



SoMLabs

**HANDS-ON LINUX ACADEMY 2019**

*MARCH 28, 2019 IN ZEMST*

*MARCH 29, 2019 IN BREDA*

**NXP**



# 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 use provided system image file and prepare an SD card 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, login through the serial console, and establish a WLAN Access Point needed for next exercises.
- **Exercise 2.** shows how various methods of reading and setting GPIOs, accessing I<sup>2</sup>C devices and transferring data through the SPI bus in the userspace. Also 1-Wire bus usage will be demonstrated. Examples will show how to handle this in C code or shell scripts.
- **Exercise 3.** 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 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 and configuring WLAN to work in Access Point mode.*

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://ftp.somlabs.com/Trainings/Hands-on-Linux-Academy-2019/linux-academy-2019.zip>

After you have downloaded and 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
```

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.

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/sdc1 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:

```
sudo 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
```

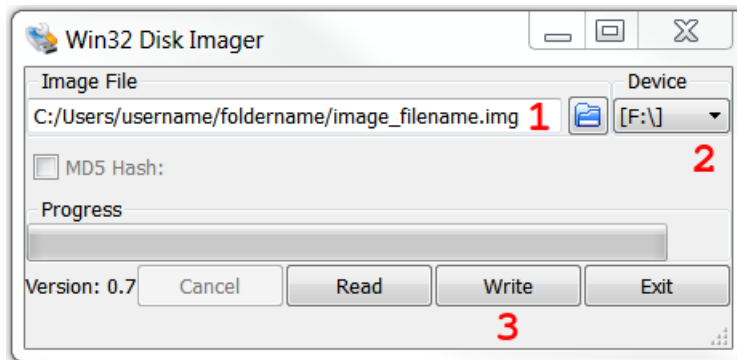
## 1.2. Preparing an SD card on a Windows PC

The 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:

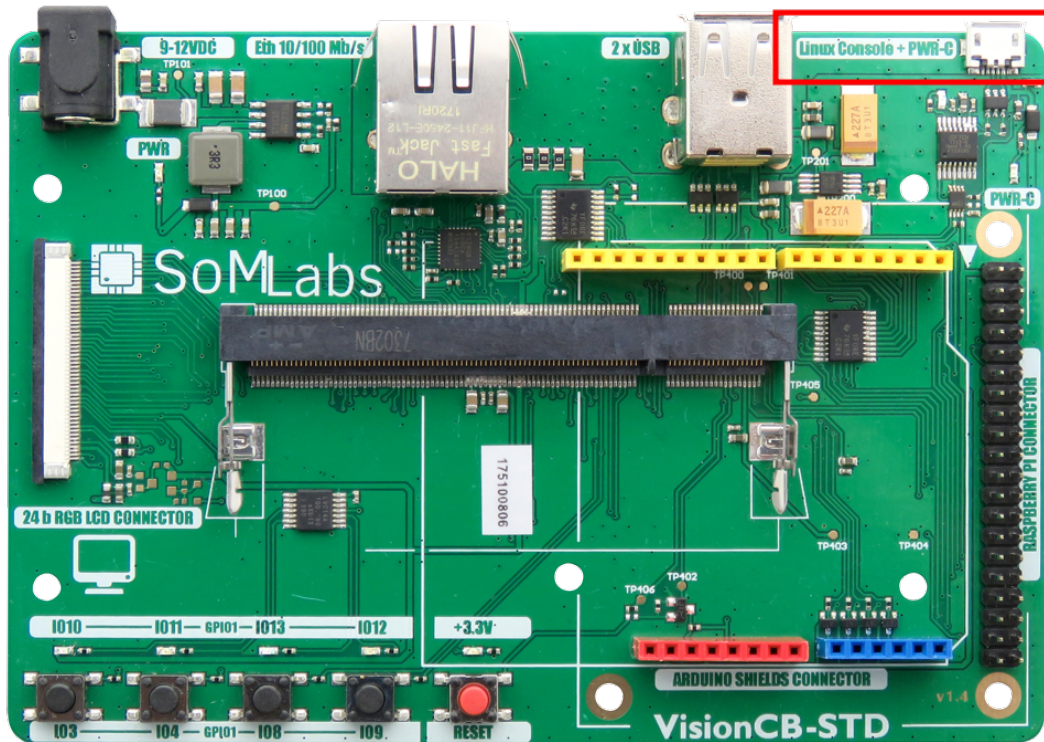
- (1) path to the \*.img file with the system image,
- (2) target device drive letter assigned by the system
- (3) press *Write*.



**Figure 1.** Win32DiskImager main window

### 1.3. First boot of *VisionSOM-6ULL* board

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. Network configuration - preparing WLAN interface to work in AP mode.**



*This chapter is just for reference - system image provided to you has all these changes already implemented.*

*You do not need to perform these steps!*

For this training, to simplify WLAN interface configuration, following changes were made to default network configuration::

- driver for used WLAN module *Murata 1DX* was installed. This driver is provided on SoMLabs web page as a compressed archive file. Installation was performed with following command:

