_get_deviceid` to look for a device on each port that it is interested in
3. and then `ieee1284_open` to open each port it finds a device it can control on.
4. The list of ports returned from `ieee1284_find_ports` can now be disposed of using `ieee1284_free_ports`.
5. Then when it wants to control the device, it will call `ieee1284_claim` to prevent other drivers from using the port
6. then perhaps do some data transfers
7. and then `ieee1284_release` when it is finished that that particular command. This claim-control-release sequence will be repeated each time it wants to tell the device to do something.
8. Finally when the application is finished with the device it will call `ieee1284_close`.

Usually a port needs to be claimed before it can be used. This is to prevent multiple drivers from trampling on each other if they both want to use the same port. The exception to this rule is the collection of IEEE 1284 Device IDs, which has an implicit open-claim-release-close sequence. The reason for this is that it may be possible to collect a Device ID from the operating system, without bothering the device with it.

## Configuration

When `ieee1284_find_ports` is first called, the library will look for a configuration file, `/etc/ieee1284.conf`.

Comments begin with a '#' character and extend to the end of the line. Everything else is freely-formatted tokens. A non-quoted (or double-quoted) backslash character '\' preserves the literal value of the next character, and single and double quotes may be used for preserving white-space. Braces and equals signs are recognised as tokens, unless quoted or escaped.

The only configuration instruction that is currently recognised is "disallow method ppdev", for preventing the use of the Linux ppdev driver.

## Environment

You can enable debugging output from the library by setting the environment variable `LIBIEEE1284_DEBUG` to any value.

## Files

`/etc/ieee1284.conf`

Configuration file.

## See Also

`parport(3)`, `parport_list(3)`, `ieee1284_find_ports(3)`, `ieee1284_free_ports(3)`, `ieee1284_get_deviceid(3)`, `ieee1284_open(3)`, `ieee1284_close(3)`, `ieee1284_claim(3)`, `ieee1284_release(3)`, `ieee1284_data(3)`, `ieee1284_status(3)`, `ieee1284_control(3)`, `ieee1284_negotiation(3)`, `ieee1284_ecp_fwd_to_rev(3)`, `ieee1284_transfer(3)`, `ieee1284_get_irq_fd(3)`, `ieee1284_set_timeout(3)`

---

# Structures

---

## Name

parport — representation of a parallel port

## Synopsis

```
#include <ieee1284.h>
```

## Description

A parport structure represents a parallel port.

## Structure members

The structure has the following members:

```
struct parport {  
    /* An arbitrary name for the port. */  
    const char *name;  
  
    /* The base address of the port, if that has any meaning, or zero. */  
    unsigned long base_addr;  
  
    /* The ECR address of the port, if that has any meaning, or zero. */  
    unsigned long hibase_addr;  
  
    /* The filename associated with this port,  
     * if that has any meaning, or NULL. */  
    const char *filename;  
};
```



---

## Name

parport\_list — a collection of parallel ports

## Synopsis

```
#include <ieee1284.h>
```

## Description

A parport\_list structure is just a vector of parport structures.

## Structure members

The structure has the following members:

```
struct parport_list {  
    /* Number of elements in the vector. */  
    int portc;  
  
    /* The ports. */  
    struct parport **portv;  
};
```

---

# Functions

---

## Name

ieee1284\_find\_ports — find ports on the system

## Synopsis

```
#include <ieee1284.h>
```

```
int ieee1284_find_ports(list, flags);  
  
struct parport_list *list;  
int flags;
```

## Description

This function should be called before the other libieee1284 functions. This gives the library a chance to look around and see what's available, and gives the program a chance to choose a port to use.

The *list* is a pointer to a parport\_list structure that will be filled in on success.

There are no *flags* defined; use zero for this parameter.

## Return value

E1284_OK	Success. <i>list</i> is filled in and must be destroyed using ieee1284_free_ports(3).
E1284_NOMEM	There is not enough memory available.
E1284_NOTIMPL	One or more of the supplied flags is not supported in this implementation.

