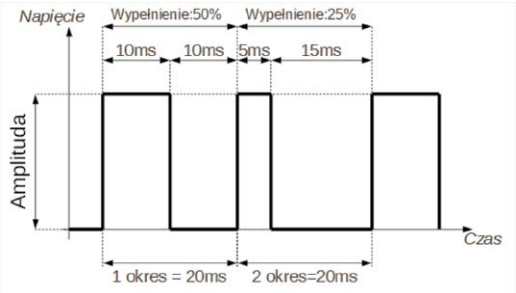
To już 13-ty odcinek kursu Raspberry Pi. Numery archiwalne MT z poprzednimi odcinkami można kupić na www.ulubionykiosk.pl

Raspberry Pi (13)
PWM dla serw

Sterowanie niektórymi elementami elektronicznymi wymaga wygenerowania serii impulsów o specyficznym wyglądzie. Kształt przebiegu, jego częstotliwość, stosunek stanu wysokiego do niskiego – wszystkie te parametry muszą być odpowiednio dobrane, aby uzyskać pożądaną reakcję. Jakość generowanych impulsów (regularność, dokładność) stanowi o stabilności odpowiedzi. W dzisiejszym odcinku zajmiemy się modulacją szerokości impulsów – czyli PWM (ang. *Pulse Width Modulation*). Zrozumienie jej pozwoli Wam na sterowanie m.in. serwami modelarskimi.

Modulacja szerokości impulsu PWM jest jedną z podstawowych metod sterowania elementami elektronicznymi. Stosuje się ją do kontrolowania serw modelarskich, silników czy taśm LED. W praktyce sprowadza się do wygenerowania prostokątnego przebiegu elektrycznego o stałej częstotliwości i amplitudzie. Częstotliwość to liczba powtórzeń, cykli w danej jednostce czasu. W układzie SI jej jednostką jest herc – w skrócie Hz (mnożniki to np. 1000 Hz = 1 kiloherc, 1 kHz; 1 MHz = 1000 kHz, 1 megaherc). Układ pracuje z częstotliwością 1 Hz, jeżeli w czasie 1 s pojawia się jeden cykl. Zauważcie, że przy częstotliwości 50 Hz, 1 cykl trwa 20 ms (tzn. w ciągu sekundy występuje 50 cykli). Dla PWM stała jest też amplituda, czyli różnica między poziomem odniesienia i wartościami szczytowymi. W praktyce taki sygnał sterujący zmienia się między poziomem „0" – masą – a napięciem zasilania lub takim, które komunikujące układy interpretują jako logiczną „1" – np. ok. 3,3 V dla Raspberry (**ilustracja 1**; [1]).



1. PWM: częstotliwość, amplituda i stopień wypełnienia

Skoro te dwie wielkości – częstotliwość i amplituda – mają pozostawać stałe, to co możemy zmieniać? Sterowanie odbywa się za pomocą tzw. stopnia wypełnienia impulsu. Stopień wypełnienia jest stosunkiem czasu trwania sygnału wysokiego do długości cyklu. Jeżeli cykl ma długość 20 ms (50 Hz), z czego sygnał wysoki trwa 10 ms (a niski przez kolejne 10 ms) – mówimy o wypełnieniu na poziomie 50%. Przy takiej częstotliwości 1 ms sygnał wysoki to 5% wypełnienia, 1,5 ms – 7,5%, a 2 ms – 10% (wrócimy do tych wartości za chwilę).

Serwa modelarskie (analogowe)

Analogowe serwa modelarskie zawierają niewielki silniczek elektryczny z przekładnią. Przekładnia ma za zadanie zwiększenie momentu obrotowego. Wał silnika obraca potencjometrem. Pozwala to wbudowanemu układowi sterującemu stwierdzić, w jakiej pozycji się znajduje i, uruchamiając silnik, zmieniać ją do zadanej przez użytkownika. Serwa mają trzy wyprowadzenia: masę (kabelek o kolorze czarnym lub brązowym), zasilanie (najczęściej w granicach 4,8-6 V, kabelek czerwony – środkowy) oraz sterowanie (kabelek biały lub pomarańczowy; zob. **ilustracja 2**).

Położenie ramienia serwa zmienia się, podając odpowiedni sygnał PWM. W zależności od parametrów sygnału, serwo może obracać się w lewo/prawo, ustawić w jedną z ustalonych pozycji skrajnych lub w neutralną. Pozycje skrajne to 0 i 180 stopni – ograniczone przez odpowiednie blokady. Pozycja neutralna to 90 stopni. Jeżeli sygnał się nie zmienia (lub przestanie być dostarczany) – serwo pozostaje nieruchome.

Serwa oczekują sygnału PWM o częstotliwości 50 Hz (czyli cyklu o długości 20 ms). Układ sterujący próbkuje długość stanu wysokiego. Może ona przyjmować pewne charakterystyczne wartości, które układ sterujący zinterpretuje jako (**ilustracja 3**):

1 ms: ustaw serwo w pozycji skrajnej 0 stopni;

1,5 ms: ustaw serwo w pozycji neutralnej (90 stopni);

2 ms: ustaw serwo w pozycji skrajnej 180 stopni.

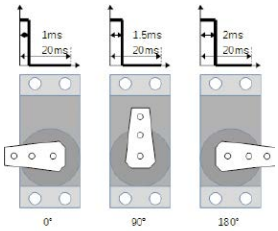
Wielkości te mogą różnić się dla poszczególnych modeli serw. Szczegółowe dane na ten temat znajdziecie na stronie **www.servodatabase.com/servos/all.** Dane zebrane doświadczalnie dla kilku wybranych serw przedstawiłem w **tabeli 1**.

