Kurs AVR – lekcja 6

Rozwiązania zadań z ostatniego odcinka

W zadaniu numer 1 mieliśmy wykonać przesuwanie tekstu na wyświetlaczu. Służy do tego funkcja „Cursor or display shift" (makro LCD_COMMAND_SHIFT), której musimy podać, czy przesuwamy kursor, czy tekst, i w którą stronę. Ponieważ przesuwamy tekst, musimy ustawić bit 3. Mamy dla niego stworzone makro LCD_PARAM_SHIFT_DISPLAY. Spróbujmy np. przesuwać w lewo. Nie trzeba więc ustawiać bitu 2. Aby uzyskać ciągłe przesuwanie, komendę przesuwającą trzeba umieścić w pętli i dodać niewielkie opóźnienie (**listing 1**).

Zadanie drugie było analogiczne, ale przesuwanie miało być wywoływane z klawiatury. Wystarczy więc odczytywać stan przycisków, np. S1 i S2 a następnie wydać odpowiednią komendę przesuwającą. Należy tylko pamiętać, że klawiatura będzie podłączona do innego portu niż LCD, np. do portu D. Konieczna jest więc zmiana makr w bibliotece LCD lub modyfikacja funkcji readKeyboard() (**listing 2**).

Zadanie 3 było zachętą do podejścia do tematu wyświetlania tekstu wpisywanego z klawiatury. Najpierw musimy pomyśleć, jakie znaki mają być pod danymi przyciskami. Przypisanie znaków do przycisków da nam mapę, np. taką:

```
uint8_t keyMap[16] = {
 'A', 'B', 'C', 'D',
 'a', 'b', 'c', 'd',
 '1', '2', '3', '4',
 '5', '6', 126, 127
};
```

Oczywiście zamiast liter i cyfr przedstawionych powyżej można użyć innych znaków obsługiwanych przez wyświetlacz. Jeśli znak nie daje się wpisać z klawiatury naszego komputera, zamiast umieszczać go w apostrofach, można bezpośrednio wpisać liczbę będącą kodem ASCII tego znaku. Przykładem są tu kody 126 i 127, które dają symbole strzałek (dla japońskiej wersji wyświetlacza).

Następnie wystarczy odczytywać stan przycisków i w zależności od numeru przycisku wybierać element z mapy klawiatury (listing 3):

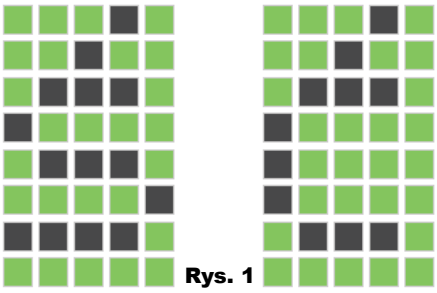
```
int main(void) {
 lcdInit();
 DDRD = 0x0f;
 while(1) {
  uint8_t key = readKeyboard();
  if (key) {
   lcdWriteData(keyMap[key - 1]);
   _delay_ms(200);
  }
 }
}
```

Od numeru klawisza odejmujemy 1, ponieważ klawisze są numerowane od 1, a elementy tablicy od 0.

Definiowanie własnych znaków

W poprzedniej lekcji poznaliśmy możliwości wyświetlacza opartego na sterowniku HD44780, nie omówiliśmy jednak dokładnie tworzenia własnych znaków. Definiujemy je w ten sposób, że w pamięci CG RAM wyświetlacza umieszczamy bajty określające wygląd znaku piksel po pikselu. Znaki wyświetlane są na matrycy 5x8 pikseli, czyli mają 5 kolumn i 8 wierszy. Na każdy wiersz przypada jeden bajt pamięci CG RAM, potrzebujemy więc 8 bajtów na znak. Z każdego bajtu używanych jest 5 młodszych bitów na zdefiniowanie wyglądu danego wiersza. Definiowanie własnego znaku najlepiej zacząć od narysowania go.

Załóżmy, że chcemy wyświetlić napis „Cześć!", więc potrzebujemy liter ś oraz ć. Narysujmy je (rysunek 1).



Rys. 1

Oczywiście nie ma problemu, aby tworzyć różne znaki, niekoniecznie polskie ogonki, np. ikony baterii czy termometru, zależnie od potrzeb. A dysponując odpowiednim zestawem ikon i kolejno je podmieniając możemy łatwo stworzyć animację.

