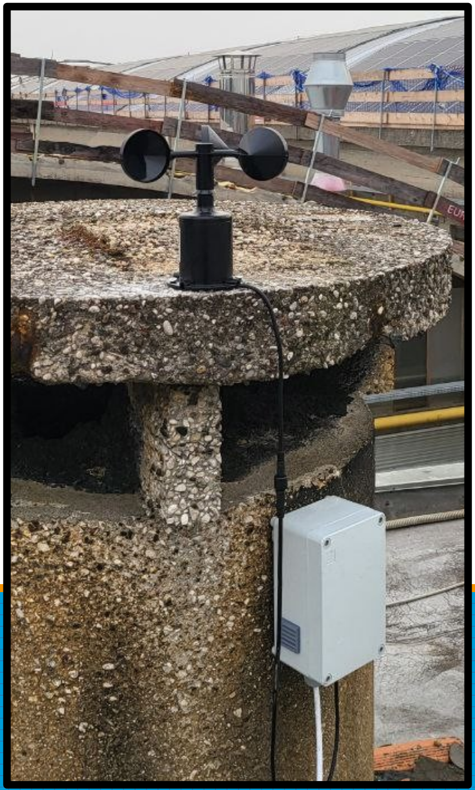
EuroBSDCon 2024 - Dublin



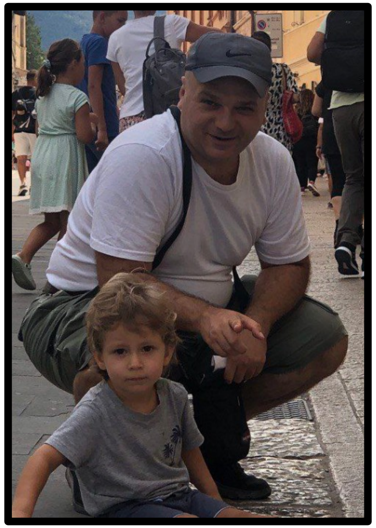
An introduction to GPIO in RPi3B+ and NetBSD, building a Wind-speed logger as an application.

CAVEAT. This document did not go through a detailed external review, double check the content before using these procedures in production.

Dr. Nicola Mingotti

Sep 2024, document release 1.6

Who am I



- Dr. Nicola Mingotti -- Linkedin -- nmingotti-THING-gmail.com
- PhD in Applied Math (Carlos III Madrid, Spain) [laude]
  . Master Mathematics (Bologna) [laude]
  . Master in Finance and Risk Management (Pisa) [top score]
- Relevant experience:
  . Technologist INFN (Firenze)
  . Research associate at Stanford/SLAC (Menlo Park, USA)
  . Software developer for Università di Bologna, Telefonica (Madrid) and more
- I have my own little company, mostly consultant for Borghi SRL (Melara, IT)
- husband and father

Our family company

Borghi SRL

Table of content

. **Installation**, complete recipe to install NetBSD 10.0 in RPi3B+
. **Exclusions and limits**, what we will not talk about
. **GPIO**, what they are and where you can find them
. **GPIO basics**, manually turn them on, off and read their state; from the shell
.. Applications: toggle an LED, read a switch state
. **GPIO in C**, as fast as possible, under self imposed limits. bit-banging and polling
.. Applications: dim an LED (as an exercise)
. GPIO drivers. **gpioirq** and **gpiopwm**
.. Applications: blinky, read push button, read isolated signals in loopback
. **Exercises**
. The Wind-speed **Logger**
.. Steps building a functional prototype with RPi and NetBSD
. Further readings and conclusions

Section - 1 | NetBSD Installation

Objective. Get NetBSD 10.0 running in your RPi3B+

Quick NetBSD install procedure.1

- **Needed hardware**: **(1)** RPi3B+ [*] **(2)** microSD class A1 64GB [*] **(3)** microSD reader/writer [*] **(4)** USB keyboard **(5)** HDMI display **(6)** USB-microUSB cable to power the RPI3B+
- Go to this web site for bootable ARM images https://nycdn.netbsd.org/pub/arm/
- Get the **OFFICIAL RELEASE NetBSD 10.0**, **GENERIC 32 BIT**
  You download the file named NetBSD-10-earmv7hf--generic.img.gz
- Decompress the file
  You get: NetBSD-10-earmv7hf--generic.img
- Burn this last file, the *.img, on the microSD. This depends on the OS of your computer. I will show the procedure on **Linux/Debian.**
  DON'T BLINDLY COPY MY COMMAND HERE, RISK OF WRECKING YOUR COMPUTER , what goes in **of=/dev/xxx** must be understood.
  $> sudo dd if=NetBSD-10-earmv7hf--generic.img of=/dev/sdb

Quick install procedure.2

- Wait for the writing the micro-SD to complete, when done the computer may ask to mount it, don’t.
- Grab a RPi3B+, not powered and plug the micro-SD in it
- Plug a USB keyboard and a HDMI screen in the RPi3B+
- Give power to the RPi3B+ via the micro-USB port
- The first time the system boots it will grow the filesystem then reboot automatically, takes about 5 mins.
- login as **root**, no password required
- If you did not connect an Ethernet cable you might want to disable ntpd
  or it will print error messages and disrupt your typing.
  `#> /etc/rc.d/ntpd stop`

Quick install procedure.3

- You are ready to work !
- Suggestions.
  . configure Ethernet and work via **ssh**
  . learn how to use the **serial** console
  . learn how use **packages**, e.g. for
  doas, emacs, ruby …
  . make a **non root user** and work with that
  . get a **2.5 A power supply**. Powering
  from your PC USB can give problems
  if you connect devices to the RPi. [*]

Section 1 | GPIO - a first look

Objective1. State the limits of our discussion

Objective2. What is a GPIO ? What can it do ?