---

## Name

ieee1284\_free\_ports — safely deallocate a port list

## Synopsis

```
#include <ieee1284.h>
```

```
void ieee1284_free_ports(list);
```

```
struct parport_list *list;
```

## Description

When the port list will no longer be used, the program should call `ieee1284_free_ports` giving it a pointer to the `parport_list` structure that holds the list of ports returned by `ieee1284_find_ports(3)`. The ports are reference counted with the `ieee1284_open` and `ieee1284_close` functions, and so the port list may be freed even if it contains pointers to ports that are still open.

---

## Name

ieee1284\_get\_deviceid — retrieve an IEEE 1284 Device ID

## Synopsis

```
#include <ieee1284.h>
```

```
ssize_t ieee1284_get_deviceid(port, daisy, flags, buffer, len);

struct parport *port;
int daisy;
int flags;
char *buffer;
size_t len;
```

## Description

This function is for retrieving the IEEE 1284 Device ID of the specified device. The device is specified by the *port* to which it is attached, and optionally an address (*daisy*) on the daisy chain of devices on that port.

*daisy* should be -1 to indicate that the device is not participating in a IEEE 1284.3 daisy chain, meaning it is the last (or only) device on the port, or should be a number from 0 to 3 inclusive to indicate that it has the specified daisy chain address (0 is next to the port).

The *flags* parameter should be a bitwise union of any flags that the program wants to use. Available flags are:

F1284\_FRESH

Guarantee a fresh Device ID. A cached or OS-provided ID will not be used.

The provided *buffer* must be at least *len* bytes long, and will contain the Device ID including the initial two-byte length field and a terminating zero byte on successful return, or as much of the above as will fit into the buffer.

## Return value

A return value less than zero indicates an error as below. Otherwise, the return value is the number of bytes of *buffer* that have been filled. A return value equal to the length of the buffer indicates that the Device ID may be longer than the buffer will allow.

E1284\_NOID

The device did not provide an IEEE 1284 Device ID when interrogated (perhaps by the operating system if F1284\_FRESH was not specified).

E1284\_NOTIMPL

One or more of the supplied flags is not supported in this implementation, or if no flags were supplied then this function is not implemented for this type of port or this type of system. This can also be returned if a daisy chain address is specified but daisy chain Device IDs are not yet supported.

E1284\_NOTAVAIL

F1284\_FRESH was specified and the library is unable to access the port to interrogate the device.

`ieee1284_get_deviceid`

`E1284_NOMEM`

There is not enough memory.

`E1284_INIT`

There was a problem initializing the port.

`E1284_INVALIDPORT`

The *port* parameter is invalid.

## Notes

Unless the `F1284_FRESH` flag is given, the library will try to find the device's ID as unobtrusively as possible. First it will ask the operating system if it knows it, and then it will try actually asking the device for it. Because of this, the Device ID may be partially computed (the length field, for example) or even partially missing if the operating system has only remembered some parts of the ID. To guarantee that you are getting the bytes that the device sent, use `F1284_FRESH`. Be aware that the operating system may allow any user to inspect the Device IDs that it provides, whereas device access is normally more restricted.

The initial two-byte length field is a big-endian 16 bit unsigned integer provided by the device and may not be accurate. In particular, it is meant to indicate the length of the entire string including the length field itself; however, some manufacturers exclude the length field or just set the length field to some arbitrary number greater than the ID length.

---

## Name

ieee1284\_open — open a port

## Synopsis

```
#include <ieee1284.h>
```

```
int ieee1284_open(port, flags, capabilities);

struct parport *port;
int flags;
int *capabilities;
```

## Description

In order to begin using a port it must be opened. Any initial set-up of the port is done at this stage. When an open port is no longer needed it should be closed with `ieee1284_close(3)`.

The possible *flags* are:

F1284\_EXCL

This device cannot share the port with any other device. If this is the case it must be declared at this stage, so that other drivers trying to access the port know not to bother; otherwise they will wait until this driver releases the port, i.e. never.

