

## **How to use the IOWarrior kernel module**

In this document I will try to provide some basic knowledge about how to access IOWarriors on Linux using the basic functions provided by the Kernel-Module for the device.

If you are looking for information about the lowKit library you're in the wrong place, sorry. Please refer to the documentation that is distributed along with the lowKit library. This document here is strictly about '**Low-level IO**' with the IOWarrior. But if you want to understand what's going on inside the lowKit library, keep reading. The lowKit library provides some nice example code for the concepts described in this document.

## **1 What you need to know about Linux and the IOWarrior**

Before we get into the business of coding for the IOWarrior, there are some Linux-specific topics to be explained concerning the IOWarrior.

### **1.1 A tiny little bit of “IOWarrior-on-Linux” Kernel-History**

The IOWarrior kernel module for Linux is available since version 2.6.10. Up to version 2.6.20 of the Linux kernel you have to compile and install the module manually. Since the modules source-code comes with an install-script, it's that not difficult. But you need administrator privileges to do this and you have to do this *every* time you update your kernel to a more recent version.

Note:

A limitation of all the Linux IOWarrior kernel module versions **prior to 2.6.21** is that the driver is not able to handle an IOWarrior24-PowerVampire. (Product-Ids: 0x1511/0x1512). These devices will simply not be classified as IOWarriors and are therefore not accessible through the driver.

With the release of version 2.6.21 the module is part of the default Linux kernel and will be shipped with every distribution based on kernel 2.6.21 or later. There is no need to download and compile the source files anymore. Bug fixes and necessary updates caused by changes to the Linux kernel itself are now taken care of by the Linux kernel developers.

### **1.2 Linux specific hardware issues with the IOWarrior**

I assume you have read the data-sheet for the chip, so you are familiar with the difference between the IO-Pin and SpecialMode functions of the device. On the USB level the IOWarrior provides two interfaces, which on Linux (and Windows) actually show up as two independent devices. You don't really need to know anything about USB interfaces, but you have to understand that by accessing one of the interfaces you will read/write to either the IO-Pin or the SpecialMode functions. If your application needs both features you'll have to open both devices.

---

## The IOWarrior Kernel Module Howto

---

### Note:

Even if you don't need any of the SpecialMode functions I would suggest to open both devices. Otherwise another application would be able to open the SpecialMode device behind your back and enable for instance the LCD mode. Here you will loose control over some of the IO-Pins without even noticing it from your application.

On most systems the location and names of the device files for the IOWarrior will be set by the udev-daemon. If you know how to create and apply a set of rules for the udev-daemon you're free to choose an location and name you like.

### Note

At least the lowKit library expects all IOWarrior device nodes to be created with a default location/name-scheme of

`/dev/usb/iowarrior*`

Keep this in mind before you start tweaking the udev rules for an IOWarrior.

Here is the directory listing for a single IOWarrior plugged into my machine.

```
wyd@rebooter:~ > ls -all /dev/usb/
drwxr-xr-x  2 root root      80 2007-03-22 12:44 .
drwxr-xr-x 13 root root    13580 2007-03-22 12:44 ..
crw-rw-rw-  1 root users 180, 208 2007-03-22 12:44 iowarrior0
crw-rw-rw-  1 root users 180, 209 2007-03-22 12:44 iowarrior1
wyd@rebooter:~ >
```

As already mentioned, the two files represent the two interfaces of the IOWarrior.

But at this stage there is no way to tell which one of the files links to the IO-Pin functions and which one links to the SpecialModes. We will see how to find find out, but you can't tell from simply looking at the name of the device-file.

The same applies if you connect more than one IOWarrior:

```
wyd@rebooter:~ > ls -all /dev/usb/
drwxr-xr-x  2 root root      120 2007-03-22 13:01 .
drwxr-xr-x 13 root root    13580 2007-03-22 12:44 ..
crw-rw-rw-  1 root users 180, 208 2007-03-22 12:44 iowarrior0
crw-rw-rw-  1 root users 180, 209 2007-03-22 12:44 iowarrior1
crw-rw-rw-  1 root users 180, 210 2007-03-22 13:01 iowarrior2
crw-rw-rw-  1 root users 180, 211 2007-03-22 13:01 iowarrior3
wyd@rebooter:~ >
```

Do not assume that the device-files iowarrior0 and iowarrior1 (iowarrior2 and iowarrior3 respectively) **always** represent the two interfaces of a single IOWarrior. The enumeration of USB devices (and therefore the interfaces of IOWarriors) might happen on separate threads. You should be prepared for situations in which the files `/dev/usb/iowarrior0` and `/dev/usb/iowarrior3` link to the same IOWarrior. (But as we will see later, there are ways to easily solve this problem with just a few lines of code...)

