Kurs AVR – lekcja 7

Rozwiązania zadań z ostatniego odcinka

Zadania z poprzedniego numeru dotyczyły przykładowego kodu odliczającego sekundy. W pierwszym z zadań trzeba było dodać sygnalizację upłynięcia czasu. Aktualna liczba sekund przechowywana była w zmiennej s. Trzeba więc było dodać instrukcję warunkową sprawdzającą, czy zmienna ta osiągnęła wartość zero. Może się wtedy wykonać kod sygnalizujący upływ czasu, np. migający LED-em. Nasze pierwsze podejście wyglądałoby jak na **listingu 1**.

Po osiągnięciu przez zmienną s wartości 0, uruchamiana jest pętla migająca diodą. Przedtem zmieniana jest wartość rejestru OCR1A na mniejszą, aby uzyskać krótkie opóźnienia i szybkie miganie. Wartość 1562 da półokres 100ms i częstotliwość 5Hz. Oczywiście można tę wartość zmieniać zgodnie z własnymi upodobaniami. Druga czynność przed uruchomieniem pętli to ustawienie pinu portu B do pracy jako wyjście, bo właśnie ten pin jest potem używany do sterowania diodą. Można wykorzystać inny wolny pin. Ustawienie trybu pracy pinu można przenieść na początek funkcji main().

Nasz kod w zasadzie działa poprawnie, ale w momencie włączenia się migania na wyświetlaczu widać liczbę 1. Niby w warunku można zmienić 0 na –1, ale to pogorszy sytuację. Miganie bowiem włącza się w odpowiednim momencie, tylko nie jest już wykonywany kod wyświetlający aktualną liczbę. Rozwiązanie jest proste: linijkę z wywołaniem funkcji printf() w pętli while() trzeba przenieść za linijkę dekrementującą zmienną s:

```
while(!(TIFR & _BV(OCF1A)));
TIFR |= _BV(OCF1A);
s--;
lcdGotoXY(0, 0);
printf("%3u", s);
```

Listing 1

```
#include <avr/io.h>
#include "lcd.h"
#include "keyb.h"
int main(void)
{
  lcdInit();
  lcdInitPrintf();
  keybInit();
  int s = readNumber();
  printf("%d", s);
  TCCR1B = _BV(WGM12) | _BV(CS11) | _BV(CS10);
  OCR1A = 15624;
  while(1) {
    lcdGotoXY(0, 0);
    printf("%3u", s);
    while(!(TIFR & _BV(OCF1A)));
    TIFR |= _BV(OCF1A);
    s--;
    if (s == 0) {
      OCR1A = 1562;
      DDRB = _BV(DDB0);
      while (1) {
        while(!(TIFR & _BV(OCF1A)));
        TIFR |= _BV(OCF1A);
        PORTB = _BV(PB0);
        while(!(TIFR & _BV(OCF1A)));
        TIFR |= _BV(OCF1A);
        PORTB = 0;
      }
    }
  }
}
```

Zadanie mamy rozwiązane, ale dobrze byłoby się zastanowić nad strukturą kodu. W naszym programie wyróżnić można trzy etapy: wczytanie liczby sekund, odliczenie sekund oraz sygnalizacja upływu czasu. Odliczanie oraz sygnalizację mamy w nieskończonej pętli while(), przy czym sygnalizacja też ma swoją nieskończoną pętlę i zatrzymuje wykonanie pętli nadrzędnej. Prosi się, aby pętla odliczająca nie była nieskończona, tylko sprawdzała osiągnięcie zera, a sygnalizacja była oddzielnym blokiem (najlepiej funkcją), wyniesionym poza pętlę odliczającą. Przebudujmy więc nasz kod – **listing 2**.

Program działa tak samo, ale kod jest bardziej czytelny i elastyczny. Do oddzielnej funkcji dobrze byłoby również wynieść odliczanie sekund. Dzięki temu łatwo przerobimy nasz program tak, aby nie działał jednorazowo i nie wymagała resetu w celu odmierzenia kolejnych sekund. Przykładowa realizacja na **listingu 3**.

Funkcja main() znacznie nam się uprościła. Na początku inicjalizujemy wyświetlacz i klawiaturę oraz Timer1. Reszta kodu main() to nieskończona pętla, w której wczytywana jest liczba początkowa oraz wyświetlana jest na ekranie, a następnie wykonywane jest odliczanie funkcją countdown() i miganie funkcją blink().

Funkcja countdown() zawiera pętlę odliczającą, bez zmian w stosunku do poprzedniego przykładu. Nowością jest inicjalizacja rejestru OCR1A, ponieważ jest on również modyfikowany przez funkcję blink(). Ponadto czyszczona jest flaga OCF1A, ponieważ podczas wpisywania liczby na pewno licznik zdążył się „przekręcić" i ustawić tę flagę. Flagi można nie czyścić, ale wtedy liczenie nie będzie odbywało się od wpisanej liczby, tylko od o jeden mniejszej. Powodem będzie natychmiastowe wykonanie się pętli opóźniającej z powodu już ustawionej flagi. Funkcję blink() należało rozbudować tak, aby użytkownik miał możliwość jej przerwania. Dodano więc sprawdzanie stanu klawiatury. Jeśli zostanie wciśnięty przycisk, pętla kończy się i następuje oczekiwanie na zwolnienie przycisku.

