Użytkowanie Odroid-C1+ (3)

System kontroli wersji Git, moduł do zarządzania urządzeniami udev

Jedną z zalet ODROIDa jest zgodność pinów złącza J2 ze złączem P1 w Raspberry Pi. Dzięki temu możliwe jest użycie wielu dostępnych modułów rozszerzeń przeznaczonych dla Raspberry Pi. Na koniec zobaczymy jak ułatwić sobie pracę z edytorem VIM i systemem kontroli wersji Git, a także zapoznamy się z modułem udev, służącym do dynamicznego zarządzania urządzeniami.

Jako przykład tezy zawartej we wstępnie, posłużą nam moduły KAmodRPi PwrRELAY zawierający dwa przekaźniki elektromechaniczne oraz KAmodRPi ADC_DAC z przetwornikami A/C i C/A. Ich użycie znacznie ułatwia samodzielne eksperymentowanie z Odroidem.

KAmodRPi PwrRELAY – powtórka z GPIO

Pierwszy z modułów, KAmodRPi PwrRELAY (*https://goo.gl/6rcPAM*) wyposażono w dwa przekaźniki elektromechaniczne oraz dwa wyjścia tranzystorowe, którymi możemy sterować za pomocą linii GPIO. Można go zatem wykorzystać na przykład do sterowania domowymi urządzeniami elektrycznymi. W tym prostym przykładzie jeden z przekaźników posłuży do włączania i wyłączania lampki biurkowej za pośrednictwem terminala.

Na początku warto przyjrzeć się dokumentacji modułu, ponieważ znajdują się w niej m. in. informacje o sposobie jego podłączenia. Według schematu elektrycznego, przekaźniki są podłączone do linii 22 i 23 GPIO, które odpowiadają liniom 115 i 104 ODROIDa. Jeżeli poprawnie wykonaliśmy wszystkie kroki z poprzedniego artykułu, możemy je skonfigurować za pomocą terminala (bez uprawnień roota):

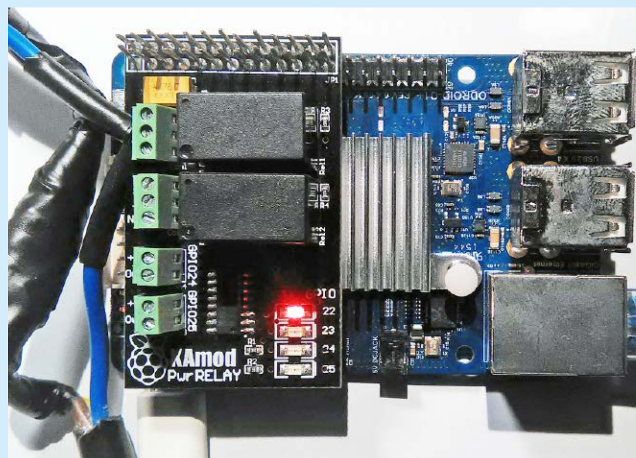
```
echo 115 > /sys/class/gpio/export
echo out > /sys/class/gpio/gpio115/direction
```

Ustawienie lub zerowanie GPIO wykonujemy poleceniami:

```
echo 1 > /sys/class/gpio/gpio115/value
echo 0 > /sys/class/gpio/gpio115/value
```

Oczywiście, do obsługi portu możemy wykorzystać również jedną z przykładowych aplikacji, omówionych w poprzednim artykule. O poprawnym działaniu modułu, a tym samym o stanie przekaźnika, poinformuje nas czerwony LED – świecący, gdy pin zostanie ustawiony. Mając działający i skonfigurowany moduł możemy zabrać się za ciąg dalszy, czyli podłączenie lampki do przekaźnika.

Każdy z przekaźników zamontowanych na module ma trzy styki: NC (Normally Closed), COM (Common), NO (Normally Open). Przy braku zasilania przekaźnika styki NC i COM są zwarte. Po ustawieniu wyjścia sterującego, zostaną zwarte styki COM i NO, co w module jest sygnalizowane świeceniem LED. Aby przyłączyć lampkę do przekaźnika, powinniśmy dołączyć ją do wejścia NO, natomiast do wejścia COM jeden z przewodów z gniazdka elektrycznego (lub odwrotnie). Dzięki temu, po odłączeniu zasilania obwód będzie rozwarty. Drugi przewód z gniazdka należy na stałe zewrzeć z drugim przewodem lampki. Teraz pozostaje przetestować zmontowany układ (**fotografia 1**).



