

## Eliminacja drgań styków, wielozadaniowość, wyświetlacz LCD

Opracował opiekun przedmiotu dr inż. Krzysztof Chudzik

Wydział Informatyki i Telekomunikacji  
Politechnika Wrocławskiego

Wrocław, 2021.08.25 19:54:32

---

### Spis treści

<b>1</b>	<b>Eliminacja drgań styków elektromechanicznych elementów przełączających</b>	<b>1</b>
1.1	Czym są drgania styków? . . . . .	1
1.2	Kiedy występuje problem w programie? . . . . .	2
1.3	Eliminacja wpływu drgań styków na przebieg programu (ang. <i>debouncing</i> ) . . . . .	4
<b>2</b>	<b>Wielozadaniowość</b>	<b>6</b>
2.1	„Współbieżność” wykonywania zadań na mikrokontrolerze jednordzeniowym . . . . .	6
2.2	Wzajemne blokowanie się zadań . . . . .	7
2.3	Jak nie blokować wzajemnie wykonywanych zadań . . . . .	7
<b>3</b>	<b>Wyświetlacz LCD</b>	<b>8</b>
3.1	Wyświetlacz LCD w zestawie laboratoryjnym . . . . .	8
3.2	Magistrala I2C . . . . .	9
3.3	Sterowanie wyświetlaczem LCD w zestawie laboratoryjnym. . . . .	9

---

### Lista zadań

1	Program z eliminacją wpływu drgania styków . . . . .	6
2	Program wykonujący współbieżnie różne zadania . . . . .	8
3	Wykorzystanie wyświetlacza LCD - Program „Stoper elektroniczny” . . . . .	11

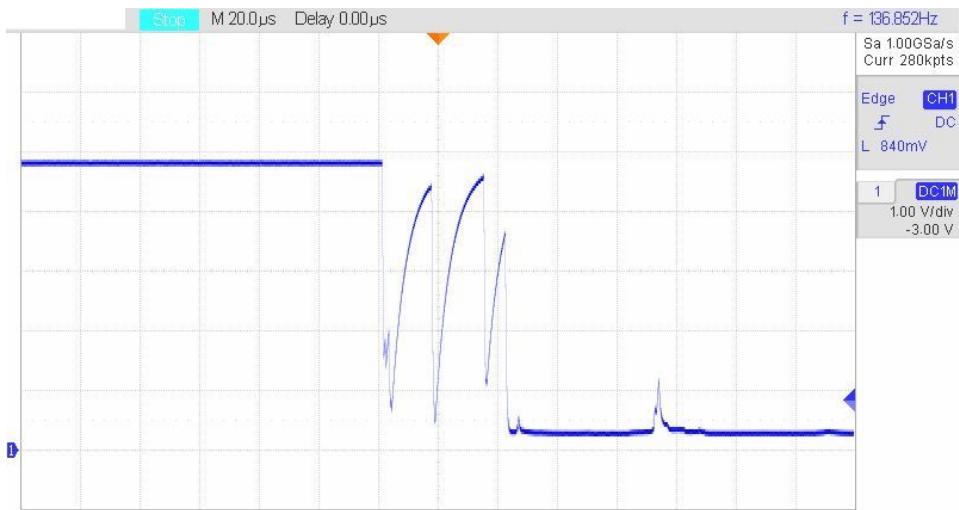
---

## 1 Eliminacja drgań styków elektromechanicznych elementów przełączających

### 1.1 Czym są drgania styków?

W ramach poprzedniego laboratorium przedstawiono sposób odczytu stanu przycisku przyłączonego do płytki Arduino. Zasygnalizowaliśmy, że to elementy elektromechaniczne. Zamknięcie lub otwarcie obwodu elektrycznego wiąże się tu ze zjawiskiem określonym mianem „drgania styków”. Powoduje ono, że mamy pewne stany przejściowe, które mogą być interpretowane jako bardzo szybkie wielokrotne zamknięcie i otwarcie obwodu.

Drgania styków przedstawiono na oscylogramie z rysunku 1.



Rysunek 1: Oscylogram przebiegu napięcia zawierający drgania styków po naciśnięciu przycisku w zestawie laboratoryjnym.

Oscylogram przedstawia zjawiska zachodzące przy naciśnięciu przełącznika przyciskanego (przycisku). Niebieska linia przedstawia wartość napięcia w czasie. Jedna działka („kratka”) w poziomie to 20 mikrosekund, zaś w pionie to 1 volt. Naciśnięcie przycisku to przejście ze stanu napięciowego wysokiego do niskiego. W idealnym przełączeniu oczekiwaliśmy jednego „schodka” w dół. W praktyce, jak na oscylogramie, widzimy wahania napięcia, które mikrokontroler może zinterpretować jako następujące bardzo szybko po sobie wyłączenia iłączenia. Ostatecznie napięcie osiąga poziom niski, ale stabilizacja zajmuje pewien czas, który dla mikrokontrolera wcale krótki nie jest.

