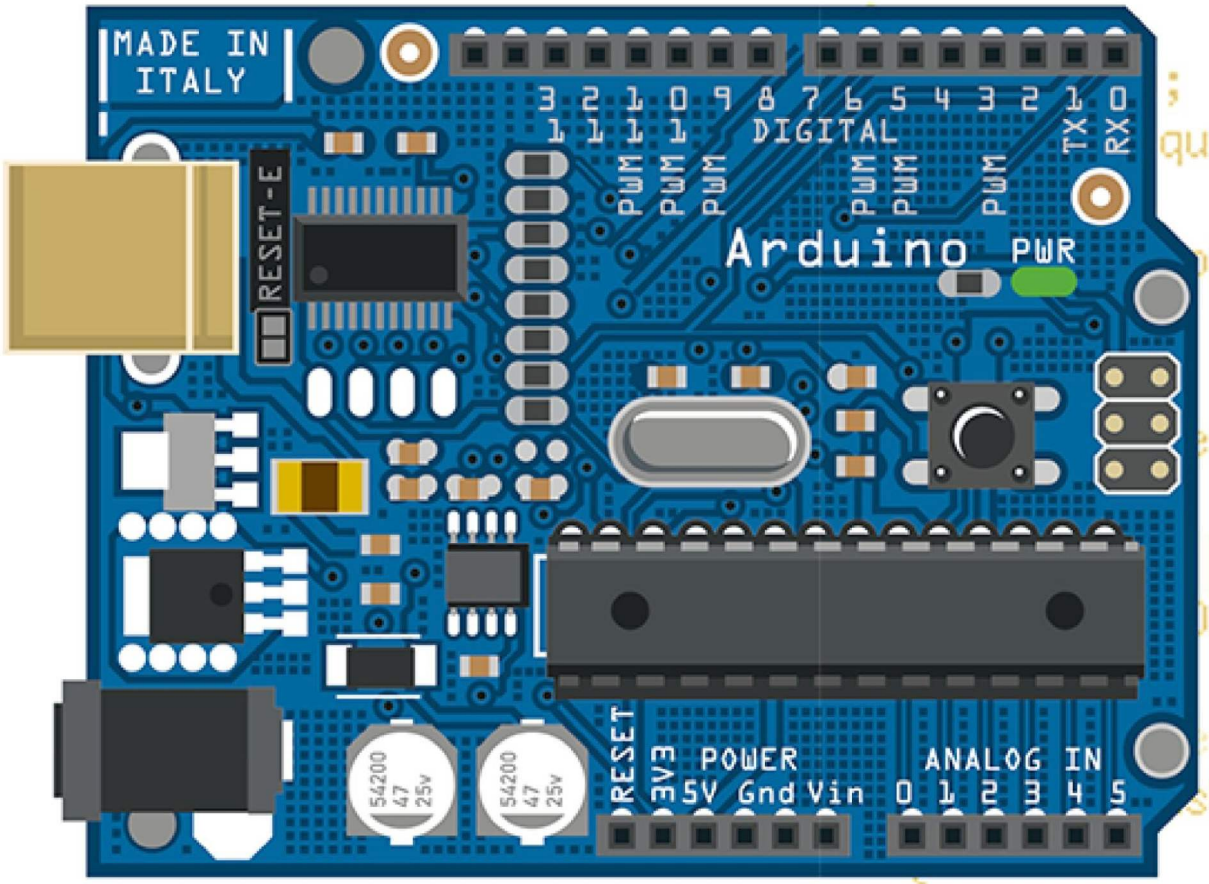
Arduino
dla początkujących
Podstawy i szkice

Podręcznik programisty Arduino!



Simon Monk

Tytuł oryginału: Programming Arduino Getting Started with Sketches
Tłumaczenie: Konrad Matuk
ISBN: 978-83-246-8737-4

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)

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

Printed in Poland.

Jako dumny ojciec książkę dedykuję moim chłopcom, Stephenowi i Matthew.

Spis treści

O autorze ... 9
Podziękowania ... 10
Wstęp ... 11
Czym jest Arduino? ... 11
Co będzie potrzebne? ... 12
Korzystanie z niniejszej książki ... 12
Pomoce ... 13

Rozdział 1. Oto Arduino ... 15
Mikrokontrolery ... 15
Płyty rozwojowe ... *16*
Płyta Arduino ... 17
Zasilanie ... *17*
Złącza zasilania ... *17*
Wejścia analogowe ... *18*
Złącza cyfrowe ... *18*
Mikrokontroler ... *18*
Pozostałe podzespoły ... *19*
Początki Arduino ... 19
Rodzina płyt Arduino ... 21
Uno, Duemilanove i Diecimila ... *21*
Mega ... *22*
Nano ... *22*
Bluetooth ... *23*
Lilypad ... *24*
Inne „oficjalne" płytki ... *24*
Inne klony i odmiany Arduino ... 25
Podsumowanie ... 25

Rozdział 2. Rozpoczynamy przygodę z Arduino 27
Zasilanie 27
Instalacja oprogramowania 28
Ładowanie pierwszego szkicu 28
Aplikacja Arduino 33
Podsumowanie 34

Rozdział 3. Podstawy języka C 35
Programowanie 35
Czym jest język programowania? 36
Blink po raz kolejny 40
Zmienne 42
Eksperymentowanie w języku C 44
Zmienne numeryczne i arytmetyka *45*
Polecenia 47
if *47*
for *49*
while *51*
Dyrektywa #define 52
Podsumowanie 52

Rozdział 4. Funkcje 53
Czym jest funkcja? 53
Parametry 54
Zmienne globalne, lokalne i statyczne 55
Zwracanie wartości 58
Zmienne innych typów 59
float *59*
boolean *60*
Inne typy danych *61*
Styl zapisu kodu 62
Wcięcia *62*
Nawiasy klamrowe otwierające *63*
Białe znaki *63*
Komentarze *64*
Podsumowanie 65

Rozdział 5. Tablice i łańcuchy 67
Tablice 67
Zastosowanie tablic do alfabetu Morse'a i sygnału SOS *70*
Tablice łańcuchów 71
Literały łańcuchowe *71*
Zmienne łańcuchowe *72*
Tłumacz alfabetu Morse'a 73
Dane *73*
Zmienne globalne i funkcja setup *74*

Funkcja loop ... *75*
Funkcja flashSequence ... *77*
Funkcja flashDotOrDash ... *78*
Składanie całości programu ... *78*
Podsumowanie ... 80

Rozdział 6. Wejścia i wyjścia ... 81
Wyjścia cyfrowe ... 81
Wejścia cyfrowe ... 84
Rezystor podwyższający ... *85*
Wewnętrzny rezystor podwyższający ... *88*
Usuwanie stuków ... *88*
Wyjścia analogowe ... 93
Wejścia analogowe ... 95
Podsumowanie ... 96

Rozdział 7. Standardowa biblioteka Arduino ... 97
Liczby losowe ... 97
Funkcje matematyczne ... 99
Manipulacja bitami ... 99
Zaawansowane funkcje wejścia i wyjścia ... 102
Generowanie tonów ... *102*
Wprowadzanie rejestru przesuwnego ... *103*
Przerwania ... 103
Podsumowanie ... 105

Rozdział 8. Zapisywanie danych ... 107
Stałe ... 107
Dyrektywa PROGMEM ... 108
EEPROM ... 109
Przechowywanie wartości zmiennej typu int w pamięci EEPROM ... *110*
Przechowywanie wartości typu float w pamięci EEPROM (unie) ... *110*
Przechowywanie łańcucha w pamięci EEPROM ... *111*
Wymazywanie zawartości pamięci EEPROM ... *112*
Kompresja ... 112
Kompresja zakresu ... *112*
Podsumowanie ... 113

Rozdział 9. Wyświetlacze LCD ... 115
Tablica wyświetlająca komunikaty za pośrednictwem interfejsu USB ... 116
Korzystanie z wyświetlacza ... 118
Inne funkcje biblioteki wyświetlacza LCD ... 119
Podsumowanie ... 120

Rozdział 10. Programowanie aplikacji sieci Ethernet 121
Płytki pozwalające na pracę w sieci Ethernet 122
Komunikacja z serwerami sieciowymi 122
HTTP *122*
HTML *122*
Arduino w roli serwera sieci Web 123
Konfigurowanie złączy Arduino za pośrednictwem sieci 126
Podsumowanie 131

Rozdział 11. C++ i biblioteki 133
Mechanizmy obiektowe 133
Klasy i metody *133*
Przykład wbudowanej biblioteki 134
Tworzenie bibliotek 134
Plik nagłówkowy *134*
Plik implementacji *136*
Uzupełnianie swojej biblioteki *137*
Podsumowanie 139

Skorowidz 141

O autorze

Simon Monk jest inżynierem cybernetykiem i informatykiem. Posiada doktorat z zakresu inżynierii oprogramowania. Od dzieciństwa interesował się elektroniką. Jest autorem artykułów zamieszczanych na łamach czasopism dla elektroników hobbistów. Napisał także książki *Arduino + Android Projects for the Evil Genius* oraz *15 Dangerously Mad Projects for the Evil Genius*.

Podziękowania

Dziękuję Lindzie za poświęcanie czasu i wspieranie mnie podczas pracy nad tą książką. Dziękuję za radzenie sobie z bałaganem, który powstaje w domu, gdy pracuję nad swoimi projektami.

Dziękuję również moim dwóm synom za zainteresowanie moją pracą, a także za udzieloną mi pomoc.

Na koniec chciałbym podziękować Rogerowi Stewartowi i Sapnie Rastogi, a także każdej innej osobie związanej z tworzeniem niniejszej książki. Praca w tak dobrym zespole była przyjemnością.

Wstęp

Interfejs Arduino jest tanią i łatwą w użyciu technologią pozwalającą na tworzenie projektów opartych o mikrokontrolery. Wykorzystując podstawy elektroniki, możesz sprawić, że Twoje Arduino będzie sterowało oświetleniem podczas wystawy sztuki lub zarządzało energią wytwarzaną przez panele słoneczne.

Powstało wiele książek skupiających się na projektach korzystających z Arduino. Jedną z takich książek jest napisana przeze mnie *30 Arduino Projects for the Evil Genius*. Niniejsza książka skupia się na tematyce związanej z programowaniem Arduino.

Dowiesz się, jak uczynić programowanie Arduino czynnością prostą i przyjemną oraz jak uniknąć trudności związanych z niedziałającym kodem programu, który jest źródłem wielu problemów podczas tworzenia projektów. Proces programowania Arduino zostanie opisany krok po kroku. Poznasz także podstawy języka C — języka, z którego korzysta Arduino.

Czym jest Arduino?

Arduino jest mikrokontrolerem mającym postać płytki wyposażonej w złącze uniwersalnej magistrali szeregowej (USB) służące do komunikacji z komputerem, a także inne złącza służące do podłączania zewnętrznych elementów elektronicznych takich jak silniki, przekaźniki, fotodiody, diody laserowe, głośniki, mikrofony itp. Urządzenia podłączone do Arduino mogą być zasilane prądem pobieranym ze złącza USB, ogniwa 9 V lub zewnętrznego zasilacza. Urządzenia te mogą być sterowane za pośrednictwem komputera lub — po uprzednim zaprogramowaniu i odłączeniu od komputera — przez samo Arduino.

Projekt Arduino jest otwarty. Każdy może wykonywać układy zgodne z Arduino. Dzięki temu płytki układu Arduino są tanie.

Podstawowe płytki Arduino są dostarczane wraz z dodatkowymi płytkami, które mogą być wpinane na wierzch płytki Arduino. Będziemy korzystać z dwóch takich płytek — wyświetlacza LCD, a także interfejsu sieci Ethernet. Pozwoli to na stworzenie miniaturowego serwera sieci web.

Oprogramowanie służące do programowania Arduino jest łatwe w użyciu i jest dostępne za darmo na platformy Windows, Mac i Linux.

Co będzie potrzebne?

Niniejsza książka jest przeznaczona dla osób początkujących, jednakże będzie ona również przydatna dla tych, którzy korzystają już z Arduino i chcą dowiedzieć się więcej na temat programowania tego mikrokontrolera.

Nie musisz posiadać doświadczenia w zakresie programowania. Nie będzie Ci potrzebna również duża wiedza techniczna. Ćwiczenia zaprezentowane w książce nie wymagają od czytelnika tworzenia połączeń lutowniczych. Wymagają jedynie chęci do pracy twórczej.

Jeżeli jednakże chcesz w pełni skorzystać z niniejszej książki, powinieneś mieć pod ręką:

- kawałek jednożyłowego kabla elektrycznego,
- tani multimetr cyfrowy.

Obie te rzeczy można nabyć za kilkanaście złotych w każdym sklepie dla elektroników hobbystów. Oczywiście będziesz również potrzebować układu Arduino Uno.

Jeżeli chcesz posunąć się dalej i eksperymentować z siecią Ethernet oraz wyświetlaczem ciekłokrystalicznym, musisz zakupić odpowiednie płytki za pośrednictwem sklepów internetowych. Więcej informacji na ten temat znajdziesz w rozdziałach 9. i 10.

Korzystanie z niniejszej książki

Niniejsza książka pozwala na naukę podstaw, a następnie rozbudowywanie swojej wiedzy o kolejne zagadnienia. Jeżeli jesteś już obeznany z pewnymi zagadnieniami, możesz przeskoczyć bezpośrednio do kolejnych rozdziałów.

Książka została podzielona na następujące rozdziały:

- **Rozdział 1: Oto Arduino** — w rozdziale tym przedstawiono podstawowe informacje o sprzętowym aspekcie Arduino. Opisano możliwości sprzętowe Arduino, a także różne dostępne wersje tego układu.
- **Rozdział 2: Rozpoczynamy przygodę z Arduino** — w rozdziale tym przedstawiono podstawowe czynności związane z Arduino: instalowanie oprogramowania, zasilanie układu, a także ładowanie pierwszego szkicu.

- **Rozdział 3: Podstawy języka C** — w rozdziale tym opisano podstawy języka C. Laikom w dziedzinie programowania rozdział ten może służyć za ogólny wstęp do programowania.
- **Rozdział 4: Funkcje** — w rozdziale tym wyjaśniono tworzenie i stosowanie funkcji w szkicach przeznaczonych dla Arduino. Działanie szkiców zademonstrowano przy użyciu przykładów wykonywalnego kodu.
- **Rozdział 5: Tablice i łańcuchy** — w tym rozdziale dowiesz się, jak tworzyć struktury danych bardziej złożone od obiektów całkowitoliczbowych oraz jak korzystać z takich struktur. W celu wyjaśnienia tych zagadnień tworzony będzie przykładowy projekt związany z obsługą alfabetu Morse'a.
- **Rozdział 6: Wejścia i wyjścia** — po przeczytaniu tego rozdziału dowiesz się, jak należy korzystać z analogowych wejść i wyjść Arduino. W celu analizy tego, co się dzieje na złączach analogowych wejść i wyjść mikrokontrolera, przydatny będzie cyfrowy multimetr.
- **Rozdział 7: Standardowa biblioteka Arduino** — rozdział ten wyjaśnia sposób korzystania ze standardowych funkcji Arduino zdefiniowanych w standardowej bibliotece Arduino.
- **Rozdział 8: Zapisywanie danych** — rozdział ten opisuje tworzenie szkiców, które mogą być następnie zapisane w programowalnej pamięci stałej wymazywalnej elektrycznie (EEPROM). Dowiesz się, jak można korzystać z wbudowanej w Arduino pamięci flash.
- **Rozdział 9: Wyświetlacze LCD** — w tym rozdziale będziesz tworzyć, przy użyciu biblioteki obsługującej płytkę stykową wyświetlacza LCD, projekt wyświetlacza wiadomości tekstowych obsługiwanego za pośrednictwem złącza USB.
- **Rozdział 10: Programowanie aplikacji sieci Ethernet** — dowiesz się, jak sprawić, by Arduino zachowywało się jak serwer sieciowy. Poznasz podstawy protokołu HTTP, a także tworzenia dokumentów HTML.
- **Rozdział 11: C++ i biblioteki** — wyjdziemy poza ograniczenia języka C. Poznasz elementy programowania obiektowego. Dowiesz się, jak tworzyć własne biblioteki Arduino.

Pomoce

Na stronie *http://www.helion.pl/ksiazki/ardupo.htm* znajdziesz między innymi kod źródłowy, z którego korzystamy w niniejszej książce, oraz uzupełnianą w razie konieczności erratę.

Rozdział 1.

Oto Arduino

Arduino jest mikrokontrolerem. Platforma Arduino została bardzo ciepło przyjęta przez miłośników elektroniki. Jest ona otwarta i łatwa w obsłudze, a więc może się przydać każdej osobie, która chce zrealizować swój własny projekt związany z elektroniką.

Arduino kontroluje urządzenia podłączone do swoich złączy. Na przykład włącza i wyłącza światła lub silniki. Może być również stosowane do odczytu temperatury lub ilości padającego światła. Z tego powodu Arduino określane jest czasem mianem **fizycznego komputera.** Wszystkie elementy podłączone do Arduino mogą być również sterowane z poziomu komputera, do którego podłączony jest ten mikrokontroler. W tym wypadku Arduino odgrywa rolę interfejsu.

Niniejszy rozdział stanowi wstęp do Arduino. Opisano tutaj rys historyczny tego urządzenia, a także omówiono jego budowę.

Mikrokontrolery

Sercem Arduino jest mikrokontroler. Wszystkie pozostałe elementy na płytce układu zajmują się zasilaniem układu lub pozwalają na komunikację układu z komputerem.

Mikrokontroler jest tak naprawdę małym komputerem zamkniętym w pojedynczym chipie. Ma on wszystko to, co miały pierwsze komputery przeznaczone do użytku domowego, a nawet więcej. Arduino posiada procesor, kilobajt lub dwa kilobajty pamięci o dostępie bezpośrednim (RAM) służącej do przechowywania danych, kilka kilobajtów pamięci stałej programowalnej elektrycznie (EEPROM) lub pamięci flash służącej do przechowywania programów, a także złącza wejścia i wyjścia. Złącza te służą do łączenia mikrokontrolera z innymi elementami elektronicznymi. Wejścia mogą odbierać zarówno sygnał cyfrowy (stan zwarcia i rozwarcia), jak i analogowy (napięcie na złączu). Pozwala to między innymi na podłączenie różnorodnych czujników temperatury, czujników ciśnienia akustycznego oraz światłomierzy.

Wyjścia mogą działać również w trybie cyfrowym lub analogowym. Możesz skonfigurować dany pin tak, aby działał w trybie zwarcia lub rozwarcia (0 woltów lub 5 woltów). A więc można w ten sposób bezpośrednio włączać lub wyłączać diodę elektroluminescencyjną (LED). Wyjścia mogą być również stosowane do sterowania urządzeniami większej mocy, takimi jak silniki. Złącza wyjściowe mogą również działać w trybie analogowym. A więc można ustalić jakąś wartość napięcia na danym złączu, a tym samym sterować prędkością obrotową silnika albo jasnością diody.

Mikrokontroler zainstalowany na płytce Arduino jest dwudziestoośmiostykowym chipem umieszczonym w gnieździe znajdującym się na środku płytki. Ten pojedynczy chip posiada wbudowane układy procesora oraz pamięci. Wbudowano w nim również całą elektronikę odpowiedzialną za styki wejścia i wyjścia. Firma Atmel — jeden z największych producentów mikrokontrolerów — jest odpowiedzialna za produkcję tego układu. Każdy producent mikrokontrolerów produkuje różne układy zgrupowane w rodziny. Nie wszystkie mikrokontrolery są tworzone z myślą o elektronikach hobbystach. Jesteśmy tylko niewielką cząstką tego ogromnego rynku. Mikrokontrolery są tworzone w celu stosowania ich w takich produktach konsumpcyjnych jak samochody, pralki, odtwarzacze DVD, zabawki, a nawet odświeżacze powietrza.

Dużą zaletą Arduino jest to, że ogranicza on tę zatrważającą liczbę kontrolerów możliwych do wyboru do jednego wystandaryzowanego układu. Później dowiesz się, że zdanie to nie jest do końca prawdziwe, ale na chwilę obecną możemy je za takie uznać.

Oznacza to, że rozpoczynając pracę nad nowym projektem, nie musisz zastanawiać się nad zaletami i wadami mikrokontrolerów różnych typów.

Płyty rozwojowe

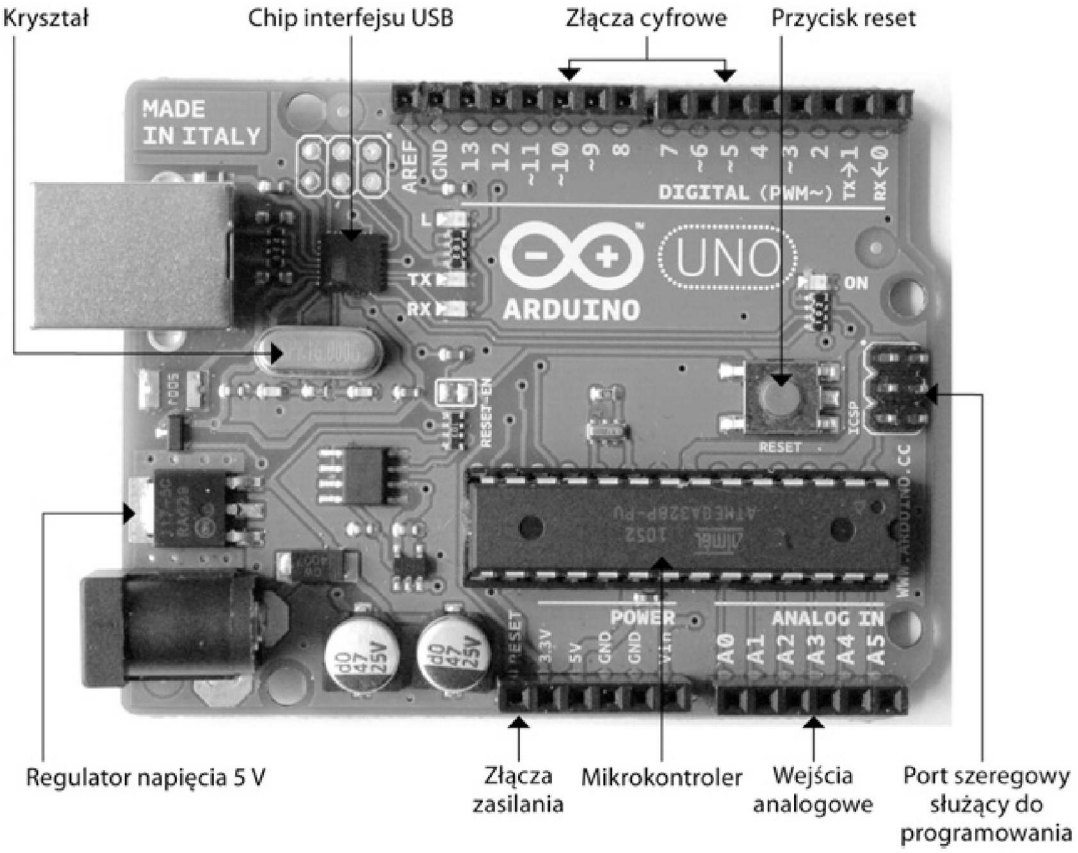
Powiedzieliśmy, że mikrokontroler to tak naprawdę tylko układ scalony. Pojedynczy chip nie jest jednakże w stanie działać bez elektroniki odpowiednio go zasilającej (mikrokontrolery są wrażliwe na problemy związane z zasilaniem) oraz bez elektroniki pozwalającej na komunikację z komputerem programującym mikrokontroler.

Wszystkie te funkcje zapewniają płyty rozwojowe. Płyta Arduino jest tak naprawdę płytą rozwojową mikrokontrolera, której projekt jest niezależny i otwarty. Oznacza to, że pliki projektowe płytki drukowanej układu, a także jej schematy są ogólnodostępne i każdy może na ich podstawie tworzyć i sprzedawać swoje własne płytki Arduino.

Każdy producent mikrokontrolerów — w tym również Atmel produkujący mikrokontroler ATmega328 stosowany w płytach Arduino — dostarcza własne płyty rozwojowe i oprogramowanie służące do programowania układu. Co prawda rzeczy te zwykle nie są drogie, jednakże ich docelowym odbiorcą są inżynierowie, a nie elektronicy hobbyści. Takie płyty i oprogramowanie są trudniejsze w użyciu i wymagają dłuższej nauki obsługi przed uzyskaniem jakichkolwiek użytecznych rezultatów.

Płyta Arduino

Na rysunku 1.1 przedstawiono płytę Arduino. Przyjrzyjmy się elementom na niej umieszczonym.



Rysunek 1.1. *Płyta Arduino Uno*

Zasilanie

Popatrz na rysunek 1.1. Bezpośrednio pod złączem USB znajduje się regulator napięcia 5 V. Układ ten generuje stałe napięcie 5 V niezależnie od podanego mu napięcia (w zakresie od 7 V do 12 V).

Układ regulatora napięcia jest dość duży jak na element montowany powierzchniowo. Układ ten ma taki rozmiar, ponieważ ułatwia to rozproszenie ciepła powstającego podczas regulacji napięcia przy dużych prądach. Jest to przydatne podczas zasilania urządzeń zewnętrznych.

Złącza zasilania

Następnie przyjrzyjmy się złączom zasilania znajdującym się na dole rysunku 1.1. Z rysunku możesz odczytać nazwy złączy. Pierwsze złącze jest oznaczone jako „Reset" — ma ono taką samą funkcję jak przycisk *Reset*. Podobnie jak ponowne uruchomienie komputera, skorzystanie z przycisku *Reset* powoduje rozpoczęcie wykonywania programu przez mikrokontroler od początku. Aby zresetować mikrokontroler, należy połączyć na chwilę pin Reset z pinem masy.

Pozostałe piny w tej sekcji dostarczają, zgodnie z oznaczeniami, prąd o odpowiednim napięciu (3,3 V, 5 V, GND i 9 V). Złącze GND, zwane również masą, dostarcza napięcie 0 V. Jest to napięcie, do którego odnoszą się wszystkie pozostałe napięcia na płytce układu.

Wejścia analogowe

Sześć złączy opisanych jako *Analog In* o oznaczeniach od A0 do A5 można stosować do pomiaru przyłożonego do nich napięcia. Wynik pomiaru może zostać wykorzystany w szkicu. Zauważ, że mierzone jest napięcie, a nie prąd. Pomiędzy tymi złączami a masą mogą płynąć tylko niewielkie prądy. Dzieje się tak, ponieważ złącza te posiadają duży opór wewnętrzny.

Złącza te są oznaczone jako złącza analogowe, jednakże są one po prostu złączami działającymi domyślnie w trybie analogowym. Złącza te mogą również pracować jako cyfrowe wejścia lub wyjścia.

Złącza cyfrowe

Teraz zajmiemy się omówieniem złączy pokazanych w prawym górnym rogu rysunku 1.1. Znajdują się tam piny o numerach od 0 do 13. Złącza te mogą służyć zarówno jako wejścia, jak i wyjścia. Działając jako wyjścia, zachowują się podobnie do opisanych wcześniej źródeł napięcia; jedyną różnicą jest to, że podają napięcie 5 V i można je włączać lub wyłączać z poziomu szkicu. Złącza te po wyłączeniu będą posiadać potencjał 0 V. Stosując te złącza jako źródła prądowe, należy uważać na to, aby ich zbytnio nie obciążać. Pierwsze dwa złącza (oznaczone 0 i 1) posiadają również oznaczenia RX (odbiór) i TX (transmisja). Złącza te są stosowane do transmisji danych. Są one pośrednio połączone ze złączem USB służącym do komunikacji z komputerem.

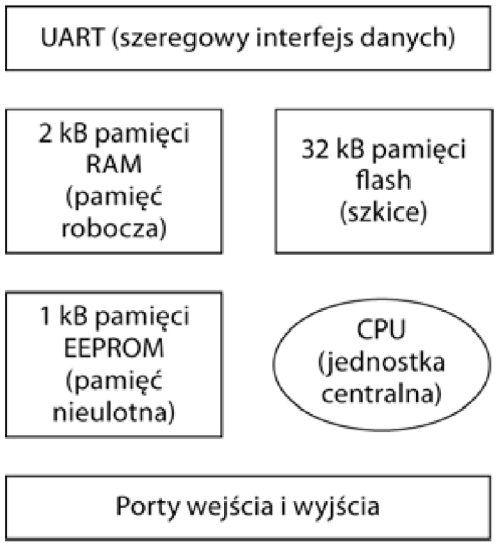
Omawiane złącza mogą zapewnić prąd o maksymalnym natężeniu 40 mA (miliamperów) przy napięciu 5 V. Jest to wystarczająca wartość do zasilania standardowej diody LED, jednakże jest to zbyt mały prąd, by bezpośrednio zasilać silnik elektryczny.

Mikrokontroler

Opisując kolejne elementy znajdujące się na płytce Arduino, dochodzimy do samego chipu mikrokontrolera. Jest to prostokątny układ scalony posiadający 28 styków. Układ ten znajduje się w dwurzędowym podłużnym gnieździe. Można go z łatwością wymienić. Mikrokontroler umieszczony na płytce Arduino Uno to ATmega328. Na rysunku 1.2 przedstawiono diagram blokowy ilustrujący budowę tego układu.

Sercem, a właściwie mózgiem tego układu jest jednostka centralna (CPU). Kontroluje ona wszystko to, co dzieje się w układzie. Jednostka ta pobiera i wykonuje instrukcje zawarte w pamięci flash. Takie operacje mogą wymagać pobrania danych z pamięci roboczej (RAM), modyfikacji tych danych, a następnie zapisania ich z powrotem w pamięci roboczej. Instrukcje mogą również włączać lub wyłączać napięcie podawane na złącza cyfrowe.

Pamięć EEPROM jest nieulotna, podobnie jak pamięć flash. A więc po wyłączeniu zasilania urządzenia dane w niej zapisane nie zostaną utracone. Pamięć flash ma przechowywać instrukcje programu (szkice), a pamięć EEPROM ma przechowywać dane, których nie chcesz utracić w wyniku wciśnięcia przycisku *Reset* lub wyłączenia zasilania.



Rysunek 1.2. *Schemat blokowy układu ATmega328*

Pozostałe podzespoły

Nad procesorem znajduje się mały, srebrny, prostokątny element. Jest to kwarcowy generator drgań. Generuje on 16 milionów impulsów na sekundę. Z każdym impulsem mikrokontroler wykonuje jedną operację matematyczną (np. dodawania lub odejmowania).

Na prawo od kryształu znajduje się przycisk *Reset*. Wciśnięcie tego przycisku spowoduje wysłanie sygnału logicznego sprawiającego, że mikrokontroler wyczyści pamięć operacyjną i uruchomi od początku wykonywany program. Warto zwrócić uwagę na to, że program jest przechowywany w pamięci nieulotnej — zawartość tej pamięci nie jest kasowana w wyniku wciśnięcia przycisku *Reset* lub odłączenia zasilania.

Na prawo od przycisku *Reset* znajduje się port szeregowy służący do programowania. Pozwala on na programowanie Arduino bez potrzeby korzystania z interfejsu USB. Nie będziemy poruszać kwestii związanych z możliwością programowania Arduino poprzez port szeregowy, ponieważ Arduino posiada złącze USB i odpowiednie oprogramowanie, co pozwala na wygodną pracę.

W lewym górnym rogu płytki (obok gniazda USB) znajduje się chip interfejsu USB. Układ ten przetwarza standardowy sygnał USB na sygnał o poziomie akceptowanym przez płytkę Arduino.

Początki Arduino

Arduino powstało jako pomoc mająca służyć studentom. Następnie (w 2005 r.) Massimo Banz i David Cuartielles zaczęli komercyjnie rozwijać ten projekt. Z powodu łatwej obsługi i wytrzymałości Arduino jest szczególnie popularne wśród konstruktorów, studentów i artystów.

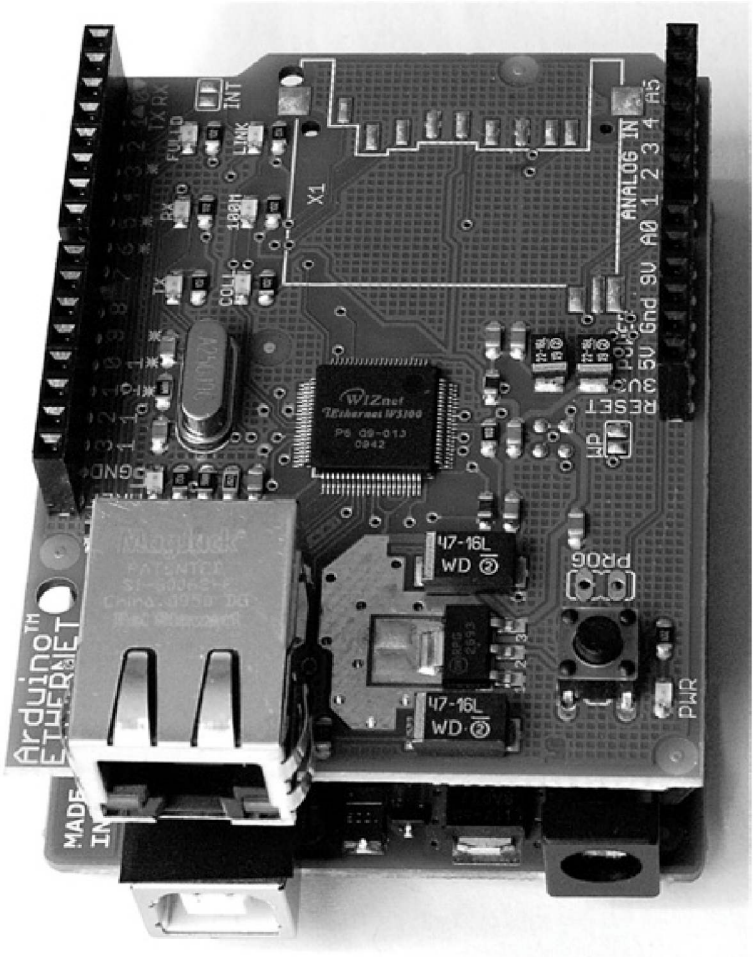
Kolejnym czynnikiem mającym wpływ na sukces tego mikrokontrolera jest to, że jego dokumentacja jest ogólnodostępna na zasadach licencji Creative Commons. Pozwoliło to na powstanie wielu tańszych odpowiedników płyty. Nazwa Arduino jest chroniona, a więc

takie klony mają często nazwy utrzymane w stylistyce „*duino" (np. Boarduino, Seeeduino i Freeduino). Oryginalna płyta, produkowana we Włoszech, wciąż się jednakże bardzo dobrze sprzedaje. Więksi dystrybutorzy sprzedają często tylko ładnie zapakowane oficjalne płyty, których jakość nie budzi zastrzeżeń.

Kolejnym czynnikiem mającym wpływ na sukces Arduino jest fakt, że Arduino nie jest ograniczone tylko do płytki mikrokontrolera. Istnieje wiele płyt stykowych kompatybilnych z Arduino. Płytki takie są wpinane bezpośrednio na płytę Arduino. Dzięki temu, że istnieje wiele takich płytek stykowych (zwanych również potocznie „shieldami"), możliwe jest uniknięcie korzystania z lutownicy. Wystarczy tylko wpiąć w siebie płytki. Poniżej wymieniono tylko najpopularniejsze płytki stykowe:

- Ethernet — zapewnia obsługę sieci.
- Motor — pozwala obsługiwać silniki elektryczne.
- Host USB — pozwala na obsługę urządzeń wyposażonych w interfejs USB.
- Moduł przekaźników — pozwala na obsługę przekaźników za pośrednictwem Arduino.

Na rysunku 1.3 pokazano Arduino Uno z założoną płytą stykową służącą do obsługi sieci Ethernet.



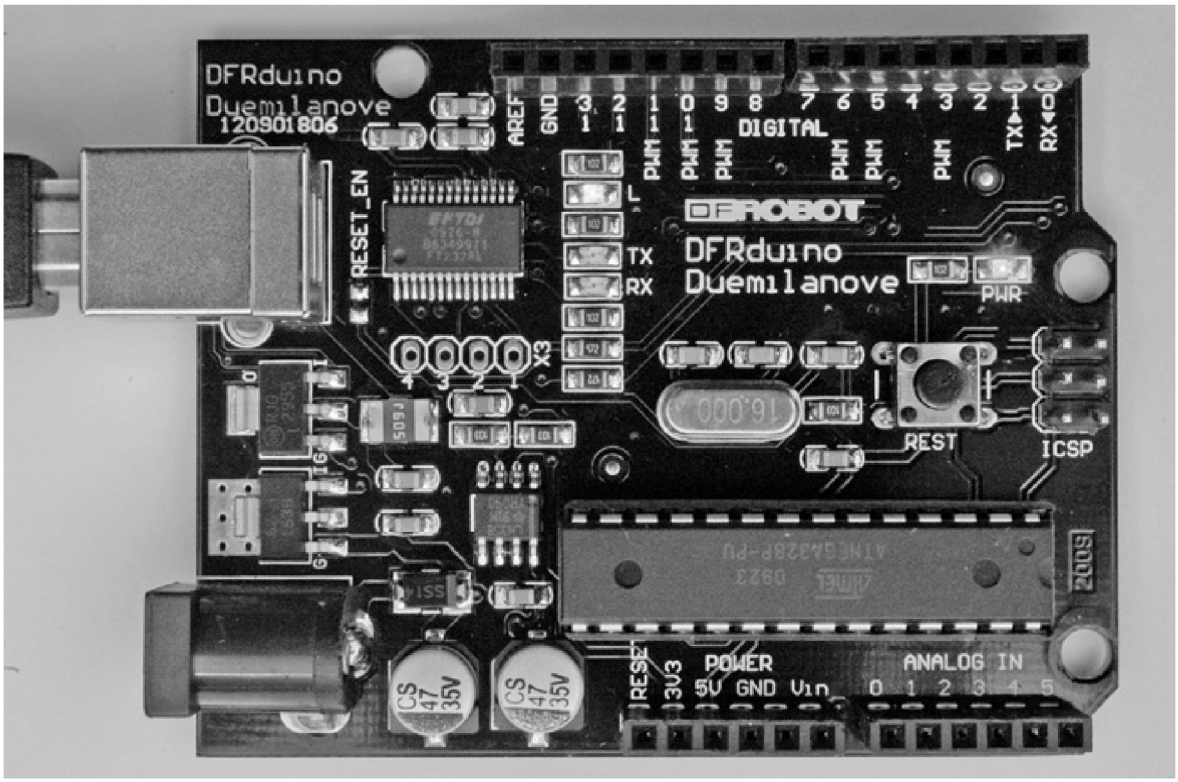
Rysunek 1.3. *Arduino Uno z płytką stykową Ethernet*

Rodzina płyt Arduino

Podstawowa wiedza na temat różnych płyt Arduino może okazać się przydatna. Płyty Arduino Uno będziemy używać jako standardowego urządzenia. Jest to najczęściej stosowana płyta Arduino. Wszystkie płyty Arduino są programowane w tym samym języku i mają podobną konfigurację wejść i wyjść, a więc ewentualne zastosowanie innej płyty nie stanowi większego problemu.

Uno, Duemilanove i Diecimila

Arduino Uno jest najnowszą wersją najbardziej popularnej serii płyt Arduino. Do serii tych płyt zaliczają się również włoskie konstrukcje Diecimila i Duemilanove. Na rysunku 1.4 znajduje się klon Arduino. Jak już pewnie zauważyłeś, Arduino jest włoskim wynalazkiem.



Rysunek 1.4. *Płytka Arduino Duemilanove*

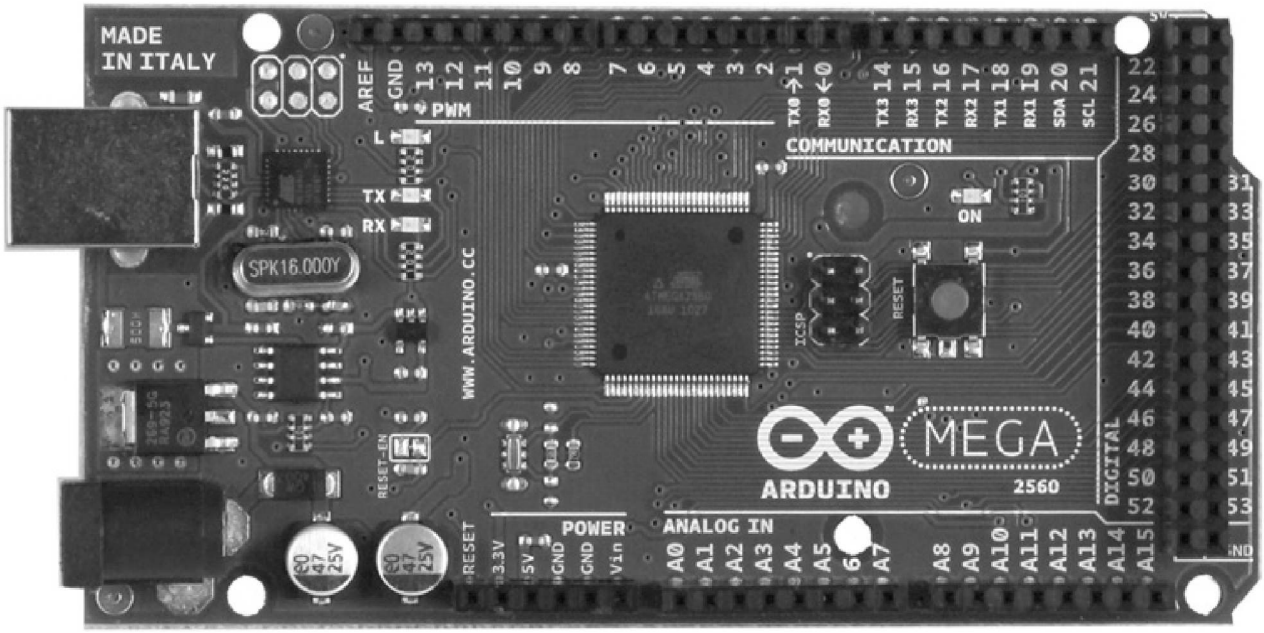
Te starsze płyty wyglądają bardzo podobnie do Arduino Uno. Posiadają one takie same złącza oraz port USB. Ponadto są ze sobą kompatybilne.

Główną różnicą pomiędzy płytami jest to, że Uno posiada inny chip USB niż jego poprzednicy. Chip ten nie wpływa na sposób użytkowania płyty, jednakże sprawia, że instalacja oprogramowania Arduino jest łatwiejsza, a komunikacja z komputerem przebiega szybciej.

Płyta Uno może dostarczyć większy prąd na złączu o napięciu 3,3 V. Ponadto płyta ta jest zawsze wyposażona w układ ATmega328. Wcześniejsze płyty były wyposażane w układy ATmega328 lub ATmega168. Mikrokontroler ATmega328 posiada więcej pamięci, ale o ile nie tworzysz dużego szkicu, nie jest to istotna różnica.

Mega

Arduino Mega (patrz rysunek 1.5) to prawdziwe Ferrari wśród płyt Arduino. Płyta ta została wyposażona w wiele dodatkowych złączy wejścia i wyjścia. Złącza te zostały jednakże dodane na skraju płytki, a więc Arduino Mega jest w pełni kompatybilne pod względem układu złączy z Arduino Uno, w związku z czym możliwe jest stosowanie płytek stykowych zgodnych z Arduino Uno.



