Kurs AVR – lekcja 12

Rozwiązania zadań z ostatniego odcinka

Przed miesiącem omawialiśmy przerwania zewnętrzne i komparator analogowy. ***W pierwszym zadaniu domowym*** trzeba było zaświecać LED na 1 sekundę po wystąpieniu przerwania zewnętrznego. Jest to mikroprocesorowa wersja przerzutnika monostabilnego. Zadanie nie było trudne, wystarczyło lekko przerobić kod z listingu 6. W przykładzie zaświecenie diody było wykonywane w funkcji obsługującej przerwanie, natomiast w zadaniu faktyczne włączenie diody miało następować w głównej pętli. W funkcji obsługi przerwania trzeba więc jakoś poinformować główną pętlę o konieczności włączenia LED-a. Można to zrobić, wykorzystując globalną zmienną, oczywiście zadeklarowaną jako volatile, która będzie odczytywana w głównej pętli. Jeśli zmienna zostanie ustawiona, włączony zostanie LED, a po sekundzie będzie mógł zostać wyłączony. Kompletny kod rozwiązania znajduje się poniżej na **listingu 1.**

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
volatile uint8_t interrupt = 0;

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) {
  if (interrupt) {
   PORTB |= _BV(PB1);
   _delay_ms(1000);
   PORTB &= ~_BV(PB1);
   interrupt = 0;
  }  } }
ISR(INT2_vect) {  //włącz LED
 interrupt = 1;
}
```

W drugim zadaniu mieliśmy zrobić zmierzchowy wyłącznik światła, korzystając z komparatora analogowego. Jedno z porównywanych przez niego napięć musi zależeć od natężenia światła. Potrzebujemy więc dzielnika napięcia, który w jednej gałęzi będzie miał fotorezystor, a w drugiej zwykły rezystor o dobranej wartości lub potencjometr. W zależności od tego, czy fotorezystor zostanie w dzielniku umieszczony od strony masy, czy od plusa zasilania oraz czy jego rezystancja będzie malała, czy rosła wraz ze wzrostem natężenia oświetlenia, maleć lub rosnąć będzie napięcie z dzielnika.

Drugie napięcie może pochodzić z potencjometru pracującego jako dzielnik napięcia, tak jak to robiliśmy w poprzedniej lekcji. Pozwoli to na wygodną regulację progu przełączania się naszego wyłącznika zmierzchowego. Dzielnik można oczywiście wykonać z rezystorów lub też zamiast dzielnika wykorzystać wewnętrzne napięcie odniesienia (ustawiony bit ACBG w rejestrze ACSR), wtedy jednak próg przełączania będzie można regulować, tylko dobierając wartości rezystorów.

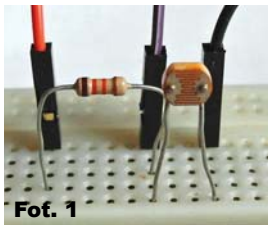
Załóżmy, że źródłem napięcia odniesienia będzie potencjometr znajdujący się na naszej płytce testowej. Potrzebujemy jeszcze dzielnika napięcia z fotorezystorem. Na płytce nie ma fotorezystora, dzielnik więc trzeba zmontować poza płytką testową, np. na płytce stykowej (**fotografia 1**). Jeśli w dzielniku fotorezystor będzie od strony masy, a jego rezystancja będzie rosła wraz ze spadającym natężeniem oświetlenia, to zapadający zmrok będzie powodował wzrost napięcia na wyjściu dzielnika.

Mamy więc dwa źródła napięcia (potencjometr oraz dzielnik z fotorezystorem) i dwa wejścia komparatora: nieodwracające (AIN0/PB2) i odwracające (AIN1/PB3). Wyjście komparatora, czyli bit AC0 w rejestrze ACSR, będzie ustawiony, gdy na AIN0 będzie większe napięcie niż na AIN1. Jeśli więc dzielnik z fotorezystorem podłączymy do PB2, a potencjometr do PB3, to zmrok spowoduje ustawienie bitu AC0. Sprawdzając stan tego bitu, możemy włączyć element wykonawczy, np. LED. Jeśli miałby to być prawdziwy wyłącznik sterujący oświetleniem, wówczas zamiast diody podłączymy tranzystor z przekaźnikiem lub optotriak z triakiem. Kod źródłowy, zakładając podłączenie elementu wykonawczego do PB0, będzie wyglądał jak na **listingu 2**.