Spróbujmy oszacować, wyliczyć, jak szybko następują te zmiany, które widzimy na oscylogramie, dla mikrokontrolera w zestawie Arduino. Patrząc na oscylogram, przyjmijmy, że okres jednego cyklu wyłączenie-załączenie to 20 mikrosekund (jedna kratka). Gdyby działało się to cyklicznie, to w ciągu sekundy mielibyśmy 50 tysięcy cykli. Oczywiście człowiek z taką szybkością nigdy nie będzie w stanie ręcznie przełączać. Uświadommy sobie jednak, że mikrokontroler w Arduino pracuje z częstotliwością zegara 16 MHz, czyli wykonuje 16 milionów operacji na sekundę. Jest to wartość ponad 300 razy większa niż częstotliwość drgań. Oznacza to, że pomiędzy każdym zarejestrowanym drganiem jest ponad 300 cykli zegara. W tym czasie można wykonać całkiem sporo instrukcji programu (mniej niż 300 oczywiście, bo część zajmuje kilka cykli). Mikrokontroler jest więc w stanie zaobserwować drgania, jako kolejne cykle naciśnięcia przycisku.

Powyzsza analiza może wypaść zupełnie inaczej dla innych zarejestrowanych przebiegów napięcia. Powtarzalność tych przebiegów jest bardzo niska. Proszę przedstawioną analizę na przykładzie praktycznym traktować jedynie jako poglądową. Oddaje ona istotę problemu.

## 1.2 Kiedy występuje problem w programie?

Spróbujmy zaobserwować to zjawisko. Postawmy sobie za cel, aby program reagował jedynie na naciśnięcie przycisku. Nie chcemy, aby program reagował na zwolnienie przycisku. Gdy przycisk jest zwolniony, albo przyciśnięty, nie powinno się nic dziać.

Przypomnijmy, że stan portu zwolnionego przycisku to HIGH, a naciśniętego to LOW.

Program, po naciśnięciu zielonego przycisku ma zgasić zaświeconą diodę i zapalić kolejną dokładnie w cyklu R,G,B,R,G,... . Program jest przedstawiony jako kod 1.

Kod 1: Zapalanie kolorów R,G,B,R,G,... po naciśnięciu przycisku - NIE DZIAŁA dobrze.

```

1 #define LED_RED 6
2 #define LED_GREEN 5
3 #define LED_BLUE 3
4
5 #define RED_BUTTON 2
6 #define GREEN_BUTTON 4
7
8 int led[] = {LED_RED, LED_GREEN, LED_BLUE};
9
10 void initRGB()
11 {
12     pinMode(LED_BUILTIN, OUTPUT);
13     digitalWrite(LED_BUILTIN, LOW);
14
15     pinMode(LED_RED, OUTPUT);
16     digitalWrite(LED_RED, HIGH);
17
18     pinMode(LED_GREEN, OUTPUT);
19     digitalWrite(LED_GREEN, LOW);
20
21     pinMode(LED_BLUE, OUTPUT);

```

```

22     digitalWrite(LED_BLUE, LOW);
23 }
24
25 void initButtons()
26 {
27     pinMode(RED_BUTTON, INPUT_PULLUP);
28     pinMode(GREEN_BUTTON, INPUT_PULLUP);
29 }
30
31 bool isGreenButtonPressed()
32 {
33     static int previous_reading = HIGH;
34     int current_reading = digitalRead(GREEN_BUTTON);
35
36     bool isPressed = false;
37     if (previous_reading == HIGH && current_reading == LOW)
38     {
39         isPressed = true;
40     }
41
42     if (previous_reading != current_reading)
43     {
44         previous_reading = current_reading;
45     }
46
47     return isPressed;
48 }
49
50 void setup()
51 {
52     initRGB();
53     initButtons();
54 }
55
56 int led_index = 0;
57 void loop()
58 {
59     if (isGreenButtonPressed())
60     {
61         digitalWrite(led[led_index], LOW);
62         led_index = ++led_index % 3;
63         digitalWrite(led[led_index], HIGH);
64     }
65 }

```

Najpierw mała uwaga dotycząca języka C. Zmienna lokalna zadeklarowana wewnątrz funkcji ze słowem kluczowym **static** to statyczna zmienna lokalna. Zakres widoczności jest taki sam jak zmiennej lokalnej, ale istnieje ona pomiędzy wywołaniami funkcji i zachowuje swoją wartość. Więcej na stronie Wikipedii w artykule [Static \(keyword\) \(link\)](#).

W kodzie 1 funkcja **loop()** realizuje przełączanie kolorów diody RGB w zadany cyklu.

Zwróćmy uwagę, że gdybyśmy jedynie sprawdzili czy przycisk jest naciśnięty, to znaczy, warunek w linii 59 miałby postać **digitalRead(GREEN\_BUTTON) == LOW**, to po przytrzymaniu przycisku zaobserwujemy świecenie diody w kolorze zbliżonym do białego. Zrób taki eksperyment i zastanów się dlaczego tak to działa.

