Kurs AVR – lekcja 5

Rozwiązania zadań z ostatniego odcinka

Pierwszym zadaniem domowym z poprzedniego odcinka była poprawa prostego miernika częstotliwości tak, aby działał także z małą częstotliwością oraz w przypadku jej braku. Możemy to zrobić w ten sposób, że będziemy sprawdzać, czy pomiędzy zdarzeniami przepełnienia się licznika pojawiło się jakieś zbocze na wejściu ICP1. Gdy licznik się przepełnia i ustawiona zostaje flaga TOV1, ustawiamy zmienną overflow na 1. Wcześniej jednak sprawdzamy, czy nie została ona wyzerowana przez funkcję Input Capture. Jeśli nie została wyzerowana, to ustawiamy rejestr PORTA na 1, co spowoduje zaświecenie pierwszej diody. Rozwiązujemy w ten sposób problem z niepoprawnym wskazaniem przy braku sygnału wejściowego.

Ponadto trzeba poprawić sprawdzanie, czy wykryliśmy drugie zdarzenie Input Capture, które jest sygnałem, że można wykonać obliczenia. Może się bowiem zdarzyć tak, że między dwoma zdarzeniami upłynie wiele okresów licznika, ale odejmowanie wartości rejestru ICR1 da wynik taki, jakby zdarzenia te były w jednym okresie licznika. Dlatego przy przepełnieniu licznika możemy dodać zerowanie zmiennej prevICR1 i obliczenia wykonywać tylko wtedy, gdy zmienna ta jest różna od zera (**listing 1**).

```
#include <avr/io.h>
int main(void) {
 DDRA = 0b11111111;
 TCCR1B = _BV(CS10);
 uint16_t prevICR1 = 0;
 uint8_t overflow = 0;
 while(1) {
  if(TIFR & _BV(ICF1)) {
   TIFR |= _BV(ICF1);
   if (ICR1 > prevICR1 && prevICR1) {
    uint16_t period = ICR1 - prevICR1;
    uint16_t freq = 500 / period;
    PORTA = _BV(freq);
    overflow = 0;
   }
   prevICR1 = ICR1;
  }
  if(TIFR & _BV(TOV1)) {
   TIFR |= _BV(TOV1);
   if (overflow) PORTA = 1;
   overflow = 1;
   prevICR1 = 0;
  }
 }
}
```

Listing 1

Drugim zadaniem była zmiana koloru diody RGB za pomocą klawiatury. Potrzebujemy do tego celu trzech sygnałów PWM i możliwości sterowania ich wypełnieniem. Jeden sygnał PWM możemy wygenerować za pomocą Timer0, a dwa za pomocą Timer1, gdyż ma on dwa kanały. Timery pracują w trybie 8-bitowym. Funkcja PWM ma włączoną korekcję fazy, aby wyeliminować niepożądane szpilki dla zerowego wypełnienia i uzyskać całkowicie zgaszoną diodę dla wypełnienia zerowego. W zależności od wciśniętego przycisku inkrementowane lub dekrementowane są zmienne dla każdego koloru. Klawisze pierwszego rzędu klawiatury zwiększają intensywność świecenia, a klawisze dolnego rzędu zmniejszają. Używana jest funkcja readKeyboard() znana z poprzednich lekcji. Rejestry timerów ustawiane są na wartości będące kwadratami wartości zmiennych, aby uzyskać sterowanie nieliniowe. Po ustawieniu rejestrów wypełnienia PWM program czeka na puszczenie przycisku klawiatury. Zostały użyte zmienne ze znakiem (typu int8_t) ze względu na możliwość obsługi wartości –1, która może się pojawić przy próbie ściemnienia zgaszonego już koloru. W **listingu 2** znajduje się sama funkcja main().

```
int main(void) {
 DDRD = _BV(DDD4) | _BV(DDD5);
 DDRB = _BV(DDB3);
 DDRA = _BV(DDA0) | _BV(DDA1) | _BV(DDA2) | _BV(DDA3);
 TCCR1A = _BV(WGM10) | _BV(COM1A1) |_BV(COM1B1);
 TCCR1B = _BV(CS10);
 TCCR0 = _BV(WGM00) | _BV(COM01) | _BV(CS01);
 int8_t r = 0;
 int8_t g = 0;
 int8_t b = 0;
 while(1) {
  uint8_t val = readKeyboard();
  if (val) {
   switch (val) {
    case 1: r++; break;
    case 2: g++; break;
    case 3: b++; break;
    case 5: r--; break;
    case 6: g--; break;
    case 7: b--; break;
   }
   if (r < 0) r = 0;
   if (g < 0) g = 0;
   if (b < 0) b = 0;
   if (r > 15) r = 15;
   if (g > 15) g = 15;
   if (b > 15) b = 15;
   OCR1A = r * r;
   OCR1B = g * g;
   OCR0 = b * b;
   while(readKeyboard());
  }
 }
}
```

Listing 2

Tematem trzeciego zadania było samoczynne zmienianie się kolorów diody. Możemy wykorzystać kod z poprzedniego zadania, ale zmiany koloru nie uzależniać już od klawiatury. Kolory składowe mogą być zmieniane według różnych algorytmów. Prostym rozwiązaniem może być np. zmienianie intensywności każdego koloru z różną częstotliwością (**listing 3**). Najwolniej zmieniana jest intensywność koloru czerwonego. Kolor zielony zmienia intensywność dwa razy szybciej, a niebieski trzy razy. W ten sposób w ciągu czasu otrzymujemy różne kombinacje kolorystyczne. Przy osiągnięciu wartości skrajnej odwracany jest kierunek zmian. Podział przez 5 i 10 to przykład sposobu na zmniejszenie jasności niektórych kolorów w przypadku, gdyby dioda RGB miała niektóre kolory intensywniejsze od innych.

