OD PODSTAW

Witold Wrotek

Helion

Redaktor prowadzący: Małgorzata Kulik

Projekt okładki: Studio Gravite / Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki
Grafika na okładce została wykorzystana za zgodą Shutterstock.com.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

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

ISBN: 978-83-283-9715-6

Printed in Poland.

SPIS TREŚCI

3

NIEPRZERWANIE O PRZERWANIACH

Przerwania zasługują na poświęcenie im sporej uwagi, gdyż pozwalają mikrokontrolerowi reagować na zdarzenia zewnętrzne bez konieczności ustawicznego generowania zapytań, czy analizowany stan nie uległ zmianie. Powoduje to zredukowanie liczby wykonywanych instrukcji do naprawdę niezbędnych, a przez to zwolnienie zasobów dla pozostałych zadań.

Gdy zaczynałem naukę programowania, bardziej doświadczony kolega podpowiedział mi, abym (jeżeli tylko się da) do pisania nowych programów wykorzystywał programy, które już napisałem i sprawdziłem. Tą radą dzielę się z Tobą. Gromadząc gotowe programy, stworzysz bibliotekę, która pozwoli Ci zaoszczędzić wiele czasu.

Przerwania mają ograniczenia. Jeżeli podczas wykonywania poleceń znajdujących się w procedurze obsługującej przerwanie zdarzy się kolejne, wykonywana procedura nie zostanie przerwana. Innymi słowy, sygnał nowego przerwania przychodzący podczas wykonywania bieżącego przerwania jest ignorowany. Dlatego program

```
void setup()
{
   pinMode(diodaPin, OUTPUT);
   pinMode(przerwaniePin, INPUT);
   digitalWrite(przerwaniePin, HIGH);
   attachInterrupt(0, szybciej, FALLING);
   attachInterrupt(0, wolniej, RISING);
}
```

nie będzie działał poprawnie. Gdy zacznie obsługiwać przerwanie `attachIn ↪terrupt(0, szybciej, FALLING)` powodujące przyspieszone miganie diody, będzie ślepy i głuchy na wszystkie inne przerwania. Instrukcję

```
attachInterrupt(0, wolniej, RISING);
```

będzie ignorował.

Zapobiega to zaburzeniu procedury obsługi przerwania przez kolejne przerwanie. Ma to swoje uzasadnienie. Wyobraź sobie, że Arduino UNO użyto do utrzymywania wilgotności i temperatury w szklarni: włącza pompę, gdy jest za sucho, a ogrzewanie, gdy jest za zimno. Załóżmy, że temperatura w szklarni spadła. Arduino włączył ogrzewanie. Wzrost temperatury zintensyfikował parowanie i gleba się przesuszyła. Przyszła noc i temperatura na zewnątrz spadła. Grzanie zostało zintensyfikowane. Podlewanie nie włączy się, póki nie zakończy się grzanie. W rezultacie zamiast szklarni mamy suszarnię.

Z zegarów i przerwań korzysta funkcja `millis`. Z funkcji `millis` korzysta funkcja `delay`. Obie nie nadają się do obsługi jednoczesnych przerwań.

Funkcja `Verify` (rysunek 1.17) nie sygnalizuje błędu po umieszczeniu programu obsługi przerwania w ten sposób, że blokuje inne przerwanie. Ona sprawdza składnię. Za sens odpowiada programista.

Alternatywnym rozwiązaniem jest użycie programu działającego w pętli:

```
int ledPin = 13;     // dioda LED - digital pin 13
int inPin = 5;       // sterowanie - digital pin 5
int val = 0;         // zmienna - wczytany stan

void setup() {
   pinMode(ledPin, OUTPUT);      // deklaracja - pin 13. wyjście
   pinMode(inPin, INPUT);        // deklaracja - pin 5. wejście
}

void loop() {
   val = digitalRead(inPin);     // odczyt stanu pinu
   digitalWrite(ledPin, val);    // wyświetlenie wczytanego stanu
}
```

Podanie na wejście cyfrowe o numerze 5 stanu niskiego np. zworą włożoną w wyjście GND po stronie analogowej powoduje zgaszenie diody LED L.

Podanie na wejście cyfrowe o numerze 5 stanu wysokiego np. zworą włożoną w wyjście 5 V po stronie analogowej powoduje zapalenie diody LED L.

Po uruchomieniu powyższego programu dioda LED L jest włączona nawet przy braku sygnału podanego na wejście 5.

Teraz zmienię program tak, aby stan na wejściu 5. był sygnalizowany częstotliwością migania diody.

