Książka przeznaczona jest dla elektroników i hobbystów, którzy chcą szybko, w oparciu o interesujące przykłady, poznać język C przeznaczony dla mikrokontrolerów AVR i nauczyć się pisać dla nich programy. Jest to język wysokiego poziomu o nieograniczonych możliwościach, pozwala również łatwo i wygodnie dokonywać połączeń z językiem maszynowym asembler. W sposób przystępny opisana została także architektura oraz możliwości samych mikrokontrolerów AVR wchodzących w skład dwóch rodzin: ATmega i ATtiny. Prezentowany materiał podzielony jest na trzy części. Pierwsza obejmuje zagadnienia związane z budową mikrokontrolerów, druga to wykład na temat podstaw samego języka, a trzecia zawiera szereg ćwiczeń wraz z kodami źródłowymi, komentarzami i bogatymi opisami.

Opracowanie graficzne: Mirosław Kardaś
Redakcja: Małgorzata Koczańska

© Copyright by Wydawnictwo Atnel
Szczecin 2011

ISBN 978-83-931797-0-1

Wydawnictwo ATNEL
ul. Jasna 15/38
70-777 Szczecin

fax: 91 4635 683
http://www.atnel.pl
e-mail: biuro@atnel.pl

Wydanie I

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli. Autor oraz wydawnictwo Atnel dołożyli wszelkich starań, by publikowane tu informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz wydawnictwo Atnel nie ponoszą także żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce. Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentów niniejszej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie wszelkiego rodzaju kopii na dowolnych nośnikach powoduje naruszenie praw autorskich niniejszej publikacji.

Spis treści

Przedmowa **7**

1 Wstęp **8**

2 Zaczynamy **9**

2.1 Pierwszy „pusty” program w C 9

2.2 Od programu do procesora 10

2.2.1 Kompilacja 10

2.2.2 Środowisko 12

2.2.3 Programator sprzętowy 13

2.2.4 Programowanie procesora 14

2.2.5 Uruchamiamy AVR Studio 15

2.2.6 Platforma sprzętowa 23

3 Procesory AVR **25**

3.1 Informacje ogólne 25

3.2 Programowanie ISP 28

3.3 Sposoby taktowania procesorów 29

3.3.1 Wewnętrzny oscylator 30

3.3.2 Zewnętrzny rezonator kwarcowy 30

3.3.3 Zewnętrzny oscylator RC 31

3.3.4 Zewnętrzny generator 32

3.4 Zagadnienia związane z zasilaniem 32

3.5 Układ resetu mikrokontrolera AVR 34

3.6 Wewnętrzne moduły procesorów AVR 34

3.6.1 Pamięć FLASH, RAM, EEPROM 34

3.6.2 Przerwania 38

3.6.3 Timery sprzętowe 40

3.6.3.1 Podstawowe tryby pracy Timerów 42

3.6.3.1.1 Tryb zwykłego LICZNIKA 42

3.6.3.1.2 Tryb CTC – jeden z najważniejszych 44

3.6.3.1.3 Tryb PWM 45

3.6.4 Przetwornik ADC 48

3.6.5 Moduł komparatora analogowego 50

3.6.6 Moduł UART/USART, (czyli RS232) 51

3.6.7 Moduł SPI 52

3.6.8 Moduł TWI, (czyli I2C) 52

3.6.9 Watchdog 53

3.6.10 Tryby oszczędzania energii 53

3.6.11 FUSE BITS (ustawienia konfiguracji AVR) 54

3.6.12 LOCK BITS (zabezpieczenia AVR) 55

3.6.13 Bootloader – niesamowite możliwości . . . 56

4 Podstawy języka C . . . 58

4.1 Zagadnienia ogólne . . . 58

4.1.1 Komentarze . . . 58

4.1.2 Definicja a deklaracja . . . 59

4.1.3 Wyrażenia logiczne (warunki) . . . 60

4.2 Najważniejsze instrukcje . . . 60

4.2.1 Instrukcja warunkowa If , else . . . 60

4.2.2 Pętla while . . . 63

4.2.3 Pętla do..while . . . 64

4.2.4 Pętla for . . . 64

4.2.5 Instrukcja break . . . 66

4.2.6 Instrukcja switch . . . 66

4.2.7 Instrukcja continue . . . 68

4.2.8 Nawiasy klamrowe . . . 69

4.2.9 Instrukcja goto . . . 69

4.3 Typy . . . 70

4.3.1 Systematyka typów języka C . . . 71

4.3.1.1 Typy złożone . . . 74

4.3.1.2 Zakres widoczności zmiennych . . . 76

4.3.1.3 Typ void . . . 77

4.3.1.4 Specyfikator const . . . 78

4.3.1.5 Specyfikator volatile . . . 79

4.3.1.6 Specyfikator register . . . 80

4.3.1.7 Instrukcja Typedef . . . 80

4.3.1.8 Typy wyliczeniowe enum . . . 82

4.3.2 Stałe w języku C . . . 85

4.3.2.1 Stałe jako liczby całkowite . . . 85

4.3.2.2 Stałe jako liczby zmiennoprzecinkowe . . . 86

4.3.2.3 Stałe znakowe . . . 86

4.3.2.4 Stałe tekstowe, stringi . . . 88

4.4 Operatory . . . 89

4.4.1 Arytmetyczne . . . 89

4.4.1.1 Modulo, czyli % . . . 89

4.4.1.2 Inkrementacja i dekrementacja ++ -- . . . 91

4.4.1.3 Operator przypisania = . . . 92

4.4.2 Operatory Logiczne . . . 93

4.4.2.1 Operatory relacji . . . 93

4.4.2.2 Suma || oraz iloczyn && logiczny . . . 94

4.4.2.3 Negacja – wykrzyknik ! . . . 95

4.4.2.4 Operatory bitowe . . . 95

4.4.3 Pozostałe operatory przypisania . . . 102
4.4.4 Operator pobierania adresu & . . . 102
4.4.5 Wyrażenie warunkowe ? : . . . 103
4.4.6 Operator sizeof() . . . 104
4.4.7 Priorytety operatorów . . . 105
4.4.8 Operatory rzutowania . . . 106
4.5 Funkcje *** . . . 107
4.5.1 Wynik działania funkcji – jak to działa? . . . 110
4.5.2 Stos – ujarzmianie "potwora" . . . 112
4.5.3 Przekazywanie argumentów przez wartość . . . 114
4.5.4 Funkcje typu inline . . . 116
4.5.5 Zakresy widoczności nazw . . . 123
4.5.5.1 Zakres globalny . . . 123
4.5.5.2 Zakres lokalny i zmienne automatyczne . . . 123
4.5.5.3 Zmienne i funkcje statyczne . . . 124
4.5.6 Funkcje w różnych plikach projektu . . . 126
4.6 Preprocesor . . . 132
4.6.1 Dyrektywa #define . . . 132
4.6.2 Makrodefinicje . . . 134
4.6.3 Dyrektywa #undef. . . . 135
4.6.4 Operator ## - sklejanie nazw . . . 136
4.6.5 Operator zamiany na string # . . . 136
4.6.6 Dyrektywy kompilacji warunkowej . . . 137
4.6.7 Dyrektywy #ifdef oraz #ifndef . . . 139
4.6.8 Dyrektywy #error i pozostałe . . . 140
4.6.9 Dyrektywa #include . . . 140
4.7 Tablice . . . 141
4.7.1 Tablice wielowymiarowe . . . 144
4.7.2 Tablica jako argument funkcji . . . 145
4.7.3 Tablice znakowe . . . 147
4.8 Wskaźniki . . . 153
4.9 Struktury, unie, pola bitowe . . . 164
4.9.1 Struktury . . . 164
4.9.2 Unie . . . 167
4.9.3 Połączenie struktury z unią . . . 168
4.9.4 Pola bitowe . . . 171
5 Warsztaty – zajęcia praktyczne . . . 173
5.1 Przygotowanie procesora do pracy . . . 173
5.2 Migocząca dioda LED . . . 174
5.3 Obsługa klawiszy typu micro-switch . . . 177
5.4 Multipleksowanie LED - przerwania . . . 182

5.5 Wyświetlacz LCD (HD44780) ... 202
5.6 Sterowanie PWM (kolorowa dioda RGB) ... 223
5.7 Pomiar napięcia za pomocą ADC ... 235
5.7.1 Klawiatura analogowa ... 246
5.7.2 Różnicowy pomiar napięcia - amperomierz ... 246
5.8 Komunikacja RS232 / RS485 ... 257
5.8.1 Inicjalizacja, kalibracja ... 257
5.8.2 UART, przerwania, bufor cykliczny ... 266
5.9 Odczyt-zapis magistrali I2C (RTC, EEPROM) ... 277
5.9.1 RTC – sprzętowa obsługa I2C ... 278
5.9.2 Programowa implementacja I2C ... 285
5.9.3 EEPROM – I2C ... 289
5.10 Moduł SPI ... 291
5.10.1 Sprzętowa obsługa SPI ... 291
5.10.2 Programowa obsługa SPI ... 297
5.11 Magistrala 1Wire ... 299
5.12 Odbiór kodów RC5 w podczerwieni ... 307
5.13 Sterowanie silnikami DC ... 316
5.14 Silnik krokowy unipolarny ... 320
5.15 Silnik krokowy bipolarny ... 326
5.16 Odczyt/zapis kart pamięci SD (FAT) ... 330
5.16.1 FatFS ... 333
5.16.2 PetitFS ... 348
6 FuseBity – MkAvrCalculator ... 356
6.16.1 Fusebity, Lockbity ... 356
6.16.2 MkAvrCalculator ... 360
7 Bootloader ... 368
8 Projekty ... 371
8.1 Pilot na podczerwień ... 371
8.2 Moduł Bluetooth (BTM-112/222) ... 379
8.3 Ściemniacz – płynna regulacja mocy 230V ... 384
8.4 Wstęp do systemów czasu rzeczywistego ... 395
8.5 Obsługa stosu AVR - TCP/IP ... 417
8.5.1 Karta sieciowa ethernet – ENC28J60 ... 419
8.5.2 Serwer http ... 422
8.5.3 Sterownik urządzeń – protokół UDP ... 430
8.6 Programator USBASP ... 454
9 Środowisko ECLIPSE ... 455

4 Podstawy języka C

Wreszcie dotarliśmy do miejsca, gdzie będzie można poznać więcej informacji na temat samego języka C. Podobnie jak w przypadku omawiania podstawowych zagadnień dotyczących całej rodziny mikrokontrolerów, teraz będę musiał omówić składnię języka.

4.1 Zagadnienia ogólne

W języku C stosujemy tzw. „wolny format" jeśli chodzi o pisanie kodu. Nie obowiązują tu reguły jak w innych językach, gdzie trzeba się ograniczać do pisania rozkazów w jednej linii. Nie ma tu żadnych przymusów. Wszystko, co chcemy zapisać, może się znaleźć w każdym miejscu linii, a nawet można to samo rozpisać na kilka linijek. Związane jest to z tym, że koniec instrukcji, jaką wydajemy, jest określony przez średnik, który stawiamy na końcu, a nie przez to, że kończy się linia programu.

Wewnątrz instrukcji może znajdować się dowolna ilość tzw. białych znaków, do których zaliczamy spacje czy tabulatory. Są one ignorowane przez kompilator. Z tego względu nie ma różnicy w tym, jak zapiszemy poniższą linię - możemy to zrobić tak:

```
int main(void) {return 0;}
```

lub tak:

```
int main(void)
{
	// od tego miejsca zaczyna się start programu.
	/*
	komentarze
	*/
	return		0	; // koniec programu
}
```

4.1.1 Komentarze

Białe znaki są ignorowane przez kompilator, służą one jedynie programiście. Słyszałeś zapewne przy okazji pisania kodów programu o tzw. „wcięciach". Dobrze napisany kod jest wtedy, gdy ma stosowane wcięcia. Bez nich kod staje się mało czytelny i bardzo ciężko wrócić do jego analizy po dłuższym czasie.

Zauważyłeś powyżej w jednym z przykładów dwie charakterystyczne linie, w których widać tzw. komentarze. To opisy, które można wstawić do kodu w celu zwiększenia czytelności programowanych zagadnień. Jeśli w dowolnym miejscu linii kompilator napotka dwa znaki `//` następujące po sobie, to ignoruje wszystkie kolejne aż do końca tej linii. Inna forma do oznaczania całego bloku linii, w których chcemy umieścić opisy, może być zawarta pomiędzy dwoma znacznikami, gdzie jeden rozpoczyna blok `/*` natomiast drugi `*/` kończy taki blok.

Zapamiętaj, że komentarze w języku C są bardzo istotnym elementem. Program napisany bez żadnych komentarzy czy krótkich chociaż objaśnień, nie jest napisany w dobrym stylu programistycznym.

