Kurs AVR – lekcja 3

Rozwiązania zadań z ostatniego odcinka

Tradycyjnie odcinek zaczynamy od analizy zadania z poprzedniego numeru. Celem było wygenerowanie dźwięku o wysokości zależnej od wciśniętego przycisku. W pierwszym podejściu moglibyśmy spróbować napisać kod jak poniżej (zmieniamy wnętrze głównej pętli while(1), reszta kodu bez zmian):

```
while(1) {
 uint8_t key = readKeyboard();
 if (key) {
   PORTD = _BV(PD0);
   _delay_ms(key);
   PORTD = 0;
   _delay_ms(key);
 }
}
```

Niestety kompilacja się nie powiedzie, bo funkcja _delay_ms() może przyjmować tylko wartości znane w momencie kompilacji. Otrzymamy błąd __builtin_avr_delay_cycles expects a compile time integer constant. Nie można więc przekazać do niej zmiennej. Obejściem tego problemu może być stworzenie funkcji, która wywoła _delay_ms() ze stałą wartością (np. 0.1) określoną liczbę razy. W ten sposób będziemy mogli sterować długością opóźnienia za pomocą zmiennej. Cały kod będzie wyglądał jak na **listingu 1**.

Wprowadziliśmy tutaj dodatkową funkcję delay(), która pobiera parametr count. W pętli while() wielokrotnie wykonywana jest funkcja _delay_ms() i za każdym razem zmniejszana jest wartość zmiennej, w której dostępny jest parametr count. Gdy osiągnie ona wartość zero, pętla przerywa się i funkcja kończy swoje działanie.

Cechą tego rozwiązania jest to, że opóźnienie będzie wielokrotnością wartości przekazywanej do funkcji _delay_ms(). Jeśli przekazywaną wartością jest 0.1 (0,1ms), to będzie można uzyskać opóźnienia 100 µs, 200 µs, 300 µs itd. A jeśli potrzebne opóźnienia nie mają być wielokrotnościami? Np. miałyby być użyte do generowania dźwięków z oktawy muzycznej? Wtedy można użyć konstrukcji switch, która w zależności od wartości zmiennej key uruchomi funkcję _delay_ms() z różnymi parametrami (**listing 2**):

```
void delay(uint8_t key) {
 switch(key) {
  case 1:
  _delay_ms(1.9); break;
  case 2:
   _delay_ms(1.7); break;
  case 3:
   _delay_ms(1.52); break;
  case 4:
   _delay_ms(1.43); break;
  case 5:
   _delay_ms(1.27); break;
  case 6:
   _delay_ms(1.13); break;
  case 7:
   _delay_ms(1.01); break;
  case 8:
   _delay_ms(0.95); break;
 }
}
```

Mamy więc kod spełniający założenia zadania i można powiedzieć, że osiągnęliśmy cel. Czy jednak kod działa dokładnie tak, jak moglibyśmy się spodziewać? Podłączając miernik częstotliwości, zauważymy, że generowane częstotliwości są niższe, niż wynikałoby z obliczeń. Bierze się to z tego, że opóźnienia są generowane nie tylko przez funkcję _delay_ms(), ale też wykonywanie pozostałych operacji, przede wszystkim przez odczyt klawiatury. Sprawdzenie każdego przycisku zajmuje trochę czasu. Nie jest to też stały czas. Sprawdzane są kolejne przyciski i gdy okaże się, że któryś jest wciśnięty, to kolejne już sprawdzane nie są. Czas odczytu klawiatury zależy więc od tego, który przycisk jest wciśnięty. Funkcja najkrócej się wykona, gdy wciśnięty będzie przycisk 1, a najdłużej, gdy nie będzie wciśnięty żaden przycisk.

Można sobie z tym poradzić na kilka sposobów. Jednym z nich może być modyfikacja funkcji readKeyboard() tak, żeby zawsze wykonywała się tak samo długo. W tym celu w czasie przeczesywania klawiszy i znalezienia wciśniętego, należałoby jego numer zapamiętać w zmiennej, kontynuować przeczesywanie i wartość ze zmiennej zwrócić dopiero na końcu. Trzeba by następnie zmierzyć czas wykonywania funkcji, np. pisząc program, który ustawi stan pinu na 1, wywoła funkcję readKeyboard() a następnie ustawi pin w stan 0. Długość stanu 1 można by zmierzyć potem oscyloskopem. Otrzymaną wartość można by następnie uwzględnić przy obliczaniu opóźnień.

