TECHNOLOGY IN ACTION™

Apress®

Arduino

dla zaawansowanych

Rick Anderson, Dan Cervo

Helion

Tytuł oryginału: Pro Arduino

ISBN: 978-83-246-8222-5

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: http://helion.pl (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/arduza.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/arduza
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

ROZDZIAŁ 6

Symulacja czujników

Układy Arduino mogą być użyte do symulowania czujników podłączonych do innych płyt Arduino lub urządzeń współpracujących z czujnikami. Symulowanie czujników umożliwia generowanie określonych i powtarzalnych danych, które mogą być użyte do testowania systemów i usuwania błędów, jak również do badania czujników, które nie są aktualnie dostępne. W tym rozdziale w większym stopniu skupiliśmy się na koncepcjach podłączenia różnego typu czujników niż na przesyłanych danych. Na potrzeby opisywanych zagadnień dane są odpowiednio uproszczone, niemniej jednak są wysyłane w taki sam sposób jak przez prawdziwe czujniki. Aby lepiej zademonstrować, jak Arduino może bezpośrednio symulować różne czujniki, każdy opisany przykład zawiera kod odczytujący dane z różnego typu interfejsów. Kody pochodzą z różnych źródeł i nie zostały zmodyfikowane.

Celem koncepcji symulacji nie jest zastąpienie czujników. W przypadku małych projektów przygotowanie symulacji może zająć więcej czasu niż użycie rzeczywistych czujników. Techniki symulacji są przydatne w zastosowaniach wymagających kontrolowanego przetwarzania danych, takich jak budowa robotów, testowanie platform lub badanie działania czujników. Podstawowym celem tego rozdziału jest pomoc przy pokonywaniu przeszkód, jakie możesz napotkać podczas budowania systemów symulujących czujniki lub tworzenia wykorzystujących je skomplikowanych projektów.

Czujniki służą do zamiany różnych parametrów fizycznych na impulsy elektryczne, które mogą zostać odczytane przez systemy komputerowe. Temperatura, położenie lub skład chemiczny przedmiotów to przykłady cech fizycznych, które mogą być badane za pomocą czujników. W symulacji nie jest najważniejsze pełne odtworzenie pracy lub funkcjonalności czujników. Niemniej jednak dane muszą być wysyłane w tym samym momencie, w takiej samej kolejności i tą samą metodą, jak w prawdziwym czujniku. Dokumentacja dostarcza niezbędnych informacji o ważnych funkcjach czujnika (np. zakres danych, ich typ, rodzaj komunikacji). W tym rozdziale będą potrzebne dwie płyty kompatybilne z Arduino oparte na układzie ATmega 328P i dość szeroki zestaw podstawowych elementów do budowy prototypów projektów. Jedna z płyt Arduino zostanie użyta do odczytywania czujnika, a druga do jego symulacji. Dwie płyty mogą być zastosowane do zasymulowania szerokiego spektrum czujników, przy czym szkic kodu czujnika pozostanie niezależny od szkicu kodu czytnika. Dzięki temu symulacje będą najbardziej dokładne w kontekście synchronizacji czasowej urządzeń, a jeżeli symulowany czujnik zostanie zamieniony na prawdziwy, nie będzie wymagana modyfikacja szkicu kodu czujnika.

Czujniki analogowe

Jest bardzo wiele czujników mierzących np. temperaturę, ruch lub położenie. Tego rodzaju czujniki sterują w sposób ciągły napięciem wyjściowym, ściśle skorelowanym ze stanem czujnika. Dostarczana informacja może być odczytana przez analogowe piny Arduino. Dane analogowe można zasymulować za pomocą potencjometru, lecz ponieważ potencjometr jest sam w sobie czujnikiem, więc nie jest odpowiedni do zastosowań w automatycznym sterowania urządzeniami.

Arduino posiada wejścia analogowe, ale brakuje mu prawdziwych wyjść. Istnieją metody złagodzenia braku wyjść analogowych poprzez zastosowanie przetwornika cyfrowo-analogowego lub potencjometru cyfrowego. Są to doskonałe rozwiązania w przypadku systemów produkcyjnych, lecz rzadko można je znaleźć w zwykłym zestawie elementów. Przykłady w tej części rozdziału opisują sposób zbudowania dwóch różnych przetworników cyfrowo-analogowych do wygenerowania sygnału analogowego, wykorzystujących tylko rezystory i kondensatory. Pierwszy przykład dotyczy kodu na Arduino dla czujnika temperatury TMP35 firmy Analog Devices.

Czytnik czujnika analogowego

Listing 6.1 przedstawia kod czytnika analogowego, wykorzystany w obu przykładach. Kod musi być załadowany do płyty Arduino użytej jako czytnik. Drugie Arduino będzie użyte jako czujnik dostarczający sygnał analogowy. Kod z listingu 6.1 pochodzi z oryginalnego przykładu ze strony firmy LadyADA (pod adresem *www.ladyada.net/learn/sensors/tmp36.html*). Jego działanie nie zostało zmienione, zmodyfikowane są jedynie komentarze. Przykład dotyczy czujnika temperatury, ale koncepcja odczytu sygnału z pinu analogowego i korelacji obliczeń z sygnałem na wyjściu dotyczy również innych czujników analogowych. Kod z listingu odczytuje sygnał analogowy na pinie 0 i na monitorze portu szeregowego wyświetla przetworzone dane opisujące temperaturę.

Listing 6.1. *Kod czytnika czujnika temperatury LadyADA ze zmodyfikowanymi komentarzami*

```
int sensorPin = 0;

void setup() {
  Serial.begin(9600);
} // koniec void setup()

void loop() {
  int reading = analogRead(sensorPin);
  float voltage = reading * 5.0; // konwersja odczytu na wartość napięcia
  voltage /= 1024.0; // podzielenie odczytanego napięcia przez maksymalną rozdzielczość przetwornika ADC
  Serial.print(voltage); Serial.println(" V");
  float temperatureC = (voltage - 0.5) * 100 ; // zmniejszenie o 500 mV i przemnożenie
                                               // przez 100, by otrzymać stopnie Celsjusza
  Serial.print(temperatureC); Serial.println(" stopni C");
  float temperatureF = (temperatureC * 9.0 / 5.0) + 32.0; // zamiana stopni C na F
  Serial.print(temperatureF); Serial.println("stopni F");
  delay(1000);
} // koniec void loop()
```

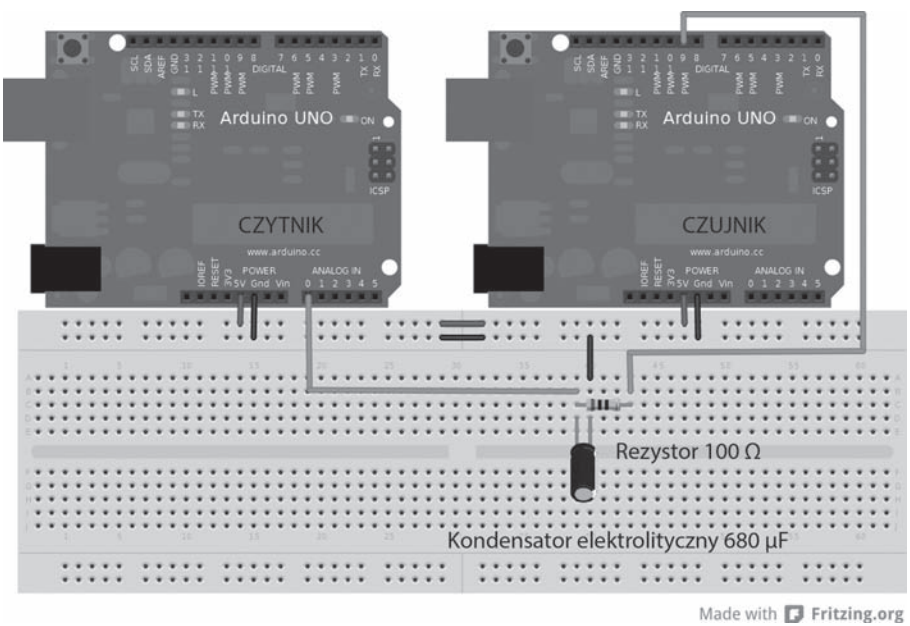
Filtr dolnoprzepustowy

Pierwszą metodą uzyskania sygnału analogowego jest zastosowanie filtru dolnoprzepustowego. Filtr zbudowany jest z rezystora i kondensatora połączonych szeregowo. Kondensator jest ładowany sygnałem modulacji PWN z Arduino i rozładowywany przez rezystor do wejścia analogowego płyty. Taka metoda konwersji sygnału

cyfrowego na analogowy sprawdza się, ponieważ pełne naładowanie kondensatora trwa pewien czas i poprzez kontrolę czasu utrzymania wysokiego stanu na wyjściu cyfrowym można uzyskać poziom napięcia stanowiący określony procent maksymalnego napięcia na wyjściu cyfrowym. Impuls PWM o poziomie wypełnienia 50% naładuje kondensator do 50% pojemności, co pozwoli uzyskać połowę maksymalnego napięcia. Jeżeli napięcie na pinie cyfrowym wynosi 5 V, wówczas uzyska się napięcie wyjściowe ok. 2,5 V.