Fotografia 1. ODROID z podłączonymi przekaźnikami

Na koniec warto jeszcze przyjrzeć się pewnemu bardzo przydatnemu narzędziu obecnemu w systemie Linux. Jest nim cron i służy do cyklicznego wykonywania zadań. Jako przykład możemy wykonać cykliczne zapalanie i gaszenie światła.

Cron jest demonem systemu Linux, czyli programem uruchamianym przy starcie systemu i działającym w tle przez cały czas lub do momentu wyłączenia go przez użytkownika (do wyłączania demonów zwykle potrzebne są uprawnienia administratora). Program ten raz na minutę odczytuje plik zawierający listę zleconych mu zadań. Każdy użytkownik ma własną listę zadań, którą można wyświetlić poleceniem crontab -l.

Jeżeli wcześniej nie używaliśmy crona, plik ten może nie istnieć. Do jego tworzenia i edycji służy jedno polecenie: crontab -e. Powoduje ono otwarcie listy zadań w domyślnym edytorze tekstu. W ramach ćwiczenia skonfigurujemy crona, aby zmieniał stan wyjścia pinu raz na minutę. W tym celu wykorzystamy jeden z programów z poprzedniego artykułu: gpio_set_vlue. W oknie edytora dodajmy na samym końcu dwa wiersze:

```
1-59/2 * * * * /home/odroid/gpio _ set _ value 115 1
0-58/2 * * * * /home/odroid/gpio _ set _ value 115 0
```

Wpis spowoduje wywołanie w minutach parzystych (0-58/2) programu /home/pi/gpio_set_value z argumentami 115 1, natomiast w nieparzystych (1-58/2) tego samego programu z argumentami 115 0. Warto zwrócić uwagę na to, że musimy podać pełną ścieżkę do programu, który chcemy wykonać. Po zapisaniu pliku i wyjściu z edytora powinniśmy obserwować zmianę stanu przekaźnika co 1 minutę. Aby przerwać wykonywanie zadań, wystarczy usunąć wpis lub usunąć cały plik za pomocą cron -r. Należy także pamiętać, że do poprawnego działania niezbędne jest wcześniejsze wyeksportowanie pinu 115 i skonfigurowanie go jako wyjście. Jednym ze sposobów, aby działo się to automatycznie przy starcie systemu, jest użycie menedżera urządzeń udev.

Podobnie jak ostatnio, do pliku /etc/udev/rules.d/90-gpio.rules dodamy reguły:

```
SUBSYSTEM=="gpio",KERNEL=="gpiochip0",RUN+="/bin/
sh -c ,echo 115 > /sys/class/gpio/export'"
SUBSYSTEM=="gpio",KERNEL=="gpio115",RUN+="/bin/sh
-c ,echo out > /sys%p/direction'"
```

Poprzednie części kursu i dodatkowe materiały dostępne są na FTP: ftp://ep.com.pl, user: 33948, pass: 5gcckdmq

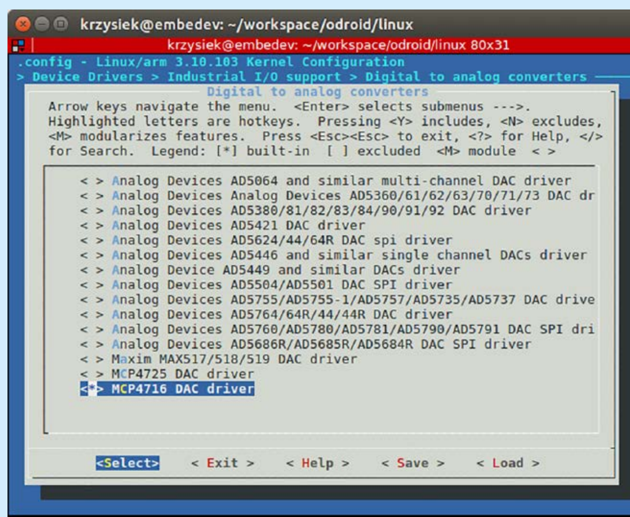
Pierwsza z nich eksportuje w sterowniku GPIO pin 115, natomiast druga konfiguruje go jako wyjście.

