

Hands-on Linux Academy

Hands-on Instructions

Welcome to *Hands-on Linux Academy*! This guide will show you how to access common peripherals and interfaces of ARM-based systems-on-chip (SoC) running Embedded Linux.

The hands-on consists of four parts:

- **Exercise 1.** The first step will be to prepare an SD card image the board can boot from. Flashing a raw system image will be demonstrated for both Linux and Windows-based host PCs. While waiting for the cards to flash, basics of embedded Linux systems design will be discussed. Once the SD cards are ready, we shall boot the VisionSOM6-ULL, log in through the serial console, and establish an Ethernet connection with the PC.
- **Exercise 2.** shows how various methods of reading and setting GPIOs, accessing I²C devices and transferring data through the SPI bus in the userspace.
- **Exercise 3.** After a short introduction to NFC technology and NXP product offering, we will run sample applications to read common NFC tags, connect to smartphones, and exchange data with connected tags.
- **Exercise 4.** Linux-based embedded systems allow for a high-level approach to solving common software problems. We will use NodeJS, a free, open-source JavaScript runtime, and three.js, a 3D graphics framework, to visualize live-streamed sensor values in a browser window.

You will need about 4.5 hours to complete this training. Many important steps in designing a Linux system are not covered, such as porting u-boot bootloader and the Linux kernel to a new board. A ready to use image has been prepared for this training, but it is not intended to be used directly in production.

SoMLabs provides both documentation and software enablement for the VisionSOM modules on their product wiki: <http://wiki.somlabs.com/index.php?title=VisionSOM-6ULL>

Additional software and documentation for the i.MX6ULL SoC is available at <http://nxp.com/imx6ull>

Let's get started!

EXERCISE 1

Preparing an SD card to boot VisionSOM-6ULL

In order to suit a wide range of applications, **SoMLabs** released three versions of the **VisionSOM-6ULL** system-on-module (SoM), each with a different type of boot memory installed:

- *on-board eMMC Flash*
- *on-board NAND Flash,*
- *micro-SD card slot.*

Together with the VisionCB-STD base-board, VisionSOM-6ULL can be used as a stand-alone embedded computing platform.

For this Hands-on Linux Academy, the VisionCB-STD base-boards have been fitted with the micro-SD version of the SoM. This allows us to use a common SD card reader to prepare the boot memory.

Download the SD card image from the link below:

<http://co.rru.pt/somlabs/somlabs-sdcard-2gb-r1.zip>

After you have extracted the image, connect the USB card reader to your PC or insert the card to the built-in card reader in your laptop.

Follow instructions in section 1.1 or 1.2, depending on the operating system you are using.

1.1. Preparing the SD card under Linux

In Linux, many devices, including storage media, are represented by files in the `/dev` directory. Open a console and use the command below to identify which block device in the system corresponds to the SD card:

```
dmesg -w
```

This will show the kernel message buffer in the console.

If using the SD card reader, you should see output similar to:

```
[21870.506727] sdb: sdb1 sdb2
[21870.509486] sd 1:0:0:0: [sdb] Attached SCSI removable disk
```

If using a built-in reader, expect the following log messages:

```
[ 52.475132] mmc0: new high speed SDHC card at address 0007
[ 52.475411] mmcblk0: mmc0:0007 SD8GB 7.42 GiB
[ 52.480792] mmcblk0: p1 p2
```

`/dev/sdb` (or `/dev/mmcblk0`), represents the entire raw SD card.

`/dev/sdb1` (or `/dev/mmcblk0p1`) corresponds to the first primary partition, `/dev/sdb2` (or `/dev/mmcblk0p2`) the second one, and so on.



To terminate a task running in a Linux console, strike `Ctrl+C`.

Some Linux systems automatically mount (map the filesystem on the block device to a directory) the SD card upon insertion. This might interfere with raw device access in the next step. In order to unmount the filesystem, first list the mount points:

```
mount
...
/dev/sdcl on /media/user/Kingston type vfat (...)
```

If you notice that any of the partitions on the SD card is mounted, unmount it:

```
umount /dev/mmcblk...
```

You can also use the File Manager window to unmount the device.

Once none of the partitions are mounted, write the image to the card:

```
dd if=/path/to/somlabs-sdcard-2gb-r1.img of=/dev/sdX bs=4M
oflag=dsync
```

The card image (`if` - **I**nput **F**ile) will be written to the card (`of` - **O**utput **F**ile) in 4M blocks (`bs` - **B**lock **S**ize) synchronously (without buffering).



Please make sure you provide the correct arguments to `dd`. An error may lead to data loss, even making it impossible to boot your PC again!

Use caution when issuing `dd`, and double check that you specified the correct target block device.



`dd`, by default, does not show progress. To see how much data has been transferred so far, you can pipe the input through the command `pv`:

```
pv file.bin | dd of=/dev/sd... bs=4M oflag=dsync
25.5MiB 0:00:02 [5.03MiB/s] [=====> ] 21% ETA 0:00:27
```

1.2. Preparing an SD card on a Windows PC

The by far easiest way to flash the SD card on Windows is to use the free Win32DiskImager software, available for download at:

<https://sourceforge.net/projects/win32diskimager/>

After inserting the card in the reader, enter the following information into Win32DiskImager's main window:

- (1) path to the *.img file with the system image,
- (2) target device drive letter assigned by the system
- (3) press *Write*.

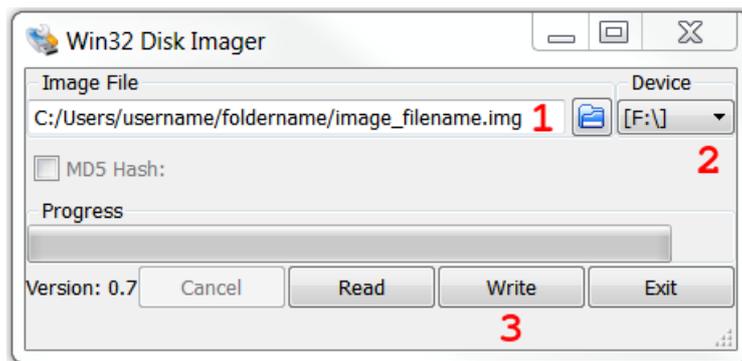


Figure 1. Win32DiskImager main window

1.3. First boot

Insert the microSD card into the SD slot on the module. Then, connect the microUSB cable to the connector marked in Figure 2.

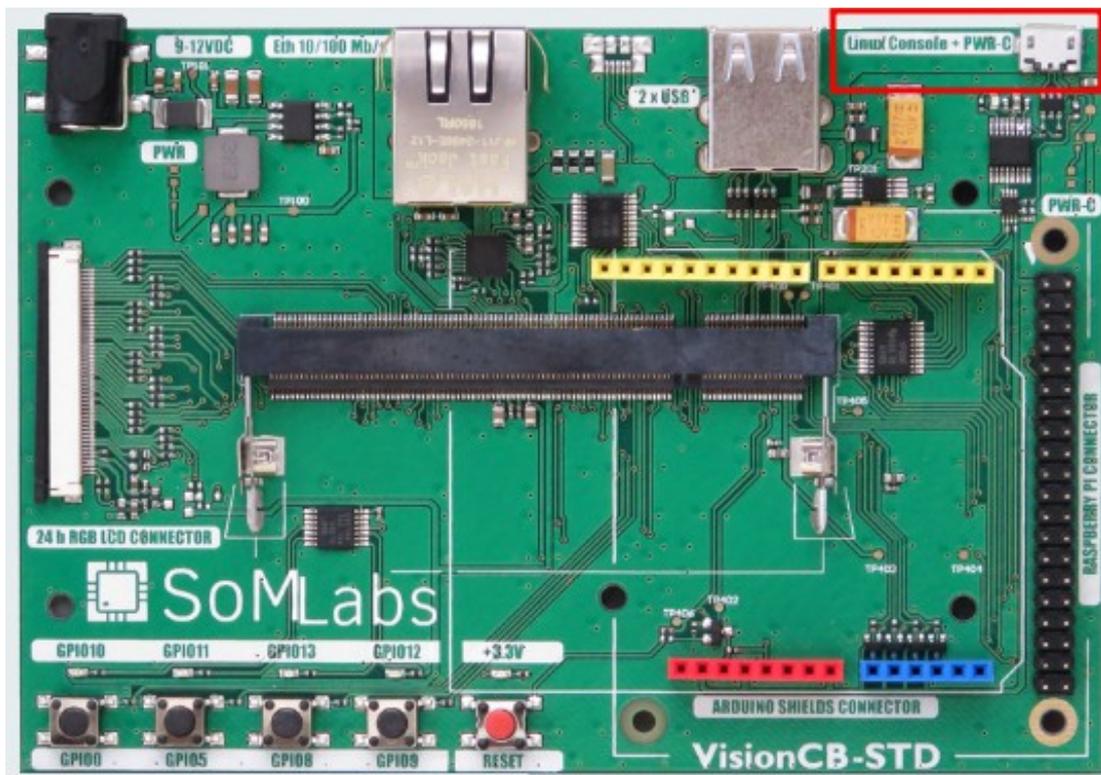


Figure 2. Serial console and power connector location on the VisionCB-STD base board

The micro-USB connector marked in Figure 2 both supplies power to the board, and connects to an FTDI-based USB↔serial converter. To access the console, open the serial port using a terminal emulator program of your choice (e.g. *minicom*, *picocom*, *screen* in Linux, or *Putty*, *HyperTerminal* in Windows). Configure the serial line for 115200 baud, 8N1. In Linux, we recommend using *picocom*:

```
picocom -b 115200 /dev/ttyUSB0
```

After opening the serial port, log in as **root**. You will not be asked to enter a password.

```
Debian GNU/Linux 9 localhost.localdomain ttymxc0
```

```
localhost login: root
root@localhost:~#
```



If you do not have sufficient permissions to run `picocom`, try adding `'sudo'`:

```
sudo picocom -b 115200 /dev/ttyUSB0
```

This will run `picocom` as the root user, who has permissions to access all files on the system.

