Kurs AVR – lekcja 10

Rozwiązania zadań z ostatniego odcinka

Pierwszym zadaniem domowym było napisanie zegara pokazującego aktualny czas na wyświetlaczu LED. Ponieważ wyświetlacz na płytce testowej ma tylko cztery cyfry, wyświetlane mogły być godziny i minuty, bez sekund. Aby zrealizować zegar w tej postaci, potrzebujemy wczytać z klawiatury aktualną liczbę godzin oraz minut a następnie je wyświetlać, co 60 sekund aktualizując wskazanie. Z poprzednich lekcji nauczyliśmy się wszystkich niezbędnych elementów: wczytywania liczb, ich wyświetlania oraz odmierzania czasu. Spróbujmy napisać odpowiedni kod (**listing 1**).

```
#include <avr/io.h>
#include <util/delay.h>
#include "keyb.h"
#include "led.h"

int main(void)
{
 ledInit();
 keybInit();
 ledClear();
 uint8_t hours = readNumber(0, 2);
 uint8_t minutes = readNumber(2, 2);
 uint8_t seconds = 0;
 ledClear();
 TCCR1B = _BV(WGM12) | _BV(CS12) | _BV(CS10);
 OCR1A = 15624;
 while(1)
 {
  ledDispNumber(hours, "%g", 0);
  ledDispNumber(minutes, "%g", 2);
  while(!(TIFR & _BV(OCF1A)));
  TIFR |= _BV(OCF1A);
  seconds++;
  if (seconds == 60) {
   seconds = 0;
   minutes++;
  }
  if (minutes == 60) {
   minutes = 0;
   hours++;
  }
  if (hours == 24) {
   hours = 0;
  }
 }
}
```

Na początku inicjalizowany jest wyświetlacz i klawiatura. Potem wczytywane są godziny i minuty jako liczby dwucyfrowe. Następnie Timer1 jest konfigurowany w trybie CTC, aby ustawiał flagę OCF1A co sekundę. Zakładamy tutaj, że fuse bity skonfigurowane są dla rezonatora kwarcowego (16MHz). W głównej pętli wyświetlane są godziny i minuty, odczekiwana jest sekunda i wykonywana jest aktualizacja liczby sekund, minut oraz godzin.

Przetestujmy nasz kod. Po zaprogramowaniu procesora ustawiamy czas, potwierdzając osobno liczbę minut i godzin Enterem (przycisk S16). Możliwe, że nie zauważymy żadnych nieprawidłowości. Jeśli jednak czas zmieni się z 23:59 na 00:00, na wyświetlaczu ukażą się cyfry 0309. Gdzie popełniony został błąd? Czy minuty i godziny są źle obliczane? Nie, tutaj jest wszystko w porządku. Problem leży w ich wyświetlaniu. Po prostu liczba zero jest wyświetlana za pomocą pojedynczej cyfry zero. W ten sposób 2359 zamienia się w 0309, bo cyfry 3 i 9 nie są czyszczone. Jak więc uzyskać 0000 na wyświetlaczu? Funkcja ledDispNumber() musi wyświetlać wszystkie liczby jako dwucyfrowe, nawet te mniejsze od 10. O formacie wyświetlania liczb decyduje drugi parametr funkcji. Musimy tutaj wrócić do lekcji 6 (**tabele 2 i 3**), gdy omawialiśmy opcje formatowania dla funkcji printf() i pokrewnych, bowiem funkcja ledDispNumber() odwołuje się do snprintf(). W łańcuchu formatującym po znaku % możemy podać m.in. flagi oraz szerokość wyświetlanej liczby. Potrzebujemy wyświetlania dwucyfrowego, więc szerokość będzie wynosić 2. Da nam to łańcuch formatujący "%2g". Dzięki temu liczby mniejsze od 10 zostaną wyświetlone na pozycji drugiej i czwartej od lewej, a nie pierwszej i trzeciej. A co będzie na pozycji pierwszej i trzeciej? Funkcja snprintf() wygeneruje tam spacje, przy czym nie zostaną one wyświetlone, gdyż funkcja ledDispNumber() nie obsługuje spacji. Jeśli po 23:59 nastąpi północ, na wyświetlaczu będą cyfry 20:50. Potrzeba aby wyrównanie do lewej wykonywane było nie spacjami, a zerami. Na szczęście jest do tego opcja formatująca i ma ona postać cyfry zero. Interesujący nas ciąg formatujący będzie więc miał ostatecznie postać "%02g".

