ES6

GPIO

Rick van Melis & Simon Lit

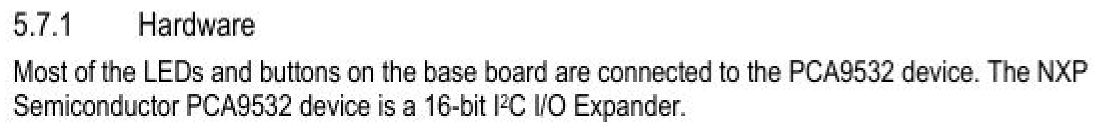
19-04-2019

Part 1 - Exploration

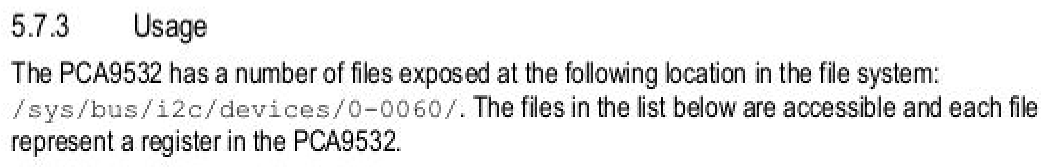
We started by trying to turn on the led's that connected through an I2C I/O expander. Quickly we figured that this is most likely not the way to go. So we focused on getting the led's working that on the OEM board which we quickly got working. At the same time, we figured out how to get reading the Joystick working. As for both of them the same registers/operations are used.

Led's through I2C I/O expander

LPC3250_Linux_BSP_Manual.pdf file page 35 states the following:

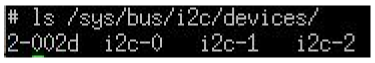

5.7.1 Hardware

Most of the LEDs and buttons on the base board are connected to the PCA9532 device. The NXP Semiconductor PCA9532 device is a 16-bit I²C I/O Expander.


5.7.3 Usage

The PCA9532 has a number of files exposed at the following location in the file system: `/sys/bus/i2c/devices/0-0060/`. The files in the list below are accessible and each file represent a register in the PCA9532.

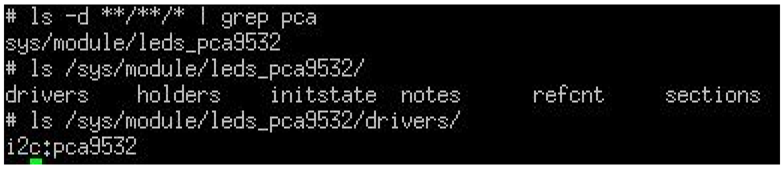
It appears that the LPC3250 configuration that we are using does not expose these device files through drivers. When scanning we received the following result:


```
# ls /sys/bus/i2c/devices/
2-002d  i2c-0   i2c-1   i2c-2
```

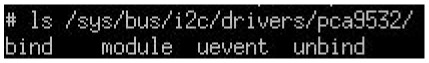
Since it is possible that the driver is simply not loaded we checked our Linux distribution that is used for cross-compiling the kernel. With locate we found the following file: `/felabs/sysdev/tinysystem/linux-2.6.34/drivers/leds/pca-9532.c`

After compiling this file we mounted the driver with `insmod`.
Unfortunately this did not return us the result we expected as the `/sys/bus/i2c/devices` were unchanged.

We found the driver mounted in 2 places:



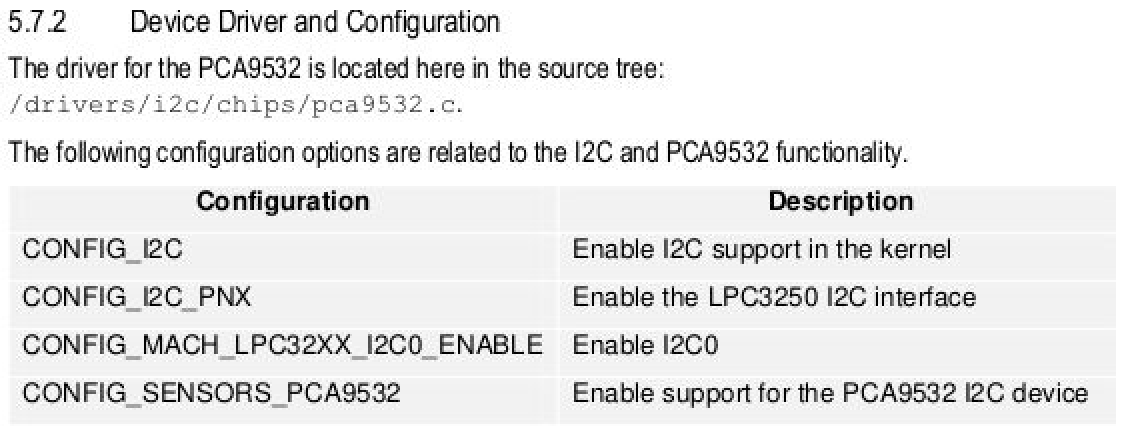
```
# ls -d **/**/* | grep pca
sys/module/leds_pca9532
# ls /sys/module/leds_pca9532/
drivers    holders    initstate  notes      refcnt     sections
# ls /sys/module/leds_pca9532/drivers/
i2c:pca9532
```



```
# ls /sys/bus/i2c/drivers/pca9532/
bind    module  uevent  unbind
```

These, however, were of no use to us.

The problem might be a configuration issue since the manual states the following configuration options which we were also unable to locate and adjust.



5.7.2 Device Driver and Configuration

The driver for the PCA9532 is located here in the source tree:
`/drivers/i2c/chips/pca9532.c`.

The following configuration options are related to the I2C and PCA9532 functionality.

Configuration	Description
CONFIG_I2C	Enable I2C support in the kernel
CONFIG_I2C_PNX	Enable the LPC3250 I2C interface
CONFIG_MACH_LPC32XX_I2C0_ENABLE	Enable I2C0
CONFIG_SENSORS_PCA9532	Enable support for the PCA9532 I2C device

Joystick

The following is said about how the joystick is connected to the development board by the assignment description. The joystick uses 5 GPIO-lines of the LPC3250 (pins P2.0, P2.1, P2.2, P2.3, and P2.4).

Looking at the datasheet we can see that these pins map to Port 2 of GPIO.

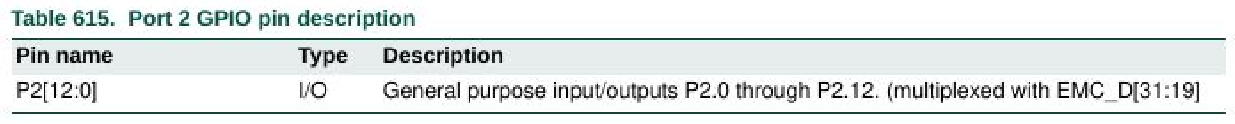


Table 615. Port 2 GPIO pin description

Pin name	Type	Description
P2[12:0]	I/O	General purpose input/outputs P2.0 through P2.12. (multiplexed with EMC_D[31:19]

The following diagram shows how the registers are internally connected and from that and the description we can figure out which registers we need. To be able to read the joystick we need to make use of the P2_INP_STATE, P2_DIR_SET_CLR, and P2_MUX_STATE