Inny sposób to po stwierdzeniu wciśnięcia przycisku wygenerować wiele okresów sygnału i dopiero później sprawdzić stan klawiatury ponownie. Można to zrobić tak:

```
while(1) {
 uint8_t key = readKeyboard();
 if (key) {
  for (uint16_t i = 0; i < 200; i++) {
    PORTD = _BV(PD0); delay(key);
    PORTD = 0; delay(key);
  }
 }
}
```

Pojawiają się tutaj dwa efekty. Jeden to ten, że odczytywanie klawiatury będzie powodowało słyszalne cykanie w momentach odczytu klawiatury. Drugi polega na tym, że długość generowanego dźwięku będzie zależała od jego wysokości. Żeby długości były równe, trzeba by przeliczać liczbę powtórzeń. Może to być kłopotliwe w przypadkach jak prezentowana opcja z konstrukcją switch, gdzie długość opóźnienia nie jest wielokrotnością numeru przycisku.

W ten sposób okazuje się, że proste z pozoru zagadnienie nie jest wcale tak trywialne w realizacji, a różne rozwiązania mogą mieć swoje efekty uboczne. Na szczęście nie wyczerpaliśmy jeszcze wszystkich możliwości i nie jesteśmy skazani na funkcję _delay_ms(). Z nieocenioną pomocą przychodzą timery i za chwilę je poznamy.

Listing 1

```
#include <avr/io.h>
#include <util/delay.h>
uint8_t readKeyboard();
void delay(uint8_t count);
int main(void) {
 DDRA = _BV(DDA0) | _BV(DDA1) | _BV(DDA2) | _BV(DDA3);
 DDRD = _BV(DDD0);
 while(1) {
  uint8_t key = readKeyboard();
  if (key) {
   PORTD = _BV(PD0); delay(key);
   PORTD = 0; delay(key);
  }
 }
}
uint8_t readKeyboard() {
 for (uint8_t row = 0; row < 4; row++){
  PORTA = ~_BV(row); _delay_us(1);
  for (uint8_t column = 0; column < 4; column++){
   if (~PINA & _BV(column + 4)) {
    return row * 4 + column + 1;
   }
  }
 }
 return 0;
}
void delay(uint8_t count) {
 while (count--) _delay_ms(0.05);
}
```

Timery/liczniki

Jedne z najważniejszych układów, jakie znajdziemy w mikrokontrolerach, to timery/liczniki. Są to specjalne układy zliczające impulsy sygnału zegarowego lub impulsy zewnętrzne, odbierane na określonej nóżce mikrokontrolera. Takie układy zliczające impulsy zegarowe świetnie nadają się do odmierzania czasu. Stąd też zwykle nazywane są timerami i przy tej nazwie pozostaniemy.

Istotną zaletą timerów jest to, że działają niezależnie od wykonującego się programu, pracują niejako w tle. Zadaniem programisty jest tylko konfiguracja timera i reagowanie na generowane przez niego zdarzenia.

Mikrokontroler ATmega32 zawiera 3 timery: 8-bitowy Timer0, 16-bitowy Timer1 oraz 8-bitowy Timer2. Przyjrzymy się najpierw temu pierwszemu.

Timer0 zawiera w sobie rejestry:

- TCCR0 służący do konfiguracji timera
- TCNT0 to licznik timera, przechowuje jego aktualną wartość
- OCR0 wykorzystywany przez funkcję Output Compare
- TIMSK służy do włączania/wyłączania poszczególnych przerwań dla wszystkich timerów
- TIFR przechowuje flagi powiązane z przerwaniami timerów

Aby timer zaczął działać, musimy określić dla niego źródło sygnału zegarowego. Domyślnie żadne źródło nie jest wybrane i timer stoi. Za wybór źródła odpowiedzialne są bity CS02...CS00 w rejestrze TCCR0. Konfiguracja odbywa się zgodnie z **tabelą 1**, clk oznacza główny zegar taktujący mikrokontrolera. Pin T0 to w przypadku ATmega32 pin PB0.

Wykonajmy więc pierwsze ćwiczenie z timerem. Włączymy go i będziemy obserwować wartość jego rejestru liczącego, czyli TCNT0. Do wyświetlania wartości rejestru wykorzystamy linijkę LED-ów, dzięki której będziemy mogli obserwować binarną reprezentację wartości licznika. Żeby dało się coś zobaczyć, zastosujemy maksymalny stopień podziału sygnału zegarowego, czyli 1024. Zgodnie z tabelą musimy ustawić bity CS00 i CS02 w rejestrze TCCR0. Jeśli mikrokontroler będzie taktowany domyślnym wewnętrznym generatorem RC o częstotliwości 1 MHz, to zmianę najstarszych bitów uda się zaobserwować gołym okiem. Kod do ćwiczenia będzie wyglądał następująco (**listing 3**):