```
#include <avr/io.h>
int main(void) {
 DDRB |= _BV(DDB0);
 while(1) {
  if (ACSR & _BV(ACO)) {
   PORTB |= _BV(PB0);
  } else {
   PORTB &= ~_BV(PB0);
  }   }   }
```

Jak widać, kod jest bardzo prosty i w sumie spełnia swoje zadanie. Jednak testując zachowanie naszego wyłącznika zmierzchowego, można zauważyć przyciemnione świecenie LED-a, gdy wyłącznik jest na granicy przełączenia. Przełączenie nie następuje więc w sposób czysty, ale na wyjściu pojawiają się śmieci będące efektem zakłóceń zbieranych przez wejścia komparatora. Zakłócenia te zwykle nie są duże, ale w okolicach progu przełączania są silnie wzmacniane przez komparator.

Ponadto jeśli nasz wyłącznik steruje źródłem światła, które w jakimś stopniu oświetla fotorezystor z naszego dzielnika, to powstaje ujemne sprzężenie zwrotne. Gdy zapada zmrok, włączane jest oświetlenie, które powoduje zwiększenie ilości światła i tym samym wyłączenie oświetlenia. Cykl ten się powtarza i wyłącznik wpada w oscylacje. Nie działa więc prawidłowo i może być niebezpieczny dla elementów takich jak przekaźnik wykonawczy, który zacznie iskrzyć. Jednym z rozwiązań jest dodatnie sprzężenie zwrotne. Wyjście komparatora, a dokładniej pin sterowany na podstawie stanu komparatora, możemy połączyć z wejściem za pomocą rezystora o dużej wartości (kilkaset kΩ) z wejściem nieodwracającym. Wtedy wyjście będzie niejako „pomagało" wejściu pozostać w stanie, w którym się aktualnie znajduje. Ze względu na to dodatkowe małe podciągnięcie, małe zakłócenia na wejściu nie będą się przenosiły na wyjście. Z progu przełączania powstaną de facto dwa progi. Aby nastąpiło włączenie, napięcie będzie musiało wzrosnąć ponad górny próg, a aby nastąpiło wyłączenie, będzie musiało spaść poniżej dolnego progu. Im mniejszy będzie rezystor dodatniego sprzężenia zwrotnego, tym odległość między tymi progami, zwana histerezą, będzie większa.


Fot. 1

Histerezę możemy też zrealizować, nie stosując dodatniego sprzężenia zwrotnego, ale dwa napięcia odniesienia. Ponieważ komparator ma tylko dwa wejścia, konieczne jest zastosowanie przełączania. Jak pamiętamy z lekcji 11, wejście odwracające jest wewnątrz mikrokontrolera połączone z pinem PB3, ale można je podłączyć też do jednego z pinów portu A, np. do PA0. Przepięcie wejścia odwracającego z domyślnego pinu na pin portu A wymaga ustawienia bitu ACME w rejestrze SFIOR. Pin wybierany jest za pomocą bitów MUX2..0 rejestru ADMUX. Możemy rejestr ten zostawić w stanie domyślnym, z wyzerowanymi bitami, co spowoduje wybór pinu PA0. Do wejścia tego podłączymy drugi potencjometr, analogicznie do tego znajdującego się na płytce testowej, podłączonego do PB3. W kodzie programu musimy ustawicznie przełączać wejście odwracające komparatora pomiędzy pinami i tym samym porównywać napięcie na wejściu nieodwracającym, pochodzącym z dzielnika z fotorezystorem, z napięciami z potencjometrów. Dzięki temu będzie można określić, czy napięcie to jest niższe od tych dwóch napięć, jest między nimi lub też jest wyższe od obu napięć. Jeśli będzie niższe od obu, nasz wyłącznik powinien być wyłączony. Jeśli wyższe od obu, powinien być włączony. Natomiast jeśli będzie pomiędzy, powinien zachowywać swój aktualny stan.

Przykładowa realizacja przedstawiona jest poniżej na **listingu 3**.