Do poprawnego działania crona niezbędne jest jeszcze ustawienie czasu w systemie. Jeżeli nasz ODROID jest przyłączony do Internetu, zrobi to dla nas ntpdate, który powinien być domyślnie zainstalowany. Jedyne, co może być potrzebne, to ustawienie odpowiedniej strefy czasowej za pomocą polecenia sudo dpkg-reconfigure tzdata. Aktualną godziną i strefę czasową możemy sprawdzić poleceniem date.

Możliwości programu cron są o wiele większe, niż te przedstawione w przykładzie. Możemy zlecać wykonywanie zadań w określonych godzinach, dniach i miesiącach podając ścieżki do skryptów lub programów. Zachęcam do eksperymentowania z tym narzędziem – w sieci znajdziemy mnóstwo materiałów dotyczących jego użycia.

KAmodRPi ADC_DAC – sterowniki modułu iio (Industrial Input Output)

Drugi z opisywanych modułów – KAmodRPi ADC_DAC, zawiera dwa przetworniki: MCP3021 i MCP4716. Co prawda w jądrze nie znajdziemy



Rysunek 2. Konfigurowanie jądra za pomocą menuconfig

Listing 1. Struktury i funkcje niezbędne do obsługi MCP4716

```
/*struktura przechowująca wskaźnik do klienta i2c oraz aktualną, 10-bitową wartość na wyjściu przetwornika*/
struct mcp4716_data {
    struct i2c_client *client;
    u16 dac_value;
};

/*struktura zawierająca dane dotyczące pojedynczego kanału przetwornika (typ obsługiwanej wartości, włączenie indeksowania kanałów,
typ kanału - wyjście, numer kanału, informacje specyficzne dla kanału - wartość wyjściowa)*/
static const struct iio_chan_spec mcp4716_channel = {
    .type           = IIO_VOLTAGE,
    .indexed        = 1,
    .output         = 1,
    .channel        = 0,
    .info_mask_separate = BIT(IIO_CHAN_INFO_RAW),
};

/*funkcja wysyłająca przez i2c nową wartość wyjściową przetwornika (według specyfikacji mcp4716, 10-bitowa wartość musi być podczas
transmisji przesunięta o dwa bity w lewo)*/
static int mcp4716_set_value(struct iio_dev *indio_dev, int val)
{
    struct mcp4716_data *data = iio_priv(indio_dev);
    u8 outbuf[2];
    int ret;

    if (val >= (1 << 10) || val < 0) return -EINVAL;
    val <<= 2;
    outbuf[0] = (val >> 8) & 0xf;
    outbuf[1] = val & 0xfc;
    ret = i2c_master_send(data->client, outbuf, 2);
    if (ret < 0)
        return ret;
    else if (ret != 2)
        return -EIO;
    else
        return 0;
}

/*funkcja obsługująca odczyt aktualnej wartości wyjściowej przetwornika (jest ona zapisywana w sterowniku po każdym zapisie, dlatego
nie ma potrzeby odczytywania jej przez i2c)*/
static int mcp4716_read_raw(struct iio_dev *indio_dev,
                            struct iio_chan_spec const *chan,
                            int *val, int *val2, long mask)
{
    struct mcp4716_data *data = iio_priv(indio_dev);

    switch (mask) {
    case IIO_CHAN_INFO_RAW:
        *val = data->dac_value;
        return IIO_VAL_INT;
    }
    return -EINVAL;
}

/*funkcja obsługująca zapis nowej wartości wyjściowej przetwornika*/
static int mcp4716_write_raw(struct iio_dev *indio_dev,
                               struct iio_chan_spec const *chan,
                               int val, int val2, long mask)
{
    struct mcp4716_data *data = iio_priv(indio_dev);
    int ret;
    switch (mask) {
    case IIO_CHAN_INFO_RAW:
        ret = mcp4716_set_value(indio_dev, val);
        data->dac_value = val;
        break;
    default:
        ret = -EINVAL;
        break;
    }
    return ret;
}

/*struktura definiująca funkcjonalność sterownika (odczyt i zapis surowych danych)*/
static const struct iio_info mcp4716_info = {
```

Listing 1. cd.