Musimy zatem wykryć zmianę stanu przycisku, i to dokładnie w kierunku od zwolnionego (HIGH) do naciśniętego (LOW). Realizuje to funkcja **isGreenButtonPressed()**. Kod jest stosunkowo prosty, więc zostawiam do samodzielnej analizy.

Jednak po uruchomieniu, kod 1 NIE DZIAŁA prawidłowo. To co najbardziej rzuca się w oczy, to zdarzające się przełączanie podczas zwalniania przycisku. Przy naciskaniu też nie zawsze pojawia się oczekiwany kolor. Co się stało?

Powiedzmy sobie czego spodziewaliśmy się w odczytach stanu przycisku. Oznaczmy H-HIGH, L-LOW. Idealna sekwencja odczytów przy naciśnięciu wyglądałaby następująco:

... H,H,H,H,H,H,L,L,L,L,L,L, ....

Dla nas ważna jest sekwencja H,L. To jest naciśnięcie przycisku.

Niestety, mamy drgania styków w czasie, a odczyty są stosunkowo szybkie. Możliwe, że odczytano, na przykład:

... H,H,H,H,H,H,L,H,L,L,L,L,L,L, ....

W tym ciągu odczytów mamy dwie sekwencje H,L i nastąpią dwa bardzo szybkie przełączenia. Efektu pierwszego, po prostu, nie jesteśmy w stanie zaobserwować, bo bardzo szybko następuje drugie przełączenie. Widzimy więc stan po drugim przełączeniu. Oczywiście takich powtórzeń sekwencji H,L może być znacznie więcej.

Teraz popatrzmy, co dzieje się przy zwalnianiu przycisku. Chcielibyśmy uzyskać sekwencję:

... L,L,L,L,L,H,H,H,H,H,H,H, ....

Ale ponieważ mamy drgania styków, więc możliwe jest odczytanie, na przykład:

... L,L,L,L,L,H,L,H,H,H,H,H,H, ....

W tym ciągu występuje sekwencja H,L, która wyzwoli przełączenie.

### 1.3 Eliminacja wpływu drgań styków na przebieg programu (ang. *debouncing*)

Znamy więc przyczynę problemu. Trzeba pozbyć się wpływu drgania styków na działanie układu. W języku angielskim nosi to miano *debouncing*. Problem wpływu drgania styków można generalnie rozwiązać na dwa sposoby: sprzętowy i programowy.

Pierwszy sposób realizowany jest na drodze dodania prostego obwodu elektronicznego filtrującego drgania styków, składającego się z rezystora i kondensatora. Nie wymaga to zmian programowych, ale zwiększa ilość elementów elektronicznych, komplikuje rozplanowanie elementów na płytce drukowanej układu, zwiększa powierzchnię płytka i podnosi koszt. Zatem, korzystniejsze jest poszukanie rozwiązania na drodze programowej i tak też uczymy się.

Drgania styków zachodzą w pewnym krótkim okresie czasu. Zatem, możemy przyjąć regułę, że jeśli wykryliśmy jakąś zmianę w stanie przycisku, to przez pewien okres czasu nie będziemy reagować. Poczekamy, aż odczyty ustabilizują się. Jednym z wielu możliwych rozwiązań jest to zaproponowane w kodzie 2 (zmodyfikowano funkcję `isGreenButtonPressed()`).

Kod 2: Zapalanie kolorów R,G,B,R,G,... po naciśnięciu przycisku - modyfikacja (wersja 1).

```
1 #define LED_RED 6
2 #define LED_GREEN 5
3 #define LED_BLUE 3
4
5 #define RED_BUTTON 2
6 #define GREEN_BUTTON 4
7
8 int led[] = {LED_RED, LED_GREEN, LED_BLUE};
9
10 void initRGB()
11 {
12     pinMode(LED_BUILTIN, OUTPUT);
13     digitalWrite(LED_BUILTIN, LOW);
14
15     pinMode(LED_RED, OUTPUT);
16     digitalWrite(LED_RED, HIGH);
17
18     pinMode(LED_GREEN, OUTPUT);
19     digitalWrite(LED_GREEN, LOW);
20
21     pinMode(LED_BLUE, OUTPUT);
22     digitalWrite(LED_BLUE, LOW);
23 }
24
25 void initButtons()
26 {
27     pinMode(RED_BUTTON, INPUT_PULLUP);
28     pinMode(GREEN_BUTTON, INPUT_PULLUP);
29 }
30
31 bool isGreenButtonPressed()
32 {
33     static int debounced_button_state = HIGH;
34     static int previous_reading = HIGH;
35     static int blocker = 0;
36     bool isPressed = false;
37
38     int current_reading = digitalRead(GREEN_BUTTON);
39
40     if (previous_reading != current_reading)
41     {
42         blocker = 1000;
43     }
44
45     if (blocker == 0)
46     {
47         if (current_reading != debounced_button_state)
48         {
49             if (debounced_button_state == HIGH && current_reading == LOW)
50             {
51                 isPressed = true;
52             }
53             debounced_button_state = current_reading;
54         }
55     }
56     else
57     {
58         --blocker;
59     }
60
61     previous_reading = current_reading;
62
63     return isPressed;
64 }
```

```

65
66 void setup()
67 {
68     initRGB();
69     initButtons();
70 }
71
72 int led_index = 0;
73 void loop()
74 {
75     if (isGreenButtonPressed())
76     {
77         digitalWrite(led[led_index], LOW);
78         led_index = ++led_index % 3;
79         digitalWrite(led[led_index], HIGH);
80     }
81 }