Można również zauważyć, że serwo ustawi się w pozycji normalnej, dostając sygnał 1,5 ms co 10 ms (czyli wypełnienie 1,5 ms z 10 ms = 15%), ale i co 40 ms (czyli wypełnienie 1,5 ms z 40 ms = 3%). W tym sensie nie jest to więc klasyczny PWM, gdzie steruje się stopniem wypełnienia jako stosunkiem czasu trwania sygnału wysokiego do okresu. Dla odróżnienia tych dwóch sposobów sterowania dla serw używa się nawet terminu RC-PWM – chociaż w większości źródeł PWM i RC-PWM są błędnie utożsamiane.

2. Typowe serwa modelarskie: TowerPro Sg90, Redox S90 (9g) i TowerPro SG5010 (36g)

Tabela 1. Zmierzone czasy trwania sygnału wysokiego, wymuszające ustawienie serwa w pozycji 0, 90 i 180 stopni, dla 50 Hz

Serwo	Pozycja neutralna [ms]	Pozycja 0 [ms]	Pozycja 180 [ms]	Obrót w lewo	Obrót w prawo
TowerPro SG5010 (36g)	1,5	0,600	2,5	między pozycją 0 i neutralną	między pozycją neutralną i 180
TowerPro SG90 (9g)	1,5	0,600	2,5		
Redox S90 (9g)	1,4	0,600	2.2		

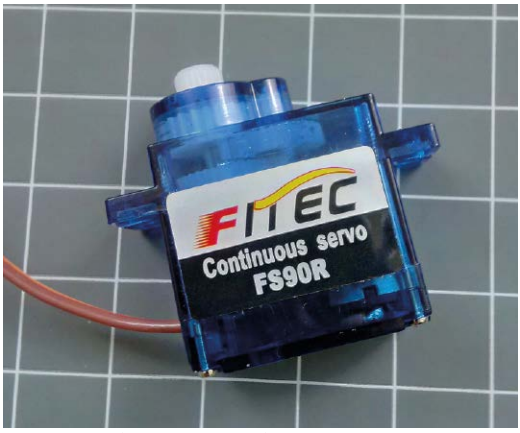


3. Ustawienie ramienia serwa w zależności od wypełnienia sygnału

Serwa 360 stopni

Trochę inną kategorię stanowią serwa tzw. 360 stopni. Nie mają one ograniczników i dzięki temu mogą się kręcić „w kółko". Sygnał sterujący neutralny (np. 1,5 ms) zatrzymuje serwo. Impulsy dłuższe lub krótsze niż neutralny zmieniają kierunek obrotów (lewo/prawo). Serwa tego typu stanowią bardzo atrakcyjną alternatywę dla silników DC (prądu stałego). Często używam ich do napędzania robotów mobilnych. Na rynku znajdziecie całkiem pokaźną ofertę modeli o różnych parametrach i rozmiarach, w cenach od 18 zł/sztukę (**ilustracja 4**).

Zamiast kupować gotowe serwo 360, możecie również sami zmodyfikować standardowe serwo. Jest kilka sposobów, żeby to osiągnąć. Najczęściej polegają one na usunięciu ograniczników i odłączeniu potencjometru (**ilustracja 5**). W jego miejsce wlutowuje się dwa rezystory. Przykład modyfikacji znajdziecie w [2]. **Uwaga:** takie modyfikacje oznaczają jednak utratę gwarancji i wykonujecie je na własną odpowiedzialność. Wymagają też trochę wprawy – są bardzo duże szanse, że pierwsza próba skończy się kompletnym zniszczeniem serwa.



4. Serwo 360: Feetech FS90R (9g)

W zależności od sposobu modyfikacji zazwyczaj dalej będziecie mogli sterować za pomocą sygnału PWM. Często jednak występują problemy z pozycją neutralną (czyli stop), a zakres wartości wypełnienia impulsu w tej pozycji ma znacząco mniejszą tolerancję. O ile Feetech FS90R jest nieruchome dla zakresu 1,43-1,51 ms – przerobione przeze mnie TowerProSG90 już jedynie 1,57-1,58 ms (długość sygnału wysokiego).

Generowanie sygnału PWM

W warunkach laboratoryjnych do generacji takich sygnałów można stosować urządzenia zwane **generatorami sygnału** (**ilustracja 6**). Są to narzędzia, które mogą wygenerować sygnały o bardzo różnych kształtach (prostokątny, sinusoidalny), zadanej częstotliwości, amplitudzie, przesunięciu, stopniu wypełnienia sygnału i innych parametrach (jak np. czas narastania). Odpowiednio dobierając parametry, możecie precyzyjnie wyznaczyć np. graniczne wartości wypełnienia sygnału dla różnych pozycji serwa.

