ĆWICZENIE 2.1:

Interfejsy komunikacyjne w przykładach - sterowanie GPIO poprzez interfejs /sys/class/gpio

Interfejs /sys/class/gpio - podstawy

- W urządzeniach wbudowanych pracujących pod kontrolą systemu operacyjnego Linux, bezpośredni dostęp do układów peryferyjnych ma wyłącznie jądro systemu,

Interfejs /sys/class/gpio - podstawy

- 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,

Interfejs /sys/class/gpio - podstawy

- 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,
- Na potrzeby portów GPIO, jądro systemu udostępnia:
    - sterownik kontroli "wyjść" cyfrowych (podsystem `Led Class Driver`),
    - sterownik kontroli "wejść" cyfrowych (podsystem `GPIO Buttons`),
    - generyczny sterownik wejść/wyjść (`sysfs gpio interface`),

Interfejs /sys/class/gpio - podstawy

- 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
```

Interfejs /sys/class/gpio - podstawy

- 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
```

- Wyeksportowanie wybranego wyprowadzenia odbywa się poprzez zapis jego numeru porządkowego do pliku *export*, np..:

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

Interfejs /sys/class/gpio - podstawy

- 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
```

- Wyeksportowanie wybranego wyprowadzenia odbywa się poprzez zapis jego numeru porządkowego do pliku *export*, np..:

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

Interfejs /sys/class/gpio - podstawy

<GPIO number> = (<GPIO bank> – 1) * 32 + <GPIO bit>

Interfejs /sys/class/gpio - podstawy

<GPIO number> = (<GPIO bank> – 1) * 32 + <GPIO bit>

- **GPIO1_IO17** -> (1-1) * 32 + 17 = 17

Interfejs /sys/class/gpio - podstawy

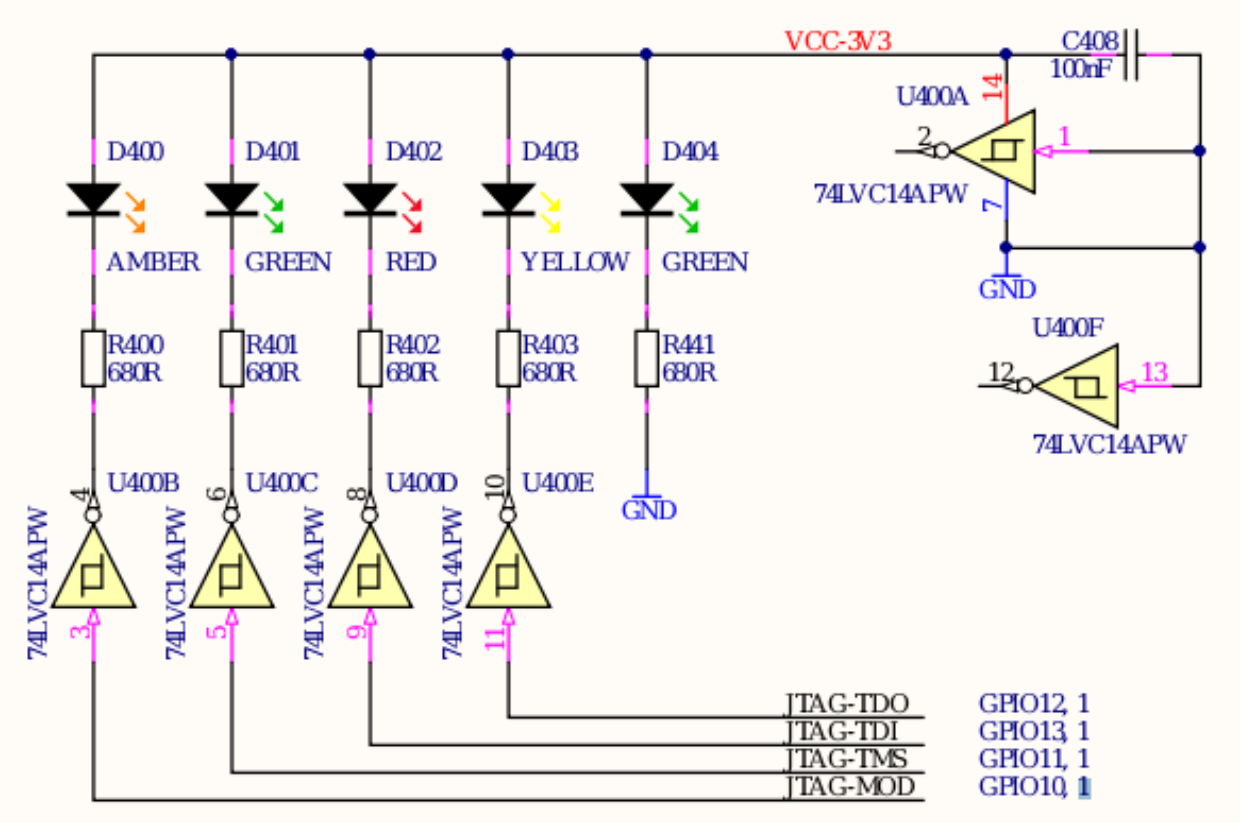
<GPIO number> = (<GPIO bank> – 1) * 32 + <GPIO bit>

- **GPIO1_IO17** -> (1-1) * 32 + 17 = 17
- **GPIO4_IO24** -> (4-1) * 32 + 24 = 120

Interfejs /sys/class/gpio - podstawy

<GPIO number> = (<GPIO bank> – 1) * 32 + <GPIO bit>

- **GPIO1_IO17** -> (1-1) * 32 + 17 = 17
- **GPIO4_IO24** -> (4-1) * 32 + 24 = 120
- **GPIO1_IO10** -> (1-1) * 32 + 17 = 10



Interfejs /sys/class/gpio - podstawy

- Każda linia GPIO wyeksportowana do przestrzeni użytkownika reprezentowana jest przez katalog */sys/class/gpio/gpioX*, gdzie X określa numer porządkowy portu:

```
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 */sys/class/gpio/gpioX*,

Interfejs /sys/class/gpio - podstawy

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
```

Interfejs /sys/class/gpio - podstawy

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
```

Interfejs /sys/class/gpio - podstawy

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
```

- wyzwolenie zboczem narastającym dla wejścia *gpioX*:

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

ĆWICZENIE 2.2:

Sterowanie GPIO poprzez interfejs /sys/class/gpio - "Hello World" [skrypt powłoki]

"Hello World" - sterowanie diodą LED [skrypt]

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***.

ĆWICZENIE 2.3:

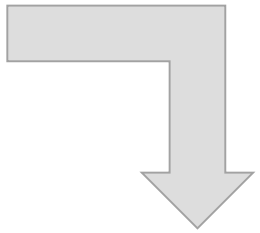
Sterowanie GPIO poprzez interfejs /sys/class/gpio - "Hello World" [Język C]

"Hello World" - sterowanie diodą LED [Język C]

- `open();`
- `write();`
- `read();`
- `close();`

"Hello World" - sterowanie diodą LED [Język C]

- `open();`
- `write();`
- `read();`
- `close();`

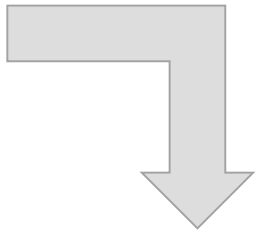


- **Eksport wyprowadzenia GPIO**
- Zmiana kierunku pracy GPIO
- Zmiana stanu na wyjściu GPIO

```
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;
}
```

"Hello World" - sterowanie diodą LED [Język C]

- `open();`
- `write();`
- `read();`
- `close();`

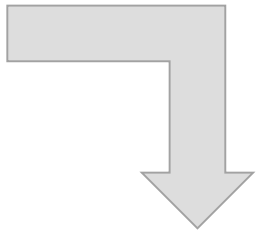


- Eksport wyprowadzenia GPIO
- **Zmiana kierunku pracy GPIO**
- Zmiana stanu na wyjściu GPIO

```
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;
}
```

"Hello World" - sterowanie diodą LED [Język C]

- `open();`
- `write();`
- `read();`
- `close();`

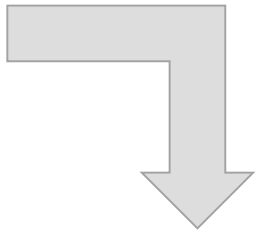


- Eksport wyprowadzenia GPIO
- Zmiana kierunku pracy GPIO
- **Zmiana stanu na wyjściu GPIO**

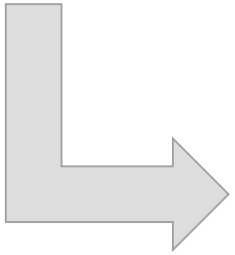
```
static int
gpio_set_value (unsigned int gpio,
                unsigned int value)
{
  int fd;
  char buf[BUF_SIZE];