Powyższy przykład służył nie tylko odświeżeniu wiadomości o formatowaniu liczb, ale też zwróceniu uwagi na testowanie warunków brzegowych. Gdybyśmy ustawili np. godzinę 12:20, to obserwując działanie programu przez pół godziny, nie zauważylibyśmy żadnych oznak nieprawidłowego działania i moglibyśmy uznać, że wszystko jest w porządku. Dlatego warto testować zachowanie programu w przypadkach skrajnych, takich jak właśnie moment na chwilę przed północą.

Czy coś jeszcze wymaga korekty? Warto byłoby dodać możliwość przestawiania czasu bez potrzeby resetu mikrokontrolera. Trzeba więc dodać sprawdzanie stanu klawiatury i w momencie wciśnięcia przycisku wczytać nowy czas. Zastanówmy się, gdzie najlepiej umieścić sprawdzanie, czy któryś z przycisków został wciśnięty. Przez praktycznie cały czas procesor znajduje się w pętli while() oczekującej na flagę OCF1A. Najlepiej więc odczyt klawiatury umieścić właśnie tam. Dzięki temu będzie ona ciągle sprawdzana. Inne miejsca, np. po linijce czyszczącej flagę, nadają się dużo mniej, ponieważ ich wykonanie następuje w odstępach jednosekundowych i użytkownik musiałby trzymać przycisk wciśnięty nawet przez sekundę, zależnie w który moment okresu 1-sekundowego akurat trafi. Rozbudujmy więc pętlę:

```
while(!(TIFR & _BV(OCF1A))) {
 if(readKeyboard()) {
  while (readKeyboard());
  ledClear();
  hours = readNumber(0, 2);
  minutes = readNumber(2, 2);
 }
}
```

W czasie oczekiwania na flagę OCF1A timera sprawdzamy, czy wciśnięty jest dowolny przycisk. Jeśli tak, czekamy na jego puszczenie a następnie czyścimy wyświetlacz i wczytujemy nowy czas.

W naszym programie możemy oczywiście dodać wiele innych funkcji, np. ustawianie sekund. Co prawda i tak ich nie będzie widać, ale zegar będzie mógł być ustawiony dokładniej i minuty będą się zmieniać we właściwych momentach. Bez tej funkcji, aby mieć czas dobrze zsynchronizowany z zegarem wzorcowym, trzeba by po wpisaniu minut zaczekać z ich zatwierdzeniem przyciskiem S16 aż do momentu, gdy w zegarze wzorcowym będą się zerowały sekundy. Oczywiście wpisana przy tym liczba minut musi być taka jak „nowa" minuta. A więc przykładowo o 20:07:40 wciskamy klawisze S2, S10, S16, S10, S8 i czekamy do 20:08:00, aby wcisnąć S16.

Ponieważ w programowaniu najważniejsze są samodzielne eksperymenty, zachęcam do wymyślania i dodawania nowych funkcji. Może to być np. odmierzanie czasu do tyłu, budzik czy funkcja daty. My tymczasem przejdziemy do zadania drugiego, które polegało na wyświetlaniu migającej kropki rozdzielającej godziny i minuty.

Najpierw zastanówmy się, jak wyświetlić kropkę. Trzeba by ją jakoś dodać do wyświetlanych minut. Niestety nie możemy napisać po prostu:

```
ledDispSymbol(LED_DOT, 2);
```

Jeśli wstawimy tę linijkę po linijkach wyświetlających czas, zniknie nam liczba jednostek godzin i pozostanie sama kropka. Jeśli umieścimy ją przed tymi linijkami, zostanie ona zaraz nadpisana przez liczbę godzin. Potrzebujemy funkcji zaświecającej nowe segmenty na wyświetlaczu obok już zaświeconych. Przydałaby się również analogiczna funkcja gasząca wybrane segmenty. Takie operacje już wykonywaliśmy, sterując pinami portów mikrokontrolera. Nie można ich bowiem ustawiać oddzielnie tylko trzeba zapisywać całą nową wartość rejestru PORTx. Aby ustawić jedynki na wybranych pinach, wykonywaliśmy opera-

cję OR na istniejącej wartości oraz nowej. Aby wyzerować wybrane piny, wykonywaliśmy operację AND na istniejącej wartości oraz zanegowanej nowej wartości. Przy segmentach możemy zrobić tak samo:

```
void ledTurnOnSegments(uint8_t segs, uint8_t position) {
 dispDigits[position] |= segs;
}

void ledTurnOffSegments(uint8_t segs, uint8_t position) {
 dispDigits[position] &= ~segs;
}
```

