Kurs AVR – lekcja 9

Rozwiązania zadań z ostatniego odcinka

Tematem pierwszego zadania było wygenerowanie na nóżce mikrokontrolera częstotliwości n-razy niższej niż występująca na innej nóżce. Aby program zależał od podawanej z zewnątrz częstotliwości, można wykorzystać funkcję taktowania timera zewnętrznym sygnałem. Załóżmy, że wykorzystamy Timer0. Aby był on taktowany zewnętrznym zegarem, trzeba ustawić bity CS1 i CS2 w rejestrze TCCR0, zgodnie z opisem w lekcji 3. Potrzebujemy też przerwania co określoną liczbę taktów timera. Tutaj użyć można funkcji Output Compare, również znanej z lekcji 3. Wymaga ona ustawienia bitu WGM01, również w rejestrze TCCR0. Cały kod pokazuje **listing 1**.

```
#include <avr/io.h>
#include <avr/interrupt.h>

int main(void)
{
 sei();
 DDRB = _BV(DDB1);
 TCCR0 = _BV(CS02) | _BV(CS01) | _BV(WGM01);
 TIMSK = _BV(OCIE0);
 OCR0 = 4;
 while(1) {}
}

ISR(TIMER0_COMP_vect) {
static uint8_t state = 0;
 if (state) {
  PORTB |= _BV(PORTB1);
 } else {
  PORTB &= ~_BV(PORTB1);
 }
 state = !state;
}
```

Listing 1

Nóżką przyjmującą sygnał jest PB0. Sygnał wyjściowy może być generowany na dowolnym pinie, tutaj jest to PB1. W funkcji obsługi przerwania przestawiany jest stan pinu w zależności od zmiennej state, która za każdym przerwaniem ma przestawianą wartość na przeciwną. Jest to zmienna statyczna, czyli zachowująca swoją wartość pomiędzy wywołaniami funkcji. Wartość wpisywana do rejestru OCR0, będącego rejestrem funkcji Output Compare, wyznacza stopień podziału. Liczba 4 da podział przez 10, ponieważ przerwanie będzie generowane co 5 taktów, a każda zmiana stanu pinu wyznacza początek jednego półokresu. Z kolei np. podział przez 20 będzie wymagał liczby 9. Maksymalny podział to 512, ponieważ rejestry Timer0 są 8-bitowe. Dla większych stopni podziału można sięgnąć po 16-bitowy Timer1 lub wykorzystać dodatkową zmienną zliczającą przerwania.

Zadanie drugie polegało na generowaniu dwóch niezależnych częstotliwości. Funkcję tę zrealizować można za pomocą dwóch timerów i ich przerwań, pochodzących z przepełnienia lub Output Compare. W przypadku jednego timera można zliczać przerwania za pomocą zmiennej i np. jeden pin przestawiać co 3 przerwania, a drugi co pięć. Trudniej jednak w takim przypadku będzie mówić o niezależności częstotliwości.

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

uint8_t dispDigits[4] = {0, 0, 0, 0};

int main(void)
{
 sei();
 TCCR0 = _BV(CS00) | _BV(CS01); // CLK/64
 TIMSK = _BV(TOIE0); // przerwanie na przepełnienie
 DDRA = 0xff;
 DDRB |= _BV(PB3) | _BV(PB2) | _BV(PB1) | _BV(PB0);
 DDRB |= _BV(PB6);
 dispDigits[1] = 0b01001111; // cyfra 3 na wyświetlaczu DISP2
 while (1)
 {
  _delay_ms(200);
  PORTB |= _BV(PB6);
  _delay_ms(200);
  PORTB &= ~_BV(PB6);
 }
}