Rozwiązując pierwsze zadanie, przepisaliśmy kod na bardziej czytelny i lepiej zorganizowany. Program też jest bardziej przyjazny, bo odliczanie może być uruchamiane wielokrotnie. Nadal jednak mamy poważną niedogodność: podczas wpisywania liczby z klawiatury poszczególne cyfry nie pokazują się na wyświetlaczu. Poprawa tego była treścią zadania drugiego.

Listing 2

```
#include <avr/io.h>
#include "lcd.h"
#include "keyb.h"
void blink();
int main(void)
{
  lcdInit();
  lcdInitPrintf();
  keybInit();
  DDRB = _BV(DDB0);
  int s = readNumber();
  printf("%3u", s);
  TCCR1B = _BV(WGM12) | _BV(CS11) | _BV(CS10);
  OCR1A = 15624;
  while(s != 0) {
    while(!(TIFR & _BV(OCF1A)));
    TIFR |= _BV(OCF1A);
    s--;
    lcdGotoXY(0, 0);
    printf("%3u", s);
  }
  blink();
}
void blink() {
  OCR1A = 1562;
  while (1) {
    while(!(TIFR & _BV(OCF1A)));
    TIFR |= _BV(OCF1A);
    PORTB = _BV(PB0);
    while(!(TIFR & _BV(OCF1A)));
    TIFR |= _BV(OCF1A);
    PORTB = 0;
  }
}
```

Listing 3

```
#include <avr/io.h>
#include "lcd.h"
#include "keyb.h"
void blink();
void countdown(int s);
int main(void)
{
  lcdInit();
  lcdInitPrintf();
  keybInit();
  TCCR1B = _BV(WGM12) | _BV(CS11) | _BV(CS10);
  DDRB = _BV(DDB0);
  while (1) {
    lcdGotoXY(0, 0);
    int s = readNumber();
    lcdGotoXY(0, 0);
    printf("%3u", s);
    countdown(s);
    blink();
  }
}
void countdown(int s) {
  OCR1A = 15624;
  TIFR |= _BV(OCF1A);
  while(s != 0) {
    while(!(TIFR & _BV(OCF1A)));
    TIFR |= _BV(OCF1A);
    s--;
    lcdGotoXY(0, 0);
    printf("%3u", s);
  }
}
void blink() {
  OCR1A = 1562;
  while (readKeyboard() == 0) {
    while(!(TIFR & _BV(OCF1A)));
    TIFR |= _BV(OCF1A);
    PORTB = _BV(PB0);
    while(!(TIFR & _BV(OCF1A)));
    TIFR |= _BV(OCF1A);
    PORTB = 0;
  }
  while (readKeyboard() != 0);
}
```

Top www www.piekarz.pl Firma Piekarz Sp. J. Hurtowania Części Elektronicznych Tel.: 22-835-50-37 Tel.: 22-835-50-41

```
lcdWriteCommand(LCD_COMMAND_ON_OFF | LCD_PARAM_ON_OFF_DISPLAY | LCD_PARAM_ON_OFF_CURSOR);
int s = readNumber();
lcdWriteCommand(LCD_COMMAND_ON_OFF | LCD_PARAM_ON_OFF_DISPLAY);
```

Listing 4

Rozwiązanie jest bardzo proste. Liczbę wczytujemy funkcją readNumber(), trzeba więc w niej umieścić wyświetlanie kolejnych cyfr. Wczytany z klawiatury znak mamy w tablicy input, pod indeksem wskazywanym za pomocą zmiennej inputIndex. Linijka wyświetlająca cyfrę będzie więc miała następującą postać:

```
printf("%c", input[inputIndex]);
```

Ponieważ chcemy wyświetlić znak, używamy specyfikatora %c. Linijkę umieszczamy przed instrukcją inkrementacji indeksu (inputIndex++;). Z racji użycia funkcji printf() trzeba też pamiętać o dodaniu dyrektywy `#include <stdio.h>` na początku pliku keyb.c.

Aby wpisywanie liczby było jeszcze przyjemniejsze, dodajmy w głównym programie wyświetlanie kursora – **listing 4**:

Wyświetlanie kursora włączamy przed wczytaniem liczby i wyłączamy po jej wpisaniu. W ten sposób wyraźnie widać, kiedy program oczekuje od użytkownika wpisania czegoś z klawiatury. Zamiast `LCD_PARAM_ON_OFF_CURSOR` można też użyć `LCD_PARAM_ON_OFF_BLINK` lub obu jednocześnie, zależnie jaki tryb wyświetlania kursora nam się bardziej podoba. W każdym razie nie można opuścić `LCD_PARAM_ON_OFF_DISPLAY`, gdyż spowoduje to wyłączenie wyświetlacza.

Trzecim zadaniem było dodanie funkcji Backspace do wybranego klawisza. O tym, która cyfra jest aktualnie wpisywana, decyduje wspomniana zmienna inputIndex. Trzeba więc ją odpowiednio zmodyfikować w przypadku wciśnięcia klawisza wybranego jako Backspace. Przyjmijmy, że tym klawiszem będzie S15. Zaraz po wywołaniu funkcji getKey() umieszczamy więc kod:

```
if (key == 15) {
  if (inputIndex > 0) inputIndex--;
  continue;
```