Pisząc program, wprowadzaj zmiany pojedynczo i testuj. Dzięki temu będziesz wiedzieć, jaki wpływ na działanie programu ma konkretna zmiana.

Program, który pozwala na sterowanie szybkością migania diody za pomocą napięcia podanego na wejście cyfrowe, ma na przykład taką postać:

```
int ledPin = 13;                  // dioda LED - digital pin 13
int inPin = 5;                    // sterowanie - digital pin 5
int val = LOW;                    // zmienna - wczytany stan
int delayPeriodH = 100;           // opóźnienie 0,1 s
int delayPeriodL = 1000;          // opóźnienie 1 s

void setup() {
   pinMode(ledPin, OUTPUT);       // deklaracja - pin 13. wyjście
   pinMode(inPin, INPUT);         // deklaracja - pin 5. wejście
}

void loop() {
   val = digitalRead(inPin);      // odczyt stanu pinu
   if (val == HIGH) {
      digitalWrite(ledPin, HIGH); // to się stanie, gdy wczytana wartość będzie
                                  // równa HIGH
      delay(delayPeriodH);        // na wejściu HIGH - dioda miga szybko
      digitalWrite(ledPin, LOW);
      delay(delayPeriodH);
   }
   if (val == LOW) {
      digitalWrite(ledPin, HIGH); // to się stanie, gdy wczytana wartość będzie
                                  // równa LOW
      delay(delayPeriodL);        // na wejściu LOW - dioda miga wolno
      digitalWrite(ledPin, LOW);
      delay(delayPeriodL);
   }
}
```

Powyższy program nie wykorzystuje przerwań.

Linie następujące po `void loop()`, a znajdujące się pomiędzy nawiasami klamrowymi są powtarzane stale. Absorbują cały czas mikrokontrolera. Każde wykonanie pętli rozpoczyna się od wczytania stanu na wejściu `inPin` i przypisania go zmiennej `val` (`val = digitalRead(inPin)`).

Następnie znajduje się instrukcja warunkowa `if`. Sprawdza ona warunek i wykonuje następujący po warunku zestaw instrukcji, jeśli warunek został spełniony. Warunek podany jest w nawiasie okrągłym (`val == HIGH`). Będzie on prawdziwy, gdy zmienna `val` ma wartość napięcia odpowiadającą stanowi wysokiemu. W tabeli 3.1 podałem operatory porównania.

Tabela 3.1. Operatory porównania

Zapis	Znaczenie
`x == y`	x jest równy y
`x != y`	x nie jest równy y
`x < y`	x jest mniejszy niż y
`x > y`	x jest większy niż y
`x <= y`	x jest mniejszy lub równy y
`x >= y`	x jest większy lub równy y

Pojedynczy znak równości (=) jest operatorem przypisania i nadaje zmiennej x wartość znajdującą się na prawo od znaku równości. Podwójny znak równości (==) jest operatorem porównania i zwraca wartość logiczną `TRUE`, gdy wartości znajdujące się po jego obu stronach są równe.

Jeżeli warunek następujący po `if` jest spełniony, wykonywane są instrukcje zawarte w nawiasie klamrowym. Jeżeli warunek nie jest spełniony, wszystkie instrukcje znajdujące się w nawiasie klamrowym są ignorowane i mikrokontroler przechodzi do wykonania kolejnej instrukcji. W moim programie jest to sprawdzenie warunku `val == LOW`.

Działanie funkcji zwracającej prawdę lub fałsz jest podobne do odpowiadania w teleturnieju (rysunek 3.1).

Rysunek 3.1. Udzielenie odpowiedzi zgodnej ze wzorcem (prawdziwej) jest warunkiem zaliczenia pytania[5]

Polecenia zawarte w nawiasie klamrowym są podobne niezależnie od sprawdzanego warunku. W pierwszym przypadku są to:

```
digitalWrite(ledPin, HIGH); // to się stanie, gdy wczytana wartość będzie równa HIGH
delay(delayPeriodH);        // na wejściu HIGH - dioda miga szybko
digitalWrite(ledPin, LOW);
delay(delayPeriodH);
```

W drugim przypadku są to

```
digitalWrite(ledPin, HIGH); // to się stanie, gdy wczytana wartość będzie równa LOW
delay(delayPeriodL);        //na wejściu LOW - dioda miga wolno
digitalWrite(ledPin, LOW);
delay(delayPeriodL);
```

W obu przypadkach do diody LED wysyłany jest stan wysoki `digitalWrite ↪(ledPin, HIGH)`, który powoduje jej zapalenie.