Mając rysunki znaków, możemy w łatwy sposób umieścić je w kodzie naszego programu w postaci tablicy bajtów. Po prostu piksel włączony (czarny) da nam bit o wartości 1, a piksel wyłączony (przezroczysty) da bit 0. W kodzie możemy skorzystać z notacji bitowej, nie trzeba przeliczać bajtów na wartości szesnastkowe lub dziesiętne.

```
uint8_t customCharS[8] = {
 0b00010,
 0b00100,
 0b01110,
 0b10000,
 0b01110,
 0b00001,
 0b11110,
 0b00000
};

uint8_t customCharC[8] = {
 0b00010,
 0b00100,
 0b01110,
 0b10000,
 0b10000,
 0b10001,
 0b01110,
 0b00000
};
```

Gdy mamy już tablice bajtów definiujące nasze znaki, musimy umieścić je w pamięci CG RAM wyświetlacza. Przed zapisem do tej pamięci musimy poinformować wyświetlacz, że wysyłane dane mają trafić do tej właśnie pamięci, a nie tak jak zwykle do DD RAM. Inaczej wysyłane przez nas bajty zostałyby potraktowane jako kody znaków do wyświetlenia a nie jako definicje własnych znaków. Patrząc na tabelę komend z poprzedniego odcinka, widzimy, że potrzebujemy wydać komendę „Set CG RAM address", której sześć młodszych bitów wyznacza adres, od którego począwszy, będziemy zapisywać wygląd naszych znaków. Ponieważ jeszcze żadnych znaków nie

Listing 1

```
int main(void)
{
 lcdInit();
 lcdString("Elektronika dla wszystkich");
 while(1) {
  lcdWriteCommand(LCD_COMMAND_SHIFT | LCD_PARAM_SHIFT_DISPLAY);
  _delay_ms(200);
 }
}
```

```
int main(void)
{
 lcdInit();
 lcdString("Elektronika dla wszystkich");
 DDRD = 0x0f;
 while(1) {
  uint8_t key = readKeyboard();
  if (key) {
   if (key == 1) lcdWriteCommand(LCD_COMMAND_SHIFT | LCD_PARAM_SHIFT_DISPLAY);
   if (key == 2) lcdWriteCommand(LCD_COMMAND_SHIFT | LCD_PARAM_SHIFT_DISPLAY | LCD_PARAM_SHIFT_RIGHT);
   _delay_ms(200);
  }
 }
}
```

Listing 2

definiowaliśmy, możemy zacząć od początku pamięci CG RAM, czyli od adresu 0. Wydajemy więc samą komendę, bez ustawiania dodatkowych bitów. Następnie wysyłamy bajty opisujące znaki, korzystając z pętli, które wyślą nasze tablice.

```
lcdWriteCommand(LCD_COMMAND_SET_CGRAM_ADDRESS);
for (uint8_t i = 0; i < 8; i++) {
 lcdWriteData(customCharS[i]);
}
for (uint8_t i = 0; i < 8; i++) {
 lcdWriteData(customCharC[i]);
}
```

Gdy zdefiniowane przez nas znaki są w pamięci, możemy je wyświetlić. Jak pamiętamy, znaki użytkownika mają kody 0 – 7, są też powtórzone pod kodami 8 – 15. Jeśli wysłaliśmy najpierw literkę ś, a potem ć, to ś będzie pod kodem 0, a ć pod 1. Zanim jednak wyślemy kody naszych literek, musimy poinformować wyświetlacz, że nie chcemy już definiować znaków, tylko je wyświetlić. Innymi słowy, trzeba przestawić wyświetlacz z zapisu do CG RAM na zapis do DD RAM, np. wykorzystując napisaną już wcześniej funkcję lcdGotoXY().

```
lcdGotoXY(0, 0);
lcdWriteData(0);
lcdWriteData(1);
```

Nasze znaki działają poprawnie. A czy da się je wykorzystać w funkcji lcdString() i umieścić w środku tekstu, a nie wyświetlać pojedynczo? Musielibyśmy kody naszych znaków wstawić do łańcucha znaków. Spróbujmy więc umieścić w kodzie taką linijkę (\xn oznacza bajt w postaci szesnastkowej):

```
lcdString("Cze\x0\x1!");
```

Niestety naszym oczom ukazują się tylko pierwsze trzy litery. Co się stało z pozostałymi? W języku C koniec łańcucha znaków wyznaczony jest przez bajt o wartości zero. Gdy umieszczamy tekst w cudzysłowie, definiując łańcuch znaków, kompilator dodaje jeszcze na końcu zero. Korzysta z tego nasza funkcja lcdString(), która wysyła dane do wyświetlacza, dopóki nie trafi na wartość 0. Problem polega na tym, że my bajt o wartości zero wstawiliśmy do środka tekstu, bo taki kod ma literka ś. Dla funkcji lcdString() nasz tekst kończy się więc zaraz za literką e. Jak to rozwiązać? Możemy po prostu literkę ś umieścić w jakimś innym miejscu CG RAM, np. za literką ć, czyli pod adresami 16–23. Wtedy otrzyma ona kod 2. A jeśli potrzebujemy zdefiniować 8 znaków, czyli wykorzystać całą pamięć CG RAM, łącznie z pierwszymi jej 8 bajtami, definiującymi znak o kodzie 0? Możemy wykorzystać tutaj fakt, że znaki użytkownika mają zdublowane kody: każdy z nich ma też kod większy o 8. Czyli nasza literka ś jest nie tylko pod kodem 0, ale też pod kodem 8. Analogicznie ć jest pod kodem 9. Sprawdźmy to:

```
lcdString("Cze\x8\x9!");
```

