

Jarosław Doliński

mikrokontrolery AVR *w praktyce*

Książka jest kompletnym przewodnikiem po rodzinie mikrokontrolerów AVR, ze szczególnym uwzględnieniem mikrokontrolera AT90S2313. Czytelnik znajdzie w książce informacje o budowie i architekturze mikrokontrolerów AVR, sposobach ich programowania, zalecanych warunkach pracy, budowie i działaniu modułów peryferyjnych, a także parametrach elektrycznych i czasowych. Szczegółowo przedstawiono listę rozkazów asemblera, programy narzędziowe (m.in. AVR Studio oraz VM Lab) i kompilatory (w tym bezpłatny kompilator języka C – AVR-GCC). Praktyków szczególnie zainteresują przykładowe projekty, opisane w ramach 11 ćwiczeń laboratoryjnych (wszystkie programy napisano w języku C). Oprócz przykładów klasycznych (jak np. obsługa alfanumerycznego wyświetlacza LCD, klawiatury, czy interfejsu RS232) przedstawiono także wiele przykładów nowoczesnych, jak choćby ilustrację sposobu dołączenia mikrokontrolera AVR do interfejsu USB, współpracę mikrokontrolerów AVR z układami wyposażonymi w interfejs 1-Wire (*iButton*), czy regulowanie obrotów silników DC za pomocą przebiegów PWM.

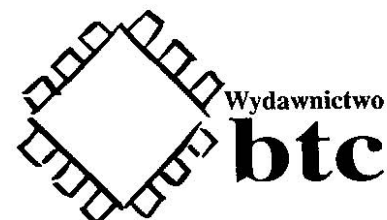
W książce przedstawiono opis programatora ISP, który umożliwia programowanie pamięci mikrokontrolerów AVR po zainstalowaniu ich w docelowym urządzeniu oraz zestawu ewaluacyjnego, który służył do zweryfikowania wszystkich projektów opisanych w książce.

Programy źródłowe do wszystkich przykładów opisanych w książce są dostępne na stronie internetowej <http://www.btc.pl/index.php?id=avr>.

Redaktor merytoryczny: Krzysztof Powął

ISBN 83-910067-6-X

© Copyright by Wydawnictwo BTC
Warszawa 2003.



Wydawnictwo BTC
ul. Inowłodzka 5
03-237 Warszawa
fax: (22) 782-42-90
<http://www.btc.pl>
e-mail: redakcja@btc.pl

Wydanie 1. Warszawa 2003.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz wydawnictwo BTC dolożyli wszelkich starań, by zawarte w tej książce 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 BTC nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentów niniejszej publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Druk i oprawa: Łódzka Drukarnia Dzielowa S.A.

Spis treści

Część 1. Informacje podstawowe

Wstęp	8
1. Trochę historii	9
2. Charakterystyka mikrokontrolerów AVR	11
3. Rodzina AVR - podstawy	13
3.1. Funkcje wyprowadzeń	16

Część 2. Budowa i działanie mikrokontrolerów AVR

4. Architektura mikrokontrolerów AVR	20
4.1. Generator taktujący	23
4.2. Rejestry ogólnego przeznaczenia	24
4.3. Jednostka arytmetyczno-logiczna (ALU)	26
4.4. Pamięć programu	26
4.5. Nieulotna pamięć danych EEPROM	26
4.6. Pamięć danych SRAM	27
4.7. Tryby adresowania pamięci danych i pamięci programu	28
4.7.1. Tryb bezpośredniego adresowania rejestrów wykorzystujący pojedynczy rejestr	28
4.7.2. Tryb bezpośredniego adresowania rejestrów wykorzystujący dwa rejestry	29
4.7.3. Tryb bezpośredniego adresowania obszaru wejścia/wyjścia	30
4.7.4. Tryb bezpośredniego adresowania pamięci danych	30
4.7.5. Tryb pośredniego adresowania danych z przemieszczeniem	31
4.7.6. Tryb adresowania pośredniego	32
4.7.7. Tryb adresowania pośredniego danych z predekrementacją	32
4.7.8. Tryb adresowania pośredniego danych z postinkrementacją	33
4.7.9. Tryb adresowania stałych z użyciem rozkazu LPM	34
4.7.10. Tryb adresowania pośredniego pamięci programu (IJMP, ICALL)	34
4.7.11. Tryb adresowania względnego pamięci programu (RJMP i RCALL)	35
4.8. Przebiegi czasowe podczas dostępu do pamięci i wykonywania rozkazów	36
4.9. Przestrzeń we/wy	38
4.9.1. Funkcje bitów w rejestrach funkcyjnych	40
4.10. Zerowanie i wektory przerw	42
4.10.1. Źródła sygnału zerującego	44
4.10.2. Uchwyty przerw	47
4.10.3. Przerwania zewnętrzne	53
4.10.4. Czas odpowiedzi na zgłoszenie przerwania	54

4.11. Tryby oszczędzania energii	56
4.11.1. Tryb Idle	57
4.11.2. Tryb Power-Down	57
5. Timery/liczniki	59
5.1. 8-bitowy Timer/Licznik0	60
5.2. 16-bitowy Timer/Licznik1	62
5.3. Timer/Licznik1 w trybie PWM	71
6. Watchdog	75
7. Pamięć danych EEPROM	79
7.1. Zapis i odczyt pamięci	79
7.2. Zapewnienie prawidłowych warunków pracy pamięci EEPROM	84
8. Układ transmisji szeregowej (UART)	85
8.1. Budowa i działanie nadajnika UART	86
8.2. Budowa i działanie odbiornika UART	87
8.3. Sterowanie transmisją	90
8.4. Generator podstawy czasu transmisji (Baud Rate Generator)	93
9. Komparator analogowy	96
10. Porty wejścia-wyjścia (I/O)	99
10.1. Budowa portu B	99
10.1.1. Port B jako cyfrowy port we/wy ogólnego przeznaczenia	101
10.1.2. Funkcje alternatywne portu B	102
10.1.3. Budowa linii portu B	103
10.2. Budowa portu D	106
10.2.1. Port D jako cyfrowy port we/wy ogólnego przeznaczenia	108
10.2.2. Funkcje alternatywne portu D	108
10.2.3. Budowa linii portu D	110
11. Pamięci nieulotne w mikrokontrolerach AVR	113
11.1. Bity zabezpieczające pamięć programu i danych	114
11.2. Bity konfiguracyjne	115
11.3. Sygnatury	116
11.4. Programowanie pamięci Flash i EEPROM	116
11.4.1. Programowanie równoległe	116
11.4.2. Programowanie szeregowe	123

Część 3. Lista rozkazów

12. Zestawienie rozkazów mikrokontrolera AT90S2313	128
12.1. Opis działania rozkazów	136

Część 4. Narzędzia i projekty przykładowe

13. Narzędzia projektowe	278
13.1. AVR Assembler for Windows	279
13.2. Kompilator języka C - AVR-GCC wersja 3.2	283
13.2.1. Instalacja kompilatora	284

13.3. AVR Studio wersja 3.56	284
13.3.1. Przygotowanie programów pisanych w assemblerze do symulacji w AVR Studio 3.56	285
13.3.2. Integracja programu AVR Studio 3.56 z kompilatorem AVR-GCC	287
13.3.3. Symulacja programów w AVR Studio V. 3.56	290
13.4. Symulator Visual Micro Lab 3.56	295
13.5. Programowanie pamięci programu w systemie (ISP)	300
13.5.1. Programator ZL2PRG	300
14. Przykładowe aplikacje	303
14.1. Zestaw uruchomieniowy ZL1AVR	304
14.1.1. Zasilanie zestawu	307
14.1.2. Taktowanie i zerowanie mikrokontrolera	307
14.1.3. Wykorzystywanie portów mikrokontrolera	308
14.1.4. Klawiatura	309
14.1.5. Interfejs RS232	310
14.1.6. Diody LED	310
14.1.7. Wyświetlacz alfanumeryczny LCD	311
14.1.8. Interfejsy I ² C i 1-Wire	311
14.1.9. Przetwornik analogowo-cyfrowy	312
14.1.10. Programowanie mikrokontrolera w systemie (ISP)	315
14.2. Ćwiczenia praktyczne	317
14.2.1. Ćwiczenie 1	317
<i>Sterowanie portami mikrokontrolera w trybie wyjściowym - efekt węzła świetlnego i biegnącego punktu na linijce diod LED</i>	
14.2.2. Ćwiczenie 2	320
<i>Wykorzystanie timera do odmierzania czasu w trybie odpytywania (generator przebiegu prostokątnego o częstotliwości 1 kHz)</i>	
14.2.3. Ćwiczenie 3	323
<i>Sterowanie portami mikrokontrolera w trybie wejściowym, wykorzystanie timera do odmierzania czasu z wykorzystaniem przerwań - obsługa przycisków dołączonych do portów mikrokontrolera</i>	
14.2.4. Ćwiczenie 4	328
<i>„Hello World!”, czyli sterowanie wyświetlaczem alfanumerycznym LCD 16×2 i 16×1. Obsługa pojedynczego przycisku</i>	
14.2.5. Ćwiczenie 5	343
<i>„Łapanie muchy”, czyli obsługa klawiatury matrycowej z wykorzystaniem przerwań timera, obsługa wyświetlacza alfanumerycznego LCD 16×2</i>	
14.2.6. Ćwiczenie 6	351
<i>6-bitowy, binarny wskaźnik napięcia. Zastosowanie komparatora analogowego do budowy przetwornika analogowo-cyfrowego. Wyzwalanie funkcji przechwytywania Timer1 za pomocą komparatora. Przerwanie od przechwytywania Timer1. Obsługa wewnętrznej pamięci EEPROM</i>	

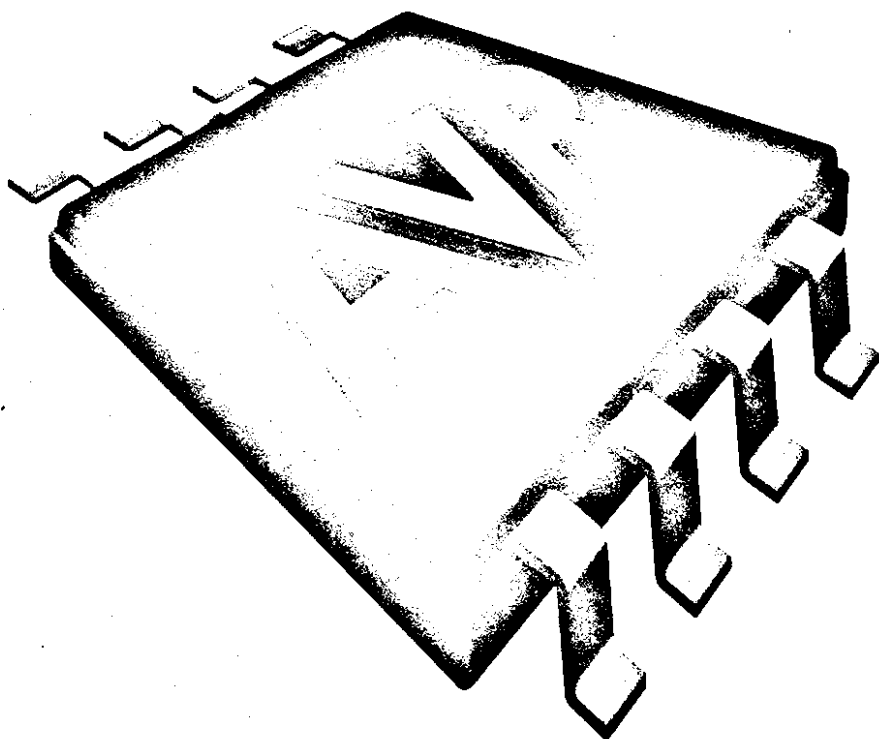
14.2.7. Ćwiczenie 7	357
<i>Regulacja obrotów silnika DC. Wykorzystanie Timer1 jako modulatora PWM. Obsługa pojedynczych klawiszy</i>	
14.2.8. Ćwiczenie 8	363
<i>Sterowanie obrotami silnika DC z komputera PC. Wykorzystanie Timer1 jako modulatora PWM. Wykorzystanie UART-a mikrokontrolera do prowadzenia transmisji szeregowej pomiędzy płytką ZL1AVR a komputerem PC</i>	
14.2.9. Ćwiczenie 9	374
<i>Obsługa interfejsu 1-Wire. Odczyt pastylki identyfikacyjnej Dallas - DS1990A. Obsługa wyświetlacza LCD 16×2</i>	
14.2.10. Ćwiczenie 10	388
<i>Obsługa interfejsu I²C. Obsługa przerwania zewnętrznego. Wykorzystanie układu PCF8583 (RTC - Real Time Clock) do budowy zegara 24-godzinnego. Obsługa wyświetlacza LCD 16×2</i>	
14.2.11. Ćwiczenie 11	408
<i>Podłączenie mikrokontrolera AVR do komputera PC przez port USB. Obsługa nadajnika i odbiornika UART z wykorzystaniem systemu przerwań. Obsługa wyświetlacza LCD 16×2</i>	