```
cd /root
wget http://ftp.somlabs.com/visionsom-6ull-bcmfirmware.tar.xz

cd /
tar -xJf /root/visionsom-6ull-bcmfirmware.tar.xz
```

- with `apt-get` tool `rfkill` package was installed (it is used to control wireless interfaces), `hostapd` package (it is used to manage Access Point) and `dnsmasq` package (this is lightweight implementation of DHCP/DNS server):

```
apt-get install rfkill
apt-get install hostapd
apt-get install dnsmasq
```

- next, all wireless interfaces in system were enabled:

```
rfkill unblock all
```

- When our board is working as Access Point, it has to provide also DHCP server. To make it working properly, board itself must be configured with static IP address. This is achieved by modifying `/etc/network/interfaces` file:

```
auto wlan0
iface wlan0 inet static
    address 192.168.1.1
```



```
netmask 255.255.255.0
```

- also */etc/dnsmasq.conf* configuration file was modified – set of IP address used by DHCP server and lease time are defined:

```
interface=wlan0
dhcp-range=192.168.1.2,192.168.1.254,255.255.255.0,24h
```

- in last stage, WLAN module has been switched to AP mode (by default it is configured for STA mode):

```
echo 2 > /sys/module/bcmdhd/parameters/op_mode
```

## 1.5. Configuring, running and testing wireless network interface

To run *hostapd*, we need to prepare configuration file for it, named *hostapd.conf*. In this file we will configure some WLAN parameters, like SSID, mode, channel and encryption settings.

At first, we have to create file by using `touch` command.

```
touch /etc/hostapd/hostapd.conf
```

Next, with your favorite text editor (we have choice between `vim` and `nano`), in this new file write following parameters:

- name of used wireless interface:

```
interface=wlan0
```

- network identifier (SSID) and password:

```
ssid=<unique network identifier - max 32. characters>
wpa_passphrase=<password - min 8 characters>
```

- desired network mode - 802.11[x] and channel number:

```
hw_mode=g
channel=11
```

- SSID broadcasting options:

```
ignore_broadcast_ssid=0
```

- encryption options:

```
auth_algs=1
wpa=2
wpa_key_mgmt=WPA-PSK
rsn_pairwise=CCMP
```



*Default text editors installed on VisionSOM Debian system are powerful, but can be difficult to use for beginners.*

*To simplify tasks from chapter 1.5 both needed files are prepared for you in this location:*

*~/linux-academy/1-5/*

*You can simply copy this files to desired location:*

*cp ~/linux-academy/1-5/hostapd.conf /etc/hostapd/*

*cp ~/linux-academy/1-5/index.html /var/www/*

Once configuration file is saved, we can start AP mode with command:

```
hostapd -B /etc/hostapd/hostapd.conf
```

This command will run `hostapd` in background, as a daemon, with parameters provided in given configuration file.

If file content is correct and `hostapd` is run successfully, you will see following messages on serial console:

```
...  
wlan0: interface state UNINITIALIZED->ENABLED  
wlan0: AP-ENABLED
```

On your PC or mobile device you have to search fir SSID defined in configuration file and connect to this network.

To validate connection, we will run WWW server with simple webpage, by using `httpd` Linux command, provided by `busybox` package.

To do this, perform following steps:

```
mkdir /var/www  
cd /var/www
```

Then create file `index.html` in directory `/var/www` containing:

```
<html>  
  <head>  
    <h1> Hello! </h1>  
  </head>  
</html>
```

You can also copy provided example file:

```
cp ~/linux-academy/1-5/index.html /var/www/
```

And finally start WWW server with command:

```
busybox httpd -f -h /var/www
```

Once this is done, we can test it by running web browser on attached device (i.e. PC) and browsing IP address `192.168.1.1`

## EXERCISE 2

*Introduction to Embedded Linux - accessing GPIOs, I<sup>2</sup>C, SPI and 1-Wire buses.*

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 file system,

Next, using SPI, I<sup>2</sup>C and 1-Wire buses will be demonstrated.

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

The I<sup>2</sup>C 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 buses and GPIOs.*

## 2.1. Accessing GPIOs via /sys/class/gpio

In embedded device powered by Linux, or even more widely - on any device powered by Linux, only kernel has direct access to all peripherals. All user space processes can access peripherals only by using dedicated kernel driver API's. For GPIO's Linux kernel provides input drivers (GPIO Buttons subsystem), output drivers (LED Class driver subsystem) and generic input/output drivers (GPIO subsystem).



*Due to limited time for our training, system image provided to you already contains properly configured and compiled Linux kernel for all this tasks.*

*Linux kernel configuration and building, as well device tree introduction will be discussed during training briefly.*

One of simplest and most generic method to control GPIO ports is use of sysfs GPIO interface. This interface allows any process running in a user space access to free GPIO's (not used by any other process nor kernel itself).

To enable this interface, corresponding option in kernel configuration needs to be set:

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

From user space process, access to information provided by GPIO subsystem is possible via set of files available in directory `/sys/class/gpio`:

```
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
```

To start using given GPIO port from user space, we have to perform “export” operation. This operation informs kernel that user space process wants to use given resource. Once we have finished and GPIO is not needed anymore in user space, we can “unexport” it - inform kernel that it is not used anymore by user space.

Only free (not used by kernel itself) GPIO can be exported. Moreover, kernel does not control which process exports and uses given GPIO. It is correct to export GPIO in one process and use it in totally different process.

To “export” given GPIO port to user space, just run following command:

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



*More details about naming and numbering GPIO's will be given during training.*

Each GPIO line exported to user space is represented by dedicated directory under `/sys/class/gpio/` hierarchy:  
`/sys/class/gpio/gpioX`, where X is give port number.

To see content of this directory, we can use `ls`` command:

```
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
```

Every ooperation on given GPIO can be performed by accessing file in this directory. Most important file here are:

- *direction* - allows controlling of port direction. To configure port as output, "out" has to be written to this file, and to configure port as input "in" has to be written:
  - set gpioX configured as output:  
echo out > /sys/class/gpio/gpioX/direction
  - set gpioX configured as input:  
echo in > /sys/class/gpio/gpioX/direction

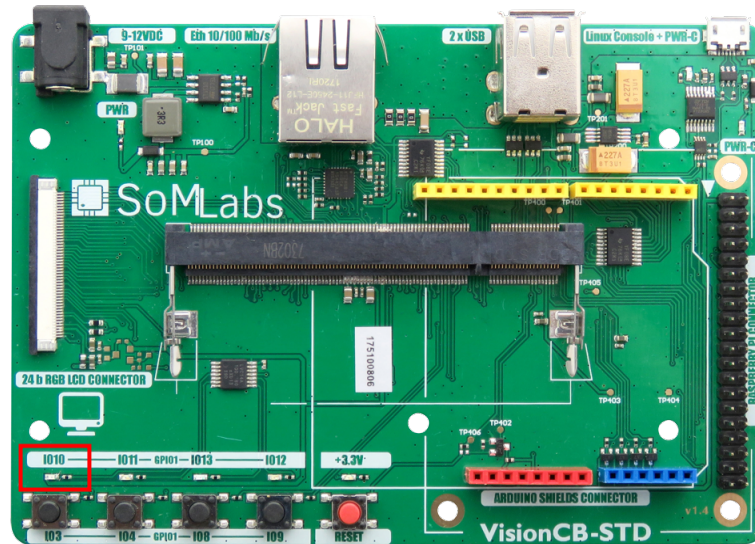
*Usage of given port as input or output can be limited in some circumstances due to hardware limitation - it is possible that given GPIO port can work only as input or only as output!*

- *value* - for GPIO configured as output, writing 0 sets low state on given port, and writing 1 set port in high state.  
In case GPIO is configured as input, port state can be read by reading this file:
  - set low state on gpioX:  
echo 0 > /sys/class/gpio/gpioX/value
  - read state of gpioX:  
cat /sys/class/gpio/gpioX/value

- *edge* - sets which edge should trigger interrupt. Possible values to be written to this file are: none, rising, falling or both, i.e.:
  - trigger on gfalling edge on `gpioX`:  
`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;
}

```

Use gcc compiler to compile the program and run it:

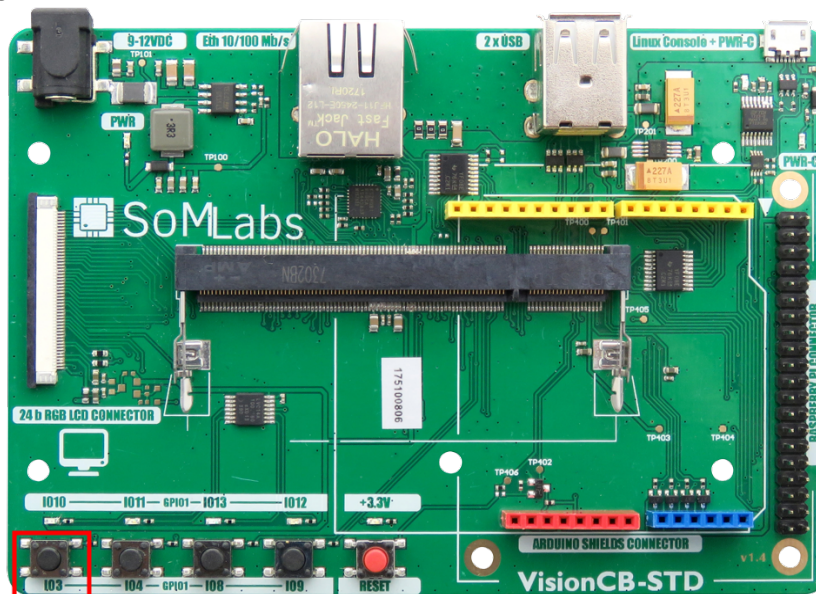
```

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

```

## 2.4. Button input with use of poll() system function

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 buton 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;
}
```

A `poll()` system call expects an array of descriptors, which will be 'monitored', and it 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, "falling") < 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;
}
```

By using functions from chapter 2.3, `GPIO1_3` is configured as input. Next, by using `gpio_set_edge()`, to edge file we are writing value "falling" - interrupt will be triggered when signal is changed from high level to low level.