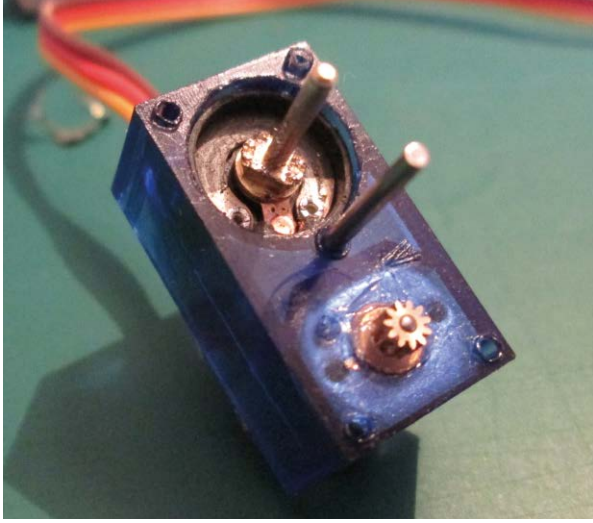
Niestety, są to urządzenia dość kosztowne (od 1 tys. zł). Tańszą alternatywę stanowią generatory w postaci płytek, np. w oparciu o projekt AVR DDS (**ilustracja 7**, zob [3]).

Na szczęście, do takiej pracy można również użyć Raspberry Pi. Wygenerowanie PWM o odpowiednich parametrach pozwoli mu na sterowanie różnymi urządzeniami.

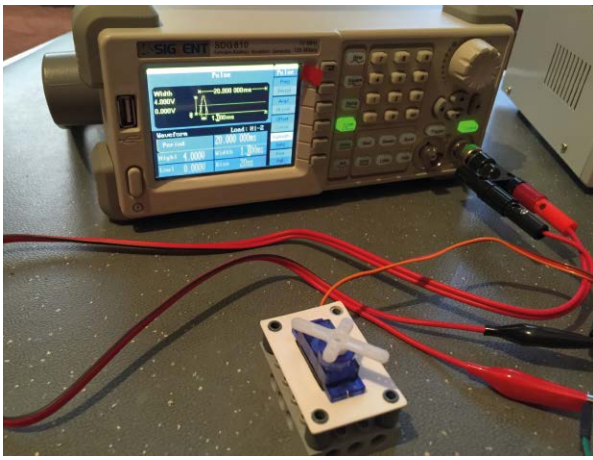
PWM z Raspberry: Python i RPi.GPIO

Skoro PWM jest sygnałem prostokątnym, wygenerowanie go za pomocą GPIO Raspberry nie powinno być większym problemem. Wydaje się, że można go uzyskać, odpowiednio zmieniając stan wybranego pinu między wysokim a niskim. Metoda ta nazywana jest *bit-banging*. Spróbujmy wygenerować PWM za pomocą prostego programu w Pythonie z użyciem biblioteki RPi.GPIO (dostarczanej razem z ostatnimi wersjami Raspbiana):

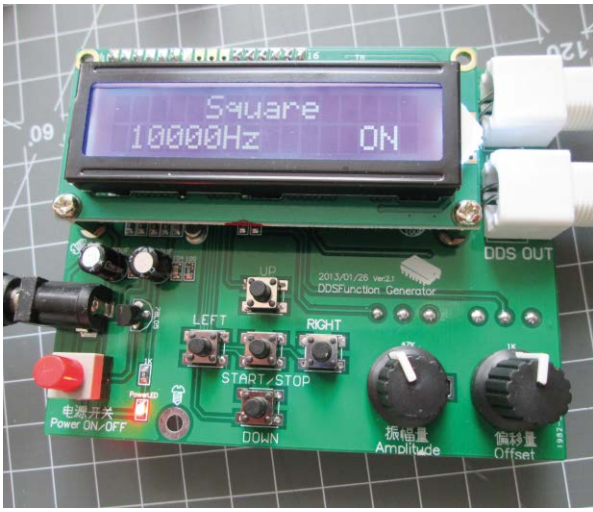
```
  $ nano test_bang.py
  import RPi.GPIO as GPIO
  import time

  #Sygnal ma byc wygenerowany na GPIO17,
  #fizyczny pin 11
  PWM_PIN = 17
  #Stan wysoki: 2ms, cykl: 20ms
  PWM_UP = 2.
  PWM_FRQ = 20.
  #ustawiamy wyjscia
  GPIO.setmode(GPIO.BCM)
  GPIO.setup(PWM_PIN, GPIO.OUT)
  print „Generuje... (przycisnij
[CTRL]+[C] zeby skonczyc)”
  try:
          while True:
                  #Pin ustawiamy w stan
wysoki
```



5. Rozmontowane TowerPro SG90 – usunięte potencjometr i ograniczniki



6. Badanie zakresów serwa za pomocą generatora Siglent SDG810



7. Prosty generator sygnałów w oparciu o projekt AVR DDS

```
                    GPIO.output(PWM_PIN,
GPIO.HIGH)
                    #Czekamy 2ms; instruk-
cja sleep pobiera czas w [s] - stad
dzielenie
                    time.sleep(PWM_UP/1000.)
                    #Pin ustawiamy w stan
niski
                    GPIO.output(PWM_PIN,
GPIO.LOW)
                    #Czekamy 20-2ms
                    time.sleep((PWM_FRQ
- PWM_UP)/1000.)
  #Elegancko konczymy po wcisnieciu
[CTRL]+[C]
  except KeyboardInterrupt:
          GPIO.cleanup()
  print "Koniec."
```

Uruchomcie skrypt:

```
$ sudo python test_bang.py
```

Powyższy kod zmienia sygnał na pinie GPIO17 (fizyczny pin 11) zgodnie z parametrami zadanymi w zmiennych PWM_UP i PWM_FRQ. Sprawdźmy na oscyloskopie efekty jego działania (**ilustracja 8**).