Za pomocą tych prostych funkcji możemy włączać i wyłączać wybrane segmenty. Funkcje te dodajemy do pliku led.c, a ich nagłówki do led.h. Parametr segs to bajt opisujący, które segmenty mają być włączone/wyłączone. Wystarczy ustawić jedynkę na danym bicie, aby odpowiadający mu segment został włączony/wyłączony. Kolejnym bitom, począwszy od najmłodszego, odpowiadają kolejne segmenty a–g, najstarszy bit odpowiedzialny jest za kropkę. Drugi parametr funkcji to pozycja od prawej, liczona od zera. Aby więc włączyć kropkę, po linijce wyświetlającej minuty napiszemy:

```
ledTurnOnSegments(0b10000000, 2);
```

Nasze nowe funkcje są bardzo zbliżone do funkcji ledDispRaw(), która również włącza wybrane segmenty, ale nie zachowuje segmentów już włączonych. Funkcja ta zawiera dodatkowo sprawdzenie, czy przekazana pozycja nie jest zbyt duża, bowiem mamy 4 wyświetlacze i prawidłowe pozycje to 0–3. Przekazanie wyższej wartości jako pozycji spowoduje zapis pamięci poza obszarem tablicy dispDigits i może spowodować nieprawidłowe działanie programu. Warto więc wzbogacić nasze nowe funkcje o walidację parametru pozycji:

```
void ledTurnOnSegments(uint8_t segs, uint8_t position) {
 if (position < sizeof(dispDigits)) dispDigits[position] |= segs;
}

void ledTurnOffSegments(uint8_t segs, uint8_t position) {
 if (position < sizeof(dispDigits)) dispDigits[position] &= ~segs;
}
```

Wyświetlanie kropki można też zrealizować w inny, dużo prostszy sposób. Wystarczy w ciągu formatującym liczbę zmienić literkę g na f:

```
ledDispNumber(hours, "%02f", 0);
```

Spowoduje to wyświetlanie liczby jako zmiennoprzecinkowej i tym samym dodanie kropki dziesiętnej. Jest to rozwiązanie najłatwiejsze, ale trzeba pamiętać, że korzystamy ze szczęśliwego zbiegu okoliczności, jakim jest wyświetlanie liczb za pomocą funkcji snprintf(). Gdyby był używany inny sposób, nie byłoby już tak łatwo. Dlatego funkcje ledTurnOnSegments() i ledTurnOffSegments() wcale nie były przez nas pisane na próżno. Poza tym mogą być używane w różnych miejscach programu, niezwiązanych z wyświetlaniem liczb.

A jak zrealizować miganie? Najłatwiej byłoby migać co sekundę. Wtedy dla sekund parzystych można by korzystać z literki formatującej g, a dla nieparzystych z literki f:

```
ledDispNumber(hours, seconds % 2 ? "%02f" : "%02g", 0);
```

Wykorzystany jest tutaj operator wyrażenia warunkowego składający się ze znaku zapytania oraz dwukropka. Przed znakiem zapytania znajduje się wyrażenie, którego wartość jest sprawdzana. Jeśli wyrażenie to zwróci wartość różną od zera, operator zwróci wartość stojącą przed dwukropkiem. Jeśli wyrażenie okaże się zerem, zwrócona zostanie druga wartość, stojąca za dwukropkiem. Nasze wyrażenie stosuje operator % (modulo) zwracający resztę z dzielenia. Tutaj zmienna seconds dzielona jest przez 2. Dla liczb parzystych resztą z dzielenia jest zero, a więc wybrany zostanie ciąg "%02g". Dla liczb nieparzystych resztą z dzielenia jest 1, co spowoduje zwrócenie ciągu "%02f". Zwrócony ciąg trafia jako drugi parametr do funkcji ledDispNumber(). Operator wyrażenia warunkowego pozwala tutaj sterować wywołaniem funkcji w dużo bardziej zwarty i zgrabny sposób, niż zrobiłoby to wyrażenie if – else. Jeśli chcielibyśmy skorzystać z funkcji ledTurnOnSegments(), napiszemy po prostu:

```
if (seconds % 2) ledTurnOnSegments(0b10000000, 2);
```