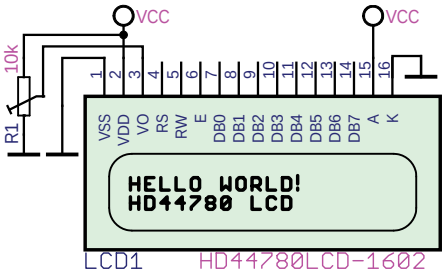
Obsługa wyświetlacza LCD

W tej lekcji przyjrzymy się największemu elementowi na naszej płytce testowej: wyświetlaczowi LCD. Jest to wyświetlacz alfanumeryczny, mogący prezentować dwie linijki po 16 znaków, wyposażony w sterownik zgodny z układem HD44780. Tego typu wyświetlacze są szeroko stosowane od wielu lat. W EdW były już szczegółowo opisywane w numerach 11/97–3/98. Tutaj z racji ograniczonego miejsca skupimy się na najważniejszych informacjach.

Co to znaczy, że wyświetlacz jest alfanumeryczny? Oznacza to, że potrafi on

```
int main(void) {
 DDRD = _BV(DDD4) | _BV(DDD5);
 DDRB = _BV(DDB3);
 DDRA = _BV(DDA0) | _BV(DDA1) | _BV(DDA2) | _BV(DDA3);
 TCCR1A = _BV(WGM10) | _BV(COM1A1) |_BV(COM1B1);
 TCCR1B = _BV(CS10);
 TCCR0 = _BV(WGM00) | _BV(COM01) | _BV(CS01);
 uint8_t r = 0;
 uint8_t rd = 1;
 uint8_t g = 0;
 uint8_t gd = 1;
 uint8_t b = 0;
 uint8_t bd = 1;
 while(1) {
  if (rd) r++; else r--;
  if (gd) g += 2; else g -= 2;
  if (bd) b += 3; else b -= 3;
  if (r == 255 || r == 0) {
   rd = !rd;
  }
  if (g == 254 || g == 0) {
   gd = !gd;
  }
  if (b == 255 || b == 0) {
   bd = !bd;
  }
  OCR1A = r;
  OCR1B = g / 5;
  OCR0 = b / 10;
  _delay_ms(10);
 }
}
```

Listing 3



Rys. 1

wyświetlać litery, cyfry i inne symbole, których kształty ma zapisane w swojej pamięci. Użytkownik nie musi zajmować się każdym pikselem oddzielnie. Określa tylko, na której pozycji jaki znak ma być wyświetlony. Gdybyśmy jednak chcieli wyświetlić znak, którego wyświetlacz nie obsługuje, też mamy taką możliwość. Dostępna jest pamięć 8 znaków, których wygląd można zdefiniować piksel po pikselu.

Podłączanie wyświetlacza. Rozkład wyprowadzeń jest widoczny na **rysunku 1**. VSS i VDD to zasilanie. Możemy używać napięcia zasilającego w zakresie 3,3–5V, w nowszych modelach 2,7–5,5V. Napięcie na nóżce VO steruje kontrastem. Zwykle podłącza się do niej potencjometr montażowy, pracujący jako dzielnik napięcia zasilającego. W ten sposób można ustawić potrzebny kontrast. Jeśli ktoś ma w swoich zbiorach bardzo stary model wyświetlacza, może być konieczne dostarczenie ujemnego napięcia dla kontrastu. Napięcie takie można wytworzyć np. za pomocą układu ICL7660. Końcówki A i K to nóżki zasilania podświetlenia, w zakresie do 5V. Niektóre wyświetlacze mogą być pozbawione podświetlenia i nie mieć tych wyprowadzeń. Pozostałe nóżki to cyfrowe linie sterujące oraz danych. Wyprowadzenie RS decyduje o tym, czy do wyświetlacza przekazujemy kod znaku do wyświetlenia (nóżka w stanie wysokim), czy też komendę sterującą jego pracą (nóżka w stanie niskim). Za pomocą nóżki R/W przełączamy się między odczytem (stan wysoki) a zapisem (stan niski) do wyświetlacza. W układach niekorzystających z funkcji odczytu z wyświetlacza nóżkę R/W łączy się na stałe z masą. Nóżka E synchronizuje przesył danych. Są one zapisywane lub odczytywane na opadającym zboczu na tejże nóżce. Jeśli więc chcemy np. przesłać kod znaku do wyświetlenia, wówczas ustawiamy RS na 1, RW na 0, kod znaku binarnie na liniach danych (DB0–DB7), a następnie przestawiamy E z 1 na 0. Jeśli linia E była już w stanie 0, trzeba ją najpierw na chwilę przestawić w stan 1.