Tabela 1

CS02	CS01	CS00	Sygnał zegarowy
0	0	0	Brak, timer wyłączony
0	0	1	clk/1
0	1	0	clk/8
0	1	1	clk/64
1	0	0	clk/256
1	0	1	clk/1024
1	1	0	Pin T0, zbocze opadające
1	1	1	Pin T0, zbocze narastające

```
#include <avr/io.h>
int main(void) {
  DDRA = 0xff;
  TCCR0 = _BV(CS02) | _BV(CS00);
  while(1) PORTA = TCNT0;
}
```

Na początku ustawiamy wszystkie piny portu A jako wyjścia. Następnie konfigurujemy źródło sygnału zegarowego jako główny zegar dzielony przez 1024. Rejestr TCNT0 będzie się więc zwiększał o 1 około 1 000 000 / 1024 = 977 razy na sekundę. W pętli while wyświetlamy zawartość rejestru TCNT0 na linijce diodowej podłączonej do portu A.

Taktowanie zewnętrznym sygnałem

Sprawdźmy, jak będzie wyglądało taktowanie timera sygnałem zewnętrznym. Na płytce testowej mamy generator sygnału prostokątnego, zbudowany na układzie NE555. Wyjście tego generatora to pin GO na złączu MISC. Połączmy więc ten pin z pinem 0 portu B, który pełni też funkcję wejścia dla naszego timera. Zgodnie z tabelą, aby taktować Timer0 sygnałem zewnętrznym, musimy ustawić bity CS1 i CS2. Od bitu CS0 będzie zależało, czy rejestr TCNT0 będzie się zwiększał wraz ze zboczem opadającym, czy narastającym. W naszym ćwiczeniu nie ma to znaczenia. Zmodyfikujmy więc kod naszego ćwiczenia. Zmiana to de facto tylko jedna cyfra: zamiast ustawiać bit CS00, ustawimy bit CS01.

Po uruchomieniu programu zaważymy prawdopodobnie tylko migotanie LED8, pozostałe będą migać zbyt szybko. Przy taktowaniu zewnętrznym sygnałem nie ma podziału częstotliwości i nawet przy taktowaniu stosunkowo niską częstotliwością, zmiany wartości TCNT0 następują bardzo szybko.

W praktyce rzadko odczytujemy bezpośrednio wartość rejestru licznika timera. Zwykle interesują nas takie zdarzenia, jak przepełnienie licznika lub osiągnięcie zadanej wartości. Timer może o takich zdarzeniach informować, generując przerwanie. W momencie wystąpienia przerwania timer ustawia odpowiednią flagę (bit) w rejestrze TIFR. Dodatkowo może zostać przerwane wykonywanie głównego programu mikrokontrolera i może zostać wykonany skok do procedury obsługi przerwania. Na razie jednak pozostańmy przy flagach. Obsłudze przerwań poświęcimy uwagę w dalszej części kursu.

Flaga przepełnienia

Timer0 jest ośmiobitowy, przepełnienie jego licznika następuje więc przy kolejnym impulsie zegarowym po osiągnięciu wartości 255. Ustawiona zostaje wtedy flaga (bit) TOV0 w rejestrze TIFR. Zmodyfikujmy więc nasz przykład, aby wykorzystywał wspomnianą flagę (**listing 4**).

```
#include <avr/io.h>
int main(void) {
 DDRA = 0xff;
 TCCR0 = _BV(CS02) | _BV(CS01);
 while(1) {
  if (TIFR & _BV(TOV0)) {
   TIFR |= _BV(TOV0);
   PORTA++;
  }
 }
}
```

Pozostajemy przy taktowaniu zewnętrznym sygnałem zegarowym, ustawione nadal są bity CS01 i CS02. W pętli sprawdzamy, czy flaga przepełnienia została ustawiona. Jeśli tak, to ją kasujemy. Kasowanie odbywa się przez ustawienie bitu flagi na 1. Może się to wydawać nieintuicyjne, ponieważ flaga została właśnie ustawiona na 1. Trzeba jednak pamiętać, że rejestry nie są zwykłymi komórkami pamięci. Ustawianie bądź zerowanie określonych bitów może powodować wykonanie określonych czynności. Tutaj ustawianie ustawionej flagi powoduje jej wyczyszczenie.

Po skasowaniu flagi inkrementowana jest wartość rejestru portu A. Jest on tutaj traktowany jako zwykła zmienna licznikowa, zwiększana o 1 po każdym przepełnieniu licznika timera. Po uruchomieniu programu zauważymy, że LED-y migają dużo wolniej, teraz da się zaobserwować, jak pojawiają się kolejne wartości. To, co teraz zrobiliśmy, to podział częstotliwości przez 256. Przedtem wyświetlany był bezpośrednio stan licznika, zwiększany po każdym impulsie zegarowym. Teraz takich impulsów musi przyjść 256, aby nastąpiło przepełnienie 8-bitowego licznika i nastąpiło zwiększenie wyświetlanej wartości o 1.