Poprzednie części kursu i dodatkowe materiały dostępne są na FTP: **ftp://ep.com.pl**, user: **33948**, pass: **5gcckdmq**

```
    .read_raw = mcp4716_read_raw,
    .write_raw = mcp4716_write_raw,
    .driver_module = THIS_MODULE,
};

/*funkcja rejestrująca sterownik w jądrze i odczytująca początkową wartość na wyjściu przetwornika*/
static int mcp4716_probe(struct i2c_client *client,
                         const struct i2c_device_id *id)
{
    struct mcp4716_data *data;
    struct iio_dev *indio_dev;
    u8 inbuf[3];
    int err;

    indio_dev = iio_device_alloc(sizeof(*data));
    if (indio_dev == NULL) {
        err = -ENOMEM;
        goto exit;
    }
    data = iio_priv(indio_dev);
    i2c_set_clientdata(client, indio_dev);
    data->client = client;
    indio_dev->dev.parent = &client->dev;
    indio_dev->name = id->name;
    indio_dev->info = &mcp4716_info;
    indio_dev->channels = &mcp4716_channel;
    indio_dev->num_channels = 1;
    indio_dev->modes = INDIO_DIRECT_MODE;
    /* read current DAC value */
    err = i2c_master_recv(client, inbuf, 3);
    if (err < 0) {
        dev_err(&client->dev, „failed to read DAC value”);
        goto exit_free_device;
    }
    data->dac_value = (inbuf[1] << 2) | (inbuf[2] >> 6);
    err = iio_device_register(indio_dev);
    if (err) goto exit_free_device;
        dev_info(&client->dev, „MCP4716 DAC registered\n”);
        return 0;
exit_free_device:
    iio_device_free(indio_dev);
exit:
    return err;
}

/*funkcja usuwająca sterownik i zwalniająca jego zasoby*/
static int mcp4716_remove(struct i2c_client *client)
{
    struct iio_dev *indio_dev = i2c_get_clientdata(client);
    iio_device_unregister(indio_dev);
    iio_device_free(indio_dev);
    return 0;
}

/*struktura zawierająca nazwy wszystkich urządzeń sterownika podłączanych do magistrali i2c (w tym przypadku jest to tylko jeden
przetwornik)*/
static const struct i2c_device_id mcp4716_id[] = {
    { „mcp4716”, 0 },
    { }
};

/*makro tworzące nazwę modułu używaną przez jądro*/
MODULE_DEVICE_TABLE(i2c, mcp4716_id);

/*struktura reprezentująca sterownik urządzenia podłączonego do magistrali i2c i zawierająca wskaźniki do funkcji rejestrującej i usu-
wającej sterownik oraz jego nazwę*/
static struct i2c_driver mcp4716_driver = {
    .driver = {
        .name   = MCP4716_DRV_NAME,
    },
    .probe          = mcp4716_probe,
    .remove         = mcp4716_remove,
    .id_table       = mcp4716_id,
};

/*makro pomocnicze do rejestracji sterownika przez jądro*/
module_i2c_driver(mcp4716_driver);
```

wsparcia dla tych urządzeń, ale możemy napisać dla nich sterowniki wzorując się na podobnych, znajdujących się w źródłach Linuksa. Do rozpoczęcia pracy potrzebujemy odpowiednich narzędzi do kompilowania oraz repozytorium ze źródłami.

KOMPILATOR I KOD JĄDRA. Zestaw narzędzi potrzebnych do skompilowania jądra (tzw. toolchain) znajdziemy na stronie ***https://goo.gl/HVdBiQ***. Po rozpakowaniu archiwum narzędzia są gotowe do użycia. Warto jednak dodać katalog gcc-linaro-arm-linux-gnueabihf-4.7-2013.04-20130415_linux/bin do zmiennej środowiskowej PATH (np. w pliku .bashrc w katalogu domowym), aby programy były dostępne z każdego miejsca w systemie, bez potrzeby podawania pełnej ścieżki.

Do pracy z jądrem Linuksa najwygodniej jest użyć systemu kontroli wersji git. Umożliwia on pobranie źródeł i ułatwia proces wprowadzania zmian. Wersję źródeł dla Odroida pobierzemy poleceniem git clone --depth 1 ***https://goo.gl/EtRpGa***. Po jego wywołaniu na dysku lokalnym zostanie utworzona kopia repozytorium, na której możemy wykonywać i zapisywać dowolne zmiany. Katalog główny repozytorium możemy rozpoznać po mieszczącym się w nim ukrytym katalogu .git. Możemy go zobaczyć wywołując polecenie ls -al w konsoli. To w nim znajduje się cała historia zmian i wersji projektu. Aby ściągnąć najnowsze zmiany z serwera wystarczy wywołać w katalogu repozytorium git pull. Polecenie to powoduje ściągnięcie wszystkich zmian z serwera w stosunku do naszej lokalnej kopii. Operacja z pewnością się powiedzie o ile nie dokonaliśmy w międzyczasie żadnych zmian. W przeciwnym razie, w pierwszej kolejności musimy zapisać własne zmiany. Pomocne przy tym będą cztery polecenia:

```
git status
git add <nazwa _ pliku>
git commit
git stash
```

