Kurs AVR – lekcja 8

Rozwiązania zadań z ostatniego odcinka

Tematem pierwszego zadania było dodanie walidacji wpisywanego czasu. Można to zrobić na kilka sposobów. Np. w samej funkcji readTime() można dodać kod, który nie pozwoli przesunąć kursora dalej, jeśli wpisane dotychczas cyfry tworzą nieprawidłową liczbę godzin, minut lub sekund. Drugi sposób to zmienić kod wykrywający „przekręcenie się" liczników poszczególnych składowych czasu. W naszym kodzie sprawdzaliśmy, czy liczniki sekund i minut osiągnęły wartość 60 oraz czy licznik godzin osiągnął 24. Ponieważ nieprawidłowe wartości będą się w praktyce sprowadzały do liczb zbyt dużych dla poprawnego czasu, można zmienić operator równości na „większe bądź równe":

```
if (seconds >= 60) {
 seconds = 0;   minutes++;  }
if (minutes >= 60) {
 minutes = 0;   hours++;  }
if (hours >= 24) {
 hours = 0; }
```

W ten sposób wpisanie godziny 33:33:33 spowoduje ustawienie godziny 00:33:33, a wpisanie godziny 55:66:77 da godzinę 00:00:00. Inna opcja to zażądanie od użytkownika wprowadzenia godziny jeszcze raz (**listing 1**).

Wykorzystujemy tu pętlę do-while. Charakteryzuje się ona tym, że zawsze wykonuje się przynajmniej raz, ponieważ warunek jej wykonywania sprawdzany jest dopiero po pierwszej iteracji. Jeśli któraś ze zmiennych sekund, minut lub godzin będzie miała wpisaną przez użytkownika zbyt dużą wartość, pętla nie zakończy się, tylko rozpocznie kolejną iterację i użytkownik zostanie poproszony o wpisanie czasu jeszcze raz.

W zadaniu drugim trzeba było dać możliwość przestawiania zegara. Musi więc istnieć możliwość poinformowania programu o chęci zmiany czasu. W tym celu program może np. sprawdzać stan klawiatury i w przypadku wciśnięcia odpowiedniego klawisza przejść do ponownego wczytywania czasu. Nasz program przez zdecydowaną większość czasu czeka na pojawienie się flagi OCF1A oznaczającej upłynięcie kolejnej sekundy. Najlepiej więc sprawdzanie stanu klawiatury umieścić w pętli while() czekającej na tę właśnie flagę. Pętla ta obecnie oprócz czekania na flagę nic nie robi i ma postać pojedynczej linijki. Możemy ją rozbudować np. jak w **listingu 2**.

Teraz pętla nie tylko sprawdza flagę, ale też odczytuje stan klawiatury. Jeśli jakiś przycisk został wciśnięty, sprawdzany jest jego numer. Jeśli był to przykładowo przycisk S14, wykonywana jest opisana już wyżej pętla do-while.

Zadanie trzecie polegało na dodaniu funkcji budzika. Tutaj można postąpić analogicznie i w pętli sprawdzającej flagę OCF1A dodać sprawdzenie, czy nie został wciśnięty klawisz np. S13. Poniżej szkielet kodu, bez wczytywania czasu:

```
while(!(TIFR & _BV(OCF1A))) {
 if (readKeyboard()) {
  uint8_t key = getKey();
  if (key == 14) {
   // wczytanie nowego czasu     }
  if (key == 13) {
   // wczytanie czasu alarmu     }
 }
}
```

Gdy klawisz ten zostanie wciśnięty, trzeba wczytać czas, ale nie umieszczać go w zmiennych zegara, tylko w dodatkowych zmiennych przechowujących czas budzika. Następnie, np. obok kodu sprawdzającego przepełnienie zmiennych czasu, konieczne jest dodane porównania zmiennych czasu alarmu ze zmiennymi bieżącego czasu. Gdy będą równe, trzeba jakoś zasygnalizować alarm. Może to być migający LED, piezo z generatorem, bez generatora lub coś innego. W przypadku piezo bez generatora można wykorzystać Timer0 i funkcję Output Compare do sterowania pinem. Opis był w lekcji 4, przy omówieniu rozwiązania zadania 1 z lekcji 3.

Nic nie stoi na przeszkodzie, aby samodzielnie dodać różne inne funkcje, np. kalendarz oraz pomyśleć nad różnymi dodatkami umilającymi pracę z programem, np. ikonką dla alarmu czy komunikatem w przypadku wpisania nieprawidłowego czasu.

```
while(!(TIFR & _BV(OCF1A))) {
 if (readKeyboard()) {
  if (getKey() == 14) {
   do {
    lcdGotoXY(0, 0);
    printf("  :  :  ");
    lcdGotoXY(0, 0);
    readTime(&hours, &minutes, &seconds);
   } while (seconds > 59 || minutes > 59 || hours > 23);
  }
 }
}
```

Listing 2

```
lcdWriteCommand(LCD_COMMAND_ON_OFF | LCD_PARAM_ON_OFF_DISPLAY | LCD_PARAM_ON_OFF_CURSOR);
uint8_t hours, minutes, seconds;
do {
 lcdGotoXY(0, 0);
 printf("  :  :  ");
 lcdGotoXY(0, 0);
 readTime(&hours, &minutes, &seconds);
} while (seconds > 59 || minutes > 59 || hours > 23);
lcdWriteCommand(LCD_COMMAND_ON_OFF | LCD_PARAM_ON_OFF_DISPLAY);
```

Listing 1

Przerwania

W tej lekcji omówimy przerwania, jedną z najważniejszych funkcji, jakie mają mikrokontrolery. Zanim jednak powiemy sobie czym są, podsumujmy sposób, w jaki dotychczas obsługiwaliśmy zdarzenia, takie jak naciśnięcie przycisku albo przepełnienie się licznika timera.