```
#include <avr/io.h>
int main(void) {
 DDRB |= _BV(DDB0);
 uint8_t v1, v2;
 while(1) {   //teraz odczyt z PB3
  SFIOR &= ~_BV(ACME);
 v1 = ACSR & _BV(ACO); //i odczyt z PA0
  SFIOR |= _BV(ACME);
  v2 = ACSR & _BV(ACO);
  if (v1 && v2) {
   PORTB |= _BV(PB0);
  }
  if (!v1 && !v2) {
   PORTB &= ~_BV(PB0);
  }  }  }
```

W głównej pętli wejście odwracające podczepiane jest pod pin PB3, a stan komparatora zapamiętywany jest w zmiennej v1. Następnie wejście jest dołączane do PA0, a wynik z komparatora trafia do zmiennej v2. W ten sposób zmienne v1 i v2 przechowują informację o tym, czy napięcie na wejściu nieodwracającym przekroczyło progi wyznaczone potencjometrami podłączonymi do PB3 i PA0. Pozostaje więc sprawdzenie warunków. Jeśli obie zmienne są różne od zera (oba warunki spełnione), LED zostaje wyłączony. Jeśli obie są zerami, LED zostaje wyłączony. Jeśli jedna ze zmiennych zawiera zero (fałsz), a druga wartość różną od zera (prawdę), wyłącznik zmierzchowy jest w obszarze histerezy i nie zmienia swojego stanu. Podczas testów widać, że przełączanie diody następuje w sposób pewny, bez stanów pośrednich.W kodzie warto zwrócić uwagę na operator AND. W przypadku sprawdzenia stanu bitu użyty jest pojedynczy symbol & oznaczający bitową operację AND. Przy sprawdzeniu warunku występuje już podwójny symbol &&, oznaczający logiczną operację AND. Wykrzyknik to oczywiście operator zaprzeczenia.

Tabela 1

REFS1	REFS0	Źródło napięcia
0	0	AREF
0	1	AVCC
1	0	-
1	1	Wewnętrzne 2,56V

Przetwornik analogowo-cyfrowy

Mikrokontroler ATmega32 wyposażony jest w przetwornik analogowo-cyfrowy (Analog to Digital Converter, ADC) umożliwiający pomiar zewnętrznych napięć. Jest to bardzo przydatna funkcja, dzięki której możemy przetwarzać różne sygnały analogowe, np. z czujników, albo odczytywać położenie potencjometru. Jak działa taki przetwornik? W naszym przypadku opiera się na sukcesywnej aproksymacji, czyli stopniowym przybliżaniu. Jego głównym składnikiem jest przetwornik cyfrowo-analogowy (DAC), który może wytwarzać różne napięcia z danego napięcia odniesienia. Wytwarzane przez niego napięcie jest porównywane z napięciem mierzonym i w kolejnych, coraz mniejszych krokach korygowane tak, aby uzyskać równość obu napięć. Nasz przetwornik nie mierzy więc napięcia bezpośrednio, ale steruje przetwornikiem cyfrowo-analogowym tak, aby uzyskać identyczne napięcia. Wartość, na jaką w ostatniej fazie został ustawiony przetwornik cyfrowo-analogowy, staje się wartością zwróconą jako wynik działania przetwornika analogowo-cyfrowego. Ze względu na wykorzystanie sukcesywnej aproksymacji pojawiają się dwie ważne obserwacje. Po pierwsze pomiar trwa pewien czas, nie otrzymamy więc wyniku pomiaru natychmiast. Po drugie wynik pomiaru staje się coraz dokładniejszy w trakcie jego trwania. Pomiar można więc skrócić kosztem dokładności.

Przeanalizujmy najprostszy przypadek użycia ADC. Jak wspomniano, przetwornik mierzy napięcie względem napięcia odniesienia. Nasz przetwornik jest 10-bitowy i wartość 0 będzie oznaczała 0V, a wartość 1023 wartość równą napięciu odniesienia (minus wartość przypadającą na 1 bit). Wybierając napięcie odniesienia, wybieramy więc zakres mierzonych napięć oraz dokładność pomiaru. W ATmega32 można wybrać trzy napięcia odniesienia: napięcie na nóżce AREF, napięcie na nóżce AVCC (zasilające ADC) oraz wewnętrzne napięcie odniesienia 2,56V. Wyboru dokonujemy za pomocą bitów REFS1 i REFS0 w rejestrze ADMUX zgodnie z **tabelą 1**.

Należy pamiętać, że jeśli wybierzemy AVCC lub wewnętrzne źródło 2,56V, to zostaną one wewnętrznie połączone z pinem AREF. Nie można więc podłączać zewnętrznych źródeł napięcia do tego pinu, jeśli wybrana jest któraś z tych dwóch opcji. Dobrze jest natomiast podłączyć między AREF i masę kondensator dodatkowo wygładzający napięcie odniesienia. Można powiedzieć, że de facto zawsze źródłem napięcia odniesienia jest AREF, tylko że raz napięcie jest tam podawane z zewnątrz mikrokontrolera, a raz z wewnątrz.

Napięcie na nóżce AVCC nie może się różnić o więcej jak 0,3V względem napięcia zasilającego VCC. Nie podłącza się więc zwykle do niej jakichś specjalnie generowanych napięć, tylko po prostu napięcie zasilające, filtrując je po drodze obwodem LC. Nasza płyta testowa zasilana jest napięciem 5V, wybranie więc AVCC jako źródła napięcia odniesienia da nam zakres pomiarowy 0–5V.

Obwody przetwornika A/C odpowiedzialne za proces sukcesywnej aproksymacji wymagają taktowania odpowiednią częstotliwością. Dla maksymalnej, 10-bitowej rozdzielczości przetwornika, ta częstotliwość musi się zawierać w przedziale od 50 do 200kHz. ADC wytwarza tę częstotliwość przez podział częstotliwości głównego zegara mikrokontrolera. Jeśli nasz mikrokontroler taktowany jest kwarcem 16MHz, to potrzebny jest podział w zakresie od 16 000 000 / 200 000 = 80 do 16 000 000 / 50 000 = 320. Podział konfigurujemy za pomocą bitów ADPS2..0 w rejestrze ADCSRA zgodnie z **tabelą 2**. Ogólnie można powiedzieć, że jest to podział przez 2^n, gdzie n jest 3-bitową liczbą utworzoną przez bity ADPS2..0. Wyjątkiem jest n = 0, dla którego podział jest tak jak dla n = 1, czyli przez 2.

Ponieważ dla taktowania 16MHz potrzebujemy podziału w zakresie 80–320, do wyboru mamy w sumie tylko jedną wartość, czyli 128. Musimy więc ustawić wszystkie trzy bity ADPSn. Normalnie konwersja A/C zajmuje 13 taktów zegara przetwornika, a pierwsza konwersja po włączeniu ADC zajmuje 25 taktów. Przy procesorze taktowanym kwarcem 16MHz i podziale przez 128, przetwornik będzie mógł więc dostarczać wyniki najwyżej ok. 9600 razy na sekundę.

Tabela 2

ADPS2	ADPS1	ADPS	Podział przez
0	0	0	2
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

Aby korzystać z ADC, musimy go najpierw włączyć. Robimy to, ustawiając bit ADEN w rejestrze ADCSRA. Następnie trzeba uruchomić pomiar, służy do tego bit ADSC w tym samym rejestrze. O zakończeniu pomiaru zostaniemy poinformowani poprzez pojawienie się jedynki w bicie ADIF, również w rejestrze ADCSRA. Alternatywnie można obserwować bit ADSC. Po zakończeniu konwersji jest on bowiem z powrotem ustawiany na 0, żeby wpisując do niego 1, móc rozpocząć kolejną konwersję.

Gdy wynik konwersji analogowo-cyfrowej jest gotowy, możemy go odczytać. Ponieważ wynik jest 10-bitowy, a rejestry mikrokontrolera są 8-bitowe, przechowywany jest on w dwóch rejestrach: ADCL i ADCH. Jak można się domyślić, ADCL zawiera niższe (młodsze) bity, a ADCH zawiera wyższe (starsze) bity. Taką sytuację znamy już z Timer1, który miał rejestry takie jak OCR1AH i OCR1AL, aby móc operować na 16-bitowych wartościach. Tak jak wtedy tak i teraz z poziomu języka C istnieje możliwość odwołania się do całej wartości przez odpowiednie makro, tutaj jest to ADCW. Inicjalizacja i odczyt wartości z ADC będzie więc wyglądać następująco:

```
ADCSRA |= _BV(ADEN); //włącz ADC
ADMUX |= _BV(REFS0); //AVCC
//preskaler 128
ADCSRA |=_BV(ADPS0)|_BV(ADPS1)|_BV(ADPS2);
//rozpocznij konwersję
ADCSRA |= _BV(ADSC);
//zaczekaj na wynik
while(ADCSRA & _BV(ADSC));
//pobierz wynik
uint16_t adcValue = ADCW;
```

Uruchomienie i test przetwornika A/C

Sprawdźmy działanie ADC w praktyce. Jako źródło napięcia można wykorzystać potencjometr znajdujący się na płytce testowej, dzięki niemu będzie można łatwo sprawdzić pełny zakres ADC. Warto poeksperymentować również z innymi źródłami napięć nieprzekraczających 5V. Napięcie podajemy na nóżkę ADC0 (PA0). Wynik wygodnie będzie prezentować na LCD. Jeśli podłączymy go do portu A, musimy zmodyfikować plik lcd.h tak, aby biblioteka LCD nie korzystała z pinu PA0. Można np. obsługę sygnałów E, RW i RS skonfigurować na pinach PA1, PA2 i PA3. Inna możliwość to skonfigurowanie LCD na porcie D. Da się też postąpić odwrotnie, czyli nie zmieniać konfiguracji LCD, a zmienić konfigurację ADC. W tym celu za pomocą rejestru ADMUX trzeba przepiąć wejście ADC na inną wolną nóżkę portu A, np. PA3. Załóżmy, że wybrana została pierwsza opcja, czyli modyfikacja biblioteki LCD do korzystania z portu D. Pozwoli nam to później przetestować tryb różnicowy, który wymaga dwóch wejść ADCn. Przykładowy kod do testów ADC będzie wyglądał jak poniżej na **listingu 4.**

```
#include <avr/io.h>
#include <util/delay.h>
#include "lcd.h"
int main(void) {
 ADCSRA |= _BV(ADEN); //włącz ADC
 ADMUX |= _BV(REFS0); //AVCC
 //preskaler 128
ADCSRA |= _BV(ADPS0)|_BV(ADPS1)|_BV(ADPS2);
 lcdInit();
 lcdInitPrintf();
 while(1) {
  ADCSRA |= _BV(ADSC);
  while(ADCSRA & _BV(ADSC));
  lcdGotoXY(0, 0);
  printf("ADC=%04d", ADCW);
  _delay_ms(200);
 }  }
```

Konfiguracja i włączenie ADC następuje w sposób umówiony wyżej. Następnie inicjalizowany jest wyświetlacz oraz funkcja printf(). W głównej pętli uruchamiana jest konwersja A/C poprzez ustawienie bitu ADSC, a następnie program czeka na wyzerowanie tego bitu przez mikrokontroler, co sygnalizuje koniec konwersji. Gotowy wynik przesyłany jest do LCD. Korzystamy tutaj z ciągu %04d, który mówi o tym, że wyświetlona ma być liczba dziesiętna, czterocyfrowa, z zerami wiodącymi. Jeśli użylibyśmy samego %d, wówczas przy spadku napięcia wejściowego a co za tym idzie coraz mniejszymi liczbami otrzymywanymi z ADC, małe wartości nie kasowałyby w pełni wartości poprzedniej. Przykładowo jeśli wynikiem konwersji była liczba 100, a następnie napięcie spadło i konwersja dała liczbę 90, to na wyświetlaczu widzielibyśmy 900, bo na pozycji trzeciej pozostałoby osierocone zero. Co prawda problem można rozwiązać, wstawiając spacje po literce d, ale byłoby to dosyć nieeleganckie i niewygodne w przypadku, gdybyśmy chcieli jeszcze coś wyświetlić na prawo od wartości ADC.

Po uruchomieniu programu z listingu 4 kręcąc potencjometrem, powinniśmy być w stanie uzyskać pełny zakres ADC, czyli liczby od 0 do 1023. Warto też odłączyć napięcie od pinu PA0 i zobaczyć, jakie będą wtedy wartości zwracane przez ADC oraz jakie będą przy dotknięciu palcem. Wejście ADC ma wysoką rezystancję i łatwo zbiera różne zakłócenia. Nawet jeśli pin połączony jest z potencjometrem, wynik na wyświetlaczu ma tendencję do migotania na ostatniej cyfrze. Warto o tym pamiętać podczas pomiarów sygnałów analogowych, szczególnie ze źródeł o wysokiej rezystancji. Pojawia się tu również kwestia rozdzielczości naszego przetwornika. 10 bitów może się wydać zbyt mało, w końcu są przetworniki np. 16- czy 24-bitowe. Jednak przy zaszumionym sygnale dodatkowe bity niewiele wniosą.

Woltomierz

Nasz testowy kod możemy łatwo przerobić tak, aby pokazywał napięcie. Jak wiemy, przetwornik A/C w naszym mikrokontrolerze ma 10 bitów rozdzielczości i skonfigurowaliśmy go na zakres 5V (napięciem odniesienia jest AVCC). Wartość zwracana przez ADC zmienia się więc o 1 najmniej znaczący bit, gdy napięcie wejściowe zmieni się o 5V/1024. Aby uzyskać woltomierz musimy zatem wartość z ADC pomnożyć przez 5 i podzielić przez 1024. Dopiszmy więc w głównej pętli dwie linijki:

```
lcdGotoXY(0, 1);
printf("U=%fV", ADCW * 5.0 / 1024);
```

Za pomocą lcdGotoXY() przechodzimy do dolnej linii wyświetlacza. Ponieważ chcemy wyświetlić liczbę ułamkową zmiennoprzecinkową, jako specyfikator została użyta literka f. Przy mnożeniu wartości z rejestru ADCW liczba 5 została zapisana jako 5.0, aby wynikiem obliczeń była liczba zmiennoprzecinkowa (typ double), a nie całkowita (typ int).

10-bitowa rozdzielczość w przypadku woltomierza o zakresie 5V oznacza rozdzielczość ok. 5 mV. Nie ma więc sensu wyświetlać więcej jak 3 cyfry po przecinku. Lepiej będzie funkcję printf() wywołać jak poniżej:

```
printf("U=%.3fV", ADCW * 5.0 / 1024);
```

Czytelnicy mogą zapytać, dlaczego maksymalne napięcie wskazywane przez nasz woltomierz to 4,995V a nie równe 5V. Jak zostało wspomniane na początku, maksymalna wartość zwracana przez przetwornik oznacza osiągnięcie napięcia odniesienia minus wartość przypadająca na najmniej znaczący bit. Stąd wartość 1023 oznacza napięcie 5V – (5V/1024), czyli ok. 4,995V. Napięciu 5V odpowiadałaby wartość 1024, ale ona wymaga już 11 bitów. Dla wartości przekraczających zakres zwracana jest wartość maksymalna (1023).

Tabela 3

-	-	-	-	-	-	ADC9	ADC8	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADC1	ADC0

Tworząc projekt w Atmel Studio, nie można zapomnieć o dopisaniu parametrów linkera z uwagi na wykorzystanie typu zmiennoprzecinkowego jako parametru funkcji printf(). Parametry dopisujemy zgodnie z opisem w poprzednich lekcjach. Nie można też zapomnieć o makrze F_CPU.

Wyrównanie bitów

Pobieranie wartości zwracanej przez przetwornik A/C jest bardzo proste, wystarczy skorzystać z makra ADCW. Jednak czasem przydać się może głębsza wiedza na temat odczytu wartości z ADC. Po pierwsze nasz przetwornik jest 10-bitowy, a odczytujemy 16 bitów. Na których więc bitach znajduje się wartość będąca wynikiem konwersji A/C? Ponieważ nic szczególnego nie robiliśmy z tą wartością, możemy spodziewać się, że ADC zwraca wartość na młodszych bitach. Czyli z 16 bitów 6 najstarszych jest nieistotna, a znaczące jest 10 młodszych (**tabela 3**). Patrząc na rozmieszczenie bitów znaczących, widzimy, że znajdują się one jakby z prawej strony 16-bitowej wartości. Mówimy w związku z tym, że bity są wyrównane do prawej. Pamiętać należy, że ADCW jest makrem, które powoduje, że z poziomu języka C widoczny jest wirtualny 16-bitowy rejestr. Mikrokontroler ATmega32 jest jednak 8-bitowy i ma 8-bitowe rejestry. Te zwracające wartość z przetwornika A/C to ADCH i ADCL. ADCH zawiera dwa najbardziej znaczące bity (ADC9 i ADC8), natomiast ADCL zawiera pozostałe bity.

Dlaczego rozłożenie bitów może być dla nas istotne? I tak przecież mamy do dyspozycji gotowe makro ADCW i nie musimy odczytywać oddzielnie ADCH i ADCL. Jak wspomniano na początku, konwersję A/C można przyspieszyć kosztem dokładności. Przy taktowaniu przetwornika zegarem 200 kHz dostępna jest dokładność rzędu 9–10 bitów i spada ona do 8 bitów w okolicach 1MHz. Przy 2MHz dokładność wynosi 7 bitów, a przy 3MHz 6 bitów. Jeśli więc potrzebowalibyśmy szybkiego próbkowania i wystarczy nam dokładność 8 bitów lub też z innego powodu 8-bitowa dokładność jest wystarczająca, to nie potrzebujemy odczytywać całej wartości z ADC, a jedynie te 8 bitów. Przy czym chodzi nam o 8 najstarszych, najbardziej znaczących bitów. Będą to więc 2 bity z ADCH i 6 bitów z ADCL. Wychodzi na to, że potrzebując odczytać 8 bitów musimy sięgnąć do dwóch 8-bitowych rejestrów. Co prawda makro