1.4. Establishing an Ethernet connection (with DHCP)

Use the included Ethernet cable to connect the board to your PC. Make sure your IPv4 settings are set to *Automatic (DHCP)*. Ethernet connectivity is needed to access the webserver in Exercise 3.



Do not type in the settings below. DHCP has already been set up. This is just for your reference.

A DHCP server (dnsmasq) has been set up in the system to allow for seamless connectivity to a PC. dnsmasq can be installed just like any other Debian package using the apt package management system:

```
sudo apt-get install dnsmasq
```

`/etc/network/interfaces` stores connection settings for each interface. By default, Debian expects there to be a DHCP server already in the network:

```
iface eth0 inet dhcp
```

The above line has been deleted and replaced with a static IPv4 configuration:

```
auto eth0
iface eth0 inet static
address 192.168.0.1
netmask 255.255.255.0
```

dnsmasq settings are stored in `/etc/dnsmasq.conf`:

```
interface=eth0
dhcp-range=192.168.0.2,192.168.0.254,255.255.255.0,12h
dhcp-option=3
dhcp-option=6
```

The DHCP server binds to interface eth0, and allocates addresses from 192.168.0.2 to 192.168.0.254. The netmask has 24 bits (255.255.255.0), and addresses are leased for 12 hours. Options 3 and 6 specify the DNS server and gateway addresses, they have been set to empty values.

EXERCISE 2

Introduction to Embedded Linux - accessing GPIOs, I²C and SPI busses

This exercise shows you a few methods to configure, sample and set the most simple of peripheral devices - GPIO (**G**eneral **P**urpose **I**nput/**O**utput) ports. GPIOs can be accessed through the /sys virtual filesystem,

Next, using SPI and I²C busses will be demonstrated.

The SPI example shifts bytes through a loopback connection (MOSI → MISO), and prints them in the console.

The I²C example uses the kernel's input APIs to read data from a gyroscope sensor.

Based on the code examples in this exercise, you should be able to create basic userspace device drivers and use input devices, such as MEMS sensors.



On microcontrollers, the bare-metal firmware or RTOS task usually has access to the entire memory map and has to directly read and write registers to control I/O peripherals. The programmer needs to know the hardware architecture of the peripheral controllers, or at least the (vendor-specific) driver APIs.

In Linux, there are common driver models for most types of peripheral devices and common APIs to access them from userspace. All sample applications in this chapter are platform-agnostic. They can run on other ARMv7-based boards and processors, or can even be compiled for other architectures, as long as these new targets have similar external connections to busses and GPIOs.

2.1. Accessing GPIOs via /sys/class/gpio



The kernel and devicetree have already been configured to support the GPIOs.

The easiest way to access GPIOs is to use the sysfs interface of the kernel GPIO driver. The driver has been enabled in the config before building the kernel:

```
Device Drivers -->
  *- GPIO Support -->
    [*]/sys/class/gpio/...(sysfs interface)
```

During boot, the kernel asks the driver to probe (take care of) the GPIO ports listed in the devicetree:

```
gpio1: gpio@0209c000 {
  compatible = "fsl,imx6ul-gpio", "fsl,imx35-gpio";
  reg = <0x0209c000 0x4000>;
  interrupts = <GIC_SPI 66 IRQ_TYPE_LEVEL_HIGH>,
```

```

        <GIC_SPI 67 IRQ_TYPE_LEVEL_HIGH>;
    gpio-controller;
    #gpio-cells = <2>;
    interrupt-controller;
    #interrupt-cells = <2>;
};

```

Every GPIO chip declared in the devicetree is represented by a file in sysfs:

```

root@localhost:~# cd /sys/class/gpio/
root@localhost:/sys/class/gpio# ls -l
total 0
--w----- 1 root root 4096 Oct  1 19:40 export
lrwxrwxrwx 1 root root    0 Oct  1 19:40 gpiochip0
lrwxrwxrwx 1 root root    0 Oct  1 19:40 gpiochip128
lrwxrwxrwx 1 root root    0 Oct  1 19:40 gpiochip32
lrwxrwxrwx 1 root root    0 Oct  1 19:40 gpiochip64
lrwxrwxrwx 1 root root    0 Oct  1 19:40 gpiochip96
--w----- 1 root root 4096 Oct  1 19:40 unexport

```

Before a GPIO pin can be used, it has to be exported to userspace. Export pin 10:

```

root@localhost:~# echo 10 > /sys/class/gpio/export

```

A GPIO pin already in use by another driver in the kernel cannot be exported. When the GPIO is no longer needed by the userspace, it can be unexported.

Once the GPIO pin is exported, a file-based interface is exposed in `/sys/class/gpio/gpioX`, where `X` is the pin number. Use `ls` to display the contents of the directory:

```

root@localhost:~# cd /sys/class/gpio/gpio10
root@localhost:/sys/class/gpio/gpio10# ls -l
total 0
-rw-r--r-- 1 root root 4096 Oct  1 23:04 active_low
lrwxrwxrwx 1 root root    0 Oct  1 23:04 device
-rw-r--r-- 1 root root 4096 Oct  1 23:04 direction
-rw-r--r-- 1 root root 4096 Oct  1 23:04 edge
drwxr-xr-x 2 root root    0 Oct  1 23:04 power
lrwxrwxrwx 1 root root    0 Oct  1 23:04 subsystem
-rw-r--r-- 1 root root 4096 Oct  1 23:04 uevent
-rw-r--r-- 1 root root 4096 Oct  1 23:04 value

```

In Linux, (almost) everything is a file. GPIOs can be controlled by reading and writing the virtual files in `/sys/class/gpio/gpioX`:

- *direction* - controls the direction of the GPIO:
 - set up gpioX as an output:


```
echo out > /sys/class/gpio/gpioX/direction
```
 - set up gpioX as an input:


```
echo in > /sys/class/gpio/gpioX/direction
```

- *value* - if the GPIO is an output, write 0 or 1 to the file. If the GPIO is an input, read the file to sample its state:
 - `gpioX` configured as output - set to low:


```
echo 0 > /sys/class/gpio/gpioX/value
```
 - show the state of `gpioX`:


```
cat /sys/class/gpio/gpioX/value
```
- *edge* - sets the interrupt trigger for inputs. Possible values: `none`, `rising`, `falling` or `both`:
 - `gpioX` configured as input and triggered by a falling edge:


```
echo falling > /sys/class/gpio/gpioX/edge
```

2.2. "Hello World" of an embedded system - blinking an LED [from a shell script]

The commands from the previous section can be automated by a shell script. *Figure 2.2.1.* shows where the LED connected to GPIO10 is located.

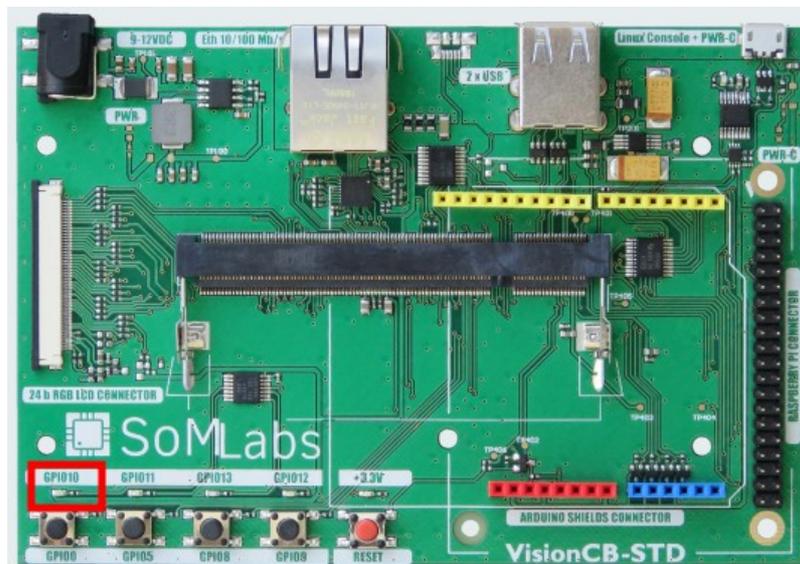


Figure 2.2.1. Location of the LED connected to `GPIO1_10`



Source code for all examples is located inside `root`'s home directory:
`/root/linux-academy/<section number>/`