  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;
}
```

"Hello World" - sterowanie diodą LED [Język C]

- `open();`
- `write();`
- `read();`
- `close();`



- Eksport wyprowadzenia GPIO
- Zmiana kierunku pracy GPIO
- Zmiana stanu na wyjściu GPIO



```
#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);

  while (1)
    {
      gpio_set_value (GPIO_PIN, 1);
      sleep (1);

      gpio_set_value (GPIO_PIN, 0);
      sleep (1);
    }
  return EXIT_SUCCESS;
}
```

ĆWICZENIE 2.4:

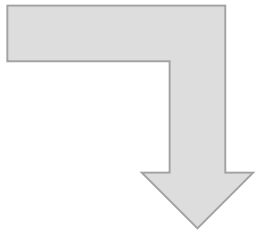
Sterowanie GPIO poprzez interfejs /sys/class/gpio - obsługa przycisku

Prosta obsługa przycisku z funkcją poll()

Bazując na kodzie z podpunktu 2.3, zaimplementujemy przerwaniową obsługę przycisku podłączonego do wyprowadzenia ***GPIO1_0***.

Prosta obsługa przycisku z funkcją poll()

- `open();`
- `write();`
- `read();`
- `close();`

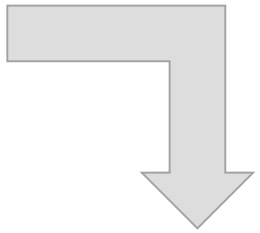


- Eksport wyprowadzenia GPIO
- Zmiana kierunku pracy GPIO
- Zmiana stanu na wyjściu GPIO
- **Konfiguracja zbocza**
- Pobranie deskryptora pliku

```
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;
}
```

Prosta obsługa przycisku z funkcją poll()

- `open();`
- `write();`
- `read();`
- `close();`

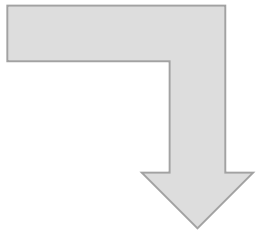


- Eksport wyprowadzenia GPIO
- Zmiana kierunku pracy GPIO
- Zmiana stanu na wyjściu GPIO
- Konfiguracja zbocza
- **Pobranie deskryptora pliku**

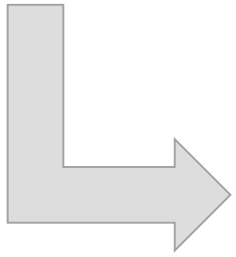
```
static int
gpio_fd_open (unsigned int gpio)
{
  int fd;
  char buf[BUF_SIZE];

  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;
}
```

Prosta obsługa przycisku z funkcją poll()

- `open();`
- `write();`
- `read();`
- `close();`



- Eksport wyprowadzenia GPIO
- Zmiana kierunku pracy GPIO
- Zmiana stanu na wyjściu GPIO
- Konfiguracja zbocza
- Pobranie deskryptora pliku



```
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)
    {
      //TODO
    }
}
```

Prosta obsługa przycisku z funkcją poll()

- `open();`
- `write();`
- `read();`
- `close();`

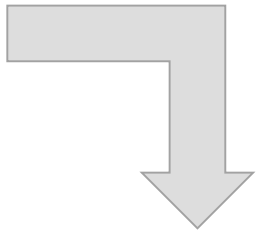
- Eksport wyprowadzenia GPIO
- Zmiana kierunku pracy GPIO
- Zmiana stanu na wyjściu GPIO
- Konfiguracja zbocza
- Pobranie deskryptora pliku

```
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)
    {
      //TODO
    }
}
```

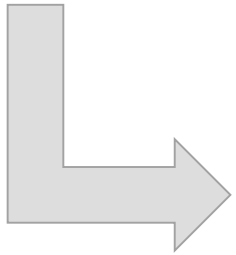
```
struct pollfd
{
  int fd;
  short events;
  short revents;
}
```

Prosta obsługa przycisku z funkcją poll()

- `open();`
- `write();`
- `read();`
- `close();`



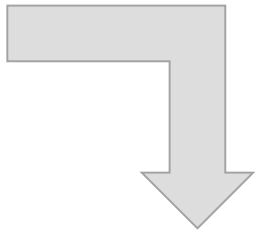
- Eksport wyprowadzenia GPIO
- Zmiana kierunku pracy GPIO
- Zmiana stanu na wyjściu GPIO
- Konfiguracja zbocza
- Pobranie deskryptora pliku



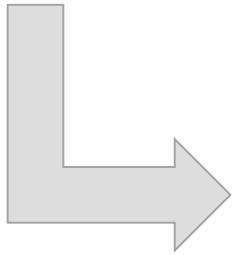
```
int main (void)
{
  struct pollfd fdset[1];
  int nfds = 1, fd, ret;

  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)
    {
      //TODO
    }
}
```

Prosta obsługa przycisku z funkcją poll()

- `open();`
- `write();`
- `read();`
- `close();`



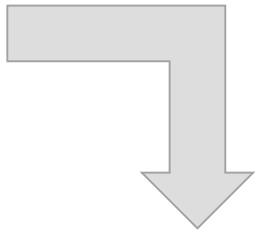
- Eksport wyprowadzenia GPIO
- Zmiana kierunku pracy GPIO
- Zmiana stanu na wyjściu GPIO
- Konfiguracja zbocza
- Pobranie deskryptora pliku



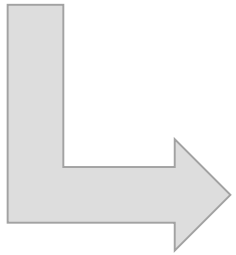
```
int main (void)
{
  struct pollfd fdset[1];
  int nfds = 1, fd, ret;

  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)
    {
      //TODO
    }
}
```

Prosta obsługa przycisku z funkcją poll()

- `open();`
- `write();`
- `read();`
- `close();`



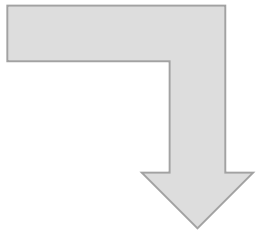
- Eksport wyprowadzenia GPIO
- Zmiana kierunku pracy GPIO
- Zmiana stanu na wyjściu GPIO
- Konfiguracja zbocza
- Pobranie deskryptora pliku



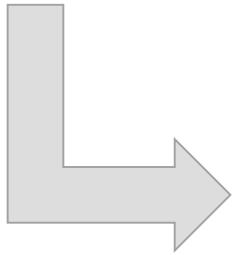
```
int main (void)
{
  struct pollfd fdset[1];
  int nfds = 1, fd, ret;

  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)
    {
      //TODO
    }
}
```

Prosta obsługa przycisku z funkcją poll()

- `open();`
- `write();`
- `read();`
- `close();`



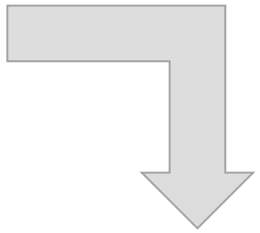
- Eksport wyprowadzenia GPIO
- Zmiana kierunku pracy GPIO
- Zmiana stanu na wyjściu GPIO
- Konfiguracja zbocza
- Pobranie deskryptora pliku



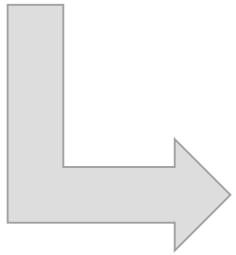
```
while (1)
  {
    memset (fdset, 0, sizeof(fdset));

    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);
  }
```

Prosta obsługa przycisku z funkcją poll()

- `open();`
- `write();`
- `read();`
- `close();`



- Eksport wyprowadzenia GPIO
- Zmiana kierunku pracy GPIO
- Zmiana stanu na wyjściu GPIO
- Konfiguracja zbocza
- Pobranie deskryptora pliku



```
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);
  }
```

ĆWICZENIE 2.5:

Obsługa przycisku z wykorzystaniem podsystemu GPIO Buttons

Obsługa przycisku – podsystem GPIO Buttons

- 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
```

Obsługa przycisku – podsystem GPIO Buttons

- 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
```

???

Kompilacja i konfiguracja jądra systemu "w pigułce"

- Domyślna konfiguracja jądra systemu:

```
cd /root/kernel/
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- visionsom-6ull-linux_defconfig
```

- Konfiguracja jądra z wykorzystaniem "graficznego" interfejsu:

```
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- menuconfig
```

- Właściwa kompilacja jądra systemu:

```
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- zImage
```

Opis konfiguracji sprzętowej z Device Tree

- Device Tree to pliki tekstowe opisujące urządzenie i konfigurację podłączonego do niego sprzętu

Opis konfiguracji sprzętowej z Device Tree

- Device Tree to pliki tekstowe opisujące urządzenie i konfigurację podłączonego do niego sprzętu
- Dla architektury ARM, pliki Device Tree umieszczone są w katalogu *arch/arm/boot/dts*

Opis konfiguracji sprzętowej z Device Tree

- Device Tree to pliki tekstowe opisujące urządzenie i konfigurację podłączonego do niego sprzętu
- Dla architektury ARM, pliki Device Tree umieszczone są w katalogu *arch/arm/boot/dts*
- Pliki Device Tree możemy podzielić na:
  - *.dtsi - opisujące konkretny model SoC-a,
  - *.dts - plik opisujący konkretną płytkę / urządzenie,
  - *.dtb - skompilowany plik Device Tree

Opis konfiguracji sprzętowej z Device Tree

- Device Tree to pliki tekstowe opisujące urządzenie i konfigurację podłączonego do niego sprzętu
- Dla architektury ARM, pliki Device Tree umieszczone są w katalogu *arch/arm/boot/dts*
- Pliki Device Tree możemy podzielić na:
  - *.dtsi - opisujące konkretny model SoC-a,
  - *.dts - plik opisujący konkretną płytkę / urządzenie,
  - *.dtb - skompilowany plik Device Tree
- Kompilacja Device Tree:

```
make ARCH=arm somlabs-visionsom-6ull.dtb
make ARCH=arm dtbs
```

Opis konfiguracji sprzętowej z Device Tree

- W Device Tree umieszczamy opis w formacie wymaganym przez sterownik danego urządzenia:

Opis konfiguracji sprzętowej z Device Tree

- W Device Tree umieszczamy opis w formacie wymaganym przez sterownik danego urządzenia:

https://www.kernel.org/doc/Documentation/devicetree/bindings/

Opis konfiguracji sprzętowej z Device Tree

- W Device Tree umieszczamy opis w formacie wymaganym przez sterownik danego urządzenia:

https://www.kernel.org/doc/Documentation/devicetree/bindings/

```
gpio-keys {

  compatible = "gpio-keys";


  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> */
  };
};
```

Opis konfiguracji sprzętowej z Device Tree

- W Device Tree umieszczamy opis w formacie wymaganym przez sterownik danego urządzenia:

https://www.kernel.org/doc/Documentation/devicetree/bindings/

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

Opis konfiguracji sprzętowej z Device Tree

- W Device Tree umieszczamy opis w formacie wymaganym przez sterownik danego urządzenia:

https://www.kernel.org/doc/Documentation/devicetree/bindings/

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

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

    pinctrl_gpio_keys: gpio-keys {
      fsl,pins = <
        MX6UL_PAD_GPIO1_IO08__GPIO1_IO08 0x1b0b0
        MX6UL_PAD_GPIO1_IO09__GPIO1_IO09 0x1b0b0
      >;
    };
  };
};
```

Opis konfiguracji sprzętowej z Device Tree

- W Device Tree umieszczamy opis w formacie wymaganym przez sterownik danego urządzenia:

https://www.kernel.org/doc/Documentation/devicetree/bindings/

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

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

    pinctrl_gpio_keys: gpio-keys {
      fsl,pins = <
        MX6UL_PAD_GPIO1_IO08__GPIO1_IO08 0x1b0b0
        MX6UL_PAD_GPIO1_IO09__GPIO1_IO09 0x1b0b0
      >;
    };
  };
};
```

Opis konfiguracji sprzętowej z Device Tree

- W Device Tree umieszczamy opis w formacie wymaganym przez sterownik danego urządzenia:

https://www.kernel.org/doc/Documentation/devicetree/bindings/

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

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

    pinctrl_gpio_keys: gpio-keys {
      fsl,pins = <
        MX6UL_PAD_GPIO1_IO08__GPIO1_IO08 0x1b0b0
        MX6UL_PAD_GPIO1_IO09__GPIO1_IO09 0x1b0b0
      >;
    };
  };
};
```

Opis konfiguracji sprzętowej z Device Tree

```
PAD_CTL_HYS                     (1 << 16)
PAD_CTL_PUS_100K_DOWN           (0 << 14)
PAD_CTL_PUS_47K_UP              (1 << 14)
PAD_CTL_PUS_100K_UP             (2 << 14)
PAD_CTL_PUS_22K_UP              (3 << 14)
PAD_CTL_PUE                     (1 << 13)
PAD_CTL_PKE                     (1 << 12)
PAD_CTL_ODE                     (1 << 11)
PAD_CTL_SPEED_LOW               (1 << 6)
PAD_CTL_SPEED_MED               (2 << 6)
PAD_CTL_SPEED_HIGH              (3 << 6)
PAD_CTL_DSE_DISABLE             (0 << 3)
PAD_CTL_DSE_240ohm              (1 << 3)
PAD_CTL_DSE_120ohm              (2 << 3)
PAD_CTL_DSE_80ohm               (3 << 3)
PAD_CTL_DSE_60ohm               (4 << 3)
PAD_CTL_DSE_48ohm               (5 << 3)
PAD_CTL_DSE_40ohm               (6 << 3)
PAD_CTL_DSE_34ohm               (7 << 3)
PAD_CTL_SRE_FAST                (1 << 0)
PAD_CTL_SRE_SLOW                (0 << 0)
```

Opis konfiguracji sprzętowej z Device Tree

```
PAD_CTL_HYS                     (1 << 16)
PAD_CTL_PUS_100K_DOWN           (0 << 14)
PAD_CTL_PUS_47K_UP              (1 << 14)
PAD_CTL_PUS_100K_UP             (2 << 14)
PAD_CTL_PUS_22K_UP              (3 << 14)
PAD_CTL_PUE                     (1 << 13)
PAD_CTL_PKE                     (1 << 12)
PAD_CTL_ODE                     (1 << 11)
PAD_CTL_SPEED_LOW               (1 << 6)
PAD_CTL_SPEED_MED               (2 << 6)
PAD_CTL_SPEED_HIGH              (3 << 6)
PAD_CTL_DSE_DISABLE             (0 << 3)
PAD_CTL_DSE_240ohm              (1 << 3)
PAD_CTL_DSE_120ohm              (2 << 3)
PAD_CTL_DSE_80ohm               (3 << 3)
PAD_CTL_DSE_60ohm               (4 << 3)
PAD_CTL_DSE_48ohm               (5 << 3)
PAD_CTL_DSE_40ohm               (6 << 3)
PAD_CTL_DSE_34ohm               (7 << 3)
PAD_CTL_SRE_FAST                (1 << 0)
PAD_CTL_SRE_SLOW                (0 << 0)
```

Interactive i.MX Pin Mux Tool

Obsługa przycisku – podsystem GPIO Buttons

Na podstawie przedstawionego opisu *Device Tree*, ze sterownikiem `gpio-keys`, skojarzone zostały przyciski podłączone do wyprowadzeń ***GPIO1_8*** oraz ***GPIO1_9*** .

Obsługa przycisku – podsystem GPIO Buttons

- Od strony procesu w przestrzeni użytkownika, dostęp do przycisków jest realizowany poprzez plik */dev/input/eventX*

Obsługa przycisku – podsystem GPIO Buttons

- Od strony procesu w przestrzeni użytkownika, dostęp do przycisków jest realizowany poprzez plik */dev/input/eventX*

- 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;
};
```

Obsługa przycisku – podsystem GPIO Buttons

- Od strony procesu w przestrzeni użytkownika, dostęp do przycisków jest realizowany poprzez plik */dev/input/eventX*

- 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 (pole `value`).

Obsługa przycisku – podsystem GPIO Buttons

Typy zdarzeń dla pola `.code`:

- `EV_REL` - używany do określenia względnego przesunięcia, np.: ruch myszą o 5 jednostek w lewo od aktualnego położenia,
- `EV_ABS` - używany do bezwzględnego określenia położenia, np.: współrzędne punktu w interfejsach dotykowych,
- `EV_SYN` - separator zdarzeń w czasie lub przestrzeni - wykorzystywany np: w ekranach dotykowych z opcją multitouch,
- `EV_KEY` - używany do opisania stanu urządzeń takich jak klawiatury, przyciski, itd.

Obsługa przycisku – podsystem GPIO Buttons

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

  fd = open ("/dev/input/event2", O_RDONLY);
  if (fd < 0)
    {
      printf ("event2: failed!\n");
      return EXIT_FAILURE;
    }

  while (1)
    {

    } /* while */

exit:
  close (fd);
  return EXIT_FAILURE;
}
```