A czy można migać kropką z inną częstotliwością? Jeśli miałaby ona być mniejsza, przed wykonaniem operacji modulo możemy podzielić liczbę sekund, np.

```
if (seconds / 2 % 2) ledTurnOnSegments(0b10000000, 2);
```

Da nam to zmianę stanu kropki co 2 sekundy. A jak migać częściej niż co sekundę? Licznik naszego timera liczy do wartości 15624. Można by więc sprawdzać, czy doliczył np. do połowy, czyli 7812. Kod sprawdzający stan licznika musi być wtedy umieszczony w pętli czekającej na upłynięcie sekundy:

```
while(!(TIFR & _BV(OCF1A))) {
 if (TCNT1 > 7812) ledTurnOnSegments(0b10000000, 2);
 if(readKeyboard()) {
  while (readKeyboard());
  ledClear();
  hours = readNumber(0, 2);
  minutes = readNumber(2, 2);
 }
}
```

Takie rozwiązanie działa poprawnie, ale gdy licznik przekroczy wartość 7812, wówczas funkcja ledTurnOnSegments() jest ciągle wołana, wiele tysięcy razy w ciągu półsekundowego okresu, zupełnie niepotrzebnie. Choć w naszym przypadku nie ma to skutków ubocznych, a jest jedynie nieeleganckie, w innych programach takie zbędne wywoływanie funkcji może przeszkadzać. Warto więc zastanowić się, jak to włączanie kropki można zrobić lepiej.

Rozwiązaniem jest np. wprowadzenie zmiennej, która byłaby ustawiana na 1 po wykonaniu ledTurnOnSegments() i której stan byłby sprawdzany przed wywołaniem tej funkcji. W ten sposób zostałaby ona wykonana tylko raz. Oczywiście zmienną tę trzeba by zerować po odmierzeniu sekundy, czyli np. przy zerowaniu flagi OCF1A. Zamiast jednak tworzyć nową zmienną, możemy wykorzystać możliwości Timer1, a mianowicie generowanie flagi OCF1B. Wystarczy do rejestru OCR1B wpisać liczbę 7812, tak jak do OCR1A wpisaliśmy 15624. Wtedy zamiast wartości rejestru licznikowego TCNT1 możemy sprawdzać flagę OCR1B:

```
if(TIFR & _BV(OCF1B)) {
 TIFR |= _BV(OCF1B);
 ledTurnOnSegments(0b10000000, 2);     }
```

Gdy flaga ta zostanie ustawiona, zerujemy ją i włączamy kropkę. Do generowania szybkich mignięć można też zastosować inne podejście, np. taktować Timer1 szybciej. Jeśli w rejestrze TCCR1B nie ustawimy bitu CS10, a zostawimy tylko CS12 i WGM12, preskaler zostanie zmieniony z 1024 na 256. Timer będzie więc działał 4 razy szybciej. Wtedy przy ustawieniu się flagi OCF1A trzeba zwiększać dodatkową zmienną, która zliczałaby ćwiartki sekund i dopiero na jej podstawie przeliczane byłyby sekundy. Natomiast na podstawie tej zmiennej można by gasić i zaświecać kropkę. Rozwiązania są więc różne i trzeba wybrać najbardziej odpowiednie.

Ostatnim zadaniem było napisanie elektronicznej kostki do gry. Nie jest ono trudne, wystarczy modyfikować wartość zmiennej, gdy przycisk jest wciśnięty, a wyświetlać ją gdy jest puszczony (**listing 2**).

```
#include <avr/io.h>
#include <util/delay.h>
#include "keyb.h"
#include "led.h"

int main(void)
{
 ledInit();
 keybInit();
 ledClear();
 uint8_t value = 1;
 while(1)
 {
  if(readKeyboard()) {
value++;
  if (value == 7) value = 1;
 } else {
  ledDispNumber(value, "%g", 3); } }}
```

W kodzie mamy tylko jedną zmienną, więc może się nazywać po prostu value. W czasie wciśnięcia przycisku jest ona ciągle inkrementowna. Gdy osiągnie wartość 7, jest przestawiana z powrotem na 1. W ten sposób po puszczeniu przycisku otrzymujemy losową wartość z przedziału 1–6.