Dodatki

Dodatek A. Podstawowe parametry mikrokontrolerów z rodziny AVR	420
Dodatek B. Zestawienie rejestrów mikrokontrolera AT90S2313	422
Dodatek C. Wybrane charakterystyki elektryczne i czasowe mikrokontrolera AT90S2313	424
Dodatek D	
D.1. Dopuszczalne parametry elektryczne mikrokontrolera AT90S2313	428
D.2. Parametry czasowe zewnętrznego sygnału zegarowego	430
Dodatek E. Wyprowadzenia typowych wyświetlaczy LCD i VFD z interfejsem równoległym	431
Dodatek F. Instalacja sterowników dla układu FT8U232BM w systemie operacyjnym Windows	432
Dodatek G	
G.1. Płytką drukowaną zestawu ZL1AVR	439
G.2. Płytką drukowaną programatora ZL2PRG	442
Dodatek H. Wybrane adresy internetowe związane z mikrokontrolerami AVR	443
Dodatek I	
I.1. Tablica kodów ASCII	445
I.2. Znaki zawarte w generatorze znaków sterownika HD44870	447
Skorowidz	449

Część 1

Informacje podstawowe



Wstęp

Postęp techniczny i technologiczny, jaki obserwujemy od wielu lat na świecie, wymusza na konstruktorach konieczność ciągłego śledzenia pojawiających się nowości i uwzględniania ich w swojej pracy. Ilość informacji, przez jaką powinien przebrnąć na co dzień każdy inżynier, aby być „na bieżąco”, coraz częściej sięga granic możliwości. Dostęp do literatury technicznej, bez porównania lepszy niż choćby dziesięć lat temu, w dalszym ciągu wydaje się jednak daleki od wystarczającego. Świadczą o tym choćby często spotykane pytania „od czego zacząć, gdzie zdobyć podstawowe narzędzia” itp., bezustannie pojawiające się na różnych internetowych grupach dyskusyjnych oraz w listach do czasopism o tematyce elektronicznej.

Książka powstała w nadziei, że przynajmniej częściowo zaspokoi takie własne potrzeby. Omówiono w niej budowę mikrokontrolerów AVR, przedstawiono listę rozkazów, a wszystko to zilustrowano wieloma przykładami praktycznymi. Zdając sobie sprawę z olbrzymiego ich znaczenia dla zrozumienia tematu, przykłady umieszczono wszędzie tam, gdzie mogą się pojawić wątpliwości z interpretacją tekstu.

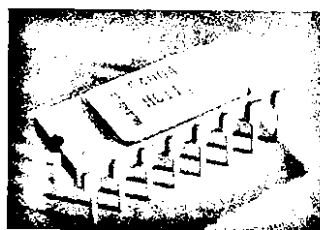
Konstruktorzy stosujący mikrokontrolery podzielili się na zwolenników programowania w assemblerze i językach wysokiego poziomu, jak np. C lub Basic. Zalety programowania w językach wysokiego poziomu są bezsporne. Trzeba jednak pamiętać, że pewnych problemów nie da się rozwiązać bez sięgnięcia po wstawki assemblerowe. Uważam, że także ci projektanci, którzy nastawiają się na programowanie w którymś z języków wysokiego poziomu, powinni znać assembler wykorzystywanego przez nich mikrokontrolera. Znajomość ta z pewnością przyczyni się do poznania budowy mikrokontrolera, ułatwi także zrozumienie sposobu jego działania. Z powyższych względów dla części przykładów zamieszczono kody źródłowe w języku C, dla pozostałych zaś w wersji assemblerowej. Zdecydowałem się na język C, chociaż budzi on nieuzasadnione obawy wśród początkujących programistów, gdyż niepodważalną zaletą przemawiającą za takim właśnie wyborem jest dostępność bezpłatnego kompilatora jakim jest AVR-GCC. Narzędzie to nie jest pozbawione pewnych wad i ułomności, co nie oznacza jednak, że nie można za jego pomocą stworzyć oprogramowania realizującego poważne zadania. Myślę, że w dobie walki z piractwem takie podejście do sprawy znajdzie zrozumienie Czytelników. Sposób pobrania kompilatora AVR-GCC z Internetu i jego instalacji opisałem w odrębnym rozdziale. Jestem przekonany, że Czytelnicy, którzy zostaną „zmuszeni” w ten sposób do nauki języka C nie będą tego żałować.

Od momentu wyprodukowania pierwszego mikrokontrolera AVR minęło już kilka lat. Do dziś firma Atmel rozwinęła tę rodzinę na tyle, że nie sposób w jednej książce omówić wszystkich dostępnych odmian mikrokontrolerów. Do celów demonstracyjnych wybrałem układ AT90S2313, na bazie którego będą omówione bloki funkcjonalne występujące również w innych mikrokontrolerach rodziny AVR. Dodatkowym czynnikiem decydującym o wyborze jest stosunkowo niska cena tego mikrokontrolera. Te argumenty powinny zachęcić konstruktorów do sięgnięcia po ten układ, na inne – bardziej rozbudowane – przyjdzie czas, gdy okaże się, że zasoby 2313 nie są wystarczające.

Jak już wspominałem, przykładam dużą wagę do praktycznego odnoszenia poruszanych w książce zagadnień. Aby ułatwić start początkującym, wszystkie eksperymenty wykonałem na uniwersalnej płytce uruchomieniowej dla mikrokontrolera AT90S2313 (oznaczenie ZL1AVR) opracowanej specjalnie na potrzeby tej książki. Jej budowę dokładnie opisałem w rozdziale 14.

1. Trochę historii

Elektronika od początku swojego istnienia przechodziła wiele momentów przełomowych. Można powiedzieć, że często decydowały one o otaczającym nas świecie. Dziś trudno powiedzieć, czy ważniejsze było wynalezienie lampy elektronowej, tranzystora, czy skonstruowanie pierwszego układu scalonego. Niemniej wiekopomnym dziełem było niewątpliwie opracowanie pierwszego mikroprocesora. To, co się dzieje we współczesnej elektronice, w większości przypadków obraca się właśnie wokół tego elementu. Aż trudno uwierzyć, że wszystko zaczęło się zupełnie niedawno, bo na początku lat 70. Niewinnie wyglądające zamówienie na specjalizowany układ, który miał być zastosowany w elektronicznych kalkulatorach, wpłynęło na losy świata. O dziwo, zlecającym była japońska firma Busicom, a wykonawcą amerykański Intel. Japończycy postrzegani byli w tamtych czasach, jako bardzo zdolny naród, który potrafi zrobić niemal wszystko. To „niemal” zdecydowało, że dzisiaj nikt nie pamięta nazwy Busicom, a słowo Intel odnieniamy na całym świecie nawet przedszkolaki. Tak powstał pierwszy mikroprocesor – Intel 4004 (wygląd jednej z jego wersji pokazano na fotografii 1.1). Było to w 1971 roku, więc nie jest to historia



Fot. 1.1. Widok opracowanego przez firmę Intel w 1971 roku mikroprocesora i4004 w obudowie ceramicznej

bardzo odległa. Na obszarze 3 na 4 mm umieszczono 2300 tranzystorów P-MOS. Układ mógł wykonać 100 000 instrukcji w ciągu sekundy, a jego lista rozkazów liczyła 45 pozycji. Chociaż żywot układu nie był zbyt długi (następca pojawił się już w 1972 roku), jego wyprodukowanie stanowiło prawdziwą „iskrę zapalną” dla prac konstrukcyjnych nad nowymi wersjami.

Dalej wydarzenia potoczyły się bardzo szybko. W krótkim czasie powstało wiele różnych odmian mikroprocesorów, ale tym, który naprawdę rozpoczął podbój świata był Intel 8080. Warto przypomnieć, że był on produkowany także w Polsce w zakładach CEMI pod oznaczeniem MCY7880. Swoją drogą zastanawiające jest jak to było możliwe, zważywszy że w halach produkcyjnych dawało się wyczuwać pod nogami drżenia posadzki, wywoływane przez przejeżdżający przed fabryką tramwaj.

Rewolucyjną rolę mikroprocesorów zauważyli też inni producenci. Jednym z nich był Zilog, którego Z80 przez długi czas skutecznie konkurował z 8080. Można obiektywnie stwierdzić, że w procesorze tym wiele rzeczy unowocześniono i ulepszono. Był wygodniejszy w użyciu zarówno w trakcie projektowania części elektronicznej, jak i w pisaniu oprogramowania. Charakteryzował się również większą niż 8080 wydajnością.

Jedną z cech ówczesnie produkowanych mikroprocesorów było to, że wymagały stosunkowo rozbudowanego otoczenia, tworzonego na bazie wyspecjalizowanych układów scalonych. Mikroprocesory były pozbawione także jakichkolwiek peryferiów (w postaci choćby interfejsu UART, wewnętrznej pamięci programu, niektóre wymagały nawet zewnętrznych kontrolerów przerwań). Dlatego właśnie kolejnym przełomem było opracowanie mikrokontrolera, czyli mikroprocesora zintegrowanego w jednej obudowie z modułami peryferyjnymi. W ten sposób powstała bardzo podobna do pierwowzoru, jednak zupełnie nowa klasa układów. Nazywano je pierwotnie mikrokomputerami jednoukładowymi, później przyjęła się nazwa mikrokontroler.

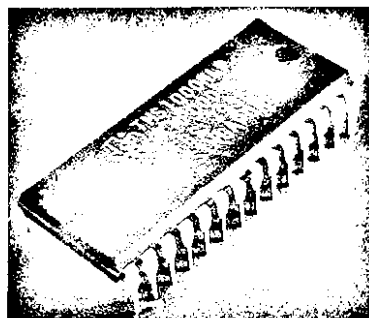
Wbrew powszechnemu mniemaniu, twórcą pierwszego na świecie mikrokontrolera jest firma Texas Instruments, która na początku 1975 roku wprowadziła do masowej produkcji 4-bitowe mikrokontrolery TMS1000 (jedną z wersji pokazano na fotografii 1.2). Z trudnych do ustalenia obecnie przyczyn mikrokontrolery te nie zdobyły wystarczająco dużego rynku, co – między innymi – spowodowało, że niedługo później zaczęły dominować mikrokontrolery produkowane przez firmę Intel (8042, 8048). Prawdziwą furorę zrobiły jednak mikrokontrolery 8051 i jego pochodne. Układ ten, choć często krytykowany, niepodzielnie króluje na rynku do dziś. Niewątpliwie powodem tego jest fakt, że wielu producentów mikrokontrolerów produkuje dziesiątki różnorodnych odmian tego układu, dzięki czemu praktycznie do każdej

aplikacji można dobrać układ z odpowiednimi peryferiami. Dodatkowym powodem popularności układów z rodziny MCS-51 jest to, że konstruktorzy znają architekturę 8051 „na pamięć” oraz zgromadzili odpowiednie narzędzia sprzętowe i programowe. Co więcej, mimo upływu wielu lat od pojawienia się pierwszych układów z rodziny MCS-51, wciąż są opracowywane nowe, udoskonalone wersje 8051. Z powyższych powodów jest on często stosowany nawet w takich projektach, do których można znaleźć bardziej odpowiednie odmiany mikrokontrolerów, z 8051 nie mających nic wspólnego.

