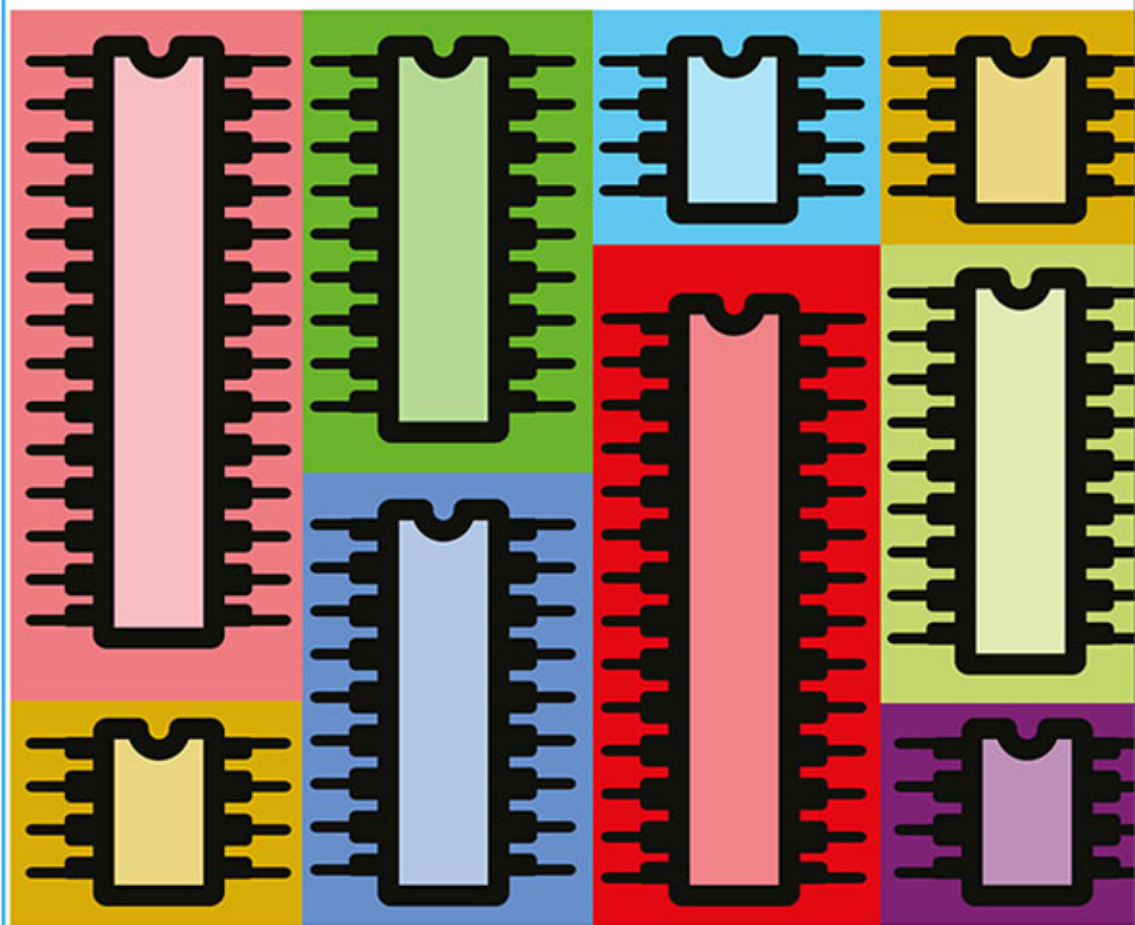


Programowanie układów AVR dla praktyków



Wykorzystaj potencjał układów AVR!

Elliot Williams

Tytuł oryginału: Make: AVR Programming

Tłumaczenie: Wojciech Moch

ISBN: 978-83-246-9501-0

© 2014 Helion S.A.

Authorized Polish translation of the English edition of Make: AVR Programming,
ISBN 9781449355784 © 2014 Elliot Williams, published by Maker Media Inc.

This translation is published and sold by permission of O'Reilly Media, Inc.,
which owns or controls all rights to sell the same.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means,
electronic or mechanical, including photocopying, recording or by any information storage retrieval system,
without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej
publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną,
fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje
naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich
właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były
kompletne i rzetelne. Nie bierze jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za
związane z tym ewentualne naruszenie praw patentowych lub autorskich. Wydawnictwo HELION nie
ponosi również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji
zawartych w książce.

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

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

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

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Spis treści

Wstęp	11
--------------------	-----------

Część I Podstawy

1. Wprowadzenie	19
Czym jest mikrokontroler? Obraz ogólny	19
Komputer w jednym układzie... ..	19
...naprawdę mały komputer	20
Co mogą mikrokontrolery?	21
Sprzęt: obraz ogólny	21
Rdzeń: procesor, pamięć i układy wejścia-wyjścia	23
Układy peryferyjne: ułatwianie sobie życia	25
2. Programowanie układów AVR	29
Programowanie układu AVR	29
Łańcuch narzędzi	29
Narzędzia programowe	32
Konfiguracja w systemie Linux	33
Konfiguracja w systemie Windows	33
Konfiguracja w systemie Mac	34
Konfiguracja modułu Arduino	34
Program make i pliki makefile	34
AVR i Arduino	35
Zalety platformy Arduino	35
Wady platformy Arduino	35

Arduino: sprzęt czy oprogramowanie? I to, i to!	37
Arduino to AVR	37
Arduino to programator układów AVR	40
Inne programatory sprzętowe	42
Moje ulubione programatory	42
Zaczynamy: błyskające diody LED	43
Podłączenia	44
Złącza ISP	46
Program AVRDUDE	48
Konfigurowanie plików makefile	51
Pamięć Flash	53
Rozwiązywanie problemów	53
3. Wyjścia cyfrowe	55
blinkLED wersja 2.0	56
Struktura kodu języka C dla układu AVR	57
Rejestry sprzętowe	57
Podsumowanie programu blinkLED	60
Zabawka świetlna	61
Budowanie układu	61
Ładne wzorki: kod zabawki	65
Eksperymentuj!	67
4. Manipulacje na bitach	69
Praca z kodem: oczy Cylonów	70
Manipulowanie bitami a oczy Cylonów	71
Przesuwanie bitów	71
Zaawansowane manipulowanie bitami: coś więcej niż oczy Cylonów	74
Włączanie bitów operatorem OR	76
Przełączanie bitów operatorem XOR	78
Wyłączanie bitów operatorami AND i NOT	79
Popisy	81
Podsumowanie	83
5. Szeregowe wejście i wyjście	85
Komunikacja szeregową	85
Implementowanie komunikacji szeregowej w układzie AVR: projekt pętli zwrotnej	88
Konfigurowanie: układ AVR	88
Konfigurowanie: komputer	90
Konfigurowanie: adapter USB-port szeregowy	90
Sprawdzanie całości: testowanie pętli zwrotnej	91
Usuwanie problemów z połączeniami szeregowymi	94

Konfigurowanie modułu USART: szczegóły	94
Organy z układu AVR	99
Muzyka z mikrokontrolera	101
Biblioteka muzyczna	103
Kod	104
Superdodatki	106
Podsumowanie	108
6. Wejścia cyfrowe	109
Przyciski, przełączniki itd.	109
Konfigurowanie wejść: rejestry DDR, PORT i PIN	111
Interpretowanie naciśnień przycisków	113
Zmiana stanu	115
Eliminowanie odbić	116
Przykład kodu obsługującego odbicia	118
Pozytywka	120
Kod programu	120
Przycisk na szefa	122
Skrypty dla komputera stacjonarnego	123
Rozszerzenia	126
7. Konwerter analogowo-cyfrowy — część 1.	129
Przegląd elementów układu AVR	130
Miernik światła	133
Schemat	133
Kod programu	137
Inicjacja konwertera AC	140
Rozszerzenia	141
Powolny oscyloskop	141
Kod dla układu AVR	143
Kod dla komputera	145
Podobieństwa	147
Oświetlenie nocne i multiplexer	147
Multiplexer	147
Konfigurowanie bitów multiplexera	148
Schemat	149
Kod	150
Podsumowanie	151

Część II Średnio zaawansowana

8. Przerwania sprzętowe	155
Przerwania zewnętrzne: przykłady użycia przycisku	157
Przykład z zewnętrznym przerwaniem INTO	158
Przykład przerwania wywołanego zmianą stanu pinu	163
Czujnik pojemnościowy	165
Czujnik	167
Kod programu	169
Zmienne ulotne i globalne	171
Debugowanie układu	173
9. Wprowadzenie do sprzętowego zegara/licznika	175
Zegary i liczniki — do czego mają służyć?	175
Sprawdź swój refleks	177
Użycie zegara numer 0 do poprawienia 8-bitowych organów	182
Radio AM	185
Schemat	187
Szybkość procesora	187
Radio AM: kod programu	191
Podsumowanie	195
10. Modułacja szerokości impulsu	197
Jasne i ciemne diody: technika PWM	198
Siłowe rozwiązanie PWM	200
Modulowanie impulsów za pomocą liczników	201
Inicjowanie liczników dla trybu PWM	203
Modulowanie szerokości impulsu na dowolnym pinie	206
Przykład modulacji impulsu na dowolnym pinie	206
Zakończenie: inne możliwości dla PWM i listy kontrolne liczników	208
11. Sterowanie serwomotorami	213
Serwomotory	214
Sekretne życie serwomotorów	215
Schemat	216
Kod programu	217
Zegar słoneczny z serwomotorem	220
Budowanie zegara	221
Przygotuj lasery!	223
Kod programu	225
Kalibracja serwomotoru zegara słonecznego	232

12. Konwerter analogowo-cyfrowy — część 2.	237
Woltomierz	238
Schemat	239
Kod programu	242
Wykrywacz kroków	246
Schemat	246
Teoria	251
Wykładniczo ważona średnia krocząca	252
Kod programu	255
Podsumowanie	258

Część III Tematy zaawansowane

13. Zaawansowane sztuczki z PWM	263
Bezpośrednia synteza cyfrowa	264
Tworzenie fali sinusoidalnej	267
Następny krok: miksowanie i głośność	269
Miksowanie	271
Dynamiczna kontrola głośności	273
Odpytywanie portu USART	276
Obwiednia ADSR	276
Pliki uzupełniające	277
14. Przetłaczalniki	279
Sterowanie dużymi prądami: przetłaczalniki	280
Tranzystory bipolarne	281
Tranzystory polowe (MOSFET)	283
Polowe tranzystory mocy	284
Przełączniki	285
Triaki i przełączniki statyczne	286
Przetłaczalniki: podsumowanie	287
Silniki prądu stałego	288
15. Zaawansowane sterowanie silnikami	295
Cofanie: mostki typu H	296
Program: zakreślmy sobie mostkiem	299
Mostek H tylko dla ekspertów	301
Mostek H i modulacja szerokości impulsu	302
Tryb napędu znak-moduł	303
Tryb napędu blokada-antyfaza	304
Porównanie trybów napędu	305
Silniki krokowe	307