Jeżeli w takiej konfiguracji zastosowany zostanie kondensator o zbyt małej pojemności, jego czas rozładowania będzie krótszy od czasu naładowania i nie będzie możliwe uzyskanie odpowiedniego napięcia na pinie analogowym. Z kolei kondensator o dużej pojemności spowoduje wydłużenie czasu spadku napięcia z poziomu przy pełnym naładowaniu. O ile pojemność kondensatora nie będzie zbyt mała, może być on użyty do symulowania wrażliwych czujników, szybko zmieniających napięcie. Do symulacji mniej wrażliwych czujników można zastosować nie tylko większy kondensator, ale również większy rezystor, przez co zmiany napięcia będą wolniejsze. Rezystor spowalnia rozładowanie kondensatora przez pin PWM. Aby nie obniżać całkowitego napięcia, należy zastosować niewielki rezystor. Opisana metoda nadaje się do konwersji sygnału cyfrowego na analogowy w przypadku, gdy dokładność nie jest istotna. W modulacji PWM dostępnych jest 256 poziomów (od 0 do 255), co przy napięciu 5 V daje ok. 0,019 – 0,2 V na jeden poziom. Filtr charakteryzuje się ponadto pewnym niewielkim rozrzutem napięcia, co zmniejsza dokładność. Rozrzut nie jest niekorzystnym zjawiskiem, szczególnie w konfiguracji, gdzie czujnik jest kontrolowany w pętli i wysyła sygnał bezpośrednio na wejście. Prawdziwy czujnik wysyłający sygnał analogowy może mieć pewną niedokładność, a więc czujnik symulowany będzie w takim przypadku lepszym przybliżeniem rzeczywistego.

Konfiguracja sprzętu jest przedstawiona na rysunku 6.1. Piny z napięciem 5 V i piny masy obu płyt Arduino są połączone ze sobą, dzięki czemu zasilana jest płyta symulująca czujnik. Taka konfiguracja zapewnia również komunikację pomiędzy oboma układami, ponieważ mają wspólną masę (taka sama konfiguracja występuje we wszystkich przykładach). Filtr dolnoprzepustowy składa się z kondensatora elektrolitycznego podłączonego biegunem ujemnym do masy, a dodatnim do wejścia analogowego czytnika. Pin 9 Arduino będącego czujnikiem jest dołączony do rezystora, którego druga strona jest dołączona do dodatniego bieguna kondensatora.



Rysunek 6.1. *Konfiguracja filtru dolnoprzepustowego*

Listing 6.2 przedstawia sposób sterowania napięciem wyjściowym. Wartość zmiennej typu *byte* jest modyfikowana i zapisywana na pinie nr 9. Wartość zmiennej *sensorOut* można zmieniać za pomocą poleceń odbieranych z portu szeregowego. Aby lepiej odwzorować działanie symulowanego czujnika, można obliczyć wartość z zakresu np. od 0 do 100°C.

Listing 6.2. *Kod dla płyty Arduino symulującej czujnik*

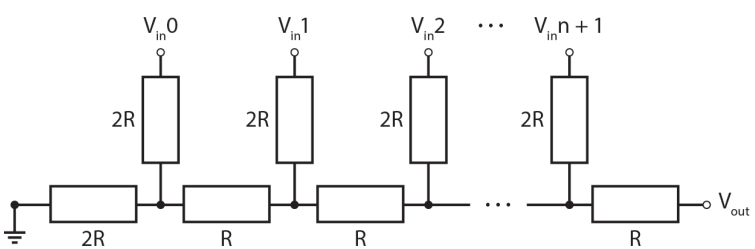
```
byte sensorOut = 0x00;

void setup() {
  pinMode(9,OUTPUT); // tu można skonfigurować port szeregowy
} // koniec void setup()

void loop() {
  sensorOut++; // przetwarzanie zmiennej wyjściowej
  analogWrite (9,sensorOut); // symulacja rzeczywistego czujnika
  delay(1000); // opóźnienie oddające prędkość odświeżania czujnika
} // koniec void loop()
```

Weryfikacja kodu

Po skonfigurowaniu układu i załadowaniu kodu podłącz port USB komputera do płyty Arduino będącej czytnikiem i otwórz monitor portu szeregowego. Czytnik będzie wyświetlał wartości napięcia na wejściu analogowym oraz temperaturę w skali Celsjusza i Fahrenheita. Arduino będące czujnikiem będzie podawać napięcie w zakresie od 0 V do ~5 V o skoku ~0,02 V, odpowiadające temperaturze od –50°C do 450°C z dokładnością ok. 2°C.

Drabinka rezystorowa

Drabinka rezystorowa jest inną metodą uzyskania sygnału analogowego za pomocą Arduino. Składa się z 20 rezystorów, z których 9 ma określoną rezystancję, a pozostałych 11 dwukrotnie większą. Drabinka jest w rzeczywistości układem dzielników napięcia. Działanie metody polega na sumowaniu wielu sygnałów cyfrowych w jeden, przy czym napięcie każdego z nich jest sukcesywnie zmieniane przez różne rezystory. Jest to metoda równoległego binarnego sterowania napięciem wyjściowym. Najmniej znaczący bit steruje sygnałem podawanym na rezystor najbliżej masy, natomiast najbardziej znaczący bit sygnałem na drugim końcu łańcucha rezystorów. Rysunek 6.2 przedstawia schemat drabinki rezystorowej, w której wejście $V_{in}0$ odpowiada najmniej znaczącemu bitowi i jego zmiana stanu na wysoki będzie wywoływać najmniejszą zmianę napięcia. Poprzez dodawanie kolejnych wejść $V_{in}n+1$ na końcu drabinki można ją rozbudować do dowolnej liczby bitów.

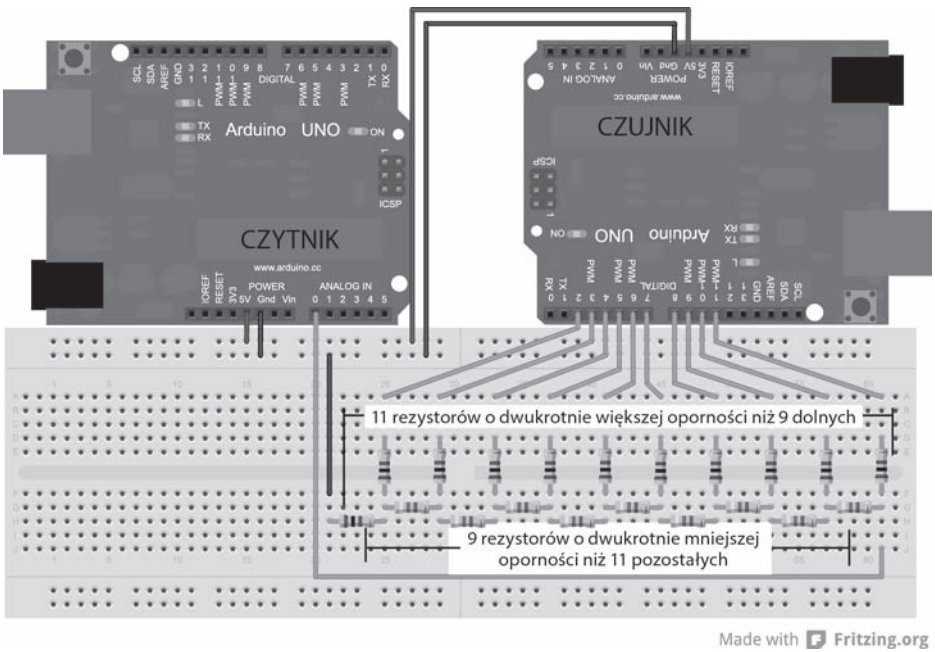


Rysunek 6.2. *Schemat drabinki rezystorowej*

Wartości rezystorów mogą być dowolne, o ile tylko jedna wartość będzie dwukrotnie większa od drugiej, przy czym nie może być zbyt duża, aby nie tłumiła napięcia wyjściowego. Odpowiednie wartości rezystancji wynoszą odpowiednio 1 kΩ oraz 470 Ω. Dobierając rezystory o dokładności 5%, można za pomocą dobrego multimetru uzyskać stosunek rezystancji 2:1. Drabinkę można wyskalować do dowolnej liczby bitów, dodając lub odejmując po dwa rezystory.