*The iopl/dev-port access methods don't support this yet, but the ppdev ones do.*

If *capabilities* is not NULL it must point to storage for an int, which will be treated as a set of flags, one per bit, which the library sets or clears as appropriate. If a capability is present it will be used when asked for. They are:

CAP1284\_RAW

Pin-level access is available. If this capability is present then the following functions are effective: `ieee1284_write_data`, `ieee1284_read_status`, `ieee1284_wait_status`, `ieee1284_write_control`, `ieee1284_read_control`, `ieee1284_frob_control`.

CAP1284\_NIBBLE

There is an implementation of nibble mode for this port.

CAP1284\_BYTE

There is an implementation of byte mode for this port.

CAP1284\_COMPAT

There is an implementation of compatibility mode for this port.

CAP1284\_ECP

There is a hardware implementation of ECP mode for this port.

CAP1284\_ECPRLE

There is an RLE-aware implementation of ECP mode for this port (the F1284\_RLE flag is recognised by the ECP transfer functions).

CAP1284\_ECPSWE

There is a software implementation of ECP mode for this port.

`ieee1284_open`

`CAP1284_BECP`

There is an implementation of bounded ECP mode for this port.

`CAP1284_EPP`

There is a hardware implementation of EPP mode for this port.

`CAP1284_EPPSWE`

There is a software implementation of EPP mode for this port.

`CAP1284_IRQ`

An interrupt line is configured for this port and interrupt notifications can be received using `ieee1284_get_irq_fd(3)`.

`CAP1284_DMA`

A DMA channel is configured for this port.

## Return value

`E1284_OK`

The port is now opened.

`E1284_INIT`

There was a problem during port initialization. This could be because another driver has opened the port exclusively, or some other reason.

`E1284_NOMEM`

There is not enough memory.

`E1284_NOTAVAIL`

One or more of the supplied flags is not supported by this type of port.

`E1284_INVALIDPORT`

The *port* parameter is invalid (for instance, the *port* may already be open).

`E1284_SYS`

There was a problem at the operating system level. The global variable `errno` has been set appropriately.

## See also

`ieee1284_close(3)`



---

## Name

ieee1284\_close — close an open port

## Synopsis

```
#include <ieee1284.h>
```

```
int ieee1284_close(port);
```

```
struct parport *port;
```

## Description

To close an open port and free any resources associated with it, call `ieee1284_close`.

## Return value

E1284\_OK

The port is now closed.

E1284\_INVALIDPORT

The *port* parameter is invalid (perhaps it is not open, for instance).

E1284\_SYS

There was a problem at the operating system level. The global variable `errno` has been set appropriately.

## See also

`ieee1284_open(3)`

---

## Name

ieee1284\_ref, ieee1284\_unref — modify a port's reference count

## Synopsis

```
#include <ieee1284.h>
```

```
int ieee1284_ref(port);
```

```
struct parport *port;
```

```
int ieee1284_unref(port);
```

```
struct parport *port;
```

## Description

If you want to free the port list from `ieee1284_find_ports` but open one of the ports later on, you will need to prevent it from being destroyed in `ieee1284_free_ports`. Each port has a reference count, and you can use `ieee1284_ref` to increment it and `ieee1284_unref` to decrement it.

If you use `ieee1284_ref` at any stage, you must later call `ieee1284_unref` to relinquish the extra reference. If you do not do this, the resources associated with the port will not be cleaned up.

If you have not previously used `ieee1284_ref` on a port, you must not use `ieee1284_unref` on it.

## Return value

These functions return the number of references held after the increment or decrement.

## See also

`ieee1284_open(3)`

---

## Name

ieee1284\_claim — claim access to the port

## Synopsis

```
#include <ieee1284.h>
```

```
int ieee1284_claim(port);
```

```
struct parport *port;
```

## Description

With the exception of `ieee1284_get_deviceid(3)`, `ieee1284_claim` must be called on an open port before any other `libieee1284` function for accessing a device on it.

## Return value

E1284\_OK

Success. Note that, unless the `F1284_EXCL` flag was specified to start with, the port should be released within a “reasonable” amount of time.

E1284\_NOMEM

There is not enough memory.