Spowoduje on cofnięcie się do poprzedniej cyfry, z wyjątkiem przypadku, gdy nic jeszcze nie wpisaliśmy i nie ma czego kasować. Możemy sprawdzić, że nasz kod w sensie logicznym działa poprawnie. Przykładowo wciśnięcie klawiszy S1, S15, S2 dla nam liczbę 2. Niestety na wyświetlaczu nie widać efektu kasowania. Zastanówmy się, jak sobie z tym poradzić. Jednym z rozwiązań może być przesunięcie kursora w lewo w momencie wciśnięcia klawisza Backspace. Można uznać to za wystarczające, ale będzie to raczej przypominało wciśnięcie strzałki w lewo niż Backspace. Trzeba więc wykasować ostatni znak, zastępując go na wyświetlaczu spacją. Należałoby przesunąć kursor w lewo (co jest jednoznaczne z zaadresowaniem poprzedniej komórki w pamięci wyświetlacza), wpisać na wyświetlaczu spację, a następnie znów przesunąć kursor w lewo – **listing5**.

```
if (key == 15) {
  if (inputIndex > 0) {
    inputIndex--;
    lcdWriteCommand(LCD_COMMAND_SHIFT);
    printf("%c", ' ');
    lcdWriteCommand(LCD_COMMAND_SHIFT);
  }
  continue;
}
```

Listing 5

Jako że używamy tutaj funkcji lcdWriteCommand(), musimy dodać dyrektywę #include "lcd.h" do pliku keyb.c. Kod sekundnika wraz z funkcjami dodanymi podczas rozwiązywania zadań umieszczony jest w materiałach do tego numeru EdW.

Zegar

Mając opanowane wprowadzanie liczb oraz odmierzanie czasu, możemy przystąpić do napisania programu pełniącego funkcję zegara, wyświetlającego aktualny czas na LCD. Chcielibyśmy jednak, aby był to zegar możliwe dokładny. Konieczne jest więc najpierw przekonfigurowanie mikrokontrolera tak, aby korzystał z rezonatora kwarcowego, a nie oscylatora RC. Jak pamiętamy z lekcji 4, potrzebujemy przestawić tzw. fusebity. Dla rezonatora kwarcowego zastosujemy konfigurację jak w **tabeli 1**. Da nam to wartość 0x81 dla fusebajtu high i 0xff dla fusebajtu low, zakładając, że pozostałe bity zostawiamy przy domyślnych wartościach.

CKSEL3	CKSEL2	CKSEL1	CKSEL0	SUT1	SUT0	CKOPT
1	1	1	1	1	1	0

Tabela 1

W okienku Device Programming w środowisku Atmel Studio mamy do dyspozycji sekcję Fuses (**rysunek 1**). Tutaj Opcję SUT_CKSEL ustawiamy na *Ext. Crystal/Resonator High Freq.; Start-up time: 16K CK + 64 ms*. Bit CKOPT powinien być zaznaczony (zaprogramowany, czyli ustawiony na 0).

Ponieważ rezonator na naszej płytce ma częstotliwość 16 MHz, musimy zastanowić się, jak ją podzielić, aby uzyskać 1 Hz. Dotychczas korzystaliśmy z 16-bitowego licznika zawartego w Timer1 i podziału częstotliwości 1 MHz przez 64. Dla częstotliwości 16 MHz potrzebujemy więc 16-krotnie większego podziału, czyli 1024. Szczęśliwie Timer1 ma taki preskaler, możemy więc nadal do rejestru OCR1A wpisywać liczbę 15624 (reset licznika przy kolejnej wartości, czyli 15625). Alternatywnie możemy wykorzystać preskaler 256 i liczbę 62499. Wybierzmy preskaler 1024 i resetowanie licznika przy wartości 15624.

Podsumujmy, co jest konieczne, aby zacząć pisać kod działający przy taktowaniu kwarcem:

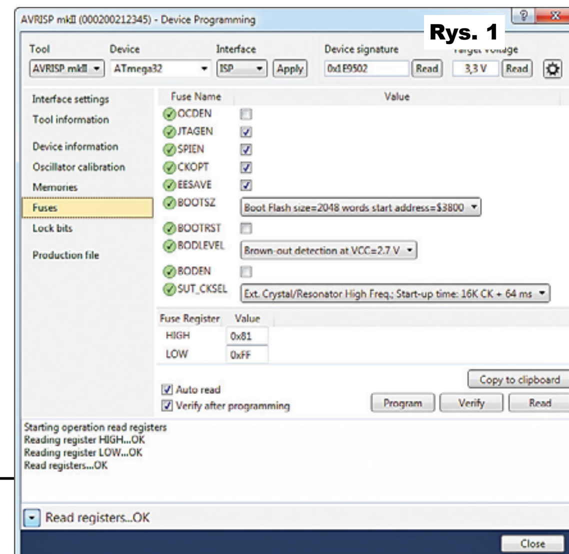
- Przeprogramowanie fusebitów
- Wybór preskalera i wartości maksymalnej dla licznika
- Modyfikacja makra F_CPU tak, aby odpowiadało częstotliwości rezonatora (właściwości projektu -> Toolchain -> Symbols -> wpisujemy F_CPU=16000000)

Na początku programu zegara potrzebujemy zainicjalizować klawiaturę i wyświetlacz, ten fragment będzie wyglądał tak jak przy sekundniku. Natomiast inaczej będzie skonfigurowany preskaler od Timer1. Patrząc na tabelę 3 w lekcji 4, widzimy, że potrzebujemy ustawić bit CS12 zamiast CS11. Bit CS10 pozostaje bez zmian. Pozostaje także bit WGM12, który włącza nam tryb CTC. Inicjalizacja Timer1 będzie więc wyglądać tak:

```
TCCR1B = _BV(WGM12)|_BV(CS12)|_BV(CS10);
```