W tym przykładzie zastosowany jest 10-bitowy konwerter odpowiadający rozdzielczości przetwornika analogowo-cyfrowego płyty Arduino. Kod implementuje 10-bitowy licznik sterujący konwerterem. Wartości mniejszych rezystorów oznaczone są jako 1R, a większych jako 2R.

Budując układ, korzystaj z rysunku 6.3. Rozpocznij od rezystora 2R, podłączając go jednym końcem do masy, a drugim do innej łączówki na tej samej stronie płyty montażowej. Połącz za pomocą tej łączówki po tej samej stronie płyty dziewięć rezystorów 1R, a ostatni dołącz do pinu nr 0 płyty Arduino będącej czytnikiem. Końcówki każdego z dziesięciu pozostałych rezystorów 2R są z jednej strony dołączone do złączonych końcówek rezystorów 1R, a z drugiej do kolejnych pinów od 2 do 11 płyty Arduino będącej czujnikiem, licząc od rezystora 2R podłączonego najbliżej masy. Pozostałe dwa połączenia doprowadzone są do pinów 5 V i masy obu płyt Arduino. Kod programu czytnika jest taki sam jak dla filtru dolnoprzepustowego.



Rysunek 6.3. *Połączenie drabinki rezystorowej*

Kod implementuje podstawowy licznik binarny i jest wprowadzeniem do programowania rejestrów AVR Arduino. Użycie rejestrów może w niektórych przypadkach skrócić kod, ale również go skomplikować. Kod musi zmieniać wartości czterech rejestrów: *DDRB*, *DDRD*, *PORTB* oraz *PORTD*. Pierwsze litery oznaczają typ rejestru, natomiast ostatnia litera oznacza pin płyty Arduino. Wszystkie opisane porty mają 8-bitów (lub dwie części po 4 bity).

- Nazwa rejestru kończąca się na literę *D* oznacza piny od 0 do 7.
- Litera *B* oznacza piny od 8 do 13, natomiast dwa ostatnie bity są nieużywane.

Bity i piny są skorelowane ze sobą, począwszy od najmniej znaczącego bitu. Aby ustawić wysoki stan na pinie 0, należy do rejestru *PORTD* wpisać wartość *0b00000001*.

- *DDRx* oznacza rejestr kierunku danych, określający, czy pin ma być wejściem (0) czy wyjściem (1). Zmianę osiąga się przez wpisanie do rejestru bajtu danych, na przykład wartość *0b11100011* ustawia piny nr 7, 6, 5, 1 i 0 jako wyjścia, natomiast piny nr 4, 3 oraz 2 jako wejścia. Za pomocą tej metody piny konfiguruje się podobnie jak przy użyciu funkcji `pinMode(pin, kierunek)` wywoływanej dla każdego pinu w funkcji `setup()`. Jeżeli wymagane jest użycie funkcji szeregowych, dwa najmłodsze bity rejestru *xxxD* muszą być pominięte, przez co nie można wykorzystać całego bajtu.
- Wpisując bajt danych do rejestru *PORTx*, można w jednym wierszu kodu zmieniać stan całej grupy pinów na wysoki lub niski, zależnie od wpisanej wartości. Natomiast wpisując do zmiennej wartość *PORTx*, można odczytać zawartość rejestru. W zależności od ustawionego trybu rejestru *DDRx* można określić, jak zostały ustawione bity danych: wewnętrznie za pomocą funkcji `output(1)` lub zewnętrznie funkcją `input(0)`.

Do Arduino będącego czujnikiem załaduj kod z listingu 6.3. Kod ustawia za pomocą rejestru stan pinów od 2 do 11, tak samo jak za pomocą funkcji `pinMode()`. Następnie kod zwiększa wartość zmiennej typu *unsigned int* do liczby 1024. Za pomocą operatora *AND* przycinanych jest 10 bitów zmiennej, dzięki czemu licznik nie osiąga maksymalnej 16-bitowej wartości równej 65535. Wartość jest następnie przesuwana tak, aby poszczególne bity odpowiadały pinom. Niepotrzebne bity są zerowane za pomocą bitowego operatora *AND*, po czym dane są umieszczane w odpowiednim rejestrze. Ponieważ środowisko Arduino IDE wykorzystuje kompilator AVR języka C/C++, nie są wymagane dołączenia ani deklaracje nazw rejestrów. Po prostu wpisuje się je i używa tak samo jak każdej innej zmiennej.

■ **Uwaga** Bezpośrednie zmienianie wartości rejestrów jest zaawansowaną techniką programistyczną, która może nie dotyczyć wszystkich płyt kompatybilnych z Arduino. Sprawdź układ pinów danej płyty i opis rejestru w dokumentacji.

Listing 6.3. *Kod czujnika*

```
unsigned int manipVar=0; // jedyna zmienna potrzebna do wygenerowania danych

void setup() {
  DDRD = DDRD | 0b11111100; // ustawienie pinów 2 – 7 jako wyjścia lub pozostawienie  pinów 1, 2 wolnych
                            // na potrzeby komunikacji szeregowej
  DDRB = DDRB | 0b00001111; // ustawienie pinów 8 – 11 jako wyjścia, pozostawienie pozostałych wolnych
} // koniec void setup()

void loop() {
  manipVar++; // zmienną manipVar można dowolnie zmodyfikować
  manipVar &= 0b0000001111111111; // maska resetująca zmienną manipVar
                                  // gdy osiągnie wartość 1024
  PORTD = (manipVar << 2) & 0b11111100; //przesunięcie w lewo o 2 bity, następnie zamaskowanie
                                        // aby wyodrębnić piny 2 – 7 ze zmiennej manipVar,
                                        // następnie wpisanie bitów do pinów 2 – 7
  PORTB = (manipVar >> 6) & 0b00001111; // przesunięcie w prawo o 6 bitów
                                        // w celu ustawienia wartości pinów 8 – 11
  delay (1000); // uwzględnienie odświeżenia danych czujnika
} // koniec void loop()
```

Weryfikacja kodu

Po załadowaniu kodu do obu płyt Arduino i umieszczeniu elementów na płycie montażowej podłącz czytnik i otwórz terminal portu szeregowego. Wyświetlana będzie ta sama informacja, jak w poprzednim przykładzie, ale z dokładnością ok. 0,0048 V lub 0,5°C w tym samym zakresie temperatur.

W tej metodzie rozrzut jest mniejszy niż w przypadku filtru dolnoprzepustowego, jak również wykorzystana jest maksymalna rozdzielczość przetwornika analogowo-cyfrowego. Jest to lepsza metoda symulowania czujników analogowych. Wadą jest natomiast większa liczba wykorzystywanych pinów, użytych elementów i zastosowanie zaawansowanej techniki programowania do odliczania wartości. W przedstawionej konfiguracji odliczanie wartości od 0 do następnego 0 wprowadza dodatkowe opóźnienie ok. 4 ms, które formuje trójkątną falę o częstotliwości ok. 250 Hz z interwałem ok. 4 µs pomiędzy kolejnymi wartościami. Krótki kod pozwala utworzyć z Arduino prosty generator funkcyjny wykorzystujący tabelę bajtów. Można również symulować piezoelektryczne czujniki pukania.