```

W kodzie 2, wprowadzono zmienną `blocker`, która nie pozwala na zapamiętanie zmiany stanu przycisku przez zadaną liczbę wywołań funkcji `isGreenButtonPressed()` od zmiany wartości odczytywanej z wejścia cyfrowego. Oczekujemy w ten sposób na stabilizację odczytów. W ten sposób odfiltrujemy też zakłócenia elektryczne, które mogłyby być traktowane jak naciśnięcie przycisku. Ujmując to innymi słowy, dokonujemy ewentualnej aktualizacji stanu, jeśli przez zadaną liczbę odczytów nic się nie zmieniło.

Liczbę zablokowanych cykli dobrano na drodze eksperymentu. Jest to rozwiążanie proste, ale czas opóźnienia silnie związany jest z częstotliwością wywołań funkcji `loop()`. Częstotliwość jej wywołań zależy od czasu jej wykonania. Zauważmy, że jeśli czas wykonania funkcji będzie wynosił 1ms, to będzie ona wywoływana nie częściej niż 1000 razy na sekundę. Dodaj do funkcji `loop()`, bezpośrednio w ciele funkcji (poza `if()`), wywołanie `delay(1)` i sprawdź efekt. Program zareagował z około sekundowym opóźnieniem. Dekrementacja zmiennej `blocker` z wartości 1000 do 0, będzie trwała, jak łatwo wyliczyć, co najmniej 1 sekundę! Co najmniej, ponieważ trzeba dodać czas wykonania funkcji `isGreenButtonPressed()`, wywoływanie funkcji `loop()`, itd.

Przedstawione w kodzie 2 rozwiązanie w pewnym stopniu rozwiązuje problem, ale ma zasadniczą wadę. W bardziej złożonych programach jest w praktyce trudno wyliczyć czas trwania wykonywania funkcji `loop()`, szczególnie jeśli użyjemy funkcji bibliotecznych, których działanie nie jest dla nas dokładnie znane. Stąd liczba zablokowanych cykli odczytu (1000) była dobrana eksperymentalnie, co na pewno nie jest dobrą praktyką. Chcielibyśmy ściśle określić czas, przez który układ nie może reagować na zmiany stanu przycisku.

W przypadku Arduino można skorzystać z funkcji `millis()`, która zwraca liczbę milisekund, które upłynęły od uruchomienia programu. Typ zwracany to `unsigned long` (32 bity w Arduino). Cykl takiego licznika to niecałe 50 dni. Zapoznaj się z [dokumentacją funkcji \(link\)](#).

Zmienną `bloker` w kodzie 2 możemy zastąpić pomiarem czasu korzystając z funkcji `millis()`. Przykładowe rozwiązanie przedstawiono w kodzie 3 (zmodyfikowano funkcję `isGreenButtonPressed()`).

Kod 3: Zapalanie kolorów R,G,B,R,G,... po naciśnięciu przycisku - modyfikacja (wersja 2).

```

1 #define LED_RED 6
2 #define LED_GREEN 5
3 #define LED_BLUE 3
4
5 #define RED_BUTTON 2
6 #define GREEN_BUTTON 4
7
8 int led[] = {LED_RED, LED_GREEN, LED_BLUE};
9
10 void initRGB()
11 {
12     pinMode(LED_BUILTIN, OUTPUT);
13     digitalWrite(LED_BUILTIN, LOW);
14
15     pinMode(LED_RED, OUTPUT);
16     digitalWrite(LED_RED, HIGH);
17
18     pinMode(LED_GREEN, OUTPUT);
19     digitalWrite(LED_GREEN, LOW);
20
21     pinMode(LED_BLUE, OUTPUT);
22     digitalWrite(LED_BLUE, LOW);
23 }
24
25 void initButtons()
26 {
27     pinMode(RED_BUTTON, INPUT_PULLUP);
28     pinMode(GREEN_BUTTON, INPUT_PULLUP);
29 }
30
31 #define DEBOUNCE_PERIOD 10UL
32

