Kurs AVR – lekcja 11

Rozwiązania zadania z ostatniego odcinka

W poprzedniej lekcji zadaniem było napisanie prostej gry zręcznościowej, w której gracz ma omijać przeszkody wyświetlające się na LCD. Ponieważ wyświetlacz jest tutaj głównym elementem, wyświetlającym stan gry, warto pomyśleć chwilę nad jego obsługą. Nie chodzi tutaj bynajmniej o napisanie nowej biblioteki, ale spojrzenie bardziej wysokopoziomowe. W naszej grze bardzo istotne jest to, co zostało wyświetlone na ekranie. Jeśli chcemy przesunąć przeszkody w lewo, musimy znać ich bieżące pozycje. Jeśli chcemy wykryć kolizję, również musimy znać pozycje przeszkód, a także pozycję gracza. Można w tym celu odczytywać dane z wyświetlacza, ale będzie to niewygodne i nieeleganckie. Najlepiej przechowywać cały stan gry w pamięci mikrokontrolera, a do LCD tylko ten stan wysyłać. Potrzebujemy więc jakoś odwzorować LCD w pamięci.

Nasz wyświetlacz jest alfanumeryczny, wyświetla dwie linie po 16 znaków. Od razu nasuwa się rozwiązanie: dwie zmienne tablicowe typu char, każda mająca po 16 elementów. W zmiennych tych będziemy mogli przechowywać to, co ma być wyświetlone na LCD. Takie podejście jest jak najbardziej poprawne, ale zwróćmy uwagę na fakt, że zmienne te są bardzo ze sobą związane, w końcu dotyczą jednego wyświetlacza. Dobrze więc byłoby umieścić je wewnątrz struktury i w programie operować na tej strukturze. W naszym kursie dotychczas nie korzystaliśmy ze struktur i choć były one omówione w kursie języka C, to mogą być one nowością dla wielu Czytelników. Przypomnienie, czym są struktury w C i jak z nich korzystać, znajduje się w ramce.

W przykładowym rozwiązaniu zadania z grą zastosowano poniższą strukturę.

```
struct disp_data {
 uint8_t upperLine[16];
 uint8_t lowerLine[16];
};
```

Ma ona dwa pola: upperLine i lowerLine, będące zmiennymi tablicowymi, przechowującymi po 16 bajtów. Spójrzmy na funkcję main(), przedstawioną na **listingu 1**, aby prześledzić działanie gry i wykorzystanie struktury.

W naszej grze korzystamy z wyświetlacza i klawiatury, program rozpoczynamy więc od ich inicjalizacji. Potrzebujemy też timera, żeby generować opóźnienia wyznaczające prędkość gry. Nie możemy generować opóźnienia funkcją _delay_ms(), ponieważ w trakcie opóźnienia gracz musi mieć możliwość zmiany swojej pozycji za pomocą klawiatury. Opóźnienie generowane jest za pomocą wielokrotnie już używanego trybu pracy Timer1, jakim jest CTC. Wartość wpisana do rejestru OCR1A wyznacza długość opóźnienia. Zakładamy, że fusebity mamy ustawione tak, że źródłem sygnału zegarowego jest nasz kwarc 16MHz. Preskaler timera został ustawiony na 1024, więc licznik timera zwiększany jest 15625 razy na sekundę. Wartość 8000 w OCR1A da więc nam opóźnienie ok. pół sekundy. Przeszkody w naszej grze będą się przesuwały mniej więcej o dwa miejsca na sekundę. Oczywiście parametr ten można dowolnie zmieniać w zakresie 1–65535 i tym samym zmieniać trudność gry.

Następnie wyświetlony jest komunikat zachęcający do wciśnięcia klawisza. Gdy użytkownik wciśnie jakiś klawisz, pobierany jest stan licznika timera (rejestr TCNT1). Po co? Chcemy zainicjalizować generator liczb pseudolosowych i potrzebujemy do tego jakiejś losowej liczby początkowej. Ponieważ przed wywołaniem funkcji getKey() oczekującej na wciśnięcie klawisza Timer1 już działa, po wciśnięciu przycisku licznik timera zdąży już doliczyć do jakiejś wartości. Będzie to wartość de facto losowa, bo gracz wciśnie przycisk zawsze z nieco innym opóźnieniem. W ten prosty sposób generator pseudolosowy będzie zawsze inaczej inicjowany i będzie zwracał różne liczby, co zapewni nam każdorazowo inne rozmieszczenie przeszkód w grze. Jeśli zamiast TCNT1 wpiszemy jakąś liczbę jako argument funkcji srand(), na przykład zero, wówczas gra będzie przebiegała zawsze tak samo.