Następnie dioda jest włączona, a mikrokontroler odlicza czas. W pierwszym przypadku jest to 0,1 sekundy (bo `delayPeriodH = 100`). W drugim przypadku jest to 1 sekunda (gdyż `delayPeriodH = 1000`).

[5] Źródło: *https://pixabay.com/illustrations/man-girl-woman-work-paper-2399982/*; dostęp 18 sierpnia 2022.

Trzecia instrukcja jest w obu warunkach identyczna i powoduje wyłączenie diody LED `digitalWrite(ledPin, LOW)`. Dioda jest wyłączona, a mikrokontroler znów odlicza czas. W pierwszym przypadku jest to 0,1 sekundy (bo `delayPeriodH ↪= 100`). W drugim przypadku jest to 1 sekunda (gdyż `delayPeriodH = 1000`).

Po wykonaniu jednej lub drugiej instrukcji warunkowej program wraca na początek pętli i sprawdza wartość na wejściu.

Przerwanie pracy w pętli jest w tym programie możliwe tylko przez jedno z działań:

- naciśnięcie klawisza *Reset*,
- wyłączenie zasilania Arduino UNO,
- wgranie nowego programu.

Blokowanie i odblokowywanie przerwania

Można sobie wyobrazić wiele sytuacji, w których przerwanie może zaburzyć pracę programu: przesyłanie danych przy użyciu interfejsu szeregowego, generowanie impulsów o dokładnie określonej szerokości lub częstotliwości itp. Pojawienie się przerwania w takim momencie spowoduje błąd w działaniu programu.

Jak zablokować przerwanie?

Wczytaj program, który zmienia częstotliwość migania diody LED podłączonej do pinu 13. w zależności od poziomu logicznego na pinie 2.

```
// Program zmienia częstotliwość migania diody LED podłączonej
// do pinu 13. w zależności od poziomu logicznego na pinie 2.
// Działanie przerwania zostało zablokowane instrukcją noInterrupts

int przerwaniePin = 2;
int diodaPin = 13;
int okres = 1000;

void setup() {
   pinMode(diodaPin, OUTPUT);
   pinMode(przerwaniePin, INPUT);
   digitalWrite(przerwaniePin, HIGH);
   noInterrupts();
   // tutaj umieść kluczowy, wrażliwy na czas kod
   attachInterrupt(0, szybciej, FALLING);
}

void loop() {
   digitalWrite(diodaPin, HIGH);
```

```
   delay(okres);
   digitalWrite(diodaPin, LOW);
   delay(okres);
}

void szybciej() {
   okres = 100;
}
```

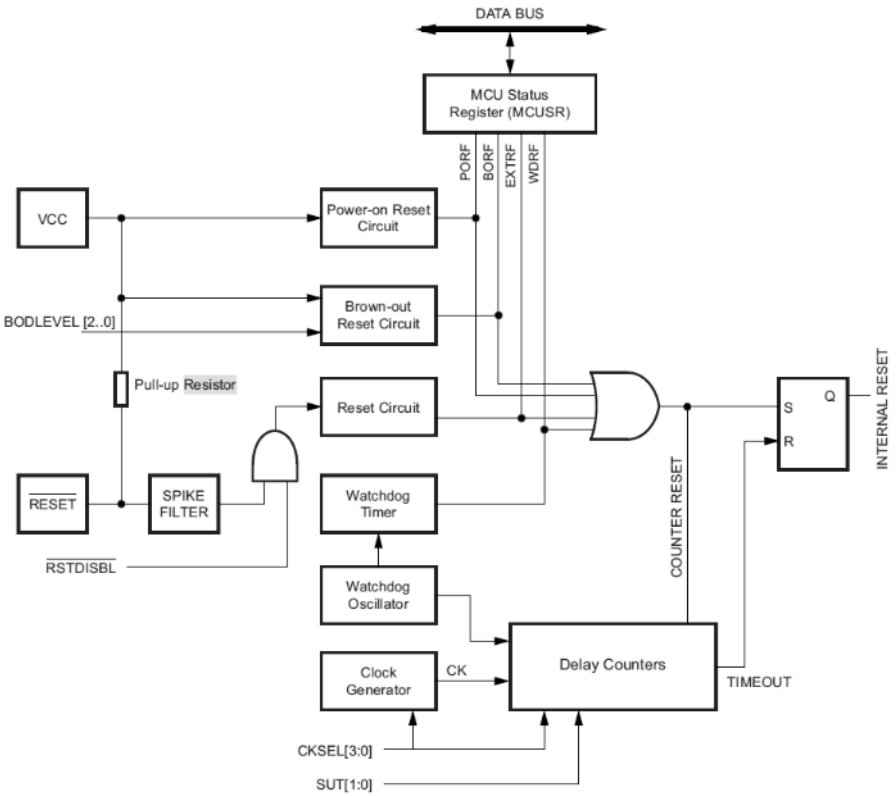
Po zweryfikowaniu programu i wysłaniu go do płytki Arduino UNO dioda LED jest zapalona cały czas. Stan podawany na obsługujące przerwanie wejście o numerze 2 nie ma wpływu na częstotliwość świecenia diody. Jak przywrócić obsługę przerwań? Zrobiłem to w kolejnym programie:

```
// Program zmienia częstotliwość migania diody LED podłączonej
// do pinu 13. w zależności od poziomu logicznego na pinie 2.
// Działanie przerwania zostało zablokowane instrukcją noInterrupts,
// a następnie odblokowane instrukcją Interrupts

int przerwaniePin = 2;
int diodaPin = 13;
int okres = 1000;

void setup() {
   pinMode(diodaPin, OUTPUT);
   pinMode(przerwaniePin, INPUT);
   digitalWrite(przerwaniePin, HIGH);
   noInterrupts();
   // tutaj umieść kluczowy, wrażliwy na czas kod
   interrupts();
   // inny kod umieść tutaj
   attachInterrupt(0, szybciej, FALLING);
}

void loop() {
   digitalWrite(diodaPin, HIGH);
   delay(okres);
   digitalWrite(diodaPin, LOW);
   delay(okres);
}

void szybciej() {
   okres = 100;
}
```

