Kurs AVR – lekcja 2

W pierwszym odcinku kursu poznaliśmy podstawy programowania mikrokontrolerów AVR. Dowiedzieliśmy się też, jak sterować stanem pinów mikrokontrolera. W tym odcinku przeanalizujemy nieco bardziej skomplikowane przykłady obsługi pinów, w tym odczyt ich stanu. Najpierw jednak zastanowimy się, jak można odpowiedzieć na pytania zamieszczone na końcu poprzedniego odcinka.

Rozwiązania poprzednich zadań

- miganie kilkoma LED-ami naraz

Żeby można było migać także innymi LED-ami, trzeba postąpić analogicznie do jednego: podłączyć je do wyprowadzeń portu, skonfigurować te wyprowadzenia jako wyjścia, a następnie przełączać stan tych wyjść. Załóżmy, że podłączamy dwie dodatkowe diody do portu A, np. do wyprowadzeń 3 i 5. Używamy więc łącznie wyprowadzeń 0, 3 i 5. Ustawiamy odpowiednie bity rejestru DDRA:

```
DDRA = _BV(DDA0) | _BV(DDA3) | _BV(DDA5);
```

Podobnie będzie wyglądało włączanie LED-ów przez ustawienie trzech bitów rejestru PORTA. Cały program będzie wyglądał następująco:

```
#include <avr/io.h>
#include <util/delay.h>
int main(void)
{
 DDRA = _BV(DDA0) | _BV(DDA3) | _BV(DDA5);
 while(1)
 {
    PORTA = _BV(PA0) | _BV(PA3) | _BV(PA5);
    _delay_ms(250);
    PORTA = 0;
    _delay_ms(250);
 }
}
```

- miganie dwoma LED-ami na przemian

Załóżmy, że LED-y podłączone będą do wyprowadzeń 0 i 3. Możemy już łatwo zgadnąć, jak będzie wyglądała inicjalizacja rejestru DDR: ustawimy bity DDA0 i DDA3. Przełączanie LED-ów będzie proste. Popatrzmy na przykład z **listingu 1.**

Ponieważ przypisujemy całą wartość rejestru, ustawienie jednego bitu będzie zerowało wszystkie pozostałe.

- zmiana szybkości migania

Szybkość migania zależy od okresu opóźnienia. Rozwiązanie jest więc bardzo proste, zmieniamy liczbę wpisywaną jako parametr wywołania funkcji _delay_ms(). Czym będzie ona mniejsza, tym miganie będzie szybsze.

- zmiana stosunku czasu zgaszenia do czasu świecenia

Jest to przypadek jak powyższy. Różnica polega na tym, że wpisujemy różne wartości opóźnienia, np. 500 i 100.

Do eksperymentów z LED-ami warto wykorzystać znajdującą się na płytce testowej diodę trójkolorową. Dostępna jest ona przez złącze o nazwie RGB, wystarczy wybrane piny mikrokontrolera połączyć z pinami R, G i B tego złącza. Piny te sterują odpowiednio kolorem czerwonym, zielonym i niebieskim. Włączając poszczególne kolory, pary lub wszystkie naraz, można otrzymać ciekawe efekty świetlne.

Sterowanie linijką LED

Miganie diodami może się wydać czymś, czemu nie warto poświęcać zbyt dużo czasu. Przecież to tylko ustawianie bitów w rejestrze portu. Jednak zastanawiając się nad niektórymi efektami, dojdziemy do ciekawych zagadnień programistycznych.

Zastanówmy się np. nad biegającym światełkiem: w linijce diod zaświecamy kolejne diody (w danej chwili świeci się tylko jedna) od lewej do prawej i z powrotem. Możemy to zrobić metodą kopiuj-wklej, wielokrotnie kopiując linijkę ustawiającą wartość rejestru PORTA, za każdym razem zmieniając bit na kolejny, a także linijkę wprowadzającą opóźnienie. Taki program będzie oczywiście działać, ale nie będzie to zbyt eleganckie ani wygodne. Jeśli np. chcielibyśmy, aby efekt odbywał się szybciej, zmianę opóźnienia musielibyśmy wykonać w kilkunastu miejscach. Rozwiązania mogą być różne. Przyjrzymy się dwóm, prezentującym różne podejścia.

```
#include <avr/io.h>
#include <util/delay.h>
int main(void)
{
 DDRA = _BV(DDA0) | _BV(DDA3);
 while(1)
 {
     PORTA = _BV(PA0);
     _delay_ms(250);
     PORTA = _BV(PA3);
     _delay_ms(250);
 }
}
```