Dane mogą być przesyłane z/do wyświetlacza 8-bitowo, z wykorzystaniem wszystkich linii DB0–DB7. Nie jest to jednak zbyt praktyczne, bo nawet jeśli nie używamy nóżki R/W, to do obsługi wyświetlacza musimy wykorzystać łącznie aż 10 nóżek mikrokontrolera. Nie jest to pożądana sytuacja, bo wymusza użycie mikrokontrolera z większą liczbą nóżek, a więc zwykle droższego, oraz komplikuje płytkę drukowaną. Na szczęście możemy wykorzystać komunikację 4-bitową, wówczas wykorzystywane są końcówki DB4–DB7, a DB0–DB3 pozostają niepodłączone. Oczywiście oznacza to pewne utrudnienie, bo każdy bajt trzeba przesyłać w dwóch krokach: najpierw starsze 4 bity, potem młodsze 4 bity. Nie jest to jednak duży problem z programistycznego punktu widzenia. Sterowanie 8-bitowe wykorzystuje się w sumie tylko wtedy, gdy w układzie już jest używana 8-bitowa magistrala.

Rodzaje pamięci. Wyświetlacze zgodne z HD44780 mają trzy pamięci: CG ROM, CG RAM i DD RAM. W CG ROM (Character Generator ROM) zapisany jest wygląd poszczególnych znaków obsługiwanych przez wyświetlacz. W CG RAM (64 bajty) użytkownik może zdefiniować własne znaki, piksel po pikselu, po 8 bajtów na każdy z 8 znaków użytkownika. Każdy bajt opisuje piksele jednej linijki znaku. 8 znaków to za mało, aby np. zdefiniować wszystkie polskie litery, małe i duże, ale zwykle nie są potrzebne wszystkie i taka pojemność pamięci CG RAM jest wystarczająca.

Kody znaków, które mają być wyświetlone, przechowywane są w pamięci DD RAM (Display Data RAM). Modyfikując tę pamięć, zmieniamy wyświetlaną treść, kolejne bajty odpowiadają kolejnym pozycjom na wyświetlaczu. Jeśli do danej komórki DD RAM wpiszemy kod od 0 do 15, na odpowiadającej tej komórce pozycji wyświetlony zostanie zdefiniowany przez nas znak z CG RAM. Pamięć ta ma tylko 8 bajtów, więc kody od 8 do 15 dają takie same znaki, jak kody od 0 do 7. Kody od 16 do 127 dają standardowe znaki ASCII, o wyglądzie zdefiniowanym w CG ROM. Np. aby wyświetlić literę E, trzeba zapisać do CG RAM liczbę 69, bo jest to jej kod ASCII. Kody od 128 w górę dają znaki zależne od producenta wyświetlacza, zwykle japońskie lub zachodnioeuropejskie.

Pamięć DD RAM ma 80 bajtów. Jeśli mamy wyświetlacz jednowierszowy, pamięć ta adresowana jest zwyczajnie, od adresu 00h do 4Fh. Nasz wyświetlacz ma jednak dwa wiersze i adresowanie wygląda inaczej: po 40 bajtów na górną i dolną linijkę, przy czym adresacja nie jest już ciągła. Pierwsze 40 bajtów ma adresy od 00h do 27h, a drugie 40 bajtów ma adresy od 40h do 67h. Chcąc więc np. wyświetlić literkę na drugiej pozycji drugiego wiersza, trzeba ją zapisać pod adresem 41h (65 dziesiętnie).

Można sobie zadać pytanie, jaki sens ma pamięć większa niż pole odczytowe wyświetlacza? Po pierwsze daje to pewną uniwersalność, jest jeden model pamięci do różnych wyświetlaczy. Po drugie wyświetlacz ma funkcję przewijania. Jeśli potrzebujemy przewijać dłuższy tekst, możemy go wysłać w całości do wyświetlacza i potem wydawać tylko komendę przesuwającą. Gdyby pamięć miała tylko 2*16=32 bajty, to chcąc uzyskać efekt przewijania dłuższego tekstu, trzeba by w każdym kroku na nowo wysyłać odpowiednio przycięty tekst. Podobnie przewijanie można wykorzystać do przełączania się między wyświetlaniem dwóch różnych komunikatów. Wystarczy zapisać oba w pamięci, jeden w części widocznej a drugi w niewidocznej, a potem w razie potrzeby przewijać. Oba wiersze wyświetlacza przewijane są niezależnie. Nie ma więc obawy, że tekst z górnego wiersza pojawi się w dolnym.