Po zweryfikowaniu i wysłaniu programu do płytki Arduino UNO częstotliwość migotania diody można zwiększyć, podając stan niski na wejście obsługi przerwania. Działanie programu nie różni się niczym od tego bez blokowania i odblokowania przerwań.

Rezystor podciągający

Rezystor podciągający znacznie upraszcza budowę układu (rysunek 3.2).



Rysunek 3.2. Rezystor podciągający zabezpiecza przed zwarciem zasilania VCC do masy po podaniu na wejście resetujące stanu niskiego[6]

Pinów analogowych można używać także jako cyfrowych. Nie da się używać pinów cyfrowych jako analogowych.

[6] Źródło: *https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf*; dostęp 19 sierpnia 2022.

Mikrokontroler wykonany jest w technologii CMOS. Ma wejścia o dużej impedancji. Na nich mogą indukować się sygnały pochodzące od zewnętrznych pól magnetycznych. W rezultacie mogą powstawać przypadkowe stany nieustalone zaburzające działanie układu. Polecenie `pinMode(2, INPUT_PULLUP)` spowoduje, że pin o numerze 2 będzie pinem wejściowym „podciągniętym" do zasilania. Przez wbudowany w mikrokontroler rezystor wejście układu będzie podłączone do dodatniego bieguna zasilania. Wyeliminuje to indukowanie się na wejściu napięć o wartościach losowych.

Wyjście typu open kolektor lub open dren z rezystorem podciągającym ma tę zaletę, że taką linię sygnałową może jednocześnie „ściągać" do masy wiele tranzystorów. Nie spowoduje to uszkodzenia układu, bo natężenie prądu ogranicza rezystor podciągający.

Rezystor podciągający stosowany jest w układach, w których po jednej linii odbywa się komunikacja w dwóch kierunkach, np. w systemie I2C.

W mikrokontrolerze AVR ATmega328 są tylko rezystory podciągające (ang. *pull-up*). W innych modelach mogą być także ściągające (ang. *pull-down*).

Rezystora podciągającego nie trzeba stosować, gdy mikrokontroler sterowany jest z wyjścia cyfrowego.

Czy warto stosować rezystor podciągający, można przekonać się, sprawdzając, co mikrokontroler „widzi" z rezystorem podciągającym, a co bez niego. Wykorzystam do tego celu program odczytujący wartość z pinu 5. i zamieniający ją na wartość cyfrową:

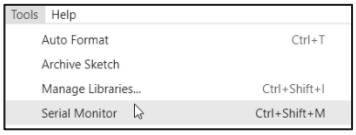
```
int inputPin = 5;

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

void loop() {
   int reading = digitalRead(inputPin);
```

```
    Serial.println(reading);
    delay(1000);
}
```

Wejściem jest pin o numerze 5. Co sekundę (1000 ms) odczytana wartość jest przesyłana do monitora szeregowego.