Stosuj komentarze zawsze, gdy przygotowujesz skomplikowane procedury, funkcje czy obliczenia tak, aby stanowiło to ułatwienie dla ciebie, gdy po dłuższym czasie wrócisz do analizy kodu. Komentarze także są istotne dla innych osób, które będą miały możliwość zapoznania się z kodem źródłowym twojego programu.

4.1.2 Definicja a deklaracja

Zapamiętaj różnice pomiędzy deklaracją a definicją, żebyśmy później dobrze się rozumieli. Brak zrozumienia tego zagadnienia na samym początku prowadzi do wielu nieporozumień, bywa także powodem rzekomych trudności w nauce języka C.

1. **Deklaracja** – określa pewne własności identyfikatora *(zmiennej czy funkcji)*, jednak nie rezerwuje pamięci.
2. **Definicja** – zajmuje pamięć dla nowego obiektu i jednocześnie go deklaruje.

Wynika z powyższego, że definicja jest równocześnie deklaracją, ale nigdy na odwrót.

Przykłady **Deklaracji**:

```
extern int a1;
extern uint8_t tab[];
int max(int a, int b);
```

1. Informuje kompilator, że identyfikator a1 oznacza zmienną typu int. Jednocześnie słówko extern oznacza, że zmienna ta jest tworzona poza aktualnym plikiem źródłowym.
2. Informuje kompilator, że identyfikator tab jest tablicą elementów jedno-bajtowych bez znaku.
3. Informuje kompilator, że identyfikator max jest funkcją zwracającą wartość typu int, oraz przyjmującą dwa argumenty typu int.

Przykłady **Definicji**:

```
int  b1;
int c2 = 5;
uint16_t tab[20];

int max(int a, int b)
{
     return (a>b) ? a : b;
}
```

1. Tworzy zmienną b1, zajmuje dla niej pamięć (w języku AVR GCC) będą to dwa bajty, oraz informuje kompilator, że identyfikator b1 oznacza zmienną typu int.
2. Tworzy zmienną c2, zajmuje dla niej pamięć, zostaje ona zainicjalizowana wartością 5, oraz informuje kompilator, że c2 oznacza zmienną typu int.
3. Tworzy tablicę tab, zajmuje dla niej pamięć 40 bajtów oraz informuje kompilator, że identyfikator tab jest tablicą dwubajtowych elementów bez znaku.
4. Tworzy funkcję max, zajmuje dla niej pamięć lecz tym razem w obszarze pamięci programu FLASH, umieszcza w niej program funkcji, oraz informuje kompilator, że funkcja max jest funkcją zwracającą wartość typu int a także o tym, że przyjmuje ona dwa argumenty także o typie int.

Nazwy zmiennych i funkcji można tworzyć dowolnie, ale z pewnymi ograniczeniami: nie mogą one być nazwami słów kluczowych używanych przez kompilator oraz nie mogą zaczynać się od cyfry. Nazwy mogą być pisane zarówno wielkimi jak i małymi literami, jednak trzeba o tym pamiętać, ponieważ jeśli zdefiniujemy zmienną o nazwie **`Rozmiar`** (zaczyna się dużą literą), to później w kodzie kompilator nie rozpozna tej nazwy, jeśli napiszesz ją tak: **`rozmiar`**.

4.1.3 Wyrażenia logiczne (warunki)

W języku C występuje wiele instrukcji sterujących programem (poznasz je w kolejnym rozdziale), które podejmują decyzje o wykonaniu lub niewykonaniu pewnych zadań w zależności od spełnienia lub niespełnienia jakiegoś warunku. Dokładniej mówiąc w zależności od tego, czy jakieś wyrażenie jest prawdziwe, czy fałszywe. Najpierw jednak musisz się dowiedzieć, co to jest prawda i fałsz w języku C. Poniżej kilka przykładów wyrażeń logicznych:

```
1.  ( x < 50 )

2.  ( x == a )

3.  ( x != a )
```

Nie znając wartości zmiennych x lub a nie jesteśmy w stanie ocenić czy te wyrażenia są prawdziwe czy fałszywe. Jeżeli jednak powiem, tobie teraz, że x=7 natomiast a=10, to jesteś w stanie szybko stwierdzić, że:

1. Wyrażenie jest prawdziwe ponieważ 7 jest mniejsze od 50
2. Wyrażenie jest fałszywe ponieważ 7 nie równa się 10
3. Wyrażenie jest prawdziwe ponieważ 7 jest różne od 10

Zaraz, zaraz ale skąd będzie o tym wiedział mikrokontroler. Okazuje się, że to nie będzie dla niego żadnym problemem. Jeśli mikrokontroler napotka np. taki warunek (x < 50), to najpierw podobnie jak my dokona obliczenia i na tej podstawie sprawdzi, czy jest on prawdziwy, czy fałszywy. Zmienna x przecież musiała być gdzieś wcześniej zdefiniowana w programie, stąd będzie znana jej wartość w momencie, gdy dojdzie do sprawdzania warunku.

ZAPAMIĘTAJ!
Wartość zero – jest zawsze rozumiana, jako stan: fałsz
Wartość inna niż zero – jest zawsze rozumiana, jako stan: prawda

Dzięki temu w wyniku operacji **`a=(5<10)`** kompilator przydzieli zmiennej **`a`** wartość jeden, natomiast w wyniku operacji **`a=(25<10)`** zmienna **`a`** przyjmie wartość zero.

Dzięki powyższemu, zamiast w instrukcji sterującej wpisywać warunek sprawdzający czy np. wartość **`x`** jest większa od zera w tradycyjny sposób: **`(x>0)`**, można zapisać to samo w prostszy **`(x)`**. Ponieważ zgodnie z powyższymi definicjami prawdy i fałszu w języku C, warunek **`(x)`** będzie spełniony *(prawdziwy)* tylko wtedy, gdy **`x`** będzie większe od zera. Przy założeniu oczywiście, że korzystamy z typu liczby całkowitej bez znaku. W związku z tym warunek zapisany z kolei w ten sposób **`(1)`** będzie zawsze prawdziwy *(spełniony)*.

4.2 Najważniejsze instrukcje

Zaczniemy od kluczowych instrukcji, bez których nie można byłoby napisać żadnego programu.

4.2.1 Instrukcja warunkowa If , else

W języku C instrukcja **`if`** *(co oznacza po polsku „jeśli”)* może występować w dwóch postaciach:

```
if(warunek) instrukcja
if(warunek) instrukcja1 else instrukcja2
```

Są to podstawowe instrukcje języka C. Pierwsza postać oznacza, że jeśli będzie spełniony

warunek, który może być dowolnym wyrażeniem, tylko wtedy zostanie wykonana instrukcja występująca w dalszej części.

Druga postać stosowana jest do tzw. „rozgałęzień” programu. Oznacza, że jeśli będzie spełniony `warunek` to zostanie wykonana `instrukcja1`, a jeśli `warunek` nie będzie spełniony, to zostanie wykonana `instrukcja2`.

Symbolicznie oznaczona instrukcja może stanowić zarówno jedną instrukcję programu, co można zapisać w kodzie tak:

```
if(x<50) wysokosc=0;
else wysokosc=1;
```

ale może także oznaczać kilka instrukcji, tyle że wtedy musimy je zebrać pomiędzy nawiasami klamrowymi **{}**

```
if(x<50)
{
      wysokosc=0;
      y=0;
}
else
{
      wysokosc=100;
      y=20;
      z=33;
}
```

W pierwszej prostszej postaci w zależności od warunku **(x<50)** była przydzielana różna wartość do zmiennej o nazwie **wysokosc**.

W drugiej postaci w zależności od spełnionego warunku lub nie, ustawiliśmy pewne wartości kilku różnym zmiennym, dlatego zastosowaliśmy nawiasy klamrowe ograniczające odpowiednio pierwszą i drugą (po else) sekcję warunku.

Należy wspomnieć także, iż instrukcje if mogą być zagnieżdżone. Spójrzmy na kod poniżej. Widać na nim dwie instrukcje warunkowe zagnieżdżone, a dokładniej mówiąc, zagnieżdżona jest instrukcja if(warunek_2). Została ona tutaj specjalnie wyróżniona szarym kolorem ramki. Kolejnym wyróżnikiem, jaki występuje w kodzie programu, są „wcięcia” tabulatorów. Widzimy, że cały zagnieżdżony warunek jest przesunięty w prawo. Bez takich wcięć analiza kodu programu byłaby prawie niemożliwa, a przynajmniej bardzo, ale to bardzo utrudniona.

```
if(warunek_1)
{
      if(warunek_2) { //instrukcje }
}
else
{
              // instrukcje
}
```

Wiemy jednak, że nawiasy klamrowe nie zawsze muszą występować, może dojść w takich sytuacjach do sporych problemów szczególnie, jeśli nie zastosujemy w odpowiedni sposób wcięć w programie.

```
if(warunek_1)
if(warunek_2) instrukcja1;
else
{
instrukcja2;
}
```

Jak przeanalizować taki kod? Do którego warunku odnosi się instrukcja else? Dla kompilatora jest to jasne jak słońce, ponieważ występuje zasada, że jeśli brak nawiasów klamrowych przed instrukcją else, to odnosi się ona zawsze do najbliższej poprzedzającej ją instrukcji if. Zobaczmy jednak, jak należy to zapisać tak, abyśmy także my mogli to spokojnie i bez błędów analizować. Znowu ważne są wcięcia.

```
if(warunek_1)
      if(warunek_2) instrukcja1;
      else
      {
            instrukcja2;
      }
```

Myślę, że teraz także dla ciebie na początku drogi w C będzie to bardzo przejrzysty zapis. Nie martw się, jeśli do tej pory miałeś problemy ze zrozumieniem różnego rodzajów kodów programów napisanych w C przez inne osoby. Nie znałeś jeszcze zasad, jakie rządzą składnią, a na dodatek mogłeś natknąć się na programy pisane bez wcięć przez niedoświadczone osoby lub takie, które już coś potrafią, ale uważają, że wcięcia nie są im potrzebne. Jednak takie podejście, uwierz mi, zawsze prędzej czy później skończy się źle.

Bywają pewne formy, gdzie musi nastąpić wybór wielowariantowy za pomocą wielu instrukcji if … else. W takich sytuacjach można pominąć tabulatory (wcięcia), o ile będzie to np. taki prosty blok:

```
if(warunek_1) instrukcja1;
else if(warunek_2) instrukcja2;
else if(warunek_3) instrukcja3;
else if(warunek_4) instrukcja4;

kolejne_instrukcje;
```

Taki blok analizujemy następująco: jeśli spełniony jest warunek_1, wykonaj instrukcję1, zakończ działanie bloku i przejdź do kolejnych instrukcji programu.

Jeśli jednak warunek_1 nie jest spełniony, to sprawdź warunek_2, jeśli jest spełniony, to zakończ działanie bloku i przejdź do kolejnych instrukcji programu.

Jeśli warunek_2 nie jest spełniony, to sprawdź warunek_3 i tak dalej.

Tego typu bloki konstrukcji wielopoziomowego wyboru od razu mogą skojarzyć się z pomysłem zastosowania tego mechanizmu do oprogramowania wielopoziomowego MENU dla użytkownika. Rzeczywiście, przy prostej budowie menu można z tego korzystać. Jednak niedługo poznamy specjalną instrukcję, która jeszcze wygodniej pozwala nam organizować wielopoziomowe wybory w kodzie programu.

Dodam jeszcze, że instrukcje warunkowe if mogą sprawdzać warunki złożone, tzn. składające się z wielu warunków bądź obliczeń. Przyjrzymy się temu bliżej ,gdy będziemy omawiać operatory. Wtedy lepiej zrozumiesz zapis typu:

```
if ( (x>50 && x<100) || (x==5) ) instrukcja1;
```

Na razie powiem tylko, że instrukcja1 zostanie tylko wtedy wykonana, jeśli zmienna x zawiera się w przedziale od 51 do 99 lub jest równa 5. Znaki **&&** oraz **||** to właśnie operatory.

4.2.2 Pętla while

Konstrukcja while *(po polsku „dopóki")* służy do realizacji jednej z podstawowych pętli programowych. Występuje ona w formie:

```
while(warunek) instrukcja(-e);
```

Oznacza to, że dopóki **warunek** będzie spełniony (prawda), dopóty będzie wykonywana **instrukcja**. Zgodnie ze składnią języka, o której pisaliśmy wyżej, pojedynczą instrukcję można zastąpić dowolnym blokiem wielu instrukcji tyle, że trzeba je umieścić wewnątrz nawiasów klamrowych **{}**. Można więc w ramach jednej pętli zapisać wiele instrukcji w ten sposób:

```
x=0;
while(x<10)
{
        // instrukcja1
        // instrukcja2
        // instrukcja3
        // ……….
        // instrukcjaN
        x=x+1;
}

// kolejne instrukcje programu
```

Zawartość pętli będzie wykonana dziesięciokrotnie. Zauważ, że przed rozpoczęciem pętli przypisaliśmy zmiennej x wartość zero. Zatem warunek **(x<10)** jest spełniony i zostaną wykonane kolejno instrukcje wewnątrz nawiasów klamrowych. Ostatnia instrukcja spowoduje zwiększenie wartości **x** o jeden, po czym znowu nastąpi sprawdzenie warunku. Jako że **x** równy będzie 1, to i tym razem warunek zostanie spełniony. Blok instrukcji będzie dotąd wykonywany, dopóki zmienna **x** w wyniku zwiększania zawartości o jeden nie osiągnie w końcu wartości równej dziesięć. W takiej sytuacji warunek **(x<10)** nie będzie już prawdziwy/spełniony i pętla nie wykona instrukcji zawartych w nawiasach klamrowych. Rozpocznie się wykonywanie kolejnych instrukcji programu.