Czy dominacja ta będzie jednak trwać wiecznie? Zapewne nie. Poważnym konkurentem dla 8051 stały się ostatnio mikrokontrolery z rodziny AVR, opracowane zresztą przez firmę, która rozpoczynała swą karierę od produkcji „klonów” 8051. Wszyscy już oczywiście wiedzą, że chodzi tu o firmę Atmel, która tak jak Intel w latach 80., obecnie staje się jednym z największych na świecie dostawców mikrokontrolerów 8-bitowych, a już na pewno bije rekordy popularności w Polsce.

2. Charakterystyka mikrokontrolerów AVR

Budowa mikrokontrolerów AVR opiera się na architekturze harwardzkiej. Jedną z głównych cech charakterystycznych architektury harwardzkiej jest rozdzielenie przestrzeni adresowej pamięci programu i przestrzeni adresowej pamięci danych, co uzyskano poprzez zastosowanie oddzielnych magistrali adresowych. Dzięki temu możliwe było zastosowanie słowa o różnej szerokości dla pamięci programu i pamięci danych, a także uchronienie się od przypadków, w których dane mogłyby być interpretowane jako instrukcje. Mikrokontrolery AVR, w przeciwieństwie do 8051, należą do grupy układów o architekturze RISC (*Reduced Instruction Set Computer*). Architektura 8051 jest określana nazwą CISC (*Complex Instruction Set Computer*). Wykonanie



Fot. 1.2. Jedna z wersji 4-bitowego mikrokontrolera TMS1000 firmy Texas Instruments

jednego rozkazu CISC wymaga zazwyczaj wykonania wielu operacji, co zwykle trwa kilka taktów zegarowych. Większość rozkazów RISC jest realizowana w jednym takcie zegara, co – pomimo krótszej listy rozkazów – zapewnia szybsze wykonywanie programu. Wbrew pozorom, programy pisane dla procesorów RISC charakteryzują się większą spójnością, a co za tym idzie mniejszym kodem wynikowym. Cechą wyróżniającą mikrokontrolery AVR jest również to, że zaimplementowano w nich wiele rejestrów wewnętrznych, z których każdy może pełnić funkcję akumulatora podczas wykonywania operacji arytmetycznych i logicznych. Dzięki temu minimalizuje się liczbę wewnętrznych przesłań międzyrejestrowych, co korzystnie wpływa na szybkość wykonywania programu i jego wielkość. Projektanci rodziny AVR przewidzieli możliwość wykorzystywania trzech par rejestrów jako rejestrów indeksowych używanych w niektórych trybach adresowania.

Na uwagę zasługuje jeszcze jeden fakt, który może na początku wydać się nieco dziwny. Mikrokontrolery AVR są zaliczane do grupy układów 8-bitowych, lecz słowo instrukcji jest 16-bitowe. Taka, a nie inna klasyfikacja wynika z długości rejestrów wewnętrznych i szerokości wewnętrznej szyny danych. Jeśli więc mówimy, że np. AT90S2313 ma 2 kB pamięci wewnętrznej, to oznacza, że możemy napisać program o wielkości 1 kśłów. Tych, u których wystąpił w tym momencie grymas na twarzy – przede wszystkim miłośników 8051 – pragnę uspokoić, że wszystkie wymienione cechy procesorów RISC powodują, że analogiczne programy powinny się zmieścić bez większych problemów zarówno w 8051 z pamięcią programu o pojemności 2 kB, jak i w AVR z pamięcią programu o wielkości 1 kśłów.

Z wymienionych powodów, mikrokontrolery AVR nadają się doskonale do pisania programów w językach wysokiego poziomu, szczególnie w języku C. Korzyści z tego wynikające z pewnością docenią ci, którzy spróbują swych sił w tej dziedzinie. Nie jest jednak bezzasadna opinia mówiąca, że do wydajnego pisania programów w C nieodzowne jest poznanie architektury używanego mikrokontrolera oraz jego listy rozkazów. Oczywiście jest, że przyjdzie to łatwiej tym Czytelnikom, którzy do tej pory nie mieli do czynienia z mikrokontrolerami innych rodzin. Przyzwyczajenia nabyte podczas prac z nimi mogą czasami przeszkadzać w poznawaniu rodziny AVR. Już na wstępie wątpliwość mogą budzić obco brzmiące mnemoniki rozkazów. Są one – jak się bliżej przyjrzeć – logiczne i konsekwentne, nie mniej dość trudne do zapamiętania. W efekcie pisać program w assemblerze często trzeba sięgać po jakąś „ściągawkę”. Dużą pomocą powinna być dołączona do książki wkładka zawierająca m.in. skrócony opis listy rozkazów.

3. Rodzina AVR – podstawy

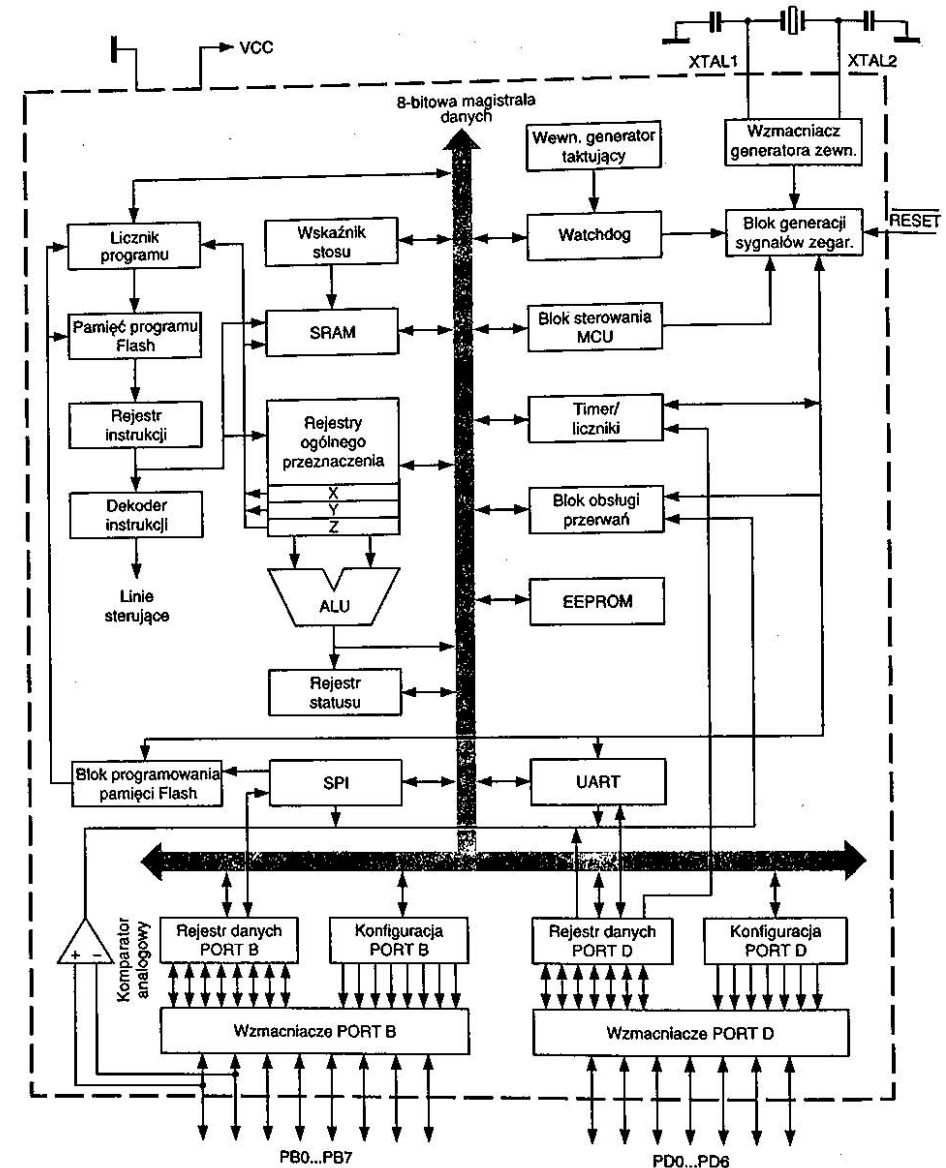
W rodzinie AVR zastosowano klasyczny sposób budowania mikrokontrolerów o różnych możliwościach, które powstają w wyniku integrowania wspólnego dla rodziny AVR rdzenia z różnymi blokami peryferyjnymi. Mikrokontrolerem o najmniejszych możliwościach w tej rodzinie jest AT90S1200, który ma ograniczoną do 3 liczbę obsługiwanych przerwań i, poza jednym 8-bitowym timerem i komparatorem analogowym, nie ma wbudowanych żadnych innych peryferii. Pomimo prostej budowy nadaje się on doskonale do stosowania w poważnych aplikacjach przemysłowych (wbrew pozorom większość aplikacji tego typu nie wymaga stosowania mikrokontrolerów o ogromnych możliwościach i dużej wydajności), a szczególnie atrakcyjny jest dla początkujących, którzy – właśnie ze względu na jego prostotę – mogą szybko go poznać.



Zestawienie najważniejszych parametrów mikrokontrolerów z rodziny AVR znajduje się w dodatku A.

Ćwiczenia prezentowane w rozdziale 14 opracowano na nieco bardziej rozbudowaną wersję mikrokontrolera – AT90S2313 – który wyposażono m.in. w szeregowy interfejs sprzętowy UART oraz 128 B pamięci danych SRAM. Dzięki temu w przykładach zostaną pokazane także sposoby komunikacji mikrokontrolera z komputerem PC (m.in. poprzez interfejs USB). Niebagatelnym atutem mikrokontrolerów AT90S2313 jest zgodność rozmieszczenia ich wyprowadzeń z produkowanymi przez Atmela mikrokontrolerami z rodziny 8051: AT89C1051/1051U, AT89C2051/2051 i AT89C4051.

Na rysunku 3.1 przedstawiono budowę wewnętrzną mikrokontrolera AT90S2313. Programista ma dostęp do 32 rejestrów ogólnego przeznaczenia, wśród których występują m.in. trzy rejestry indeksowe. Są one wykorzystywane w niektórych trybach adresowania, wspomagając operacje przesyłania danych. Wszystkie rejestry są dołączone bezpośrednio do jednostki arytmetyczno-logicznej ALU (Arithmetic Logic Unit) i mogą pełnić funkcję akumulatora. Ta cecha znacząco wpływa na zwiększenie wydajności mikrokontrolera. Zwróćmy uwagę na to, że znaczną część czasu pracy mikrokontrolera zajmuje wykonywanie operacji logicznych lub arytmetycznych. W tradycyjnych rozwiązaniach wszystkie operacje musiały być wykonywane w wydzielonym



Rys. 3.1. Budowa wewnętrzna mikrokontrolera AT90S2313

akumulatorze dołączonym do ALU, zatem jeśli program korzystał z wielu danych, bezustannie musiał wykonywać przesłania międzyrejestrowe. Ilustruje to przykład 3.1.