Wystąpienie zdarzenia stwierdzaliśmy, odczytując stan odpowiednich bitów w rejestrach takich jak PINA czy TIFR. Odczyt ten był wykonywany przez ciągłe odczytywanie tych rejestrów w oczekiwaniu na zmianę określonych bitów. Taka obsługa zdarzeń nie jest wygodna. Sprawdzenie wystąpienia zdarzenia musi być wykonywane pomiędzy innymi zadaniami realizowanymi przez program. Jeśli jakieś zadanie trwa dłuższy czas, zdarzenie zostanie odnotowane z opóźnieniem lub całkiem przegapione.

Przykładem może być proste miganie diodą z pierwszej lekcji. Jeśli chcielibyśmy podczas migania sprawdzać stan przycisku, to w przypadku, gdy przycisk został wciśnięty i puszczony w trakcie wykonywania się funkcji _delay_ms(), nie zostanie to w ogóle zauważone przez program. Możemy to rozwiązać w różny sposób. Np. zmniejszyć okres opóźnienia, a samo opóźnienie wywoływać wielokrotnie dla danego półokresu. W ten sposób dioda będzie migać z tą samą częstotliwością, ale stan przycisku będzie sprawdzany częściej. Można też zrezygnować z _delay_ms() i zastosować timer. Wówczas sprawdzając flagę w rejestrze TIFR, można też sprawdzać stan przycisku. Tak zrobiliśmy, rozwiązując zadanie 1 z lekcji 6 (rozwiązanie opublikowane w lekcji 7). W ten sposób rozwiązaliśmy problem z przegapianiem zdarzeń, ale kod stał się bardziej zagmatwany. Ponadto czym więcej zdarzeń, które program monitoruje, tym trudniej oddzielić obsługę jednych zdarzeń od drugich. Przydałoby się, żeby zdarzenia można było obsługiwać niezależnie.

Tutaj z pomocą przychodzą przerwania. W momencie wystąpienia określonego zdarzenia mikrokontroler przerywa wykonywanie programu i zaczyna wykonywać specjalną funkcję obsługującą to zdarzenie. Gdy ta funkcja się wykona, mikrokontroler wznawia wykonywanie głównego programu od miejsca, w którym przerwał. Dzieje się to asynchronicznie – funkcja obsługująca zdarzenie/przerwanie może się wykonać w każdej chwili. Trzeba o tym pamiętać, bo ma to swoje plusy i minusy. Z jednej strony zaletą przerwań jest możliwość natychmiastowej reakcji na zdarzenie. Z drugiej jednak programista musi zadbać o to, aby główny program nie był wrażliwy na wystąpienie przerwania w nieoczekiwanym momencie.

Obsługa przerwań

Jak wspomnieliśmy, przerwania obsługiwane są za pomocą specjalnych funkcji. Aby stworzyć funkcję obsługującą przerwanie, musimy ją odpowiednio nazwać. Nazwy funkcji dla poszczególnych przerwań zostały zebrane w **tabeli 1**. Dodatkowo funkcja ta musi mieć odpowiednie atrybuty, aby kompilator mógł ją wykorzystać do obsługi przerwań. Atrybuty te nadaje się za pomocą makra ISR(), które wymaga pliku nagłówkowego interrupt.h. Przykładowo funkcja dla przerwania pochodzącego z funkcji Output Compare z kanału A w Timer1 będzie wyglądała następująco:

```
ISR(TIMER1_COMPA_vect) {
 //ciało funkcji    }
```

Funkcja ta będzie się wykonywała za każdym razem, gdy w Timer1 włączy się mechanizm Output Compare, czyli gdy licznik timera osiągnie wartość zapisaną w rejestrze OCR1A. Jeśli skonfigurujemy timer tak, aby Output Compare następowało co sekundę, nasza funkcja obsługi przerwania będzie uruchamiana właśnie co sekundę. W ten sposób będziemy mogli wykonywać określoną czynność w odstępach sekundowych niezależnie od tego, co będzie robił główny program.