Bardzo często stosuje się w programach tzw. pętlę nieskończoną. Chodzi o to, aby wykonywać pewien blok instrukcji bez końca. Można wtedy posłużyć się konstrukcją:

```
while(1)
{
        // instrukcje
}
```

Zgodnie z tym co mówiliśmy o prawdzie i fałszu w języku C, wartość większa od zera będzie zawsze oznaczać prawdę. Zatem warunek (1) będzie zawsze spełniony, ponieważ liczba 1 jest większa od zera i symbolizuje w tym warunku „prawdę”.

Zauważ, proszę, istotę działania tej pętli. Otóż, zawsze przed jej pierwszym wykonaniem sprawdzany jest warunek. Gdyby nie był on spełniony (prawdziwy), to nigdy nie doszłoby do wykonania instrukcji w jej wnętrzu.

4.2.3 Pętla do..while

Konstrukcja **do** ... **while** ... oznacza z angielskiego **Rób … Dopóki …** i pozwala na realizację innego rodzaju pętli programowej. Jej forma to:

```
do instrukcja1 while(warunek);
```

Po analizie oznacza to, rób (wykonuj) **instrukcję1**, dopóki będzie spełniony **warunek**. Jak zwykle też pojedynczą instrukcję możemy zastąpić blokiem wielu instrukcji umieszczonych wewnątrz nawiasów klamrowych.

```
do
{
            Instrukcja1;
            Instrukcja2;
            Instrukcja3;
} while(warunek);
```

Zauważ, że w odróżnieniu od omawianej wyżej zwykłej pętli while, tutaj mamy do czynienia z sytuacją, w której najpierw wykonywana jest instrukcja1 lub blok instrukcji, a dopiero na końcu sprawdzany warunek. Zatem w pierwszym przebiegu tej pętli zostaną zawsze wykonane instrukcje w jej wnętrzu.

4.2.4 Pętla for

Ten typ pętli programowej wykonywany jest zdecydowanie najczęściej w różnych programach. Posiada ona postać:

```
for( init ; wyrażenie_warunkowe ; krok) treść_pętli;
```

init oznacza instrukcję bądź grupę instrukcji, które służą do inicjalizacji pracy pętli. W praktyce najczęściej będziesz stosował tu pojedynczą instrukcję.

wyrażenie_warunkowe tak jak to zwykle bywało w instrukcjach warunkowych, będzie obliczane przed każdym wykonaniem pojedynczego obiegu pętli. Jeśli wyrażenie/warunek będzie spełniony, to przebieg pętli zostanie wykonany, jeśli przestanie być prawdziwy, to przebieg nie zostanie wykonany. k**rok** to instrukcja wpływająca na licznik wykonywania pętli. Jest ona realizowana za każdym razem na zakończenie pojedynczego obiegu pętli tuż przed ponownym sprawdzeniem wyrażenia warunkowego na początku pętli.

W praktyce będzie to wyglądało tak:

```
for(i=0;i<10;i=i+1)
{
            instrukcja1;
}
```

W powyższym przykładzie sekcję **`init`** stanowi instrukcja **`i=0`**. Inicjalizujemy w ten sposób zmienną i, która będzie odpowiedzialna za **iterację** *(wielokrotnie powtarzalną czynność)*.

Sekcja **`wyrażenie_warunkowe`** to w naszym przypadku warunek **`i<10`**. Zatem pętla będzie się wykonywała do momentu, dokąd warunek będzie prawdziwy. Jako że zmienna **`i`** została zainicjalizowana wartością zero, można powiedzieć, że pierwszy przebieg pętli zostanie na pewno wykonany, gdyż warunek taki będzie prawdziwy.

Dzięki sekcji **`krok`**, która u nas ma postać **`i=i+1`** wiemy, że za każdym przebiegiem pętli, pod koniec wykonywania każdego jej obiegu zmienna **`i`** będzie zwiększana o jeden. Co za tym idzie, można śmiało wywnioskować, że pętla taka wykona się 10 razy.

Ile razy wykonana zostałaby pętla for zapisana w poniższy sposób?

```
for(i=0;i<10;i=i+2) instrukcja1;
```

Tylko pięć razy, ponieważ wartość zmiennej **`i`** w sekcji krok, jest zmieniana w większym tempie. Tym razem **`i=i+2`**. Zatem **`wyrażenie_warunek`** będzie spełnione tylko wtedy, gdy wartości zmiennej **`i`** będą wynosiły kolejno: 0, 2, 4, 6, 8.

Mam nadzieję, że ten krótki opis dał tobie dużo do myślenia i jeśli przypadkiem znasz pętle for z innych języków programowania, to śmiało stwierdzisz, że składnia tej pętli w języku C jest zdecydowanie najlepsza. Daje ogrom możliwości i nie wprowadza wielu ograniczeń.

Dodatkową ciekawostką jest to, że w języku C można śmiało pomijać niektóre bądź wszystkie części składowe pętli, pozostawiając jedynie znaki średników, które je oddzielają. Zatem poniższy zapis:

```
for(;;)
{
            // instrukcje;
}
```

Często spotkasz, jako pętlę nieskończoną. Opuszczenie sekcji **`wyrażenie_warunek`** jest zawsze równoznaczne w tym przypadku z tym, jakby warunek był zawsze spełniony. Można także skorzystać z zapisu:

```
for(i=5;x>20;) instrukcja;
```

W takim przypadku mamy do czynienia z inicjalizacją zmiennej **`i`** w pętli, następnie zostaje sprawdzany warunek **`(x>20)`**, który wcale nie musi być związany ze zmienną typu iteracyjnego, czyli **`i`**. Natomiast pominęliśmy w ogóle sekcję **krok.** Oznacza to, że pętla będzie pracować w zależności od tego, co wewnątrz niej będzie się działo z wartością zmiennej **`x`**.

Jak wspominałem wcześniej, sekcja inicjalizacji bądź sekcja krok mogą składać się z kilku instrukcji oddzielonych od siebie przecinkiem. Nie nadużywaj jednak takich konstrukcji ze względu na możliwość znacznego zmniejszenia czytelności kodu programu. Przykład:

```
for(i=0,k=10;i<10;i=i_1,k=k-1)
{
            // instrukcje pętli
}
```

W tym przypadku dostrzeżesz, iż zmienna i służy do iteracji, natomiast niejako dodatkowo można wykorzystać sekcje pętli do cyklicznych działań z innymi zmiennymi, które mogą być przydatne wewnątrz pętli. Każda pętla może także zostać przerwana w dowolnym momencie za pomocą specjalnej instrukcji, o której powiem za chwilę.

4.2.5 Instrukcja break

Instrukcja **break** z angielskiego oznacza w tym przypadku „**przerwać**”. Może zostać ona użyta wewnątrz dowolnej pętli programowej lub wewnątrz instrukcji switch. Powoduje ona natychmiastowe i bezwarunkowe przerwanie działania pętli bądź instrukcji switch oraz jej opuszczenie. W związku z czym program rozpoczyna wykonywanie kolejnych instrukcji programu, jakie znajdują się po wystąpieniu pętli lub instrukcji switch.

Oznacza to, że można przerwać działanie każdej formy tzw. pętli nieskończonej. Wystarczy w jej wnętrzu wstawić polecenie break. Oczywiście takie polecenie najczęściej w tego typu przypadkach zostaje użyte w zależności od zaistnienia pewnej sytuacji, czyli jednym słowem w zależności od spełnienia jakiegoś warunku/wyrażenia, np.

```
while(1)
{
      // instrukcje
      if(warunek) break;
      // instrukcje
}
```

Poznaliśmy już wcześniej taką konstrukcję pętli nieskończonej z użyciem pętli while, jednak równie dobrze moglibyśmy zastosować konstrukcję **`for(;;)`** zamiast **`while(1)`**. Tak czy inaczej wewnątrz za każdym obiegiem sprawdzany jest jakiś warunek, i jeśli zostanie on spełniony, wykonywanie obiegu pętli zostanie natychmiast przerwane. Nie wykona się w jej wnętrzu już żadna następna instrukcja.

4.2.6 Instrukcja switch

Switch z angielskiego oznacza „przełącznik”. Tak też zachowuje się ta instrukcja. Służy ona do podejmowania wielowariantowych decyzji. To właśnie za jej pomocą można zastąpić blok wielowariantowego wyboru, o jakim mówiłem w rozdziale poświęconym instrukcjom **`if`**... **`else`**. Oto jak wygląda postać takiej instrukcji. Jest to pewna konstrukcja, spójrz poniżej:

```
switch(wyrażenie)
{
      case wartosc1:
      instrukcje;
      [break;]
      case wartosc2:
      instrukcje;
      [break;]
      case wartosc3:
      instrukcje;
      [break;]
      default:
      instrukcje;
}
```

Wygląda to może troszkę skomplikowanie na pierwszy rzut oka, ale to tylko złudzenie, zapewniam cię. Już wyjaśniam, co to wszystko po kolei oznacza. To potężne narzędzie w języku C.

Instrukcja rozpoczyna się od sprawdzenia naszego przełącznika, którym jest wyrażenie. Oznacza to, że w nawiasach okrągłych może wystąpić sama zmienna np. **x**, ale równie dobrze może wystąpić wyrażenie matematyczne, którego wynik będzie przełącznikiem. Następnie wewnątrz nawiasów klamrowych mamy sekcje o nazwie **case** lub **default**, dzięki którym możemy zdecydować, jakie instrukcje chcemy wykonać w zależności od konkretnych wartości naszego **wyrażenia**/przełącznika. Po słówku **case** zawsze podajemy wartość przełącznika, jaka nas interesuje, co oznacza, że jeśli przełącznik będzie miał w momencie wejścia w instrukcję **switch** taką wartość, to instrukcje występujące w kolejnych liniach po słówku **case** zostaną wykonane, jeśli inną wartość, to pominięte i zostanie rozpatrzona kolejna pozycja **case**.

Po każdym pakiecie instrukcji następujących po sprawdzeniu określonej wartości przełącznika **case**, może wystąpić instrukcja **break**. Tylko dlatego ująłem ją w powyższym schematycznym w przykładzie w nawiasy kwadratowe, aby zakomunikować, że instrukcja **break** może w tym miejscu występować, ale nie musi. Nie jest to obligatoryjne. Jednak, jeśli jej nie ma, to zostaną wykonane kolejne instrukcje zawarte instrukcji następnych sekcji case, switch. Może to spowodować, że całość nie zareaguje tylko na jeden przełącznik, a na kilka. Zatem jeśli zależy nam na wykonaniu instrukcji dotyczących tylko jednego przełącznika, to najczęściej będziemy blok rozpoczynający się od słówka **case** kończyli rozkazem **break**, który przerwie dalsze wykonywanie instrukcji zawartych w **switch**, ponieważ uznajemy, iż inne są niepotrzebne w tym momencie. W praktyce może to wyglądać tak:

```
x=2;
switch(x)
{
      case 0:
      czas=10;
      break;

      case 1:
      czas=23;
      break;

      case 2:
      czas=38;
      break;

      case 3:
      czas=42;
      break;

      default:
      czas=0;
}
```

Króciutko przeanalizujemy, co stanie się w wyniku działania powyższego kodu programu. Na początku bądź „ręcznie”, bądź w wyniku wykonania jakiejś funkcji, zmienna **x** przyjmuje wartość równą dwa. Rozpoczyna się teraz instrukcja switch sprawdzająca wartość zmiennej **x**, pełniącej dla nas rolę przełącznika, od którego chcemy spowodować, aby z kolei zmienna czas przyjęła pewną konkretną wartość. Zakładamy także, że jeśli zmienna **x** nie osiągnie

żadnej z założonych wartości, to zmienna czas domyślnie zostanie wyzerowana. Po wejściu w instrukcję switch za pomocą pierwszego słówka **case**, sprawdzamy, czy nasz przełącznik, jakim jest wartość zmiennej **x**, nie posiada wartości zero. Jeśli nie, to zignorowane zostaną kolejne linijki programu aż do napotkania kolejnego momentu, w którym pojawi się słówko **case**. Oznaczać to będzie, że po raz kolejny sprawdzamy, czy nasz przełącznik nie posiada wartości równej jeden. Jeśli nie, to program przeskakuje do kolejnego słówka **case**, które tym razem sprawdza, czy **x** równa się dwa? Zgadza się, jak widać przed instrukcją **switch**, zmienna **x** jest równa dwa.