Objective3. Beware of GPIO electrical properties

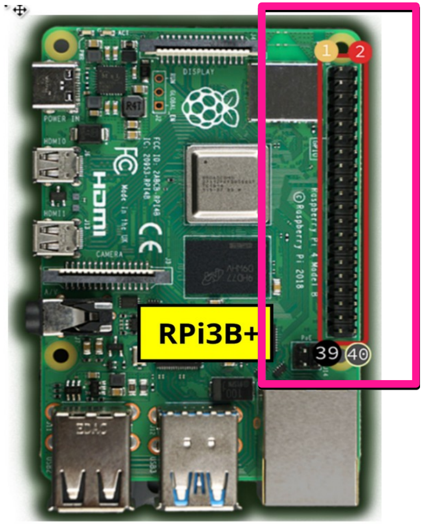
Objective4. Identification: pin-number -- gpio-number

Self imposed limits

- What we say here has been checked and tested only in **RPi3B+** and only in **NetBSD.10.0**.
  BE CAREFULL IN PARTICULAR ABOUT **ELECTRICAL CHARATERISTICS SPECIFIC FOR RPI3B+**
- No **DeviceTree** modification (DTS,DTB), we don't change pin default functions
- Only **Digital GPIO** (GPIO, in general, can have other functions e.g. A/D converter)
- We work at **securelevel 1**, the default [1]
  - => To change a pin setting we must **reboot**
  - => We don't have access to **/dev/mem**
    - With this we could change pin values bypassing the kernel (fast)
- No **transistors** or IC (keep circuits as simple as possible)
  - => We are limited to 18mA in output
    - => no relays, no motors, no speaker; those would make a far more spectacular presentation.
- We program in **user space**, use only available drivers
  - => no new modules, no kernel hacks
- We program in **sh** and **C**
  - => GPIO are controllable natively and easily via **Lua** [2]

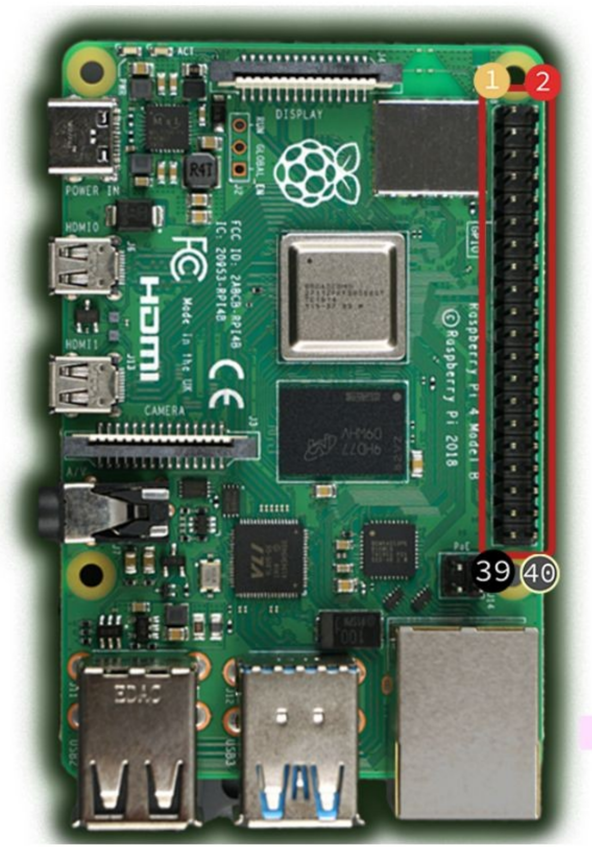
GPIO, where and what

- GPIO: **G**eneralized **P**in **I**nput **O**utput
  - **Digital** pin, meaning it has only two states, **on** and **off**.
  - Output
    - **Set a pin tension** to **0V** or **3.3V**
    - e.g. turn on and off a lamp or any other electrical device
  - Input
    - **Read a pin tension**, state is **ON** if **1.8V - 3.3V**, **OFF** if lower.
    - e.g read a push button pressure, read a switch status
  - Max reccomended out current from a gpio **18mA**
  - GPIOs, as input pins, have high impedance



ATTENTION. applying more than 3.3V to an input GPIO pin might burn your RPi3B+.

Pin.number «--» GPIO.number



			Pin	Pin			
		3V3	1	2	5V		
	I2C SDA	GPIO2	3	4	5V		
	I2C SCL	GPIO3	5	6	GND		
		GPIO4	7	8	GPIO14	UART TX	
		GND	9	10	GPIO15	UART RX	
		GPIO17	11	12	GPIO18	PCM CLK	PWM0
		GPIO27	13	14	GND		
		GPIO22	15	16	GPIO23		
		3V3	17	18	GPIO24		
	SPI MOSI	GPIO10	19	20	GND		
	SPI MISO	GPIO9	21	22	GPIO25		
	SPI SCLK	GPIO11	23	24	GPIO8	SPI CE0	
		GND	25	26	GPIO7	SPI CE1	
	I2C ID EEPROM	GPIO0	27	28	GPIO1	I2C ID EEPROM	
		GPIO5	29	30	GND		
		GPIO6	31	32	GPIO12	PWM0	
	PWM1	GPIO13	33	34	GND		
PWM1	PCM FS	GPIO19	35	36	GPIO16		
		GPIO26	37	38	GPIO20	PCM DIN	
		GND	39	40	GPIO21	PCM DOUT	

Reading Example.
pin number 11 is called (by default) in the OS GPIO17.

Note1. Some pins are associated by default to some functions, as GPIO{2,3,14,..} don't use them at the beginning.

Note2. We use here only the pins framed in a purple box.

Section 2 | GPIO basics

Objective-1. **Configure** a pin for input or output

Objective-2. Set a pin state (ON/OFF) with **gpioctl**

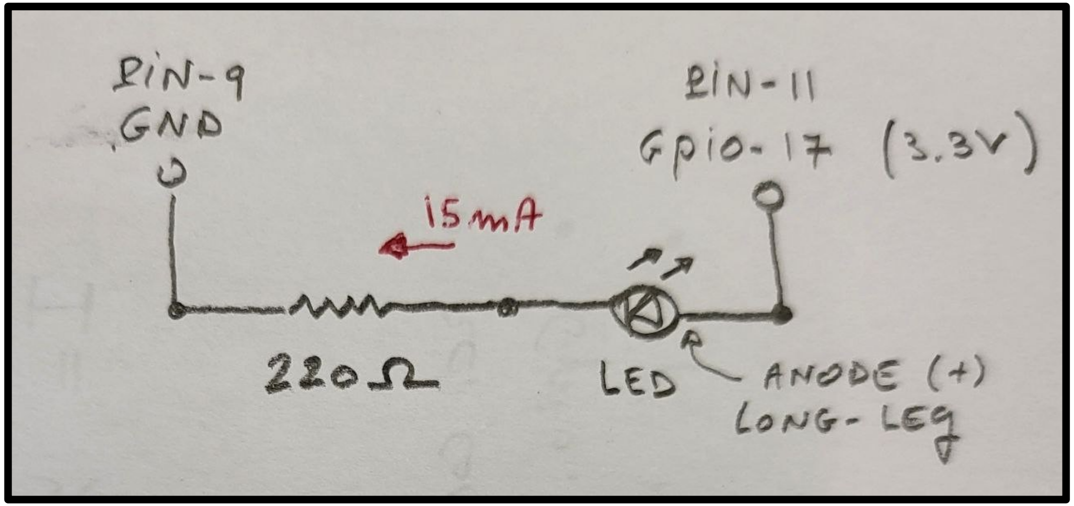
Objective-3. Read a pin state with **gpioctl**