Rodzaje silników krokowych	308
Pełne kroki i półkroki	308
Identyfikowanie przewodów silnika krokowego	311
Zbyt wiele przewodów!	312
Podwójny mostek H: układ SN754410	312
Kod programu	315
Kontrola przyspieszeń	318
Mikrokroki	320
16. SPI	325
Jak działa protokół SPI?	326
Przykład wymiany bitów	328
Rejestry przesuwające	328
Zewnętrzna pamięć EEPROM	331
Pamięć zewnętrzna	332
Połączenia elektryczne przykładu z protokołem SPI	334
Kod programu demonstracyjnego	335
Plik nagłówkowy biblioteki	337
Kod biblioteki obsługującej pamięć EEPROM SPI	339
Funkcja initSPI()	341
Funkcja SPI_tradeByte()	342
Funkcje pomocnicze	343
Podsumowanie	344
17. I2C	347
Jak działa protokół I2C?	348
Połączenia w przykładowym projekcie	352
Biblioteka obsługi magistrali I2C	353
Termometr z interfejsem I2C	356
Protokołowanie danych z użyciem protokołów I2C i SPI	359
Wskaźniki w pamięci EEPROM	363
Menu tworzone przez port szeregowy	363
Pętla główna termometru protokołującego	364
18. Używanie pamięci programu	367
Wykorzystanie pamięci programu	367
Adresy pamięci	368
Operator adresu: &	369
Wskaźniki	372
Wskaźniki w skrócie	372
Wskaźniki jak parametry funkcji	376

Podsumowanie	379
Opcjonalnie: dereferencje wskaźników	380
Gadający woltomierz	381
Struktury danych w pamięci programu i plik nagłówkowy	382
Odtwarzanie dźwięku i odczyt wartości napięcia: plik .c	386
Generowanie danych audio	391
Różnicowa modulacja kodowo-impulsowa	391
Kodowanie dźwięku metodą dwubitowego DPCM	391
Kodowanie DPCM: program wave2DPCM.py	394
19. EEPROM	399
Używanie pamięci EEPROM	400
Zapisywanie danych w pamięci	400
Odczytywanie danych z pamięci	405
Zapisywanie i odczytywanie pamięci EEPROM	407
Organizacja danych w pamięci EEPROM	408
Projekt: szyfrator kodu Vigenère'a	412
20. Wnioski, pożegnanie i zachęta	421
Układ AVR: brakujące rozdziały	421
Licznik watchdog	421
Oszczędzanie energii	422
Zewnętrzne oscylatory i inne źródła taktowania	422
Programy rozruchowe	422
Komparator analogowy	423
Debugowanie	423
Odtóż książkę i twórz!	423
Skorowidz	425

Modulacja szerokości impulsu

10

Przypaszanie diod i „analogowe” wyjście

Dotąd w naszych programach wszystko było tylko włączone albo wyłączone, czyli całkowicie cyfrowe. Diody były albo włączone, albo zgaszone. Membrana głośnika była albo w pełni wciągnięta, albo całkowicie wypchnięta. Takie przepychanie elektronów w tę lub tamtą stronę nie było zbyt finezyjne. Samo włączanie i wyłączanie może sprawiać frajdę, ale czasami, zamiast błyskać diodami, wolelibyśmy je powoli wygasić albo uruchomić silnik z połową prędkości. Można też pokusić się o wygenerowanie faktycznych fal dźwiękowych, a nie tylko fali prostokątnej, i dodatkowo zapewnić im regulację głośności.

Aby osiągnąć takie efekty, musimy znaleźć sposób generowania pośrednich wartości napięcia za pomocą jedynie logicznych sygnałów układu AVR. Jedną z najczęściej stosowanych metod jest *modulacja szerokości impulsu* (*pulse width modulation* — PWM). W skrócie polega ona na bardzo szybkim włączaniu i wyłączaniu wyjścia logicznego, na tyle szybkim, żeby element podłączony do tego wyjścia nie zdążył w pełni zareagować na zmianę. W efekcie odczuwalna wartość napięcia na tym elemencie jest proporcjonalna do średniego procenta czasu, w jakim wejście układu AVR było w stanie wysokim. (Na razie może to brzmieć bardzo dziwne, ale do końca rozdziału na pewno wszystko się wyjaśni).

W [rozdziale 13](#). wykorzystamy technikę PWM do odtwarzania dźwięków na syntezatorze. W [rozdziałach 14. i 15.](#) użyjemy jej do sterowania silnikiem z różnymi prędkościami, a nawet do poruszania nim w odwrotnym kierunku. I w końcu, w [rozdziale 18.](#), technika PWM pozwoli na zbudowanie gadającego woltomierza, który będzie podawał wartość napięcia *Twoimi słowami*. Jak widać, w kolejnych rozdziałach dość często będziemy korzystali z modulacji szerokości impulsu, dlatego proszę mi wybaczyć, że tutaj zajmujemy się prostym pulsowaniem diod LED.

Technika PWM jest tak powszechną metodą tworzenia analogowego napięcia przez cyfrowe urządzenia, że niemal wszystkie mikrokontrolery, w tym i układy AVR, mają dedykowane elementy peryferyjne, które zajmują się takim szybkim przełączaniem stanu wyjścia. Zdecydowanie zalecam korzystanie z tej możliwości. W tym rozdziale przyjrzymy się też całkowicie manualnej procedurze generowania modulowanych impulsów, która pozwoli lepiej poznać zasady tej techniki. Dodatkowo zaprezentuję też metodę wykorzystującą moduł zegara i licznika w połączeniu z przerwaniami, która pozwala na wygenerowanie modulowanych impulsów na dowolnym wyjściu mikrokontrolera. Przejdźmy zatem do rzeczy.

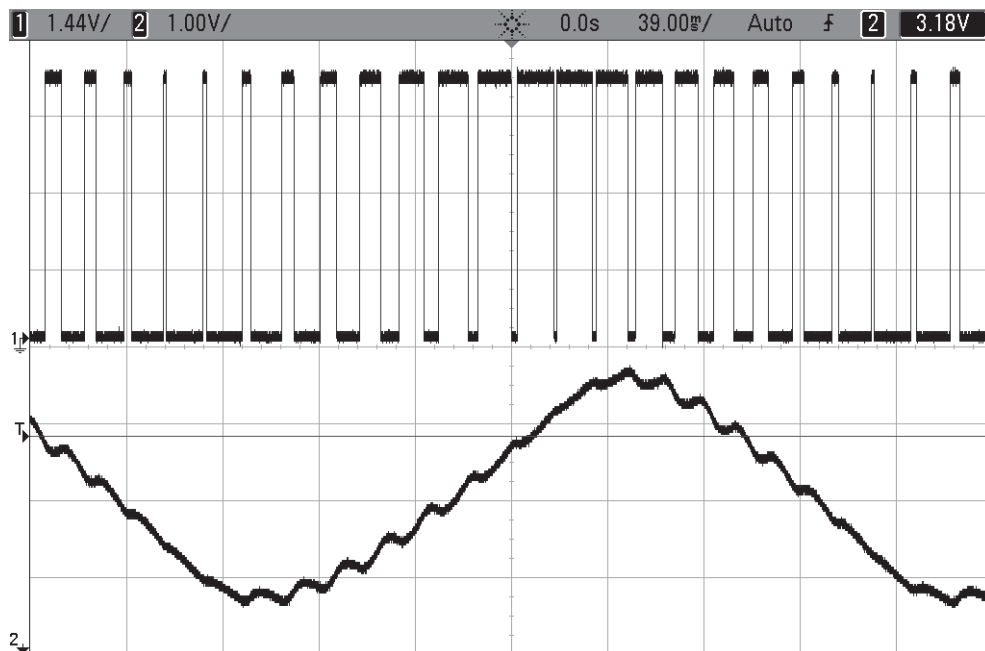
Czego potrzebujesz?

Oprócz zestawu podstawowego będziemy potrzebować:

- diod LED wraz z opornikami ograniczającymi, których używaliśmy w poprzednich rozdziałach,
- adaptera USB-port szeregowy.

Jasne i ciemne diody: technika PWM

PWM jest chyba najłatwiejszą do zaimplementowania techniką pozwalającą na uzyskanie analogowego efektu częściowego włączenia. Jak wspominałem wcześniej, musimy na tyle szybko przełączać wyjście cyfrowe między stanem wysokim i niskim, żeby podłączone urządzenie nie mogło odpowiednio zareagować na pojedyncze impulsy i w związku z tym otrzymywało tylko *średnią wartość* napięcia. Możemy wtedy ciągle powtarzać takie impulsy, zmieniając wartość średnią napięcia przez dopasowywanie procenta czasu, w którym wyjście było w stanie wysokim. Efekt takich manipulacji można zobaczyć na [rysunku 10.1](#).



Rysunek 10.1. Wykresy oscyloskopowe techniki PWM

Na [rysunku 10.1](#) górny ślad pochodzi bezpośrednio z wyjścia układu AVR. Jak widać, procent czasu, w jakim wyjście utrzymywane jest w stanie wysokim, zmienia się od niskiego do wysokiego, później znowu wraca do niskiego. Na dolnym wykresie widać wynik przesłania tych impulsów do filtra składającego się z opornika i kondensatora, podobnego do tego, z którego będziemy korzystać w [rozdziale 13](#). Taki filtr ma długi czas reakcji, dzięki czemu modulowane długości impulsów przekładają się na uśrednioną, analogową wartość napięcia.

Gdy modulowany sygnał większość czasu spędza na napięciu 5 V, wartość analogowego napięcia jest wyższa i vice versa. Oczywiście sygnał modulowanych impulsów jest całkowicie cyfrowy — może być jedynie wysoki lub niski — ale wynikowe (filtrowane), uśrednione napięcie może przyjmować dowolną wartość pomiędzy tymi dwoma stanami.

By użyć techniki PWM, musimy zdefiniować dwa ważne parametry. Procent czasu, jaki w cyklu spędzimy w stanie włączonym, nazywany jest *współczynnikiem wypełnienia* (*duty cycle*). Z kolei częstotliwość, z jaką powtarzamy pojedynczy cykl, nazywana jest częstotliwością PWM. Najczęściej regulować będziemy jednak współczynnik wypełnienia. Oznacza to, że musimy często zmieniać jego wartość, być może nawet w sposób ciągły, aby w efekcie otrzymać na wyjściu „analogowe” napięcie, takie jak na [rysunku 10.1](#).