Obsługa przycisku – podsystem GPIO Buttons

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

  fd = open ("/dev/input/event2", O_RDONLY);
  if (fd < 0)
    {
      printf ("event2: failed!\n");
      return EXIT_FAILURE;
    }

  while (1)
    {

    } /* while */

exit:
  close (fd);
  return EXIT_FAILURE;
}
```

Obsługa przycisku – podsystem GPIO Buttons

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

  fd = open ("/dev/input/event2", O_RDONLY);
  if (fd < 0)
    {
      printf ("event2: failed!\n");
      return EXIT_FAILURE;
    }

  while (1)
    {

    } /* while */

exit:
  close (fd);
  return EXIT_FAILURE;
}
```

```
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:0") : printf("KEY_DOWN:1");
    else if (ev.code == KEY_UP)
      ev.value ? printf("KEY_UP:0") : printf("KEY_UP:1");
    else
      puts ("WTF?!");
  }
```

Obsługa przycisku – podsystem GPIO Buttons

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

  fd = open ("/dev/input/event2", O_RDONLY);
  if (fd < 0)
    {
      printf ("event2: failed!\n");
      return EXIT_FAILURE;
    }

  while (1)
    {

    } /* while */

exit:
  close (fd);
  return EXIT_FAILURE;
}
```

```
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:0") : printf("KEY_DOWN:1");
    else if (ev.code == KEY_UP)
      ev.value ? printf("KEY_UP:0") : printf("KEY_UP:1");
    else
      puts ("WTF?!");
  }
```

ĆWICZENIE 2.6:

Obsługa diod LED z wykorzystaniem podsystemu LED Class Driver

LED Class Driver - konfiguracja sterownika w jądrze

- Konfiguracja sterownika *LED Class Driver* w jądrze systemu:

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

- Konfiguracja wyzwalaczy dla sterownika *LED Class Driver*:

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

LED Class Driver – opis Device Tree

- Opis w *Device Tree* definiujący podłączone diody LED:

```
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";
  };
};
```

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

    pinctrl_gpio_leds: gpio-leds {
      fsl,pins = <
        MX6UL_PAD_JTAG_TDI__GPIO1_IO13 0x17099
        MX6UL_PAD_JTAG_TDO__GPIO1_IO12 0x17099
      >;
    };
  };
};
```

https://www.kernel.org/doc/Documentation/devicetree/bindings/leds/leds-gpio.txt

ĆWICZENIE 2.8:
Magistrala I2C

I2C – konfiguracja jądra systemu "w pigułce"

- Włączenie sterownika I2C w jądrze systemu:

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

- Włączenie kontrolera magistrali I2C:

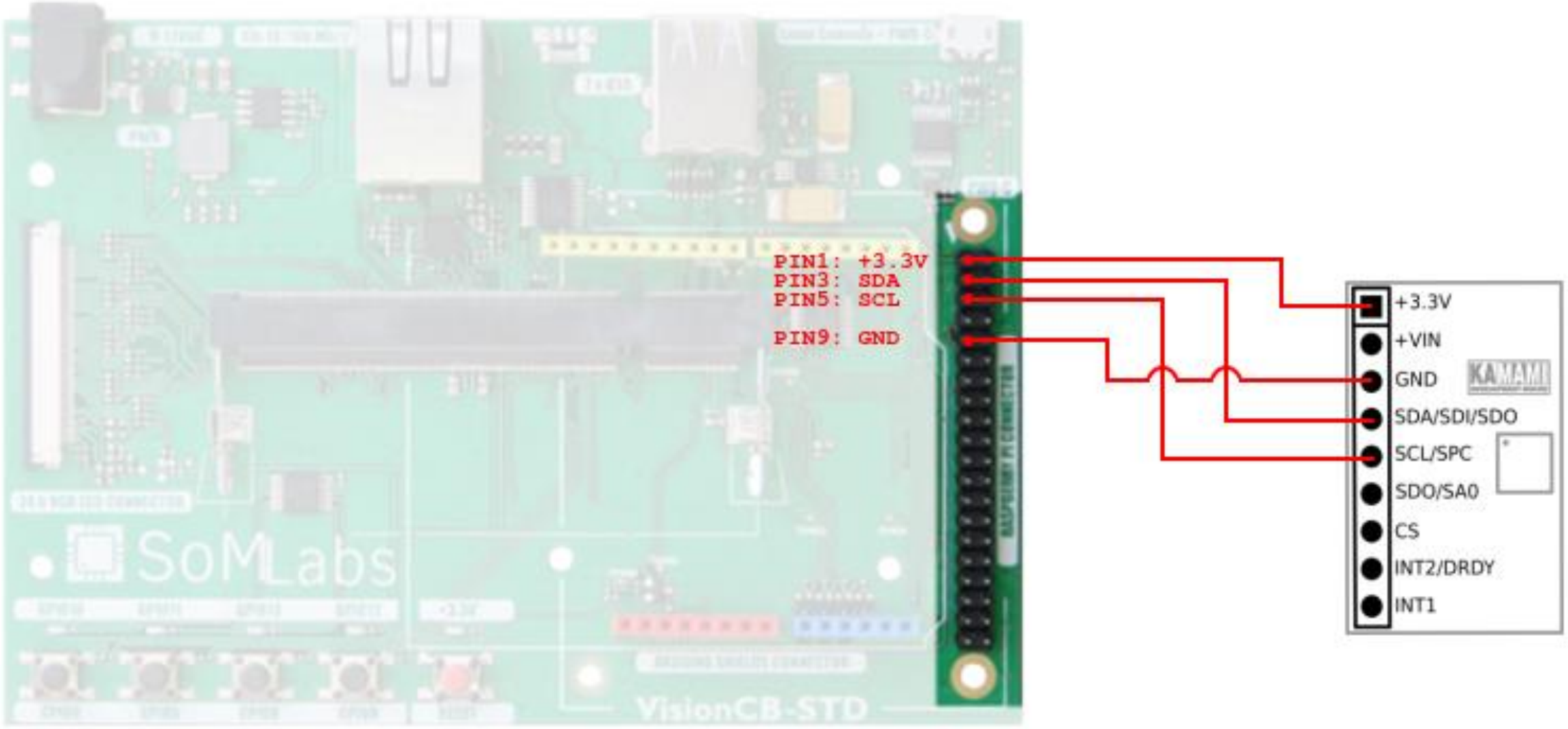
```
Device Drivers --->
  [*] I2C support --->
    [*] I2C Hardware Bus Support --->
      <*> IMX I2C interface
      < > GPIO-based bitbanging I2C
```

Magistrala I2C – opis Device Tree

```
&i2c1 {

  clock-frequency = <100000>;
  pinctrl-names = "default";
  pinctrl-0 = <&pinctrl_i2c1>;
  //status = "okay";

};

&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_i2c1: i2c1grp {
      fsl,pins = <
        MX6UL_PAD_UART4_TX_DATA__I2C1_SCL 0x4001b8b0
        MX6UL_PAD_UART4_RX_DATA__I2C1_SDA 0x4001b8b0
      >;
    };

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

Magistrala I2C - podłączenie żyroskopu



Magistrala I2C – pakiet i2c-tools

i2cdetect – umożliwia przeskanowanie wskazanej w parametrze magistrali I2C. Wynikiem działania polecenia jest tablica adresów wraz z listą dostępnych na magistrali urządzeń.

- ogólna postać polecenia:

```
i2cdetect [-y] [-a] [-q|-r] I2CBUS [FIRST LAST]
```

- przeskanowanie magistrali sprzętowej i2c_1:

```
i2cdetect -y 1
```

Magistrala I2C – pakiet i2c-tools

i2cset – umożliwia zapisanie określonej wartości pod wybrany rejestr układu Slave. Pierwszym parametrem polecenia jest numer porządkowy magistrali, następnie adres układu Slave, numer rejestru, wartość zapisywanej danej oraz jej rozmiar (domyślnie 1 bajt).

- ogólna postać polecenia:

```
i2cset [-f] [-y] [-m MASK] I2CBUS CHIP-ADDRESS DATA-ADDRESS [VALUE [MODE]]
```

- zapis wartości 0x20 do rejestru 0x10 układu o adresie 0x30 podłączonego do magistrali i2c_1