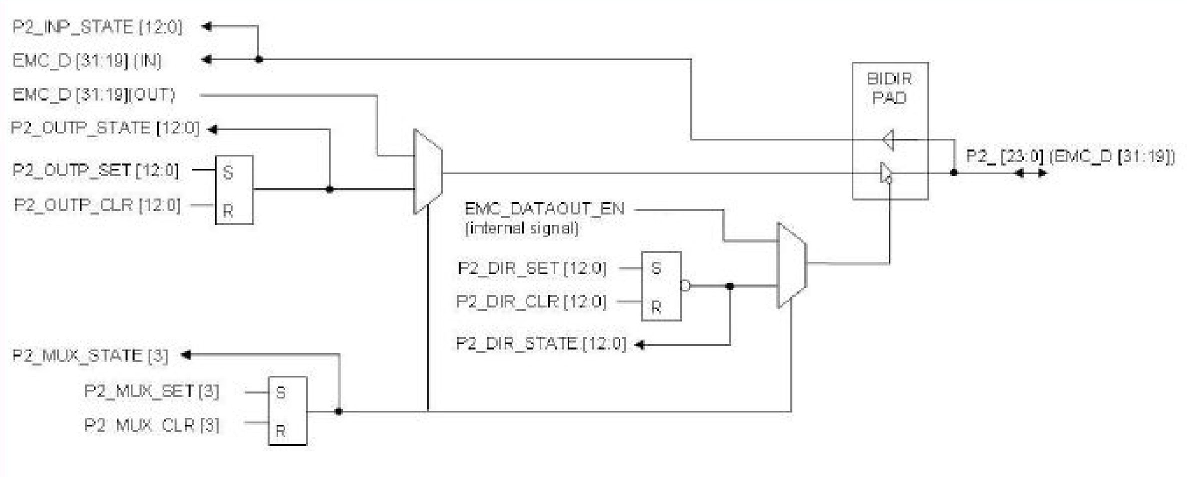
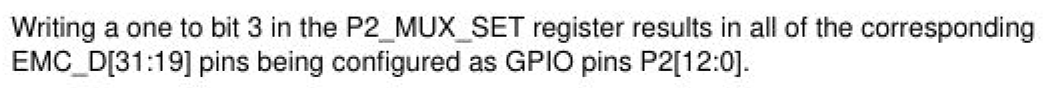


Fig 125. Port 2 EMC GPIO signals

The P2_INP_STATE reflects the actual level of the P2[12:0] pins. P2_OUT_SET/CLR control the actual level on the corresponding pins. The direction of the pins can be set with P2_DIR_SET/CLR and can be read in the P2_DIR_STATE register. This only applies when P2_MUX_STATE is set to GPIO.

Configure PORT 2 as a GPIO block is done as described in the following screengrab:


Writing a one to bit 3 in the P2_MUX_SET register results in all of the corresponding EMC_D[31:19] pins being configured as GPIO pins P2[12:0].

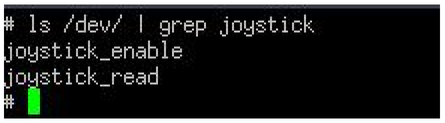
Thus the logic is as follows to read the Joystick:

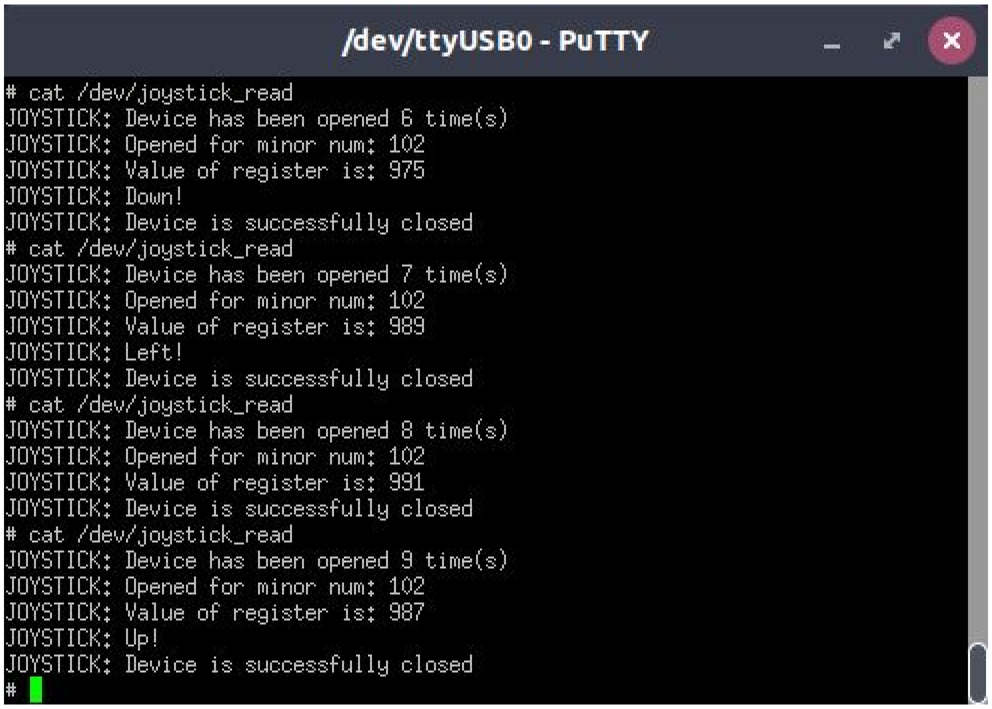
- Set a one to bit 3 in P2_MUX_SET (0x00000004)
- Set a one to bits [4:0] in P2_DIR_CLR to set direction as INPUT (0x0000001F)
- Read P2_INP_STATE register to read the input on the joystick

Register name	**Address**
P2_MUX_SET	0x40028028
P2_DIR_CLR	0x40028014
P2_INP_STATE	0x4002801C

Testing

For testing that the joystick works, we went a bit further than using our peek-and-poke module. We wrote a minimal driver, based on the design of the previous PWM driver. It has two nodes, one to enable the joystick and another to read if the joystick is clicked, left, up, right or in the down position. Respectively a number [4:0] is returned to the user space accessing the module. The screenshots below show the nodes in dev and the output in the terminal when pressing the joystick.


```
# ls /dev/ | grep joystick
joystick_enable
joystick_read
# 
```



```
# cat /dev/joystick_read
JOYSTICK: Device has been opened 6 time(s)
JOYSTICK: Opened for minor num: 102
JOYSTICK: Value of register is: 975
JOYSTICK: Down!
JOYSTICK: Device is successfully closed
# cat /dev/joystick_read
JOYSTICK: Device has been opened 7 time(s)
JOYSTICK: Opened for minor num: 102
JOYSTICK: Value of register is: 989
JOYSTICK: Left!
JOYSTICK: Device is successfully closed
# cat /dev/joystick_read
JOYSTICK: Device has been opened 8 time(s)
JOYSTICK: Opened for minor num: 102
JOYSTICK: Value of register is: 991
JOYSTICK: Device is successfully closed
# cat /dev/joystick_read
JOYSTICK: Device has been opened 9 time(s)
JOYSTICK: Opened for minor num: 102
JOYSTICK: Value of register is: 987
JOYSTICK: Up!
JOYSTICK: Device is successfully closed
#
```

The code for the joystick driver can be found in the included zip in “gpio/Joystick”.

Led’s on OEM board

The led’s on the OEM board are mapped to the following pins:

Pin	Description	GPO
P2.10	LED3	EMC_D29
P2.11	LED4	EMC_D30
P2.12	LED5	EMC_D31

Led 1 is a red led and is used for alarm, led2 is used for ...

The led’s 3 through 5 can be controlled by setting the correct bits on the P2_DIR_SET register to output. P2_MUX_SET is the same as the joystick, just setting the third bit to one. But instead of reading from P2_INP_STATE are we now going to write to P2_OUTP_SET to enable the led and P2_OUTP_CLR to reset the led.