Teraz musimy się zastanowić, jak będzie wyglądało ustawianie naszego zegara. Mamy bowiem do ustawienia trzy liczby: godziny, minuty i sekundy. Można by więc zacząć od wyświetlenia dwóch spacji, dwukropka, dwóch spacji, znów dwukropka i jeszcze raz dwóch spacji. Ewentualnie zamiast spacji mogą być zera. Następnie trzeba by ustawić kursor na pozycji godzin i wczytać liczbę od użytkownika. Tutaj pojawia się pytanie, czy użytkownik ma każdą z trzech liczb zatwierdzać Enterem. Możemy przyjąć, że po wpisaniu dwóch cyfr na pozycji godzin kursor przesunie się na pozycję minut, a po ich wpisaniu na pozycję sekund i dopiero po ich wpisaniu trzeba będzie wcisnąć Enter. Przy czym, jeśli zakładamy potwierdzanie Enterem, to powinien działać też Backspace.

Potrzebujemy więc nieco inaczej działającej funkcji wczytującej liczby. Tak właściwie trzeba by po prostu wczytać 6 cyfr i dopiero potem przekształcić je na 3 liczby. Bazując na funkcji readNumber(), napiszmy więc taką funkcję – **listing 6**.



Rys. 1

Większość kodu funkcji to nieskończona pętla przetwarzająca klawisze aż do momentu wciśnięcia klawisza S16 pełniącego funkcję Entera. Podobnie jak poprzednio, wciskane klawisze zapamiętywane są w tablicy input. Nie ma tutaj konwersji numeru klawisza do znaku ASCII, gdyż nie mamy poszczególnych liczb w oddzielnych ciągach zakończonych znakiem NULL i w związku z tym nie używamy funkcji atoi(). Pozostała część pętli while() to instrukcje warunkowe obsługujące przemieszczanie kursora i funkcję Backspace. Dwukropki oddzielające godziny od minut i minuty od sekund wymagają bowiem dodatkowego przesuwania kursora. Obsługi wymaga także skrajny przypadek, gdy użytkownik wpisze szóstą cyfrę. Kursor wtedy nie przesuwa się dalej, ale pozostaje na ostatniej pozycji. Do kasowania ostatniej cyfry nie trzeba więc używać przycisku Backspace, wystarczy wcisnąć przycisk innej cyfry. Naszą funkcję po dopisaniu jej nagłówka w pliku keyb.h możemy przetestować w prostym programie – **listing 7**.

Podczas testów może się okazać, że przyciski czasem wciskają się podwójnie. Problem ten jest tym wyraźniejszy, im większa jest częstotliwość taktowania mikrokontrolera. Jak wspomniano na początku kursu, jest to skutkiem drgań styków i najprostszym rozwiązaniem jest dodanie niewielkiego opóźnienia. W pliku keyb.c, na końcu funkcji getKey() a przed instrukcją return wstawmy zatem linijkę:

```
_delay_ms(30);
```

Napisaliśmy funkcję obsługującą wpisywanie czasu, ale funkcja ta nie zwraca żadnej wartości. Musimy jakoś zwrócić godziny, minuty i sekundy. Tymczasem funkcje w języku C zwracają jedną wartość. Jak sobie z tym poradzić? Może dałoby się zadeklarować zmienne dla tych trzech wartości i przekazać je do funkcji, która by je zmodyfikowała? Niestety zmienne w C przekazywane są przez wartość. Rozważmy prościutką funkcję:

```
void fun(uint8_t a) {
  a = 5;
}
```

A następnie taki kod:

```
uint8_t x = 1;
fun(x);
```

Jaka będzie wartość zmiennej x po wykonaniu funkcji fun()? 1 czy 5? Będzie to 1. Dlaczego? Do fun() nie trafia bowiem zmienna jako taka, tylko jej wartość, jaką miała w momencie wywołania funkcji. Wartość ta jest przypisywana do zmiennej lokalnej a, która „żyje" w tej funkcji. Dlatego nie ma znaczenia, co się dzieje wewnątrz funkcji fun(), bo nie operuje ona na oryginalnej zmiennej x, tylko na jej wartości. Można powiedzieć, że działa na kopii zmiennej x.

Jak więc przekazać zmienną? Wróćmy na chwilę do podstaw i przypomnijmy sobie, czym właściwie jest zmienna. Jest to miejsce w pamięci operacyjnej mikrokontrolera, zajmujące jeden lub więcej bajtów. Skoro jest to miejsce, to ma ono jakieś położenie w pamięci. To położenie nazywamy adresem, który w uproszczeniu jest numerem komórki. Mikrokontroler, operując na pamięci, posługuje się właśnie adresami. Ponieważ adresy są dla ludzi niewygodne w użyciu, we współczesnych językach programowania zmiennym możemy nadawać nazwy, dzięki którym łatwo możemy zorientować się, która zmienna do czego służy. Mikrokontroler jednak, aby wykonać operację na zmiennej, musi mieć jej adres. I tutaj możemy zadać sobie pytanie: czy w związku z tym, że adres pozwala „dostać się" do zmiennej, czy nie rozwiązałoby to naszego problemu z przekazywaniem zmiennej do funkcji? Jak najbardziej tak. Jeśli funkcja będzie znała adres oryginalnej zmiennej, to będzie mogła zmodyfikować jej wartość.