Listing 1

Podejście tablicowe. W pierwszym rozwiązaniu wykorzystamy tablicę, w której zdefiniowane będą kolejne stany diod świecących. W pętli będziemy je po kolei odczytywać i wpisywać do rejestru portu. Przykładowy kod wygląda następująco:

```
#include <avr/io.h>
#include <util/delay.h>
const uint8_t ledStates[] =
{
    0b10000000,
    0b01000000,
    0b00100000,
    0b00010000,
    0b00001000,
    0b00000100,
    0b00000010,
    0b00000001,
    0b00000010,
    0b00000100,
    0b00001000,
    0b00010000,
    0b00100000,
    0b01000000
};

int main(void)
{
 DDRA = 0b11111111;
 uint8_t i = 0;
 while(1)
 {
     PORTA = ledStates[i];
     _delay_ms(100);
     i++;
    if (i == sizeof(ledStates)) i = 0;
 }
}
```

Głównym elementem naszego programu jest tablica, w której zapisywane są kolejne stany diod. Ktoś mógłby tu zaprotestować, że jest to właśnie programowanie typu kopiuj-wklej. W pewnym sensie tak, ale tutaj stany diod są zebrane w jednym miejscu, w postaci tablicy, do której łatwo odwoływać się w kodzie. Ponadto pokazuje ona w czytelny sposób, które diody będą włączone, a które nie. Należy tu wspomnieć, że binarny zapis liczb w postaci 0bXXXXXXXX nie występuje w standardzie języka C, jest to rozszerzenie kompilatora AVR-GCC. Elementy tablicy są typu uint8_t, czyli ośmiobitowe, całkowite, bez znaku. Jeden element zajmuje jeden bajt. Ogólnie tablica zajmuje 14 bajtów.

Ponieważ wykorzystywany jest cały port A, ustawiamy wszystkie bity rejestru DDRA na 1. Moglibyśmy tutaj wpisać wszystkie bity rejestru, korzystając z makra _BV(), ale z racji, że ustawiamy wszystkie bity, binarny zapis 0b11111111 jest również czytelny, a przy tym znacznie krótszy.

Drugim elementem początkowej inicjalizacji w naszym programie jest zdefiniowanie zmiennej i oraz jej wyzerowanie. Zmiennej tej używamy w pętli do odwoły-

wania się do kolejnych elementów tablicy. Wskazuje ona numer elementu tablicy. Na początku więc wybrany zostanie zerowy element tablicy, czyli wartość mająca binarną postać 10 000 000 (dziesiętnie 128). Zaświecona zostanie dioda podłączona do pinu 7 portu A. Potem wykonywane jest opóźnienie. Następnie zmienna i jest inkrementowana (zwiększana o 1). Zanim pętla wykona się ponownie z wybranym kolejnym elementem tablicy, sprawdzana jest wartość zmiennej i. Jeśli bowiem dojdziemy do ostatniego elementu tablicy, musimy zacząć od pierwszego. Inaczej zmienna będzie wskazywała na bajty w pamięci, które już do tablicy nie należą. Do sprawdzenia, czy wartość zmiennej nie jest już za duża, wykorzystywany jest operator sizeof, który zwraca rozmiar tablicy. Ponieważ ma ona 14 bajtów, a jej elementy mają indeksy od 0 do 13, to indeks wynoszący 14, czyli tyle, ile rozmiar, będzie już za duży. Jeśli więc zmienna i ma wartość 14, warunek zostanie spełniony i instrukcja if spowoduje wyzerowanie zmiennej i.