Komendy sterujące. Oprócz kodów znaków do wyświetlenia, do wyświetlacza można wysyłać też różne komendy sterujące jego pracą. Protokół komunikacyjny został przedstawiony w **tabeli 1**. Jak widać, komendy wydawane są, gdy linia RS jest w stanie niskim. W stanie niskim jest też linia R/W, z wyjątkiem sytuacji, gdy odczytujemy flagę zajętości. Komendy są ośmiobitowe. Ich charakterystyczną cechą jest to, że z lewej strony, czyli od najbardziej znaczących bitów, mają określoną liczbę zer a następnie jedynkę. Miejsce, w którym wypada ta jedynka, identyfikuje komendę. Bity znajdujące się na prawo od tej jedynki, czyli mniej znaczące, są dodatkowymi parametrami komendy. Przykładem może być komenda Display ON/OFF, która ma trzy bity parametrów: D, C i B. Ustawienie 1 na tych bitach pozwala odpowiednio: włączyć wyświetlacz, włączyć kursor i włączyć miganie kursora. Kursor pokazuje, gdzie pojawi się kolejny znak, czyli wskazuje aktualną komórkę pamięci DD RAM, która zostanie zapisana, gdy wyślemy do wyświetlacza kod jakiegoś znaku. Znaczenie bitów:

- **I/D** – inkrementacja/dekrementacja, 1 to przesuwanie kursora w prawo, 0 to przesuwanie w lewo
- **S** – jeśli ustawiony na 1, to wyświetlacz przewija się w momencie wpisywania znaków
- **D** – 1 włącza wyświetlacz
- **C** – 1 włącza wyświetlanie znaku kursora (_)
- **B** – 1 włącza miganie w miejscu kursora
- **S/C** – 1 przewija wyświetlacz, 0 przesuwa kursor
- **R/L** – 1 przesuwa w prawo, 0 w lewo
- **DL** – 1 ustawia interfejs 8-bitowy, 0 ustawia 4-bitowy
- **N** – 1 dla wyświetlacza 2-wierszowego, 0 dla 1-wierszowego
- **F** – 1 ustawia czcionkę 5x10 pikseli, 0 ustawia czcionkę 5x8 pikseli

Komenda	D7	D6	D5	D4	D3	D2	D1	D0	Opis
Clear display	0	0	0	0	0	0	0	1	Czyści wyświetlacz i ustawia adres DD RAM na 0
wReturn home	0	0	0	0	0	0	1	-	Ustawia adres DD RAM na 0, resetuje ewentualne przewinięcie
Entry mode set	0	0	0	0	0	1	I/D	S	Określa kierunek przesuwania się kursora i włącza ewentualne przewijanie przy zapisie znaków
Display ON/OFF	0	0	0	0	1	D	C	B	Włącza: wyświetlacz, kursor i miganie kursora
Cursor or display shift	0	0	0	1	S/C	R/L	-	-	Przewija wyświetlacz lub przesuwa kursor
Function set	0	0	1	DL	N	F	-	-	Określa interfejs, liczbę linii i czcionkę
Set CG RAM address	0	1	ACG	ACG	ACG	ACG	ACG	ACG	Ustawia adres CG RAM, pod którym będzie wykonywany zapis lub odczyt tej pamięci
Set DD RAM address	1	ADD	ADD	ADD	ADD	ADD	ADD	ADD	Ustawia adres DD RAM, pod którym będzie wykonywany zapis lub odczyt tej pamięci

Tabela 1

Na uwagę zasługują komendy do adresacji CG RAM i DD RAM. Za ich pomocą informujemy wyświetlacz, do której komórki pamięci CG RAM lub DD RAM ma trafić przesyłany później znak. Przydatne jest to szczególnie do DD RAM, bo dzięki temu możemy łatwo umieszczać znaki w dowolnych miejscach wyświetlacza.

Inicjalizacja wyświetlacza. Zanim zaczniemy korzystać z wyświetlacza, musimy go zainicjalizować. Niestety wyświetlacze z omawianym sterownikiem HD44780 nie są od razu gotowe do pracy i konieczne jest najpierw wydanie kilku komend. Inicjalizacja przebiega różnie dla trybu 8-bitowego i 4-bitowego, jednak początek jest taki sam, niezależnie od liczby podłączonych linii danych. Po włączeniu zasilania konieczne jest odczekanie 15 milisekund. Potem trzeba wydać komendę Function Set określającą interfejs 8-bitowy. Następnie trzeba odczekać co najmniej 4,1 milisekundy i znów wydać tę samą komendę. Odczekując jeszcze 100 mikrosekund, trzeba wydać komendę po raz trzeci. Dopiero teraz można przejść w tryb 4-bitowy, wydając komendę Function Set z określeniem interfejsu 4-bitowego. Jeśli to zrobimy, kolejne komendy trzeba już wydawać w trybie zgodnym z zasadą działania interfejsu 4-bitowego, czyli w dwóch krokach, po pół bajtu. Kolejne komendy inicjalizacyjne to znów Function Set, ale już z określonymi bitami N i F, następnie Display OFF, Clear oraz Entry Mode Set. Na koniec pozostaje włączenie wyświetlacza.