E1284\_INVALIDPORT

The `port` parameter is invalid (for instance, it might not have been opened yet).

E1284\_SYS

There was a problem at the operating system level. The global variable `errno` has been set appropriately.

## See also

`ieee1284_release(3)`

---

## Name

ieee1284\_release — release a port

## Synopsis

```
#include <ieee1284.h>
```

```
void ieee1284_release(port);
```

```
struct parport *port;
```

## Description

This function undoes the effect of `ieee1284_claim(3)` by releasing the port for use by other drivers. It is good practice to release the port whenever convenient. If it is never convenient to do so, the `F1284_EXCL` flag should be specified at initialization.

---

## Name

`ieee1284_read_data`, `ieee1284_write_data`, `ieee1284_data_dir`, `ieee1284_wait_data` — control the data lines

## Synopsis

```
#include <ieee1284.h>
```

```
int ieee1284_read_data(port);

struct parport *port;

void ieee1284_write_data(port, dt);

struct parport *port;
unsigned char dt;

int ieee1284_data_dir(port, reverse);

struct parport *port;
int reverse;

int ieee1284_wait_data(port, mask, val, timeout);

struct parport *port;
unsigned char mask;
unsigned char val;
struct timeval *timeout;
```

## Description

These functions manipulate the data lines of the parallel port associated with *port* (which must have been claimed using `ieee1284_claim(3)`). The lines are represented by an 8-bit number (one line per bit) and a direction. The data lines are driven as a group; they may be all host-driven (*forward* direction) or not (*reverse* direction). When the peripheral is driving them the host must not.

For `ieee1284_data_dir` the *reverse* parameter should be zero to turn the data line drivers on and non-zero to turn them off. Some port types may be unable to switch off the data line drivers.

Setting the data lines may have side effects on some port types (for example, some Amiga ports pulse `nStrobe`).

`ieee1284_wait_data` waits, up until the *timeout*, for the data bits specified in *mask* to have the corresponding values in *val*.

## Return value

`ieee1284_read_data` returns the 8-bit number representing the data lines unless it is not possible to return such a value with this port type, in which case it returns an error code. Possible error codes:

`E1284_NOTAVAIL`

Bi-directional data lines are not available on this system.

`E1284_INVALIDPORT`

The *port* parameter is invalid (perhaps it has not been claimed, for instance).

ieee1284\_data

E1284\_SYS

There was an error at the operating system level, and `errno` has been set accordingly.

E1284\_TIMEDOUT

The *timeout* has elapsed.

Whereas `ieee1284_read_data` may return E1284\_NOTAVAIL on its first invocation on the port, if it does not do so then it cannot until `ieee1284_close` is called for that port.

---

## Name

ieee1284\_read\_status, ieee1284\_wait\_status — analyse status lines

## Synopsis

```
#include <ieee1284.h>
```

```
int ieee1284_read_status(port);

struct parport *port;

int ieee1284_wait_status(port, mask, val, timeout);

struct parport *port;
unsigned char mask;
unsigned char val;
struct timeval *timeout;
```

## Description

There are five status lines, one of which is usually inverted on PC-style ports. Where they differ, libieee1284 operates on the IEEE 1284 values, not the PC-style inverted values. The status lines are represented by the following enumeration:

```
enum ieee1284_status_bits
{
    S1284_NFAULT = 0x08,
    S1284_SELECT = 0x10,
    S1284_PERROR = 0x20,
    S1284_NACK   = 0x40,
    S1284_BUSY   = 0x80,
    /* To convert those values into PC-style register values, use this: */
    S1284_INVERTED = S1284_BUSY,
};
```

These functions all act on the parallel port associated with *port*, which must be claimed.

The purpose of `ieee1284_wait_status` is to wait until particular status lines have specified values. Its *timeout* parameter may be modified on return.

## Return value

For `ieee1284_read_status`, the return value is a non-negative integer with bits set as appropriate representing the status lines. A negative result indicates an error.

For `ieee1284_wait_status`, the return value is `E1284_OK` if the status lines now reflect the desired values (i.e. `status & mask` is `val`), or a negative result indicating an error.

