Hands-on Linux Academy – instrukcja

Witaj na szkoleniu *Hands-on Linux Academy*! Niniejsze szkolenie zostanie poświęcone praktycznym aspektom wykorzystania najbardziej popularnych układów peryferyjnych i interfejsów komunikacyjnych (takich jak porty *GPIO*, magistrale *SPI* czy *I2C*) w systemie operacyjnym *Linux* i jego aplikacjach w urządzeniach *embedded*. Część praktyczna szkolenia została podzielona na cztery części:

- **ćwiczenie 1** – zajęcia praktyczne rozpoczniemy od przygotowania karty SD z systemem dla komputera *VisionSOM-6ULL,* która posłuży nam do dalszej realizacji zadań. Prowadzący omówi proces wgrywania gotowego obrazu systemu na kartę SD z wykorzystaniem komputerów pracujących pod kontrolą systemów Linux oraz Windows. Czas oczekiwania - niezbędny na przygotowanie karty - zostanie poświęcony na krótkie omówienie zalet wykorzystania systemów operacyjnych w urządzeniach embedded. Ćwiczenie numer 1 zostanie zakończone uruchomieniem komputera *VisionSOM6-ULL*, zalogowaniem do terminala systemowego i krótkiego omówienia konfiguracji sieci (komputer jednopłytkowy pełniący rolę serwera DHCP), niezbędnej do realizacji dalszych zadań.

- **ćwiczenie 2** – w tej części szkolenia uczestnicy zostaną zapoznani z teoretyczną i praktyczną obsługą wyprowadzeń GPIO oraz magistral SPI/I2C. Omówiona zostanie obsługa wyprowadzeń GPIO z poziomu konsoli systemu, skryptów powłoki oraz prostych aplikacji w języku C (z wykorzystaniem interfejsu */sys/class/gpio* oraz podsystemów *gpio-leds* oraz *gpio-keys*). Następnie, na przykładzie aplikacji w języku C, omówione zostaną aspekty programowej obsługi interfejsów SPI oraz I2C w przestrzeni użytkownika. Ćwiczenie numer 2 zostanie zakończone przygotowaniem prostej implementacji obsługi modułu żyroskopu w przestrzeni użytkownika.

- **ćwiczenie 3** – poświęcone technologiom NFC, NTAG.

- **ćwiczenie 4** – w ćwiczeniu tym przedstawiona zostanie możliwość prostego i szybkiego tworzenia bardziej rozbudowanych projektów sprzętowo-programowych z wykorzystaniem bibliotek gotowego i darmowego oprogramowania. Wykorzystując wyłącznie minimalną funkcjonalność środowiska uruchomieniowego *Node.js* oraz biblioteki *Three.js*, przygotujemy prosty serwer WWW prezentujący wyniki danych pomiarów odczytanych z modułu żyroskopu w postaci animowanej kostki 3D.

Łączny czas trwania praktycznej części szkolenia to około 4.5h. Ze względu na szeroką tematykę poruszanych zagadnień, na potrzeby niniejszego szkolenia przygotowano dedykowany obraz systemu dla karty SD, zawierający odpowiednio skonfigurowane jądro systemu Linux oraz kody źródłowe wszystkich przykładów prezentowanych w ramach warsztatów. Niniejsza instrukcja stanowi wyłącznie podsumowanie zagadnień omawianych w ramach warsztatów i zbiór zadań do wykonania przez jego uczestników – dodatkowe szczegóły teoretyczne, praktyczne wskazówki i informacje dotyczące poszczególnych etapów zostaną przekazane przez prowadzącego ćwiczenia, który wraz z uczestnikami wykona i omówi wszystkie zagadnienia. Zaczynajmy!

ĆWICZENIE 1
Przygotowanie karty SD z systemem dla komputera VisionSOM-6ULL

Firma *SoMLabs* - producent komputerów z serii *VisionSOM-6ULL* z procesorem *NXP i.MX6ULL* - w zależności o typu zainstalowanej pamięci, udostępnia trzy różne wersje konfiguracji sprzętowej modułów:

- moduł z pamięcią typu *eMMC*,
- moduł z pamięcią *NAND Flash*,
- oraz moduł z gniazdem karty *micro-SD*.

Wszyscy uczestnicy warsztatów *Hands-on Linux Academy* zostali wyposażeni w płytkę bazową *VisionCB-STD* oraz moduły *VisionSOM-6ULL* z zainstalowanym gniazdem karty micro-SD, tak więc niniejsze warsztaty rozpoczniemy od odpowiedniego przygotowania karty SD.

Obraz karty SD w postaci archiwum *linux-academy.zip*, zawierający odpowiednio skonfigurowany system operacyjny na potrzeby niniejszego szkolenia, można pobrać z Internetu: *http://somlabs.com/os_images/linux-academy.zip*.

Po rozpakowaniu pliku, możemy przystąpić do wgrania obrazu systemu *linux-academy.img* na kartę micro-SD. Kartę umieszczamy w dołączonym do zestawu czytniku kart, a następnie podłączamy do gniazda USB w komputerze PC. W zależności od typu wykorzystywanego systemu operacyjnego, poniżej przedstawiono dwie alternatywne metody przygotowania karty SD w systemach Linux (punkt 1.1) oraz Windows (punkt 1.2).

1.1. Przygotowanie karty SD w systemie operacyjnym Linux

Jedną najprostszych metod zapisania obrazu z pliku *linux-academy.img* na karcie SD jest wykorzystanie linuksowego narzędzia `dd`. Ogólna składnia polecenia `dd` może może zostać przedstawiona następująco:

```
dd if=<pliku_wejściowy> of=<plik_wyjściowy> <dodatkowe opcje>
```

Plik wejściowy polecenia (`if` – ***I****nput* ***F****ile*) będzie stanowił pobrany w uprzednim kroku obraz systemu *linux-academy.img*. Plik wyjściowy (`of` – ***O****utput* ***F****ile*) to umieszczona w czytniku i podłączona do komputera karta SD, która jest reprezentowana w systemie jako plik w katalogu */dev*. W celu zweryfikowania, który wpis jest odpowiedzialny za podłączone urządzenie, korzystając z terminala systemowego wydajmy polecenie:

```
dmesg
```

które wyświetla komunikaty wypisywane w buforze komunikatów jądra, w tym informacje o ostatnio podłączonych urządzeniach, np:

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

W powyższym przypadku wpis */dev/sdb* odnosi się do całej karty SD, natomiast wpisy */dev/sdb1*, */dev/sdb2*, *dev/sdbX*, do kolejnych partycji urządzenia - o ile zostały utworzone.

*Uwaga! Polecenia dd używamy korzystając z praw administratora, tak więc należy się bezwzględnie upewnić, że wprowadzono poprawną wartość dla pliku wyjściowego. Błędnie określenie wyjścia (np. zamiana wpisu /dev/sdb na /dev/sdb), może spowodować nadpisanie danych na głównym dysku użytkownika. Z powodu częstych pomyłek, akronim narzędzia dd jest często rozwijany do „**d**estroy **d**isk” lub „**d**elete **d**ata”.*

Alternatywną formą sprawdzenia wpisu w katalogu */dev*, do którego została przypisana nasza karta SD, jest wykorzystanie narzędzia `df` (poprawne określenie urządzenia reprezentującego kartę SD jest niezwykle ważne – błędny wpis może spowodować np. usunięcie całej zawartości dysku twardego – patrz ramka powyżej). Narzędzie `df` (z ang. ***D****isk* ***F****ree*) jest prostą aplikacją wyświetlającą informacje o wolnym miejscu na wszystkich zamontowanych systemach plików:

```
df -h
```

Wśród wyświetlonych wpisów lokalizujemy ten odpowiedzialny za kartę SD:

```
Filesystem      Size  Used Avail Use% Mounted on
dev             7.8G     0  7.8G   0% /dev
run             7.8G  1.5M  7.8G   1% /run
/dev/sda1       235G  164G   60G  74% /
tmpfs           7.8G  353M  7.5G   5% /dev/shm
tmpfs           7.8G     0  7.8G   0% /sys/fs/cgroup
tmpfs           7.8G  8.9M  7.8G   1% /tmp
/dev/sdc1       2.0G  1.8G  106M  95% /run/media/lskalski/sdcard
```

W przypadku wykorzystania czytnika kart wbudowanego w komputer PC, sterownik urządzenia może zarejestrować kartę SD w postaci pliku o nazwie /dev/mmcblk0.

Poprawne określenie pliku wyjściowego reprezentującego podłączoną do czytnika kartę SD, pozwala na ostateczne określenie formy polecenia `dd`:

```
sudo dd if=/path/to/linux-academy.img of=/dev/sdX bs=4M oflag=dsync
```

Powyższe polecenie skopiuje obraz *linux-academy.img* na wskazaną kartę SD. Dodatkowe parametry polecenia `dd` określają zapis w blokach po 4MB (`bs` - ***B****lock* ***S****ize*) w sposób synchroniczny (bez buforowania).