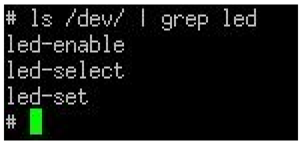
Thus the logic is as follows to turn on a led on the OEM board:

- Set a one to bit 3 in P2_MUX_SET (0x00000004)
- Set a one to bits [12:10] in P2_DIR_SET to set direction as INPUT (0x00001C00)
- Write a one to P2_OUTP_SET [12:10] register to turn a led on
- Write a one to P2_OUTP_CLR [12:10] register to turn a led off

Register name	Address
P2_MUX_SET	0x40028028
P2_DIR_SET	0x40028010
P2_OUTP_SET	0x40028020
P2_OUTP_CLR	0x40028024

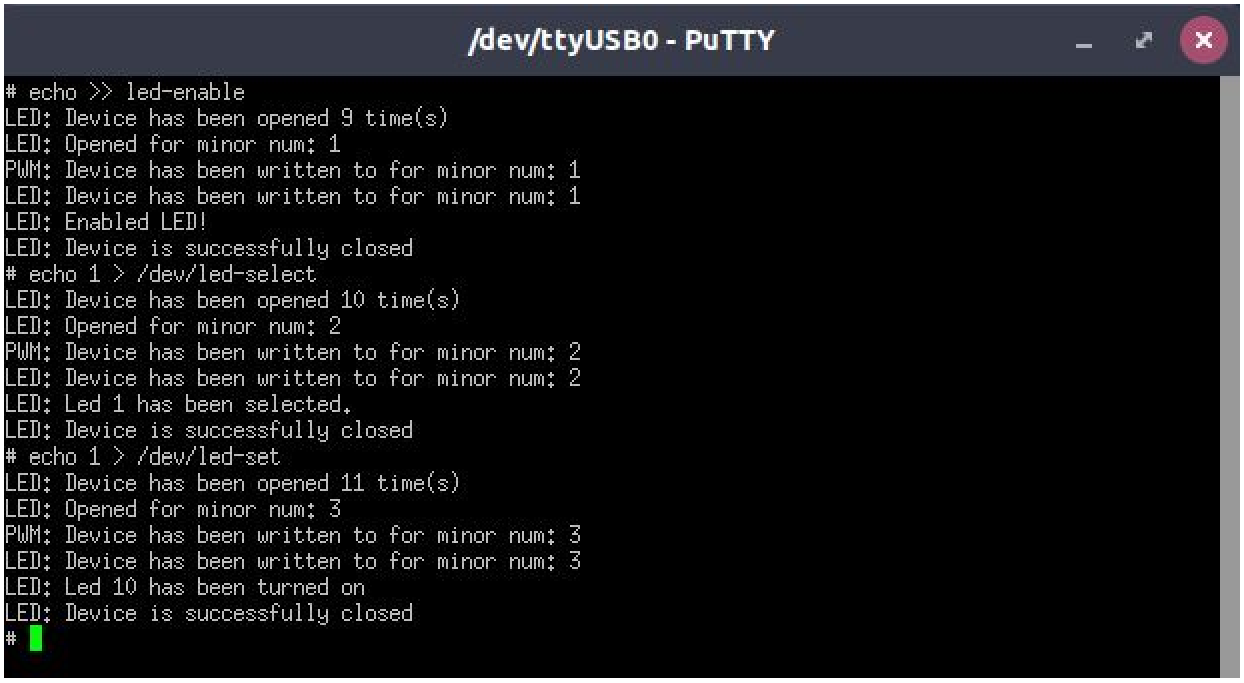
Testing

The nodes in `/dev`



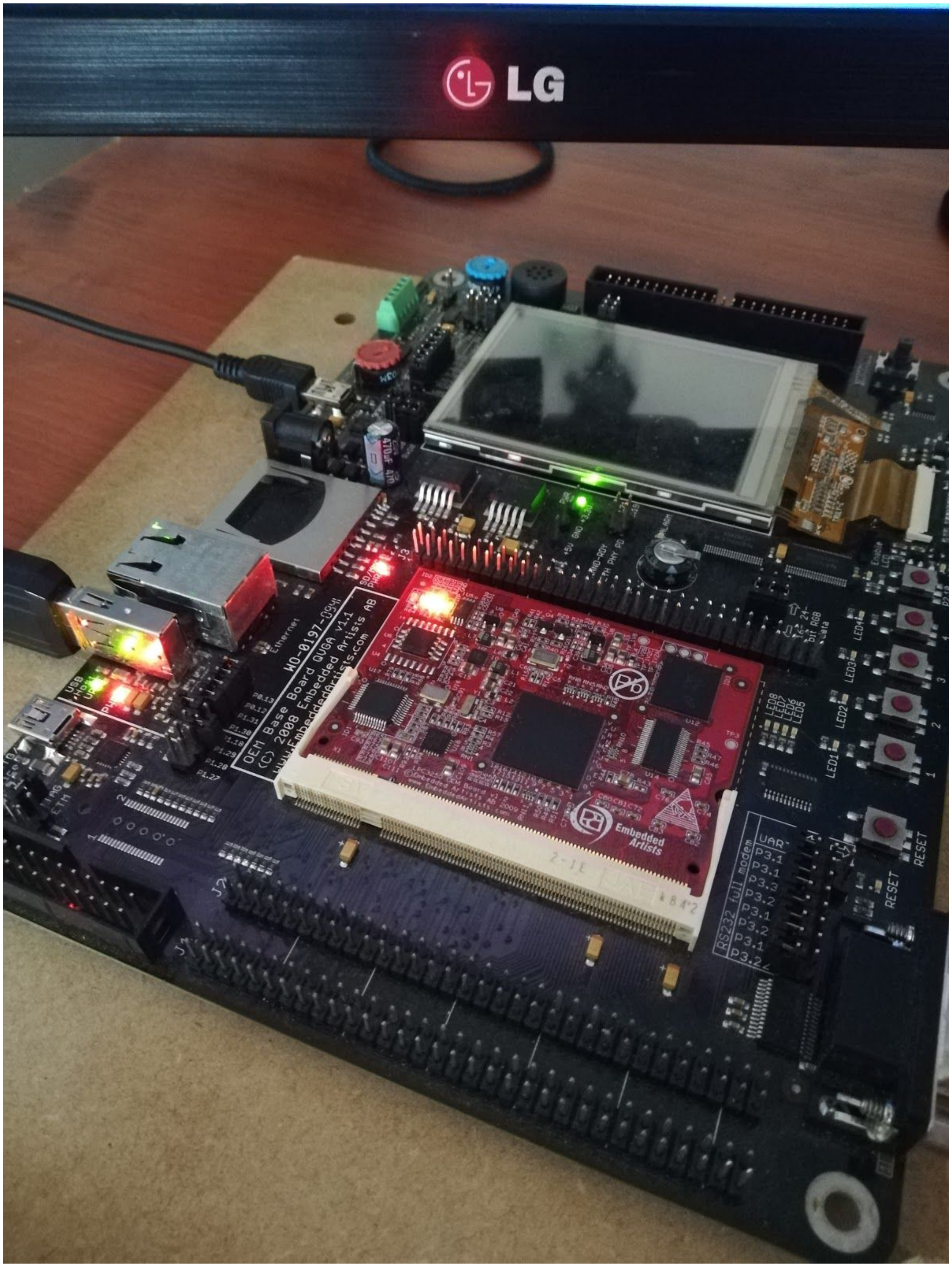
```
# ls /dev/ | grep led
led-enable
led-select
led-set
#
```

First we enable the leds then we select which one we want to turn on and then we set the led to high.



```
# echo >> led-enable
LED: Device has been opened 9 time(s)
LED: Opened for minor num: 1
PWM: Device has been written to for minor num: 1
LED: Device has been written to for minor num: 1
LED: Enabled LED!
LED: Device is successfully closed
# echo 1 > /dev/led-select
LED: Device has been opened 10 time(s)
LED: Opened for minor num: 2
PWM: Device has been written to for minor num: 2
LED: Device has been written to for minor num: 2
LED: Led 1 has been selected.
LED: Device is successfully closed
# echo 1 > /dev/led-set
LED: Device has been opened 11 time(s)
LED: Opened for minor num: 3
PWM: Device has been written to for minor num: 3
LED: Device has been written to for minor num: 3
LED: Led 10 has been turned on
LED: Device is successfully closed
#
```