Possible error codes:

`E1284_NOTIMPL`

The *port* lacks the required capability. This could be due to a limitation of this version of libieee1284, or a hardware limitation.

`ieee1284_status`

---

`E1284_NOTAVAIL`

Access to the status lines is not available on this port type.

`E1284_TIMEDOUT`

The *timeout* has elapsed.

`E1284_INVALIDPORT`

The *port* parameter is invalid (for instance, perhaps the *port* is not claimed).

## Notes

The nAck pin is often able to trigger interrupts on the host machine. With operating system help these interrupts may be visible to the application via the file descriptor returned by `ieee1284_get_irq_fd`.

Under Linux, the conditions are that the parallel port driver knows which interrupt line to use and is using it, and that the relevant `/dev/parport` device node is accessible and backed by a device driver.



---

## Name

ieee1284\_read\_control, ieee1284\_write\_control, ieee1284\_frob\_control, ieee1284\_do\_nack\_handshake — manipulate control lines

## Synopsis

```
#include <ieee1284.h>
```

```
int ieee1284_read_control(port);
```

```
struct parport *port;
```

```
void ieee1284_write_control(port, ct);
```

```
struct parport *port;
```

```
unsigned char ct;
```

```
void ieee1284_frob_control(port, mask, val);
```

```
struct parport *port;
```

```
unsigned char mask;
```

```
unsigned char val;
```

```
int ieee1284_do_nack_handshake(port, ct_before, ct_after, timeout);
```

```
struct parport *port;
```

```
unsigned char ct_before;
```

```
unsigned char ct_after;
```

```
struct timeval *timeout;
```

## Description

There are four control lines, three of which are usually inverted on PC-style ports. Where they differ, libieee1284 operates on the IEEE 1284 values, not the PC-style inverted values. The control lines are represented by the following enumeration:

```
enum ieee1284_control_bits
{
    C1284_NSTROBE   = 0x01,
    C1284_NAUTOFD   = 0x02,
    C1284_NINIT     = 0x04,
    C1284_NSELECTIN = 0x08,
    /* To convert those values into PC-style register values, use this: */
    C1284_INVERTED = (C1284_NSTROBE |
                     C1284_NAUTOFD |
                     C1284_NSELECTIN),
};
```

These functions all act on the parallel port associated with *port*, which must be claimed.

The current values on the control lines are available by calling `ieee1284_read_control`, and may be set by calling `ieee1284_write_control`.

To adjust the values on a set of control lines, use `ieee1284_frob_control`. The effect of this can be expressed by: `ctr = ((ctr & ~mask) ^ val)`; that is, the bits in *mask* are unset, and then those in *val* are inverted.

The special function `ieee1284_do_nack_handshake` is for responding very quickly in a protocol where the peripheral sets nAck and the host must respond by setting a control line. Its operation, which relies on the host machine knowing which interrupt nAck generates, is as follows:

1. Set the control lines as indicated in *ct\_before*.
2. Wait for nAck interrupt. If *timeout* elapses, return E1284\_TIMEDOUT.
3. Set the control lines as indicated in *ct\_after*.

On Linux using the ppdev driver, this is performed by the device driver in the kernel, and so is faster than normally possible in a user-space library.

## Return value

The return value of `ieee1284_read_control`, if non-negative, is a number representing the value on the control lines.

Possible error codes for `ieee1284_read_control`:

E1284\_NOTAVAIL

The control lines of this port are not accessible by the application.

E1284\_INVALIDPORT

The *port* parameter is invalid (for instance, perhaps it is not claimed).

Possible error codes for `ieee1284_do_nack_handshake`:

E1284\_OK

The handshake was successful.

E1284\_NOTAVAIL

This operation is not available on this port type or system. This could be because port interrupts are not available, or because the underlying device driver does not support the operation.

E1284\_INVALIDPORT

The *port* parameter is invalid (for instance, perhaps it is not claimed).