Rysunek 1.5. *Płytka Arduino Mega*

Arduino Mega wyposażono w procesor obsługujący dodatkowe złącza — ATmega1280. Mikrokontroler ten jest montowany powierzchniowo, a więc jest przymocowany do płyty na stałe. W przypadku uszkodzenia procesora jego wymiana nie jest możliwa. Omówiona wcześniej płytka Arduino Uno umożliwia wymianę uszkodzonego mikroukładu.

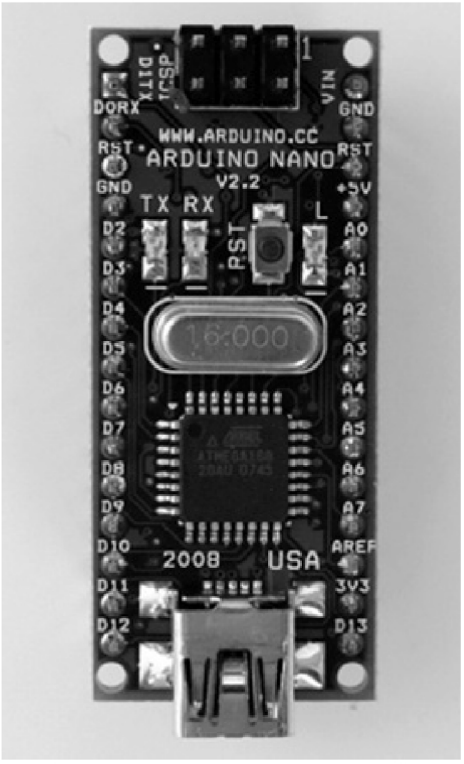
Dodatkowe złącza umieszczono na skraju płytki. Dodatkowe funkcje tej wersji Arduino to:

- 54 piny wejścia i wyjścia;
- 128 kB pamięci flash służącej do przechowywania szkiców i stałych danych (Arduino Uno posiadało tylko 32 kB tej pamięci);
- 8 kB pamięci RAM i 4 kB pamięci EEPROM.

Nano

Arduino Nano (rysunek 1.6) jest bardzo przydatnym urządzeniem w przypadku pracy z płytką prototypową. Jeżeli odpowiednio dopasujesz piny, możliwe jest wpięcie Arduino Nano w płytkę prototypową tak, jakby Arduino było chipem.

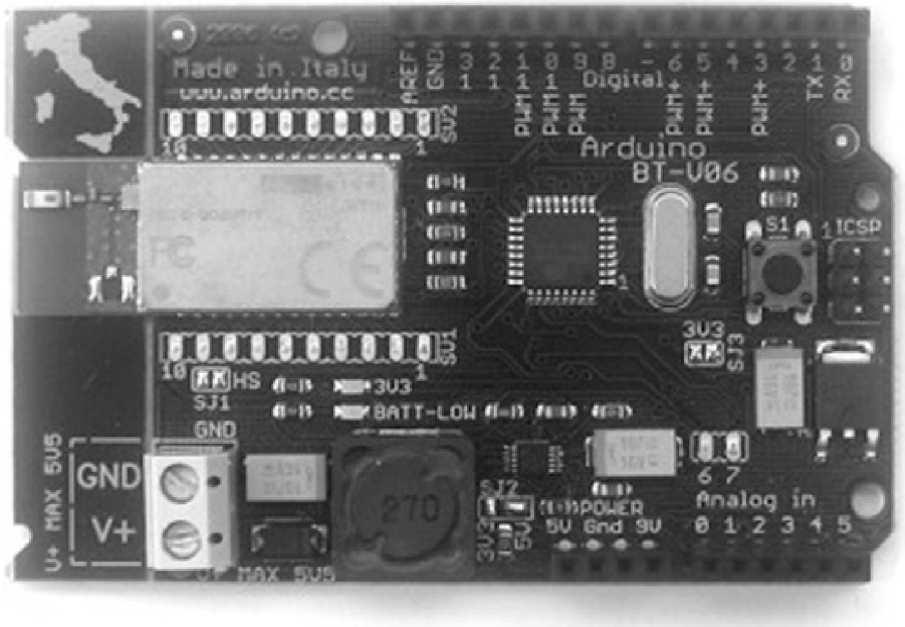
Wadą płytki Nano jest to, że z powodu jej rozmiarów nie można doczepiać do niej płytek stykowych przeznaczonych dla Arduino Uno.



Rysunek 1.6. *Płytka Arduino Nano*

Bluetooth

Arduino Bluetooth (patrz rysunek 1.7) jest ciekawym urządzeniem. W miejscu złącza USB umieszczono kontroler Bluetooth. Pozwala on na bezprzewodowe programowanie urządzenia.

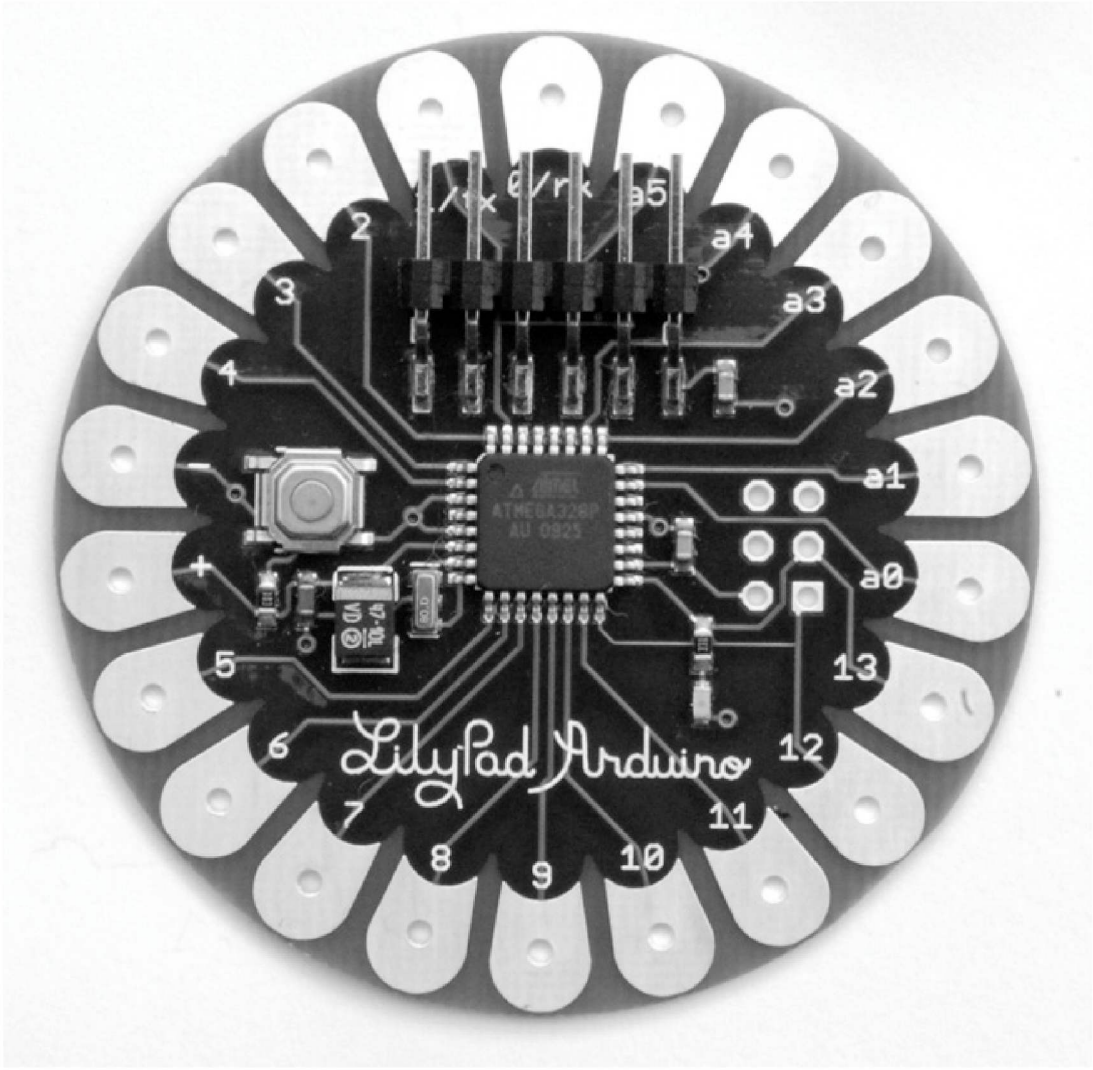


Rysunek 1.7. *Płytka Arduino Bluetooth*

Płytka Arduino Bluetooth jest droga. Bardziej opłacalny może okazać się zakup zwykłego Arduino Uno i połączenie go z modułem Bluetooth innego producenta.

Lilypad

Lilypad (patrz rysunek 1.8) jest niewielką, cienką płytką Arduino, którą można przyczepić do ubrania. Płytkę tę można stosować w przypadku tworzenia układów elektronicznych noszonych pod ubraniem.



Rysunek 1.8. *Płytka Arduino Lilypad*

Lilypad nie posiada złącza USB. Do zaprogramowania tej płytki niezbędny jest zewnętrzny programator. Projekt tej płytki jest wyjątkowo estetyczny.

Inne „oficjalne" płytki

Przedstawione wcześniej płytki Arduino są najbardziej przydatne i najczęściej stosowane. Rodzina płytek Arduino ciągle się jednakże powiększa. Aby uzyskać najświeższe informacje dotyczące płytek z rodziny Arduino, zajrzyj na oficjalną witrynę internetową Arduino pod adresem: *http://www.arduino.cc/en/Main/Hardware*.

Inne klony i odmiany Arduino

Nieoficjalne płytki można podzielić na dwie kategorie. Niektóre płytki są po prostu tańszymi, standardowymi kopiami Arduino. Do tej kategorii można zaliczyć następujące płytki:

- Roboduino,
- Freeduino,
- Seeeduino (tak, trzy litery „e" nie są błędem w druku).

Niektóre płytki kompatybilne z Arduino mają na celu usprawnienie działania Arduino lub rozszerzenie jego możliwości. Ciągle powstają nowe odmiany tego typu płytek. Niemożliwe jest wyliczenie ich wszystkich. Poniższa lista zawiera bardziej interesujące i częściej spotykane odmiany Arduino:

- Chipkit — szybka odmiana oparta o procesor PIC, wysoce kompatybilna z Arduino;
- Femtoduino — Arduino o bardzo małych rozmiarach;
- Ruggeduino — płytka Arduino z wbudowaną ochroną wejść i wyjść;
- Teensy — tanie urządzenie typu nano.

Podsumowanie

Gdy już poznałeś Arduino od strony sprzętowej, nastał czas, aby zacząć tworzyć oprogramowanie na to urządzenie.

Rozdział 2.

Rozpoczynamy przygodę z Arduino

Po zapoznaniu się z poprzednim rozdziałem już wiesz, jaki układ będziemy programować. Teraz czas na zainstalowanie na komputerze niezbędnego oprogramowania i rozpoczęcie pracy z kodem.

Zasilanie

Na nowo zakupionej płytce Arduino zwykle fabrycznie zainstalowany jest prosty program o nazwie Blink. Program ten sprawia, że wbudowana w układ dioda LED miga.

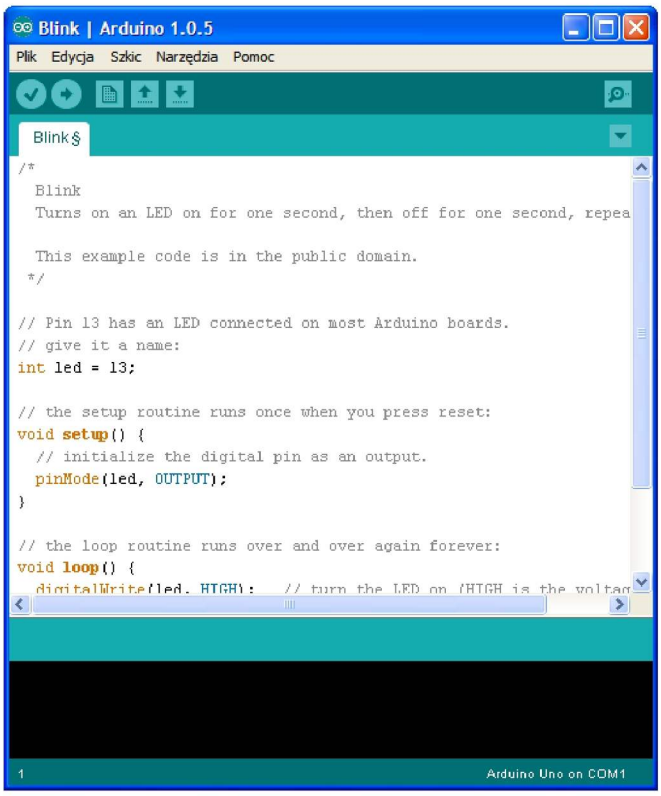
Dioda oznaczona symbolem *L* jest połączona z jednym z cyfrowych złączy wejścia i wyjścia znajdujących się na płytce. Dioda ta jest podłączona do złącza o numerze 13. Oznacza to, że złącze to może być używane tylko w charakterze wyjścia. Dioda ma mały pobór prądu, a wiec możesz również podłączać inne rzeczy do tego złącza.

Aby uruchomić Arduino, podłącz je do prądu. Najłatwiejszym sposobem na to jest podłączenie Arduino do portu USB komputera. Będzie do tego potrzebny przewód USB zakończony wtykiem typu A z jednej strony oraz wtykiem typu B z drugiej strony. Jest to taki sam przewód jak ten, który jest zwykle stosowany w celu połączenia drukarki i komputera.

Miganie diody LED oznacza, że wszystko działa poprawnie. Nowe płytki Arduino mają zainstalowany szkic programu Blink, aby można było łatwo sprawdzić, czy płyta działa poprawnie.

Instalacja oprogramowania

Do wgrywania szkiców na płytkę Arduino nie wystarczy zasilanie jej za pośrednictwem złącza USB. Aby ładować szkice, musisz zainstalować oprogramowanie Arduino na swoim komputerze (patrz rysunek 2.1).



Rysunek 2.1. *Aplikacja Arduino*

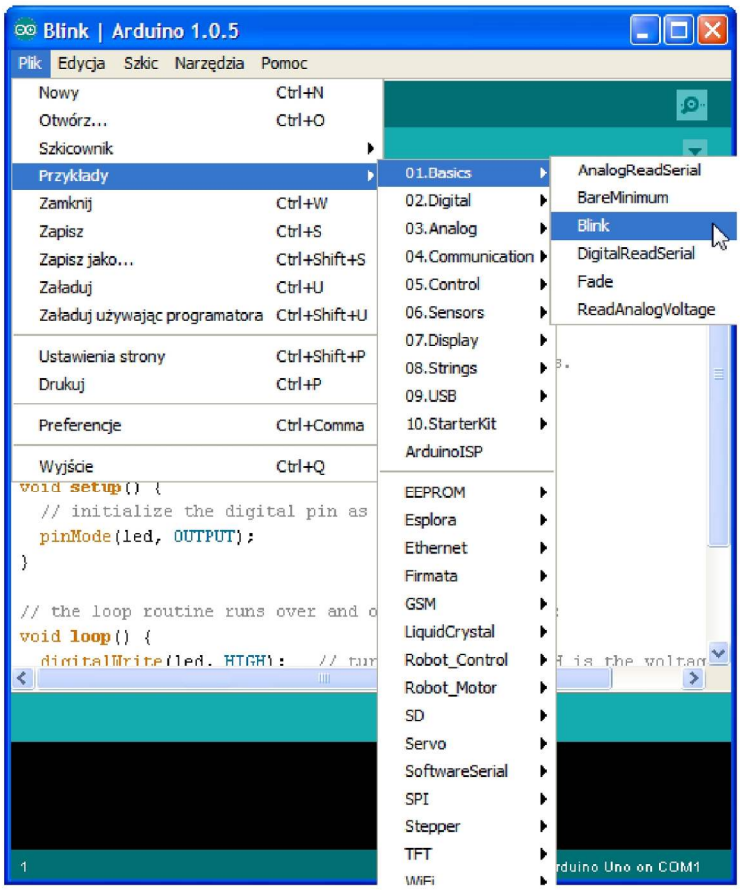
Pełna instrukcja instalacji tego oprogramowania na platformach Windows, Linux oraz Mac znajduje się w witrynie internetowej Arduino — *http://www.arduino.cc/*.

Po zainstalowaniu oprogramowania Arduino oraz sterowników USB (zależnie od używanego systemu operacyjnego) możesz ładować programy do swojego Arduino.

Ładowanie pierwszego szkicu

Program migający diodą jest odpowiednikiem programu wyświetlającego napis „Witaj" — programu uruchamianego tradycyjnie jako pierwszy podczas nauki nowego języka programowania. Sprawdźmy działanie środowiska Arduino, instalując ten program na płytce, a następnie modyfikując go.

Po uruchomieniu aplikacji Arduino na komputerze otwarty zostanie pusty szkic. Na szczęście program posiada wbudowanych wiele przydatnych przykładów. Otwórzmy przykład o nazwie Blink, co pokazano na rysunku 2.2.



Rysunek 2.2. *Szkic Blink*

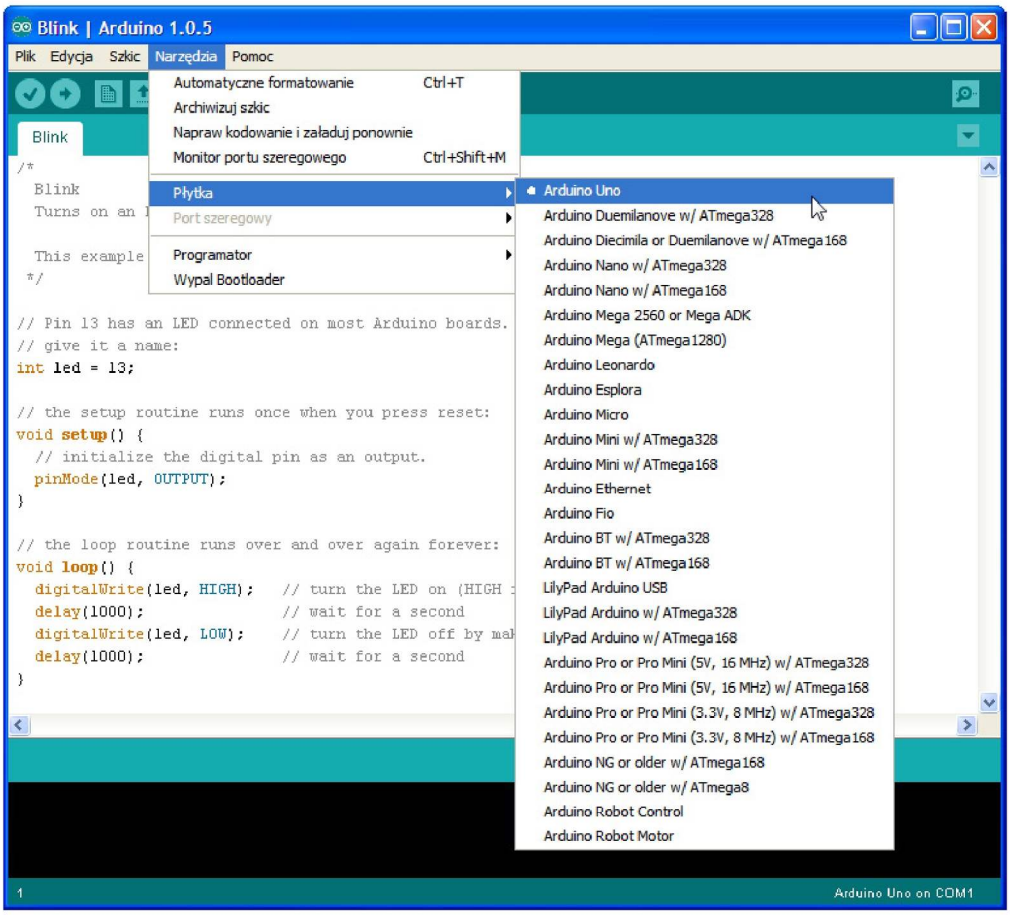
Teraz musisz zainstalować ten szkic na płytce Arduino. Podłącz płytkę Arduino do swojego komputera za pośrednictwem kabla USB. W wyniku tego na płytce Arduino powinna się zapalić zielona dioda oznaczona napisem „On". Dioda LED połączona z pinem o numerze 13 będzie już prawdopodobnie migać, ponieważ płytki Arduino są zwykle dostarczane z uprzednio zainstalowanym szkicem Blink. Niezależnie od tego spróbujmy go ponownie zainstalować, a następnie zmodyfikować.

Po podłączeniu płytki do komputera Macintosh na ekranie komputera zostanie wyświetlony komunikat informujący o wykryciu nowego interfejsu sieciowego. W takim przypadku należy kliknąć przycisk *Anuluj*. Komputer mylnie wykrywa Arduino Uno jako modem USB.

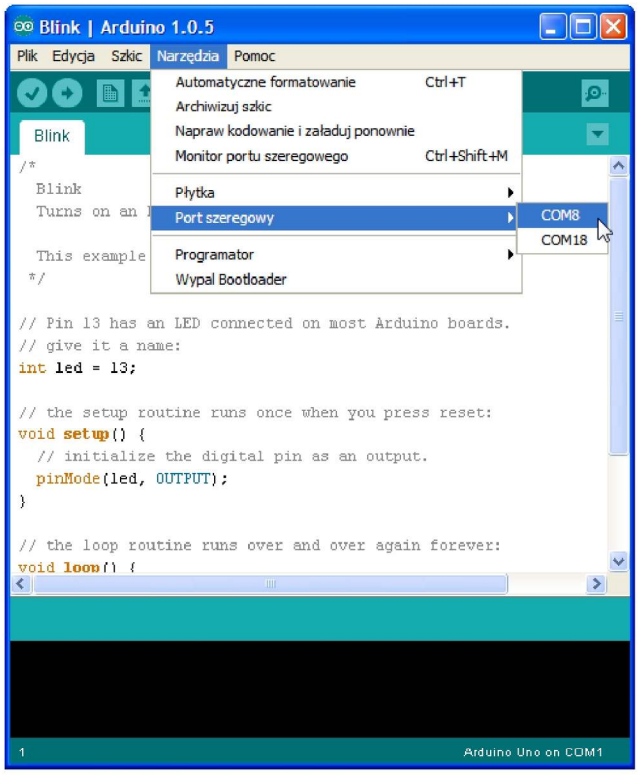
Przed załadowaniem szkicu należy w aplikacji Arduino określić typ płytki, z której korzystamy. Ponadto należy określić numer portu, do którego zostało podłączone Arduino. Na rysunkach 2.3 oraz 2.4 pokazano, jak dokonać tych operacji przy użyciu menu *Narzędzia*.

W przypadku komputera pracującego pod kontrolą systemu Windows zwykle będziesz korzystać z portu COM3. W przypadku komputerów pracujących pod kontrolą systemów Linux lub Mac OS X lista portów szeregowych będzie o wiele dłuższa (patrz rysunek 2.5). Nasze urządzenie będzie w większości przypadków znajdowało się na szczycie tej listy. Będzie ono miało nazwę podobną do `/dev/tty.usbmodem621`.

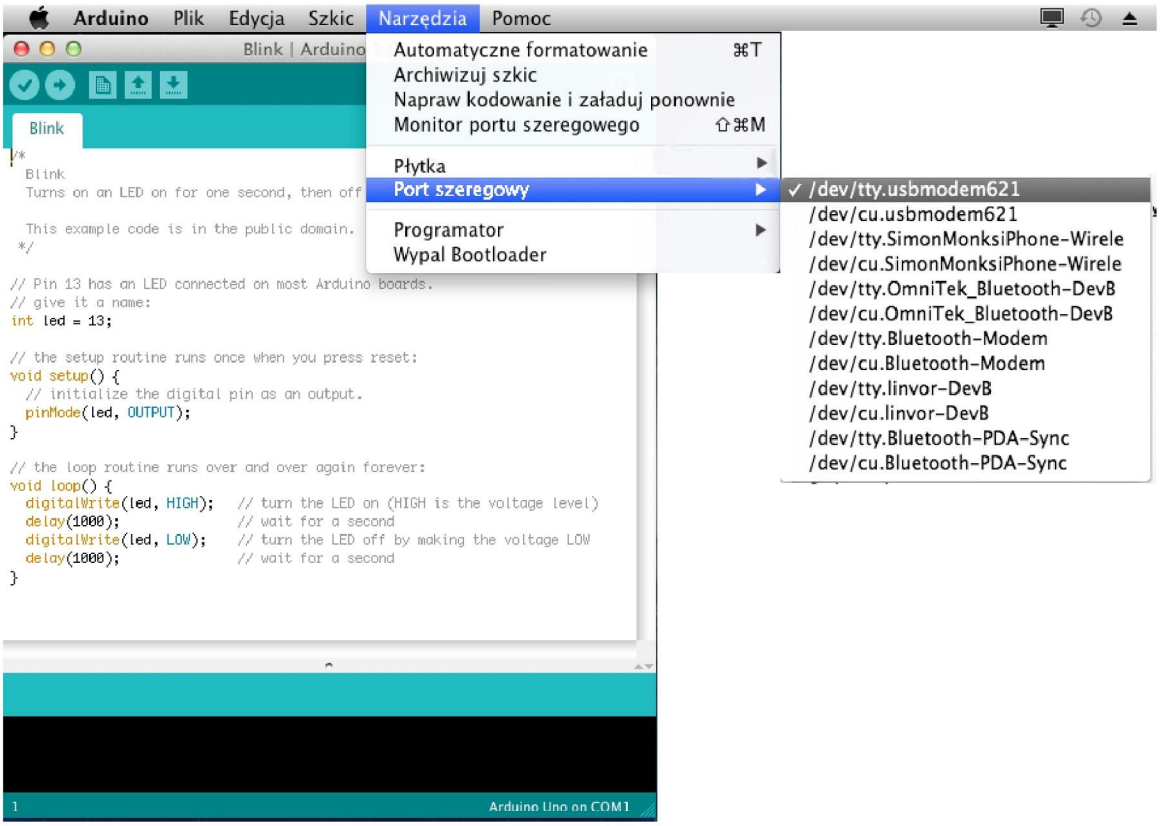
Teraz wystarczy tylko kliknąć ikonę *Załaduj*, która jest podświetlona na rysunku 2.6.



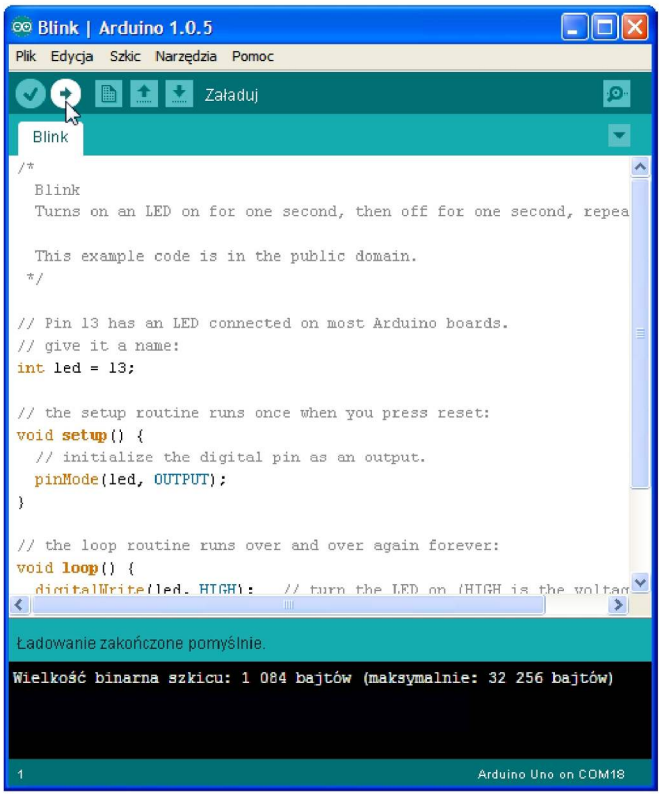
Rysunek 2.3. *Wybór typu płytki*



Rysunek 2.4. *Wybór portu szeregowego (w systemie Windows)*



Rysunek 2.5. *Wybór portu szeregowego (w systemie Mac OS X)*



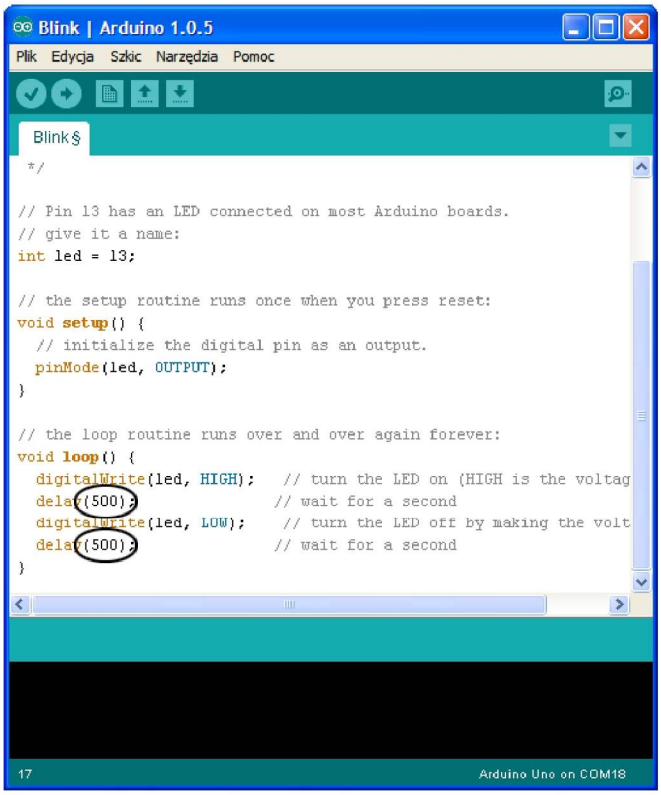
Rysunek 2.6. *Ładowanie szkicu*

Po kliknięciu ikony nastąpi kompilacja szkicu, a następnie rozpocznie się jego ładowanie. Jeżeli wszystko działa poprawnie, podczas przesyłu danych diody znajdujące się na Arduino będą migać. Po zakończeniu przesyłu danych u dołu aplikacji Arduino zostanie wyświetlony komunikat o treści „Ładowanie zakończono pomyślnie". Pod nim zostanie wyświetlony kolejny komunikat o treści podobnej do: „Wielkość binarna szkicu: 1084 bajtów (maksymalnie 32256 bajtów)".

Po załadowaniu programu płytka natychmiast zaczyna go wykonywać, a więc zobaczysz, że dioda zacznie błyskać.

Jeżeli program nie działa, sprawdź ustawienia typu płytki oraz portu szeregowego.

Teraz spróbujmy zmodyfikować szkic tak, aby zwiększyć częstotliwość błysków diody LED. W tym celu należy zmodyfikować dwa miejsca, w których zdefiniowano w szkicu opóźnienie (z ang. *delay*) na 1000 milisekund. Obie wartości 1000 należy zastąpić wartościami 500. Zmodyfikowany kod pokazano na rysunku 2.7.



Rysunek 2.7. *Modyfikacja szkicu Blink*

Ponownie kliknij przycisk *Załaduj*. Po załadowaniu szkicu zobaczysz diodę LED błyskającą dwa razy częściej, niż miało to miejsce w przypadku oryginalnego szkicu.

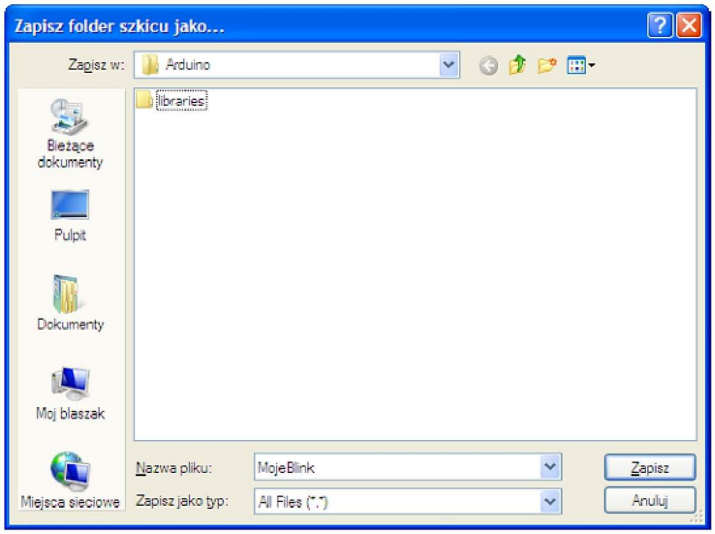
Gratuluję! Jesteś gotowy, by zacząć programować swoje Arduino. Ale wcześniej przyjrzyjmy się dokładnie aplikacji Arduino.

Aplikacja Arduino

Szkice są jak dokumenty w edytorze tekstowym. Można je otwierać i kopiować część kodu z jednego szkicu do drugiego. W menu *Plik* znajdziesz pozycje takie jak *Otwórz*, *Zapisz* i *Zapisz jako*. W większości przypadków nie będziemy korzystać z funkcji *Otwórz*, ponieważ w aplikacji Arduino funkcjonuje pojęcie szkicownika. Szkicownik to miejsce, gdzie przechowywane są foldery zawierające szkice. Dostęp do szkicownika uzyskujemy również z menu *Plik*. Szkicownik będzie w chwili obecnej pusty, ponieważ dopiero zainstalowałeś aplikację Arduino i nie napisałeś jeszcze żadnego szkicu.

Aplikacja Arduino posiada wbudowany spory zestaw przydatnych przykładowych szkiców. Podczas próby zapisu zmodyfikowanego szkicu *Blink* zostanie wyświetlony komunikat o następującej treści: „Niektóre pliki są oznaczone jako »Tylko do odczytu«. W związku z tym musisz zachować szkic w innej lokalizacji".

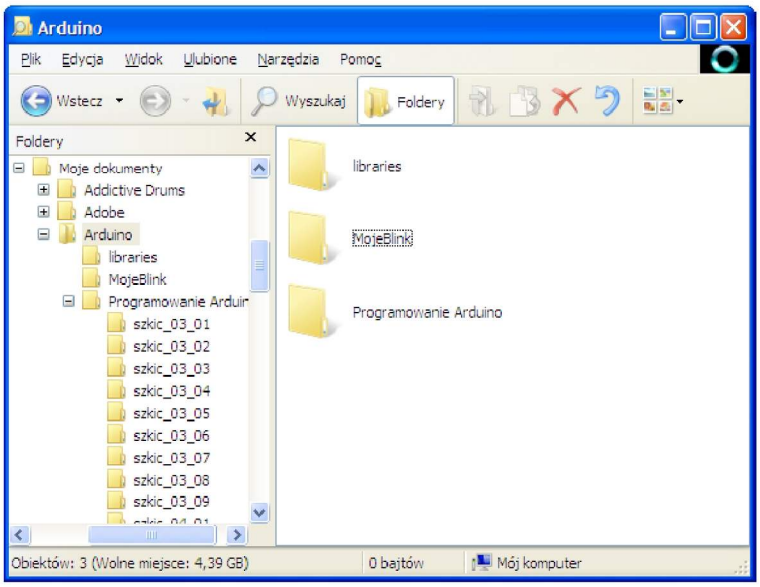
Spróbuj to zrobić. Zaakceptuj domyślną lokalizację, ale zmień nazwę pliku na *MojeBlink*, tak jak pokazano to na rysunku 2.8.



Rysunek 2.8. *Zapisywanie kopii szkicu Blink*

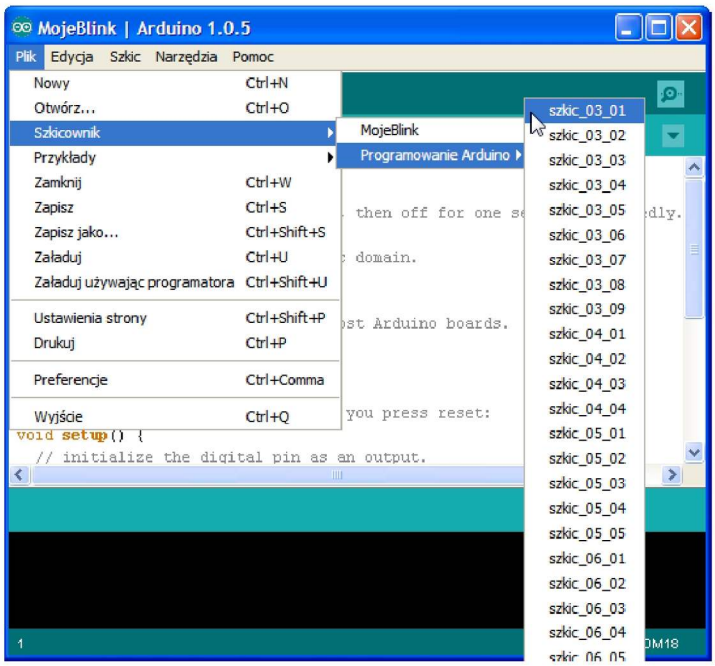
Teraz gdy najedziesz w menu *Plik* na pole *Szkicownik*, zobaczysz, że szkic *MojeBlink* został dodany do listy szkiców. W przypadku komputera klasy PC i systemu Windows szkice są zapisywane w folderze *Moje dokumenty\Arduino*. W systemie operacyjnym Linux szkice są przechowywane w folderze *Dokumenty/Arduino*.

Wszystkie szkice omówione w tej książce możesz ściągnąć w postaci archiwum z serwisu *http://www.helion.pl/ksiazki/ardupo.htm*. Dobrze by było, gdybyś ściągnął te szkice teraz i zapisał w folderze szkiców Arduino. Po rozpakowaniu archiwum do folderu ze szkicami poza katalogiem *MojeBlink* znajdzie się w nim katalog *Programowanie Arduino* (patrz rysunek 2.9). Folder *Programowanie Arduino* będzie zawierał wszystkie szkice. Ich numery określają to, w jakim rozdziale dany szkic jest stosowany. Np. szkic *03.01.* jest pierwszym szkicem omówionym w 3. rozdziale niniejszej książki.



Rysunek 2.9. *Instalacja szkiców omówionych w książce*

Szkice te będą widoczne w szkicowniku dopiero po ponownym uruchomieniu aplikacji. Uruchom ponownie aplikację Arduino. Twoje menu *Szkicownika* powinno wyglądać podobnie jak to przedstawione na rysunku 2.10.



Rysunek 2.10. *Szkicownik zawierający zainstalowane szkice omówione w książce*

Podsumowanie

Twoje środowisko jest gotowe do pracy.

W kolejnym rozdziale omówimy wybrane podstawy języka C, które są przydatne podczas pracy z Arduino.

Rozdział 3.

Podstawy języka C

Do programowania Arduino stosowany jest język C. W poniższym rozdziale przedstawimy podstawy tego języka. Wiadomości, które nabędziesz podczas lektury tego rozdziału, będą przydatne podczas pisania każdego szkicu Arduino. Zrozumienie tych podstawowych wiadomości pozwoli Ci na wykorzystywanie wszystkich możliwości Arduino.

Programowanie

Wiele osób mówi w więcej niż jednym języku. Im więcej języków się zna, tym łatwiej przychodzi nauka kolejnych języków. Dzieje się tak, ponieważ poliglota zauważa reguły występujące w gramatyce i leksyce poszczególnych języków. Taka sama prawidłowość występuje w przypadku nauki języków programowania. Nauka języka C będzie przebiegać o wiele szybciej, jeżeli znasz już jakiś inny język programowania.

Pierwsza dobra wiadomość to fakt, że zakres wyrażeń stosowanych w języku programowania jest o wiele mniejszy od zakresu słownictwa stosowanego w języku mówionym. Ponieważ język programowania jest językiem pisanym, a nie mówionym, możesz zawsze posłużyć się pomocą słownika lub podręcznika. Składnia języka programowania jest ściśle uregulowana. Gdy już opanujesz podstawowe pojęcia, nauka kolejnych zagadnień związanych z danym językiem programowania przebiega bardzo sprawnie.

Program (w przypadku Arduino zwany szkicem) można interpretować jako listę poleceń wykonywanych w kolejności, w jakiej zostały zapisane. Załóżmy, że napisałeś poniższy kod programu:

```
digitalWrite(13, HIGH);
delay(500);
digitalWrite(13, LOW);
```

Każda z tych trzech linii kodu służy do wykonania jakiejś czynności. Pierwsza linia kodu ustawia wysoki (z ang. *high*) poziom napięcia na złączu wyjściowym o numerze 13. Jest to złącze, z którym połączona jest dioda LED znajdująca się na płytce Arduino. A więc dioda

ta zostanie zapalona. Druga linia kodu informuje o tym, że Arduino zaczeka 500 milisekund (pół sekundy) i wykona polecenie zawarte w trzeciej linii kodu, czyli wyłączy diodę. Zaprezentowane trzy linijki kodu sprawią, że dioda zostanie jednokrotnie zapalona, a następnie zgaszona.

Na początku może Cię przerażać zarówno zastosowana interpunkcja, jak również i wyrazy, które nie są rozdzielane spacjami. Wielu początkujących programistów wie, co chce zrobić, ale nie wie, jak ma to zapisać. Nie ma tu niczego strasznego. Wszystko zostanie wyjaśnione.

Na początek zajmijmy się interpunkcją i sposobem tworzenia wyrażeń. Obie te rzeczy są elementami składni języka programowania. Większość języków wymaga dokładnego stosowania się do reguł składniowych. Jedna z zasad tworzenia nazw mówi, że nazwy muszą być jednowyrazowe. A więc nie mogą zawierać znaku rozdzielającego — spacji. Wyrażenie `digitalWrite` jest nazwą. Jest to nazwa wbudowanej funkcji, która dokonuje konfiguracji złączy wyjściowych na płytce Arduino. Więcej informacji na temat funkcji zostanie przedstawionych w kolejnych rozdziałach. Przy zapisie nazw należy uważać nie tylko na znaki rozdzielające. W zapisie nazw rozróżniane są wielkie i małe litery. A więc powinieneś stosować zapis `digitalWrite`, a nie `DigitalWrite` lub `Digitalwrite`.

Funkcja `digitalWrite` musi posiadać dane dotyczące tego, obsługą którego złącza ma się zająć. Funkcja powinna również posiadać informację dotyczącą napięcia, jakie ma się pojawić na złączu — wysokie (z ang. *high*) czy niskie (z ang. *low*). Te dwie informację noszą nazwę **argumentów**, które są **przekazywane** do funkcji podczas jej **wywołania**. Parametry funkcji muszą być umieszczone w nawiasach i oddzielone przecinkami.

Istnieje konwencja zapisu mówiąca o tym, że nawias należy otwierać bezpośrednio za ostatnią literą nazwy funkcji, a pomiędzy przecinkiem i kolejnym parametrem powinna być umieszczona spacja. Jeżeli jednakże czujesz taką potrzebę, możesz zastosować więcej spacji wewnątrz nawiasów.

Jeżeli funkcja posiada tylko jeden argument, nie ma potrzeby umieszczania przecinku w nawiasie.