Przed główną pętlą deklarujemy zmienną strukturalną, która będzie przechowywać położenie przeszkód w grze. Jest to zmienna typu disp_data, bo taką strukturę sobie zadeklarowaliśmy wcześniej. Ponieważ nasza zmienna jest zmienną strukturalną, musimy dodatkowo użyć słowa kluczowego struct.

Na początku głównej pętli inicjalizujemy stan gry. Do obu tablic w strukturze wpisujemy spacje, żeby potem przy wyświetlaniu uzyskać pusty wyświetlacz. Inicjalizowana jest zmienna pozycji gracza (0 – linia górna, 1 – linia dolna) oraz zmienna zapamiętująca, czy w poprzednim kroku wygenerowana została przeszkoda. Chodzi o to, żeby nie było sytuacji, gdy jest przeszkoda, a za nią, na przeciwnej linii znajduje się druga przeszkoda, co byłoby nie do przejścia dla gracza.

Wewnętrzna pętla gry, odpowiedzialna za kolejne iteracje dla każdego przesunięcia, składa się z kilku kroków. Najpierw wywoływana jest funkcja generateObstacle(), która może na skrajnej prawej pozycji wygenerować przeszkodę. Może, ponieważ to, czy przeszkoda się pojawi, czy nie, zależy od generatora pseudolosowego. Losowana jest też linia: górna bądź dolna. Funkcja zwraca liczbę 1, jeśli przeszkoda została wygenerowana. Jeśli została, w kolejnej iteracji nie generujemy przeszkody.

Listing 1

```
int main(void) {
 lcdInit();
 keybInit();
 // Timer1 w trybie CTC, preskaler /1024
 TCCR1B = _BV(CS12) | _BV(CS10) | _BV(WGM12);
 OCR1A = 8000;
 lcdString("Press key");
 lcdGotoXY(6, 1);
 lcdString("to play...");
 getKey();
 // inicjalizacja generatora pseudolosowego
 srand(TCNT1);
 struct disp_data displayData;
 while(1) {
  // wyczyszczenie danych wyświetlacza
  for (uint8_t i = 0; i < 16; i++) {
   displayData.upperLine[i] = ' ';
   displayData.lowerLine[i] = ' ';
  }
  uint8_t playerPosition = 0;
  uint8_t obstacleGenerated = 0;
  while (1) {
   // generowanie przeszkody
   if (!obstacleGenerated) {
        obstacleGenerated = generateObstacle(&displayData);
   } else {
        obstacleGenerated = 0;
   }
   // opóźnienie i odczyt klawiatury
   while(!(TIFR & _BV(OCF1A))) {
    processKeyboard(&displayData, &playerPosition);
   }
   TIFR |= _BV(OCF1A);
   // wykrycie kolizji i zakończenie gry
   if ((playerPosition == 0 && displayData.upperLine[1] == 'X') ||
    (playerPosition == 1 && displayData.lowerLine[1] == 'X')) {
    lcdGotoXY(3, 0);
    lcdString("GAME OVER");
    getKey();
    break;
   }
   // przesunięcie przeszkód w lewo
   shiftLeft(&displayData, playerPosition);
   // wysłanie stanu gry na wyświetlacz
   updateDisplay(&displayData);
  }
 }
}
```

Następnie wykonywane jest opóźnienie, podczas którego gracz może zmienić swoją pozycję. W pętli opóźniającej wywoływana jest funkcja processKeyboard(), która zmienia pozycję gracza w zależności od wciśniętego przycisku: 1 – pozycja górna, 5 – pozycja dolna.

Gdy opóźnienie się zakończy, sprawdzamy, czy nie nastąpiła kolizja gracza z przeszkodą. Ponieważ opóźnienie właśnie się skończyło i zaraz ma nastąpić przesunięcie, sprawdzamy, czy jest przeszkoda na pozycji drugiej wyświetlacza (w tablicy indeks 1). W zależności od pozycji gracza sprawdzana jest tablica odpowiedzialna za linię górną lub dolną. Przy kolizji wyświetlany jest napis GAME OVER i po wciśnięciu klawisza następuje wyjście z pętli wewnętrznej.