Listing 2.2.1 shows the contents of the `blink.sh` shell script:

```
#!/bin/sh
```

```

LED=10
LEDDIR=/sys/class/gpio/gpio$LED

if [ ! -d "$LEDDIR" ]; then
    echo "Exporting GPIO$LED"
    echo $LED > /sys/class/gpio/export
else
    echo "GPIO$LED already exported"
fi

echo out > $LEDDIR/direction

while true ; do

    echo 1 > $LEDDIR/value
    sleep 1

    echo 0 > $LEDDIR/value
    sleep 1

done

```

Listing 2.2.1. Basic shell script to blink an *LED*

To run *blink.sh*, first set its executable flag:

```

root@localhost:~# chmod +x /root/linux-acadaemy/2-2/blink.sh
root@localhost:~# /root/linux-acadaemy/2-2/blink.sh

```

2.3. Blinking an LED from a C application

Shell scripts are a convenient tool for fast prototyping, yet due to low execution speed and no compile-time error detection, it is usually preferred to control GPIOs from binary applications, most often developed in C/C++. The below example shows how an application similar to the one described in section 2.2 can be implemented in C. The same */sys/class/gpio* interface is used.

Three helper functions are used to set up and control the GPIO:

- Export the GPIO to userspace:

```

static int
gpio_export (unsigned int gpio)
{
    int fd, len;
    char buf[BUF_SIZE];

    fd = open (GPIO_DIR "/export", O_WRONLY);
    if (fd < 0)
    {
        perror ("gpio/export");
        return fd;
    }
}

```

```

len = snprintf (buf, sizeof(buf), "%d", gpio);
write (fd, buf, len);
close (fd);

return 0;
}

```

- **Set the GPIO direction:**

```

static int
gpio_set_direction (unsigned int gpio,
                   unsigned int direction)
{
    int fd;
    char buf[BUF_SIZE];

    snprintf (buf, sizeof(buf), GPIO_DIR "/gpio%d/direction", gpio);
    fd = open (buf, O_WRONLY);
    if (fd < 0)
    {
        perror ("gpio/direction");
        return fd;
    }

    if (direction)
        write (fd, "out", sizeof("out"));
    else
        write (fd, "in", sizeof("in"));

    close (fd);
    return 0;
}

```

- **Output a high or low level on the GPIO:**

```

static int
gpio_set_value (unsigned int gpio,
               unsigned int value)
{
    int fd;
    char buf[BUF_SIZE];

    snprintf (buf, sizeof(buf), GPIO_DIR "/gpio%d/value", gpio);
    fd = open (buf, O_WRONLY);
    if (fd < 0)
    {
        perror ("gpio/set-value");
        return fd;
    }

    if (value)
        write (fd, "1", 2);
    else

```

```

        write (fd, "0", 2);

    close (fd);
    return 0;
}

```

With the functions above, interfacing GPIOs becomes very simple. `Main()` is just a few lines of code:

```

#define GPIO_PIN        10
#define GPIO_DIR        "/sys/class/gpio"
#define GPIO_IN         0
#define GPIO_OUT        1

int
main (void)
{
    if (gpio_export (GPIO_PIN) < 0)
        exit (EXIT_FAILURE);

    if (gpio_set_direction (GPIO_PIN, GPIO_OUT) < 0)
        exit (EXIT_FAILURE);

    /* infinite loop */
    while (1)
    {
        gpio_set_value (GPIO_PIN, 1);
        sleep (1);

        gpio_set_value (GPIO_PIN, 0);
        sleep (1);
    }

    return EXIT_SUCCESS;
}

```

Build the program `gcc` and run it:

```

root@localhost:~# gcc blink.c -o blink
root@localhost:~# ./blink

```

2.4. Button input

This example will show you how to use the edge trigger functionality, by using it to detect when a button is pressed. We will use `GPIO1_3`, which is connected to a button - see [Figure 2.4.1 below](#):

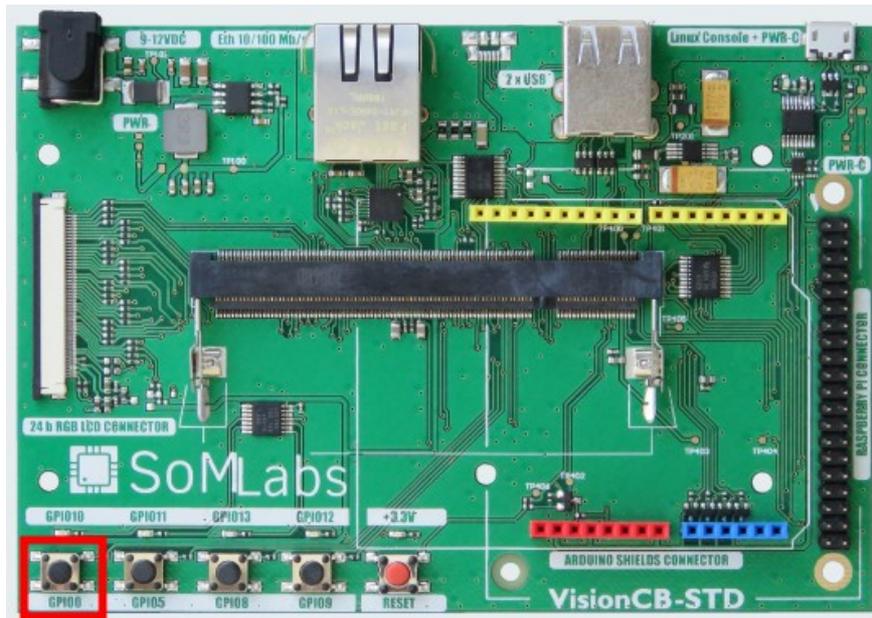


Figure 2.4.1. Location of the button connected to `GPIO1_3`

Simply checking the button state (reading `/sys/class/gpio/gpioX/value`) in a loop would take nearly 100% of the CPU time, making the system less responsive. The core would never be able to enter low-power mode, so power consumption would increase. Adding a delay would solve these problems, but then the time taken to react to the button press would vary.

A `poll()` or `select()` function (system call) can be used to wait for an event on one or more file descriptors. If a trigger event is chosen in `/sys/class/gpio/gpioX/edge`, the GPIO driver will wait for an interrupt and post an event to the file descriptor after the interrupt handler is called.

The code in this example (2-4) is based on the previous one (2-3). A function has been added to enable edge trigger by writing `/sys/class/gpioX/edge`:

```
static int
gpio_set_edge (unsigned int  gpio,
               char          *edge)
{
    int fd;
    char buf[BUF_SIZE];

    snprintf (buf, sizeof(buf), GPIO_DIR "/gpio%d/edge", gpio);

    fd = open (buf, O_WRONLY);
    if (fd < 0)
    {
        perror ("gpio/edge");
        return fd;
    }
    write (fd, edge, strlen(edge) + 1);
    close (fd);
}
```

```
    return 0;
}
```

`poll()` expects an array of descriptors, which will be 'monitored', and will block until there is an event on at least one of the descriptors, or until a timeout. The code below opens the `value` file using the `open` function, to get a file descriptor.

```
static int
gpio_fd_open (unsigned int gpio)
{
    int fd;
    char buf[BUF_SIZE];

    snprintf (buf, sizeof(buf), GPIO_DIR "/gpio%d/value", gpio);

    fd = open (buf, O_RDONLY | O_NONBLOCK );
    if (fd < 0)
        perror ("gpio/fd_open");

    return fd;
}
```

`main()` calls the helper functions and uses `poll()` to wait for the interrupt:

```
int
main (void)
{
    struct pollfd fdset[1];
    int nfds = 1, fd, ret;

    if (gpio_export (GPIO_PIN) < 0)
        exit (EXIT_FAILURE);

    if (gpio_set_direction (GPIO_PIN, GPIO_IN) < 0)
        exit (EXIT_FAILURE);

    if (gpio_set_edge (GPIO_PIN, "rising") < 0)
        exit (EXIT_FAILURE);

    fd = gpio_fd_open (GPIO_PIN);
    if (fd < 0)
        exit (EXIT_FAILURE);

    lseek (fd, 0, SEEK_SET);
    read (fd, &buf, BUF_SIZE);

    while (1)
    {
        memset (fdset, 0, sizeof(fdset));
        fdset[0].fd = fd;
        fdset[0].events = POLLPRI;
        ret = poll (fdset, nfds, -1);
        if (ret < 0) {
```

```

        printf ("poll(): failed!\n");
        goto exit;
    }

    if (fdset[0].revents & POLLPRI) {
        printf ("poll(): GPIO_%d interrupt occurred\n", GPIO_PIN);
        lseek (fdset[0].fd, 0, SEEK_SET);
        read (fdset[0].fd, &buf, BUF_SIZE);
    }
    fflush(stdout);
}

exit:
    close (fd);
    return EXIT_FAILURE;
}

```

Build and run *button.c*. Press the button to unlock the call to `poll()`:

```

root@localhost:~# gcc button.c -o button
root@localhost:~# ./button
poll(): GPIO_3 interrupt occurred
poll(): GPIO_3 interrupt occurred

```

2.5. Detecting button presses using the *Linux input system*

While it is possible to monitor buttons with the standard GPIO interface, it is more appropriate to treat them as an input device, just like a keyboard or mouse in a PC. Events on human interface devices (and some sensors) are reported through the *Linux input system*.