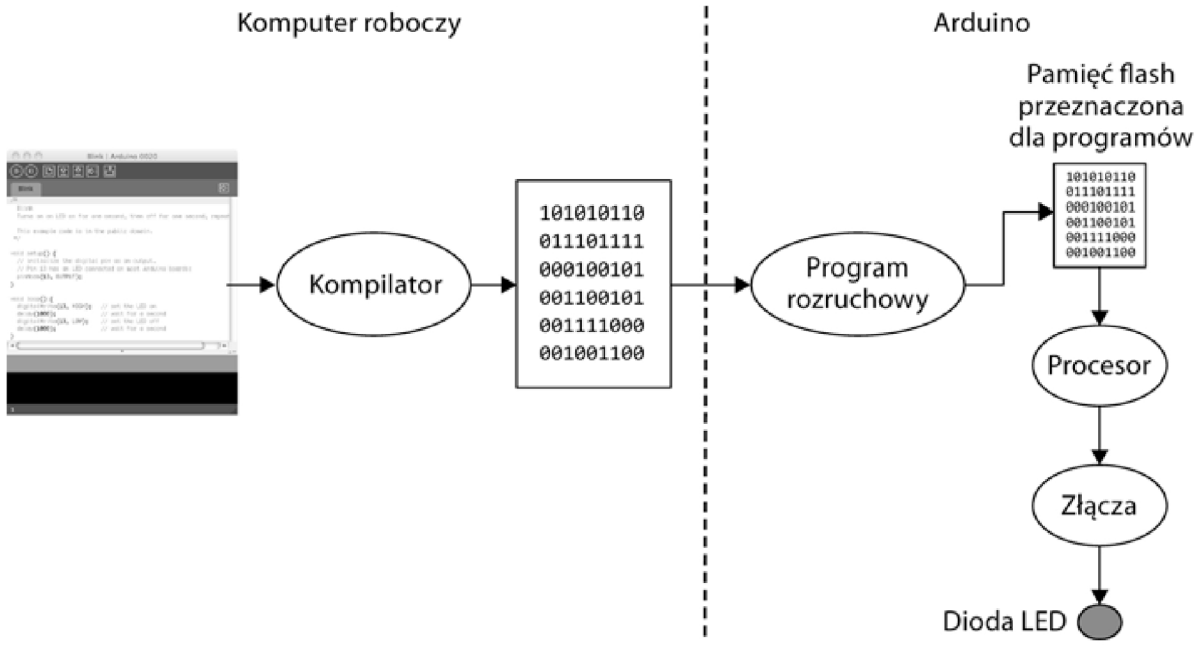
Zauważ, że każda linia kodu jest zakończona średnikiem. Bardziej logiczne byłoby kończenie każdej linii kodu kropką. Średnik sygnalizuje koniec danego polecenia, tak jak kropka sygnalizuje koniec zdania.

W kolejnym podrozdziale dowiesz się, co dzieje się, gdy w zintegrowanym środowisku programistycznym Arduino klikniesz ikonę *Załaduj* . Pozwoli Ci to na pracę z przykładowymi szkicami.

Czym jest język programowania?

To dziwne, że w książce dotyczącej programowania dopiero w trzecim rozdziale piszemy o tym, czym jest język programowania. Potrafisz rozpoznać szkic Arduino, a także masz już niewielkie pojęcie o tym, jakie operacje są w nim zdefiniowane. Musisz jednak mieć większą świadomość tego, jak kod zapisany w języku programowania jest tłumaczony z zapisanych wyrażeń na realnie wykonywaną czynność, taką jak włączenie i wyłączenie diody LED.

Na rysunku 3.1 przedstawiono proces zaczynający się od wpisania kodu programu w zintegrowanym środowisku Arduino, a kończący się na wykonaniu szkicu przez mikrokontroler.



Rysunek 3.1. *Od kodu do jego wykonania przez płytkę*

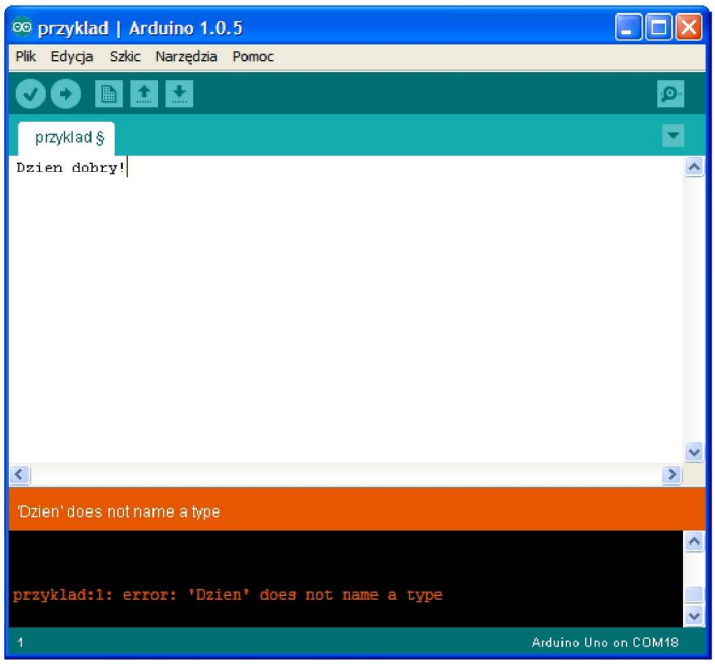
Gdy klikniesz ikonę *Załaduj* w środowisku Arduino, środowisko to uruchamia cały szereg operacji, w wyniku których szkic zostaje zainstalowany i uruchomiony na płytce Arduino. Jest to zadanie o wiele bardziej skomplikowane od prostego skopiowania wpisanego przez Ciebie tekstu z komputera do Arduino.

Pierwszą operacją wykonywaną przez środowisko jest **kompilacja**. Operacja ta polega na przetłumaczeniu kodu wpisanego przez użytkownika na kod maszynowy — język binarny rozumiany przez Arduino. Jeżeli klikniesz ikonę *Weryfikuj*, środowisko Arduino przystąpi do próby kompilacji programu napisanego w języku C. Program nie będzie przesyłany do płytki Arduino. Podczas próby kompilacji kod szkicu jest sprawdzany pod kątem zgodności z zasadami języka C.

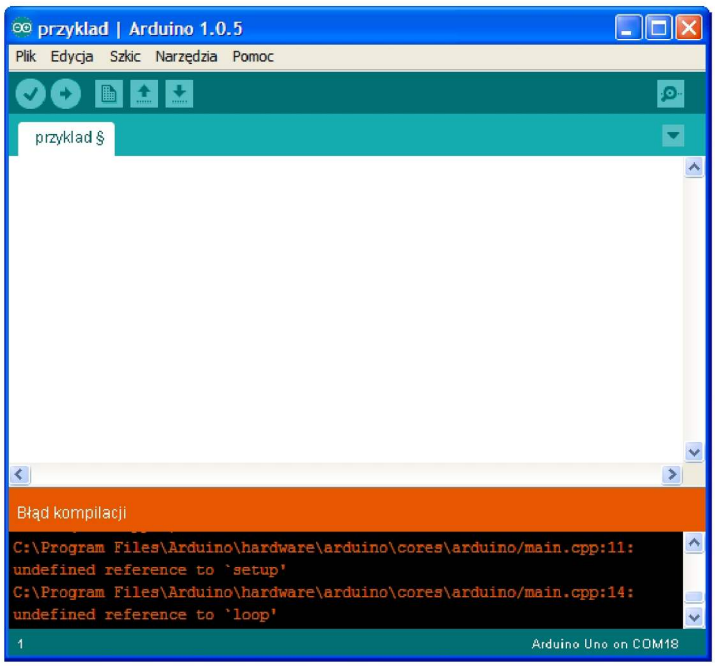
Jeżeli wpiszesz `Dzien dobry!` do środowiska Arduino i klikniesz ikonę *Weryfikuj*, zostanie wyświetlony komunikat taki, jak pokazano na rysunku 3.2.

Środowisko Arduino próbowało dokonać kompilacji słów „Dzien dobry", jednakże pomimo polskiej wersji interfejsu użytkownika oprogramowanie nie ma pojęcia, o czym mówisz. Wpisany tekst nie jest kodem języka C. A więc u dołu ekranu został wyświetlony komunikat błędu o treści: „error: 'Dzien' does not name a type". Informuje on nas o tym, że wpisany przez nas kod jest niepoprawny.

Przeprowadźmy kolejne doświadczenie. Tym razem spróbujmy skompilować szkic niezawierający żadnego kodu (patrz rysunek 3.3).



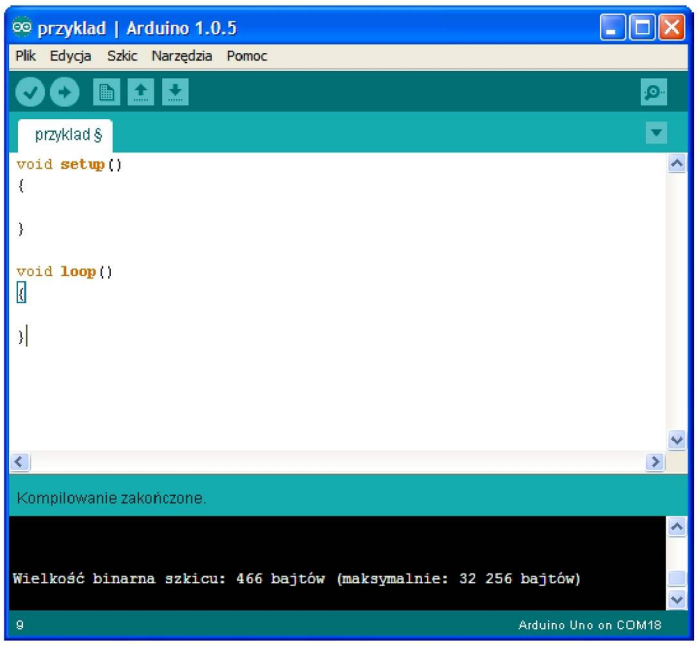
Rysunek 3.2. *Arduino nie posługuje się językiem polskim*



Rysunek 3.3. *Brak funkcji setup i loop*

Tym razem kompilator informuje nas o tym, że szkic nie zawiera funkcji `setup` i `loop`. Jak już zauważyłeś, analizując w rozdziale 2. przykład *Blink*, szkic musi zawierać pewien „szablonowy" kod, w którym dopiero umieszczamy nasze własne polecenia. W przypadku programowania Arduino taki „szablon" ma postać dwóch funkcji o nazwach „setup" i „loop". Te funkcje muszą występować w każdym szkicu.

Czytając kolejne rozdziały, poszerzysz swoją wiedzę dotyczącą funkcji. Teraz po prostu przyjmij za regułę to, że program powinien zawierać pewien formalny kod niezbędny do kompilacji szkicu (patrz rysunek 3.4).



Rysunek 3.4. *Szkic, który zostanie prawidłowo skompilowany*

Środowisko programistyczne Arduino przeanalizowało wprowadzony przez Ciebie kod i stwierdziło, że jest on prawidłowy. Stwierdzenie to jest sygnalizowane komunikatem „Kompilowanie zakończone". Poza komunikatem wyświetlona zostaje informacja o tym, że szkic ma rozmiar 466 bajtów, a maksymalny rozmiar szkicu wynosi 32 256 bajtów. Masz więc miejsce, aby tworzyć o wiele bardziej rozbudowane szkice.

Przeanalizujmy wpisany przez nas kod, który będzie punktem wyjścia podczas tworzenia każdego szkicu. Występują tu nowe elementy, takie jak np. słowo `void` i nawiasy klamrowe. Naszą analizę zacznijmy od tego słowa.

Linia kodu `void setup()` definiuje funkcję o nazwie `setup`. Arduino posiada zasób gotowych, zdefiniowanych funkcji (np. `digitalWrite` i `delay`). Niektóre funkcje musimy jednakże samodzielnie definiować. Funkcje `setup` i `loop` są funkcjami, które będziesz musiał definiować w każdym pisanym szkicu.

Musisz zrozumieć, że w przytoczonym przykładzie nie wywołujesz funkcji `setup` oraz `loop` tak, jak wcześniej wywoływaliśmy funkcję `digitalWrite`. Tworzysz te funkcje po to, aby Arduino mogło je samodzielnie wywoływać. Może być to trudne do zrozumienia, ale można o tym myśleć jak o definicji znajdującej się w akcie prawnym.

Większość aktów prawnych posiada sekcję zawierającą definicje. Można znaleźć w niej na przykład coś takiego:

```
Definicje.
Autor: Osoba odpowiedzialna za treści umieszczone w książce.
```

Po podaniu takiej definicji w dalszej części dokumentu możliwe jest stosowanie słowa „autor" jako skrótu oznaczającego „osobę odpowiedzialną za treści umieszczone w książce". Dzięki takiemu zabiegowi dokumenty tworzone przez prawników mogą być krótsze. Funkcje w języku programowania odgrywają taką samą rolę jak definicje w specjalistycznym języku stosowanym przez prawników. Możesz zdefiniować funkcję, która może być później zastosowana w szkicu przez Ciebie lub system.

Słowo kluczowe `void` zastosowane w kontekście funkcji `setup` i `loop` oznacza, że funkcje te nie zwracają żadnych wartości. Jeżeli wyobrazisz sobie funkcję o nazwie `sin`, która będzie obliczać wartość funkcji trygonometrycznej sinus, wtedy taka funkcja będzie zwracała pewną liczbę. Zwracaną wartością będzie sinus kąta, który został uprzednio przekazany do tej funkcji.

Funkcje w języku C stosuje się w tym samym celu co zdefiniowane terminy w dokumentach posiadających moc prawną.

Po słowie kluczowym `void` umieszczona jest nazwa funkcji, a za nią nawiasy zawierające argumenty. W zaprezentowanym przykładzie nie ma żadnych argumentów, jednakże umieszczenie nawiasów było konieczne. Po nawiasie zamykającym nie umieszczono średnika, ponieważ definiujemy funkcję, a nie wywołujemy ją. Musimy więc określić to, co powinno nastąpić, gdy opisywana przez nas funkcja zostanie wywołana.

Operacje, które maja zostać wykonane po wywołaniu funkcji, są zapisane pomiędzy nawiasami klamrowymi. Nawiasy klamrowe oraz kod umieszczony pomiędzy nimi nazywamy **blokiem** kodu. Pojęcie to napotkasz jeszcze wielokrotnie w dalszej części książki.

Zauważ, że musisz zdefiniować funkcje `setup` oraz `loop`, ale nie musisz umieszczać w nich żadnego kodu. Nieumieszczenie kodu w tych funkcjach sprawi jednakże, że Twój szkic będzie dosyć „niewyraźny".

Blink po raz kolejny

W Arduino stosujemy funkcje `setup` oraz `loop` w celu oddzielenia od siebie rzeczy, które muszą być wykonane raz w wyniku uruchomienia programu, od rzeczy, które będą wykonywane w sposób ciągły.

Funkcja `setup` jest uruchamiana jednokrotnie po uruchomieniu szkicu. Dodajmy do naszej funkcji kod, który sprawi, że dioda wbudowana w Arduino zostanie zapalona. Zmodyfikuj napisany wcześniej kod w sposób pokazany poniżej, a następnie załaduj gotowy szkic na płytkę.

```
void setup()
{
   pinMode(13, OUTPUT);
   digitalWrite(13, HIGH);
}
void loop()
{
}
```

Funkcja `setup` wywoła dwie wbudowane funkcje — `pinMode` i `digitalWrite`. Funkcja `digitalWrite` została omówiona wcześniej. Funkcja `pinMode` określa to, czy dane złącze (pin) ma działać jako wejście, czy ma odgrywać rolę wyjścia. A więc włączanie diody LED jest tak naprawdę procesem składającym się z dwóch etapów. Najpierw musimy określić to, że złącze o numerze 13 ma działać jako wyjście, a następnie musimy ustawić sygnał wyjściowy na wysoki (5 V).

Po uruchomieniu tego szkicu zobaczysz, że dioda zostaje zapalona i nie gaśnie. Efekt ten nie jest zachwycający, a więc spróbujmy sprawić, naprzemiennie włączając i wyłączając diodę, że będzie ona migać. Instrukcje włączającą i wyłączającą diodę umieścimy raczej w funkcji `loop` niż w funkcji `setup`.

Pozostawmy wywołanie funkcji `pinMode` wewnątrz funkcji `setup`, ponieważ funkcja `pinMode` musi zostać wywołana tylko jednokrotnie. Szkic nadal działałby, gdybyś przeniósł wywołanie funkcji `pinMode` do funkcji `loop`, ale nie ma takiej potrzeby. Dobrą praktyką programistyczną jest robienie tylko raz rzeczy, które muszą być wykonane jednokrotnie. Zmodyfikuj swój szkic, aby wyglądał w następujący sposób:

```
void setup()
{
   pinMode(13, OUTPUT);
}
void loop()
{
   digitalWrite(13, HIGH);
   delay(500);
   digitalWrite(13, LOW);
}
```

Uruchom ten szkic i zobacz, co się stanie. Efekt działania szkicu może odbiegać od oczekiwań. Dioda LED świeci światłem ciągłym. Dlaczego tak się dzieje?

Przeanalizujmy kod krok po kroku:

1. Uruchom funkcję `setup` i zainicjuj złącze o numerze 13 tak, aby działało jako wyjście.
2. Uruchom funkcję `loop` i włącz wysoki sygnał na złączu o numerze 13 (włącz diodę LED).
3. Odczekaj pół sekundy.
4. Podaj niski sygnał na złącze o numerze 13 (wyłącz diodę LED).
5. Uruchom ponownie funkcję `loop` — przejdź z powrotem do punktu drugiego i ponownie włącz wysoki sygnał na złączu o numerze 13 (włącz diodę LED).

Problem znajduje się pomiędzy punktem czwartym i piątym. Dioda jest wyłączana, ale jest natychmiast włączana ponownie. Dzieje się to tak szybko, że wydaje się, że dioda LED jest cały czas włączona.

Mikrokontroler znajdujący się na płytce Arduino jest w stanie przetwarzać 16 milionów instrukcji na sekundę. Nie jest to wykonywanie 16 milionów poleceń napisanych w języku C, lecz pomimo tego jest to bardzo duża wartość. Dioda będzie wyłączona tylko przez niewielki ułamek sekundy.

Aby rozwiązać ten problem, należy dodać kolejną funkcję opóźniającą `delay` po wyłączeniu diody LED. Twój kod powinien wyglądać w następujący sposób:

```
// szkic 03.01.

void setup()
{
  pinMode(13, OUTPUT);
}
```

```
void loop()
{
 digitalWrite(13, HIGH);
 delay(500);
 digitalWrite(13, LOW);
 delay(500);
}
```

Teraz dioda powinna migać około raz na sekundę.

Zauważyłeś zapewne komentarz o treści „szkic 03.01." znajdujący się na początku kodu. Umieściliśmy wszystkie szkice w witrynie internetowej, żebyś nie musiał tracić czasu na przepisywanie kodów z książki. Wszystkie szkice umieszczone na naszym serwerze są opatrzone komentarzem tego typu. Szkice możesz pobrać, korzystając z adresu: *http://www.helion.pl/ksiazki/ardupo.htm*

Zmienne

W przytoczonym wcześniej szkicu korzystaliśmy ze złącza o numerze 13. Odwoływaliśmy się do niego trzykrotnie. Jeżeli zdecydowałbyś się na korzystanie z innego złącza, musiałbyś zmodyfikować kod programu w trzech miejscach. Podobnie jeżeli chciałbyś zmienić częstotliwość błysków, musiałbyś zmienić wartość argumentu funkcji `delay` w więcej niż jednym miejscu.

Zmienne mogą być traktowane jako wartości, którym nadano nazwy. Mają one o wiele szersze zastosowanie, jednakże na razie skupimy się jedynie na tym.

Definiując zmienną w języku C, musisz określić jej typ. Chcemy, aby nasze zmienne były liczbami całkowitymi — w języku C zmienne takie noszą nazwę `int` (od angielskiego słowa *integer*). Poniżej przedstawiono deklarację zmiennej o nazwie `ledPin`. Przypisano jej wartość 13:

```
int ledPin 13;
```

Wyrażenie `ledPin` jest nazwą zmiennej. Do nazw zmiennych stosuje się te same reguły co do nazw funkcji. A więc nazwy nie mogą zawierać żadnych spacji. Istnieje konwencja dotycząca nazewnictwa zmiennych mówiąca o tym, że nazwy zmiennych zaczyna się z małej litery, a każde kolejne słowo rozpoczyna się wielką literą. Programiści określają ten sposób notacji terminem „camelCase".

Zmodyfikujmy nasz szkic w następujący sposób:

```
// szkic 03.02.
int ledPin = 13;
int delayPeriod = 500;

void setup()
{
  pinMode(ledPin, OUTPUT);
}
```

```
void loop()
{
 digitalWrite(ledPin, HIGH);
 delay(delayPeriod);
 digitalWrite(ledPin, LOW);
 delay(delayPeriod);
}
```

W zaprezentowanym szkicu zastosowano również kolejną zmienną — `delayPeriod`.

Wszędzie tam, gdzie w poprzednim szkicu stosowaliśmy wartość 13, teraz stosujemy zmienną `ledPin`, a wszędzie tam, gdzie stosowaliśmy wartość 500, teraz stosujemy zmienną `delayPeriod`.

Jeżeli chcesz, aby dioda błyskała z większą częstotliwością, wystarczy zmienić w jednym miejscu wartość zmiennej `delayPeriod`. Spróbuj zmienić wartość tej zmiennej na 100 i uruchomić szkic na płytce Arduino.

Są inne rzeczy, do których możemy sprytnie zastosować funkcje. Zmodyfikujmy szkic tak, aby na początku miganie przebiegało z bardzo dużą częstotliwością, a następnie częstotliwość błysków malała — tak jakby Arduino „męczyło się". Aby wykonać szkic dający taki efekt, musimy po każdym błysku dodawać jakąś wartość do zmiennej `delayPeriod`.

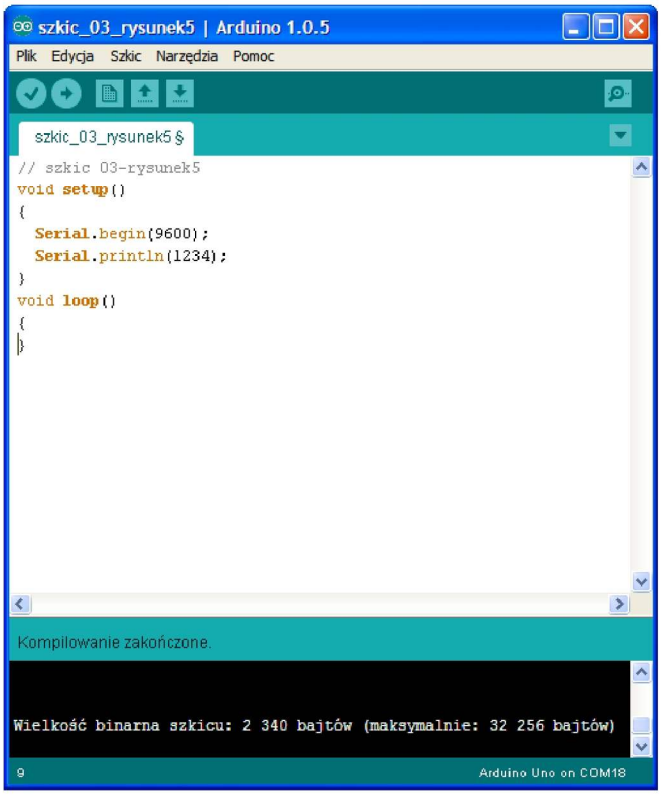
Zmodyfikuj szkic, dodając linię kodu na końcu funkcji `loop` tak, jak to zrobiono w poniższym kodzie. Uruchom szkic na płytce Arduino. Wciśnij przycisk *Reset* i obserwuj efekt działania szkicu.

```
// szkic 03.03.
int ledPin = 13;
int delayPeriod = 100;

void setup()
{
   pinMode(ledPin, OUTPUT);
}

void loop()
{
   digitalWrite(ledPin, HIGH);
   delay(delayPeriod);
   digitalWrite(ledPin, LOW);
   delay(delayPeriod);
   delayPeriod = delayPeriod + 100;
}
```

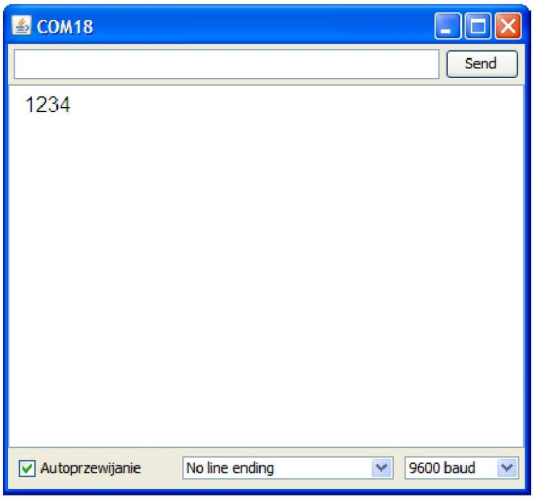
W zaprezentowanym przykładzie Arduino wykonuje operacje arytmetyczne. Przy każdym wywołaniu funkcji `loop` Arduino mignie diodą LED oraz równocześnie doda 100 do zmiennej `delayPeriod`. Niedługo wrócimy do zagadnień związanych z operacjami arytmetycznymi, ale najpierw poznajmy inne funkcje Arduino.

Eksperymentowanie w języku C

Musisz w jakiś sposób testować pisany przez siebie kod. Jednym ze sposobów na to jest umieszczenie kodu, który chcesz sprawdzić, w funkcji `setup`, sprawdzenie kodu przy użyciu Arduino, a następnie wyświetlenie wyniku przez Arduino za pomocą tak zwanego monitora portu szeregowego, co pokazano na rysunkach 3.5 i 3.6.



Rysunek 3.5. *Pisanie kodu języka C w funkcji setup*



Rysunek 3.6. *Monitor portu szeregowego*

Monitor portu szeregowego jest częścią środowiska programistycznego Arduino. Dostęp do niego możesz uzyskać, klikając ikonę znajdującą się po prawej stronie paska narzędzi. Monitor działa jako kanał komunikacyjny pomiędzy Twoim komputerem i Arduino.

Możesz wpisać komunikat w górnej części okna monitora portu szeregowego, a kiedy klikniesz ikonę *Send* lub wciśniesz klawisz *Enter*, wpisany kod zostanie wysłany do Arduino. Jeżeli Arduino ma coś do zakomunikowania, to zostanie to wyświetlone w oknie monitora portu szeregowego. Dane są przesyłane w obu kierunkach za pośrednictwem interfejsu USB.

Jak zapewne się spodziewałeś, istnieje wbudowana funkcja pozwalająca na wysyłanie wiadomości do monitora portu szeregowego. Funkcja ta nosi nazwę `Serial.println`. Jest to funkcja jednoargumentowa. Argumentem tym jest treść, którą chcemy przesłać. Treść ta jest zwykle pewną zmienną.

Będziesz korzystać z tego mechanizmu do sprawdzania rzeczy związanych ze zmiennymi oraz arytmetyką w języku C. Tak naprawdę jest to jedyny sposób na wizualizowanie rezultatów Twoich eksperymentów w języku C.

Zmienne numeryczne i arytmetyka

Ostatnią rzeczą, jaką zrobiłeś, modyfikując szkic sterujący migającą diodą, było stopniowe zwiększanie czasu trwania błysku przy użyciu następującego kodu:

```
delayPeriod = delayPeriod + 100;
```

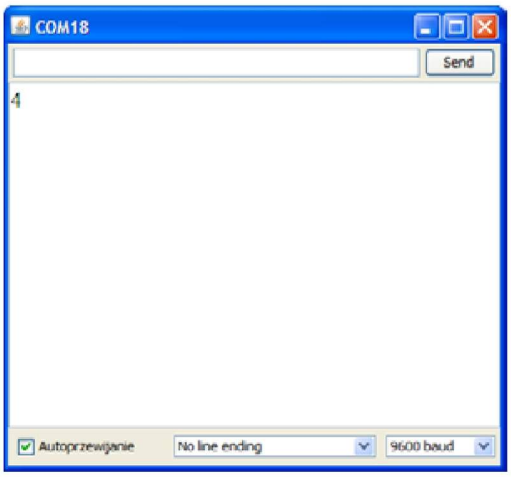
Przyjrzyjmy się temu kodowi. Składa się on z nazwy zmiennej, znaku równości, a następnie wyrażenia `delayPeriod + 100`. Znak równości wykonuje działanie zwane przypisaniem, czyli przypisuje nową wartość do zmiennej. Nowa wartość zmiennej jest określana przez to, co znajduje się pomiędzy znakiem równości a średnikiem. W omawianym przypadku nowa wartość zmiennej `delayPeriod` jest poprzednią wartością zmiennej `delayPeriod` zwiększoną o 100.

Sprawdźmy działanie omówionego wcześniej mechanizmu poprzez uruchomienie poniższego szkicu i otwarcie monitora portu szeregowego.

```
// szkic 03.04.
void setup()
{
  Serial.begin(9600);
   int a = 2;
   int b = 2;
   int c = a + b;
   Serial.println(c);
}
void loop()
{}
```

Na rysunku 3.7 pokazano to, co powinien wyświetlić monitor portu szeregowego po wykonaniu powyższego programu.

A teraz przyjrzyjmy się bardziej złożonemu przykładowi. Aby przekształcić wartość temperatury wyrażoną w skali Celsjusza na temperaturę wyrażoną w skali Fahrenheita, należy pomnożyć podaną wartość przez 5, podzielić ją przez 9, a następnie dodać do wyniku 32. Działanie to przedstawia poniższy szkic:



Rysunek 3.7. *Proste działania arytmetyczne*

```
// szkic 03.05.
void setup()
{
   Serial.begin(9600);
   int degC = 20;
   int degF;
   degF = degC * 9 / 5 + 32;
   Serial.println(degF);
}
void loop()
{}
```

Podany szkic zawiera kilka elementów, na które powinieneś zwrócić uwagę. Zacznijmy od poniższej linii kodu:

```
int degC = 20;
```

Pisząc tę linię kodu, przeprowadziliśmy tak naprawdę dwie operacje. Deklarujemy zmienną typu `int` o nazwie `degC`, a jednocześnie określamy wartość początkową tej zmiennej — 20. Możesz również wykonać te dwie operacje, korzystając z dwóch oddzielnych linii kodu:

```
int degC;
degC = 20;
```

Zmienną musisz deklarować tylko raz. Deklarując zmienną, informujesz kompilator o jej typie — w tym przypadku zmienna była typu `int`. Zadeklarowanej zmiennej możesz wielokrotnie przypisywać nowe wartości:

```
int degC;
degC = 20;
degC = 30;
```

W przykładzie dotyczącym zamiany jednostki temperatur definiujesz zmienną `degC` i nadajesz jej początkową wartość — 20. Natomiast definiując zmienną `degF`, nie nadajesz jej żadnej wartości początkowej. W kolejnej linii kodu zmiennej tej przypisywana jest wartość zgodnie z przeprowadzonymi obliczeniami. Dopiero po ich wykonaniu wynik jest przesyłany do monitora portu szeregowego.

Gdy przyjrzysz się obliczeniom, zauważysz, że znak gwiazdki (*) jest stosowany do operacji mnożenia, a znak kreski ukośnej (/) jest stosowany do operacji dzielenia. Operacje matematyczne określane przez operatory (+, –, * i /) są wykonywane w ustalonej kolejności. Najpierw wykonywane są operacje mnożenia, następnie dzielenia, a dopiero później dodawania i odejmowania. Jest to kolejność, w jakiej z reguły przeprowadza się operacje arytmetyczne. Czasem, w przypadku wyrażeń arytmetycznych, warto zastosować nawiasy. W tym celu możesz zastosować następujący zapis:

```
degF = ((degC * 9) / 5) + 32;
```

Wyrażenia tego typu mogą być dowolnie długie i dowolnie złożone. Ponadto poza podstawowymi operacjami matematycznymi możliwe jest stosowanie olbrzymiej liczby dostępnych funkcji matematycznych. Funkcje te omówimy jednak później.

Polecenia

W języku C zaimplementowano wiele poleceń. W tym podrozdziale omówimy niektóre z nich i zobaczymy, jak można je zastosować w szkicach.

if

Do tej pory zakładaliśmy, że linie kodu umieszczone w szkicu będą wykonywane jedna po drugiej, bez żadnych wyjątków. Ale co, jeżeli tego nie chcesz? Co, jeżeli chcesz wykonać tylko część kodu, zależnie od jakichś uwarunkowań?

Wróćmy do przykładu, w którym dioda LED błyskała z coraz niższą częstotliwością. W tej chwili dioda będzie migać coraz to wolniej i wolniej, aż błysk będzie trwał godzinami. Przyjrzyjmy się temu, jak można sprawić, by częstotliwość błyskania diody po osiągnięciu pewnej dolnej wartości znów wracała do wartości początkowej.

Aby tego dokonać, musisz zastosować polecenie `if`. Poniżej zamieszczono zmodyfikowany szkic. Wypróbuj go.

```
// szkic 03.06.
int ledPin = 13;
int delayPeriod = 100;

void setup()
{
   pinMode(ledPin, OUTPUT);
}

void loop()
{
   digitalWrite(ledPin, HIGH);
   delay(delayPeriod);
   digitalWrite(ledPin, LOW);
   delay(delayPeriod);
   delayPeriod = delayPeriod + 100;
   if (delayPeriod > 1000)
```

```
    {
      delayPeriod = 100;
    }
}
```

Polecenie `if` wygląda podobnie do definicji funkcji, jednakże jest to tylko powierzchniowe podobieństwo. Wyrażenie zawarte w nawiasach nie jest argumentem. Wyrażenie to nazywamy **warunkiem**. W zaprezentowanym przykładzie warunkiem jest to, że zmienna `delayPeriod` ma wartość większą od 3000. Jeżeli ten warunek będzie spełniony, wykonane zostaną polecenia objęte nawiasami klamrowymi. W tym przypadku kod zawarty w nawiasach klamrowych nadaje zmiennej `delayPeriod` ponownie wartość 100.

Jeżeli warunek nie będzie spełniony, Arduino zignoruje kod zawarty w nawiasach klamrowych i będzie wykonywało kolejne operacje. W naszym przykładzie po poleceniu `if` nie ma żadnej kolejnej instrukcji, a więc Arduino będzie ponownie wykonywało funkcję `loop`.

Przeanalizowanie sekwencji kodu pozwoli Ci zrozumieć jego działanie. Oto lista przeprowadzanych operacji:

1. Arduino uruchamia funkcję `setup` i inicjalizuje złącze diody LED tak, aby działało ono jako wyjście.
2. Arduino uruchamia funkcję `loop`.
3. Dioda zostaje włączona.
4. Arduino odczekuje określoną ilość czasu.
5. Dioda zostaje wyłączona.
6. Arduino odczekuje określoną ilość czasu.
7. Do wartości zmiennej `delayPeriod` zostaje dodana liczba 100.
8. Jeżeli wartość zmiennej delayPeriod jest większa od 3000, zmiennej tej przypisywana jest ponownie wartość 100.
9. Operacje zostają wykonane ponownie, zaczynając od punktu drugiego.

Zastosowaliśmy znak większości (>). Jest to jeden z przykładów operatorów porównywania. Operatory te zostały omówione w poniższej tabeli:

Tabela 3.1. *Operatory porównywania*

Operator	Znaczenie	Przykłady	Wynik logiczny operacji
<	mniejszy od	9 < 10	prawda
		10 < 10	fałsz
>	większy od	10 > 10	fałsz
		10 > 9	prawda
<=	mniejszy lub równy	9 <= 10	prawda
		10 <= 10	prawda
>=	większy lub równy	10 >= 10	prawda
		10 >= 9	prawda
==	równy	9 == 9	prawda
!=	nierówny	9 != 9	fałsz

W celu porównania dwóch liczb stosowany jest operator ==. Ten podwójny znak równości jest często mylony z pojedynczym znakiem równości (=), który jest stosowany do przypisywania wartości do zmiennych.

Istnieje również nieco inna forma polecenia `if`, która pozwala wykonać jeden zestaw operacji, gdy warunek jest spełniony (jest prawdziwy), i inny zestaw operacji, gdy warunek nie jest spełniony (jest fałszywy). Taka operacja zostanie omówiona na podstawie przykładów zamieszczonych w dalszej części książki.

for

Poza wykonywaniem różnych poleceń ze względu na różne warunki możesz często potrzebować wykonania w programie serii instrukcji określoną liczbę razy. Wiesz, że możesz to wykonać przy użyciu funkcji `loop`. Po wykonaniu wszystkich instrukcji znajdujących się w pętli `loop` instrukcje te zostaną automatycznie wykonane ponownie. Czasem potrzebna jest jednakże bardziej ścisła kontrola nad takim zapętleniem operacji.

Na przykład chcąc napisać szkic powodujący dwadzieścia błysków, a następnie po trzech sekundach przerwy ponownie uruchamiający błyski, możesz powtarzać wielokrotnie tę samą sekwencję kodu w funkcji `loop`, co pokazano poniżej:

```
// szkic 03.07.
int ledPin = 13;
int delayPeriod = 250;

void setup()
{
   pinMode(ledPin, OUTPUT);
}
void loop()
{
   digitalWrite(ledPin, HIGH);
   delay(delayPeriod);
   digitalWrite(ledPin, LOW);
   delay(delayPeriod);

   digitalWrite(ledPin, HIGH);
   delay(delayPeriod);
   digitalWrite(ledPin, LOW);
   delay(delayPeriod);

   digitalWrite(ledPin, HIGH);
   delay(delayPeriod);
   digitalWrite(ledPin, LOW);
   delay(delayPeriod);
//powyższe cztery linie kodu należy powtórzyć jeszcze 17 razy
   delay(3000);
}
```

Taka operacja wymaga napisania dużej ilości kodu. Istnieją lepsze sposoby na przeprowadzenie takiej operacji. Zacznijmy od przeanalizowania zastosowania w tym celu pętli `for`, a następnie przeanalizujmy sposób przeprowadzenia tej operacji przy użyciu licznika i polecenia `if`.

Jak widzisz, poniższy szkic, wykonujący to samo zadanie przy użyciu pętli `for`, jest o wiele krótszy.

```
// szkic 03.08.
int ledPin = 13;
int delayPeriod = 250;

void setup()
{
   pinMode(ledPin, OUTPUT);
}

void loop()
{
   for (int i = 0; i < 20; i ++)
   {
      digitalWrite(ledPin, HIGH);
      delay(delayPeriod);
      digitalWrite(ledPin, LOW);
      delay(delayPeriod);
   }
delay(3000);
}
```

Pętla `for` przypomina funkcję, do której przekazywane są trzy argumenty. W przypadku tej pętli „argumenty" są jednakże oddzielane średnikami, a nie przecinkami. Jest to jedna z zasad języka C. Kompilator poinformuje Cię, gdy zastosujesz nieprawidłowy znak separujący.

Nawiasy po `for` zawierają deklarację zmiennej. Zmienna ta jest stosowana jako licznik. W nawiasach nadajemy również jej wartość początkową — w naszym przypadku jest to 0.

Drugim elementem zawartym w nawiasie jest warunek, który musi być prawdziwy, aby polecenia zawarte w funkcji `loop` były dalej wykonywane. W analizowanym przypadku polecenia będą wykonywane, dopóki zmienna `i` ma wartość mniejszą od 20. Gdy zmienna `i` osiągnie wartość równą 20 lub większą od 20, program przestanie wykonywać instrukcje zawarte wewnątrz funkcji `loop`.

Trzeci element zawarty w nawiasie określa to, co powinno się stać po wykonaniu wszystkich poleceń zawartych w funkcji `loop`. W naszym przykładzie wartość zmiennej `i` jest zwiększana o 1. A więc po dwudziestokrotnym wykonaniu operacji zawartych w funkcji `loop` wartość zmiennej `i` będzie wynosić 20, co spowoduje zakończenie działania funkcji `loop`.

Spróbuj wprowadzić ten kod, a następnie go uruchomić. Jedynym sposobem na opanowanie reguł składni oraz irytującej interpunkcji jest samodzielne wpisywanie kodu. Kompilator poinformuje Cię, gdy popełnisz jakiś błąd. Z czasem wszystkie te skomplikowane zapisy zaczną nabierać sensu.

Potencjalną wadą takiego rozwiązania jest to, że wykonanie funkcji `loop` zabierze dużo czasu. Nie jest to problemem w przypadku poprzedniego szkicu, ponieważ szkic nie wykonuje żadnych innych czynności poza sterowaniem diodą LED. W innych szkicach funkcja `loop` będzie jednakże sprawdzała, czy nie wciśnięto jakiegoś przycisku lub czy komunikacja z urządzeniem zewnętrznym została zakończona. Jeżeli procesor będzie zajęty długotrwa-

łym przetwarzaniem pętli `for`, wymienione operacje mogą zostać niewykonane. Ogólnie rzecz biorąc, dobrą praktyką jest tworzenie funkcji `loop` tak, aby działała jak najszybciej i mogła być jak najczęściej uruchamiana. Poniższy szkic prezentuje sposób, w jaki można osiągnąć taki efekt:

```
// szkic 03.09.
int ledPin = 13;
int delayPeriod = 250;
int count = 0;

void setup()
{
   pinMode(ledPin, OUTPUT);
}

void loop()
{
   digitalWrite(ledPin, HIGH);
   delay(delayPeriod);
   digitalWrite(ledPin, LOW);
   delay(delayPeriod);
   count ++;
   if (count == 20)
   {
      count = 0;
      delay(3000);
   }
}
```

Poniższa linia kodu mogła przyciągnąć Twoją uwagę.

```
count ++;
```

Linia ta zawiera skrót często stosowany w języku C. Poniżej przedstawiono kod wykonujący dokładnie tę samą operację.

```
count = count + 1;
```

Wykonanie operacji zawartych w funkcji `loop` za każdym razem będzie zabierać trochę więcej niż 200 milisekund. Wykonanie operacji zawartych w tej funkcji po raz dwudziesty zajmie ponadto o 3 sekundy dłużej z powodu zaprogramowanego oczekiwania pomiędzy błyskami. W niektórych aplikacjach taki czas może się okazać za długi. Puryści generalnie odradzają stosowanie funkcji `delay`. Nie ma jednego uniwersalnego, najlepszego rozwiązania tego problemu. Rozwiązanie należy dobierać indywidualnie zależnie od potrzeb danej aplikacji.

while

Pętle w języku C można tworzyć również przy użyciu polecenia `while`. Poniżej znajduje się przykład jego zastosowania. W przykładzie tym wykonywana jest ta sama operacja co w przytoczonym wcześniej przykładzie użycia polecenia `for`.

```
int i = 0;
while (i < 20)
{
   digitalWrite(ledPin, HIGH);
   delay(delayPeriod);
   digitalWrite(ledPin, LOW);
   delay(delayPeriod);
   i ++;
}
```

Aby pętla `while` była wykonywana, wyrażenie zawarte w nawiasach musi być prawdziwe. Gdy wyrażenie to przestanie być prawdziwe, program wykona kolejne polecenia, które znajdą się w programie za ostatnim nawiasem klamrowym.

Dyrektywa #define

Istnieje alternatywne rozwiązanie dla stosowania zmiennych do opisu wartości, które nie zmieniają się podczas działania programu (np. do określania numeru złącza). Zamiast zmiennej można w takiej sytuacji zastosować dyrektywę `#define`. Dyrektywa ta pozwala na skorelowanie nazwy z wartością. Każde miejsce, gdzie zostanie użyta dana nazwa, zostanie przed kompilacją szkicu wypełnione określoną wartością.

Przypisanie numeru złącza diody LED można zdefiniować w następujący sposób:

```
#define ledPin 13
```

Zauważ, że w dyrektywie `#define` pomiędzy nazwą a wartością nie stosuje się znaku „=". Na końcu dyrektywy nie umieszcza się nawet średnika. Dzieje się tak, ponieważ dyrektywa nie jest elementem języka C. Jest to dyrektywa przedkompilacyjna, która jest obsługiwana przed kompilacją programu.