Tak działa funkcja main() i tak ogólnie wygląda działanie programu. Popatrzmy jeszcze na funkcje pomocnicze. Każda z nich operuje na naszej strukturze i pobiera wskaźnik na nią. W związku z tym przekazywanym typem jest struct disp_data *.

Funkcja updateDisplay() służy do wyświetlenia stanu gry na LCD (**listing 2**). Najpierw ustawia kursor na początku górnej linii i wyświetla zawartość tablicy upperLine ze struktury. Następnie funkcja ustawia kursor na początku dolnej linii i wyświetla tablicę lowerLine.

Za pomocą funkcji shiftLeft() realizowane jest przesunięcie przeszkód (**listing 3**). Do kolejnych pól obu tablic wpisywane są wartości z pól następnych, przez co realizowane jest przesunięcie. Pola o indeksie 15, odpowiadające skrajnym prawym pozycjom, uzupełniane są spacjami. Na koniec wyświetlany jest znak > symbolizujący pozycję gracza, na dolnej lub górnej linii.

GenerateObstacle() to funkcja zawierająca logikę generującą przeszkody w losowych miejscach (**listing 4**). Źródłem danych losowych jest standardowa funkcja biblioteczna rand(). Zwraca ona wartości liczbowe całkowite, a nam potrzebna jest tylko decyzja tak/nie, więc operatorem modulo (%) sprawdzamy, czy zwrócona liczba jest parzysta. W zależności od obliczonej wartości przechodzimy do linii górnej bądź dolnej i znów pobieramy losową liczbę, aby sprawdzić, czy ma być wygenerowana przeszkoda. Przeszkody są oznaczane dużą literką X. Jeśli przeszkoda została wygenerowana, czyli wstawiona do tablicy odpowiedzialnej za daną linijkę, zwracana jest liczba 1. W przeciwnym wypadku liczba 0.

Obsługę przemieszczania gracza zapewnia funkcja processKeyboard(), przedstawiona na listingu 5. Jeśli wciśnięty został klawisz, sprawdzany jest jego numer. Dla klawisza 1 gracz jest przenoszony na górną linię, a dla klawisza 5 na dolną. Nowa pozycja jest oznaczana symbolem >, a poprzednia jest kasowana przez nadpisanie spacją.

Pełny kod gry dostępny jest w materiałach dodatkowych dla tego numeru EdW na Elportalu. Zachęcam do przejrzenia go i przerabiania gry na różne sposoby.

```
void updateDisplay(struct disp_data * displayData) {
 lcdGotoXY(0, 0);
 for (uint8_t i = 0; i < 16; i++)
  lcdWriteData(displayData->upperLine[i]);
 lcdGotoXY(0, 1);
 for (uint8_t i = 0; i < 16; i++)
  lcdWriteData(displayData->lowerLine[i]);
}
```

Listing 2

```
void shiftLeft(struct disp_data * displayData, uint8_t playerPosition) {
 for (uint8_t i = 0; i < 15; i++) {
  displayData->upperLine[i] = displayData->upperLine[i + 1];
  displayData->lowerLine[i] = displayData->lowerLine[i + 1];
 }
 displayData->upperLine[15] = ' ';
 displayData->lowerLine[15] = ' ';
 if (playerPosition) displayData->lowerLine[0] = '>';
  else displayData->upperLine[0] = '>';
}
```

Listing 3

```
uint8_t generateObstacle(struct disp_data * displayData) {
 int randomNumber = rand();
 if (randomNumber % 2) {
  randomNumber = rand();
  if (randomNumber % 2) {
   displayData->upperLine[15] = 'X';
   return 1;
  } else {
   displayData->upperLine[15] = ' ';
   return 0;
  }
 } else {
  randomNumber = rand();
  if (randomNumber % 2) {
   displayData->lowerLine[15] = 'X';
   return 1;
  } else {
   displayData->lowerLine[15] = ' ';
   return 0;
  }
 }
}
```

Listing 4

Przerwania zewnętrzne

Zdecydowana większość przerwań, jakie mogą być generowane w mikrokontrolerze ATmega32, pochodzi z peryferii takich jak timery czy układy transmisji szeregowej. Przerwania można jednak też generować bezpośrednio za pomocą pinów INT0, INT1 i INT2. W naszym mikrokontrolerze są to odpowiednio piny PD2, PD3 i PB2. W przypadku pinów INT0 i INT1 przerwanie może być wyzwalane zarówno stanem logicznym (niskim), jak i zboczem, a dla pinu INT2 tylko zboczem. Jeśli skonfigurowane będzie wyzwalanie niskim stanem logicznym, przerwanie będzie generowane ciągle, tak długo jak będzie się utrzymywało logiczne zero na pinie.