There is no hard limit on how many IOWarriors can be connected at the same time, but performance will degrade at some point. From my own experience, I can tell that four IOWarriors work fine even on a rather slow machine, but you need to verify this for your own setup.

## 2 Writing application-code for the IOWarrior

Right in the first paragraph I used the term '**Low-level IO with the IOWarrior**' as the main topic for the whole document. But since the IOWarrior provides us with a very simple way to accomplish complex tasks, 'LowLevel-IO' could also be translated into 'NotVeryDifficult-IO'.

It will only take five different function calls in your application to communicate with an IOWarrior. Only *one* these five calls is specific to an IOWarrior, the other four have familiar sounding names like `open()`, `read()`, `write()` and `close()`.

But let's do this step by step...

### 2.1 Basic header files to be included.

Besides the usual header-files like `<stdlib.h>`, `<stdio.h>` your application will need to include two more header files which are not that common.

The first one is `<linux/types.h>` for some architecture independent data-types needed by the second file `<linux/usb/iowarrior.h>` which provides the definitions for the `ioctl()`-calls into the IOWarrior driver.

Since all errors that are returned by the driver match `errno`-constants you will also need include `errno.h` into your application code.

```
#include<errno.h>
#include<linux/types.h>
#include<linux/usb/iowarrior.h>
```

#### Note:

If you want your application to compile on a machine running a Linux kernel version between 2.6.10 and 2.6.20 you have to include the file `iowarrior.h` from the kernel-module sources you have downloaded.

### 2.2 #define a few useful constants

As you might know there are four different types of IOWarriors available. They all have their specific product-id to tell them apart. Your application will need to check for the product-id's of all the devices before it starts any IO operation, since every type of IOWarrior uses a different layout for the reports that are to be written or read.

So we set up some constants for the product-ids to make our code more readable.

```
#define USB_VENDOR_ID_CODEMERCS 1984
/* low speed iowarrior */
#define USB_DEVICE_ID_CODEMERCS_IOW40 0x1500
#define USB_DEVICE_ID_CODEMERCS_IOW24 0x1501
#define USB_DEVICE_ID_CODEMERCS_IOWPV1 0x1511
#define USB_DEVICE_ID_CODEMERCS_IOWPV2 0x1512
/* full speed iowarrior */
#define USB_DEVICE_ID_CODEMERCS_IOW56 0x1503
```

With this setup we are now ready to start with our application code.

### 2.3 Opening the IOWarrior

Obviously the first thing we need to do is open the device file to which the IOWarrior is connected.

But there is one important point to be made before we go on: An IOWarrior will **always** be opened in **exclusive** mode. Only one process at a time will have access to the device.

```
int fd = -1;

if(( fd = open( "/dev/usb/iowarrior0", O_RDWR)) < 0 ) {
    printf( "iowarrior open failed %d\n",errno );
    exit( 1 );
}
```

As you can see, it is a plain `open()` call. The only interesting part is the value of `errno` when `open()` fails. In the simple code shown above we just print a message to the console. In a real application you will have to check the value of `errno` against one of the following constants to allow for some more sophisticated error-handling.

`errno=EBADF`

The file is not where it is supposed to be. In other words there is no IOWarrior attached to this device file (or the file doesn't even exist).

`errno=EBUSY`

As already mentioned, an IOWarrior can only be accessed by **one** process at a time. This error tells you that some other process is already using the IOWarrior.

`errno=ENODEV`

Let take this as : The device should be there, but it isn't! Something prevented the kernel module from setting up the device for you. This sounds rather vague ... but I never actually saw that error returned. (It could be the result of a race condition where your application opens the device, and you pull the plug at the same time)

`errno=EFAULT`

There was a problem initializing the device before use. Could be that something is already wrong with the USB core on your machine. But there is definitely nothing you can do about it in your application.

But let's be positive, let pretend `open()` returned a file descriptor for us and move on to the next chapter.

### 2.4 Getting information about the device

With the device open, let's see what it has to offer. The driver module implements an `ioctl()`-call that tells us more about each open IOWarrior. The device-specific information is stored in an instance of type `struct iowarrior_info` that is defined in file `iowarrior.h`

```
struct iowarrior_info {
    __u32 vendor;
    __u32 product;
    __u8 serial[9];
    __u32 revision;
    __u32 speed;
    __u32 power;
    __u32 if_num;
    __u32 report_size;
};
```

Here's a description of the struct members:

#### vendor

Not very interesting, it's the Code Mercenaries Vendor ID, the company that produces the IOWarrior. The kernel needs this value to decide which driver is responsible for a device. It's there for completeness.