Według autora niniejszej książki taki zapis jest trudniejszy do odczytania niż zapis zmiennej. Jego zaletą jest to, że do przechowywania tej dyrektywy nie używa się pamięci. Warto jest rozważyć zastosowanie tej techniki, gdy zależy nam na tym, aby nasz program zajmował jak najmniej pamięci.

Podsumowanie

W niniejszym rozdziale poznałeś podstawy języka C. Możesz sprawić, że dioda LED będzie błyskać na różne ciekawe sposoby. Potrafisz użyć funkcji `Serial.println` do wysłania danych z Arduino do komputera za pośrednictwem interfejsu USB. Dowiedziałeś się, jak można stosować polecenia `if` i `for` w celu kontroli kolejności wykonywania operacji. Umiesz również zmusić Arduino do przeprowadzania podstawowych operacji arytmetycznych.

W kolejnym rozdziale przyjrzymy się bliżej zagadnieniom związanym z funkcjami. W rozdziale tym zaprezentowane zostaną również typy zmiennych inne niż omówiony w tym rozdziale typ `int`.

Rozdział 4.

Funkcje

W tym rozdziale skupimy się głównie na typach funkcji, które będziesz tworzyć samodzielnie. Nie będziemy tu omawiać funkcji wbudowanych i zdefiniowanych — takich jak `digitalWrite` i `delay`.

Warto nauczyć się tworzenia własnych funkcji. Gdy będziesz tworzyć coraz to bardziej skomplikowane szkice, funkcje `setup` i `loop` mogą być bardzo długie i złożone, w wyniku czego analiza tego, co wykonują, może być trudna.

Opanowanie złożoności aplikacji stanowi główny problem związany z tworzeniem oprogramowania. Najlepsi programiści piszą programy, których kod jest łatwy do odczytania i zrozumienia, a do tego nie wymaga zbyt wielu dodatkowych wyjaśnień.

Funkcje są głównym narzędziem służącym do tworzenia przejrzystych szkiców, które można łatwo modyfikować bez ryzyka uszkodzenia całej aplikacji.

Czym jest funkcja?

Funkcja odgrywa rolę małego programu występującego w pisanym przez nas większym programie. W funkcji można zawrzeć pojedyncze czynności, które chcemy wykonać. Zdefiniowana funkcja może być użyta w dowolnym miejscu szkicu, zawiera własne zmienne i własną listę poleceń. Gdy polecenia zawarte w funkcji zostaną wykonane, program powraca do miejsca w szkicu, gdzie znajdował się kod wywołujący funkcję, a następnie wykonywane są kolejne polecenia.

Na przykład szkic powodujący błyskanie diody LED jest przykładem kodu, w którym pewne elementy powinny zostać zamknięte wewnątrz funkcji. A więc zmodyfikujmy szkic powodujący dwudziestokrotny błysk diody. Stworzymy nowy szkic, w którym umieścimy funkcję o nazwie `flash` (z ang. błysk):

```
// szkic 04.01.
int ledPin = 13;
int delayPeriod = 250;
```

```
void setup()
{
   pinMode(ledPin, OUTPUT);
}

void loop()
{
   for (int i = 0; i < 20; i ++)
   {
      flash();
   }
   delay(3000);
}

void flash()
{
   digitalWrite(ledPin, HIGH);
   delay(delayPeriod);
   digitalWrite(ledPin, LOW);
   delay(delayPeriod);
}
```

Tak naprawdę jedyne, co zrobiliśmy, to przeniesienie kodu powodującego błyskanie diody LED ze środka pętli `for` do oddzielnej funkcji o nazwie `flash`. Teraz możesz spowodować dowolną liczbę błysków, wywołując nową funkcję przy użyciu polecenia `flash()`. Zwróć uwagę na puste nawiasy umieszczone bezpośrednio za nazwą funkcji. Oznaczają one, że funkcja nie pobiera żadnych parametrów. Wartość opóźnienia jest określana przez funkcję `delayPeriod`, która była zastosowana wcześniej.

Parametry

Dzieląc szkic na funkcje, warto pomyśleć o rolach, jakie mają odgrywać poszczególne funkcje. Rola funkcji `flash` jest oczywista. Spróbujmy dodać do funkcji parametry określające, ile razy ma nastąpić błysk oraz jak długo te błyski mają trwać. Przeanalizuj zamieszczony poniżej kod. Działanie zastosowanych w nim parametrów omówię za chwilę.

```
// szkic 04.02.
int ledPin = 13;
int delayPeriod = 250;

void setup()
{
   pinMode(ledPin, OUTPUT);
}

void loop()
{
   flash(20, delayPeriod);
   delay(3000);
}
```

```
void flash(int numFlashes, int d)
{
   for (int i = 0; i < numFlashes; i ++)
   {
      digitalWrite(ledPin, HIGH);
      delay(d);
      digitalWrite(ledPin, LOW);
      delay(d);
   }
}
```

Przyjrzyj się funkcji `loop`. Funkcja ta zawiera tylko dwie linie kodu. Dużą porcję kodu przełożyliśmy do funkcji `flash`. Zauważ, że wywołując funkcje `flash`, przekazujemy do niej dwa argumenty umieszczone w nawiasach. Definiując funkcję pod koniec szkicu, musimy zadeklarować typy zmiennych umieszczonych w nawiasach. W omawianym przykładzie obie zmienne są typu `int`. Tak naprawdę definiujemy nowe zmienne. Zmienne `numFlash` i `d` mogą być stosowane tylko wewnątrz funkcji `flash`.

Funkcja `flash` jest napisana poprawnie, ponieważ obejmuje wszystko, co jest niezbędne do wykonania operacji błyskania diodą LED. Jedyną informacją, jaką musisz dostarczyć do tej funkcji z zewnątrz, jest numer złącza, do którego podłączona jest dioda LED. Numer złącza może również zostać potraktowany jako parametr. Byłoby to potrzebne, gdybyśmy posiadali więcej diod LED podłączonych do Arduino.

Zmienne globalne, lokalne i statyczne

Jak już wspominałem wcześniej, parametry przekazane do funkcji mogą być stosowane tylko wewnątrz danej funkcji. A więc gdybyś próbował skompilować poniższy kod, wyświetliłby się komunikat błędu:

```
void indicate(int x)
{
   flash(x, 10);
}
x = 15;
```

A teraz załóżmy, że napisałeś taki kod:

```
int x;
void indicate(int x)
{
   flash(x, 10);
}
x = 15;
```

Kod ten nie spowodowałby wyświetlenia błędu podczas kompilacji. Musisz jednakże uważać, ponieważ stworzyłeś dwie zmienne o nazwie `x`, a każda z tych zmiennych ma inną wartość. Zmienna zadeklarowana w pierwszej linii kodu jest **zmienną globalną**. Może być ona zastosowana w dowolnym miejscu programu, wewnątrz dowolnej funkcji.

Nie możesz jednakże stosować globalnej zmiennej x wewnątrz funkcji, ponieważ nazwałeś zmienną znajdującą się wewnątrz funkcji tą samą nazwą, a „lokalna" wersja zmiennej x ma pierwszeństwo przed zmienną globalną. Można powiedzieć, że parametr x usunął w cień globalną zmienną o tej samej nazwie. Może to prowadzić do problemów podczas debugowania projektu.

Poza parametrami możesz również definiować zmienne, które nie są parametrami, a które mogą być zastosowane tylko w obrębie danej funkcji. Zmienne takie nazywane są **zmiennymi lokalnymi**. Poniżej znajduje się przykład takiej zmiennej:

```
void indicate(int x)
{
   int timesToFlash = x * 2;
   flash(timesToFlash, 10);
}
```

Lokalna zmienna o nazwie `timesToFlash` będzie istniała tylko podczas wykonywania funkcji. Funkcja przestanie istnieć w chwili, w której zostanie wykonane ostatnie polecenie zawarte w funkcji. Nie można uzyskać dostępu do zmiennych lokalnych z jakiegokolwiek miejsca programu znajdującego się poza funkcją, w której zdefiniowano daną zmienną. A więc poniższy kod jest błędny:

```
void indicate(int x)
{
   int timesToFlash = x * 2;
   flash(timesToFlash, 10);
}
timesToFlash = 15;
```

Doświadczeni programiści starają się unikać stosowania zmiennych globalnych, ponieważ zmienne te nie są zgodne z zasadami hermetyzacji. **Hermetyzacja** polega na zamykaniu wszystkiego, co ma służyć jednemu konkretnemu celowi, wewnątrz jakiejś określonej struktury. Funkcje są doskonałym narzędziem służącym hermetyzacji. Problem ze zmiennymi globalnymi polega na tym, że są one zwykle definiowane na samym początku szkicu i mogą być stosowane w dowolnym jego miejscu. Czasami tworzenie takich zmiennych ma swoje uzasadnienie. Czasami jednak zmienne globalne są stosowane z powodu lenistwa, w miejscach, gdzie bardziej odpowiednim zabiegiem byłoby przekazywanie parametrów. W dotychczas omówionych szkicach przykładem dobrego zastosowania zmiennej globalnej była zmienna `ledPin`. Umieszczenie tej zmiennej na początku szkicu sprawia, że jest ona łatwa do odnalezienia i zmodyfikowania. Tak naprawdę zmienna `ledPin` jest stałą. Możesz zmienić wartość tej zmiennej i ponownie skompilować szkic, jednakże wartość przechowywana przez tę zmienną nie ulega zmianie podczas działania programu. Dlatego możesz w tym przypadku zastosować dyrektywę `#define` omówioną w rozdziale 3.

Wartości lokalnych zmiennych są inicjalizowane podczas każdego wywołania funkcji. Inicjalizacja taka ma miejsce w funkcji `loop`, co może być kłopotliwe. Spróbujmy zmodyfikować jeden z omówionych wcześniej przykładów i zamiast zmiennej globalnej zastosować zmienną lokalną:

```
// szkic 04.03.
int ledPin = 13;
int delayPeriod = 250;

void setup()
{
   pinMode(ledPin, OUTPUT);
}

void loop()
{
   int count = 0;
   digitalWrite(ledPin, HIGH);
   delay(delayPeriod);
   digitalWrite(ledPin, LOW);
   delay(delayPeriod);
   count ++;
   if (count == 20)
   {
      count = 0;
      delay(3000);
   }
}
```

Szkic *04.03.* jest oparty na szkicu *03.09.*, jednakże zastosowano w nim zmienną lokalną w miejscu uprzednio stosowanej zmiennej globalnej, która miała zliczać liczbę błysków.

Zaprezentowany szkic zawiera błąd. Nie będzie on działał prawidłowo, ponieważ przy każdym wykonaniu funkcji `loop` zmiennej `count` będzie przypisywana ponownie wartość 0, a wiec zmienna `count` nigdy nie osiągnie wartości 20. Miganie diody LED nigdy nie zostanie przerwane. Zmienna `count` była zmienną globalną, ponieważ nie chcieliśmy, aby jej wartość była kasowana po każdym wykonaniu funkcji `loop`. Funkcja `loop` jest jednakże jedynym miejscem, gdzie stosowana jest zmienna `count`, a więc teoretycznie zmienna ta powinna być umieszczona wewnątrz tej funkcji.

Na szczęście język C zawiera mechanizm pozwalający na rozwiązanie tego problemu. Tym rozwiązaniem jest słowo kluczowe `static`. Stosując to słowo kluczowe przed deklaracją zmiennej w funkcji, sprawiasz, że zmienna ta będzie inicjalizowana tylko podczas pierwszego uruchomienia funkcji. Jest to idealne rozwiązanie naszego problemu. Możemy umieścić zmienną w funkcji, a wartość tej zmiennej nie będzie zmieniana na 0 przy każdym uruchomieniu funkcji. Omawiane rozwiązanie zastosowano w szkicu *04.04.*:

```
// szkic 04.04.
int ledPin = 13;
int delayPeriod = 250;

void setup()
{
   pinMode(ledPin, OUTPUT);
}

void loop()
{
   static int count = 0;
```

```
    digitalWrite(ledPin, HIGH);
    delay(delayPeriod);
    digitalWrite(ledPin, LOW);
    delay(delayPeriod);
    count ++;
    if (count == 20)
    {
       count = 0;
       delay(3000);
    }
}
```

Zwracanie wartości

Informatyka jest dyscypliną naukową zrodzoną z połączenia matematyki i techniki. Dziedzictwo związane z tymi dwoma dziedzinami widać w nazewnictwie stosowanym w informatyce. Słowo **funkcja** jest terminem matematycznym. W matematyce argument (to, co dostarczamy funkcji) decyduje o tym, co dana funkcja zwraca. Tworzyliśmy funkcje, do których przekazywaliśmy jakieś wartości, ale funkcje te nie zwracały nam niczego. Wszystkie nasze funkcje były funkcjami typu „void". Jeżeli funkcja ma zwracać jakąś wartość, musisz uprzednio określić typ zwracanej wartości.

Przyjrzyjmy się zapisowi funkcji, do której przekazuje się temperaturę wyrażoną w stopniach Celsjusza. Funkcja ta zwraca tę samą temperaturę wyrażoną w skali Fahrenheita:

```
int centToFaren(int c)
{
   int f = c * 9 / 5 + 32;
   return f;
}
```

Definicja funkcji teraz zaczyna się od int, a nie tak jak wcześniej od void. Wskazuje to, że funkcja zwróci wartość typu `int` do elementu, który ją wywoła. Poniżej znajduje się przykładowy kod wywołujący funkcję:

```
int pleasantTemp = centToFaren(20);
```

Każda funkcja typu innego niż `void` musi zawierać polecenie `return`. Kompilator poinformuje Cię, jeżeli polecenie to zostanie pominięte. Funkcja może zawierać więcej niż jedno polecenie `return`. Do takiej sytuacji może dojść, gdy w funkcji zastosowano instrukcję warunkową `if`. Może być to problematyczne dla programistów. Jeżeli będziesz jednakże tworzyć zwięzłe funkcje (a właśnie takie powinny one być), nie powinieneś mieć żadnych kłopotów z tym mechanizmem.

Nie ma obowiązku umieszczania nazwy zmiennej po poleceniu `return`. Zamiast nazwy zmiennej można tam umieścić wyrażenie. A więc wcześniejszy przykład można zapisać w następujący sposób:

```
int centToFaren(int c)
{
   return (f = c * 9 / 5 + 32);
}
```

Jeżeli zwracane wyrażenie jest czymś więcej niż tylko nazwą zmiennej, wtedy wyrażenie to powinno być umieszczone w nawiasach tak jak w poprzednim przykładzie.

Zmienne innych typów

Wszystkie omówione dotychczas zmienne były zmiennymi typu `int`. Jest to najczęściej stosowany typ zmiennych. Istnieją również jednakże inne typy zmiennych, o których także powinieneś coś wiedzieć.

float

Typ `float` był używany w przykładzie zamieniającym skale temperatur. Zmienne tego typu przechowują liczby zmiennoprzecinkowe. Są to liczby, które zawierają separator dziesiętny, np. 1,23. Zmienne tego typu są stosowane, gdy nie można zapisać jakichś danych wystarczająco dokładnie przy użyciu liczb całkowitych.

Przeanalizuj poniższy zapis:

```
f = c * 9 / 5 + 32
```

Jeżeli zmiennej `c` przypiszesz wartość 17, to zmienna `f` po wykonaniu obliczeń będzie miała wartość 62,6. Gdyby zmienna `f` była typu `int`, to jej wartość zostałaby zapisana jako 62.

Problem ten może się pogłębić, jeżeli nie zwracamy uwagi na kolejność wykonywanych obliczeń. Załóżmy, że najpierw przeprowadzimy operację dzielenia, tak jak pokazano to poniżej:

```
f = (c / 5) * 9 + 32
```

Teoretycznie według reguł matematycznych wynik powinien w dalszym ciągu wynosić 62,6, jednakże w przypadku gdy wszystkie liczby są typu `int`, obliczenia przebiegają w następujący sposób:

1. 17 jest dzielone przez 5. Daje to w wyniku 3,4. Wynik ten jest jednakże ucinany do 3.
2. 3 jest następnie mnożone przez 9, a do wyniku tej operacji dodaje się 32, co w rezultacie daje 59 — wartość wyraźnie różną od 62,6.

W tego typu przypadkach należy stosować zmienne typu `float`. W poniższym przykładzie funkcja dokonująca konwersji skali temperatur została napisana tak, aby zastosowane w niej zmienne były typu `float`:

```
float centToFaren(float c)
{
   float f = c * 9.0 / 5.0 + 32.0;
   return f;
}
```

Zauważ, że dodano `.0` na końcu stałych. Dzięki temu kompilator wie, że liczby te należy traktować jako dane typu `float`, a nie `int`.

boolean

Wartości typu `boolean` są wartościami logicznymi. Wartości takie mogą reprezentować dwa stany — prawdę lub fałsz.

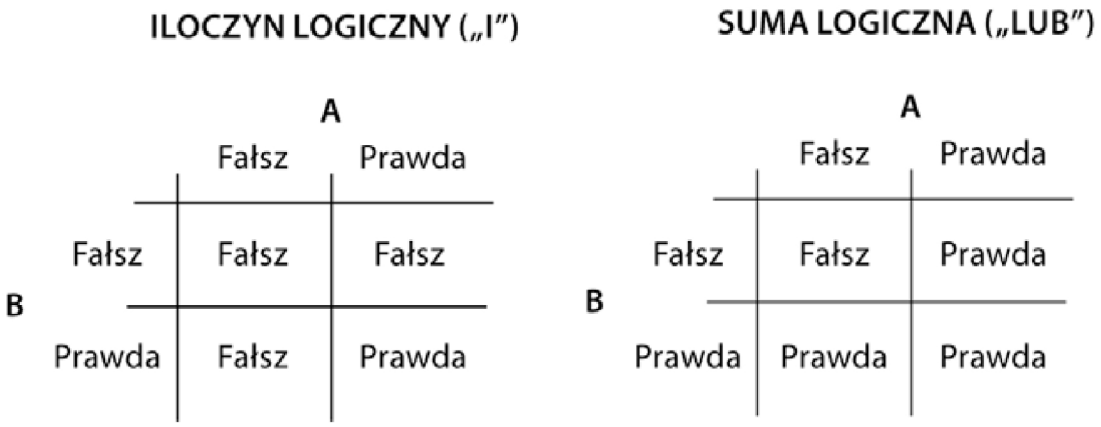
W języku C słowo **boolean** jest zapisywane z małej litery, jednakże wyraz ten jest zapisywany w języku angielskim zwykle z wielkiej litery. Wyraz ten jest przymiotnikiem utworzonym od nazwiska George'a Boole'a — matematyka, wynalazcy algebry Boole'a, która jest bardzo ważna w informatyce.

Możesz nie zdawać sobie z tego sprawy, ale wartości logiczne spotkałeś już wcześniej podczas analizy polecenia `if`. Warunek zawarty w poleceniu `if` — taki jak np. `(count==20)` jest tak naprawdę wyrażeniem dającym w wyniku wartość logiczną. Operator == jest operatorem porównywania. Operator + jest operatorem arytmetycznym dodającym do siebie dwie wartości, natomiast operator == jest operatorem dokonującym operacji porównania dwóch liczb i zwracającym wartość, która określa prawdę lub fałsz.

Poniżej przedstawiono sposób definiowania i stosowania zmiennych przechowujących wartości logiczne:

```
boolean tooBig = (x > 10);
if (tooBig)
{
   x = 5;
}
```

Na wartościach logicznych można przeprowadzać operacje za pomocą operatorów logicznych. Operacje te mają podobny charakter do operacji arytmetycznych przeprowadzanych na liczbach. Najczęściej stosowanymi operatorami logicznymi są: **operator iloczynu logicznego** (zapisywany symbolicznie jako &&) oraz **operator sumy logicznej** (zapisywany symbolicznie jako ||). Na rysunku 4.1 przedstawiono tablice prawdy tych operatorów.



Rysunek 4.1. *Tablice prawdy*

Analizując tablice znajdujące się na rysunku 4.1, możesz zauważyć, że **iloczyn logiczny** daje w wyniku prawdę, gdy zarówno A, jak i B są prawdą. W pozostałych przypadkach wynikiem będzie fałsz.

Z drugiej strony **suma logiczna** daje w wyniku fałsz tylko, gdy zarówno A, jak i B są fałszem. W pozostałych przypadkach w wyniku otrzymamy prawdę.

Poza dwoma omówionymi operatorami logicznymi stosowany jest jeszcze **symbol negacji** zapisywany jako `!`. Chyba nie dziwi Cię, że zaprzeczeniem prawdy jest fałsz, a zaprzeczeniem fałszu jest prawda.

Możesz łączyć operatory logiczne, tworząc bardziej złożone instrukcje warunkowe. Poniżej znajduje się przykład takiej instrukcji:

```
if ((x > 10) && (x < 50))
```

Inne typy danych

Jak już pewnie zauważyłeś, najczęściej stosowanymi typami danych są `int` oraz `float`. Istnieją jednakże sytuacje, w których lepiej byłoby zastosować inny typ zmiennych. W przypadku szkiców Arduino zapis zmiennej typu `int` zajmuje 16 bitów (cyfr binarnych). Pozwala to na zapis cyfr z przedziału od –32 768 do 32 767.

W tabeli 4.1 przedstawiono inne typy danych, które możesz stosować w szkicach. Tabela ta ma charakter poglądowy. Zastosowanie niektórych z zaprezentowanych typów danych zostanie omówione w dalszej części książki.

Tabela 4.1. *Typy danych w języku C*

Typ	Zajmowany obszar pamięci (w bajtach)	Przedział	Uwagi
`boolean`	1	prawda lub fałsz (0 lub 1)	
`char`	1	–128 do +128	Używany do zapisu znaków kodu ASCII. Np. liczba 65 reprezentuje literę A. Zwykle stosowane są tylko wartości dodatnie.
`byte`	1	0 do 255	Stosowany często jako pojedyncza jednostka danych podczas komunikacji za pośrednictwem portu szeregowego. Patrz rozdział 9.
`int`	2	–32768 do +32767	
`unsigned int`	2	0 do 65536	Może być stosowany w celu zwiększenia zakresu opisywanych liczb tam, gdzie niepotrzebny jest zapis liczb ujemnych. Przeprowadzanie działań arytmetycznych na zmiennych typu `unsigned int` i `int` może prowadzić do błędnych wyników.
`long`	4	–2147483648 do +2147483647	Potrzebny tylko do zapisu bardzo dużych wartości.
`unsigned long`	4	0 do 4294967295	Patrz `unsigned int`.
`float`	4	–3,4028235E+38 do +3,4028235E+38	
`double`	4	Taki sam jak w przypadku `float`	Normalnie ten typ danych byłby ośmiobitowy i posiadałby większą precyzję od typu `float`. W przypadku Arduino zakresy tych dwóch typów danych są jednakże identyczne.

Należy uważać na to, żeby nie przekroczyć zakresów poszczególnych typów danych. Jeżeli do zmiennej typu `byte` o wartości 255 dodasz 1, to w wyniku takiej operacji otrzymasz 0. Jeżeli do zmiennej typu `int` o wartości 32 767 dodasz 1, to w wyniku otrzymasz wartość –32 768.

Jeżeli nie czujesz się pewnie, wybierając odpowiedni typ danych, to powinieneś trzymać się typu `int`. Ten typ danych sprawdza się w większości zastosowań.

Styl zapisu kodu

W przypadku języka C sposób zapisu kodu — jego układ na stronie — jest bez znaczenia z punktu widzenia kompilatora. Tak naprawdę możesz zapisać cały program w jednej linii kodu, stosując średniki pomiędzy poszczególnymi poleceniami. Czytelnie zapisany kod jest jednakże o wiele łatwiejszy do przeglądania i poprawiania. Sposób formatowania programu ułatwia jego czytanie tak, jak odpowiednie rozmieszczenie tekstu na stronie książki ułatwia jej lekturę.

Do pewnego stopnia formatowanie jest kwestią indywidualnych preferencji. Nikt nie chciałby zostać posądzony o brak gustu, a więc dyskusja na temat formatowania kodu może dotykać spraw osobistych. Programiści, pracując w zespołach, często zaczynają swoją pracę od przeformatowania czyjegoś kodu tak, aby odpowiadał preferowanemu przez nich stylowi.

Aby rozwiązać wynikające z tego problemy, powstały standardy kodowania. Zachęcają one programistów do zapisywania kodu w podobny sposób i do stosowania „dobrych praktyk" podczas pisania programów.

Język C posiada pewien standard, który ewoluował przez lata. Niniejsza książka jest generalnie zgodna z tym standardem.

Wcięcia

Zapewne zauważyłeś, że w omawianych szkicach stosujemy wcięcia — odsuwamy fragmenty kodu od lewego marginesu. A więc zapisując definicję funkcji typu `void`, słowo kluczowe `void` zapisujemy obok lewego marginesu, tak samo jak nawias klamrowy znajdujący się w kolejnej linii kodu. Cały kod zawarty pomiędzy klamrami jest jednak nieco bardziej odsunięty od lewego marginesu. Rozmiar wcięcia tak naprawdę jest bez znaczenia. Niektórzy programiści stosują w tym celu trzy lub cztery spacje. Wcięcie można również wykonać za pomocą klawisza tabulacji. W niniejszej książce wcięcia są wykonywane przy użyciu trzech znaków spacji.

Gdybyś umieszczał instrukcję `if` wewnątrz definicji funkcji, musiałbyś wtedy dodać kolejne wcięcie dla dwóch linii kodu znajdujących się pomiędzy nawiasami klamrowymi instrukcji `if`:

```
void loop()
{
   static int count = 0;
   count ++;
   if (count == 20)
```

```
    {
      count = 0;
      delay(3000);
    }
}
```

Wewnątrz instrukcji `if` możesz umieścić kolejną instrukcję `if`, a to spowodowałoby konieczność wykonania kolejnego wcięcia (w naszym przypadku byłoby to dziewięć spacji od lewego marginesu).

Może to się wydawać nieco trywialne, ale odczytanie źle sformatowanego szkicu napisanego przez inną osobę może okazać się bardzo trudne.

Nawiasy klamrowe otwierające

Istnieją dwie teorie mówiące o tym, gdzie w definicjach funkcji, instrukcji `if` lub pętli `for` ma znajdować się pierwszy nawias klamrowy. Pierwsza teoria mówi o tym, że nawias ten ma się znajdować w kolejnej linii, pod resztą polecenia — tak jak było to pokazane w dotychczas omawianych przykładach. Nawias ten może jednakże być umieszczony w tej samej linii co polecenie:

```
void loop() {
   static int count = 0;
   count ++; if (count == 20) {
      count = 0;
      delay(3000);
   }
}
```

Taki styl zapisu jest najczęściej spotykany w programach napisanych w języku Java. Język ten ma składnię bardzo podobną do języka C. Ja jednakże preferuję opisany wcześniej sposób zapisu, który jest bardziej popularny w świecie szkiców Arduino.

Białe znaki

Kompilator ignoruje znaki spacji, znaki tabulacji i znaki nowego wiersza. Służą one jedynie rozdzielaniu symboli i słów w szkicu. A więc poniższy przykład zostanie bezproblemowo skompilowany:

```
void loop() {static int
count=0;count++;if(
count==20){count=0;
delay(3000);}}
```

Zapis jest poprawny, jednakże bardzo trudny do odczytania.

Niektórzy programiści zapisują operację przypisania w następujący sposób:

```
int a = 10;
```

Inni wolą stosować następujący zapis:

```
int a=10;
```

Każdy z tych styli jest poprawny. Wybór stylu zależy od indywidualnych preferencji programisty. Ważne jest, żeby trzymać się raz obranego stylu. Ja stosuję pierwszy styl zapisu.

Komentarze

Komentarze są tekstem umieszczonym w szkicu obok kodu programu, jednakże nie pełnią one żadnej funkcji w pisanym programie. Komentarze mają na celu przypominać Tobie lub innym programistom o tym, dlaczego dany fragment kodu został zapisany w taki, a nie inny sposób. Komentarz może być również stosowany do zapisania tytułu.

Komentarze są ignorowane przez kompilator podczas kompilacji programu. Dotychczas widziałeś, że tytuły szkiców zaprezentowanych w tej książce są zapisane w formie komentarzy.

Istnieją dwa sposoby zapisu komentarzy:

- komentarz jednoliniowy rozpoczynający się od // i kończący wraz z końcem danego wiersza,
- komentarz wieloliniowy rozpoczynający się od /*, a kończący na */.

W poniższym przykładzie pokazano zastosowanie obu typów komentarzy:

```
/* niezbyt przydatna funkcja loop.
Napisana przez: Simona Monka
W celu zademonstrowania zastosowania komentarzy
*/
void loop() {
   static int count = 0;
   count ++; // komentarz jednoliniowy
   if (count == 20) {
      count = 0;
      delay(3000);
   }
}
```

W niniejszej książce korzystam głównie z komentarzy jednoliniowych.

Dobry komentarz pomaga zrozumieć, co dzieje się w szkicu lub do czego dany szkic może być zastosowany. Komentarze przydają się innym osobom korzystającym z Twoich szkiców. Mogą się one również przydać Tobie podczas analizy szkicu, który napisałeś kilka tygodni wcześniej.

Niektóre kursy programowania uczą, że im więcej komentarzy zawiera dany program, tym lepiej. Doświadczeni programiści twierdzą jednakże, że dobrze napisany program nie wymaga zbyt wielu komentarzy. Powinieneś stosować komentarze w celu:

- wyjaśniania wszystkiego, co zrobiłeś, a co na pierwszy rzut oka może wydawać się skomplikowane;
- opisywania czynności wymaganych od użytkownika, a niebędących jednocześnie częścią programu, np. *// do tego złącza należy podłączyć tranzystor sterujący przekaźnikiem*;
- wykonywania notatek przypominających o konieczności zrobienia czegoś, np. *//zrób to: uporządkuj ten fragment kodu.*

Ostatni punkt pokazuje bardzo przydatne zastosowanie komentarzy. Programiści bardzo często wstawiają w kodzie programu komentarze, które mają im przypomnieć o czymś, co trzeba będzie później wykonać. Aby znaleźć tego typu komentarze, wystarczy skorzystać z funkcji wyszukiwania oferowanej przez środowisko, w którym pracujemy, i odnaleźć sekwencję `//zrób to:` w kodzie programu.

Komentarzy **nie** należy stosować do:

- opisywania rzeczy oczywistych, np: `a = a+1;` *// dodaje 1 do a*;
- wyjaśniania źle zapisanego kodu.

Powinieneś poprawić zapis takiego kodu.

Podsumowanie

Był to dosyć teoretyczny rozdział. Musiałeś jednak zrozumieć pewne zagadnienia związane z podziałem szkicu na funkcje, a także ze stylem programowania. Pozwoli Ci to później zaoszczędzić dużo czasu.

W kolejnym rozdziale zaczniemy stosować zdobytą wiedzę w praktyce. Omówimy również zagadnienia związane z tworzeniem struktur danych, a także z zastosowaniem łańcuchów znaków.

Rozdział 5.

Tablice i łańcuchy

Po lekturze rozdziału 4. już wiesz, jak należy dzielić szkic tak, aby łatwiej się z nim pracowało. Każdy dobry programista lubi, gdy praca przebiega łatwo i sprawnie. Teraz przyjrzymy się strukturom danych, które będziesz stosować w swoich szkicach.

Książka pod tytułem *Algorytmy + struktury danych = programy* napisana przez Niklausa Wirtha została wydana już jakiś czas temu, jednakże wciąż dobrze opisuje sedno informatyki, a zwłaszcza programowania. Tę książkę mogę polecić każdej osobie zmagającej się z błędami w tworzonych programach. Książka opisuje ideę, w myśl której pisząc program, należy myśleć zarówno o algorytmie (przeprowadzanych operacjach), jak również o stosowanych strukturach danych.

Przeanalizowałeś już pewne funkcje, pętle i instrukcje, czyli coś, co można określić mianem „algorytmicznej" strony programowania Arduino. Teraz przeanalizujemy różne struktury danych.

Tablice

Tablice służą do przechowywania zbioru zmiennych. Zmienne, które do tej pory poznałeś, przechowywały tylko jedną wartość. Zwykle była to wartość typu `int`. Tablica natomiast zawiera listę zmiennych. Korzystając z tej listy, możesz uzyskać dostęp do dowolnej wartości zapisanej w tablicy.

W języku C, tak jak w większości języków programowania, numeracja indeksów zaczyna się od 0, a nie od 1. A więc pierwszym elementem tablicy jest element o numerze zero.

W celu zademonstrowania działania tablic stworzymy przykładową aplikację, która za pomocą błysków wbudowanej w Arduino diody LED będzie generować alfabetem Morse'a sygnał „SOS".

Alfabet Morse'a był stosowany do komunikacji w XIX i XX wieku. Składa się z kombinacji dwóch znaków — kropki i kreski. Dzięki temu mógł on być przesyłany za pośrednictwem kabli telegraficznych i łącz radiowych. Kod ten również mógł być przekazywany

przy pomocy sygnałów świetlnych. Skrót „SOS" (od ang. *save our souls*) wciąż odgrywa rolę międzynarodowego sygnału oznaczającego wołanie o pomoc.

Litera *S* jest reprezentowana przez trzy krótkie błyski (kropki), a litera *O* — przez trzy długie błyski (kreski). Utworzymy tablicę, która będzie przechowywać elementy typu `int` zawierające informacje o długości każdego błysku. Elementy tablicy zostaną następnie zastosowane w pętli `for` w celu określenia odpowiedniego czasu trwania błysków.

Najpierw przyjrzyjmy się sposobowi tworzenia tablicy elementów typu `int`, które mają określać czas trwania poszczególnych błysków.

```
int durations[] = {200, 200, 200, 500, 500, 500, 200, 200, 200};
```

Umieszczając nawiasy kwadratowe [] za nazwą zmiennej, wskazujemy, że zmienna ta zawiera tablicę.

W zaprezentowanym przykładzie określamy wartości długości błysków w momencie tworzenia tablicy. Deklaracja tych elementów polega na zastosowaniu nawiasów klamrowych, a następnie wpisaniu wartości oddzielonych od siebie przecinkami. Nie zapomnij o umieszczeniu średnika na końcu linii kodu.

Dostęp do dowolnego elementu tablicy możesz uzyskać przy użyciu zapisu z nawiasem kwadratowym. Jeżeli chcesz uzyskać dostęp do pierwszego elementu tablicy, zastosuj poniższy kod:

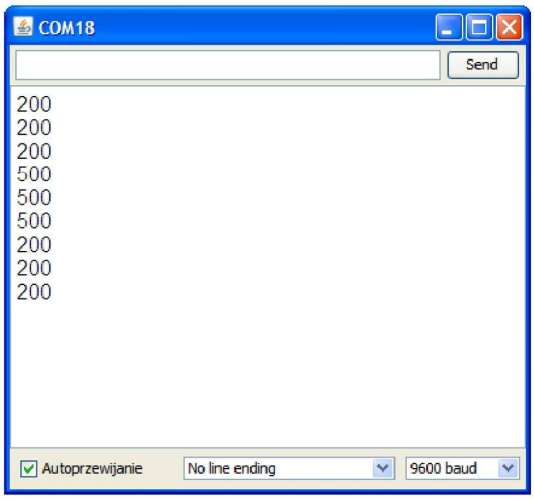
```
durations[0]
```

Aby zademonstrować to w praktyce, stwórzmy tablicę, a następnie wyświetlmy wartości zapisane w tej tablicy przy użyciu monitora portu szeregowego:

```
// szkic 05.01.
int ledPin = 13;

int durations[] = {200, 200, 200, 500, 500, 500, 200, 200, 200};

void setup()
{
   Serial.begin(9600);
   for (int i = 0; i < 9; i++)
   {
      Serial.println(durations[i]);
   }
}

void loop() {}
```

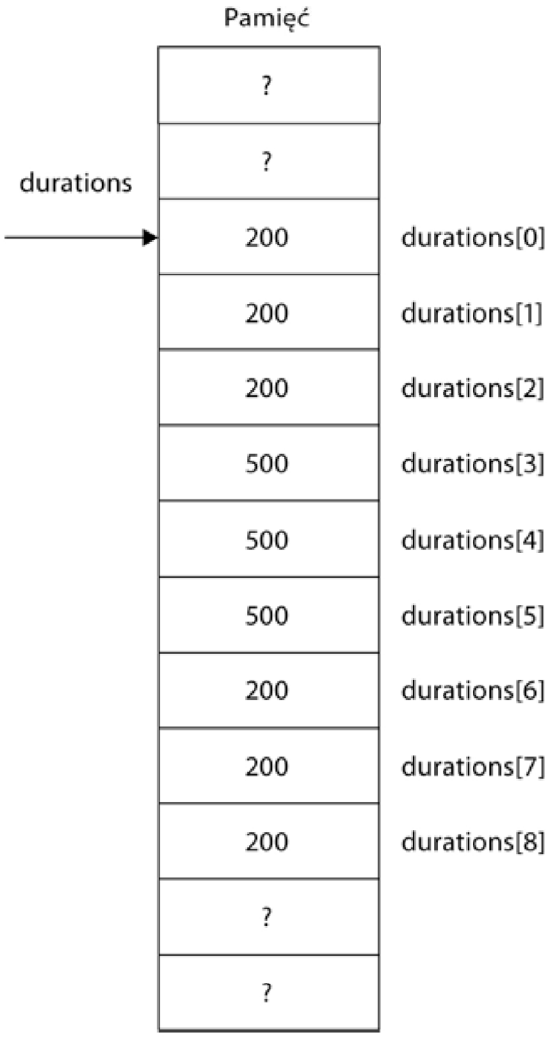
Załaduj szkic do Arduino, a następnie otwórz monitor portu szeregowego. Efekt działania programu powinien być podobny do tego, co przedstawiono na rysunku 5.1.

Powstał zgrabny program, który można łatwo zmodyfikować w celu dodania kolejnych wartości długości błysków. Wystarczyłoby tylko dodać je do listy znajdującej się w nawiasach klamrowych, a następnie w pętli `for` zmienić liczbę *9* na nowy rozmiar tablicy.



Rysunek 5.1. *Monitor portu szeregowego wyświetlający dane wyjściowe szkicu 05.01.*

Podczas pracy z tablicami musisz zachować pewną ostrożność. Kompilator nie powstrzyma próby uzyskania dostępu do danych znajdujących się poza obszarem tablicy. Dzieje się tak, ponieważ tablica jest tak naprawdę wskaźnikiem do adresu w pamięci, co pokazano na rysunku 5.2.



Rysunek 5.2. *Tablice i wskaźniki*

Programy przechowują dane (zarówno zwyczajne zmienne, jak i tablice) w **pamięci**. Pamięć komputera jest zorganizowana w sposób mniej elastyczny od pamięci człowieka. Pamięć Arduino można porównać do zbioru szuflad. Podczas definiowania dziewięcioelementowej tablicy rezerwowana jest przestrzeń kolejnych dziewięciu szuflad. Zmienna tablicowa jest wskaźnikiem pierwszej szuflady, zwanej pierwszym **elementem** tablicy.

Wróćmy do zagadnienia uzyskania dostępu do obszaru pamięci znajdującego się poza obszarem tablicy. Gdybyś zdecydował się uzyskać dostęp do elementu `duration[10]`, zostałaby Ci zwrócona jakaś wartość typu `int`. Ta wartość może być jednakże dosłownie wszystkim. Uzyskanie dostępu do obszaru pamięci znajdującego się poza tablicą nie jest niczym groźnym. Uzyskasz po prostu dostęp do jakiejś przypadkowej wartości niebędącej elementem tablicy. Może to spowodować nieprawidłowe wyniki działania Twojego programu.

O wiele gorsze skutki może spowodować próba zmiany wartości znajdującej się poza obszarem tablicy. Na przykład gdybyś zamieścił w swoim programie poniższy zapis, mógłbyś poważnie zakłócić działanie programu.

```
durations[10] = 0;
```

Szuflada, w której chcemy dokonać zapisu, może już zawierać jakąś inną zmienną. A więc powinieneś bardzo uważać, żeby nie dokonać zapisu poza granicami tablicy. Jeżeli Twój szkic działa w dziwny sposób, powinieneś sprawdzić, czy nie występuje w nim problem tego typu.

Zastosowanie tablic do alfabetu Morse'a i sygnału SOS

W szkicu *05.02.* zaprezentowano zastosowanie tablic w celu wygenerowania sygnału SOS.

```
// szkic 05.02.
int ledPin = 13;

int durations[] = {200, 200, 200, 500, 500, 500, 200, 200, 200};

void setup()
{
   pinMode(ledPin, OUTPUT);
}

void loop()
{
   for (int i = 0; i < 9; i++)
   {
      flash(durations[i]);
   }
   delay(1000);
}

void flash(int duration)
{
   digitalWrite(ledPin, HIGH);
   delay(duration);
   digitalWrite(ledPin, LOW);
   delay(duration);
}
```

Oczywistą zaletą tej techniki jest to, że treść wiadomości można zmienić bardzo łatwo. Wystarczy po prostu zmodyfikować tablicę `durations`. W szkicu *05.05.* zastosujemy tablice w sposób bardziej zaawansowany — stworzymy generator alfabetu Morse'a ogólnego przeznaczenia.

Tablice łańcuchów

W programowaniu łańcuch nie ma nic wspólnego z długim cienkim przedmiotem składającym się z ogniw. Łańcuch jest sekwencją znaków. Dzięki łańcuchom Arduino może obsługiwać teksty. Szkic *05.03.* będzie cyklicznie, co sekundę, przesyłał wiadomość tekstową o treści „Witaj" do monitora portu szeregowego:

```
// szkic 05.03.
void setup()
{
   Serial.begin(9600);
}

void loop()
{
   Serial.println("Witaj");
   delay(1000);
}
```

Literały łańcuchowe

Literały łańcuchowe są zapisywane w podwójnych cudzysłowach. Literały to łańcuchy, które są stałe, w przeciwieństwie do np. zmiennej typu `int`, którą chcemy później modyfikować.

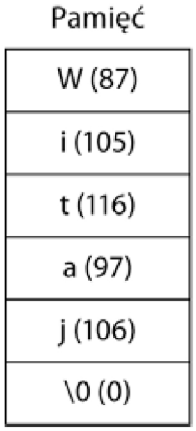
Tak jak zapewne się tego spodziewasz, łańcuchy można umieszczać w zmiennych. Istnieje również zaawansowana biblioteka obsługująca łańcuchy. Na razie będziemy jednakże używać standardowych łańcuchów języka C — takich jak ten, który znajduje się w szkicu *05.03.*

W języku C literał łańcuchowy jest tak naprawdę tablicą elementów typu `char`. Element typu `char` jest trochę podobny do elementu typu `int` — jest to również cyfra — jednakże mieści się ona w zakresie od 0 do 127 i reprezentuje jeden znak. Tym znakiem może być litera alfabetu, cyfra, znak interpunkcyjny lub znak specjalny, taki jak np. znak tabulacji lub znak przesuwu o wiersz. Kody numeryczne zawarte w łańcuchach są oparte na standardzie o nazwie ASCII. Najczęściej stosowane kody ASCII przedstawiono w tabeli 5.1.

Tabela 5.1. *Najczęściej stosowane kody ASCII*

Znak	Kod ASCII (dziesiętny)
a – z	97 – 112
A – Z	65 – 90
0 – 9	48 – 57
spacja	32

Literał łańcuchowy `"Witaj"` jest tak naprawdę tablicą znaków, co pokazano na rysunku 5.3.