Konfigurację wyzwalania przerwań dla pinów INT0 i INT1 przeprowadza się w rejestrze MCUCR. Za pin INT0 odpowiedzialne są bity ISC01 ISC00, a za pin INT1 piny ISC11 ISC10. Ustawienia bitów przedstawione są w **tabeli 1**.

W przypadku INT2 możemy tylko wybrać, czy przerwanie ma być dla narastającego, czy opadającego zbocza. Odpowiada za to bit ISC2 w rejestrze MCUCSR. Zero konfiguruje przerwanie dla zbocza opadającego, jedynka dla narastającego. Ponieważ zmiana tego bitu może spowodować wyzwolenie przerwania, najlepiej konfigurować go przed włączeniem przerwań.

Przerwania dla pinów INT0, INT1 i INT2 włącza się, ustawiając jedynki w bitach o tych samych nazwach w rejestrze GICR. Z kolei wystąpienie przerwania można sprawdzić, odczytując flagi w rejestrze GIFR, są to bity INTF0, INTF1 i INTF2.

Załóżmy, że chcielibyśmy wyzwolić przerwanie INT2 za pomocą zbocza opadającego. Zbocze to musi więc pojawić się

```
void processKeyboard(struct disp_data * displayData, uint8_t * playerPosition) {
 uint8_t key = readKeyboard();
 if (key) {
  if (key == 1) {
   *playerPosition = 0;
   displayData->upperLine[0] = '>';
   if (displayData->lowerLine[0] == '>') displayData->lowerLine[0] = ' ';
  }
  if (key == 5) {
   *playerPosition = 1;
   displayData->lowerLine[0] = '>';
   if (displayData->upperLine[0] == '>') displayData->upperLine[0] = ' ';
  }
  updateDisplay(displayData);
 }
}
```

Listing 5

Tabela 1

ISC01/ISC11	ISC00/ISC10	Źródło przerwania
0	0	Niski stan logiczny
0	1	Każda zmiana stanu logicznego
1	0	Zbocze opadające
1	1	Zbocze narastające

Struktury w języku C

Struktury są sposobem organizacji zmiennych w większe zmienne. Używamy ich do wartości, które są logicznie ze sobą powiązane i zwykle przetwarzane razem. Dzięki strukturom możemy wiele wartości przechowywać w jednej zmiennej i łatwo przekazywać do funkcji. Strukturę deklarujemy następująco:

```
struct typ_struktury {
   typ_pola_1 pole1;
   typ_pola_2 pole2;
} zmienna_strukturalna;
```

Słowo kluczowe struct jest obowiązkowe. Ponieważ nasza struktura jest nowym typem zmiennych, musimy wymyślić nazwę dla tego typu. Będziemy go potem używać tak, jak używamy typów uint8_t, float itd. W nawiasach klamrowych definiujemy składniki struktury, podobnie jak deklarujemy zwykłe zmienne. Tutaj deklarację możemy zakończyć, tak akurat jest w naszej grze. Będzie to sama definicja typu strukturalnego. Ale przed średnikiem możemy jeszcze podać nazwę zmiennej. W ten sposób nie tylko zadeklarujemy typ strukturalny, ale od razu stworzymy zmienną tego typu. Jeśli chcemy deklarować zmienne na podstawie naszego typu strukturalnego później, piszemy:

```
struct typ_struktury zmienna_strukturalna;
```

Czyli po prostu opuszczamy nawias klamrowy, bo deklaracja typu była już wcześniej i kompilator go zna. Deklaracja wygląda więc podobnie jak przy standardowych typach, ale musimy zacząć od słowa kluczowego struct.