Przykład 3.1. Porównanie operacji dodawania dwóch 2-bajtowych liczb, wykonywanej przez mikrokontrolery rodziny '51 i AVR

```
;Program dla mikrokontrolera '51
;dana1 - R1R0, dana2 - R3R2, wynik R1R0
MOV   A,R0
ADD   A,R2
MOV   R0,A
MOV   A,R1
ADDC  A,R3
MOV   R1,A

;Program dla mikrokontrolera AVR
;dana1 - R1R0, dana2 - R3R2, wynik R1R0
ADD   R0,R2
ADDC  R1,R3
```

Zwróćmy uwagę na to, że mikrokontroler '51 na wykonanie każdego z powyższych rozkazów potrzebuje 12 cykli zegara, natomiast AVR wykonuje je w jednym cyklu. Biorąc pod uwagę częstotliwości oscylatorów równe odpowiednio 12 MHz i 4 MHz, czas wykonania pierwszego fragmentu programu będzie równy 6 μ s, drugiego zaś tylko 0,5 μ s. Kod dla 8051 zajmie 6 bajtów pamięci programu, dla AVR-a natomiast jedynie dwa słowa. Ten prosty przykład z całą bezwzględnością pokazuje przewagę wydajności mikrokontrolera AVR nad 8051.

Jak widać na **rysunku 3.1**, większość bloków funkcjonalnych mikrokontrolerów AVR komunikuje się między sobą za pośrednictwem wewnętrznej 8-bitowej magistrali. Dodatkowo prowadzone są pomiędzy nimi niezbędne sygnały sterujące.

Oprócz typowych i na ogół dobrze znanych układów peryferyjnych można wyróżnić bloki, do których użytkownik nie ma bezpośredniego dostępu. Są to: rejestr instrukcji, dekodery instrukcji, wewnętrzny oscylator i układ generujący wewnętrzne sygnały zegarowe. Rejestr instrukcji przechowuje kod aktualnie wykonywanego rozkazu. Jest to w większości przypadków 16-bitowa liczba. Wyjątkiem są rozkazy wymagające podania argumentu, np. wywołania podprogramów lub skoki. Kody te są podawane na liście rozkazów, choć większego znaczenia praktycznego dla użytkownika nie mają. Na podstawie zawartości rejestru instrukcji, dekodery instrukcji generuje odpowiednie sygnały sterujące dla automatu realizującego funkcje procesora. Automat ten jest układem synchronicznym. Do pracy wymaga odpowiedniego przebiegu zegarowego (jednego lub wielu). Od strony wyprowadzeń (na styku mikrokontroler-otoczenie) niezbędny jest jedynie zegar jednofazowy.

Na schemacie z **rysunku 3.1** nie widać stosu jako wydzielonego bloku. W mikrokontrolerze AT90S2313 stos został umiejscowiony w wewnętrznej pamięci danych (SRAM). Oznacza to, że jego głębokość jest uwarunkowana

jedynie dostępnym obszarem tej pamięci. Pamiętajmy, że są tu lokowane również zmienne wykorzystywane przez program. Niewykorzystany obszar pamięci SRAM kompilator przeznaczają na stos.

Większość aktualnie produkowanych mikrokontrolerów dysponuje mechanizmami oszczędzania energii. Wyposażono w nie również układ AT90S2313. Podczas normalnej pracy wszystkie bloki mikrokontrolera pracują bez żadnych ograniczeń pobierając ze źródła zasilającego prąd o wartości ok. 2,8 mA. W trybie *Idle* zostaje wstrzymana praca jednostki centralnej (*CPU – Central Processing Unit*), pozostałe bloki (pamięć SRAM, timery/liczniki, port SPI, system przerwań) pracują normalnie. W tym trybie układ pobiera prąd o wartości ok. 0,8 mA. Istnieje jeszcze trzeci tryb oszczędzania energii – *power-down*, w którym wszystkie wewnętrzne bloki są wyłączone (łącznie z generatorem taktującym). Zostają jednak zachowane stany rejestrów. Jediną metodą „obudzenia” mikrokontrolera jest w tym przypadku zgłoszenie przerwania zewnętrznego lub wyzerowanie mikrokontrolera. W tym trybie prąd zasilający jest mniejszy od 1 μ A.

Firma Atmel była jedną z pierwszych, która opanowała technologię wytwarzania pamięci Flash i od samego początku stosowała ją jako pamięć programu w swoich mikrokontrolerach. Można wręcz powiedzieć, że zrewolucjonizowało to światowy rynek w tej branży. Pamięć Flash można wielokrotnie programować, do tego jest możliwe jej programowanie po zamontowaniu mikrokontrolera w systemie. Dzięki temu znacznie uprościły się prace konstrukcyjne i skomplikowane (drogie) programatory stały się zbędne. Do zaprogramowania pamięci wystarczy jedynie prosty interfejs. Dla amatorów jest to bardzo cenna cecha, gdyż wejście w świat mikrokontrolerów nie wymaga inwestowania dużych kwot w specjalistyczne oprzyrządowanie. W Internecie można znaleźć schematy wielu rozwiązań interfejsów programujących możliwych do wykonania niemal w przysłowiowe „pięć minut”, w dodatku z elementów, jakich na ogół jest pełno w szufladzie. W dalszej części książki zostaną zaprezentowane adresy internetowe, pod którymi można znaleźć wiele bardzo pomocnych informacji, a nawet pobrać darmowe oprogramowanie wspomagające projektowanie urządzeń z mikrokontrolerami AVR.

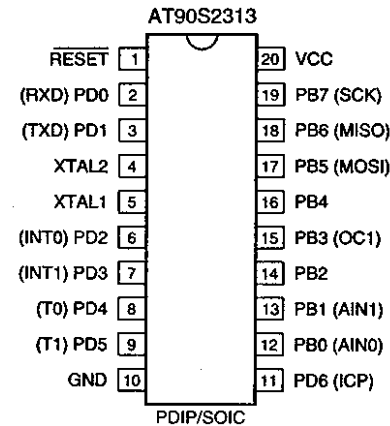
3.1. Funkcje wyprowadzeń

Na **rysunku 3.2** przedstawiono rozmieszczenie wyprowadzeń mikrokontrolera AT90S2313 w obudowie PDIP lub SOIC. Funkcje wyprowadzeń są następujące:

VCC – plus napięcia zasilającego.

GND – masa zasilania.

Port B (PB0...PB7) – 8-bitowy, dwukierunkowy port wejścia/wyjścia ogólnego przeznaczenia. Wszystkie linie portu mają indywidualnie konfigurowalne wewnętrzne podciąganie do plusa napięcia zasilania (*pull-up*). Maksymalny prąd wejściowy (w kierunku od plusa zasilania do masy) każdej linii wynosi 20 mA, co umożliwia bezpośrednie sterowanie np. diodami świecącymi. Po zerowaniu mikrokontrolera wyprowadzenia portu B ustawiają się w stan wysokiej impedancji, także wtedy, gdy nie jest generowany sygnał zegarowy.



Rys. 3.2. Rozmieszczenie wyprowadzeń mikrokontrolera AT90S2313



Jeśli linie portów PB i PD są skonfigurowane jako wejścia i z zewnątrz mają dołączone podciąganie do masy oraz włączone wewnętrzne rezystory podciągające (*pull-up*), to stanowią one źródła prądu.

Niektóre linie portu B mogą pełnić też dodatkowe funkcje (będą one dokładnie omówione w rozdziale 10.):

PB0 (AIN0) – wejście nieodwracające wewnętrznego komparatora.

PB1 (AIN1) – wejście odwracające wewnętrznego komparatora.

PB3 (OC1) – wyjście wyniku porównania Timera/Licznika 1.

PB5 (MOSI) – szeregowe wejście danych w trybie programowania i weryfikacji.

PB6 (MISO) – szeregowe wyjście danych w trybie programowania i weryfikacji.

PB7 (SCK) – wejście zegarowe dla trybu programowania i weryfikacji.

Port D (PD6...PD0) – 7-bitowy, dwukierunkowy port wejścia/wyjścia ogólnego przeznaczenia. Wszystkie linie portu mają indywidualnie konfigurowalne wewnętrzne podciąganie do plusa zasilania (*pull-up*). Maksymalny prąd wejściowy każdej linii wynosi 20 mA. Po zerowaniu mikrokontrolera wyprowadzenia portu D ustawiają się w stan wysokiej impedancji, także wtedy, gdy nie jest generowany sygnał zegarowy.

Linie portu D mogą pełnić też dodatkowe funkcje (będą one dokładnie omówione w rozdziale 10.):

PD0 (RXD) – wejście szeregowe układu UART.

PD1 (TXD) – wyjście szeregowe układu UART.

PD2 (INT0) – wejście przerwania zewnętrznego.

PD3 (INT1) – wejście przerwania zewnętrznego.

PD4 (T0) – wejście zewnętrznego przebiegu zegarowego licznika T0.

PD5 (T1) – wejście zewnętrznego przebiegu zegarowego licznika T1.

PD6 (ICP) – wejście przechwytywania zegara.

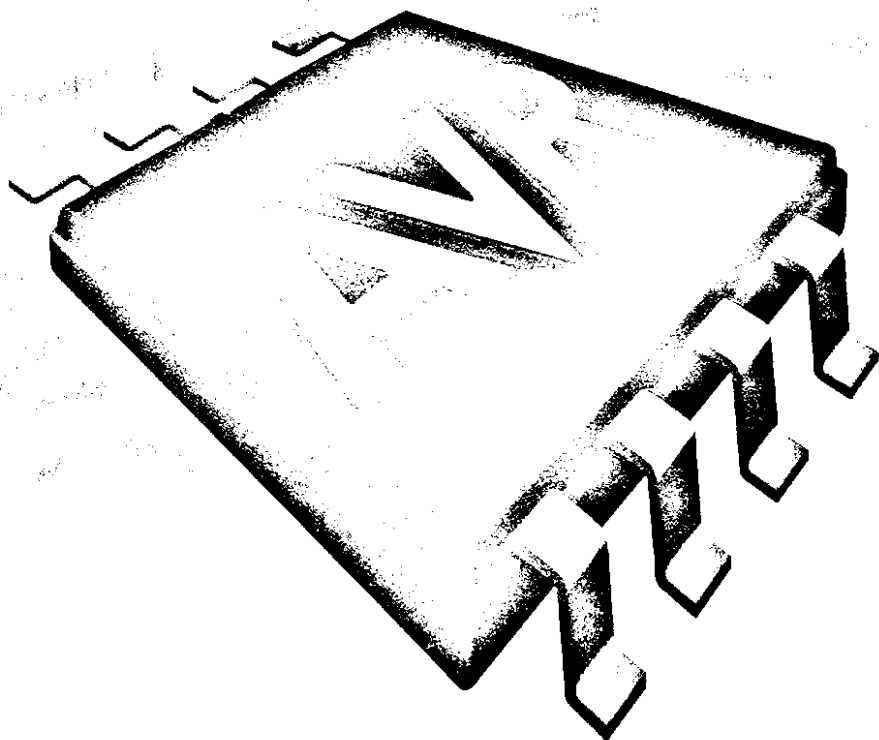
RESET – wejście zerowania mikrokontrolera. Do wygenerowania prawidłowego sygnału zerującego konieczne jest, aby stan niski na tym wyprowadzeniu trwał co najmniej 50 ns. Sygnał zerowania wystąpi nawet, gdy nie pracuje zegar. Impulsy krótsze niż 50 ns nie gwarantują wygenerowania prawidłowego sygnału zerującego.

XTAL1 – wejście odwracające wzmacniacza oscylatora mogące pełnić funkcję wejścia zewnętrznego przebiegu zegarowego.

XTAL2 – wyjście wzmacniacza oscylatora (odwracającego fazę).