W języku C adresy zmiennych nazywamy wskaźnikami, a zmienne przechowujące wskaźniki nazywamy zmiennymi wskaźnikowymi. Często w języku potocznym te pojęcia są używane zamiennie. Działanie wskaźników sprawia problemy początkującym programistom, ale nie jest to wcale takie trudne. Powróćmy do naszego przykładu, tym razem w wersji wskaźnikowej.

```
void fun(uint8_t * a) {
  *a = 5;
}
```

W naszej funkcji pojawiły się dwie gwiazdki. Pełnią one różne funkcje. Jeśli gwiazdka znajduje się za typem (tutaj uint8_t), oznacza, że mamy do czynienia nie ze zmienną tego typu, tylko ze zmienną wskaźnikową zawierającą adres zmiennej tego typu. Czyli w naszym przykładzie a jest zmienną wskaźnikową przyjmującą adres zmiennej typu uint8_t. Innymi słowy, funkcja fun() dostaje wskaźnik na zmienną typu uint8_t i ten wskaźnik przechowuje w zmiennej a. Funkcja ta nie dostaje wartości, tylko lokalizację tej wartości.

Druga gwiazdka też występuje przy zmiennej a, ale tutaj obecny jest operator przypisania. Nie mamy więc tutaj operacji „do zmiennej a przypisz wartość 5", tylko „do miejsca w pamięci, którego adres jest w zmiennej a, wpisz wartość 5". Ze zmienną a w tej funkcji nic się nie dzieje, jej wartość pozostaje taka sama, nadal wskazuje to samo miejsce w pamięci. Tyle tylko, że w tym miejscu jest już zapisania inna wartość. Użyta tutaj gwiazdka jest nazywana operatorem wyłuskania. Nazwa ta może być dziwna, ale staje się jaśniejsza w sytuacji odwrotnej:

```
uint8_t b = *a;
```

Tutaj pod zmienną b podstawiana jest wartość wskazywana przez a. Możemy to odczytać jako „wyłuskaj wartość wskazywaną przez a i podstaw pod b".

A teraz popatrzmy na kod wywołujący funkcję fun():

```
uint8_t x = 1;
fun(&x);
```

Deklarujemy zmienną x i wpisujemy do niej jakąś wartość. Następnie za pomocą operatora adresu (&) pobieramy adres tej zmiennej i przekazujemy do funkcji fun(). W ten sposób funkcja fun() może zmodyfikować wartość zmiennej x.

Zmienne typu wskaźnikowego możemy deklarować w dowolnym miejscu, nie muszą być to wyłącznie parametry funkcji. Poprzedni przykład możemy rozpisać następująco:

```
uint8_t x = 1;
uint8_t * px = &x;
fun(px);
```

Listing 6

```
void readTime() {
  uint8_t input[6];
  uint8_t inputIndex = 0;
  uint8_t key = 0;
  while(1) {
    key = getKey();
    if (key == 15) {
      if (inputIndex > 0) {
        if (inputIndex == sizeof(input) - 1) {
          printf("%c", ' ');
          lcdWriteCommand(LCD_COMMAND_SHIFT | LCD_PARAM_SHIFT_LEFT);
        }
        inputIndex--;
        if (inputIndex == 1 || inputIndex == 3)
          lcdWriteCommand(LCD_COMMAND_SHIFT | LCD_PARAM_SHIFT_LEFT);
        lcdWriteCommand(LCD_COMMAND_SHIFT | LCD_PARAM_SHIFT_LEFT);
        printf("%c", ' ');
        lcdWriteCommand(LCD_COMMAND_SHIFT | LCD_PARAM_SHIFT_LEFT);
      }
      continue;
    }
    if (key < 10) input[inputIndex] = key;
    if (key == 10) input[inputIndex] = 0;
    if (key == 16) {
      break;
    }
    printf("%c", '0' + input[inputIndex]);
    if (inputIndex == 1 || inputIndex == 3)
      lcdWriteCommand(LCD_COMMAND_SHIFT | LCD_PARAM_SHIFT_RIGHT);
    if (inputIndex < sizeof(input) - 1) {
      inputIndex++;
    } else {
      lcdWriteCommand(LCD_COMMAND_SHIFT | LCD_PARAM_SHIFT_LEFT);
    }
  }
}
```

Listing 7

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

int main(void)
{
  lcdInit();
  lcdInitPrintf();
  keybInit();
  TCCR1B = _BV(WGM12) | _BV(CS12) | _BV(CS10);
  lcdWriteCommand(LCD_COMMAND_ON_OFF |
    LCD_PARAM_ON_OFF_DISPLAY | LCD_PARAM_ON_OFF_CURSOR);
  printf("  :  :  ");
  lcdGotoXY(0, 0);
  readTime();
}
```

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

int main(void)
{
  lcdInit();
  lcdInitPrintf();
  keybInit();
  TCCR1B = _BV(WGM12) | _BV(CS12) | _BV(CS10);
  OCR1A = 15624;
  lcdWriteCommand(LCD_COMMAND_ON_OFF |
   LCD_PARAM_ON_OFF_DISPLAY | LCD_PARAM_ON_OFF_CURSOR);
  printf("  :  :  ");
  lcdGotoXY(0, 0);
  uint8_t hours, minutes, seconds;
  readTime(&hours, &minutes, &seconds);
  lcdWriteCommand(LCD_COMMAND_ON_OFF |
    LCD_PARAM_ON_OFF_DISPLAY);
  TIFR |= _BV(OCF1A);
  while(1) {
    while(!(TIFR & _BV(OCF1A)));
    TIFR |= _BV(OCF1A);
    lcdGotoXY(0, 0);
    printf("%.2u:%.2u:%.2u", hours, minutes, seconds);
    seconds++;
    if (seconds == 60) {
      seconds = 0;
      minutes++;
    }
    if (minutes == 60) {
      minutes = 0;
      hours++;
    }
    if (hours == 24) {
      hours = 0;
    }
  }
}
```