A jak się dostać do wnętrza zmiennej strukturalnej? Służy do tego operator kropki:

```
zmienna_strukturalna.pole1 = 5;
```

W stosunku do zmiennych strukturalnych bardzo często korzystamy ze wskaźników, np. przekazując te zmienne do funkcji. Posiłkujemy się wtedy zmienną wskaźnikową pokazującą na typ strukturalny:

```
struct typ_struktury * wskaznik_na_strukture;
wskaznik_na_strukture = &zmienna_strukturalna;
```

Ponieważ wskaźnik nie jest strukturą, nie możemy użyć kropki, aby dostać się do pól struktury, na którą pokazuje. Musimy do tego użyć operatora strzałki:

```
wskaznik_na_strukture->pole1 = 5;
```

Ewentualnie można wyłuskać najpierw oryginalną zmienną operatorem gwiazdki i potem użyć kropki, ale jest to rzadko spotykane:

```
(*wskaznik_na_strukture).pole1 = 5;
```

na pinie PB2, bo to on jest odpowiedzialny za przerwanie INT2. Wygenerowanie zbocza opadającego można spowodować, podłączając do pinu układ, który wymusi na pinie zmianę stanu wysokiego na niski. Mogą to być najróżniejsze układy cyfrowe, a przy zapewnieniu odpowiedniej stromości zbocza także analogowe. Skorzystajmy z najprostszej opcji, jaką jest przycisk.

W ogólnym przypadku podciągnęlibyśmy pin mikrokontrolera do plusa za pomocą rezystora, a przycisk włączylibyśmy między pin a masę. Naciśnięcie przycisku powodowałoby zmianę stanu na pinie z jedynki na zero. Jak to jednak zrobić na naszej płytce testowej, która nie ma pojedynczych przycisków, tylko całą ich matrycę 4x4? Brak też rezystorów podciągających. Jak pamiętamy z pierwszych lekcji, ATmega32 zawiera wewnętrzne rezystory podciągające. Wystarczy do rejestru PORTx wpisać jedynkę na bicie odpowiadającym wybranemu pinowi, przy czym pin pozostaje skonfigurowany jako wejście. W naszym przypadku napiszemy więc:

```
PORTB |= _BV(PB2);
```

A co ze zwarciem do masy? W naszej klawiaturze matrycowej nie ma zwierania do masy ani plusa zasilania a jedynie tworzenie zwarć między pinami złącza klawiatury w momencie naciśnięcia przycisku: pin wiersza zostaje zwarty z pinem kolumny. Przykładowo więc przycisk S1 powoduje zwarcie pinów W1 i K1 złącza KEYB. Wystarczy jeden z tych pinów, np. W1, połączyć kabelkiem z masą, a drugi (K1) z pinem PB2. Wtedy naciśnięcie przycisku S1 zmieni stan na pinie PB2 z jedynki na zero, czyli wygeneruje nam zbocze opadające.

Ponieważ interesuje nas zbocze opadające, do rejestru MCUCSR nic nie wpisujemy, bo na bicie ISC2 już jest domyślnie zero. Musimy natomiast włączyć przerwanie INT2 w rejestrze GICR, ustawiając bit o tej samej nazwie. W kodzie musi też znaleźć się funkcja obsługi przerwania. Z lekcji 8, z tabeli 1, wiemy, że musi mieć ona postać:

```
ISR(INT2_vect) {
}
```

We wnętrzu funkcji dodajmy coś, co zasygnalizuje wywołanie przerwania, np. zaświecenie diody na pinie 1 portu B. Całość kodu będzie wyglądała jak na **listingu 6** poniżej:

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

int main(void)
{
 //podciągnięcie pinu
 PORTB |= _BV(PB2);
 //pin z LEDem jako wyjście
 DDRB |= _BV(DDB1);
 //włącz przerwanie INT2
 GICR |= _BV(INT2);
 //włącz przerwania globalnie
 sei();
 while (1)
 {
 }
}

ISR(INT2_vect) {
 //włącz LED
 PORTB |= _BV(PB1);
}
```

Przerwania zewnętrzne działają niezależnie od tego, czy pin jest skonfigurowany jako wejście, czy jako wyjście. Jeśli więc pin skonfigurowany jest jako wyjście, możemy ustawić na nim określony stan logiczny i w ten sposób spowodować wyzwolenie się przerwania całkowicie programowo. Popatrzmy na przykład, **listing 7** poniżej:

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

int main(void) {
 //podciągnięcie pinu INT2
 PORTB |= _BV(PB2);
 //pin z LEDem jako wyjście
 DDRB |= _BV(DDB1);
 //PD3(INT1) jako wyjście
 DDRD |= _BV(DDD3);
 //przerwanie przy zboczu narastającym
 MCUCR = _BV(ISC11) | _BV(ISC10);
 //włącz przerwania INT1 i INT2
 GICR = _BV(INT1) | _BV(INT2);
 //włącz przerwania globalnie
 sei();
 while (1) {}
}

ISR(INT1_vect) {
 //włącz LED
 PORTB |= _BV(PB1);
}

ISR(INT2_vect) {
 //wyzwól INT1
 PORTD |= _BV(PD3);
}
```