W większości przypadków częstotliwość PWM wybiera się tylko raz, zaraz na początku pracy układu. Jak już mówiłem wcześniej, ważne jest, żeby ta częstotliwość była na tyle wysoka, by sterowany system nie miał czasu odpowiednio szybko zareagować na poszczególne impulsy i musiał pracować z ich uśrednioną wartością. O jakiej częstotliwości zatem mówimy? To zależy od konkretnego rozwiązania i tolerancji układu na *tętnienia*, czyli niewielkie pozostałości impulsów modulujących, które z pewnością przedostaną się przez filtr. Na [rysunku 10.1](#) wybrałem częstotliwość PWM na poziomie zbliżonym do częstotliwości fali wyjściowej, dlatego na wykresie sygnału wyjściowego widać całkiem sporo takich tętnień. Jeżeli częstotliwość PWM byłaby znacznie większa, tętnienia zmniejszyłyby się znacznie (ale nie dałoby się wtedy przygotować tak ładnego przykładu).

Częstotliwość PWM wymagana do uzyskania zakładanej wielkości tętnień zależy od tego, jak wolno dane urządzenie lub filtr reagują na zmiany sygnału wejściowego. Przykładowo silniki zwykle bardzo powoli dopasowują swoją prędkość, co wynika z ich dużej bezwładności ramienia, koła lub innego napędzanego przez nie elementu. Dla większości silników całkowicie wystarczające byłyby częstotliwości PWM na poziomie od 50 do 500 Hz. Mówię byłyby, ponieważ są to częstotliwości słyszalne dla człowieka, a większość ludzi woli jednak, gdy silniki nie „śpiewają”.

Wiele nowoczesnych układów sterowania silnikami wykorzystuje częstotliwości PWM o wartości lekko przekraczającej 20 kHz (to mniej więcej maksymalna częstotliwość, jaką może usłyszeć człowiek), dzięki czemu unika się drżeń wywoływanych w uzwojeniach. Od tej reguły istnieją też pewne wyjątki. Większość silników napędzających wagony metra zasilana jest za pomocą słyszalnych częstotliwości PWM. To dlatego doskonale słychać proces rozpędzania i hamowania silników. Po prostu budujący je inżynierowie bardziej koncentrowali się na wydajności tych układów, a mniejszą wagę przywiązywali do generowanych dźwięków.

Tworzenie sygnałów audio odpowiednich dla ludzkiego uszu, czym będziemy się zajmować w [rozdziale 13.](#), wymaga znacznie wyższych częstotliwości PWM. Potrzebna jest częstotliwość przynajmniej dwukrotnie większa od najwyższej częstotliwości, jaką chcemy odwzorować, a to oznacza, że do modulowania musimy użyć częstotliwości przynajmniej 40 kHz. (W naszym przykładzie z 8-bitowym sygnałem audio i wbudowanym zegarem procesora o wartości 8 MHz zostaniemy ograniczeni do 32,5 kHz, ale i to powinno wystarczyć).

Oko, w przeciwieństwie do ludzkiego ucha, możemy zaliczyć do elementów reagujących powoli. Widzieliśmy to już w przykładzie wykorzystującym bezwładność wzroku, w którym generowaliśmy impulsy o długości 2 milisekund (500 Hz). Okazuje się, że aby uniknąć migotania diod, wcale nie musimy używać aż tak wysokich częstotliwości. W moim przypadku okres o długości 18 milisekund (nieco poniżej 60 Hz) całkowicie wystarczał do wyeliminowania migotania. Przypominam, że mówię tu o czasie reakcji ludzkiego oka, a nie diod LED, które mogą się włączać i wyłączać z częstotliwością dochodzącą do megaherców, choć tego nie zauważymy. W tym przypadku ludzki system optyczny jest tym powolnym mechanizmem, którego użyjemy do uśredniania wartości impulsów.

Siłowe rozwiązanie PWM

Nie musisz mi wierzyć na słowo. Na [listingu 10.1](#) zapisałem w pełni konfigurowalną, manualną procedurę generowania modulowanych impulsów. Zapisz w mikrokontrolerze program [pwm.c](#) i sprawdź, gdzie leżą Twoje granice możliwości widzenia migotania. Wystarczy, że zmienisz wartość opóźnienia zdefiniowaną na początku programu. Potem przyjrzymy się pętli zdarzeń i wykonywanym przez nią działaniom.

Listing 10.1. Kod programu pwm.c

```
/* Prosty przykład modulacji impulsu */

// ----- Preambula ----- //
#include <avr/io.h> /* Definicje pinów, portów itp. */
#include <util/delay.h> /* Funkcje marnujące czas */
#include "pinDefines.h"

#define LED_DELAY 20 /* mikrosekundy */

void pwmAllPins(uint8_t brightness) {
    uint8_t i;
    LED_PORT = 0xff; /* włączenie */
    for (i = 0; i < 255; i++) {
        if (i >= brightness) { /* po odczekaniu dość długiego czasu */
            LED_PORT = 0; /* wyłączenie */
        }
        _delay_us(LED_DELAY);
    }
}

int main(void) {

    uint8_t brightness = 0;
    int8_t direction = 1;

    // ----- Inicjacja ----- //

    // Inicjowanie wszystkich diod
    LED_DDR = 0xff;
    // ----- Pętla zdarzeń ----- //
    while (1) {
        // Rozjaśnianie i przygaszanie
        if (brightness == 0) {
            direction = 1;
        }
        if (brightness == 255) {
            direction = -1;
        }
        brightness += direction;
        pwmAllPins(brightness);

        /* Koniec pętli zdarzeń */
    }
    return (0); /* ta instrukcja nie zostanie wykonana */
}
```

Powyższy kod został podzielony na dwie części. Funkcja `pwmAllPins()` zajmuje się implementacją modulowania impulsu, a pozostały kod w pętli zwiększa i zmniejsza wartość zmiennej `brightness`. Wywoływanie funkcji `pwmAllPins()` ze zmieniającą się wartością zmiennej `brightness` powoduje rozjaśnianie i wygaszanie diod LED.

W funkcji `pwmAllPins()` mamy pętlę sterowaną zmienną `i`, która wykonuje dokładnie 256 iteracji. Wewnątrz tej pętli porównujemy wartość zmiennej `i` z wartością parametru `brightness` i utrzymujemy diody w stanie włączonym, dopóki `i` jest mniejsze. W efekcie diody włączone na początku każdego zestawu 256 iteracji będą świeciły, aż pętla wykona `brightness` iteracji. Oznacza to, że dla większych wartości parametru diody będą zapalone przez dłuższą część każdego z tych cykli. Oto mamy modulację szerokości impulsu! (Pamiętaj jednak, że na potrzeby rozjaśniania i przyciemniania diod zużywamy całą moc obliczeniową procesora).

Dodatkowo możemy też modyfikować czas opóźnienia każdej iteracji i w ten sposób sprawdzić, w którym momencie zauważymy migotanie diod. (Jako zadanie domowe możesz podłączyć do układu potencjometr i regulować nim czas opóźnienia iteracji pętli). Dość szybko powinno Ci się udać ustalić minimalną częstotliwość, przy której nie widzisz jeszcze migotania. Zauważ, że poruszając głową, ponownie zobaczysz efekty związane z bezwładnością wzroku.

Sprawdzając różne wartości opóźnienia, pamiętaj, że w każdym cyklu wykonujemy 256 kroków, co w połączeniu z opóźnieniem 20 mikrosekund na iterację sprawia, że każdy cykl będzie trwał 5120 mikrosekund, czyli 5,12 milisekundy. W każdej sekundzie mieści się 1000 milisekund, a zatem nasza częstotliwość PWM wynosi $1000/5,12$ czyli 195 Hz.

Kod znajdujący się w funkcji `main()` ustala wartość zmiennej `direction` (liczba całkowita ze znakiem), która umożliwia zwiększanie i zmniejszanie aktualnej jasności diod. Gdy w końcu osiągnięta zostanie maksymalna lub minimalna jasność, następuje zmiana znaku wartości zmiennej `direction`. W efekcie otrzymujemy naprzemiennie rozjaśniające i przyciemniające się diody.

Przeglądając się efektom pracy tego programu, możesz zauważyć, że ludzkie oko nie reaguje jednakowo na stałe zmiany jasności. W naszym przykładzie diody sprawiają wrażenie, jakby szybciej przechodziły przez ciemniejszą część cyklu niż przez jego jaśniejszą część. To tylko pozory, ponieważ dobrze wiemy, że w każdym cyklu jasność zmienia się o 1 w każdą stronę.

Ludzkie oko znacznie lepiej radzi sobie z odróżnianiem niższych poziomów oświetlenia niż przy bardzo jasnym świetle. Oznacza to, że reakcje oka na światło nie są liniowe, co jest wykorzystywane w wielu zastosowaniach diod LED sterowanych modulowanymi impulsami. Jeżeli przesuwasz szybko głowę przed światłami sygnalizacyjnymi na skrzyżowaniu, zauważysz ich migotanie, wynika ono właśnie ze stosowania modulacji impulsów. Powodem tak częstego stosowania techniki PWM przy sterowaniu diod LED jest to, że człowiek nie zauważa różnicy między 100% a 90% współczynnikiem wypełnienia, co pozwala sterować lampami na skrzyżowaniu na poziomie 90% i oszczędzić tym samym 10% kosztów energii.

Modulowanie impulsów za pomocą liczników

Jeżeli nie wierzysz, że Twoje oczy nieliniowo reagują na światło, przeprowadźmy mały eksperyment. Na [listingu 10.2](#) program `pwmTimer.c` pobiera przez port szeregowy wpisywane przez Ciebie znaki i przekształca je w liczbowe współczynniki wypełnienia. Przy okazji poznasz znacznie lepszą metodą implementowania techniki PWM — przerzucenie szczegółów na sprzęt.

Zapisz program w mikrokontrolerze, a potem w oknie terminala wpisz wartość 10, 20 lub 30 i sprawdź, przy której z tych wartości diody będą świeciły najjaśniej. A teraz wpisz 210, 220 albo 230 i spróbuj jeszcze raz.

Przyznasz, że znacznie łatwiej odróżnić jasność ciemniejszych diod. Reakcja naszych oczu na zmiany jasności jest zbliżona do procentowej wartości takiej zmiany, a nie do bezwzględnej wartości natężenia światła. Warto o tym pamiętać, przygotowując oświetlenie diodowe — równe odstępy wartości modulacji impulsu nie przekładają się na równe odstępy jasności oświetlenia.

Możesz się zastanawiać, dlaczego zapalamy tutaj jedynie trzy diody. Przykrą przyczyną takiego ograniczenia jest fakt, że sprzęt generujący modulowane impulsy został ograniczony i jeden licznik może sterować co najwyżej dwoma pinami. W sumie daje to maksymalnie sześć pinów: cztery sterowane dwoma licznikami 8-bitowymi oraz dwa piny sterowanie licznikiem 16-bitowym. Nasz program zapala zatem jedynie trzy diody LED, wykorzystując do tego wartości wpisane na klawiaturze. Teraz przyjrzyjmy się zasadzie działania programu przedstawionego na [listingu 10.2](#).