Rysunek 5.3. *Literał łańcucha o treści „Witaj”*

Zauważ, że literał kończy się specjalnym znakiem `null` (`\0`). Znak ten jest stosowany do oznaczania końca łańcucha.

Zmienne łańcuchowe

Zmienne łańcuchowe są bardzo podobne do zmiennych tablicowych. Różnicą jest to, że ich wartość początkową można zdefiniować w bardzo prosty sposób:

```
char name[] = "Witaj";
```

Zapis ten definiuje tablicę znaków i inicjalizuje ją łańcuchem znaków o treści „Witaj”. Automatycznie zostanie dodana wartość zerowa (0 w kodzie ASCII) oznaczająca koniec łańcucha.

Wcześniejszy przykład był co prawda zgodny z posiadanymi przez Ciebie wiadomościami dotyczącymi łańcuchów, jednakże częściej spotykany jest następujący zapis:

```
char *name = "Witaj";
```

Jest to zapis równoznaczny z zapisem omówionym wcześniej. Znak * symbolizuje, że mamy do czynienia ze wskaźnikiem. Wskaźnik `name` wskazuje na pierwszy element typu `char` tablicy elementów typu `char`. Jest to miejsce w pamięci, gdzie zapisana jest litera *W*.

Możesz zmodyfikować szkic *05.03.* tak, aby zastosować w nim zarówno zmienne, jak i stałe łańcuchowe:

```
// szkic 05.04.
char message[] = "Witaj";

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

void loop()
```

```
{
   Serial.println(message);
   delay(1000);
}
```

Tłumacz alfabetu Morse'a

Zastosujmy naszą wiedzę dotyczącą tablic i łańcuchów w celu napisania bardziej złożonego szkicu, który będzie odbierał komunikaty przesłane za pośrednictwem monitora portu szeregowego, a następnie przekazywał je w formie alfabetu Morse'a przy użyciu wbudowanej diody LED.

Tabela 5.2. zawiera litery i cyfry stosowane w alfabecie Morse'a.

Tabela 5.2. *Litery i cyfry w alfabecie Morse'a*

A	.-	N	-.	0	-----
B	-...	*O*	---	*1*	.----
C	-.-.	*P*	.--.	*2*	..---
D	-..	*Q*	--.-	*3*	...--
E	.	*R*	.-.	*4*	....-
F	..-.	*S*	...	*5*	.....
G	-.	*T*	-	*6*	-....
H	....	*U*	..-	*7*	--...
I	..	*V*	...-	*8*	---..
J	.---	*W*	.--	*9*	----.
K	-.-	*X*	-..-		
L	.-..	*Y*	-.--		
M	--	*Z*	--..		

Jedna z zasad tego kodu mówi, że kreska powinna być trzy razy dłuższa od kropki. Odstęp pomiędzy każdą kreską lub kropką jest równy długości kropki. Przerwa pomiędzy dwoma literami trwa tyle samo czasu co kreska. Odstęp pomiędzy dwoma słowami powinien trwać tyle samo co siedem kropek.

W tym projekcie nie będziemy dbali o interpunkcję. Samodzielne dodanie do szkicu obsługi interpunkcji byłoby jednakże bardzo ciekawym ćwiczeniem. Pełną listę znaków stosowanych w alfabecie Morse'a znajdziesz pod adresem *http://pl.wikipedia.org/wiki/Kod_Morse'a.*

Dane

Nasz szkic będziemy budować krok po kroku. Zaczniemy od struktury danych, którą będziesz stosować do reprezentacji kodów.

Warto zrozumieć, że każdy problem natury programistycznej ma więcej niż jedno rozwiązanie. Różni programiści tworzą różne rozwiązania tych samych problemów. Nie warto jest więc myśleć, że „nigdy bym na to nie wpadł". Prawdopodobnie wpadłbyś na coś innego, a być może lepszego. Każdy myśli według innych schematów, a zaprezentowane przeze mnie rozwiązania po prostu wpadły mi do głowy jako pierwsze.

Reprezentacja danych sprowadza się do przełożenia na język C zawartości tabeli 5.2. Utworzymy dwa oddzielne łańcuchy — jeden na litery, a drugi na liczby. Struktura danych obsługująca litery ma następującą postać:

```
char* letters[] = {
   ".-", "-...", "-.-.", "-..", ".", //A-I
   "..-.", "--.", "....", "..",
   ".---", "-.-", ".-..", "--", "-.", //J-R
   "---", ".--.", "--.-", ".-.",
   "...", "-", "..-", "...-", ".--", //S-Z
   "-..-", "-.--", "--.."
};
```

Stworzyłeś tablicę literałów łańcuchowych. A przypominam, że literał łańcuchowy jest tak naprawdę tablicą elementów typu `char`, a więc tak naprawdę otrzymałeś tablicę tablic — coś, co jest w pełni zgodne z zasadami języka C, a ponadto jest bardzo przydatne.

Aby odnaleźć odpowiednik litery *A* w alfabecie Morse'a, musisz uzyskać dostęp do `letter[0]`. W wyniku takiej operacji zostałby zwrócony łańcuch `.-`. Metoda ta nie jest efektywna, ponieważ wykorzystujesz cały bajt (8 bitów) pamięci do zapisania kreski lub kropki, które mogłyby być zapisywane przy użyciu pojedynczych bitów. Możesz to jednakże usprawiedliwić faktem, że zajmujesz tylko 90 bajtów, a dostępnych masz około 2000. Ponadto zaprezentowana technika sprawia, że kod jest łatwy do zrozumienia, co jest również ważne.

Wykonajmy podobną tablicę dla liczb:

```
char* numbers[] = {
   "-----", ".----", "..---", "...--",
   "....-", ".....", "-....", "--...", "---..", "----."};
```

Zmienne globalne i funkcja setup

Musisz zdefiniować parę zmiennych globalnych. Jedna z nich będzie określała czas trwania kropki w alfabecie Morse'a, a druga będzie definiowała złącze, do którego podłączona będzie dioda LED:

```
int dotDelay = 200;
int ledPin = 13;
```

Funkcja `setup` jest dosyć prosta. Musisz tylko zaprogramować złącze tak, żeby działało jako wyjście, a także uruchomić port szeregowy:

```
void setup()
{
   pinMode(ledPin, OUTPUT);
   Serial.begin(9600);
}
```

Funkcja loop

Teraz zaczniesz prawdziwą pracę związaną z przetwarzaniem danych. Funkcja `loop` ma następujący algorytm:

- Jeżeli istnieje znak do wczytania, to wczytaj go za pośrednictwem interfejsu USB:
  - Jeżeli jest to litera, wyświetl ją przy użyciu tablicy liter.
  - Jeżeli jest to liczba, wyświetl ją przy użyciu tablicy liczb.
  - Jeżeli jest to spacja, odczekaj czterokrotność czasu trwania kropki.

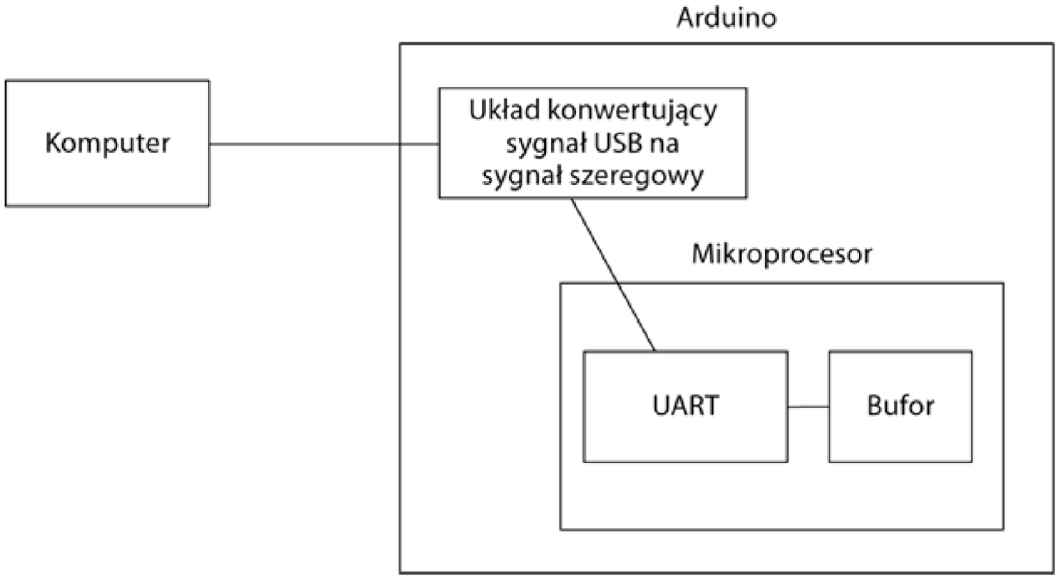
To tyle. Nie powinieneś wybiegać myślami za bardzo do przodu. Algorytm określa to, co chcesz zrobić, lub to, jakie masz **intencje**. Taki styl programowania nosi nazwę **programowania intencyjnego**.

Jeżeli przeniesiesz ten algorytm na język C, to otrzymasz następujący kod:

```
void loop()
{
   char ch;
   if (Serial.available() > 0)
   {
      ch = Serial.read();
      if (ch >= 'a' && ch <= 'z')
      {
         flashSequence(letters[ch - 'a']);
      }
      else if (ch >= 'A' && ch <= 'Z')
      {
         flashSequence(letters[ch - 'A']);
      }
      else if (ch >= '0' && ch <= '9')
      {
         flashSequence(numbers[ch - '0']);
      }
      else if (ch == ' ')
      {
         delay(dotDelay * 4); // odstęp pomiędzy słowami
      }
   }
}
```

Kilka rzeczy ujętych w tym kodzie wymaga wyjaśnienia. Pierwszą rzeczą jest zapis `Serial.available()`. Aby go zrozumieć, musisz posiąść pewną wiedzę dotyczącą komunikacji pomiędzy Arduino a komputerem poprzez złącze USB. Schemat tej komunikacji został przedstawiony na rysunku 5.4.

Dane z monitora portu szeregowego są przesyłane z komputera na płytkę Arduino. Następnie sygnał zgodny ze standardem i protokołem USB jest przetwarzany na sygnał akceptowany przez mikrokontroler znajdujący się na płytce Arduino. Później specjalny układ dokonuje konwersji sygnału. Następnie dane są odbierane przez UART (*Universal Asynchronous Receiver/Transmitter* — z ang. uniwersalny asynchroniczny nadajnik i odbiornik). Jest to element mikrokontrolera, który umieszcza otrzymane dane w buforze. Bufor jest specjalnym miejscem w pamięci (128 bajtów), gdzie przechowywane są dane, które są usuwane zaraz po ich odczytaniu.



Rysunek 5.4. *Komunikacja komputera z Arduino*

Komunikacja przebiega niezależnie od wykonywanego szkicu. Dane będą dostarczane do bufora, gdzie będą czekać na odczyt, nawet podczas wykonywania operacji migania diodami LED. Pracę bufora możesz sobie wyobrazić jako działanie skrzynki odbiorczej poczty elektronicznej.

Nadejście „nowych wiadomości" sprawdzasz za pomocą funkcji `Serial.available()`. Funkcja ta zwraca liczbę bajtów danych czekających na odczyt w buforze. Jeżeli bufor jest pusty, funkcja zwraca wartość 0. Dlatego zastosowaliśmy instrukcję `if`, która sprawdza, czy do odczytu jest więcej niż 0 bajtów danych. Jeżeli w buforze znajdują się jakieś dane do odczytu, to kolejny dostępny element typu `char` jest odczytywany przy użyciu funkcji `Serial.read()`. Funkcja ta zostaje przypisana do zmiennej lokalnej `ch`.

Następnie powinniśmy umieścić kolejną instrukcję `if`, która sprawdzi, jaki znak będziesz wyświetlać za pomocą błysków diody LED:

```
if (ch >= 'a' && ch <= 'z')
{
   flashSequence(letters[ch - 'a']);
}
```

Na początku używanie operatorów <= i >= do porównywania znaków może wydawać się czymś dziwnym. Operatory te możemy jednakże stosować w ten sposób. Każdy znak jest reprezentowany liczbą (kodem ASCII). Jeżeli kod opisuje znaki od *a* do *z* (mieści się w przedziale od 97 do 122), to wiesz, że z komputera została przekazana mała litera. Następnie możesz wywołać nienapisaną jeszcze funkcję `flash.Sequence`. Do tej funkcji przekażemy łańcuch składający się z kropek i kresek. Na przykład w celu wyświetlenia za pomocą błysków diody LED litery *a* przekażemy do tej funkcji w roli argumentu ciąg `.-`.

Przenosisz do tej funkcji całkowitą odpowiedzialność za wykonywanie błysków diodą. Nie próbuj przeprowadzać tej operacji wewnątrz funkcji `loop`. W ten sposób kod będzie bardziej przejrzysty.

Poniżej znajduje się kod określający ciąg kropek i kresek, które następnie będziesz musiał przekazać funkcji `flashSequence`:

```
letters[ch - 'a']
```

Również ten zapis może wydawać się dziwny. Funkcja wydaje się odejmować od siebie znaki. Zapis ten jest jednakże poprawny. W naszym programie odejmujemy wartości będące kodami ASCII.

Pamiętaj o tym, że kody liter przechowujesz w tablicy, a więc pierwszy element tablicy przechowuje ciąg kropek i kresek symbolizujących literę *A*. Drugi element tablicy przechowuje ciąg symbolizujący literę *B* itd. Musisz odnaleźć pozycję w tablicy, której odpowiada litera pobrana z bufora. Pozycja małych liter będzie taka sama jak pozycja liter wielkich. Wystarczy przeprowadzić proste działanie matematyczne polegające na odjęciu wartości kodu symbolizującego literę *a*. Działanie *a–a* to tak naprawdę operacja 97–97 = 0. Podobnie *c–a* to tak naprawdę 99–97 = 2. Jeżeli w przytoczonym powyżej kodzie zmienna `ch` będzie przechowywała literę *c*, to działanie umieszczone w nawiasie da w wyniku wartość 2, a z tablicy zostanie pobrany element o numerze 2, czyli ciąg `-.-..`

Podrozdział, który właśnie przeczytałeś, opisuje obsługę małych liter. Musisz także obsłużyć wielkie litery i cyfry. Ich obsługa będzie przebiegała w podobny sposób.

Funkcja flashSequence

Wcześniej zakładaliśmy istnienie funkcji `flashSequence` i stosowaliśmy ją w kodzie programu. Teraz musimy ją napisać. Planowaliśmy, że funkcji tej będziemy przekazywać łańcuch składający się z serii kropek i kresek, a funkcja ta będzie wykonywała odpowiednią liczbę błysków w odpowiednim czasie.

Obmyślając algorytm tej funkcji, możesz podzielić go na dwa etapy:

- Dla każdego elementu łańcucha kresek i kropek (np. `.-.-`)
  - wykonuj odpowiednio długi błysk.

Stosując zasady programowania intencyjnego, postarajmy się, aby funkcja była jak najprostsza.

Kody Morse'a różnych liter mają różną długość. Będziesz wiec musiał zastosować pętlę przeglądającą łańcuch aż do napotkania znaku `\0`. Będzie Ci również potrzebna zmienna o nazwie `i`, o początkowej wartości 0. Wartość tej zmiennej będzie zwiększana podczas przeglądania kolejnych kropek i kresek:

```
void flashSequence(char* sequence)
{
   int i = 0;
   while (sequence[i] != '\0')
   {
      flashDotOrDash(sequence[i]);
      i++;
   }
   delay(dotDelay * 3); // odstęp pomiędzy literami
}
```

Odpowiedzialność za włączanie i wyłączanie diody LED zostaje przeniesiona na nową funkcję o nazwie `flashDotOrDash`. Gdy program wygeneruje odpowiednie błyski dla wszystkich kropek i kresek, następuje pauza o długości trzech kropek. Zauważ, jak przydatne w tym miejscu okazało się zastosowanie komentarza.

Funkcja flashDotOrDash

Ostatnia brakująca funkcja będzie bezpośrednio włączała i wyłączała diodę LED. Funkcja ta przyjmuje pojedynczy argument, który będzie kropką (`.`) lub kreską (`-`).

Funkcja włącza diodę LED i oczekuje określony okres czasu, jeżeli do funkcji została przekazana kropka. Jeżeli do funkcji została przekazana kreska, funkcja odczekuje trzy razy dłuższy okres czasu. Po tym czasie następuje wyłączenie diody LED. Na koniec funkcja musi odczekać czas równy czasowi „wyświetlania" kropki — w ten sposób zaznaczany jest odstęp pomiędzy błyskami.

```
void flashDotOrDash(char dotOrDash)
{
   digitalWrite(ledPin, HIGH);
   if (dotOrDash == '.')
   {
      delay(dotDelay);
   }
   else //jeżeli nie jest kropką, to musi być kreską
   {
      delay(dotDelay * 3);
   }
   digitalWrite(ledPin, LOW);
   delay(dotDelay); // odstęp pomiędzy błyskami
}
```

Składanie całości programu

Cały program zawierający wszystkie funkcje znajduje się w szkicu *05.05*. Załaduj go na płytkę Arduino i uruchom. Pamiętaj, że aby korzystać z tego szkicu, musisz otworzyć monitor portu szeregowego, wpisać jakiś tekst, a następnie kliknąć ikonę *Send*. Wpisany przez Ciebie tekst powinien zostać przekształcony na alfabet Morse'a przy użyciu wbudowanej w Arduino diody LED.

```
// szkic 05.05.
int ledPin = 13;
int dotDelay = 200;

char* letters[] = {
   ".-", "-...", "-.-.", "-..", ".", "..-.", "--.", "....", "..",     //A-I
   ".---", "-.-", ".-..", "--", "-.", "---", ".--.", "--.-", ".-.",   //J-R
   "...", "-", "..-", "...-", ".--", "-..-", "-.--", "--.."           //S-Z
};

char* numbers[] = {
   "-----", ".----", "..---", "...--", "....-", ".....", "-....", "--...", "---..", "----."};
```

```
void setup()
{
   pinMode(ledPin, OUTPUT);
   Serial.begin(9600);
}

void loop()
{
   char ch;
   if (Serial.available() > 0)
   {
      ch = Serial.read();
      if (ch >= 'a' && ch <= 'z')
      {
         flashSequence(letters[ch - 'a']);
      }
      else if (ch >= 'A' && ch <= 'Z')
      {
         flashSequence(letters[ch - 'A']);
      }
      else if (ch >= '0' && ch <= '9')
      {
         flashSequence(numbers[ch - '0']);
      }
      else if (ch == ' ')
      {
         delay(dotDelay * 4);   // odstęp pomiędzy błyskami
      }
   }
}

void flashSequence(char* sequence)
{
   int i = 0;
   while (sequence[i] != NULL)
   {
      flashDotOrDash(sequence[i]);
      i++;
   }
   delay(dotDelay * 3);      // odstęp pomiędzy literami
}

void flashDotOrDash(char dotOrDash)
{
   digitalWrite(ledPin, HIGH);
   if (dotOrDash == '.')
   {
      delay(dotDelay);
   }
   else // jeżeli nie jest kropką, to musi być kreską
   {
      delay(dotDelay * 3);
   }
   digitalWrite(ledPin, LOW);
   delay(dotDelay); // odstęp pomiędzy błyskami
}
```

W szkicu umieszczono funkcję `loop`, która jest wywoływana automatycznie. Funkcja ta wywołuje napisaną przez Ciebie funkcję `flashSequence`, która to z kolei wywołuje napisaną przez Ciebie funkcję `flashDotOrDash`, a ta wywołuje funkcje `digitalWrite` i `delay`, które to są funkcjami „wbudowanymi" w Arduino.

Tak powinny wyglądać Twoje szkice. Podział szkicu na funkcje pozwala na łatwiejszą implementację kodu. Ułatwia on również powrót do pracy nad szkicem po dłuższej przerwie.

Podsumowanie

W tym rozdziale poznałeś podstawowe wiadomości dotyczące łańcuchów i tablic. Napisałeś dość złożony szkic tłumaczący wprowadzany tekst na alfabet Morse'a. Mam nadzieję, że dzięki temu szkicowi zrozumiałeś, jak ważny jest podział programu na funkcje.

W kolejnym rozdziale opiszemy zagadnienia związane z wejściami i wyjściami. Omówimy wprowadzanie do Arduino, a także wyprowadzanie z niego, sygnałów analogowych i cyfrowych.

Rozdział 6.

Wejścia i wyjścia

Płytkę Arduino można określić mianem komputera, do którego można podłączyć wiele zewnętrznych podzespołów elektronicznych. Musisz więc nauczyć się korzystać z licznych opcji związanych ze złączami znajdującymi się na Arduino.

Wyjścia mogą działać w trybie cyfrowym — napięcie na nich może być przełączane pomiędzy wartościami 0 V i 5 V. Mogą one również działać w trybie analogowym — napięcie na nich może przybierać dowolną wartość w zakresie od 0 V do 5 V. W praktyce nie jest to tak proste, jak Ci się może teraz wydawać.

Wejścia, podobnie jak wyjścia, mogą być cyfrowe (np. mogą wykrywać, czy podłączony przycisk został wciśnięty) lub analogowe (np. obsługiwać światłomierz). Książka, którą czytasz, dotyczy tematyki związanej z oprogramowaniem, a nie ze sprzętem, a więc nie będziemy zbyt szczegółowo omawiać zagadnień związanych z elektroniką. Podczas lektury niniejszego rozdziału może Ci się jednakże przydać multimetr oraz kawałek przewodu.

Wyjścia cyfrowe

W poprzednich rozdziałach korzystaliśmy z diody LED podłączonej do cyfrowego złącza o numerze 13. W rozdziale 5. stosowaliśmy tę diodę do generowania alfabetu Morse'a. Arduino dysponuje wieloma złączami cyfrowymi.

Poeksperymentujmy trochę z innym złączem znajdującym się na płytce Arduino. Będziesz korzystać ze złącza cyfrowego o numerze 4. Aby sprawdzić, co dzieje się na tym złączu, będziesz łączyć multimetr i płytkę Arduino za pomocą przewodów. Połączenie to zostało pokazane na rysunku 6.1. Jeżeli Twój multimetr jest wyposażony w zaciski typu „krokodylki", powinieneś zerwać izolację z końców przewodów i jeden koniec każdego z przewodów podłączyć do zacisków multimetru. Pozostałe końce przewodów należy wpiąć w odpowiednie złącza znajdujące się na Arduino. Jeżeli Twój multimetr nie posiada takich zacisków, to owiń końce przewodów, z których zerwano izolację, wokół probówek. Najlepiej zastosować w tym celu sztywne przewody jednożyłowe.

Rysunek 6.1. *Stosowanie multimetru do pomiarów na złączach Arduino*

Multimetr należy ustawić w tryb umożliwiający pomiar prądu stałego w zakresie od 0 V do 20 V. Ujemny (czarny) przewód miernika należy podłączyć do złącza o zerowym potencjale (GND), a przewód dodatni (czerwony) należy podłączyć do złącza D4. Przewody po prostu łączą miernik z odpowiednimi gniazdami znajdującymi się na płytce Arduino.

Załaduj szkic *06.01.*:

```
//szkic 06.01.

int outPin = 4;

void setup()
{
   pinMode(outPin, OUTPUT);
   Serial.begin(9600);
   Serial.println("Wybierz 1 lub 0");
}
```

```
void loop()
{
   if (Serial.available() > 0)
   {
      char ch = Serial.read();
      if (ch == '1')
      {
         digitalWrite(outPin, HIGH);
      }
      else if (ch == '0')
      {
         digitalWrite(outPin, LOW);
      }
   }
}
```

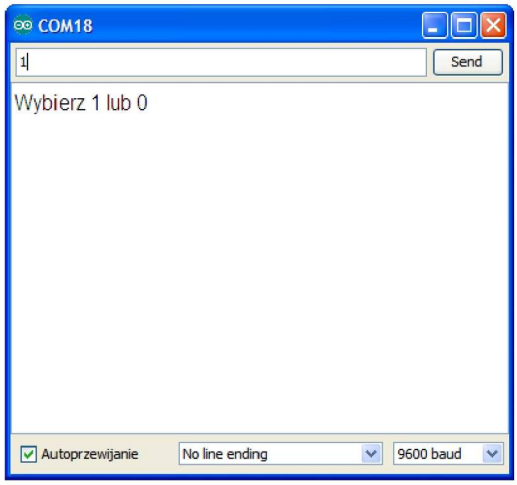
Na początku szkicu znajduje się polecenie `pinMode`. Powinieneś stosować to polecenie dla każdego złącza, z którego korzystasz w projekcie. Dzięki temu Arduino będzie w stanie odpowiednio skonfigurować układy elektroniczne podłączone do danego złącza tak, aby złącze działało jako wejście lub wyjście. Na przykład:

```
pinMode(outPin, OUTPUT);
```

Jak mogłeś się już domyślić, `pinMode` jest funkcją wbudowaną. Pierwszym argumentem funkcji jest numer interesującego nas złącza (`int`), a drugim argumentem jest zdefiniowanie tego, czy interesujące nas złącze ma działać jako wejście (`INPUT`), czy jako wyjście (`OUTPUT`). Drugi argument musi być wpisany wielkimi literami.

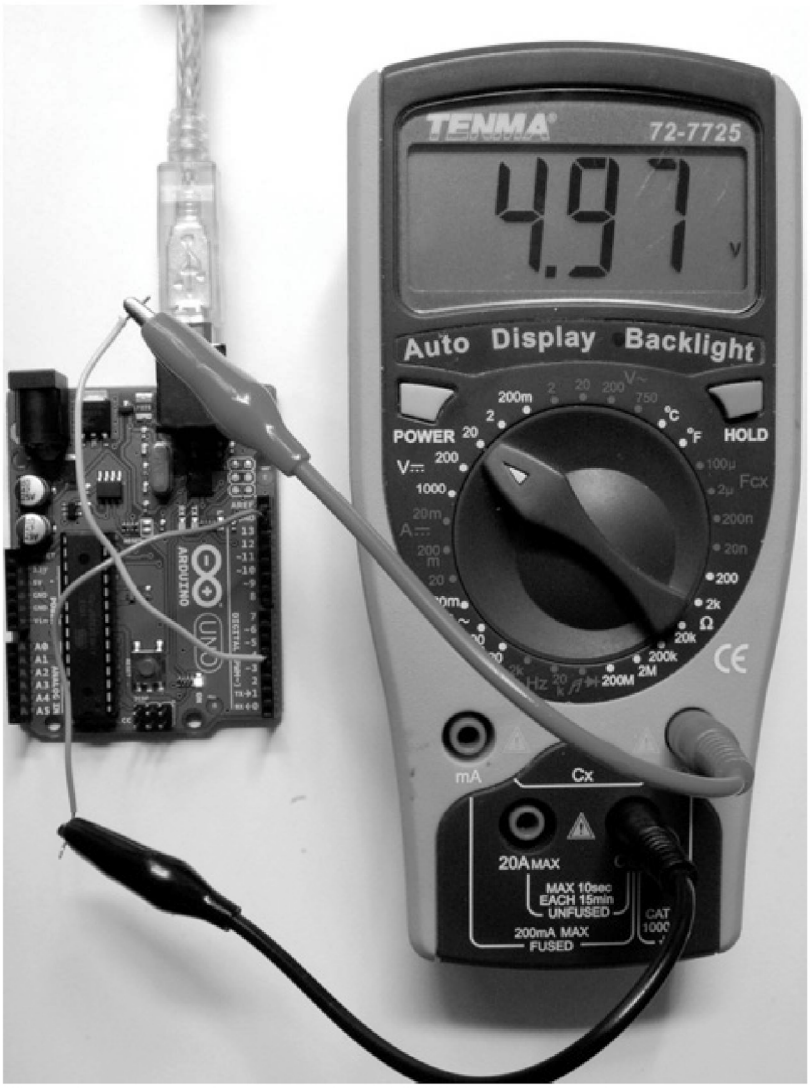
Funkcja `loop` oczekuje na wprowadzenie `1` lub `0` z komputera za pośrednictwem monitora portu szeregowego. Jeżeli zostanie wprowadzona liczba 1, to złącze numer 4 zostanie włączone, w innym przypadku pozostanie ono wyłączone.

Załaduj szkic na swoje Arduino i otwórz monitor portu szeregowego (patrz rysunek 6.2).



Rysunek 6.2. *Monitor portu szeregowego*

Po podłączeniu multimetru do Arduino powinieneś widzieć, jak po przesyłaniu do Arduino (za pośrednictwem monitora portu szeregowego) poleceń w postaci zera lub jedynki napięcie na złączu zmienia się z 0 V do około 5 V. Na rysunku 6.3 przedstawiono wskazania multimetru po wysłaniu 1 za pośrednictwem monitora portu szeregowego.



Rysunek 6.3. *Złącze cyfrowe działające w trybie HIGH*

Jeżeli Twój projekt będzie wymagał większej liczby złączy cyfrowych niż ich liczba na płytce Arduino (złącza oznaczone literą D), możesz posługiwać się złączami oznaczonymi literą A (analogowe). Złącza analogowe mogą działać również jako cyfrowe wyjścia. Aby korzystać z tych złączy, wystarczy dodać 14 do numeru złącza analogowego. Możesz wypróbować działanie tych złączy, modyfikując pierwszy wiersz ostatnio omawianego szkicu tak, aby korzystał on ze złącza o numerze 14. W tym wypadku dodatni przewód multimetru podłącz do złącza A0 znajdującego się na płytce Arduino.

To chyba tak naprawdę wszystko na temat cyfrowych wyjść. Teraz zajmiemy się omawianiem zagadnień związanych z cyfrowymi wejściami.

Wejścia cyfrowe

Wejścia cyfrowe są najczęściej stosowane do detekcji tego, czy obwód włącznika został zamknięty. Złącze cyfrowe może odczytywać dwa stany: włączony i wyłączony. Jeżeli napięcie na złączu jest niższe od 2,5 V (połowa 5 V), odczytana zostanie wartość 0 (wyłączony). Jeżeli napięcie to będzie wynosić więcej niż 2,5 V, odczytana zostanie wartość 1 (włączony).

Odłącz multimetr od płytki Arduino, a następnie załaduj na nią szkic *06.02.*:

```
//szkic 06.02.

int inputPin = 5;

void setup()
{
   pinMode(inputPin, INPUT);
   Serial.begin(9600);
}

void loop()
{
   int reading = digitalRead(inputPin);
   Serial.println(reading);
   delay(1000);
}
```

W funkcji `setup` musisz poinformować Arduino o tym, że masz zamiar korzystać z danego złącza w charakterze wejścia, podobnie jak to miało miejsce w przypadku korzystania ze złącza w charakterze wyjścia. Funkcja `digitalRead` zwraca wartość odczytywaną przez wejście. Funkcja ta zwraca wartości 0 lub 1.

Rezystor podwyższający

Szkic odczytuje stan wejścia i wyświetla odczytaną wartość na monitorze portu szeregowego. Operacja ta przebiega raz na sekundę. Załaduj szkic na Arduino i uruchom monitor portu szeregowego. Powinieneś widzieć wartości pojawiające się w odstępach sekundowych. Podłącz jeden koniec przewodu do gniazda D5, a drugi koniec tego samego przewodu chwyć palcami tak, jak to pokazano na rysunku 6.4.

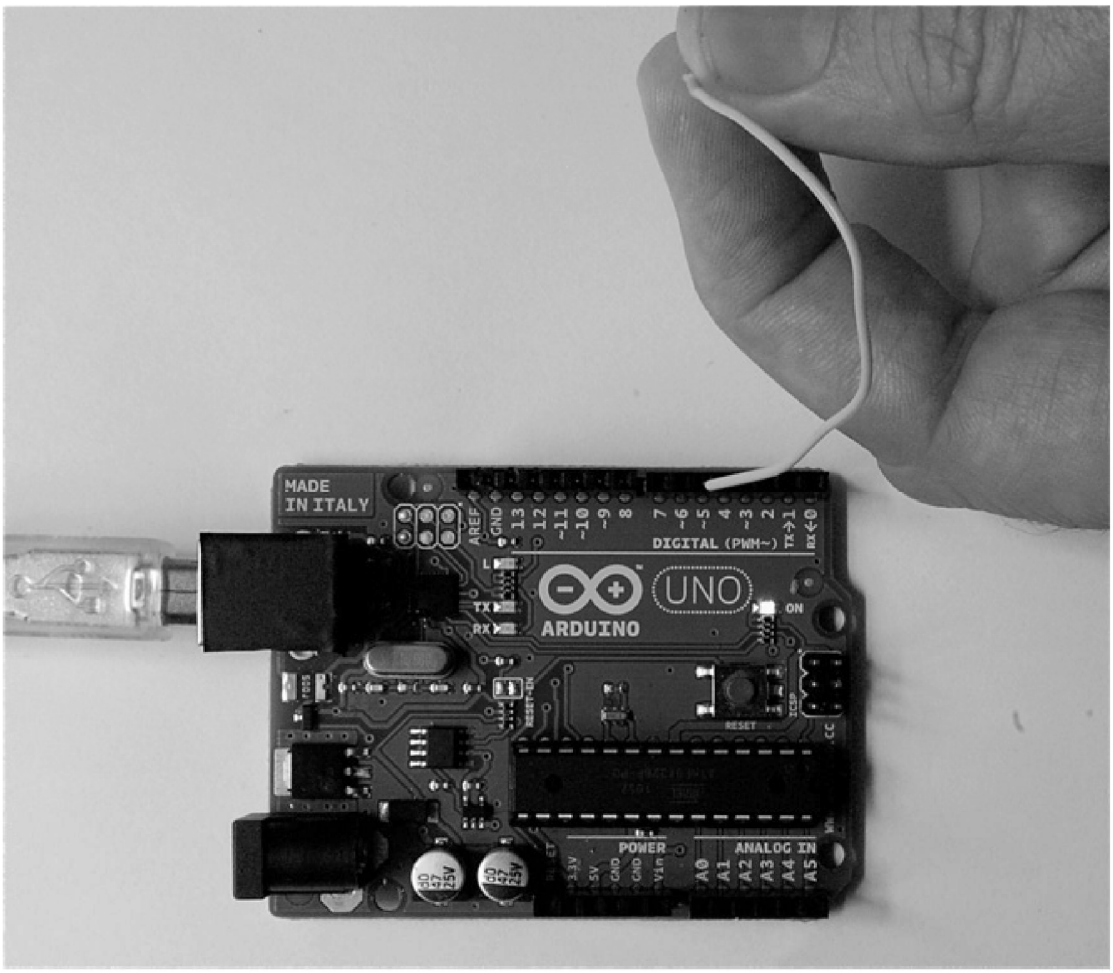
Ściskaj i popuszczaj palcami koniec przewodu oraz obserwuj zera i jedynki pojawiające się na monitorze portu szeregowego. Dzieje się tak, ponieważ wejścia Arduino są bardzo czułe. Twoje ciało działa jak antena odbierająca interferencje elektromagnetyczne.

Podłącz końcówkę przewodu, którą przed chwilą trzymałeś w dłoni, do złącza +5 V. Operację tę pokazano na rysunku 6.5. Monitor portu szeregowego powinien teraz wyświetlać ciąg jedynek.

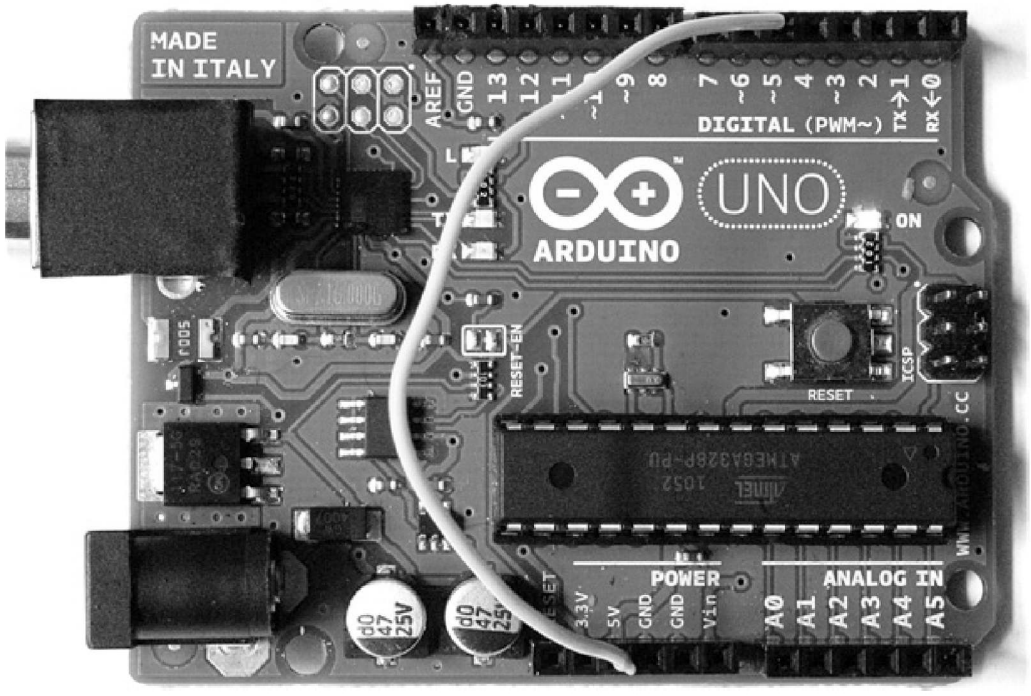
Teraz odłącz końcówkę kabla od złącza +5 V i podłącz ją do jednego ze złączy oznaczonych GND. Tak jak się zapewne spodziewasz, monitor portu szeregowego będzie wyświetlał ciąg zer.

Złącza wejściowe są często stosowane do podłączania przełączników. Na rysunku 6.6 przedstawiono przykładowy sposób podłączenia przełącznika.

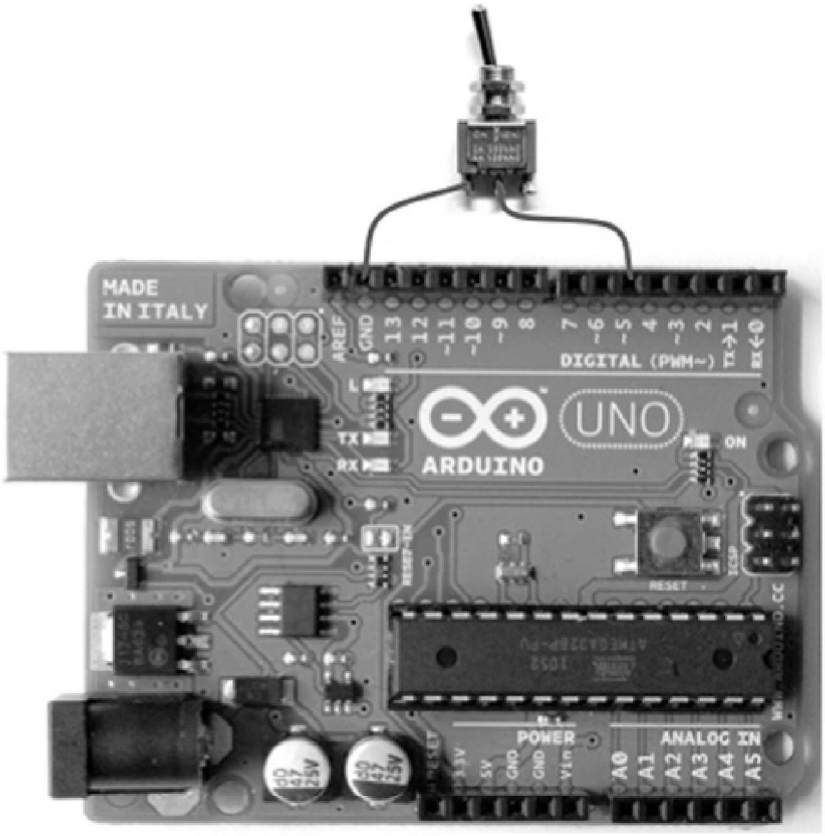
Takie połączenie może być problematyczne — gdy obwód przełącznika będzie otwarty, złącze wejściowe nie będzie podłączone do niczego. Można je określić mianem pływającego. W takim stanie złącze może generować fałszywe odczyty. Złącze musi działać w sposób bardziej przewidywalny. Możesz to osiągnąć przy użyciu tak zwanego rezystora podwyższającego.



Rysunek 6.4. *Ciało człowieka odgrywające rolę „anteny" podłączonej do złącza cyfrowego*

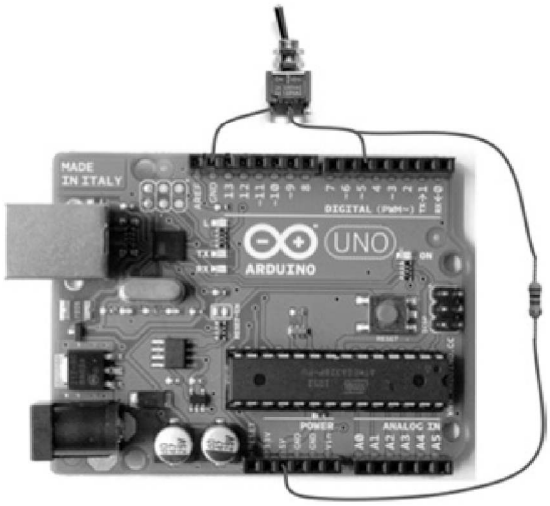


Rysunek 6.5. *Złącze numer 5 połączone ze złączem +5 V*



Rysunek 6.6. *Podłączanie przełącznika do płytki Arduino*

Na rysunku 6.7 przedstawiono standardowy sposób użycia rezystora podwyższającego. Takie połączenie powoduje, że po otwarciu obwodu przełącznika płynie przez rezystor prąd o napięciu 5 V. Gdy zamkniesz obwód przełącznika, na złącze zostanie podane napięcie o potencjale 0 V. Skutkiem ubocznym takiego połączenia jest to, że gdy obwód przełącznika będzie zamknięty, dojdzie do przepływu prądu o napięciu 5 V przez rezystor. A więc należy dobrać rezystor o wartości na tyle małej, aby był on odporny na interferencje elektromagnetyczne, ale jednocześnie tak dużej, aby po zamknięciu obwodu przełącznika nie płynął przez układ zbyt duży prąd.



Rysunek 6.7. *Przełącznik wraz z rezystorem podwyższającym*

Wewnętrzny rezystor podwyższający

Na szczęście płytka Arduino posiada rezystory podwyższające wbudowane w złącza cyfrowe. Istnieje możliwość programowej konfiguracji tych rezystorów. Domyślnie są one wyłączone. W celu uruchomienia rezystora podwyższającego na złączu o numerze 5 wystarczy dodać do szkicu *06.02.* poniższą linię kodu:

```
digitalWrite(inputPin, HIGH);
```

Kod ten należy wpisać w funkcji `setup` bezpośrednio po zdefiniowaniu tego, że złącze ma pracować w charakterze wejścia. Stosowanie funkcji `digitalWrite` w odniesieniu do wejścia może wydawać się nieco dziwne, aczkolwiek tak właśnie wygląda uruchomienie wbudowanego rezystora.

Modyfikacja ta została wprowadzona do szkicu *06.03.* Załaduj go na swoje Arduino i sprawdź działanie szkicu, ponownie ściskając przewód palcami. Tym razem monitor portu szeregowego powinien wyświetlać nieprzerwany ciąg jedynek.