Polecenie dd nie umożliwia nam monitorowania postępu zapisu danych na kartę SD, który w zależności od wielkości obrazu systemu może być dość czasochłonne. Wygodnym rozwiązaniem jest wykorzystanie narzędzia pv, który wyświetla informacje o postępie odczytu danych z pliku. Składnia polecenia z wykorzystaniem narzędzia pv, wygląda następująco:

pv linux-academy.img | dd of=/dev/zero bs=4M oflag=dsync

25.5MiB 0:00:02 [5.03MiB/s] [====>   ] 21% ETA 0:00:27

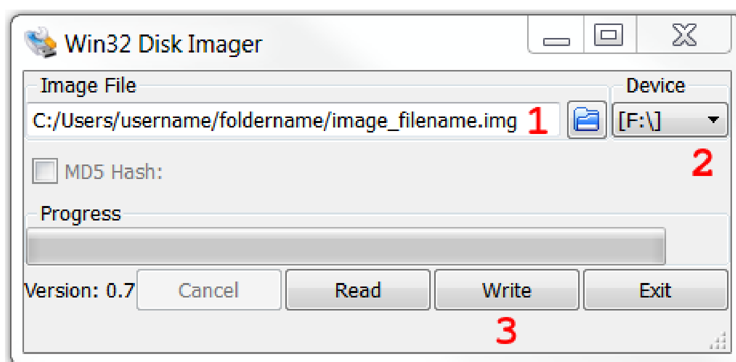
1.2. Przygotowanie karty SD w systemie operacyjnym Windows

Jedną z najprostszych opcji w przygotowaniu karty SD w systemie operacyjnym Windows jest wykorzystanie darmowego narzędzia *Win32DiskImager*, które jest dostępna do pobrania pod adresem:

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

Po umieszczeniu czytnika z kartą w slocie USB oraz zainstalowaniu i uruchomieniu oprogramowania *Win32DiskImager* użytkownik zostanie poproszony o (*Rysunek 1*):

- (1) wskazanie ścieżki do pliku *.*img* z obrazem systemu,
- (2) wskazanie urządzenia docelowego (wybieramy oznaczenie literowe dysku przypisane przez system do podłączonego czytnika kart SD),
- (3) przeprowadzenie operacji zapisu poprzez wybranie przycisku *Write*.

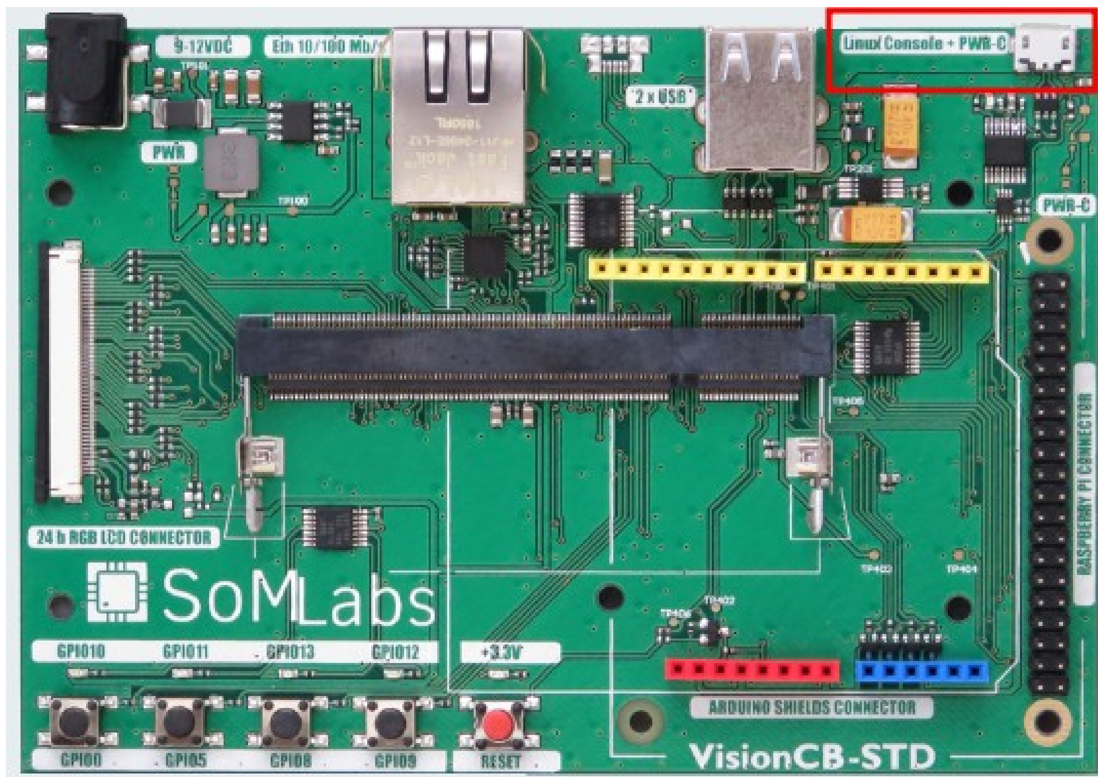


Rys. 1. *Główne okno aplikacji Win32DiskImager*

1.3. Uruchomienie płytki *VisionSOM-6ULL*

Po poprawnym przygotowaniu karty micro-SD z systemem i umieszczeniu jej w czytniku modułu *VisionSOM-6ULL*, możemy przystąpić do pierwszego uruchomienia zestawu. W tym celu łączymy moduł płytki z komputerem PC za pomocą dołączonego do zestawu kabla typu *PatchCord* (połączenie to nie jest wymagane na obecnym etapie wykonywania ćwiczeń, jednak będzie niezbędne do realizacji ćwiczenia 4), a następnie, za pomocą kabla typu *USB A↔microUSB B*,

łączymy komputer PC z gniazdem "*Linux Console + PWR-C*" umieszczonym na płycie zestawu *VisionCB-STD* (*Rysunek 2*).



Rys. 2. *Gniazdo USB – "Linux Console + PWR-C"*

Wyprowadzenie "*Linux Console + PWR-C*", pełni rolę gniazda zasilającego oraz jest podłączone do umieszczonego na płycie *VisionCB-STD* konwertera sygnałów UART↔USB. Aby uzyskać dostęp do konsoli systemu Linux, należy uruchomić dowolny program emulatora terminalu (np. *minicom*, *picocom*, *screen* w systemie Linux lub *Putty* w systemie Windows) z ustawieniami transmisji: **115200 8N1**. Jednym z najprostszych narzędzi dostępnych w większości dystrybucji systemu Linux, jest program `picocom`:

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

Po otwarciu połączenia należy zalogować się konto użytkownika **root** (bez hasła):

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

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

W przypadku systemu operacyjnego Linux i występujących problemów z nawiązaniem połączenia, należy upewnić się, że aktualnie zalogowany użytkownik posiada stosowane prawa odczytu/zapisu danego pliku urządzenia, tj: /dev/ttyUSB0, /dev/ttyACM0, itd.

1.4. Konfiguracja sieci – serwer DHCP

Poniższy opis ma charakter informacyjny – ta część instrukcji nie wymaga od uczestnika szkolenia wprowadzania modyfikacji w konfiguracji sieci – wszystkie poniżej opisane zmiany zostały już wprowadzone w domyślnie instalowanym obrazie.

Na potrzeby niniejszego szkolenia (uszczegóławiając - ćwiczenia numer 4 w którym zostanie przygotowany prosty serwer WWW), w stosunku do domyślnej konfiguracji sieci systemu *Debian* wprowadzono kilka zmian, których zwięzły opis przedstawiono poniżej:

- w docelowym systemie, za pomocą narzędzia `apt-get`, został zainstalowany pakiet `dnsmasq`, będący lekką implementacją serwera DHCP/DNS:

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

- w przyjętej konfiguracji sieci, komputer jednopłytkowy ma pełnić rolę serwera DHCP ze statycznie przypisanym adresem IP (192.168.0.1). W tym celu dokonano edycji pliku */etc/network/interfaces* w którym usunięto wpis:

  ```
  iface eth0 inet dhcp
  ```

  oraz dodano statyczną konfigurację sieci:

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

- dokonano modyfikacji pliku konfiguracyjnego */etc/dnsmasq.conf* w którym zdefiniowano między innymi pulę adresów przydzielanych przez serwer DHCP (192.168.0.2-254) oraz czas dzierżawy adresu:

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

Powyższa konfiguracja umożliwia szybkie ustanowienie bezpośredniego połączenia sieciowego pomiędzy komputerem jednopłytkowym a komputerem użytkownika i odczyt danych pomiarowych z żyroskopu, udostępnianych przez serwer WWW (którego przygotowanie będzie tematyką ćwiczenia numer 4).

ĆWICZENIE 2

Praktyczne wprowadzenie do Linux Embedded - obsługa portów GPIO oraz magistral SPI oraz I2C.

Ćwiczenie numer 2 rozpoczniemy od zapoznania się z jednym z najprostszych układów peryferyjnych (zarówno pod względem obsługi, jak i budowy) – portami *GPIO* (ang. ***G**eneral **P**urpose **I**nput/**O**utput* – wejścia/wyjścia ogólnego przeznaczenia). W kolejnych podpunktach niniejszego ćwiczenia, uczestnicy warsztatów zostaną zapoznani z programową obsługą magistral SPI oraz I2C. Umiejętności nabyte na tym etapie są niezbędne do realizacji najprostszych systemów wbudowanych, pracujących pod kontrolą systemu operacyjnego Linux. Przekonajmy się zatem czy naprawdę w Linuksie *„wszystko jest plikiem”*.

Programując „małe” 8-bitowe mikrokontrolery (bez wykorzystania systemów operacyjnych), wszystkie operacje wejścia-wyjścia realizowane są poprzez bezpośrednie operacje na rejestrach. Przed przystąpieniem do napisania programu, nieodzownym elementem jest wówczas dokładnie zapoznanie się z notą katalogową nowego mikrokontrolera. Po wykonaniu wszystkich zadań z ćwiczenia numer 2, warto zwrócić uwagę na fakt, że w jego treści nie umieszczono żadnych nazwy rejestru układu i.MX6ULL. Co więcej, wszystkie przygotowane kody źródłowe, można z powodzeniem uruchomić na dowolnych platformach z systemem Linux.