```

```

33 bool isGreenButtonPressed()
34 {
35     static int debounced_button_state = HIGH;
36     static int previous_reading = HIGH;
37     static unsigned long last_change_time = 0UL;
38     bool isPressed = false;
39
40     int current_reading = digitalRead(GREEN_BUTTON);
41
42     if (previous_reading != current_reading)
43     {
44         last_change_time = millis();
45     }
46
47     if (millis() - last_change_time > DEBOUNCE_PERIOD)
48     {
49         if (current_reading != debounced_button_state)
50         {
51             if (debounced_button_state == HIGH && current_reading == LOW)
52             {
53                 isPressed = true;
54             }
55             debounced_button_state = current_reading;
56         }
57     }
58
59     previous_reading = current_reading;
60
61     return isPressed;
62 }
63
64 void setup()
65 {
66     initRGB();
67     initButtons();
68 }
69
70 int led_index = 0;
71 void loop()
72 {
73     if (isGreenButtonPressed())
74     {
75         digitalWrite(led[led_index], LOW);
76         led_index = ++led_index % 3;
77         digitalWrite(led[led_index], HIGH);
78     }
79 }

```

Uwaga do kodu: Wartości liczbowe (literaly) typu `unsigned long` muszą być zapisywane z przyrostkiem `UL` (lub `ul`). Stąd zapis, na przykład, `static unsigned long last_change_time = 0UL;` w kodzie 3.Więcej informacji można znaleźć, na przykład, na stronie [Integer literal \(link\)](#).

Zastanów się co robić, gdy eliminacja drgania styków (*debouncing*) będzie zachodzić w tym samym czasie, gdy licznik milisekund osiągnie koniec okresu.

## Zadanie 1: Program z eliminacją wpływy drgania styków

Napisz program, Program, po naciśnięciu i zwolnieniu jednego z przycisków (zielony i czerwony) ma zgasić zaświeconą diodę i zapalić kolejną dokładnie w cyklu R,G,B,R,G,... .

Jest to zadanie przykładowe. Prowadzący może zmodyfikować lub zmienić treść zadania.

## 2 Wielozadaniowość

### 2.1 „Współbieżność” wykonywania zadań na mikrokontrolerze jednordzeniowym

Pojęcie „współbieżność” w wykonywaniu zadań na mikrokontrolerze, który jest jednordzeniowy, może budzić pewne emocje. Nie, nie będziemy mówić tu o klasycznym programowaniu współbieżnym. Zajmiemy się problemem, jak uniknąć zawłaszczenia czasu mikrokontrolera przez jedno wykonywane zadanie. Mówiąc nieco nieformalnie, przez zadanie będącym rozumieć tutaj zespół czynności, które musi zrealizować mikrokontroler, aby spełnić pewne wymaganie funkcjonalne. Przykładami zadań, które należy wykonywać równolegle, mogą być: reagowanie na zdarzenia z przełączników lub klawiatury, obsługa elementów sygnalizacyjnych (na przykład, diody świecące), pomiar temperatury otoczenia, komunikacja z innymi urządzeniami, itd.

## 2.2 Wzajemne blokowanie się zadań

Wróćmy do doskonale znanego nam przykładu z Arduino IDE, programu `Blink.ico`. Zaświeca on i gasi diodę. Zmiana stanu diody następuje co sekundę. Ale co dzieje się pomiędzy przełączniami? Wywoływana jest funkcja `delay()`, która w czasie swojego działania wstrzymuje wykonanie innych możliwych operacji.

Spróbujmy napisać program, który będzie wykonywał to co program `Blink.ico`, oraz jeśli przycisk czerwony będzie naciśnięty, to będzie świecić się czerwona dioda (pomijamy *debouncing*). Zaczniemy od wersji mocno naiwnej przedstawionej jako kod 4.

Kod 4: Miganie diodą i odczyt przycisku równocześnie (Tak NIE programować!).

```
1 #define LED_RED 6
2 #define RED_BUTTON 2
3
4 void setup()
5 {
6     pinMode(LED_BUILTIN, OUTPUT);
7     digitalWrite(LED_BUILTIN, LOW);
8
9     pinMode(LED_RED, OUTPUT);
10    digitalWrite(LED_RED, LOW);
11
12    pinMode(RED_BUTTON, INPUT_PULLUP);
13 }
14
15 void blinkBuiltInLed()
16 {
17     digitalWrite(LED_BUILTIN, HIGH);
18     delay(1000);
19     digitalWrite(LED_BUILTIN, LOW);
20     delay(1000);
21 }
22
23 void readButtonSetLed()
24 {
25     if (digitalRead(RED_BUTTON) == LOW)
26         digitalWrite(LED_RED, HIGH);
27     else
28         digitalWrite(LED_RED, LOW);
29 }
30
31 void loop()
32 {
33     blinkBuiltInLed();
34     readButtonSetLed();
35 }
```

Funkcja `readButtonSetLed()` działa stosunkowo szybko. Wykonanie funkcji `blinkBuiltInLed()` trwa jednak dwie sekundy, przez które, z punktu użytkownika, nic innego nie dzieje się. Proszę ten program uruchomić i sprawdzić.