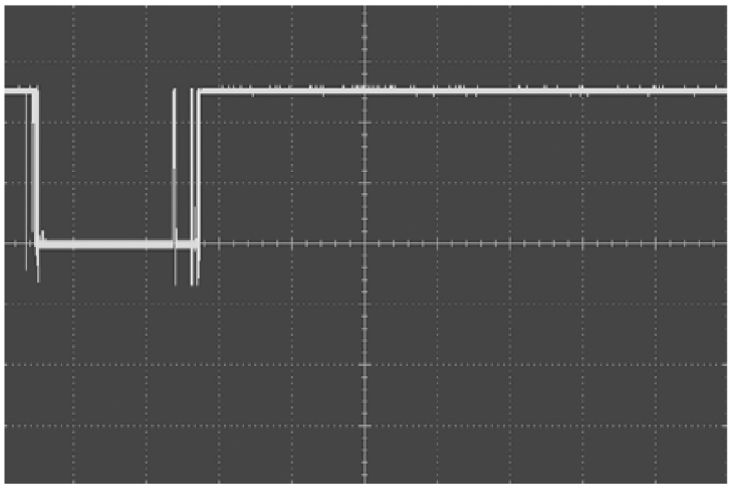
```
//szkic 06.03.

int inputPin = 5;

void setup()
{
   pinMode(inputPin, INPUT);
   digitalWrite(inputPin, HIGH);
   Serial.begin(9600);
}

void loop()
{
   int reading = digitalRead(inputPin);
   Serial.println(reading);
   delay(1000);
}
```

Usuwanie stuków

Zapewne spodziewasz się, że wciśnięcie przycisku spowoduje pojedynczą zmianę sygnału odbieranego przez wejście (z rezystorem podwyższającym) z 1 na 0. Na rysunku 6.8 pokazano, co może się dziać po wciśnięciu przycisku. Metalowe styki znajdujące się wewnątrz przycisku poruszają się. Wciśnięcie przycisku powoduje wywołanie czegoś, co może być zinterpretowane jako seria przyciśnięć, która ulega po chwili stabilizacji.

Wszystko to trwa bardzo krótko. Czas przedstawionego na oscyloskopie wciśnięcia przycisku to zaledwie około 200 milisekund. Przedstawiony wykres ilustruje działanie dość starego przełącznika marnej jakości. Nowy przełącznik dotykowy lub przycisk typu „klik" może w ogóle nie generować tego typu stuków.

Czasami omawiane zjawisko nie wpływa w żaden sposób na działanie programu. Na przykład szkic *06.04.* po wciśnięciu przycisku powoduje zapalenie diody LED. Tak naprawdę nigdy nie będziesz stosować Arduino w tym celu, aczkolwiek posłużmy się tym przykładem dla rozważań czysto teoretycznych.



Rysunek 6.8. *Ślad wciśnięcia przycisku widoczny na ekranie oscyloskopu*

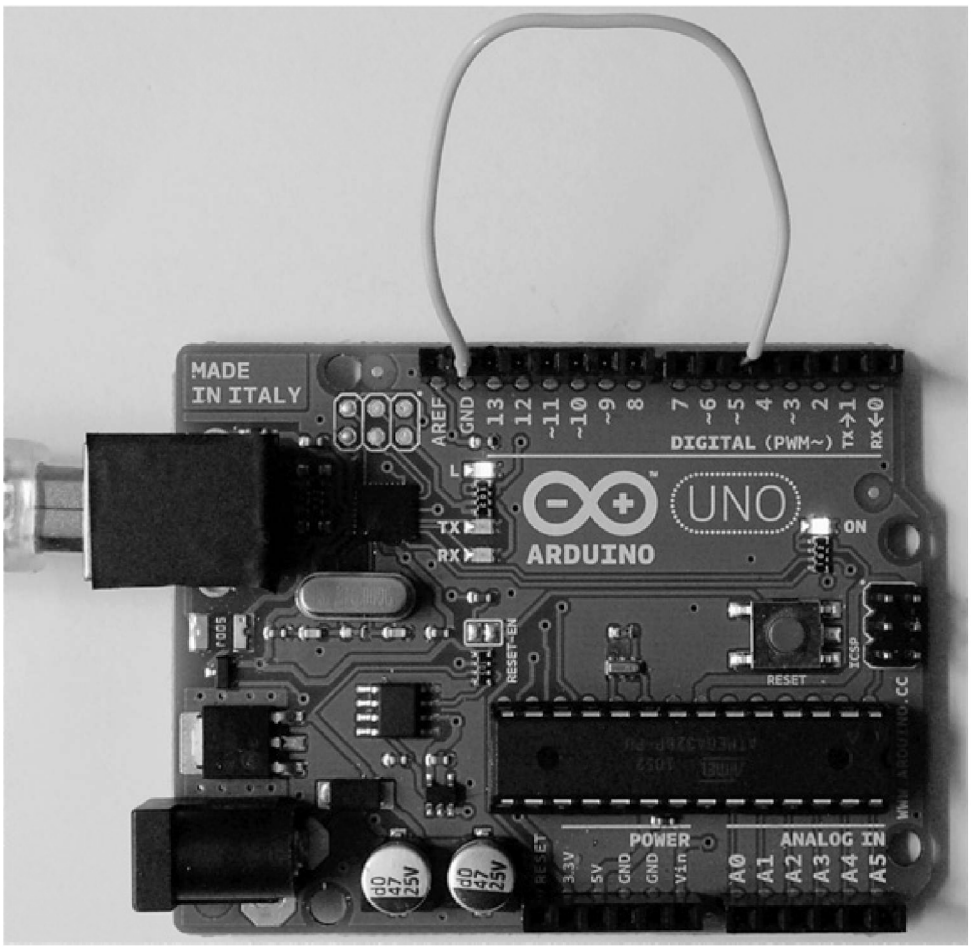
```
//szkic 06.04.

int inputPin = 5;
int ledPin = 13;

void setup()
{
   pinMode(ledPin, OUTPUT);
   pinMode(inputPin, INPUT);
   digitalWrite(inputPin, HIGH);
}

void loop()
{
   int switchOpen = digitalRead(inputPin);
   digitalWrite(ledPin, ! switchOpen);
}
```

Przeanalizujmy szkic *06.04*. Funkcja `loop` odczytuje dane z wejścia cyfrowego i przypisuje je do zmiennej `switchOpen`. Zmienna ta będzie przyjmowała wartość 0, gdy przycisk jest wciśnięty, lub 1, gdy przycisk nie jest wciśnięty. Pamiętaj, że stosujemy rezystor podwyższający, a więc gdy przycisk nie będzie wciśnięty, z wejścia będzie odczytywana wartość 1.

Tworząc funkcję `digitalWrite` służącą do włączania i wyłączania diody LED, pamiętaj o tym, że musisz odwrócić odczytaną wartość. Możesz to zrobić przy użyciu operatora `!` lub `not`.

Jeżeli załadujesz ten szkic i połączysz przewodem złącza D5 i GND, tak jak to pokazano na rysunku 6.9, dioda LED powinna świecić. Może tu dochodzić do tak zwanych stuków, ale prawdopodobnie będą one zbyt szybkie i nie dostrzeżesz ich wpływu na świecenie diody LED. Stuki nie wpływają na działanie tego programu.

Stuki miałyby wpływ na program, w którym stosowalibyśmy przełącznik do włączania i wyłączania diody LED. To znaczy, że po wciśnięciu przycisku dioda byłaby włączana na stałe. Dioda byłaby gaszona dopiero po kolejnym wciśnięciu przycisku. Gdyby Twój przycisk generował tak zwane stuki, o tym, czy dioda pozostanie włączona, czy wyłączona, decydowałby przypadek — to, czy doszłoby do parzystej, czy nieparzystej liczby stuków.



Rysunek 6.9. *Stosowanie przewodu w roli przełącznika*

Szkic *06.05.* po prostu uruchamia diodę LED w sposób opisany powyżej. W programie nie zastosowano żadnych technik mających na celu redukcję stuków. Wypróbuj działanie szkicu, stosując w roli przełącznika przewód łączący złącza D5 oraz GND:

```
//szkic 06.05.

int inputPin = 5;
int ledPin = 13;
int ledValue = LOW;

void setup()
{
   pinMode(inputPin, INPUT);
   digitalWrite(inputPin, HIGH);
   pinMode(ledPin, OUTPUT);
}

void loop()
{
   if (digitalRead(inputPin) == LOW)
   {
      ledValue = ! ledValue;
      digitalWrite(ledPin, ledValue);
   }
}
```

Zapewne zauważysz, że możliwe jest włączenie lub wyłączenie diody, jednakże nie udaje się to za każdym razem. Tak, to właśnie skutek uboczny tak zwanych stuków!

Prostym sposobem na rozwiązanie tego problemu jest dodanie opóźnienia — czasu, przez który program nie będzie odczytywał stanu wejścia — po wykryciu pierwszego wciśnięcia przycisku. Technikę tę zastosowano w szkicu *06.06.*:

```
//szkic 06.06.
int inputPin = 5;
int ledPin = 13;
int ledValue = LOW;

void setup()
{
   pinMode(inputPin, INPUT);
   digitalWrite(inputPin, HIGH);
   pinMode(ledPin, OUTPUT);
}

void loop()
{
   if (digitalRead(inputPin) == LOW)
   {
      ledValue = ! ledValue;
      digitalWrite(ledPin, ledValue);
      delay(500);
   }
}
```

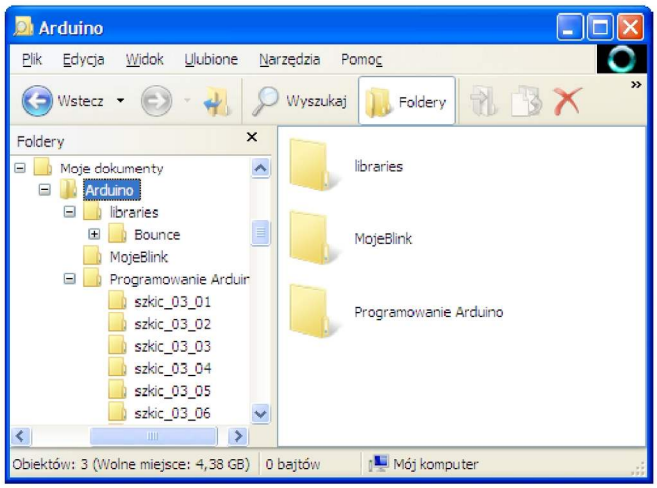
Po uruchomieniu funkcji `delay` żadne operacje nie będą przeprowadzane przez 500 milisekund, a przez ten czas stuki z pewnością ustaną. Dzięki takiemu zabiegowi możemy polegać na działaniu przełącznika. Ciekawym skutkiem ubocznym takiego zabiegu jest to, że przytrzymując przycisk, sprawisz, że dioda LED będzie błyskać.

Jeżeli szkic nie wykonuje żadnych innych operacji, zastosowanie funkcji `delay` nie jest problemem. Gdyby jednakże funkcja `loop` wykonywała więcej operacji, taki zabieg byłby problematyczny. Na przykład program nie mógłby w ciągu tych 500 milisekund wykryć wciśnięcia jakiegoś innego przycisku.

Zaprezentowana technika nie sprawdzi się w każdej sytuacji; czasem będziesz musiał zastosować bardziej skomplikowany zabieg. Możesz stworzyć samodzielnie zaawansowany kod redukujący stuki. Jest to co prawda zadanie dość skomplikowane, ale istnieją gotowe rozwiązania tego problemu.

Aby skorzystać z takiego gotowego rozwiązania, wystarczy do swojej aplikacji dodać odpowiednią bibliotekę. Bibliotekę taką można pobrać w formie archiwum ZIP ze strony internetowej książki: *http://www.helion.pl/ksiazki/ardupo.htm*

Po pobraniu archiwum należy rozpakować. Rozpakowany folder o nazwie *Bounce* należy przenieść do podfolderu *libraries* znajdującego się w folderze, w którym zapisywane są wszystkie szkice. W systemie Windows będzie to folder *Moje dokumenty\Arduino*, a w przypadku systemów Mac OS X lub Linux będzie to folder *Dokumenty/Arduino*. Jeżeli w folderze *Arduino* nie posiadasz folderu *libraries*, to będziesz musiał go utworzyć samodzielnie. Na rysunku 6.10 pokazano strukturę folderów po dodaniu biblioteki w systemie Windows.



Rysunek 6.10. *Dodawanie biblioteki Bounce w systemie Windows*

Po dodaniu biblioteki musisz uruchomić ponownie aplikację Arduino. Gdy to zrobisz, będziesz mógł stosować bibliotekę Bounce w swoich szkicach.

Szkic *06.07.* ilustruje zastosowanie biblioteki Bounce. Załaduj ten szkic do swojego Arduino i zobacz, jak pewne stało się włączanie i wyłączanie diody LED.

```
//szkic 06.07.

#include <Bounce.h>

int inputPin = 5;
int ledPin = 13;

int ledValue = LOW;
Bounce bouncer = Bounce(inputPin, 5);

void setup()
{
   pinMode(inputPin, INPUT);
   digitalWrite(inputPin, HIGH);
   pinMode(ledPin, OUTPUT);
}

void loop()
{
   if (bouncer.update() && bouncer.read() == LOW)
   {
      ledValue = ! ledValue;
      digitalWrite(ledPin, ledValue);
   }
}
```

Bibliotekę stosuje się w sposób dość prosty. Pierwsze, na co powinieneś zwrócić uwagę, to poniższa linia kodu:

```
#include <Bounce.h>
```

Powyższy fragment kodu jest niezbędny, aby poinformować kompilator o tym, że będziesz korzystać z biblioteki Bounce.

Do szkicu dodano również poniższy fragment kodu:

```
Bounce bouncer = Bounce(inputPin, 5);
```

Nie przejmuj się niestandardową składnią tego kodu. Tak naprawdę zastosowano tu składnię języka C++, a nie języka C. O języku C++ będziemy mówić dopiero w rozdziale 11. Na razie wystarczy, żebyś miał świadomość tego, że zaprezentowana linia kodu określa obiekt `bouncer` dla danego złącza, o okresie tłumienia stuków o długości 5 milisekund.

Od teraz aby odczytać stan wejścia (zamiast bezpośrednio odczytywać stan złącza), będziesz stosować obiekt o nazwie `bouncer`. Obiekt ten stanowi pewne opakowanie złącza wejściowego. Określanie tego, czy przycisk został wciśnięty, jest zadaniem tej linii kodu:

```
if (bouncer.update() && bouncer.read() == LOW)
```

Funkcja `update` zwraca prawdę, jeżeli stan obiektu `bouncer` uległ zmianie. Druga część warunku sprawdza, czy przycisk został wciśnięty (osiągnął stan `LOW`).

Wyjścia analogowe

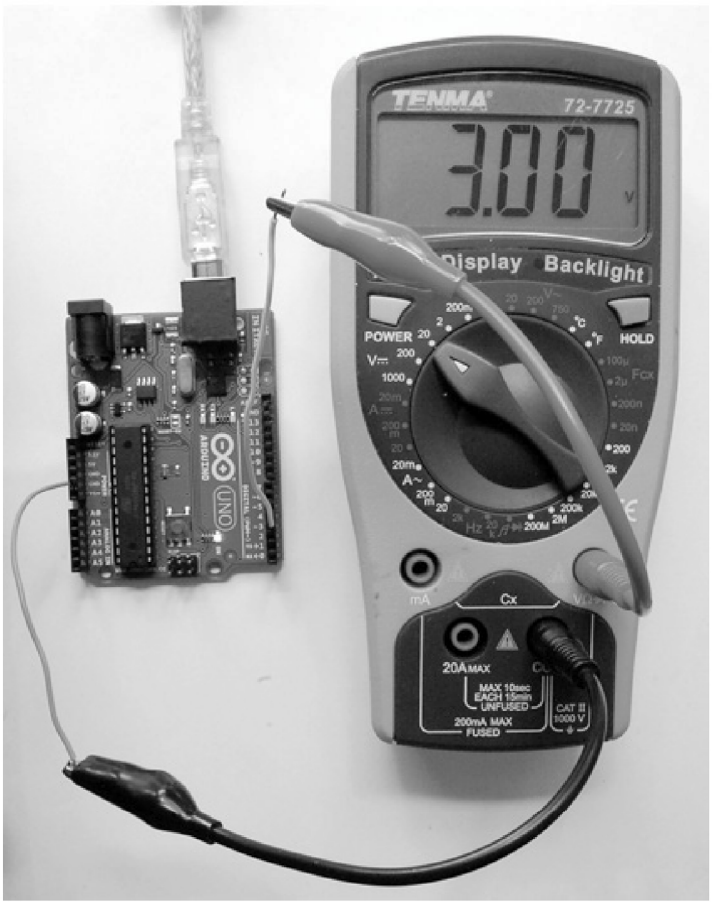
Część złączy cyfrowych — złącza cyfrowe o numerach 3, 5, 6, 9, 10 i 11 — może generować sygnały wyjściowe inne niż 5 V i 0 V. Złącza te są oznaczone ~ lub „PWM". PWM to skrót od angielskiego *Pulse Width Modulation* („modulacja czasu trwania impulsu") — określenie to odnosi się do sposobu sterowania ilością mocy na wyjściu. Efekt ten jest osiągany poprzez bardzo szybkie naprzemienne włączanie i wyłączanie wyjścia.

Impulsy są dostarczane zawsze z tą samą częstotliwością (około 500 na sekundę), jednakże mogą mieć różną długość. Gdybyś zastosował układ PWM do sterowania jasnością diody LED, to gdyby impulsy były długie, sterowana dioda byłaby ciągle włączona. Gdyby impulsy były krótkie, dioda świeciłaby tylko przez krótkie okresy czasu. Dzieje się to zbyt szybko, żeby człowiek mógł zauważyć takie migotanie diody. Obserwator ma wrażenie, że dioda świeci po prostu jaśniej lub ciemniej.

Zanim sprawdzimy te wyjścia przy użyciu diody LED, przetestujmy ich działanie za pomocą multimetru. Podłącz multimetr tak, aby mierzył napięcie pomiędzy złączami GND i D3 (patrz rysunek 6.11).

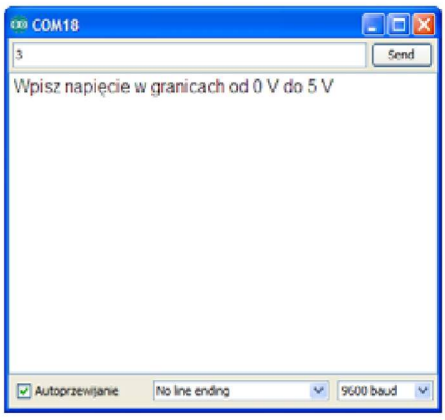
Teraz załaduj szkic *06.08.* na swoją płytę Arduino, a następnie otwórz monitor portu szeregowego (patrz rysunek 6.12). Wprowadź cyfrę 3 i wciśnij klawisz *Enter*. Twój multimetr powinien wskazać napięcie o wartości około 3 V. Możesz wprowadzić również inną cyfrę z zakresu od 0 do 5.

```
//szkic 06.08.

int outputPin = 3;

void setup()
{
   pinMode(outputPin, OUTPUT);
   Serial.begin(9600);
```



Rysunek 6.11. *Pomiar napięcia na wyjściu analogowym*

```
   Serial.println("Wpisz napięcie w granicach od 0 V do 5 V");
}

void loop()
{
   if (Serial.available() > 0)
   {
      char ch = Serial.read();
      int volts = (ch - '0') * 51;
      analogWrite(outputPin, volts);
   }
}
```



Rysunek 6.12. *Regulacja napięcia na wyjściu analogowym*

Program ustala wartość generowaną na wyjściu przez układ PWM w skali od 0 do 255. Wartość ta jest wyliczana poprzez mnożenie wybranej wartości napięcia (od 0 do 5) przez 51. Jeżeli chcesz dowiedzieć się więcej na temat układu PWM, zajrzyj do Wikipedii.

Wartość wyjścia jest ustawiana przy użyciu funkcji `analogWrite`. Funkcja ta przyjmuje jako argument wartość w przedziale od 0 do 255, gdzie 0 oznacza wyłączenie wyjścia, a 255 oznacza pełną moc. Jest to bardzo dobry sposób kontrolowania jasności diody LED. Gdybyś próbował kontrolować jasność diody przy użyciu napięcia zasilającego, zobaczyłbyś, że do około 2 V nic się nie dzieje, a po przekroczeniu tej granicy jasność światła diody LED rośnie bardzo szybko. Kontrolując jasność diody przy użyciu układu PWM (zmieniając średni czas, przez jaki dioda jest włączona) osiągniesz bardziej liniową kontrolę nad diodą.

Wejścia analogowe

Wejścia cyfrowe dostarczają tylko informacji na temat tego, czy element podłączony do złącza Arduino jest włączony, czy nie. Wejścia analogowe generują wartości w zakresie od 0 do 1023, w zależności od napięcia przyłożonego do danego złącza.

Program odczytuje stan wejścia analogowego przy użyciu funkcji `analogRead`. Szkic *06.09.* wyświetla co pół sekundy, przy użyciu monitora portu szeregowego, wartość napięcia na złączu analogowym A0. Otwórz monitor portu szeregowego i obserwuj pojawiające się w nim dane.

```
//szkic 06.09.

int analogPin = 0;

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

void loop()
{
   int reading = analogRead(analogPin);
   float voltage = reading / 204.6;
   Serial.print("Odczyt=");
   Serial.print(reading);
   Serial.print("\t\tV=");
   Serial.println(voltage);
   delay(500);
}
```

Po uruchomieniu szkicu zobaczysz, że odczytywane wartości zmieniają się. Jest to skutek tego, że wejście jest „pływające". Podobna sytuacja miała miejsce w przypadku wejścia cyfrowego.

Połącz złącza A0 i GND przy użyciu przewodu. Teraz odczytywana powinna być tylko wartość 0. Odłącz końcówkę przewodu wpiętą do złącza GND i podłącz ją do złącza 5 V. Arduino powinno odczytywać wartość około 1023, która to jest wartością maksymalną

możliwą do odczytania. Gdybyś połączył przewodem złącza A0 i 3,3 V, Arduino powinno poinformować Cię, że właśnie dokonujesz pomiaru prądu o napięciu 3,3 V.

Podsumowanie

Na tym kończą się rozważania dotyczące podstaw odbierania oraz generowania sygnałów przez Arduino. W kolejnym rozdziale przyjrzymy się pewnym funkcjom, jakie oferuje standardowa biblioteka Arduino.

Rozdział 7.

Standardowa biblioteka Arduino

Biblioteka jest miejscem, gdzie kryje się wiele przydatnych rzeczy. Dotąd korzystałeś tylko z podstawowych możliwości języka C. Do pracy nad szkicami przyda Ci się wiele innych funkcji.

Dotychczas używałeś funkcji `pinMode`, `digitalWrite` i `analogWrite`. Funkcji, z których możesz korzystać, jest jednakże o wiele więcej. Funkcje mogą być stosowane do przeprowadzania operacji matematycznych, generowania losowych liczb, przeprowadzania operacji na bitach, wykrywania pulsacji na złączu wejściowym i obsługi tak zwanych przerwań.

Język, w którym programujemy Arduino, jest oparty na starszej bibliotece o nazwie Wiring. Ponadto jest on pewnym dopełnieniem innej biblioteki — biblioteki o nazwie Processing. Biblioteka Processing jest bardzo podobna do biblioteki Wiring, jednakże jest ona oparta w większym stopniu na języku Java niż na języku C. Biblioteka ta jest stosowana przez komputer do obsługi Arduino za pośrednictwem interfejsu USB. Aplikacje Arduino uruchamiane na komputerze korzystają z biblioteki Processing. Gdybyś chciał stworzyć jakiś ciekawy interfejs pozwalający na komunikację komputera z Arduino, więcej informacji dotyczących tej biblioteki znajdziesz pod adresem *http://www.processing.org/*.

Liczby losowe

Pomimo naszych doświadczeń wynikających z pracy z komputerem jest on tak naprawdę bardzo przewidywalnym urządzeniem. Czasami możesz jednakże potrzebować, żeby Arduino wygenerowało coś nieprzewidywalnego. Na przykład może zaistnieć potrzeba, aby skonstruowany przez Ciebie robot poruszał się w sposób „losowy": przez losowo określony czas poruszał się w jakimś kierunku, potem obrócił o losowo wybraną liczbę stopni, a następnie znów rozpoczął ruch w losowo obranym kierunku. Możesz również chcieć zastosować Arduino w roli swoistej kostki do gry, która będzie generowała losowo liczby z zakresu od 1 do 6.

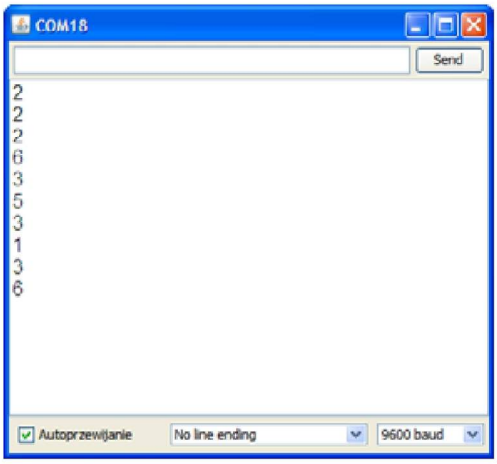
Na wykonanie takich operacji pozwala standardowa biblioteka Arduino. Funkcja `random` zwraca wartość typu `int`. Funkcja ta może przyjąć jeden lub dwa argumenty. Jeżeli do funkcji `random` przekażemy jeden argument, funkcja ta zwróci liczbę z zakresu od zera do liczby o jeden mniejszej od przekazanego argumentu.

Po przekazaniu dwóch argumentów funkcja `random` zwraca liczbę z zakresu od wartości pierwszego argumentu do wartości drugiego argumentu pomniejszonej o jeden. Na przykład wywołanie funkcji `random(1, 10)` zwróci losową liczbę z zakresu od jednego do dziewięciu.

Szkic *07.01.* wyświetla za pośrednictwem monitora portu szeregowego liczby z zakresu od jednego do sześciu.

```
//szkic 07.01.

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

void loop()
{
   int number = random(1, 7);
   Serial.println(number);
   delay(500);
}
```

Jeżeli załadujesz ten szkic na Arduino i otworzysz monitor portu szeregowego, to zobaczysz coś podobnego do treści zaprezentowanej na rysunku 7.1.



Rysunek 7.1. *Liczby losowe*

Po kilkukrotnym uruchomieniu tego szkicu może Cię zdziwić to, że za każdym razem generowany jest ten sam ciąg liczb„losowych".

Tak naprawdę liczby te nie są losowane. Takie liczby noszą nazwę **liczb pseudolosowych**. Rozkład tych liczb jest losowy. Jeżeli uruchomisz ten program i zgromadzisz milion liczb, uzyskasz podobną ilość jedynek, dwójek, trójek i tak dalej. Liczby te nie są losowe w sensie ich nieprzewidywalności. Nieprzewidywalność nie leży w naturze pracy mikrokontrolera. Bez pomocy z zewnątrz mikrokontroler nie jest w stanie wygenerować liczb, które będą zupełnie przypadkowe.

Aby sekwencja generowanych liczb była mniej przewidywalna, możesz zastosować technikę zwaną **osadzaniem** generatora liczb losowych. Sprowadzi się to do rozpoczęcia sekwencji. Nie można w tym celu korzystać wyłącznie z funkcji `random`. Często korzysta się z wartości odczytywanych z „pływającego" złącza wejściowego (bez rezystora podwyższającego). A więc do osadzenia generatora liczb losowych możesz zastosować wartość odczytaną ze złącza analogowego.

Funkcja `randomSeed` wykonuje tę operację. Szkic *07.02.* przedstawia sposób generowania bardziej przypadkowych wartości.

```
//szkic 07.02.

void setup()
{
   Serial.begin(9600);
   randomSeed(analogRead(0));
}

void loop()
{
   int number = random(1, 7);
   Serial.println(number);
   delay(500);
}
```

Wciśnij kilkukrotnie przycisk *Reset*. Wyświetlana sekwencja liczb losowych powinna być za każdym razem inna.

Generator liczb losowych tego typu nie mógłby zostać użyty w żadnej loterii. W celu wygenerowania jeszcze bardziej przypadkowych wartości musiałbyś korzystać ze sprzętowego generatora liczb losowych. Generator taki korzysta z dużo bardziej losowych zdarzeń, takich jak np. zmiany widma promieniowania kosmicznego.

Funkcje matematyczne

Czasami będziesz wymagać od Arduino przeprowadzenia wielu bardziej złożonych operacji matematycznych. W takim wypadku będziesz mógł skorzystać z dużej biblioteki dostępnych funkcji matematycznych. Najbardziej przydatne funkcje omówiono w tabeli 7.1.

Manipulacja bitami

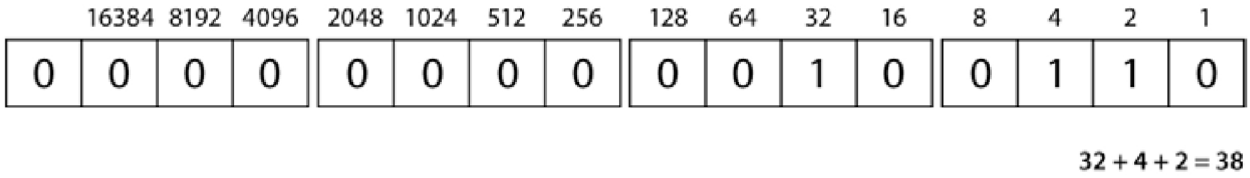
Bit jest jednocyfrowym nośnikiem informacji binarnej. Bit może przybierać wartość 0 lub 1. Słowo **bit** w języku angielskim jest skróconą formą wyrażenia *binary digit*, co po polsku oznacza **liczbę w systemie dwójkowym**. Najczęściej stosowane są zmienne typu `int`, które składają się z 16 bitów. Jeżeli chcesz zapisać tylko wartość określającą prawdę lub fałsz (1 lub 0), to zapisanie takiej wartości jako zmiennej typu `int` skutkuje marnotrawieniem pamięci. Tak naprawdę nie jest to duży problem, o ile nie brakuje Ci pamięci. Lepiej jest stworzyć

Tabela 7.1. *Funkcje matematyczne*

Funkcja	Opis	Przykład
`abs`	Zwraca wartość bezwzględną argumentu.	`abs(12)` zwróci 12 `abs(-12)` zwróci 12
`constrain`	Funkcja ogranicza liczbę tak, aby nie przekraczała ona określonego przedziału. Pierwszy argument jest liczbą, która będzie poddana operacji ograniczania, drugi argument jest początkiem zakresu dopuszczalnych wartości, a trzeci argument jest końcem tego zakresu.	`constrain(8, 1, 10)` zwróci 8 `constrain(11, 1, 10)` zwróci 10 `constrain(0, 1, 10)` zwróci 1
`map`	Funkcja dokonuje mapowania liczby z jednego zakresu na liczbę jej odpowiadającą, ale znajdującą się w innym zakresie. Pierwszym argumentem jest liczba, którą chcemy poddać operacji mapowania. Drugi i trzeci argument określają zakres źródłowy, a ostatnie dwa argumenty określają zakres docelowy. Funkcja ta przydaje się do mapowania wartości odczytywanych z wejść analogowych.	`map(x, 0, 1023, 0, 5000)`
`max`	Zwraca większy z przekazanych argumentów.	`max(10, 11)` zwróci 11
`min`	Zwraca mniejszy z przekazanych argumentów.	`min(10, 11)` zwróci 10
`pow`	Zwraca pierwszy argument podniesiony do potęgi określonej przez drugi argument.	`pow(2, 5)` zwróci 32
`sqrt`	Zwraca pierwiastek argumentu.	`sqrt(16)` zwróci 4
`sin, cos, tan`	Funkcje te wykonują przekształcenia trygonometryczne. Są one rzadko stosowane.	
`log`	Funkcja ta może być zastosowana np. do obliczenia temperatury wskazywanej przez termistor logarytmiczny.	

przejrzysty kod, który marnuje trochę pamięci, niż tworzyć bardzo skomplikowany kod programu. Czasem upakowanie danych na mniejszym obszarze pamięci może jednakże okazać się przydatne.

Każdy bit pamięci, w której zapisano zmienną `int`, można rozumieć jako element posiadający wartość dziesiętną. Wartość zmiennej typu `int` w systemie dziesiętnym możesz obliczyć, sumując wartości wszystkich bitów, które są jedynkami. Przypatrz się rysunkowi 7.2. Wartość zaprezentowanej tam zmiennej wynosi 38. Obsługa liczb ujemnych jest jeszcze bardziej skomplikowana. W przypadku liczb ujemnych skrajny bit znajdujący się z lewej strony przybiera wartość 1.



Rysunek 7.2. *Przykładowa wartość typu int*

Analizowanie poszczególnych bitów przy użyciu wartości dziesiętnych nie sprawdza się w praktyce. Bardzo trudno byłoby wyobrazić sobie zapis binarny liczby 123. Z tego powodu programiści często stosują system **szesnastkowy**, zwany również **heksadecymalnym**. W systemie dziesiętnym stosujemy cyfry od 0 do 9. W systemie szesnastkowym mamy do dyspozycji dodatkowe cyfry symbolizowane literami od A do F. Każda cyfra w systemie szesnastkowym reprezentuje cztery bity. Poniższa tabela pokazuje zależności pomiędzy liczbami w systemie dziesiętnym i szesnastkowym a ich zapisem binarnym.

Tabela 7.2. *Liczby od 0 do 15 w systemie dziesiętnym i szesnastkowym*

Liczba w systemie dziesiętnym	Liczba w systemie szesnastkowym	Zapis binarny (czterocyfrowy)
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

W systemie szesnastkowym wartość zmiennej typu `int` może być zapisana przy użyciu czterech cyfr. A więc liczba 10001100 w systemie dwójkowym jest równoważna liczbie 8C w systemie szesnastkowym. Aby stosować w języku C liczbę w systemie szesnastkowym, należy korzystać ze specjalnej składni. Wartość zapisaną w systemie szesnastkowym możesz przypisać do zmiennej typu `int`:

```
int x = 0x8C;
```

Standardowa biblioteka Arduino pozwala na manipulowanie pojedynczo szesnastoma bitami składającymi się na wartość typu `int`. Funkcja `bitRead` zwraca wartość każdego bitu wartości typu `int`. W poniższym przykładzie następuje przypisanie wartości 0 do zmiennej o nazwie `bit`:

```
int x = 0x8C; // 10001100
bit = bitRead(x, 0);
```

Początkowa pozycja bitu to 0, a końcowa to 15. Funkcja rozpoczyna działanie od najmniej znaczącego bitu. Skrajny bit znajdujący się z prawej strony jest bitem 0. Następny bit jest bitem 1 i tak dalej.

Bliźniaczą funkcją `bitRead` jest funkcja `bitWrite`. Funkcja ta przyjmuje trzy argumenty. Pierwszy argument jest liczbą, na której będą przeprowadzane operacje, drugi argument określa pozycję bitu, a trzeci określa jego wartość. Poniższy przykład zmienia wartość typu `int` z 2 na 3 (w zapisie dziesiętnym lub szesnastkowym):

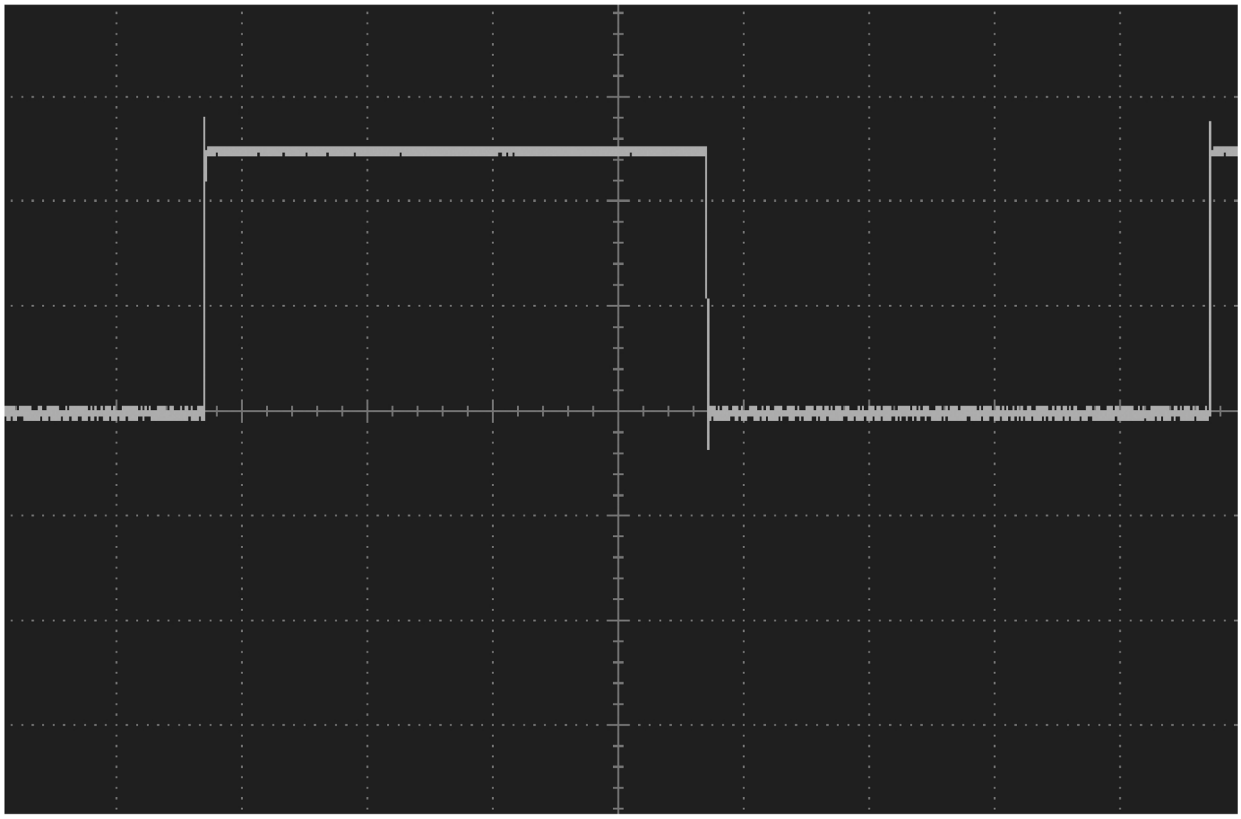
```
int x = 2; // 0010
bitWrite(x, 0, 1);
```

Zaawansowane funkcje wejścia i wyjścia

Istnieją pewne niewielkie funkcje, które mogą Ci ułatwić wykonywanie zadań związanych z obsługą wejść i wyjść.

Generowanie tonów

Funkcja `tone` pozwala na generowanie sygnału o fali kwadratowej (patrz rysunek 7.3) na jednym ze złączy cyfrowych. Najczęściej funkcja ta jest stosowana do generowania słyszalnego dźwięku przy użyciu głośnika lub brzęczyka.



Rysunek 7.3. *Sygnał o fali kwadratowej*

Funkcja ta przyjmuje dwa lub trzy argumenty. Pierwszy argument określa numer złącza, na którym chcemy wygenerować ton. Drugim argumentem jest częstotliwość tonu wyrażona w hercach (Hz). A opcjonalnym, trzecim argumentem jest czas trwania tonu. Jeżeli nie określimy tego czasu, to dźwięk będzie generowany w nieskończoność, tak jak ma to miejsce w przypadku szkicu *07.03*. Dlatego też wywołanie funkcji `tone` umieściliśmy w funkcji `setup`, a nie w funkcji `loop`.

```
//szkic 07.03.

void setup()
{
  tone(4, 500);
}

void loop() {}
```

Aby przerwać sygnał dźwiękowy, możesz zastosować funkcję `noTone`. Funkcja ta przyjmuje tylko jeden argument — numer złącza, na którym chcemy zakończyć generowanie dźwięku.

Wprowadzanie rejestru przesuwnego

Czasami liczba złączy dostępna na płytce Arduino może okazać się za mała. Podczas sterowania dużą liczbą diod LED często stosowaną techniką jest użycie układu rejestru przesuwnego. Układ ten odczytuje dane bit po bicie. Gdy odczyta wystarczającą ilość danych, przenosi te dane na zestaw wyjść (jeden bit do jednego wyjścia).

Pomocna w implementacji tej techniki może okazać się funkcja `shiftOut`. Funkcja ta przyjmuje cztery argumenty:

- Numer złącza, na którym pojawi się wysłany bit.
- Numer złącza, które będzie pracować w charakterze zegara. Złącze to będzie sygnalizować każdorazowe wysłanie bitu.
- Znacznik określający to, czy bity zostaną wysłane, zaczynając od najmniej znaczących bitów, czy zaczynając od najbardziej znaczących bitów. Powinna to określać stała `MSBFIRST` lub `LSBFIRST`.
- Bajt danych do wysłania.

Przerwania

Jedną z rzeczy, która irytuje programistów tworzących rozbudowane programy, jest to, że Arduino może wykonywać jednocześnie tylko jedną operację. Jeżeli chcesz napisać program, który wykonuje równolegle kilka wątków, to niestety muszę Cię zmartwić. Jest to zadanie niemożliwe. Pomimo tego powstało kilka projektów, w których obsługiwanych jest kilka wątków. Arduino zostało jednakże stworzone do celów, które nie wymagają wielowątkowych operacji. Wątki wykonywane przez Arduino mogą być przełączane za pomocą przerwań.

Dwa złącza Arduino (D2 i D3) mogą odbierać sygnał przerwania. Jeżeli na złączu pojawi się określony sygnał przerwania, to procesor Arduino zawiesza aktualnie wykonywaną operację i zaczyna wykonywać funkcję określoną przez przerwanie.

Szkic *07.10.* steruje błyskaniem diody LED. Okres błysku ulegnie zmianie po otrzymaniu sygnału przerwania. W celu symulacji sygnału przerwania możesz połączyć złącza D2 i GND, stosując wewnętrzny rezystor podwyższający w celu utrzymania wysokiego sygnału przez większość czasu.

```
//szkic 07.04.

int interruptPin = 2;
int ledPin = 13;
int period = 500;

void setup()
{
   pinMode(ledPin, OUTPUT);
   pinMode(interruptPin, INPUT);
   digitalWrite(interruptPin, HIGH); //podwyższ
   attachInterrupt(0, goFast, FALLING);
}

void loop()
{
   digitalWrite(ledPin, HIGH);
   delay(period);
   digitalWrite(ledPin, LOW);
   delay(period);
}

void goFast()
{
   period = 100;
}
```

Poniżej znajduje się najważniejszy fragment kodu umieszczony w funkcji `setup` powyższego szkicu.

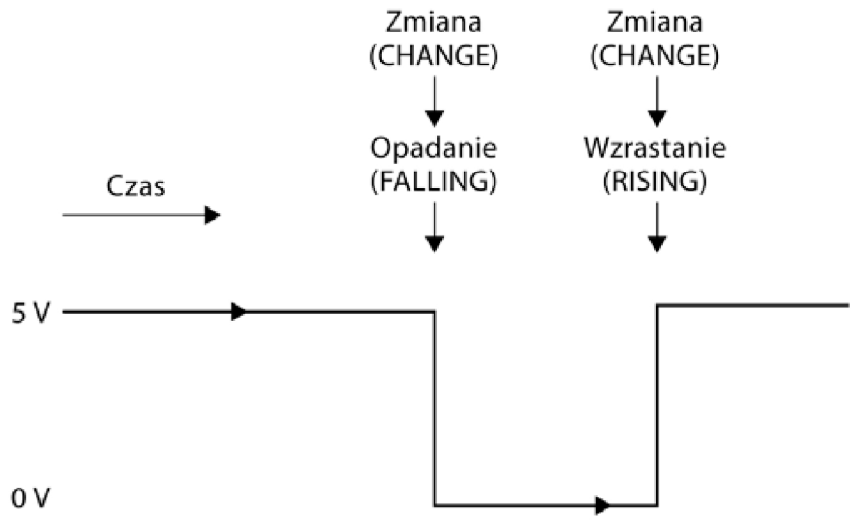
```
attachInterrupt(0, goFast, FALLING);
```

Pierwszy argument określa to, które z dwóch przerwań chcemy stosować. Może być to mylące, ale `0` oznacza użycie złącza o numerze 2, a `1` oznacza użycie złącza o numerze 3.