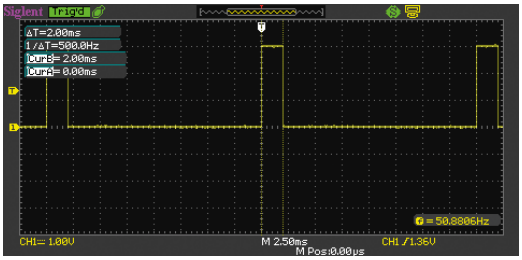
Niestety, w praktyce metoda – choć tak łatwa i przejrzysta – nie da zadowalających rezultatów. Zajęty poważniejszymi zadaniami, procesor Raspberry zepchnie wykonanie poleceń naszego skryptu na dalszy plan. W rezultacie generacja sygnału może zostać poważnie zakłócona. Sprawdźcie to sami. Stwórzcie drugi skrypt, który trochę zajmie procesor:

```
  $ nano work_me.py
  from random import randint
  while True:
          z = randint(1,2000)
          print z
```

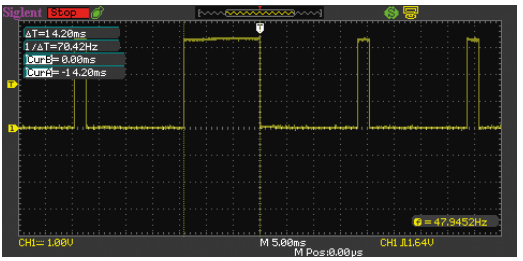
Uruchomcie teraz obydwa skrypty:

```
  $ sudo python work_me.py &
  $ sudo python test_bang.py
```

Nawet odrobinę obciążony procesor spowoduje, że otrzymacie sygnał o znacznie mniejszej regularności. Długość impulsu wysokiego w niektórych momentach może nawet przekroczyć 10 ms (**ilustracja 9**). Wyobraźcie sobie, jak będzie on wyglądał, gdy dodatkowo uruchomicie inne aplikacje czy np. środowisko graficzne.



8. Sygnał PWM wygenerowany za pomocą Pythona



9. Generowany programowo sygnał PWM z Raspberry może ulec zakłóceniu

Oczywiście nie zawsze musi to być wielki problem. W niektórych zastosowaniach takie zakłócenia nie wpłyną znacząco na sterowany obiekt. Jeżeli jednak potrzebujecie precyzyjnego sygnału, jak przy sterowaniu serwami, efekty tych niedokładności mogą być bardzo dokuczliwe.

Spójrzmy jeszcze dokładniej na samą bibliotekę RPi.GPIO. Oferuje ona bardziej „elegancki" sposób generowania PWM poprzez klasę GPIO.PWM:

```
  import RPi.GPIO as GPIO
  import time
  #Tym razem na pinie GPIO18, fizyczny 12
  PWM_PIN = 18
  #Korzystamy ze schematu BCM, ustawiamy
pin jako wyjscie
  GPIO.setmode(GPIO.BCM)
  GPIO.setup(PWM_PIN, GPIO.OUT)
  #Tworzymy obiekt klasy PWM; ustawiamy
50Hz (co 20ms)
  pwm = GPIO.PWM(PWM_PIN, 50)
  #Inicjalizacja, wypelnienie 10% czyli
2ms (20ms/dc)
  pwm.start(10)

  print „Generuje... (przycisnij
[CTRL]+[C] zeby skonczyc)"
  try:
          while 1:
                  pass
  except KeyboardInterrupt:
          pwm.stop()
          GPIO.cleanup()
  print „Koniec."
```

Ten program (podobnie jak poprzedni) będzie generował impulsy do momentu, gdy nie wciśniecie [CTRL]+[C]. Niestety, nie rozwiązuje on podstawowego problemu stabilności sygnału PWM. Zgodnie z notatkami na stronie projektu **https://pypi.python.org/pypi/RPi.GPIO** – *although hardware PWM is not available yet, software PWM is available to use on all channels*. Dowiadujemy się, że PWM możemy używać na dowolnym, wybranym pinie. Czym jest jednak owo brakujące *hardware PWM*?

Sprzętowe generowanie sygnału PWM polega na wykorzystaniu pewnych właściwości samego sprzętu (ang. *hardware*) do generowania

odpowiednich impulsów. W odróżnieniu od realizacji opartych wyłącznie na oprogramowaniu, ten sposób nie obciąża samego procesora, jest realizowany przez dodatkowe układy. Nie będzie więc zakłócany działaniem systemu operacyjnego. Jak się zorientowaliście, RPi.GPIO nie oferuje takiej opcji. Ale są biblioteki, dla których nie jest to problemem. Oczywiście wymaga to pewnych dodatkowych zabiegów.

Lepsze PWM: wiringPi2 i C

Powyższe przykłady opierały się na generowaniu PWM za pomocą oprogramowania. SoC napędzający Raspberry Pi model A/B ma również dwa sprzętowe generatory PWM, z których jeden – PWM0 – można podłączyć do GPIO18 (fizyczny pin 12). Generator ten jest również używany do tworzenia dźwięku na wyjściu 3,5 mm audio.