## 2.3 Jak nie blokować wzajemnie wykonywanych zadań

Samo zapalenie, czy zgaszenie diody zachodzi stosunkowo szybko. Musimy zatem przepisać funkcję `blinkBuiltInLed()` w taki sposób, aby co sekundę zmieniała stan diody wbudowanej, ale nie blokowała całego programu.

Skorzystamy z funkcji `millis()` (Jest też jej odpowiednik do krótszych czasów: `micros()`).

Problem można spróbować rozwiązać jak w kodzie 5 (zmodyfikowano funkcję `blinkBuiltInLed()`).

Kod 5: Miganie diodą i odczyt przycisku równocześnie - Nieblokująca funkcja migania.

```
1 #define LED_RED 6
2 #define RED_BUTTON 2
3
4 void setup()
5 {
6     pinMode(LED_BUILTIN, OUTPUT);
7     digitalWrite(LED_BUILTIN, LOW);
8
9     pinMode(LED_RED, OUTPUT);
10    digitalWrite(LED_RED, LOW);
11
12    pinMode(RED_BUTTON, INPUT_PULLUP);
13 }
14
15 void blinkBuiltInLed()
16 {
17     const unsigned long BlinkChangePeriod = 1000UL;
18     static int ledState = LOW;
```

```

19 static unsigned long lastBlinkChange = 0UL;
20
21 if (millis() - lastBlinkChange >= BlinkChangePeriod)
22 {
23     if (ledState == HIGH)
24     {
25         ledState = LOW;
26     }
27     else
28     {
29         ledState = HIGH;
30     }
31
32     digitalWrite(LED_BUILTIN, ledState);
33     lastBlinkChange += BlinkChangePeriod;
34 }
35 }
36
37 void readButtonSetLed()
38 {
39     if (digitalRead(RED_BUTTON) == LOW)
40         digitalWrite(LED_RED, HIGH);
41     else
42         digitalWrite(LED_RED, LOW);
43 }
44
45 void loop()
46 {
47     blinkBuiltInLed();
48     readButtonSetLed();
49 }

```

Funkcja jest nieco bardziej skomplikowana. Sprawdza czy upłynął czas przełączenia diody. Jeśli nie to natychmiast oddaje sterowanie do funkcji `loop()`. Jeśli czas upłynął to przełącza diodę i również oddaje sterowanie do funkcji `loop()`. Czyli zajmuje tyle czasu, ile jest wymagane na sprawdzenie potrzeby przełączenia i ewentualnego przełączenia.

Mamy tu też drugą korzyść. Czasy następnego przełączenia obliczane są jako dokładna wielokrotność wartości stałej `BlinkChangePeriod`. I nawet jeśli z jakiegoś powodu wywołanie funkcji spóźni się i zmiana stanu diody też, to kolejna zmiana zaplanowana będzie prawidłowo. W przypadku wersji z funkcją `delay()` opóźnienia kumulują się.

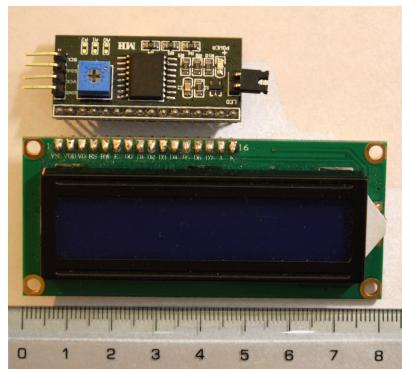
## Zadanie 2: Program wykonujący współbieżnie różne zadania

Napisz program, który będzie zmieniał stany kolorowych diod w diodzie RGB co 0,9s czerwonej, 1,0s zielonej i 1,1s niebieskiej. Program przygotować tak, aby nie blokować przełączania diod wzajemnie, nie blokować wykonania programu. Program ma pozwalać na dodanie kolejnych zadań w funkcji `loop()`, które będą wykonywane bez znacznego opóźnienia. Jest to zadanie przykładowe. Prowadzący może zmodyfikować lub zmienić treść zadania.

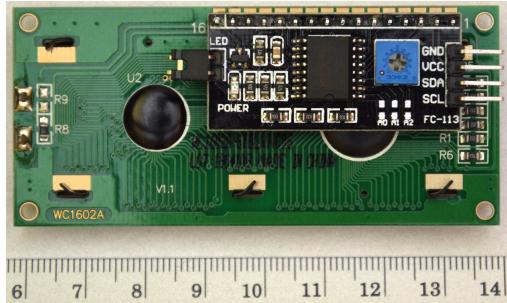
## 3 Wyświetlacz LCD

### 3.1 Wyświetlacz LCD w zestawie laboratoryjnym

Zestaw laboratoryjny Arduino wyposażony jest w jeden z najpowszechniej wykorzystywanych wyświetlaczów znakowych LCD. Wyświetla on po 16 znaków w 2 liniach (16x2). Zdjęcie wyświetlacza z zestawu laboratoryjnego znajduje się na fotografiach 1 i 2. Sam wyświetlacz może być podłączony do płytki Arduino bezpośrednio, ale wtedy wykorzystuje dużą liczbę pinów. Aby zmniejszyć liczbę pinów wykorzystywanych przez wyświetlacz skorzystano z magistrali I2C, która jest dostępna na płytce Arduino. Potrzebny jest jednak dodatkowy układ elektroniczny, który będzie konwertował sygnały I2C do wymaganych przez wyświetlacz. Jest on również widoczny na fotografiach 1 i 2.