Small video showing the leds in working: https://youtu.be/nOheirUo-48



Part 2 - Design and implementation

Existing Implementations

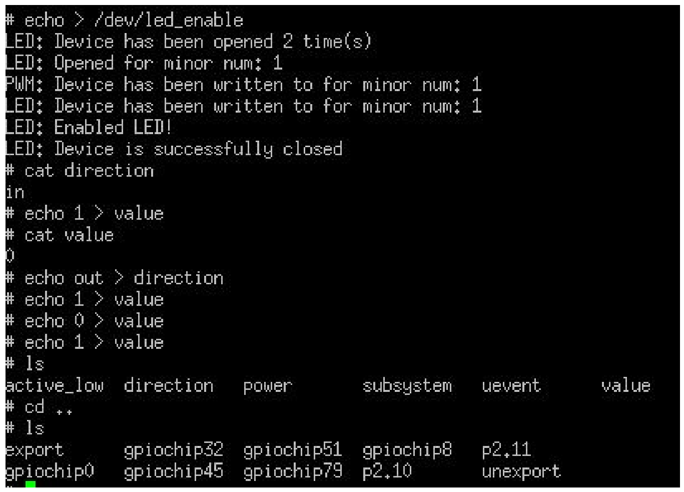
Linux offers a sysfs GPIO driver that uses PINCTL to map pins on corresponding pin controllers so that pins can be handled in an abstract way. Each type of pin controller will have its own functions to handle the pins. It is also in PINCTL where constraints are handled, pinMuxes, multiplexes, etc.

From PINCTRL

https://www.kernel.org/doc/html/v4.17/driver-api/pinctl.html

- Registers (or fields within registers) that control electrical properties of the pin such as biasing and drive strength should be exposed through the pinctrl subsystem, as “pin configuration” settings.
- Registers (or fields within registers) that control muxing of signals from various other HW blocks (e.g. I2C, MMC, or GPIO) onto pins should be exposed through the pinctrl subsystem, as mux functions.
- Registers (or fields within registers) that control GPIO functionality such as setting a GPIO’s output value, reading a GPIO’s input value, or setting GPIO pin direction should be exposed through the GPIO subsystem, and if they also support interrupt capabilities, through the irqchip abstraction.

To the right is an example of using sysfs gpio with our /dev driver to set the mux. We used some of the design principles behind this system to design our own, simplified and customized driver.



```
# echo > /dev/led_enable
LED: Device has been opened 2 time(s)
LED: Opened for minor num: 1
PWM: Device has been written to for minor num: 1
LED: Device has been written to for minor num: 1
LED: Enabled LED!
LED: Device is successfully closed
# cat direction
in
# echo 1 > value
# cat value
0
# echo out > direction
# echo 1 > value
# echo 0 > value
# echo 1 > value
# ls
active_low  direction  power      subsystem  uevent     value
# cd ..
# ls
export      gpiochip32  gpiochip51  gpiochip8   p2.11
gpiochip0   gpiochip45  gpiochip79  p2.10       unexport
```

Design

The goal of the driver is to expose as many pins as possible to the user by a simple interface. To achieve this we design a system that uses sysfs for configuring a pin as an output or input and setting the required bits and bytes in configuration registers for these pins to be accessible through the expansion connectors on the board. We use devfs for actual setting output to high or low for reading the current value of a pin.

For implementation, we designed a struct that contains all the information needed for setting and reading the registers, the mapping of the expansion connectors to the four internal ports of the board as well which operations can be executed on the pins. For each port on the board, we have an instance of this struct. These are defined in a header which allows us to include one implementation of the struct into both the sysfs and devfs driver.

```
struct Pinctrl {
   struct Registers registers;
   unsigned int npins;
   struct Pin pins[PINS_PER_CONNECTOR];
   Get_direction get_direction;
   Set_direction set_direction;
   Get_value get_value;
   Set_value set_value;
   Init init;
};
```

This struct contains a struct of registers, array of pins and function pointers to operations that are possible on the pin controller.

```
struct Pin {
   const char* connector;
   unsigned int pin;
   unsigned int index_read;
   unsigned int index_write;
   struct Pinctrl *pinctrl; // pointer to parent
};
```

A pin has a connector and pin number (par exemple J3 51) and an index what tells the possible operations which bit in the registers need to be set. And as last each pin contains a pointer to the parent pin controller. This allows us to search for a pin an return a reference to the pin and all data that is needed can be accessed this way.

Interface

The direction of the GPIO can be set through the sysfs driver, the syntax for this is as follows: `echo J3 40 O >> /sys/kernel/gpio/config`

The value of the GPIO can be set and read through the devfs driver as follows:

Write `echo J3 40 1 >>  /dev/gpio-set`

Read `echo J3 40 >>  /dev/gpio_read && cat /dev/gpio-read`

GPIO Pins

The goal of the driver is by using sysfs and devfs to be able to use all of the GPIO pins that are available on the Jx connectors on the board. These connectors are accessible through port 0,1,2 and 3.

Port 0

Port zero has eight GPIO's multiplexed with the LCD. Because of this the LCD needs to be turned off to be able to use these pins. This can be done by setting by setting the register the LcdEn bit to zero in register LCD_CTRL.

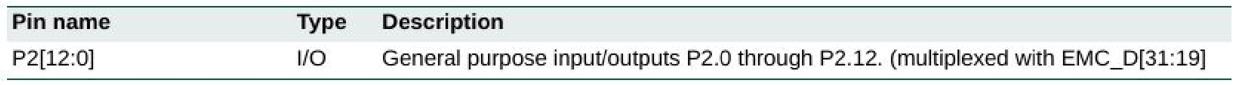
Pin name	Type	Description
p0[7:0]	I/O	General purpose input/outputs P0.0 through P0.7.

Port 1

Nothing on Port 1 is available because this port is used by SDRAM, if not SDRAM is connected these pins would be available. The SDRAM connected is a 16 bits bus. If it was a 32bit bus port 2 would not be available either.

Pin name	Type	Description
P1[23:0]	I/O	General purpose input/outputs P1.0 through P1.23. (multiplexed with EMC_A[23:0])

Port 2

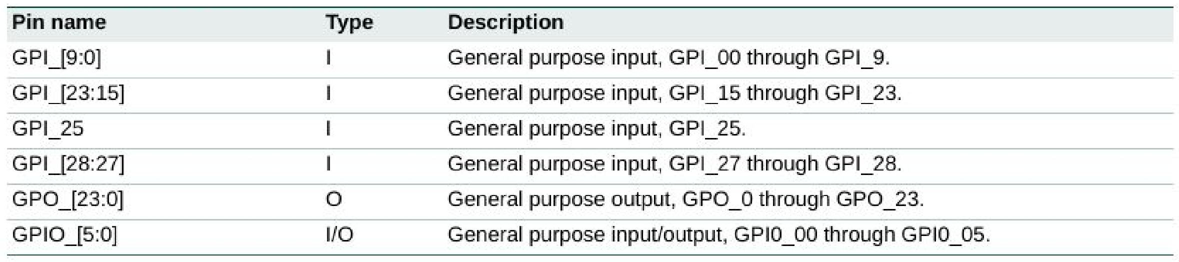


Pin name	Type	Description
P2[12:0]	I/O	General purpose input/outputs P2.0 through P2.12. (multiplexed with EMC_D[31:19]

By setting the MUX register the EMC_D[31:19] can be configured as GPIO pins.