---

## Name

ieee1284\_negotiate, ieee1284\_terminate — IEEE 1284 negotiation

## Synopsis

```
#include <ieee1284.h>
```

```
int ieee1284_negotiate(port, mode);
```

```
struct parport *port;  
int mode;
```

```
void ieee1284_terminate(port);
```

```
struct parport *port;
```

## Description

These functions are for negotiating to and terminating from IEEE 1284 data transfer modes. The default mode is called compatibility mode, or in other words normal printer protocol. It is a host-to-peripheral mode only. There are special modes that allow peripheral-to-host transfer as well, which may be negotiated to using `ieee1284_negotiate`. IEEE 1284 negotiation is a process by which the host requests a transfer mode and the peripheral accepts or rejects it. An IEEE 1284-compliant device will require a successful negotiation to a particular mode before it is used for data transfer (but simpler devices may not if they only speak one transfer mode).

To terminate the special mode and go back to compatibility mode use `ieee1284_terminate`.

These functions act on the parallel port associated with `port`, which must be claimed.

With a device strictly complying to IEEE 1284 you will need to call `ieee1284_terminate` in between any two calls to `ieee1284_negotiate` for modes other than `M1284_COMPAT`.

## Available modes

### Uni-directional modes

- `M1284_COMPAT`: Compatibility mode. Normal printer protocol. This is not a negotiated mode, but is the default mode in absence of negotiation. `ieee1284_negotiate(port, M1284_COMPAT)` is equivalent to `ieee1284_terminate(port)`. This host-to-peripheral mode is used for sending data to printers, and is historically the mode that has been used for that before IEEE 1284.
- `M1284_NIBBLE`: Nibble mode. This peripheral-to-host mode uses the status lines to read data from the peripheral four bits at a time.
- `M1284_BYTE`: Byte mode. This peripheral-to-host mode uses the data lines in reverse mode to read data from the peripheral a byte at a time.

### Bi-directional modes

- `M1284_ECP`: ECP mode. On entry to ECP mode it is a host-to-peripheral (i.e. forward) mode, but it may be set to reverse mode using `ieee1284_ecp_fwd_to_rev(3)`. It is common for PC hardware to provide assistance with this mode by the use of a FIFO which the host (or, in reverse mode, the peripheral) may fill, so that the hardware can do the handshaking itself.
- `M1284_EPP`: EPP mode. In this bi-directional mode the direction of data transfer is signalled at each byte.

## Mode variations

- `M1284_FLAG_DEVICEID`: Device ID retrieval. This flag may be combined with a nibble, byte, or ECP mode to notify the device that it should send its IEEE 1284 Device ID when asked for data.
- `M1284_BECP`: Bounded ECP is a modification to ECP that makes it more robust at the point that the direction is changed. (Unfortunately it is not yet implemented in the Linux kernel driver.)
- `M1284_ECPRLE`: ECP with run length encoding. In this mode, consecutive data bytes of the same value may be transferred in only a few cycles.

## Return value

<code>E1284_OK</code>	The negotiation was successful.
<code>E1284_NOTAVAIL</code>	Negotiation is not available with this port type.
<code>E1284_REJECTED</code>	Negotiation was rejected by the peripheral.
<code>E1284_NEGFAILED</code>	Negotiation failed for some reason. Perhaps the device is not IEEE 1284 compliant.
<code>E1284_SYS</code>	A system error occurred during negotiation.
<code>E1284_INVALIDPORT</code>	The <i>port</i> parameter is invalid (for instance, perhaps the <i>port</i> is not claimed).

---

## Name

ieee1284\_ecp\_fwd\_to\_rev, ieee1284\_ecp\_rev\_to\_fwd — ECP direction switching

## Synopsis

```
int ieee1284_ecp_fwd_to_rev(port);  
  
struct parport *port;  
  
int ieee1284_ecp_rev_to_fwd(port);  
  
struct parport *port;
```

## Description

These functions are used to switch directions when in ECP mode. On negotiation to ECP mode the direction is forward (in other words, host-to-peripheral). Use `ieee1284_ecp_fwd_to_rev` to switch from forward to reverse, and `ieee1284_ecp_rev_to_fwd` to switch from reverse to forward.