Listing 10.2. Kod programu pwmTimers.c

```
/* Modulacja szerokości impulsu sterowana portem szeregowym */

// ----- Preambula ----- //
#include <avr/io.h>                                /* Definicje pinów, portów itp. */
#include <util/delay.h>                             /* Funkcje marnujące czas */
#include "pinDefines.h"
#include "USART.h"

static inline void initTimers(void) {
    // Licznik 1 A,B
    TCCR1A |= (1 << WGM10);                        /* Szybki tryb PWM, 8-bitów */
    TCCR1B |= (1 << WGM12);                        /* Szybki tryb PWM, pkt. 2 */
    TCCR1B |= (1 << CS11);                        /* Częstotliwość PWM = F_CPU/8/256 */
    TCCR1A |= (1 << COM1A1);                      /* Wyjście PWM na OCR1A */
    TCCR1A |= (1 << COM1B1);                      /* Wyjście PWM na OCR1B */

    // Licznik 2
    TCCR2A |= (1 << WGM20);                        /* Szybki tryb PWM */
    TCCR2A |= (1 << WGM21);                        /* Szybki tryb PWM, pkt. 2 */
    TCCR2B |= (1 << CS21);                        /* Częstotliwość PWM = F_CPU/8/256 */
    TCCR2A |= (1 << COM2A1);                      /* Wyjście PWM na OCR2A */
}

int main(void) {
    uint8_t brightness;

    // ----- Inicjacja ----- //

    initTimers();
    initUSART();
    printString("-- Przykład PWM z diodami LED --\r\n");

    /* włącza wyjścia diod przełączanych przez sprzęt do PWM */
    LED_DDR |= (1 << LED1);
    LED_DDR |= (1 << LED2);
    LED_DDR |= (1 << LED3);
}
```

```
// ----- Pętla zdarzeń ----- //
while (1) {

    printString("\r\nWprowadź współczynnik wypełnienia cyklu (0-255): ");
    brightness = getNumber();
    OCR2A = OCR1B;
    OCR1B = OCR1A;
    OCR1A = brightness;

}
return (0);
}
```

/ Koniec pętli zdarzeń */
/* ta instrukcja nie zostanie wykonana */*

Podobnie jak we wszystkich dotychczasowych programach używających liczników, pętla zdarzeń zawiera raczej niewiele kodu. Pobiera ona liczbę z portu szeregowego i przesuwa wartości pomiędzy rejestrami porównującymi, sterującymi diodami LED1, LED2 i LED3. Zauważ, że nie musimy korzystać z dodatkowych zmiennych do przechowywania „poprzednich” wartości otrzymanych z portu szeregowego, ponieważ są one przechowywane w rejestrach OCR n x, które traktujemy tu jak zmienne.

Przepisanie tych wartości zajmuje procesorowi kilka mikrosekund, nawet jeżeli działa z szybkością 1 MHz. Pozostałą część czasu procesor spędza na oczekiwaniu na dane z portu szeregowego i odpowiednim przetwarzaniu otrzymanych danych. Zauważ, że w wersji z programowym modulowaniem impulsów (plik [pwm.c](#)) w ogóle nie mielibyśmy takiej opcji. Gdybyśmy czekali na dane z portu szeregowego, diody przestawałyby błyskać. Jeśli nawet jedynie sprawdzalibyśmy, czy port szeregowy odebrał jakieś dane, to i tak powodowałoby to zmianę zależności czasowych modulowanych impulsów. Żeby teraz utrzymywać błyskanie diod, nasz kod musi tylko zapisywać właściwe wartości współczynnika wypełnienia do odpowiednich rejestrów OCR n x. Całą resztą prac, czyli odliczaniem, porównywaniem oraz włączaniem i wyłączaniem pinów, zajmują się elementy sprzętowe. W tym przykładzie zyskujemy zatem możliwość wykorzystania procesora do obsługi portu szeregowego.

Inicjowanie liczników dla trybu PWM

Konfigurowanie licznika na potrzeby modulacji impulsów jest bardzo podobne do sposobu ich użycia, jaki opisałem w [rozdziale 9.](#), z tym że zamiast trybu normalnego albo CTC, do generowania przebiegu wybieramy tryb PWM.

Przyjrzyjmy się teraz dokładniej funkcji `initTimers()`, żeby zobaczyć kod przypominający ten, którego używaliśmy już w [rozdziale 9.](#) Konfigurujemy tryb generowania przebiegów, ustalamy wartość podzielnika zegara i definiujemy piny wyjściowe, żeby sygnały skierować bezpośrednio do pinów OCR1A, OCR1B i OCR2A. I to wszystko.

Jeszcze raz musimy zajrzeć do arkusza danych, aby upewnić się, że wiemy przynajmniej, skąd wzięły się nazwy bitów ustawianych w funkcji inicjującej. Zauważ, że licznik numer 1 jest bardziej złożony, ponieważ jest licznikiem 16-bitowym, a układ AVR daje możliwość działania w trybie 8-, 10- i 16-bitowym. Pełną rozdzielczość licznika wykorzystamy w [rozdziale 11.](#) do sterowania serwowmotorem, ale na razie wybieramy tylko 8-bitową rozdzielczość licznika, aby uzyskać zgodność działań z licznikiem numer 2.



Szybki tryb PWM

Szybki tryb PWM jest chyba najczęściej używanym trybem tej kategorii, a jednocześnie jest zdecydowanie najprostszy w użyciu. Licznik odlicza w pętli od zera do wartości maksymalnej (w zależności od licznika i trybu będzie to 255, 65535 albo inna wartość zapisana w rejestrze OCRxA) z szybkością definiowaną przez podzielnik sygnału zegara. Przy okazji wartość licznika porównywana jest z zawartością rejestru porównującego OCRnx. Jeżeli obie wartości są takie same, wyjście PWM może zostać wyczyszczone albo ustawione, a dodatkowo można wywołać odpowiednie przerwanie.

Szybki tryb PWM to sprzętowa wersja programu *pwm.c*, w której przechodziliśmy w pętli przez wartości od 0 do 255 i porównywaliśmy wartość licznika z wartością zmiennej współczynnika wypełnienia. Jednak użycie sprzętu do modulowania szerokości impulsu jest znacznie szybszym rozwiązaniem niż wykonywanie tych wszystkich operacji programowo, a dodatkowo zupełnie nie obciąża procesora.

Wyliczanie częstotliwości PWM w trybie szybkim jest naprawdę proste. Wystarczy podzielić częstotliwość zegara procesora przez wartość podzielnika, a potem podzielić ją jeszcze przez liczbę kroków wykonywanych w cyklu. Przykładowo przy podzielniku ustawionym na 8, przy użyciu 8-bitowego licznika z 256 krokami i procesora o szybkości 1 MHz uzyskamy częstotliwość PWM: $1\,000\,000\text{ Hz} / 8 / 256 = 488\text{ Hz}$.

Tutaj inaczej niż w trybie CTC wygląda też sposób konfigurowania wyjścia za pomocą bitów COM. Po pierwsze, poszczególne bity mają inne znaczenie niż w trybie CTC, szybkim PWM i trybie PWM z korekcją fazy, dlatego upewnij się, że ustawienia odczytujesz z właściwej tabeli. W naszym przypadku ustawienie bitów COM1A1, COM1B1 i COM2A1 odpowiadało włączeniu trybu „nieodwracającego”, w którym układ PWM włącza pin w momencie przepelnienia licznika, a wyłącza go po osiągnięciu wartości porównywanej. Dokładnie tak samo postępowaliśmy w wersji w pełni programowej na początku tego rozdziału. Po podaniu wyższej wartości porównującej otrzymywaliśmy większy współczynnik wypełnienia i jaśniejsze diody LED.

Konfigurowanie podzielnika zegara wygląda dokładnie tak samo jak w przykładach z trybem CTC, dlatego w tym kodzie nie znajdziemy już nic interesującego. Przy ustalaniu wartości podzielnika dla trybu CTC najbardziej interesowała nas uzyskana częstotliwość, ale tym razem nie ma ona większego znaczenia. W trybie PWM chodzi o uzyskanie na tyle szybko powtarzających się cykli, żeby układ odbierający mógł reagować jedynie na ich uśrednioną wartość. Wypróbuj różne wartości podzielnika, żeby samodzielnie się przekonać, w jakim zakresie częstotliwości najłatwiej można sterować diodami LED. Możesz też użyć rozwiązania inżynierskiego opisanego w tym rozdziale, w ramce „Szybki tryb PWM”, które pozwala na wyliczenie właściwej częstotliwości trybu PWM.

Zanim opuścimy sekcję inicjacji, muszę wspomnieć, że używamy tu jeszcze rejestru DDR, żeby włączyć wyjścia na trzy diody LED. Do tej pory chyba każdemu udało się zauważyć, że po naciśnięciu klawisza *Enter* albo wpisaniu wartości 0 nawet najciemniejsza dioda nie jest całkowicie wyłączona. Dlaczego tak się dzieje? Wpisanie wartości 0 do rejestru OCR nie powoduje wyłączenia pinu sterującego diodą. Pin jest włączany za każdym razem, gdy licznik sprzętowy przepelnia się z wartości 255 do 0. Następnie pin jest wyłączany, w momencie gdy jednostka porównująca zauważy równość licznika z wartością rejestru OCR. To wszystko sprawia, że pin będzie w stanie włączonym przez 1/256 część cyklu (0,4 %), co przekłada się na niewielką, ale widoczną jasność diody.

Co można zrobić, żeby naprawdę wyłączyć diodę? Najprostszym sposobem jest odłączenie jej od zasilania przez przełączenie pinu w tryb wejścia, czyli wyzerowanie odpowiadającego mu bitu w rejestrze DDR. Czasami jednak lepszym wyjściem jest przestawienie pinu w trwały stan niski (uziemiaenie go), co sprawdza się np. przy wyłączaniu tranzystora podłączonego do tego pinu. Aby na stałe uzyskać logiczne niskie napięcie, musimy wyzerować bit COM w rejestrze konfiguracji licznika, co spowoduje odłączenie go od źródła taktowania, upewnić się, że pin jest skonfigurowany jako wyjście w rejestrze DDR, a w rejestrze PORT odpowiadający mu pin ma wartość 0.

Warto też wiedzieć, że takich zabiegów nie trzeba wykonywać, jeśli chcemy na stałe włączyć diodę. Wpisanie wartości 255 do rejestru OCR [sprawia](#), że dioda będzie zawsze włączona. Jedynie całkowite wyłączenie danego pinu wymaga pominięcia modułu sprzętowego generatora PWM.