`gpio_fd_open()` function is used to open `value` file and return its file descriptor, needed for `poll()` function call.

A `poll()` function as first parameter expects pointer to array with `pollfd` structures:

```
struct pollfd
{
    int fd;           /* file descriptor */
    short events;    /* expected events */
    short revents;   /* occurred events */
}
```

In this example, array will contain just single file descriptor - pointing to our "value" file, and we will wait for "POLLPRI" event (data to read). We will put -1 as last parameter, and its meaning is "infinity" - so no any timeout for event to occur.

So our `poll()` function call will look like:

```
struct pollfd fdset[1];
int nfds = 1;

fdset[0].fd = fd;
fdset[0].events = POLLPRI;

ret = poll (fdset, nfds, -1);
```

After pressing a button, on GPIO `GPIO1_3` falling edge will be generated, at it will "unlock" `poll()` call. By checking condition:

```
if (fdset[0].revents & POLLPRI)
```

we are testing if `POLLPRI` event occurred for give file descriptor(of course we need this check mainly if there is more than one descriptor stored in `pollfd`).

Again, we compile `button.c` file with `gcc`:

```
root@localhost:~# gcc button.c -o button
```

And now we can run it - each button press should trigger message on console:  
`"poll(): GPIO_3 interrupt occurred"`

```
root@localhost:~# ./button
poll(): GPIO_3 interrupt occurred
poll(): GPIO_3 interrupt occurred
```

## 2.5. GPIO Buttons support with Linux input subsystem

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 device tree 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
```

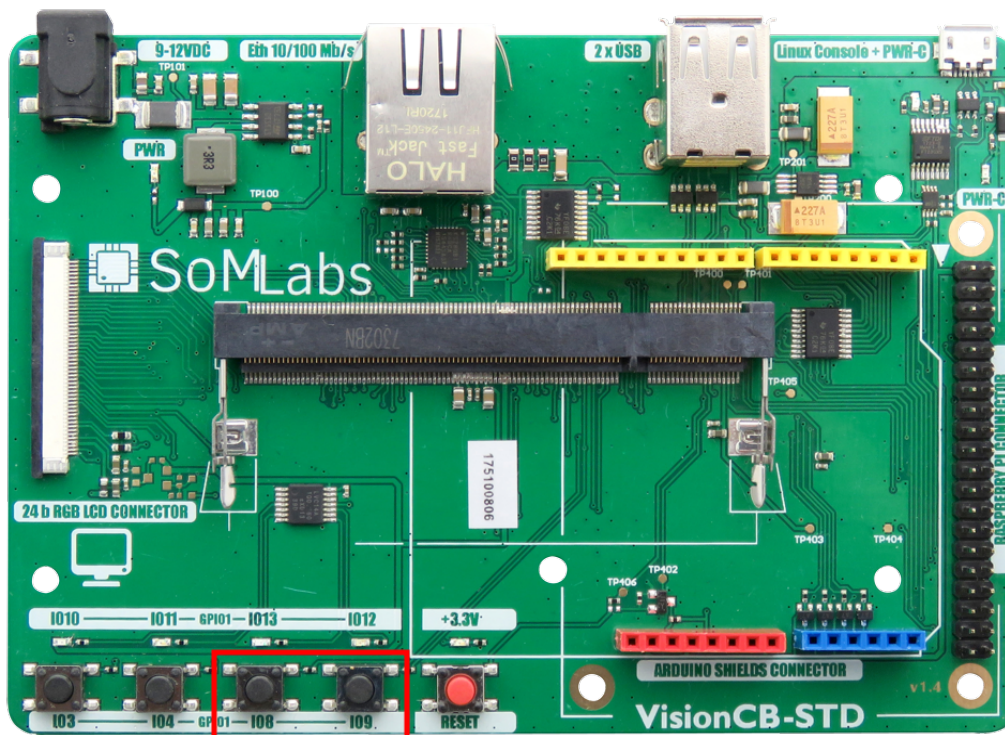
Every button needs to have an entry to map it to a key code:

```
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
```

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
00000010 44d5 59d1 da16 0000 0000 0000 0000 0000
00000020 44d5 59d1 d5f7 0002 0001 006c 0001 0000
00000030 44d5 59d1 d5f7 0002 0000 0000 0000 0000
```

Notice that 0x6c equals 108 decimal, which is the *KEY\_DOWN* keycode.