Sprzętowe generowanie PWM może być wykorzystane za pomocą popularnej biblioteki *wiringPi*. Jej autorem jest Gordon „Drogon" Henderson. Instalacja wymaga ściągnięcia i przekompilowania kodu źródłowego z *github*'a:

```
$ git clone git://git.drogon.net/wiringPi
$ cd wiringPi
$ sudo ./build
```

Biblioteka jest dostarczana z zestawem poręcznych narzędzi wywoływanych z linii komend:

```
$ gpio -v
gpio version: 2.26
Copyright (c) 2012-2015 Gordon Henderson
This is free software with ABSOLUTELY NO WARRANTY.
...
$ gpio readall
...
```

Możecie również zainstalować dodatki pozwalające używać jej z poziomu Pythona (wymaga rozszerzenia pip):

```
$ sudo apt-get install python-dev python-pip
$ sudo pip install wiringpi2
```

Napiszmy teraz program w C. Sygnał sterujący wygenerujemy na GPIO18 (fizyczny pin 12) – to właśnie na niego można przełączyć wewnętrzny generator PWM:

```
$ nano pwm.c

#include <wiringPi.h>
int main (void){
      if (wiringPiSetupGpio() != -1){
            pinMode(18,PWM_OUTPUT);
            //Wyjasnienia ponizej

            pwmSetMode(PWM_MODE_MS);
            pwmSetRange (1000);
            pwmSetClock(384);
            pwmWrite (18, 500);
      }else{
            return 1;
      }
      return 0;
}
```

Zwróćcie uwagę na kilka szczegółów ([3]):

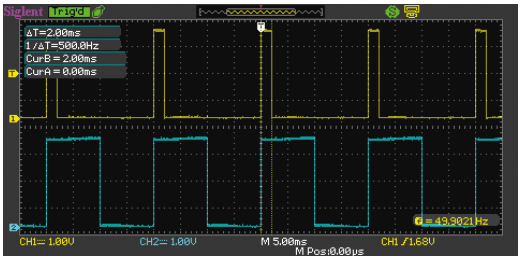
- instrukcja *pwmSetMode()* ustawia tryb pracy generatora w bardziej przewidywalny i niezależny od wypełnienia „mark:space";
- instrukcja *wiringPiSetupGpio()* inicjuje bibliotekę *wiringPi2*. Zakłada ona, że piny będą adresowane jak dla BCM (GPIOxx). Możecie również zastosować funkcję *wiringPiSetupPhys()*, ale wtedy używajcie numerów fizycznych pinów w złączu GPIO (np. 3,3v to 1, GND to 6). Kolejna funkcja *wiringPiSetup()* wymusza posługiwanie się numeracją specyficzną dla tej biblioteki – innej niż BCM i fizyczna. Upewnijcie się, której numeracji chcecie używać;
- instrukcja *pwmSetRange()* ustala liczbę przedziałów w jednym cyklu; dla wygody programowania wybrałem 1000 przedziałów. Dla 50 Hz (cykl 20 ms) oznacza to, że każdy przedział będzie miał szerokość 20 uS (mikrosekund – 20 ms/1000). Przykładowy 1,5 ms sygnał wymaga więc 75 przedziałów (1,5 ms = 75*20 us) – stąd *pwmWrite(18,75)* – „18" to numer GPIO. Żeby zapewnić taką rozdzielczość, generator układu PWM musi działać z częstotliwością 1/0.02 = 50 kHz;
- instrukcja *pwmSetClock()* określa dzielnik częstotliwości generatora PWM; dla Raspberry działa on z częstotliwością 19,2 MHz; żeby uzyskać 50 kHz, musimy podzielić ją na 384 (19,2 MHz/384 = 50 kHz);

Nasz nowy kod trzeba teraz skompilować i można go uruchomić:

```
$ gcc -L/usr/local/lib  pwm.c  -lwiringPi -lm -o pwm
$ sudo ./pwm
```

Taki sygnał będzie stabilny, ponieważ pochodzi z przeznaczonego do tego układu. Oczywiście jeden sprzętowy PWM to raczej niewiele. Nowsze modele A+/B+/Pi 2 mogą wyprowadzić dodatkowy PWM1 na piny GPIO13 i jego kopię na GPIO19 (fizyczne 33 i 35 z rozszerzonego zestawu). Oprócz GPIO18, kopię sygnału PWM0 znajdziecie na GPIO12 – fizycznym pinie 32.

```
$ nano pwm2.c

#include <wiringPi.h>
int main (void)
{
        if (wiringPiSetupGpio() != -1)
{
                pinMode(18,
PWM_OUTPUT);
                pinMode(19,
PWM_OUTPUT);
```



10. Dwa niezależne sygnały PWM generowane przez Raspberry

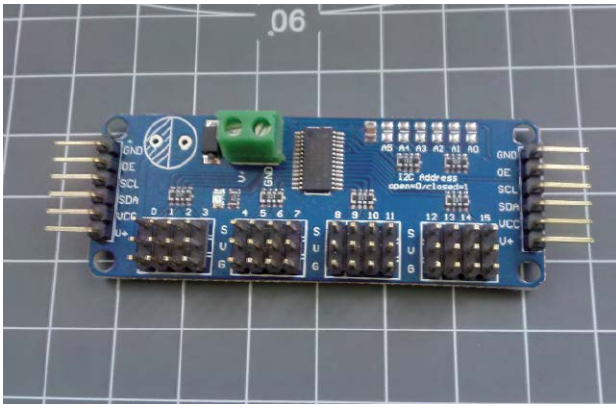
```
pwmSetMode(PWM_MODE_MS);
                    pwmSetRange (1000);
                    pwmSetClock(384);
                    //2ms - pozycja 180st
                    pwmWrite (18, 100);
                    pwmWrite (19, 500);
            }else{
                  return 1;
          }
          return 0;
    }
```

W efekcie uzyskacie dwa różne sygnały na pinach GPIO18 i GPIO19 (**ilustracja 10**; oczywiście sygnał z wypełnieniem 50% jest kompletnie nieprzydatny do sterowania serwami – za to wyraźnie widać różnicę). Zauważcie, że są zsynchronizowane.