Tym razem się udało, na wyświetlaczu widać cały tekst. Czy zostało nam jeszcze coś do zrobienia? Na pewno warto by opakować definiowanie znaku w wygodną do użycia funkcję i umieścić ją w naszej bibliotece, aby nie trzeba było tego robić w głównym programie. Funkcja taka pobierałaby tablicę opisującą znak oraz adres, pod którym miałby się on znaleźć w CG RAM:

```
void lcdDefineChar(uint8_t charDefinition[], uint8_t address) {
 lcdWriteCommand(LCD_COMMAND_SET_CGRAM_ADDRESS + address);
 for (uint8_t i = 0; i < 8; i++) {
  lcdWriteData(charDefinition[i]);
 }
}
```

Funkcję tę umieszczamy w naszej bibliotece, czyli powyższą definicję w pliku lcd.c, a nagłówek w lcd.h. Dzięki temu staje się ona dostępna w głównym programie:

```
lcdDefineChar(customCharS, 0);
lcdDefineChar(customCharC, 8);
```

Przy definiowaniu znaku za pomocą lcdDefineChar() podajemy adres w CG RAM, a przy jego wykorzystaniu podajemy jego kod. Ponieważ znak zajmuje 8 bajtów, kod znaku jest 8 razy mniejszy od jego adresu. Przeliczanie nie jest więc trudne. Ale jeśli ktoś chce, może przepisać tak funkcję, aby przyjmowała kod znaku. Obliczanie adresu zostanie w ten sposób schowane wewnątrz biblioteki i pisząc główny program, nie trzeba będzie się przejmować adresami w CG RAM. Wtedy pierwsza linijka funkcji będzie wyglądać następująco:

```
lcdWriteCommand(LCD_COMMAND_SET_CGRAM_ADDRESS + code * 8);
```

Zakładamy tu zmianę nazwy drugiego parametru funkcji z *address* na *code*. Jak wspomniano, oprócz kodów znaków 0–7 wyświetlacz przyjmuje też zduplikowane kody 8–15. Wykorzystaliśmy to przed chwilą aby rozwiązać problem z kodem 0. Aby obsługiwać takie kody, nasza zmodyfikowana funkcja musi przed wykonaniem mnożenia wykonać operację modulo:

```
lcdWriteCommand(LCD_COMMAND_SET_CGRAM_ADDRESS + (code % 8) * 8);
```

Dzięki temu w głównym programie możemy napisać (**listing 4**):

```
lcdDefineChar(customCharS, 8);
lcdDefineChar(customCharC, 9);
```

Tego typu modyfikacje mogą się wydać nieistotnymi szczegółami, w końcu program działa i tak, i tak. Jednak w większych projektach stają się one istotne. Pozwalają utrzymać spójność, porządek i pomagają w uniknięciu pomyłek.

Funkcja sprintf()

W dotychczasowych przykładach wyświetlaliśmy prosty tekst, posiłkując się co najwyżej samodzielnie zdefiniowanymi znakami. Jak jednak wyświetlić np. liczbę? Oczywiście wśród znaków obsługiwanych przez wyświetlacz są cyfry i możemy np. wyświetlić napis „123”. Jak jednak uzyskać w trakcie działania programu odpowiedni łańcuch cyfr, mając wartość liczbową zapisaną w zmiennej? Dla pojedynczej cyfry jest jeszcze łatwo, bo aby uzyskać odpowiednią cyfrę, wystarczy wartość danej zmiennej dodać do kodu ASCII cyfry 0:

```
uint8_t i = 5;
lcdWriteData('0' + i);
```

Zero umieszczone w apostrofach oznacza pojedynczy znak, który w języku C jest tak naprawdę kodem ASCII tego znaku. Cyfra 0 ma kod 48, więc zamiast ‘0’ moglibyśmy napisać po prostu 48. Kolejne cyfry mają kolejne kody, dlatego dodając daną wartość do kodu cyfry 0, otrzymamy cyfrę odpowiadającą tej wartości. W powyższym przykładzie wyświetlona zostanie cyfra 5.