A co się stanie, jeśli wyjdziemy poza zakres tabeli? Możemy to sprawdzić, usuwając lub zakomentowując linijkę z instrukcją if. W tym programie akurat nie wydarzy się nic strasznego. Po prostu LED-y będą wyświetlać w postaci binarnej zawartość pamięci, która znajduje się za tabelą. W końcu zmienna i, która jest ośmiobitowa, przepełni się i znów będzie wyświetlana zawartość tablicy od jej początku. Program będzie więc oprócz zdefiniowanych przez nas danych wyświetlał także de facto śmieci. W innych przypadkach wyjście poza zakres tablicy może spowodować niestabilne działanie programu lub utratę danych. Należy więc zawsze dbać, aby kontrolować, które elementy tablic są używane.

Podejście algorytmiczne. W poprzednim przykładzie mieliśmy przygotowane dane, a program był niejako odtwarzaczem, który je kolejno odczytywał i używał bez głębszej analizy. Możliwe jest odwrotne podejście, w którym sterowanie odbywa się na podstawie określonego algorytmu. Tutaj jest to algorytm przesuwania punktu świetlnego. Popatrzmy na poniższy przykład. Zależnie od kierunku przesuwania punktu świetlnego program wpisuje 1 do najstarszego lub najmłodszego bitu rejestru PORTA, a następnie przesuwa bity portu w odpowiednią stronę. Gdy po 8 przesunięciach jedynka „wypadnie" z rejestru i będą w nim same zera, kierunek jest zmieniany. Dla większej czytelności kodu zdefiniowane zostały makra LEFT i RIGHT:

```
#include <avr/io.h>
#include <util/delay.h>
#define LEFT 0
#define RIGHT 1

int main(void)
{
    DDRA = 0xff;
    uint8_t direction = LEFT;
    while(1)
    {
        if (!PORTA) {
           direction = !direction;
           if (direction == LEFT)
               PORTA = 1;
            else
               PORTA = 0x80;
        }
        _delay_ms(100);
        if (direction == LEFT)
            PORTA <<= 1;
        else
            PORTA >>= 1;
    }
}
```

Jak widać, nie ma tu żadnej tablicy. Deklarujemy tylko zmienną, w której zapamiętywany jest kierunek przesuwania punktu świetlnego. Ważną funkcję pełni też rejestr PORTA, na którym wykonywana jest operacja przesunięcia bitowego.

W powyższych przykładach konfigurowaliśmy piny portu mikrokontrolera jako wyjścia i sterowaliśmy diodami świecącymi. Zamiast diod możemy podłączyć oczywiście wiele innych układów. Maksymalna wydajność pojedynczego pinu to 40 mA. Jeśli konieczny będzie większy prąd, zastosować trzeba tranzystor lub inny odpowiedni element sterujący.

A co z pracą jako wejście?

Odczyt stanu pinów

Jak zostało wspomniane, jest to domyślny stan portów: rejestry DDRx mają po resecie wyzerowane bity. Do odczytu stanu pinów służą rejestry PINx (PINA, PINB, PINC, PIND). Są to oddzielne rejestry od rejestrów PORTx. Odczyt tych drugich da nam wartości, które sami do nich wpisaliśmy, niezwiązane z faktycznym napięciem na pinie mikrokontrolera. Co ciekawe, rejestry PORTx także mają znaczenie przy pracy pinów jako wejścia. Jeśli bit PORTxn (np. PORTA0) jest ustawiony na 0, wówczas odpowiadający mu pin (w przykładzie pin 0 portu A) jest w stanie wysokiej rezystancji. Jeśli natomiast ustawimy go na 1, wówczas pin zostanie podciągnięty do plusa zasilania przez wewnętrzny rezystor.

Kontrolowanie diody przyciskiem. Korzystając z tych informacji, napiszmy prosty program, który będzie włączał LED, gdy wciśnięty jest przycisk. Wykorzystamy w tym celu znajdujący się na płytce testowej naszego kursu LED1 oraz przycisk S1. Dioda wraz z towarzyszącym jej rezystorem zostanie podłączona do pinu 0 portu A, a przycisk do pinu 1 tego samego portu. Ponieważ port mikrokontrolera obsługuje podciąganie do plusa, przycisk musi być włączony od strony masy. Na płytce testowej przyciski połączone są w układzie klawiatury matrycowej, w rzędy i kolumny. Musimy więc podłączyć pierwszy rząd/wiersz (pin W1 złącza KEYB) do pinu mikrokontrolera (PA1) a pierwszą kolumnę (pin K1 złącza KEYB) do masy lub oczywiście odwrotnie: kolumnę do masy, a rząd do pinu, gdyż przyciski są niebiegunowe.