Zamek szyfrowy

W EdW opublikowano już kilka zamków szyfrowych, a pewien nietypowy zamek stał się głównym projektem pierwszego numeru miesięcznika 21 lat temu. My natomiast rozważmy zamek działający podobnie jak w domofonie, gdzie jest czterocyfrowy PIN a wciśnięcie ostatniej cyfry otwiera zamek. Rolę zamka będzie odgrywała dioda świecąca. Oczywiście zamiast niej można podłączyć np. tranzystor sterujący elektrycznym ryglem (**listing 3**).

```
#include <avr/io.h>
#include <util/delay.h>
#include "keyb.h"
#include "led.h"
const uint8_t PIN[] = {4, 6, 0, 1};
int main(void)
{
 ledInit();
 keybInit();
 DDRB |= _BV(DDB4);
 while(1)
 {
  ledClear();
  uint8_t match = 1;
  //wczytanie PINu
  for (uint8_t i = 0; i < 4; i++) {
   uint8_t input = getKey();
   if (input == 10) input = 0;
   if (input != PIN[i]) match = 0;
   ledDispSymbol(LED_MINUS, 3 - i);
  }
  //otwarcie zamka
  if (match) {
   PORTB |= _BV(PORTB4);
   _delay_ms(1000);
   PORTB &= ~_BV(PORTB4);
  } } }
```

Dioda będzie podłączona do pinu 4 portu B. W związku z tym po inicjalizacji klawiatury i wyświetlacza ustawiamy ten pin do pracy jako wyjście. W głównej pętli czyścimy wyświetlacz i ustawiamy na 1 zmienną oznaczającą wpisanie poprawnego PIN-u. Dlaczego z góry zakładamy poprawność PIN-u? Ponieważ przy wczytywaniu PIN-u każda niezgodność będzie zerowała tę zmienną. Wymagana jest zgodność wszystkich cyfr, aby zmienna match pozostała przy wartości 1. Przed porównaniem danej cyfry musimy sprawdzić czy nie został wciśnięty klawisz S10 oznaczający zero. Wtedy wartość 10 zmieniamy na 0. Ponieważ nie chcemy aby wpisywany PIN był widoczny, zamiast cyfr wyświetlane są minusy na kolejnych pozycjach wyświetlacza, od lewej strony. Po wpisaniu 4 cyfr, jeśli PIN był poprawny, na 1 sekundę ustawiany jest stan wysoki na pinie 4 portu B. Następnie cykl się powtarza.

Listing 4

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
#include <util/atomic.h>
#include "lcd.h"
volatile int32_t overflows = 0;
volatile uint8_t pulseNumber = 1;
volatile int32_t firstPulseTime = 0;
volatile int32_t secondPulseTime = 0;
volatile int32_t period = 0;