Biblioteka dla wyświetlacza. Spróbujmy napisać kod do obsługi wyświetlacza. Dobrze by było, aby miał postać biblioteki, którą łatwo będzie dołączać do tworzonych projektów. Stwórzmy więc w Atmel Studio nowy projekt i obok głównego pliku .c z kodem utwórzmy plik lcd.c. Będziemy w nim umieszczać tylko to, co jest związane z obsługą LCD. Potrzebny będzie też plik nagłówkowy lcd.h, za pomocą którego będziemy udostępniać funkcje LCD głównemu plikowi projektu. W okienku Solution Explorer klikamy więc prawym przyciskiem na projekcie i z menu wybieramy Add -> New Item… Z listy wybieramy C File i jako nazwę wpisujemy lcd.c. Następnie dodajemy podobnie plik nagłówkowy, wybierając Include File i wpisując lcd.h.

Założeniem jest też, że nasza biblioteka, którą można też nazwać sterownikiem, ma obsługiwać zarówno interfejs 8-, jak i 4-bitowy. Oprócz zapisu do modułu będzie też obsługiwany odczyt. Bibliotekę będziemy pisać od funkcji najniższego poziomu, przechodząc do wysokopoziomowych.

Na najniższym poziomie znajduje się sterowanie wyprowadzeniami wyświetlacza. Wyprowadzenia te są połączone z określonymi pinami mikrokontrolera, potrzebujemy więc ustawiać te piny w odpowiednie stany. Przerabialiśmy to już w pierwszych lekcjach kursu, ale pomyślmy, jak to zrobić, aby nasza biblioteka była konfigurowalna. Zdefiniujmy więc makra dla pinów połączonych z wyświetlaczem.

```
#define LCD_E_PIN PORTA0
#define LCD_RW_PIN PORTA1
#define LCD_RS_PIN PORTA2
#define LCD_E_PORT PORTA
#define LCD_RW_PORT PORTA
#define LCD_RS_PORT PORTA
#define LCD_E_DDR DDRA
#define LCD_RW_DDR DDRA
#define LCD_RS_DDR DDRA
```

W ten sposób, podłączając wyświetlacz do innych pinów, musimy tylko zmienić definicje makr w jednym miejscu. Nie trzeba przeglądać całego kodu w poszukiwaniu odwołań do nich. Za pomocą makr możemy też zdefiniować operacje ustawiające stan pinu:

```
#define LCD_E_LOW LCD_E_PORT &= ~_BV(LCD_E_PIN)
#define LCD_RW_LOW LCD_RW_PORT &= ~_BV(LCD_RW_PIN)
#define LCD_RS_LOW LCD_RS_PORT &= ~_BV(LCD_RS_PIN)
#define LCD_E_HIGH LCD_E_PORT |= _BV(LCD_E_PIN)
#define LCD_RW_HIGH LCD_RW_PORT |= _BV(LCD_RW_PIN)
#define LCD_RS_HIGH LCD_RS_PORT |= _BV(LCD_RS_PIN)
```

Dzięki temu, aby np. ustawić linię R/W w stan niski wystarczy, że napiszemy:

```
LCD_RW_LOW;
```

Można w ten sposób zwiększyć czytelność kodu. A co z liniami danych? Dla nich też moglibyśmy zdefiniować takie makra. Dzięki temu moglibyśmy korzystać z dowolnych, niezajętych pinów mikrokontrolera. Zwykle jednak nie ma takiej potrzeby i można wykorzystać po prostu cały port (dla trybu 8-bitowego) lub jego połówkę. Upraszcza to kod, bo nie trzeba wystawiać wyświetlaczowi bajtu bit po bicie, tylko po prostu wystawić całą liczbę (lub jej połówkę dla trybu 4-bitowego) na danym porcie. Tak też zrobimy: dla trybu 8-bitowego wykorzystamy cały port, a dla trybu 4-bitowego jego starsze piny (bity).

```
#define LCD_DATA_PORT PORTA
#define LCD_DATA_INPUT PINA
#define LCD_DATA_DDR DDRA
```

W założeniach napisaliśmy, że nasz sterownik ma obsługiwać zarówno tryb 8-, jak i 4-bitowy. Zdefiniujmy więc makro, które będzie oznaczało, że sterownik jest kompilowany dla trybu 4-bitowego. Brak tego makra (usunięcie lub zakomentowanie) będzie oznaczał tryb 8-bitowy.

```
#define BITS4
```

Możemy teraz przejść do pisania funkcji zapisującej dane do wyświetlacza. Będzie to funkcja wykonująca jeden transfer, a więc dla trybu 4-bitowego przesłanie połówki bajtu. Przesyłana może być dana lub komenda, więc funkcja musi wiedzieć, jak ma ustawić linię RS. Funkcja musi więc pobierać co najmniej dwa parametry: wartość do przesłania oraz stan RS. Ponadto może zachodzić konieczność sprawdzenia stanu zajętości modułu, żeby nie wysyłać do niego danych zbyt szybko. Można co prawda stosować opóźnienia, ale bardziej eleganckie jest sprawdzanie flagi zajętości, szczególnie jeśli zdecydowaliśmy się na obsługę odczytu danych z wyświetlacza. Zajętość sprawdza się na początku transferu bajtu. Jeśli więc używamy trybu 4-bitowego i mamy dwa transfery na bajt, to flagę sprawdza się tylko przed pierwszym transferem. Trzecim parametrem naszej funkcji będzie więc wartość określająca konieczność sprawdzenia flagi. Otrzymujemy w ten sposób poniższą deklarację:

```
void lcdWrite(uint8_t data, uint8_t rs, uint8_t checkBusyFlag);
```