Tutaj konfigurujemy dwa przerwania zewnętrzne INT1 i INT2. Tylko to drugie jest wyzwalane przyciskiem. Przerwanie INT1 wyzwalane jest zboczem narastającym na pinie 3 portu D, ale do portu tego nic nie podłączamy. Natomiast w funkcji obsługującej INT2 ustawiamy jedynkę na tym pinie. Powoduje to wywołanie przerwania INT1. Jest to więc swego rodzaju kaskada przerwań. Ale jedynkę na PD3 możemy ustawić też w głównym programie, bez udziału przycisku – **listing 8**:

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

int main(void)
{
 //pin z LEDem jako wyjście
 DDRB |= _BV(DDB1);
 //PD3(INT1) jako wyjście
 DDRD |= _BV(DDD3);
 //przerwanie przy zboczu narastającym
 MCUCR = _BV(ISC11) | _BV(ISC10);
 //włącz przerwanie INT1
```

```
 GICR = _BV(INT1);
 //włącz przerwania globalnie
 sei();
 //ustaw jedynkę na PD3(INT1)
 PORTD |= _BV(PD3);
 while (1)
 {
 }
}

ISR(INT1_vect) {
 //włącz LED
 PORTB |= _BV(PB1);
}
```

Po uruchomieniu tego programu LED zaświeci się od razu. Przy pracy z przerwaniami zewnętrznymi warto pamiętać, że mogą one zostać wywołane także całkowicie wewnętrznie.

Komparator analogowy

Mikrokontroler ATmega32 jest układem cyfrowym, ale ma też obwody analogowe. Jednym z nich jest komparator porównujący napięcia na pinach AIN0 (PB2) i AIN1 (PB3). Komparator ten bazuje na wzmacniaczu operacyjnym, którego wejście nieodwracające połączone jest z AIN0, a odwracające z AIN1. W związku z tym na wyjściu komparatora jedynka będzie obecna, gdy napięcie na AIN0 będzie większe niż na AIN1.

Za konfigurację komparatora analogowego odpowiedzialny jest rejestr ACSR. Zawiera on następujące bity:

- ACD – wpisanie 1 wyłącza komparator, domyślnie 0 – komparator włączony
- ACBG – użycie wewnętrznego napięcia odniesienia zamiast napięcia z AIN0
- ACO – wyjście komparatora
- ACI – flaga przerwania z komparatora
- ACIE – włączenie przerwania
- ACIC – uruchomienie funkcji Input Capture w Timer1
- ACIS1, ACIS0 – opcje przerwania

Jak widać, komparator domyślnie jest włączony i wystarczy odczytywać wartość bitu ACO z rejestru ACSR, aby wiedzieć, czy napięcie na AIN0 jest większe niż na AIN1, czy też odwrotnie. Sprawdźmy to prostym programem, który stan bitu ACO będzie prezentował za pomocą LED-a podłączonego do pinu PB0 – **listing 9**:

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

int main(void)
{
 DDRB |= _BV(DDB0);
 while(1)
 {
  if (ACSR & _BV(ACO)) {
   PORTB |= _BV(PB0);
  } else {
   PORTB &= ~_BV(PB0);
  }
 }
}
```

Aby przetestować kod, potrzebujemy dwóch napięć podłączonych do PB2 i PB3. Na płytce testowej mamy potencjometr, który działa jako dzielnik napięcia. Podłączmy go więc (pin Pot na złączu MISC) do pinu PB3 (AIN1). W ten sposób na AIN1 będziemy mogli ustawić dowolne napięcie z zakresu 0–5V. Do PB2 (AIN0) podłączamy zaś jakieś inne źródło napięcia, np. baterię „paluszek” (ok. 1,5V) czy baterię z telefonu komórkowego (ok. 3,7V).

Gdy napięcie z podłączonego źródła będzie większe niż napięcie z potencjometru, dioda podłączona do PB0 będzie świecić. Jeśli będzie mniejsze, dioda zgaśnie.