#### product

This is important! It tells you which type of IOWarrior is connected here. Check this value against the constants `USB_DEVICE_ID_CODEMERS_IOWxx` we defined earlier to find out which type of IOWarrior is connected.

#### serial

Every single IOWarrior-Product has a unique serial number which is returned as string with 8 Hex digits, terminated by `'\0'`. When it gets printed it shows up as something like `"00000c64"`. I used the term "IOWarrior-Product" to emphasize that there will be for instance no two IOWarrior24 that have the same serial, but there sure is an IOWarrior40 and an IOWarrior24 that have the same serial. You really want to save that value for later usage, because this is the way to match up the IO-Pin functions device and the Special-Mode functions device for an IOWarrior. If you find the same product and the same serial its the same silicon!

I lied!

There are some IOWarrior40 that actually have NO serial number. The driver will consequently return the empty string for them. These were the first devices produced and if you don't already have one, there is no chance you'll buy one by accident.

### revision

Another item to remember. This tells you the revision number of the IOWarrior. The number is incremented on bug-fix-releases or when new features are added to the SpecialModes. The data-sheets list all revision changes, so be prepared for missing features on older devices.

### speed

Tells you the speed on the USB-bus on which the device operates. This will always be the same along a product. The value is set to one of the constants

- 0 - when there was a problem detecting the speed
- 1 - USB 1.1 low-speed (IOWarrior24 and IOWarrior40)
- 2 - USB 1.1 full-speed (IOWarrior56)
- 3 - USB 2.0 high-speed (no IOWarrior supports this yet)

### power

Gives you the power consumption the device demands in milli amps. Since the IOWarriors can be configured to use either 100 or 500 mA, it's one of these values you will get.

### if\_num

This is the value that tells you whether you are looking at the IO-Pin device (`if_num==0`) or the SpecialMode device (`if_num==1`).

### report\_size

The number of bytes to be written to or read from the device. It is a fixed value based on whether it is the IO-Pin or SpecialMode functions you're looking at and the specific product-id of the IOWarrior. (More on this in the next chapter.)

Here's some example-code on how to get all this info from a device:

```
struct iowarrior_info info;

if( (ioctl( fd, IOW_GETINFO, &info) == -1 ) {
    printf( "Unable to retrieve device info %d\n",errno );
    goto exit;
}
```

The first argument to the `ioctl()` 'fd' is the file-descriptor from the `open()` call. The second 'IOW\_GETINFO' is a constant defined in 'iowarrior.h' And the third argument '&info' is a pointer to a `struct iowarrior_info` that is to be updated with the device information. In the above code we use a struct created on the stack.

#### Note:

The `ioctl()` expects a pointer as the third argument, so we use the address-operator for the info.

If the `ioctl()` returns 0, the members of the struct have been updated with the information about the device.

Otherwise `errno` will have been set to one of the following values

`errno=ENODEV`

The device was unplugged! If your open call succeeded, someone pulled the plug in the meantime.

`errno=EFAULT`

The kernel detected that your third argument, the pointer to a struct `iowarrior_info` was referencing an illegal address (most likely `NULL`)

`errno=ENOTTY`

Something was wrong with the second argument (use the `#define` value from the header like I did).

## 2.5 Reading

Reading from an IOWarrior works almost like reading from any other file on your system. I put in the "*almost*" because with an IOWarrior you are limited in the number of bytes you can read in a single call. When new data arrives from an IOWarrior it comes in chunks of bytes (reports). The size of a report is determined by two factors : the type of product and whether the data comes from. The IO-Pin device or the SpecialMode functions device. Here is a table for the different sizes of reports you should use :

<b>Table 1 : Report Sizes of the IOWarriors</b>		
<b>Product</b>	<b>IO-Pin Interface</b>	<b>SpecialMode Interface</b>
IOWarrior24	2	8
IOWarrior24PV	2	8
IOWarrior40	4	8
IOWarrior56	7	64

You have to read the exact number of bytes (see Table 1) from an IOWarrior with a single `read()`. The read will either return all the bytes you requested or none. Since we don't want to check the product type of our device on each `read()` call, we simply use the value `iowarrior_info.report_size` (see previous chapter). If no data is immediately available, the `read()` will block the thread from which it was called until the IOWarrior sends data.

A read application will wrap the `read()`-call with `select()` .

```
char buf[64]; //this is big enough for all IOWarriors
int size=info.report_size; //taken from struct iowarrior_info

if(read(fd,buf,size)!=size) {
    printf( "Error in read %d\n",errno );
    goto exit;
}
else {
    //do something with the data in buf
}
```

Here are the `errno` values that are returned when `read()` failed

`errno=ENODEV`

The device was unplugged

`errno=EINVAL`

You requested to read more or less than the number of bytes allowed (see Table 1)

`errno=EAGAIN`

`F_NONBLOCK` is set for your file descriptor, but no data is currently available.