Stworzenie samej funkcji obsługującej dane przerwanie to jednak za mało, abyśmy mogli z niego korzystać. Generowanie przerwań przez mikrokontroler trzeba najpierw włączyć, i to na dwóch poziomach. Po pierwsze trzeba przerwania włączyć globalnie funkcją sei(). Inaczej przerwania nie będą się w ogóle wykonywać. Po drugie trzeba włączyć przerwanie dla danego peryferium, które jest źródłem zdarzeń. W przypadku timerów konfiguruje się to rejestrem TIMSK (**tabela 2**). Popatrzmy na przykład migania diodą za pomocą przerwania – **listing 3**: Na początku zaczynamy od globalnego włączenia przerwań za pomocą sei(). Ponieważ będziemy migać diodą podłączoną do pinu 0 portu B, konfigurujemy go jako wyjście, ustawiając pin DDB0 w rejestrze DDRB. Tak jak w zegarze z lekcji 7, źródłem odmierzającym czas jest Timer1 i jego funkcja Output Compare. Tak samo też konfigurujemy jego rejestr OCR1A, aby uzyskać przerwanie co sekundę. Liczba 15624 wynika z przyjętego wtedy taktowania kwarcem 16 MHz. Oczywiście dla innego źródła taktowania lub innego okresu musimy obliczyć inną wartość dla OCR1A, zgodnie z konfiguracją taktowania opisaną w lekcji 4. Na koniec ustawiamy bit OCIE1A w rejestrze TIMSK, aby Timer1 generował przerwania dla funkcji Output Compare w kanale A. Główna pętla programu jest pusta. Zadanie realizowane przez nasz program, czyli miganie diodą, realizowane jest w funkcji TIMER1_COMPA_vect(). Przełączanie LED-a wykonywane jest w tym przykładzie w najprostszy sposób, przez logiczną negację stanu rejestru portu. Jeśli w rejestrze jest liczba 0, to wpisana zostanie do niego 1, a jeśli wpisane jest 1, zostanie wpisane 0. W ten sposób zmienia się stan tylko jednego, najmłodszego bitu (pinu), reszta jest zerowana. Gdybyśmy użyli nie operatora negacji logicznej (!), a operatora negacji bitowej (~), przestawiane byłyby wszystkie bity. W każdym razie sterowane są wszystkie piny portu B i gdyby główny program zmienił stan jednego z pinów tego portu, to w przerwaniu byłoby to nadpisane. W naszym prostym przykładzie jednak to nie przeszkadza, bo główny program nie przestawia żadnego pinu.

Listing 3

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

int main(void)
{
 sei(); //włącz przerwania globalnie
 DDRB = _BV(DDB0); //pin PORTB0 jako wyjście
 TCCR1B = _BV(WGM12) | _BV(CS12) | _BV(CS10);
 OCR1A = 15624;
 TIMSK = _BV(OCIE1A); //włącz przerwanie dla funkcji output compare timera 1
 while (1) {}
}

ISR(TIMER1_COMPA_vect) {
 PORTB = !PORTB;
}
```

Przekazywanie danych do funkcji obsługi przerwania

Uważni Czytelnicy zauważą, że definicja funkcji wygląda nietypowo, bo po jej nazwie brakuje nawiasów z ewentualnymi parametrami, a te nawiasy są przecież wymagane przez język C. Otóż są one dodawane przez makro ISR, nie ma więc odstępstwa od standardu. Ważne jest to, że są one dodawane w postaci (void), czyli funkcja obsługi przerwania nie przyjmuje żadnych parametrów. Jest ona wywołana asynchronicznie, gdy wykonanie głównego programu odbywa się w bliżej nieokreślonym miejscu. Trudno więc, aby jakieś parametry były przekazane.

W jaki więc sposób funkcja obsługi przerwania może uzależnić swoje działanie od stanu głównego programu? Skoro nie można przekazać parametrów, funkcja musi skorzystać ze zmiennych globalnych lub, jak nasza funkcja migająca, odczytywać stan rejestrów. Jak wiemy, zmiennych globalnych należy unikać, ale tutaj będą właśnie potrzebne.

Nazwa funkcji	Źródło przerwania
INT0_vect	Zewnętrzne przerwanie 0
INT1_vect	Zewnętrzne przerwanie 1
INT2_vect	Zewnętrzne przerwanie 2
TIMER2_COMP_vect	Output Compare w Timer2
TIMER2_OVF_vect	Przepełnienie licznika w Timer2
TIMER1_CAPT_vect	Input Compare w Timer1
TIMER1_COMPA_vect	Output Compare w Timer1 (kanał A)
TIMER1_COMPB_vect	Output Compare w Timer1 (kanał B)
TIMER1_OVF_vect	Przepełnienie licznika w Timer1
TIMER0_COMP_vect	Output Compare w Timer0
TIMER0_OVF_vect	Przepełnienie licznika w Timer0
SPI_STC_vect	Koniec transmisji SPI
USART_RXC_vect	Odebranie bajtu przez port szeregowy
USART_UDRE_vect	Rejestr nadawczy gotowy
USART_TXC_vect	Zakończenie transmisji bajtu przez port szeregowy
ADC_vect	Zakończenie konwersji A/C
EE_RDY_vect	EEPROM gotowy
ANA_COMP_vect	Stan wysoki na wyjściu komparatora analogowego
TWI_vect	Przerwanie I^2C
SPM_RDY_vect	Gotowość Store Program Memory

Tabela 1

Załóżmy, że chcemy migać linijką 8 LED-ów, ale niekoniecznie wszystkimi w danej chwili, tylko wybierać z klawiatury, które diody mają migać. Do przechowywania informacji o tym, który LED ma aktualnie włączone miganie, wystarczy zmienna typu uint8_t. Zadeklarujemy ją jako globalną, czyli jej deklarację umieścimy przed deklaracjami funkcji.

W naszym kodzie będziemy potrzebować jeszcze jednej zmiennej związanej z obsługą migania. Gdy migaliśmy jedną diodą, mogliśmy w danej iteracji sprawdzać obecny stan pinu, do którego podłączony jest LED i ustawiać jego stan na przeciwny. Teraz już nie jest to możliwe, bo sterować będziemy diodami niezależnie i miganie będzie mogło być np. w danej chwili wyłączone dla wszystkich z nich. Potrzebna będzie więc zmienna (ledON), która będzie mówiła, czy w danej iteracji jesteśmy w fazie wyłączenia wszystkich diod, czy też włączenia wybranych z nich.

Ponieważ zmienna ta musi zachowywać swój stan pomiędzy wywołaniami funkcji obsługi przerwania, nie możemy zadeklarować tej zmiennej wewnątrz tejże funkcji jako zwykłej zmiennej automatycznej. Pamięć dla nich jest bowiem rezerwowana tylko na czas wykonywania się funkcji, w których są zadeklarowane i może być później nadpisana innymi danymi. Pasowałaby nam tutaj więc zmienna globalna, która będzie „żyła" przez cały czas działania programu. Byłoby to jednak nieeleganckie, gdyż ideą zmiennych globalnych jest ich dostępność z różnych miejsc kodu, a nie zapewnienie nieograniczonego „czasu życia". Na szczęście mamy do dyspozycji zmienne statyczne. Są one deklarowane wewnątrz funkcji, w której są używane i nie ma problemu z ich niepotrzebną widocznością poza tą funkcją. Jednocześnie umieszczane są nie na stosie, ale w obszarze pamięci zwanym stertą, tak jak zmienne global-

Tabela 2

TICIE1	Input Compare w Timer1
OCIE1A	Output Compare w Timer1 (kanał A)
OCIE1B	Output Compare w Timer1 (kanał B)
TOIE1	Przepełnienie licznika w Timer1
OCIE0	Output Compare w Timer0
TOIE0	Przepełnienie licznika w Timer0

ne. Aby zmienna była statyczna, używamy modyfikatora static. Nasza funkcja obsługi przerwania będzie wyglądała następująco:

```
uint8_t ledConf = 0;

ISR(TIMER1_COMPA_vect) {
 static uint8_t ledON = 0;
 ledON = !ledON;
 if (ledON) {
  PORTB = ledConf;  }
  else {
  PORTB = 0;    }
}
```

Zmienną ledON przestawiamy za każdym wywołaniem w przeciwny stan logiczny. Zależnie od bieżącego stanu włączamy wybrane diody według bitów w zmiennej ledConf lub wyłączamy wszystkie diody. Zmienna statyczna lenON jest inicjalizowana zerem tylko za pierwszym wywołaniem funkcji obsługi przerwania.

Mając gotową obsługę przerwania, przejdźmy do funkcji main(), aby dopisać w niej obsługę klawiatury i wpisywanie odpowiednich wartości do zmiennej globalnej ledConf. Dla klawiatury mamy naszą bibliotekę, możemy więc skopiować pliki keyb.h i keyb.c do katalogu z naszym nowym projektem, obok pliku main.c. Następnie trzeba je dodać do projektu, klikając na nim prawym przyciskiem w okienku Solution Explorer i wybierając z menu Add-> Existing Item… Aby sprawdzić, czy wszystko jest w porządku, wciśnijmy F7, aby skompilować projekt. Niestety pojawi się błąd:

Error lcd.h: No such file or directory.

Jak widać, przyczyną jest brak pliku lcd.h w naszym projekcie. Wynika to z tego, że biblioteka klawiatury odwołuje się do biblioteki LCD, bo w poprzedniej lekcji dopisaliśmy obsługę klawisza Backspace i konieczne było kasowanie znaków na wyświetlaczu oraz przesuwanie kursora z poziomu funkcji wczytującej czas z klawiatury. W ten sposób biblioteka klawiatury została powiązana z biblioteką alfanumerycznego wyświetlacza ciekłokrystalicznego. Dodając obsługę Backspace, sprawiliśmy, że biblioteka stała się mniej uniwersalna. Można powiedzieć, że nie jest to duży problem, bo w każdym projekcie mamy oddzielne pliki keyb.h oraz keyb.c i w związku z tym nic nie stoi na przeszkodzie, aby dostosować je do potrzeb danego projektu. Możemy np. w pliku keyb.c wykasować dyrektywę #include dołączającą plik lcd.h oraz wykasować funkcje readTime() i readNumber(). Wykonajmy więc teraz tę operację, a później zastanowimy się, jak możemy przepisać bibliotekę klawiatury, aby pasowała do różnych projektów. Po skasowaniu wymienionych fragmentów kod powinien się już skompilować. Może ewentualnie pojawić się ostrzeżenie odnośnie do braku definicji makra F_CPU. Zdefiniujmy je, jeśli jeszcze tego nie zrobiliśmy w naszym projekcie. Ponieważ pozostajemy przy taktowaniu częstotliwością 16 MHz, definicja będzie miała postać F_CPU=16000000.

Aby korzystać z biblioteki klawiatury, na początku funkcji main() dodajemy wywołanie funkcji keybInit(). Następnie w głównej pętli będziemy odczytywać klawisz funkcją getKey(). Jak pamiętamy, funkcja ta oczekuje na wciśnięcie klawisza i zwraca jego numer. Ponieważ mamy 8 LED-ów, załóżmy, że będziemy nimi sterować za pomocą klawiszy S1–S8. Naszym celem jest, aby wciśnięcie klawisza włączało lub wyłączało miganie odpowiedniego LED-a. Włączanie/wyłączanie migania ustawiamy przez odpowiedni bit w zmiennej ledConf. Przykładowo aby migały nam diody LED3 i LED5, do zmiennej ledConf musi być wpisana wartość 00010100b, czyli muszą być ustawione bity 2 i 4. Aby te diody przestały migać, te bity muszą zostać wyzerowane. Skoro naciśnięcie klawisza ma włączać/wyłączać miganie danej diody, to na podstawie numeru wciśniętego klawisza musimy przestawić odpowiedni bit na przeciwny.

Omawialiśmy już ustawianie i zerowanie wybranych bitów, ale nie zajmowaliśmy się jeszcze przestawianiem na stan przeciwny. Oczywiście możemy to zrobić na piechotę: sprawdzić stan bitu a następnie go wyzerować lub ustawić. Okazuje się, że można to zrobić znacznie prościej: operacją XOR. Czytelnicy, którzy mieli styczność z podstawami elektroniki cyfrowej, na pewno pamiętają bramki XOR. Działają one w ten sposób, że zwracają 1, jeśli na wejściach mają różne stany, a 0 jeśli takie same. Wynika też z tego, że jeśli na jednym z wejść będzie 1, to na wyjściu będzie stan przeciwny niż na drugim wejściu **(tabelka obok)**.

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

Operacje logiczne w C działają na zmiennych tak, jakby było wiele bramek logicznych, a każda obsługiwała po jednym bicie z obu zmiennych. A więc XOR dwóch zmiennych 1-bajtowych to tak jakby 8 XOR-ów: jeden na bitach 0 jednej i drugiej zmiennej, drugi na bitach 1, trzeci na bitach 2 itd. Załóżmy teraz, że mamy dwie zmienne. W jednej przechowujemy jakąś daną wejściową, a w drugiej wszystkie bity są wyzerowane oprócz jednego. W wyniku operacji XOR na tych zmiennych otrzymamy wartość wyglądającą jak pierwsza zmienna, z wyjątkiem bitu, który był na tej pozycji, co bit ustawiony na 1 w drugiej zmiennej. Innymi słowy, jedynka w drugiej zmiennej przestawi bit w wartości pochodzącej z pierwszej zmiennej na przeciwny (**tabela poniżej**).

A	1	0	1	1	0	1	0	0
B	0	0	1	0	0	0	0	0
A XOR B	1	0	0	1	0	1	0	0

Dzięki operacji XOR możemy w głównej pętli przestawiać bity w zmiennej, znajdujące się na pozycji wskazanej przez numer klawisza:

```
uint8_t key = getKey();
ledConf ^= _BV(key - 1);
```

Wykorzystujemy tutaj znane nam makro wykonujące przesunięcie bitowe i zwracające bajt, który ma jedną jedynkę na wybranym miejscu. Parametrem jest tutaj numer klawisza pomniejszony o jeden, ponieważ bity są numerowane od 0, a klawisze od 1. W języku C operatorem XOR jest symbol ^ (daszek). Ponieważ wynik zapisujemy do tej samej zmiennej (ledConf), wykorzystana jest notacja ^=. W ten sposób w głównym programie czekamy na wciśnięcie klawisza, a gdy ono nastąpi, aktualizujemy odpowiednio bity w zmiennej ledConf. Uruchamiające się cyklicznie przerwanie odczytuje tą zmienną i odpowiednio zaświeca lub gasi LED-y. Na **listingu 4** pokazany jest kompletny program główny, bez funkcji obsługi klawiatury z naszej biblioteki.

Listing 4

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

uint8_t ledConf = 0xff;

int main(void)
{
 sei(); //włącz przerwania globalnie
 keybInit();
 DDRB = 0xff;
 TCCR1B = _BV(WGM12) | _BV(CS12) | _BV(CS10);
 OCR1A = 15625;
 TIMSK = _BV(OCIE1A); //włącz przerwanie dla funkcji output compare timera 1
 while (1)
 {
  uint8_t key = getKey();
  ledConf ^= _BV(key - 1);
 }
}

ISR(TIMER1_COMPA_vect) {
 static uint8_t blinkON = 0;
 blinkON = !blinkON;
 if (blinkON) {
  PORTB = ledConf;
 } else {
  PORTB = 0;
 }
}
```

Niespodzianka ulotna

Jak wspomnieliśmy, przerwania są bardzo pożyteczną rzeczą, ale ze względu na swoje asynchroniczne działanie wymagają większej uwagi od programisty. Bez przerwań kod wykonuje się linijka po linijce i generalnie nie ma tutaj niespodzianek. Jeśli jakaś funkcja się wykonuje, to dlatego, że w kodzie umieściliśmy jej wywołanie. Funkcja obsługi przerwania

wykonuje się jednak, mimo że nigdzie jej bezpośrednio nie wywoływaliśmy. Wywołanie wykonywane jest sprzętowo, gdy zaszło pewne zdarzenie, jest dla niego włączone przerwanie i zdefiniowana funkcja obsługi przerwania. Na tym jednak nie koniec trudności związanych z przerwaniami. Rozważmy prosty przypadek: oczekiwanie na wystąpienie przerwania. Ilustruje to **listing 5**.