W takim razie, rozpoczną się wykonywać kolejne instrukcje, które znajdują się po tym właśnie sprawdzeniu słówkiem **case**. W naszym przypadku jest to tylko jedna instrukcja, ale można równie dobrze w kolejnych liniach napisać ich więcej. Tutaj można, ale nie trzeba, koniecznie stosować do bloku instrukcji, nawiasów klamrowych. Zauważ jednak, że na zakończenie tych instrukcji zostaje wykonana instrukcja **break**. Powoduje ona zakończenie działania całości. O to nam chodziło. Aby w zależności od konkretnej wartości zmiennej **x** odpowiednio ustawić zmienną **czas**.

Dodajmy na koniec, że gdyby wartość zmiennej **x** była różna od **0, 1, 2, 3** *(bo takie wartości zostają sprawdzane za pomocą słówek **case**)*, to zrealizowana zostałaby sekcja instrukcji na końcu po słówku **default**. W naszym przypadku zmienna **czas** zostałaby wyzerowana. Jeśli taka sekcja case lub default występuje na samym końcu, to zbędne jest już użycie instrukcji **break**.

4.2.7 Instrukcja continue

Instrukcja ta bywa przydatna wewnątrz każdej z omawianych pętli programowych. Może czasem wystąpić sytuacja, gdy pętla zawiera długi blok instrukcji programu występujących jedna po drugiej, że w zależności od jakiegoś czynnika chcemy pominąć wykonywanie części bloku tychże instrukcji. Jej postać przedstawia się następująco:

```
for(;;)
{
      instrukcja1;
      instrukcja2;
      instrukcja3;

      if(warunek) continue;

      instrukcja4;
      instrukcja5;
}
```

Oczywiście rodzaj pętli może być dowolny, równie dobrze w tym przykładzie moglibyśmy zastosować **while()** czy też **do**...**while()**. Jak to działa? Otóż zakładając, że jeśli warunek nie jest spełniony, to dokładnie w każdym obiegu pętli wykonywane są wszystkie instrukcje od 1 do 5. Jeśli jednak w konkretnym czy też w wielu przebiegach warunek zacznie być spełniony/prawdziwy, to instrukcje od 4 do 5 są całkowicie pomijane. Można powiedzieć, że instrukcja continue powoduje przejście na sam koniec pętli. Efekt będzie taki, jakby całość została wykonana, następuje zakończenie obiegu, po czym zostaje sterowanie przekazane znowu na początek pętli, gdzie sprawdzane są jej warunki pracy.

4.2.8 Nawiasy klamrowe

Kilka praktycznych porad jak ich używać aby uniknąć pomyłek, o które szczególnie łatwo, jeśli stosować będziemy wiele zagnieżdżonych instrukcji **`if`**, a w nich jeszcze rozbudowanych **`pętli`**, które na dodatek także mogą zawierać kolejne instrukcje **`if`**. Poniżej przedstawię często spotykane trzy sposoby używania nawiasów klamrowych w programach.

```
while(1) {
     instrukcje;                        I sposób
}

while(1)
     {
           instrukcje;                  II sposób
     }

while(1)
{
     Instrukcje;                        III sposób
}
```

Każdy sposób jest generalnie prawidłowy, gdyż zawiera odpowiednie wcięcia. Jednak warto zdecydować się na jeden z nich taki, który tobie będzie sprawiał najmniej problemów. Dla mnie najlepszym sposobem, jakiego najczęściej korzystam, gdy piszę własne kody, jest ten trzeci. Powiem więcej, żeby uniknąć pomyłek związanych z pisaniem długiego kodu programu i zagnieżdżonych instrukcji, po których stosuję klamry, zawsze podchodzę do tego właśnie tak. Po napisaniu instrukcji warunkowej czy pętli wciskam klawisz ENTER, po czym równo pod rozpoczynającą się instrukcją stawiam otwarty nawias klamrowy, ponownie klikam klawisz ENTER (nawet dwukrotnie) i wstawiam zamknięty nawias klamrowy równiutko pod tym otwartym powyżej. Dopiero wtedy przenoszę kursor pomiędzy oba nawiasy i zaczynam wpisywać kod programu pomiędzy nimi. Dzięki temu rzadko mylę się, jeśli chodzi o stosowanie tych nawiasów.

Dodam, że niektóre zaawansowane środowiska jak np. ECLIPSE, opisane przeze mnie wyżej czynności wykonują za mnie automatycznie! Oznacza to, że gdy po napisaniu instrukcji warunkowej lub pętli wcisnę raz klawisz ENTER, to automatycznie pod spodem umieszczone zostają od razu dwa nawiasy klamrowe a kursor umiejscawia się wraz z poprzedzającym go tabulatorem/wcięciem w linii pomiędzy nimi, dzięki czemu bez uciążliwych wyżej opisanych czynności przystępuję do pisania kodu. Inne środowiska i edytory oferują jeszcze inne narzędzia/gadżety wspomagającą pracę programisty w tym zakresie. Dlatego pisanie programu w zwykłym lub lekko zaawansowanym programie typu notatnik, który oferuje tylko kolorowanie składni, bywa w dzisiejszych czasach bardzo uciążliwe.

4.2.9 Instrukcja goto

Pozostawiłem tę instrukcję na koniec. Najchętniej w ogóle bym jej nie omawiał, ponieważ jej istnienie powoduje, że początkujący często nabierają złych nawyków programowania, gdy się przyzwyczają zbytnio do tej instrukcji. Niemniej jednak jest kilka drobnych sytuacji, gdzie może się ona przydać. Wtedy nie jest wstydem jej używanie. W pozostałych przypadkach jej nadmierne stosowanie wręcz świadczy tylko źle o programiście. Cóż to za „wstydliwa” instrukcja? Jej składnia to:

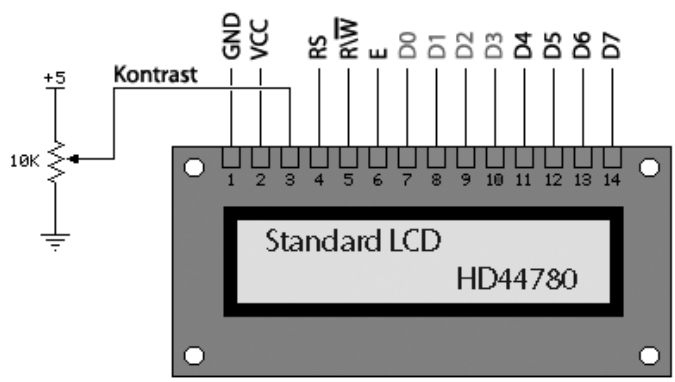
Zwróć uwagę, że zmiennej tablicowej o nazwie cyfry[] nie deklarujemy w pliku nagłówkowym, tym samym ukrywamy jej istnienie przed innymi modułami programu. Jest ona tak naprawdę potrzebna i wykorzystywana tylko w module d_led.c

Na zakończenie pliku, występują jeszcze deklaracje funkcji, które chcemy udostępnić na zewnątrz. W naszym przypadku jest to tylko jedna funkcja o nazwie d_led_init(); Wspomniałem też, że nasz plik nagłówkowy pełni jeszcze jedną ważną funkcję. Zauważ, że właśnie w tych plikach zwykle zdefiniowane są wszelkie zależności programu od sprzętu. Jeśli więc chciałbyś zmienić porty, do których podłączasz segmenty czy też anody, to wszelkich zmian trzeba dokonać tylko w tym jednym miejscu. Nie trzeba błądzić w gąszczu całego kodu i modyfikować zawartość setek linii, żeby pozmieniać wszędzie odwołania do portu A czy C.

5.5 Wyświetlacz LCD (HD44780)

Zajmiemy się teraz wyświetlaniem danych na różnego rodzaju wyświetlaczach alfanumerycznych LCD zgodnych ze standardem scalonego sterownika HD44780. Mamy dostępnych na rynku wiele odmian takich wyświetlaczy. Różnią się nie tylko rozmiarem fizycznym, ale także ilością wyświetlanych linii i kolumn. Najbardziej typowym, używanym do testów bywa LCD 2x16. Posiada on dwa wiersze, z których każdy może wyświetlić 16 znaków.

Zgodność ze standardem HD44780 oznacza, że mamy do dyspozycji 14 pinów wyświetlacza, za pomocą których możemy podłączyć go np. do naszego układu mikroprocesorowego. Standard stanowi o tym, że każdy kolejny pin pełni ściśle określoną funkcję niezależnie, czy korzystamy z wyświetlacza 2x16, czy może 2x20, czy też 4x20 lub innego.



Rysunek 38.

(Oczywiście ich rozmieszczenie na płytce może się różnić od tego, które tu przedstawiłem).

Dwa pierwsze piny to zasilanie. Trzeci pin podłączamy jak wyżej do suwaka potencjometru 10K przyłączonego pomiędzy VCC i GND. Potencjometr ten służy do ustawiania kontrastu wyświetlacza. Kolejne trzy piny RS, RW oraz E to linie sterujące natomiast następnych 8 linii D7..D0 to linie danych wyświetlacza. Jest ich wprawdzie osiem, jednak standard HD44780 umożliwia korzystanie z niego także przy pomocy mniejszej ilości połączeń, dokładnie mówiąc, umożliwia pracę w trybie 4-bitowym. Korzystamy wtedy tylko z linii D7..D4 natomiast pozostałe nieużywane D3..D0 nie są do niczego podłączone. „Wiszą w powietrzu”. Sposób ten umożliwia podłączenie wyświetlaczy LCD tego typu do mikrokontrolera za pomocą

tylko 7 linii. Znacznie upraszcza to połączenia elektryczne w układzie. Żeby obsługiwać taki wyświetlacza z poziomu języka C, trzeba albo sobie napisać funkcje do jego obsługi, albo znaleźć gotowe rozwiązania w Internecie. Zajmiemy się tym pierwszym rozwiązaniem. W końcu umiejętność obsługi tego typu wyświetlaczy to także podstawa. Zanim zaczniemy pisać te funkcje, musimy się najpierw dowiedzieć, w jaki sposób korzystać z linii sterujących oraz linii danych w trybie 4-bitowym. Wyświetlacze posiadają wewnątrz dwa rodzaje pamięci. CGRAM jest to pamięć generatora własnych znaków użytkownika (można zdefiniować 8 takich znaków). DDRAM to pamięć danych, w której przechowywane są kody ASCII wyświetlanych znaków. Sterownik LCD posiada stały zdefiniowany zbiór komend, jakie można do niego wysyłać w celu realizacji różnych zadań na wyświetlaczu. Poniżej przedstawiam zbiór tych poleceń w postaci typowej tabeli:

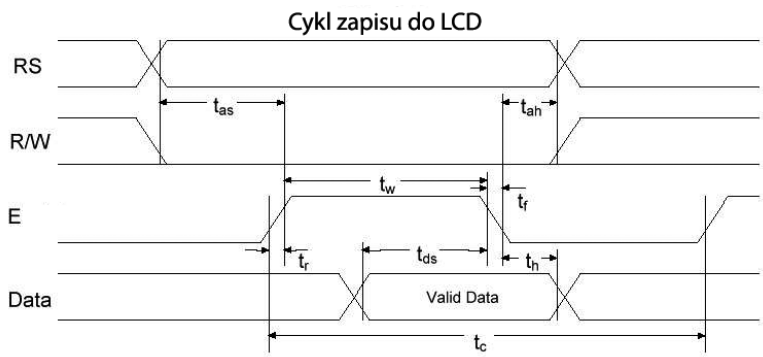
Rozkaz	**RS**	**RW**	**D7**	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**	**Funkcja**
Clear display	0	0	0	0	0	0	0	0	0	1	Czyści wyświetlacz
Cursor home	0	0	0	0	0	0	0	0	1	x	Ust. kursora w poz 0,0
Entry mode set	0	0	0	0	0	0	0	1	I/D	S	Określenie trybu pracy kursora/okna wyśw.
Display On/Off	0	0	0	0	0	0	1	D	C	B	Wł/Wył wyświetlacza.
Cursor/displ. shift	0	0	0	0	0	1	S	R	x	x	Przesuwanie kursora
Function set	0	0	0	0	1	D	N	F	x	x	Tryb pracy wyświe-tlacza
CGRAM set	0	0	0	1	Adres pamięci CGRAM						Ustawia adr. w CGRAM
DDRAM set	0	0	1	Adres pamięci DDRAM							Ustawia adr. w DDRAM
Busy flag read	0	1	B	Adres pamięci DDRAM lub CGRAM							Odczyt flagi zajętości
Data write	1	0	Zapisywany bajt danych								Zapis znaków
Data read	1	1	Odczytywany bajt danych								Odczyt znaków
I/D = 1: kursor lub okno wyświetlacza przesuwa się w prawo *I/D=0: kursor lub okno wyświetlacza przesuwa się w lewo* *S=1: po wpisaniu znaku do LCD kursor stoi w miejscu a przesuwa się zawartość okna* *S=0: po wpisaniu znaku do LCD przesuwa się kursor, zawartość okna pozostaje* *D=1/0: włączenie/wyłączenie wyświetlacza* *C=1/0: włączenie/wyłączenie kursora* *B=1/0: włączenie/wyłączenie migania kursora* *R=1/0: kierunek przesuwu kursora w prawo/lewo* *D=1: interfejs 8-bitowy* *D=0: interfejs 4-bitowy* *N=1: wyświetlacz dwuwierszowy* *N=0: wyświetlacz jednowierszowy* *F=1: rozmiar znaku 5x10 punktów* *F=0: rozmiar znaku 5x7 punktów*											