2.1. Sterowanie portami GPIO poprzez interfejs /sys/class/gpio

W urządzeniach wbudowanych pracujących pod kontrolą systemu operacyjnego Linux, bezpośredni dostęp do układów peryferyjnych ma wyłącznie jądro systemu. Procesy pracujące w przestrzeni użytkownika mogą uzyskać dostęp do sprzętu wyłącznie z wykorzystaniem dedykowanych sterowników sprzętu. Dla portów GPIO, jądro systemu Linux udostępnia sterowniki wyjścia (podsystem *Led Class Driver*) oraz wejścia (podsystem *GPIO Buttons*). Alternatywnym rozwiązaniem - przydatnym zwłaszcza w pracach projektowych z wykorzystaniem płytek deweloperskich w których urządzenia takie jak diody LED i przyciski mogą być podłączane „ad hoc” - jest wykorzystanie ogólnego sterownika GPIO, od którego rozpoczniemy niniejsze ćwiczenie.

Ze względu na ograniczony czas trwania warsztatów, przygotowany obraz systemu zawiera odpowiednio skonfigurowane i skompilowane jądro systemu do wykonania wszystkich prezentowanych zadań – zagadnienia związane z konfiguracją jądra systemu i Device Tree zostaną omówione przez prowadzącego podczas trwania warsztatów.

Jednym z najprostszych i najbardziej generycznym sposobem sterowania portami GPIO jest wykorzystanie ogólnego sterownika GPIO (*sysfs gpio interface*). Interfejs ten umożliwia dostęp aplikacjom użytkownika do aktualnie nieużywanych (przez inny sterownik) linii wejścia-wyjścia. Włącznie wsparcia dla ogólnego sterownika GPIO wymaga przedstawionej poniżej konfiguracji jądra systemu:

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

Z poziomu przestrzeni użytkownika, dostęp do informacji udostępnianych przez sterownik jest realizowany poprzez szereg plików dostępnych w katalogu *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
```

Z punktu widzenia obsługi wyprowadzeń GPIO, do dwóch najważniejszych pozycji należą pliki *export* i *unexport*. Pliki te umożliwiają udostępnienie/zwolnienie wybranego wyprowadzenia do przestrzeni użytkownika, o ile w danej chwili nie jest ono wykorzystywane przez inny sterownik. Wyeksportowanie wybranego wyprowadzenia odbywa się poprzez zapis jego numeru porządkowego do pliku *export*, np:

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

Bliższe szczegóły dotyczące zasad nazewnictwa, numeracji wyprowadzeń GPIO, konfiguracji jądra systemu oraz Device Tree, zostaną omówione przez prowadzącego warsztaty.

Każda linia GPIO wyeksportowana do przestrzeni użytkownika reprezentowana jest przez katalog */sys/class/gpio/gpioX*, gdzie X określa numer porządkowy portu. Poleceniem `ls` wyświetlmy zatem zawartość katalogu reprezentującego, uprzednio wyeksportowane, wyprowadzenie *GPIO_10*:

```
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
```

Ponieważ w Linuksie "wszystko jest plikiem", sterowanie wybranym portem GPIO realizowane jest poprzez zapis/odczyt plików umieszczonych w katalogu z oznaczeniem danego wyprowadzenia. Najważniejsze z nich to:

- plik *direction* – umożliwia sterowanie kierunkiem pracy danego wyprowadzenia (wejście/wyjście). Ustawienie kierunku pracy odbywa się poprzez zapis wartości `out` (dla wyjścia) lub `in` (dla wejścia) Dla przykładu:

    - port `gpioX` pracujący jako wyjście:
      ```
      echo out > /sys/class/gpio/gpioX/direction
      ```

    - port `gpioX` pracujący jako wejście:
      ```
      echo in > /sys/class/gpio/gpioX/direction
      ```

- plik *value* – jeżeli wyprowadzenie GPIO pracuje jako wyjście, wówczas zapis wartości 0 ustawia stan niski na wyjściu, natomiast zapis 1 - stan wysoki. W przypadku konfiguracji jako wejście, odczyt zawartości pliku umożliwia odczyt stanu logicznego linii:

    - ustawienie stanu niskiego na wyprowadzeniu `gpioX`:
      ```
      echo 0 > /sys/class/gpio/gpioX/value
      ```

    - odczyt stanu linii wejściowej `gpioX`:
      ```
      cat /sys/class/gpio/gpioX/value
      ```

- plik *edge* – umożliwia określenie zbocza sygnału jakie wyzwoli zgłoszenie zmiany stanu dla danego wyprowadzenia GPIO, skonfigurowanego jako wejście. Dopuszczane wartości to: `none`, `rising`, `falling` oraz `both`, np.:

    - wyzwolenie zboczem opadającym dla wejścia `gpioX`:
      ```
      echo falling > /sys/class/gpio/gpioX/edge
      ```

2.2. "Hello World" w systemach embedded - sterowanie diodą LED [skrypt powłoki]

Krótki teoretyczny wstęp zawarty w podpunkcie 2.1, umożliwia nam przygotowanie pierwszego prostego skryptu powłoki, którego zadaniem będzie sekwencyjne miganie diodą LED, podłączoną w płycie bazowej *VisionCB-STD* do wyprowadzenia `GPIO1_10` – *Rysunek 2.2.1*.

Rys. 2.2.1. *Lokalizacja diody LED podłączonej do wyprowadzenia GPIO1_10*

Kompletne kody źródłowe dla wszystkich omawianych zagadnień zostały umieszczone w katalogu:

- */root/linux-academy/<numer_rozdziału>/*

Kod prostego skryptu powłoki *blink.sh*, przedstawiono na *Listingu 2.2.1*.

```
#!/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. *Prosty skrypt powłoki realizujący miganie diodą LED*

Uruchomienie skryptu *blink.sh*:

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

2.3. "Hello World" w systemach embedded - sterowanie diodą LED [język C]

Skrypty powłoki są wygodnym narzędziem do szybkiego prototypowania prostych aplikacji, jednak większość programistów związanych dotychczas ze środowiskiem mikrokontrolerów, w realizowanych projektach preferuje rozwiązania przygotowane w języku C. W ramach podpunktu 2.3 zaimplementujemy funkcjonalny odpowiednik skryptu *blink.sh* z wykorzystaniem języka C.

Jak już wiemy, sterowanie wyprowadzeniami GPIO poprzez interfejs */sys/class/gpio* odbywa się na zasadzie prostych operacji odczytu i zapisu wybranych plików. Tak więc przygotowanie aplikacji w języku C, w głównej mierze opierać się będzie na operacjach plikowych – ich otwieraniu, zapisie, odczycie i zamykaniu. Korzystając z funkcji systemowych `open()`, `write()`, `read()` oraz `close()`, w pliku */root/linux-academy/2-3/blink.c*, przygotujmy zestaw trzech funkcji umożliwiających:

- wyeksportowanie wybranego wyprowadzenia do przestrzeni użytkownika:

```
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;
}
```

- zmianę kierunku pracy wyprowadzenia GPIO:

```
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;
}
```

- zmianę stanu na wyjściu wyprowadzenia 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;
}
```

Posiadając tak przygotowany zestaw funkcji, uzupełnienie ciała funkcji `main()`, sprowadza się do kilku linii kodu:

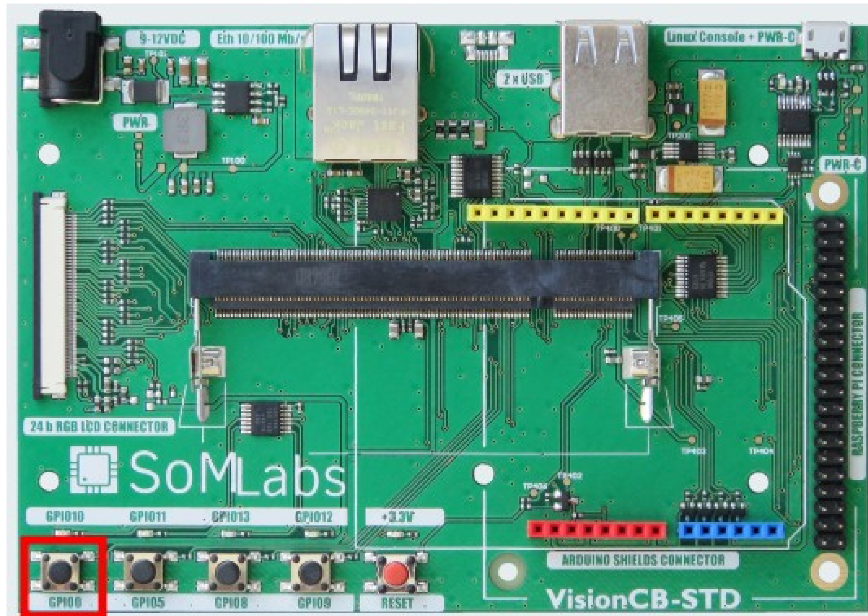
```
#define GPIO_PIN        10
#define GPIO_DIR        "/sys/class/gpio"
#define GPIO_IN         0
#define GPIO_OUT        1