■ **Uwaga** Aby lepiej poznać kod, zastąp drabinkę rezystorową serią diod LED, dołącz potencjometr do pinu analogowego nr 0. Następnie do zmiennej `manipVar` przypisz funkcję `analogRead(0)` i zmniejsz argument funkcji `delay` do 100. Włącz zasilanie i obserwuj konwersję sygnału z potencjometru na wartość binarną.

Czujniki cyfrowe

Pracując z czujnikami cyfrowymi, można odnieść wrażenie, że jest tyle sposobów obsługi, ile typów czujników. W tej części rozdziału opisany jest przekrój sposobów komunikacji. W symulacjach tych czujników ważna jest zgodność ze specyfikacjami różnych protokołów. Dane mogą być wysyłane lub odbierane, wysyłane w dowolnej kolejności do wielu urządzeń lub żądane w dowolnym momencie, przez co implementacja niektórych urządzeń może być trudna. Cenną pomocą przy wyborze najlepszej metody symulacji jest dokumentacja do czujników i układów Atmel.

Czujniki PWM

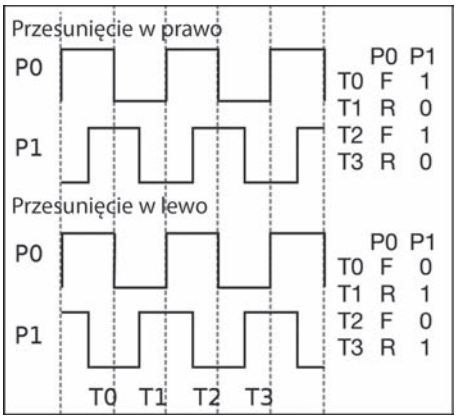
Czujniki PWM nie są tak popularne, jak inne typy czujników, ale wciąż zasługują na uznanie. Są powszechnie używane do sterowania silnikami. W pewnym sensie czujnik PWM zastępuje odbiornik RC, który też jest swego rodzaju czujnikiem. Chociaż w mikrokontrolerze Atmel brakuje niektórych elementów ściśle odpowiadających specyfikacji większości czujników wykorzystujących modulację PWM, niemniej jednak możliwy jest ich odczyt. Funkcja `pulseIn()` może odczytywać wygnał wyjściowy PWM z innego pinu z wystarczającą dokładnością umożliwiającą korelację danych. Kod, który można wykorzystać do symulowania czujników tego typu, jest podobny do przedstawionego w listingu 6.2. Jeżeli weźmiemy pod uwagę fakt, że nie ma czujników implementujących modulację PWM odpowiadającą tolerancji czasowej Arduino, okaże się, że nie znajdziemy przykładów do tej części rozdziału. Taki sposób przekazywania informacji cyfrowej może być przydatny do budowy innych układów czujników.

Kod Graya

Kod Graya jest metodą generowania fali prostokątnej na dwóch lub więcej pinach, przy czym fale te są przesunięte w fazie. Metoda ustawiania faz wielu sygnałów pozwala na odczyt w dowolnym momencie zmiany położenia lub kierunku poruszania się przedmiotów. Przesunięcie fazowe fal prostokątnych pozwala określić, czy bity są przesunięte w lewo, czy w prawo. Kod Graya jest również nazywany **cyklicznym kodem dwójkowym** i jest powszechnie używany w czujnikach zamieniających liniowy lub kątowy ruch elementów na impulsy umożliwiające określenie ich położenia oraz kierunku i prędkości przemieszczania. W ten sposób działa kółko przewijania w myszy komputerowej. Kod Graya jest szeroko stosowany w robotyce w obrotomierzach. Jeżeli

jedna wartość wyjściowa stanowi sygnał odniesienia z przyjętą zasadą narastającego lub opadającego zbocza, wówczas druga wartość wyjściowa odczytana w momencie zmiany sygnału pozwala określić kierunek obrotów. Jeżeli sygnał na drugim wyjściu jest niski przed odczytem sygnału na pierwszym wyjściu, oznacza to, że urządzenie porusza się w określonym kierunku, natomiast jeżeli sygnał jest wysoki — w przeciwnym.

W czujniku tego typu potrzebne są przynajmniej dwa piny dla danych i jeden do zasilania/masy. Im więcej pinów ma czujnik, tym jest dokładniejszy, dzięki możliwości wykrywania błędów i brakujących impulsów. Rysunek 6.4 przedstawia impulsy z dwuwyjściowego dekodera. Odczytywana jest zmiana stanu na wysoki lub niski pierwszego sygnału. Stan drugiego zależy od kierunku przesunięcia pierwszego sygnału w chwili odczytu.



Rysunek 6.4. *Impulsy dwuwyjściowego dekodera*

Śledzenie impulsów i wykrycie pełnego obrotu jest zadaniem czytnika lub kontrolera. Kod czytnika musi również rozpoznawać kierunek przesunięcia kodu Graya. Dr Ayars jest autorem artykułu na temat odczytu obrotomierza (SparkFun, nr COM-09117). W tym przykładzie kod zmniejsza lub zwiększa wartość zmiennej w zależności od kierunku obrotu dekodera w momencie minięcia zapadki, ale nie w zależności od liczby wykonanych obrotów. Więcej informacji na temat odczytu tego typu czujników można znaleźć na blogu dr. Ayarsa pod adresem *http://hacks.ayars.org/2009/12/using-quadrature-encoder-rotary-switch.html*.

Technika zastosowana w listingu 6.4 jest jedną z metod odczytu kodu Graya i doskonale nadaje się do dekoderów dwuwyjściowych. W przypadku dekoderów o trzech lub więcej wyjściach niezbędna jest bardziej zaawansowana metoda umożliwiająca korektę błędów odczytu i zliczania impulsów. W pierwszej części przykładu należy załadować poniższy kod do Arduino używanego jako czytnik.

Listing 6.4. *Kod dr Ayarsa ze zmienionymi komentarzami*

```
byte Blinker = 13;
int Delay = 250;
byte A = 2; // pin wyjściowy pierwszego czujnika
byte B = 3; // pin wyjściowy drugiego czujnika
volatile int Rotor = 0; // licznik kliknięć czujnika

void setup() {
  pinMode(Blinker, OUTPUT);
  pinMode(A, INPUT);
  pinMode(B, INPUT);
  digitalWrite(A, HIGH); // włączenie rezystorów podwyższających
  digitalWrite(B, HIGH);
  attachInterrupt(0, UpdateRotation, FALLING); // użycie przerwania na pinie A
```

```
  Serial.begin(9600);
} // koniec setup()

void loop() {
  digitalWrite(Blinker, HIGH); // miganie diody
  delay(Delay);                // tu może być wykonany dowolny kod, czujnik będzie odświeżony
  digitalWrite(Blinker, LOW); // po zgłoszeniu przerwania na pinie 2
  delay(Delay);
} // koniec loop()

void UpdateRotation() {
  // odświeżenie odczytu czujnika przy opadającym sygnale na pinie 2
  if (digitalRead(B)) {
    Rotor++; // zwiększenie kierunku, gdy w momencie zgłoszenia przerwania
  }          // na drugim pinie jest stan wysoki
  else {
    Rotor--; // zmniejszenie kierunku, gdy w momencie zgłoszenia przerwania
  }          // na drugim pinie jest stan niski
  Serial.println(Rotor, DEC);
} // koniec UpdateRotation()
```