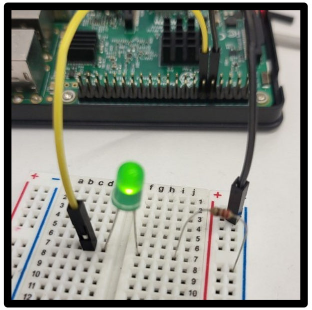
Application-1. Turn an LED ON and OFF

Application-2. Read the state of a switch

GPIO output pin - basic circuit

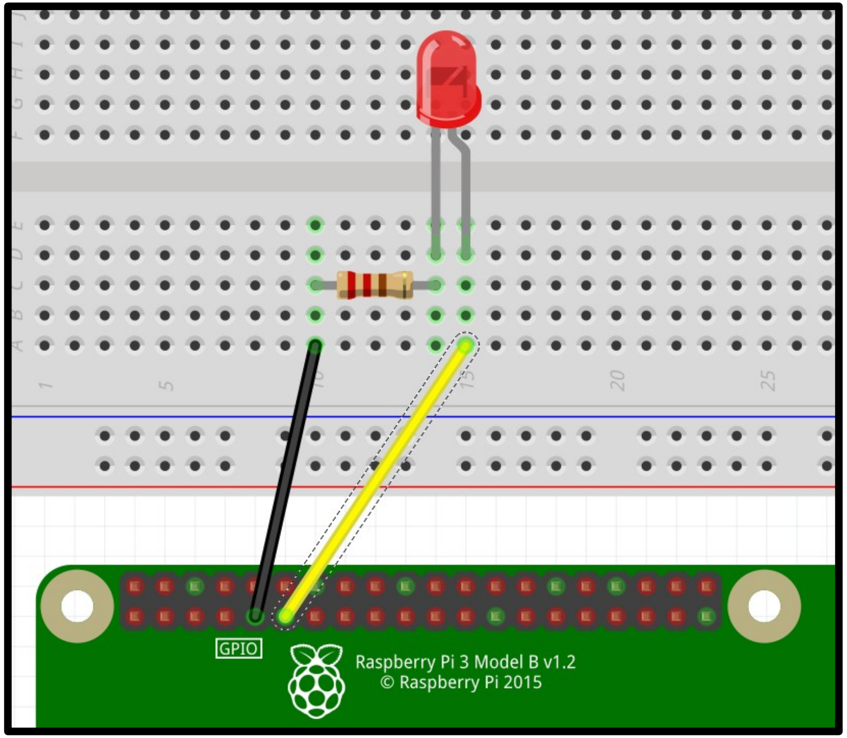
Scheme





Reality

Fritzing



GPIO output pin configuration

```
---- /etc/rc.conf ------------------------
# gpio must be configured at a proper securelevel , boot time is ok
# evaluates what is in /etc/gpio.conf
# try to read /etc/rc.d/gpio , see gpioctl(8)
gpio=YES
-----------------------------------------
```

. **gpio0** below refers to the device name **/dev/gpio0** [see gpio(4)]

```
$> dmesg | grep gpio          # check the name of the gpio device
gpio0 at bcmgpio0: 54 pins
----- /etc/gpio.conf ------------------
gpio0 17 set out      # gpio.17 will be an output pin
gpio0 17 0            # gpio.17 will be by default off (0)
-----------------------------------------
```

change GPIO state with gpioctl

. if you are working as **root** don't type **doas** in the following commands

. if you made a user let it toggle pins, my user here is `p`

```
$> doas usermod -G _gpio p          # add `p` to group _gpio
```

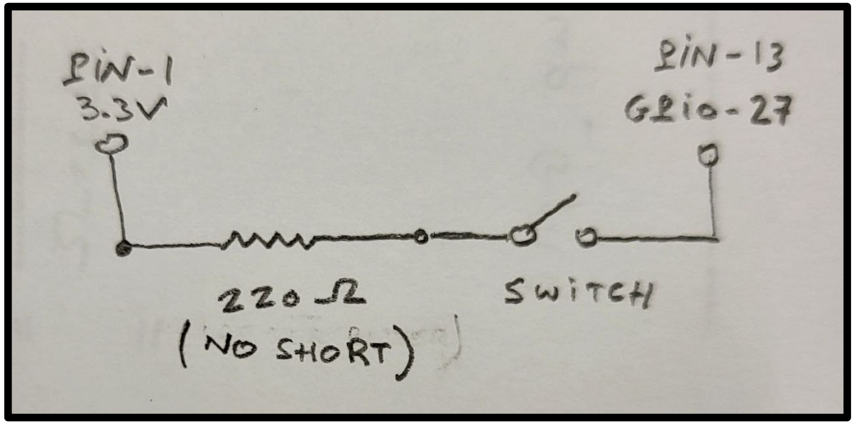
. reboot with with your new pin configuration

```
$> doas reboot
```

. check available pins

```
$> gpioctl gpio0 list               # check configured gpios
17: GPIO17                          # gpio.17 is configured for use
$> gpioctl gpio0 17 1               # turn gpio.17 on
$> gpioctl gpio0 17 0               # turn gpio.17 off
$> gpioctl gpio0 17 2               # toggle gpio.17 state
```