Tak jak w poprzednim przykładzie, włączamy przerwania, ustawiamy piny portu B jako wyjścia i konfigurujemy timer. W głównej pętli czekamy, aż zmienna counter będzie różna od zera. Gdy to nastąpi, wpisujemy 1 do rejestru portu B, aby włączyć LED-a. Zmienna counter jest ustawiana na 1 w funkcji obsługującej przerwanie z timera. W ten sposób LED ma się włączyć po sekundzie od uruchomienia programu. Kompilujemy nasz program, programujemy procesor i… nic. Dioda nie świeci. Czyżby przerwanie się nie wyzwoliło? Sprawdźmy. W obsłudze przerwania wpiszmy PORTB = 1; zamiast counter = 1;. Po zmianie faktycznie dioda się włącza po upływie sekundy. Czyżby było coś nie tak ze zmienną counter? Ustawiamy ją na 1, a pętla while() zachowuje się, jakby zmienna ta była ciągle wyzerowana. I tak właśnie jest, problem tkwi w tej zmiennej.

Żeby zobaczyć, co się tak właściwie dzieje, w okienku *Solution Explorer* rozwińmy katalog *Output Files* i otwórzmy plik z rozszerzeniem .lss. W pliku tym mamy pokazane, w jaki sposób kompilator przetłumaczył kod C na asembler. Popatrzmy, jak wygląda nasza pętla while() (**rysunek 1**). Instrukcją lds procesor wczytuje wartość zmiennej counter przechowywanej w pamięci RAM pod adresem 0x0060 do rejestru R24. Robi tak, ponieważ nie potrafi wykonywać operacji arytmetycznych na komórkach pamięci, tylko właśnie na rejestrach. Stąd też konieczne jest skopiowanie wartości z RAM-u do rejestru. Gdy już wartość zmiennej jest w rejestrze, wykonuje na niej operację and, jako drugi argument biorąc tę samą wartość. W ten sposób sprawdza, czy wartość ta jest różna od zera. Następnie

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

uint8_t counter = 0;

int main(void)
{
 sei(); //włącz przerwania globalnie
 DDRB = 0xff;
 TCCR1B = _BV(WGM12) | _BV(CS12) | _BV(CS10);
 OCR1A = 15624;
 TIMSK = _BV(OCIE1A); //włącz przerwanie dla funkcji output compare timera 1
 while (!counter){}
 PORTB = 1;
}

ISR(TIMER1_COMPA_vect) {
 counter = 1;
}
```

Listing 5

mamy instrukcję breq, która jest skokiem warunkowym. Skok wykonywany jest wtedy, gdy w procesorze ustawiona jest flaga Z (zero).

```
while (!counter)
92:   80 91 60 00
96:   88 23
98:   f1 f3
```

```
while (!counter)
92:   80 91 60 00
96:   88 23
98:   e1 f3
```

Flaga ta jest ustawiana przez różne instrukcje arytmetyczne, m.in. and, gdy ich wynikiem jest zero. Jeśli więc wczytana do rejestru wartość była zerem, to flaga Z zostaje ustawiona i następuje skok. Jak widzimy, argumentem jest tu liczba –4, czyli procesor skacze o 4 bajty do tyłu w kodzie programu. Ponieważ wczytana już była instrukcja breq (licznik rozkazów jest już ustawiony na kolejną instrukcję), skok o –4 bajty będzie skokiem do instrukcji and.

I tu jest właśnie przyczyna naszych kłopotów. Procesor ciągle sprawdza instrukcją and stan rejestru R24, nie uwzględnia w ogóle zmiennej w RAM. Wczytuje ją tylko raz. Dlaczego tak się dzieje? Czy to błąd kompilatora? Otóż efekt ten jest wynikiem procesu optymalizacji wykonywanej przez kompilator. Wewnątrz pętli while() kompilator nie napotyka nic, co sugerowałoby konieczność wczytywania zmiennej z pamięci i uniemożliwiałoby posługiwanie się jej kopią przechowywaną w rejestrze R24. Nie uwzględnia faktu, że może nastąpić przerwanie, w którego obsłudze oryginalna zmienna zostanie zmodyfikowana.

W tym miejscu wnikliwi Czytelnicy oglądający asemblerowy kod obsługujący przerwanie zapewne zaprotestują, bo widać że tam również modyfikowany jest rejestr R24. To prawda. Jednak najpierw wartość rejestru R24 jest zachowywana na stosie (instrukcja push), następnie jest on używany do modyfikacji zmiennej (instrukcje ldi i sts), a na koniec jest on odtwarzany ze stosu (instrukcja pop). W ten sposób różne funkcje używające tych samych rejestrów nie przeszkadzają sobie.

Kończąc dygresję, zostajemy z pytaniem, jak poradzić sobie ze skutkami optymalizacji. Otóż rozwiązanie jest bardzo proste: słowo kluczowe volatile. Zmieńmy deklarację zmiennej counter na poniższą:

```
volatile uint8_t counter = 0;
```

Kompilujemy program, programujemy mikrokontroler i okazuje się, że nasz program działa w pełni poprawnie. Za pomocą słowa kluczowego volatile poinformowaliśmy kompilator, że zmienna counter może zmienić swoją wartość w nieprzewidywalnym momencie i w związku z tym musi on w generowanym

Rys. 1

```
lds r24, 0x0060 ; 0x800060 <_edata>
and r24, r24
breq    .-4         ; 0x96 <main+0x1a>
```

Rys. 2