int main(void) {
 lcdInit();
 lcdInitPrintf();
 //włącz przerwania globalnie
 sei();
 //CLK/1
 TCCR1B = _BV(CS10);
 //przerwania na przepełnienie i input capture
 TIMSK = _BV(TOIE1) | _BV(TICIE1);
 while(1) {
  float f = 0;
  //obliczenie częstotliwości
  if (period) f = 16000000.0 / period;
  lcdGotoXY(0, 0);
  printf("%9.2f Hz", f);
  _delay_ms(500);
 }
}
ISR(TIMER1_OVF_vect){
 overflows++;
}
ISR(TIMER1_CAPT_vect){
 if (pulseNumber == 1) {
  firstPulseTime = (uint16_t)ICR1;
  pulseNumber = 2;
  overflows = 0;
 } else {
  secondPulseTime = (uint16_t)ICR1;
  pulseNumber = 1;
  period = secondPulseTime - firstPulseTime + overflows * 65536;
 } }
```

Miernik częstotliwości

W lekcji 4 jako ilustrację działania funkcji Input Capture oferowanej przez Timer1 napisaliśmy prymitywny miernik częstotliwości. Warto podjąć próbę napisania miernika bardziej dokładnego. Częstotliwość możemy mierzyć na dwa sposoby: albo klasycznie zgodnie z definicją zliczając impulsy w odcinku czasu, albo też mierząc okres sygnału i częstotliwość obliczając jako jego odwrotność. Dla niskich częstotliwości lepsza będzie ta druga metoda, ponieważ mikrokontroler, pracując z dużo większą częstotliwością, będzie w stanie w jednym okresie mierzonego sygnału zliczyć wiele taktów swojego zegara. Przy taktowaniu częstotliwością 1 MHz okres będzie mierzony z dokładnością 1 mikrosekundy. Dla częstotliwości mierzonej rzędu 1Hz da to rozdzielność jednej milionowej herca. Jednak dla sygnału 0,5MHz rozdzielczość wyniesie 100kHz, a więc błąd rzędu ±20%. Odwrotnie będzie przy zliczaniu impulsów (okresów) w zadanym czasie. Jeśli mikrokontroler będzie zliczał impulsy przez 1 sekundę, pomiar będzie miał rozdzielczość 1Hz. Dla dużych częstotliwości, np. 100kHz, jest to rozdzielczość bardzo dobra. Jednak dla częstotliwości rzędu pojedynczych herców błąd na poziomie 1 Hz będzie już błędem względnym na poziomie kilkudziesięciu %.

Zacznijmy od pomiaru małych częstotliwości, czyli od pomiaru okresu. Wykonywaliśmy to właśnie w lekcji 4, korzystając z funkcji Input Capture. Pomiar odbywał się w całości podczas jednego cyklu pracy licznika, podczas którego musiały przyjść dwa impulsy sygnału mierzonego. Nie można więc było mierzyć sygnałów o okresie dłuższym niż okres, w jakim licznik timera ulegał przepełnieniu. Ponadto, aby zwiększyć rozdzielczość pomiaru, trzeba by zwiększyć szybkość taktowania, co również spowodowałoby zbyt szybkie przepełnienie się licznika timera. A może przepełnienia nie są wcale przeszkodą i mogą okazać się pomocne? Jeśli przyjdzie impuls przy jakimś stanie licznika timera, następnie nastąpi kilka przepełnień licznika timera, a potem przyjdzie drugi impuls przy innym stanie licznika timera, to aby obliczyć czas pomiędzy impulsami, brakuje nam właśnie liczby tych przepełnień. A przepełnienia możemy zliczyć, bo są sygnalizowane flagami i przerwaniami. Spójrzmy na przykładową realizację (**listing 4**). Założone tutaj jest taktowanie kwarcem 16 MHz.

Program na początku inicjalizuje wyświetlacz i funkcję printf(). Następnie włączane są przerwania. Timerem wykorzystywanym do pomiaru częstotliwości jest Timer1. Za pomocą bitu CS10 w rejestrze TCCR1B preskaler timera ustawiany jest na 1, czyli timer taktowany jest z pełną częstotliwością zegara mikrokontrolera. Za pomocą rejestru TIMSK zostają włączone przerwania na przepełnienie licznika timera i dla funkcji Input Capture. Główna pętla to pobranie zmierzonego okresu ze zmiennej globalnej period, obliczenie częstotliwości i wyświetlenie jej na LCD. Skąd bierze się wartość zmiennej