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;
}
```

Korzystając z narzędzia `gcc`, wykonajmy kompilację kodu *blink.c*:

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

2.4. Obsługa przycisku z wykorzystaniem funkcji systemowej poll()

W dotychczas zrealizowanych zadaniach nie wykorzystano jeszcze funkcjonalności dostarczanej przez plik */sys/class/gpio/gpioX/edge*. Praktycznie zastosowanie zostanie przedstawione na przykładzie obsługi przycisku umieszczonego na płycie *VisionCB-STD* i podłączonego do wyprowadzenia `GPIO1_0` – *Rysunek 2.4.1*.



Rys. 2.4.1. *Lokalizacja przycisku podłączonego do wyprowadzenia GPIO1_0*

Nie ma wątpliwości, że próba programowej obsługi przycisku - w której program w nieskończonej pętli czyta zawartość pliku */sys/class/gpio/gpioX/value* - jest rozwiązaniem nieoptymalnym (dodanie funkcji `sleep()` pomiędzy kolejne odczyty zmniejsza responsywność urządzenia, natomiast brak opóźnień pomiędzy odczytami, pochłonie zasoby CPU). Dobrym rozwiązaniem tego problemu jest konfiguracja wejścia GPIO z wykorzystaniem pliku */sys/class/gpio/gpioX/edge* oraz jednej z systemowych funkcji z tzw. grupy „zwielokrotnień wejścia/wyjścia”, czyli `poll()` lub `select()`. Takie połączenie, umożliwia programiście np. wstrzymanie wykonywania procesu do czasu zmiany stanu przycisku lub łatwe „wpięcie” obsługi przycisku do głównej pętli programu (np. w `GMainLoop` z frameworku `GLib`).

Bazując na kodzie */root/linux-academy/2-3/blink.c*, utwórzmy plik *button.c* w którym zaimplementujemy przerwaniową obsługę przycisku podłączonego do wyprowadzenia `GPIO1_0` (program będzie blokował swoje działanie do momentu wystąpienia opadającego zbocza).

Edycję pliku *button.c* rozpoczynamy od dodania kodu funkcji `gpio_set_edge()`, która pozwoli na konfigurację zbocza wyzwalającego przerwanie:

```
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;
}
```

Konfiguracja funkcji systemowej `poll()`, której użyjemy w głównej pętli programu, wymaga przekazania zbioru deskryptorów, które będą przez tę funkcję "monitorowane" (funkcja `poll()` zostanie zablokowana do momentu gdy przekazane deskryptory plików będą gotowe do operacji wejścia/wyjścia lub dopóki nie zostanie przekroczony opcjonalnie określony limit czasowy). W realizowanym przykładzie do wywołania `poll()` przekazany zostanie deskryptor pliku `value`, który pobierzemy za pomocą generycznej funkcji `gpio_fd_open()`:

```
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;
}
```

Na ostatnim etapie edycji pliku *button.c*, zmodyfikujmy również zawartość głównej funkcji `main()`:

```
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;
}
```

Korzystając z funkcji przygotowanych w podpunkcie 2.3, konfigurujemy wyprowadzenie `GPIO1_0` do pracy jako 'wejście'. Następnie, korzystając z funkcji `gpio_set_edge()`, do pliku `edge` wpisujemy wartość `rising` - zmiana stanu wyprowadzenie zostanie zgłoszona przy wystąpieniu zbocza opadającego. Poprzez

funkcję `gpio_fd_open()` otwieramy i uzyskujemy deskryptor pliku `value`, który będzie niezbędny do konfiguracji funkcji systemowej `poll()`. Funkcja `poll()` jako pierwszy argument przyjmuje wskaźnik na tablicę struktur typu `pollfd`:

```
struct pollfd
{
  int fd;            /* deskryptor pliku               */
  short events;      /* zdarzenia oczekiwane           */
  short revents;     /* zdrzenia, które wystąpiły      */
}
```

W realizowanym przykładzie, w tablicy struktur `pollfd` umieścimy wyłącznie deskryptor pliku `value`, dla którego będziemy oczekiwać na zdarzenie `POLLPRI` ("istnieją pilne dane do odczytu"). Na zdarzenie to będziemy oczekiwać "w nieskończoność" – ostatni z argumentów wywołania funkcji `poll()` przyjmuje wówczas wartość -1. Finalna postać kodu dla wywołania systemowego `poll()`, będzie zatem wyglądać następująco:

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

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

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

Po wciśnięciu przycisku, na linii `GPIO1_0` wygenerujemy zbocze opadające, a tym samym odblokujemy funkcję `poll()`. Za pomocą warunku:

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

sprawdzamy, czy na danym deskryptorze wystąpiło zdarzenie `POLLPRI` (oczywiście sprawdzenie to jest zasadne wyłącznie wtedy, gdy tablica struktur `pollfd` zwiera więcej deskryptorów plików).
Wykorzystując narzędzia `gcc`, wykonajmy kompilację kodu *button.c*:

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

Oraz przetestujmy jego działanie (każde wciśnięcie przycisku powinno wygenerować komunikat o treści "*poll(): GPIO_0 interrupt occurred*"):

```
root@localhost:~# ./button
poll(): GPIO_0 interrupt occurred
poll(): GPIO_0 interrupt occurred
```

2.5. Obsługa przycisków z wykorzystaniem podsystemu GPIO Buttons

Jak wspomniano we wstępie do podpunktu 2.1, wykorzystanie generycznego interfejsu */sys/class/gpio* do sterowania pracą diod LED i przycisków jest zasadne w użyciu w przypadku prac badawczo-rozwojowych z płytkami deweloperskimi, w których konfiguracja sprzętowa nie została jeszcze jasno określona. W docelowych

rozwiązaniach sprzętowych (w których określono już liczbę diod LED oraz liczbę przycisków i pełnione przez nie funkcje), zastosowanie znajdują dedykowane sterowniki wyjścia (podsystem *Led Class Driver*) oraz wejścia (podsystem *GPIO Buttons*). Bohaterem tego podrozdziału będzie sterownik *GPIO Buttons.*

Analogicznie jak w poprzednich podrozdziałach, konfiguracja jądra systemu oraz Device Tree została wykonana na etapie przygotowywania obrazu. Niniejsza instrukcja zawiera wyłącznie krótki opis konfiguracji jądra – szczegóły zostaną przedstawione w trakcie trwania szkolenia.

Konfigurację sterownika *GPIO Buttons* rozpoczynamy od jego włączenia w jądrze systemu:

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

Niezbędne jest również włączenie tzw. interfejsu zdarzeń (*Event interface*) z podsystemu *Linux Input System*:

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

Do sterownika zostaną podłączone przyciski podpięte do wyprowadzeń `GPIO1_8` oraz `GPIO1_9`, których umiejscowienie na płycie *VisionCB-STD*, zostało przedstawione na *Rysunku 2.5.1*.

W opisie *Device Tree*, przyciskom `GPIO1_8` oraz `GPIO1_9` nadano odpowiednio oznaczenia `btn3` oraz `btn4` i przypisano funkcje `KEY_UP` oraz `KEY_DOWN`:

```
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> */
      };
};
```

Rys. 2.5.1. *Lokalizacja przycisku podłączonego do wyprowadzenia GPIO1_0*

Od strony procesu w przestrzeni użytkownika, dostęp do przycisków jest realizowany poprzez plik */dev/input/event2*. Bazując na informacjach z dotychczas zrealizowanych ćwiczeń, moglibyśmy łatwo wywnioskować, że odczyt tego pliku, np. poleceniem `cat`, będzie informował nas o stanie położenia przycisku (jeśli tak to którego – podłączonego do wyprowadzenia `GPIO1_8` czy `GPIO1_9`?). Przekonajmy się zatem:

```
root@localhost:~# cat /dev/input/event2
T
 ����T
       �T
          �
           ��T
```

Próba wykonania polecenia `cat` na pliku specjalnym */dev/input/event2* zakończy się odczytem "krzaków". Dlaczego? Zdarzenia informujące o wciśnięciu przycisku przekazywane są do przestrzeni użytkownika przez generyczny interfejs zdarzeń (*Event interface*). Każde z takich zdarzeń opisane jest strukturą `input_event`:

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

Struktura ta, poprzez określone pola dostarcza procesom użytkownika informacje o czasie wystąpienia zdarzenia (pole `time`), typie zdarzenia (pole `type`), kodzie użytego przycisku (pole `code`) oraz jego aktualnym położeniu przycisku (pole `value`). Przykładowy kod programu, realizujący odczyt danych z pliku */dev/input/event2* oraz ich interpretację, został zaimplementowany w pliku */root/linux-academy/2-5/gpio-keys.c* oraz przedstawiony na *Listingu 2.5.1*:

```
#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/event2", O_RDONLY);
  if (fd < 0)
    {
      printf ("Open /dev/input/event2 failed!\n");
      return EXIT_FAILURE;
    }

  while (1)
    {
      if (read(fd, &ev, size) < size)
        {
            printf ("Reading from /dev/input/event2 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. *Zawartość pliku /root/linux-academy/2-5/gpio-keys.c*

Wykonajmy kompilację kodu */root/linux-academy/2-5/gpio-keys.c*:

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

Oraz przetestujmy poprawność jego działania:

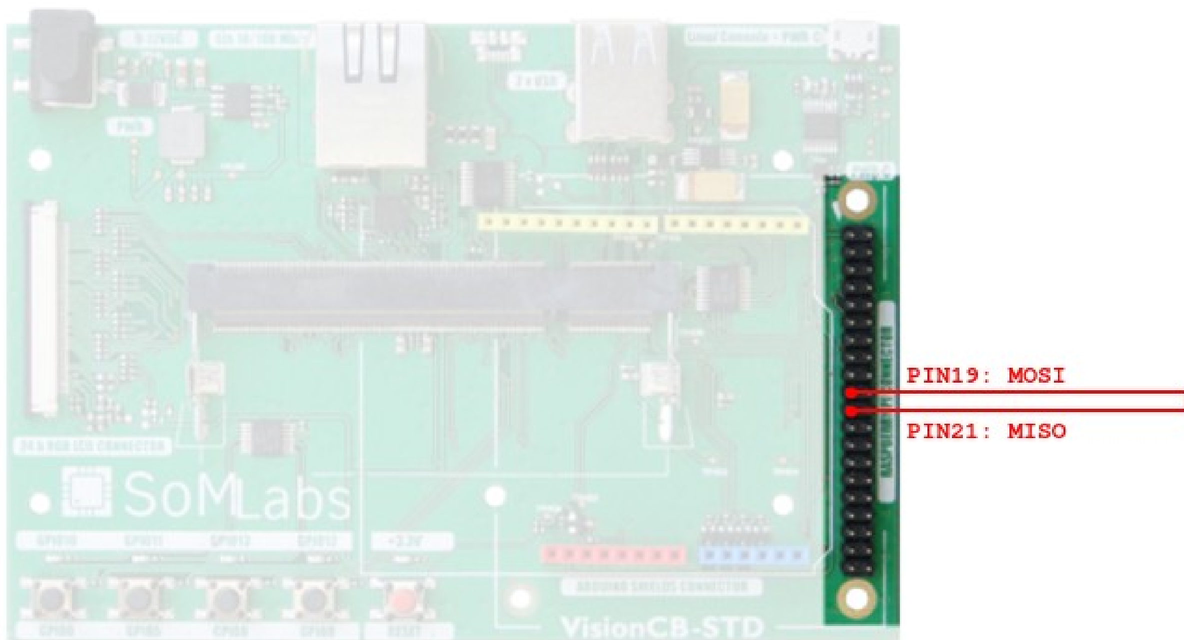
```
root@localhost:~# ./gpio-keys
KEY_UP: press
KEY_UP: release
KEY_UP: press
KEY_UP: release
KEY_DOWN: press
KEY_DOWN: release
KEY_DOWN: press
KEY_DOWN: release
```

2.6. Obsługa diod LED z wykorzystaniem podsystemu LED Class Driver

Ze względu na bardzo duże podobieństwo programowej obsługi sterownika LED Class Driver oraz /sys/class/gpio, dla podpunktu 2.6 nie przygotowano wyodrębnionego kodu źródłowego. Konfiguracja jądra, DT oraz obsługa sterownika w przestrzeni użytkownika (w tym konfiguracja tzw. wyzwalaczy) zostanie przedstawiona przez prowadzącego podczas sesji warsztatów.

2.7. Obsługa magistrali SPI na przykładzie aplikacji "loopback"

[1] Schemat połączeń sprzętowych:



[2] Kompilacja i uruchomienie programu testowego *loopback-spi.c*

```
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
```

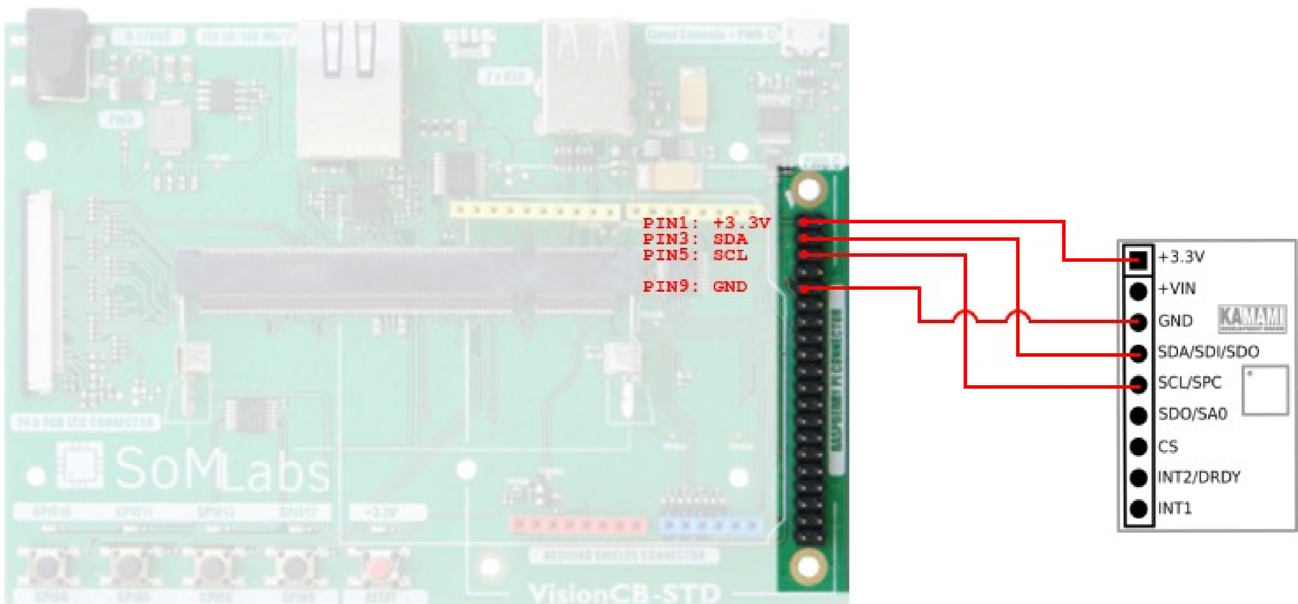
```
FF FF FF FF FF FF
40 00 00 00 00 95
FF FF FF FF FF FF
FF FF FF FF FF FF
FF FF FF FF FF FF
DE AD BE EF BA AD
F0 0D
```

W przypadku nieprawidłowego wykonania połączeń sprzętowych, program *loopback-spi*, zwróci wartości:

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

2.8. Magistrala I2C na przykładzie obsługi modułu żyroskopu

[1] Schemat połączeń sprzętowych:

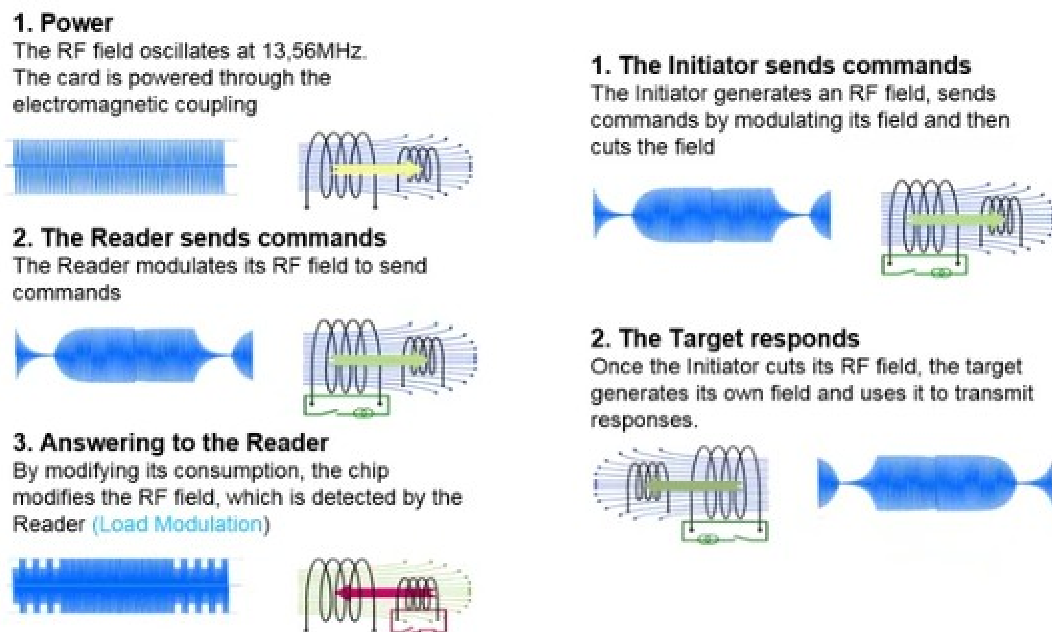


[2] Kompilacja i uruchomienie programu testowego *gyro-i2c.c*

```
root@localhost:~# cd /root/linux-academy/2-8
root@localhost:~/linux-academy/2-8# gcc gyro-i2c.c -o gyro-i2c
root@localhost:~/linux-academy/2-8# ./gyro-i2c
-8.1 23.6 -13.0
-8.1 24.3 -13.5
```

ĆWICZENIE 3

NFC (*Near-Field Communications*) to interfejs służący do bezprzewodowej wymiany danych i, w ograniczonym zakresie, do przesyłania energii. Wykorzystuje oddziaływanie magnetyczne między dwiema antenami, które, zbliżone do siebie, tworzą transformator.



Komunikacja z tagiem (po lewej) oraz tryb P2P.

W komunikacji NFC zawsze występują dwa urządzenia: *initiator* oraz *target*. *Initiator* musi zawsze być urządzeniem aktywnym, czyli musi być w stanie zasilić antenę. *Target* może być zarówno urządzeniem aktywnym, jak i pasywnym. Pasywny *target* jest zasilany prądem zindukowanym w antenie, i komunikuje się przez modulowanie (obciążanie) istniejącego pola. W zasadzie wszystkie tagi NFC są pasywnymi *target*-ami.

NDEF (*NFC Data Exchange Format*) to format danych używany do przesyłania wiadomości (*messages*) przez interfejs NFC. Większość urządzeń konsumenckich wykorzystujących NFC, w tym smartfony z systemem Android, wspiera format NDEF. Wiadomości NDEF składają się z rekordów (*records*). Zdefiniowane są standardowe typy rekordów dla różnego rodzaju danych, np.:

- tekst
- numer telefonu
- adres e-mail
- współrzędne geograficzne
- dane sieci WiFi (SSID, metoda zabezpieczenia, hasło)
- MIME (np. image/png)
- URI (np. ftp://ftp.slackware.com/pub/slackware/)

Możliwe jest zdefiniowanie nowych typów rekordów na potrzeby danej aplikacji, wprowadzając nowy schemat URI lub typ MIME.

Dla celów tego szkolenia zostały przygotowane trzy aplikacje, prezentujące różne zastosowania technologii NFC:

- odczytywanie tagów,
- komunikacja ze smartfonem (tryb P2P),
- wykorzystanie inteligentnych tagów (NTAG I^2C Plus).