Po zapoznaniu się z tym przykładem każdy powinien już skonfigurować i wykorzystać tryb PWM w swoich programach. Warto zauważyć, że po skonfigurowaniu tego trybu nasz kod musi już naprawdę niewiele robić, żeby zmienić średnią wartość generowaną przez modulowane impulsy. Wystarczy, że do rejestru porównującego wpisujemy nową wartość, a sprzęt zajmie się całą resztą. To wszystko sprawia, że używanie trybu PWM jest niezwykle proste. Jeżeli chcesz uzyskać napięcie dokładnie w połowie między 5 V i 0 V, wpisujesz do rejestru OCR1A wartość **127**. To wszystko! Wygląda to tak, jakby nasz kod miał możliwość bezpośredniego definiowania „analogowej” wartości napięcia.

„Analogowe” wyjścia modułów Arduino

Osoby używające modułów Arduino mogą się zastanawiać, dlaczego po prostu nie użyjemy pinów wyjść analogowych. Przyczyna jest prozaiczna: takie wyjścia *nie istnieją!* Otwierając tę książkę, przełknęliście czerwoną pigułkę, a teraz powoli sprawdzacie, jak głęboka jest królicza norka. Twórcy platformy Arduino was okłamują. To wszystko to wielka konspiracja, która ma przed wami ukryć fakt istnienia techniki PWM. Teraz znacie już prawdę!

A mówiąc serio, platforma Arduino stara się przestonąć wszystkie drobne szczegóły działania mikrokontrolerów, dlatego jej twórcy nie zaprzatają sobie głowy różnicami między wyjściem „analogowym” a wyjściem PWM. Nie zmuszają nas też do zapamiętywania, które piny łączą się z danymi licznikami. Przyjrzyj się funkcji `analogWrite()` znajdującej się w kodzie źródłowym platformy Arduino w pliku [wiring_analog.c](#), który z kolei umieszczony jest w katalogu [arduino-1.0/hardware/arduino/cores/arduino](#) (dopasuj numer do swojej wersji środowiska). Przekonasz się, że w gruncie rzeczy funkcja ta robi dokładnie to samo.

Funkcja `analogWrite()` najpierw sprawdza, czy zapisywana wartość jest równa 0 lub 255, co odpowiada stałemu wyłączeniu i włączeniu pinu. Następnie odsu-

kuje właściwy licznik (`digitalPinToTimer(pin)`), który trzeba skonfigurować dla wybranego pinu, a potem ustawia odpowiednio bity konfiguracyjne COM i na koniec zapisuje wartość do rejestru OCR. Dokładnie to samo robiliśmy w naszych programach! (Wersja z biblioteki Arduino przy każdej zmianie współczynnika wypełnienia ustawia też bity COM, co jest operacją nadmiarową, ale w ten sposób mamy pewność, że są one właściwie skonfigurowane).

Jednak na platformie Arduino płacimy za taką wygodę. Operacje, które w naszym programie zajmują od dwóch do trzech cykli procesora, w wersji Arduino wymagają przynajmniej 50 cykli. Najpierw sprawdzenie, czy chcemy całkowicie włączyć albo wyłączyć pin, potem użycie instrukcji `switch` i przeszukiwanie pamięci, żeby wybrać rejestry właściwego licznika, a wszystko po to, żebyśmy nie musieli przeglądać arkusza danych mikrokontrolera.

Jeżeli w swoim programie rzadko korzystasz z funkcji `analogWrite()`, nie będzie to miało większego wpływu na wydajność. Jeśli jednak często zmieniasz wartości bitów OCR, tak jak będziemy to robić w [rozdziale 13.](#), ten dodatkowy kod może sprawić,

„Analogowe” wyjścia modułów Arduino — ciąg dalszy

że określone zadanie stanie się niewykonalne. Ktoś, kto używał wcześniej platformy Arduino, może narzekać, że teraz musi poznać tyle szczegółów na temat mikrokontrolera, ale gdy już opanuje sposoby używania poszczególnych urządzeń peryferyjnych, okaże się,

że otwierają się całkiem nowe możliwości. A gdy już się wie, jak działają poszczególne części mikrokontrolera, ich właściwe skonfigurowanie okazuje się całkiem proste.

Modulowanie szerokości impulsu na dowolnym pinie

Poznaliśmy już dwie metody implementowania techniki PWM w kodzie dla układu AVR. Jedna z nich polegała na całkowicie programowym tworzeniu pętli i bezpośrednim przełączaniu stanu wybranego pinu. Druga, sprzętowa metoda działała zdecydowanie szybciej, ale pozwalała wykorzystać jedynie sześć pinów układu, po dwa na każdy z liczników.

Gdybyśmy chcieli zastosować technikę PWM na dowolnym pinie, *można* użyć sztuczki, która jest nieco niestandardowa. Zamiast przełączać stan pinu za pomocą wbudowanych elementów samych liczników, możemy wykorzystać przerwania do wywoływania własnego kodu i w nim włączać oraz wyłączać wybrany pin. Nie musimy wiązać procesora zadaniem odliczania i oczekiwania, tak jak robiliśmy w pełni programowym wariantcie. Tym razem odliczaniem zajmie się moduł licznika działający w trybie normalnym.

Przerwania wywoływane na początku każdego cyklu pozwalają włączać piny, natomiast przerwania wynikające z działania rejestrów porównujących umożliwiają ich wyłączenie. Oznacza to, że metoda ta jest swego rodzaju hybrydą metody całkowicie programowej i całkowicie sprzętowej. Dzięki zastosowaniu przerwań i liczników zmniejsza się obciążenie procesora, ale obsługa przerwań wymaga choć *trochę* czasu procesora, więc całość nie jest aż tak szybka i stabilna jak rozwiązanie w pełni sprzętowe.

Ze względu na to, że przełączaniem stanu wybranych pinów zajmują się procedury obsługi przerwań, musimy mieć pewność, że parametry modulacji będą na tyle duże, by każda z tych procedur mogła zostać do końca wykonana. Wyobraź sobie, co się stanie, jeżeli wybierzemy współczynnik wypełnienia o wielkości 6, a dzielnik zegara przełączymy w najszybszy tryb działania. W takiej sytuacji mamy zaledwie 6 cykli na zakończenie obsługi przerwania włączającego pin na początku każdego cyklu, co jest nierealne, ponieważ większość przerwań wymaga przynajmniej 10 cykli na przełączenie kontekstu. (Można, co prawda, postarać się obejść takie ograniczenie, ale w wielu przypadkach jest to po prostu nieopłacalne).

Do prawidłowego działania metody modulowania szerokości impulsu na dowolnym pinie musimy zatem przypisać dzielnikowi wartość przynajmniej 64. Wtedy będziemy mieli dość czasu na obsługę przerwań i wszystko będzie działało bez problemów.

Przykład modulacji impulsu na dowolnym pinie

Podsumowując, mogę stwierdzić, że modulowanie impulsów na dowolnym pinie realizowane jest przez wybranie trybu normalnego konkretnego licznika, odliczanie od 0 do 255 i takie skonfigurowanie go, żeby przy każdym cyklu wywoływał przerwania. Pierwsze przerwanie wywoływane jest przy przepelnieniu licznika, gdy wraca on do początkowej wartości 0. To w tym przerwaniu włączamy wybrany pin. Drugie przerwanie wywoływane jest przez rejestr porównujący i zajmuje się wyłączeniem pinu, w momencie gdy wartość licznika

zrówna się z wartością rejestru. W ten sposób większe wartości wpisywane do rejestru OCR powodować będą, że pin będzie włączony przez większą część cyklu. Na tym polega modulowanie szerokości impulsu! Kod całego rozwiązania przedstawiam na [listingu 10.3](#).

Listing 10.3. Kod programu pwmOnAnyPin.c

```
// Szybki przykład modulowania impulsu na dowolnym pinie z wykorzystaniem przerwań
// ----- Preambula ----- //
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>
#include "pinDefines.h"

#define DELAY 3

volatile uint8_t brightnessA;
volatile uint8_t brightnessB;

// ----- Funkcje ----- //
static inline void initTimer0(void) {
    /* musi mieć wartość /64 ze względu na zależności czasowe */
    TCCR0B |= (1 << CS01) | (1 << CS00);
    /* oba przerwania porównywania wyjścia */
    TIMSK0 |= ((1 << OCIE0A) | (1 << OCIE1B));
    TIMSK0 |= (1 << TOIE0);
    /* włączenie przerwania przepelnienia */
    sei();
}

ISR(TIMERO_OVF_vect) {
    LED_PORT = 0xff;
    OCR0A = brightnessA;
    OCR0B = brightnessB;
}

ISR(TIMERO_COMP_A_vect) {
    LED_PORT &= 0b11110000;
    /* wyłączyć dolne cztery diody */
}

ISR(TIMERO_COMP_B_vect) {
    LED_PORT &= 0b00001111;
    /* wyłączyć górne cztery diody */
}

int main(void) {
    // ----- Inicjacja ----- //

    uint8_t i;
    LED_DDR = 0xff;
    initTimer0();

    // ----- Pętla zdarzeń ----- //
    while (1) {

        for (i = 0; i < 255; i++) {
            _delay_ms(DELAY);
        }
    }
}
```

```

brightnessA = i;
    brightnessB = 255 - i;
}

for (i = 254; i > 0; i--) {
    _delay_ms(DELAY);
    brightnessA = i;
    brightnessB = 255 - i;
}

}
return (0);
}

```

/ Koniec pętli zdarzeń */*
/ ta instrukcja nie zostanie wykonana */*

Tutaj również muszę wyjaśnić kilka szczegółów. Po pierwsze zauważ, że tym razem zdefiniowałem dwie zmienne globalne — `brightnessA` i `brightnessB` — które na początku każdego cyklu ładowane są do rejestrów porównujących. Dlaczego nie możemy wpisywać tych wartości bezpośrednio do rejestrów `OCROA` i `OCROB`? Po prostu w zależności od momentu, w którym wartość zostanie zapisana do tego rejestru, może się zdarzyć, że modulacja impulsu przeskoczy o jeden cykl. Rozwiązaniem tego problemu jest wpisywanie nowej wartości do rejestrów w ściśle określonym momencie, czyli w przerwaniu wywoływanym przepelnieniem licznika. Obie zmienne globalne mogą być w dowolnej chwili modyfikowane w funkcji `main()`, ale ich wartości zostaną przepisane do rejestrów `OCRO` dopiero po wywołaniu przerwania przepelnienia. Są zatem najprostszym buforem.

Wywoływanie trzech różnych przerw z jednego licznika jest zadziwiająco proste. W kodzie funkcji `init` ↪ `Timer0()` można zobaczyć, że wymaga to ustawienia zaledwie trzech bitów w rejestrze maski przerw `TIMSK`. Jak już wspominałem, źródło taktowania licznika musi być znacznie wolniejsze od zegara taktującego pracę procesora, bo wtedy umożliwiałaby prawidłową pracę procedur obsługi przerw. Z tego powodu wybrałem dla podzielnika wartość 64. Na koniec, choć ten punkt dotyczy wszystkich przypadków użycia przerw, nie można zapomnieć o włączeniu całego systemu przerw za pomocą instrukcji `sei()`.