Co jednak z liczbami dwu-, trzycyfrowymi lub większymi? Musielibyśmy obliczyć liczbę jedności, dziesiątek, setek itd. i w ten sposób otrzymać kody kolejnych cyfr liczby. Napisanie programu, który by to wykonywał, nie jest trudne, ale istnieje inny prosty sposób: mamy do dyspozycji funkcję sprintf(), znajdującą się w standardowej bibliotece stdio.h. sprintf() ma następujący nagłówek:

```
int sprintf(char str[], const char format[], ...);
```

Działa ona w ten sposób, że we wskazanym miejscu pamięci (tablica str) umieszcza ciąg znaków będący wynikiem podstawiania wartości do wzorca formatującego (tablica format). Może to brzmieć zawile, więc popatrzmy na przykład użycia:

```
char str[15];
sprintf(str, "Napiecie: %dV", 230);
```

Definiujemy tablicę typu char, która ma 15 elementów (bajtów). To jest obszar, na którym pracujemy. Tablicę tę przekazujemy do funkcji sprintf() jako jej pierwszy parametr. Drugim parametrem jest ciąg znaków, który funkcja sprintf() kopiuje do naszej tablicy. Nie jest to jednak zwykłe kopiowanie. Gdy sprintf() napotka na znak %, wie, że jeden lub więcej kolejnych znaków to specjalne symbole, pod które trzeba podstawić wartości przekaza-

Numer bajtu	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Znak	N	a	p	i	e	c	i	e	:		2	3	0	V	NULL
Wartość bajtu (kod ASCII znaku)	78	97	112	105	101	99	105	101	58	32	50	51	48	86	0

Tabela 1

ne w kolejnych parametrach. W naszym przykładzie %d oznacza, że ma być podstawiona liczba dziesiętna. W rezultacie w tablicy str znajdą się znaki jak pokazuje **tabela 1**.

Ogólnie symbol formatujący ma następującą postać:

```
%[flagi][szerokość][.precyzja][długość]specyfikator
```

Tylko specyfikator jest obowiązkowy. Mówi on o tym, jaki typ danych chcemy wyświetlić. Jak już wiemy, specyfikator d oznacza liczbę dziesiętną. Pełna lista specyfikatorów, np. dla liczb zmiennoprzecinkowych i szesnastkowych, znajduje się w **tabeli 2**. Flagi (**tabela 3**) określają takie właściwości, jak wyrównanie czy wstawianie zer wiodących. Szerokość oznacza minimalną liczbę znaków do wyświetlenia. Jeśli liczba ma mniej cyfr niż podana szerokość, zostaną dodane spacje. Jako szerokość może być podana gwiazdka (*). Wtedy szerokość wyznaczana jest w kolejnym parametrze przekazywanym do sprintf(). Dzięki temu szerokość da się ustawić dynamicznie, a nie jest określona z góry na etapie pisania programu. Precyzja to liczba cyfr po kropce dziesiętnej (nie ma możliwości stosowania przecinka jako symbolu dziesiętnego, jak to jest przyjęte w Polsce). Tutaj też można podać gwiazdkę i liczbę cyfr po przecinku przekazać przez parametr. W miejscu trzeciego parametru funkcji sprintf() znajduje się trzykropek. Oznacza to, że jest to funkcja o zmiennej liczbie parametrów i możemy do niej przekazać kilka wartości do podstawienia.

Jak zostało wspomniane, pierwszym parametrem sprintf() jest tablica, w której umieszczany jest wynik działania funkcji. Jak duża powinna być ta tablica? Musi się w niej zmieścić cały napis wraz z bajtem o wartości zero (znakiem NULL) oznaczającym koniec tego napisu. To na programiście spoczywa odpowiedzialność za zapewnienie, że przekazywana tablica będzie wystarczająco duża. Funkcja sprintf() „nie zna" rozmiaru przekazywanej tablicy. Jeśli będzie ona za mała, sprintf() nadpisze obszar pamięci znajdujący się za wskazaną tablicą, co może doprowadzić do nadpisania innych zmiennych lub niekontrolowanych przeskoków.

Bezpieczniejsza w użyciu jest funkcja snprintf(), która dodatkowo pobiera rozmiar przekazywanej tablicy w drugim parametrze:

```
int snprintf(char str[],  size_t  n, const char * format, … );
```

Rozmiar tablicy jest znany w czasie kompilacji, da się więc go pobrać operatorem sizeof i przekazać do snprintf():

```
snprintf(str, sizeof(str), "Napiecie: %dV", 230);
```

Obie funkcje zwracają wartość typu int oznaczającą liczbę bajtów, które zostały przez nie zapisane do tablicy.