The kernel and devicetree have already been configured to support the on-board GPIOs.

GPIO Buttons support needs to be enabled in the kernel:

```
Device Drivers --->
  Input device support --->
    [*] Keyboards --->
      <*> GPIO Buttons
```

Key presses are reported to the userspace via the *Event interface*, part of the *Linux Input System*. The appropriate driver has to be enabled as well:

```
Device Drivers --->
  Input device support --->
    <*>Event interface
```

Every button needs to have an entry to map it to a key code:

```
gpio-keys {
    compatible = "gpio-keys";
    pinctrl-0 = <&pinctrl_gpio_keys>;
    pinctrl-names = "default";

    btn3 {
        label = "btn3";
        gpios = <&gpio1 8 GPIO_ACTIVE_HIGH>;
        linux,code = <103>; /* <KEY_UP> */
    };

    btn4 {
        label = "btn4";
        gpios = <&gpio1 9 GPIO_ACTIVE_HIGH>;
        linux,code = <108>; /* <KEY_DOWN> */
    };
};
```

Two buttons are connected to GPIO1_8 and GPIO1_9, and assigned keycodes 103 (*KEY_UP*) and 108 (*KEY_DOWN*), respectively. Their placement on the VisionCB base-board is shown in Figure 2.5.1.

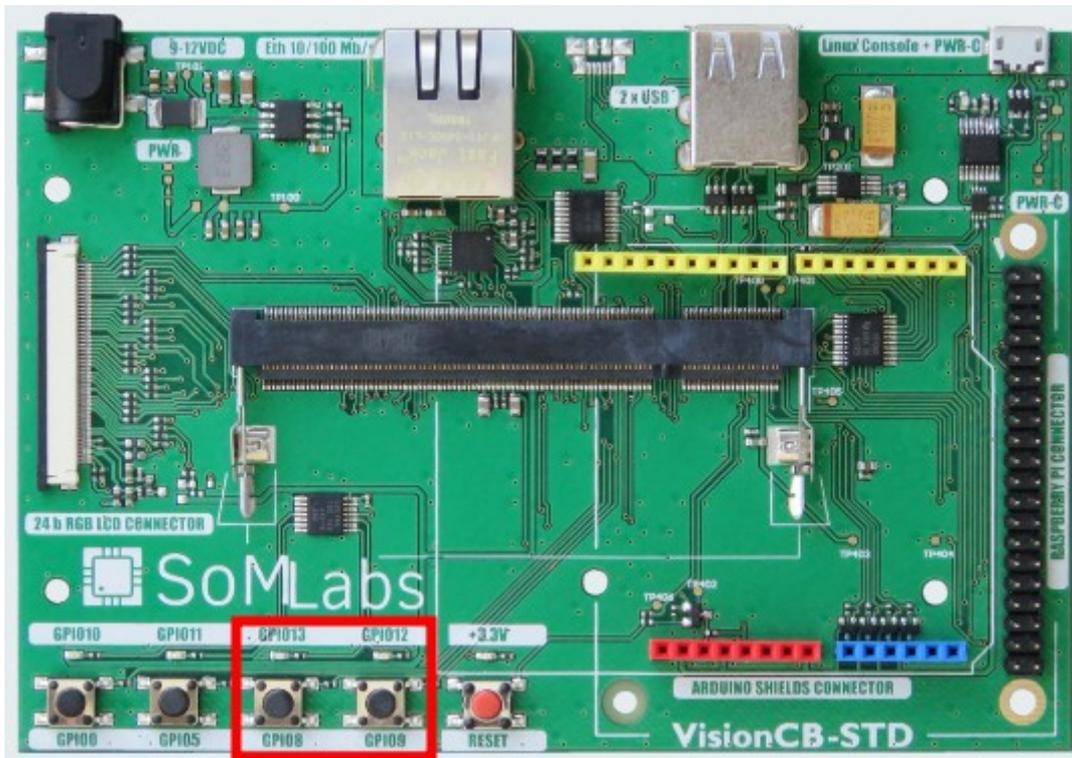


Figure 2.5.1. Location of buttons connected to GPIOs 1_8 and 1_9.

The kernel exposes `/dev/input/event1` for each input device. Let's see if we can read those event files just like we did previously with GPIOs:

```
root@localhost:~# cat /dev/input/event1
T
♦♦♦♦T
♦♦T
♦T
♦♦T
```

The *Event interface* uses a binary format, so printing the data with `cat` results in garbage on the terminal. Events are reported by `input_event` structures:

```
struct input_event {
    struct timeval time;
    unsigned short type;
    unsigned short code;
    unsigned int value;
};
```

hexdump can be used to make the raw data more readable:

```
root@localhost:~# hexdump /dev/input/event1
00000000 44d5 59d1 da16 0000 0001 006c 0000 0000
00000100 44d5 59d1 da16 0000 0000 0000 0000 0000
00000200 44d5 59d1 d5f7 0002 0001 006c 0001 0000
00000300 44d5 59d1 d5f7 0002 0000 0000 0000 0000
```

Notice that 0x6c equals 108 decimal, which is the *KEY_DOWN* keycode.

/root/linux-academy/2-5/gpio-keys.c (Listing 2.5.1 below) shows how to read and parse events received from the input system:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <linux/input.h>

int
main (void)
{
    struct input_event ev;
    int size = sizeof(ev), fd;

    fd = open ("/dev/input/event1", O_RDONLY);
    if (fd < 0)
    {
        printf ("Open /dev/input/event1 failed!\n");
        return EXIT_FAILURE;
    }

    while (1)
    {
        if (read(fd, &ev, size) < size)
        {
            printf ("Reading from /dev/input/event1 failed!\n");
            goto exit;
        }

        if (ev.type == EV_KEY)
        {
            if (ev.code == KEY_DOWN)
                ev.value ? printf("KEY_DOWN:release\n") :
printf("KEY_DOWN:press\n");
            else if (ev.code == KEY_UP)
                ev.value ? printf("KEY_UP:release\n") :
printf("KEY_UP:press\n");
            else
                puts ("WTF?!");
        } /* ev_key */
    } /* while */
}
```

```

exit:
    close (fd);
    return EXIT_FAILURE;
}

```

Listing 2.5.1. */root/linux-academy/2-5/gpio-keys.c*

To build and run the *gpio-keys* example application, issue the following commands:

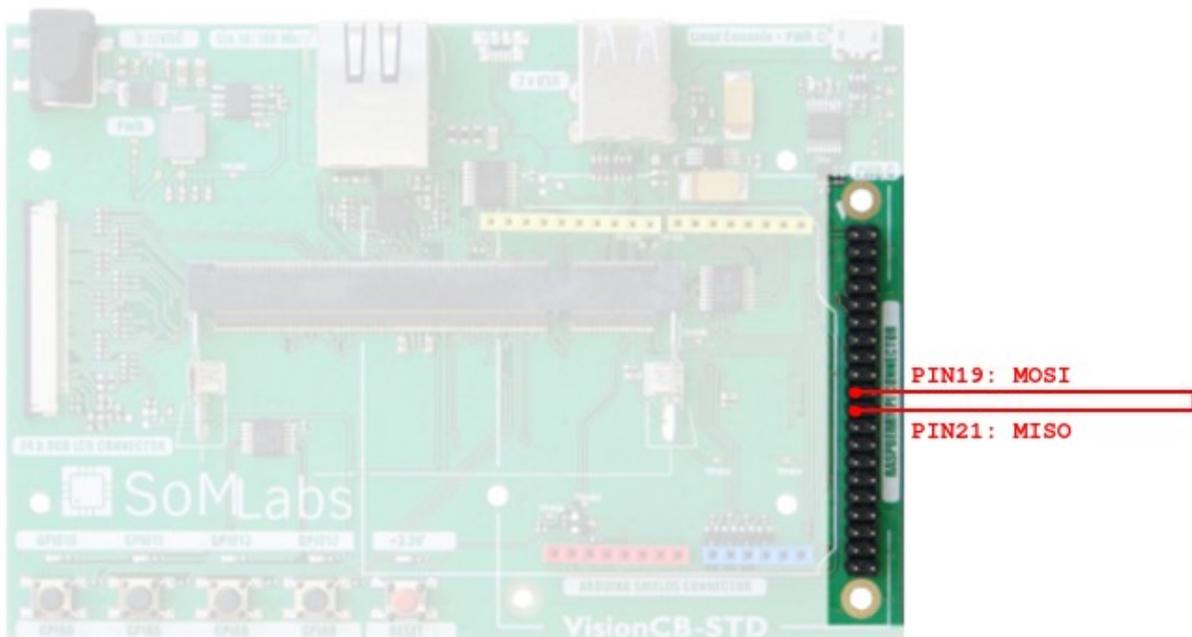
```

root@localhost:~# cd /root/linux-academy/2-5
root@localhost:~/linux-academy/2-5# gcc gpio-keys.c -o gpio-keys
root@localhost:~/linux-academy/2-5# ./gpio-keys
KEY_UP: press
KEY_UP: release
KEY_UP: press
KEY_UP: release
KEY_DOWN: press
KEY_DOWN: release
KEY_DOWN: press
KEY_DOWN: release

```

2.7. Testing SPI with a loopback connection

[1] Connect pins 19 and 21 with a jumper wire:



[2] Build and run *loopback-spi* example application:

```

root@localhost:~# cd /root/linux-academy/2-7
root@localhost:~/linux-academy/2-7# gcc loopback-spi.c -o loopback-spi
root@localhost:~/linux-academy/2-7# ./loopback-spi
FF FF FF FF FF FF
40 00 00 00 00 95
FF FF FF FF FF FF

```

```
FF FF FF FF FF FF
FF FF FF FF FF FF
DE AD BE EF BA AD
F0 0D
```

The MISO line has an on-chip pull-up enabled. Without the cable connected properly, you may see only FF values:

```
root@localhost:~/linux-academy/2-7# ./loopback-spi
FF FF FF FF FF FF
FF FF
```

2.X. NXP FXAS2100X gyroscope

FXAS2100x is a triple-axis digital gyroscope with 16-bit resolution and adjustable range from 250°/s to 2000°/s, and up to 800 Hz sample rate. We will use it to show how to communicate with more complex sensors on the I²C bus.



The kernel driver for the gyroscope has already been enabled in the kernel config, and the device has been added to the devicetree.

To enable the sensor's driver in the kernel, the following option has to be selected:

```
Device Drivers --->
  Misc devices --->
    <*> Freescale FXAS2100X gyroscope sensor
```

The gyroscope needs to be declared in the devicetree:

```
&i2c2 {
    status = "okay";
    fxas2100x@20 {
        compatible = "fsl,fxas2100x";
        reg = <0x20>;
    };
};
```



Do not connect the shield to the Arduino-style connector on VisionCB. The connector on VisionCB has a 5V supply and uses 5V logic. The shield accepts only 3.3V.

Make sure the jumpers *J6* and *J7* are both set to positions 1-2:

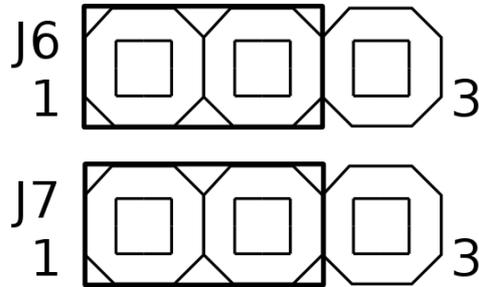


Figure 2.X.1. Gyroscope shield jumper settings

Connect the Arduino-style board with the gyroscope to the Raspberry Pi-compatible connector on VisionCB, as shown in Figure 2.X.1:

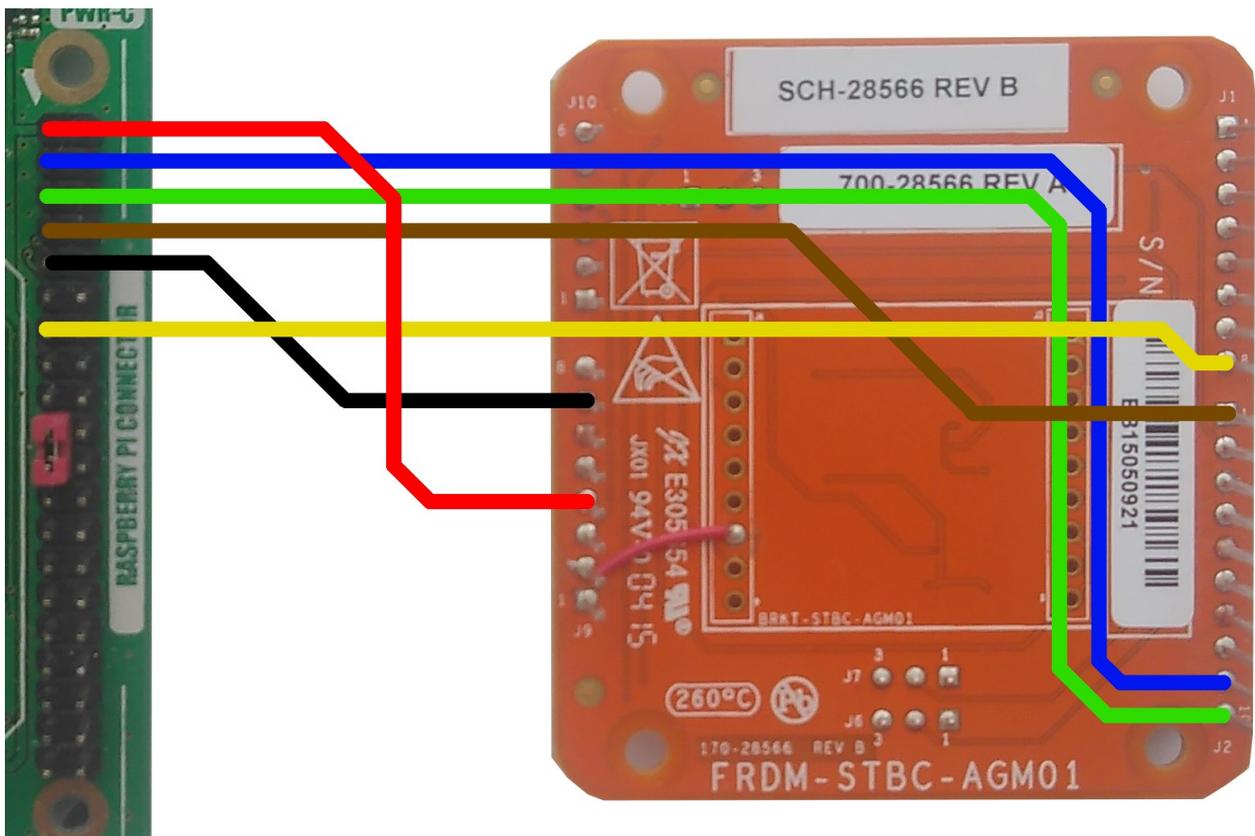


Figure 2.X.2. Gyroscope shield connections

Enable the gyroscope:

```
root@localhost:~# cd fexas2100x
root@localhost:~/fexas2100x# ./fexas2100x_enable.sh
[ 1470.052799] misc FreescaleGyroscope: mma enable setting active
```

Build and run the gyroscope test application:

```
root@localhost:~/fexas2100x# make
make: Warning: File 'Makefile' has modification time 2963329 s in
the future
cc -Wall -O0 -g -funsigned-char -I. -c fexas2100x.c -o fexas2100x.o
cc -g -O0 -Wl,--gc-sections,--relax -L/usr/local/lib
-lnfc_nci_linux -lpthread fexas2100x.o -o gyro-i2c
make: warning: Clock skew detected. Your build may be
incomplete.
root@localhost:~/fexas2100x# ./gyro-i2c
0.0 0.0 0.0
0.1 0.0 -3.0
-0.3 0.3 -6.9
^C
root@localhost:~/fexas2100x#
```

EXERCISE 3

NFC (Near-Field Communications) is a contactless interface enabling power and bi-directional data transfer, based on inductive coupling between two antennae.

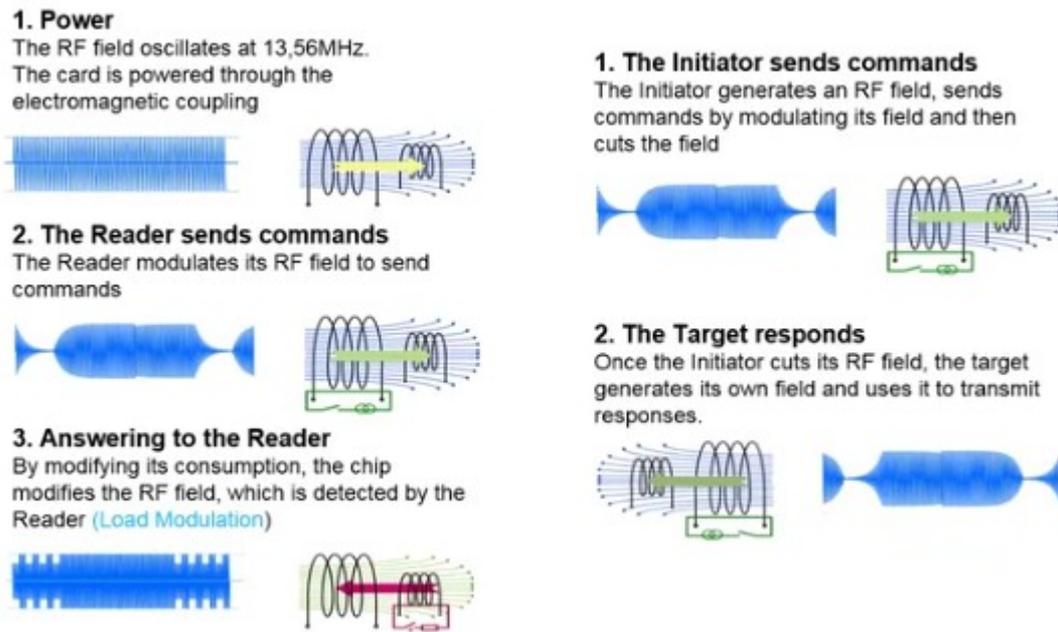


Figure 3.1. Accessing a passive tag (left) and P2P mode

Two devices are always involved in NFC communication: an initiator and a target. The initiator must always be active, i.e. able to power the antenna. The target can be either active or passive. A passive target is powered by energy harvested by the antenna, and uses the antenna to modulate the existing field. Virtually all tags are passive targets.

NDEF (NFC Data Exchange Format) is used to carry messages via the NFC interface. It is supported by most consumer devices, including NFC-enabled Android smartphones. An NDEF message consists of one or more records. Standard record types are defined for many common payloads:

- Text
- Phone number
- E-mail address
- Geographical location
- WiFi network credentials
- MIME types
- URIs

Application-specific record types can be defined by specifying new URI schemes or MIME types.

Three example applications implementing different NFC use cases have been created for this training:

- reading tags,
- exchanging NDEF messages with a smartphone (P2P mode)
- interacting with connected tags (NTAG I²C Plus)

3.1. Hardware set-up

Stack the PN7120 shield over the gyroscope shield.

3.2. Reading tags

This application supports NDEF-formatted MIFARE Ultralite tags. It reads text records and prints them to the console.

To build and run the example:

```
root@localhost:~# cd ~/libnfc-nci-demos/read_ntag
root@localhost:~/libnfc-nci-demos/read_ntag# make
root@localhost:~/libnfc-nci-demos/read_ntag# ./read_ntag
```

Jeżeli do anteny na PCB zostanie zbliżony tag zawierający rekord tekstowy, jego treść zostanie wyświetlona w konsoli.

3.3. Exchanging NDEF messages with a smartphone (P2P mode)

P2P mode allows two active devices to exchange NDEF messages.

Run the following commands in the Linux console:

```
root@localhost:~# cd ~/libnfc-nci-demos/ndef_p2p
root@localhost:~/libnfc-nci-demos/card_emu# make
root@localhost:~/libnfc-nci-demos/card_emu# ./ndef_p2p "Hello!"
```

Enable NFC in your smartphone: *Settings* → *More* → *NFC*

Tap your smartphone to the on-board NFC antenna. A dialog should pop up with the message supplied in the command-line argument.

3.4. Connected tag (NTAG I²C Plus)

NTAG I²C Plus is an NFC tag with an I²C port. It is possible to access the internal memory both from the I²C bus and NFC interface. The tag can harvest energy to power external components, and it has a field detect output. A 64-byte page of SRAM can be used for fast, two-way communication between the MCU and an external NFC device, such as a smartphone.

Run the following commands in the console:

```
root@localhost:~# cd ~/libndc-nci-demos/ntag_i2c
root@localhost:~/libndc-nci-demos/ntag_i2c# make
```

```
root@localhost:~/libndc-nci-demos/ntag_i2c# ./ntag_i2c
```

Hold the NTAG I²C Plus demoboard above the on-board NFC antenna. The RGB LED will start blinking. While holding the demoboard in place, press one or more buttons with colored caps. The buttons pressed will be listed in the console.



If you hold the middle button while bringing the tag to the reader, the demoboard will enter programming mode and re-write the tag's EEPROM with default content.

EXERCISE 4

4.1. Node.js - Embedded Linux and Javascript?

What is *Node.js*?

Node.js is a multi-platform JavaScript runtime, based on Google's V8 engine - the same one used in the Chrome browser. Instead of implementing the Document Object Model, Node provides APIs for common server-side tasks, such as opening files, accessing databases, establishing TCP/IP connections, or implementing various network services. Due to its flexible architecture, it can also be used in embedded systems to interact with device drivers.

Node.js is used by many large web companies, such as *Netflix*, *PayPal*, *LinkedIn* or *Uber*. Node's online package manager, *npm*, hosts over 470 000 packages of free, reusable code.

In Debian, Node can be installed just like any other package, using the Apt package manager:

```
root@localhost:~# apt-get install nodejs
Selecting previously unselected package libuv1:armhf.e will be used.
(Reading database ... 34198 files and directories currently installed.)
Preparing to unpack .../libuv1_1.9.1-3_armhf.deb ...
Unpacking libuv1:armhf (1.9.1-3) ...
Selecting previously unselected package nodejs.
Preparing to unpack .../nodejs_4.8.2~dfsg-1_armhf.deb ...
Unpacking nodejs (4.8.2~dfsg-1) ...
```



Node.js has already been installed.

To check which version of Node is in the system:

```
root@localhost:~# nodejs -v
v4.8.2
```

4.2. Node.js - A basic web server



Full source code for example 4.2:

- `/root/linux-academy/4-2/main.js`

To implement the HTTP server, we shall use the built-in Node module, `http`:

```
var http = require ('http');
```

The server will run on port 8080:

```
var PORT = 8080;
```

An `http` server object needs to be created. In JavaScript, a function is a first-class citizen. Functions can be passed as arguments to other functions to declare callbacks, and assigned to structure members to form objects. The `http.createServer` constructor takes a handler function as an argument, and returns an `http.Server` object.

```
var server = http.createServer (function handler (request,
response) {
  response.writeHead (200, {'Content-Type': 'text/plain'});
  response.end ('Hello World!');
});
```

Once the server receives a request, the handler function is called with two arguments:

- `request` - contains the requested URL, access method, and headers,
- `response` - an object the handler function can write the response to.

In this case, every request results in a *200 OK* status code, and the server returns a plaintext document with just the *Hello World!* phrase.

Finally, `listen()` is called, and the server starts listening for connections on the port specified in the argument:

```
server.listen (PORT);
```

Start the server using the command below::

```
nodejs main.js
```

You should now be able to reach `http://192.18.0.1:8080` from the web browser on your PC - see *Figure 4.2.1*.

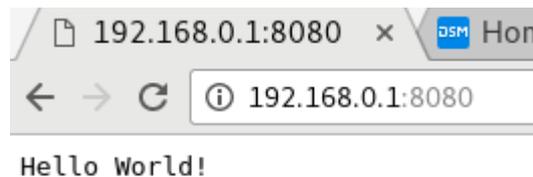


Figure 4.2.1. NodeJS serving static content with the 'http' module

4.3. Node.js – Serving local files



Full source code for example 4.3:

- `/root/linux-academy/4-3/main.js`
- `/root/linux-academy/4-3/index.html`

It is usually a better idea to store the content to be served in a separate file, rather than in the source code of the server itself. *Listing 4.3.1* below shows how to read and serve a file:

```
var http = require ('http');
var fs = require ('fs');

var index = fs.readFileSync (__dirname + '/index.html');

var PORT = 8080;

var server = http.createServer (function handler (request,
response) {
  response.writeHead (200, {'Content-Type': 'text/html'});
  response.end (index);
});

server.listen (PORT);
```

Listing 4.3.1. Serving a local file via HTTP

The built-in module `fs` implements synchronous file operations. Because `index.html` is a hypertext document, not a plaintext file, the `Content-type` response header has been changed to `text/html`.

`index.html` is just a simple web page:

```
<!DOCTYPE html>
<html>

  <head>
  </head>
```

```
<body>
  <h1>Hello World!</h1>
</body>
</html>
```

Run *Node.js*:

```
nodejs main.js
```

Then, refresh the page in the browser.

4.4. Node.js - front-end to back-end communication using socket.io



Full source code for example 4.4:

- `/root/linux-academy/4-4/main.js`
- `/root/linux-academy/4-4/index.html`

Introducing a clear division between front- and back-end is tricky, since we now need to establish real-time communication between the two. HTTP isn't particularly suited to this, since it's a request-response type protocol and relies on the client initiating communication - but what if it is the server that has new data for the web application running in the browser? Trying to periodically refresh the file on a timer will work, but leaves a lot to be desired and isn't the way to go.

To get around this limitation, we'll use a JavaScript library called `socket.io` - it will allow us to link the front-end to the back-end through persistent, bi-directional network sockets, „piggybacked“ on top of HTTP. In short, it simplifies handling the WebSocket protocol, which itself is part of the HTML5 specification. Socket.io is comprised of two parts - the server-side (a module for the *Node.js* platform), and the client-side (code written for web browsers).

Basing on the *main.js* code from the previous example, let's move into discussing a practical implementation.

We begin amending *main.js* by importing the `socket.io` module (details on installing this module are discussed in more detail in the aside below):

```
var io = require ('socket.io').listen(server);
```



socket.io is not part of the *Node.js* core platform and requires separate installation. To do that, you can use the *npm* package manager:

```
npm install socket.io
```

Notably, *socket.io* is distributed along the source code of the examples using it in the default image.

For our next step, we need to create an event handler for the incoming connections. This handler will be executed each time a new client connects to our socket server. Let's also have it log a status update to the screen, informing us of the new connection:

```
io.on ('connection', function (socket) {
  console.log ('We have new connection!');
});
```

The goal of example 4 is to have the server application update the user's web browser with information read from the gyroscope module. The method of linking the *gyro-i2c* application (from example 2) with the web server will be discussed in the next stage of this exercise. For our current needs, we will prepare a simple `send_time()` function, which will send the current time at one-second intervals, to all connected clients:

```
function send_time() {
  io.emit ('time', {message: new Date().toISOString()});
}
setInterval (send_time, 1000);
```

In the body of this function, we are broadcasting a message with the current time to all connected clients. The full source listing of *main.js*, along with clearly delineated departures from the code in example 4.3, is shown in *Listing 4.4*.