ADCW zrobi to za nas automatycznie, ale taki dostęp jest nieoptymalny. A gdyby tak wartość z ADC była wyrównana do lewej? Wtedy 8 najstarszych, najbardziej znaczących bitów byłoby w ADCH, a pozostałe 2 w ADCL. Ponieważ na tych 2 najmniej znaczących bitach nam nie zależy, możemy odczytać tylko ADCH i zignorować ADCL. Upraszcza to znacznie przetwarzanie danych, co w niektórych zastosowaniach może mieć duże znaczenie. Wyrównanie do lewej włączamy, ustawiając bit ADLAR w rejestrze ADMUX.

Jeśli odwoływalibyśmy się kiedyś do rejestrów ADCL i ADCH bezpośrednio, bez używania makra ADCW, musimy pamiętać o jednej rzeczy. Odczytanie ADCL powoduje niejako „zamrożenie" wyniku konwersji do momentu, aż odczytany zostanie rejestr ADCH. Dzięki temu, jeśli przetwornik działa w trybie autonomicznym, nie ma obaw, że po odczytaniu części wyniku konwersji wartość zmieni się, zanim odczytamy drugą część. W związku tym trzeba pamiętać, aby najpierw odczytywać ADCL, a potem ADCH. Jeśli zastosowane jest wyrównanie do lewej i potrzebujemy tylko 8 starszych bitów, wówczas można odczytywać tylko ADCH.

Odczyt 8-bitowy możemy wykorzystać przy wyborze wewnętrznego źródła napięcia odniesienia o wartości 2,56V. Wówczas wynik pomiaru będzie wyrażony w dziesiątkach miliwoltów, np. liczba 100 będzie oznaczała napięcie 1V. Oczywiście dotyczy to odczytu rejestru ADCH i traktowania go jako wartość 1-bajtową. Jeśli skorzystamy z ADCW, wynik będzie na starszych 8 bitach z 16 i będzie musiał być przesunięty w prawo o 8 miejsc.

Pomiar różnicowy i zmiana wzmocnienia

Przetwornik A/C wbudowany w mikrokontroler ATmega32 potrafi nie tylko mierzyć napięcia względem masy, ale też napięcia między wybranymi wejściami ADC0..7. Za konfigurację tej funkcji odpowiedzialny jest znany nam już rejestr ADMUX. Jeśli jego bity MUX4 i MUX3 są wyzerowane, wówczas podobnie jak przy komparatorze analogowym bity MUX2..0 pozwalają wybrać pin wejściowy spośród pinów ADC0–ADC7 (PA0–PA7). Przetwornik mierzy wtedy napięcie między wybranym pinem a masą. Dodatkowo są dwa specjalnie ustawienia pozwalające jako sygnał wejściowy wybrać napięcie odniesienia V_{BG} lub masę (**tabela 4**). Napięcie V_{BG} (bandgap voltage) ma wartość 1,22V i na jego podstawie generowane jest napięcie 2,56V. Jeśli jednak jeden lub oba bity MUX4 i MUX3 są ustawione na 1, wówczas przetwornik mierzy napięcie między dwoma wybranymi pinami, jednym traktowanym jako dodatni, a drugim jako ujemny. Dodatkowo, jeśli MUX4 jest wyzerowany a MUX3 ustawiony, wówczas mierzone napięcie jest dodatkowo wzmacniane zgodnie z określonym mnożnikiem. Ilustruje to **tabela 5**.

Dokładniejsza analiza tabeli 5 może zaskakiwać, ponieważ wynika z niej, że można ustawić ten sam pin jako wejście dodatnie i ujemne. Jaki sens może mieć pomiar napięcia między tym samym pinem? Czy nie da to zawsze zera? Otóż niedoskonałości przetwornika wprowadzają pewien offset pomiędzy jego wejściami. Podłączając oba wejścia do tego samego pinu, możemy ten offset zmierzyć a następnie uwzględnić, przeprowadzając już normalne pomiary.

Jak dokładnie działa pomiar różnicowy? Jak wiemy, jeden z pinów jest konfigurowany jako dodatni a drugi jako ujemny. Jeśli napięcie na pinie dodatnim będzie większe niż na pinie ujemnym, wynik będzie dodatni, a w przeciwnym wypadku ujemny. W normalnym trybie wynik konwersji był nieujemny i miał wartość od 0 do 1023. W przypadku pomiaru różnicowego wynik konwersji A/C jest liczbą z przedziału od –512 do +511. Zakres pomiaru jest więc niejako podzielony na pół, aby mógł obejmować też wartości ujemne.

Zapis liczb ujemnych

