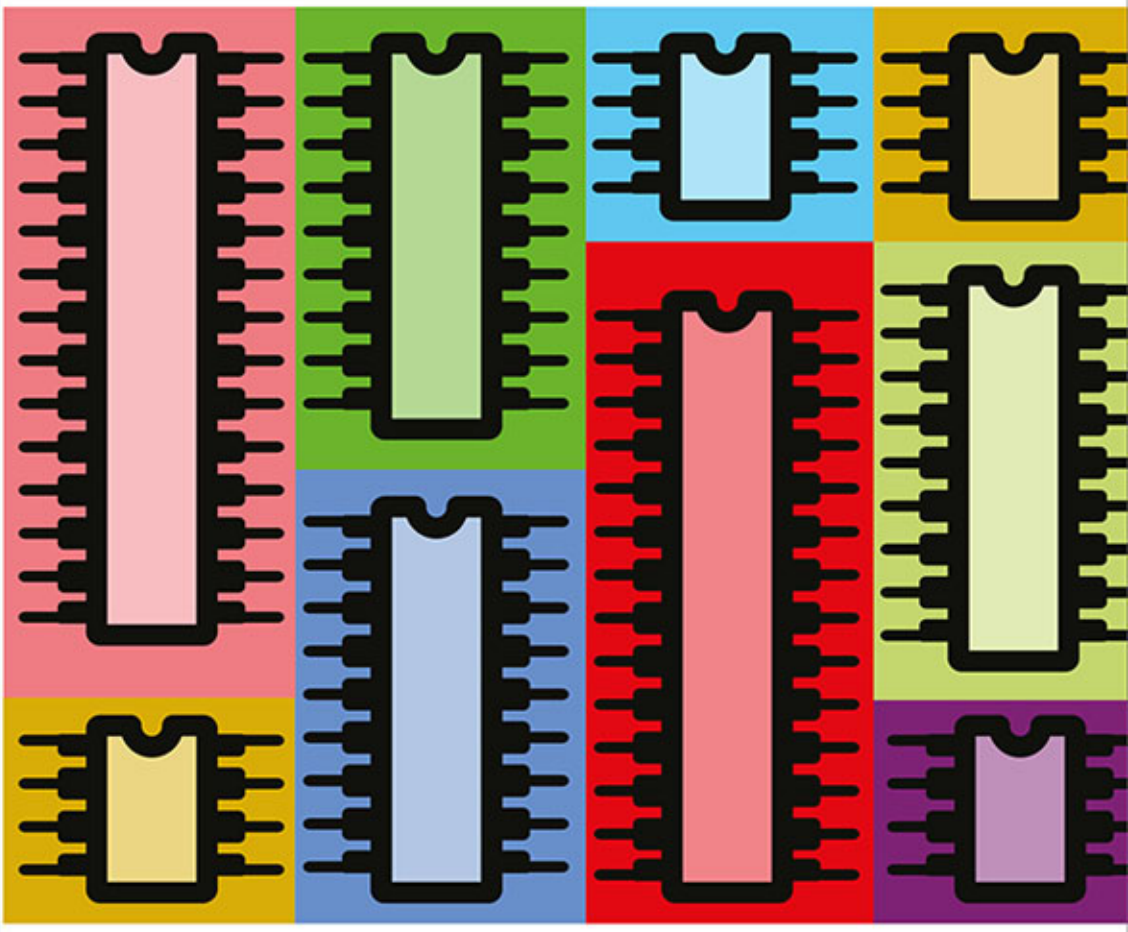
Programowanie układów AVR dla praktyków



Wykorzystaj potencjał układów AVR!

Elliot Williams

Helion

Tytuł oryginału: Make: AVR Programming

Tłumaczenie: Wojciech Moch

ISBN: 978-83-246-9501-0

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.

Spis treści

Część II Średnio zaawansowana

Modulacja szerokości impulsu

10

Przygaszanie 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 dziwnie, 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 zajmiemy 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.