Część 2

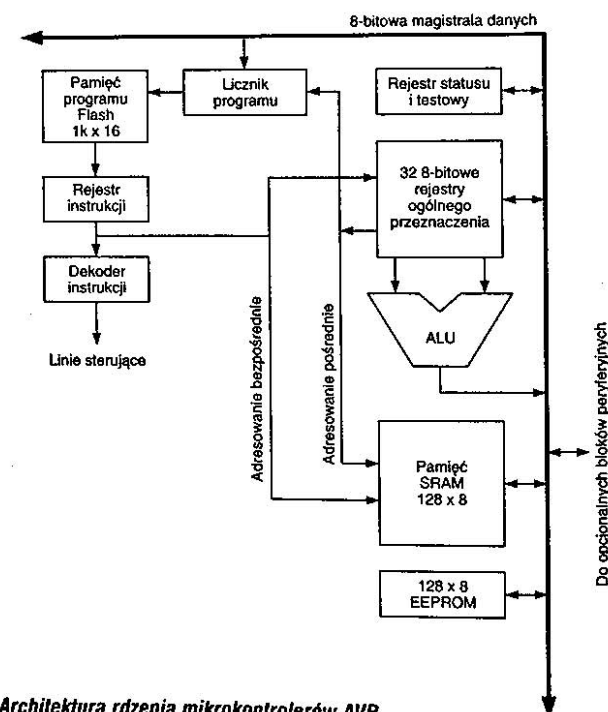
Budowa i działanie mikrokontrolerów AVR



4. Architektura mikrokontrolerów AVR

Wszystkie bloki funkcjonalne mikrokontrolera AT90S2313 komunikują się z jednostką arytmetyczno-logiczną (ALU) za pośrednictwem wewnętrznej 8-bitowej magistrali danych. Jednym z ważniejszych elementów architektury (rysunek 4.1) jest zestaw 32 8-bitowych rejestrów ogólnego przeznaczenia. Gwarantują one szybki dostęp do podręcznych danych. Wszystkie rejestry mają bezpośredni dostęp do ALU, mogą więc pełnić rolę operandów działań arytmetyczno-logicznych bez pośrednictwa dodatkowych rejestrów przejściowych i specjalnego akumulatora.

Rezultat wykonywanej operacji jest umieszczany w dowolnym rejestrze, którym nie zawsze musi być akumulator, jak ma to miejsce w niektórych innych rodzinach mikrokontrolerów. Spośród 32 rejestrów, 6 wydzielono do specjalnych zadań, chociaż nadal pozostają rejestrami ogólnego przeznaczenia. Tworzą one trzy 16-bitowe rejestry indeksowe, wykorzystywane w trybach adresowania pośredniego, stanowiąc bardzo wydajny mechanizm obliczania adresu. Duża liczba rejestrów ułatwia pracę kompilatorom języków wysokie-



Rys. 4.1. Architektura rdzenia mikrokontrolerów AVR

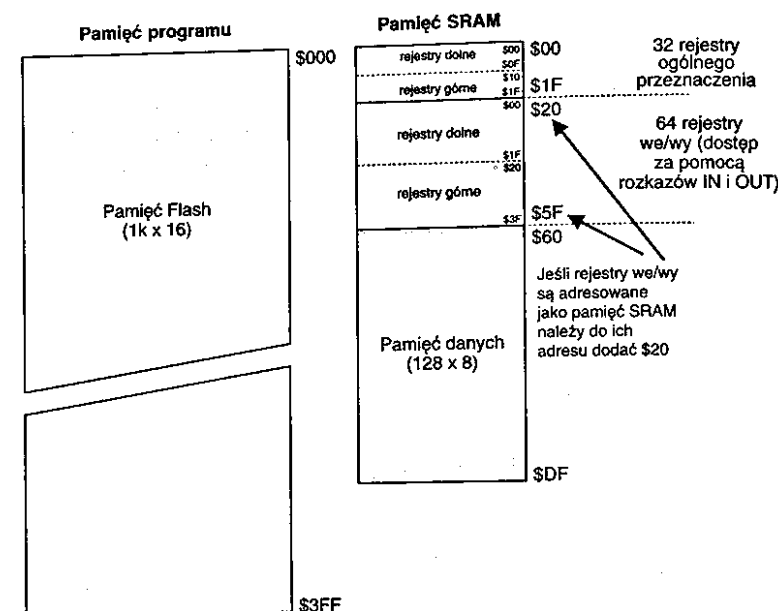
go poziomu, umożliwiając przyspieszenie wykonywania programu. Jeden z rejestrów indeksowych może być wykorzystywany jako wskaźnik (*pointer*) na stałe umieszczone w tablicy tzw. *look-up*. Do rozróżniania rejestrów stosuje się oznaczenia od R0 do R31. Trzy 16-bitowe rejestry indeksowe zostały nazwane: *X-register*, *Y-register* i *Z-register*. Jednostka arytmetyczno-logiczna wykonuje obliczenia na parze danych umieszczonych w dwóch rejestrach lub na danej umieszczonej w rejestrze i stałej umieszczonej w tablicy. Możliwe są też operacje na jednym rejestrze. Typowe tryby adresowania pamięci mogą być stosowane również do adresowania zestawu rejestrów. Dzieje się tak, gdyż rejestry są ułożone w jednej przestrzeni adresowej z pamięcią danych, zajmując najmłodsze adresy (\$00...\$1F). Kolejne 64 lokacje (\$20...\$5F) w mikrokontrolerach AVR przeznaczono dla urządzeń peryferyjnych: rejestrów sterujących, timerów/liczników, przetworników A/C (w AT90S2313 ich nie ma!) i innych. Będziemy je nazywać obszarem we/wy.



W książce przyjęto konwencję zapisu liczb heksadecymalnych, zgodnie z którą są one poprzedzane znakiem dolara (\$). Liczby dziesiętne są zapisywane w sposób standardowy. W przykładach programów pisanych w języku C, liczby heksadecymalne są zapisywane w obowiązującym w nim formacie z przedrostkiem 0x. Na przykład: 16=\$10=0x10.

Mikrokontrolery AVR mają architekturę harwardzką. Przestrzeń adresowa pamięci programu i pamięci danych jest w niej rozdzielona (rysunek 4.2). Wykonywanie rozkazów jest realizowane poprzez 2-stopniowe przetwarzanie potokowe (*pipeline*). Wykonanie instrukcji odbywa się z jednoczesnym wstępnym pobraniem kodu następnej instrukcji. Koncepcja ta umożliwia wykonanie w każdym cyklu jednego rozkazu.

Pamięć programu to programowana w systemie pamięć typu Flash. Cały obszar 1 kół jest dostępny bezpośrednio dla instrukcji skoków względnych i wywołań podprogramów. Większość instrukcji mikrokontrolera AVR mieści się w 16-bitowym słowie, ale zdarzają się także instrukcje 32-bitowe. W wyniku przyjęcia przerwania, a także podczas wywoływania podprogramów, adres powrotu, jakim jest aktualny stan licznika programu (*Program Counter – PC*) jest automatycznie zachowywany na stosie. Stos jest umieszczony w obszarze pamięci danych SRAM, w związku z czym jego pojemność jest limitowana tylko jej dostępnym obszarem. Dostęp do danych zachowanych na stosie jest możliwy za pomocą wskaźnika stosu (*Stack Pointer –*



Rys. 4.2. Mapa pamięci mikrokontrolera AT90S2313 (pamięć EEPROM leży poza standardowym obszarem adresowym, dostęp do niej jest możliwy poprzez rejestry EEAR i EEDR)

SP). Należy bezwzględnie pamiętać o odpowiednim ustawieniu jego wartości, w przeciwnym przypadku może dojść do nadpisania danych lub utraty wartości stosu przez nadpisanie go danymi, co w konsekwencji może doprowadzić do błędnego działania programu. Ustawienie SP powinno odbywać się na samym początku programu, jako jedna z pierwszych czynności, koniecznie przed wywołaniami procedur i włączeniem przerwań. W przypadku jednoczesnego wystąpienia dwóch przerwań, w pierwszej kolejności jest obsługiwane to o wyższym priorytecie.

Wskaźnik stosu w mikrokontrolerze AT90S2313 to 8-bitowy rejestr dostępny w przestrzeni adresowej we/wy. Umieszczenie stosu w obszarze pamięci danych jest z jednej strony korzystne, bo nie ogranicza jego wielkości. Jednak przy rozrzuconym gospodarowaniu zmiennymi wykorzystywanymi przez program może się okazać, że na stos pozostaje mało miejsca. Na domiar złego nie ma żadnego mechanizmu kontrolującego jego przekroczenie. Może to prowadzić do niełatwych w wykryciu błędów. Z drugiej strony dane odłożone na stosie, jako dane w pamięci danych są również dostępne poprzez zwykłe tryby adresowania. Stwarza to możliwość wykorzystywania wyrafinowanych trików polegających na podmianie danych odłożonych na stosie z innymi. Można w ten sposób przekazywać np. argumenty do procedur.

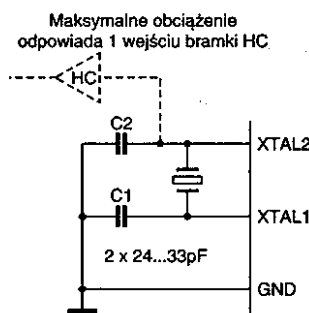
Cała przestrzeń adresowa od \$00 do \$DF, do której należy 128 bajtów danych pamięci SRAM, 64 bajty przestrzeni we/wy oraz zbiór rejestrów roboczych może być w prosty sposób udostępniona poprzez pięć trybów adresowania obsługiwanych w architekturze mikrokontrolerów AVR. Jest to przestrzeń liniowa i regularna, czyli dostęp do każdej z komórek jest możliwy za pomocą każdego polecenia wykonującego operację na rejestrach (rysunek 4.2).

System przerwań mikrokontrolera AVR wykorzystuje niektóre rejestry sterujące znajdujące się w przestrzeni we/wy. Każde z przerwań ma indywidualny wektor przerwania, czyli adres skoku do procedury obsługi przerwania. Tablica wektorów przerwań jest zawsze umieszczana na początku pamięci programu. Wszystkie przerwania charakteryzują się przypisanym priorytetem obsługi. Im jest niższy adres danego wektora przerwania, tym wyższy ma on priorytet.

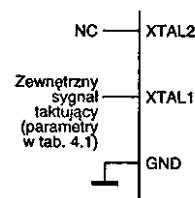
4.1. Generator taktujący

W mikrokontroler wbudowano wzmacniacz odwracający fazę, przeznaczony do generowania przebiegu zegarowego synchronizującego pracę mikrokontrolera. Wyprowadzenia XTAL1 i XTAL2 pełnią funkcje odpowiednio jego wejścia i wyjścia. Po dołączeniu zewnętrznego rezonatora wzmacniacz pracuje jako generator sygnału taktującego mikrokontroler. Sposób dołączenia rezonatora pokazano na rysunku 4.3. Można stosować zarówno rezonatory kwarcowe, jak i ceramiczne.

Przewidziano także możliwość pracy mikrokontrolera z zewnętrznym przebiegiem zegarowym. Wyprowadzenie XTAL2 powinno w takim przypadku pozostać niepodłączone, natomiast do XTAL1 jest doprowadzany sygnał zegarowy (rysunek 4.4). Bardzo ważne dla poprawnego działania mikrokontrolera jest, aby sygnał zewnętrzny spełniał warunki czasowe zawarte w tablicy 4.1 i miał amplitudę dostosowaną do napięcia zasilającego mikrokontroler.



Rys. 4.3. Zalecany sposób dołączenia zewnętrznego rezonatora kwarcowego do wyprowadzeń mikrokontrolera AVR (sygnał z wyjścia XTAL2 może być wykorzystywany – po dodaniu bufora – także przez obwody zewnętrzne)



Rys. 4.4. Dołączenie zewnętrznego generatora taktującego do mikrokontrolera

Tab. 4.1. Zalecane przez producenta warunki czasowe wymagane od zewnętrznego sygnału taktującego

Parametr	$V_{cc}=2,7...6,0\text{ V}$		$V_{cc}=4,0...5,0\text{ V}$		Jednostka
	Min.	Maks.	Min.	Maks.	
Częstotliwość	0	4	0	10	MHz
Czas trwania poziomu H	100		40		ns
Czas trwania poziomu L	100		40		ns
Czas narastania		1,6		0,5	μs
Czas opadania		1,6		0,5	μs

Sygnał wytwarzany przez wewnętrzny generator mikrokontrolera może być również wykorzystywany do sterowania innymi układami zewnętrznymi. Wyjście XTAL2 musi być wówczas buforowane za pomocą bramki serii HC.