W większości zastosowań wykorzystuje się tylko kilka podstawowych funkcji, więc nie przerażaj się obszernością tej tabeli. Linie D7..D0 służą do zapisywania informacji do wyświetlacza lub do odczytywania. O tym, czy chcemy zapisać informacje, np. przesłać jedną z funkcji lub znak, decyduje stan linii RW. Jeśli ustawimy na niej stan niski, to oznacza, że dokonujemy zapisu do wyświetlacza. Jeśli zaś ustawimy stan wysoki, to znaczy, że będziemy dokonywać odczytu danych z wyświetlacza. W praktyce bardzo często spotyka się rozwiązania, gdzie linia RW wyświetlacza na stałe podłączona jest do GND, występuje więc na niej cały czas stan niski, czyli nawet nie trzeba koniecznie do normalnej pracy korzystać z

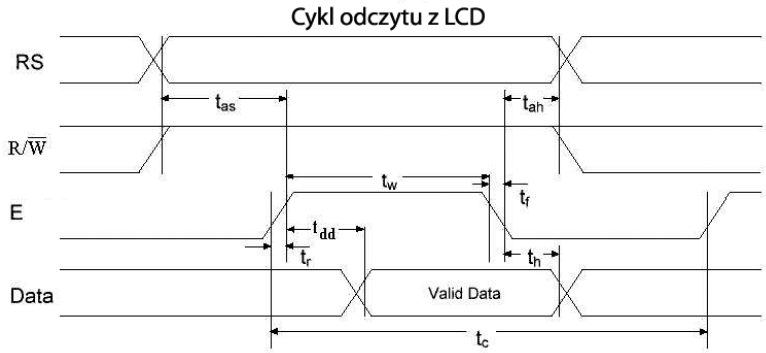
odczytu. Taki sposób okupiony jest jednak nieco dłuższym dostępem do wyświetlacza LCD. Nie stanowi to wprawdzie dużej przeszkody, ponieważ sporo tanich wyświetlaczy ma tak słabe odświeżanie, że próby szybkich zmian zawartości są prawie niewidoczne, wszystko się rozmywa. Jednak są także dużo lepsze i szybsze wyświetlacze ze sterownikiem zgodnym z HD44780, wykonane w technologiach np. VFD (lampowy) czy PLED/OLED. Także cześć lepszych wyświetlaczy LCD jest wystarczająco szybka i warto dla nich pokusić się jednak o to, aby dokonywać odczytu Busy Flag *(Flagi zajętości)*. Odczyt jednego bitu tej flagi pozwala na maksymalne wykorzystanie prędkości transmisji i operacji na wyświetlaczu.

Dzięki temu zastosowanie we własnych programach różnego rodzaju pseudoanimacji pozwala opracować ciekawe dla oka użytkownika efekty. Zawsze korzystam z trybu pracy wykorzystującego odczyt BF. O takim trybie też będziemy rozmawiać. Wiemy już, jak przełączyć wyświetlacza linią RW w stan zapisu lub odczytu. Jednak czasem zmuszeni jesteśmy wysłać do wyświetlacza tylko zwykły znak alfanumeryczny, który ma się na nim ukazać, a czasem musimy przesłać komendę.

Do tego posłuży nam druga linia sterująca o nazwie RS. Jeśli chcemy przesłać komendę, musimy ją ustawić w stan niski, jeśli zaś zwykłe dane *(znaki do wyświetlenia),* to musimy ją ustawić w stan wysoki. Jeśli spojrzysz w tabelkę powyżej, to zobaczysz, w jakich sytuacjach należy ustawiać te linie i w jakim stanie. Pozostaje nam jeszcze trzecia linia sterująca o nazwie E. Używamy jej na bardzo krótki okres. Normalnie jest ona cały czas w stanie niskim. Jeśli ustawimy dane do wysłania na liniach portu mikrokontrolera, jeśli też ustawimy odpowiednio linie RW oraz RS, to wtedy włączamy stan wysoki na linii E i po chwili przywracamy na niej stan niski. Taki impuls wyświetlacz traktuje jako zezwolenie na wykonanie żądanej operacji. Występują tu ścisłe zależności czasowe, których należy przestrzegać, żeby sterownik poprawnie interpretował nasze polecenia.

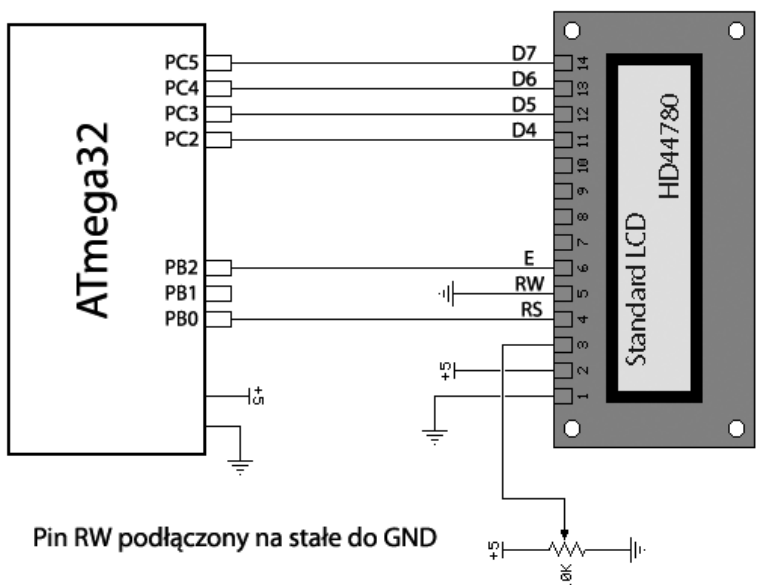


Item		Symbols	Standard min.	Standard max.	Units
Enable cycle time		tc	1000	-	ns
Enable pulse width	High level	tw	50	-	ns
Enable rise and fall time		tr, tf	-	25	ns
Setup time	RS,R/-W -› E	tas	140	-	ns
Address hold time		tah	10	-	ns
Data setup time		tds	195	-	ns
Data hold time		th	10	-	ns

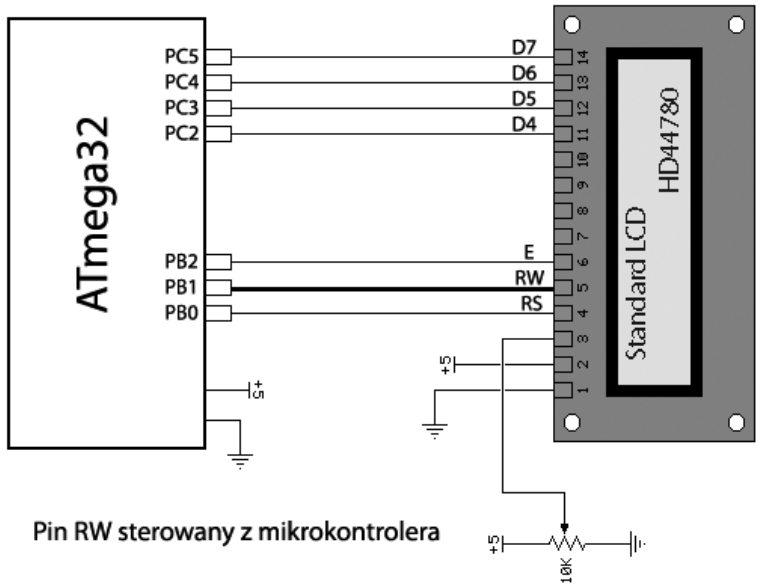


Item		Symbols	Standard		Units
			Min.	Max.	
Enable cycle time		tc	1000	-	ns
Enable pulse width	High level	tw	450	-	ns
Enable rise and fall time		tr, tf	-	25	ns
Setup time	RS,R/-W -› E	tas	140	-	ns
Address hold time		tah	10	-	ns
Data delay time		tdd	-	320	ns
Data hold time		th	20	-	ns

Tak wyglądają dokładnie charakterystyki przebiegów czasowych podczas operacji zapisu do LCD oraz podczas odczytu danych z LCD.



Na rysunku powyżej przedstawiam najprostszy sposób podłączenia wyświetlacza LCD do mikrokontrolera. Pin RW jest na stałe podłączony do GND, zatem możliwe są tylko operacje zapisu do wyświetlacza. Powoduje to, że dostęp do niego jest troszeczkę wolniejszy niż w przypadku, gdy pin RW sterujemy bezpośrednio z mikrokontrolera, jak przedstawione jest to z kolei na rysunku poniżej:



Obydwa sposoby wykorzystują 4-bitowy tryb współpracy z mikrokontrolerem i tylko takim trybem będziemy się zajmować. Postaramy się utworzyć własne, proste i uniwersalne biblioteki do obsługi wyświetlaczy LCD zgodnych z HD44780. Aby mogły być uniwersalne, będziemy musieli jak zwykle funkcje do obsługi zawrzeć w oddzielnym pliku, a w zasadzie plikach, bo z góry wiadomo, że będą, co najmniej dwa. Plik z funkcjami oraz plik nagłówkowy.

Zanim zaczniemy, musimy się zapoznać ogólnie z procedurą inicjalizacji wyświetlacza oraz ze sposobem przekazywania danych 8-bitowych za pomocą takiej uproszczonej 4-bitowej magistrali. Zacznę od sposobu przesyłania danych. W związku z tym, że brak 4 linii, polecenie 8-bitowe przesyłamy do wyświetlacza (bądź je odczytujemy) „na raty". Oznacza to ni miej ni więcej, że najpierw dokonujemy zapisu starszej części bajtu (bity D7..D4) stosując oczywiście tak jak należy sygnały RS, RW oraz impuls E. Następnie w ten sam sposób przesyłamy młodszą część bajty, czyli bity D3..D0. Kolejna rzecz, nad którą musimy się zatrzymać, to sposób, w jaki programowo można przesyłać taki półbajt biorąc pod uwagę odpowiednie wysterowanie linii RS, RW oraz E. Dużego problemu by nie było, gdybyśmy się zdecydowali na to, że zawsze linie D7..D4 wyświetlacza będziemy podłączać tylko do pinów jednego portu procesora. Z poprzedniego rozdziału wiemy już, jak modyfikować tylko część bitów portu bez przypadkowych zmian pozostałych. Jednak „utrudnimy" sobie troszkę zadanie, chociaż ten trud się opłaci, ponieważ będziemy mogli wg naszych założeń podłączać dowolne linie wyświetlacza LCD do zupełnie dowolnych linii portów mikrokontrolera. Żadnych ograniczeń. Kolejnym założeniem do naszych uniwersalnych procedur będzie takie napisanie naszych funkcji, żeby można było ustawić także to, czy korzystamy z podłączonego pinu RW, czy też jest on podłączony na stałe do GND. Obydwie wersje będziemy mogli przetestować na naszej płytce startowej ATB rev1.xx, ponieważ można na niej podłączać pin RW za pomocą zworki albo do GND, albo do wyjścia mikrokontrolera.