```
i2cset -y 1 0x30 0x10 0x20
```

Magistrala I2C – pakiet i2c-tools

i2cget – pozwala na odczyt danej z wybranego układu slave i określonego rejestru. Pierwszym parametrem polecenia jest numer porządkowy magistrali, następnie adres układu slave, numer rejestru spod którego będziemy wykonywać odczyt oraz rozmiar odczytywanej danej (domyślnie jednobajtowej).

- ogólna postać polecenia:

```
i2cget [-f] [-y] I2CBUS CHIP-ADDRESS [DATA-ADDRESS [MODE]]
```

- odczyt dwubajtowej zawartości rejestru 0x0F układu o adresie 0x6b podłączonego do i2c_1:

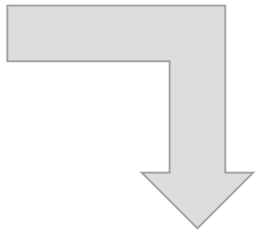
```
i2cget -y 1 0x6b 0x0F w
```

Magistrala I2C - obsługa programowa

- `open();`
- `write();`
- `read();`
- `close();`

Magistrala I2C - obsługa programowa

- `open();`
- `write();`
- `read();`
- `close();`

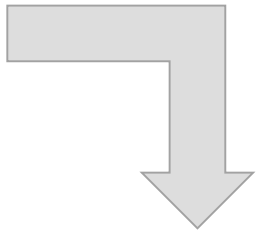


- **Otworzenie pliku urządzenia /dev/i2c-X**
- Wybranie adresu urządzenia SLAVE
- Zapis danych do urządzenia SLAVE
- Odczyt danych z urządzenia SLAVE

```
/* open i2c device */
int i2c_fd = open ("/dev/i2c-1", O_RDWR);
if (i2c_fd < 0)
  return EXIT_FAILURE;

/* set slave address */
int ret = ioctl (i2c_fd, I2C_SLAVE, SLAVE_ADDR);
  return EXIT_FAILURE;

/* write data */
char buf[3];
buf[0] = 0x01;
buf[1] = 0x0F;
buf[2] = 0xFF;

if (write (i2c_fd, buf, 3) != 3)
  return EXIT_FAILURE;

/* read data */
if (read (i2c_fd, buf, 3) != 3)
  return EXIT_FAILURE;
```

Magistrala I2C - obsługa programowa

- `open();`
- `write();`
- `read();`
- `close();`

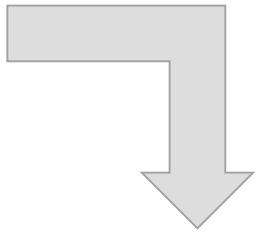


- Otworzenie pliku urządzenia /dev/i2c-X
- **Wybranie adresu urządzenia SLAVE**
- Zapis danych do urządzenia SLAVE
- Odczyt danych z urządzenia SLAVE

```
/* open i2c device */
int i2c_fd = open ("/dev/i2c-1", O_RDWR);
if (i2c_fd < 0)
  return EXIT_FAILURE;

/* set slave address */
int ret = ioctl (i2c_fd, I2C_SLAVE, SLAVE_ADDR);
  return EXIT_FAILURE;

/* write data */
char buf[3];
buf[0] = 0x01;
buf[1] = 0x0F;
buf[2] = 0xFF;

if (write (i2c_fd, buf, 3) != 3)
  return EXIT_FAILURE;

/* read data */
if (read (i2c_fd, buf, 3) != 3)
  return EXIT_FAILURE;
```

Magistrala I2C - obsługa programowa

- `open();`
- `write();`
- `read();`
- `close();`

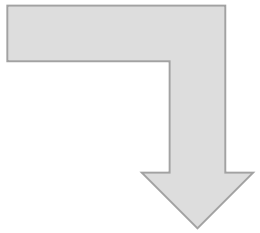


- Otworzenie pliku urządzenia /dev/i2c-X
- Wybranie adresu urządzenia SLAVE
- **Zapis danych do urządzenia SLAVE**
- Odczyt danych z urządzenia SLAVE

```
/* open i2c device */
int i2c_fd = open ("/dev/i2c-1", O_RDWR);
if (i2c_fd < 0)
  return EXIT_FAILURE;

/* set slave address */
int ret = ioctl (i2c_fd, I2C_SLAVE, SLAVE_ADDR);
  return EXIT_FAILURE;

/* write data */
char buf[3];
buf[0] = 0x01;
buf[1] = 0x0F;
buf[2] = 0xFF;

if (write (i2c_fd, buf, 3) != 3)
  return EXIT_FAILURE;

/* read data */
if (read (i2c_fd, buf, 3) != 3)
  return EXIT_FAILURE;
```

Magistrala I2C - obsługa programowa

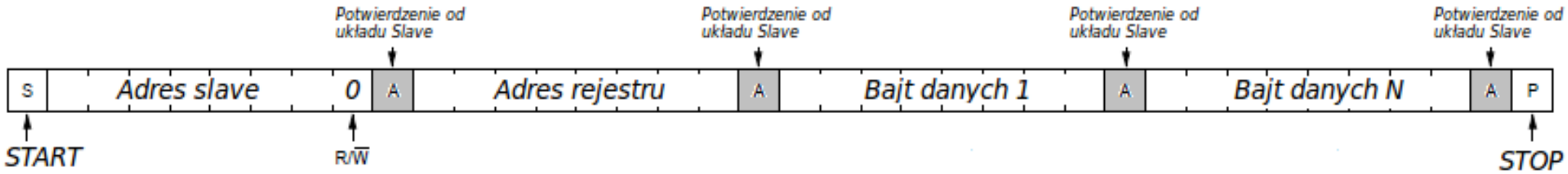
- `open();`
- `write();`
- `read();`
- `close();`



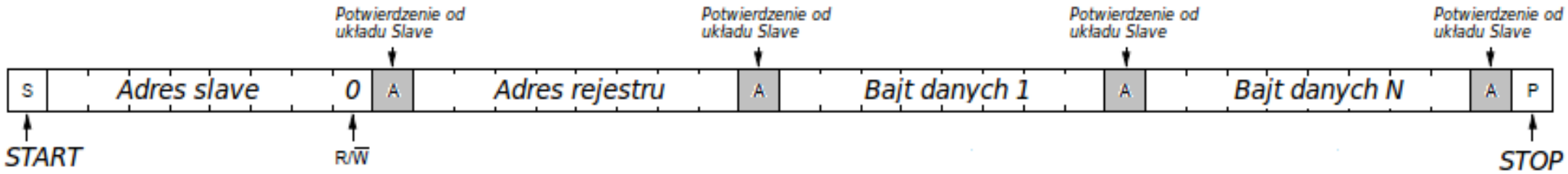
- Otworzenie pliku urządzenia /dev/i2c-X
- Wybranie adresu urządzenia SLAVE
- Zapis danych do urządzenia SLAVE
- **Odczyt danych z urządzenia SLAVE**

```
/* open i2c device */
int i2c_fd = open ("/dev/i2c-1", O_RDWR);
if (i2c_fd < 0)
  return EXIT_FAILURE;

/* set slave address */
int ret = ioctl (i2c_fd, I2C_SLAVE, SLAVE_ADDR);
  return EXIT_FAILURE;

/* write data */
char buf[3];
buf[0] = 0x01;
buf[1] = 0x0F;
buf[2] = 0xFF;

if (write (i2c_fd, buf, 3) != 3)
  return EXIT_FAILURE;

/* read data */
if (read (i2c_fd, buf, 3) != 3)
  return EXIT_FAILURE;
```

Magistrala I2C – komunikacja na poziomie bajtowym

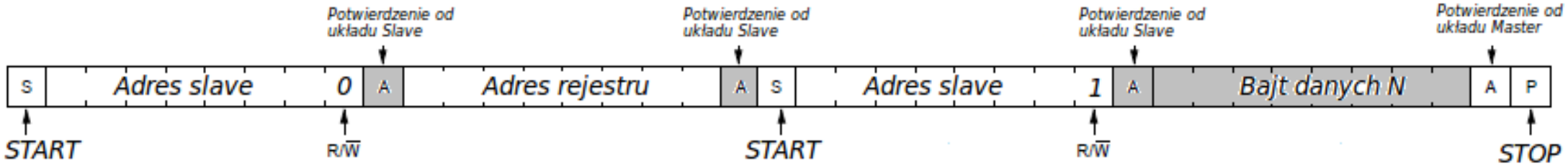


Magistrala I2C – komunikacja na poziomie bajtowym

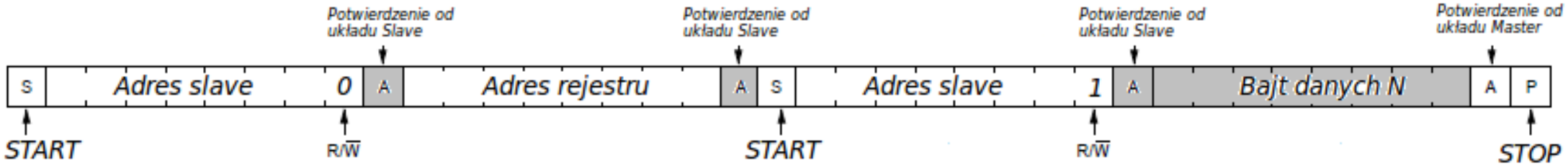