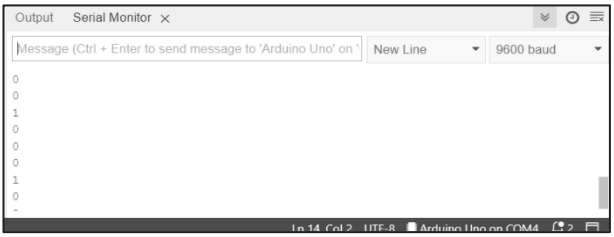
Weryfikuję program i przesyłam do Arduino UNO. Świecą się diody: ON, L. Miga dioda RX.

Aby dowiedzieć się, co program odczytuje, uruchamiam monitor szeregowy (rysunek 3.3).



Rysunek 3.3. Polecenia, które muszę wybrać, aby uruchomić monitor szeregowy

W dolnej części ekranu wyświetlone zostało okno *Serial Monitor* (rysunek 3.4). Żadne wejście Arduino UNO nie jest podłączone. Monitor wyświetla to, co Arduino UNO „łapie" z otoczenia.

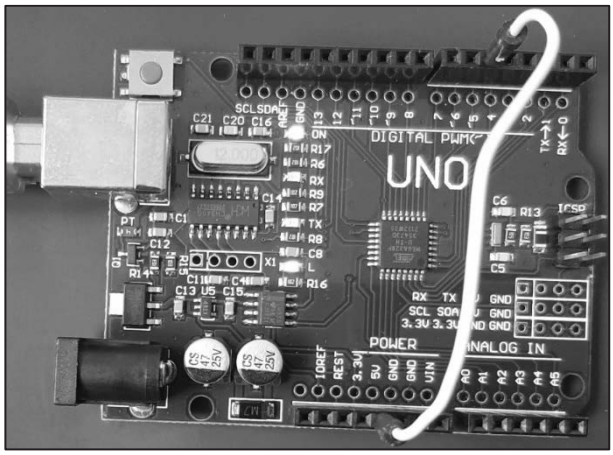


Rysunek 3.4. Zewnętrzne pole elektromagnetyczne powoduje indukowanie napięć rozpoznawanych błędnie jako stan wysoki (wtedy mikrokontroler odczytuje, że na wejściu 5. jest wartość 1)

Niepodłączone wejście działa jak antena.

Podłączenie wejścia 5. do 5 V (rysunek 3.5) powoduje, że monitor szeregowy prawidłowo odczytuje stan wejścia jako wysoki (rysunek 3.6).

Czy można spowodować, aby wejście mikrokontrolera było mniej czułe na zakłócenia? Tak. Wystarczy w programie dodać polecenie aktywujące rezystor podciągający:



Rysunek 3.5. Na wejście 5. znajdujące się na listwie górnej podane zostało napięcie 5 V z listwy dolnej

Rysunek 3.6. Arduino prawidłowo interpretuje 5 V jako stan wysoki

```
int inputPin = 5;

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

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

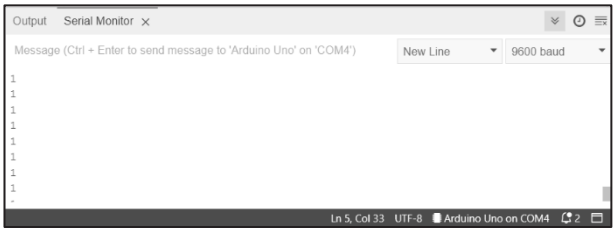
Zmiana polega na zastąpieniu polecenia