period? Aby obliczyć okres sygnału, sprawdzamy stan licznika timera w momencie wystąpienia kolejnych zboczy narastających mierzonego sygnału. Zapamiętanie stanu licznika wykonuje funkcja Input Capture timera, która stan ten zapisuje w rejestrze ICR1. Generowane jest wtedy również przerwanie, dzięki któremu możemy stan tego rejestru od razu skopiować do zmiennych, raz firstPulseTime a raz secondPulseTime. Przy drugim razie następuje obliczenie okresu przez odjęcie poprzedniego stanu licznika. Ponieważ przy niskiej częstotliwości sygnału licznik może się jeden lub więcej razy przepełnić, obliczenie musi brać pod uwagę także przepełnienia. Ponieważ licznik jest 16-bitowy, przepełnia się on co 65 536 cykli zegara taktującego. Każde więc przepełnienie wymaga dodania tylu właśnie cykli.

Ponieważ korzystamy z funkcji Input Capture, sygnał wejściowy musimy dostarczyć do pinu ICP1 (PD6) mikrokontrolera. Tak jak w poprzednich ćwiczeniach z pomiarem częstotliwości, do testów możemy skorzystać z generatora sygnału prostokątnego, którego wyjście jest dostępne na pinie GO złącza MISC na płytce testowej. Ze względu na dużą rezystancję wejściową portów mikrokontrolera pin PD6 można po prostu dotknąć palcem i wówczas nasz miernik pokaże częstotliwość ok. 50Hz. Nasze ciało stanie się bowiem anteną odbierającą przydźwięk sieci.

Napiszmy jeszcze miernik częstotliwości według drugiego podejścia, w którym przez

jedną sekundę będziemy zliczać nadchodzące impulsy mierzonego sygnału. Czym możemy je zliczać? Oczywiście najprościej licznikiem. Liczniki taktowane zewnętrznym sygnałem znajdziemy w Timer0 oraz Timer1. Ponieważ Timer1 dobrze nadaje się do odliczania okresów 1-sekundowych, impulsy sygnału wejściowego będziemy zliczać za pomocą Timer0. W przypadku tego timera sygnał wejściowy podaje się na pin T0 (PB0). Spójrzmy na kod (**listing 5**).

Struktura programu jest bardzo podobna do poprzedniej: inicjalizacja timerów, główna pętla zajmująca się wyświetlaniem wyniku oraz dwie funkcje obsługi przerwań. Timer0 konfigurowany jest do bycia taktowanym zboczem narastającym na pinie T0 (lekcja 3, tabela 1). Można też skonfigurować zbocze opadające, nie ma to znaczenia w tym przykładzie. Timer1 działa w trybie CTC, podobnie jak w wielu poprzednich przykładach. Włączone są dwa przerwania: na przepełnienie licznika Timer0 oraz dla funkcji Output Compare Timer1. Przerwanie od Output Compare następuje co sekundę. Analogicznie do poprzedniego przykładu pobieramy stan licznika i dodajemy przepełnienia. Tym razem mnożnikiem jest 256, bo licznik w Timer0 jest 8-bitowy. Tutaj od razu dostajemy częstotliwość, ponieważ Timer0 zlicza przychodzące impulsy, a nie takty zegara wzorcowego, jakim wcześniej był zegar mikrokontrolera. Wynik przechowywany w zmiennej frequency jest wyświetlany w głównej pętli while().

Listing 5

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
#include <util/atomic.h>
#include "lcd.h"

volatile uint32_t frequency = 0;
volatile uint32_t overflows = 0;

int main(void) {
 lcdInit();
 lcdInitPrintf();
 //włącz przerwania globalnie
 sei();
 //sygnał zegarowy na T0
 TCCR0 = _BV(CS02) | _BV(CS01) | _BV(CS00);
 //CLK/1024
 TCCR1B = _BV(WGM12) | _BV(CS12) | _BV(CS10);
 OCR1A = 15624;
 //przerwania na przepełnienie Tim0
   i output capture Tim1
 TIMSK = _BV(TOIE0) | _BV(OCIE1A);
 while(1) {
  lcdGotoXY(0, 0);
  printf("%6g Hz", (double)frequency);
  _delay_ms(500);
 }
}

ISR(TIMER0_OVF_vect){
 overflows++;
}

ISR(TIMER1_COMPA_vect){
 frequency = TCNT0 + overflows * 256;
 overflows = 0;
 TCNT0 = 0;
}
```