```
unsigned char init_seq[6];

init_seq[0] = (CTRL_REG1 | AUTO_INCREMENT);
init_seq[1] = 0xCF; /* CTRL_REG1: normal mode, xyz enable */
init_seq[2] = 0x01; /* CTRL_REG2: <default value> */
init_seq[3] = 0x00; /* CTRL_REG3: <default value> */
init_seq[4] = 0x80; /* CTRL_REG4: 250dps, Block Data Update */
init_seq[5] = 0x02; /* CTRL_REG5: <default value> */

if (write (i2c_fd, init_seq, 6) != 6)
  return -1;
```

Magistrala I2C – komunikacja na poziomie bajtowym

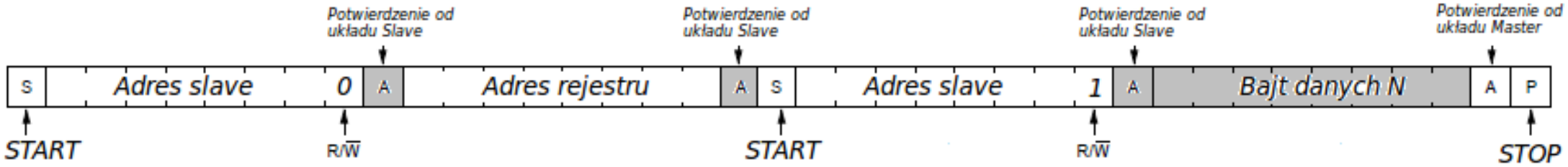


Magistrala I2C – komunikacja na poziomie bajtowym

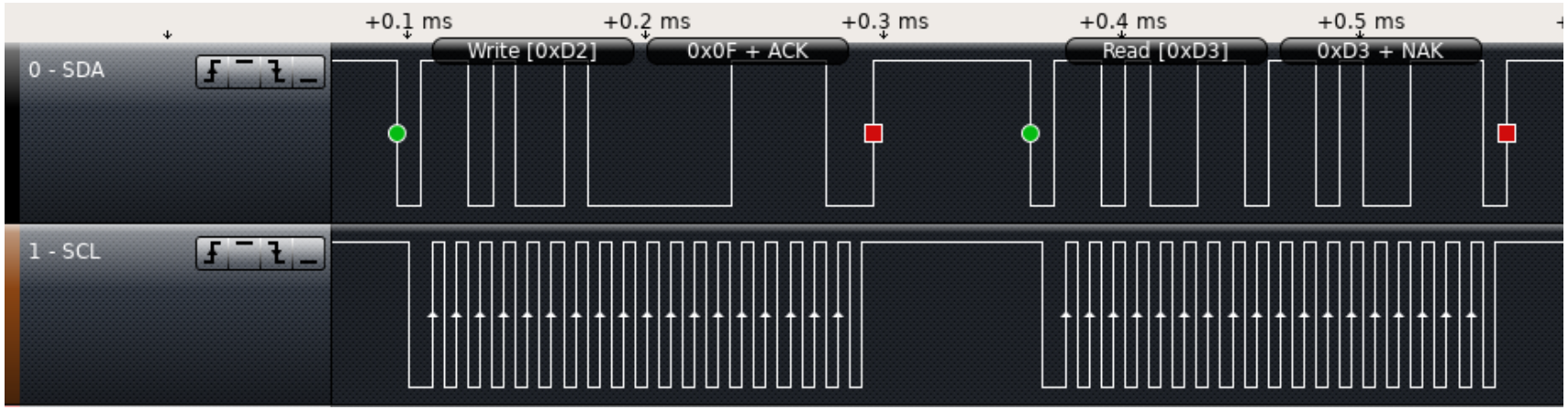



```
buf[0] = reg;
write(i2c, buf, 1);
read(i2c, buf, 1);
return buf[0];
```

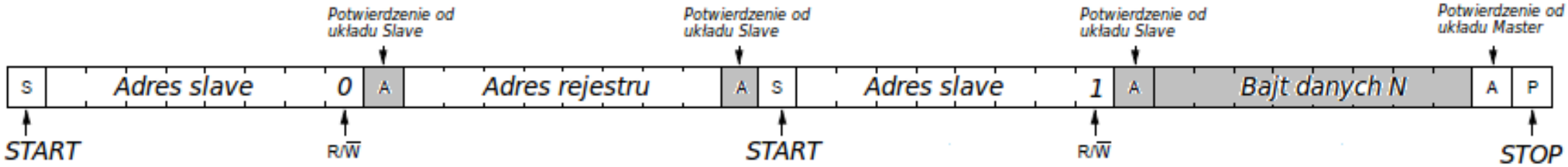
Magistrala I2C – komunikacja na poziomie bajtowym



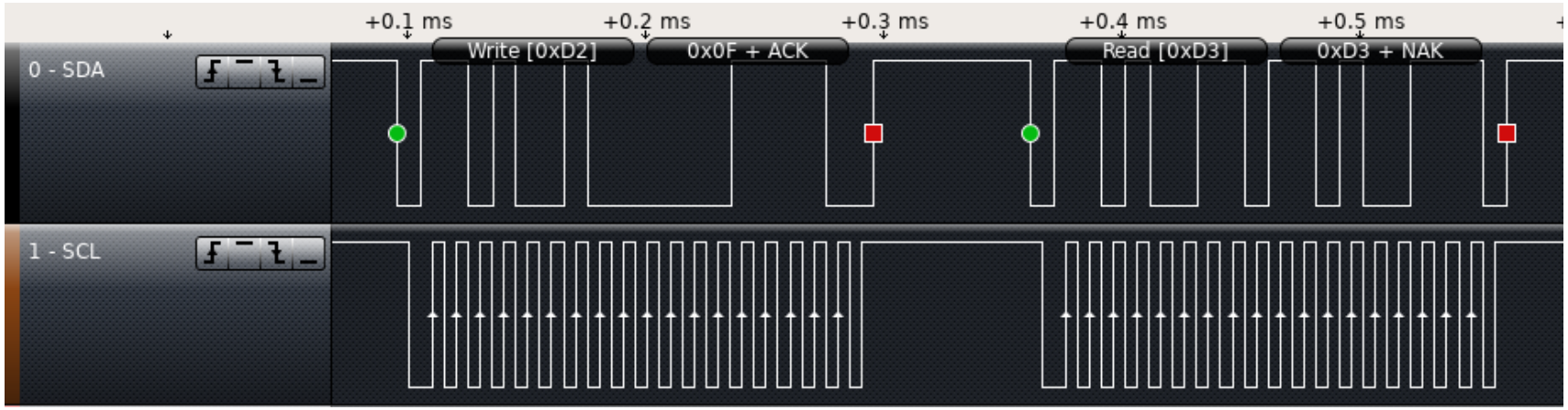