Uwaga! W przypadku użycia specyfikatora zmiennoprzecinkowego, np. %f lub %g, konieczne jest we właściwościach projektu (Toolchain -> AVR/GNU Linker -> Miscellaneous) dodanie następujących flag linkera: -Wl,-u,vfprintf -lprintf_flt

Tabela 2

specyfikator	opis	przykład
d lub i	Liczba dziesiętna, całkowita, ze znakiem	392
u	Liczba dziesiętna, całkowita, bez znaku	7235
o	Liczba ósemkowa bez znaku	610
x	Liczba szesnastkowa, całkowita, bez znaku	7fa
X	Liczba szesnastkowa, całkowita, bez znaku	7FA
f	Liczba dziesiętna, zmiennoprzecinkowa	123.45
F	Liczba dziesiętna, zmiennoprzecinkowa	123.45
e	Notacja naukowa	1.23e+4
E	Notacja naukowa (wielka litera E)	1.23E+4
g	Najkrótsza notacja spośród %e i %f	123.45
G	Najkrótsza notacja spośród %E i %F	123.45
a	Liczba szesnastkowa, zmiennoprzecinkowa	-0xc.90fep-2
A	Liczba szesnastkowa, zmiennoprzecinkowa	-0XC.90FEP-2
c	Znak	e
s	Ciąg znaków	Elektronika
p	Adres wskaźnikowy	b12D0F9B3
n	Nic nie jest wypisywane. Pod adresem wskazywanym przez argument zostanie wpisana liczba dotychczas wypisanych znaków	
%	Znak % (znak procent wyświetlamy za pomocą %%)	%

Funkcja printf()

W kursie języka C poznaliśmy funkcję printf() służącą do wyświetlania tekstu na ekranie. Czy jest możliwe, aby wykorzystać ją do wyświetlania tekstu na naszym LCD? Byłaby prostsza w użyciu, bo nie musielibyśmy się martwić tablicą roboczą. printf() kieruje generowany tekst na tzw. standardowe wyjście, które jest jednym z kilku strumieni dostępnych w systemie operacyjnym. W naszym przypadku nie ma systemu operacyjnego, ale możemy zarządzać strumieniami dostarczanymi przez biblioteki AVR-GCC. Kluczowa jest tu funkcja fdevopen() służąca do otwarcia strumienia dla danego urządzenia. Jej nagłówek jest następujący:

```
FILE* fdevopen(int(*)(char, FILE *)put, int(*)(FILE *)get);
```

Może on wyglądać niezrozumiale, ale idea jest prosta: musimy dostarczyć wskaźniki na dwie funkcje: put() i get(), które będą zapisywały i odczytywały po jednym znaku z danego urządzenia. Aby działała nam funkcja printf(), potrzebujemy funkcji zapisującej, czyli put. Ma ona przyjmować dwa parametry: znak oraz wskaźnik na typ FILE. Wskaźnik ten nie będzie przez nas wykorzystywany. Z nagłówka funkcji put () widzimy, że ma ona zwracać typ int. Konkretnie ma być to wartość 0 przy prawidłowym zapisie i różna od 0, gdy zapis się nie powiedzie. Nasza funkcja lcdWriteData() nie sprawdza błędów, więc put() będzie prostym opakowaniem dla lcdWriteData() zwracającym zawsze zero:

```
int put(char c, FILE * file) {
 lcdWriteData(c);
 return 0;
}
```

Możemy teraz zainicjalizować strumień:

```
devopen(&put, NULL);
```

Pierwszym parametrem jest nasza funkcja put. Nie piszemy za nią nawiasów, bo nie chcemy jej wykonać, a jedynie przekazać jej adres do funkcji devopen(). Podkreśla to dodatkowo operator adresu &. Jako wskaźnik na funkcję get() podajemy NULL, liczbowo 0, czyli pusty wskaźnik. Otrzymujemy strumień z opcją zapisu, skojarzony z naszą biblioteką LCD. A jak go podpiąć pod printf()? Nic przecież w powyższej linijce nie robimy z wartością zwróconą przez fdevopen(). Otóż działa to bardzo prosto. Pierwszy strumień otwarty do zapisu staje się standardowym wyjściem (stdout) oraz standardowym wyjściem błędów (stderr).

flaga	opis
-	Wyrównanie do lewej w ramach zadanej szerokości
+	Wyświetla + także przed liczbami dodatnimi
spacja	Spacja, jeśli brak znaku (wyrównanie liczb ze znakiem i bez)
#	Dla liczb ósemkowych lub szesnastkowych dodaje odpowiednio 0 lub 0x (0X). Dla notacji z przecinkiem dodaje kropkę dziesiętną, nawet jeśli nie ma części dziesiętnej
0	Wyrównuje zerami do lewej zamiast spacjami

Tabela 3

Z kolei pierwszy otwarty strumień odczytu staje się standardowym wejściem (stdin). A więc nic już nie musimy robić, możemy korzystać z printf() i tekst będzie się wyświetlał na LCD.