Listing 8

Tworzymy dwie zmienne: zwykłą oraz wskaźnikową. Tę pierwszą inicjalizujemy jakąś wartością. Drugą zmienną inicjalizujemy adresem pierwszej zmiennej. Następnie zmienną wskaźnikową (a właściwie jej wartość) przekazujemy do funkcji. W ten sposób funkcja dostała adres pierwszej zmiennej.

Znając podstawy działania wskaźników, możemy dokończyć naszą funkcję readTime(). Zmieńmy jej nagłówek tak, aby przyjmowała zmienne wskaźnikowe. Oczywiście jednocześnie trzeba zmodyfikować jej nagłówek w keyb.h.

```
void readTime(uint8_t * hours, uint8_t * minutes, uint8_t * seconds);
```

Na końcu, po głównej pętli, dopisujemy zwracanie wartości.

```
*hours = 10 * input[0] + input[1];
*minutes = 10 * input[2] + input[3];
*seconds = 10 * input[4] + input[5];
```

Mając gotową funkcję wczytującą czas, możemy dokończyć nasz program, co pokazuje **listing 8**.

Program rozpoczynamy od inicjalizacji klawiatury, wyświetlacza oraz Timer1, którego będziemy używać od odmierzania upływu czasu. Następnie włączane jest wyświetlanie kursora i wyświetlane są dwukropki, między którymi użytkownik wpisuje bieżącą godzinę, minutę i sekundę. Są one wczytywane do odpowiednich zmiennych. Następnie kursor jest wyłączany i w głównej pętli wyświetlany jest bieżący czas. Kropka i liczba 2 oznaczają liczbę w formacie dwucyfrowym, z zerami wiodącymi. Co sekundę zwiększany jest licznik sekund oraz ewentualnie liczniki minut i godzin.

Rozwiązywanie problemów i debugowanie programu

Nasze programy stają się coraz większe i coraz bardziej skomplikowane. Rośnie więc ryzyko popełniania błędów podczas pisania. Jak sobie z tym radzić? Są różne metody.

Po pierwsze, warto najpierw się zastanowić, jak się chce zrealizować daną funkcję. Pomyśleć przez chwilę, może rozrysować sobie na kartce. Początkujący programiści mają tendencję aby natychmiast pisać kod, bez dłuższego zastanowienia się nad problemem. Oczywiście proste rzeczy nie wymagają długiego kombinowania, ale nieraz dobrze jest najpierw poukładać sobie w głowie swój pomysł na realizację danej funkcji. Jakie peryferie mikrokontrolera będą odpowiednie? Czy wiem dokładnie, jak działają? Jakich użyć typów zmiennych? Jaki algorytm byłby najlepszy?

Ważny jest także styl pisania. Nasze funkcje powinny być zwięzłe, mają realizować jedną konkretną czynność, a nie kilka. Przykładem mogą być tutaj nasze biblioteki dla klawiatury i wyświetlacza, które składają się z wielu małych funkcji. Podział kodu na mniejsze części wykonywaliśmy też w tym odcinku, przy rozwiązywaniu zadań z odcinka poprzedniego. Taka organizacja sprawia, że łatwiej odnaleźć fragment, w którym potencjalnie może być błąd. Jest to ważne nie tylko dla nas, ale też gdy się zdarzy, że będziemy szukać pomocy na jakimś forum. Osobom chcącym nam pomóc będzie dużo łatwiej, jeśli nasz program będzie miał logiczną strukturę.

Styl dotyczy także deklaracji zmiennych. Szczególnie chodzi o to, żeby unikać zmiennych globalnych, czyli tych, które nie są zadeklarowane wewnątrz żadnej funkcji. Funkcje powinny potrzebne im zmienne otrzymywać poprzez parametry. Dzięki temu można łatwo się zorientować, jak wygląda przepływ danych. W przypadku zmiennych globalnych jest to gorzej widoczne, trudniej zauważyć, w którym momencie następuje odwołanie do zmiennej. Oczywiście zmienne globalne nie są z definicji złe i bywają potrzebne, jednak nie należy ich nadużywać.

Kłopoty mogą sprawiać zawarte w mikrokontrolerze różne układy peryferyjne, jak timery czy interfejsy komunikacyjne. Tutaj nieoceniona jest dokumentacja do mikrokontrolera. Odpowiedni PDF można znaleźć, wpisując do wyszukiwarki słowa ATmega32 datasheet. Dokumentacja jest długa i w języku angielskim, przez co odstraszająca dla niektórych osób. Jednak jest wyczerpująca i zawiera odpowiedzi na zdecydowaną większość problemów, z jakimi styka się programista tego mikrokontrolera. Warto więc do niej zaglądać. Firma Atmel udostępnia też wiele not aplikacyjnych, w których są szersze omówienia różnych zagadnień a także gotowe programy. Przykładowo układ Apollo Roger Beep z EdW 07/2016 powstał z wykorzystaniem noty aplikacyjnej AVR314.