```
buf[0] = reg;
write(i2c, buf, 1);
read(i2c, buf, 1);
return buf[0];
```

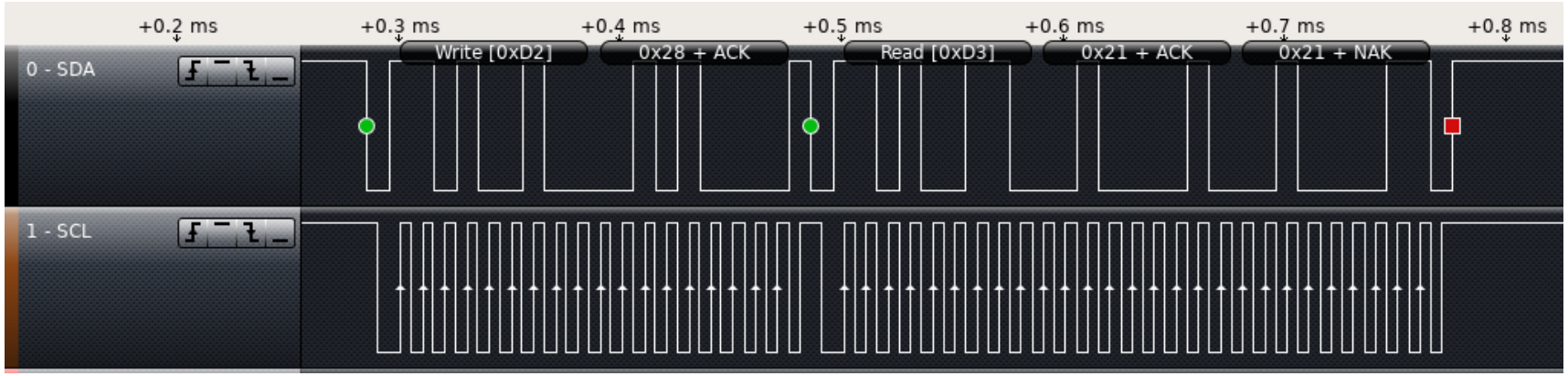


Magistrala I2C – komunikacja na poziomie bajtowym

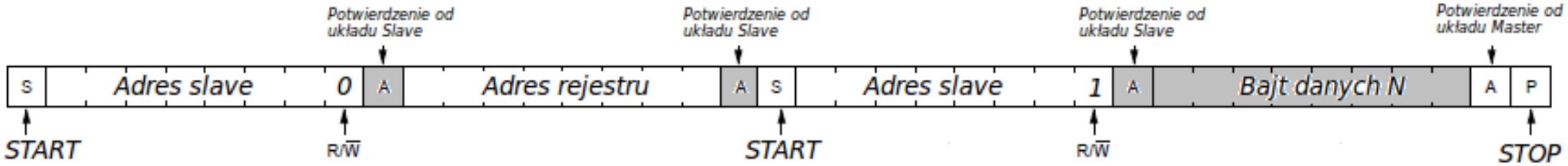



```
buf[0] = reg;
write(i2c, buf, 1);
read(i2c, buf, 1);
return buf[0];
```

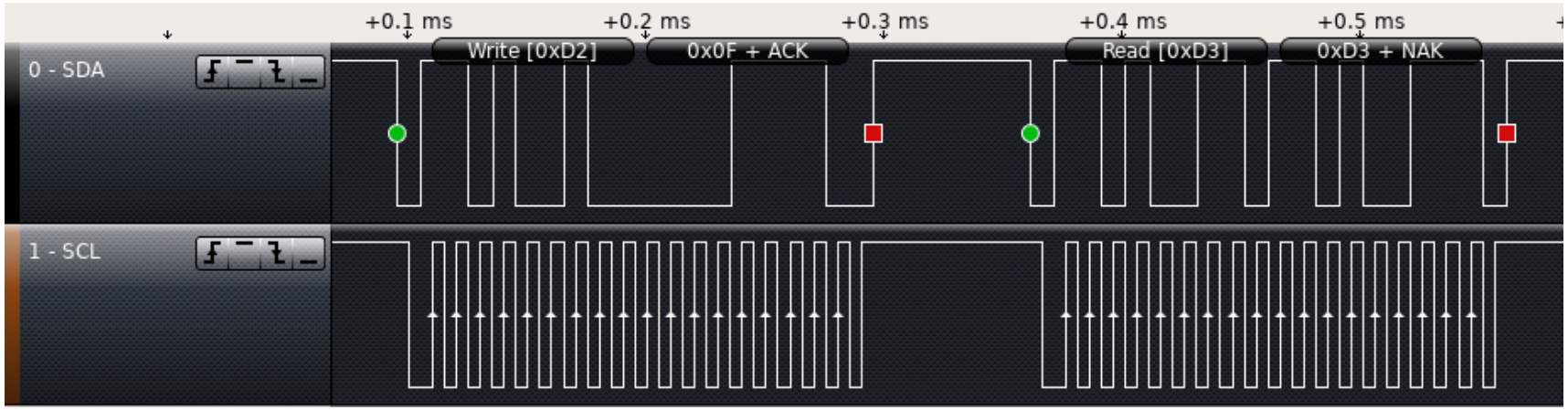




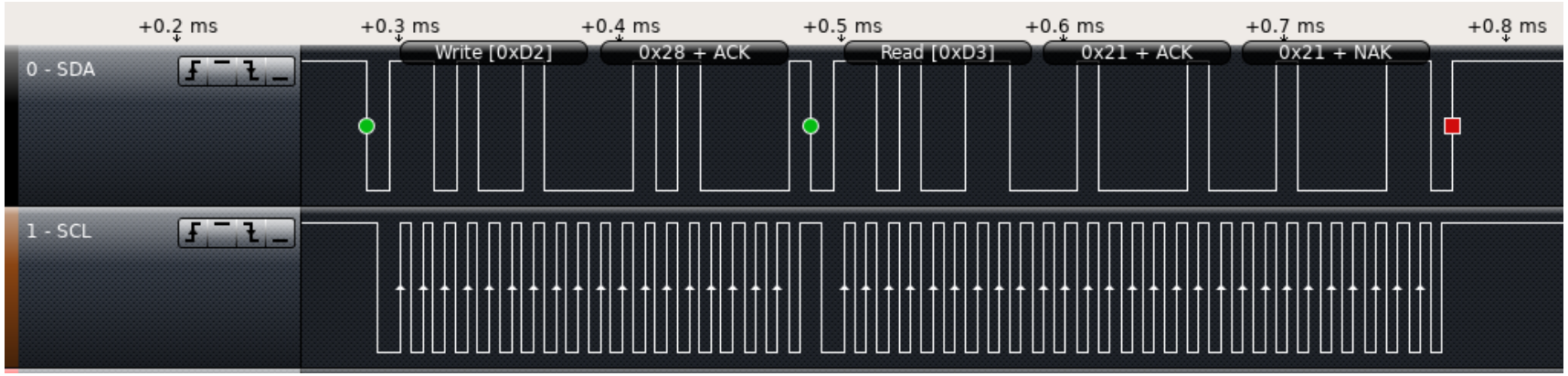
Magistrala I2C – komunikacja na poziomie bajtowym




```
buf[0] = reg;
write(fd, buf, 1);
read(fd, buf, 1);
return buf[0];
```




```
ioctl (fd, I2C_RDWR, …);
```



Magistrala I2C – ioctl (I2C_RDWR)

```
unsigned char reg_addr = STATUS_REG;
unsigned char reg_data[1];
int ret;

struct i2c_msg messages[] =
  {
    {
      GYRO_ADDR,         /* slave address */
      0,                 /* flags: 0 */
      sizeof(reg_addr), /* transfer size */
      &reg_addr          /* data */
    },

    {
      GYRO_ADDR,         /* slave adddress */
      I2C_M_RD,          /* flags: READ */
      sizeof(reg_data), /* transfer size */
      reg_data           /* data */
    }
  };
```

Magistrala I2C – ioctl (I2C_RDWR)

```
unsigned char reg_addr = STATUS_REG;
unsigned char reg_data[1];
int ret;

struct i2c_msg messages[] =
  {
    {
      GYRO_ADDR,         /* slave address */
      0,                 /* flags: 0 */
      sizeof(reg_addr),  /* transfer size */
      &reg_addr          /* data */
    },

    {
      GYRO_ADDR,         /* slave adddress */
      I2C_M_RD,          /* flags: READ */
      sizeof(reg_data),  /* transfer size */
      reg_data           /* data */
    }
  };
```

```
struct i2c_rdwr_ioctl_data packets =
  {
    messages,
    sizeof(messages) / sizeof(struct i2c_msg)
  };