3.1. Odczytywanie tagów

Aplikacja pozwala odczytywać rekordy tekstowe NDEF z kart MIFARE Ultralite, np. ntag216.

Aby skompilować i uruchomić aplikację, w konsoli Linuksa należy wpisać:

```
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.

3.2. Komunikacja ze smartfonem (P2P)

Tryb P2P pozwala dwóm urządzeniom aktywnym wymieniać wiadomości NDEF.

Ponownie, należy przejść do właściwego katalogu, skompilować i uruchomić aplikację:

```
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 World!"
```

Upewniamy się, że kontroler NFC w smartfonie jest włączony: *Ustawienia → Więcej → Komunikacja NFC*

Po zbliżeniu smartfona do anteny NFC, na ekranie powinien pokazać się komunikat o treści wpisanej jako argument podany przy uruchamianiu programu.

3.3. Inteligentny tag (NTAG I^2C Plus)

NTAG I^2C Plus to układ taga NFC z interfejsem I^2C. Dodanie interfejsu I^2C pozwala modyfikować ustawienia i zawartość pamięci przez mikrokontroler. Ponadto, tag może zasilać zewnętrzne układy i sygnalizować obecność sygnału NFC. Oprócz pamięci EEPROM, dysponuje 64 bajtami pamięci SRAM, dzięki czemu możliwa jest szybka, dwukierunkowa komunikacja z innymi urządzeniami.

Komendy do uruchomienia aplikacji:

```
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
```

Antenę demoboard-u umieszczamy nad anteną modułu NFC. Dioda RGB zacznie migać. Wciśnięcie przycisków oznaczonych kolorami będzie zasygnalizowane na standardowym wyjściu programu.

Jeżeli podczas wprowadzania anteny w pole NFC będzie wciśnięty środkowy przycisk, demoboard wejdzie w tryb programowania pamięci. Do pamięci EEPROM zostanie zaprogramowana domyślna wiadomość NDEF.

ĆWICZENIE 4

Time-to-market w systemach Linux Embedded, czyli wykorzystanie gotowych komponentów oprogramowania do sprawnej realizacji bardziej rozbudowanych projektów programowo-sprzętowych.

Dzisiejszy rynek elektroniki użytkowej wymaga od projektantów, by konstruowane przez nich urządzenia nie tylko były użyteczne, funkcjonalne i poprawnie zrealizowane. Przy dużej i wciąż rosnącej konkurencji na rynku elektroniki użytkowej, należy również zadbać, by urządzenie miało nowoczesny design a czas realizacji produktu był jak najkrótszy - od momentu złożenia zamówienia przez potencjalnego klienta do jego pełnej realizacji. Co zrobić, by nadążyć za dynamicznie rozwijającym się rynkiem? Korzystać z otwartych i gotowych pakietów oprogramowania.

Jedną z niewątpliwych zalet wykorzystania systemu operacyjnego Linux w procesie projektowania urządzeń wbudowanych jest szybki i łatwy dostęp do otwartych, darmowych i wolnych (w sensie wolności) repozytoriów oprogramowania implementujących m.in. rozbudowane stosy graficzne, protokoły sieciowe czy złożone algorytmy obliczeń. Fakt ten odgrywa szczególnie ważną rolę, kiedy zadanie stawiane przed naszym urządzeniem może zostać chociaż częściowo zrealizowane za pomocą istniejących już pakietów oprogramowania. Samodzielna implementacja stosów obsługi USB, Ethernet, Bluetooth czy bardziej złożonych interfejsów graficznych użytkownika może być czasochłonna i podatna na błędy. Wykorzystując gotowe sterowniki i pakiety oprogramowania (o ile są one udostępnione na dogodnej licencji), zyskujemy nie tylko czas, ale i pewność, że oprogramowanie zostało przetestowane przez tysiące innych użytkowników Linuksa.

W ćwiczeniu numer 4, przedstawiony zostanie przykład prostego i szybkiego tworzenia bardziej rozbudowanych projektów sprzętowo-programowych, z wykorzystaniem bibliotek gotowego i darmowego oprogramowania. Wykorzystując wyłącznie minimalną funkcjonalność środowiska uruchomieniowego *Node.js* oraz biblioteki *Three.js*, przygotujemy prosty serwer WWW, prezentujący wyniki danych pomiarów odczytanych z modułu żyroskopu w postaci animowanej kostki 3D. Zaczynajmy!

4.1. Node.js – systemy wbudowane i JavaScript?

Czym jest *Node.js*? Jest to wieloplatformowe środowisko uruchomieniowe JavaScript udostępnione na licencji *open-source*. Inaczej ujmując ogólnodostępną definicję – platforma *Node.js* umożliwia uruchomienie kodu JavaScript poza przeglądarką internetową. Należy podkreślić, że samo *Node.js* nie jest serwerem, umożliwia jednak proste i szybkie utworzenie serwera HTTP oraz bardziej złożonych aplikacji internetowych. Ponieważ kod programu jest uruchamiany poza przeglądarką, programista ma możliwość tworzenia typowych rozwiązań "*server-side*", czyli takich w których aplikacja może realizować odczyt/zapis systemu plików, baz danych oraz co równie istotne pod kątem realizacji urządzeń wbudowanych – wchodzić w interakcję z dołączonym do systemu urządzeniami peryferyjnymi – czujnikami, aktuatorami, itp.

Ze względu na dużą popularność platformy *Node.js* (warto nadmienić, że korzystają z niej takie serwisy jak *Netflix*, *PayPal*, *LinkedIn* czy *Uber*), gotowe pakiety oprogramowania są obecnie dostępne w niemal wszystkich dystrybucjach linuksowych. Dla dystrybucji *Debian*, instalacja pakietu `nodejs` przebiega w sposób standardowy dla narzędzia `apt-get`:

```
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) ...
```

Przygotowany na potrzeby niniejszych warsztatów obraz systemu linux-academy.img, posiada zainstalowany pakiet nodejs – zadaniem uczestnika warsztatów jest sprawdzenie poprawności jego instalacji poprzez wykonanie polecenia:

root@localhost:~# nodejs -v

Aby przetestować poprawność instalacji, wywołajmy komendę `nodejs -v`:

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

Jeśli w wyniku powyższego polecenia został wyświetlony numer wersji oprogramowania *Node.js*, możemy przystąpić do realizacji zadań z punktu 4.2 w którym przygotujemy najprostszą implementację serwera WWW.

4.2. Node.js – prosta implementacja serwera WWW

Kompletny kod źródłowy dla zagadnień omawianych w punkcie 4.2:

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

Implementację serwera WWW rozpoczynamy od utworzenia pliku *main.js*. W pierwszej linii kodu zaimportujmy wbudowany w *Node.js* moduł `http`:

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

oraz zdefiniujmy numer portu na którym będzie nasłuchiwał tworzony serwer:

```
var PORT = 8080;
```

Kolejnym krokiem będzie utworzenie właściwego serwera poprzez wywołanie metody `createServer()` na module `http`:

```
var server = http.createServer (/*…*/);
```

Metoda `createServer()` jako argument przyjmuje funkcje zwrotną, której zadaniem jest obsługa zapytań przychodzących do serwera. Funkcja ta przyjmuje dwa argumenty:

- `request` – argument zawiera informacje o szczegółach zapytania,
- `response` – obiekt zawierający metody i własności do obsługi odpowiedzi.

W naszej prostej implementacji serwera, w odpowiedzi na zapytanie klienta prześlemy: informację z kodem odpowiedzi (*200-OK*), typ zwracanego dokumentu (*text/plain* – czyste dane tekstowe) oraz napis *„Hello World!”*. Uzupełniony kod metody `createServer()` o obsługę zapytania został przedstawiony poniżej:

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

Ostatnim etapem, jest wywołanie metody `listen()` wraz z przekazaniem numeru portu, na którym serwer będzie nasłuchiwał nadchodzących połączeń:

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

Kompletna zawartość pliku main.js została przedstawiona na *Listingu 4.2.1*.

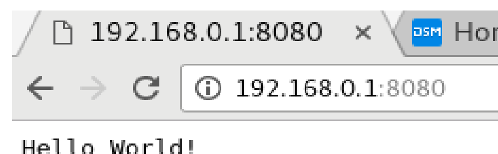
```
var http = require ('http');

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 4.2.1. *Skrypt main.js z implementacją prostego serwera WWW*

Po zapisaniu zmian w pliku *main.js*, możemy uruchomić skrypt za pomocą polecenia:

```
nodejs main.js
```

Jeżeli próba uruchomienia skryptu została zakończona sukcesem, wpisując w oknie przeglądarki adres *http://192.18.0.1:8080* zobaczymy rezultat działania serwera WWW – *Rysunek 4.2.1*.



Rys. 4.2.1. *Implementacja prostego serwera WWW*

4.3. Node.js – serwer WWW z podziałem na funkcje front-end/back-end

Kompletny kod źródłowy dla zagadnień omawianych w punkcie 4.3:

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

Bezpośrednie umieszczenie „kodu strony” - w postaci napisu *„Hello World!”* - w funkcji `response.end()` nie wpływa znacząco na czytelność kodu, jednak nietrudno wyobrazić sobie sytuację, że budowany przez nas serwis zaczyna się rozrastać, a wprowadzane znaczniki HTML znacznie zwiększają objętość kodu strony, powodując że skrypt *main.js* może stać się mało czytelny i trudny w zarządzaniu. W takiej sytuacji niezbędne jest wprowadzenie jasnego podziału na *front-end* (czyli właściwą stronę udostępnianą użytkownikowi) oraz *back-end* (czyli kod realizujący zadania stawiane przed serwerem). Wprowadzenie podziału na *front-end* i *back-end* wymaga jedynie kosmetycznych zmian w skrypcie *main.js* z punktu 4.2. Zmodyfikowany skrypt *main.js* przedstawiono na *Listingu 4.3.1*.

```
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);
});

server.listen (PORT);
```

Listing 4.3.1. *Serwer WWW z podziałem na funkcje fron-end/back-end*

Z wykorzystaniem wbudowanego modułu `fs` (pozwalającego na przeprowadzanie szeregu operacji I/O na plikach) w sposób synchroniczny wczytujemy zawartość pliku *index.html*, umieszczonego w tym samym folderze jak skrypt *main.js*. Ponieważ wczytany plik jest prostą stroną HTML, zmieniamy zawartość pola `Content-Type` na `text/html`. W wywołaniu `response.end()` przesyłamy użytkownikowi zawartość pliku *index.html*.