Dla wygody funkcję put() oraz przekierowanie strumienia umieśćmy w naszej bibliotece. Definicja (treść) funkcji put() trafi więc do pliku lcd.c. W lcd.h umieścimy nagłówek tej funkcji oraz dyrektywę #include dołączającą bibliotekę stdio.h, aby przy przetwarzaniu pliku lcd.h kompilator znał typ FILE. Potrzebujemy jeszcze jakoś opakować wywołanie devopen(). Stwórzmy więc funkcję o nazwie np. lcdInitPrintf():

```
void lcdInitPrintf() {
 fdevopen(&put, NULL);
}
```

Umieszczamy ją w lcd.c, a jej nagłówek w lcd.h. Teraz w naszym głównym programie możemy wywołać lcdInitPrintf() i następnie korzystać z printf() (**listing 5**):

```
int main(void) {
 lcdInit();
 lcdInitPrintf();
 printf("Elektronika");
 while(1) {}
}
```

Sekundnik

Wiemy już, jak odmierzać czas, umiemy też wyświetlać dane na LCD. Nie ma zatem przeszkód, żeby stworzyć prosty zegar. Dojdziemy do tego jednak etapami, zaczynając od najprostszych programów. Na początek weźmy odliczanie sekund mijających od uruchomienia programu. Jak odmierzać sekundy? Dobrze nadawałby się tutaj 16-bitowy Timer1, którego możliwości były przedstawione w części 4. kursu. Zaznajomiliśmy się wtedy również z kilkoma rzeczami, które trzeba wziąć pod uwagę przy odmierzaniu czasu.

Załóżmy, że korzystamy z wewnętrznego generatora RC do taktowania naszego mikrokontrolera i jest on ustawiony na domyślną częstotliwość 1MHz. Timer1 może być taktowany sygnałem zegarowym podzielonym przez 1, 8, 64, 256 lub 1024. Przez które z tych liczb da się podzielić 1000000 (Hz), aby uzyskać liczbę całkowitą? Będą to 1, 8 i 64. Czy możemy wybrać dowolny z tych dzielników? Licznik jest 16-bitowy, więc po podziale 1MHz musimy otrzymać liczbę mieszczącą się w przedziale do 65 535. Warunek ten spełnia tylko dzielnik 64 dający liczbę 15625. Wiemy więc już, jak skonfigurować Timer1: musi być taktowany sygnałem zegarowym podzielonym przez 64 i resetować się po doliczeniu do 15625. Resetowanie po osiągnięciu tej wartości możemy zrealizować funkcją CTC. Do rejestru OCR1A wpisujemy liczbę 15624, ponieważ licznik nie resetuje się w momencie osiągnięcia wpisanej liczby, tylko dopiero w następnym cyklu. Nasz przykładowy kod będzie wyglądał jak na **listingu 6**.

```
int main(void) {
 lcdInit();
 lcdInitPrintf();
 TCCR1B = _BV(WGM12) | _BV(CS11) | _BV(CS10);
 OCR1A = 15624;
 uint32_t s = 0;
 lcdInit();
 while(1) {
  lcdGotoXY(0, 0);
  printf("%d", s);
  while(!(TIFR & _BV(OCF1A)));
  TIFR |= _BV(OCF1A);
  s++;
 }
}
```

Listing 6

Nie jest on skomplikowany, korzystamy z elementów omówionych wcześniej: inicjalizacja wyświetlacza, konfiguracja trybu pracy Timer1 (CTC, podział zegara przez 64), inicjalizacja zmiennej liczącej sekundy, a w pętli głównej: ustawienie kursora na początku wyświetlacza, wyświetlenie liczby, oczekiwanie na osiągnięcie przez licznik wartości CTC (15625), wyczyszczenie flagi i wreszcie zwiększenie zmiennej sekund.

Rozpatrzmy odwrotne działanie programu: odliczanie sekund od zadanej wartości do zera. W powyższym przykładzie wystarczy zainicjalizować zmienną s odpowiednią wartością, a inkrementację zmiennej zastąpić dekrementacją. Wszystko OK, ale taki program będzie mało praktyczny. Przydałoby się, żeby można było wpisywać wartość początkową z klawiatury oraz żeby była jakaś sygnalizacja dojścia do zera. Teraz zmienna się „przekręca" i wyświetlane są wartości ujemne. Uważni Czytelnicy zapytają zapewne, dlaczego otrzymujemy liczby ujemne, skoro zadeklarowaliśmy zmienną s jako bez znaku (unsigned). Otóż funkcja printf() dla symbolu %d traktuje przekazany parametr jako int, który w przypadku naszego kompilatora AVR-GCC jest równoważny typowi int16_t, czyli 16-bitowemu typowi ze znakiem. Nasza zmienna s jest do tego typu konwertowana. Mamy nawet o tym wyświetlane ostrzeżenie w okienku rezultatów kompilacji:

Warning: format '%d' expects argument of type 'int', but argument 2 has type 'uint32_t' [-Wformat=]

Zadeklarujmy więc zmienną s jako po prostu int. Jeśli jednak nie pasuje nam oferowany przez ten typ zakres wartości (-32768 – 32767 dla AVR-GCC), możemy symbol %d zastąpić symbolem %u, który oznacza również liczbę dziesiętną, ale bez znaku (nieujemną). Wtedy po dojściu do zera otrzymamy 65535, 65534 itd. Jeśli nadal jednak chcemy mieć pełny zakres oferowany przez typ uint32_t, to musimy użyć symbolu %lu (long unsigned). Wówczas bez przeszkód wyświetlimy liczby od 0 do 4294967295, oczywiście deklarując zmienną s jako uint32_t.