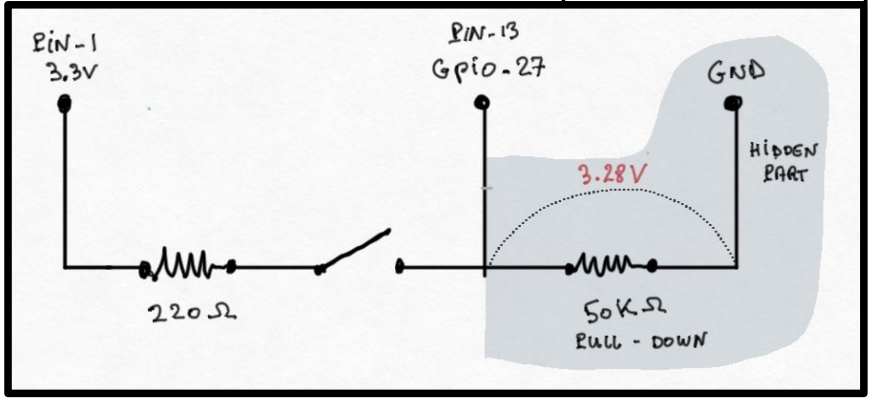
GPIO input pin - basic circuit



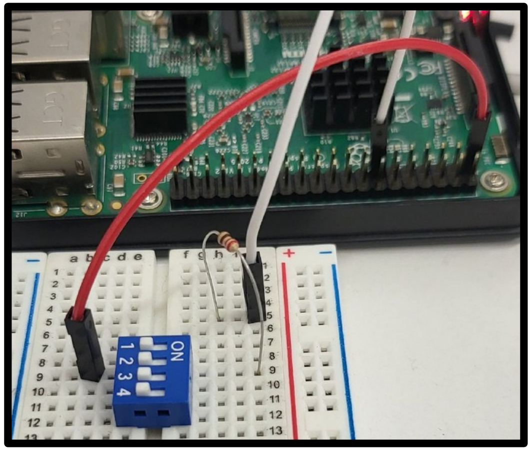
Scheme

(*) the resistor is not really necessary in this circuit because GPIO input pins have high impedance.

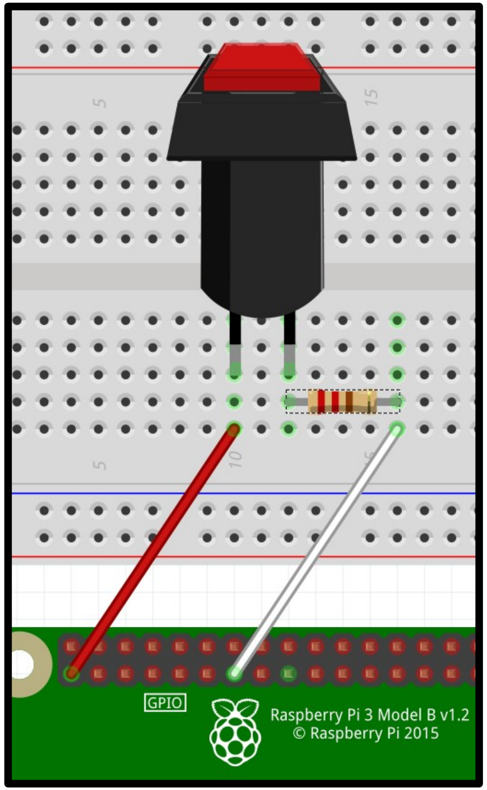
more datailed scheme



Reality



Fritzing



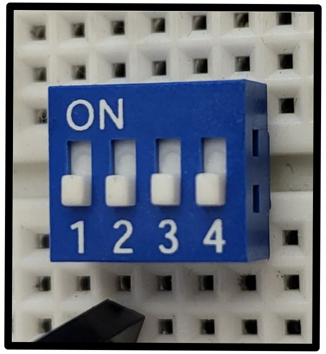
GPIO input ping configuration

```
---- /etc/rc.conf -----------------------
# . as before
gpio=YES
----------------------------------------------

------- /etc/gpio.conf -----------------
gpio0 27 set in pd    # gpio.17 will be a input pin, in pull-down.
                      # `pd`: equivalent to a ~50kΩ resistor to ground in
                      # parallel with the pin. Set to 0V the voltage
                      # of the pin when nothing is connected.
----------------------------------------------
```

read GPIO state with gpioctl

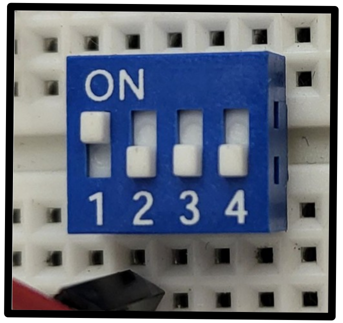
. reboot with with your new configuration
. check available pins




```
$> gpioctl gpio0 list          # check configured gpios
27: GPIO27                     # gpio.27 is configured

$> gpioctl gpio0 27            # read gpio.27 state
pin 27: state 0                # exit code ZERO

$> gpioctl gpio0 27            # read gpio.27 state
pin 27: state 1                # exit code ZERO
```



Section 3 | Manipulate GPIOs in **C** via **ioctl**

Objective-1. See how fast we can be with **gpiotcl** [~ 6.3 ms/op, 80Hz] [1]

Objective-2. Turn a pin ON or OFF with **ioctl** (bit-banging)

Objective-3. Read a pin state with **ioctl** (polling)

Objective-4. See how fast can we be with **ioctl** [~ 3.1 us/op, 160 kHz]

Objective-5. Fast GPIO → Square waves → many applications

CAVEAT. In this section we will be using our OS much like and **Arduino**, using as much CPU as we can. We are not taking into account other processes are running. Numbers found are just indicative. Square waves can be quite irregular.

[1] X ms/op: means X milliseconds per operation. An operation can be a GPIO state toggle or a GPIO read.

How fast can gpioctl be ?

--- Test approximate maximum write time --- [conf. as circuit.1]

. Count the time it takes to toggle (2) an output pin state 1_000 times.

$> time for i in `seq 1000`; do gpioctl -q gpio0 17 2; done

6.09s real   0.11s user   0.83s system

. => (1_000 / 6.09) = 164.20 toggles/sec ~ **150 toggles/sec**

. **Bit banging** from gpioctl can make up to ~ **80 Hz** square wave

--- Test approximate max read frequency --- [conf. as circuit.2]

. Count the time it takes to read an input pin state 1_000 times.