Kolejny argument jest nazwą funkcji, która ma zostać wywołana w wyniku przerwania. Ostatni argument jest jedną z następujących stałych: `CHANGE`, `RISING` i `FALLING`. Działanie tych argumentów ilustruje rysunek 7.4.

Jeżeli trybem przerwania jest `CHANGE`, to przerwanie zostanie wywołane w wyniku wzrostu sygnału z 0 do 1 (`RISING`), a także w wyniku opadnięcia sygnału z 1 do 0 (`FALLING`).

Funkcja `noInterrupts` wyłącza przerwania. Funkcja ta zatrzymuje wykonywanie przerwań sygnalizowanych na obu złączach. Obsługę przerwań możesz włączyć ponownie, stosując funkcję `interrupts`.



Rysunek 7.4. *Rodzaje sygnałów przerwania*

Podsumowanie

W niniejszym rozdziale omówiliśmy niektóre przydatne funkcje oferowane przez standardową bibliotekę Arduino. Omówione funkcje oszczędzą Ci wiele wysiłku. Jedną z rzeczy, które lubią dobrzy programiści, jest możliwość zastosowania efektów dobrze wykonanej przez innych pracy.

W kolejnym rozdziale rozszerzymy wiadomości na temat struktur danych, które zdobyłeś podczas lektury rozdziału 5. Omówimy także sposoby zapisywania danych, których nie chcemy utracić w wyniku odłączenia Arduino od zasilania.

Rozdział 8.

Zapisywanie danych

Arduino pamięta wartości przypisane do zmiennych tylko do momentu odłączenia płytki od źródła zasilania. Po wyłączeniu zasilania lub wciśnięciu przycisku *Reset* następuje utrata danych.

W poniższym rozdziale przyjrzymy się sposobom na zachowanie takich danych.

Stałe

Jeżeli dane, które chcesz zachować, nie ulegają zmianie, możesz po prostu wprowadzać te dane przy każdym uruchomieniu Arduino. Przykładem to ilustrującym jest tablica liter umieszczona w programie tłumaczącym wprowadzany tekst na alfabet Morse'a (rozdział 5. — szkic *05.05*.).

W tamtym szkicu stosowałeś poniższy kod w celu zdefiniowania odpowiednich zmiennych i wypełnienia ich niezbędnymi danymi:

```
char* letters[] = {
   ".-", "-...", "-.-.", "-..", ".",
      "..-.", "--.", "....", "..", //A-I
   ".---", "-.-", ".-..", "--", "-.",
      "---", ".--.", "--.-", ".-.", //J-R
   "...", "-", "..-", "...-", ".--",
      "-..-", "-.--", "--.." //S-Z
};
```

Może pamiętasz, że dokonaliśmy wtedy obliczeń i zdecydowaliśmy, że mamy do dyspozycji jeszcze dość dużo dwukilobajtowej pamięci. Gdyby jednakże brakowało Ci pamięci, lepiej byłoby zapisać te dane w pamięci flash o pojemności 32 KB. Pamięć ta jest częściej stosowana do przechowywania programów od pamięci RAM o pojemności 2 KB. Dyrektywa `PROGMEM` służy do zapisywania danych w pamięci flash. `PROGMEM` znajduje się w jednej z bibliotek dołączonych do środowiska Arduino. Stosowanie tej dyrektywy może wydawać się dość nieintuicyjne.

Dyrektywa PROGMEM

W celu zapisania danych w pamięci flash należy do szkicu dołączyć bibliotekę PROGMEM:

```
#include <avr/pgmspace.h>
```

Powyższa linia kodu informuje kompilator o tym, że w szkicu będziesz korzystać z biblioteki `pgmspace`. Biblioteka w tym akurat przypadku jest zestawem funkcji, które ktoś już stworzył, a które możesz stosować w swoich szkicach. Nie musisz rozumieć wszystkich szczegółów dotyczących działania tych funkcji.

Korzystanie z biblioteki pozwala na stosowanie słowa kluczowego `PROGMEM`, a także funkcji `pgm_read_word`. Z obu tych elementów będziesz korzystać w kolejnych szkicach.

Biblioteka `pgmspace` jest oficjalnie obsługiwana przez środowisko, w którym pracujesz. Stanowi ona jeden ze składników oprogramowania Arduino. Istnieje wiele oficjalnych bibliotek, a ponadto w internecie możesz znaleźć wiele bibliotek nieoficjalnych — stworzonych przez osoby takie jak na przykład Ty i ja. Biblioteki nieoficjalne muszą zostać zainstalowane w środowisku Arduino. Więcej na temat tych bibliotek, a także na temat tworzenia własnych bibliotek, dowiesz się podczas lektury rozdziału 11.

Korzystając z dyrektywy `PROGMEM`, musisz pamiętać o tym, że dyrektywa ta obsługuje tylko wybrane typy danych. Niestety tablice znaków nie są obsługiwane przez tę dyrektywę. Musisz zdefiniować zmienną (łańcuch typu `PROGMEM`) dla każdego łańcucha, który chcesz zapisać w pamięci flash. Następnie powinieneś umieścić te łańcuchy w tablicy typu `PROGMEM`. Operacja ta będzie wyglądała następująco:

```
PROGMEM prog_char sA[] = ".-";
PROGMEM prog_char sB[] = "-...";
// i tak dalej dla reszty liter
PROGMEM const char* letters[] =
{
   sA, sB, sC, sD, sE, sF, sG, sH, sI, sJ, sK, sL, sM,
   sN, sO, sP, sQ, sR, sS, sT, sU, sV, sW, sX, sY, sZ};
}
```

Nie zamieszczam tutaj szkicu *08.01.*, ponieważ jest on dosyć długi. Możesz go jednakże załadować i przekonać się, że działa dokładnie tak samo jak wcześniejszy szkic, którego działanie było oparte na pamięci RAM.

Zapis danych w pamięci flash przebiega w specyficzny sposób. Ich odczyt jest równie skomplikowany. Kod odczytujący łańcuch zawierający litery w alfabecie Morse'a musi zostać zmodyfikowany w następujący sposób:

```
strcpy_P(buffer, (char*)pgm_read_word(&(letters[ch - 'a'])));
```

Powyższy zapis stosuje zmienną `buffer`, do której kopiowany jest łańcuch `PROGMEM`. Dzięki tej operacji możliwe jest korzystanie z tych danych jak ze zwyczajnej tablicy elementów typu `char`. Zmienna ta musi być zadeklarowana jako zmienna globalna:

```
char buffer[6];
```

Omówioną technikę można stosować tylko wtedy, gdy dane są stałymi — nie zamierzasz ich modyfikować podczas pracy programu. W kolejnym podrozdziale dowiesz się, jak korzystać z pamięci EEPROM, która powinna służyć do zapisywania danych mogących podlegać modyfikacji.

EEPROM

Układ ATMega328 będący sercem Arduino posiada kilobajt pamięci EEPROM (pamięci tylko do odczytu kasowanej elektrycznie). Pamięć ta jest zaprojektowana tak, aby zapisane w niej dane można było odczytać po wielu latach. Tak naprawdę pomimo swojej nazwy pamięć ta nie służy tylko do odczytu. Można w niej również zapisywać dane.

Polecenia stosowane podczas zapisu danych w pamięci EEPROM i ich odczytu są równie nieintuicyjne co komendy używane do obsługi dyrektywy `PROGMEM`. Możliwy jest jednoczesny odczyt lub zapis tylko jednego bajta pamięci EEPROM.

Przykładowy szkic *08.02.* pozwala na wprowadzenie jednocyfrowego kodu za pośrednictwem monitora portu szeregowego. Szkic zapamiętuje tę cyfrę, a następnie wyświetla ją cyklicznie za pośrednictwem monitora portu szeregowego.

```
// szkic 08.02.
#include <EEPROM.h>

int addr = 0;
char ch;

void setup()
{
   Serial.begin(9600);
   ch = EEPROM.read(addr);
}

void loop()
{
   if (Serial.available() > 0)
   {
      ch = Serial.read();
      EEPROM.write(0, ch);
      Serial.println(ch);
   }
   Serial.println(ch);
   delay(1000);
}
```

Aby sprawdzić działanie tego szkicu, należy otworzyć monitor portu szeregowego, wprowadzić nowy znak, a następnie odłączyć Arduino. Po ponownym podłączeniu płytki do komputera i ponownym otwarciu monitora portu szeregowego zobaczysz, że wprowadzony znak został zapamiętany.

Funkcja `EEPROM.write` przyjmuje dwa argumenty. Pierwszy argumentem jest adres pamięci EEPROM. Wartość ta powinna się znajdować w przedziale od 0 do 1023. Drugim

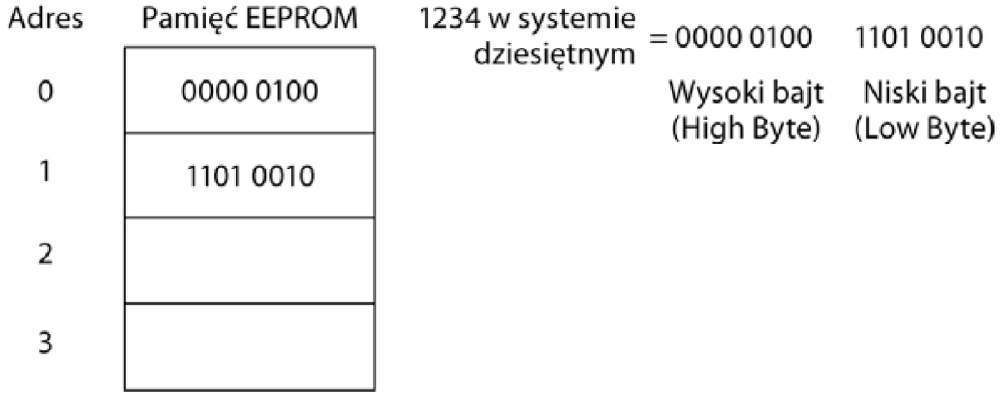
argumentem są dane, które chcemy zapisać w określonej lokacji. Dane muszą być pojedynczym bajtem. Znak jest reprezentowany za pomocą ośmiu bitów, a więc jest to wystarczająca przestrzeń, jednakże nie możesz bezpośrednio zapisać w niej szesnastobitowej zmiennej typu `int`.

Przechowywanie wartości zmiennej typu int w pamięci EEPROM

W celu zapisania dwubajtowej wartości typu `int` w pamięci EEPROM pod adresem 0 i 1 musisz wykonać następujące operacje:

```
int x = 1234;
EEPROM.write(0, highByte(x));
EEPROM.write(1, lowByte(x));
```

Funkcje `highByte` i `lowByte` przydają się do podzielenia wartości typu `int` na dwa bajty. Na rysunku 8.1 pokazano sposób przechowywania takiej liczby w pamięci EEPROM.



Rysunek 8.1. *Zapisywanie szesnastobitowej wartości typu integer w pamięci EEPROM*

Aby wczytać wartość typu `int` z pamięci EEPROM, musisz odczytać oba bajty pamięci, a następnie dokonać rekonstrukcji liczby w następujący sposób:

```
byte high = EEPROM.read(0);
byte low = EEPROM.read(1);
int x = (high << 8) + low;
```

Operator << jest operatorem przesuwającym bity. Przemieszcza on osiem bitów wyższego rzędu na właściwe pozycje liczby typu `int`, a następnie dodaje bajt niższego rzędu.

Przechowywanie wartości typu float w pamięci EEPROM (unie)

Zapisanie liczby typu float w pamięci EEPROM wymaga jeszcze bardziej złożonych operacji. Przydadzą się do tego tak zwane **unie**. Unia jest strukturą danych, która umożliwia zapisanie więcej niż jednej zmiennej w tym samym obszarze pamięci. Takie zmienne mogą być różnych typów, muszą jedynie posiadać ten sam rozmiar (zajmować tyle samo bajtów).

Poniższa definicja unii pozwala zmiennym typu `float` i `int` odnosić się do tych samych bajtów pamięci:

```
union data
{
   float f;
   int i;
} convert;
```

Liczbę typu `float` możesz umieścić wewnątrz unii w następujący sposób:

```
float f = 1.23;
convert.f = f;
```

Liczbę typu `int` możesz podzielić na dwa bajty (w celu zapisania w pamięci EEPROM) w następujący sposób:

```
EEPROM.write(0, highByte(convert.i));
EEPROM.write(1, lowByte(convert.i));
```

Odczytanie takiej liczby z pamięci wymaga wykonania operacji odwrotnej. Najpierw musisz połączyć dwa bajty w jeden obiekt typu `int`, a następnie umieścić ten element w unii w celu odczytania go jako liczby typu `float`.

```
byte high = EEPROM.read(0);
byte low = EEPROM.read(1);
convert.i = (high << 8) + low;
float f = convert.f;
```

Przechowywanie łańcucha w pamięci EEPROM

Zapisywanie łańcuchów w pamięci EEPROM i ich odczytywanie przebiega w dosyć prosty sposób. Musisz wprowadzać kolejne znaki pojedynczo, tak jak w poniższym przykładzie:

```
char *test = "Witaj";
int i = 0;
while (test[i] != '\0')
{
   EEPROM.write(i, test[i]);
   i++;
}
EEPROM.write(i, "\0");
```

W celu wczytania łańcucha z powrotem z pamięci możesz przeprowadzić następującą operację:

```
char test[10];
int i = 0;
char ch;
ch = EEPROM.read(i);
while (ch != '\0' && i < 10)
{
   test[i] = ch;
   ch = EEPROM.read(i);
   i++;
}
```

Wymazywanie zawartości pamięci EEPROM

Pamiętaj, że danych zapisanych w pamięci EEPROM nie skasuje nawet załadowanie do Arduino nowego szkicu. Jest to problematyczne, ponieważ podczas Twojej pracy nad kolejnym projektem w pamięci wciąż mogą znajdować się dane dotyczące poprzednich projektów. Szkic *08.03.* wypełnia całą przestrzeń pamięci EEPROM zerami:

```
// szkic 08.03.
#include <EEPROM.h>

void setup()
{
   Serial.begin(9600);
   Serial.println("EEPROM: wymazywanie");
   for (int i = 0; i <= 1023; i++)
   {
      EEPROM.write(i, 0);
   }
   Serial.println("EEPROM wymazano");
}

void loop()
{
}
```

Pamiętaj o tym, że zapis do lokacji znajdującej się w pamięci EEPROM można wykonać zaledwie około 100 000 razy. Po przekroczeniu tej liczby zapisów pamięć może okazać się zawodna. A więc w pamięci EEPROM umieszczaj tylko te wartości, których naprawdę potrzebujesz. Wadą tej pamięci jest również to, że jest ona bardzo wolna. Zapis jednego bajta zajmuje około 3 milisekund.

Kompresja

Zapisując dane do pamięci EEPROM lub korzystając z dyrektywy `PROGMEM`, możesz napotkać problem braku miejsca w pamięci na zapisanie wszystkich danych. W takiej sytuacji warto znaleźć jak najbardziej wydajny sposób reprezentacji danych.

Kompresja zakresu

Istnieją wartości, które można zapisać jako liczbę typu `int` (16 bitów) lub `float` (32 bity). Na przykład w celu określenia temperatury wyrażonej w stopniach Celsjusza możesz stosować wartość typu `float` — np. 20,25. Gdybyś chciał zapisać tę wartość w pamięci EEPROM, dobrze byłoby, żebyś mógł ją umieścić w pojedynczym bajcie pamięci. W ten sposób zaoszczędziłbyś połowę miejsca zajmowanego standardowo przez liczbę typu `float`.

Taką operację możesz przeprowadzić, modyfikując dane przed ich zapisaniem. Pamiętaj o tym, że bajt pozwala na zapisanie wartości dodatniej z przedziału od 0 do 255. A więc jeżeli zależy Ci na pomiarze temperatury z dokładnością do najbliższej całkowitej wartości wyrażonej w stopniach Celsjusza, to możesz po prostu dokonać konwersji liczby typu

`float` na liczbę typu `int`. W ten sposób utracisz część wartości będącą ułamkiem dziesiętnym. Poniższy przykład pokazuje, jak to zrobić:

```
int tempInt = (int)tempFloat;
```

Zmienna `tempFloat` przechowuje liczbę zmiennoprzecinkową. Polecenie `(int)` jest tak zwanym **rzutowaniem**. Rzutowanie polega na konwersji zmiennej jednego typu na inny, kompatybilny typ. W tym przypadku następuje konwersja typu `float` wartości 20,25 na wartość typu `int`, która zostanie ucięta do liczby 20.

Jeżeli wiesz, że pomiar, którego dokonujesz, mieści się w przedziale od 0 do 60 stopni Celsjusza, to możesz zmierzoną wartość temperatury pomnożyć przez cztery (przed dokonaniem konwersji tej wartości i przed zapisaniem jej w pamięci). Odczytując dane z pamięci EEPROM, zapisaną liczbę musiałbyś w takim przypadku podzielić przez cztery. Otrzymasz wartość, która będzie określała wartość mierzonej temperatury z dokładnością do 0,25 stopnia Celsjusza.

Poniższy przykładowy szkic zapisuje w taki właśnie sposób wartość temperatury w pamięci EEPROM, a następnie odczytuje ją z pamięci i wyświetla za pomocą monitora portu szeregowego:

```
//szkic 08.04.

#include <EEPROM.h>

void setup()
{
   float tempFloat = 20.75;
   byte tempByte = (int)(tempFloat * 4);
   EEPROM.write(0, tempByte);

   byte tempByte2 = EEPROM.read(0);
   float temp2 = (float)(tempByte2) / 4;
   Serial.begin(9600);
   Serial.println("\n\n\n");
   Serial.println(temp2);
}

void loop(){}
```

Istnieją również inne sposoby kompresowania danych. Jeżeli dokonujesz pomiaru wartości, które zmieniają się wolno — dobrym przykładem takiego pomiaru jest wielokrotny pomiar temperatury — możesz zapisać tylko pierwszą wartość z pełną dokładnością, a następnie zapisywać tylko informacje dotyczące zmiany temperatury. Zmiany będą małe, a więc zajmą mniejszą ilość pamięci.

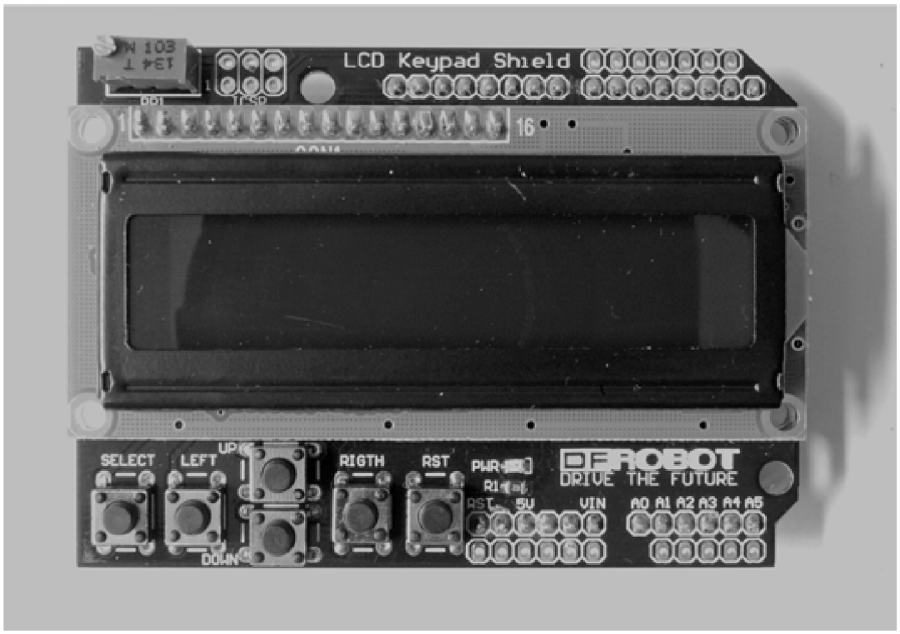
Podsumowanie

Posiadasz już pewne wiadomości na temat sposobów zapisywania danych tak, aby nie zostały one utracone po odłączeniu Arduino od zasilania. W kolejnym rozdziale omówimy zagadnienia związane z wyświetlaczami LCD.

Rozdział 9.

Wyświetlacze LCD

W tym rozdziale przeanalizujemy techniki pomocne podczas pisania programów służących do obsługi wyświetlaczy LCD. Na rysunku 9.1 pokazano wyświetlacz, który będę stosował.



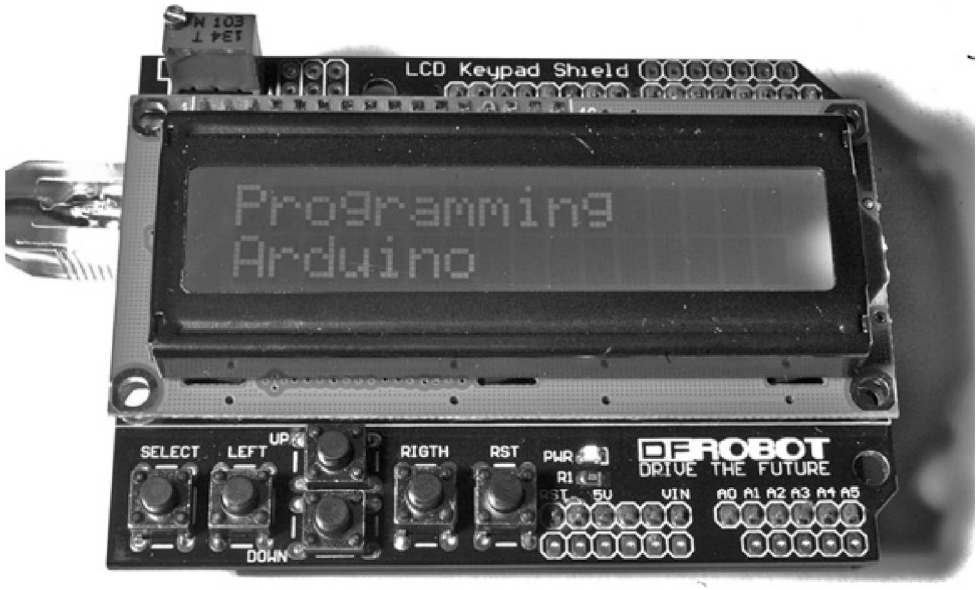
Rysunek 9.1. *Płytka wyświetlacza alfanumerycznego*

Jest to książka dotycząca oprogramowania, a nie sprzętu. W tym rozdziale muszę jednakże wyjaśnić podstawy działania elektroniki wyświetlacza LCD. Pozwoli Ci to na zrozumienie tego, jak należy nim sterować.

Używany przeze mnie moduł wyświetlacza LCD jest gotową płytką, którą można wpiąć bezpośrednio na wierzch płytki Arduino. Na płytce tego modułu umieszczono poza samym wyświetlaczem kilka przycisków. Istnieje wiele płytek z wyświetlaczami, jednakże prawie wszystkie zostały zbudowane w oparciu o ten sam kontroler (układ HD44780). Powinieneś szukać płytki z tym właśnie kontrolerem.

Korzystam z płytki o nazwie DFRobot LCD Keypad Shield. Moduł ten jest sprzedawany przez DFRobot *(http://www.robotshop.com/)*. Płytka ta jest tania, posiada wyświetlacz LCD wyświetlający po 16 znaków w każdym z dwóch rzędów, a ponadto wyposażona jest w sześć przycisków.

Płytka jest w pełni produkowana przez fabrykę, a więc nie musisz niczego lutować. Płytkę wyświetlacza wystarczy założyć na wierzch płytki Arduino (patrz rysunek 9.2).



Rysunek 9.2. *Płytka wyświetlacza założona na płytkę Arduino*

Płytka do obsługi wyświetlacza korzysta z siedmiu złączy oraz z jednego złącza analogowego do obsługi przycisków. A więc nie możemy zastosować tych ośmiu złączy znajdujących się na płytce Arduino w żadnym innym celu.

Tablica wyświetlająca komunikaty za pośrednictwem interfejsu USB

W celu zaprezentowania przykładu ilustrującego proste zastosowanie wyświetlacza zamienimy Arduino w tablicę wyświetlającą komunikaty za pośrednictwem interfejsu USB. Wyświetlane będą komunikaty wprowadzone przy użyciu monitora portu szeregowego.

Środowisko programistyczne Arduino jest wyposażone w bibliotekę obsługującą wyświetlacze LCD. Biblioteka ta ułatwia korzystanie z wyświetlaczy. Zapewnia ona obsługę wywołań następujących funkcji:

- `clear` — usuwa dowolny tekst wyświetlony na ekranie.
- `setCursor` — ustawia kursor w rzędzie i kolumnie, w której pojawi się następny wprowadzony przez Ciebie tekst.
- `print` — wyświetla łańcuch we wskazanej pozycji.

Poniżej znajduje się szkic *09.01.*:

```
// szkic 09.01. tablica wyświetlająca komunikaty za pośrednictwem interfejsu USB

#include <LiquidCrystal.h>

// lcd(RS, E, D4, D5, D6, D7)
LiquidCrystal lcd(8, 9, 4, 5, 6, 7);
int numRows = 2;
int numCols = 16;


void setup()
{
   Serial.begin(9600);
   lcd.begin(numRows, numCols);
   lcd.clear();
   lcd.setCursor(0,0);
   lcd.print("Arduino");
   lcd.setCursor(0,1);
   lcd.print("jest super");
}

void loop()
{
   if (Serial.available() > 0)
   {
      char ch = Serial.read();
      if (ch == '#')
      {
         lcd.clear();
      }
      else if (ch == '/')
      {
         // nowy wiersz
         lcd.setCursor(0, 1);
      }
      else
      {
         lcd.write(ch);
      }
   }
}
```

Korzystając z bibliotek, musisz je zadeklarować na początku szkicu — dzięki temu kompilator będzie wiedzieć, że używasz danej biblioteki.

Kolejna linia kodu określa to, które złącza są używane przez płytkę wyświetlacza. Określony również zostaje cel użycia tych złączy. Inne płytki wyświetlacza mogą komunikować się z Arduino za pośrednictwem innych złączy (pinów). Informacje na ten temat są podane w dokumentacji płytki.

W moim przypadku do sterowania wyświetlaczem używane są złącza D4, D5, D6, D7, D8 i D9. W tabeli 9.1 przedstawiono funkcje tych złączy.

Tabela 9.1. *Funkcje złączy płytki wyświetlacza LCD*

Parametry funkcji LCD()	Złącze (pin) Arduino	Funkcja
RS	8	Wybór rejestru. Parametr ten przybiera wartość 1 lub 0 w zależności od tego, czy Arduino wysyła dane zawierające znaki, czy dane zawierające instrukcje. Instrukcja może np. włączyć miganie kursora.
E	9	Złącze to przekazuje układowi sterującemu wyświetlaczem informację na temat gotowości do przesłania danych za pośrednictwem czterech poniższych złączy.
Data 4	4	Te cztery złącza służą do przesyłania danych. Mikrokontroler znajdujący się na płytce wyświetlacza może obsługiwać dane ośmiobitowe lub czterobitowe. Moja płytka obsługuje dane czterobitowe. W tym przypadku stosowany jest zakres bitów od 4 do 7 zamiast zakresu od 0 do 7.
Data 5	5	
Data 6	6	
Data 7	7	

Funkcja `setup` jest napisana przejrzyście. Uruchamiasz komunikację z monitorem portu szeregowego, aby ten mógł przesyłać polecenia. Następnie inicjalizujesz bibliotekę wyświetlacza LCD o określonych wymiarach. Potem wyświetlasz komunikat o treści „Arduino jest super". Komunikat ten jest wyświetlany w dwóch wierszach. Najpierw kursor jest umieszczany w lewym górnym rogu wyświetlacza, a następnie wyświetlany jest napis „Arduino". Później kursor jest przenoszony na początek drugiego wiersza i wyświetlony zostaje napis „jest super".

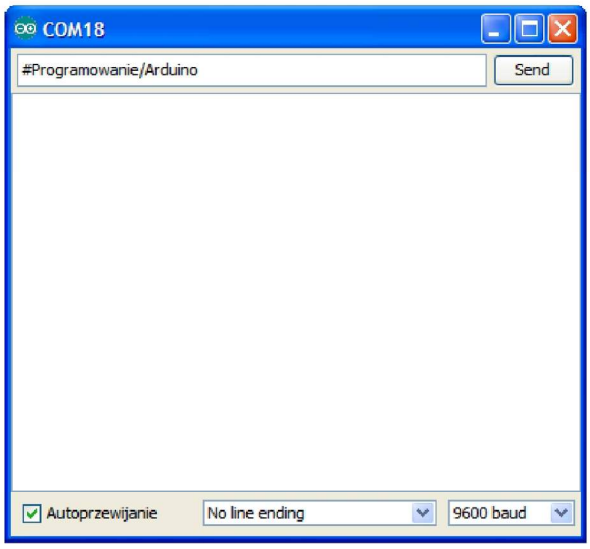
Większość rzeczy umieszczono w funkcji `loop`. Funkcja ta sprawdza, czy nie wysłano nowych znaków za pośrednictwem monitora portu szeregowego. Szkic przetwarza znaki pojedynczo.

Poza zwyczajnymi znakami, które zostaną po prostu wyświetlone na ekranie, możesz stosować również pewne znaki specjalne. Gdy wpiszesz #, szkic wyczyści całość ekranu. Znak / spowoduje przeniesienie kursora do nowego wiersza. Pozostałe znaki będą po prostu wyświetlane przy użyciu funkcji `write`, w miejscu, gdzie akurat będzie znajdował się kursor. Funkcja `write` działa podobnie do funkcji `print`, jednakże służy ona tylko do wyświetlania pojedynczych znaków. Nie można jej stosować do wyświetlania łańcuchów znaków.

Korzystanie z wyświetlacza

Załaduj szkic *09.01.* na swoją płytkę Arduino, a następnie załóż płytkę wyświetlacza. Pamiętaj o tym, że Arduino należy bezwzględnie odłączyć od zasilania przed zakładaniem płytki.

Otwórz monitor portu szeregowego i spróbuj wpisać tekst pokazany na rysunku 9.3.



Rysunek 9.3. *Wysyłanie poleceń do wyświetlacza*

Inne funkcje biblioteki wyświetlacza LCD

Poza funkcjami, z których korzystałeś w ostatnim szkicu, biblioteka wyświetlacza LCD obsługuje również wiele innych funkcji, takich jak:

- `home` — działa tak samo jak `setCursor(0,0)` — ustawia kursor w lewym górnym rogu wyświetlacza.
- `cursor` — wyświetla kursor.
- `noCursor` — powoduje niewyświetlanie kursora.
- `blink` — powoduje, że kursor miga.
- `noBlink` — powoduje, że kursor przestaje migać.
- `noDisplay` — wyłącza wyświetlacz, nie usuwając jego zawartości.
- `display` — włącza monitor z powrotem po uprzednim zastosowaniu funkcji `noDisplay`.
- `scrollDisplayLeft` — przesuwa całość tekstu wyświetlanego na ekranie o jedną pozycję w lewo.
- `scrollDisplayRight` — przesuwa całość tekstu wyświetlanego na ekranie o jedną pozycję w prawo.
- `autoscroll` — uruchamia tryb, w którym nowe znaki są dodawane w miejscu wyświetlania się kursora, a wcześniej wyświetlany tekst jest spychany w kierunku określonym przez funkcję `leftToRight` lub `rightToLeft`.
- `noAutoscroll` — wyłącza tryb `autoscroll`.

Podsumowanie

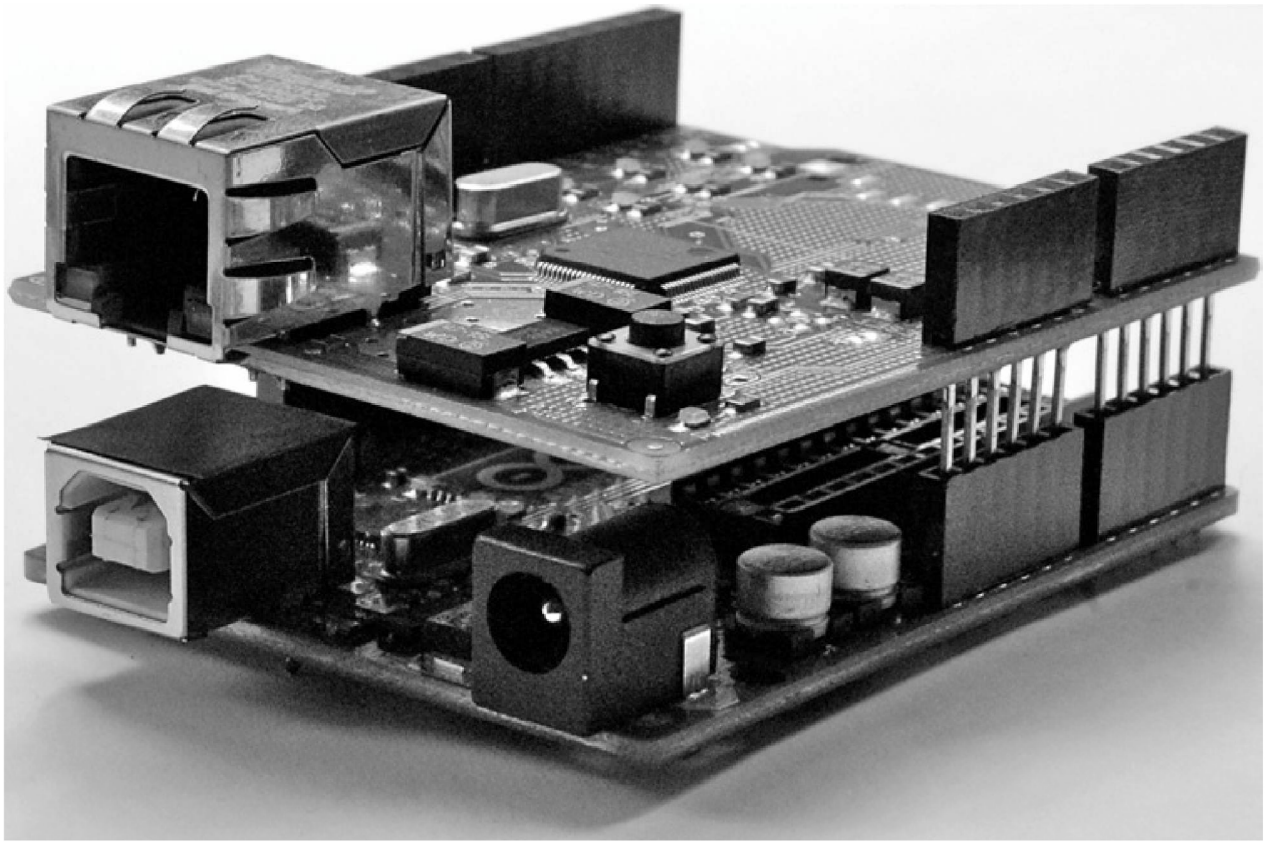
Widzisz? Programowanie dodatkowych płytek nie jest wcale trudne, zwłaszcza gdy istnieje biblioteka, która oszczędza Ci wiele pracy.

W kolejnym rozdziale omówimy zagadnienia związane z płytką pozwalającą Arduino łączyć się z siecią Ethernet.

Rozdział 10.

Programowanie aplikacji sieci Ethernet

W tym rozdziale omówimy zagadnienia związane z płytką zawierającą interfejs Ethernet. Pozwala ona Arduino pracować w domowej sieci Ethernet (patrz rysunek 10.1).



Rysunek 10.1. *Arduino z płytką pozwalającą na pracę w sieci Ethernet*

Płytki pozwalające na pracę w sieci Ethernet

Planując zakup płytki pozwalającej Arduino na pracę w sieci Ethernet, musisz zachować pewną ostrożność. Powinieneś kupić „oficjalną" płytkę opartą o mikroukład Wiznet. Unikaj tańszych, ale o wiele trudniejszych w obsłudze płytek opartych o układ kontrolera Ethernet ENC28J60.

Płytki obsługujące sieć Ethernet charakteryzują się dużym poborem prądu, a więc będziesz potrzebować źródła zasilania o napięciu 9 V lub 12 V zdolnego dostarczyć prąd o natężeniu 1 A lub większym.

Komunikacja z serwerami sieciowymi

Zanim omówimy zagadnienia związane z komunikacją pomiędzy Arduino, przeglądarką internetową i serwerem sieci Web, musisz zrozumieć pewne zagadnienia związane z protokołem przesyłania hipertekstu (HTTP), a także językiem hipertekstowego znakowania informacji (HTML).

HTTP

Protokół przesyłania hipertekstu służy do komunikacji pomiędzy przeglądarką a serwerem sieci Web.

Gdy otwierasz stronę w przeglądarce, przeglądarka wysyła do serwera, na którym znajduje się dana strona, zapytanie o potrzebne treści. Taką treścią może być po prostu zawartość strony HTML. Serwer stale oczekuje na tego typu żądania i przetwarza otrzymane żądania. W naszym przypadku przetwarzanie żądania oznacza po prostu odesłanie treści HTML, którą zdefiniowałeś w szkicu Arduino.

HTML

Rola języka hipertekstowego znakowania informacji sprowadza się do dodania do zwykłego tekstu informacji dotyczących jego formatowania. Formatowanie ma sprawić, że przesłany tekst będzie wyglądać atrakcyjnie po wyświetleniu go przez przeglądarkę. Poniżej znajduje się przykładowy kod HTML. Kod ten wyświetlony w przeglądarce wygląda tak, jak to pokazano na rysunku 10.2.

```
<html>
<body>
<h1>Programowanie Arduino</h1>
<p>Informacje o programowaniu Arduino</p>
</body>
</html>
```

Rysunek 10.2. *Przykładowy dokument HTML*

Dokument HTML zawiera znaczniki. Znaczniki mają symboliczne oznaczenia początku i końca. Wewnątrz nich mogą znajdować się inne znaczniki. Znaczniki otwierające rozpoczynają się od symbolu <, następnie umieszczana jest nazwa znacznika, a na końcu zapisywany jest symbol > — przykład to `<html>`. Znacznik zamykający wygląda podobnie, jednakże po symbolu < znajduje się symbol /. W zaprezentowanym powyżej przykładzie najbardziej zewnętrznym znacznikiem jest `<html>`, w którym znajduje się znacznik o nazwie `<body>`. Wszystkie witryny sieciowe powinny zaczynać się od tych znaczników. Na końcu przykładowego kodu znajdują się odpowiadające im znaczniki zamykające. Zauważ, że znaczniki zamykające muszą zostać umieszczone we właściwej kolejności — znacznik `body` musi być zamknięty przed znacznikiem `html`.

Teraz przyjrzyjmy się bardziej interesującym, znajdującym się w środku znacznikom `h1` i `p`. Treści oznaczone tymi znacznikami są wyświetlane przez przeglądarkę.

Znacznik `h1` służy do oznaczania nagłówka pierwszego poziomu. Sprawia on, że tekst jest wyświetlany przy użyciu większej, pogrubionej czcionki. Znacznik `p` służy do oznaczania akapitów — tekst oznaczony tym znacznikiem będzie wyświetlany jako akapit.

Omówione rzeczy stanowią zaledwie wierzchołek góry lodowej. HTML jest bardzo szerokim zagadnieniem. W celu dokładniejszego zapoznania się z technikami HTML możesz zajrzeć do wielu książek, a także licznych zasobów sieci Web.

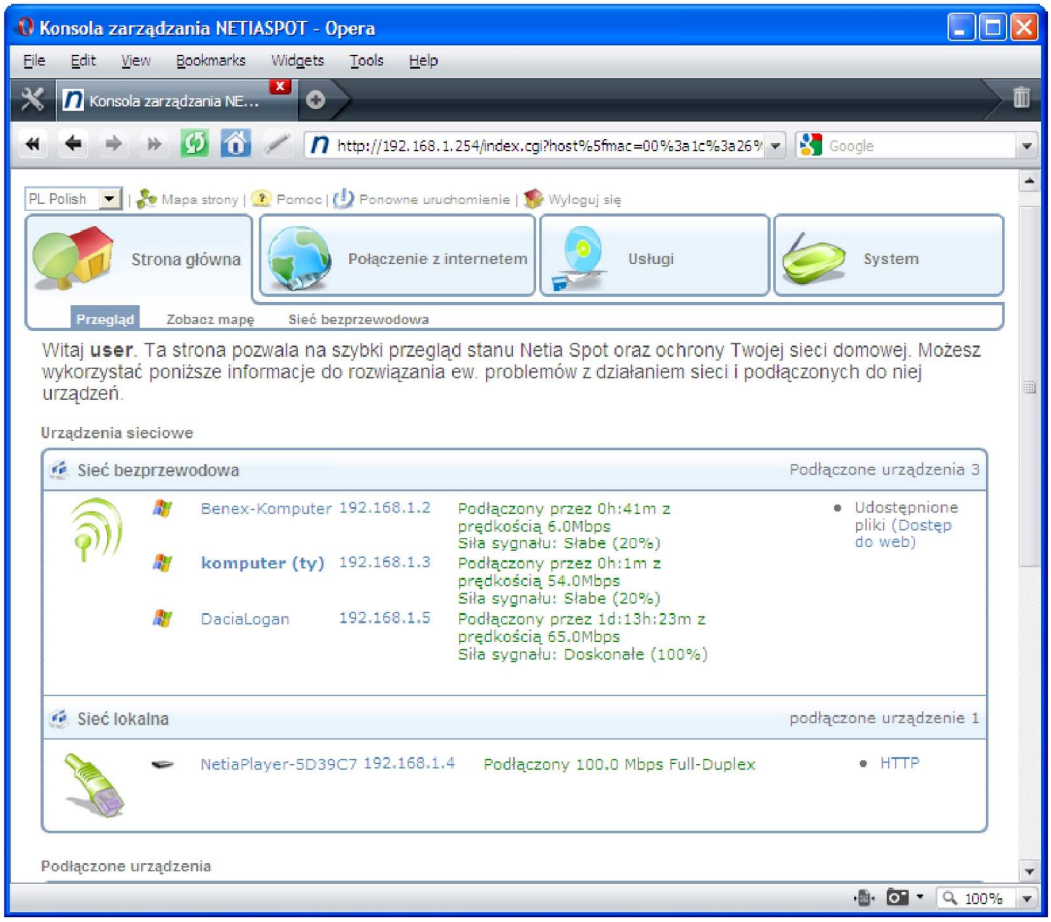
Arduino w roli serwera sieci Web

Pierwszy szkic zamieni Twoje Arduino wyposażone w płytkę obsługującą sieć Ethernet w mały serwer sieci Web. Z pewnością taki serwer nie będzie mógł pracować jako serwer wyszukiwarki Google, jednakże pozwoli on Arduino na obsługę żądań wysyłanych przez sieć Ethernet i wyświetlanie wyników operacji w przeglądarce internetowej zainstalowanej na Twoim komputerze.