Nasz kod będzie wyglądał następująco:

```
#include <avr/io.h>
int main(void)
{
    DDRA = _BV(DDA0);
    PORTA = _BV(PA1);
    while(1)
    {
        if (PINA & _BV(PINA1)) {
            PORTA &= ~_BV(PA0);
        } else {
            PORTA |= _BV(PA0);
        }
    }
}
```

Tak jak w pierwszym przykładzie ustawiamy pin 0 portu A jako wyjście. Pin 1 pozostaje wejściem, ale włączone zostaje dla niego podciągnięcie do plusa zasilania. W pętli odczytywany jest rejestr PINA przechowujący stan wejść portu A. Za pomocą operatora sumy logicznej sprawdzamy, czy jest w nim ustawiony bit odpowiadający pinowi 1. Jeśli tak, oznacza to, że przycisk zwierający pin do masy nie został wciśnięty i dioda powinna być zgaszona. Zerowany jest więc bit 0 w rejestrze PORTA. Gdy przycisk zostanie wciśnięty, bit PINA1 będzie wyzerowany i wtedy włączamy diodę, ustawiając bit PA0 w rejestrze PORTA. Linijki ustawiające wartość rejestru PORTA możemy zamienić miejscami i wtedy przycisk będzie gasił diodę.

Zatrzymajmy się jeszcze chwilę przy zapisie do rejestru PORTA. Wykonywane operacje logiczne mogą się wydać bardzo niejasne. Problem polega na tym, że procesory AVR nie pozwalają na niezależne sterowanie pinami portów. Operuje się na rejestrze jako całości, ustawiając stan wszystkich jego bitów. Jeśli więc chce się zmienić jeden pin, trzeba najpierw odczytać wszystkie pozostałe, aby móc je zapisać bez ich zmieniania. We wcześniejszych przykładach nam to nie przeszkadzało, ale tutaj chcemy bit PA1 w rejestrze PORTA ustawić na początku programu i już go potem nie ruszać. Co prawda można by go dodawać do późniejszych linijek, które operują na PORTA, ale będzie to niewygodne i nieeleganckie.

Jak więc manipulować pojedynczymi bitami? Przy ustawianiu bitu na 1 jest to bardzo proste. Wystarczy wykonać sumę logiczną obecnej wartości rejestru oraz wartości, w której ustawiony jest tylko potrzebny nam pin. Taką wartość zwraca nam makro _BV(). Otrzymujemy w ten sposób prosty zapis w rodzaju

PORTA |= _BV(PA0);

Niestety przy zerowaniu bitu jest trudniej. Suma logiczna nam się nie przyda, bo „nie potrafi" zerować bitów w istniejącej wartości. Musimy skorzystać z iloczynu logicznego. Używając wartości, która ma dany bit wyzerowany, będziemy mogli za pomocą iloczynu logicznego wyzerować dany bit rejestru. Żeby jednak nie naruszyć pozostałych bitów, nasza wartość zerująca musi mieć pozostałe bity ustawione na 1. Stąd wartość zwracana przez makro _BV() musi mieć najpierw odwrócone wszystkie bity. Ostatecznie otrzymujemy zapis typu PORTA &= ~_BV(PA0);

Przełączanie stanu LED-a na przeciwny. Spróbujmy dodać więcej inteligencji do naszego programu. Niech naciśnięcie przycisku przełącza stan diody. Naciśnięcie, czyli sytuacja gdy przycisk zmienia stan. Musimy więc sprawdzić stan przycisku i porównać go z poprzednim. Stan przycisku przechowywany jest w zmiennej *switchState*, poprzedni stan w zmiennej *lastSwitchState*. Wykonywana jest negacja odczytu stanu pinu, ponieważ wygodniej jest z programistycznego punktu widzenia, jeśli wciśnięcie przycisku będzie dawało wartość 1 w zmiennej. Po odczycie sprawdzamy, czy przycisk jest wciśnięty i czy poprzednio nie był wciśnięty. Jeśli tak, wówczas przestawiana na przeciwną jest wartość zmiennej stanu LED-a **(listing 2)**.