They act on the parallel port associated with *port*, which must be claimed.

## Return value

E1284_OK	Direction switched successfully.
E1284_NOTIMPL	The <i>port</i> lacks the required capability. This could be due to a limitation of this version of libieee1284, or a hardware limitation.
E1284_INVALIDPORT	The <i>port</i> parameter is invalid (for instance, perhaps the <i>port</i> is not claimed).

---

## Name

ieee1284\_nibble\_read, ieee1284\_compat\_write, ieee1284\_byte\_read, ieee1284\_epp\_read\_data, ieee1284\_epp\_write\_data, ieee1284\_epp\_read\_addr, ieee1284\_epp\_write\_addr, ieee1284\_ecp\_read\_data, ieee1284\_ecp\_write\_data, ieee1284\_ecp\_read\_addr, ieee1284\_ecp\_write\_addr — data transfer functions

## Synopsis

```
#include <ieee1284.h>
```

```
ssize_t ieee1284_nibble_read(port, flags, buffer, len);
```

```
struct parport *port;  
int flags;  
char *buffer;  
size_t len;
```

```
ssize_t ieee1284_compat_write(port, flags, buffer, len);
```

```
struct parport *port;  
int flags;  
const char *buffer;  
size_t len;
```

```
ssize_t ieee1284_byte_read(port, flags, buffer, len);
```

```
struct parport *port;  
int flags;  
char *buffer;  
size_t len;
```

```
ssize_t ieee1284_epp_read_data(port, flags, buffer, len);
```

```
struct parport *port;  
int flags;  
char *buffer;  
size_t len;
```

```
ssize_t ieee1284_epp_write_data(port, flags, buffer, len);
```

```
struct parport *port;  
int flags;  
const char *buffer;  
size_t len;
```

```
ssize_t ieee1284_epp_read_addr(port, flags, buffer, len);
```

```
struct parport *port;  
int flags;  
char *buffer;  
size_t len;
```

```
ssize_t ieee1284_epp_write_addr(port, flags, buffer, len);
```

```
struct parport *port;  
int flags;  
const char *buffer;  
size_t len;
```

```
ssize_t ieee1284_ecp_read_data(port, flags, buffer, len);
```

`ieee1284_transfer`

```
struct parport *port;
int flags;
char *buffer;
size_t len;

ssize_t ieee1284_ecp_write_data(port, flags, buffer, len);

struct parport *port;
int flags;
const char *buffer;
size_t len;

ssize_t ieee1284_ecp_read_addr(port, flags, buffer, len);

struct parport *port;
int flags;
char *buffer;
size_t len;

ssize_t ieee1284_ecp_write_addr(port, flags, buffer, len);

struct parport *port;
int flags;
const char *buffer;
size_t len;
```

## Description

This set of functions is for transferring bytes in the relevant transfer mode. For ECP and EPP modes two types of transfer are possible: *data* and *address* (usually referred to as *channel* in ECP).

The supplied *port* must be a claimed port.

The supplied *buffer* must be at least *len* bytes long. When reading, the transferred data is stored in the buffer; when writing the data to be transferred is taken from the buffer.

For reads (peripheral to host): if no data is available and F1284\_NONBLOCK is not in effect, the inactivity timer is started. If data becomes available before the inactivity time-out elapses it is read; otherwise the return value will be E1284\_TIMEDOUT.

For writes (host to peripheral): if the peripheral is not willing to accept data and F1284\_NONBLOCK is not in effect, the inactivity timer is started. If the peripheral indicates that it is willing to accept data before the inactivity time-out elapses it is sent; otherwise the return value will be E1284\_TIMEDOUT

The *flags* may alter the behaviour slightly:

F1284\_NONBLOCK

For reads (peripheral to host): if no data is available, return immediately (with E1284\_TIMEDOUT).

For writes (host to peripheral): if the peripheral is not willing to accept data, return immediately (with E1284\_TIMEDOUT).

F1284\_SWE

Don't use hardware assistance for the transfer, but instead set the parallel port pins according to the wire protocol.