ISR(TIMER0_OVF_vect)
{
 static uint8_t dispNum = 0;
 PORTB |= 0x0f;
 PORTB &= ~_BV(dispNum);
 PORTA = ~dispDigits[dispNum];
 dispNum++;
 dispNum %= 4;
}
```

Listing 2

Sterowanie wyświetlaczem LED

Wiemy już, jak działają przerwania i poznaliśmy niektóre niuanse związane z ich obsługą. Naszą wiedzę wykorzystaliśmy do prostego odmierzania czasu i migania LED-ami. Czas na wykorzystanie przerwań do czegoś bardziej praktycznego.

Na naszej płytce testowej znajdują się cztery wyświetlacze 7-segmentowe LED, tworząc jeden duży wyświetlacz czterocyfrowy. Pojedynczym wyświetlaczem sterowaliśmy już w lekcji 2. Sterowanie jednak wszystkimi naraz wymaga multipleksowania ze względu na ograniczoną liczbę pinów mikrokontrolera. Sterowanie multipleksowe polega na tym, że aktywujemy jeden wyświetlacz i włączamy odpowiednie jego segmenty. Następnie wyłączamy go i włączamy segmenty drugiego wyświetlacza. Robimy tak też z wyświetlaczem trzecim i czwartym, a następnie zaczynamy od początku. W danej chwili świecą segmenty tylko jednego wyświetlacza, ale jeśli będziemy przełączać się między wyświetlaczami odpowiednio szybko, oko ludzkie tego nie wychwyci i wrażenie będzie takie, jakby wszystkie wyświetlacze działały jednocześnie, bez migotania. Do sterowania czterema wyświetlaczami wystarczy 12 pinów mikrokontrolera: osiem dla segmentów (razem z kropką) oraz 4 do włączania wyświetlaczy.

Na naszej płytce testowej dostępne jest złącze LED_DISP, które ma właśnie 12 pinów pozwalających na sterowanie wyświetlaczami LED. Segmenty (katody) połączone są z pinami oznaczonymi literami A–G oraz kropką. Z kolei tranzystory PNP podłączające anody do plusa zasilania są dostępne na pinach 1–4. Aby włączyć dany segment danego wyświetlacza, trzeba ustawić niski stan logiczny na pinie tego segmentu oraz niski stan logiczny na pinie wyświetlacza. Przykładowo zaświecenie środkowego poziomego segmentu na wyświetlaczu drugim od lewej (DISP3) wymaga podania zera (czyli zwarcia do masy) pinów G i 3.

Jak wspomnieliśmy, sterowanie multipleksowe wymaga szybkiego przełączania wyświetlaczy. Jest to trudne do zrealizowania, jeśli mikrokontroler zajmuje się jednocześnie innymi rzeczami. Tutaj z pomocą przychodzą właśnie przerwania. Konfigurując odpowiednio częste przerwanie w timerze, możemy sterować wyświetlaczem LED niezależnie od reszty programu. Popatrzmy na prosty przykład, pokazany na **listingu 2**.

Do sterowania wyświetlaczem wykorzystujemy Timer0. Taktowany jest on częstotliwością mikrokontrolera podzieloną przez 8. Przerwanie generowane jest w momencie przepełnienia licznika, czyli z częstotliwością 16000000 / 8 / 256 = 7812,5Hz. Może się ona wydawać zbyt duża, ale przy tych ustawieniach timera migotanie nie będzie widoczne, nawet jeśli zmniejszymy taktowanie mikrokontrolera z 16 MHz na 1 MHz. Do sterowania segmentami wyświetlaczy wykorzystany jest port A, natomiast do wyboru wyświetlacza wykorzystywane są piny 0–3 portu B. Zarówno włączenie/wybór wyświetlacza, jaki włączenie segmentu wykonywane są niskim stanem logicznym. Wynika to z tego, że wyświetlacze na płytce testowej są wyświetlaczami ze wspólną anodą i sterowane są od strony katod. Z kolei anody dołączane są do plusa przez tranzystory PNP włączane logicznym zerem.

W funkcji obsługującej przerwanie zadeklarowana jest statyczna zmienna dispNum przechowująca numer aktualnego wyświetlacza. W zależności od jej wartości na odpowiednim pinie ustawiane jest zero. Wcześniej na pozostałych 4 pinach sterujących wyświetlaczem ustawiane są jedynki, aby wyłączyć ewentualnie włączone wyświetlacze. Następnie

```
void ledDispNumber(float num, char format[], uint8_t shift) {
 uint8_t dispNum = 3 - shift;
 char buff[6];
 snprintf(buff, sizeof(buff), format, num);
 buff[sizeof(buff) - 1] = 0;
 for (uint8_t i = 0; i < strlen(buff); i++) {
  char ch = buff[i];
  if (ch >= '0' && ch <= '9') dispDigits[dispNum] = segments[ch - '0'];
  if (ch == '-') dispDigits[dispNum] = segments[LED_MINUS];
  if (ch == '.') {
   dispDigits[dispNum + 1] |= segments[LED_DOT];
  } else {
   if (dispNum > 0) dispNum--; else return;
  }
 }
}
```

Listing 3

z globalnej tablicy dispDigits wybierany jest bieżący element i wystawiany na porcie A w postaci zanegowanej, ze względu na sterowanie od strony katod. Gdy już odpowiednie segmenty są wysterowane, zwiększana jest zmienna z numerem aktualnego wyświetlacza. Gdy osiągnie wartość 4, jest zerowana. Stosowany jest tutaj operator modulo, czyli reszta z dzielenia, mający w języku C symbol %.

W głównym programie tablica dispDigits inicjowana jest zerami, a następnie do elementu drugiego (z numerem 1) wpisywany jest bajt z bitami odpowiadającymi zaświeconym segmentom układającym się na wyświetlaczu w cyfrę 3. W głównej pętli miga dioda. Wyświetlacz odświeżany jest niezależnie, pobiera tylko wartości z dispDigits.

Warto zauważyć, że tablica dispDigits jest swego rodzaju interfejsem pomiędzy głównym programem a kodem obsługującym wyświetlacz. Pomijając kwestię inicjalizacji, sterowanie wyświetlaczem odbywa się bez wywoływania funkcji, tylko przez zapis do tablicy. Docelowo jednak tego typu interfejs nie zawsze jest pożądany, uzasadnienie poznamy w dalszej części lekcji.

Aby przetestować omawiany przykład, należy utworzyć w Atmel Studio nowy projekt i wkleić kod do main.c. Oczywiście jak zwykle nie zapominamy o zdefiniowaniu makra F_CPU o wartości odpowiadającej aktualnie ustawionej fusebitami częstotliwości taktowania mikrokontrolera. Kolejne piny portu A łączymy z kolejnymi pinami segmentów na złączu LED_DISP, czyli piny 0–6 portu z pinami A–G złącza LED_DISP i pin 7 z pinem oznaczonym kropką. Z kolei piny 0–3 portu B łączymy z pinami podpisanymi cyframi 1–4 na złączu LED-DISP.

Wyświetlanie liczb

Napisany przez nas program jest prostą demonstracją sterowania multipleksowego wyświetlaczem LED. Jeśli chcielibyśmy wykorzystać go np. do napisania zegara, konieczna będzie jego rozbudowa o funkcje do wygodnego wyświetlania pojedynczych cyfr oraz całych liczb.

Teraz w naszym przykładzie, aby wyświetlić cyfrę 3, ustawiliśmy bezpośrednio określone bity tak, aby odpowiadające im segmenty ułożone były w kształt cyfry 3. Aby ułatwić sobie życie i nie musieć wgłębiać się w układanie segmentów, możemy skorzystać ze sposobu z lekcji 2, czyli tablicy zawierającej definicje segmentów dla wszystkich 10 cyfr. Przypomnijmy jak wyglądała ta tablica:

```
const uint8_t segments[10] =
{
 0b00111111,
 0b00000110,
 0b01011011,
 0b01001111,
 0b01100110,
 0b01101101,
 0b01111101,
 0b00000111,
 0b01111111,
 0b01101111
};
```

Za jej pomocą wyświetlenie cyfry 3 na drugim wyświetlaczu możemy zrealizować w wygodniejszy sposób:

```
dispDigits[1] = segments[3];
```

Wyświetlacze 7-segmentowe mają ograniczone możliwości prezentowania symboli innych niż cyfry. Mimo to da się wyświetlić m.in. takie znaki jak kropka, minus, podkreślenie czy niektóre litery: A, b, c, C, d, E, F, h, H, J, L, Ł, n, o, P, q, r, S, u, U, y. Dla pozostałych liter trudno jest znaleźć takie układy segmentów, które byłyby wystarczająco czytelne oraz nie myliłyby się z cyframi (np. l, I, 1). W każdym razie naszą 10-elementową tablicę możemy rozszerzyć o potrzebne nam symbole. Dodajmy np. minus oraz kropkę, dopisując na końcu, przed nawiasem zamykającym, dwie linijki:

```
0b01000000,
0b10000000
```

Ponieważ tablica będzie miała teraz 12 elementów, w jej deklaracji trzeba zmienić rozmiar z 10 na 12 lub po prostu nawiasy kwadratowe zostawić puste. Symbol minus otrzymał indeks 10, a kropka 11. Aby nie musieć pamiętać indeksów, możemy stworzyć makra:

```
#define LED_MINUS 10
#define LED_DOT 11
```

Załóżmy, że chcemy wyświetlić liczbę –2,5. Zrobimy to w następujący sposób:

```
dispDigits[3] = segments[LED_MINUS];
dispDigits[2] = segments[2];
dispDigits[1] = segments[LED_DOT];
dispDigits[0] = segments[5];
```

W ten sposób wyświetlona liczba nie wygląda jednak zbyt dobrze, bo jest duży odstęp między dwójką a kropką. Byłoby estetyczniej, gdyby włączona była kropka na tym samym wyświetlaczu co dwójka, a nie w kolejnym. Trzeba więc w danym wyświetlaczu włączyć segmenty zarówno od dwójki, jak i od kropki:

```
dispDigits[3] = 0;
dispDigits[2] = segments[LED_MINUS];
dispDigits[1] = segments[2] | segments[LED_DOT];
dispDigits[0] = segments[5];
```

W ten sposób skrajny lewy wyświetlacz jest wygaszony, na drugim wyświetlaczu jest minus, na trzecim dwójka wraz z kropką, a na czwartym cyfra 5.

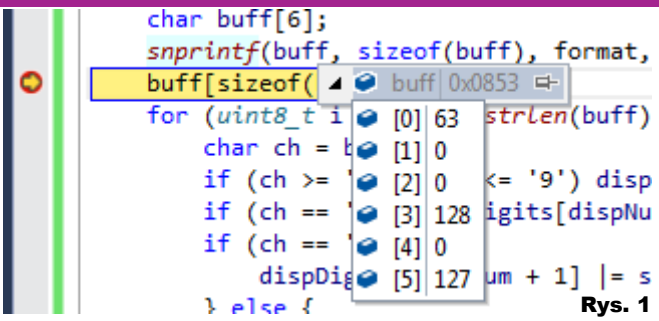
Mamy opanowane wyświetlanie symboli na poszczególnych wyświetlaczach, nadal jednak nie mamy prostego sposobu na wygodne wyświetlanie całych liczb ze zmiennych typu int czy float. Z pomocą przyjść może znana już nam funkcja sprintf(), dzięki której liczbę możemy zmienić na ciąg znaków. Na podstawie uzyskanych znaków możemy następnie włączać odpowiednie segmenty wyświetlaczy. Na **listingu 3** znajduje się przykładowa implementacja funkcji wyświetlającej liczby.

Funkcja ledDispNumber() przyjmuje trzy parametry: liczbę do wyświetlenia, format oraz przesunięcie mówiące, od którego miejsca od lewej ma się rozpocząć wyświetlanie. Jako liczbę do wyświetlenia przekazujemy float, ale może być to też typ całkowity, konwersja odbędzie się automatycznie. Format to ciąg znaków określający, jak ma wyglądać liczba, np. ile ma mieć pozycji po przecinku. Zasady tworzenia tych ciągów poznaliśmy w lekcji 6. Dla wyświetlacza LED najbardziej przydatny będzie ciąg %g. Dla liczb całkowitych zachowuje się jak %d, zwracając ewentualny znak minus oraz cyfry części całkowitej. Dla liczb ułamkowych dodaje też kropkę dziesiętną i cyfry części ułamkowej (**tabela 1**).

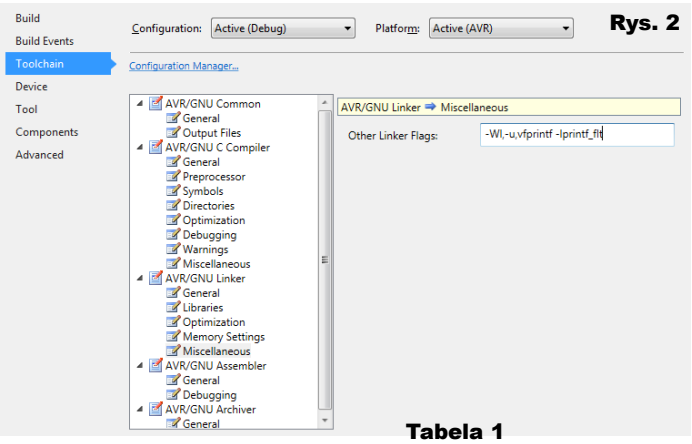
Parametr shift wynoszący zero oznacza wyświetlanie od pierwszego wyświetlacza z lewej strony na płytce testowej, czyli od DISP4. Ponieważ w naszym kodzie wyświetlacze numerowane są od zera od prawej strony, aby uzyskać numer początkowego wyświetlacza, parametr shift odejmowany jest od liczby 3.

Przekazana liczba jest konwertowana na ciąg znaków za pomocą funkcji snprintf(), która działa jak sprintf(), ale pobiera rozmiar tablicy wynikowej, co pozwala uniknąć jej przepełnienia. Ponieważ mamy 4 wyświetlacze, razem z kropką możemy przetworzyć 5 znaków. Tablica buff deklarowana jest jako 6-elementowa ze względu na zero kończące ciąg.

Główna część naszej funkcji to pętla iterująca po kolejnych znakach ciągu. Jeśli znak jest cyfrą, wyświetlamy go



Rys. 1



Rys. 2

Tabela 1

Liczba	%g	%.2g	%f	%.2f
2	2	2	2.000000	2.00
-2,5	-2.5	-2.5	-2.500000	-2.50

Listing 4

```
void ledInit() {
  sei();
  TCCR0 = _BV(CS01); // Timer0 CLK/8
  TIMSK = _BV(TOIE0); // przerwanie na przepełnienie
  DDRA = 0xff;
  DDRB |= _BV(PB3) | _BV(PB2) | _BV(PB1) | _BV(PB0);
}
```

Listing 5

```
 sei();
 TCCR0 = _BV(CS01); // Timer0 CLK/8
 TIMSK = _BV(TOIE0); // przerwanie na przepełnienie
 LED_SEG_DDR = 0xff;
 LED_DISP_DDR |= _BV(LED_DISP4_PIN) | _BV(LED_DISP3_PIN)
  | _BV(LED_DISP2_PIN) | _BV(LED_DISP1_PIN);
}
```

na bieżącym wyświetlaczu. W tym celu z tablicy segments wybierany jest układ segmentów danej cyfry i wpisywany do tablicy dispDigits pod indeksem bieżącego wyświetlacza. Od znaku (jego kodu ASCII) odejmowany jest kod ASCII cyfry 0, aby w ten sposób skonwertować cyfrę (znak) na liczbę 0–9. Jeśli znak nie był cyfrą, ale znakiem minus, włączany jest odpowiedni segment. Natomiast w przypadku kropki włączany jest segment kropki poprzedniego wyświetlacza (mającego wyższy numer, stąd dispNum + 1). Jeśli wyświetlony znak nie był kropką i nie był to ostatni wyświetlacz, przechodzimy do następnego, dekrementując zmienną dispNum.

Jak używać naszej funkcji? Nie jest to trudne:

```
ledDispNumber(-2.5, "%g", 0);
```

Powyższa przykładowa linijka spowoduje wyświetlenie liczby –2,5, począwszy od pierwszego (lewego) wyświetlacza. Na pierwszym wyświetlaczu będzie minus, na drugim dwójka z kropką, na trzecim piątka, a ostatni będzie pusty. Jeśli trzecim parametrem będzie 1, a nie 0, cała liczba zostanie przesunięta w prawo i będzie zaczynać się od drugiego wyświetlacza.

Tak przynajmniej jest w teorii. Kompilujemy kod, programujemy mikrokontroler i… nic. Program działa, bo miga LED, ale wyświetlacz jest pusty. Co może być nie tak? Nasza funkcja opiera się na wywołaniu funkcji snprintf(), może tutaj jest coś nie tak? Korzystając z JTAG-a ustawiamy punkt przerwania (breakpoint) na wywołaniu tej funkcji i uruchamiamy program. Gdy wykonanie zatrzyma się na punkcie przerwania, sprawdzamy parametry format i num. Okazują się one w porządku. Z kolei przekazywana tablica buff ma odpowiedni rozmiar. Naciskamy F10, aby funkcja snprintf() się wykonała i aby program zatrzymał się na następnej linijce. Zaglądamy do tablicy buff (**rysunek 1**). W pierwszym jej elemencie jest wartość 63 (kod ASCII znaku zapytania), potem zera i inne wartości. Jako że zero kończy ciąg, buff przechowuje de facto jednoznakowy napis „?”. Co mogliśmy zrobić źle? snprintf() nie powinien zwracać czegoś takiego.

Otóż problem wcale nie leży w naszym kodzie. Nie działa on, ponieważ domyślnie linkowane biblioteki nie zawierają funkcji obsługujących liczby zmiennoprzecinkowe. W związku z tym musimy zmienić ustawienia linkera. Wchodzimy w tym celu do właściwości naszego projektu, do sekcji Toolchain -> AVR/GNU Linker -> Miscellaneous (**rysunek 2**). Mamy tam pole, do którego możemy wpisać dodatkowe parametry dla linkera. Wpisujemy:
-Wl,-u,vfprintf -lprintf_flt

Kompilujemy kod, programujemy mikrokontroler i tym razem wszystko działa poprawnie, liczba jest widoczna na wyświetlaczu. O dodaniu tych parametrów trzeba pamiętać także przy wykorzystaniu snprintf() czy printf() do obsługi LCD, gdy wyświetlane mają być liczby zmiennoprzecinkowe.

Biblioteka wyświetlacza LED

Zanim przejdziemy dalej, wydzielmy nasz kod obsługujący wyświetlacz LED do oddzielnego pliku. Zyskamy porządek i większą kontrolę nad kodem. W tym celu stwórzmy plik led.c oraz plik nagłówkowy led.h. Do pliku led.c przenieśmy funkcje ledDispNumber() oraz ISR(TIMER0_OVF_vect). Dobrze byłoby też stworzyć funkcję inicjalizującą nasz sterownik wyświetlacza LED. Umieścimy tam konfigurację timera, inicjalizację przerwania oraz konfigurację wykorzystywanych pinów. W pierwszym podejściu funkcja inicjalizująca może wyglądać jak na **listingu 4.**

Dobrze byłoby jednak, aby można było łatwo zmieniać porty, do których podłączony jest wyświetlacz, tak jak było to w przypadku LCD i klawiatury. Stwórzmy więc odpowiednie makra, analogicznie jak wtedy:

```
#define LED_DISP_PORT PORTB
#define LED_SEG_PORT PORTA
#define LED_DISP_DDR DDRB
#define LED_SEG_DDR DDRA
#define LED_DISP1_PIN PB0
#define LED_DISP2_PIN PB1
#define LED_DISP3_PIN PB2
#define LED_DISP4_PIN PB3
```

W ten sposób możemy przepisać naszą funkcję inicjalizacyjną , jak na **listingu 5**.

A co z resztą biblioteki? Odwołania do portów mamy jeszcze w funkcji obsługi przerwania. Nasze makra się tam nie przydadzą, bo obecnie funkcja ta zakłada, że sterowane będą cztery młodsze piny portu, a nie dowolne. Co prawda zdefiniowaliśmy makra pinów jako PB0, PB1, PB2 i PB3, ale to tylko przykład. Moglibyśmy potrzebować wykorzystać np. PB2, PB4, PB6 i PB7. Co wtedy? Do przepisania mamy dwie linijki:

```
PORTB |= 0x0f;
PORTB &= ~_BV(dispNum);
```

Pierwsza wyłącza wszystkie wyświetlacze, a druga włącza wyświetlacz bieżący. Wyłączenie wyświetlaczy to wpisanie jedynek na sterujących nimi pinach, jest więc proste:

```
LED_DISP_PORT |= _BV(LED_DISP4_PIN) | _BV(LED_DISP3_PIN)
 | _BV(LED_DISP2_PIN) | _BV(LED_DISP1_PIN);
```

Z drugą linijką jest gorzej, ponieważ nie możemy zakładać, że numer pinu będzie taki sam jak numer wyświetlacza przechowywany w zmiennej dispNum. Potrzebujemy więc tablicy, która zmapuje nam numer wyświetlacza na numer pinu:

```
const uint8_t dispPin[] = {LED_DISP1_PIN, LED_DISP2_PIN, LED_DISP3_PIN, LED_DISP4_PIN};
```

Dzięki temu aktywację wyświetlacza możemy zapisać w ten sposób:

```
LED_DISP_PORT&=~_BV(dispPin[dispNum]);
```

Z kolei modyfikacja w linijce sterującej segmentami będzie już tylko prostym podstawieniem makra w miejsce konkretnego portu:

```
LED_SEG_PORT=~dispDigits[dispNum];
```

To, co nam pozostało do przeniesienia do biblioteki, to deklaracje tablic dispDigits oraz segments, a także definicje makr LED_MINUS i LED_DOT. Deklaracje tablic umieszczamy w led.c, deklaracje makr w led.h. W led.h powinny się też znaleźć nagłówki funkcji ledInit() i ledDispNumber(). Na początku pliku led.c trzeba też umieścić dyrektywy #include dołączające potrzebne pliki nagłówkowe:

```
#include <avr/interrupt.h>
#include <stdio.h>
#include <string.h>
#include "led.h"
```

Nasza biblioteka jest już gotowa do użytku. Możemy ją przetestować za pomocą poniższego pliku main.c:

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

int main(void)
{
 ledInit();
 DDRB |= _BV(PB6);
 ledDispNumber(-3, "%g", 1);
 while (1)
 {
  _delay_ms(200);
  PORTB |= _BV(PB6);
  _delay_ms(200);
  PORTB &= ~_BV(PB6);
 }
}
```

Widoczność tablicy sterującej wyświetlaczem

W powyższym przykładzie wykorzystaliśmy funkcję ledDispNumber() do wyświetlenia liczby. Co się jednak stało z bezpośrednim sterowaniem segmentami wyświetlaczy? Czy możemy odwołać się do tablicy dispDigits z poziomu pliku main.c? Spróbujmy np. wyświetlić minus na drugiej pozycji, umieszczając poniższą linijkę przed pętlą while:

```
dispDigits[1] = segments[LED_MINUS];
```

Niestety otrzymamy błąd:

Error 'dispDigits' undeclared (first use in this function)

Wyświetlony będzie też taki sam błąd w stosunku do tablicy segments. Nie mamy bowiem deklaracji tablic dispDigits i segments w pliku main.c, są one zadeklarowane w led.c. Umieśćmy więc w main.c, pod dyrektywami #include następujące deklaracje:

```
extern uint8_t dispDigits[4];
extern const uint8_t segments[];
```

Sprawdźmy – program kompiluje się poprawnie. Rozwiązaliśmy więc problem, ale czy na pewno nasz program ma właściwą strukturę? Dobre praktyki mówią, że sterownik/biblioteka nie powinien dawać dostępu do swoich wewnętrznych struktur. Nasz kod jest stosunkowo prosty i ta zasada może się wydawać pozbawiona sensu. Jednak przy bardziej rozbudowanych projektach pozwala zabezpieczyć się przed przypadkowym zmodyfikowaniem pamięci używanej przez bibliotekę w sposób, który uniemożliwi lub zakłóci jej prawidłowe działanie. Wprowadza też większy porządek w kodzie.

Ponadto zauważmy, że bezpośrednie pisanie do tablicy dispDigits nie jest zbyt wygodne, gdyż wymaga odwoływania się do tablicy segments. Użytkownik biblioteki obsługującej wyświetlacz zainteresowany jest tym, jaki symbol chce wyświetlić, zwykle cyfrę, oraz na której pozycji. Dopóki nie będzie chciał definiować nowych symboli (układów segmentów), nie powinien mieć konieczności odwoływania się do tablicy segments.

Z wyżej wymienionych powodów tablice dispDigits oraz segments powinny być „schowane” wewnątrz biblioteki i nie być dostępne na zewnątrz. Dobrze jest nawet ich deklaracje poprzedzić słowem kluczowym static, dzięki temu przestaną być widoczne w main.c, nawet jeśli umieścimy tam ich deklaracje ze słowem kluczowym extern. Natomiast aby była możliwość wyświetlania pojedynczych symboli, zdefiniujemy funkcje jak na **listingu 6**:

```
void ledDispSymbol(uint8_t symbol, uint8_t position) {
 if (symbol < sizeof(segments) && position < sizeof(dispDigits))
  dispDigits[position] = segments[symbol];
}

void ledDispRaw(uint8_t seg, uint8_t position) {
 if (position < sizeof(dispDigits)) dispDigits[position] = seg;
}
```

Listing 6

Pierwsza z nich służy do wyświetlania symboli zdefiniowanych w tablicy symbols. Za pomocą instrukcji if sprawdzane jest, czy przekazane parametry nie mają przypadkiem zbyt dużych wartości. W przeciwnym wypadku przekroczony zostałby zakres tablic i program odwołałby się do przypadkowego miejsca w pamięci. Druga funkcja pozwala włączyć dowolne symbole na wybranej pozycji wyświetlacza. Popatrzmy na przykładowy program:

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

int main(void) {
 ledInit();
 ledDispNumber(-5, "%g", 1);
 ledDispSymbol(LED_MINUS, 3);
 while (1)  {
  _delay_ms(200);
  ledDispRaw(0b00000001, 0);
  _delay_ms(200);
  ledDispRaw(0b00000110, 0);
  _delay_ms(200);
  ledDispRaw(0b00001000, 0);
  _delay_ms(200);
  ledDispRaw(0b00110000, 0);
 }
}
```

Tutaj zamiast migającej diody w głównej pętli wyświetlana jest prosta animacja na pierwszym wyświetlaczu od prawej, mająca postać biegającej dookoła kreski. Na wyświetlaczach 3 i 4 od prawej wyświetlane są minusy, a na wyświetlaczu drugim cyfra 5.

Brakuje nam jeszcze funkcji czyszczącej wyświetlacz. Możemy ją zrealizować następująco:

```
void ledClear() {
 memset(dispDigits, 0, sizeof(dispDigits));
}
```

Wykorzystana została tutaj funkcja memset() z biblioteki stdio.h. Tablicę wskazaną w pierwszym parametrze wypełnia ona wartością wskazaną w parametrze drugim. Czyli tablica dispDigits zostanie wypełniona zerami. Ostatni parametr to rozmiar obszaru do wypełnienia. Ponieważ wyzerować chcemy całą tablicę, podajemy jej rozmiar pobierany operatorem sizeof.

```
int readNumber(uint8_t position, uint8_t maxLength) {
 // walidacja parametrów
 if (position > 3) position = 3;
 if (maxLength > 4) maxLength = 4;
 if (position + maxLength > 4) maxLength = 4 - position;
 uint8_t dispNum = 3 - position;
 char input[5];
 uint8_t inputIndex = 0;
 uint8_t lastDigit = 0;
 while(1) {
  uint8_t key = 0;
  key = getKey();
  if (key == 15) {
   // obsługa Backspace
   if (inputIndex > 0) {
    if (lastDigit) {
     ledDispRaw(0, dispNum);
     lastDigit = 0;
    } else {
     inputIndex--;
     ledDispRaw(0, ++dispNum);
    }
   }
   continue;
  }
  if (key < 10) input[inputIndex] = '0' + key;
  if (key == 10) input[inputIndex] = '0';
  if (key == 16) {
   input[inputIndex + 1] = 0;
   break;
  }
  // wyświetl cyfrę
  ledDispSymbol(key < 10 ? key : 0, dispNum);
  // dla cyfr przed limitem długości
  if (inputIndex < maxLength - 1) {
   inputIndex++;
   if (dispNum > 0) {
    dispNum--;
   }
  }
  } else { // dla ostatniej cyfry
   lastDigit = 1;
  }
 }
 return atoi(input);
}
```

Listing 7

Wpisywanie liczb

Tak jak w przypadku LCD, tak i teraz chcielibyśmy mieć możliwość wpisywania liczb z klawiatury z jednoczesnym ich wyświetlaniem. Posłużymy się tutaj funkcją readNumber(), którą napisaliśmy dla LCD. Na **listingu 7** przykładowa modyfikacja dla wyświetlacza LED:

Funkcja jest dosyć długa, ale jej analiza nie powinna być trudna, szczególnie po zapoznaniu się z lekcjami 6 i 7. Do funkcji przekazywane są dwa parametry: position oraz maxLength. Pierwszy mówi, od którego wyświetlacza ma się rozpocząć wyświetlanie wpisywanej liczby. Jest to przydatne, gdybyśmy chcieli wpisać obok siebie dwie różne liczby, np. godzinę i minutę. Drugi parametr to maksymalna liczba cyfr. Przykładowo jeśli jako maxLength podamy 2, użytkownik będzie mógł wpisać liczbę maksymalnie dwucyfrową. W pierwszych trzech linijkach sprawdzamy, czy przekazane parametry nie są zbyt duże i jeśli tak, zostają odpowiednio zmniejszone.

W głównej pętli wczytywany jest klawisz i sprawdzany jest jego numer. Dla numerów klawiszy poniżej 10 zapamiętywany jest ich numer jako cyfra w tablicy znakowej input. Klawisz 10 powoduje zapamiętanie cyfry 0. Ostatni klawisz, noszący numer 16, działa jako Enter i powoduje wyjście z pętli. Klawisz przedostatni, z numerem 15, działa jak Backspace. Rozróżnia on dwa przypadki, zależnie od tego, czy użytkownik wpisał już wszystkie cyfry, jakie mógł wpisać, np. dwie cyfry liczby dwucyfrowej. W obu przypadkach zostanie wyczyszczona ostatnia wpisana cyfra i dla użytkownika wygląda to tak samo, jednak w kodzie jest to obsługiwane różnie z uwagi na to, że po wpisaniu ostatniej cyfry przestaje zwiększać się zmienna inputIndex wskazująca na aktualne miejsce w tablicy input oraz przestaje zmniejszać się zmienna aktualnego wyświetlacza. Wykorzystywana jest tutaj zmienna pomocnicza lastDigit, ustawiana na 1, gdy użytkownik wpisał ostatnią cyfrę.

Wpisane cyfry są wyświetlane funkcją ledDispSymbol(). Pobierany jest tutaj bezpośrednio numer klawisza z wyjątkiem klawisza 10, dla którego wyświetlane jest 0. Został tutaj użyty operator warunkowy. Sprawdza on, czy wyrażenie przed znakiem zapytania jest prawdą. Jeśli tak, zwraca to, co jest przed dwukropkiem (wartość zmiennej key), w przeciwnym razie zwraca to, co jest po dwukropku (liczbę zero).

Jeśli użytkownik wpisał cyfrę, a nie była to ostatnia cyfra, zwiększany jest licznik cyfr. Jeśli wpisał ostatnią, ustawiana jest zmienna lastDigit dla obsługi Backspace. Wartość końcowa obliczana jest funkcją atoi(), która tworzy liczbę na podstawie wpisanych cyfr (znaków). Przykładowe wywołanie funkcji readNumber():