Generowanie kodu Graya

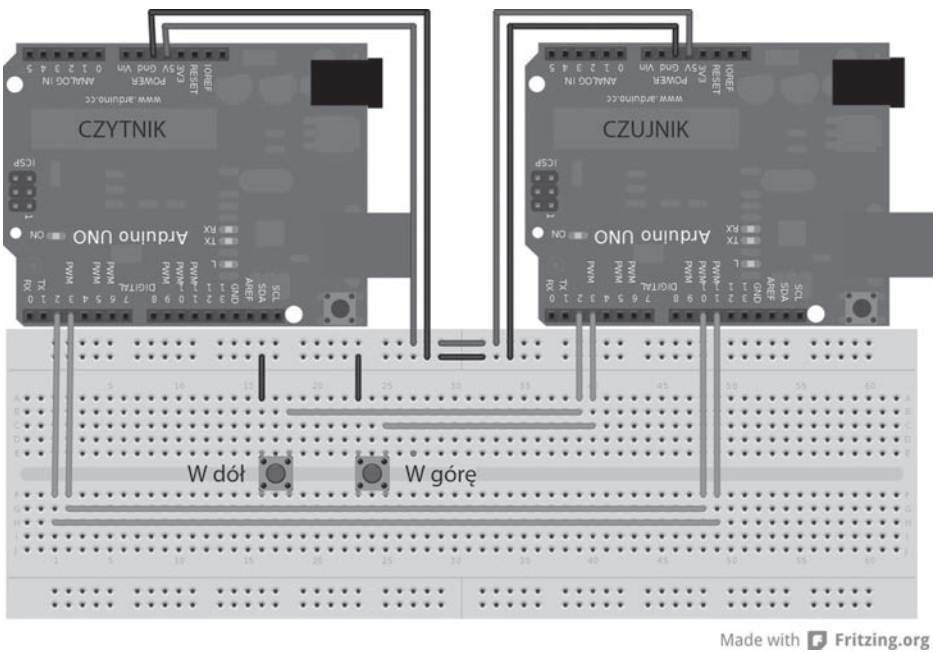
Aby Arduino mogło udawać czujnik wysyłający kod Graya, musi generować fale prostokątne, równomiernie przesunięte w fazie między sobą. Najlepszym sposobem sterowania serią sygnałów cyfrowych wysyłanych w określonej kolejności jest zastosowanie funkcji szeregowej `digitalWrite()` do ustawiania stanów pinów wyjściowych oraz funkcji `delay()` do kontroli fazy sygnału. Do sterowania każdym wyjściem symulowanego obrotomierza jest użyta osobna funkcja `digitalWrite()`, natomiast po każdym zapisie stanu jest użyta funkcja `delay()` przesuwająca sygnał cyfrowy. W obrotomierzu z dwoma wyjściami stany pinów zmieniane są w pętli. W każdym przebiegu pętli wywoływana jest dwukrotnie funkcja `digitalWrite()`, a po każdym zapisie funkcja `delay()`. W ten sposób jest generowana fala prostokątna, której okres jest dwukrotnie dłuższy od całkowitego opóźnienia. W każdym przebiegu pętli generowana jest jedna połowa kodu Graya. Kolejność zmian stanów pinów określa kierunek obrotów. Zmiana stanu pinów w kolejności od 1 do 3 oznacza jeden kierunek, natomiast od 3 do 1 oznacza kierunek przeciwny. Procentowe przesunięcie fazy sygnałów jest kontrolowane za pomocą funkcji `delay()` wywoływanej po funkcji `digitalWrite()`. W celu obliczenia różnicy fazy indywidualne opóźnienie wprowadzane za pomocą funkcji `delay()` jest podzielone przez całkowite opóźnienie. Jeżeli całkowite opóźnienie sygnału na dwóch wyjściach jest równe 6 ms, a indywidualne 3 ms, wówczas drugi sygnał jest przesunięty w fazie o 50%.

Niektóre obrotomierze generują sygnały przesunięte w fazie o 100%, czyli mające całkowicie przeciwne stany. Aby w obrotomierzu o czterech wyjściach trzeci sygnał był przeciwny do pierwszego, a przesunięcie fazowe było wciąż równomierne, pierwszy sygnał musi zmieniać stan w tej samej chwili co trzeci sygnał, natomiast czwarty sygnał nie może mieć opóźnienia. W ten sposób okres sygnału będzie równy 6 ms, a przesunięcie fazowe równe 1 ms. Wygenerowany okres obrotu będzie zależał od szybkości zmian sygnałów czujnika.

Aby obliczyć maksymalną częstotliwość symulowanych zmian, należy podzielić liczbę 60 przez całkowity okres obrotu i pomnożyć przez liczbę kroków na określonym dystansie. Dystans w odniesieniu do obrotów oznacza jeden obrót, natomiast odległość liniowa może być wyrażona w metrach lub innych jednostkach. W tym przykładzie symulacji obrotomierza przyjętych jest 12 kroków na jeden obrót. Najkrótszy czas jednego cyklu zaimplementowanego w czytniku jest równy 8 ms, co daje wynik 60 s / (0,008 s / krok × 12 kroków / obrót) = 625 obrotów na minutę. Czas wykonywania funkcji `digitalWrite()` można zaniedbać w obliczeniach maksymalnej prędkości obrotowej. Wprowadzane dodatkowe opóźnienie jest równe ok. 6,75 µs, co daje tolerancję błędu równą 0,3%. Jeżeli opóźnienie zostanie usunięte, czujnik może generować sygnał odpowiadający prędkości 1,8 mln obrotów na minutę.

Obliczenia maksymalnej prędkości obrotowej nie są wykonywane w celu określenia opóźnienia, ale do pozyskania informacji potrzebnych do zastosowania symulowanego sprzętu w pętli sterującej. Opóźnienia pomiędzy zapisami stanu na pinach powinny być równe co najmniej 1 ms, a ponadto powinno być uwzględnione dodatkowe opóźnienie związane ze sterowaniem i zmianami prędkości. Jeżeli kod czytnika ma problemy z dokładnym odczytem wyjścia czujnika, należy zwiększyć opóźnienie pomiędzy wywołaniami funkcji `digitalWrite()`.

Rysunek 6.5 przedstawia konfigurację sprzętu. Piny nr 2 i 3 czytnika są dołączone odpowiednio do pinów 10 i 11 czujnika. Do sterowania kierunkiem sygnału wyjściowego służą dwa przyciski monostabilne dołączone do masy i osobno do pinów nr 2 (poziom wysoki) i 3 (poziom niski). Kod z listingu 6.5 należy załadować do Arduino użytego jako czujnik. Do czytnika natomiast należy załadować kod z listingu 6.4. W odróżnieniu od rzeczywistego obrotomierza, do symulacji potrzebny jest jeszcze jeden pin, a mianowicie muszą być połączone piny zasilające 5 V oraz piny masy obu układów Arduino.



Rysunek 6.5. *Układ symulujący kod Graya*

Listing 6.5. *Kod Arduino użytego jako czujnik*

```
byte first , second; // kierunek zmiany pinów
boolean click , stateChang; // umieszczenie kliknięcia i zmiany stanu w zmiennych

void setup() {
  pinMode(2 , INPUT); // przycisk dekodera w dół
  pinMode(3 , INPUT); // przycisk dekodera w górę
  pinMode(11 , OUTPUT); pinMode(10 , OUTPUT); // wyjścia dekodera
  digitalWrite(2 , HIGH); digitalWrite(3 , HIGH); // wejścia rezystorów podwyższających
  digitalWrite(10 , HIGH); // stan początkowy
  digitalWrite(11 , LOW);
  stateChang = true;
} // koniec void setup()
```

```
void loop() {
  if (digitalRead(2) == 0 ){ // w dół
    first = 10; second = 11; // zapis na pinie 10, następnie na 11 dla kierunku w dół
    click = true;
  }
  if (digitalRead(3) == 0 ){ // w górę
    first = 11; second = 10; // zapis na pinie 11, następnie na 10 dla kierunku w górę
    click = true;
  }
  if (click == true ) { // wysłanie 1/2 impulsu, gdy naciśnięty jest przycisk
    stateChang = !stateChang; // zmiana stanu do zapisu
    digitalWrite(first, stateChang); // zmiana 1. pinu
    delay (2); // opóźnienie przed zmianą następnego pinu
    digitalWrite(second , stateChang); // zmiana 2. pinu
    delay (2); // opóźnienie przed zmianą następnego pinu przy największej prędkości
    click = false ; // reset
  }
  delay (100); // spowolnienie kodu = wolniejsza praca dekodera
} // koniec void loop()
```

Weryfikacja kodu

Po skonfigurowaniu układu podłącz czytnik Arduino do komputera i otwórz monitor portu szeregowego. Czujnik będzie wyświetlał liczby, gdy sygnał na pinie 2 zmieni stan na niski. Liczby będą się zmniejszać lub zwiększać w zależności od przychodzącego sygnału. Po każdorazowym naciśnięciu przycisku Arduino pracujące jako czujnik będzie wysyłać połowę kodu Graya. Jeżeli przytrzymasz przycisk, wysyłany będzie ciągły sygnał z maksymalną prędkością 208 ms, określoną w kodzie. Jeżeli przyciski nie są naciśnięte, Arduino będzie w stanie spoczynku. Taka symulacja jest bardzo pomocna przy debugowaniu kodu w obrabiarkach CNC lub w innych systemach wykorzystujących pętle sterujące.