F1284\_RLE (for ECP only)

Use run length encoding. If the peripheral is in ECP mode with RLE, calls to `ieee1284_ecp_read_data` *must* set this flag in order for the RLE from the peripheral to be interpreted correctly, and calls to `ieee1284_ecp_write_data` *may* set this flag in order to take advantage of RLE.

F1284\_FASTEPP (for EPP only)

Use multi-byte transfers. Several bytes at a time are transferred using hardware assistance, if supporting hardware is present. The price of this increased speed is that the return value will be less reliable when this flag is used.

For ECP mode, a given direction is in force at any particular time, and it is up to the application to ensure that it is only writing when in forward mode, and reading when in reverse mode.

## Return value

The return value is the number of bytes successfully transferred or, if negative, one of:

E1284\_NOTIMPL

This transfer mode and flags combination is not yet implemented in `libieee1284`.

E1284\_TIMEDOUT

Timed out waiting for peripheral to handshake.

E1284\_NOMEM

Not enough memory is available.

E1284\_SYS

There was a problem at the operating system level. The global variable `errno` has been set appropriately.

E1284\_INVALIDPORT

The `port` parameter is invalid (for instance, perhaps the `port` is not claimed).

If any bytes are successfully transferred, that number is returned. An error is returned only if no bytes are transferred.

For host-to-peripheral transfers, all data is at the peripheral by the time the call returns.

## See also

`ieee1284_ecp_fwd_to_rev(3)`



---

## Name

ieee1284\_get\_irq\_fd, ieee1284\_clear\_irq — interrupt notification

## Synopsis

```
#include <ieee1284.h>
```

```
int ieee1284_get_irq_fd(port);

struct parport *port;

int ieee1284_clear_irq(port, count);

struct parport *port, , unsigned int *count;
```

## Description

If the *port* has a configured interrupt line and the port type supports interrupt notification, it is possible to obtain a file descriptor that may be used for `select(2)` or `poll(2)`. Any event (readable, writable or exception) means that an interrupt has been triggered. No operations other than `select` or `poll` may be performed on the file descriptor.

The port must be open in order to call `ieee1284_get_irq_fd`, and must be claimed when using `select` or `poll`.

The caller must not close the file descriptor, and may not use it at all when the port is not claimed.

When an interrupt has been detected, the caller must call `ieee1284_clear_irq` to clear the interrupt condition, at which point the number of interrupts raised can be obtained by supplying a non-NULL *count*.

## Return value

For `ieee1284_get_irq_fd`: If the return value is negative then it is an error code listed below. Otherwise it is a valid file descriptor.

E1284\_NOTAVAIL

No such file descriptor is available.

E1284\_INVALIDPORT

The *port* parameter is invalid (for instance, perhaps the *port* is not open).

For `ieee1284_clear_irq`:

E1284\_OK

The interrupt has been cleared. If *count* was not NULL the count of interrupts has been atomically stored to *count* and reset.

E1284\_NOTAVAIL

The *count* parameter was not NULL but interrupt counting is not supported on this type of port. The interrupt has been cleared.

E1284\_SYS

There was a problem clearing the interrupt.

E1284\_INVALIDPORT

The *port* parameter is invalid (for instance, perhaps the *port* is not claimed).

---

## Name

ieee1284\_set\_timeout — modify inactivity timeout

## Synopsis

```
#include <ieee1284.h>
```

```
struct timeval *ieee1284_set_timeout(port, timeout);  
  
struct parport *port;  
struct timeval *timeout;
```

## Description

This function sets a new value for the inactivity timeout (used for block transfer functions), and returns the old value.

The *port* must be claimed.

The *timeout* parameter may be NULL, in which case the old value is left unchanged.

## Return value

This function returns a pointer to a struct timeval representing the old value. This uses the same storage as the *port* structure, and so is not valid after closing the port.

## Notes

Note that this is an inactivity time-out, not an absolute time-out. During a data transfer, if the peripheral is inactive for the length of time specified then the host gives up.

It is also advisory; no guarantee is made that the transfer will ever complete.