Example code provided on *Listing 2.5.1* below shows how 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*

Compile source code as usual:

```
root@localhost:~# gcc gpio-keys.c -o gpio-keys
```

And next test it:

```
root@localhost:~# ./gpio-keys
KEY_UP: press
KEY_UP: release
```

```
KEY_DOWN: press
KEY_DOWN: release
```

## 2.6. LED handling with *LED Class Driver*



*There is no separated C source code for this chapter, as handling LED class driver is very similar to generic GPIO interface.*

*Most interesting things about LED class driver are triggers, which can drive LED on various system states/operations, etc.*

### [1] Configuration of *LED Class Driver*:

```
Device Drivers --->
[*] LED Support --->
  <*> LED Class Support
  <*> LED Support for GPIO connected LEDs
```

### [2] Triggers for *LED Class Driver*:

```
Device Drivers --->
[*] LED Support --->
  <*> LED Trigger Support
    <*> LED Heartbeat Trigger
    <*> LED CPU Trigger
    <*> LED Default ON Trigger
```

### [3] Part of *Device Tree* file defining LED's connected to board:

```
leds {

    compatible = "gpio-leds";
    pinctrl-0 = <&pinctrl_gpio_leds>;
    pinctrl-names = "default";

    led3 {
        label = "led3";
        gpios = <&gpio1 13 GPIO_ACTIVE_HIGH>;
        linux,default-trigger = "heartbeat";
    };

    led4 {
        label = "led4";
        gpios = <&gpio1 12 GPIO_ACTIVE_HIGH>;
        linux,default-trigger = "mmc1";
    };
};
```



## 2.7. I2C bus example - reading data from gyroscope module



*For exercises 2.7, 2.8 and 2.9, instruction contains only short description needed to perform this exercises. More details will be provided during training.*

### [1] Enabling I2C bus in kernel:

```
Device Drivers --->
[*] I2C support --->
  <*> I2C device interface
```

### [2] Enabling I2C controller driver for iMX6:

```
Device Drivers --->
[*] I2C support --->
  [*] I2C Hardware Bus Support --->
    <*> IMX I2C interface
    < > GPIO-based bitbanging I2C
```

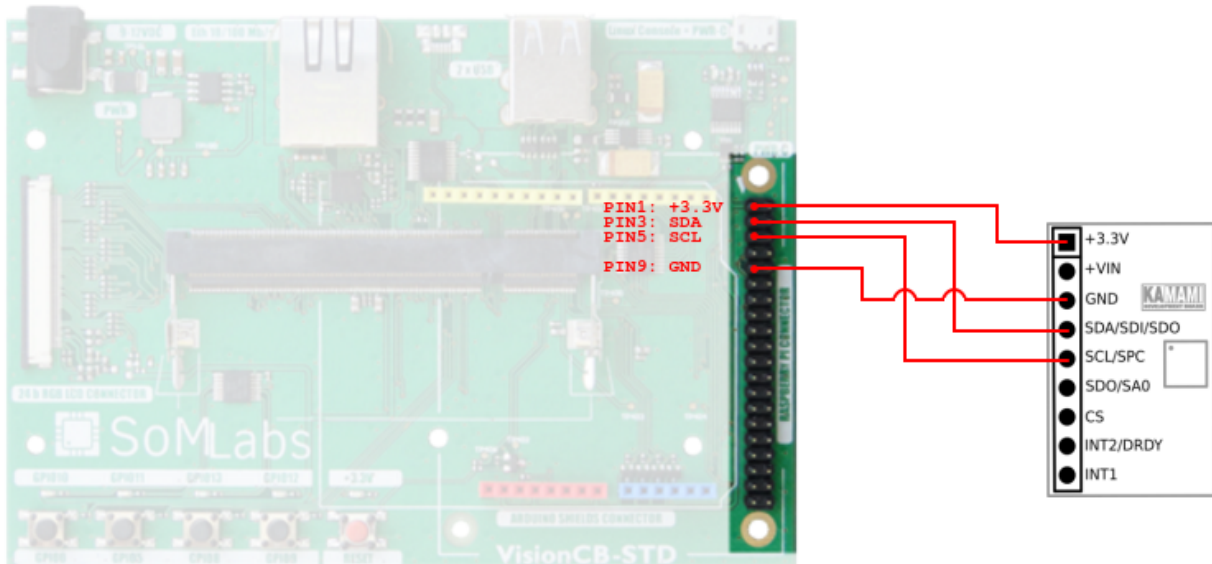
### [3] Device Tree description for I2C bus:

```
&i2c2 {
    clock_frequency = <100000>;
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_i2c2>;
    status = "okay";
};

&iomuxc {
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_hog_1>;
    imx6ul-evk {

        pinctrl_i2c2: i2c2grp {
            fsl,pins = <
                MX6UL_PAD_UART5_TX_DATA__I2C2_SCL 0x4001b8b0
                MX6UL_PAD_UART5_RX_DATA__I2C2_SDA 0x4001b8b0
            >;
        };
    };
};
```