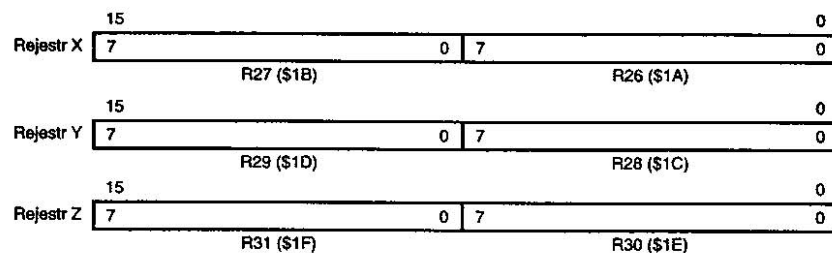
4.2. Rejestry ogólnego przeznaczenia

Prawie wszystkie rozkazy należące do listy obsługiwanych przez mikrokontrolery AVR umożliwiają wykonywanie operacji na rejestrach ogólnego przeznaczenia. Zapewniają one bezpośredni dostęp do każdego z nich i są wykonywane w jednym cyklu. Wyjątkiem jest pięć rozkazów operacji arytmetycz-

Adres	
\$00	R0
\$01	R1
\$02	R2
...	...
\$0D	R13
\$0E	R14
\$0F	R15
\$10	R16
\$11	R17
...	...
\$1A	R26
\$1B	R27
\$1C	R28
\$1D	R29
\$1E	R30
\$1F	R31

Rejestry
ogólnego
przeznaczenia

Rys. 4.5. Rozmieszczenie rejestrów ogólnego przeznaczenia w mikrokontrolerze AVR



Rys. 4.6. Budowa logiczna rejestrów indeksowych X, Y i Z

no-logicznych wykorzystujących stałe jako parametry – SBCI, SUBI, CPI, ANDI i ORI oraz rozkaz bezpośredniego ładowania stałej – LDI. Te rozkazy wykorzystują jedynie drugą połowę zestawu rejestrów, czyli R16...R31. Rozkazy SBC, SUB, CP, AND, OR i inne wymagające jednego lub dwóch argumentów pracują na całym zestawie rejestrów.

Wszystkie rejestry są widoczne w przestrzeni adresowej pamięci SRAM zajmując pierwsze 32 lokacje. Dzieje się tak mimo tego, że rejestry te nie są fizycznie zaimplementowane jako pamięć SRAM. Taka organizacja pamięci pozwala na bardzo elastyczny dostęp do rejestrów, także poprzez rejestry indeksowe X, Y i Z.

Rejestry R26...R31 są rejestrami ogólnego przeznaczenia, pełnią jednak dodatkowe funkcje. Mogą być wykorzystywane jako 16-bitowe rejestry indeksowe (wskaźniki), pozwalające na adresowanie pośrednie pamięci danych. Umożliwiają więc adresowanie pośrednie rejestrów roboczych, rejestrów we/wy, jak i pamięci SRAM (danych). Ich organizację przedstawiono na rysunku 4.6.

Bardzo interesującym trybem adresowania z wykorzystaniem rejestrów indeksowych jest adresowanie różnicowe. Rejestry X, Y i Z pełnią w tym przypadku rolę wskaźników przemieszczenia względem adresu bazowego, a ich zawartość może być automatycznie dekrementowana lub inkrementowana przed lub po wykonaniu określonej operacji.

Zaimplementowanie kilku rejestrów indeksowych zostało podyktowane ukierunkowaniem architektury rdzenia mikrokontrolerów AVR na języki wysokiego poziomu. Rejestry te są intensywnie wykorzystywane do indeksowania argumentów i wyniku operacji arytmetyczno-logicznych wykonywanych przez ALU. Dzięki takiemu rozwiązaniu uzyskuje się znaczne przyspieszenie wykonywania obliczeń.

4.3. Jednostka arytmetyczno-logiczna (ALU)

Mikrokontrolery AVR są wyposażone w bardzo wydajną jednostkę arytmetyczno-logiczną współpracującą bezpośrednio z 32 rejestrami ogólnego przeznaczenia. Operacje wykonywane przez ALU można podzielić na trzy kategorie: arytmetyczne, logiczne i operacje bitowe. Listę rozkazów wraz z ich dokładnym opisem przedstawiono w rozdziale 12.

4.4. Pamięć programu

Mikrokontroler AT90S2313 wyposażono w 2 kB pamięci Flash programowej w systemie, która pełni funkcję pamięci programu. Choć w danych technicznych jej wielkość jest określana jako 2 kB, to trzeba pamiętać, że wszystkie instrukcje mikrokontrolera AT90S2313 są 16- lub 32-bitowe. Pamięć programu ma organizację $1k \times 16$ bitów. Szacowana wytrzymałość pamięci wynosi 1000 cykli zapisu/kasowania. Do całkowitego zaadresowania obszaru 1 kSłów jest potrzebny 10-bitowy licznik programu, taki też zaimplementowano w mikrokontrolerze AT90S2313. Dokładniejsze informacje dotyczące programowania pamięci Flash podano w rozdziale 11.

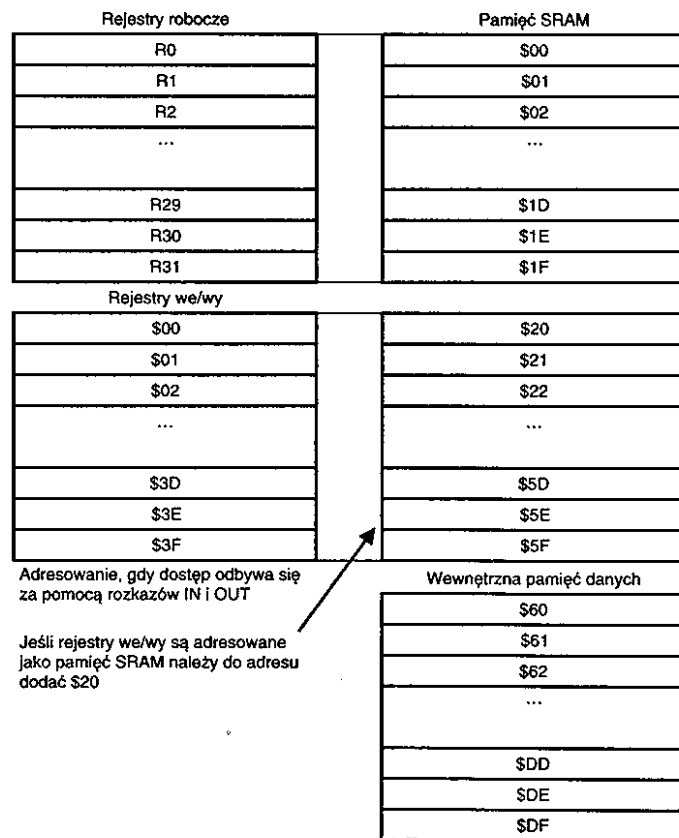
4.5. Nieulotna pamięć danych EEPROM

Współczesne urządzenia budowane w oparciu o mikrokontrolery bardzo często wymagają przechowywania jakichś parametrów także po zaniku zasilania. Najczęściej są to dane określające konfigurację układów peryferyjnych mikrokontrolera lub np. ostatnią konfigurację cyfrowych elementów regulacyjnych. Do przechowywania danych tego typu przewidziano 128-bajtową pamięć typu EEPROM. Jest ona widoczna jako wydzielony obszar adresowy, do którego można zapisywać lub odczytywać pojedyncze bajty. Żywotność pamięci EEPROM jest szacowana na co najmniej 100000 cykli zapisu/kasowania. Dostęp do niej jest realizowany poprzez wydzielone rejestry: adresowy, danych i sterujący. Dokładniejsze informacje dotyczące wykorzystywania pamięci EEPROM podano w rozdziale 11.

4.6. Pamięć danych SRAM

Na rysunku 4.7 przedstawiono organizację pamięci SRAM (*Static Random Access Memory*) mikrokontrolera AT90S2313. Wewnętrzna pamięć danych SRAM zajmuje lokacje od adresu \$60 do \$DF. Łącznie jest więc do wykorzystania 128 bajtów tej pamięci. Młodsze adresy zajmują: zestaw rejestrów roboczych (od \$00 do \$1F – 32 lokacje) oraz rejestry we/wy (od \$20 do \$5F – 64 lokacje).

Wymienione obszary tworzą przestrzeń adresową danych. Zapis lub odczyt tego obszaru może być realizowany w jednym z pięciu trybów adresowania: bezpośrednim, pośrednim z przemieszczeniem, pośrednim, pośrednim z predekrementacją, pośrednim z postinkrementacją. Zostaną one dokładnie omó-



Rys. 4.7. Organizacja pamięci danych SRAM. Rejestry we/wy do których dostęp odbywa się za pomocą rozkazów IN i OUT muszą być określone adresami od \$00 do \$3F mimo tego, że w pamięci SRAM zajmują lokacje od \$20 do \$5F

wione w dalszej części rozdziału. Do indeksowania danych w trybach pośrednich są wykorzystywane rejestry R26 do R31. W trybie adresowania bezpośredniego jest dostępna cała przestrzeń adresowa danych. Adresując dane pośrednio z przemieszczeniem uzyskuje się dostęp do 63 lokacji począwszy od przyjętego adresu bazowego. Do indeksowania danych w tym trybie wykorzystywane są rejestry Y i Z. Tryby pośredni z predekrementacją i pośredni z postinkrementacją wykorzystują rejestry X, Y, i Z.

4.7. Tryby adresowania pamięci danych i pamięci programu

Mikrokontrolery AVR dysponują bogatym zestawem trybów adresowania pamięci programu oraz danych, gwarantując tym samym dużą wydajność i efektywność dostępu do obu rodzajów pamięci. W rozdziale tym zostaną wyjaśnione szczegóły dotyczące zasad adresowania stosowanych w mikrokontrolerach AVR.

Umieszczony na kilku kolejnych rysunkach symbol *OP* oznacza część kodu operacji słowa rozkazu.

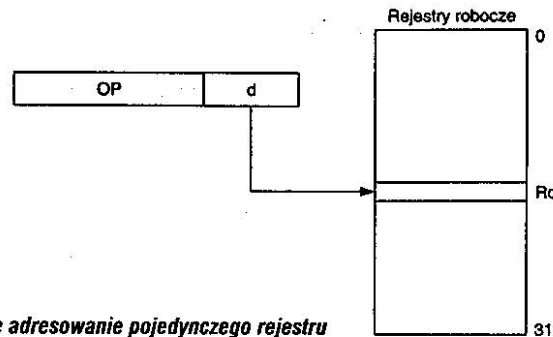
UWAGA

W dalszej części rozdziału przyjęto następujące oznaczenia:

- Rd* – rejestr źródłowy i przeznaczenia ulokowany w obszarze Register File,
- Rs* – rejestr źródłowy ulokowany w obszarze Register File,
- n* – 6-bitowy adres rejestru źródłowego lub docelowego,
- a* – 6-bitowe przesunięcie,
- k* – adres względny zapisany w kodzie uzupełnienia do 2 (*U2*), może przybierać wartości od -2048 do 2047,
- P* – 6-bitowy adres docelowego rejestru we/wy.

4.7.1. Tryb bezpośredniego adresowania rejestrów wykorzystujący pojedynczy rejestr

Operand – rejestr *Rd* – jest wskazany bezpośrednio w kodzie rozkazu poprzez podanie jego numeru *d*.