■ **Uwaga** Jeżeli do wizualizacji sygnału czujnika nie może być użyty oscyloskop, zwiększ opóźnienie do ok. 200 ms, a zamiast czytnika zastosuj dwie diody LED.

Czujniki szeregowe

Transmisja szeregowa jest jedną z podstawowych form transmisji w informatyce i wiele czujników ją wykorzystuje. Czujniki szeregowe mogą wysyłać i odbierać więcej informacji niż czujniki analogowe, wysyłające pojedyncze bajty. Symulacja czujnika szeregowego za pomocą Arduino jest prosta dzięki wbudowanym funkcjom szeregowym i programowemu portowi szeregowemu. Trick polega na dobraniu prędkości transmisji odpowiedniej dla wysyłanych danych. Potrzebne informacje powinny być dostępne w dokumentacji czujnika. Zaleca się użycie programowego portu szeregowego, dzięki czemu inne porty szeregowe mogą być użyte do sterowania i monitoringu.

Szeregowe wysyłanie danych

W tym przykładzie został użyty czujnik radiowy Parallax RFID wysyłający dane z prędkością 9600 bodów. Czujnik odbiera specjalne tagi zawierające 40-bitowy identyfikator i wysyła je w postaci dziesięciu szesnastkowych liczb w kodzie ASCII. Na początku tagu jest wysyłany bajt o wartości 10, a na końcu bajt o wartości 13. Dostępny jest również pin do aktywacji czujnika. Kod dla Arduino służący do odczytu informacji z czujnika

jest dostępny pod adresem *http://arduino.cc/playground/Learning/PRFID*. W celu użycia portu programowego część kodu została zmodyfikowana przez Worapoht K. Do Arduino, które będzie odbierać sygnał radiowy RFID, załaduj kod z listingu 6.6.

Listing 6.6. *Kod autorstwa Worapoht K. ze zmienionymi komentarzami*

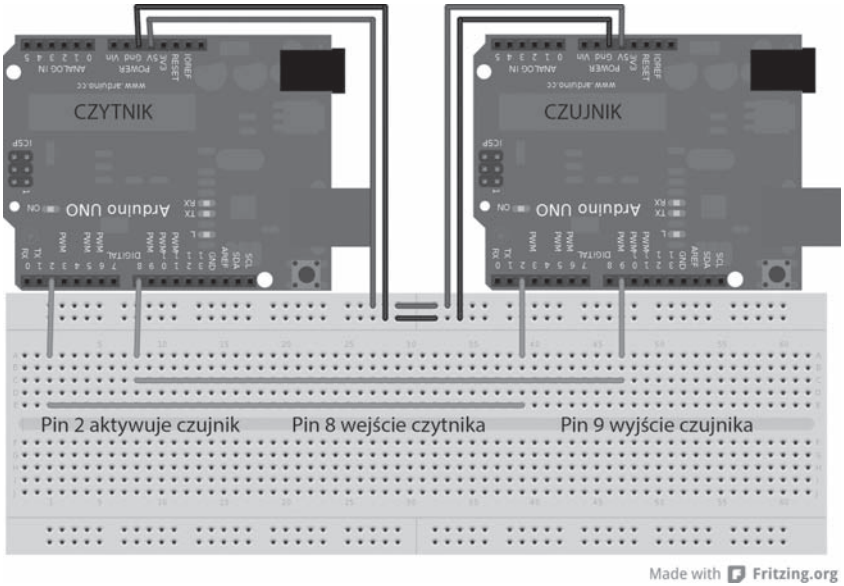
```
#include <SoftwareSerial.h>
int val = 0;        // tymczasowe dane
char code[10];      // Tag
int bytesread = 0; // liczba bajtów
#define rxPin 8     // pin SOUT  czytnika RFID
#define txPin 9     // brak połączenia

void setup() {
  Serial.begin(2400);     // sprzętowy port szeregowy 2400 bps do monitorowania
  pinMode(2,OUTPUT);      // pin RFID ENABLE
  digitalWrite(2, LOW); // aktywacja czytnika RFID
} // koniec void setup()

void loop() {
  SoftwareSerial RFID = SoftwareSerial(rxPin,txPin);
  RFID.begin(2400);
  if((val = RFID.read()) == 10) { // sprawdzenie nagłówka
    bytesread = 0;
    while(bytesread<10) { // odczyt 10-cyfrowego kodu
      val = RFID.read();
      if((val == 10)||(val == 13)) { // sprawdzenie wartości 10 lub 13
        break; // koniec odczytu
      }
      code[bytesread] = val; // dodanie cyfry
      bytesread++; // gotowość do odczytania następnej cyfry
    }
    if(bytesread == 10) { // jeżeli odczytanych jest 10 cyfr
      Serial.print("Kod TAG: "); // prawdopodobnie dobry TAG
      Serial.println(code); // wyświetlenie kodu TAG
    }
    bytesread = 0; // reset liczby bajtów
    delay(500);
  }
} // koniec void loop()
```

Jak pokazuje rysunek 6.6, konfiguracja symulowanego czujnika jest bardzo podobna do rzeczywistego: piny nr 2 obu płyt Arduino są ze sobą połączone, a pin nr 8 czytnika jest połączony z pinem nr 9 czujnika. Analogicznie jak poprzednio, muszą być podłączone piny 5 V i GND.

Listing 6.7 zawiera kod symulujący sygnał radiowy RFID.

Listing 6.7. *Symulator czujnika radiowego*

```
#include <SoftwareSerial.h>

void setup() {
  Serial.begin(2400); // sprzętowy port szeregowy 2400 bps do monitorowania
  pinMode(2,INPUT);
} // koniec void setup()
```



Rysunek 6.6. *Konfiguracja czujnika radiowego*

```
void loop() {
  SoftwareSerial RFID = SoftwareSerial(8,9); // pin 8 niepodłączony, pin 9 wysyłający
  RFID.begin(2400);
  if(LOW == digitalRead(2)) { // jeżeli czujnik ma być aktywny
    RFID.write(10); // wysłanie nagłówka
    RFID.write("1234567890"); // wysłanie kodu Tag
    RFID.write(13); // koniec transmisji
  }
} // koniec void loop()
```

Weryfikacja kodu

Po połączeniu elementów i załadowaniu kodu otwórz monitor portu szeregowego skonfigurowanego na prędkość 2400 bodów. Kod symulujący czujnik ustawia prędkość transmisji programowej na 2400 bodów, a następnie czeka na pojawienie się stanu niskiego na pinie nr 2 i rozpoczyna wysyłanie serii danych. Dane wysyłane do czytnika Arduino rozpoczynają się od bajtu o wartości 10, a kończą bajtem o wartości 13. Następnie wysyłany jest napis *Kod TAG: 1234567890*. Napis *1234567890* zawiera dokładnie dziesięć znaków i może być zastąpiony innymi znakami. Czasami mogą pojawić się przypadkowe dane, spowodowane brakiem synchronizacji z portem szeregowym. Aby weryfikować dane, należy dopisać dodatkowy kod, ale w tym przypadku do sprawdzenia wystarczy, by został poprawnie odebrany przynajmniej jeden kod RFID i porównany z listą poprawnych kodów.

Transmisja I2C

Transmisja danych I2C, znana również jako transmisja dwuliniowa, jest metodą synchronicznej komunikacji szeregowej wykorzystującej jedną linię do nadawania sygnału zegara, a drugą do przesyłania danych. Transmisja I2C jest podoba do zwykłej transmisji szeregowej. Nieliczne różnice dotyczą pracy sprzętu

podczas transmisji. Czujniki wykorzystujące ten rodzaj komunikacji mogą obsługiwać szeroki wachlarz danych, urządzeń i poleceń. Jeden czujnik może być wyposażony w wiele funkcji do pomiaru różnych parametrów. Tutaj symulowany będzie czujnik SRF10 Ultrasonic Ranger Finder. Jego kod jest dostępny w środowisku Arduino IDE w menu *Plik/Przykłady/Wire/SFRRange_reader* i należy go załadować do płyty Arduino pracującej jako czytnik.