Pozostała część kodu jest już bardzo prosta. Procedury obsługi przerw powinny być jak najkrótsze, ponieważ będą wywoływane naprawdę często. W tym przypadku przerwanie porównania składają się z pojedynczej instrukcji. I w końcu, w funkcji `main()` zademonstrowany został sposób użycia tej metody modulowania impulsu na dowolnym pinie. Pozwala ona na ustalenie współczynnika wypełnienia w dowolnym momencie pracy programu, a licznik i przerwanie zajmą się już resztą.

Zakończenie: inne możliwości dla PWM i listy kontrolne liczników

W tym rozdziale zaprezentowałem trzy metody generowania „analogowego” wyjścia realizowanego przez szybkie przełączanie stanu cyfrowego pinu. Na początku całość zaprogramowaliśmy samodzielnie, do przygotowania wyjścia z modulowaną szerokością impulsu wykorzystując całą moc obliczeniową procesora. Następnie wszystkie związane z tymi pracami zadania przerzuciliśmy na moduł zegara i licznika. To zdecydowanie

najczęściej stosowana i najwszechstronniejsza metoda tworzenia modulowanego wyjścia, choć ograniczona jest do sterowania jedynie dwoma pinami na licznik. Pamiętaj o tych ograniczeniach podczas projektowania własnych układów, a sprzętowa funkcja generowania PWM nigdy Cię nie zawiedzie.

Na koniec zademonstrowałem specjalną metodę pozwalającą na modulowanie impulsów na dowolnym pinie, wykorzystującą moduł zegara i licznika w połączeniu z automatycznie wywoływanymi przerwaniem, które zajmowały się przełączaniem stanu pinów. W porównaniu do metody całkowicie sprzętowej, wymaga to niewielkiego nakładu pracy procesora i redukuje maksymalną szybkość modulacji ze względu na czas potrzebny na obsługę przerwania. Poza tym, niezbędne jest zastosowanie zmiennych globalnych, aby uniknąć przeskoków przy zmianie wartości rejestrów porównujących. Z drugiej strony, jeżeli *naprawdę* musisz *modulować* impulsy na niestandardowych pinach i masz niewykorzystany licznik, to jak najbardziej możesz przełączyć go w tryb normalny i pozwolić na wywoływanie przerwania.

Tymi trzema opcjami opisanymi rozdziale praktycznie wyczerpałem temat modulowania szerokości impulsu. Na zakończenie chciałbym wspomnieć o innych metodach tworzenia analogowych wyjść z układu AVR. Jeżeli potrzebujesz większej wydajności zarówno pod względem redukcji cyfrowych szumów, jak i większej częstotliwości wyjścia, musisz znać inne rozwiązania.

Konwertery cyfrowo-analogowe (CA).

Prosty podzielnik napięcia

Jeżeli są Ci potrzebne tylko cztery dyskretne wartości napięcia, możesz połączyć dwa końce podzielnika napięcia zbudowanego z oporników do dwóch pinów wyjściowych układu AVR. Gdy oba piny będą miały stan wysoki, na wyjściu pojawi się napięcie zasilające. Gdy oba piny przełączymy w stan niski, na wyjściu odczytamy 0 V. Gdy natomiast jeden z pinów będzie w stanie wysokim, a drugi w niskim, napięcie na wyjściu podzielnika znajdzie się pomiędzy tymi wartościami granicznymi.

Jeżeli w takim podzielniku napięcia jeden opornik będzie miał wartość dwukrotnie większą od drugiego, uzyskamy podział w stosunku 1/3. Włączenie jednego pinu przy wylączonym drugim pozwoli uzyskać na wyjściu napięcie $1/3 \times 5\text{ V}$ albo (w odwrotnej konfiguracji) $2/3 \times 5\text{ V}$.

Jeżeli potrzebujesz tylko jednej pośredniej wartości napięcia, ale musi być ona bardzo dokładna, możesz odpowiednio dopasować wartości oporników składających się na podzielnik. Opcjonalnie możesz też wykorzystać potencjometr, co pozwoli na dopasowywanie napięcia wyjściowego.

Samodzielnie złożony konwerter CA typu R-2R

Rozwijając pomysł z prostym podzielnikiem napięcia, możesz zbudować wielowejściowy, zagnieżdżony podzielnik napięcia wykorzystujący znacznie więcej oporników. Powstanie wtedy cała drabinka podzielników, która umożliwia wpisanie binarnej wartości na wyjścia układu AVR w celu uzyskania dopasowanego do tej wartości napięcia. Sztuczka polega na tym, żeby odpowiednio dobrać wartości oporników.

Przykładowo budując 8-bitowy konwerter tego rodzaju, musimy podłączyć wszystkie 8 pinów portu B do oporników o wartości 2R, te połączyć ze sobą opornikami o wartości R, a opornikiem o wartości 2R połączyć najmniej znaczący bit z uziemieniem. (W sieci WWW na pewno łatwo znajdziesz schemat połączeń tego konwertera). W efekcie każdy pin układu AVR będzie wpływał na wyjściowe napięcie układu zgodnie ze swoją wagą w bajcie, dzięki czemu będziemy mogli po prostu wpisać wartość napięcia do portu mikrokontrolera. Przykładowo wpisanie do portu wartości 63 spowoduje wygenerowanie napięcia $5\text{ V}/4$, przy napięciu zasilającym wynoszącym 5 V.

Zbudowanie konwertera typu R-2R jest dość proste, a przy zastosowaniu oporników o tolerancji 1% można uzyskać całkiem niezłą dokładność odwzorowania napięć. Jeszcze lepsze wyniki uzyskasz, stosując oporniki o jednakowej wartości, ale układając po dwa równolegle w celu uzyskania wartości R.

Jeśli wyszukasz w sieci schematy takiego konwertera, nie zapomnij, że jego wyjście zazwyczaj wymaga zastosowania wzmacniacza, który wyeliminuje ryzyko przeciążenia wyjścia sieci oporników.

Kolejną zaletą konwertera typu R–2R (obok jego prostoty) jest to, że poszczególne bity są obciążane równolegle podczas generowania napięcia wyjściowego. Oznacza to, że zmiany napięcia są równie szybkie, co operacje zapisywania nowych wartości do portu mikrokontrolera. W ten sposób możemy generować napięcia o częstotliwości dochodzącej do megaherców. W miejscach zmiany wartości napięcia nie powstają też tętnienia, ponieważ przejścia pomiędzy poziomami są zwykle łagodne. Wadą tego rozwiązania jest to, że wymaga zastosowania ośmiu pinów, których można by użyć do innych celów.

Zewnętrzne układy konwerterów CA

Samodzielne przygotowanie konwertera R–2R dla rozdzielczości 8 bitów nie powinno sprawiać trudności, jednak już przy 10 bitach trzeba bardzo dokładnie wybierać oporniki, a uzyskanie więcej niż 12 bitów jest praktycznie niemożliwe. Aby zatem generować wysokiej jakości sygnał audio, musimy wykorzystać gotowe układy konwerterów CA. Niektóre z nich to starannie przygotowane drabinki oporników R–2R, ale większość zawiera złożone cyfrowe układy logiczne działające z bardzo wysoką częstotliwością.

Dzięki temu, że układy te są względnie często używane w konsumenckich produktach audio, z pewnością łatwo znajdziesz konwerter CA doskonale dopasowany do generowania sygnałów stereo (16- lub 24-bitowy i 44,1 lub 96 kHz) kosztujący zaledwie kilka złotych. Takie nowoczesne konwertery najczęściej pobierają dane poprzez szeregowe magistrale SPI lub I2C, o których napiszę w [rozdziałach 16. i 17.](#)

Poszukując właściwego układu CA, obok rozdzielczości mierzonej w bitach, trzeba brać pod uwagę pożądaną szybkość konwersji oraz dokładność odwzorowania napięcia. Jeżeli chcesz mieć układ o absolutnie najlepszych parametrach, jego koszt może się okazać astronomiczny! Po prostu konwertery przystosowane do generowania sygnałów audio nie mają idealnej czułości i odwzorowania, a ich maksymalna częstotliwość pracy nie jest trudna do osiągnięcia, co przekłada się na względnie niską cenę. Jeśli jednak potrzebujesz konwertera pracującego z precyzją do mikrowolta albo działającego z częstotliwością mierzoną w megahercach, musisz głębiej sięgnąć do kieszeni.

Na zakończenie ostatnich dwóch rozdziałów, w których koncentrowaliśmy się na różnym wykorzystaniu modułów zegara i licznika, przygotowałem listę kontrolną wszystkich poleceń konfiguracyjnych niezbędnych do ich prawidłowego użycia. Nie ma ona zastępować sekcji „Register Description” z arkusza danych mikrokontrolera, ale jej celem jest bezpieczne poprowadzenie Cię przez cały system.

Oto lista kontrolna konfiguracji liczników.

1. Najpierw zdecyduj, którego licznika chcesz użyć. Zwykle będzie to zależało od potrzebnej Ci rozdzielczości licznika. Jeżeli całkowicie wystarczy Ci rozdzielczość 8 bitów, wybierz licznik numer 0 lub 2. Jeżeli niezbędna będzie rozdzielczość 16 bitów albo licznik ten jest po prostu nieużywany, wybierz licznik numer 1.
2. Następnie zdecyduj, w jakim trybie ma pracować licznik: ustaw odpowiednio bity `WGMn0` i `WGMn1` w rejestrze `TCCRnA` oraz bit `WGMn2` w rejestrze `TCCRnB`. W arkuszu danych mikrokontrolera przejrzyj tabelę „Waveform Generation Mode Bit Description”.
 - a. Zliczasz zdarzenia albo odliczasz czas? Potrzebujesz trybu normalnego. (Nie trzeba tu ustawiać bitów konfiguracyjnych).
 - b. Używasz licznika jak generatora podstawy czasu albo częstotliwości? Najłatwiej będzie to zrobić w trybie CTC. Ustaw bit `WGMn1` w rejestrze `TCCRnA`.
 - c. Używasz licznika do modulowania szerokości impulsu? Zazwyczaj najlepiej będzie użyć trybu PWM. Jeżeli nie musisz regulować częstotliwości cyklu, ustaw bity `WGMn0` i `WGMn1` w rejestrze `TCCRnA`, a w przeciwnym przypadku ustaw bit `WGMn2` w rejestrze `TCCRnB`.