Flagę odczytuje się w ten sposób, że wydajemy komendę odczytującą bieżący adres pamięci. Ponieważ adres jest 7-bitowy, najstarszy bit używany jest jako flaga zajętości. Jeśli moduł jest zajęty, wykonujemy opóźnienie i sprawdzamy ponownie. Początek naszej funkcji będzie wyglądał następująco:

```
if (checkBusyFlag)
  while (lcdReadAddress() & 0b10000000)
    _delay_us(20);
```

Funkcji odczytującej adres jeszcze nie

mamy, napiszemy ją później. Gdy w odczytanej wartości występuje jedynka na najstarszym bicie, wykonywane jest 20-mikrosekundowe opóźnienie za pomocą standardowej funkcji opóźniającej. Po sprawdzeniu zajętości modułu możemy ustawić linie R/W oraz RS. Ponieważ wykonujemy zapis, linia R/W będzie w stanie niskim. Natomiast stan linii RS będzie zależał od przekazanego parametru.

```
LCD_RW_LOW;
if (rs) LCD_RS_HIGH; else LCD_RS_LOW;
```

Ze względu na przeprowadzanie zapisu, musimy piny danych ustawić jako wyjścia. W trybie 8-bitowym dla całego portu, w trybie 8-bitowym dla starszej połówki:

```
#ifdef BITS4
 LCD_DATA_DDR |= 0xf0;
#else
 LCD_DATA_DDR = 0xff;
#endif
```

Kolejny krok to wystawienie transmitowanej danej. W trybie 4-bitowym chcemy ustawić tylko 4 piny z całego portu, nie ruszając pozostałych. Wykonywane jest to w dwóch krokach: w pierwszym ustawiane są zera, w drugim jedynki.

```
#ifdef BITS4
 LCD_DATA_PORT &= data | 0x0f; //zerowanie bitów
 LCD_DATA_PORT |= data & 0xf0; //ustawianie bitów
#else
 LCD_DATA_PORT = data;
#endif
```

Gdy dane są już wystawione, wysyłamy je przez wygenerowanie krótkiego impulsu na linii E.

```
LCD_E_HIGH;
_delay_us(1);
LCD_E_LOW;
_delay_us(1);
```

W ten sposób mamy gotową funkcję wykonującą zapis do modułu. Funkcja odczytująca jest bardzo podobna. Różnice polegają na tym, że linia R/W jest w stanie wysokim, piny danych pracują jako wejścia, a dane są z nich odczytywane po narastającym zboczu na linii E.

```
uint8_t lcdRead(uint8_t rs, uint8_t checkBusyFlag) {
 if (checkBusyFlag)
  while (lcdReadAddress() & 0b10000000)
   _delay_us(20);
 LCD_RW_HIGH;
 if (rs) LCD_RS_HIGH; else LCD_RS_LOW;
 #ifdef BITS4
  LCD_DATA_DDR &= 0x0f;
 #else
  LCD_DATA_DDR = 0x00;
 #endif
 LCD_E_HIGH;
 _delay_us(1);
 uint8_t data = LCD_DATA_INPUT;
 LCD_E_LOW;
 _delay_us(1);
 return data;
}
```

Możemy teraz przejść do funkcji wyższego poziomu. Zacznijmy od wspomnianej wcześniej funkcji odczytującej bieżący adres z wyświetlacza. Może ona wyglądać następująco:

```
uint8_t lcdReadAddress() {
 if (switchedTo4BitMode){
  return ((lcdRead(0, 0) & 0xf0) |
(lcdRead(0, 0) >> 4));
 }else{
  return lcdRead(0, 0);
 }
}
```

switchedTo4BitMode jest globalną zmienną informującą o tym, czy wyświetlacz został przełączony w tryb 4-bitowy. Jest to konieczne, ponieważ na początku procesu inicjalizacji działa on w trybie 8-bitowym. Zmienna ma modyfikator static, aby była dostępna tylko w obrębie pliku lcd.c. Jeśli nastąpiło przełączenie w tryb 4-bitowy, odczyt wykonywany jest w dwóch krokach. Zwrócone wartości są następnie sklejane w jeden bajt. Odbywa się to w ten sposób, że po odebraniu starszych bitów, młodsze bity są zerowane. Gdy odbierane są młodsze bity, zwracane są one przez funkcję lcdRead na starszych pozycjach, stąd konieczne jest ich przesunięcie w prawo, na pozycje młodsze. Na koniec obie połówki bajtu są łączone i cały bajt jest zwracany. Przy trybie 8-bitowym wszystko sprowadza się do zwrócenia tego, co zwróciła funkcja lcdRead.