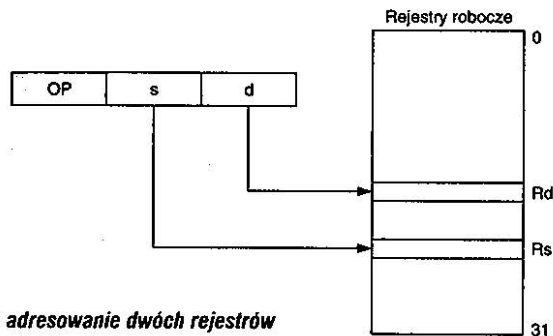


Rys. 4.8. Bezpośrednie adresowanie pojedynczego rejestru

Przykład 4.1. Inkrementacja rejestru R0

```
...
INC R0
...
```

4.7.2. Tryb bezpośredniego adresowania rejestrów wykorzystujący dwa rejestry



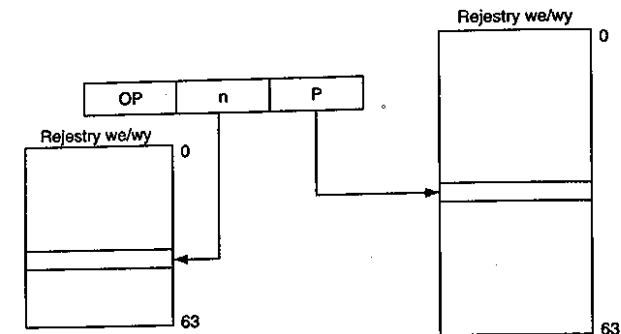
Rys. 4.9. Bezpośrednie adresowanie dwóch rejestrów

Operandy znajdują się w rejestrach *Rs* i *Rd*, które są wskazywane bezpośrednio w kodzie rozkazu (*s* i *d*). Rezultat operacji jest umieszczany w rejestrze *Rd*.

Przykład 4.2. Suma logiczna rejestrów R0 i R1. Wynik umieszczony w R0

```
...
OR R0, R1
...
```

4.7.3. Tryb bezpośredniego adresowania obszaru wejścia/wyjścia



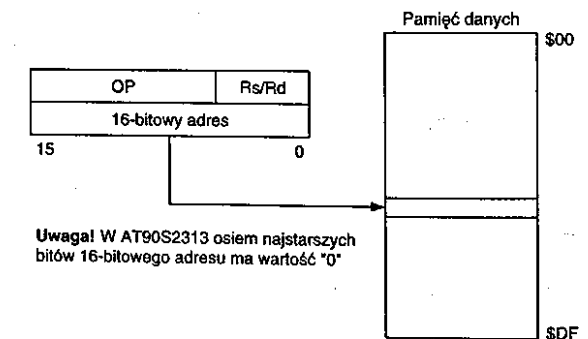
Rys. 4.10. Bezpośrednie adresowanie obszaru wejścia/wyjścia

Rezultat operacji jest umieszczany w rejestrze we/wy, którego adres jest bezpośrednio wskazywany w kodzie rozkazu. Adres operandu (*P*) zajmuje 6 bitów słowa rozkazowego. Pole *n* określa adres rejestru źródłowego lub docelowego.

Przykład 4.3. Przepisanie zawartości rejestru R0 do portu B

```
...
OUT PORTB, R0
...
```

4.7.4. Tryb bezpośredniego adresowania pamięci danych



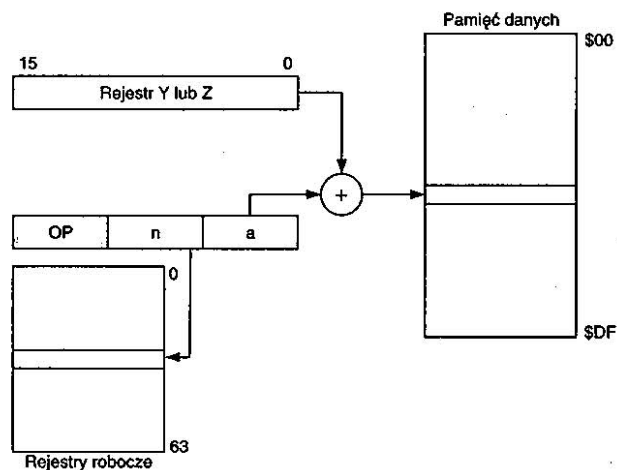
Rys. 4.11. Bezpośrednie adresowanie pamięci danych

Rozkaz wykorzystujący adresowanie bezpośrednie składa się z dwóch słów. Pierwsze słowo zawiera kod operacji i określenie rejestru źródłowego lub docelowego (*Rd/Rs*). Szesnastobitowy adres jest zawarty w drugim słowie rozkazu.

Przykład 4.4. Przepisanie zawartości komórki pamięci danych o adresie \$65 do rejestru R0

```
...
LDS R0,$65
...
```

4.7.5. Tryb pośredniego adresowania danych z przemieszczeniem



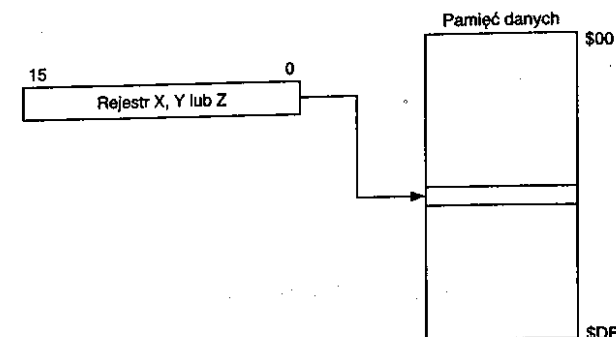
Rys. 4.12. Adresowanie pośrednie z przemieszczeniem

Adres operandu jest obliczany poprzez dodanie zawartości rejestrów Y lub Z stanowiących adres bazowy oraz przesunięcia podanego na sześciu bitach w słowie rozkazu. Mechanizm ten pozwala na wydajne przetwarzania rekordów i tablic. Jest wykorzystywany przez kompilatory języków wysokiego poziomu.

Przykład 4.5. Umieszczenie w rejestrze R0 danej z 8-bitowej tablicy zaczynającej się od adresu \$65. Pobierany jest 6. element tablicy (przesunięcie wskazuje przesunięcie względem pierwszego elementu o indeksie 0)

```
LDI R28,$65 ;adres bazowy tablicy danych 8-bitowych
LDD R0,Y+5 ;umieszczenie szóstego bajtu tablicy
           ;w rejestrze R0
```

4.7.6. Tryb adresowania pośredniego



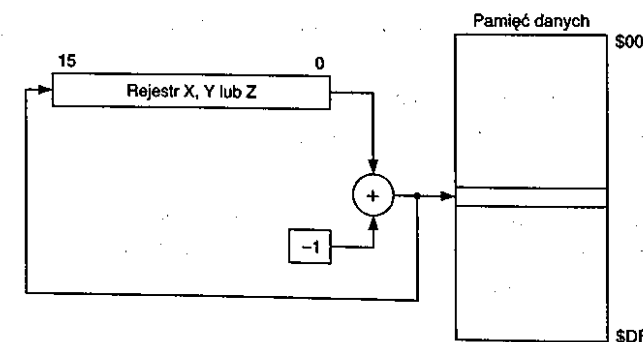
Rys. 4.13. Adresowanie pośrednie

Adres operandu jest umieszczony w rejestrze indeksowym X, Y lub Z. Rejestry te pełnią więc funkcję wskaźnika na operand. Mechanizm ten jest powszechnie wykorzystywany np. w języku C.

Przykład 4.6. Umieszczenie zawartości rejestru R16 w pamięci SRAM

```
LDI R28,$65 ;adres pośredni umieszczony w rejestrze indeksowym Y
ST Y,R16 ;umieszczenie zawartości rejestru R16 w pamięci
          ;adresowanej przez rejestr Y (w tym przypadku
          ;w komórce o adresie $65)
```

4.7.7. Tryb adresowania pośredniego danych z predekrementacją



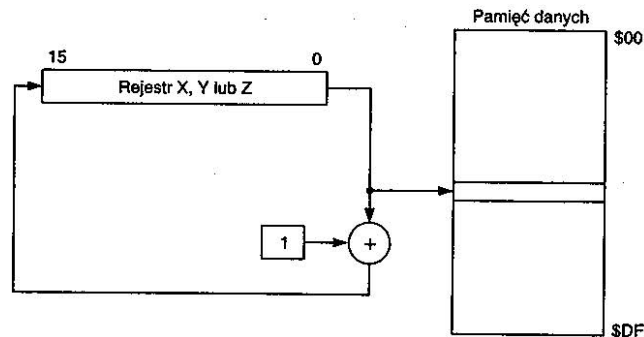
Rys. 4.14. Adresowanie pośrednie z predekrementacją

W tym trybie rejestry indeksowe X, Y i Z są dekrementowane przed wykonaniem operacji. Takie adresowanie doskonale nadaje się do przetwarzania długich struktur danych.

Przykład 4.7. Fragment pętli wypełniającej tablicę danych zawartością rejestru R16. Do wskazywania elementów tablicy wykorzystywany jest rejestr X, do którego przed wejściem do pętli powinien być wpisany odpowiedni adres (następny po ostatnim elemencie tablicy)

```
...
ST  -X,R16 ;wypełnianie kolejnych elementów tablicy
...      ;ewentualne inne operacje i rozkazy realizujące pętlę
```

4.7.8. Tryb adresowania pośredniego danych z postinkrementacją



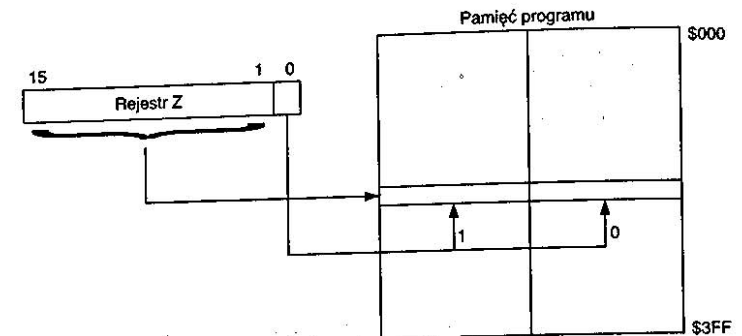
Rys. 4.15. Adresowanie pośrednie z postinkrementacją

Tryb adresowania z postinkrementacją jest podobny do opisanego wyżej. Różnica polega na tym, że rejestry indeksowe X, Y, i Z są inkrementowane po wykonaniu operacji.

Przykład 4.8. Pobieranie do rejestru R0 kolejnych elementów z obszaru pamięci danych wskazywanych przez rejestr Z

```
...
LD  R0,Z+ ;pobranie elementu z automatycznym przesunięciem
      ;wskaznika na kolejny
...      ;ewentualne inne operacje i rozkazy realizujące
      ;pętlę
```

4.7.9. Tryb adresowania stałych z użyciem rozkazu LPM



Rys. 4.16. Adresowanie stałej umieszczonej w pamięci programu z użyciem rozkazu LPM

Ten tryb adresowania służy do obsługi struktur zawierających stałe, umieszczonych w pamięci programu. Adres stałej jest umieszczany w rejestrze Z. Adres słowa pamięci programu (0...1023) jest wybierany przez piętnaście najstarszych bitów rejestru. Bit najmłodszy decyduje o tym, czy odczytywany jest młodszy (LSB=0), czy starszy (LSB=1) bajt słowa programu.