Funkcja Output Compare

Wykorzystując funkcję Output Compare, możemy niejako skrócić licznik timera. Wartość licznika jest przy każdej jego zmianie porównywana z wartością zapisaną w rejestrze OCR0. Gdy są zgodne, następuje ustawienie odpowiedniej flagi (OCF0) w rejestrze TIFR. Domyślnie licznik będzie liczył dalej, aż do wartości 255. Aby był zerowany po osiągnięciu liczby z OCR0, trzeba włączyć tryb CTC (Clear Timer on Compare Match). Wykonujemy to, ustawiając bit WGM01 w rejestrze TCCR0 według

Tabela 2

Tryb pracy	WGM01	WGM00	Maksymalna wartość licznika (TOP)	Aktualizacja rejestru OCR0 przy wartości	Ustawienie flagi TOV0 przy wartości
Normalny	0	0	0xFF	Natychmiastowo	MAX
PWM, Phase Correct	0	1	0xFF	TOP	BOTTOM
CTC	1	0	OCR0	Natychmiastowo	MAX
Fast PWM	1	1	0xFF	BOTTOM	MAX

tabeli 2. Użyte w tabeli oznaczenia BOTTOM, TOP i MAX oznaczają odpowiednio: najniższą możliwą wartość licznika, czyli 0, najwyższą wartość, do której liczy licznik w danym trybie, oraz maksymalną możliwą wartość, jaka może znaleźć się w rejestrze licznika (0xFF).

W tym ćwiczeniu powrócimy do taktowania licznika zegarem systemowym. Będzie on dzielony przez 1024 (ustawione bity CS00 i CS02) - **listing 5**.

Do rejestru OCR0 wpisana zostaje liczba, do której ma liczyć licznik. W trybach PWM nie jest ona uwzględniana od razu, ale w momencie osiągnięcia wartości TOP lub BOTTOM. Dzięki temu nie ma zakłóceń w sygnale PWM podczas zmiany wartości rejestru. Wartość 25 da mniej więcej 10-krotnie skrócenie cyklu pracy licznika. Można oczywiście poeksperymentować z różnymi wartościami, mieszczącymi się w 8 bitach. W głównej pętli sprawdzane jest, czy ustawiona została flaga OCF0 w rejestrze TIFR. Jeśli tak, jest czyszczona, a liczba wyprowadzana na port A zostaje zwiększona o 1.

Korzystając z posiadanej wiedzy, możemy przepisać nasz pierwszy program migający diodą. Zamiast funkcji opóźniającej zostanie wykorzystany Timer0.

A teraz **listing 6**: pin 0 portu A zostaje skonfigurowany jako wyjście, włączona zostaje funkcja CTC. Wartość 250 w rejestrze OCR0 przy preskalerze 1024 i zegarze 1 MHz da miganie z częstotliwością ok. 2Hz. W głównej pętli czekamy na ustawienie się flagi OCF0. Wykonane jest to jednolinijkową pętlą while(), która kręci się bezczynnie tak długo, jak nie jest spełniony warunek ustawienia bitu OCF0 w rejestrze TIFR. Gdy zostanie ustawiony, jest czyszczony jak poprzednio i następuje włączenie stanu wysokiego na pinie 0 portu A. Potem wykonywane jest to samo, ale na pinie ustawiany jest stan niski.

Aby kod był bardziej elegancki, pętlę sprawdzającą flagę można wydzielić do oddzielnej funkcji, która odgrywałaby rolę funkcji opóźniającej. Jeślibyśmy chcieli napisać taką funkcję, bo z jakichś względów nie pasowałaby nam biblioteczna funkcja _delay_ms(), to trzeba by dodać zerowanie licznika. W powyższym przykładzie nie wiemy tak naprawdę, jaki jest stan licznika w momencie gdy startuje pętla sprawdzająca flagę. Dobrze byłoby też sparametryzować funkcję, aby można było osiągać różne czasy opóźnienia. Kod mógłby wyglądać jak na **listingu 7**.

Ponieważ timer działa cały czas, osiągnięcie przez jego licznik zadanej wartości może się zdarzyć także poza powyższą pętlą opóźniającą, czyszczenie flagi zostało umieszczone przed pętlą, która ją sprawdza.

Automatyczne sterowanie pinem przez timer

W przedstawionych ćwiczeniach na podstawie flag timera była wykonywana określona czynność – zwiększanie wartości rejestru portu A lub ustawianie stanu pojedynczego pinu tego portu. Okazuje się, że jeśli chcielibyśmy wykorzystać timer do sterowania stanem pinu, to dostępny w naszym mikrokontrolerze Timer0 może to robić automatycznie. Pinem tym jest OC0, pracujący też jako pin 3 portu B. Zostańmy przy trybie CTC, w którym licznik zlicza od zera aż do wartości zapisanej w rejestrze OCR0, po czym zeruje się i cykl się powtarza. W momencie osiągnięcia przez licznik wartości OCR0, stan pinu OC0 może się zmienić na trzy sposoby, zależnie od konfiguracji bitów COM01 i COM00 w rejestrze TCCR0. Dostępne tryby przedstawione są w **tabeli 3**.