3. Chcesz skierować wyjście licznika bezpośrednio na pin układu? Ustaw bity COMxA i COMxB w rejestrze TCCxA.
4. Oblicz, jaką wartość podzielnika zegara ustawić w bitach CS_nx w rejestrze TCCR_nB.
5. Jeżeli używasz wartości porównującej, dobrze jest wpisać jej domyślną wartość do rejestrów OCR_nA oraz OCR_nB. Jeżeli chcesz, żeby na pinach pojawiały się impulsy modulowane licznikiem w trybie PWM, nie zapomnij o odpowiednim skonfigurowaniu rejestru DDR.
6. Czy w połączeniu z licznikiem używasz też przerwań?
 - a. W trybie normalnym włącz przerwanie przepelnienia licznika, ustawiając bit TOIE_n w rejestrze TIMSK_n.
 - b. W trybach PWM lub CTC włącz przerwania wartości porównywanych, ustawiając bity OCIE_nA i OCIE_nB w rejestrze TIMSK_n.
 - c. Nie zapomnij włączyć całego systemu przerwań za pomocą instrukcji sei (). Pamiętaj o przygotowaniu procedur obsługi przerwań.

Skorowidz

A

adaptery programatorów, 47
ADC, Analog to Digital Converter, 129
adresy pamięci, 368
ADSR, 274, 276
akumulator, 265
amplituda podtrzymania, sustain, 274
analogowe wyjścia modułów, 206
arkusz danych, 23
atak, attack, 273
automatyzacja kompilacji, 31

B

bezpiecznik, 188
bezpośrednia synteza cyfrowa, 264, 267
bezwładność wzroku, 61
biblioteka
 25LC256, 337
 avr/power.h, 422
 pgmspace.h, 371
bieg jałowy, 305

bit, 69
 EESAVE, 411
 TWEA, 353
 TWSTA, 353
 TWSTO, 353
bitowe operatory logiczne, 75
bity
 bezpiecznika, 189
 multipleksera, 148
 przełączanie, 78
 przesuwanie, 71
 włączanie, 76
 wyłączanie, 79
 wymiana, 328
blinkerlights, 62
błędy programu AVRDUDE, 49

C

cofanie, 296
CPU, 23
czas
 opadania, decay, 273
 życia pamięci, 401
czasomierze, 26

częstotliwość
 cykli, 178
 impulsów sterujących, 219
 nośna, 195
 próbki, 269
czujnik, 130, 136
 drgań, 255
 LM75, 350
pojemnościowy, 165, 167

D

DDS, Direct Digital Synthesis, 264
deasemblacja, 181
debugowanie, 32, 173, 273, 423
dereferencje wskaźników, 380
detektor wibracji, 251
dioda LED, 44
dioda zabezpieczająca, 291
DPCM, Differential Pulse-Code Modulation, 368
 dwubitowe, 391
 jednobitowe, 391
dynamiczna
 kontrola głośności, 273
 obwiednia dźwięku, 274

dysk piezoelektryczny, 246
działanie
konwertera AC, 132
protokołu I2C, 348

E

edytor
gedit, 30
Notepad++, 30
Programmer's Notepad, 30
TextMate, 30
EEPROM, 23, 332, 364, 399–420
efekt odbicia przycisku, 117
eliminowanie odbić, 116
EMF, Electro-Motive Force, 290

F

fala sinusoidalna, 267
faza, 343
filtr dolnoprzepustowy, 270
Flash, 23, 53, 332
format Intel hex dump, 407
fotorezystor, 134–137
funkcja

_delay_ms(), 159, 195
createHeader(), 397
debounce(), 119
decodeVigenere(), 420
displayCodes(), 418
eeprom_block_update(), 402
eeprom_read_block(), 406,
407
eeprom_update_byte(), 400
eeprom_update_word(), 402
eeprom_write_byte(), 400
EEPROM_writeByte(), 344
encodeVigenere(), 420
enterText(), 415, 418
getDifferences(), 397
i2cWaitForComplete(), 353,
356
initI2C(), 353
initInterrupt0(), 159
initSPI(), 341

initTimer0(), 192
initTimers(), 203
ISR(), 160
main(), 82, 225, 390
packTwoBitDPCM(), 397
pgm_read_byte(), 370, 377,
380
playNote(), 176, 182
POVDisplay(), 65
printFromEEPROM(), 414, 416
printString(), 368, 410, 414
pwmAllPins(), 201
quantize(), 397
return(), 58
scaleData(), 397
setTime(), 235
sizeof(), 379
SPI_tradeByte(), 342
takeSteps(), 317, 320
trapezoidMove(), 319
unpackMono(), 397
funkcje
pomocnicze, 343
w języku C, 67

G

generowanie danych audio, 391
głośność, 269, 273
gniazdo ZIF, 47
granice mikrokroków, 323

I

IDE, 31, 37
IDE Arduino, 37–39
identyfikowanie przewodów
silnika krokowego, 311
inicjowanie
konwertera AC, 140
liczników, 203
pamięci EEPROM, 410
instalowanie języka Python, 124
instrukcja #define, 72
instrukcje przesuwające, 72

interfejs
I2C, 356
SPI, 31

J

język
C, 38, 57
Python, 124

K

kable DAPA, 42
kadowanie danych, 326
kalibracja serwomotoru, 232
kalkulator bitów
bezpiecznikowych, 190
karty pamięci SD, 333
kasowanie pamięci, 411
kod programu, *Patrz* program
kodowanie
DPCM, 393, 394
dźwięku, 391
kody błędów I2C, 356
komparator analogowy, 423
kompilacja, 32
kompilator
avr-gcc, 31
GCC, 31, 113
komunikacja
szeregowa, 25
z czujnikiem, 350
konfiguracja
bitów multiplexera, 148
liczników, 203, 210
modułu Arduino, 34
plików makefile, 51
programu AVRDUDE, 51
środowiska IDE, 38
w systemie Linux, 33
w systemie Mac, 34
w systemie Windows, 33
wejść, 111
wyjścia, 60
konstruowanie blinklightów, 62

- kontrola
 - głośności, 273
 - przyspieszeń, 318
- konwerter
 - AC, 25, 129, 142, 237
 - CA, 130, 209
 - CA typu R–2R, 209
- kroki, 308, 310
- kwalifikator EEMEM, 409

L

- laserowy zegar słoneczny, 222
- lewe ukośniki, 371
- liczby
 - całkowite, 269
 - zmiennoprzecinkowe, 401
- licznik, 26
 - sprzętowy, 175
 - watchdog, 421
- linia
 - MISO, 327
 - MOSI, 327
- listy kontrolne liczników, 208

Ł

- łańcuch narzędzi, 29

M

- magistrala SPI, 327
- makra kompilatora GCC, 113
- makro
 - _BV(), 75
 - CURRENT_LOCATION_POINT
ER, 363
 - PROGMEM, 370, 372
 - SLAVE_SELECT, 339
 - SLAVE_SELECT i
SLAVE_DESELECT, 339
- manipulowanie bitami, 69, 71, 74, 84
- martwa strefa, dead band, 215
- maska
 - bitowa, bitmask, 77
 - pinów, 163

- metoda DPCM, 391
- metody debugowania, 273
- miernik czasu reakcji, 177
- mikrokontroler, 19, 21
 - ATmega168, 26
 - ATmega328P, 38
 - AVR, 33
- mikrokroki, 320
- miksowanie sygnałów, 269
- modulacja
 - amplitudy, 185–187
 - impulsów, 201
 - kodowo-impulsowa, 391
 - szerokości impulsu, 197, 206, 302
- moduł Arduino, 32, 34
 - pin, 37
 - wady, 35
 - zalety, 35
- moduł TWI, 353
- moduły w języku Python, 236
- MOSFET, 283
- mostek H, 295–302
- mostek H podwójny, 312
- multiplekser, 147

N

- nadpróbkowanie, 237, 245
- napięcie wsteczne, 290
- narzędzia programowe, 32
- nastawianie napięć, biasing, 237
- natężenie światła, 134
- nazwy pinów, 37
- nieulotne parametry
 - konfiguracyjne, 364
- numer przerwań, 164
- numeracja, 73

O

- obsługa
 - magistrali I2C, 353
 - odbić, 118
 - przerwania, 170, 173, 389
 - zewnętrznej pamięci, 325

- obwiednia
 - ADSR, 276
 - dźwięku, 274
- odbicia, 119
- odczyt
 - nieistniejącego wskaźnika, 419
 - wartości napięcia, 386
 - pamięci EEPROM, 405, 407
- odpytywanie portu, 276
- odtworzenie dźwięków, 265, 386
- okablowanie modułu Arduino, 40
- opcje
 - podzielnika zegara, 140
 - programu AVRDUDE, 49
- operator
 - =, 115
 - adresu, 369
 - AND, 79, 80, 113
 - NOT, 79
 - OR, 76, 78
 - XOR, 78
- operatory logiki bitowej, 76
- opornik, 241
 - podciągający, 110, 335
 - podciągający wewnętrzny, 111
- oprogramowanie ładujące, 30
- organizacja danych, 408
- oscylatory, 422
- oszczędzanie energii, 422

P

- pakiet
 - Atmel Studio, 31
 - Eclipse, 31
 - WinAVR, 30
- pamięci ulotne, 332
- pamięć
 - EEPROM, 23, 332, 364, 399–420
 - EEPROM SPI, 339
 - Flash, 23, 53, 332
 - RAM, 23
- pętla
 - for, 83, 172
 - while, 58, 74
 - zdarzeń, 139