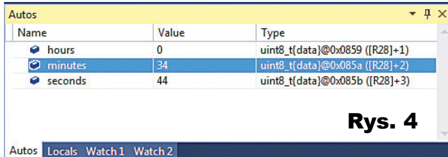
Oczywiście staranne pisanie kodu czy czytanie dokumentacji to nie wszystko. Nieraz konieczne staje się przeanalizowanie zachowania programu podczas jego działania. Tutaj w lepszej sytuacji są posiadacze programatorów z interfejsem JTAG. Może być to AVR Dragon lub Atmel-ICE (w wersji pełnej lub Basic). Są one dużo droższe od programatorów ISP, ale pozwalają na zatrzymywanie programu na danej linijce lub w dowolnym momencie oraz manipulację zmiennymi i rejestrami. Korzystanie z wbudowanego w Atmel Studio debuggera zostało opisane w kolejnym rozdziale.

Co zrobić, jeśli mamy tylko programator ISP? Musimy wtedy wykorzystać peryferie mikrokontrolera do informowania świata zewnętrznego o stanie programu. Możemy np. dopisać fragment kodu, który włączy LED-a, gdy jakaś zmienna będzie miała określoną wartość. Jeśli mamy wolny port, wówczas zmienną możemy wystawić na tym porcie. Za pomocą LED-ów lub woltomierza można wtedy sprawdzić, na których pinach są zera, a na których jedynki i obliczyć wartość dziesiętną. Dane możemy też wysłać sobie do komputera za pomocą portu szeregowego, który będziemy omawiać w jednym z kolejnych odcinków. Na naszej płytce testowej jest też wyświetlacz, na którym możemy prezentować dane testowe.

Gdy coś nie działa zgodnie z oczekiwaniami, początkujący programiści mają tendencję do obwiniania używanych narzędzi. Podejrzewają kompilator, szukają błędu w standardowych bibliotekach lub dopatrują się pomyłki w dokumentacji mikrokontrolera. Owszem, nic nie jest doskonałe i takie problemy się zdarzają, jednak zwykle błąd jest w naszym kodzie. Gdzieś jest niezainicjalizowana zmienna, gdzieś wychodzimy poza zakres tablicy, gdzieś jest zamieniony plus z minusem. Czasem programista nie doczyta dokładnie, jak coś działa. Np. przeczyta, że zapis do określonego rejestru powoduje wysłanie bajtu przez port szeregowy. Zapisuje dwa razy i dziwi się, że została wysłana tylko druga wartość. Tymczasem okazuje się, że zapis powoduje tylko rozpoczęcie wysyłania i wykonanie natychmiastowo drugiego zapisu, zanim bajt się zdąży wysłać, spowoduje tylko nadpisanie wartości, która ma być wysłana. Aby wysłać drugi znak, trzeba poczekać na koniec transmisji pierwszego. Warto więc czytać dokumentację dokładnie.

Błędy często występują przy przetwarzaniu jakiegoś zakresu wartości i program przetwarza o jedną wartość za mało lub za dużo. W języku angielskim określane są mianem off by one errors. Zazwyczaj wiąże się to z zapominaniem, jak są indeksowane tablice, np. że jeśli tablica ma 10 elementów, to jej elementy mają indeksy od 0 do 9. Powoduje to potem nieprawidłowe sprawdzanie warunków brzegowych, np. porównanie i <= 10 zamiast i < 10. Ciekawe są też błędy przy korzystaniu z funkcji kopiuj-wklej:

a = t[0];
b = t[1];
c = t[1];



Rys. 4

Tutaj programista, chcąc pobrać trzy pierwsze elementy z tablicy, przekleił dwukrotnie pierwszą linijkę. Zmienną a zmienił na b i na c, 0 zmienił na 1, ale za drugim razem się zagapił i też zmienił na 1 zamiast na 2. Tego typu błędy mogą się wydawać śmieszne, ale łatwo je przeoczyć i nieraz znalezienie ich zajmuje sporo czasu.

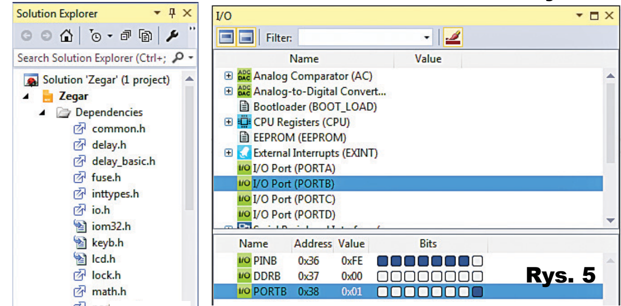
Przed błędami próbuje nas ustrzec wbudowane w Atmel Studio sprawdzanie pisowni. Dzięki niemu trudniej zrobić literówkę albo zapomnieć o przecinku czy średniku. Jest to przydatna funkcja, ale ma jeden uciążliwy błąd. Otóż makro _BV() jest podkreślane jako nieznane, mimo że dołączamy w naszym kodzie bibliotekę avr/io.h, która z kolei dołącza bibliotekę sfr_defs.h, zawierającą definicję tego makra. Na szczęście rozwiązanie jest proste. Jeśli nasz kod skompiluje się prawidłowo, w okienku Solution Explorer pojawiają się pliki w folderze Dependencies (**rysunek 2**). Trzeba po prostu kliknąć dwukrotnie plik sfr_defs.h, aby go otworzyć. Wtedy sprawdzanie pisowni przeanalizuje go i przestanie zgłaszać zastrzeżenia do makra _BV().