Dla kompletności zadania, utwórzmy również prosty plik HTML – *index.html*:

```
<!DOCTYPE html>
<html>

  <head>
  </head>

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

Po zakończonej edycji plików *main.js* oraz *index.html*, sprawdźmy poprawność naszego kodu poprzez ponowne uruchomienie serwera:

```
nodejs main.js
```

4.4. Node.js – komunikacja front-end<->back-end z wykorzystaniem socket.io

Kompletny kod źródłowy dla zagadnień omawianych w punkcie 4.4:

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

Wprowadzenie wyraźnego podziału na sekcje *front-end* i *back-end* stawia przed nami kolejne zadanie do wykonania – zapewnienie sprawnej komunikacji i wymiany danych w „czasie rzeczywistym" pomiędzy tymi modułami. Dlaczego w czasie rzeczywistym? Protokół HTTP jest typowym protokołem typu żądanie-odpowiedź, w którym to rolę żądającego pełni klient/przeglądarka internetowa. Rozwiązanie to spełnia swoje zadanie w przypadku gdy to klient chce przesłać dane do serwera. Niestety w sytuacji gdy serwer chce poinformować odbiorcę o aktualizacji danych (np. nowych odczytach z czujników temperatury, których wyniki powinny być wyświetlane w czasie rzeczywistym w interfejsie przeglądarkowym), nie może on zainicjować połączenia z klientem. Modyfikacja „w locie" pliku *index.html* przez kod serwera oraz cykliczne odświeżanie strony przez klienta nie brzmią jak idealne rozwiązanie problemu. W takiej sytuacji pomocną dłoń wyciąga do nas biblioteka *socket.io* zapewniająca połączenie pomiędzy stroną WWW (front-endem) a skryptem uruchomionym na serwerze (back-endem). *Socket.io* jest biblioteką języka JavaScript, której zadaniem jest ułatwienie pracy z protokołem *WebSocket* (który to natomiast jest częścią specyfikacji HTML5, umożliwiającą dwustronną komunikację klient-serwer w czasie rzeczywistym). Biblioteka *socket.io* składa się z tzw. części serwerowej (będącej modułem dla platformy *Node.js*) oraz klienckiej (dla przeglądarek internetowych). Bazując na kodzie skryptu *main.js* oraz strony *index.html* z poprzedniego podpunktu, przejdźmy do praktycznej implementacji.

Rozbudowę skryptu *main.js* rozpoczynamy od zaimportowania modułu *socket.io* (szczegóły dotyczące instalacji zewnętrznego pakietu *socket.io* zostały przedstawione w ramce poniżej):

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

Pakiet socket.io nie jest częścią platformy Node.js i wymaga dodatkowej instalacji. Do instalacji dodatkowych pakietów można wykorzystać dystrybuowany wraz z Node.js, manager pakietów npm:

npm install socket.io

W domyślnym obrazie, pakiet socket.io jest dystrybuowany wraz z kodem źródłowym poszczególnych ćwiczeń, stąd uczestnicy warsztatów nie muszą dokonywać samodzielnej instalacji poszczególnych pakietów.

W następnym kroku utwórzmy *event-handler* dla zdarzenia `connection` (które jest wywoływane każdorazowo, gdy do serwera podłączony zostanie nowy klient), wyświetlający krótki komunikat na standardowym wyjściu:

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

W docelowym rozwiązaniu realizowanym w ramach ćwiczenia numer 4, aplikacja serwera będzie przesyłała do przeglądarki użytkownika informacje odczytane z modułu żyroskopu. Sposób w jaki zostanie zrealizowana komunikacja pomiędzy aplikacją *gyro-i2c* (z ćwiczenia numer 2) a serwerem WWW, zostanie omówiony w kolejnym podpunkcie niniejszego ćwiczenia. Na potrzeby obecnego zadania, przygotujmy prostą funkcję `send_time()`, która z interwałem jednej sekundy, prześle do wszystkich podłączonych klientów aktualny czas:

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

W ciele funkcji `send_time()` wysyłamy rozgłoszeniową wiadomość `time` z aktualnym czasem serwera, skierowaną do wszystkich aktualnie podłączonych klientów. Pełny kod skryptu *main.js* wraz z wyszczególnieniem wprowadzonych zmian (w odniesieniu do podpunktu 4.3) został przedstawiony na *Listingu 4.4.1*.

```
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 4.4.1. *Skrypt main.js zintegrowany z bilioteką socket.io*

Ostatnim etapem zadania z podpunktu 4.3 jest integracja biblioteki *socket.io* z udostępnianą przez serwer stroną *index.html*. Integrację biblioteki rozpoczniemy od dołączenia w sekcji `<head>` biblioteki *socket.io*:

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

Również w sekcji `<head>` utwórzmy prosty skrypt realizujący nawiązanie połączenia z serwerem oraz odbiór komunikatów (należy pamiętać, że kod zawarty w tagach `<script></script>` zostanie uruchomiony przez przeglądarkę, a więc komputer PC użytkownika):

```
var socket = io();

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

Zanim przystąpimy do uzupełnienia kodu *event-handler'a* dla zdefiniowanego przez nas zdarzenia `time`, w sekcji `<body>` strony HTML utwórzmy akapit z identyfikatorem `test`, w miejscu którego wyświetlone zostaną dane otrzymane z serwera WWW:

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

Mając określone pole w który otrzymane dane będą wyświetlane, możemy uzupełnić implementację *event-handler'a* dla zdarzenia `time`:

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

Pełna zawartość pliku *index.html* (wraz z wyróżnieniem zmian wprowadzonych w stosunku do podpunktu 4.3) została przedstawiona na *Listingu 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. *Strona index.html zintegrowana z biblioteką socket.io*

Przy ponownym uruchomieniu serwera poleceniem:

```
nodejs main.js
```

oraz odświeżeniu zawartości adresu *http://192.168.0.1:8080*, powinniśmy uzyskać efekt przedstawiony na *Rysunku 4.4.1*.

Rys. 4.4.1. *Przykład komunikacji między serwer WWW a przeglądarką internetową*

4.5. Node.js – serwer WWW z odczytem danych z modułu żyroskopu

Kompletny kod źródłowy dla zagadnień omawianych w punkcie 4.5:

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

Na obecnym etapie realizacji ćwiczenia wiemy już jak nawiązać prostą komunikację pomiędzy serwerem a klientem. Do pełnej realizacji celu postawionego przez ćwiczeniem numer 4, brakuje wciąż informacji w jaki sposób „poinformować” serwer o aktualnych danych pomiarowych, zwracanych przez moduł żyroskopu. Do najprostszych rozwiązań tego problemu możemy zaliczyć np. bezpośrednią implementacją obsługi żyroskopu w kodzie serwera – z wykorzystaniem operacji na plikach lub gotowych modułów *Node.js,* instalowanych poprzez menadżer pakietów *npm*. Przykładem takiego modułu może być pakiet `i2c`, instalowany poleceniem:

```
npm install i2c
```

który udostępnia proste API do realizacji niskopoziomowych operacji zapisu/odczytu danych na magistrali, np.:

```
var i2c = require('i2c');
```

```
var wire = new i2c(address, {device: '/dev/i2c-1'});

wire.writeByte(byte, function(err) {});

wire.writeBytes(command, [byte0, byte1], function(err) {});

wire.readByte(function(err, res) { // result is single byte })

wire.readBytes(command, length, function(err, res) {});
```

Pomimo tego, że API modułu `i2c` jest bardzo zbliżone do rozwiązań wykorzystanych przy implementacji obsługi żyroskopu w języku C a sama reimplementacja kodu `gyro-i2c` (z ćwiczenia numer 2) na kod JavaScript nie powinna być zbyt czasochłonna, do realizacji kolejnego zadania użyjemy alternatywnego podejścia. Aby uniknąć ponownego przygotowywania obsługi żyroskopu, do realizacji zadania wykorzystamy gotowy plik binarny `gyro-i2c` oraz wbudowany w *Node.js* moduł `child_process`. Za pomocą metody `spawn()` utworzymy nowy proces potomny oraz zdefiniujemy dla niego funkcje zwrotną obsługi standardowego wyjścia (wywoływaną w chwili gdy program `gyro-i2c` zwróci kolejną porcję danych z wynikami pomiarów).

Analogicznie jak w poprzednich podpunktach, do realizacji tego etapu wykorzystamy pliki z etapu 4.4. Edycję rozpoczniemy od skryptu *main.js* w którym zaimportujemy wbudowany moduł `child_process`:

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

W kolejnym kroku, za pomocą wywołania `spawn()` utworzymy nowy proces potomny realizujący kod programu `gyro-i2c` (skopiowany uprzednio do folderu */tmp)*:

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

Ostatnią modyfikacją w skrypcie *main.js* jest dodanie funkcji zwrotnych do obsługi kanałów `stdout` (funkcja przesyła odczytane dane do przeglądarki w postaci komunikatu `xyz`) oraz `stderr` (funkcja wypisuje w konsoli dane odczytane ze standardowego strumienia błędów):