- piezoelementy, 247
- pin
 - RESET, 64, 335
 - SPEAKER, 183
- piny
 - mikrokontrolera, 37
 - programowania układu, 44
 - układu AVR, 22, 164
 - wyjściowe, 342
 - zasilania, 136
- pisanie kodu, 32
- planowanie, 32
- platforma Arduino, 35
- plik
 - _servoClickFunctions.c, 230
 - _servoSerialHelpers.c, 231
 - 25LC256.c, 339
 - 25LC256.h, 337
 - allDigits.h, 384
 - avr/sleep.h, 244
 - blinkLED.c, 57
 - fatSaw.h, 272
 - i2c.h, 354
 - makefile, 31, 34, 42, 51
 - portpins.h, 38, 39
 - quickDemo.c, 402
 - servoSundial.h, 228
 - talkingVoltmeter.h, 382
 - vigenereCipher.c, 413
- pliki
 - .hex, 408
 - .wav, 397
 - nagłówkowe, 382
- podłączenie
 - przycisku, 110, 111
 - serwomotoru, 216
- podzielnik
 - napięcia, 133–138, 209, 239
 - zegara, 185
- polaryzacja, 343
- polecenie
 - acr-objdump, 181
 - dump eeprom, 403
 - make flash, 42, 54
 - make main, 34
 - make size, 373
 - sizeof(), 379
- polowe tranzystory mocy, 284
- połączenia cewek w silniku, 312
- pomiar czasu, 166
- port
 - równoległy, 42
 - szeregowy, 231, 363
 - USART, 276
- potencjometr, 137, 138
- półkroki, 308, 310
- procedura obsługi przerwania, 25, 173, 389
- procesor, 23
- profil szybkości, 320
- program
 - adsr.c, 274
 - amRadio.c, 193
 - AVRDUDE, 38, 48, 189, 403–407
 - avrMusicBox.c, 120
 - blinkLED, 56, 60
 - bossButton.c, 125
 - bossButton.py, 124
 - calibrateTime.py, 233
 - capSense.c, 169
 - cylonEyes.c, 70
 - dcMotorWorkout.c, 291
 - dds.c, 267
 - debouncer.c, 118
 - eememDemo.c, 409
 - fatSaw.c, 271
 - favoriteColor.c, 406
 - footstepDetector.c, 255
 - generateScale.py, 278
 - generateWavetables.py, 277
 - hBridgeWorkout.c, 300
 - helloInterrupt.c, 158
 - i2cThermometer.c, 357
 - lightSensor.c, 138
 - loggingThermometer.c, 359
 - make, 34
 - nightLight.c, 150
 - povToy.c, 66
 - progmemDemo1.c, 370
 - progmemDemo2.c, 374
 - progmemDemo3.c, 376
 - progmemDemo4.c, 378
 - pwm.c, 200
 - pwmOnAnyPin.c, 207
 - pwmTimers.c, 202
 - reactionTimer.c, 177, 178
 - serialScope.py, 146, 173
 - servoSundial.c, 226
 - servoWorkout.c, 217
 - showingOffBits.c, 81
 - simpleButton.c, 114
 - slowScope.c, 143
 - spiEEPROMDemo.c, 336
 - stepperWorkout.c, 315, 319
 - talkingVoltmeter.c, 387
 - timerAudio.c, 182
 - toggleButton.c, 116
 - vigenereCipher_outline.c, 415
 - voltmeter.c, 243
 - wave2DPCM.py, 394
- programator
 - Atmel AVRISP mkII, 42
 - LadyAda's, 43
 - USBasp, 42
 - USBTiny, 42, 47
 - USBTinyISP, 43
- programatory
 - pamięci Flash, 30, 31
 - układów AVR, 40
 - sprzętowe, 42
- programowanie układu AVR, 29
- programy rozruchowe, 422
- projekt
 - czujnik pojemnościowy, 165
 - gadający woltomierz, 381
 - miernik czasu reakcji, 177
 - miernik światła, 133
 - oczy Cylonów, 70
 - oświetlenie nocne, 147
 - popisy, 81
 - powolny oscyloskop, 141
 - pozytywka, 120
 - protokołujący termometr cyfrowy, 352

- przycisk na szefa, 122
- przygaszanie diod, 198
- radio AM, 185
- szyfrator kodu Vigenère'a, 412
- syntezator dźwięków, 263
- woltomierz, 238
- wykrywacz kroków, 246
- zabawka świetlna, 61
- zegar słoneczny, 220
- protokołowanie danych, 359
- protokół
 - I2C, 347, 351, 359
 - SPI, 325, 342, 359
- próbki fali sinusoidalnej, 266
- przebieg czasowy transmisji, 349
- przechowywanie danych, 367
- przekazniki, 285
- przekazniki statyczne, 286
- przelączanie bitów, 78
- przetłączniki, 279, 287
 - dolne, 281
 - górne, 281
- przepalanie bezpieczników, 189, 190
- przepełnienie licznika, 276
- przerwania, 25, 162
 - sprzętowe, 155
 - wywoływane wewnętrznie, 156
 - wywoływane zewnętrznie, 157
 - zmiany stanu pinów, 164, 165
- przerwanie INTO, 163
- przesuwanie bitów, 71
- przewody magistrali I2C, 352
- przycisk, 110, 157
- przyspieszenie, 318
- PWM, pulse width modulation, 197, 200

R

- RAM, 23
- rdzeń, 23
- rejestr
 - ADMUX, 148
 - DDR, 111
 - DDRx, 59

- OCRn, 177
- OSCCAL, 422
- PIND, 112
- PINx, 59
- PORTD, 112
- PORTx, 59
- TWAR, 353
- TWCR, 356
- rejstry
 - przesuwające, 328–330
 - sprzętowe, 57, 59
 - wejścia-wyjścia, 22
- rodzaje
 - dysków piezoelektrycznych, 248
 - pamięci, 23
 - silników krokowych, 308
- rodzina ATmegaxx8, 27
- równanie średniej kroczącej, 254

S

- samokalibracja, 251
- samorozładowanie, 365
- schemat
 - mostka H, 297
 - programatora AVR, 45, 46
 - układu z czujnikiem pojemnościowym, 168
 - układu z piezoelementem, 249
 - zabawki świetlnej, 63
- serwomotory, 213, 214
- silnik
 - krokowy, 307, 309, 314
 - pozycjonujący, 214
 - prądu stałego, 288, 307
 - unipolarny, 313
 - z przekładniami, 294
- skalowanie napięcia, 241
- skrypty, 123
- słowo kluczowe volatile, 171
- specyfikacja silnika, 293
- SPI, Serial Peripheral Interface, 31, 326
- sprawdzanie stanu bitów, 113
- stany napędu, 303

- sterowanie
 - prądami, 280
 - serwomotorami, 213
 - silnikami, 295
 - szybkością procesora, 191
- struktury danych, 382
- sygnał audio, 391
- symbole tranzystorów polowych, 283
- synteza cyfrowa, 264, 267
- system PCINT, 158
- szerokość impulsu, 206
- szybki tryb PWM, 204, 219
- szybkość
 - magistrali I2C, 355
 - pracy silnika, 290
 - pracy zegara, 180, 190
 - procesora, 187, 191
 - wybrzmiewania, release rate, 274
- szyfr Vigenère'a, 412
- szyfrator, 412

Ś

- średnia
 - krocząca, 237
 - krocząca wykładniczo ważona, 252
 - wartość napięcia, 198
- środowisko IDE Arduino, 31, 38

T

- tablice, 278, 375
- tablice wskaźników, 384, 419
- technika
 - DPCM, 368
 - PWM, 198
- termometr
 - protokołujący, 352, 364
 - z interfejsem, 356
- testowanie, 32
- testowanie wartości bitów, 113
- transmitter częstotliwości radiowych, 185

- transoptory, 143
- tranzystory, 64
 - bipolarne, 281
 - Darlingtona, 282
 - mocy, 284
 - polowe, 64, 283
- triaki, 286
- tryb
 - CTC, 178, 317
 - napędu blokada-antyfaza, 304
 - napędu znak-moduł, 303
 - PWM, 203, 204
 - uśpienia, 244
 - wyłączenia zasilania, 246
- tryby
 - danych, 343
 - generatora prostych
 - przebiegów, 178
 - pracy zegara, 178
 - uśpienia, 245
- tworzenie
 - fali sinusoidalnej, 267
 - menu, 363
- typ
 - char, 405
 - int, 405
 - long, 405
 - long long, 405
 - word, 405
- typy danych, 404

U

- układ
 - ATtiny44, 27
 - ATtiny45, 26
 - blokady-antyfazy, 304
 - nocnego oświetlenia, 150

- SN754410, 312
- wykrywacza kroków, 249
- układy
 - mostków H, 306, 307
 - peryferyjne, 25
- urządzenie
 - główne, master, 327
 - podległe, slave, 327
- używanie
 - pamięci EEPROM, 400
 - pamięci EEPROM SPI, 335
 - pamięci programu, 367

W

- wady platformy Arduino, 35
- wejścia, 24
- wejścia cyfrowe, 109
- wektory przerwań, 162
- wirtualny rejestr, 139
- włączanie bitów, 76, 78
- wskaźnik, pointer, 363, 370, 375
 - jako parametr, 376
 - na typ, 373, 379
- wskaźniki laserowe, 224
- wybieranie
 - opornika, 284
 - szybkości zegara, 190
- wyjścia, 24
 - analogowe, 205
 - cyfrowe, 55
- wykorzystanie pamięci programu, 367
- wykrywacz kroków, 250
- wyłączanie bitów, 79
- wymiana bitów, 328
- wyznaczanie bitów
 - bezpiecznikowych, 190

Z

- zabawka świetlna, 63
- zalety platformy Arduino, 35
- zaokrąglanie liczb, 254
- zapis w pamięci, 404
- zapisywanie
 - danych, 400
 - pamięci EEPROM, 407
 - programów, 32, 39, 41
- zasilanie cewek, 311
- zastosowania rejestrów
 - przesuwających, 330
- zegar, 24, 229
 - słoneczny, 221
 - sprzętowy, 175
 - typu watchdog, 36
- zewnętrzne przerwanie INT0, 158
- ZIF, Zero Insertion Force, 47
- złącza
 - ISP, 46
 - programatora, 44
- zmiana stanu
 - pinów, 163–165
 - przycisku, 115
- zmienna
 - SENSE_TIME, 173
 - THRESHOLD, 173
- zmiennne
 - globalne, 171
 - ulotne, 171, 172
- znak liczby, 269

Ż

- źródła taktowania, 422

Ż

- żądanie przerwania, 162

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA WYDAWNICZA



Helion SA

Programowanie układów AVR

Współczesny świat elektroniki jest podbijany przez układ Arduino. Przemawiają za nim wygoda oraz proste tworzenie całkiem zaawansowanych projektów. Jeżeli jednak wymagasz najwyższej wydajności, reakcji na zdarzenia w czasie rzeczywistym lub wielozadaniowości, warto, żebyś wykonał kolejny krok i poznał układy AVR firmy Atmel. Brzmi zachęcająco?

Doskonale! W Twoje ręce oddajemy książkę, która pozwoli Ci poznać pasjonujący świat tych układów. W kolejnych rozdziałach nauczysz się wykorzystywać ich potencjał do pisania swoich własnych programów w języku C oraz komunikowania się ze światem zewnętrznym. Ponadto dowiesz się, jak korzystać z komunikacji szeregowej, wejść cyfrowych oraz przerwań sprzętowych. Na sam koniec, w części poświęconej zaawansowanym zagadnieniom, zobaczysz, jak używać przełączników i protokołu I2C oraz sterować silnikami. Książka ta przyda się wszystkim pasjonatom elektroniki, którzy pragną odkryć potencjał układów AVR.

Dzięki tej książce:

- » skompletujesz potrzebne narzędzia
- » nauczysz się programować układy AVR
- » wykorzystasz Arduino do programowania AVR
- » zastosujesz przerwania sprzętowe
- » użyjesz układu AVR w zaawansowanych projektach

helion.pl
księgarnia
internetowa

Nr katalogowy: 25456



Księgarnia internetowa:
<http://helion.pl>



Zamówienia telefoniczne:
0 801 339900



0 601 339900



Helion

Sprawdź najnowsze promocje:

• <http://helion.pl/promocje>

Książki najchętniej czytane:

• <http://helion.pl/bestsellery>

Zamów informacje o nowościach:

• <http://helion.pl/novosci>

Helion SA
ul. Kościuski 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
<http://helion.pl>

sięgnij po **WIĘCEJ**



KOD KORZYŚCI

ISBN 978-83-246-9501-0



cena 89,00 zł

Informatyka w najlepszym wydaniu

Make:
makezine.com