```
var http = require ('http');
var fs = require ('fs');

var index = fs.readFileSync (__dirname + '/index.html');

var PORT = 8080;

var server = http.createServer (function handler (request, response) {
  response.writeHead (200, {'Content-Type': 'text/html'});
  response.end (index);
});

var io = require ('socket.io').listen(server);

io.on ('connection', function (socket) {
  console.log ('We have new connection!');
});

function send_time() {
  io.emit ('time', {message: new Date().toISOString()});
}
setInterval (send_time, 1000);

server.listen (PORT);
```

Listing 4.4.1. *main.js* with *socket.io* support

The last step we need to perform in the course of example 4.3, is to integrate the *socket.io* client-side code with the *index.html* file. We will start by including our *socket.io* library in the `<head>` section:

```
<script src='/socket.io/socket.io.js'></script>
```

Right below that (still in the `<head>` section), we will add a simple script that'll take care of establishing the connection and relaying messages. The code inside `<script></script>` tags will be ran by the client - the web browser, on the PC;

```
var socket = io();

socket.on ('time', function (data) {
  /* TODO */
});
```

Before we fill out the event-handler code for our custom-defined time event, we should also include a new paragraph with a 'test' identifier in the `<body>` section, so that the data we want to display will have a place to go:

```
<p id="test">JavaScript can change HTML content.</p>
```

Once we have a destination for our data, we can fill out the time event-handler:

```
socket.on ('time', function (data) {
  document.getElementById("test").innerHTML = data.message;
});
```

The complete contents of the *index.html* file, along with clearly delineated departures from the code found in example 4.3, are shown in *Listing 4.4.2*.

```
<!DOCTYPE html>
<html>

  <head>
    <script src='/socket.io/socket.io.js'></script>
    <script>

      var socket = io();

      socket.on ('time', function (data) {
        document.getElementById("test").innerHTML = data.message;
      });

    </script>
  </head>

  <body>
    <h1>Hello World!</h1>
    <p id="test">JavaScript can change HTML content.</p>
  </body>

</html>
```

Listing 4.4.2. *index.html with socket.io support*

Once we restart the server by issuing:

```
nodejs main.js
```

and having refreshed the website at <http://192.168.0.1:8080> we should now be observing the effects shown in *Figure 4.4.1*.



Figure 4.4.1. An example of communication from the web server to the browser

4.5. Node.js - live streaming gyroscope readings



Source code for example 4.5:

- `/root/linux-academy/4-5/main.js`
- `/root/linux-academy/4-5/index.html`

In order to avoid having to rewrite our gyroscope handling code, we are going to reuse the `gyro-i2c` executable, along with a built-in Node.js module called `child_process`. We'll create a new child process by using the `spawn()` method, and define a callback for it to handle its `stdout` - it will be called each time `gyro-i2c` gives a new data point.

Just like in the previous examples, we'll base our code on what we wrote in the previous exercise.

```
var spawn = require('child_process').spawn;
```

In the next step, `spawn()` is used to create the child process - it will be handling running `gyro-i2c`:

```
var child = spawn ('/tmp/gyro-i2c');
```

The `gyro-i2c` binary must be copied to `/tmp`:

```
root@localhost:~# cp ~/fxas2100x/gyro-i2c /tmp
```

The final change we need to make in `main.js` is to add callback functions to handle `stdout` (which will send the read data to the browser via an `xyz` message) and `stderr` (which will report any errors generated by `gyro-i2c` to the console) of the process:

```
child.stdout.on ('data', function (data) {  
  io.emit ('xyz', {message: data.toString().split('\n')[0]});  
});
```

```
child.stderr.on ('data', function (data) {
  console.log ('stderr: ' + data);
});
```

It may also be worth it to implement a close event handler, so that we can be notified of the exit code returned by the child process:

```
child.on ('close', function (code) {
  console.log ('exit: ' + code);
});
```

The complete source code of *main.js*, with the changes marked in bold, is shown in *Listing 4.5.1*.

```
var http = require ('http');
var fs = require ('fs');
var spawn = require('child_process').spawn;

var index = fs.readFileSync (__dirname + '/index.html');

var PORT = 8080;

var server = http.createServer (function handler (request, response) {
  response.writeHead (200, {'Content-Type': 'text/html'});
  response.end (index);
});
var io = require ('socket.io').listen(server);

io.on ('connection', function (socket) {
  console.log ('We have new connection!');
});

var child = spawn ('/tmp/gyro-i2c');

child.stdout.on ('data', function (data) {
  io.emit ('xyz', {message: data.toString().split('\n')[0]});
});

child.stderr.on ('data', function (data) {
  console.log ('stderr: ' + data);
});

child.on ('close', function (code) {
  console.log ('exit: ' + code);
});

server.listen (PORT);
```

Listing 4.5.1. *main.js* spawning a child process

Now, we need to modify *index.html* so that it can receive and report the readings for the the X, Y and Z axes. To do this, let us create a simple table in the `<body>` section to contain `x_val`, `y_val` and `z_val` fields:

```
<table>
  <tr>
    <th>X [deg]</th>
    <td><p id="x_val">---</p></td>
  </tr>
  <tr>
    <th>Y [deg]</th>
    <td><p id="y_val">---</p></td>
  </tr>
  <tr>
    <th>Z [deg]</th>
    <td><p id="z_val">---</p></td>
  </tr>
</table>
```

In the `<head>` section, let's now add a function to receive the rotation vector messages. Each read line will be split by the ' ' separator (space), and the results will be then assigned to the corresponding table fields:

```
<script>

  var socket = io();

  socket.on ('xyz', function (data) {
    var arr = data.message.split(" ");
    document.getElementById("x_val").innerHTML = arr[0];
    document.getElementById("y_val").innerHTML = arr[1];
    document.getElementById("z_val").innerHTML = arr[2];
  });

</script>
```

To improve the visual aesthetics of the table, we've included a few CSS formatting directives. The complete source code of *index.html*, along with clearly delineated departures from the code found in example 4.4, is shown in *Listing 4.5.2*.

```
<!DOCTYPE html>
<html>

  <head>

    <style>
      table, th, td {
        border: 1px solid black;
      }
      th, td {
        border: 1px solid black;
        padding: 15px;
      }
    </style>
  </head>
```

```

</style>

<script src='/socket.io/socket.io.js'></script>
<script>

  var socket = io();

  socket.on ('xyz', function (data) {
    var arr = data.message.split(" ");
    document.getElementById("x_val").innerHTML = arr[0];
    document.getElementById("y_val").innerHTML = arr[1];
    document.getElementById("z_val").innerHTML = arr[2];
  });