```
lds r24, 0x0060 ; 0x800060 <_edata>
and r24, r24
breq    .-8         ; 0x92 <main+0x16>
```

kodzie zawsze umieszczać bezpośrednie odwołania do niej. Warto w tym momencie ponownie zajrzeć do pliku .lss. Okazuje się, że zmiana jest niewielka: skok wykonywany jest nie o 4, ale o 8 bajtów do tyłu, do instrukcji ładującej zmienną do rejestru (**rysunek 2**). W ten sposób instrukcja and otrzymuje w rejestrze R24 zawsze aktualną wartość zmiennej.

Korzystając z przerwań, musimy więc pamiętać o stosowaniu słowa kluczowego volatile do zmiennych, które są używane i w przerwaniach, i w „zwykłym" kodzie. Jednak optymalizacja może nam przeszkadzać nie tylko w przypadku przerwań. Załóżmy, że chcemy wstawić w kodzie opóźnienie za pomocą pętli:

```
for(uint32_t i = 0; i < 1000000; i++);
```

Niby wszystko jest w porządku. Procesor będzie zwiększał zmienną aż jej wartość osiągnie milion i wykona kolejne instrukcje. Załóżmy jednak, że zależy nam, aby nasz program zajmował jak najmniej pamięci Flash w mikrokontrolerze i we właściwościach projektu ustawimy optymalizację na -Os. Okaże się, że kompilator usuwa tę pętlę, nie umieszcza jej w kodzie wynikowym. Jest to bowiem pusta pętla, zajmująca się jedynie modyfikacją zmiennej nieużywanej nigdzie indziej. Z logicznego punktu widzenia pętla ta jest więc zbędna. Nam jednak zależy na opóźnieniu i tutaj również z pomocą przychodzi volatile:

```
for(volatile uint32_t i = 0; i < 1000000; i++);
```

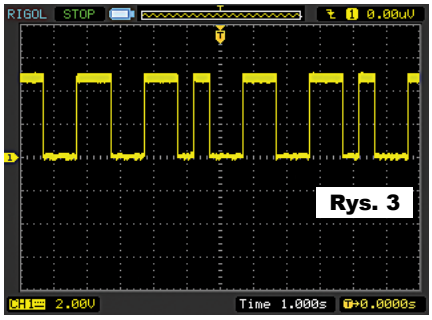
Taka pętla już nie zostanie usunięta w procesie optymalizacji jako niepotrzebna.

Niespodzianka atomowa

Przy pracy z przerwaniami czeka na nas jeszcze jedna niespodzianka. Załóżmy, że chcemy migać diodą z częstotliwością ok. 0,5Hz, korzystając z Timer0 z preskalerem wynoszącym 1024. Przy podziale częstotliwości 16MHz przez 1024 licznik timera będzie się zwiększał 15625 razy na sekundę. Ponieważ chcemy migać dużo wolniej, korzystamy z funkcji CTC, aby uzyskać przerwania z częstotliwością np. 32-krotnie mniejszą, czyli ok. 488Hz. Do rejestru OCR0 wpisujemy liczbę 31, ponieważ po osiągnięciu przez licznik wartości z rejestru OCR0 nie resetuje się on od razu, tylko dopiero w następnym cyklu swojego zegara. Wpisanie liczby 32 spowodowałoby odliczenie 33 cykli.

W funkcji obsługującej przerwanie dekrementujemy zmienną zainicjowaną liczbą wynoszącą tyle, ile wynosi częstotliwość taktowania licznika wyrażona w hercach, czyli 488. W ten sposób wartość zmiennej spadnie do zera w ciągu 1 sekundy. Dwa takie półokresy, dla diody włączonej i zgaszonej, dadzą 2 sekundy całkowitego okresu, czyli miganie z częstotliwością 0,5 Hz. Nasz kod będzie wyglądał jak na **listingu 6**.

Dioda miga, ale jakoś dziwnie. Co 2 –3 okresy zdarza się szybsze mignięcie. Widać to dobrze na oscyloskopie (**rysunek 3**).



Dlaczego tak się dzieje? Czyżby timer zmieniał swoje taktowanie? Odpowiedź znów kryje się w kodzie asemblerowym. Zaglądając do pliku .lss, widzimy, że nasza pętla while(counter) została skompilowana jako dwie instrukcje lds, instrukcja or, oraz instrukcja brne. Nasza zmienna licznikowa jest typu uint16_t, czyli zajmuje 2 bajty. Inaczej nie zmieściłaby się w niej wartość 488. Mikrokontroler, aby sprawdzić jej wartość, ładuje najpierw jej młodszy bajt spod adresu 0x0060 do rejestru R24, a następnie starszy, z adresu 0x0061, do rejestru R25. Dzięki temu może potem instrukcją or sprawdzić, czy oba bajty (połówki zmiennej 16-bitowej) są wyzerowane. Nasz mikrokontroler jest 8-bitowy i nie może wczytać zmiennej 16-bitowej w jednej instrukcji.

Listing 6

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

volatile uint16_t counter = 0;

void delay();

int main(void)
{
 sei();
 DDRB = 0xff;
 TCCR0 = _BV(WGM01) | _BV(CS02) | _BV(CS00);
 OCR0 = 9;
 TIMSK = _BV(OCIE0);
 while(1) {
  delay();
  PORTB = 1;
  delay();
  PORTB = 0;
 }
}

ISR(TIMER0_COMP_vect) {
 counter--;
}

void delay() {
 counter = 488;
 while(counter);
}
```