$> time for i in `seq 1000`; do gpioctl -q gpio0 27; done

6.15s real   0.09s user   0.91s system

. **Polling** maximum frequency, again, **~ 150 read/sec**

Write a pin state in C

```c
#include <stdio.h>        // See $> man 4 gpio
#include <fcntl.h>        // This file is named "c-write-pin-state.c", compile it with:
#include <sys/ioctl.h>    // $> gcc c-write-pin-state.c -o c-write-pin-state
#include <sys/gpio.h>     // To toggle pin state:
#include <string.h>       //   - change GPIOWRITE with GPIOTOGGLE
#include <stdlib.h>       //   - gp_value will be ignored
#include <err.h>
#include <unistd.h>
int main(int argc, char* argv[]) {
  int i, value, devfd = 0;
  struct gpio_req req;
  memset(&req, 0, sizeof(req));
  req.gp_pin = 17;
  req.gp_value = 1; // 1 on, 0 off
  char dev[] = "/dev/gpio0";
  if ((devfd = open(dev, O_RDWR)) == -1) { err(EXIT_FAILURE, "%s", dev); exit(1); }
  if (ioctl(devfd, GPIOWRITE, &req) == -1) { err(EXIT_FAILURE, "GPIOWRITE"); exit(1); }
  close(devfd);
}
```

see the doc gpio(4)

Read a pin state in C

```
#include <stdio.h>          // See $> man 4 gpio
#include <fcntl.h>          // This file is named "c-read-pin-state.c", compile it with:
#include <sys/ioctl.h>      // $> gcc c-read-pin-state.c -o c-read-pin-state
#include <sys/gpio.h>
#include <string.h>
#include <stdlib.h>
#include <err.h>
#include <unistd.h>
int main(int argc, char* argv[]) {
  int i, value, devfd = 0;
  struct gpio_req req;
  memset(&req, 0, sizeof(req));
  req.gp_pin = 27;
  char dev[] = "/dev/gpio0";
  if ((devfd = open(dev, O_RDWR)) == -1) { err(EXIT_FAILURE, "%s", dev); exit(1); }
  if (ioctl(devfd, GPIOREAD, &req) == -1) { err(EXIT_FAILURE, "GPIOREAD"); exit(1); }
  close(devfd);
  printf("%d\n", req.gp_value);
}
```

How fast are we in C ?

. let’s toggle 1 milion times the state of a pin

```
// file named: "c-toggle-speed-test.c", variation on "write-pin-state"
if ((devfd = open(dev, O_RDWR)) == -1) { err(EXIT_FAILURE, "%s", dev); exit(1); }
for(i = 0; i <= 1e6; i++) {
  if (ioctl(devfd, GPIOTOGGLE, &req) == -1) { err(EXIT_FAILURE, "GPIOTOGGLE"); exit(1); }
}
close(devfd);
```

$> time ./c-toggle-speed-test

3.10s real 0.04s user 3.03s system

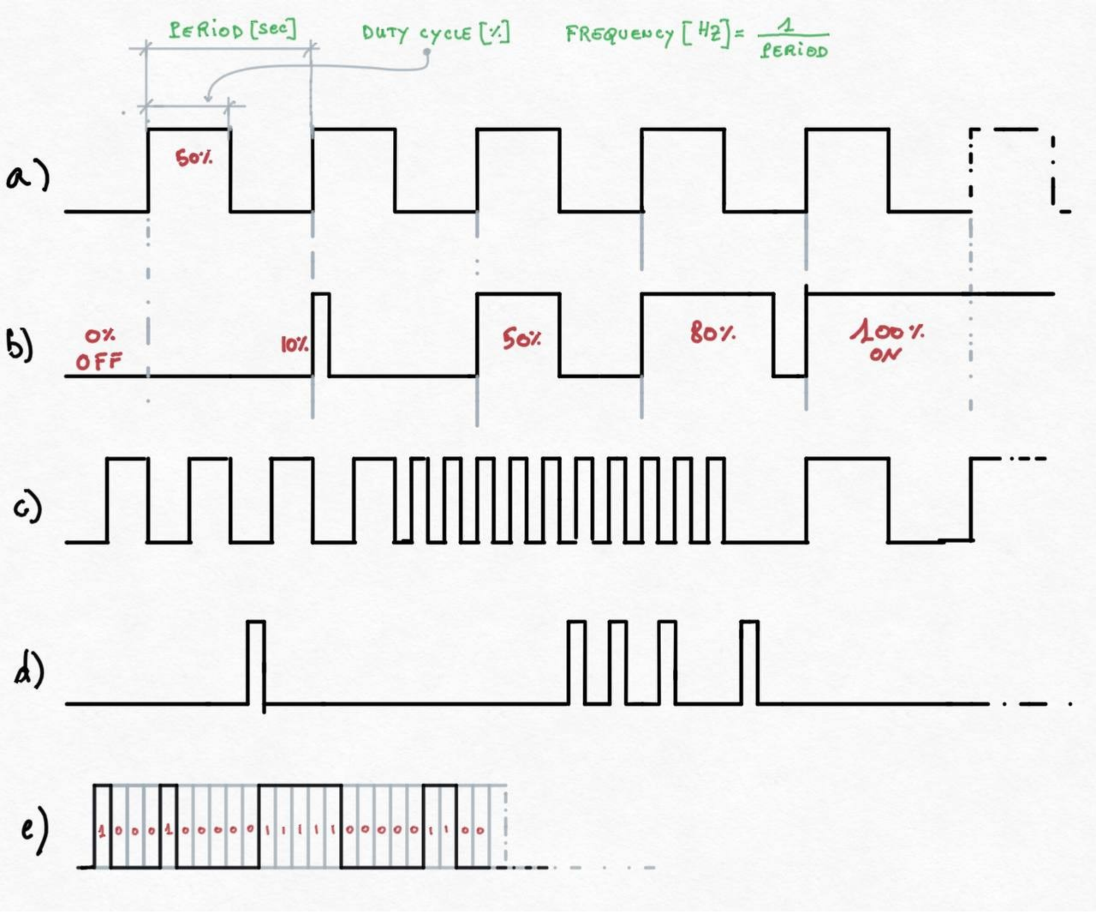
=> 1e6 / 3.10 = 322580.64 => ~ **320_000 toggles/sec**

=> **160 kHz** square wave, period **6.25 us**

. Looping for more iterations, 1e7 is enough, we observe:

=> The **load** on **top** at WCPU column is **98%** ! Reading ? The same.