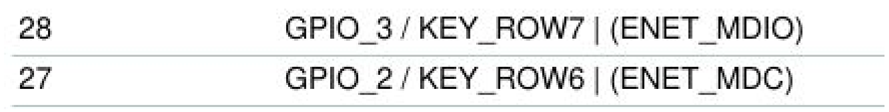
LPC3250	**SODIMM**	**J Connector**
P2.0	X1-120	J3 Pin 47
P2.1	X1-121	J3 Pin 56
P2.2	X1-122	j3 Pin 48
P2.3	X1-123	J3 Pin 57
P2.4	X1-124	J3 Pin 49
P2.5	X1-125	J3 Pin 58
P2.6	X1-126	J3 Pin 50
P2.7	X1-116	J3 Pin 45
P2.8	X1-176	J1 Pin 49
P2.9	X1-178	J1 Pin 50
P2.10	X1-180	J1 Pin 51
P2.11	X1-182	J1 Pin 52
P2.12	X1-184	J1 Pin 53

Port 3



Pin name	Type	Description
GPI_[9:0]	I	General purpose input, GPI_00 through GPI_9.
GPI_[23:15]	I	General purpose input, GPI_15 through GPI_23.
GPI_25	I	General purpose input, GPI_25.
GPI_[28:27]	I	General purpose input, GPI_27 through GPI_28.
GPO_[23:0]	O	General purpose output, GPO_0 through GPO_23.
GPIO_[5:0]	I/O	General purpose input/output, GPI0_00 through GPI0_05.

Port three does not need anything like a Mux set. There are six GPIO pins on this port, although only 4 of them are available because of because two of the ports are connected are used by the ethernet controller. As can be read here:



28	GPIO_3 / KEY_ROW7 | (ENET_MDIO)
27	GPIO_2 / KEY_ROW6 | (ENET_MDC)

LPC3250	**SODIMM**	**J Connector**
P3.25	X1-117	J3 Pin 54
P3.26	X1-118	J3 Pin 46
P3.27	/	/
P3.28	/	/
P3.29	X1-96	J3 Pin 36
P3.30	X1-85	J1 Pin 24

Number of pins

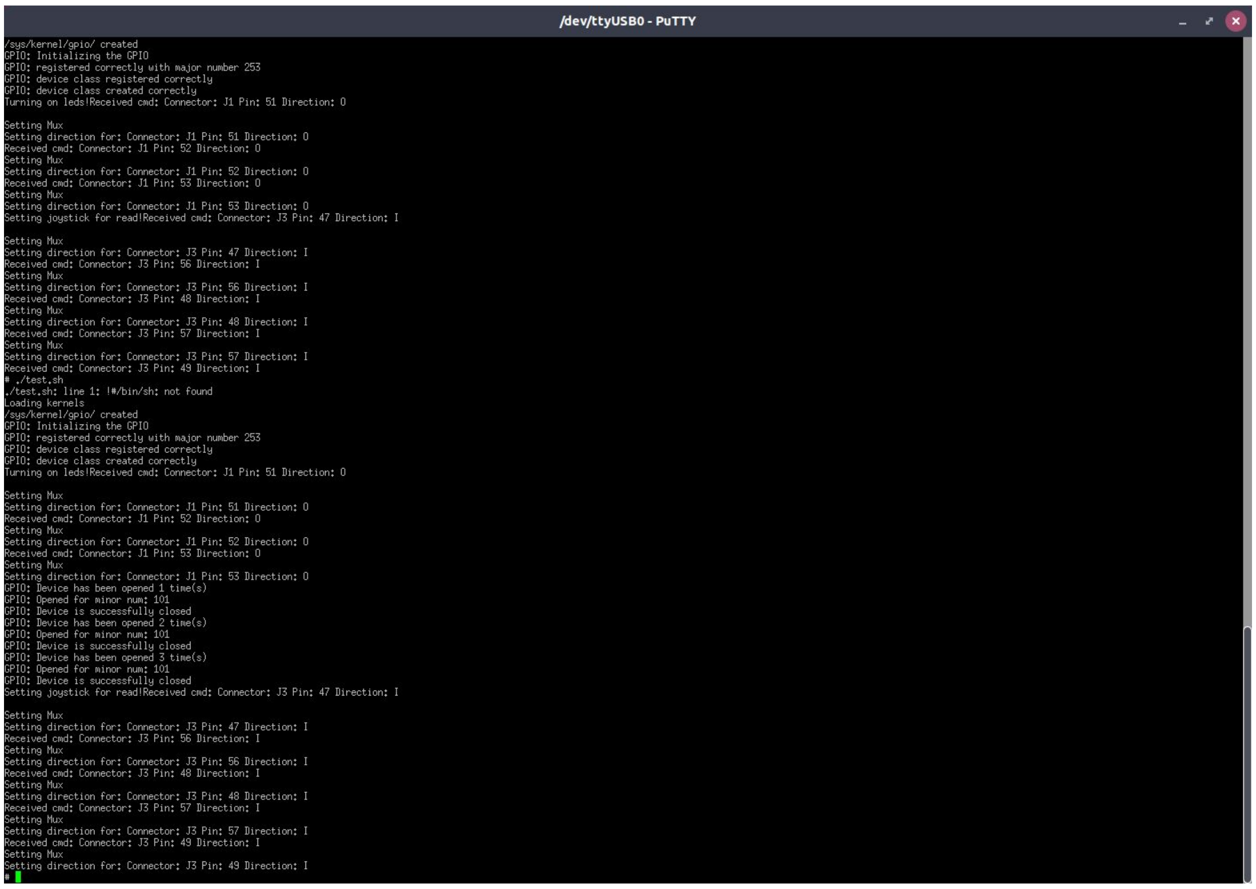
In total we expect to be able to have 25 GPIO pins available and working though our driver. Eight pins from port zero, thirteen from port two and four from port three.

Part 3 - Testing the driver

Testing of the driver is done in a few ways. First we test if pins of every port work so we test the joystick and leds that are connected to port two, we test if we can get a high signal on our GPIO's pins of the other ports with a logic analyzer. And finally we do a few tests to see how our driver handles incorrect input and request pins that do not exist.