Jak powiedzieliśmy na początku, oba programy różnią się pod względem dokładności pomiaru w zakresie małych i dużych częstotliwości. Obserwując ich pracę, można też zauważyć inny szczegół: jeśli badamy niezbyt stabilny generator, jak np. ten znajdujący się na płytce testowej, w przypadku pierwszego programu ta niestabilność jest wyraźnie widoczna, wynik na wyświetlaczu ciągle się zmienia. W przypadku drugiego programu wynik jest dużo bardziej stabilny. Drugi program stosując okres pomiaru o długości 1 sekundy, uśrednia wynik. Jeśli np. mamy sygnał 10kHz, wynik będzie średnią z 10 000 impulsów. W pierwszym programie pomiar trwa tyle, ile wynosi pół okresu sygnału, nie ma więc żadnego uśredniania. Należy tutaj podkreślić, że nie ma to związku z półsekundowym opóźnieniem występującym w pętli wyświetlającej wynik.

Zadanie

Tym razem zadanie domowe jest jedno: napisać prostą grę zręcznościową. Choć do gier dobrze byłoby mieć wyświetlacz graficzny, można też wykorzystać to, co jest dostępne na płytce testowej, np. alfanumeryczny LCD. To tylko 2x16 znaków, ale można za ich pomocą zasymulować choćby drogę z przeszkodami. Niech na pierwszej pozycji będzie wyświetlony jakiś znak, np. „>” symbolizujący pojazd użytkownika. Za pomocą klawiatury można go przenosić między górną i dolną linią, ciągle pozostając w skrajnej lewej kolumnie. Natomiast z prawej strony wyświetlacza będą co jakiś czas pojawiać się znaki symbolizujące przeszkody. Znaki te przesuwać się będą w lewo, w stronę „pojazdu” gracza, który będzie musiał je omijać, przeskakując między górną i dolną linią. Jeśli jeden z przesuwających się w lewo znaków znajdzie się na tej samej pozycji co gracz, nastąpi koniec gry.

Grę można w różny sposób uatrakcyjnić, np. wprowadzając też znaki, które trzeba zbierać/łapać i za które będą przyznawane punkty. W grze można wykorzystać znaki ASCII, takie jak nawiasy, literka O, symbole matematyczne i dowolne inne, lub też zdefiniować własne, bardziej pasujące do gry. Grę można wzbogacić również o efekty świetlne (LED-y) czy dźwiękowe (brzęczyk piezo).

Żeby gra nie była nudna, musi być nieprzewidywalna. Pojawiające się znaki powinny pojawiać się w losowych momentach na losowych pozycjach. Do generowania liczb pseudolosowych w języku C służy funkcja rand() zwracająca typ int. Wymaga ona inicjalizacji, aby nie generowała za każdym razem tego samego ciągu. Trzeba więc i tak wziąć skądś choć jedną losową liczbę. Można w tym celu np. poprosić gracza o wciśnięcie na początku gry przycisku na krótką chwilę i zmierzyć, ile mikrosekund ta chwila trwała. Liczbę tę przekazujemy do funkcji srand(). Obie funkcje wymagają dołączenia biblioteki stdlib.h. Na początku kodu napiszemy więc:

```
#include <stdlib.h>
```

Z kolei na początku programu, po kodzie obsługującym wciśnięcie przycisku, wstawiamy:

```
srand(keyPressTime);
```

gdzie keyPressTime to przykładowa zmienna przechowująca zmierzony czas wciśnięcia przycisku. Natomiast kolejne pseudolosowe liczby uzyskamy przez wywołanie:

```
randomNumber = rand();
```

Jeśli potrzebować będziemy liczb z wąskiego zakresu, otrzymaną losową liczbę wystarczy odpowiednio podzielić. Można też stosować operację modulo, jak to robiliśmy w tej lekcji.

W grze jest także ważne, aby dało się w nią wygrać. Jeśli nasz program będzie pozwalał, aby pojawiły się dwie przeszkody w tej samej kolumnie, gracz nie będzie w stanie ich ominąć. Kod generujący przeszkody nie może więc umieszczać ich na ekranie zupełnie losowo, musi eliminować układy, które będą niemożliwe do ominięcia. Chyba że dodamy możliwość strzelania do przeszkód, tu już ograniczeniem jest tylko nasza wyobraźnia.

Do napisania gry bardzo zachęcam. Zadanie tego typu pozwala z jednej strony oswoić się ze znanymi już elementami, jak LCD czy timery, a z drugiej strony stawia różne nowe wyzwania, jak choćby zapewnienie, żeby gra nie była zbyt trudna. Ponadto nawet do prostej gry można dodawać coraz to nowe ciekawe funkcje. Warto spróbować, nawet jeśli programowanie mikrokontrolera wciąż sprawia nam problemy. Zacząć można od najprostszej rzeczy: wyświetlenia losowej literki. Potem wyświetlić więcej literek, a następnie cały ekran przesunąć. Gdy to się uda pomyśleć, jak przesunąć cały ekran z wyjątkiem wybranej literki. W kolejnym kroku dopisać przesuwanie literki po ekranie. Najważniejsze, żeby samodzielnie próbować, nawet pisząc najprostsze funkcje. Przykładowe rozwiązanie jak zwykle zamieszczone zostanie w kolejnym numerze EdW.

Grzegorz Niemirowski
grzegorz@grzegorz.net