From switches to square waves



[a] Square wave, duty-cycle 50%
[b] PWM. **Width modulation.** change the duty-cycle, e.g. dimming lights, DC motors speed, servo motor angle …
[c] Frequency modulation. e.g. sound, La (A) - 440 Hz .
[d] Sparse pulses. e.g. Lovato power meter in S0, 1 pulse of 20ms per each 0.01kWh consumed.
[e] Digital communication e.g. serial line, I2C …

Status

. We can read/write a pin status with **gpioctl** => simple, but slow

. We can read/write a pin status with **ioctl** => fast, resource intensive

. RPi3B+ has PWM hardware (no driver) => **bit-banging** (write fast, ioctl)

. RPI3B+ has interrupt support (no driver?) => **polling** (read fast, ioctl)

---- Possible mitigations [NetBSD specific] ----

. gpiopwm(4). Provide PWM on a GPIO pin (min pulse 1 tick: 10ms, max freq. 50 Hz)

. gpioirq (4). Provide IRQ support on a GPIO pin

Section 4 | be fast & lean

Objective-1. learn to use the **gpiopwm** driver (avoid bit-banging)

Objective-2. learn to use the **gpioirq** driver (avoid polling)

Application-1. **blinky**: intermittent LED via gpiopwm

Application-2. read a **push button** with gpioirq (visible bouncing)

Application-3. read a **sparse pulse** in loopback with gpioirq

gpiopwm | (alternative to bit-banging)

. **HARDER**. need to compile the kernel to have this **driver**

. change the GENERIC kernel configuration file as

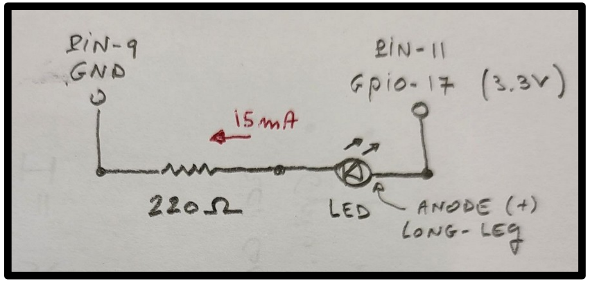
```
----- GENERIC --------------- [ i call the new file EBCON24-1]
gpio*         at gpiobus?
gpiopwm*      at gpio?              # === add this line
-----------------------------------------
```

. cross-compile the kernel image and put it in the RPi3B+

. reboot and check you booted with the right kernel

```
$> uname -a
NetBSD pulce4 10.0 NetBSD 10.0 (EBCON24-1) ....
```

gpiopwm application.1 - blinky



. configure the 17 pin as a PWM, same circuit as circuit.1 →

---- /etc/gpio.conf --------

```
gpio0 17 set out                  # see gpiopwm(4)
gpio0 attach gpiopwm 17 1         # 1 → we attach only gpio.17
```

. The gpiopwm ON and OFF times are expressed in term of system **ticks**

. See your tick frequency with $> sysctl kern.clockrate => hz=100

=> 1 tick = 10ms => to make an HIGH + LOW state we need
at least 2 ticks => 20ms. Then the minium
perdiod is 20ms and the max highest frequency is 50 Hz.

. We set our blinking LED to be 500ms ON and 500ms OFF:

```
$> doas sysctl -w hw.gpiopwm0.on=50
$> doas sysctl -w hw.gpiopwm0.off=50
```

Video here

gpioirq | (alternative to polling)

. gpioirq module is included in the default kernel, just load it at boot

```
----- /etc/rc.conf ---------------
modules=YES
-------------------------------------
------ /etc/modules.conf ----
gpioirq
-------------------------------------
```

. reboot and check you have the module loaded

```
$> modstat | grep gpioirq
gpioirq          driver   filesys  -     0     - gpio
```

. What gpioirq finds **must** be red from **/dev/gpioirq0**, 3 bytes per event

. bytes: [device unit, pin number[1], state of the pin]

[1] For GPIO27, `pin number` means 27.

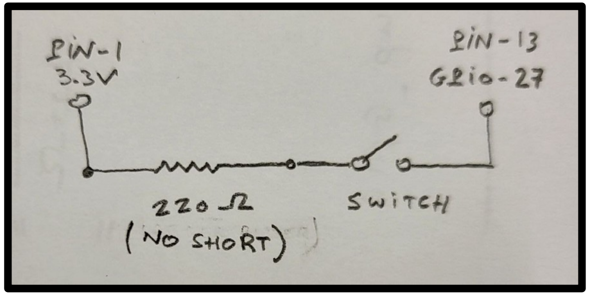
gpioirq - read gpioirq0

```c
#include <stdio.h>   // this file is called: c-read-gpioirq.c
#include <stdlib.h>  // compile with
#include <unistd.h>  // $> gcc c-read-gpioirq.c -o c-read-gpioirq
#include <err.h>
#include <fcntl.h>
int main() {
  int devfd;
  char dev[] = "/dev/gpioirq0";
  char buffer[3];
  printf("start reading : \n");
  if ((devfd = open(dev, O_RDONLY)) == -1) { err(EXIT_FAILURE, "%s", dev); exit(1); }
  while (1) {
    read(devfd, &buffer, 3);
    printf("%x, %x, %x \n", buffer[0], buffer[1], buffer[2]);
  }
  close(devfd);
}
```

$> ./c-read-gpioirq

gpioirq application.1 - push button

. circuit as circuit.2, change the switch with a push button



. tell the system that gpio27 is an input pin controlled by gpioirq

------ /etc/gpio.conf -------- # see gpioirq(4)
gpio0 27 set in pd # first 0x01 → only gpio27 controlled
gpio0 attach gpioirq 27 0x01 0x01 # second 0x01 → catch rising edges
----

. reboot & read gpioirq0 Video here

gpioirq application.2 - loopback

. We connect now an output pin to an input pin

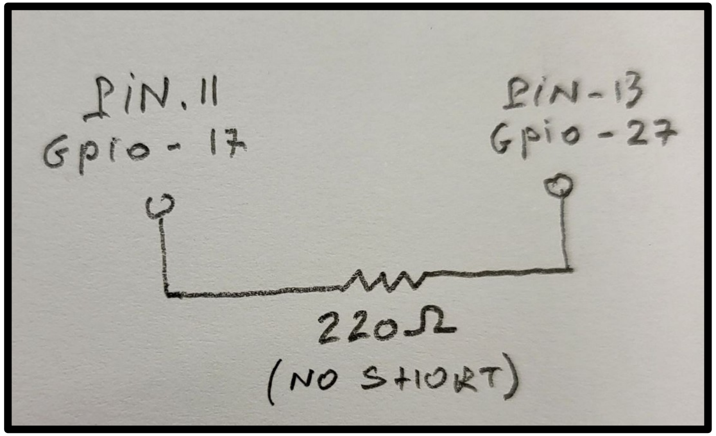
------ /etc/gpio.conf ------

```
gpio0 17 set out
gpio0 17 0
gpio0 27 set in pd
gpio0 attach gpioirq 27 0x01 0x01
```

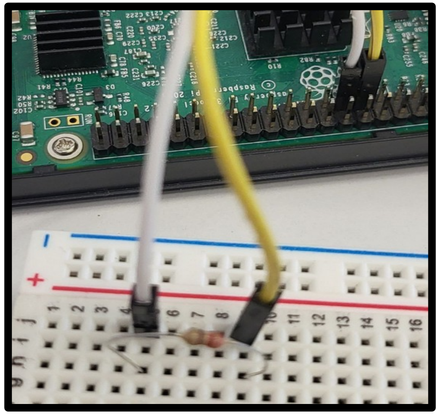
. reboot

. read /dev/gpioirq0 in a terminal and toggle the state of gpio.17 with gpioctl from another terminal.