ret = ioctl (i2c_fd, I2C_RDWR, &packets);
if (ret < 0)
  return ret;
```

Magistrala I2C - moduł żyroskopu

Name	Type	Register address		Default
		Hex	Binary	
Reserved	-	00-0E	-	-
WHO_AM_I	r	0F	000 1111	11010100
Reserved	-	10-1F	-	-
CTRL_REG1	rw	20	010 0000	00000111
CTRL_REG2	rw	21	010 0001	00000000
CTRL_REG3	rw	22	010 0010	00000000
CTRL_REG4	rw	23	010 0011	00000000
CTRL_REG5	rw	24	010 0100	00000000
REFERENCE	rw	25	010 0101	00000000
OUT_TEMP	r	26	010 0110	output
STATUS_REG	r	27	010 0111	output
OUT_X_L	r	28	010 1000	output
OUT_X_H	r	29	010 1001	output
OUT_Y_L	r	2A	010 1010	output
OUT_Y_H	r	2B	010 1011	output
OUT_Z_L	r	2C	010 1100	output
OUT_Z_H	r	2D	010 1101	output

Magistrala I2C - moduł żyroskopu

Name	Type	Register address		Default
		Hex	Binary	
Reserved	-	00-0E	-	-
WHO_AM_I	r	0F	000 1111	11010100
Reserved	-	10-1F	-	-
CTRL_REG1	rw	20	010 0000	00000111
CTRL_REG2	rw	21	010 0001	00000000
CTRL_REG3	rw	22	010 0010	00000000
CTRL_REG4	rw	23	010 0011	00000000
CTRL_REG5	rw	24	010 0100	00000000
REFERENCE	rw	25	010 0101	00000000
OUT_TEMP	r	26	010 0110	output
STATUS_REG	r	27	010 0111	output
OUT_X_L	r	28	010 1000	output
OUT_X_H	r	29	010 1001	output
OUT_Y_L	r	2A	010 1010	output
OUT_Y_H	r	2B	010 1011	output
OUT_Z_L	r	2C	010 1100	output
OUT_Z_H	r	2D	010 1101	output

Magistrala I2C - moduł żyroskopu

Name	Type	Register address		Default
		Hex	Binary	
Reserved	-	00-0E	-	-
WHO_AM_I	r	0F	000 1111	11010100
Reserved	-	10-1F	-	-
CTRL_REG1	rw	20	010 0000	00000111
CTRL_REG2	rw	21	010 0001	00000000
CTRL_REG3	rw	22	010 0010	00000000
CTRL_REG4	rw	23	010 0011	00000000
CTRL_REG5	rw	24	010 0100	00000000
REFERENCE	rw	25	010 0101	00000000
OUT_TEMP	r	26	010 0110	output
STATUS_REG	r	27	010 0111	output
OUT_X_L	r	28	010 1000	output
OUT_X_H	r	29	010 1001	output
OUT_Y_L	r	2A	010 1010	output
OUT_Y_H	r	2B	010 1011	output
OUT_Z_L	r	2C	010 1100	output
OUT_Z_H	r	2D	010 1101	output

Magistrala I2C - moduł żyroskopu

Name	Type	Register address		Default
		Hex	Binary	
Reserved	-	00-0E	-	-
WHO_AM_I	r	0F	000 1111	11010100
Reserved	-	10-1F	-	-
CTRL_REG1	rw	20	010 0000	00000111
CTRL_REG2	rw	21	010 0001	00000000
CTRL_REG3	rw	22	010 0010	00000000
CTRL_REG4	rw	23	010 0011	00000000
CTRL_REG5	rw	24	010 0100	00000000
REFERENCE	rw	25	010 0101	00000000
OUT_TEMP	r	26	010 0110	output
STATUS_REG	r	27	010 0111	output
OUT_X_L	r	28	010 1000	output
OUT_X_H	r	29	010 1001	output
OUT_Y_L	r	2A	010 1010	output
OUT_Y_H	r	2B	010 1011	output
OUT_Z_L	r	2C	010 1100	output
OUT_Z_H	r	2D	010 1101	output

ĆWICZENIE 2.7:
Magistrala SPI

SPI – konfiguracja jądra systemu "w pigułce"

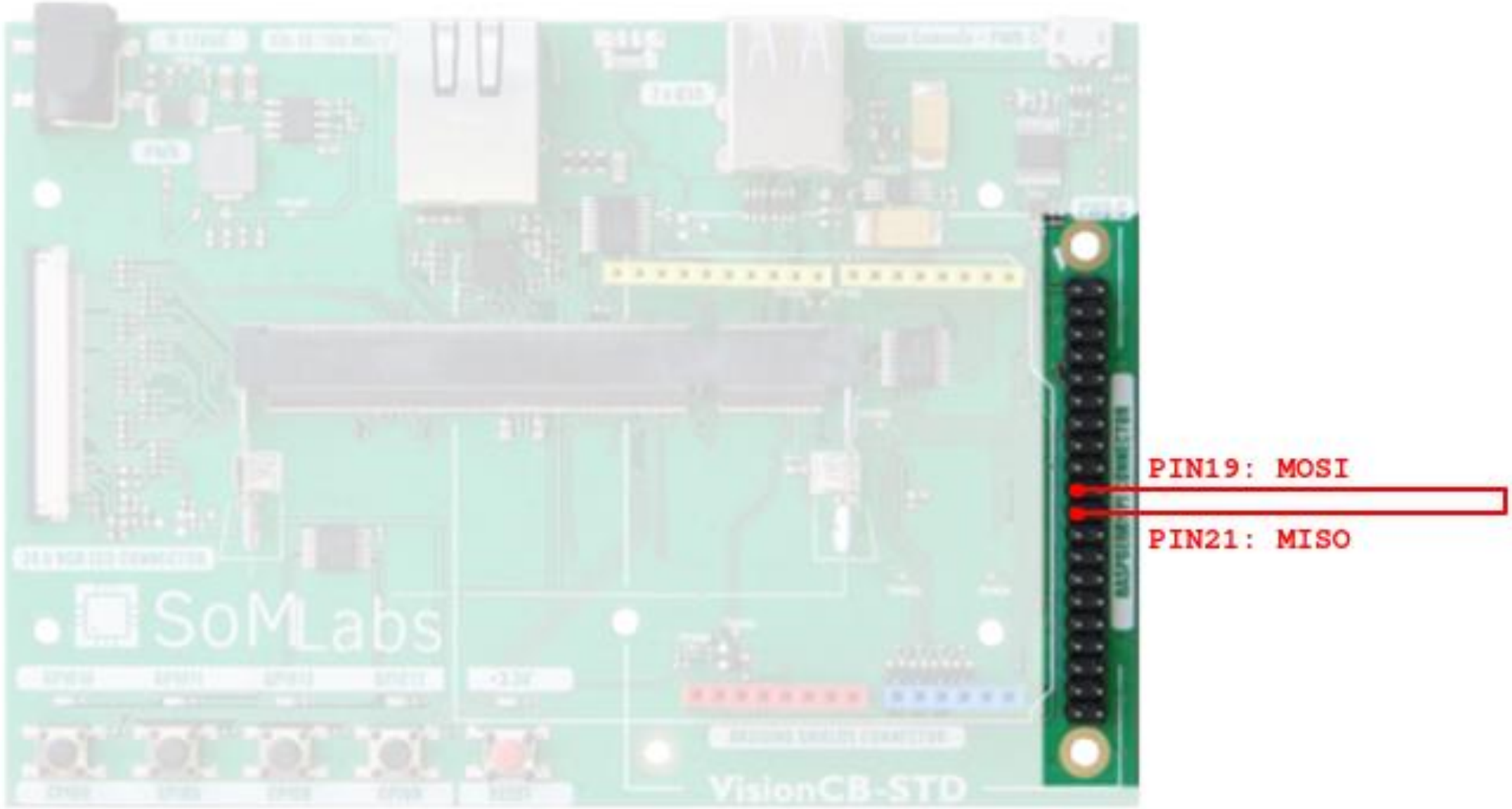
- Włączenie kontrolera magistrali SPI w jądrze systemu:

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

- Włączenie sterownika `spidev` umożliwiającego uzyskanie dostępu do magistrali z przestrzeni użytkownika:

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

Magistrala SPI - połączenie "loopback"



Magistrala SPI - obsługa programowa

```
long param;
ioctl(handle, SPI_IOC_xxx, &param);
```

```
ioctl(handle, SPI_IOC_MESSAGE(n), tab)
```

Argument SPI_IOX_xxx	Opis/parametry
SPI_IOC_RD_MODE SPI_IOC_WR_MODE	Umożliwia ustawienie lub odczytanie trybu pracy magistrali SPI, Dopuszczalne parametry to SPI_MODE_0 do SPI_MODE_3 , które umożliwiają wybór sposobu pracy magistrali.
SPI_IOC_RD_LSB_FIRST SPI_IOC_WR_LSB_FIRST	Umożliwiają odczyt lub zmianę kolejności bitów wysyłanych magistralą. Wartość równa zero oznacza tryb pracy MSB, natomiast wartość różna od zera tryb pracy LSB
SPI_IOC_RD_BITS_PER_WORD SPI_IOC_WR_BITS_PER_WORD	Umożliwia odczytanie lub zapisanie, liczby bitów które będą przesyłane magistralą SPI podczas transmisji pojedynczego słowa
SPI_IOC_RD_MAX_SPEED_HZ SPI_IOC_WR_MAX_SPEED_HZ	Pozwala na zapisanie lub odczytanie maksymalnej dopuszczalnej prędkości transmisji SPI.

```
struct spi_ioc_transfer {
    __u64           tx_buf;
    __u64           rx_buf;
    __u32           len;
    __u32           speed_hz;
    __u16           delay_usecs;
    __u8            bits_per_word;
    __u8            cs_change;
    __u32           pad;
};
```