`errno=EFAULT`

The address for the buffer into which the data was to be read is invalid (Most likely `NULL`)



errno=ERESTART

An uncaught signal was sent to your application. (You hit `ctrl-C` for an app that was stuck in a blocking read ?)

Internally the module implements a buffer for the last 16 reports sent by the IOWarrior. When this buffer is full and a new report arrives the oldest value from buffer is lost. So you should check for new values according to the expected data rate of the hardware. The IOWarrior24 and IOWarrior40 can send a new report at an interval of 8 ms, the IOWarrior 56 needs only 1 ms for this. If you expect fast bursts of signals that toggle the IO-Pins on the device, you better be fast not to loose any reports.

## 2.6 Writing

Starting with version 0.4 of the IOWarrior kernel module, there is an implementation for a generic `write()` function. The number of bytes you want to write to an IOWarrior must match the values already presented in Table 1 (see the section 2.5 about reading from the device.)

The `write()` looks like you already expected it :

```
char buf[64]; //make sure the write buffer is big enough for any IOWarrior
int size = info.report_size; //taken from struct iowarrior_info

buf[0] = 0xFF; //setup the data to be written
buf[1] = 0x??;
.....

if( write(fd,buf,size) != size ) {
    printf( "Error in write %d\n",errno );
    goto exit;
}
else {
    //do something on successful write
}
```

and here are the `errno` values that are returned if `write()` failed

`errno=ENODEV`

The device was unplugged

`errno=EINVAL`

You requested to write more or less than the number of bytes allowed (please refer to the values in Table 1)

`errno=EAGAIN`

You have set `F_NONBLOCK` for your file descriptor but the write would block because the device is busy.

`errno=EFAULT`

The address for the buffer that contains the data to be written is invalid (Most likely `NULL`)

`errno=ERESTART`

An uncaught signal was sent to your application. (You hit `ctrl-C` for an app that was stuck in a blocking read ?)

`errno=ENOMEM`

The driver was unable to allocate some resources in kernel memory.

But we're not finished yet, sorry...

Inside the `write()` function the data is handed over to the Linux USB core which might report errors not shown above. Since this list of error conditions simply too long to be

repeated (and might grow or shrink with kernel versions) it will not be repeated here. But since it's already installed with the Linux kernel sources you might want to have a look at file

```
'/usr/src/linux/Documentation/usb/error-codes.txt'
```

If `write()` returns something unusual this might help to track down the problem.

### **How long does a write take?**

This depends not only on the product type of IOWarrior but also on the type of USB host controller (UHCI vs. OHCI) to which the IOWarrior is connected.

For the IOWarrior24 and the IOWarrior40 you can expect something like 3-5 milliseconds for a single write access through an UHCI host controller. If the IOWarrior is plugged into an OHCI host controller you'll get down to 1millisecond.

For an IOWarrior56 it's 1 millisecond no matter what kind of hub you're using.

I **strongly** suggest the use of `select()` to check whether the IOWarrior is ready to accept new data to be written. If you send fast bursts of data and the device is busy, your application will be put to sleep inside the kernel. The application will be woken up when the new data can be written, but some tests showed that it is much faster to wait in a `select()`-call than to block in kernel mode. Timing on an IOWarrior56 can degrade from 1 millisecond down to 2 milliseconds for a single `write()`.

### 2.7 Closing the device

This is going to be short! Please do it! Even when an IOWarrior was unplugged there are still kernel resources allocated by the module. These resources cannot be released until you call `close()` on your file-descriptor.

We all know what `close` looks like:

```
if(close(fd)) {  
    printf( "Error in close %d\n",errno );  
    goto exit;  
}
```

Even if `close()` returns something else than 0 you're done.

The only error that is special for an IOWarrior is

`errno=ENODEV`

The device was unplugged... but that was probably the reason why you closed it.

#### **Legal disclaimer**

This document is ©2006-2007 by Eberhard Fahle.

Questions, objections, corrections to <[e.fahle@wayoda.org](mailto:e.fahle@wayoda.org)>

I make no claims as to the completeness or correctness of the information contained in this document. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.

Document Revision History 'The IOWarrior Kernel Modul Howto'

12/11/2006    Version 0.1.0.0

First public release

03/27/2007    Version 0.1.1.0

Updates reflecting changes to the header,  
removed sections on `read/write-ioctl()` calls