### [4] Connection of gyroscope module:



[5] Compile and run example code *gyro-i2c.c*

```
root@localhost:~# cd /root/linux-academy/2-7
root@localhost:~/linux-academy/2-7# gcc gyro-i2c.c -o gyro-i2c
root@localhost:~/linux-academy/2-7# ./gyro-i2c
-8.1 23.6 -13.0
```

## 2.8. Testing SPI with a loopback connection

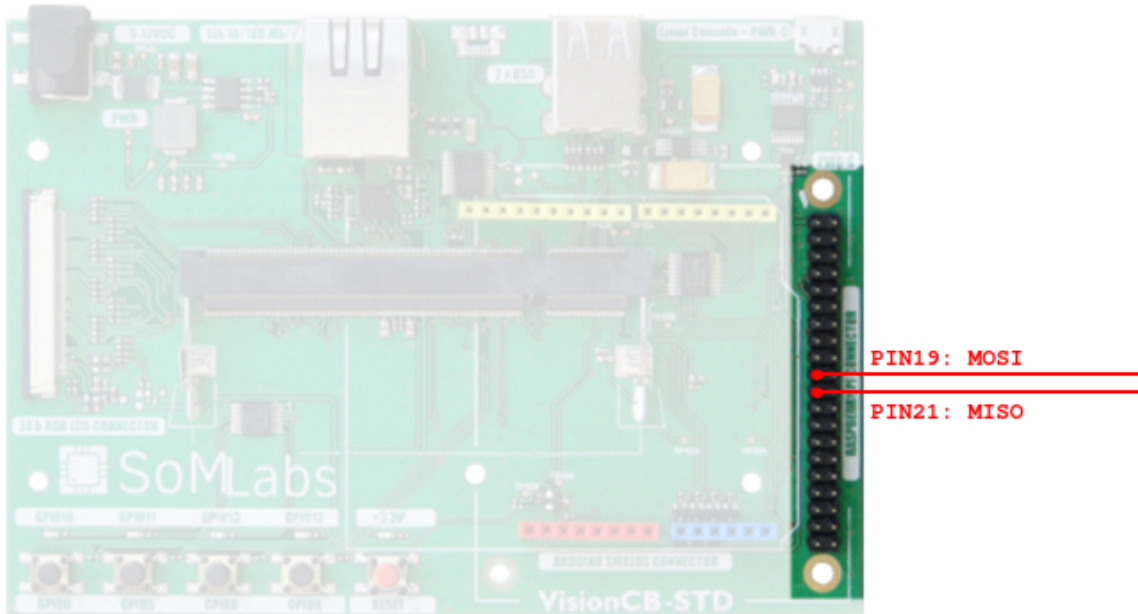
[1] Enable SPI bus support in kerne:

```
Device Drivers --->
[*] SPI support --->
  <*> Freescale i.MX SPI controllers
  < > GPIO-based bitbanging SPI Master
```

[2] enable user space access to SPI bus:

```
Device Drivers --->
[*] SPI support --->
  <*> User mode SPI device driver support
```

[3] Hardware connection:



#### [4] Compile and run test application *loopback-spi.c*

```
root@localhost:~# cd /root/linux-academy/2-8
root@localhost:~/linux-academy/2-8# gcc loopback-spi.c -o loopback-spi
root@localhost:~/linux-academy/2-8# ./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
```

If hardware connection is not correct *loopback-spi*, will print:

```
root@localhost:~/linux-academy/2-8# ./loopback-spi
FF FF FF FF FF FF
FF FF FF FF FF FF
FF FF FF FF FF FF
FF FF FF FF FF FF
FF FF FF FF FF FF
FF FF FF FF FF FF
FF FF
```

## 2.9. 1-Wire bus - DS18B20 temperature sensor

### [1] Enable 1-Wire bus support in kernel:

```
Device Drivers --->
  <*> Dallas's 1-wire support --->
    1-wire Bus Masters --->
      < >DS2490 USB <-> W1 transport layer for 1-wire
      < > Maxim DS2482 I2C to 1-Wire bridge
      <*> GPIO 1-wire busmaster
```

### [2] Enable device driver for 1-Wire temperature sensors:

```
Device Drivers --->
  <*> Dallas's 1-wire support --->
    1-wire Slaves --->
      <*> Thermal family implementation
      < > 1kb EEPROM family support (DS2431)
```