Transmisja danych odbywa się na jednej linii, co oznacza, że tylko jedno urządzenie może wysyłać dane w danej chwili, przy czym za pomocą dwóch linii mogą być połączone ze sobą więcej niż dwa urządzenia. W większości konfiguracji magistrali I2C jest jedno główne urządzenie (*master*), które odbiera dane i steruje komunikacją innych urządzeń. Oprogramowanie Arduino zawiera bibliotekę implementującą ten rodzaj komunikacji, działającą poprawnie w większości zastosowań, szczególnie gdy jest użyta na głównym urządzeniu. Niemniej jednak brakuje w niej niektórych funkcjonalności potrzebnych do symulowania czujników I2C.

W symulacjach czujników I2C uzyskanie najlepszego efektu wymaga manipulowania rejestrami sprzętowymi. Ta metoda konfiguracji magistrali I2C jest nieco bardziej skomplikowana, ale niezbyt trudna, jeżeli posiądzie się podstawową wiedzę na temat rejestrów.

Uwaga Zapoznaj się z dokumentacją do układu ATmega 328P (sekcja 22.5, str. 223 – 247). Opisany tam został moduł I2C zawarty w mikrokontrolerze Arduino.

Komunikacja I2C odbywa się na pinie analogowym nr 5 dla zegara (SCL) oraz pinie analogowym nr 4 dla danych (SDL). Do skonfigurowania transmisji w trybie slave wykorzystywane są cztery rejestry: TWAR, TWCR, TWDR oraz TWSR. W module I2C jest jeszcze piąty rejestr, TWBR, niewykorzystywany w trybie slave. Jest on używany w trybie master do sterowania prędkością linii SCL. Rejestr SREG jest jedynym rejestrem spoza modułu I2C, który będzie zmieniany w tym przykładzie. Rejestry są używane i zmieniane w kodzie w taki sam sposób jak zmienne. Nazwy rejestrów zostały już zdefiniowane w głównych bibliotekach środowiska Arduino IDE i nie trzeba ich deklarować. Wszystkie rejestry użyte w tej części rozdziału mają wielkość 1 bajtu. Niektóre rejestry służą do transmisji danych, inne do sterowania.

Rejestr TWRC

Rejestr TWRC służy do sterowania obiema liniami i określa podstawowy przebieg transmisji I2C. Każdy bit rejestru steruje inną funkcją modułu. Nazwa bitu określa jego pozycję w bajcie.

- Aby skonfigurować Arduino w tryb slave, ustaw bity TWI Enable Acknowledge (TWEA) oraz TWI Enable (TWEN) rejestru TWRC na wartość 1. Bit TWEN (nr 2) aktywuje moduł I2C, natomiast bit TWEA (nr 6) konfiguruje moduł do wysyłania potwierdzeń, jeżeli są potrzebne. Jeżeli bit TWEA nie jest ustawiony, urządzenie nie będzie odpowiadać na próby nawiązania komunikacji przez inne urządzenia.
- Bity TWI Interrupt (TWINT, nr 7) oraz TWI Interrupt Enable (TWIE, nr 0) są innymi ważnymi bitami rejestru, służącymi do sterowania oprogramowaniem. Bit TWINT jest flagą, której wartość 1 oznacza pojawienie się zdarzenia wymagającego obsługi przez program. Program po obsłudze zdarzenia powinien skasować flagę, ustawiając bit TWINT na wartość 1. Bity TWINT oraz TWIE można skonfigurować jako wewnętrzne przerwanie.

Transmisja I2C jest wrażliwa na opóźnienia, dlatego dobrą metodą obsługi komunikacji jest użycie wewnętrznych przerwań Arduino. Należy w tym celu ustawić bit TWIE i bit globalnego przerwania w rejestrze SREG. Rejestr SREG należy ustawić przy użyciu bitowego operatora *OR* (znak |), dzięki czemu pozostałe bity nie będą zmienione. Bit należy resetować po każdym zgłoszeniu przerwania. Jeżeli bit TWINT zostanie sprzętowo ustawiony na 1, wówczas zgłaszane jest przerwanie i wywoływana procedura jego obsługi (`ISR(wektor)`). Procedura `ISR()` działa bardzo podobnie jak zwykła funkcja, np. `setup()` lub `loop()`. Procedura `ISR()` może

być zapisana bezpośrednio w szkicu Arduino bez poprzedzających ją informacji, lecz wymagany jest wektor. Termin *wektor* jest nazwą opisującą przerwanie, na które odpowiada procedura `ISR()`.

■ **Uwaga** Opisy nazw wektorów wykorzystywanych w bibliotekach AVR oprogramowania Arduino są dostępne pod adresem *www.nongnu.org/avr-libc/user-manual/group__avr__interrupts.html*. Wektor przerwania I2C w płycie Arduino z układem 328P nosi nazwę *TWI_vect*.

Rejestr TWAR

Ostatnim rejestrem, który należy skonfigurować, aby urządzenie w trybie slave odpowiadało na dane transmitowane przez magistralę, jest adres. Adres jest ustawiany w rejestrze TWI Address Register (TWAR). Siedem najstarszych bitów (7 – 1) stanowi adres. Bit nr 0 informuje urządzenie, że może odpowiadać na wywołania przysyłane z ogólnego adresu. Ogólny adres ma wartość 0 i kiedy urządzenie master go wysyła, odpowiada każde urządzenie, które ma włączoną odpowiedź. Adres ustawiony w rejestrze TWAR musi być przesunięty w lewo o jeden bit. W ten sposób do magistrali I2C może być dołączonych 126 urządzeń.

Rejestr TWDR

Przez rejestr TWI Data Register (TWDR) przepływają wszystkie transmitowane dane. Jeżeli do rejestru wpisana zostanie wartość (instrukcja `TWDR = FOO;`), rozpoczyna się przesyłanie danych. W celu odczytania przysyłanych danych należy wpisać zawartość rejestru do zmiennej (`FOO = TWDR;`). Transmisja I2C wykorzystuje unikalne wartości oznaczające początek i koniec transmisji, obejmujące przesyłane dane. Dzięki temu można przesyłać całe bajty, w odróżnieniu od zwykłej transmisji szeregowej, przedstawionej w poprzednim przykładzie, gdzie część bajtu jest używana do oznaczenia początku i końca większej porcji danych. Rejestr TWI Status Register (TWSR) ułatwia przesyłanie większych ilości danych w odpowiedniej kolejności.

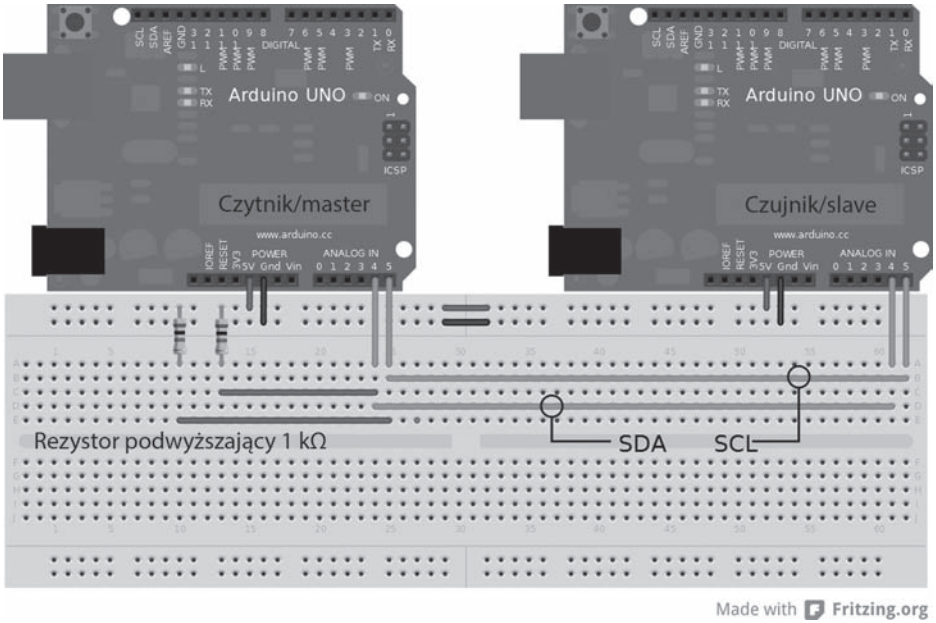
Rejestr TWSR

Rejestr TWSR zawiera informacje o tym, co się dzieje na magistrali I2C, np. kierunek danych, błędy, żądania transmisji. Ważne jest, aby program sterujący odczytywał wartość tego rejestru. W dokumentacji do modułu TWI układu Atmel można znaleźć listę kodów stanu. W symulacji czujnika ważne są stany *0x80* oraz *0xA8*. Stan *0x80* oznacza, że nadeszły dane, które należy odczytać, natomiast stan *0xA8* informuje, że czujnik ma wysłać dane. Rejestr zawiera trzy bity o numerach od 2 do 0, które są ważne dla urządzenia pracującego w trybie slave. Bity te muszą być wyzerowane bitowym operatorem *AND* (`&`) z wartością *0b11111000* i nie wymagają przesunięcia. W ten sposób obliczane są kody stanu.