```
pinMode(inputPin, INPUT);
```

poleceniem

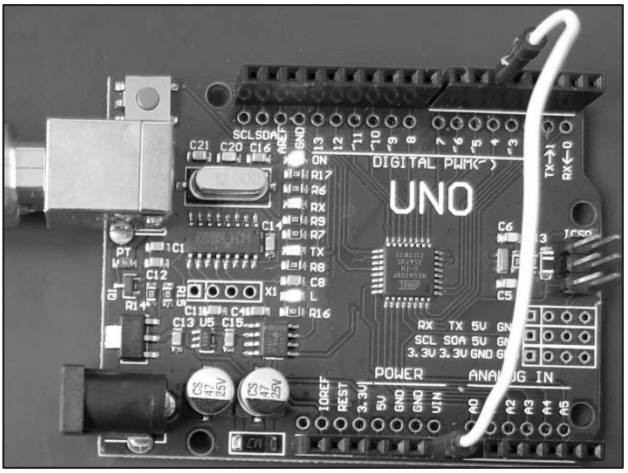
```
pinMode(inputPin, INPUT_PULLUP);
```

Weryfikuję i wysyłam program do Arduino UNO. Odłączam przewód podający napięcie na wejście 5. Włączam i obserwuję monitor szeregowy (rysunek 3.7).



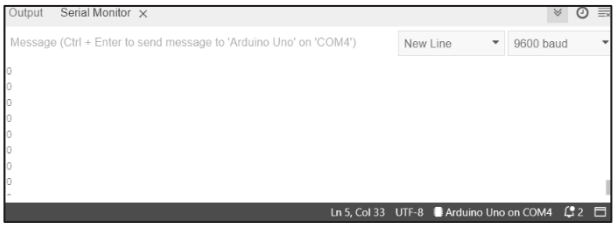
Rysunek 3.7. Mimo że wejście 5. płytki nie jest podłączone, stan jest rozpoznawany jako logiczne 1

A może polecenie `INPUT_PULLUP` zepsuło płytkę i teraz będzie odczytywała tylko stan wysoki? Podłączam wejście 5. do masy (rysunek 3.8).



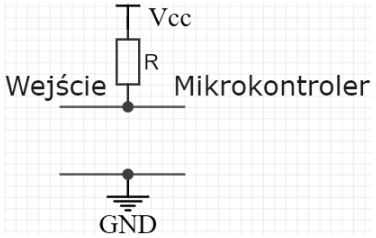
Rysunek 3.8. Na wejście 5. podany został stan niski (GND)

Monitor szeregowy pokazuje ciąg zer (rysunek 3.9).



Rysunek 3.9. Monitor szeregowy odczytał wartości

Na rysunku 3.10 pokazałem, jak działa rezystor podciągający R. Jeżeli R nie jest włączony, **wejście** podłączone jest bezpośrednio do **mikrokontrolera**. Kawałek ścieżki działa jak antena.



Rysunek 3.10. Rezystor podciągający podnosi stan napięcia na niepodłączonym wejściu do logicznego 1, ale nie przeszkadza w ustawieniu go w stan 0

Gdy rezystor podciągający R jest włączony, a na wejściu nie ma sygnału, do mikrokontrolera podawane jest logiczne 1 (Vcc za pośrednictwem R).

Podanie na wejście logicznego 1 powoduje, że mikrokontroler odczytuje 1.

Podanie na wejście logicznego 0 powoduje, że rezystor R podłączony jest pomiędzy Vcc i masę. Powstaje na nim spadek napięcia (Vcc nie jest zwierane do GND, bo oddziela je rezystor), a mikrokontroler odczytuje 0.

ARDUINO OD PODSTAW

Płytka Arduino to platforma pozwalająca na realizację milionów interesujących projektów, między innymi z zakresu robotyki, automatyzacji, ale można z niej korzystać wszędzie tam, gdzie potrzebny jest solidny mikrokontroler o dużych możliwościach i ograniczonym zapotrzebowaniu na zasoby. Przez lata Arduino doczekało się licznych wyspecjalizowanych wariantów, a także rozsianej po całym świecie wielomilionowej społeczności użytkowników. Czas do niej dołączyć!

Arduino od podstaw to praktyczny przewodnik adresowany do wszystkich, którzy chcą się zapoznać z możliwościami urządzenia — od tych, którzy dotąd nie mieli styczności z komputerami jednopłytkowymi, po tych bardziej zaawansowanych. Począwszy od podstaw, jak również przedstawienia możliwości i potencjalnych zastosowań, książka wprowadza w konkretne zagadnienia, w tym programowanie kontrolera. W przystępny sposób wyjaśnia konstrukcję Arduino i działanie zintegrowanego środowiska programistycznego, pozwala także na stworzenie swoich pierwszych projektów.

Dzięki książce poznasz:

- **dostępne warianty sprzętowe**
- **możliwości poszczególnych modeli**
- **tajniki budowy i architektury**
- **sposoby na efektywne programowanie**
- **podstawy składni używanych języków programowania**
- **standardowe biblioteki**
- **sposoby tworzenia własnych bibliotek**
- **metody zapisu danych**
- **zasady obsługi operacji wejścia i wyjścia**

Zacznij tworzyć z Arduino!

Helion

helion.pl

HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-283-9715-6

9 788328 397156

Cena: 49,90 zł