Dzięki opcji przestawiania stanu pinu na przeciwny możemy łatwo zrealizować generator sygnału prostokątnego. Podłączmy więc pin OC0 (PB3) np. do diody LED i uruchommy program z **listingu 8**.

Jest on o tyle ciekawy, że główna pętla jest zupełnie pusta. Wszystko, co najważniejsze, dzieje się zaraz po resecie mikrokontrolera. Pin OC0 (PB3) konfigurowany jest jako wyjście. Dla timera Timer0 źródłem sygnału zegarowego jest jak poprzednio zegar wewnętrzny, dzielony przez 1024. Włączony jest tryb CTC oraz przełączanie pinu OC0 przy osiągnięciu wartości z rejestru OCR0. Do rejestru tego wpisana jest wartość 199, co razem z zerem daje 200 stanów. Ponieważ pin zmienia stan na przeciwny co 200 impulsów zegarowych, pełny okres wyniesie 400 impulsów. Zrealizowaliśmy więc dzielnik częstotliwości przez 400. Biorąc jeszcze pod uwagę preskaler o wartości 1024, zegar systemowy jest dzielony przez 409600. Przy taktowaniu mikrokontrolera zegarem 1 MHz otrzymujemy na

Listing 5

```
#include <avr/io.h>
int main(void) {
 DDRA = 0xff;
 TCCR0 = _BV(CS02) | _BV(CS00) | _BV(WGM01);
 OCR0 = 25;
 while(1) {
  if (TIFR & _BV(OCF0)) {
   TIFR |= _BV(OCF0);
   PORTA++;
  }
 }
}
```

Listing 6

```
#include <avr/io.h>
int main(void) {
 DDRA = _BV(DDA0);
 TCCR0 = _BV(CS02) | _BV(CS00) | _BV(WGM01);
 OCR0 = 250;
 while(1) {
  while(!(TIFR & _BV(OCF0)));
  TIFR |= _BV(OCF0);
  PORTA |= _BV(PA0);
  while(!(TIFR & _BV(OCF0)));
  TIFR |= _BV(OCF0);
  PORTA &= ~_BV(PA0);
 }
}
```

Listing 7

```
#include <avr/io.h>
void delay(uint8_t val);
int main(void) {
 DDRA = _BV(DDA0);
 TCCR0 = _BV(CS02) | _BV(CS00) | _BV(WGM01);
 while(1) {
  delay(250); PORTA |= _BV(PA0);
  delay(250); PORTA &= ~_BV(PA0);
 }
}
void delay(uint8_t val) {
 TCNT0 = 0;
 OCR0 = val;
 TIFR |= _BV(OCF0);
 while(!(TIFR & _BV(OCF0)));
}
```

Listing 8

```
#include <avr/io.h>
int main(void) {
 DDRB = _BV(DDB3);
 TCCR0 = _BV(CS02) | _BV(CS00) | _BV(WGM01) | _BV(COM00);
 OCR0 = 199;
 while(1) {}
```

Tabela 3

COM01	COM00	Opis
0	0	Brak sterowania pinem OC0 przez timer
0	1	Przestawienie na stan przeciwny
1	0	Ustawienie w stan niski
1	1	Ustawienie w stan wysoki

pinie częstotliwość ok. 2,44 Hz. W ten oto sposób otrzymujemy generator przebiegu prostokątnego, który działa całkowicie sprzętowo. Jedynie jego konfiguracja wykonywana jest programowo. W trakcie pracy program wykonuje tylko skoki w pustej pętli.

Oczywiście pętlę można wypełnić kodem realizującym różne inne funkcje, np. rekonfigurację timera. Dodajmy więc możliwość zmiany generowanej częstotliwości z poziomu klawiatury matrycowej. Jej obsługę już poznaliśmy. Ponieważ korzysta ona z funkcji opóźniającej, w opcjach projektu musi być zdefiniowane makro F_CPU o wartości 1000000 – **listing 9**.

Tym razem w głównej pętli programu coś się dzieje. Wywoływana jest funkcja odczytująca numer wciśniętego klawisza. Jeśli klawisz jest wciśnięty, czyli funkcja zwróciła wartość różną od zera, numer klawisza jest mnożony przez 10 i wpisywany do rejestru OCR0. W ten sposób otrzymujemy wartości od 10 do 160. Oczywiście można użyć innego mnożnika, np. 15. Przy mnożniku 16 ostatni klawisz spowoduje wpisanie do OCR0 liczby 0, ponieważ 256 (16*16) nie zmieści się w 8-bitowym rejestrze. Licznik będzie się ciągle resetował z częstotliwością ok. 977 Hz (1 MHz / 1024), co spowoduje generowanie na pinie OC0 (PB3) częstotliwości ok. 488 Hz.