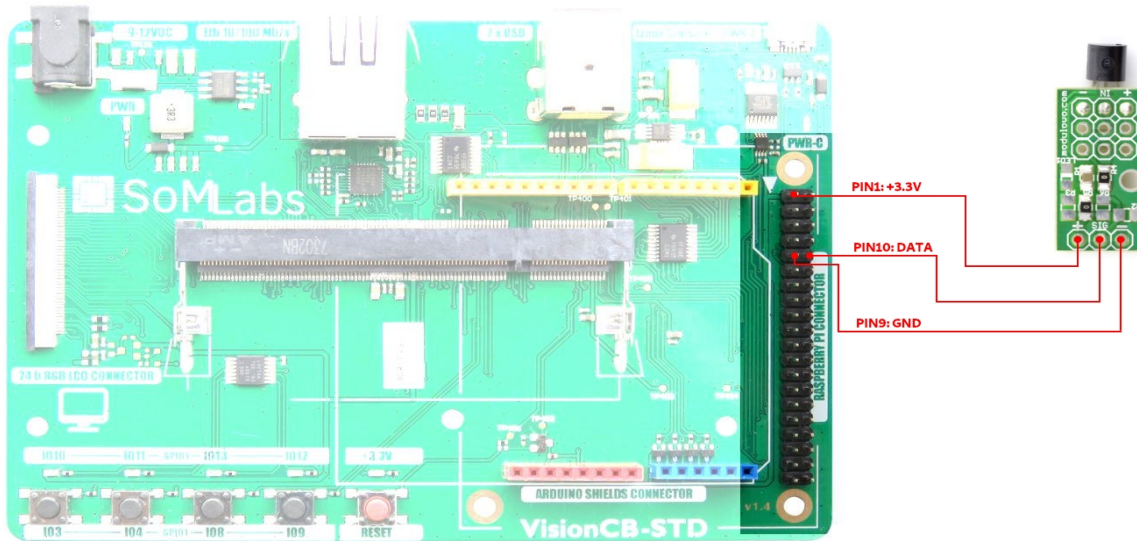
### [3] Device Tree entry:

```
onewire {
    compatible = "w1-gpio";
    pinctrl-0 = <&pinctrl_w1_gpio>;
    pinctrl-names = "default";
    gpios = <&gpio1 29 0>;
};

&iomuxc {
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_hog_1>;
    imx6ul-evk {

        pinctrl_w1_gpio: onewire {
            fsl,pins = <
                MX6UL_PAD_UART4_RX_DATA__GPIO1_IO29    0x4001b8b1
            >;
        };
    };
};
```

### [4] Hardware connection:



### Exercise 3

#### ***Time-to-market in embedded systems - usage of existing software components to build complex products.***

On today's consumer electronics market, customers expects more and more functional products, with perfect user experience, sophisticated UI and connectivity. And from developer point of view, product must be build in short time, with lowest possible cost which will not affect quality and reliability. One of possible solutions to achieve this is usage of existing software components, either open source or commercial.

And here Linux comes in - if you use it as underlying OS for your embedded system, you directly got access to lot of open source software - communication stacks, GUI libraries, connectivity libraries, etc. Lot of this software is already proved on mass market (Linux is used by Android as well!) and well tested.

In this chapter, we will show how simple it is to implement web interface for embedded device with use of free, open source libraries. By use of only minimal functionality of *Node.js* and *Three.js* library, with few lines of code, we will prepare web server visualizing gyroscope data as 3D cube!

### **3.1. 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 is already installed on your system.*

*You can check this by running command:*

```
root@localhost:~# nodejs -v
```

To test if it is installed correctly, run following command:

```
root@localhost:~# nodejs -v  
v8.11.4
```

If version number is shown properly, we can continue with our exercise.

## 3.2. Node.js - A basic web server



Full source code for example 3.2:

- `/root/linux-academy/3-2/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:

Complete content of `main.js` file is shown on *Listing 3.2.1*.

```
var http = require ('http');  
  
var PORT = 8080;  
  
var server = http.createServer (function handler (request, response) {  
  response.writeHead (200, {'Content-Type': 'text/plain'});  
  response.end ('Hello World!');  
});  
  
server.listen (PORT);
```

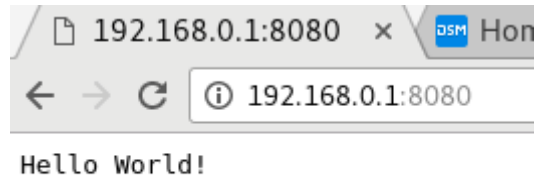
**Listing 3.2.1.** *main.js* implementing basic web server



Start the server using the command below:

```
nodejs main.js
```

You should now be able to reach `http://192.168.1.1:8080` from the web browser on your PC – see *Figure 3.2.1*.



**Figure. 3.2.1.** *NodeJS serving static content with the 'http' module*

### 3.3. Node.js - Serving local files



Full source code for this exercise is stored here:

- `/root/linux-academy/3-3/main.js`
- `/root/linux-academy/3-3/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 3.3.1* 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 3.3.1.** *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.