Jest jeszcze jeden problem: jeśli wyświetlona jest liczba 10, a następnie wyświetlimy 9, to na wyświetlaczu mamy 90, bo wyświetlana jest po prostu dziewiątka i zero nie jest kasowane. Rozwiązanie już poznaliśmy: przy symbolu można podać długość, np. %3d (liczba dziesiętna trzycyfrowa). Dzięki temu liczba 10 będzie mieć postać [spacja]10, a liczba 9 będzie mieć postać [spacja][spacja]9.

Wpisywanie liczby z klawiatury

Wracamy do pytania: jak możemy zainicjalizować odliczanie sekund? Mamy opanowane odczytywanie stanu klawiszy, potrzebujemy móc wpisać kilkucyfrową liczbę. Żeby to zrobić, potrzebne jest zdefiniowanie, jak dokładnie przebiegać będzie wpisywanie liczby. Na pewno trzeba poczekać na wciśnięcie przycisku. Gdy ono nastąpi, trzeba zapamiętać jego numer. Następnie trzeba poczekać na zwolnienie przycisku. W tym miejscu algorytm można powtórzyć, aby obsłużyć kolejne wciśnięcie przycisku. Jednak użytkownik musi jakoś zakończyć wpisywanie liczby. Jeden z klawiszy będzie więc pełnił funkcję klawisza Enter i jego wciśnięcie będzie oznaczało koniec wpisywania.

Przydałaby się tutaj funkcja czekająca na wciśnięcie klawisza i zwracająca jego numer. Nasza funkcja readKeyboard() zwraca bowiem informację o stanie klawiatury w danej chwili i nie jest zbyt wygodna. Napiszmy więc taką funkcję:

```
uint8_t getKey() {
 uint8_t key = 0;
 while (!(key = readKeyboard()));
 while (readKeyboard());
 return key;
}
```

Pierwsza pętla while może się wydać dziwna, dlaczego jest tam pojedynczy znak równości? Nie chcemy jednak sprawdzać, czy zmienna key jest równa wartości zwracanej przez funkcję readKeyboard(),

```
int readNumber() {
 char input[6];
 uint8_t inputIndex = 0;
 while(inputIndex < (sizeof(input) - 1)) {
  uint8_t key = 0;
  key = getKey();
  if (key < 10) input[inputIndex] = '0' + key;
  if (key == 10) input[inputIndex] = '0';
  if (key == 16) {
   break;
  }
  inputIndex++;
 }
 input[inputIndex] = 0;
 return atoi(input);
}
```

Listing 7

```
int main(void)
{
 lcdInit();
 lcdInitPrintf();
 DDRD = 0x0f;
 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--;
 }
}
```

Listing 8

więc nie użyjemy podwójnego znaku równości. Jak pamiętamy, pojedynczy znak równości oznacza przypisane. Wartość zwrócona przez readKeyboard() jest podstawiana do zmiennej key, a następnie wartość ta po negacji operatorem wykrzyknika jest sprawdzana przez pętlę while. Możemy powiedzieć, że przypisanie zwraca wartość. Pętla while wykonuje się tak długo, jak długo readKeyboard() zwraca zero. Przedstawiona konstrukcja jest równoważna poniższej pętli:

```
do {
 key = readKeyboard();
} while (!key);
```

W kolejnej pętli czekamy na zwolnienie klawisza. Działanie funkcji getKey() możemy przetestować np. w następujący sposób:

```
char k[3];
k[0] = '0' + getKey();
k[1] = '0' + getKey();
k[2] = 0;
lcdString(k);
```

Zdefiniowany został łańcuch znaków, do którego da się wpisać dwa znaki. Trzeci element to oczywiście zero kończące łańcuch. Po wpisaniu cyfr łańcuch jest wyświetlany na LCD. A po co jest dodawanie zer? Funkcja getKey() zwraca numer klawisza, czyli liczby od 1 do 16. Aby otrzymać znaki/cyfry, do numeru klawisza trzeba dodać kod ASCII cyfry zero. Czyli numer klawisza staje się przesunięciem względem znaku zero.