W trybie różnicowym wartość zwracana przez ADC jest reprezentowana nieco inaczej niż w trybie z pojedynczym wejściem. W tamtym przypadku z makra ADCW otrzymywaliśmy 16-bitową wartość, z czego wynik pomiaru był na 10 młodszych bitach. Zwracane wartości były więc z przedziału od 0000000000000000b do 0000001111111111b. Gdy mamy do czynienia z liczbami ujemnymi, musimy jakoś przechować informację o znaku. W trybie różnicowym wykorzystywany jest do tego bit dziesiąty (o indeksie 9 licząc od zera). Stąd sama liczba jest reprezentowana na 9 bitach i zakres wynosi właśnie –512...+512 a nie –1024...+1023. Gdy wspomniany 10. bit jest ustawiony na 1, liczba jest traktowana jako ujemna, a jeśli na 0, jako dodatnia. Uwaga! Nie zmieniamy liczby dodatniej na ujemną i odwrotnie przez proste przestawianie tego bitu. Stosowany sposób zapisu liczb to tzw. kod uzupełnień do dwóch (ang. two's complement), w skrócie U2. W kodzie tym liczbę ujemną zapisujemy jako wynik odejmowania od potęgi dwójki zależnej od tego, ilu bitowa jest liczba. Załóżmy, że liczba jest 8-bitowa, czyli przechowywana w jednym bajcie pamięci. Odpowiednią potęgą będzie więc $2^8 = 256$. Jeśli chcemy przykładowo zapisać liczbę –1 w kodzie U2, to od 256 odejmujemy 1 i otrzymujemy 255. Szesnastkowo będzie to FFh, a binarnie 11111111b. Z kolei –100 da nam 156, czyli szesnastkowo 9Ch, dwójkowo 10011100b, a liczba –128d to 80h i 10000000b.

Taki sposób zapisu może nam się wydać dziwny i nieintuicyjny, ale warto go poznać, bo to jest właśnie najpopularniejsza forma prezentacji liczb ujemnych w informatyce, także w naszym mikrokontrolerze. Czym więc zyskała sobie taką popularność? Możemy domyślić się, że jeśli coś jest dla nas nieintuicyjne, to zapewne jest proste dla komputera. Tak właśnie jest w tym przypadku. Otóż liczby zapisywane w formie U2 można dodawać i odejmować za pomocą tych samych jednostek arytmetyczno-logicznych, co zwykłe liczby dodatnie bez znaku. Procesory dzięki temu mają znacznie prostszą konstrukcję. Polecam sprawdzić to wykonując operacje na liczbach U2 na kartce. Dodajmy np. –1 i 1, traktując te liczby i wynik jako zmienne 1-bajtowe. 11111111b + 00000001b = 100000000b. Wynik ma się zmieścić w 8 bitach, więc tę jedynkę z przodu obcinamy i dostajemy bajt o wartości zero, czyli poprawny wynik.

Tabela 4

MUX4..0	Pin wejściowy
00000	ADC0
00001	ADC1
00010	ADC2
00011	ADC3
00100	ADC4
00101	ADC5
00110	ADC6
00111	ADC7
11110	1,22V (V_{BG})
11111	0V (GND)

Wiemy już, że ADC w trybie różnicowym może nam zwrócić liczbę ujemną zapisaną w formacie U2. Liczby ujemne w mikrokontrolerze też są reprezentowane w kodzie U2, jednak są one 8-, 16- lub 32-bitowe, zależnie od tego, czy skorzystamy z typu int8_t, int16_t, czy int32_t. ADC zwraca nam jednak liczbę 10-bitową i nie możemy jej tak po prostu podstawić do zmiennej czy przekazać do funkcji. Dla liczby ujemnej bit znaku musi być bowiem najstarszym bitem, a nie mamy 10-bitowego typu danych w języku C. Trzeba by więc ten bit jakoś przesunąć. Jeśli wartość z ADC mamy w zmiennej 16-bitowej (typu int16_t), to możemy sprawdzić, czy bit dziesiąty jest

Tabela 5

MUX4..0	Pin wejściowy +	Pin wejściowy -	Wzmocnienie
01000	ADC0	ADC0	10x
01001	ADC1	ADC0	
01010	ADC0	ADC0	200x
01011	ADC1	ADC0	
01100	ADC2	ADC2	10x
01101	ADC3	ADC2	
01110	ADC2	ADC2	200x
01111	ADC3	ADC2	
10000	ADC0	ADC1	1x
10001	ADC1	ADC1	
10010	ADC2	ADC1	
10011	ADC3	ADC1	
10100	ADC4	ADC1	
10101	ADC5	ADC1	
10110	ADC6	ADC1	
10111	ADC7	ADC1	
11000	ADC0	ADC2	
11001	ADC1	ADC2	
11010	ADC2	ADC2	
11011	ADC3	ADC2	
11100	ADC4	ADC2	
11101	ADC5	ADC2	

TOP WWW SKLEP.SEMICON.COM.PL
- podzespoły elektroniczne - szablony laserowe
- montaż SMT - diody laserowe
- materiały dla elektroniki - taśmy i kleje 3M

ustawiony. Jeśli tak, to ustawiamy też bity od szesnastego do jedenastego (od 15 do 10, jeśli liczymy od zera). W ten sposób z 16 bitów 7 starszych będzie ustawionych na 1 a 9 młodszych będzie przechowywać wartość.

Aby przetestować tryb różnicowy, włączmy go dla pinów ADC0 i ADC1 (PA0 i PA1). W tym celu do kodu inicjalizującego ADC dopisujemy linijkę:

```
ADMUX |= _BV(MUX4); //ADC0 i ADC1
```

Do PA0 mamy podłączony potencjometr. Do PA1 podłączmy jakieś źródło napięcia około 1–4V aby móc potencjometrem swobodnie ustawić napięcie niższe i wyższe. Może więc to być np. paluszek 1,5V, bateria 3,7V z telefonu lub zasilacz regulowany. Gdy na potencjometrze ustawimy napięcie wyższe, na LCD zobaczymy napięcie różnicowe. Jeśli jednak ustawimy napięcie niższe, wyświetli się duża wartość. Jeśli od napięcia dodatniego będziemy schodzić potencjometrem powoli w dół, przekraczając zero, zobaczymy wartość 1023 przeliczaną na napięcie 4,995V. Tymczasem chcielibyśmy zobaczyć wartość ujemną. Musimy więc przekonwertować naszą wartość U2 z liczby 10-bitowej na 16-bitową. Jak wspomniano, można to zrobić, przesuwając bit znaku. Zmodyfikujmy więc główną pętlę w następujący sposób wg **listingu 5**:

```
while(1) {
 ADCSRA |= _BV(ADSC);
 while(ADCSRA & _BV(ADSC));
 lcdGotoXY(0, 0);
 int16_t adc = ADCW;
 if (adc & 0b0000001000000000) {
  adc |= 0b1111110000000000;
 }
 printf("ADC=%03d", ADCW);
 lcdGotoXY(0, 1);
 printf("U=% .3fV", adc * 5.0 / 512);
 _delay_ms(200);
}
```

Wprowadzamy tutaj dodatkową zmienną o typie int16_t, czyli 16-bitową ze znakiem, mogącą przechowywać liczby ujemne. Podstawiamy do niej wartość z ADC. Jeśli ustawiony jest bit dziesiąty (9 licząc od zera), to ustawiamy też starsze od niego bity. W ten sposób otrzymujemy prawidłową 16-bitową wartość U2. Spacja po znaku % w wywołaniu funkcji printf() powoduje wstawienie spacji przed liczbami nieujemnymi. W ten sposób zajmą one tyle samo miejsca co liczby ujemne, poprzedzane minusem i unikniemy problemów powstających, gdy po dłuższym ciągu znaków wyświetlamy w tym samym miejscu ciąg krótszy. Aby otrzymać wynik w woltach, w trybie różnicowym jako dzielnik stosujemy 512 zamiast 1024.

Tabela 6

ADTS2	ADTS1	ADTS0	Źródło wyzwalania
0	0	0	tryb samobieżny
0	0	1	komparator analogowy
0	1	0	przerwanie zewnętrzne EXT0
0	1	1	Timer0 Compare Match
1	0	0	Timer0 przepełnienie
1	0	1	Timer1 Compare Match B
1	1	0	Timer1 przepełnienie
1	1	1	Timer1 Input Capture

Nasz kod działa zgodnie z założeniami. Czy możemy jeszcze coś zrobić w kwestii liczb ujemnych? Tym, co nam było potrzebne, to bit znaku znajdujący się na najstarszej pozycji. Czy przypadkiem nie mieliśmy tego efektu, stosując wyrównanie bitów do lewej? Tak, ale wówczas także bity przechowujące wartość są przesunięte. Nie jest to jednak duży problem, bo możemy je przesunąć na właściwe miejsce. Kompletny kod znajduje się poniżej, na **listingu** 6.

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

int main(void) {
 ADCSRA |= _BV(ADEN); //włącz ADC
 ADMUX |= _BV(REFS0); //AVCC
 //preskaler 128
  ADCSRA |= _BV(ADPS0)|_BV(ADPS1)|_BV(ADPS2);
 ADMUX |= _BV(MUX4); //ADC0 i ADC1
 ADMUX |= _BV(ADLAR); //wyrównanie do lewej
 lcdInit();
 lcdInitPrintf();
 while(1) {
  ADCSRA |= _BV(ADSC);
  while(ADCSRA & _BV(ADSC));
  lcdGotoXY(0, 0);
  int16_t adc = (int16_t)ADCW >> 6;
  printf("ADC=%03d ", adc);
  lcdGotoXY(0, 1);
  printf("U=% .3fV", adc * 5.0 / 512);
  _delay_ms(200);
 }
}
```

Konwersja liczby nam się uprościła. Nie ma już instrukcji if, jest tylko przesunięcie bitowe przesuwające wartość zwróconą przed ADCW o sześć miejsc w prawo. Wartość zwracana przez makro ADCW jest rzutowana na typ int16_t, aby była ona traktowana jako wartość ze znakiem. Dzięki temu przy przesunięciu bitowym bit znaku będzie prawidłowo zachowany, a sama wartość będzie prawidłowo wyświetlana i przeliczana na wolty.

Przetwornik samobieżny

W dotychczasowych przykładach uruchamialiśmy proces konwersji analogowo/cyfrowej niejako ręcznie, przez ciągłe ustawianie bitu ADSC w głównej pętli programu. Konwersja może jednak być też uruchamiana automatycznie w zależności od różnych zdarzeń. Automatyczną pracę przetwornika włączamy, ustawiając bit ADATE w rejestrze ADCSRA. Wówczas możemy wybrać źródło wyzwalania konwersji za pomocą bitów ADTS2..0 w rejestrze SFIOR, zgodnie z **tabelą 6**.

W trybie samobieżnym (free running) konwersja odbywa się cały czas. Gdy tylko jedna się zakończy, zaraz zaczyna się druga. W ten sposób możemy w każdej chwili odczytać aktualny wynik konwersji. W pozostałych przypadkach musi nastąpić określone zdarzenie pochodzące z peryferii takich jak komparator analogowy, przerwanie zewnętrzne lub timer. Są to bardzo przydatne funkcje w sytuacji, gdy chcemy znać wartość napięcia w określonym momencie. Szczególnym przypadkiem jest tutaj wyzwalanie konwersji timerem, dzięki któremu możemy próbkować sygnał analogowy w równomiernych odstępach czasu. Typowym przykładem jest nagrywanie i przetwarzanie dźwięku.

Jeśli przetwornik A/C będzie wyzwalany przez różne zdarzenia, przydatne jest wtedy takie skonfigurowanie przerwania, aby móc zareagować na koniec konwersji i dostępność nowej próbki sygnału analogowego. Przerwanie dla ADC włączamy, ustawiając bit ADIE w rejestrze ADCSRA. Przerwanie będzie mogło być obsłużone za pomocą funkcji:

```
ISR(ADC_vect) {
 //obsługa przerwania
}
```

Zadania

Przetwornik analogowo-cyfrowy jest bardzo przydatnym peryferium mikrokontrolera. Pozwala realizować różne funkcje, od najprostszych, jak sprawdzanie stanu baterii, do bardziej zaawansowanych, jak przetwarzanie dźwięku. Warto więc poeksperymentować z ADC i poznać jego specyfikę. Oprócz własnych eksperymentów proponuję dwa ćwiczenia.

1. Jeszcze raz napisać wyłącznik zmierzchowy, z wykorzystaniem ADC. Tym razem progi przełączania można ustawić nie potencjometrami, ale w postaci liczb, np. z klawiatury.
2. Miernik pojemności akumulatorków z monitorowaniem napięcia i obliczaniem liczby mAh. W jaki sposób można wykonać część analogową?

Szczegółowe informacje na temat przetwornika analogowo-cyfrowego można znaleźć w dokumentacji mikrokontrolera ATmega32 oraz w dokumencie AVR127: Understanding ADC Parameters.

W materiałach dodatkowych znajduje się projekt woltomierza oraz pozostałe listingi.

Grzegorz Niemirowski
grzegorz@grzegorz.net