W ramach samodzielnych ćwiczeń warto pokusić się o różne modyfikacje programu. Można np. dodać wyłączanie generatora przy naciśnięciu któregoś z klawiszy. Wyłączenie można zrealizować, zatrzymując timer (wyzerowanie bitów CS00..03) lub wyłączając funkcję sterowania pinem (wyzerowanie bitu COM00).

Generowanie pojedynczego impulsu

Jak mogliśmy przeczytać w tabeli, funkcja Output Compare jest w stanie nie tylko przełączać stan pinu OC0 (przestawiać go w stan przeciwny), ale też ustawiać konkretny stan. Dzięki temu można zrealizować na timerze przerzutnik monostabilny, który wygeneruje pojedynczy impuls. Spróbujmy więc napisać program, który po naciśnięciu przycisku wywoła krótkie mignięcie diody. Długość mignięcia będzie zależała od numeru wciśniętego przycisku klawiatury matrycowej. Na **listingu 10** sama funkcja main(), reszta kodu bez zmian.

Niby wszystko jest w porządku. Zerujemy licznik, wpisujemy wartość, którą ma on osiągnąć, ustawiamy stan wysoki na pinie, a na koniec uruchamiamy timer z opcją zerowana pinu po osiągnięciu dopasowania wartości. Program jednak nie działa, LED podłączony do pinu się nie zaświeca. Okazuje się, że timer, przejmując kontrolę nad pinem, nie tylko ustawia jego stan w momencie osiągnięcia przez licznik wartości z rejestru OCR0, ale kontroluje go od momentu, gdy zostaje włączony. Dokumentacja wspomina o tym fakcie, ale w sposób mało czytelny. Podaje też dosyć nietypowe rozwiązanie. Aby ustawić początkowy stan pinu, trzeba wygenerować dopasowanie wartości „ręcznie”. Jeśli naszym założeniem jest gaszenie LED-a po określonym czasie, trzeba timer najpierw skonfigurować w tryb ustawiania stanu wysokiego na pinie. Zamiast ustawiać bit PB3 rejestru PORTB, należy w rejestrze konfiguracyjnym timera zerowego włączyć tryb CTC oraz funkcję ustawiania stanu wysokiego dla zdarzenia dopasowania wartości. Czyli ustawione zostają bity WGM01, COM01 i COM00. Następnie należy wywołać zdarzenie osiągnięcia przez licznik wartości zadanej w OCR0. Wykonuje się to, ustawiając bit FOC0 (funkcja Force Output Compare). Spowoduje to ustawienie stanu wysokiego na pinie. Nie nastąpi przy tym wyzerowanie licznika ani ustawienie flagi OCF0. Potem można już normalnie włączyć timer ze skonfigurowanym sygnałem zegarowym oraz włączoną funkcją ustawiania stanu niskiego na pinie po osiągnięciu przez licznik timera wartości z rejestru OCR0. Nasza funkcja main() będzie więc wyglądać jak na **listingu 11**:

Jeśli chcielibyśmy uzyskać efekt odwrotny, czyli dioda normalnie włączona i gaszenie na krótki moment przyciskiem, trzeba odwrotnie ustawić bity COM01 i COM00: najpierw tylko COM01 a potem oba. Dodatkowo na samym początku, przed pętlą while, trzeba ustawić bit PB3 w rejestrze PORTB. Inaczej LED włączy się dopiero po pierwszym przyciśnięciu przycisku.

Listing 9

```
#include <avr/io.h>
#include <util/delay.h>
uint8_t readKeyboard();
int main(void) {
 DDRB = _BV(DDB3);
 DDRA = _BV(DDA0) | _BV(DDA1) | _BV(DDA2) | _BV(DDA3);
 TCCR0 = _BV(CS02) | _BV(CS00) | _BV(WGM01) | _BV(COM00);
 OCR0 = 199;
 while(1) {
  uint8_t key = readKeyboard();
  if (key) OCR0 = 10 * key;
 }
}
uint8_t readKeyboard() {
 for (uint8_t row = 0; row < 4; row++){
  PORTA = ~_BV(row); _delay_us(1);
  for (uint8_t column = 0; column < 4; column++){
   if (~PINA & _BV(column + 4)) {
    return row * 4 + column + 1;
   }
  }
 }
 return 0;
}
```