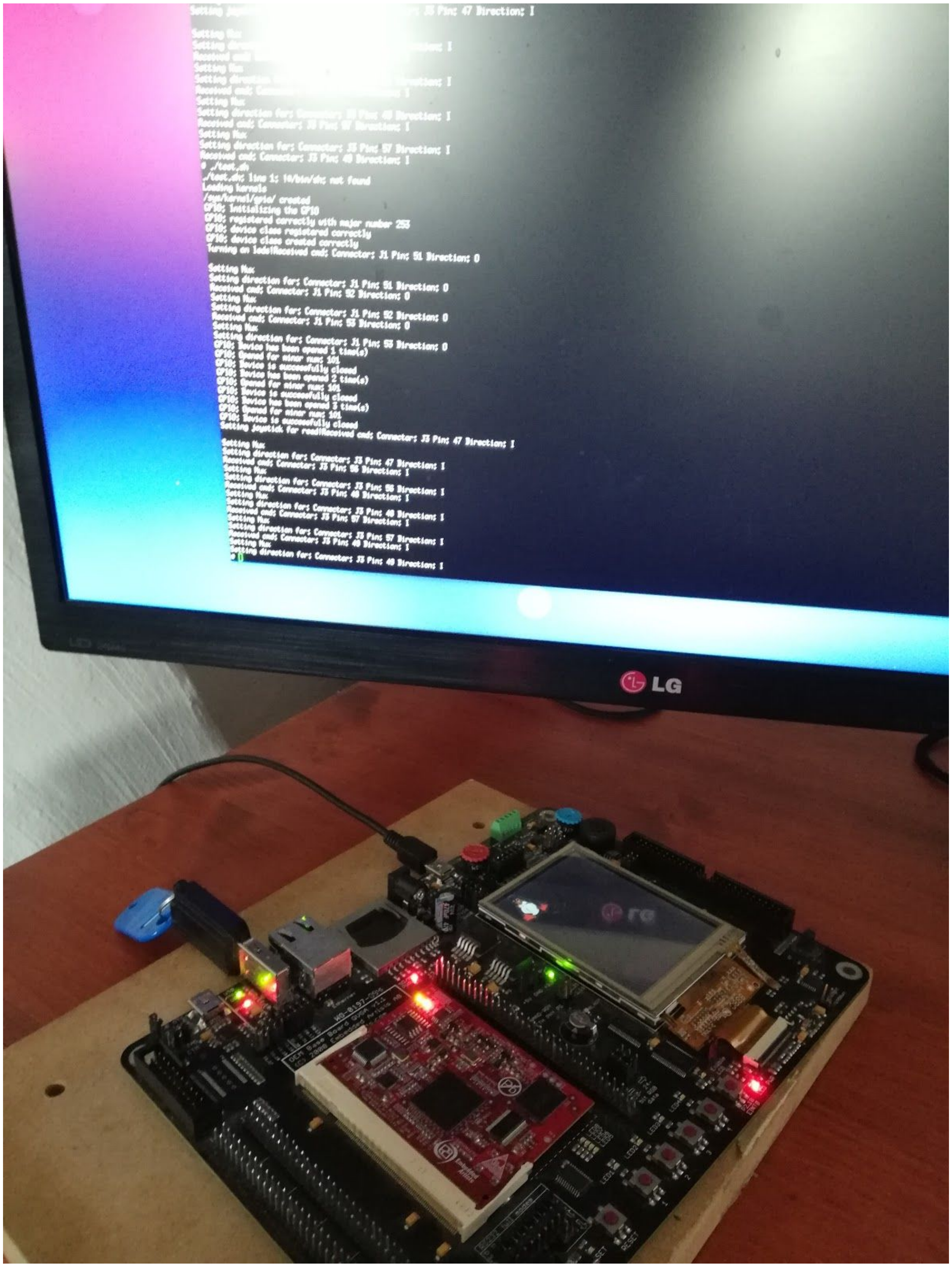
Port 2

Port two we tested with a script that turns on the three leds on the oem board and enables the joystick for reading. Below is a screenshot of the output in our terminal as well as a video of running the script and an photo of the board after turning everything on.



```
/sys/kernel/gpio/ created
GPIO: Initializing the GPIO
GPIO: registered correctly with major number 253
GPIO: device class registered correctly
GPIO: device class created correctly
Turning on leds!Received cmd: Connector: J1 Pin: 51 Direction: 0

Setting Mux
Setting direction for: Connector: J1 Pin: 51 Direction: 0
Received cmd: Connector: J1 Pin: 52 Direction: 0
Setting Mux
Setting direction for: Connector: J1 Pin: 52 Direction: 0
Received cmd: Connector: J1 Pin: 53 Direction: 0
Setting Mux
Setting direction for: Connector: J1 Pin: 53 Direction: 0
Setting joystick for read!Received cmd: Connector: J3 Pin: 47 Direction: I

Setting Mux
Setting direction for: Connector: J3 Pin: 47 Direction: I
Received cmd: Connector: J3 Pin: 56 Direction: I
Setting Mux
Setting direction for: Connector: J3 Pin: 56 Direction: I
Received cmd: Connector: J3 Pin: 48 Direction: I
Setting Mux
Setting direction for: Connector: J3 Pin: 48 Direction: I
Received cmd: Connector: J3 Pin: 57 Direction: I
Setting Mux
Setting direction for: Connector: J3 Pin: 57 Direction: I
Received cmd: Connector: J3 Pin: 49 Direction: I
# ./test.sh
./test.sh: line 1: !#/bin/sh: not found
Loading kernels
/sys/kernel/gpio/ created
GPIO: Initializing the GPIO
GPIO: registered correctly with major number 253
GPIO: device class registered correctly
GPIO: device class created correctly
Turning on leds!Received cmd: Connector: J1 Pin: 51 Direction: 0

Setting Mux
Setting direction for: Connector: J1 Pin: 51 Direction: 0
Received cmd: Connector: J1 Pin: 52 Direction: 0
Setting Mux
Setting direction for: Connector: J1 Pin: 52 Direction: 0
Received cmd: Connector: J1 Pin: 53 Direction: 0
Setting Mux
Setting direction for: Connector: J1 Pin: 53 Direction: 0
GPIO: Device has been opened 1 time(s)
GPIO: Opened for minor num: 101
GPIO: Device is successfully closed
GPIO: Device has been opened 2 time(s)
GPIO: Opened for minor num: 101
GPIO: Device is successfully closed
GPIO: Device has been opened 3 time(s)
GPIO: Opened for minor num: 101
GPIO: Device is successfully closed
Setting joystick for read!Received cmd: Connector: J3 Pin: 47 Direction: I

Setting Mux
Setting direction for: Connector: J3 Pin: 47 Direction: I
Received cmd: Connector: J3 Pin: 56 Direction: I
Setting Mux
Setting direction for: Connector: J3 Pin: 56 Direction: I
Received cmd: Connector: J3 Pin: 48 Direction: I
Setting Mux
Setting direction for: Connector: J3 Pin: 48 Direction: I
Received cmd: Connector: J3 Pin: 57 Direction: I
Setting Mux
Setting direction for: Connector: J3 Pin: 57 Direction: I
Received cmd: Connector: J3 Pin: 49 Direction: I
Setting Mux
Setting direction for: Connector: J3 Pin: 49 Direction: I
#
```