Wysyłanie danych magistralą I2C

Do zbudowania układu przedstawionego na rysunku 6.7 potrzebne będą dwa rezystory podwyższające w celu uzyskania wysokiego stanu na liniach SCL i SDA, zgodnie z wymaganiami transmisji I2C. Płyty Arduino są połączone pinami analogowymi nr 4 i 5 oraz pinami zasilania i masy. Kod z listingu 6.8 należy załadować do płyty pracującej jako czujnik.

Przykład pokazuje, jak zaimplementować komunikację I2C, bezpośrednio zmieniając rejestry sprzętowe w połączonym kodzie AVR C i Arduino. W celu uzyskania lepszej czytelności wartości wpisywane do rejestrów są zapisane w formacie dwójkowym, odpowiadającym pozycjom bitów w rejestrze. Kod zwiększa wartość zmiennej `manipVar` w każdym wywołaniu funkcji `loop()`. Dioda LED na płycie zapala się lub gaśnie w zależności od polecenia odebranego z urządzenia master. Cała komunikacja jest realizowana w funkcji `ISR()`. Zmienne używane w funkcji muszą być zadeklarowane jako globalne. Nie jest możliwe przekazywanie danych do funkcji `ISR()`, ponieważ jest wywoływana przez sprzęt, a nie przez kod programu.



Rysunek 6.7. *Konfiguracja magistrali I2C*

Listing 6.8. *Kod symulowanego czujnika I2C*

```
byte address = 112;          // adres czujnika
unsigned int manipVar = 0; // zmienna do zmiany danych
byte bytessent = 2 ;         // liczba bajtów do wysłania
byte bytestosend[2] ;        // przygotowanie danych do wysłania
byte command = 0 ;           // przechowywanie poleceń

void setup() {
  TWAR = (address << 1) | 0b00000001; // ustawienie adresu i ogólnej odpowiedzi na wywołania
  TWCR = 0b01000101; // ustawienie TWEA, TWEN i TWIE na 1
  SREG |= 0b10000000; // globalne włączenie przerwań
  pinMode(13 , OUTPUT);
} // koniec void setup()

void loop() {
  if (command == 0x50){ // włączenie diody dla polecenia 0x50
    digitalWrite (13 , HIGH);
  }
  if (command == 0x02){ // wyłączenie diody dla polecenia 0x02
    digitalWrite (13 , LOW);
  }
  manipVar++; // główna zmienna do manipulowania dwoma bajtami wyjściowymi
  bytestosend [0] = manipVar; // przygotowanie starszego i młodszego bajtu zmiennej manipVar
  bytestosend [1] = manipVar >> 8 ; // starszy bajt manipVar
  delay (250); // coś innego do zrobienia podczas oczekiwania
} // koniec void loop()
```

```
ISR (TWI_vect){ // procedura obsługi przerwania przypisana do wektora
  if (TWCR & (1 << TWINT)) { // powtórne sprawdzenie, czy przerwanie jest właściwe
    if ((TWSR & 0b11111000) == 0x80){ // dane przychodzące
      command = TWDR; // skopiowanie danych polecenia do wykorzystania później
      TWCR = 0b11000100; // reset oryginalnej konfiguracji
    }
    if ((TWSR & 0b11111000) == 0xA8 ) { // żądanie danych wychodzących
      while (bytessent > 0 ){ // wysłanie bajtów do mastera
        bytessent--;
        TWDR = bytestosend [bytessent]; // wysłanie danych od najstarszego do najmłodszego bajtu
        TWCR = 0b11000101; // reset po każdym wysłaniu
        delay (5); // chwilowa pauza po wysłaniu
      }
      if (bytessent == 0 ){ // reset liczby bajtów i jeżeli bufor jest pusty
        bytessent = 2;
      }
    } // koniec if ((TWSR & 0b11111000) == 0xA8 )
    TWCR = 0b11000101; // jeszcze jeden reset na wszelki wypadek
    SREG |= 0b10000000; // włączenie z powrotem przerwań
  } // koniec if (TWCR & (1 << TWINT))
} // koniec ISR (TWI_vect)
```

Weryfikacja kodu

Po połączeniu elementów i załadowaniu kodu do odpowiednich płyt Arduino podłącz czytnik do portu USB i otwórz monitor portu szeregowego. Powinny pojawiać się kolejne liczby, a dioda LED powinna migać, gdy urządzenie master będzie wysyłać odpowiednie polecenia. Bezpośrednie odwoływanie się do rejestrów pozwala na maksymalną kontrolę interfejsu I2C, nieosiągalną przy użyciu bibliotek.

■ **Uwaga** W rozdziale 10., poświęconym wieloprocesowości, opisana jest komunikacja SPI, która może być użyta w symulacji czujników.

Podsumowanie

Techniki opisane w tym rozdziale mogą być zastosowane nie tylko do obsługi czujników, ale również innych systemów przesyłających między sobą dane. W rozdziale skupiliśmy się na połączeniu czujników z Arduino, ponieważ jest to najtrudniejsze zadanie w symulacji czujników.

Aby uniknąć komplikacji, przy tworzeniu kodu symulującego czujniki pracuj powoli i skupiaj się na określonej części czujnika. Warto poświęcić więcej czasu na pisanie kodu symulującego rzeczywiste czujniki, które są dostępne i mogą być użyte do weryfikacji kodu.

Skorowidz

C

D

E

F

G

H

I

J

K

P

R

S

Ś

T

U

V

W

X

Z

Ż

Obowiązkowa lektura dla każdego pasjonata elektroniki!

Platforma Arduino to cudowne dziecko inżynierów, dzięki któremu świat elektroniki zyskał tysiące nowych entuzjastów. Skąd oni się wzięli? Dzięki Arduino nawet kompletny laik może zbudować atrakcyjny układ elektroniczny, który ułatwi mu życie. Jeżeli do tego dołożymy świetną dokumentację oraz środowisko przeznaczone specjalnie do tworzenia oprogramowania, to staje się jasne, dlaczego Arduino jest tak popularne.

Na rynku wydawniczym obecnych jest już kilka pozycji poświęconych Arduino, jednak zazwyczaj zawierają one zbiory projektów, które czytelnik może wykonać we własnym zakresie. Tymczasem jeżeli masz ambicję tworzyć nowatorskie rozwiązania, których nie spotkasz w sieci ani w książkach, musisz zdobyć zdecydowanie szerszą wiedzę. Ta książka Ci jej dostarczy. W trakcie lektury dowiesz się, jak wykorzystać sieci radiowe XBee, komunikować się z systemem Android oraz integrować Arduino z niestandardowymi układami, takimi jak Atmel. Ponadto dowiesz się, jak wykorzystać wiele platform Arduino do pracy nad konkretnym problemem, a potem nauczysz się tworzyć biblioteki dla Arduino i udostępniać je społeczności. W tym tkwi największa siła platformy!

Dowiedz się:

- jak zwiększyć wydajność Arduino
- jak zbudować sieć sensorową
- do czego wykorzystać moduły radiowe XBee
- jak stworzyć grę na Arduino

TECHNOLOGY IN ACTION™ Apress®

ISBN 978-83-246-8222-5

9 788324 682225

Cena: 54,90 zł