Uwaga: w tym momencie korzystamy zarówno z klawiatury, jak i wyświetlacza. Muszą one więc działać na różnych portach, nie możemy podłączyć obu do jednego portu. Jeśli LCD mamy podłączony do portu A, to klawiaturę możemy podłączyć do portu D. Oczywiście wymaga to modyfikacji funkcji readKeyboard(), w tym także linijki ustawiającej cztery piny jej portu jako wyjścia:

```
DDRD = 0x0f;
```

W powyższym przykładzie wczytywaliśmy dwie cyfry. Jak wczytać inną liczbę cyfr? Wygodnie by było, gdyby jeden z klawiszy pełnił funkcję klawisza Enter. Jego wciśnięcie oznaczałoby koniec wpisywania. Musimy więc zapamiętywać wciśnięte klawisze tak długo, jak nie zostanie wciśnięty ten klawisz.

Obsługując klawiaturę w przedstawiany sposób, otrzymamy ciąg znaków. Co zrobić, aby otrzymać liczbę? Z pomocą przychodzi nam biblioteczna funkcja atoi(), która właśnie zwraca wartość liczbową na podstawie podanego ciągu znaków. Deklaracja atoi() znajduje się w bibliotece stdlib.h, musimy więc dołączyć ją do naszego programu za pomocą #include.

Mamy już wszystko, co potrzebne, aby napisać funkcję wczytującą liczbę z klawiatury (**listing 7**):

Ponieważ atoi() zwraca typ int, tak samo zwraca go nasza funkcja. Powoduje to też, że maksymalną obsługiwaną liczbą jest 32767, a więc pięciocyfrowa. Deklarujemy więc sześcioelementową tablicę znaków o nazwie input. Bieżący element tej tablicy będzie wskazywany przez zmienną inputIndex. Dopóki zmienna ta jest w zakresie 0–4, czyli jest jeszcze miejsce w tablicy, odczytujemy klawisz z klawiatury. Dla klawiszy S1–S9 zamieniamy numer klawisza na cyfrę. Klawisz S10 pełni funkcję zera, a klawisz S16 funkcję Entera. Gdy wczytywanie znaków zostanie zakończone, czy to Enterem, czy na skutek braku miejsca, wpisujemy końcowe zero i zwracamy wartość liczbową za pomocą atoi().

Gdy mamy już gotową funkcję readNumber(), możemy zmodyfikować nasz główny program tak, aby zaczynał liczenie od liczby wpisanej przez użytkownika (**listing 8**):

Na początku funkcji main() inicjalizujemy wyświetlacz i piny klawiatury. Następnie czekamy na wciśnięcie przez użytkownika klawiszy i potwierdzenie klawiszem S16 (Enter). Gdy zostanie to wykonane, inicjalizowany jest Timer1 w trybie CTC. W głównej pętli ustawiamy kursor na początku ekranu, wyświetlamy bieżący stan licznika sekund i czekamy na upłynięcie sekundy. Gdy to nastąpi, licznik sekund jest zmniejszany o 1 i pętla wykonuje się od początku.

Ponieważ mamy już trzy funkcje do obsługi klawiatury, warto byłoby umieścić je w oddzielnym pliku, np. keyb.c i stworzyć bibliotekę taką jak dla LCD. Pozwoli to zachować większy porządek w głównym pliku programu. Dodajmy więc plik keyb.c do projektu i umieśćmy w nim funkcje readKeyboard(), getKey() i readNumber(). Stwórzmy też plik keyb.h i umieśćmy w nim nagłówki tych funkcji. Na początku keyb.c zawrzemy też dyrektywy dołączające bibliotekę stdlib.h dla atoi() oraz util/delay.h dla _delay_us(). Aby nasza biblioteka była wygodniejsza w użyciu, powinna być konfigurowalna podobnie jak biblioteka LCD. Dodajmy więc definicje portów dla funkcji readKeyboard():

```
#define KEYB_PORT PORTD
#define KEYB_PIN PIND
#define KEYB_DDR DDRD
```

Makro KEYB_DDR wykorzystamy w funkcji inicjalizującej klawiaturę:

```
void keybInit() {
 KEYB_DDR = 0x0f;
}
```

Ta funkcja to tylko jedna linijka, ale pomoże utrzymać porządek. Jej nagłówek także umieszczamy w pliku keyb.h, a wywołanie możemy umieścić w naszym programie zamiast bezpośredniego odwołania do rejestru DDRn.

W materiałach dodatkowych do tego numeru EdW znajduje się projekt zawierający kod odliczający sekundy oraz biblioteki klawiatury i wyświetlacza.

Zadania

Jak zwykle zachęcam do samodzielnych eksperymentów. Definiowanie własnych znaków, wczytywanie i wyświetlanie liczb czy odmierzanie czasu dają spore pole manewru w realizacji różnych funkcji. Inspiracją mogą być poniższe zagadnienia.

1. W naszym programie brakuje sygnalizacji upłynięcia zadanego czasu. Jak zrealizować zatrzymanie odliczania i sygnalizację, np. LED-em?
2. Co zrobić, aby cyfry były widoczne podczas wpisywania?
3. Jak dodać klawisz Backspace?

Grzegorz Niemirowski
grzegorz@grzegorz.net