Fotografia 1: Wyświetlacz LCD 16x2 (dół fotografii) i konwerter magistrali I2C (góra).



Fotografia 2: Wyświetlacz LCD 16x2(widok z tyłu) z zamontowanym konwerterem I2C.

Warto wiedzieć, że samym wyświetlaczem ciekłokrystalicznym steruje układ scalony HD44780 lub układ z nim kompatybilny zamontowany na płytce drukowanej wyświetlacza. Warto o tym wiedzieć, ponieważ sterownik HD44780 (lub kompatybilny) wykorzystywany jest w wielu modelach wyświetlaczy tekstowych różnych producentów i o innej geometrii niż 16x2 również. Ze względu na jego popularność nie ma problemów ze znalezieniem informacji na temat jego użycia w Internecie. Proponuję zatrzymać się na Wikipedię do artykułu [Hitachi HD44780 LCD controller](#) ([link](#)) oraz do noty katalogowej dostępnej, na przykład, na witrynie [SparkFun Electronics](#) ([link](#)).

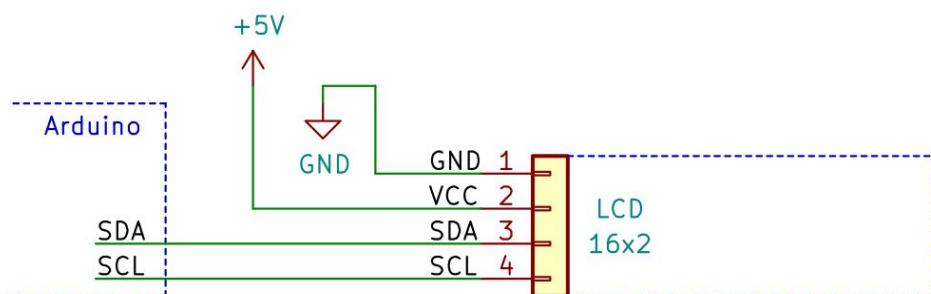
### 3.2 Magistrala I2C

Magistralę I2C zaprojektowano do połączenia wielu mikrokontrolerów, a także wielu urządzeń podłączanych, na tej samej magistrali dwuprzewodowej.

Wykorzystuje dwa połączenia sygnałowe

- sygnał zegarowy (SCL - serial clock),
- dane szeregowe (SDA - serial data).

Przełączenie wyświetlacza z adapterem do płytki Arduino w zestawie laboratoryjnym przedstawia schemat 1.



Schemat 1: Podłączenie modułu LCD z adapterem I2C do płytki Arduino w zestawie laboratoryjnym.

Literatura na temat tej magistrali jest obszerna i łatwo dostępna. Poszukującym informacji proponuję zacząć od [I2C – What’s That?](#) ([link](#)) oraz [UM10204: I2C-bus specification and user manual](#) ([link](#)).

### 3.3 Sterowanie wyświetlaczem LCD w zestawie laboratoryjnym.

Do programowania wyświetlacza podłączonego poprzez magistralę I2C proponuje się bibliotekę [Liquid Crystal I2C](#) by Marco Schwarz ver. 1.1.2. Biblioteka definiuje klasę `LiquidCrystal_I2C` o następującym interfejsie (wybrane ważniejsze metody, pełna informacja w kodzie źródłowym):

- `LiquidCrystal_I2C(uint8_t lcd_Addr,uint8_t lcd_cols,uint8_t lcd_rows)`  
Konstruktor obiektu wyświetlacza, który wymaga podania adresu uwyświetlacza na magistrali I2C, liczby wierszy i kolumn. Dla zestawu laboratoryjnego to 0x27, 16, 2.
- `void init()` - Inicjalizuj wyświetlacz.
- `void backlight() / void noBacklight()` - Wyłącz/wyłącz podświetlenie wyświetlacza.
- `void display() / void noDisplay()` - Włącz/wyłącz wyświetlanie tekstu (szybko). To nie jest zupełne wyłączenie i nie trzeba ponownie wysyłać tekstu.
- `void clear()` - Wyczyszczenie wyświetlacza, ustaw pozycję kurSORA na zero. Zajmuje dużo czasu!
- `void home()` - Ustaw pozycję kurSORa na zero. Zajmuje dużo czasu!