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

int main(void)
{
    DDRA |= _BV(DDA0);
    PORTA |= _BV(PA1);
    uint8_t ledState = 0;
    uint8_t lastSwitchState = 0;
    while(1)
    {
       uint8_t switchState = !(PINA & _BV(PINA1));
       if (!lastSwitchState && switchState) {
           ledState = !ledState;
       }
       lastSwitchState = switchState;
       if (ledState) {
           PORTA |= _BV(PA0);
       } else {
           PORTA &= ~_BV(PA0);
       }
    }
}
```

Listing 2

Uruchommy nasz program i sprawdźmy, czy działa poprawnie. Możliwe, że Czytelnik dostrzeże czasami nieprawidłowe przełączanie stanu diody. Dochodzimy tutaj do problemu drgań styków przycisków. Podczas naciskania styki mogą się odbijać od siebie i mikrokontroler może to zinterpretować jako więcej niż jedno naciśnięcie przycisku. Zależy to od budowy przycisku a także częstotliwości odczytu stanu pinu. Dlatego najprościej zmniejszyć częstotliwość odczytu przycisku, np. dodając opóźnienie rzędu kilkudziesięciu milisekund. Jeśli wykorzystamy funkcję _delay_ms(), należy pamiętać o włączeniu pliku nagłówkowego <util/delay.h> jak w naszym pierwszym programie.

```
if (!lastSwitchState && switchState)
{
    ledState = !ledState;
    _delay_ms(30);
}
```

Jako że ten przykład jest pierwszym naszym programem dla AVR, w którym używamy zmiennych, konieczna jest dygresja. Jak widać, użyte zostały zmienne typu uint8_t, który nie był wspomniany w części teoretycznej kursu. Otóż w przypadku mikrokontrolerów dużo ważniejsze jest, jaki rozmiar mają poszczególne zmienne, niż jest to w przypadku programowania na PC. A przykładowo typ int ma różny rozmiar na różnych architek-

Ściągawka z C

#define

Dyrektywa dla preprocesora, która mówi, że pod dany symbol ma być podstawione inne wyrażenie. Podstawienie odbywa się zaraz przed kompilacją. #define LEFT 0 oznacza, że wszędzie tam, gdzie występuje LEFT, ma być podstawione 0.

uint8_t zmienna;

Deklaracja zmiennej. Oznacza, że rezerwujemy miejsce w pamięci operacyjnej i że będziemy odwoływać się do niego przez nazwę „zmienna". W tym przykładzie zmienna ma typ uint8_t, czyli zajmować będzie 8 bitów (1 bajt), będzie przechowywać liczbę całkowitą i nie będzie mieć znaku (będzie zawsze dodatnia). Przy deklaracji od razu możemy wpisać do zmiennej określoną wartość, np. uint8_t i = 0; spowoduje deklarację zmiennej i oraz wpisanie do niej liczby 0.

const typ zmienna;

Deklaracja stałej. Jest to zmienna, której wartość nie zmienia się w trakcie działania programu.

uint8_t zmienna_tablicowa[] = {…};

Deklaracja tablicy, czyli zmiennej, która przechowuje wiele elementów danego typu. Jeśli tablica inicjalizowana jest od razu, elementy wpisujemy w nawiasach klamrowych. Jeśli nie, wówczas w nawiasach kwadratowych trzeba podać ile ma mieć ona elementów, a deklarację zakończyć zaraz średnikiem, bez znaku równości i nawiasów kwadratowych.

zmienna_tablicowa[n]

Odwołanie się do n-tego elementu zmiennej tablicowej. Elementy numerowane są od zera. Zwykle n to zmienna, ale może być również inne wyrażenie zwracające wartość liczbową.

uint8_t readKeyboard();

Deklaracja funkcji. Informuje ona kompilator, jaki typ zwraca funkcja (tutaj uint8_t) oraz jakie przyjmuje parametry (tutaj akurat brak). Kompilator analizuje kod po kolei, z góry do dołu. Gdy natrafi na miejsce wywołania funkcji, musi znać jej deklarację. Co prawda moglibyśmy po prostu definicję (treść) funkcji umieścić przed funkcją, która ją wywołuje, ale takie rozwiązanie jest na dłuższą metę niewygodne, a nieraz też niemożliwe.

if () {} else {}

Sprawdzenie warunku. Jeśli wyrażenie w nawiasach okrągłych jest prawdziwe (różne od zera), zostaną wykonane instrukcje umieszczone w pierwszych nawiasach klamrowych. W przeciwnym wypadku będą wykonane instrukcje z drugich nawiasów, umieszczonych po słowie kluczowym else. Słowa tego i drugich nawiasów może nie być, czyli nie będzie przewidziany żaden kod na okoliczność niespełnienia warunku. W przypadku gdy do wykonania jest pojedyncza instrukcja, nawiasy klamrowe mogą być opuszczone.

!

Logiczny operator zaprzeczenia. Dla wyrażeń równych 0 zwraca 1, a dla różnych od 0 zwraca 0.

~

Operator negacji bitowej. Każdy bit w wyrażeniu (np. w zmiennej) zamienia na przeciwny. Np. dla wartości binarnej 00101011 zwróci 11010100.

&&

Logiczny operator I (AND). Zwraca 1, jeśli wyrażenia z jego lewej i prawej strony są prawdą (różne od zera).

||

Logiczny operator LUB (OR). Zwraca 0, jeśli przynajmniej jedno z wyrażeń z jego lewej i prawej strony są prawdą (różne od zera).

==

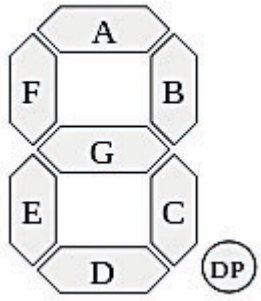
Operator porównania. Nie należy go mylić z operatorem przypisania (=). Operator porównania zwraca 1, jeśli wyrażenia po jego lewej i prawej stronie zwracają taką samą wartość liczbową. W przeciwnym wypadku zwraca 0.