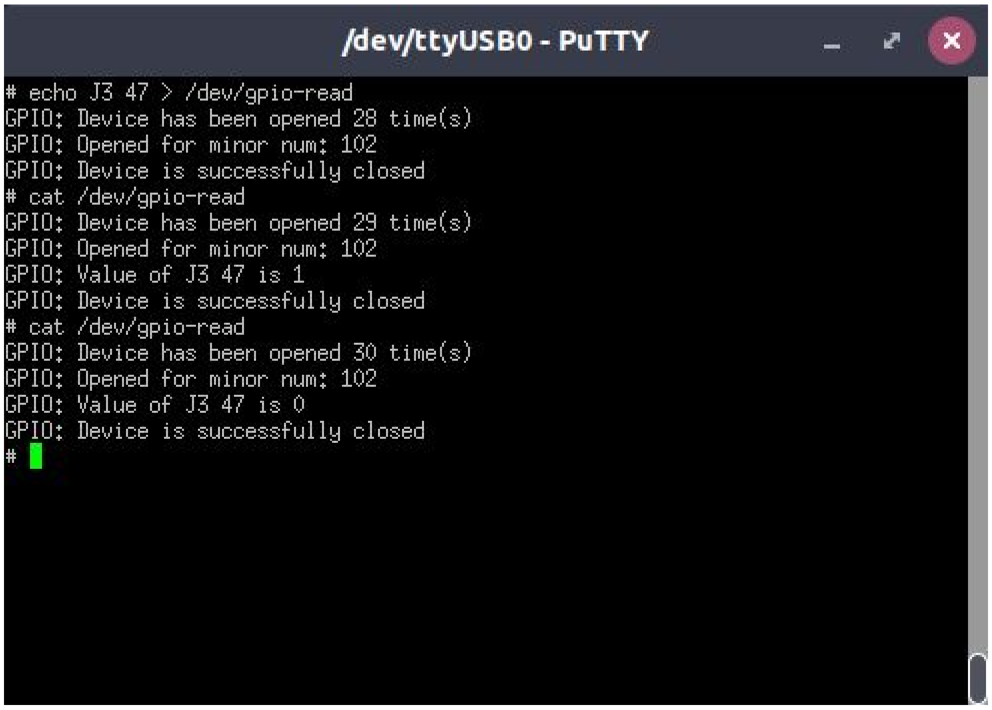
The video: https://youtu.be/fQE2E3F0pwM



```
Setting joy[illegible]: J3 Pin: 47 Direction: I

Setting Mux
Setting dir[illegible]ection: I
Received cmd[illegible]
Setting Mux
Setting direction [illegible]irection: I
Received cmd: Conn[illegible]ection: I
Setting Mux
Setting direction for: Connector: J3 Pin: 49 Direction: I
Received cmd: Connector: J3 Pin: 57 Direction: I
Setting Mux
Setting direction for: Connector: J3 Pin: 57 Direction: I
Received cmd: Connector: J3 Pin: 49 Direction: I
# ./test.sh
./test.sh: line 1: !#/bin/sh: not found
Loading kernels
/sys/kernel/gpio/ created
GPIO: Initializing the GPIO
GPIO: registered correctly with major number 253
GPIO: device class registered correctly
GPIO: device class created correctly
Turning on leds!Received cmd: Connector: J1 Pin: 51 Direction: O

Setting Mux
Setting direction for: Connector: J1 Pin: 51 Direction: O
Received cmd: Connector: J1 Pin: 52 Direction: O
Setting Mux
Setting direction for: Connector: J1 Pin: 52 Direction: O
Received cmd: Connector: J1 Pin: 53 Direction: O
Setting Mux
Setting direction for: Connector: J1 Pin: 53 Direction: O
GPIO: Device has been opened 1 time(s)
GPIO: Opened for minor num: 101
GPIO: Device is successfully closed
GPIO: Device has been opened 2 time(s)
GPIO: Opened for minor num: 101
GPIO: Device is successfully closed
GPIO: Device has been opened 3 time(s)
GPIO: Opened for minor num: 101
GPIO: Device is successfully closed
Setting joystick for read!Received cmd: Connector: J3 Pin: 47 Direction: I

Setting Mux
Setting direction for: Connector: J3 Pin: 47 Direction: I
Received cmd: Connector: J3 Pin: 56 Direction: I
Setting Mux
Setting direction for: Connector: J3 Pin: 56 Direction: I
Received cmd: Connector: J3 Pin: 48 Direction: I
Setting Mux
Setting direction for: Connector: J3 Pin: 48 Direction: I
Received cmd: Connector: J3 Pin: 57 Direction: I
Setting Mux
Setting direction for: Connector: J3 Pin: 57 Direction: I
Received cmd: Connector: J3 Pin: 49 Direction: I
Setting Mux
# Setting direction for: Connector: J3 Pin: 49 Direction: I
```

And for port two we have as well a clip for reading the joystick:

https://youtu.be/8ZiKEXGpXeg



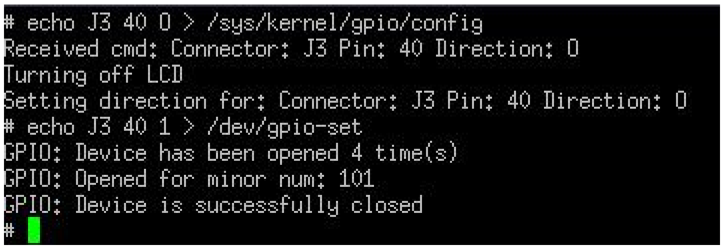
```
# echo J3 47 > /dev/gpio-read
GPIO: Device has been opened 28 time(s)
GPIO: Opened for minor num: 102
GPIO: Device is successfully closed
# cat /dev/gpio-read
GPIO: Device has been opened 29 time(s)
GPIO: Opened for minor num: 102
GPIO: Value of J3 47 is 1
GPIO: Device is successfully closed
# cat /dev/gpio-read
GPIO: Device has been opened 30 time(s)
GPIO: Opened for minor num: 102
GPIO: Value of J3 47 is 0
GPIO: Device is successfully closed
#
```

Port 0

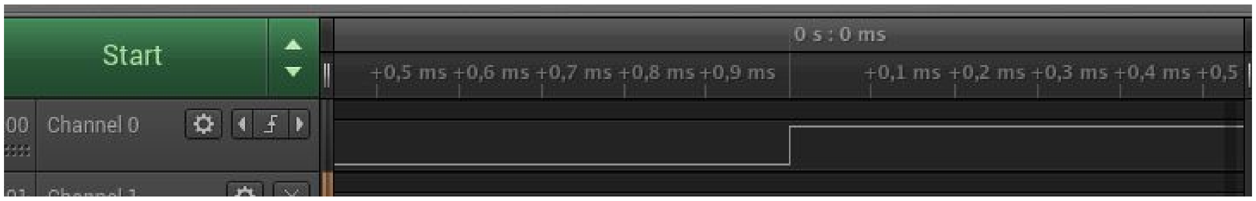
Because we need to turn off the lcd here is a small video showing that

https://youtu.be/PQP-G1o2ffk

Also a small video of turning the pin to high and that is detected by our logic analyzer that starts sampling on a positive flank.



```
# echo J3 40 0 > /sys/kernel/gpio/config
Received cmd: Connector: J3 Pin: 40 Direction: 0
Turning off LCD
Setting direction for: Connector: J3 Pin: 40 Direction: 0
# echo J3 40 1 > /dev/gpio-set
GPIO: Device has been opened 4 time(s)
GPIO: Opened for minor num: 101
GPIO: Device is successfully closed
#
```