Zanim przedstawię ostateczną wersję plików, postaramy się wspólnie popracować nad najbardziej newralgicznymi punktami naszych funkcji. Skoro chcemy podłączać wyświetlacz do dowolnych linii procesora, zacznijmy jak zwykle od dobrego rozpisania definicji preprocesora, które bardzo nam w tym pomogą:

```
// rozdzielczość wyświetlacza LCD (wiersze/kolumny)
#define LCD_Y 2      // ilość wierszy wyświetlacza LCD
#define LCD_X 16     // ilość kolumn wyświetlacza LCD

// tu ustalamy za pomocą zera lub jedynki czy sterujemy pinem RW
#define USE_RW 1

// tu konfigurujemy port i piny do jakich podłączymy linie D7..D4 LCD
#define LCD_D7PORT  C
#define LCD_D7 5
#define LCD_D6PORT  C
#define LCD_D6 4
#define LCD_D5PORT  C
#define LCD_D5 3
#define LCD_D4PORT  C
#define LCD_D4 2

// tu definiujemy piny procesora do których podłączamy sygnały RS,RW, E
#define LCD_RSPORT B
#define LCD_RS 2

#define LCD_RWPORT B
#define LCD_RW 1

#define LCD_EPORT B
#define LCD_E 0

// tu definiujemy adresy w DDRAM dla różnych wyświetlaczy
// różne są w wyświetlaczach 2wierszowych inne w 4wierszowych
#if ( (LCD_Y == 4) && (LCD_X == 20) )
#define LCD_LINE1 0x00     // adres 1 znaku 1 wiersza
#define LCD_LINE2 0x28     // adres 1 znaku 2 wiersza
#define LCD_LINE3 0x14     // adres 1 znaku 3 wiersza
#define LCD_LINE4 0x54     // adres 1 znaku 4 wiersza
#else
#define LCD_LINE1 0x00     // adres 1 znaku 1 wiersza
#define LCD_LINE2 0x40     // adres 1 znaku 2 wiersza
#define LCD_LINE3 0x10     // adres 1 znaku 3 wiersza
#define LCD_LINE4 0x50     // adres 1 znaku 4 wiersza
#endif
```

Dzięki takiej konstrukcji zawsze, gdy będziesz trzeba użyć naszych procedur w innym projekcie z innymi ustawieniami i podłączeniami LCD do procesora, wystarczy, że wyedytujesz plik nagłówkowy i dokonasz w nim stosownych zmian. Jeśli nie zechcesz sterować pinem RW z procesora, to ustawisz wartość zero dla parametry **USE_RW**, i pozmieniasz porty oraz numery pinów, jeśli zajdzie taka potrzeba. Kolejny krok to zainicjalizowanie wszystkich pinów sterujących. W związku z tym, że każdy z nich może być w dowolnym projekcie ustawiony na innym porcie i mieć różny nr pinu, musimy każdy ustawiać oddzielnie. W różnych liniach programu. Spowoduje to przyrost o kilka dodatkowych bajtów po kompilacji w pamięci programu, jednak biblioteka będzie w pełni uniwersalna. Coś za coś.

Zapewne widzisz już, że w jakiś „dziwny” sposób zdefiniowałem nazwy portów. Określiłem je tylko za pomocą litery oznaczającej port zamiast pełnej nazwy np. PORTC czy PORTB. Nie zrobiłem tego przypadkowo. Chciałbym pokazać ci bowiem kilka ciekawych makr preprocesora, z których często korzystam.

```
// Makra upraszczające dostęp do portów
// *** PORT
#define PORT(x) SPORT(x)
#define SPORT(x) (PORT##x)
// *** PIN
#define PIN(x) SPIN(x)
#define SPIN(x) (PIN##x)
// *** DDR
#define DDR(x) SDDR(x)
#define SDDR(x) (DDR##x)
```

Dzięki takim makrodefinicjom będzie można w bardzo prosty sposób odwoływać się do różnych portów. Nie trzeba definiować nazw np. dla rejestru kierunku portu C (DDRC), dla rejestru wyjściowego portu C (PORTC), czy też dla rejestru wejściowego portu C (PINC). Teraz inicjalizacja pinów na początku funkcji inicjalizującej mikrokontroler do współpracy z wyświetlaczem LCD będzie mogła wyglądać tak:

```
// inicjowanie pinów portów ustalonych do podłączenia z wyświetlaczem LCD
// ustawienie wszystkich jako wyjścia
DDR(LCD_D7PORT) |= (1<<LCD_D7);
DDR(LCD_D6PORT) |= (1<<LCD_D6);
DDR(LCD_D5PORT) |= (1<<LCD_D5);
DDR(LCD_D4PORT) |= (1<<LCD_D4);
DDR(LCD_RSPORT) |= (1<<LCD_RS);
DDR(LCD_EPORT) |= (1<<LCD_E);
#if USE_RW == 1
     DDR(LCD_RWPORT) |= (1<<LCD_RW);
#endif
```

Tutaj, jak widzisz, skorzystałem z „upraszczającego makra” DDR(x), gdzie zamiast x podałem wcześniej zdefiniowane nazwy portów w postaci pojedynczej litery. Makro jest dwustopniowe (tłumaczyłem we wcześniejszych rozdziałach, z czego to wynika) i korzysta z dyrektywy preprocesora ##. Jak pamiętasz, służy ona do sklejania. W rozwinięciu pod makra SDDR(x) (DDR##x) preprocesor podstawi w miejsce x argument, jakim będzie litera portu i sklei ją z napisem DDR. Zapis do portów także bardzo się uprości, np.

```
// wyzerowanie wszystkich linii sterujących
PORT(LCD_RSPORT) &= ~(1<<LCD_RS);
PORT(LCD_EPORT) &= ~(1<<LCD_E);
#if USE_RW == 1
     PORT(LCD_RWPORT) &= ~(1<<LCD_RW);
#endif
```

Tutaj makra PORT(x) działają analogicznie jak wyżej opisane DDR(x). Przy okazji mamy kolejny fragment inicjalizacji, ponieważ dokonujemy domyślnie wyzerowania stanu wszystkich linii sterujących RS, RW oraz E. Jakby nie patrzeć, mamy już „kawałek” pracy za sobą. Takie przygotowanie plików nagłówkowych, *(chociaż to jeszcze nie koniec)* też wymaga pracy. Jednakże trzeba ją wykonać raz i porządnie, dzięki czemu później nie będzie trzeba wracać do szczegółów, za to osiągniemy dwa cele. Napiszemy własną bibliotekę obsługi LCD oraz poznamy od „podszewki”, jak programuje się tego typu układy. W związku z tym, że bardzo często będziemy korzystali z ustawiania naszych trzech linii sterujących w stan wysoki lub niski, przygotujemy sobie także makra do tego celu:

```
// makrodefinicje operacji na sygnałach sterujących RS,RW oraz E
```

```
// stan wysoki na linii RS
#define SET_RS PORT(LCD_RSPORT) |= (1<<LCD_RS)
// stan niski na linii RS
#define CLR_RS PORT(LCD_RSPORT) &= ~(1<<LCD_RS)

// stan wysoki na RW - odczyt
#define SET_RW PORT(LCD_RWPORT) |= (1<<LCD_RW)
// stan niski na RW - zapis
#define CLR_RW PORT(LCD_RWPORT) &= ~(1<<LCD_RW)

// stan wysoki na linii E
#define SET_E       PORT(LCD_EPORT) |= (1<<LCD_E)
// stan niski na linii E
#define CLR_E       PORT(LCD_EPORT) &= ~(1<<LCD_E)
```

Musimy także przygotować sobie prostą funkcję, która będzie miała za zadanie przesłać nam do wyświetlacza LCD połówkę bajtu tak, żeby odpowiednie bity połówki trafiły na odpowiednie piny linii wyświetlacza D7..D4:

```
static inline void lcd_sendHalf(uint8_t data)
{
     if (data&(1<<0)) PORT(LCD_D4PORT) |= (1<<LCD_D4);
     else PORT(LCD_D4PORT) &= ~(1<<LCD_D4);

     if (data&(1<<1)) PORT(LCD_D5PORT) |= (1<<LCD_D5);
     else PORT(LCD_D5PORT) &= ~(1<<LCD_D5);

     if (data&(1<<2)) PORT(LCD_D6PORT) |= (1<<LCD_D6);
     else PORT(LCD_D6PORT) &= ~(1<<LCD_D6);

     if (data&(1<<3)) PORT(LCD_D7PORT) |= (1<<LCD_D7);
     else PORT(LCD_D7PORT) &= ~(1<<LCD_D7);
}
```

Po pierwsze, przed funkcją występuje słówko static, co oznacza, że nie będzie ona udostępniana na zewnątrz do innych modułów. Po drugie, występuje słówko inline, co oznacza, że sugerujemy, aby kompilator funkcję tę przerobił na makro, którego kod będzie za każdym razem rozwijany w miejscu wywołania funkcji. Decyzję taką podejmujemy, aby nie tracić czasu na skok do funkcji i powrót. Dane muszą jak najszybciej pojawić się na pinach LCD. Argumentem funkcji jest jeden bajt o nazwie data. Za każdym razem wewnątrz funkcji w warunkach if() sprawdzane są po kolei jego cztery najmłodsze bity D3..D0 i w zależności od ich wartości oddzielnie ustawiane linie sterujące liniami danych do LCD. Robimy tak (każda linia z osobna), ponieważ każda z tych linii może być zdefiniowana na różnym porcie czy pinie mikrokontrolera. Nie można zatem skorzystać ze znanego już wcześniej sposobu:

```
PORTB = (PORTB & 0xF0)  | (data & 0x0F);
```

Moglibyśmy tak zrobić, gdyby założenia do naszej obsługi LCD przewidywały, iż wymuszamy na siłę podłączania zawsze linii danych D7..D7 wyświetlacza to dowolnego portu, ale zawsze jego najstarszych bitów, także D7..D4. Wtedy mając do czynienia np. z mikrokontrolerem ATmega8 nie można byłoby nigdy użyć portu C do tego celu, gdyż posiada on tylko do dyspozycji piny PC5..PC0.

Skoro otrzymaliśmy już podstawową funkcję do przesyłania połówki bajtu, to teraz możemy spokojnie napisać prostą funkcję do przesyłania całego bajtu:

```
void _lcd_write_byte(unsigned char _data)
{
      // Ustawienie pinów portu LCD D4..D7  jako wyjścia
      data_dir_out();

#if USE_RW == 1
      CLR_RW;
#endif

      SET_E;
      lcd_sendHalf(_data>>4);// wysłanie starszej części bajtu danych D7..D4
      CLR_E;

      SET_E;
      lcd_sendHalf(_data); // wysłanie młodszej części bajtu danych D3..D0
      CLR_E;

#if USE_RW == 1
      while( check_BF() & (1<<7) );
#else
      _delay_us(120);
#endif
}
```

Tutaj jest kilka ciekawostek. Do funkcji przekazujemy już oczywiście cały bajt o nazwie _data. W pierwszej linii ustawiamy sposobem opisanym wyżej wszystkie linie danych LCD jako wyjścia. Tyle, że zebraliśmy znowu te cztery linijki w jedną funkcję typu static inline. Następnie stosujemy sprawdzanie za pomocą dyrektywy preprocesora #if, czy zdeklarowane jest używanie pinu RW podłączonego do procesora, czy do GND. Jeśli do procesora, to RW będzie równe 1 i dzięki temu skompilowana zostanie także linia zerująca sygnał RW. W kolejnej części funkcji widać już jak na dłoni przesłanie dwóch połówek naszego bajtu _data przy użyciu wcześniej zdefiniowanej funkcji lcd_sendHalf(). Z tym, że najpierw musimy wysłać do LCD starszą część bajtu, dlatego przesyłamy (_data >> 4) korzystając uprzednio z przesunięcia starszej części do młodszych bitów D3..D0, gdyż pamiętamy, że funkcja ta przesyła dalej tylko 4 młodsze bity. Za każdym razem przed wysłaniem połówki bajtu zgodnie z wykresami czasowymi ustawiamy sygnał sterujący E w stan wysoki, a po przesłaniu w stan niski.

Dalej znowu mamy warunek preprocesora #if. Jeśli okaże się, że korzystamy z sygnału RW podłączonego do mikrokontrolera, to linia while(check_BF() & (1<<7)); powoduje, że następuje oczekiwanie na flagę zajętości sterownika LCD. Jeśli będzie ona równa jeden, to oznacza, iż zakończył on swoje wewnętrzne działania i można przesyłać do niego kolejne dane. Właśnie taki jest sposób postępowania ze sprawdzaniem Busy Flag. Za chwilę omówimy dokładnie tę funkcję. Jeśli jednak warunek preprocesora nie jest spełniony, czyli RW = 0, a sygnał RW podłączony na stałe do GND, to nie mamy jak sprawdzać stanu zajętości sterownika HD44780. Na szczęście, w notach aplikacyjnych PDF są zwykle podane czasy, jakich trzeba przestrzegać wprowadzając zwykłe opóźnienia, żeby sterownik zdążył wykonać wewnętrzne operacje po odebraniu bajtu. Zatem czas potrzebny na obsłużenie takiego bajtu to minimum 120us. Takie opóźnienie za pomocą _delay_ms() stosujemy w naszej funkcji.