I tutaj właśnie daje o sobie znać asynchroniczność przerwań. Może się przecież zdarzyć, że przerwanie nastąpi między jedną a drugą instrukcją lds. Nasza zmienna inicjowana jest liczbą 488 i ciągle zmniejszana w przerwaniu. W którymś momencie otrzymuje wartość 256, czyli 0000000100000000b. Popatrzmy, co się teraz dzieje. Mikrokontroler ładuje młodszy bajt do rejestru R24. Jest w nim więc zero. I nagle, jeśli mamy (nie)szczęście, następuje przerwanie. Zmienna counter zostaje zdekrementowana i otrzymuje wartość 255, czyli 0000000011111111b. Przerwanie się kończy, a kod sprawdzający wartość zmiennej wykonuje się dalej. Mikrokontroler ładuje starszy bajt do rejestru R25. W tym starszym bajcie nie ma już jednak wartości 1, tylko 0. Oba rejestry R24 i R25 otrzymują wartość 0, mimo że zmienna counter wcale nie miała wartości 0. Instrukcja or stwierdza, że w obu rejestrach są zera, nie ma skoku instrukcją brne, funkcja delay() kończy się. Zakończenie tej funkcji następuje przedwcześnie, zmienna counter nie osiągnęła wartości 0 i obserwujemy krótsze mignięcie. Patrząc na oscyloskop, widzimy, że zamiast 1 sekundy zostało odmierzonych ok. 480 milisekund. Jeśli zmienna licznikowa nie odlicza do 0, tylko do 256, to zamiast 488 dekrementacji wykonywanych jest tylko 232, co daje ok 48% zakładanego czasu. Obserwacja potwierdza więc, że przyczyną nieprawidłowego działania jest wystąpienie przerwania, gdy sprawdzana zmienna ma wartość zmniejszaną z 256 na 255.

Jak sobie z tym poradzić? Nasz mikrokontroler jest 8-bitowy i musi do zmiennych zajmujących więcej niż 1 bajt odwoływać się „na raty". Pozostaje więc wyłączenie przerwań na czas sprawdzania wartości zmiennej:

```
void delay() {
 counter = 488;
 volatile uint16_t tmp;
 do {
  cli();
  tmp = counter;
  sei();
 } while (tmp);
}
```

Funkcją cli() wyłączamy przerwania, a funkcją sei() włączamy. Konieczne przy tym było użycie dodatkowej zmiennej, by w ten sposób oddzielić pobranie wartości zmiennej counter od sprawdzenia tejże wartości. Zmienną tę dobrze zadeklarować jako volatile, aby podczas optymalizacji nie została usunięta jako zbędna.

Zamiast bezpośrednio wywoływać funkcje cli() oraz sei(), możemy skorzystać z makra ATOMIC_BLOCK:

```
void delay() {
 counter = 488;
 volatile uint16_t tmp;
 do {
  ATOMIC_BLOCK(ATOMIC_FORCEON) {
   tmp = counter;
  }
 } while (tmp);
}
```

Aby z niego skorzystać, trzeba dołączyć plik util/atomic.h za pomocą dyrektywy #include. Makro ATOMIC_BLOCK może przyjmować następujące parametry, które określają jego zachowanie po wykonaniu się bloku kodu, którego dotyczy:

- **ATOMIC_RESTORESTATE – pozostaw przerwania w takim stanie w jakim były**
- **ATOMIC_FORCEON – włącz przerwania**
- **ATOMIC_FORCEOFF – wyłącz przerwania**

Generalnie działanie tego parametru sprowadza się do tego, czy na koniec ma być wykonana funkcja sei(), czy nie.

Wyłączając przerwania na czas wykonania jakiegoś fragmentu kodu, tworzymy w tym miejscu tzw. sekcję krytyczną, czyli kod, którego wykonanie nie zostanie przerwane. Jeśli wystąpi jakieś przerwanie, to jego funkcja obsługująca zostanie wykonana dopiero po wykonaniu funkcji sei(). Innymi słowy, funkcja obsługi przerwania nie zaingeruje w nasz kod oznaczony jako sekcja krytyczna. Zostanie on wykonany w jednym kawałku, atomowo.

Kiedy powinniśmy korzystać z sekcji krytycznych (bloków atomowych)? Wszędzie tam, gdzie funkcja obsługi przerwania mogłaby w niepożądany sposób zaingerować w wykonywanie się głównego programu i trudno byłoby znaleźć lepsze rozwiązanie.

Zadania

Do przerwań będziemy jeszcze wracać wielokrotnie. Tymczasem zachęcam do samodzielnych eksperymentów z przerwaniami pochodzącymi z timerów. Jako ćwiczenia można wykonać np.:

1. Dzielnik częstotliwości: częstotliwość na jednej nóżce mikrokontrolera ma być n-krotnie niższa niż na innej nóżce.
2. Generowanie dwóch różnych częstotliwości jednocześnie.

Zadania powinny być wykonane przy użyciu przerwań, bez angażowania głównej pętli programu.

Jak zwykle kody źródłowe do lekcji są dostępne w materiałach dodatkowych do tego numeru EdW w Elportalu. W przypadku jakichkolwiek problemów z realizacją kursu AVR, niejasności lub wątpliwości proszę o kontakt mailowy lub zadanie pytania na forum Elportalu.

Grzegorz Niemirowski
grzegorz@grzegorz.net