```
#include <avr/io.h>
#include <util/delay.h>
uint8_t readKeyboard();
const uint8_t segments[10] =
{  0b00111111,
   0b00000110,
   0b01011011,
   0b01001111,
   0b01100110,
   0b01101101,
   0b01111100,
   0b00000111,
   0b01111111,
   0b01101111 };

int main(void)
{
 DDRA = _BV(DDA0) | _BV(DDA1) | _BV(DDA2) | _BV(DDA3);
 DDRD = 0xFF;
 uint8_t value = 0;
 while(1)
 {
    PORTD = ~segments[value];
    uint8_t key = readKeyboard();
    if (key) {
       if (key < 10) {
          value = key;
       }
       if (key == 10) value = 0;
       if (key == 12)
          if (value < 9) value++;
       if (key == 16)
          if (value > 0) value--;
       _delay_ms(200);
    }
  }
}
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 3

Rys. 1



turach. Dlatego używane są typy takie jak int8_t, int16_t czy int32_t oraz ich wersje z literką u dla typów bez znaku (unsigned). Dzięki temu programista od razu widzi, ile bitów ma zmienna i może bardziej świadomie zarządzać pamięcią, której mikrokontrolery często mają niewiele. Wspomniane typy są zdefiniowane na bazie standardowych typów języka C. Można się o tym przekonać, klikając w Atmel Studio na uint8_t w jednej z linijek naszego programu i wciskając Alt+G, który powoduje przejście do definicji klikniętego wyrażenia (typu, zmiennej, funkcji, makra itd.). Zobaczymy, że uint8_t to w rzeczywistości unsigned char.

Obsługa klawiatury matrycowej

W eksperymentach warto wykorzystać klawiaturę oraz wyświetlacze LED. W przykładzie użyty jest jeden wyświetlacz (jedna cyfra), ponieważ większą ich liczbę najwygodniej obsługiwać multipleksowo z wykorzystaniem timera, co będzie omówione w dalszej części kursu. Natomiast teraz możemy omówić multipleksowy odczyt klawiatury matrycowej. Polega on na tym, że sprawdzany jest stan pinów, do których podłączone są kolejne kolumny przycisków. Piny są podciągnięte do plusa. Naciśnięcie przycisku w kolumnie zwiera pin do masy i powoduje pojawienie się zera na odpowiednim bicie rejestru wejściowego portu. Pozostaje jeszcze identyfikacja wiersza. W tym celu logiczne zero podawane jest na kolejne wiersze i wiedząc, na którym wierszu było zero, można jednoznacznie określić wciśnięty przycisk.