Parametrami wywołania lcdRead są wszędzie zera, gdyż odczytanie adresu wymaga linii RS ustawionej na 0, a sprawdzanie flagi zajętości jest wyłączone. Funkcja odczytująca dane wygląda analogicznie do funkcji odczytującej adres, z tym że linia RS jest w stanie wysokim. Ponadto jest sprawdzanie flagi zajętości przy pierwszym transferze w trybie 4-bitowym, a w trybie 8-bitowym zależnie od wartości zmiennej globalnej checkBusyFlag, która jest ustawiana podczas inicjalizacji. Flagi zajętości nie można bowiem sprawdzać od początku procedury inicjalizacji.

```
uint8_t lcdReadData(){
 if (switchedTo4BitMode){
  return ((lcdRead(1, 1) & 0xf0) | (lcdRead(1, 0) >> 4));
 }else{
  return lcdRead(1, checkBusyFlag);
 }
}
```

Popatrzmy teraz na funkcje zapisu komend i danych – **listing 4.**

Są one skonstruowane analogicznie. Dla trybu 4-bitowego przesyłamy najpierw starszy półbajt, potem młodszy. Przy zapisie danych ustawiamy RS na 1. Pierwszy transfer ma włączone sprawdzanie flagi zajętości, a w trybie 8-bitowym zależnie od zmiennej checkBusyFlag.

```
void lcdWriteCommand(uint8_t command) {
 if (switchedTo4BitMode){
  lcdWrite(command, 0, 1);
  lcdWrite(command << 4, 0, 0);
 }else {
  lcdWrite(command, 0, 1);
 }
}

void lcdWriteData(uint8_t data) {
 if (switchedTo4BitMode){
  lcdWrite(data, 1, 1);
  lcdWrite(data << 4, 1, 0);
 }else {
  lcdWrite(data, 1, checkBusyFlag);
 }
}
```

Listing 4

Mając zdefiniowane podstawowe funkcje do komunikacji z wyświetlaczem, możemy przejść do napisania funkcji służącej do jego inicjalizacji. Inicjalizacja polega na wydawaniu komend, zdefiniujmy więc dla nich makra. Wartości poniższych makr wynikają z tabeli 1.

```
#define LCD_COMMAND_CLEAR 0x01
#define LCD_COMMAND_RETURN_HOME 0x02
#define LCD_COMMAND_ENTRY_MODE_SET 0x04
#define LCD_COMMAND_ON_OFF 0x08
#define LCD_COMMAND_SHIFT 0x10
#define LCD_COMMAND_FUNCTION_SET 0x20
#define LCD_COMMAND_SET_CGRAM_ADDRESS 0x40
#define LCD_COMMAND_SET_DDRAM_ADDRESS 0x80
```

A także makra dla parametrów dla tych komend:

```
#define LCD_PARAM_ENTRY_MODE_SET_SHIFT 0x01
#define LCD_PARAM_ENTRY_MODE_SET_INCREMENT 0x02
#define LCD_PARAM_ON_OFF_BLINK 0x01
#define LCD_PARAM_ON_OFF_CURSOR 0x02
#define LCD_PARAM_ON_OFF_DISPLAY 0x04
#define LCD_PARAM_SHIFT_RIGHT 0x04
#define LCD_PARAM_SHIFT_DISPLAY 0x08
#define LCD_PARAM_FUNCTION_SET_5X10 0x04
#define LCD_PARAM_FUNCTION_SET_2LINES 0x08
#define LCD_PARAM_FUNCTION_SET_8BIT 0x10
```

Funkcja inicjalizująca wyświetlacz będzie wyglądała jak w **listingu 5**.

W ten oto sposób otrzymaliśmy bibliotekę, z której już możemy korzystać do wyświetlania tekstu na wyświetlaczu. Aby ją przetestować, musimy upewnić się, że mamy we właściwościach projektu zdefiniowane makro F_CPU, ponieważ nasza biblioteka korzysta z funkcji opóźniających. W naszym głównym pliku projektu dołączamy plik nagłówkowy biblioteki za pomocą dyrektywy:

```
#include "lcd.h"
```

Teraz w funkcji main możemy wykonać inicjalizację wyświetlacza i np. wysłać do niego jakieś litery do wyświetlenia:

```
int main(void)
{
```

```
 lcdInit();
 lcdWriteData('E');
 lcdWriteData('d');
 lcdWriteData('W');
 while (1) { }
}
```

Jeśli wszystko wykonaliśmy poprawnie, musimy jeszcze połączyć mikrokontroler z wyświetlaczem. Zależnie od wybranego interfejsu (4 lub 8 bitów), wybranego portu dla danych oraz pinów sterujących wpisanych w pliku lcd.h, łączymy odpowiednie piny portów z pinami złącza LCD na płytce testowej. Jeśli używamy podanych na początku przykładowych makr, to wykorzystanych zostanie 7 pinów portu A, 4 na dane i 3 sterujące. Piny złącza LCD są opisane na płytce, ale niestety w pierwszej partii płytek opis został obrócony o 180 stopni. Prawidłowy opis złącza znajduje się na **rysunku 2**.