Pierwsze polecenie pokazuje aktualny stan repozytorium. Po jego wywołaniu dowiemy się, które pliki zostały dodane lub zmienione. W pierwszym przypadku musimy wydać kolejne z poleceń służące dodawaniu plików do repozytorium. Należy o tym pamiętać, ponieważ pliki utworzone w katalogu głównym, lub podkatalogach repozytorium nie są dodawane automatycznie do kontroli wersji. Jeżeli sami o to nie zadbamy, możemy utracić zmiany, np. po przypadkowym usunięciu pliku.

Poprzednie części kursu i dodatkowe materiały dostępne są na FTP: ftp://ep.com.pl, user: 33948, pass: 5gcckdmq

Trzecie polecenie z powyższej listy wprowadza zmiany do repozytorium, w tym także pliki dodane poleceniem git add. Po wywołaniu jesteśmy proszeni o podanie komentarza. Warto zadbać o to, aby komentarze krótko, ale wyczerpująco opisywały wprowadzone zmiany, gdyż potem ułatwia to przeglądanie repozytorium i przywracanie poprzednich wersji plików.

Nie zawsze jednak chcemy, żeby nasze zmiany trafiły do repozytorium. Może się to zdarzyć, gdy ciągle pracujemy nad kodem źródłowym, ale nie jest on jeszcze gotowy. W tej sytuacji pomoże nam ostatnie z poleceń: git stash. Umożliwia ono bezpieczne zachowanie wszelkich dokonanych zmian, ale bez zapisywania ich w repozytorium. Dzięki temu możemy ściągnąć nową wersję projektu z serwera, a następnie przywrócić swoją pracę poleceniem git stash apply.

STEROWNIK MCP4716. Mając aktualne źródła Linuxa możemy rozpocząć pracę nad sterownikami. Przetworniki A/C i C/A, oraz wszelkiego rodzaju sensory, takie jak czujniki temperatury, akcelerometry itp. działają w ramach podsystemu iio (Industrial Input Output) [2]. Dzięki temu, urządzenia te, podłączone za pośrednictwem I²C lub SPI mają wspólny interfejs ułatwiający pisanie programów korzystających z ich sterowników.

Sterownik układu MCP4716 najłatwiej utworzyć korzystając z istniejącego sterownika MCP4725. Układy te różnią się rozdzielczością (MCP4716 ma 10 bitów, natomiast MCP4725 – 12 bitów) oraz obecnością pamięci EEPROM w drugim z nich. Zaczynamy więc od utworzenia nowego pliku w katalogu ze źródłami Linuxa: ./drivers/iio/dac/mcp4716.c i skopiowaniu do niego zawartości pliku ./drivers/iio/dac/mcp4725.c. Na potrzeby przykładu można pozostawić tylko podstawowe funkcje sterownika, a mianowicie zapis i odczyt wartości na wyjściu przetwornika. W ostatecznej wersji pliku powinny znaleźć się struktury i funkcje pokazane na **listingu 1**. Kod źródłowy sterownika można pobrać z repozytorium poleceniem git clone ***https://goo.gl/Nk5Ixr***.

Aby sterownik mógł zostać skompilowany z całym jądrem należy jeszcze dokonać zmian w pliku ./drivers/iio/dac/Makefile, dopisując na końcu obj-$(CONFIG_MCP4716) += mcp4716.o oraz dodając opis przygotowanego sterownika w pliku ./drivers/iio/dac/Kconfig:

```
config MCP4716
    tristate „MCP4716 DAC driver”
    depends on I2C
    ---help---
      Say Y here if you want to build a driver for
the Microchip
      MCP 4716 10-bit digital-to-analog converter
(DAC) with I2C
      interface.
     To compile this driver as a module, choose M
here: the module
      will be called mcp4716.
```