W naszym przykładzie wierszami klawiatury sterują cztery młodsze piny portu A, a z czterech starszych odczytywana jest kolumna. Jeśli np. zero będzie wystawione na pinie 2 oraz zostanie odczytane zero na pinie 3, będzie to oznaczało, że wciśnięty został przycisk w 3. rzędzie i 4. kolumnie czyli S12 na naszej płytce testowej. Po wystawieniu zera na danym wierszu, w programie umieszczone jest mikrosekundowe opóźnienie, aby zdążył się zaktualizować stan rejestru PINA. Funkcja readKeyboard() zwraca numer wciśniętego przycisku (1–12). Jeśli nic nie zostało wciśnięte, zwraca zero. Numer wciśniętego przycisku (1–9) prezentowany jest na wyświetlaczu 7-segmentowym. Dla przycisku S10 wyświetlana jest wartość 0. Program obsługuje też zwiększanie i zmniejszanie wyświetlanej wartości za pomocą przycisków S12 i S16. Wyświetlacz podłączony jest do portu D, kolejne segmenty a–g do kolejnych pinów 0–7. Rozmieszczenie segmentów wyświetlacza przedstawia **rysunek 1**. Końcówka wspólna wyświetlacza sterowana jest tranzystorem PNP, konieczne jest więc dołączenie do masy pinu włączającego używany przez nas wyświetlacz (pin 1–4 złącza LED_DISP).

Kombinacje segmentów dla poszczególnych cyfr zostały umieszczone w tablicy *segments*, co pozwala na łatwy do nich dostęp. Przy zapisie wartości do portu D wykonywana jest negacja z uwagi na sterowanie wyświetlacza od strony masy na naszej płytce **(listing 3)**.

Sterowanie brzęczykim piezo

Nasza płytka testowa wyposażona jest w przetwornik piezoelektryczny (bez generatora). Dzięki niemu możemy generować proste dźwięki, podając na niego sygnał prostokątny. Generowanie przebiegu prostokątnego omówiliśmy już na przykładzie naszego pierwszego programu migającego diodą. Teraz wystarczy zamiast diody podłączyć brzęczyk do pinu mikrokontrolera. Oczywiście generowana częstotliwość jest niska, więc zamiast dźwięku usłyszymy stukanie. Trzeba więc ją podwyższyć, zmniejszając okres. Jak wiemy, za okres odpowiedzialne są opóźnienia realizowane za pomocą funkcji _delay_ms(). Jeśli jako parametr jej wywołania podamy liczbę 1, na pinie mikrokontrolera przez 1 milisekundę będzie utrzymywał się stan niski i przez 1 milisekundę stan wysoki. Łącznie okres wyniesie więc 2 ms, co da częstotliwość 500Hz. Jeśli w obu wywołaniach funkcji _delay_ms() podamy jako parametr liczbę 2, otrzymamy 250Hz. Możemy też stosować wartości ułamkowe. Np. dla częstotliwości 1 kHz konieczne będą wywołania

```
_delay_ms(0.5);
```

Zadanie

W tym odcinku zachęcam do zastanowienia się nad generowaniem dźwięku w momencie naciśnięcia przycisku klawiatury matrycowej. Jak wygenerować dźwięk, gdy przycisk jest wciśnięty? Jak generować dźwięki o różnej wysokości w zależności od wciśniętego przycisku? Jak poradzić sobie z faktem, że funkcja _delay_ms() nie przyjmuje jako parametru zmiennych czy wartości zwracanych przez funkcje, a jedynie wyrażenia, których wartość jest znana w momencie kompilacji?

Grzegorz Niemirowski
grzegorz@grzegorz.net

Uwaga do lekcji 1 kursu (EdW 5/2016):
Lekcję można zrealizować z wartościami Fusebitów z rysunku 8, jednak w tekście podana jest prawidłowa wartość bajtu HIGH (0x99).