Debugowanie programu za pomocą JTAGa

Jedną z najważniejszych rzeczy, jakie daje JTAG, jest możliwość zatrzymania programu w wybranym miejscu i sprawdzenie jego stanu. Aby program zatrzymał się na konkretnej linijce kodu, wystarczy kliknąć po jej lewej stronie, na brzegu obszaru edycji. Pojawi się czerwona kropka oznaczająca, że w tym miejscu jest tzw. breakpoint. Aby program wykonywał się pod kontrolą debuggera, z menu Debug wybieramy Continue lub wciskamy F5. Wtedy nastąpi zaprogramowanie mikrokontrolera i uruchomienie programu. Gdy mikrokontroler wykonując program, osiągnie linijkę, która ma breakpoint, nastąpi zatrzymanie programu. Linijka zostanie podświetlona na żółto (**rysunek 3**). Oznacza to, że wykonana została linijka poprzednia a ta podświetlona zostanie wykonana, gdy uruchomimy program dalej. Innymi słowy, breakpoint powoduje zatrzymanie programu przed wskazaną linijką.

Gdy program jest zatrzymany, możemy obejrzeć zawartość zmiennych. Są one widoczne w okienkach Autos i Locals. Wyświetlane są też typy zmiennych i ich adresy w pamięci (**rysunek 4**). Przydatną funkcją jest możliwość edytowania wartości zmiennych. Oprócz wspomnianych okienek możemy też najechać myszą na daną zmienną w kodzie i wtedy pojawi się mała etykietka z wartością tej zmiennej. Ona też pozwala na zmianę wartości zmiennej. Dzięki interfejsowi JTAG możemy też kontrolować stan rejestrów. Jeśli okienko rejestrów jest niewidoczne, z menu Debug -> Windows wybieramy pozycję I/O. Pojawi się okienko, w którym mamy podgląd oraz możliwość edycji wartości rejestrów peryferii (**rysunek 5**). Dzięki jego pomocy mamy wygodny dostęp do rejestrów. Wartości rejestrów można zmieniać, nie tylko wpisując do nich wartości szesnastkowe, ale też przestawiać pojedyncze bity, klikając poszczególne pola. Jak to działa, można zobaczyć na przykładzie rejestrów portów. Gdy mamy podłączoną diodę świecącą do pinu określonego portu, możemy ją włączać i wyłączać, klikając odpowiedni bit rejestru sterującego należącego do tego portu.

Bardzo przydatną funkcją jest możliwość ręcznej kontroli wykonywania się programu. Klawisz F10 (Step Over) powoduje wykonanie bieżącej linijki i zatrzymanie się na następnej. Jeśli bieżąca linijka to wywołanie funkcji, wówczas możemy do niej wejść klawiszem F11 (Step Into). Z kolei Shift+F11 (Step Out) powoduje wykonanie bieżącej funkcji do końca, wyjście z niej i zatrzymanie się w funkcji wywołującej w miejscu wywołania. Klawisz F5 „puszcza" program dalej, do ewentualnego kolejnego breakpointu. W menu Debug mamy też funkcję Run To Cursor, która powoduje kontynuację wykonywania programu i zatrzymanie na wskazanej linijce. Można powiedzieć, że jest to szybkie wstawienie wirtualnego breakpointu. W menu Debug mamy jeszcze m.in. polecenie Start Debugging and Break, które powoduje uruchomienie programu, z tym że od razu się on zatrzymuje na początku funkcji main(). Dzięki temu można śledzić wykonanie programu do samego początku bez potrzeby wstawiania breakpointu. Polecenie Break All służy do zatrzymania programu w dowolnym momencie. Debugowanie przerywamy poleceniem Stop Debugging.



Rys. 3
Rys. 2
Rys. 5

Uważni czytelnicy mogą zapytać, w jaki sposób można wykonywać program krokowo, skoro w mikrokontrolerze jest program skompilowany do języka maszynowego, a nie kod w języku C. Na szczęście za pomocą tzw. symboli debugger jest w stanie powiązać sobie kod w C z kodem maszynowym. Potrafi też powiązać nazwy zmiennych z adresami w pamięci. Może jednak mieć problemy, gdy ustawimy bardzo duży stopień optymalizacji w ustawieniach kompilatora lub będziemy chcieli postawić breakpoint na wywołaniu funkcji inline, takich jak _delay_ms() czy _delay_us().

Zadania

Wiadomości z tego odcinka pozwalają zrealizować wiele ciekawych projektów, np. sterowników włączających określone urządzenia o różnych porach. Zamiast diody świecącej możemy przecież sterować np. tranzystorem włączającym przekaźnik lub silniczek. Warto też pomyśleć o obwodach wejściowych, jak np. kontaktronowy czujnik otwarcia drzwi i włączenie sygnału dźwiękowego po określonym czasie. Możliwości jest dużo, a im więcej samodzielnych eksperymentów, choćby najprostszych, tym lepiej. Tradycyjnie też zamieszczam kilka przykładowych zadań, które zostaną rozwiązane w kolejnym odcinku kursu.

1. Do naszego zegara można wpisać nieprawidłowe wartości, np. godzinę 30. Dopisać walidację wpisywanych przez użytkownika liczb.
2. Dodać możliwość przestawiania zegara w trakcie pracy.
3. Dodać funkcję budzika.

Grzegorz Niemirowski
grzegorz@grzegorz.net