Ostatnią zmianą jest dodanie przetwornika do drzewa urządzeń wczytywanego przez jądro i znajdującego się w pliku ./arch/arm/boot/dts/meson8b_odroidc.dts. W tym celu należy dopisać informację o nowym urządzeniu w sekcji magistrali I2C-A:

```
i2c@c1108500{ /*I2C-A*/
    compatible = „amlogic,aml _ i2c”;
    dev _ name = „i2c-A”;
    status = „ok”;
    reg = <0xc1108500 0x20>;
    device _ id = <1>;
    pinctrl-names=”default”;
    pinctrl-0=<&a _ i2c _ master>;
    #address-cells = <1>;
    #size-cells = <0>;
    use _ pio = <0>;
    master _ i2c _ speed = <100000>;
    mcp4716@60{
        compatible = „microchip,mcp4716”;
        reg = <0x60>;
    };
};
```

KOMPILOWANIE JĄDRA. Z uwagi na to, że dodany sterownik ma znaleźć się bezpośrednio w jądrze, należy je teraz skonfigurować i skompilować. Pierwszym krokiem jest przygotowanie konfiguracji korzystając z domyślnych ustawień make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- odroidc_defconfig. Polecenie to możemy wywołać, jeżeli ścieżka do kompilatora znajduje się w zmiennych środowiskowych. Generuje ono plik .config w katalogu ze źródłami jądra, zawierający domyślną konfigurację jądra dla Odroida. Aby dostosować ją do własnych potrzeb można skorzystać z polecenia make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- menuconfig umożliwiającego zmianę konfiguracji poprzez wygodny interfejs graficzny (**rysunek 2**). Komponenty mogą zostać dołączone do jądra (za pomocą klawisza „Y”), lub skompilowane jako moduły (za pomocą klawisza „M”). W tym przykładzie wszystko zostanie skompilowane razem z jądrem. Należy więc znaleźć i dodać następujące komponenty:

```
Device Drivers > Amlogic Device Drivers > I2C
Hardware Bus support > Amlogic I2C Driver
Device Drivers > Industrial I/O support
Device Drivers > Industrial I/O support > Digital
to analog converters > MCP4716 DAC driver
```

Po wyjściu z narzędzia menuconfig i zapisaniu zmian można przystąpić do kompilowania jądra wydając polecenie make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- -j4. Parametr „-j” umożliwia równoległą kompilację na kilku rdzeniach procesora. Po zakończeniu kompilacji jądra należy jeszcze przygotować potrzebne moduły:

```
make ARCH=arm
CROSS _ COMPILE=arm-linux-gnueabihf- modules
make ARCH=arm CROSS _ COMPILE=arm-linux-gnueabihf-
INSTALL _ MOD _ PATH=../modules modules _ install
```

Pierwsze polecenie kompiluje moduły, natomiast drugie kopiuje je do wskazanego katalogu. Pozostało już tylko przygotować obraz jądra (może być konieczne zainstalowanie pakietu u-boot-tools) i drzewo urządzeń dla bootloadera:

```
make ARCH=arm CROSS _ COMPILE=arm-linux-gnueabihf- uImage
make ARCH=arm CROSS _ COMPILE=arm-linux-gnueabihf- dtbs
```

Zbudowane komponenty należy skopiować na kartę pamięci. Na partycję BOOT muszą trafić obraz jądra (./arch/arm/boot/uImage) i drzewo urządzeń (./arch/arm/boot/dts/meson8b_odroidc.dtb), natomiast katalog z modułami należy skopiować na partycję z plikami systemowymi do katalogu lib.

Teraz można uruchomić system i sprawdzić działanie sterownika. W katalogu /sys/bus/iio/devices powinno znajdować się nowe urządzenie reprezentujące omawiany przetwornik, co można sprawdzić w pliku /sys/bus/iio/devices/iio\:device0/name. Najważniejszym plikiem w podkatalogu urządzenia jest natomiast /sys/bus/iio/devices/iio\:device0/out_voltage0_raw, do którego możemy wpisywać lub odczytywać 10-bitowe wartości wyjściowe przetwornika.

Krzysztof Chojnowski

Literatura:

https://goo.gl/DedF0O
https://goo.gl/bLNzvk
https://goo.gl/exmP06
https://goo.gl/XzUaUS
https://goo.gl/Rx6K4i
https://goo.gl/ugeYpH
https://goo.gl/mfHn27
https://goo.gl/sBQvM5

Poprzednie części kursu i dodatkowe materiały dostępne są na FTP: **ftp://ep.com.pl**, user: **33948**, pass: **5gccldmq**