- `void setCursor(uint8_t col, uint8_t row)` - Ustawia kurs w zadanej kolumnie i wierszu. Lewy górny róg ekranu wyświetlacza ma współrzędne (0, 0), a prawy dolny (15,1). Kursor może być niewidoczny, w postaci podkreślenia, migania pola znaku lub podkreślenia i migania pola znaku równocześnie..
- `void cursor() / void noCursor()` - Włącz/wyłącz kurSOR w postaci podkreślenia.
- `void blink() / void noBlink()` - Włącz/wyłącz miganie całego pola znaku w miejscu kurSora.
- `void scrollDisplayLeft() / void scrollDisplayRight()` - Przesuń (przewiń) zawartość wyświetlacza bez zmiany pamięci RAM w lewo/prawo. Każda linia ma bufor 40 znaków i można po nim przesuwać się. Więcej informacji w [karcie katalogowej \(link\)](#).
- `void createChar(uint8_t, uint8_t[])` - Pozwala zdefiniować własny znak. Wyświetlacz w zestawie laboratoryjnym ma znaki o rozmiarze 5 na 8 pikseli. Takich własnych znaków można zdefiniować 8 i w ten sposób wprowadzić, na przykład, polskie znaki. Więcej informacji w [karcie katalogowej \(link\)](#). Przykład takiej definicji i użycia zawiera kod 6.

Ponadto klasa `LiquidCrystal_I2C` dziedziczy po klasie `Print` i można używać, miedzy innymi, następujących metod odziedziczonych po klasie `Print`:

- `size_t print(const String &)`
- `size_t print(const char[])`
- `size_t print(char)`
- `size_t print(unsigned char, int = DEC)`
- `size_t print(int, int = DEC)`
- `size_t print(unsigned int, int = DEC)`
- `size_t print(long, int = DEC)`
- `size_t print(unsigned long, int = DEC)`
- `size_t print(double, int = 2)`

przy czym, specyfikatorem formatu liczby w drukowanym łańcuchu tekstowym dla liczb całkowitych są: DEC, HEX, OCT i BIN. Działanie tych metod jest oczywiste, więc dodatkowe wyjaśnienia są zbędne. Zwróćmy jednak uwagę, że metody `println()`, choć zdefiniowane w klasie `Print`, na wyświetlaczu LCD nie działają prawidłowo.

Uwaga praktyczna do programowania: Jeśli należy zmienić tylko fragment wyświetlonego tekstu, nie należy czyścić zawartości całego wyświetlacza i pisać od nowa, ale ustawić kurSOR na pozycji, od której ma nastąpić zmiana i tylko tam dokonywać zmian. Będzie to działać szybciej. Wykonanie funkcji `clear()` zajmuje stosunkowo dużo czasu, a potem trzeba wysłać jeszcze pełną zawartość tekstu na wyświetlacz. Tej wady nie posiada zmiana w miejscu modyfikacji zawartości, choć trzeba uważać, szczególnie przy liczbach, gdzie ilość cyfr się zmienia, aby nie pozostały fragmenty poprzednich wartości na ekranie. Przykładowo, liczba z 3 cyframi, nie nadpisze do końca liczby z 4 cyframi, jeśli nie podejmimy dodatkowych kroków. Należy o tym pamiętać i zatroszczyć się w kodzie programu, aby do tego nie doszło.

Przykład programu wypisującego na ekranie wyświetlacza LCD w zestawie laboratoryjnym napis „Hello World!” w dwóch linijkach przedstawiono jako kod 6.

Kod 6: Program typu „Hello World!” dla wyświetlacza LCD w zestawie laboratoryjnym.

```

1 #include <LiquidCrystal_I2C.h>
2
3 LiquidCrystal_I2C lcd(0x27, 16, 2);
4
5 byte smile[8] = {
6     B10101,
7     B00000,
8     B00000,
9     B10001,
10    B00000,
11    B00000,
12    B10001,
13    B01110,
14 };
15
16 void setup()
17 {
18     lcd.init();
19     lcd.backlight();
20
21     lcd.createChar(0, smile);
22 }
```

```
23    lcd.setCursor(0, 0);
24    lcd.print("Hello");
25    lcd.setCursor(3, 1);
26    lcd.print("World!");
27
28    lcd.setCursor(15, 0);
29    lcd.write(byte(0));
30 }
31
32 void loop()
33 {
34 }
```

### Zadanie 3: Wykorzystanie wyświetlacza LCD - Program „Stoper elektroniczny”

Napisz program, który będzie działał jako stoper elektroniczny, podając czas na wyświetlaczu LCD. Klawisz zielony uruchamia i zatrzymuje stoper. Klawisz czerwony zatrzymuje stoper jeśli działał, i resetuje wartość czasu do wartości 0. Precyza pomiaru to 1 sekunda.

Jest to zadanie przykładowe. Prowadzący może zmodyfikować lub zmienić treść zadania.