Przed załadowaniem szkicu *10.01.* do pamięci Arduino musisz dokonać w nim kilku modyfikacji. Na początku szkicu znajdują się dwie linie kodu:

```
byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
byte ip[] = { 192, 168, 1, 30 };
```

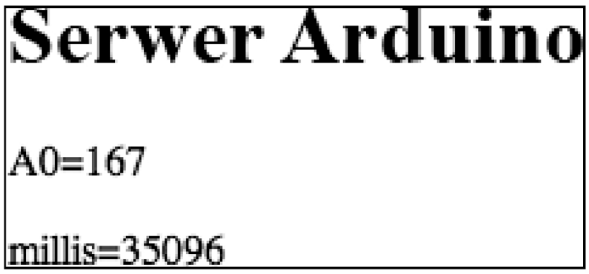
Pierwsza linia określa adres MAC. W sieci nie mogą się znajdować dwa urządzenia posiadające identyczny adres MAC. Druga linia kodu określa numer IP. Większość urządzeń, z których korzystasz w swojej sieci domowej, posiada adres IP przypisany automatycznie przez protokół DHCP (protokół dynamicznej konfiguracji węzłów). Nie ma możliwości automatycznego nadania adresu IP dla płytki pozwalającej Arduino na obsługę sieci Ethernet. Będziesz musiał samodzielnie określić adres IP tego urządzenia. Adres ten nie może być dowolny. Powinieneś przypisać do Arduino adres IP należący do puli adresów wewnętrznych sieci. Ponadto adres ten musi być obsługiwany przez Twój router. Adres będzie mieć postać 10.0.1.x lub 192.168.1.x. „x" w podanych adresach może być dowolną liczbą z zakresu od 0 do 255. Część adresów IP może być już zajęta przez urządzenia pracujące w Twojej sieci. Aby określić, który adres jest aktualnie nieużywany, otwórz panel administracyjny swojego routera. Poszukaj w nim informacji na temat urządzeń sieciowych (zakładki *DHCP*). Powinieneś znaleźć listę urządzeń i ich adresów IP podobną do listy pokazanej na rysunku 10.3. Kolejny wolny adres możesz przydzielić dla Arduino. W przypadku mojej sieci takim wolnym adresem jest np. 192.168.1.30.



Rysunek 10.3. *Poszukiwanie wolnego adresu IP*

Połącz Arduino z komputerem za pośrednictwem interfejsu USB i załaduj szkic. Odłącz kabel USB. Podłącz do Arduino przewód zasilający oraz kabel sieci Ethernet.

W przeglądarce internetowej wpisz adres IP, który wcześniej przypisałeś do Arduino. Przeglądarka powinna wyświetlić treść podobną do tej, jaka znajduje się na rysunku 10.4.



Rysunek 10.4. *Efekt działania prostego serwera Arduino*

Poniżej znajduje się kod szkicu *10.01.*:

```
// szkic 10.0.1 Przykładowy prosty serwer Arduino

#include <SPI.h>
#include <Ethernet.h>


// Adres MAC musi być unikalny. Poniższy numer nie powinien powodować konfliktów.
byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
// Adres IP jest zależny od Twojej lokalnej sieci
byte ip[] = { 192, 168, 1, 30 };

EthernetServer server(80);

void setup()
{
   Ethernet.begin(mac, ip);
   server.begin();
   Serial.begin(9600);
}

void loop()
{
   // nasłuchiwanie klienta
   EthernetClient client = server.available();
   if (client)
   {
      while (client.connected())
      {
         // wysyła standardowy nagłówek odpowiedzi http
         client.println("HTTP/1.1 200 OK");
         client.println("Content-Type: text/html");
         client.println();

         // wysyła treść
         client.println("<html><body>");
         client.println("<h1>Serwer Arduino</h1>");
         client.print("<p>A0=");
         client.print(analogRead(0));
```

```
            client.println("</p>");
            client.print("<p>millis=");
            client.print(millis());
            client.println("</p>");
            client.println("</body></html>");
            client.stop();
        }
        delay(1);
    }
}
```

Standardowa biblioteka Arduino obsługuje płytkę sieci Ethernet w sposób podobny do tego, w jaki biblioteka wyświetlacza LCD obsługiwała wyświetlacz LCD, co omówiono w rozdziale 9.

Funkcja `setup` inicjalizuje bibliotekę Ethernet, korzystając z uprzednio określonych adresów MAC oraz IP.

Funkcja `loop` odpowiada za obsługę żądań wysyłanych przez przeglądarkę internetową do serwera sieci Web. Jeżeli żądanie oczekuje na odpowiedź, to funkcja `server.available` zwróci klienta. Klient jest obiektem, który zostanie szerzej omówiony w rozdziale 11. Teraz wystarczy, że wiesz, że jeżeli klient istnieje (warunek ten jest sprawdzany przez pierwsze polecenie `if`), to jego połączenie z serwerem możesz sprawdzić, wywołując funkcję `client.connected`.

Kolejne trzy linie kodu zwracają nagłówek, który informuje przeglądarkę o tym, jaką treść ma wyświetlać. W naszym przypadku przeglądarka ma wyświetlać treść dokumentu HTML.

Po wygenerowaniu nagłówka pozostaje nam tylko stworzenie pozostałego kodu HTML, który będzie wysłany do przeglądarki. Kod ten musi zawierać standardowe znaczniki `<html>` i `<body>`. Umieścimy w nim znacznik nagłówka `<h1>`, a także dwa znaczniki `<p>`, które będą wyświetlały wartości odbierane przez analogowe złącze A0, a także wartość zwróconą przez funkcję `millis`. Wartość ta jest liczbą milisekund, które upłynęły od ostatniego uruchomienia Arduino.

Na koniec `client.stop` informuje przeglądarkę o tym, że przesłano już wszystkie dane. Teraz przeglądarka może wyświetlić stronę.

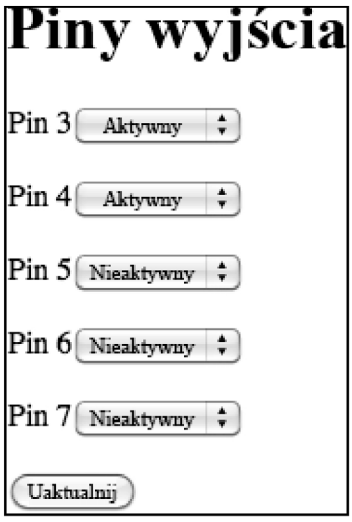
Konfigurowanie złączy Arduino za pośrednictwem sieci

Drugi przykład zastosowania płytki obsługującej sieć Ethernet pozwala na włączanie i wyłączanie (przy użyciu formularza wyświetlanego przez przeglądarkę internetową) złączy o numerach od D3 do D7.

W tym przykładzie będziesz musiał znaleźć sposób na przesłanie do Arduino danych dotyczących ustawień złączy.

Skorzystamy w tym celu z metody o nazwie **przesyłanie danych**. Jest ona częścią standardu HTTP. Aby zastosować tę metodę, będziesz musiał wbudować w kod HTML mechanizm wysyłający z Arduino do przeglądarki formularz. Formularz ten przedstawiono na

rysunku 10.5. Korzystając z niego, możesz włączyć lub wyłączyć każde ze złączy. Kliknięcie przycisku *Uaktualnij* wyśle do Arduino wybrane przez Ciebie ustawienia.



Rysunek 10.5. *Formularz*

Po kliknięciu przycisku *Uaktualnij* do Arduino wysyłane jest kolejne żądanie. To drugie żądanie jest podobne do pierwszego, jednakże zawiera ono parametry decydujące o włączeniu lub wyłączeniu złączy (pinów).

Parametr żądania jest czymś podobnym do parametru funkcji. Parametr funkcji pozwala Ci na przesłanie danych do funkcji. Mogą to być dane dotyczące liczby błysków diody. Natomiast parametr żądania pozwala Ci na przesyłanie danych do Arduino. Gdy Arduino odbierze żądanie przesłane za pośrednictwem sieci, może odczytać z niego dane dotyczące ustawień złączy, a następnie skonfigurować złącza zgodnie z przesłanymi parametrami.

```
// szkic 10.02. Obsługa złączy za pośrednictwem sieci

#include <SPI.h>
#include <Ethernet.h>

// Adres MAC musi być unikalny. Poniższy numer nie powinien powodować konfliktów.
byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
// Adres IP jest zależny od Twojej lokalnej sieci
byte ip[] = { 192, 168, 1, 30 };
EthernetServer server(80);

int numPins = 5;
int pins[] = {3, 4, 5, 6, 7};
int pinState[] = {0, 0, 0, 0, 0};
char line1[100];

void setup()
{
   for (int i = 0; i < numPins; i++)
   {
      pinMode(pins[i], OUTPUT);
   }
   Serial.begin(9600);
   Ethernet.begin(mac, ip);
```

```
    server.begin();
}

void loop()
{
    EthernetClient client = server.available();
    if (client)
    {
        while (client.connected())
        {
            readHeader(client);
            if (! pageNameIs("/"))
            {
                client.stop();
                return;
            }
            client.println("HTTP/1.1 200 OK");
            client.println("Content-Type: text/html");
            client.println();

            // wysyła właściwą treść formularza
            client.println("<html><body>");
            client.println("<h1>Piny wyjścia</h1>");
            client.println("<form method='GET'>");
            setValuesFromParams();
            setPinStates();
            for (int i = 0; i < numPins; i++)
            {
                writeHTMLforPin(client, i);
            }
            client.println("<input type='submit' value='Uaktualnij'/>");
            client.println("</form>");
            client.println("</body></html>");

            client.stop();
        }
    }
}

void writeHTMLforPin(EthernetClient client, int i)
{
    client.print("<p>Pin ");
    client.print(pins[i]);
    client.print("<select name='");
    client.print(i);
    client.println("'>");
    client.print("<option value='0'");
    if (pinState[i] == 0)
    {
        client.print(" selected");
    }
    client.println(">Nieaktywny</option>");
    client.print("<option value='1'");
    if (pinState[i] == 1)
    {
        client.print(" selected");
    }
```

```
   client.println(">Aktywny</option>");
   client.println("</select></p>");
}

void setPinStates()
{
   for (int i = 0; i < numPins; i++)
   {
      digitalWrite(pins[i], pinState[i]);
   }
}

void setValuesFromParams()
{
   for (int i = 0; i < numPins; i++)
   {
      pinState[i] = valueOfParam(i + '0');
   }
}

void readHeader(EthernetClient client)
{
   // wczytuje pierwszą linię nagłówka
   char ch;
   int i = 0;
   while (ch != '\n')
   {
      if (client.available())
      {
         ch = client.read();
         line1[i] = ch;
         i ++;
      }
   }
   line1[i] = '\0';
   Serial.println(line1);
}

boolean pageNameIs(char* name)
{
   // nazwa strony zaczyna się od znaku znajdującego się na pozycji o numerze 4
   // kończy się spacją
   int i = 4;
   char ch = line1[i];
   while (ch != ' ' && ch != '\n' && ch != '?')
   {
      if (name[i-4] != line1[i])
      {
         return false;
      }
      i++;
      ch = line1[i];
   }
   return true;
}
```

```
int valueOfParam(char param)
{
   for (int i = 0; i < strlen(line1); i++)
   {
      if (line1[i] == param && line1[i+1] == '=')
      {
         return (line1[i+2] - '0');
      }
   }
   return 0;
}
```

W szkicu zastosowano dwie tablice służące do sterowania złączami. Pierwsza tablica `pins` określa, które złącza będą użyte. Tablica `pinState` przechowuje dane dotyczące stanu każdego ze złączy — jest to 0 lub 1.

Odczyt nagłówka generowanego przez przeglądarkę jest niezbędny do pobrania informacji przesłanych do Arduino za pomocą formularza. Tak naprawdę wystarczą do tego dane zawarte w pierwszej linii nagłówka. Tablica znaków `line1` przechowuje te dane. Użytkownik po kliknięciu przycisku *Uaktualnij* wysyła dane z przeglądarki. Adres URL strony będzie wyglądał następująco:

```
http://192.168.1.30/?0=1&1=1&2=0&3=0&4=0
```

Parametry znajdują się po znaku ? i są oddzielone separatorem &. Przyjrzyjmy się tym parametrom. Zapis `0=1` oznacza, że pierwsze złącze zdefiniowane w tablicy (`pins[0]`) powinno przyjąć wartość 1 (zostać włączone). Gdybyś miał możliwość przyjrzenia się pierwszej linii nagłówka, dostrzegłbyś w niej te same parametry żądania:

```
GET /?0=1&1=1&2=0&3=0&4=0 HTTP/1.1
```

Przed parametrami znajduje się zapis `GET /`. Określa on stronę żądaną przez przeglądarkę. W tym przypadku / wskazuje stronę główną.

W funkcji `loop` wywołana zostaje funkcja `readHeader`. Odczytuje ona pierwszą linię nagłówka. Następnie funkcja `pageNameIs` sprawdza, czy żądana jest strona główna.

Następnie szkic generuje nagłówek i kod HTML wyświetlanego formularza. Przed napisaniem kodu HTML dotyczącego każdego ze złączy szkic uruchamia funkcję `setValues ↪FromParams`. Funkcja ta odczytuje każdy z parametrów żądania i wpisuje odpowiednie wartości do tablicy `pinStates`. Złącza wyjściowe są następnie konfigurowane zgodnie z danymi zawartymi w tej tablicy. Później osobno dla każdego ze złączy wywoływana jest funkcja `writeHTMLforPin`. Funkcja ta dla każdego złącza generuje listę wyboru. Lista ta jest budowana stopniowo. Polecenie `if` zapewnia wybór właściwych opcji.

Funkcje `readHeader`, `pageNameIs` i `valueOfParam` są przydatnymi funkcjami ogólnego przeznaczenia. Możesz z nich korzystać we własnych szkicach.

W celu sprawdzenia tego, czy złącza są naprawdę włączane i wyłączane, możesz skorzystać z multimetru, tak samo jak to miało miejsce w przykładach omówionych w rozdziale 6. Jeżeli chcesz, to możesz podłączyć do Arduino diody LED lub przekaźniki sterujące pracą jakichś urządzeń zewnętrznych.

Podsumowanie

W poprzednich dwóch rozdziałach omówiliśmy zastosowanie płytek stykowych i związanych z nimi bibliotek. Teraz czas, abyś poznał techniki związane z tworzeniem własnych bibliotek.

Rozdział 11.

C++ i biblioteki

Arduino jest prostym mikrokontrolerem. W większości przypadków szkice nie są zbyt obszerne, a więc podczas ich tworzenia znakomicie sprawdza się język C. Tak naprawdę programy uruchamiane przez Arduino są tworzone w języku C++. Jest to rozszerzenie języka C. Rozszerzenie to dodaje do tego języka obsługę **mechanizmów obiektowych**.

Mechanizmy obiektowe

Niniejsza książka jest za krótka, żeby przedstawić w niej dogłębną analizę zagadnień związanych z programowaniem w C++. Odnajdziesz tutaj jednakże podstawy C++ i programowania obiektowego. Naszym głównym celem będzie zwiększenie **hermetyzacji** tworzonych programów. Hermetyzacja pozwala na umieszczenie ważnych rzeczy w jednym miejscu. Dzięki temu C++ jest bardzo dobrym narzędziem do pisania bibliotek takich jak te, z których korzystaliśmy w poprzednich rozdziałach, w szkicach związanych z wyświetlaczem LCD oraz siecią Ethernet.

Istnieje wiele dobrych książek dotyczących zagadnień związanych z C++ i programowaniem obiektowym. Poszukaj jakichś wysoko ocenianych książek na ten temat w swojej ulubionej księgarni internetowej.

Klasy i metody

W programowaniu obiektowym występuje pojęcie **klas** wspomagających hermetyzację. Ogólnie rzecz biorąc, klasę można porównać do sekcji w programie — posiada ona zmienne (**zmienne składowe**) i **metody**. Metody odgrywają rolę podobną do roli funkcji w programie, jednakże są stosowane w klasach. Metody mogą być **publiczne** (mogą być stosowane w innych klasach) lub **prywatne** (mogą być wywoływane wyłącznie przez inne metody znajdujące się wewnątrz tej samej klasy).

Szkic Arduino jest jednym plikiem. Podczas pracy w C++ istnieje jednakże tendencja do jednoczesnej pracy z większą liczbą plików. Dla każdej klasy są tworzone dwa pliki. **Plik nagłówkowy**, który posiada rozszerzenie *.h*, a także **plik implementacji**, który posiada rozszerzenie *.cpp*.

Przykład wbudowanej biblioteki

Biblioteka wyświetlacza LCD była już stosowana w poprzednich rozdziałach. Przyjrzyjmy się jej bliżej.

Analizując szkic *09.01.* (otwórz go w środowisku programistycznym Arduino), zauważysz, że polecenie `include` dołącza plik *LiquidCrystal.h*:

```
#include <LiquidCrystal.h>
```

Jest to plik nagłówkowy klasy o nazwie `LiquidCrystal`. Plik ten dostarcza do szkicu Arduino informacje niezbędne do użycia biblioteki. Możesz edytować ten plik. Wystarczy odnaleźć folder, w którym zainstalowałeś środowisko Arduino, a następnie wejść do katalogu *libraries\LiquidCrystal*. Plik należy otworzyć za pomocą edytora tekstowego. Jeżeli pracujesz na komputerze typu Mac, kliknij prawym przyciskiem aplikację Arduino, a następnie wybierz opcję *Pokaż zawartość pakietu*. Następnie kieruj się do katalogu *Contents/Resources/Java/libraries/LiquidCrystal*.

Plik *LiquidCrystal.h* zawiera dużą ilość kodu. Jest to dość duża klasa biblioteki. Kod zawierający instrukcje bezpośrednio obsługujące wyświetlacz znajduje się w pliku *LiquidCrystal.cpp*.

W kolejnym podrozdziale stworzymy prostą przykładową bibliotekę. Powinno to w praktyce wyjaśnić Ci zagadnienia związane z bibliotekami.

Tworzenie bibliotek

Tworzenie bibliotek może wydawać się zadaniem, któremu podoła tylko doświadczony programista. Utworzenie biblioteki jest jednakże tak naprawdę dosyć proste. Na przykład możesz utworzyć bibliotekę z funkcji `flash` przedstawionej w rozdziale 4. Funkcja ta zapalała diodę LED określoną liczbę razy.

Potrzebne pliki C++ będziesz tworzyć w edytorze tekstowym. Będzie Ci potrzebny edytor podobny do programu TextPad (przeznaczony dla komputerów działających pod kontrolą systemu Windows) lub TextMate (przeznaczony dla komputerów działających pod kontrolą systemu OS X).

Plik nagłówkowy

Pracę rozpocznij od stworzenia folderu zawierającego wszystkie pliki biblioteki. Musisz go stworzyć w podfolderze przeznaczonym na biblioteki znajdującym się w folderze dokumentów Arduino. W systemie Windows folder *libraries* znajduje się w katalogu *Moje dokumenty*\

Arduino. W systemie OS X folder ten znajdziesz w katalogu *Dokumenty/Arduino*. W systemie Linux folder ten będzie umieszczony w katalogu zawierającym szkice. Jeżeli w katalogu *Arduino* nie znajduje się folder o nazwie *libraries*, to musisz takowy folder utworzyć samodzielnie.

Wszystkie biblioteki, które napiszesz sam, a także wszystkie „nieoficjalne" biblioteki muszą być umieszczone w folderze *libraries*.

W folderze *libraries* utwórz nowy folder o nazwie *Flasher*. Uruchom edytor tekstowy i umieść w nim następujący kod:

```
// Biblioteka błyskania diodą LED

#include "WProgram.h"

class Flasher
{
   public:
      Flasher(int pin, int duration);
      void flash(int times);
   private:
      int _pin;
      int _d;
};
```

Zapisz ten plik w folderze *Flasher* pod nazwą *Flasher.h*. Właśnie utworzyłeś plik nagłówkowy dla klasy biblioteki. Plik ten określa różne elementy klasy. Jest on podzielony na część publiczną (`public`) i prywatną (`private`).

W części publicznej znajduje się coś, co wygląda jak początek dwóch funkcji. Są to metody. Przyjmij, że na razie różnią się one od funkcji tylko tym, że są związane z klasą. Mogą być zastosowane tylko w charakterze części klasy. W przeciwieństwie do funkcji nie mogą być zastosowane samodzielnie.

Nazwa pierwszej metody — `Flasher` — rozpoczyna się wielką literą. Nazwy funkcji nie rozpoczynają się w ten sposób. Metoda ta ma taką samą nazwę co klasa. Metodę tę określamy mianem **konstruktora**. Możesz ją stosować w celu utworzenia nowego obiektu `Flasher`. Obiekt taki możesz następnie stosować w szkicu.

W szkicu możesz umieścić następujący kod:

```
Flasher slowFlasher(13, 500);
```

Stworzyłoby to nowy obiekt `Flasher` o nazwie `slowFlasher`, który powodowałby błyski diody podłączonej do złącza D13. Błyski te miałyby czas trwania o długości 500 milisekund.

Druga metoda występująca w klasie nosi nazwę `flash`. Metoda ta przyjmuje pojedynczy argument będący liczbą błysków. Element ten jest związany z klasą, a więc jeżeli chciałbyś go wywołać, to musiałbyś odnieść się do obiektu stworzonego wcześniej:

```
slowFlasher.flash(10);
```

Kod ten spowodowałby, że dioda błysnęłaby dziesięciokrotnie, a okres błysków diody został już wcześniej określony kodem definiującym obiekt `Flasher`.

Sekcja `private` klasy zawiera definicję dwóch zmiennych. Jedna zmienna określa numer złącza, a druga — `d` — czas. Każdy stworzony przez Ciebie obiekt klasy `Flasher` będzie zawierał te dwie zmienne. Pozwala to na zastosowanie określonego numeru złącza i czasu podczas tworzenia nowego obiektu `Flasher`.

Zmienne te określamy mianem zmiennych składowych — stanowią one jeden ze składników klasy. Ich nazwy zaczynają się od znaku podkreślenia, co jest dość niezwykłe. Jest to jednakże powszechnie stosowana konwencja, a nie konieczny zabieg. Inną popularną konwencją nazewnictwa takich zmiennych jest rozpoczynanie ich nazwy od małej litery *m* (od pierwszej litery angielskiego słowa *member*).

Plik implementacji

Plik nagłówkowy definiuje wygląd klasy. Przed Tobą utworzenie pliku, który tak naprawdę wykona całą pracę. Plik taki nosi nazwę pliku implementacji. Posiada on rozszerzenie *.cpp*.

Utwórz nowy plik zawierający poniższy kod, a następnie zapisz go w folderze *Flasher* pod nazwą *Flasher.cpp*.

```
#include "WProgram.h"
#include "Flasher.h"

Flasher::Flasher(int pin, int duration)
{
   pinMode(pin, OUTPUT);
   _pin = pin;
   _d = duration / 2;
}
void Flasher::flash(int times)
{
   for (int i = 0; i < times; i++)
   {
      digitalWrite(_pin, HIGH);
      delay(_d);
      digitalWrite(_pin, LOW);
      delay(_d);
   }
}
```

Składnia tego pliku może wydawać Ci się nieco dziwna. Przed nazwami obu metod znajduje się prefiks `Flasher::`. Zapis ten informuje o tym, że metody te należą do klasy `Flasher`.

Metoda-konstruktor (`Flasher`) przypisuje każdy ze swoich parametrów do odpowiedniej zmiennej prywatnej. Parametr `duration` jest dzielony przez dwa, a następnie przypisywany do zmiennej składowej `_d`. Zabieg ten zastosowano ze względu na fakt, że funkcja `delay` jest wywoływana dwukrotnie, ponieważ całkowity czas powinien być sumą czasu trwania błysku oraz pauzy pomiędzy błyskami.

Funkcja `flash` przeprowadza właściwą operację błyskania. Funkcja ta zostaje zapętlona odpowiednią liczbę razy, włączając i wyłączając diodę na określony czas.

Uzupełnianie swojej biblioteki

Posiadasz już podstawową wiedzę niezbędną do utworzenia biblioteki. W zasadzie tworzenie biblioteki mógłbyś zakończyć na tym etapie, a działałaby ona poprawnie. Są jeszcze jednakże dwie rzeczy, które powinieneś zrobić przed ukończeniem pracy nad biblioteką. Po pierwsze, możesz zdefiniować słowa kluczowe stosowane w bibliotece. W wyniku tej operacji środowisko programistyczne Arduino będzie wyróżniać je odpowiednim kolorem, gdy użytkownik wprowadzi je podczas edycji kodu. Ponadto powinieneś dołączyć do biblioteki jakieś przykłady jej użycia.

Słowa kluczowe

W celu zdefiniowania słów kluczowych musisz utworzyć w katalogu *Flasher* plik o nazwie *keywords.txt*. Plik ten będzie zawierać dwie poniższe linie kodu:

```
Flasher KEYWORD1
flash   KEYWORD2
```

Kod musi koniecznie zostać umieszczony w pliku tekstowym w formie dwukolumnowej tabeli. Lewa kolumna zawiera słowa kluczowe, a informacje zawarte w prawej kolumnie określają rodzaje poszczególnych słów kluczowych. Nazwy klas powinny być oznaczone jako `KEYWORD1`, a nazwy metod jako `KEYWORD2`. Bez znaczenia jest liczba spacji i tabulatorów zastosowanych w celu oddzielenia kolumn, jednakże każde kolejne słowo kluczowe musi rozpoczynać się od nowego wiersza.

Przykłady

Jako porządny członek społeczności Arduino powinieneś również dodać do swojej biblioteki folder zawierający przykłady. W omawianym przypadku biblioteka jest na tyle prosta, że wystarczy dodanie jednego przykładu.

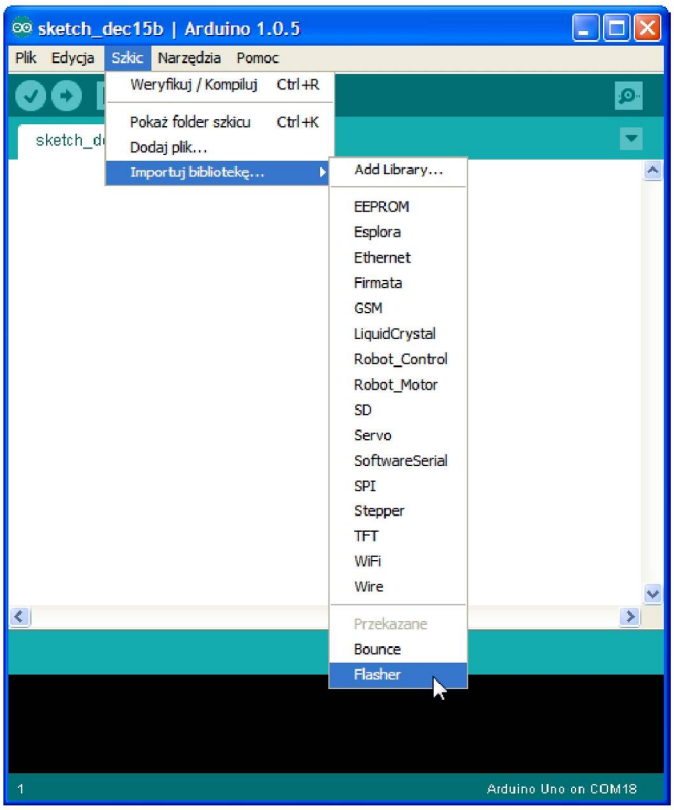
Przykład musi zostać umieszczony w folderze o nazwie *examples*, który ma znajdować się w folderze *Flasher*. Plik, który stworzysz, będzie tak naprawdę szkicem, który możesz napisać za pomocą zintegrowanego środowiska programistycznego Arduino. Przed przystąpieniem do pracy musisz uruchomić ponownie aplikację Arduino. Po ponownym uruchomieniu aplikacja ta wykryje nową bibliotekę.

W celu utworzenia nowego szkicu w środowisku Arduino z menu *Plik* wybierz opcję *Nowy*. Następnie w menu *Szkic* najedź kursorem na opcję *Importuj bibliotekę*. Na ekranie swojego komputera powinieneś widzieć to, co przedstawiono na rysunku 11.1.

W widocznym podmenu znajdują się oficjalne biblioteki. Pod nimi znajduje się linia, pod którą umieszczone zostały biblioteki „nieoficjalne". Jeżeli wykonałeś wszystko poprawnie, to na tej liście powinieneś widzieć bibliotekę Flasher.

Jeżeli na liście nie ma tej biblioteki, to prawdopodobnie folder *Flasher* nie został umieszczony w folderze *libraries* znajdującym się w folderze zawierającym Twoje szkice. Sprawdź jeszcze raz, gdzie umieściłeś folder *Flasher*.

Wpisz poniższy kod do otwartego przed chwilą okna szkicu:

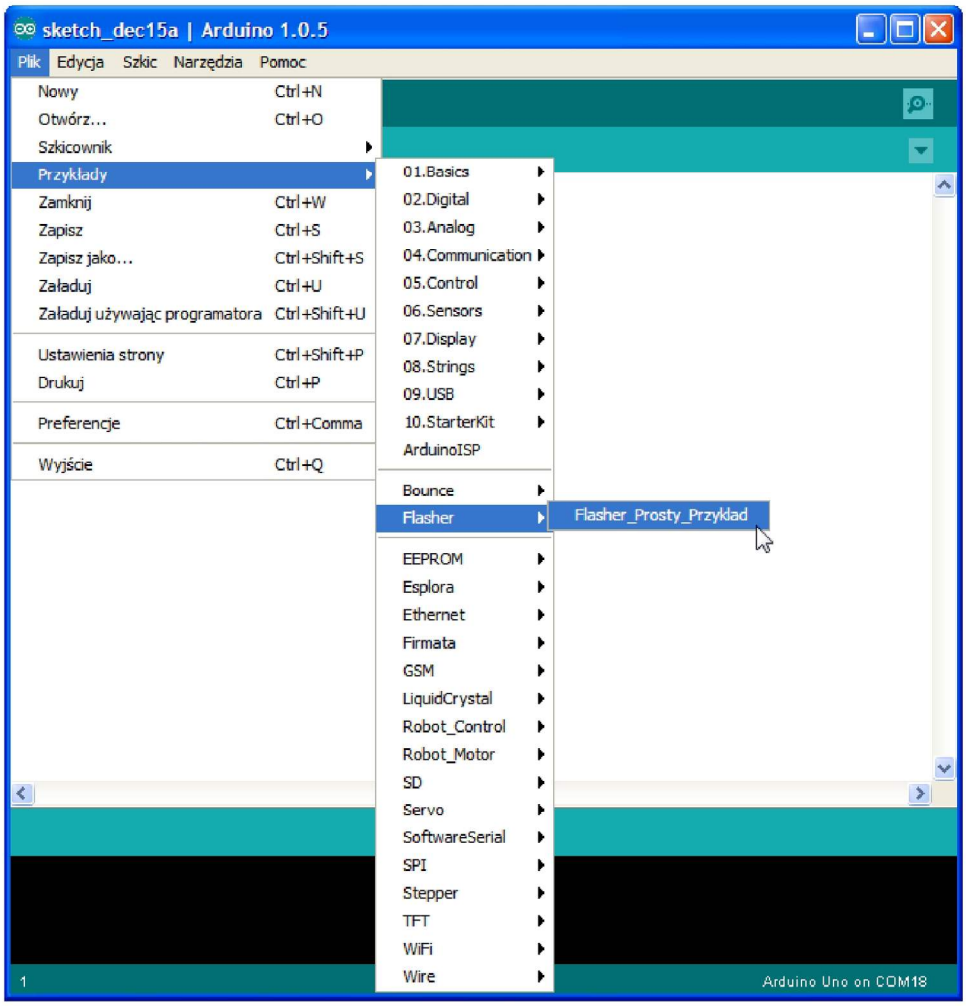


Rysunek 11.1. *Importowanie biblioteki Flasher*

```
#include <Flasher.h>

int ledPin = 13;
int slowDuration = 300;
int fastDuration = 100;

Flasher slowFlasher(ledPin, slowDuration);
Flasher fastFlasher(ledPin, fastDuration);

void setup(){}

void loop()
{
   slowFlasher.flash(5);
   delay(1000);
   fastFlasher.flash(10);
   delay(2000);
}
```

Zintegrowane środowisko programistyczne Arduino nie pozwoli Ci na zapisanie tego przykładowego szkicu bezpośrednio w folderze bibliotek, a więc zapisz go w dowolnej lokalizacji pod nazwą *Flasher_Prosty_Przyklad*. Następnie przenieś cały folder szkicu do folderu *examples* znajdującego się w folderze Twojej biblioteki.

Uruchom ponownie aplikację Arduino. Teraz powinieneś mieć możliwość skorzystania z przykładowego szkicu w sposób pokazany na rysunku 11.2.



Rysunek 11.2. *Otwieranie przykładowego szkicu*

Podsumowanie

C++ ma o wiele więcej praktycznych zastosowań niż pisanie bibliotek. Ten rozdział należy traktować jako wstęp do C++. Przedstawiliśmy Ci zagadnienia związane z C++, które pozwolą Ci na sprawne posługiwanie się Arduino. Arduino jest prostym urządzeniem. Często możesz czuć pokusę stworzenia zbyt skomplikowanego programu, który mógłby być zastąpiony czymś o wiele prostszym.

Jest to ostatni rozdział książki. Jeżeli chcesz uzyskać więcej informacji na temat Arduino, dobrym punktem wyjścia jest oficjalna witryna Arduino: *http://www.arduino.cc/*. Warto, abyś odwiedził również stronę internetową niniejszej książki: *http://www.helion.pl/ksiazki/ardupo.htm*. Znajdziesz tam wiele przydatnych materiałów.

Jeżeli chcesz uzyskać pomoc lub poradę, możesz zwrócić się o pomoc na forum społeczności Arduino: *http://forum.arduino.cc/*. Forum to jest bardzo przydatne podczas pracy z Arduino — korzysta z niego także autor niniejszej książki, jako użytkownik o pseudonimie Si.

Skorowidz

A

adres MAC, 124
akapit, 123
alfabet Morse'a, 67, 71, 73, 107
algorytm, 67, 75
antena, 85
Arduino
  aplikacja, 33
  dokumentacja, 19
  historia, 19
  uruchamianie, 27

B

Banz Massimo, 19
biblioteka, 91
  Bounce, 91, 92
  Ethernet, 126
  funkcji matematycznych, 99
  nieoficjalna, 108
  obsługująca wyświetlacze LCD, 116, 134
  pgmspace, 108
  Processing, 97
  PROGMEM, 108
  tworzenie, 134
  Wiring, 97
bit, 99
Boole George, 60
bufor, 75

C

camelCase, 42
CPU, 18
Cuartielles David, 19

D

dane
  kompresja, 112, 113
  struktura, 67
  typ, Patrz: zmienna typ
  zapis
    do pamięci EEPROM, 109, 110, 111, 112
    do pamięci flash, 108
    unia, 110
  zapisywanie, 107
debugowanie, 56
dioda LED, Patrz: LED
dyrektywa
  #define, 52, 56
  #include, 92
  PROGMEM, 107, 108, 112

E

EEPROM, 15, 18, 109
  wymazywanie, 112
  zapis danych, 109, 110, 111, 112
Ethernet, 121

F

formularz, 126
funkcja, 53, 58
  abs, 100
  argument, 36, 58
  bitRead, 101
  bitWrite, 102
  client.connected, 126
  client.stop, 126
  constrain, 100
  cos, 100
  definiowanie, 40
  delay, 39, 51
  digitalWrite, 36, 39, 88
  EEPROM.write, 109
  highByte, 110
  interrupts, 104
  log, 100
  loop, 38, 39, 40, 41, 75
  lowByte, 110
  map, 100
  matematyczna, 47, 99
  max, 100
  millis, 126
  min, 100
  noTone, 103
  pageNameIs, 130
  parametr, 127
  parametry, 54, 55
  pgm_read_word, 108
  pinMode, 40, 83

funkcja
pow, 100
random, 98
readHeader, 130
Serial.available, 75, 76
Serial.println, 45
server.available, 126
setup, 38, 39, 40, 44, 74
setValuesFromParams, 130
shiftOut, 103
sin, 100
sqrt, 100
tan, 100
tone, 102
trygonometryczna, 100
typu int, 58
typu void, 58
valueOfParam, 130
wartość zwracana, 58
wejścia, 102
writeHTMLforPin, 130
wyjścia, 102
wywołanie, 36
wywoływanie, 40

G

generator
alfabetu Morse'a, 71
drgań, 19
liczb losowych, 97, 99
osadzanie, 99
sprzętowy, 99

H

hermetyzacja, 56, 133
HTML, 122
znacznik, Patrz: znacznik

I

interfejs, 15
USB, 19, 45, 97
interferencja
elektromagnetyczna, 87

J

jednostka centralna, Patrz: CPU
język
C, 35
funkcja wbudowana, 36
nazwa, 36
składnia, 36, 42
C++, 93, 133
hipertekstowego znakowania informacji, Patrz: HTML
Java, 97

K

klasa, 133
LiquidCrystal, 134
kod, Patrz też: szkic
ASCII, 71, 76
blok, 40
HTML, Patrz: HTML
testowanie, 44
wcięcia, 62
zapis, 62
komentarz, 64
komunikat błędu, 37
konstruktor, 135, 136
kontroler Bluetooth, 23

L

LED, 78
liczba
całkowita, 42
losowa, 97
mapowanie, 100
ograniczenie, 100
pseudolosowa, 98
w systemie dwójkowym, Patrz: bit
zmiennoprzecinkowa, 59
licznik, 50
literał łańcuchowy, 71

M

mechanizm obiektowy, 133
metoda, 133, 135
Flasher, 135
prywatna, 133
publiczna, 133
mikrokontroler
ATmega1280, 22
ATmega168, 21
ATmega328, 16, 18, 21
bufor, Patrz: bufor
PIC, 25
wymiana, 22
modem USB, 29
modulacja czasu trwania impulsu, Patrz: PWM
moduł wyświetlacza LCD, Patrz: wyświetlacz LCD
monitor portu szeregowego, 44, 83, 85
multimetr, 81

N

nagłówek, 123, 130
numer IP, 124

O

obiekt bouncer, 93
operacja
dodawania, 47
dzielenia, 47
iloczynu logicznego, 60
logiczna, 60
matematyczna, 97, 99
kolejność, 47
mnożenia, 47
na bitach, 97
odejmowania, 47
porównywania, 60
sumy logicznej, 60
operator porównywania, 48
opóźnienie, 91
oprogramowania
instalowanie, 28

P

pamięć, 70
flash, 18, 107
zapis danych, 108
o dostępie bezpośrednim, Patrz: RAM
operacyjna, 19
robocza, 18
stała programowalna, Patrz: EEPROM

pętla
for, 49
argumenty, 50
while, 51
pierwiastek, 100
pin, Patrz: złącze
plik
.cpp, 134, 136
.h, 134
implementacji, 134, 136
LiquidCrystal.h, 134
nagłówkowy, 134, 136
płyta
Arduino, 17, 24, 25
nieoficjalna, 25
zasilanie, Patrz: zasilanie
złącze analogowe, Patrz: złącze analogowe
złącze cyfrowe, Patrz: złącze cyfrowe
Arduino Bluetooth, 23
Arduino Lilypad, 24
Arduino Mega, 22
Arduino Nano, 22
Arduino Uno, 21, 22, 29
Chipkit, 25
Diecimila, 21
Duemilanove, 21
Femtoduino, 25
Freeduino, 25
Roboduino, 25
rozwojowa, 16
Ruggeduino, 25
Seeeduino, 25
stykowa, 20, 22
Ethernet, Patrz: Ethernet
Host USB, Patrz: Host USB
Motor, Patrz: Motor
Teensy, 25
wyświetlacza LCD, Patrz: wyświetlacz LCD
polecenie, 47
if, 47, 48, 49
warunek, Patrz: warunek
include, 134
int, 113
return, 58
port
COM3, 29
szeregowy, 19, 29
monitor, Patrz: monitor portu szeregowego
program, Patrz: szkic
programator zewnętrzny, 24
programowanie intencyjne, 75
protokół
DHCP, 124
HTTP, 122
przeglądarka, 126
przekaźnik, 20
przełącznik, 85
dotykowy, 88
typu klik, 88
przerwanie, 103
CHANGE, 104
FALLING, 104
RISING, 104
przypisanie, 45, 49
Pulse Width Modulation, Patrz: PWM
PWM, 93, 95

R

RAM, 15, 18, 107
regulator
napięcia, 17
rezystor
podwyższający, 85, 89
wewnętrzny, 88
rzutowanie, 113

S

serwer
sieci Web, 123
wyszukiwarki Google, 123
shield, Patrz: płyta stykowa
silnik elektryczny, 18, 20
słowo kluczowe
int, 58
PROGMEM, 108
static, 57
void, 39, 40
stała, 56, 109
standard ASCII, Patrz: kod ASCII
stuki, 89
redukcja, 91
sygnał przerwania, 104
system
dwójkowy, 99
heksadecymalny, Patrz: system szesnastkowy
szesnastkowy, 101
szkic, 18, Patrz też: kod
Blink, 27
kompilacja, 32, 37, 39
ładowanie, 28, 32, 37
wbudowany, 33
wykonywanie, 32
szkicownik, 33

T

tablica, 67
element, 68, 70
typu char, 71
indeks, 67
łańcuchów, 71
tablic, 74
znaków, 108
temperatura, 45

U

UART, 75
układ
ENC28J60, 122
HD44780, 115
regulatora napięcia, Patrz: regulator napięcia
rejestru przesuwnego, 103
unia, 110
Universal Asynchronous Receiver/Transmitter, Patrz: UART
urządzenie sieciowe, 124

W

wartość logiczna, 60
wejście, Patrz: złącze:wejściowe
Wirth Niklaus, 67
wyjście, Patrz: złącze:wyjściowe
wyświetlacz LCD, 115, 116

Z

zasilanie, 122
złącze, 17
zegar, 103

złącze
analogowe, 18
cyfrowe, 18
D2, 104
D3, 104
GND, 85
sterowanie, 130
zasilające, 17
USB, 21, 23, 24, 75
wejściowe, 40, 85
analogowe, 81
cyfrowe, 81, 84
pływające, 85, 99
włączanie, 126
wyjściowe, 40
13, 35
analogowe, 81, 93
cyfrowe, 81, 95
wyłączanie, 126
zmienna, 42, 45
boolean, 60, 61
byte, 61
char, 61, 71
deklarowanie, 46
double, 61
float, 59, 61, 112
globalna, 55, 56, 74
int, 42, 46, 59, 61, 100, 112
lokalna, 56
inicjalizowanie, 56, 57
long, 61
łańcuchowa, 72, 111
nazwa, 42
składowa, 133
tablicowa, 68, 70
typ, 61
konwersja, 113
unsigned int, 61
unsigned long, 61
wartość początkowa, 46
znacznik
body, 123, 126
h1, 123, 126
html, 123, 126
otwierający, 123
p, 123, 126
zamykający, 123
znak
!=, 48
&, 130
&&, 60
*/, 64
/*, 64
//, 64
?, 130
||, 60
<, 48, 123
<=, 48
=, 49, 52
==, 48, 49
>, 48, 123
>=, 48
biały, 63
cudzysłowu podwójnego, 71
gwiazdki, 47
końca łańcucha, 72
kreski ukośnej, 47
nawiasu, 40, 47
nawiasu klamrowego, 40, 63, 68
nawiasu kwadratowego, 68
nowego wiersza, 63
null, 72
przecinka, 36, 50
równości, 45
separatora dziesiętnego, 59
spacji, 62, 63
średnika, 36, 40, 50, 52
tabulacji, 62, 63
większości, 48

Ż

żądanie, 127