Dodatkowe piny zostaną uruchomione, gdy je ustawicie *pinMode()* i zapiszecie *pwmWrite()*. Jeżeli ustawicie GPIO18 – kopię sygnału znajdziecie na pinie GPIO12 lub odwrotnie (podobnie dla GPIO13 i GPIO19).

Rozszerzenia do generacji sygnału

Posiadanie jednego pinu (modele A/B) lub nawet dwóch (A+/B+/Pi2) do generacji sprzętowego sygnału PWM jest dość poważnym ograniczeniem. Można jednak wzbogacić naszą Raspberry o dodatkowe rozszerzenia, które skutecznie zwiększą liczbę podłączanych serw. Przykładowo, taki moduł może być oparty o układ PCA9685 (**ilustracja 11**, karta katalogowa [5]). Płytki o niego oparte oferują nawet szesnaście niezależnych kanałów PWM. Na każdym z nich można ustawiać częstotliwość 40-1000 Hz z dowolnym stopniem wypełnienia sygnału. Co jeszcze ciekawsze – rozszerzenie to jest sterowane przez interfejs i2c.



11. Rozszerzenie generujące sygnały PWM oparte na PCA9685

Oznacza to, że wystarczą dwie linie sygnałowe, żeby kontrolować nawet szesnaście serw!

Instalacja tego rozszerzenia wymaga następujących kroków (dla jądra Linuksa 3.18+):

- włączcie obsługę i2c: używając programu raspi-config (polecenie: *$ sudo raspi-config*). Odpowiednią opcję znajdziecie w menu *Advanced settings->I2C* (wymagany restart);
- do pliku /etc/modules dodajcie wpis: *i2c_dev*;
- podłączcie SDA (fizyczny pin 3) oraz SCL (fizyczny pin 5) do odpowiednich pinów płytki rozszerzenia;
- podłączcie masę Raspberry (np. fizyczny pin 6) oraz zasilanie 3,3 V (fizyczny pin 1) do płytki rozszerzenia;
- zainstalujcie narzędzia i2c poleceniem: *$ sudo apt-get i2c-tools*;
- zainstalujcie bibliotekę do obsługi i2c pod Pythonem: *$ sudo apt-get install python-smbus*;

Po podłączeniu rozszerzenia sprawdźcie, czy jest widziane przez Raspberry:

```
  $ i2cdetect -y 1
  0 1 2 3 4 5 6 7 8 9 a b c d e f
00: -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- --
40: 40 -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- --
70: 70 -- -- -- -- -- -- --
```

Tabela 2. Wybrane biblioteki generujące sygnał PWM

Biblioteka	Adres projektu	Wsparcie sprzętowe	Uwagi
Rpi.GPIO	https://pypi.python.org/pypi/RPi.GPIO	nie	wykorzystana w tekście, domyślnie dostarczana z Raspbianem
WiringPi2	http://wiringpi.com/	tak	wykorzystana w tekście
PigPio	http://abyz.co.uk/rpi/pigpio/	tak	więcej: [4]
RPIO.PWM	https://pythonhosted.org/RPIO/rpio_py.html	tak	rozszerzenie RPi.GPIO

Adresy 0x40 i 0x70 należą do naszego rozszerzenia. Jeżeli uzyskaliście na konsoli wydruk jak poniżej – możecie zabrać się do pobierania przykładów. Ja użyłem dostarczonego przez Adafruit.com (na podstawie [6]):

```
$ git clone https://github.com/adafruit/Adafruit-Raspberry-Pi-Python-Code.git
$ cd Adafruit-Raspberry-Pi-Python-Code
$ cd Adafruit_PWM_Servo_Driver
```

Spójrzcie na przykład *ServoExample.py*. Zawiera właściwie wszystko, co potrzeba; w skrócie:

```
from Adafruit_PWM_Servo_Driver import PWM
import time
#Rozszerzenie znajduje sie na adresie 0x40
pwm = PWM(0x40)
#Wybieramy znana czestotliwosc 50Hz
pwm.setPWMFreq(50)
#ustaw sygnał o na wyjsciu 0 rozszerzenia
pwm.setPWM(0, 0, 307)
```

Ostatnia instrukcja wymaga wyjaśnień, choć działa bardzo podobnie do opisywanej poprzednio *wiringPi2*. Biblioteka dzieli ustawiony częstotliwością okres (tu: 20 ms) na 4096 przedziałów (podobnie jak *pwmSetRange()* z wiringPi2) – tu każdy z nich ma długość ok. 5 us. Stąd sygnał pozycji neutralnej 1,5 ms to ok. 307 przedziałów.