Listing 10

```
int main(void) {
 DDRB = _BV(DDB3);
 DDRA = _BV(DDA0) | _BV(DDA1) | _BV(DDA2) | _BV(DDA3);
 while(1) {
  uint8_t key = readKeyboard();
  if (key) {
   TCNT0 = 0;
   OCR0 = key * 15;
   PORTB |= _BV(PB3);
   TCCR0 = _BV(CS02) | _BV(CS00) | _BV(WGM01) | _BV(COM01);
  while (readKeyboard());
  }
 }
}
```

Listing 11

```
int main(void) {
 DDRB = _BV(DDB3);
 DDRA = _BV(DDA0) | _BV(DDA1) | _BV(DDA2) | _BV(DDA3);
 while(1) {
  uint8_t key = readKeyboard();
  if (key) {
   TCNT0 = 0;
   OCR0 = key * 15;
   TCCR0 = _BV(COM01) | _BV(COM00)| _BV(WGM01);
   TCCR0 |= _BV(FOC0);
   TCCR0 = _BV(CS02) | _BV(CS00) | _BV(WGM01) | _BV(COM01);
   while (readKeyboard());
  }
 }
}
```

PWM

Jak już wiemy, mikrokontroler może zmieniać automatycznie stan pinu w momencie osiągnięcia przez licznik timera określonej wartości albo przy jego przepełnieniu. A czy jest możliwe, aby obie te funkcje działały jednocześnie i przełączały pin zarówno dla zdarzenia osiągnięcia określonej wartości, jak i dla przepełnienia? W ten sposób można by zrealizować funkcję PWM. Jak najbardziej jest to możliwe.

Zatrzymajmy się na chwilę i przypomnijmy, czym jest PWM (Pulse Width Modulation). Funkcja ta polega na generowaniu sygnału prostokątnego o określonej częstotliwości i zmienianiu jego

wypełnienia, czyli długości zera i jedynki. Suma czasów trwania zera i jedynki pozostaje stała, zmienia się ich stosunek. Załóżmy, że generowany jest sygnał o częstotliwości 1 kHz, czyli jeden okres trwa 1 milisekundę. Zwykle wypełnienie wynosi 50%, czyli zero trwa przez 0,5 ms i jedynka tak samo. Ale wypełnienie może być inne, np. 10% i wtedy pin wyjściowy będzie w stanie niskim przez 0,9ms, a w stanie wysokim przez 0,1ms. Takie modyfikowanie szerokości impulsu (jedynki) znajduje bardzo wiele zastosowań. Można np. sterować jasnością LED-a, włączonego pomiędzy pin PWM a masę. Czym wypełnienie będzie większe, tym dioda będzie świecić jaśniej. Jeśli dioda będzie włączona między pin a plus zasilania, będzie odwrotnie – mniejsze wypełnienie da jaśniejsze świecenie. Oprócz LED-ów czy żarówek, PWM można wykorzystać także do sterowania silnikami lub grzałkami. Oczywiście przy urządzeniach większej mocy konieczne będzie dodanie odpowiednich tranzystorów wykonawczych, gdyż wydajność portów mikrokontrolera jest ograniczona. PWM to prymitywny przetwornik cyfrowo-analogowy, używany tam, gdzie potrzebne są wartości pomiędzy zerem a jedynką. Generowany sygnał prostokątny jest często wygładzany filtrem RC.

Timer0 w mikrokontrolerze ATmega32 ma dwa tryby PMW: szybki (Fast PWM Mode) oraz z korekcją fazy (Phase Correct PWM Mode). W trybie szybkim licznik timera liczy od zera do wartości maksymalnej, czyli 255 (FFh), a następnie przepełnia się i liczy od nowa. Przełączenie stanu pinu wyjściowego następuje po osiągnięciu przez licznik wartości takiej, jak jest wpisana w rejestrze OCR0. Wartość tego rejestru wyznacza więc wypełnienie generowanego sygnału PWM. W trybie nieodwracającym na początku cyklu pin wyjściowy jest ustawiany w stan wysoki, a przełączany jest w stan niski po osiągnięciu wartości z OCR0. Im wartość ta będzie większa, tym przełączenie z jedynki na zero będzie następowało później i średnie napięcie na pinie wyjściowym będzie wyższe. W trybie odwracającym sytuacja jest przeciwna: pin na początku jest w stanie niskim i po osiągnięciu przez licznik wartości z rejestru OCR0 następuje przełączenie w stan wysoki. A więc większe wartości OCR0 będą oznaczały mniejsze wypełnienie.

Tryb wybieramy bitami COM01 i COM00 (**tabela 4**).