Mamy już funkcję przesyłającą bajt, ale trzeba jeszcze za pomocą stanu linii RS określać, czy przesyłamy komendę do sterownika, czy też kod znaku do wyświetlenia. Idąc tym tropem przygotowujemy dwie kolejne funkcje wyższego rzędu:

```
void lcd_write_cmd(uint8_t cmd)
{
      CLR_RS;
      _lcd_write_byte(cmd);
}

void lcd_write_data(uint8_t data)
{
      SET_RS;
      _lcd_write_byte(data);
}
```

Myślę, że te nie wymagają już komentarza. Wcześniej napracowaliśmy się troszkę nad utworzeniem podstawowych funkcji, żeby kolejne mogły być coraz bardziej uproszczone. W takim razie przyjrzyjmy się, jak można dokonać sprawdzenia stanu Busy Flag w sterowniku LCD. Zasada działania jest taka, że trzeba koniecznie, zgodnie z tabelą komend, zmienić stan sygnału RW na wysoki (dlatego musi on być podłączony i sterowany z procesora). Po tym trzeba odczytać ze sterownika bajt. Wewnątrz tego odczytanego bajtu dla nas będzie miał znaczenie tylko najstarszy bit nr 7. To jego wartość odpowiada Busy Flag. Inne możemy zupełnie zignorować. Zatem podobnie, jak pisaliśmy wcześniej funkcję do zapisu połówki bajtu, a później do zapisu całego bajtu, teraz musimy sytuację odwrócić i napisać najpierw funkcję do odczytu połówki bajtu z LCD, a następnie do odczytu całego bajtu, żeby na końcu można było sprawdzić stan bitu nr 7. Dzięki temu, jeśli zechcesz, możesz pójść dalej tą drogą i odczytywać w ogóle dane zapisywane do pamięci DDRAM lub CGRAM sterownika. Tutaj jednak tym się nie będziemy zajmować, ponieważ praktycznie nigdy się tego nie wykorzystuje. Odczyt połówki bajtu:

```
#if USE_RW == 1
static inline uint8_t lcd_readHalf(void)
{
      uint8_t result=0;

      if(PIN(LCD_D4PORT)&(1<<LCD_D4)) result |= (1<<0);
      if(PIN(LCD_D5PORT)&(1<<LCD_D5)) result |= (1<<1);
      if(PIN(LCD_D6PORT)&(1<<LCD_D6)) result |= (1<<2);
      if(PIN(LCD_D7PORT)&(1<<LCD_D7)) result |= (1<<3);

      return result;
}
#endif
```

Spójrz, tym razem cała funkcja zawarta jest w warunku #if preprocesora, a to tylko po to, żeby w razie czego jej nie kompilować i nie zajmować miejsca w pamięci FLASH, jeśli mamy wyświetlacz podłączony z sygnałem RW do GND na stałe. Funkcja zwraca nam tym razem rezultat, gdyż zgodnie z nazwą odczytuje półbajt. Wewnątrz deklarujemy zmienną automatyczną *(lokalną)* o nazwie result oraz koniecznie inicjalizujemy ją zerem *(gdyż zmienne automatyczne, tworzone na stosie nie są inicjalizowane tak jak globalne zerami)*. Następnie mamy znowu cztery warunki if(), które w działają w odwrotny sposób niż przy

funkcji zapisu półbajtu. Najpierw sprawdzają kolejno stan panujący na liniach sterownika na poszczególnych bitach, i jeśli jest to stan wysoki, to zostaje ustawiony odpowiadający mu bit w młodszej części zmiennej result. Zera nie musimy ustawiać, gdyż już cała zmienna została zainicjalizowana zerami na pozycji każdego bitu. Na koniec, poleceniem return zwracamy odczytany półbajt, który znajduje się w młodszej części wyniku.

```
#if USE_RW == 1
uint8_t _lcd_read_byte(void)
{
      uint8_t result = 0;
      data_dir_in();

      SET_RW;

      SET_E;
      // odczyt starszej części bajtu z LCD D7..D4
      result |= (lcd_readHalf() << 4);
      CLR_E;

      SET_E;
      // odczyt młodszej części bajtu z LCD D3..D0
      result |= lcd_readHalf();
      CLR_E;

      return result;
}
#endif
```

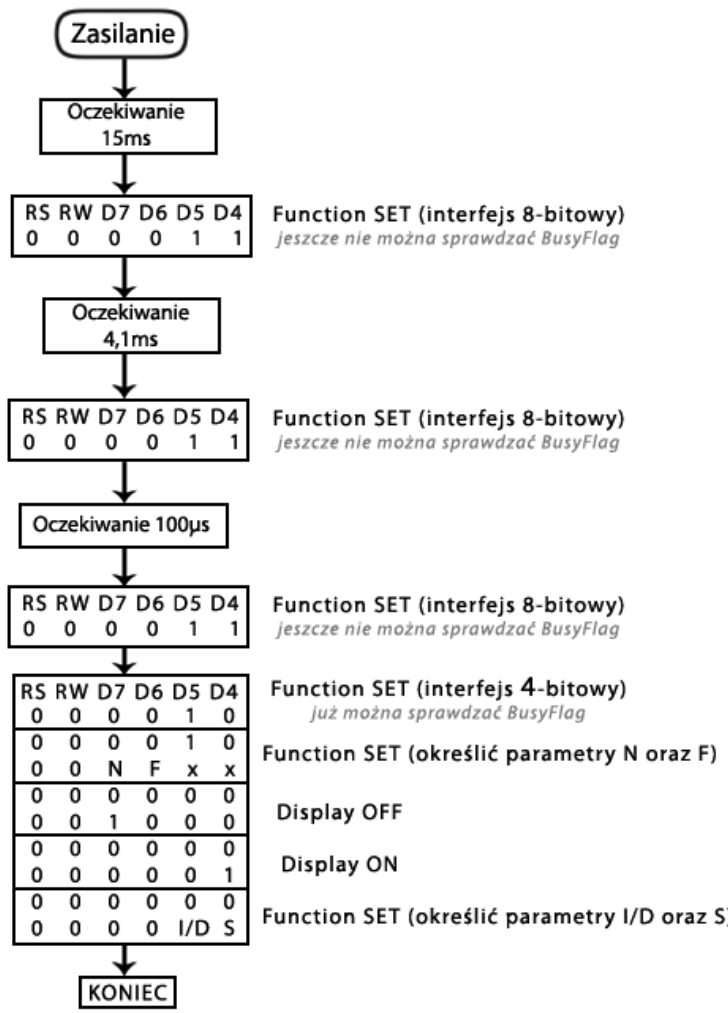
Podobnie ta funkcja nie będzie kompilowana w przypadku zadeklarowanego podłączenia sygnału RW do GND na stałe. Na początku także definiujemy zmienną result wraz z inicjalizacją zerem dla porządku. Ustawiamy piny procesora podłączone do linii danych sterownika jako wejścia, a także wcześniej zdefiniowaną funkcją typu static inline. Nie będę już jej podawał, ponieważ jest praktycznie taka sama jak data_dir_out(), tyle że zamiast ustawiać bity rejestrów DDRx na jedynki, tu ustawiamy je na zera.

W tej funkcji już bez sprawdzania warunku #if USE_RW == 1 ustawiamy stan linii RW na wysoki. A następnie analogicznie jak w funkcji do zapisu, tutaj dokonujemy odczytu dwóch półbajtów, zaczynając od starszego: `result |= (lcd_readHalf() << 4);`. Jednocześnie dokonujemy od razu przesunięcia czterech młodszych bitów w lewo na pozycję starszych bitów, ponieważ sterownik także w odpowiedzi przy magistrali 4-bitowej najpierw przesyła starszą część bajtu. Teoretycznie można się zastanawiać, dlaczego na tym nie zakończymy tej funkcji skoro mamy już w starszej części bajtu wartość Busy Flag na bicie nr 7. Jednak zakończyć nie można, ponieważ sterownik na siłę będzie chciał wysłać młodszą część bajtu, musimy ją zatem odebrać, żeby nie doszło do kolizji z następnymi operacjami. Drugą, młodszą część półbajtu także zapisujemy za pomocą sumy logicznej (OR) do zmiennej result, mamy więc już cały odebrany bajt i możemy go zwrócić w postaci rezultatu funkcji za pomocą polecenia return. Teraz już napisanie naszej funkcji check_BF(), którą widziałeś wcześniej, jest bajecznie proste:

```
#if USE_RW == 1
uint8_t check_BF(void)
{
      CLR_RS;
```

```
        return _lcd_read_byte();
}
#endif
```

Jak zwykle kompilacja nastąpi tylko wtedy, gdy RW podłączone jest do mikrokontrolera, przed odczytem ustawiamy linię RS w stan niski zgodnie z tabelą polecenia odczytu flagi zajętości, i jako rezultat funkcji zwracamy cały odczytany bajt. Po czym, jak wspomniano wyżej, w warunku powodującym oczekiwanie na stan flagi zajętości, `while( checkBF() & (1<<7) );` maskujemy iloczynem bitowym AND pozostałe bity, gdyż interesuje nas tylko stan siódmego BF. Teraz muszę przedstawić jeden z algorytmów inicjalizacji wyświetlacza umożliwiający jego pracę w trybie 4-bitowym.



Napiszmy, więc zgodnie z podanym algorytmem naszą funkcję inicjalizacyjną:

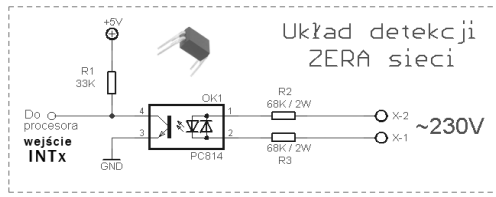
```
void lcd_init(void)
{
        // inicjowanie pinów portów ustalonych do podłączenia z wyświetlaczem LCD
        // ustawienie wszystkich jako wyjścia
```

„**ATH1**”, spowodującym, że nasze moduły nie będą ujawniały się w otoczeniu BT. Zatem nawet potencjalny haker nie będzie w stanie namierzyć naszych urządzeń. Uwaga! Po zastosowaniu polecenie „**ATH1**”, należy odczekać minimum 15 sekund, aby odniosło ono zamierzony skutek. Pamiętaj o tym, gdy zastosujesz tę opcję, ponieważ nie będzie można już wykryć modułu nawet za pomocą własnego komputera. Niemniej jednak nie ma obaw. Można przywrócić możliwość wykrywania poleceniem „**ATH0**”. Pozostaje jeszcze sporo innych poleceń AT, o których już spokojnie sam możesz doczytać w nocie PDF. Opisałem te, które mają największe znaczenie ,oraz niektóre całkowicie pominięte w nocie, jak np.: „**+++**” czy też „ATO”. Podobnie niezbyt jasno wynika z noty PDF, jak uruchomić tryb Master oraz co powoduje niewyłączenie echa. Mam nadzieję, że ten pakiet informacji okaże się bardzo pomocny i spokojnie zaczniesz używać transmisji Bluetooth w swoich projektach. Niestety nie jestem w stanie w ramach tej książki zmieścić sposobów komunikacji modułów z telefonami komórkowymi oraz metod, jak napisać proste aplikacje w języku Java, które umożliwiają pełnowartościowe wykorzystanie takiej komunikacji do własnych celów. Być może poświęcę temu zagadnieniu nieco więcej miejsca w kolejnej książce, gdyż uważam, że także zasługuje na dobrą i pełną prezentację od podstaw. Na zakończenie dodam, iż w modułach BTM-xxx w ogóle nie działają linie CTS oraz RTS. Wprawdzie na niektórych schematach spotkasz się z tym, że są one w jakiś sposób połączone, jednak zapewniam, że ktoś, kto narysował taki schemat, po prostu powielił błędny znaleziony w Internecie. Podobnie nie ma najmniejszego sensu używanie i wyprowadzanie linii RESET z modułu. Tutaj także pokutuje na zasadzie plotek internetowych opinia, jakoby czasem układ potrafił się zawiesić i dobrze jest mieć możliwość jego restartu tym sygnałem. Możesz tego typu opinie włożyć między bajki. Poza tym nie działają również żadne inne linie modułu. Być może są jakieś możliwości ich wykorzystania, jednak dokumentacja producenta milczy na ten temat.