```
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);
});
```

Warto również zaimplementować obsługę zdarzenia `close`, która poinformuje nas o zakończeniu procesu potomnego i zwróconym przez niego kodzie wyjścia:

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

Pełna zawartość pliku *main.js* (wraz z wyróżnieniem zmian wprowadzonych w stosunku do podpunktu 4.4) została przedstawiona na *Listingu 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. *Skrypt main.js z zaimplementowaną obsługą procesu potomnego*

Przystosujmy również plik *index.html* do nowych wymagań projektu, tj. wyświetlenia wartości trzech pomiarów dla osi X, Y oraz Z. W tym celu w sekcji `<body>` utwórzmy prostą tabelę zawierającą identyfikatory pól `x_val`, `y_val` oraz `z_val`:

```
<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>
```

W sekcji `<head>` zmodyfikujmy kod obsługi wiadomości `xyz`. Odczytana linia danych zostanie podzielona względem separatora ‘ ‘ (spacja), a wyniki pomiarów przypisane do poszczególnych identyfikatorów pól:

```
<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>
```

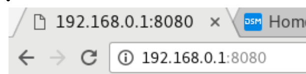
Dla poprawienia estetyki utworzonej strony w sekcji head dodano wpis formatujący wygląd tabeli. Pełna zawartość pliku *index.html* (wraz z wyróżnieniem zmian wprowadzonych w stosunku do podpunktu 4.4) została przedstawiona na *Listingu 4.5.2*.

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

Listing 4.5.2. *Skrypt main.js z zaimplementowaną obsługą procesu potomnego*

Przy ponownym uruchomieniu serwera poleceniem `nodejs main.js` oraz odświeżeniu zawartości adresu *http://192.168.0.1:8080*, powinniśmy uzyskać efekt przedstawiony na *Rysunku 4.5.1*.



Gyroscope I2C

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

Rys. 4.5.1. *Prezentacja wyników pomiarów w oknie przeglądarki internetowej*

4.6. Node.js – rozbudowa interfejsu o proste elementy grafiki 3D (Three.js)

Kompletny kod źródłowy dla zagadnień omawianych w punkcie 4.6:

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

W ostatnim podpunkcie ćwiczenia numer 4 rozbudujemy interfejs graficzny aplikacji o prostą grafikę 3D w postaci sześciennej kostki, odwzorowującej ruch podłączonego modułu żyroskopu. Do realizacji operacji graficznych wykorzystamy bibliotekę *Three.js*, która to natomiast korzysta z API *WebGL* – oficjalnego rozszerzenia możliwości języka JavaScript o interfejs grafiki 3D. Bezpośrednie wykorzystanie interfejsu *WebGL* jest dość uciążliwe, choćby ze względu na dużą liczbę operacji niskiego poziomu, jakie spoczywają na programiście – definicja wierzchołków, buforów, macierzy transformacji, operacje związane z wyświetlaniem sceny, obsługa *shaderów*, oświetlenia, modeli, kamer i wiele innych. W bibliotece *Three.js* scena budowana jest z obiektów (sama scena jest również obiektem w którym umieszczamy inne obiekty). Do podstawowych obiektów możemy zaliczyć: figury geometryczne (biblioteka posiada zdefiniowane kilka gotowych do użycia obiektów takich jak sfera czy sześcian), materiały przypisywane do figur geometrycznym (określające m.in. ich kolor i fizykę odbijania światła), źródła światła oraz obserwatora sceny (czyli „kamerę”, która obserwuje scenę w określonym położeniu). Choć przedstawione w tym akapicie definicje mogę wydawać się na tym etapie mało nieczytelne, przystąpmy do praktycznej realizacji zadania, która powinna wyjaśnić wprowadzone pojęcia.

Ponieważ kod odpowiedzialny za animację 3D jest wykonywany przez przeglądarkę po stronie komputera użytkownika, należy upewnić się, że wybrana przez nas przeglądarka internetowa wspiera API WebGL v1:

http://webglreport.com/

Rozbudowę aplikacji rozpoczynamy od pobrania kodu biblioteki *Three.js* (plik *three.min.js*) do katalogu w którym umieszczono skrypt *main.js* oraz stronę *index.html*:

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

W trakcie trwania warsztatów nie ma potrzeby samodzielnego pobierania kodu biblioteki Three.js – wszystkie niezbędne pliki do wykonania tego etapu zostały umieszczone w katalogu /root/linux-academy/4-6.

Edycję pliku *index.html* rozpoczynamy od zdefiniowania w sekcji `<head>` „płótna” `canvas` (o wymiarach 500x500px oraz identyfikatorze `mycanvas`) w którym będzie renderowana docelowa animacja:

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

Następnie w sekcji `<head>` dołączamy bibliotekę *Three.js*:

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

W dalszej części skryptu definiujemy zmienne w których będziemy przechowywać wyniki pomiarów w osi x, y, z oraz informacje o tworzonej scenie i dołączonych do niej obiektach:

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

Następnie implementujemy funkcję `init()`, której zadaniem jest zbudowanie sceny z określonych obiektów:

```
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);
}
```

W pierwszej linii kodu funkcji `init()` tworzymy scenę, do której będziemy dołączali kolejno definiowane obiekty (kamerę, figurę geometryczną oraz materiał dla tej figury):

```
scene = new THREE.Scene();
```

W następnym kroku tworzymy obiekt kamery określając kąt jej widzenia (70 stopni), proporcje kadru, zakresy widzenia: bliski i daleki, a także jej umiejscowienie:

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

Korzystając ze zdefiniowanych w bibliotece *Three.js* kształtów, tworzymy obiekt reprezentujący sześcian (`BoxGeometry`):

```
geometry = new THREE.BoxGeometry (0.2, 0.2, 0.2);
```

oraz obiekt stanowiący „materiał” z jakiego wykony jest nasz sześcian (decyduje on m.in. o kolorze obiektu i sposobie rozpraszania światła) – wykorzystamy tutaj predefiniowany materiał `MeshNormalMaterial`:

```
material = new THREE.MeshNormalMaterial();
```

Z połączenia figury z materiałem możemy utworzyć obiekt klasy `Mesh`, który dodajemy do tworzonej sceny:

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

W ostatnich liniach funkcji `init()`, określamy rozmiar i identyfikator powierzchni (`mycanvas`) na której będzie renderowana animacja:

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

Utwórzmy również funkcję animate(), która dokona obrotu obiektu, zgodnie z kątem obrotu zapisanym w zmiennych x, y, z:

```
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);
}
```

Pełna zawartość pliku *index.html* (wraz z wyróżnieniem zmian wprowadzonych w stosunku do podpunktu 4.5) została przedstawiona na *Listingu 4.6.1*.

```
<!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 4.6.1. *Plik index.html z wbudowaną animacją 3D*

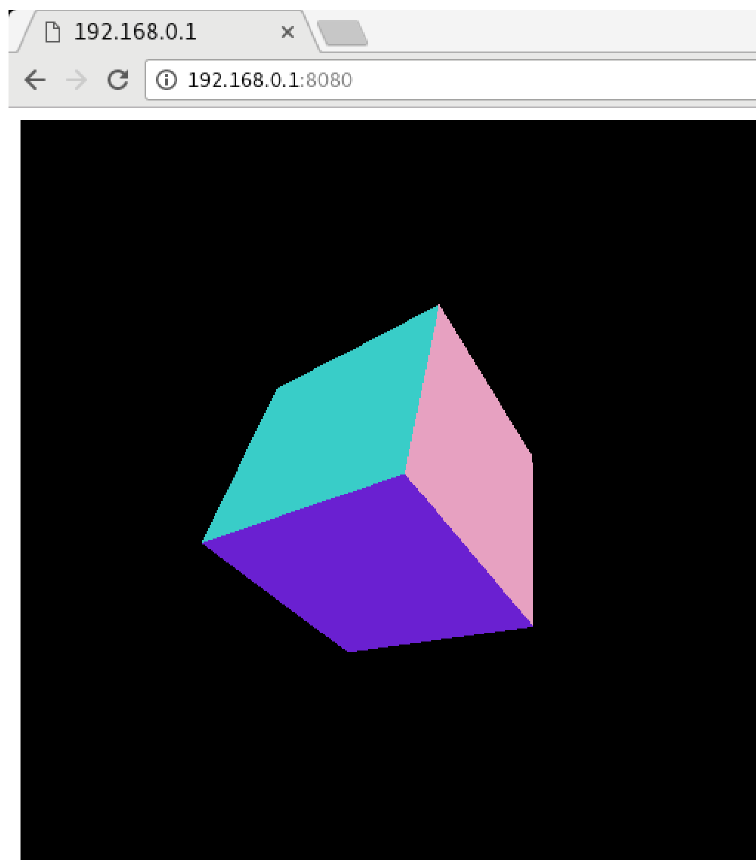
Niewielkiej modyfikacji wymaga również sam kod serwera *main.js*. Dotychczas serwer na żądnie klienta udostępniał wyłącznie plik *index.html*. W aktualnie formie, przy ładowaniu strony głównej, klient zażąda również pliku *three.min.js* – serwer powinien to żądanie obsłużyć i dostarczyć klientowi wymaganą bibliotekę.

```
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();
});
```

Przy ponownym uruchomieniu serwera poleceniem:

```
nodejs main.js
```

oraz odświeżeniu zawartości adresu *http://192.168.0.1:8080*, powinniśmy uzyskać efekt przedstawiony na *Rysunku 4.6.1*.



Gyroscope I2C

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

Rys. 4.6.1. *Prezentacja danych odczytanych z żyroskopu w postaci animacji 3D*