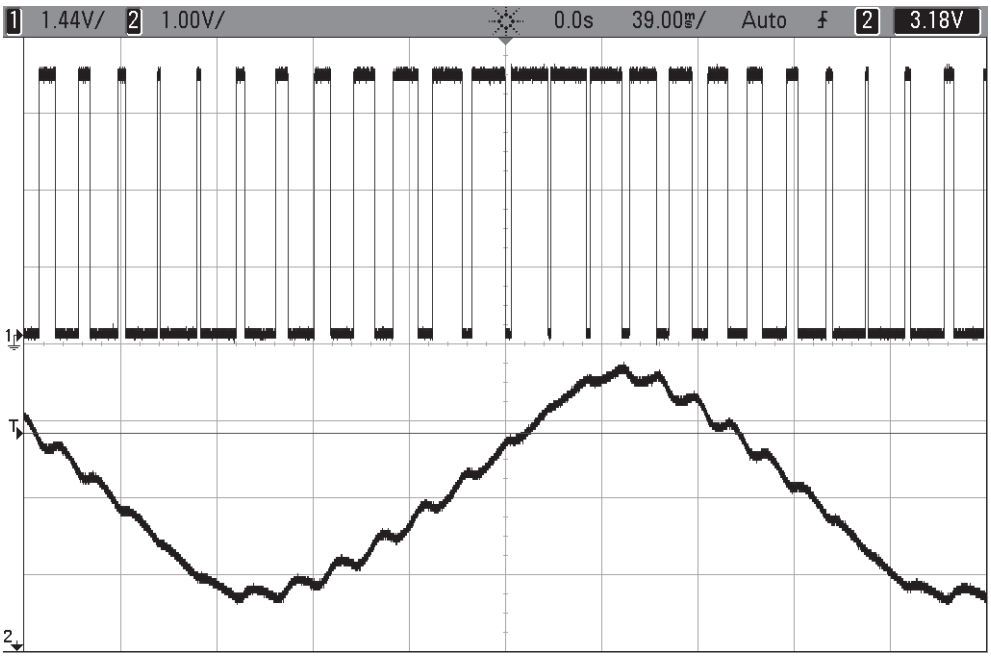
Czego potrzebujesz?

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 ludzkich 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 */

// ------- Preambuła -------- //
#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.

Przyglądając 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 przesuwając 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 */

// ------- Preambuła -------- //
#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;

  }                                                    /* Koniec pętli zdarzeń */
  return (0);                                  /* 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`*nx*, 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 `OCS`*nx*. 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 serwomotorem, 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 OCR*x*A) z szybkością definiowaną przez podzielnik sygnału zegara. Przy okazji wartość licznika porównywana jest z zawartością rejestru porównującego OCR*nx*. 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 Hz / 8 / 256 = 488 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 przepełnienia 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 inżynieryjnego 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 przepełnia 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 (uziemienie 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 wpiszemy 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 przesłonić wszystkie drobne szczegóły działania mikrokontrolerów, dlatego jej twórcy nie zaprzątają 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 odszukuje 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 wariancie. 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 podzielnik 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 podzielnikowi 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 przepełnieniu 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
// ------- Preambuła -------- //
#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 przepełnienia */
  sei();
}

ISR(TIMER0_OVF_vect) {
  LED_PORT = 0xff;
  OCR0A = brightnessA;
  OCR0B = brightnessB;
}
ISR(TIMER0_COMPA_vect) {
  LED_PORT &= 0b11110000;                          /* wyłącz dolne cztery diody */
}
ISR(TIMER0_COMPB_vect) {
  LED_PORT &= 0b00001111;                          /* wyłącz 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;
    }

  }                                          /* Koniec pętli zdarzeń */
  return (0);                    /* 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 `OCR0A` i `OCR0B`? 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 przepełnieniem licznika. Obie zmienne globalne mogą być w dowolnej chwili modyfikowane w funkcji `main()`, ale ich wartości zostaną przepisane do rejestrów `OCR0` dopiero po wywołaniu przerwania przepełnienia. Są zatem najprostszym buforem.

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

Pozostała część kodu jest już bardzo prosta. Procedury obsługi przerwań powinny być jak najkrótsze, ponieważ będą wywoływane naprawdę często. W tym przypadku przerwania 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 przerwania 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 przerwaniami, 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 przerwań.

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 miał 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 wyłączonym drugim pozwoli uzyskać na wyjściu napięcie 1/3 × 5 V albo (w odwrotnej konfiguracji) 2/3 × 5 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 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 konwerter 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 `CSnx` w rejestrze `TCCRnB`.
5. Jeżeli używasz wartości porównującej, dobrze jest wpisać jej domyślną wartość do rejestrów `OCRnA` oraz `OCRnB`. 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 przepełnienia licznika, ustawiając bit `TOIEn` w rejestrze `TIMSKn`.
   b. W trybach PWM lub CTC włącz przerwania wartości porównywanych, ustawiając bity `OCIEnA` i `OCIEnB` w rejestrze `TIMSKn`.
   c. Nie zapomnij włączyć całego systemu przerwań za pomocą instrukcji `sei()`. Pamiętaj o przygotowaniu procedur obsługi przerwań.

Skorowidz

E

F

G

I

J

K

U

W

Z

Ź

Ż

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/nowosci

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

Informatyka w najlepszym wydaniu

sięgnij po WIĘCEJ
KOD KORZYŚCI

ISBN 978-83-246-9501-0
cena 89,00 zł
9 788324 695010

Make:
makezine.com