Jeżeli jednak spróbujecie ustawić taką liczbę przedziałów, okaże się, że serwo... nadal się porusza. Nic nie zrobiliście źle – zwłaszcza jeżeli porównacie wersje kodu dla Raspberry i Arduino. Okaże się, że wybrana częstotliwość dla Arduino jest skalowana przez 0,9. Jak podają źródła, stała ta ma skompensować niedokładność zegara rozszerzenia (25 MHz; zob. [8]). W kodzie dla Raspberry brakuje tego skalowania (pobrany: maj 2015 r.). W rezultacie zamiast 50 Hz biblioteka wygeneruje ok. 56 Hz – a więc pozycja neutralna będzie wymagała 343 przedziałów. Zgodnie ze źródłem [7], możecie dodać to skalowanie w pliku „Adafruit_PWM_Servo_Driver.py", linia 59:

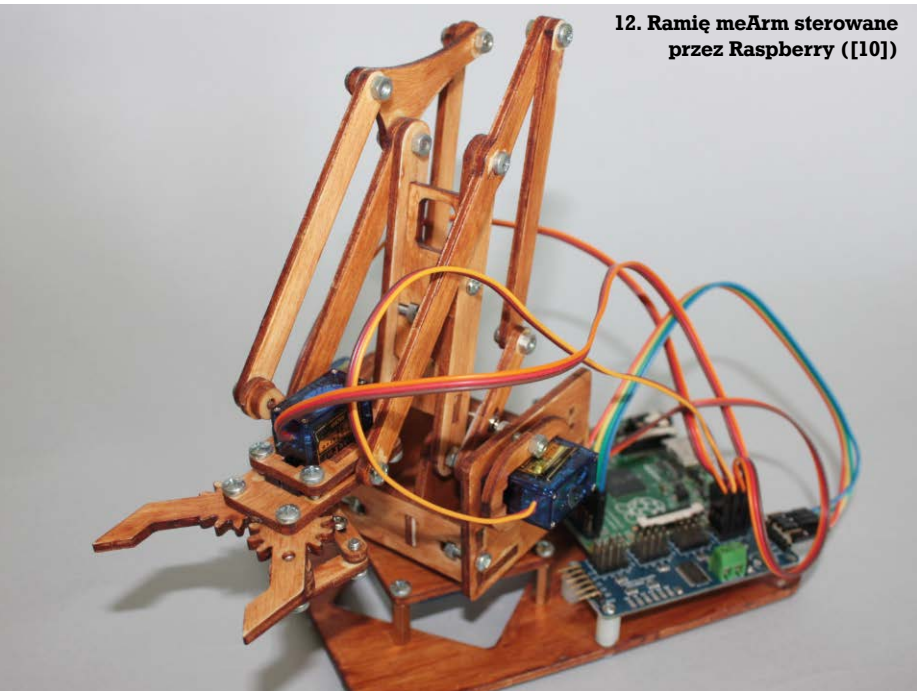
```
$ sudo nano Adafruit_PWM_Servo_Driver.py
    def setPWMFreq(self, freq):
      #Dodaj:
      freq *= 0.9
      "Sets the PWM frequency"
      prescaleval = 25000000.0    # 25MHz
      prescaleval /= 4096.0       # 12-bit
      prescaleval /= float(freq)
```

Dzięki tej zmianie odwzorowanie częstotliwości będzie znacznie dokładniejsze.

Jeżeli już opanujecie szczegóły obsługi tego rozszerzenia, możecie go wykorzystać do sterowania np. ramieniem MeArm (**ilustracja 12** – więcej o MeArm: [9], [10]). Ramię wymaga kontrolowania aż czterech serw.

A może Arduino?

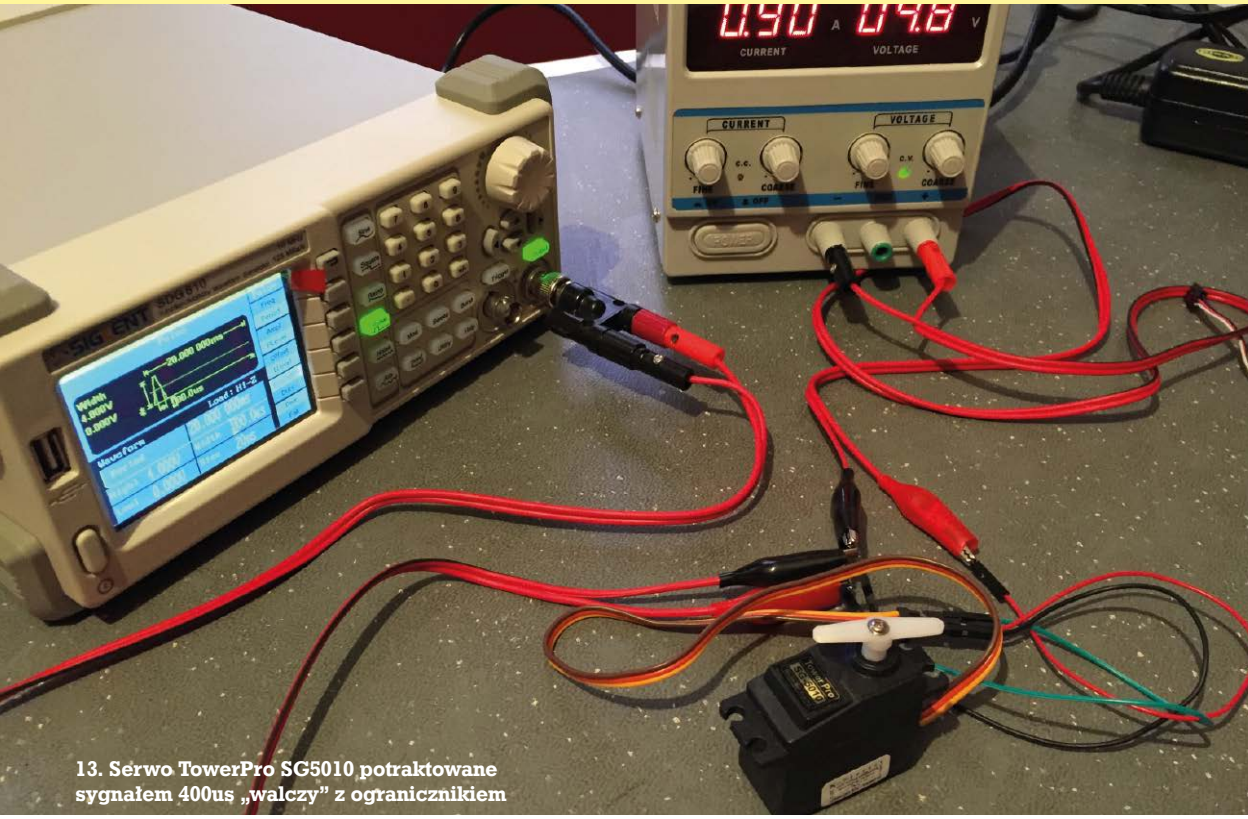
Kolejną opcją jest podłączenie do Raspberry Arduino. Możliwości Arduino w zakresie generacji PWM są znacznie większe. I nie jest obciążony bagażem systemu operacyjnego! Spróbujcie użyć do tego nawet niewielkie Arduino Nano. W odróżnieniu do Raspberry, sprzętowy sygnał PWM wygeneruje niezależnie aż na sześciu pinach. Podłączenie do Raspberry? Wystarczy zwykły kabel USB-miniUSB. Jedną końcówkę wkładacie do wolnego portu na Raspberry, a drugą – do gniazda na Nano. Raspberry zapewni zasilanie. Pozostaje wymyślić prosty protokół szeregowy, za pomocą którego będziecie instruowali Arduino, jaki sygnał i na jakim pinie wygeneruje. Oczywiście nic nie stoi na przeszkodzie, żeby połączyć oba układy bezprzewodowo – np. za pomocą modułów NRF24L01 czy WiFi przez ESP8266.