Video here



(*) the resistor is not really necessary in this circuit because GPIO input pins have high impedance.



Exercises

Exercise.1. Redo blinky using gpioctl in a shell script. Change the blinking frequency. Verify that attempts to sleep for less then 10ms are ignored.
Exercise.2. Dim an LED brightness in progressive steps, as illustrated here.
Exercise.3. (HAL.2001) Make an LED light dim in ramp increasing and then decreasing, as illustrated here.
Exercise.4. (toggle the daemon). When the service **ntpd** [*] (or other of your choice) is ON an LED must be ON. Toggling a switch must toggle the state of ntpd. Read the state of the switch every 1 second. RATIO: ntpd produces a lot of annoying messages on the console when you boot and login without a working network, it might be practical to disable it with a physical switch.

SUGGESTIONS.
1. To dim LED Use a PWM of frequency 4kHz, this will avoid flickering
1.1 You can't use gpiopwm for 4 kHz since its maximum frequency is 50 Hz
1.2 You can't use nanosleep(3) in NetBSD to sleep for more than 10ms, loose time in another way, e.g. an empty loop.

Break

- At this point you know quite a bit about GPIO
- <u>Limitless applications adding very few extra components</u>

Section 5 | a real world application

Objective-1. Use built knowledge to make something real, here it is a wind-speed logger

Objective-2. Show the steps needed to build a working and usable embedded system prototype

The Wind-sped Logger (aka WSL)

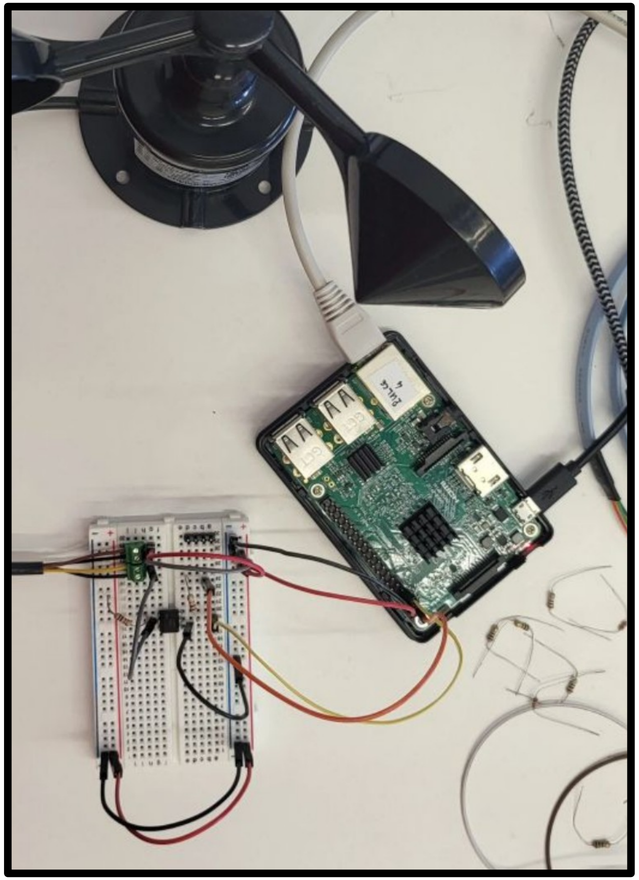
- I learnt GPIO in NetBSD to build this →
- Wind-speed logger specs
  - measures wind speed every 2s
  - computes 1min and 5min averages
  - saves data to a local db every 5 minutes
  - data accessible via HTTP
  - accessible in the local LAN or via VPN
  - powered via POE

WSL | The sensor

- The sensor available is this →
- The manufacturer lets us know that for each 360° turn the sensor will produce 20 pulses in output.
- The manufacturer tells us that 20 pulses per second => wind speed 1.75 m/s. Nothing more. Implied linearity
- We just need to count pulses, we saw how to do it, for example with **gpioirq**

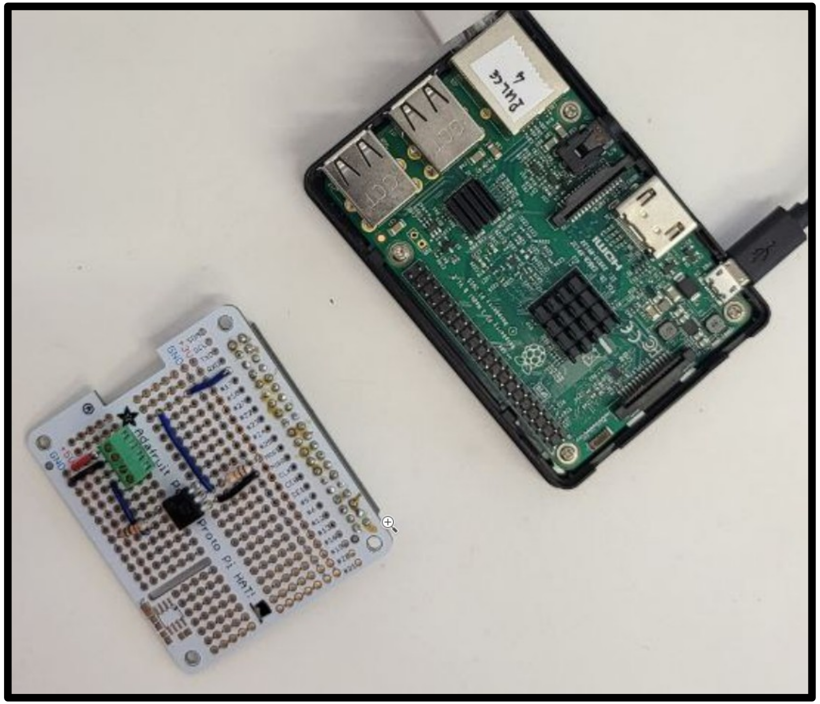
WSL | Soft prototype

- Test on the breadboard
- **problem**. electronics interfacing
  . this sensor can run at 5V not at 3V
  . it can’t communicate directly with RPi3B+, need to build a little circuit.