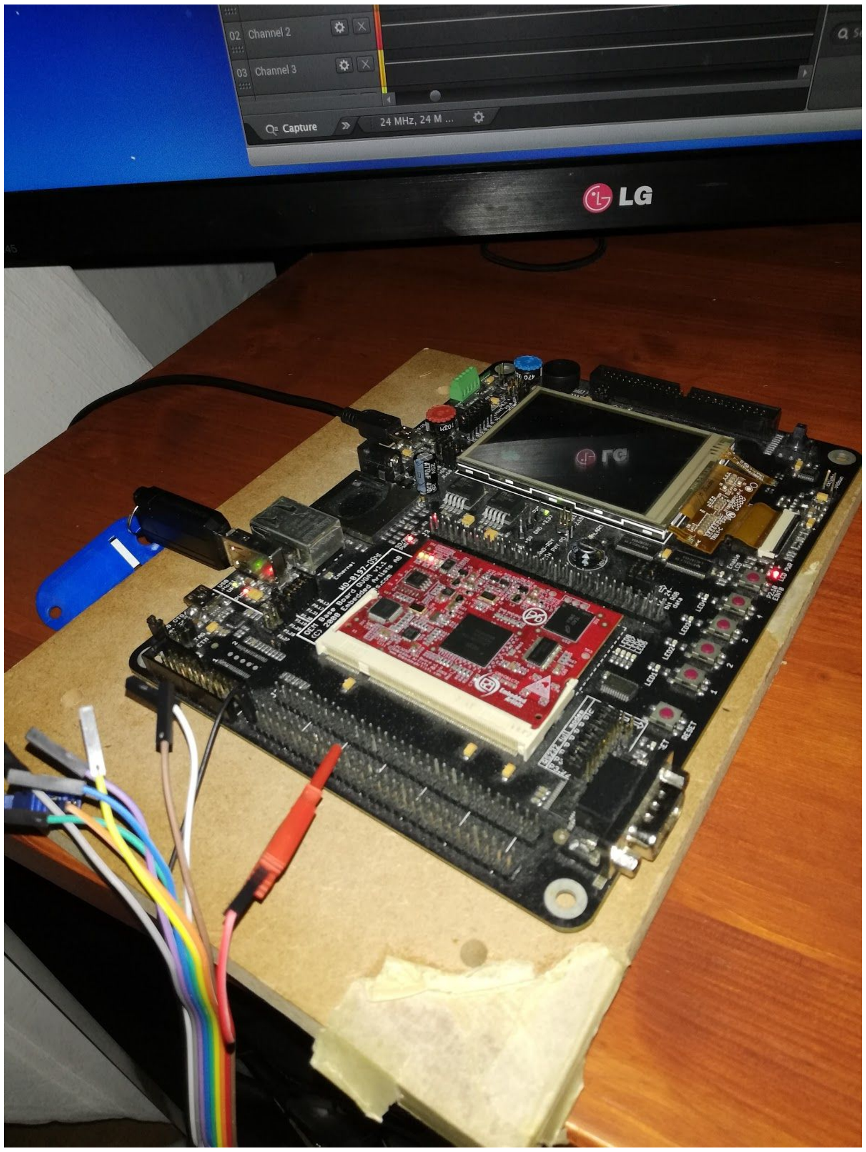

Start
0 s : 0 ms
+0,5 ms +0,6 ms +0,7 ms +0,8 ms +0,9 ms
+0,1 ms +0,2 ms +0,3 ms +0,4 ms +0,5
00 Channel 0

Clip of turning the pin on: https://youtu.be/6UPfJGV0qT4

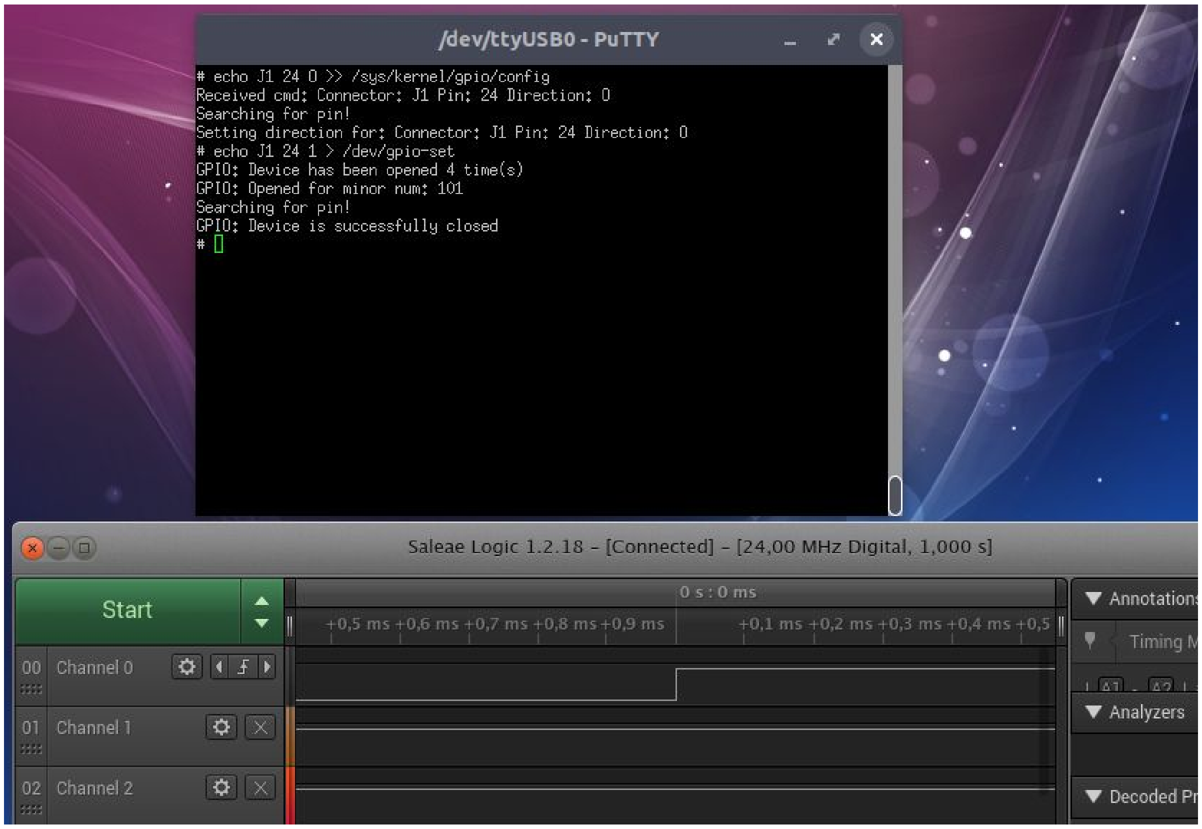
Port 3

Clip of setting pin J1 24 high: https://youtu.be/JOyEXnHPLVE

Picture showing how we are connected to the board.

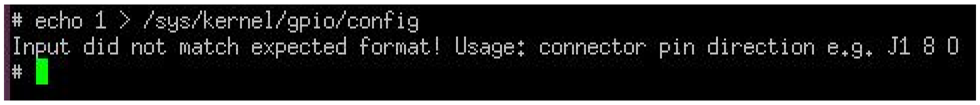


And ofcourse the output in terminal.

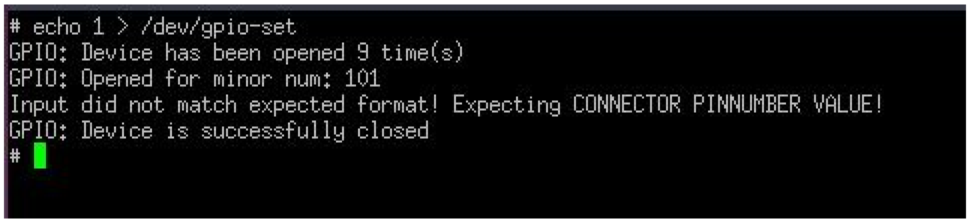

/dev/ttyUSB0 - PuTTY
Saleae Logic 1.2.18 - [Connected] - [24,00 MHz Digital, 1,000 s]
Start
0 s : 0 ms
+0,5 ms +0,6 ms +0,7 ms +0,8 ms +0,9 ms
+0,1 ms +0,2 ms +0,3 ms +0,4 ms +0,5
00 Channel 0
01 Channel 1
02 Channel 2
Annotations
Timing M
Analyzers
Decoded Pr

```
# echo J1 24 0 >> /sys/kernel/gpio/config
Received cmd: Connector: J1 Pin: 24 Direction: 0
Searching for pin!
Setting direction for: Connector: J1 Pin: 24 Direction: 0
# echo J1 24 1 > /dev/gpio-set
GPIO: Device has been opened 4 time(s)
GPIO: Opened for minor num: 101
Searching for pin!
GPIO: Device is successfully closed
#
```

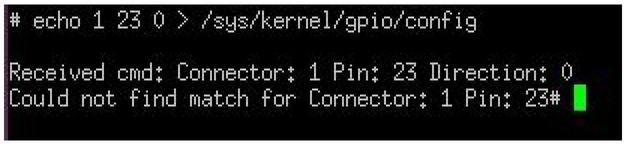
Invalid input tests



```
# echo 1 > /sys/kernel/gpio/config
Input did not match expected format! Usage: connector pin direction e.g. J1 8 0
#
```



```
# echo 1 > /dev/gpio-set
GPIO: Device has been opened 9 time(s)
GPIO: Opened for minor num: 101
Input did not match expected format! Expecting CONNECTOR PINNUMBER VALUE!
GPIO: Device is successfully closed
#
```



```
# echo 1 23 0 > /sys/kernel/gpio/config

Received cmd: Connector: 1 Pin: 23 Direction: 0
Could not find match for Connector: 1 Pin: 23#
```