```
int n = readNumber(2, 2);
```

Wczytana zostanie liczba dwucyfrowa, przy czym wpisywane cyfry pojawią się na 3. i 4. pozycji od lewej.

W materiałach dodatkowych znajduje się kompletny projekt zawierający obsługę wyświetlacza LED oraz klawiatury z funkcją wyświetlania cyfr podczas wpisywania liczb.

Zadania

Jak zwykle zachęcam do pisania jak największej liczby własnych programów, choćby najprostszych. Aby przećwiczyć wiadomości z tego odcinka, proponuję napisać:

1. Zegar wyświetlający godziny i minuty, z możliwością ustawienia czasu
2. Zegar z migającą kropką na drugim wyświetlaczu od lewej, oddzielającą godziny od minut
3. Kostkę do gry – gdy wciśnięty jest przycisk, program bardzo szybko odliczać liczby 1–6, po puszczeniu przycisku odliczanie jest przerywane i wyświetlana jest ostatnia wylosowana w ten sposób liczba

Grzegorz Niemirowski
grzegorz@grzegorz.net

Extern dla dociekliwych

W jednym z przykładów użyliśmy słowa kluczowego extern. Jaka jest dokładnie jego rola? Otóż tworzenie wynikowego kodu wykonywalnego, który wpisujemy do pamięci Flash mikrokontrolera, składa się z trzech głównych etapów: przetwarzania kodu źródłowego przez preprocesor, kompilacji oraz linkowania. Często potocznie kompilacją nazywa się cały proces, ale lepiej mówić o budowaniu. Stąd w Atmel Studio mamy menu oraz polecenia Build. Kompilacja to tłumaczenie kodu źródłowego na kod wykonywalny, przy czym odbywa się ona plik po pliku.

Kompilator, przetwarzając dany plik, musi znać podstawowe dane o wszystkich występujących w nim symbolach, bo „nie wie” o innych plikach, chyba że są to pliki .h, które dołączyliśmy dyrektywą #include. Stosując słowo kluczowe extern, przekazujemy właśnie tylko te niektóre informacje, m.in. o typie zmiennej. Natomiast jej zawartość początkowa i ewentualnie rozmiar będą określone w innym pliku. Przy kompilacji tamtego pliku kompilator zadba o dodanie kodu rezerwującego pamięć dla tej zmiennej oraz inicjującego jej wartość. Nie możemy w jednym pliku napisać int a = 5; a w drugim extern int a = 5; ponieważ kod inicjalizujący zmienną może być tylko jeden.

Z punktu widzenia architektury kodu extern oznacza „chcę użyć zmiennej globalnej zdefiniowanej w innym pliku”. Natomiast z punktu widzenia zarządzania pamięcią oznacza „nie chcę używać nowego obszaru pamięci, ale odwołać się do obszaru zajmowanego przez zmienną zdefiniowaną w innym pliku”. Tak więc extern nie tworzy nowej zmiennej i przy kompilacji danego pliku nie są tworzone odwołania do odpowiedniego miejsca w pamięci. Te odwołania zostaną uzupełnione na etapie linkowania, gdy linker będzie miał już skompilowane wszystkie pliki i będzie je łączył w jedną całość.

Należy pamiętać, że extern jest niejako domyślne. Jeśli w dwóch plikach będą zadeklarowane zmienne globalne o tych samych nazwach, to będzie to jedna, wspólna zmienna. Aby zmienne te były niezależne, trzeba nadać im różne nazwy lub użyć słowa kluczowego static. Dlatego jeśli deklarujemy zmienną globalną, która ma być używana tylko w jednym pliku, zadeklarujmy ją jako static. Uchroni nas to przed przypadkowym odwołaniem do niej z innego pliku. Natomiast jeśli ma być dostępna z innych plików, to w tych plikach zadeklarujmy ją jako extern, pozostawiając ją bez extern i bez static w jej głównym pliku.

Przedstawione informacje tyczą się zmiennych globalnych, niezwiązanych z żadną funkcją, deklarowanych zwykle na początku pliku .c. Zaś słowo kluczowe static dla zmiennej wewnątrz funkcji oznacza, że jej wartość jest zapamiętywana pomiędzy wywołaniami funkcji. Wynika to z tego, że umieszczona jest w tym samym obszarze co zmienne globalne, które „żyją” przez cały czas działania programu. Widoczna jest jednak tylko w swojej funkcji, nie jest globalna.