WSL | Sturdy prototype

- Once you are happy with the circuit you must make it resistant, maybe a **hat** for the RPI, if the circuit fits.
- My preference in prototyping goes to **solderable breadboard**
- Extremely practical for RPi is this Adafruit PermaProto HAT Mini.

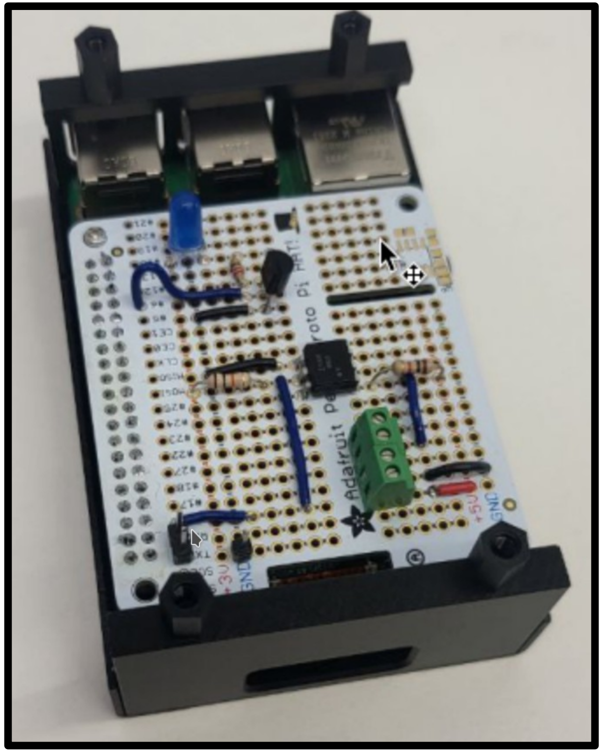


WSL | RPi case

- It is not easy to find a proper case,
  I landed on this one (Zkeeshop)

. You may need to do some adjustment to the case, see the **spacers** on the top.

. Keep in your lab a selection of **M2** and **M3** **screws** and **spacers**.



WSL | Housing

- I adapted a common industrial case for outdoor circuits.
- Added holes for ventilation.
- 3D printed gable vents to block rain from entering.

WSL | Software

- **Services**
  cron : general orchestration
  dhcpdcd : get address in the local LAN
  ntpd : keeps time updated
  mdnsd : reachable locally as pf-wind.local
  postgresql / sqlite : data is saved every 5 minutes in the local DB (cron)
  wireguard : reachable in a VPN of mine (run & check by cron)
  windspeed-service.rb : Program that collects data from gpioirq
  and makes it available via HTTP (run & check by cron)
- **User interface to the logger**
  $> curl http://pf-wind.local:8080/windspeed-json
  {"avg-speed-m/s-2-sec":0.61,"avg-speed-m/s-1-min":1.61,"avg-speed-m/s-5-min":1.38,"n.pulses-in-2-seconds":14,"loops-per-second":0.35}

Section 6 | conclusions

Subject-1. Where to learn more ?

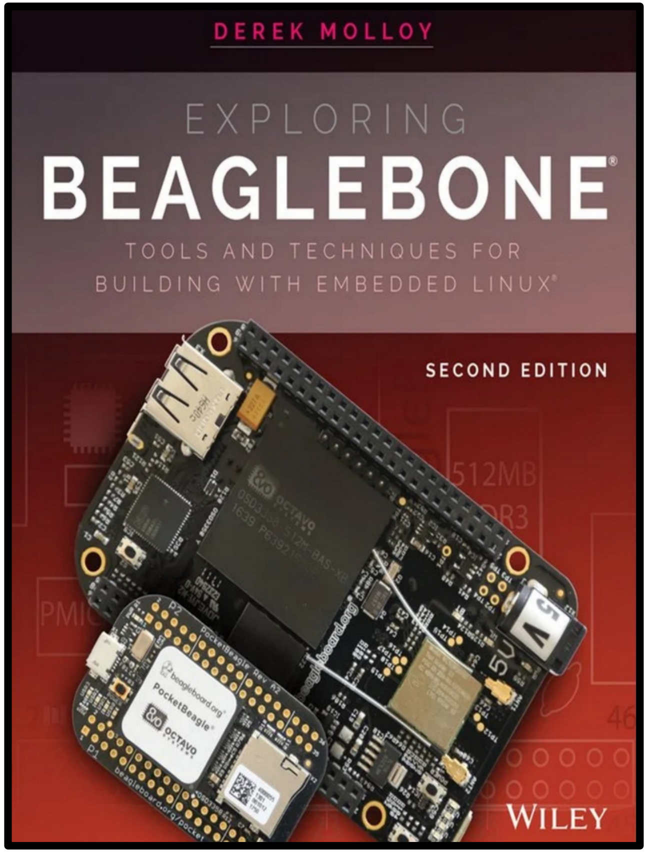
Subject-2. Why BSD ? why NetBSD ?

Where to learn more ?

- Best book I have red on the subject →

  Try in Linux on the BeagleBone then “export“ your knowledge to *BSD, and any SBC you like.

- NetBSD man pages and guide
- NetBSD ARM mailing list
- If you are very new you might start asking in Reddit or other social.



Why *BSD and NetBSD ?

- I built quite a few of these machines, mostly loggers, usually I use BeagleBone Black with Debian or RPi* with Raspian.
- One day, recently, **/sys/class/gpio** suddenly was removed from Linux. This annoyed me quite a bit. I decided then to try something alternative, that changes in a less astonishing way. (POLA principle)
- I am familiar with OpenBSD and FreeBSD. I like the BSD style, I love their **documentation** approach.
- I needed interrupt on GPIO, I found **gpioirq** in NetBSD so I chose it.
- The thing that surprised me most about NetBSD is that one is encouraged to change the system. In particular, NetBSD makes it quite easy to **cross compile**. For the first time I cross compiled for ARM under a x86. It was like going back to the early years of Linux, where things where less complex and compiling the kernel was a must. **A refreshing return to simplicity**.

The end