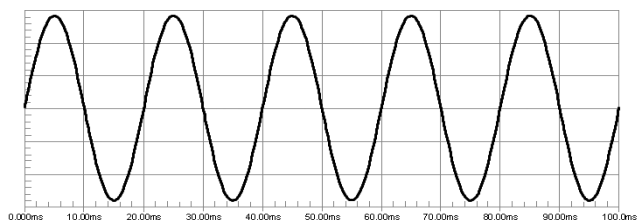
8.3 Ściemniacz – płynna regulacja mocy 230V

Zastanawiałem się, czy poruszyć ten temat, głównie ze względu na duży stopień niebezpieczeństwa podczas konstruowania układów mających styczność z napięciem sieci elektrycznej. Testy w tym zakresie polecam tylko osobom zaawansowanym, którym tego typu konstrukcje nie są obce. Pamiętaj, że niewielka błąd może zakończyć się porażeniem prądem lub awarią zasilania w całym mieszkaniu lub warsztacie. Ale też tą tematyką interesuje się spora ilość początkujących elektroników, a rozwiązanie programowe, które przedstawię, jest bajecznie proste i pozwala na wielokanałowe sterowanie urządzeniami z jednego mikrokontrolera AVR. Główny kod odpowiedzialny za funkcje regulacji zajmie nie więcej niż 300-400bajtów pamięci Flash. W Internecie można wprawdzie znaleźć wiele różnorakich rozwiązań, które jednak w opinii wielu użytkowników bądź to nie sprawdzają się w praktyce, bądź są zbyt skomplikowane, jeśli chodzi o strukturę kodu programu, trudne do zrozumienia oraz do uruchomienia we własnych układach, na innym mikrokontrolerze i przy innej częstotliwości taktowania. Zajmiemy się przygotowaniem absolutnie uniwersalnego kodu, który uruchomisz bez problemów w swoim układzie, niezależnie, jaką częstotliwością będzie taktowany twój mikrokontroler *(oczywiście w granicach rozsądku, co oznacza dla mnie częstotliwości od 1MHz do 20MHz)*. Dodam, że nie będzie trzeba nic w związku z tym przeliczać we własnym zakresie, zmusimy do tego preprocesor, niech on się męczy. Zacząć jednak musimy od prezentacji precyzyjnego schematu elektronicznego oraz kilku słów teorii wyjaśniającej, na czym polega „sterowanie fazowe”, bo takim się zajmiemy. Istnieje także „sterowanie grupowe”, ale o tym w skrócie później. Jak zapewne zadajesz sobie świetnie sprawę, napięcie zmienne znajdujące się w naszych domowych gniazdkach elektrycznych generowane jest w postaci sinusoidy z częstotliwością równą 50Hz. Jest ona w miarę dokładna, dlatego używana jest nawet przez niektóre zegarki, jako wzorzec czasu.

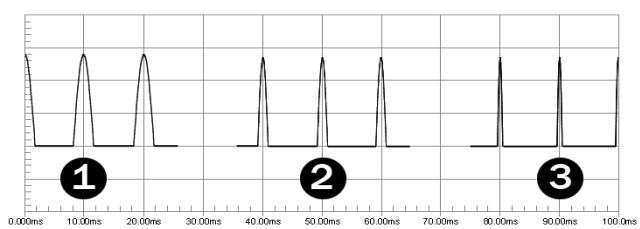
Sterowanie fazowe polega na tym, że za pomocą mikrokontrolera będziemy ograniczali prąd/napięcie w każdej połówce sinusoidy. Na podobnej zasadzie jak w przypadku sterowania PWM. Tyle, że tutaj to nie my generujemy sygnał 50Hz. Otrzymujemy go na wejściu, musimy się z nim zsynchronizować, aby precyzyjnie od początku każdej połówki rozpocząć załączanie/wyłączanie *(w całym jej obszarze)*.Pierwsza rzecz, jaka będzie nam potrzebna, to układ detekcji *ZERA* sieci. W Internecie znajdziesz mnóstwo rozwiązań. Proponuję ci w pełni sprawdzony i uruchomiony układ:



Chciałbym zwrócić szczególną uwagę na sposób dobierania elementów oraz ich wpływu na parametry pracy układu detekcji. Znaczenie w pierwszej kolejności ma rezystor R1 o wartości 33K. Ta wartość jest krytyczna. Zmniejszenie rezystora np. do 22K spowoduje szersze „szpilki" impulsów. Im szersza szpilka, jak zaraz się przekonasz, tym gorsze parametry pracy całego układu regulatora mocy.



Sinusoida 230V



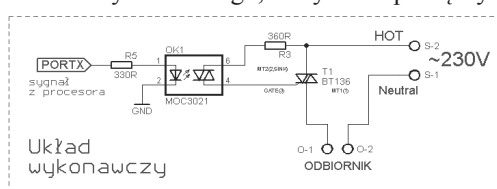
Detekcja przejścia przez ZERO

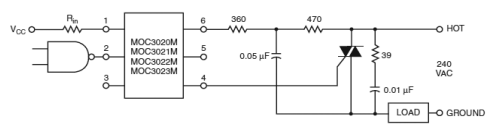
Układ przewidziany został do pracy z napięciem +5V. Aby nasz układ nie miał styczności z napięciem 230V, użyjemy optoizolatora PC814. Jest on łatwo dostępny, więc nie będzie problemów z jego zdobyciem. Jak widać ze schematu, układ scalony posiada wewnątrz dwie diody połączone przeciwsobnie. Dzięki temu mamy załatwione dwie sprawy jednocześnie. Po pierwsze, diody nawzajem stanowią dla siebie zabezpieczenie, po drugie, na wyjściu układu otrzymujemy impuls „szpilkę" o bardzo krótkim czasie trwania dla każdego przejścia przez *ZERO*. Uwaga! Rezystory R2 oraz R3 muszą być odpowiedniej mocy, w tym przypadku koniecznie minimalnie 2W. Nie stosuj mniejszych mocy. Natomiast wartości rezystorów nie są już tak krytyczne. Mogą one być w przedziale 47K do 100K. Ja użyłem wartości ze środka tego przedziału. Niestety im większa wartość tych rezystorów, tym szerszy będzie impuls na wyjściu, natomiast im mniejsza wartość, tym impuls będzie krótszy, co jest bardzo pożądane z punktu widzenia obsługi programowej. Niestety „coś za coś" i im mniejsze rezystory, np. 47K, tym więcej ciepła będzie się na nich wydzielać, natomiast im większe, np. 100K, tym

mniej ciepła. Przy budowie w/w urządzenia rezystory 68K, które zastosowałem były lekko ciepłe. Nie wpłynie to jednak na ich uszkodzenie, jeśli ich moc = 2W. Rysunek powyżej przedstawia kształt sinusoidy sieci 230V, jaką można zaobserwować na ekranie oscyloskopu. Poniżej pokazałem, jak wyglądają impulsy na wyjściu naszego układu detekcji *ZERA sieci*. Przedstawiłem trzy warianty oznaczone punktami 1,2 i 3.

- Wariant **1** – uzyskamy stosując np. R1=10K, R2, R3=100K
- Wariant **2** – uzyskamy stosując np. R1=33K, R2, R3=68K
- Wariant **3** – uzyskamy stosując np. R1=33K, R2, R3=47K

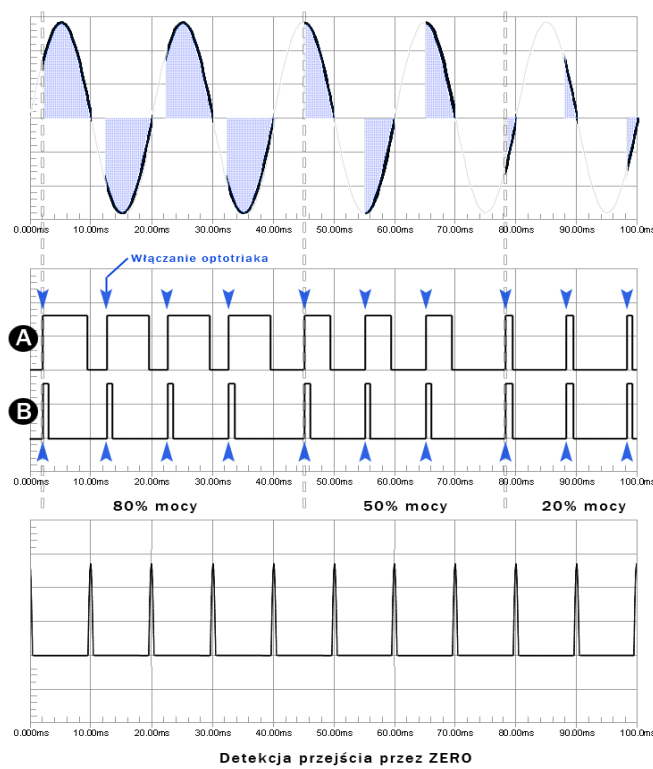
Z programistycznego punktu widzenia zadowoli nas już wariant 2, gdy szerokość impulsu będzie wynosiła ok. 1ms *(plus, minus 0,2ms)*. Zwykle problemem wielu elektroników amatorów jest jednak brak oscyloskopu, co uniemożliwia podejrzenie na jego ekranie szerokości generowanych impulsów, aby prawidłowo skalibrować cały układ. Zapewniam cię jednak, że za chwilę przekonasz się, jak można uruchomić całość bez posiadania oscyloskopu. Podam dokładną procedurę kalibracji bez jego użycia. Na pewno sobie poradzisz. Jak się zapewne domyślasz, impulsy detekcji ZERA sieci są dla nas niezbędne po to, aby załączać każdą połówkę sinusoidy w dowolnym momencie. Jeśli załączymy ją bezpośrednio po wykryciu impulsu to triak sterujący będzie przesuszał prąd/napięcie przez cały okres jej trwania *(10ms)*, co spowoduje, iż urządzenie będzie otrzymywało 100% mocy. Pełna sinusoida. Zanim jednak przyjrzymy się, jak sterować zwykłym załączaniem bądź wyłączaniem odbiornika 230V, spójrzmy na schemat modułu wykonawczego, który także podłączymy do mikrokontrolera.





Moduł wykonawczy jest równie prosty, nie pomyl się jednak podczas podłączania triaka. Sprawdź dobrze notę PDF, aby podłączyć odpowiednie końcówki w odpowiednie miejsce. Złe podłączenie może skutkować albo zwarciem (bardzo groźne) lub po prostu tym, że układ będzie działał nieprawidłowo. Przedstawiłem dwa różne schematy. Pierwszy w ramce otoczonej przerywaną linią jest uproszczony do minimum. Drugi natomiast pochodzi wprost z noty aplikacyjnej optotriaków serii MOC302x. Schemat uproszczony nie posiada przede wszystkim tzw. „gasika", tj. układu wytłumiającego w podstawowy sposób zakłócenia pojawiające się na wyjściu triaka podczas przełączania. Rolę gasika na schemacie z noty PDF pełni para elementów po prawej stronie triaka, czyli rezystor 39R *(Ohm)* oraz kondensator ceramiczny 10nF *(400V)*. Gdyby twoje urządzenie miało pracować w pobliżu np. sprzętu RTV, warto taki „gasik" zastosować. Rezystor w tym układzie może mieć moc 1/4W. W praktyce ocenisz, czy będzie ci potrzebny układ gasika, czy nie. Układ z noty PDF posiada jeszcze dodatkowy rezystor 470R oraz kondensator 50nF. Para tych elementów to układ „gasika", ale dla optotriaka. W praktyce w konstrukcjach amatorskich rzadko korzysta się z

„gasików”, jeśli układ nie wprowadza specjalnych zakłóceń do otoczenia, w którym pracuje. Zastosowaliśmy optotriak bez wbudowanej wewnątrz detekcji przejścia przez *ZERO sieci*. Jest to konieczne, ponieważ do naszych celów musimy posiadać własny niezależny układ detekcji. A triaki posiadające wbudowany układ detekcji (seria MOC304x) bardziej przydatne są, gdy trzeba dokonać tylko prostego sterowania typu WŁĄCZ/WYŁĄCZ. Dzięki temu, że potrafią wykryć ZERO sieci, dokonują załączania triaka automatycznie podczas właśnie samego przejścia sinusoidy przez ZERO, co diametralnie zmniejsza ilość generowanych zakłóceń na wyjściu w porównaniu do układów MOC302x. Jednak nie nadają się one przez to do naszych celów. Możesz, zatem zakupić dowolny optotriak z serii MOC302x, ja użyłem takiego jak na schemacie, czyli MOC3021. Układ ten pozwala także na wyjściu w pełni odizolować nasz układ od niebezpiecznego napięcia sieci. Sterowanie takim modułem jest banalne w przypadku zwykłego włączania/wyłączania. Jeśli podamy stan wysoki na anodę diody wewnątrz optotriaka, spowoduje to załączenie triaka na stałe. *(W rzeczywistości, ponieważ, triak sam rozłącza się podczas przejścia sinusoidy przez ZERO, to jednak nasz włączony optotriak ciągle go uaktywnia na początku każdej połówki sinusoidy)*. Jednakże nasze zadanie będzie zgoła inne. Mając już na wejściu sygnał w postaci „impulsu szpilki” z naszego układu detekcji ZERA, za pomocą kodu programu będziemy w odpowiednim momencie załączali triaka. Nie zawsze jednak na początku połówki, ale także w trakcie jej trwania, by zmniejszyć moc dostarczaną do sterowanego urządzenia. W przypadku podłączonej żarówki, będziemy obserwowali płynne jej ściemnianie i rozjaśnianie, jeśli prawidłowo to oprogramujemy.



Teraz kilka słów na temat podłączenia omawianych modułów z mikrokontrolerem. Wyjście układu detekcji ZERA sieci podłączymy wprost do dowolnego wejścia przerwania typu INTx. Ja wybrałem do testów INT0. Z uwagi na to, że nasza „szpilka” to impuls, który zmienia swój stan z niskiego na wysoki i bardzo szybko z powrotem z wysokiego na niski, to trzeba