Przetestujmy tryb Fast PWM w roli sterownika jasności LED. Aby włączyć tryb Fast PWM, potrzebujemy ustawić bity WGM01 oraz WGM00 w rejestrze TCCR0. Jasność diody będziemy kontrolować przez ustawianie wartości rejestru OCR0 za pomocą klawiatury, analogicznie do poprzedniego ćwiczenia. Częstotliwość nie jest krytyczna. Możemy wykorzystać główny sygnał zegarowy dzielony przez 8 (ustawiamy bit CS01 w TCCR0). Da to częstotliwość 1 MHz/8/256 ≈ 488 Hz. Na **listingu 12** funkcja main(), deklaracje oraz definicja funkcji readKeyboard() pozostają bez zmian.

```
int main(void) {
 DDRB = _BV(DDB3);
 DDRA = _BV(DDA0) | _BV(DDA1) | _BV(DDA2) | _BV(DDA3);
 TCCR0 = _BV(WGM01) | _BV(WGM00) | _BV(COM01) | _BV(CS01);
 while(1) {
  uint8_t key = readKeyboard();
  if (key) {
   OCR0 = key * 15;
   while (readKeyboard());
  }
 }
}
```

Listing 12

Testując program, można zauważyć, że nawet przy najmniejszym wypełnieniu dioda świeci dosyć jasno, a przecież dla pierwszego przycisku wypełnienie wyniesie tylko 1 * 15 / 256 ≈ 6%. Związane jest to z nieliniową czułością oka ludzkiego. Rozwiązaniem może być podnoszenie liczby odczytanej z klawiatury do kwadratu:

```
OCR0 = key * key;
```

Wrażenie jest dużo lepsze. Ale przy okazji można też zauważyć coś nietypowego: naciśnięcie przycisku S16 powoduje lekkie świecenie LED-a. Naciśnięcie tego przycisku powoduje zwrócenie przez funkcję readKeyboard() liczby 16, która podniesiona do kwadratu daje 256. Wartość ta nie mieści się w 8-bitowym rejestrze i pozostanie z niej samo zero. Wartość 0 w rejestrze OCR0 powinna dać więc zerowe wypełnienie i zgaszoną diodę. Tak się jednak nie dzieje. Nie jest to błąd w programie, ale efekt specyfiki działania trybu Fast PWM w naszym mikrokontrolerze. Przy przepełnieniu się licznika generowana jest krótka szpilka. Należy o tym pamiętać, korzystając z tego trybu PWM, może to przeszkadzać w sterowaniu niektórymi układami.

Tryb Phase Correct PWM działa w ten sposób, że licznik, osiągając wartość maksymalną, zaczyna liczyć w dół. Gdy osiągnie zero, znów liczy w górę. Nie ma więc przepełniania się licznika po osiągnięciu wartości maksymalnej. A jak sterowany jest pin wyjściowy OC0? Podobnie jak w trybie Fast PWM, w trybie nieodwracającym, na początku cyklu pin ustawiany jest w stan wysoki. Po osiągnięciu przez licznik wartości równej rejestrowi OCR0, pin ustawiany jest w stan niski. Gdy licznik osiągnie wartość maksymalną, a następnie będzie odliczać w dół, znów nastąpi przełączenie w stan niski przy zrównaniu wartości licznika z wartością rejestru OCR0. Zauważyć tutaj można dwie różnice w stosunku do poprzednio omawianego trybu. Ponieważ wykorzystane jest liczenie zarówno w górę, jak i w dół, cały cykl trwa dwa razy dłużej. Dlatego tryb ten nie jest „fast". Drugi efekt to to, że środek stanu wysokiego wypada zawsze w tym samym momencie, gdy licznik jest wyzerowany. Daje to zgodność fazy, która może być potrzebna w niektórych zastosowaniach. Generowanych jest też mniej harmonicznych. Ponadto w trybie Phase Correct PWM nie występuje niepożądana szpilka, którą mogliśmy obserwować w trybie Fast PWM. Aby przetestować Phase Correct PWM, wystarczy, że w naszym przykładzie nie będziemy ustawiać bitu WGM01. Inicjalizacja rejestru TCCR0 będzie więc wyglądać następująco:

```
TCCR0 = _BV(WGM00) | _BV(COM01) | _BV(CS01);
```

Tabela 4

COM01	COM00	Opis
0	0	Brak sterowania pinem OC0 przez timer
0	1	Zarezerwowany
1	0	Tryb nieodwracający
1	1	Tryb odwracający

Zadania

1. Wykonajmy kolejne podejście do generowania dźwięku przy wciśniętym przycisku, tym razem z wykorzystaniem timera Timer0, bez uciekania się do funkcji _delay_ms(). Wysokość dźwięku ma zależeć od tego, który przycisk klawiatury matrycowej jest wciśnięty.

2. Pulsujący LED: naprzemienne stopniowe zaświecanie i gaszenie diody świecącej.

Grzegorz Niemirowski
grzegorz@grzegorz.net