Wpisując 1 do bitu ACBG, przełączamy wejście nieodwracające komparatora na wewnętrzne źródło napięcia odniesienia. Producent mikrokontrolera podaje, że napięcie odniesienia ma wartość 1,15–1,35V, typowo 1,23V. Zmierzmy jego wartość. Ale jak to zrobić, skoro nie jest wyprowadzone na zewnątrz? Wystarczy za pomocą potencjometru zmieniać napięcie na AIN1 i uchwycić moment, gdy dioda będzie się przełączała. Wtedy wystarczy zmierzyć napięcie z potencjometru, będzie ono mniej więcej równe napięciu odniesienia. Jedyne, co trzeba zmienić w kodzie, to przed pętlą while(1) dopisać linijkę:

```
ACSR = _BV(ACBG);
```

Jeśli potrzebujemy, aby komparator generował przerwanie przy zmianie swojego stanu, ustawiamy bit ACIE. Tak jak i w innych przypadkach, przerwanie można obsłużyć na dwa sposoby. Jeden to sprawdzanie flagi (tutaj jest to bit ACI) i jej czyszczenie przez wpisanie 1, drugi to funkcja obsługi przerwania. Dla komparatora analogowego będzie miała ona postać ISR(ANA_COMP_vect) {}. Oczywiście aby funkcja działała, oprócz włączenia przerwania komparatora trzeba też włączyć przerwania globalnie funkcją sei(). A kiedy właściwie będzie włączało się przerwanie? Określają to bity ACIS1 i ACIS0 zgodnie z **tabelą 2**.

ACIS1	ACIS0	Przerwanie dla
0	0	Każda zmiana stanu komparatora
0	1	-
1	0	Zbocze opadające (zmiana z 1 na 0)
1	1	Zbocze narastające (zmiana z 0 na 1)

Tabela 2

Ciekawą opcją konfiguracyjną jest bit ACIC. Ustawienie go na 1 powoduje, że w momencie wystąpienia przerwania z komparatora uruchomiona zostaje funkcja Input Capture w Timer1, czyli skopiowanie aktualnego stanu jego licznika do rejestru ICR1. Funkcję tę omawialiśmy w lekcji 4. Dzięki wyzwalaniu Input Capture przy zmianie stanu komparatora otrzymujemy informację o tym, w którym momencie cyklu pracy Timer1 nastąpiła zmiana napięcia.

MUX2	MUX1	MUX0	Wejście komparatora
0	0	0	ADC0
0	0	1	ADC1
0	1	0	ADC2
0	1	1	ADC3
1	0	0	ADC4
1	0	1	ADC5
1	1	0	ADC6
1	1	1	ADC7

Tabela 3

Wspomnieliśmy, że wejście nieodwracające komparatora może być podłączone do pinu AIN0 lub do wewnętrznego napięcia odniesienia. Z kolei wejście odwracające jest połączone z AIN1, ale może być też połączone z jednym z wejść ADC0–ADC7. Wejścia te w naszym mikrokontrolerze to piny PA0–PA7. Aby przełączyć komparator na któryś z tych pinów, musimy ustawić na 1 bit ACME w rejestrze SFIOR. Ponadto nie może być włączony przetwornik analogowo-cyfrowy, który będziemy poznawać w następnej lekcji. Jeśli oba te warunki są spełnione, wejście dla komparatora można wybrać za pomocą bitów MUX2..0 w rejestrze ADMUX. Wartości tych bitów tworzą liczbę dwójkową odpowiadającą numerowi wejścia. Zobrazowane to zostało w **tabeli 3**.

Do czego może być nam potrzebne takie przełączanie wejść? Daje to dużo większą elastyczność, bo dzięki temu napięcie z AIN0 lub wewnętrznego źródła może być porównywane nie z jednym (AIN1), ale łącznie aż dziewięcioma napięciami doprowadzonymi do pinów AIN1 i ADC0–ADC7.

Zadania

Zarówno przerwania zewnętrzne, jak i komparator analogowy są stosunkowo proste do opanowania, napisanie własnych programów z ich wykorzystaniem nie powinno być więc trudne. Oto przykładowe zadania:

1. Zewnętrzne przerwanie wyzwalane przyciskiem powoduje zaświecenie LED-a na 1 sekundę w głównym programie
2. Zmierzchowy wyłącznik światła w oparciu o fotorezystor, najlepiej z histerezą.

Grzegorz Niemirowski
grzegorz@grzegorz.net