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



Full source code for this exercise is stored here:

- `/root/linux-academy/3-4/main.js`
- `/root/linux-academy/3-4/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 3.3, is shown in *Listing 3.4.1*

```
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 3.4.1.** *main.js with socket.io support*

The last step we need to perform in the course of example 3.3, 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 3.3, are shown in *Listing 3.4.2*.

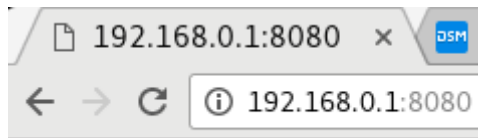
```
<!DOCTYPE html>  
<html>  
  
  <head>  
    <script src='/socket.io/socket.io.js'></script>  
    <script>  
  
      var socket = io();  
  
      socket.on ('time', function (data) {  
        document.getElementById("test").innerHTML = data.message;  
      });  
  
    </script>  
  </head>  
  
  <body>  
    <h1>Hello World!</h1>  
    <p id="test">JavaScript can change HTML content.</p>  
  </body>  
  
</html>
```

**Listing 3.4.2.** *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 3.4.1.



# Hello World!

2017-10-01T19:44:59.317Z

**Figure. 3.4.1.** *An example of communication from the web server to the browser*

### 3.5. Node.js - live streaming gyroscope readings



Full source code for this exercise is stored here:

- `/root/linux-academy/3-5/main.js`
- `/root/linux-academy/3-5/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 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 3.5.1*.

```
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 3.5.1.** *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 3.5.2.*

```

<!DOCTYPE html>
<html>

<head>

    <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>

        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>

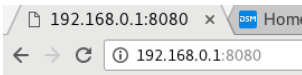
<body>
    <h1>Gyroscope I2C</h1>
    <table>
        <tr>
            <th>X [deg]</th>
            <td><p id="x_val">---</p></td>
        </tr>
        <tr>
            <th>Y [deg]</th>
            <td><p id="y_val">---</p></td>
        </tr>
    </table>

```

```
<tr>
  <th>Z [deg]</th>
  <td><p id="z_val">---</p></td>
</tr>
</table>
</body>
</html>
```

**Listing 3.5.2.** *main.js with child process creation implemented*

After starting the server via `nodejs main.js` and refreshing the view of `http://192.168.1.1:8080` we should be seeing results presented in Figure 3.5.1.



The screenshot shows a web browser window with the address bar containing '192.168.0.1:8080'. The page title is 'Gyroscope I2C'. Below the title is a table with three rows of data.

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

**Figure. 3.5.1.** *Presenting readouts in the web browser view*

### 3.6. Node.js - Adding 3D graphics with Three.js



Complete source code for this exercise is provided here:

- `/root/linux-academy/3-6/main.js`
- `/root/linux-academy/3-6/index.html`
- `/root/linux-academy/3-6/three.min.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.



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>
```

Next, we define needed variables:

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

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.

```
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 3.6.1.* shows the 3D graphics implementation in index.html. Changes from example 3.5 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);
}

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 3.6.1.** *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) {

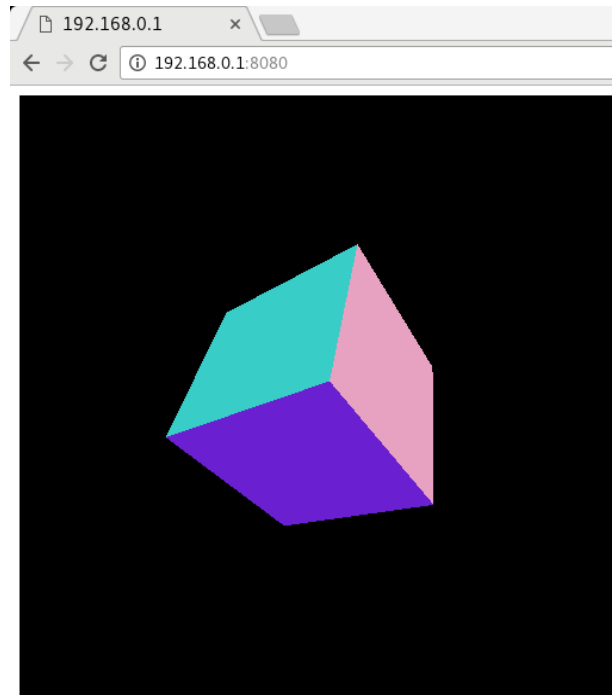
    var pathname = url.parse(request.url).pathname;
    console.log("Request for " + pathname + " received.");

    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://192.168.1.1:8080>). You should now see the cube rotate as you move the gyroscope shield*



### Gyroscope I2C

<b>X [deg]</b>	153.19
<b>Y [deg]</b>	125.43
<b>Z [deg]</b>	73.18

**Figure. 3.6.1.** Gyroscope orientation represented by a WebGL-rendered 3D model