12. Ramię meArm sterowane przez Raspberry ([10])

Słowo o mocy...

Powyższe rozważania skupiły się tylko i wyłącznie na logicznych aspektach sterowania. Oczywiście serwa będą również potrzebowały odpowiedniego napięcia zasilania (zwykle 4,8-6 V) oraz



13. Serwo TowerPro SG5010 potraktowane sygnałem 400us „walczy" z ogranicznikiem

prądu. Niestety, parametry poboru prądu są rzadko wymieniane w instrukcjach. Małe serwa, żeby przemieścić ramię przy pracy bez obciążenia, potrzebują ok. 150 mA – większe 250 mA i więcej.

Jeżeli chcecie, żeby ramię serwa pokonało jakieś siły (co ma miejsce w praktyce) – zapotrzebowanie na prąd wzrośnie razem z obciążeniem – wartości 2 A dla serw 36g wcale nie są rzadkością (w zależności od rodzaju serwa)!

Podobna sytuacja może się również zdarzyć, gdy podacie sygnał niewiele poza granicami położeń skrajnych. Serwo będzie wtedy walczyć z ogranicznikami i zacznie pobierać duży prąd (**ilustracja 13**).

Oczywiście sama Raspberry nie jest w stanie sprostać takim wymaganiom prądowym. Przykłady, gdzie serwa (tylko i wyłącznie te najmniejsze i bez obciążenia!) zasilane są bezpośrednio z pinów 5 V Raspberry, mają zastosowanie jedynie edukacyjne. W praktycznych sytuacjach będziecie musieli użyć zewnętrznego zasilania o odpowiednich parametrach. Z pomocą mogą Wam tu przyjść np. modelarskie akumulatory LiPo lub LiIon. Te ostatnie występują nawet w bardzo wygodnym rozmiarze AA (np. 14 500). Oczywiście będziecie musieli jakoś ustabilizować ich napięcie (1 naładowana cela to nawet ponad 4 V) – ale odpłacą Wam się dużym prądem, a w rezultacie stosunkowo długim czasem pracy.

Podsumowanie

PWM to jeden z podstawowych typów sygnału sterującego. W ten sposób kontroluje się nie tylko serwomechanizmy, ale również silniki DC, taśmy LED i inne. Rozumiejąc podstawy, będziecie mogli swobodnie modyfikować przedstawiony w tekście kod źródłowy, ulepszać go i wykorzystywać w swoich projektach. ■

Arkadiusz Merta

Źródła:
[1] http://www.mikrokontrolery.org/artykuly/elektronika/130-pwm-modulacja-szerokoci-impulsu
[2] http://uczymy.edu.pl/wp/blog/2015/03/15/modyfikacja-serwa-towerpro-sg90-na-360st-cz-1
[3] http://www.scienceprog.com/avr-dds-signal-generator-v20/
[4] http://www.mikrokontroler.pl/content/pigpio-sposob-na-wielokanalowy-pwm-w-raspberry-pi
[5] http://www.nxp.com/documents/data_sheet/PCA9685.pdf
[6] https://github.com/adafruit/Adafruit-Raspberry-Pi-Python-Code
[7] https://forums.adafruit.com/viewtopic.php?f=19&t=72554
[8] https://github.com/adafruit/Adafruit-PWM-Servo-Driver-Library/issues/11
[9] http://www.phenoptix.com/products/mearm-pocket-sized-robot-arm
[10] http://uczymy.edu.pl/wp/mearm-dziennik-budowy/