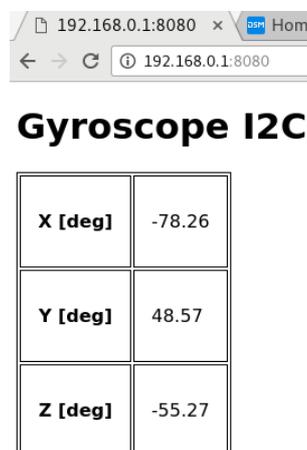
</script>
</head>

<body>
<h1>Gyroscope I2C</h1>
<table>
  <tr>
    <th>X [deg]</th>
    <td><p id="x_val">---</p></td>
  </tr>
  <tr>
    <th>Y [deg]</th>
    <td><p id="y_val">---</p></td>
  </tr>
  <tr>
    <th>Z [deg]</th>
    <td><p id="z_val">---</p></td>
  </tr>
</table>
</body>
</html>

```

Listing 4.5.2. *main.js with child process creation implemented*

After starting the server via nodejs main.js and refreshing the view of <http://192.168.0.1:8080> we should be seeing results presented in *Figure 4.5.1*.



The screenshot shows a web browser window with the address bar displaying '192.168.0.1:8080'. The page title is 'Gyroscope I2C'. Below the title is a table with three rows, each representing a coordinate: X [deg], Y [deg], and Z [deg]. The values are -78.26, 48.57, and -55.27 respectively.

X [deg]	-78.26
Y [deg]	48.57
Z [deg]	-55.27

Figure 4.5.1. *Presenting readouts in the web browser view*

4.6. Node.js - Adding 3D graphics with Three.js)

Showing three numerical values does not tell much about how an object moves in three-dimensional space. Fortunately, modern web browsers support a variety of APIs connecting the client-side Javascript to various pieces of the client's software and hardware. Among those APIs is WebGL, a wrapper around OpenGL, an API to render 3D graphics with the acceleration of the system GPU. WebGL, like OpenGL, is a fairly low-level API. Instead of calling its functions directly, we shall use a free library called *three.js* to create a 3D model and render it on an HTML `<canvas>` element.



Source code for example 4.6:

- `/root/linux-academy/4-6/main.js`
- `/root/linux-academy/4-6/index.html`
- `/root/linux-academy/4-6/three.min.js`



Make sure your browser supports the WebGL v1 API:

<http://webglreport.com/>



The three.js library, in its compacted form, has to be available to the webpage. The following command can be used to download it from the project's site:

`wget http://threejs.org/build/three.min.js`

During the hands-on, here is no need to download three.min.js. It has already been included in the project.

The library uses the HTML `<canvas>` element to draw onto. A 500x500px canvas is placed in the document:

```
<canvas id="mycanvas" width="500" height="500"></canvas>
```

Link the *Three.js* library in the `<head>` section to use it:

```
<script src='three.min.js'></script>
```

An `init()` function is declared, where the perspective, geometry and materials are set up, and a mesh is added to the scene:

```
function init() {  
    scene = new THREE.Scene();
```

```

camera = new THREE.PerspectiveCamera (70, 500/500, 0.01, 10);
camera.position.z = 0.5;

geometry = new THREE.BoxGeometry (0.2, 0.2, 0.2);
material = new THREE.MeshNormalMaterial();

mesh = new THREE.Mesh (geometry, material);
scene.add (mesh);

renderer = new THREE.WebGLRenderer ({ canvas: mycanvas});
renderer.setSize (500, 500);
document.body.appendChild (renderer.domElement);
}

```

`THREE.PerspectiveCamera()` sets the viewing angle, aspect ratio, near and far rendering depth limits.

The camera is placed at $(0.0, 0.0, 0.5)$ and looks at $(0.0, 0.0, 0.0)$:

```

camera = new THREE.PerspectiveCamera (70, 500/500, 0.01, 10);
camera.position.z = 0.5;

```

Next, a cube mesh is created:

```

geometry = new THREE.BoxGeometry (0.2, 0.2, 0.2);
material = new THREE.MeshNormalMaterial();
mesh = new THREE.Mesh (geometry, material);
scene.add (mesh);

```

The material used to render the faces of the cube is set to `MeshNormalMaterial`. It is a special type of material which maps the normal vector of a surface (i.e. a perpendicular unit vector) to its RGB color, giving a nice visual effect.

Finally, a WebGL renderer is created and assigned to the canvas:

```

renderer = new THREE.WebGLRenderer ({ canvas: mycanvas});
renderer.setSize (500, 500);
document.body.appendChild (renderer.domElement);

```

The `animate()` function rotates the mesh to follow the orientation of the sensor:

```

function animate() {

    requestAnimationFrame (animate);

    mesh.rotation.x = THREE.Math.degToRad(x);
    mesh.rotation.y = THREE.Math.degToRad(y);
    mesh.rotation.z = THREE.Math.degToRad(z);

    renderer.render (scene, camera);
}

```

Listing 4.6.1. shows the 3D graphics implementation in `index.html`. Changes from example 4.5 are in bold.

```

<!DOCTYPE html>
<html>

<head>

  <canvas id="mycanvas" width="500" height="500"></canvas>

  <style>
    table, th, td {
      border: 1px solid black;
    }
    th, td {
      border: 1px solid black;
      padding: 15px;
    }
  </style>

  <script src='/socket.io/socket.io.js'></script>
  <script src='three.min.js'></script>

  <script>

    var camera, scene, renderer;
    var geometry, material, mesh;
    var x, y, z;

    function init() {

      scene = new THREE.Scene();

      camera = new THREE.PerspectiveCamera (70, 500/500, 0.01, 10);
      camera.position.z = 0.5;

      geometry = new THREE.BoxGeometry (0.2, 0.2, 0.2);
      material = new THREE.MeshNormalMaterial();

      mesh = new THREE.Mesh (geometry, material);
      scene.add (mesh);

      renderer = new THREE.WebGLRenderer ({ canvas: mycanvas});
      renderer.setSize (500, 500);
      document.body.appendChild (renderer.domElement);
    }

    function animate() {

      requestAnimationFrame (animate);

      mesh.rotation.x = THREE.Math.degToRad(x);
      mesh.rotation.y = THREE.Math.degToRad(y);
      mesh.rotation.z = THREE.Math.degToRad(z);

      renderer.render (scene, camera);
    }
  </script>

```

```

    init();
    animate();

    var socket = io();

    socket.on ('xyz', function (data) {

        var arr = data.message.split(" ");

        x = arr[0];
        y = arr[1];
        z = arr[2];

        document.getElementById("x_val").innerHTML = x;
        document.getElementById("y_val").innerHTML = y;
        document.getElementById("z_val").innerHTML = z;
    });

</script>
</head>

<body>
  <h1>Gyroscope I2C</h1>
  <table>
    <tr>
      <th>X [deg]</th>
      <td><p id="x_val">---</p></td>
    </tr>
    <tr>
      <th>Y [deg]</th>
      <td><p id="y_val">---</p></td>
    </tr>
    <tr>
      <th>Z [deg]</th>
      <td><p id="z_val">---</p></td>
    </tr>
  </table>
</body>

</html>

```

Listing 4.6.1. *index.html with 3D animation*

index.html now links to *three.min.js*, and the browser will request it. The file's path needs to be added to *main.js*:

```

var url = require('url');
var server = http.createServer (function handler (request, response) {

    var pathname = url.parse(request.url).pathname;
    console.log("Request for " + pathname + " received.");

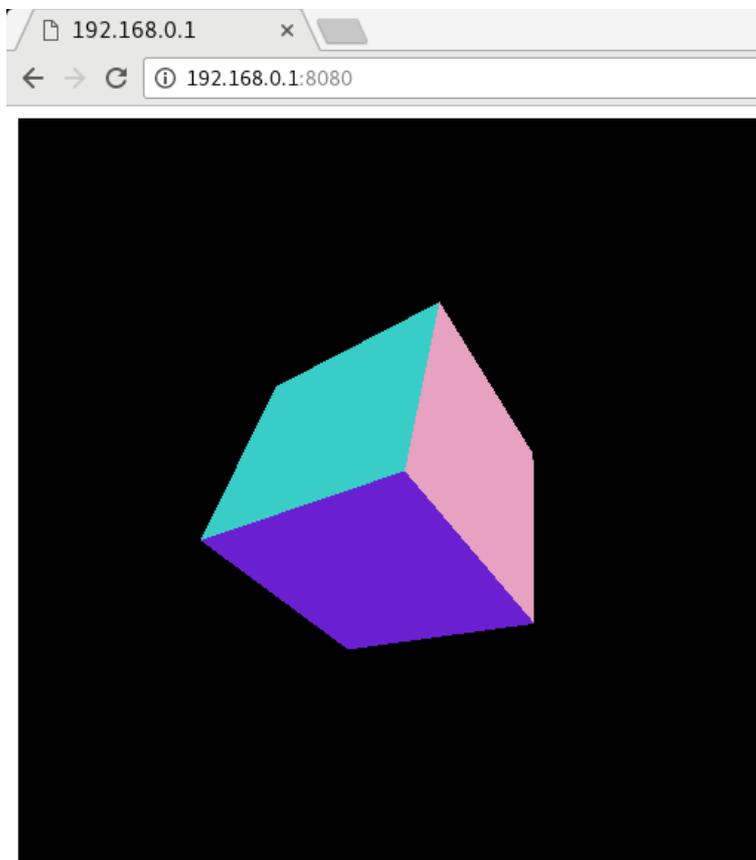
```

```
response.writeHead (200, {'Content-Type': 'text/html'});
if(pathname == "/") {
  var index = fs.readFileSync (__dirname + '/index.html');
  response.write (index);
} else if (pathname == "/three.min.js") {
  var script = fs.readFileSync (__dirname + '/three.min.js');
  response.write (script);
}
response.end();
});
```

Run the webserver with NodeJS:

```
nodejs main.js
```

Reload the URL in your browser (*http://192.168.0.1:8080*). You should now see the cube rotate as you move the gyroscope shield.



Gyroscope I2C

X [deg]	153.19
Y [deg]	125.43
Z [deg]	73.18

Figure 4.6.1. Gyroscope orientation represented by a WebGL-rendered 3D model.