Gdy już wykonamy połączenia, możemy skompilować projekt i zaprogramować mikrokontroler. Na wyświetlaczu powinien pojawić się napis „EdW".

A czy możemy sobie uprościć życie i wyświetlić napis za jednym zamachem, a nie po literze? Oczywiście tak. Możemy wzbogacić naszą bibliotekę o funkcję, która będzie wyświetlała kolejne znaki z podanej tablicy:

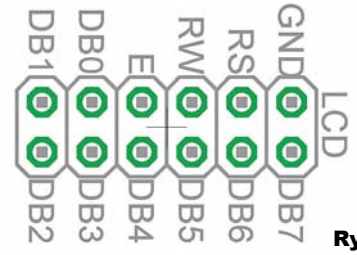
```
void lcdString(char str[]) {
 uint8_t i = 0;
 while (str[i] != 0) {
  lcdWriteData(str[i++]);
 }
}
```

Funkcja ta bierze znaki z podanej tablicy i wyświetla je tak długo, aż natrafi na znak NULL (bajt o wartości 0). Bajt ten jest dodawany automatycznie przez kompilator na końcu łańcucha znaków umieszczonego w cudzysłowie. Wywołanie funkcji jest bardzo proste:

```
lcdString("Elektronika");
```

Listing 5

```
void lcdInit() {
 switchedTo4BitMode = 0;
 checkBusyFlag = 0;
 LCD_E_DDR |= _BV(LCD_E_PIN);
 LCD_RW_DDR |= _BV(LCD_RW_PIN);
 LCD_RS_DDR |= _BV(LCD_RS_PIN);
 _delay_ms(15);
 lcdWriteCommand(LCD_COMMAND_FUNCTION_SET | LCD_PARAM_FUNCTION_SET_8BIT);
 _delay_ms(4.1);
 lcdWriteCommand(LCD_COMMAND_FUNCTION_SET | LCD_PARAM_FUNCTION_SET_8BIT);
 _delay_ms(0.1);
 lcdWriteCommand(LCD_COMMAND_FUNCTION_SET | LCD_PARAM_FUNCTION_SET_8BIT);
 #ifdef BITS4
  _delay_ms(1);
  lcdWriteCommand(LCD_COMMAND_FUNCTION_SET);
  _delay_ms(1);
  switchedTo4BitMode = 1;
  checkBusyFlag = 1;
  lcdWriteCommand(LCD_COMMAND_FUNCTION_SET | LCD_PARAM_FUNCTION_SET_2LINES);
 #else
  checkBusyFlag = 1;
  lcdWriteCommand(LCD_COMMAND_FUNCTION_SET | LCD_PARAM_FUNCTION_SET_8BIT | LCD_PARAM_FUNCTION_SET_2LINES);
 #endif
 lcdWriteCommand(LCD_COMMAND_ON_OFF);
 lcdWriteCommand(LCD_COMMAND_ON_OFF | LCD_PARAM_ON_OFF_DISPLAY);
 lcdWriteCommand(LCD_COMMAND_ENTRY_MODE_SET | LCD_PARAM_ENTRY_MODE_SET_INCREMENT);
 lcdWriteCommand(LCD_COMMAND_CLEAR);
}
```



Rys. 2

Przydałaby się również funkcja, za pomocą której moglibyśmy określić, gdzie ma zostać wyświetlony dany napis. Jak wiemy pozycja znaku na wyświetlaczu zależy od tego, do której komórki pamięci DD RAM go wpiszemy, oraz że adresy dolnego wiersza zaczynają się od 40h. Przed wysłaniem więc znaku/znaków należy podać wyświetlaczowi adres odpowiedniej komórki. Możemy to wykonać prostą funkcją:

```
void lcdGotoXY(uint8_t x, uint8_t y){
  lcdWriteCommand(LCD_COMMAND_SET_DDRAM_ADDRESS | (x + y * 0x40));
}
```

Zakładamy tutaj, że pozycje na wyświetlaczu numerowane są od 0, górny wiersz ma numer 0, a dolny 1. Wyświetlenie więc napisu „Elektronika" w górnym wierszu, a „dla wszystkich" w dolnym będzie wyglądało tak:

```
lcdString("Elektronika");
lcdGotoXY(0, 1);
lcdString("dla wszystkich");
```

Oczywiście można sobie przyjąć inną konwencję numeracji i odpowiednio zmodyfikować wyrażenie obliczające adres.

Materiały do tego numeru EdW zawierają dokumentację HD44780 oraz przykładowy projekt.

Zadania

W tym miejscu nie kończymy jeszcze naszej przygody z wyświetlaczem alfanumerycznym. Materiał z tej lekcji jest już jednak wystarczający do wykonania własnych ćwiczeń. Oto kilka przykładowych:

1. Automatyczne przewijanie napisu
2. Przewijanie napisu za pomocą klawiatury
3. Wpisywanie tekstu lub innych symboli z klawiatury. Klawiatura ma tylko 16 klawiszy, więc mogą to być np. cyfry i wybrane litery lub inne symbole ASCII.

Grzegorz Niemirowski
grzegorz@grzegorz.net