Przykład 4.9. Pobranie do rejestru R0 elementu tablicy umieszczonej w pamięci programu

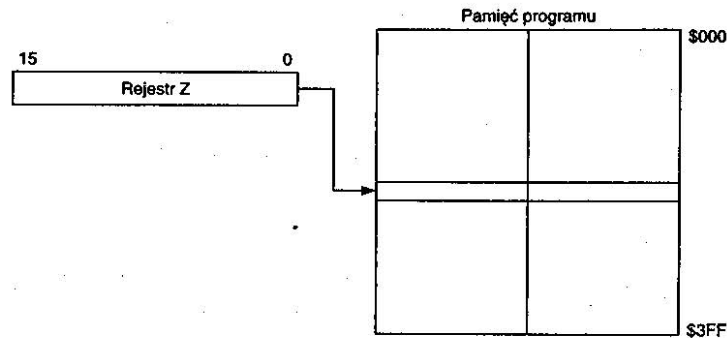
```
LDI  R30,(TAB<<1) ;umieszczenie w młodszej części rejestru Z adresu
                  ;elementu tablicy przesuniętego o jedną pozycję
                  ;w lewo, by mógł być wykorzystany przez rozkaz LPM

LDI  R31,0
LPM                      ;pobranie do rejestru R0 pierwszego elementu
                  ;tablicy TAB. Po wykonaniu rozkazu R0 będzie miał
                  ;wartość $55
```

```
...
TAB: .DB  $55
...
```

4.7.10. Tryb adresowania pośredniego pamięci programu (IJMP, ICALL)

Tryb adresowania pośredniego pamięci programu jest wykorzystywany do realizacji skoków i wywołań podprogramów. Po wykonaniu rozkazu IJMP lub ICALL program jest kontynuowany od adresu umieszczonego w rejestrze Z.



Rys. 4.17. Adresowanie pośrednie pamięci programu wykorzystywane przez rozkazy IJMP i CALL

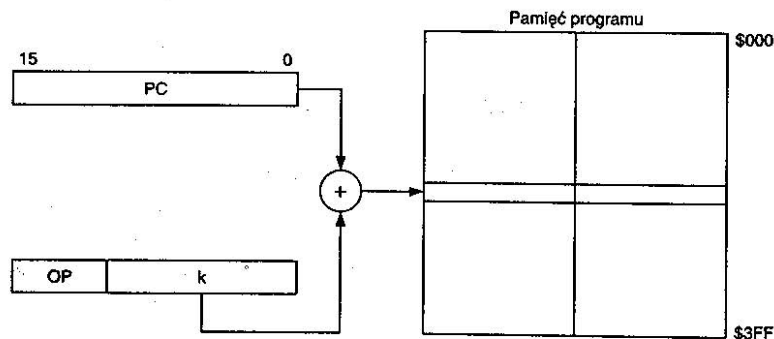
Przykład 4.10. Przeniesienie wykonania programu do adresu opatrzonego etykietą „dalej”

```

LDI    ...    R30, DALEJ
LDI    ...    R31, 0
L0:    IJMP           ;skok do rozkazu z etykietą DALEJ
...
DALEJ: ...

```

4.7.11. Tryb adresowania względnego pamięci programu (RJMP i RCALL)



Rys. 4.18. Adresowanie względne pamięci programu wykorzystywane przez rozkazy RJMP i RCALL

Adresowanie względne pamięci programu umożliwia realizację skoków względnych. Przeniesienie programu następuje do adresu obliczanego formułą $PC = PC + k + 1$. Parametr k jest zapisany w kodzie uzupełnienia do 2 (U2), może przybierać wartości od -2048 do 2047. Skok może być więc wykonany zarówno do przodu jak i do tyłu względem bieżącego stanu licznika programu.



Na liście rozkazów mikrokontrolerów AVR występuje skok bezwarunkowy JMP. Nie został on zaimplementowany w mikrokontrolerze AT90S2313.

Przykład 4.11. Przeniesienie wykonania programu przy wykorzystaniu skoku względnego

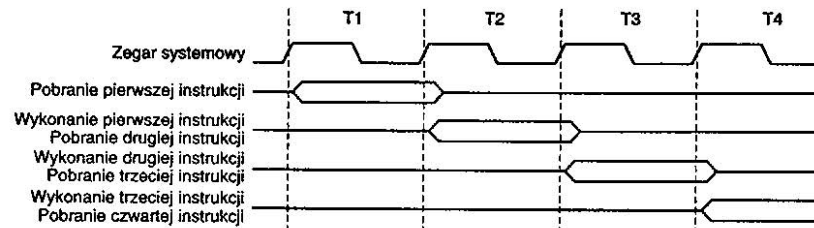
```

BLIZEJ: RJMP DALEJ    ;skok względny do adresu opatrzonego
                    ;etykietą „DALEJ”
...
DALEJ:  RJMP BLIZEJ    ;skok względny do adresu opatrzonego
                    ;etykietą „BLIZEJ”

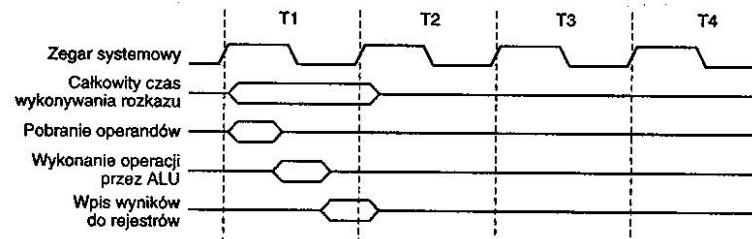
```

4.8. Przebiegi czasowe podczas dostępu do pamięci i wykonywania rozkazów

Mikrokontrolery AVR jako układy synchroniczne wymagają taktowania przebiegiem zegarowym. Odpowiedni jednofazowy przebieg zegarowy jest wytwarzany przez wewnętrzny oscylator, którego częstotliwość pracy jest ustalana zewnętrznym rezonatorem kwarcowym dołączanym do wyprowadzeń XTAL1 i XTAL2. Częstotliwość oscylatora nie jest dzielona tak, jak to ma miejsce w przypadku mikrokontrolerów rodziny 8051, ale czy nie jest wewnętrznie powielana? Wiele wskazuje na to, że układ timingu zawiera w sobie pętlę PLL lub jakiś inny układ powielania częstotliwości podstawowej. Świadczą o tym dość charakterystyczne przesunięcia czasowe niektórych sygnałów sterujących i wysoka szybkość działania mikrokontrolera. Jedyną wątpliwość może budzić deklarowana w pełni statyczna budowa rdzenia. W oficjalnych danych katalogowych udostępnianych przez Atmelę nie ma jednak ani słowa na ten temat. Z punktu widzenia użytkownika nie jest to w sumie istotne, jeśli tylko spełnione są parametry podawane w dokumentacji układu.



Rys. 4.19. Zasada równoległego pobierania i wykonywania rozkazów

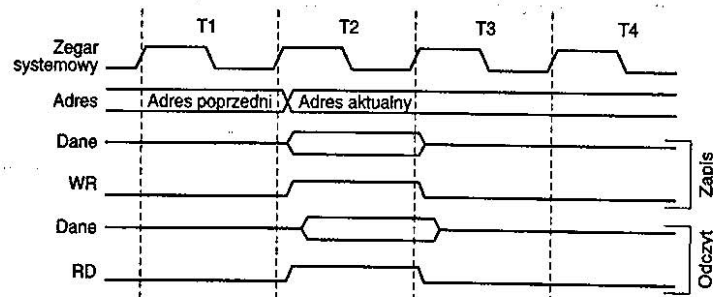


Rys. 4.20. Timing operacji wykonywanej w jednym cyklu zegarowym

Na **rysunku 4.19** przedstawiono zasadę potokowego przetwarzania rozkazów, zgodnie z którą pracują mikrokontrolery AVR. Każdy kolejny rozkaz jest pobierany w tym samym czasie, kiedy wykonywany jest poprzedni. Dzięki temu mechanizmowi wydajność mikrokontrolerów AVR wynosi 1 MIPS/MHz.

Timing operacji wykonywanej w jednym cyklu zegarowym, wykorzystującej zestaw rejestrów przedstawiono na **rysunku 4.20**. Dane są pobierane z dwóch rejestrów, a po wykonaniu operacji jej rezultat jest umieszczany w rejestrze wynikowym.

Nieco inaczej wygląda timing dostępu do wewnętrznej pamięci SRAM. W tym przypadku do wykonania rozkazu niezbędne są dwa cykle zegarowe (**rysunek 4.21**).



Rys. 4.21. Cykle dostępu do wewnętrznej pamięci SRAM

4.9. Przestrzeń we/wy

Wszystkie układy peryferyjne mikrokontrolera AT90S2313 są umiejscowione w obszarze we/wy pamięci SRAM. Lokacje te są dostępne poprzez rozkazy IN i OUT przekazujące dane pomiędzy rejestrami ogólnego przeznaczenia, a obszarem we/wy. Rejestry należące do obszaru od \$00 do \$1F charakteryzują się dostępem bitowym. Poszczególne ich bity mogą być sprawdzane przy wykorzystaniu rozkazów SBIS i SBIC i w zależności od ich stanu przenosić wykonywanie programu w inne miejsce.



Rejestry we/wy do których dostęp odbywa się za pomocą rozkazów IN i OUT muszą być określane adresami od \$00 do \$3F, chociaż w pamięci SRAM zajmują lokacje od \$20 do \$5F. Jeśli rejestry we/wy są adresowane jako pamięć SRAM należy do adresu dodać wartość \$20.



W książce adresy rejestrów we/wy podawane jako adresy pamięci SRAM są ujmowane w nawiasy.

W celu zachowania kompatybilności programu z nowymi wersjami mikrokontrolerów, bity zarezerwowane powinny mieć wartość zero. Zarezerwowanych adresów przestrzeni we/wy nie wolno używać w ogóle.

Niektóre znaczniki (*flags* – w dalszej części książki będą wymiennie nazywane flagami) są zerowane poprzez wpisanie do nich logicznej „1”. Rozkazy CBI i SBI działają na wszystkich bitach rejestrów we/wy. Mogą być jednak wykorzystywane tylko rejestry o adresach od 00 do \$1F. W **tablicy 4.2** zestawiono rejestry specjalne związane z obszarem we/wy.



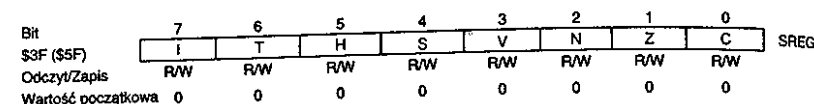
Niektóre wskaźniki są zerowane poprzez wpisanie do nich logicznej „1”.

Tab. 4.2. Rejestry przestrzeni we/wy mikrokontrolera AT90S2313

Adres	Nazwa	Funkcja
\$3F(\$5F)	SREG	Rejestr statusu
\$3D(\$5D)	SPL	Wskaźnik stosu – rejestr mniej znaczący (w AT90S2313 występuje tylko SPL)
\$3B(\$5B)	GIMSK	Rejestr maskowania przerw ogólnych
\$3A(\$5A)	GIFR	Rejestr wskaźników przerw ogólnych
\$39(\$59)	TIMSK	Rejestr maskowania przerw timera/licznika
\$38(\$58)	TIFR	Rejestr wskaźników przerw timera/licznika
\$35(\$55)	MCUCR	Rejestr sterowania MCU
\$33(\$53)	TCCR0	Rejestr sterujący Timera/Licznika0
\$32(\$52)	TCNT0	Timer/Licznik0 (8-bitowy)
\$2F(\$4F)	TCCR1A	Rejestr sterujący A Timera/Licznika1
\$2E(\$4E)	TCCR1B	Rejestr sterujący B Timera/Licznika1
\$2D(\$4D)	TCNT1H	Timer/Licznik1 – starszy bajt
\$2C(\$4C)	TCNT1L	Timer/Licznik1 – młodszy bajt
\$2B(\$4B)	OCR1AH	Rejestr porównania – starszy bajt
\$2A(\$4A)	OCR1AL	Rejestr porównania – młodszy bajt
\$25(\$45)	ICR1H	Rejestr przechwytywania Timera/Licznika1 – starszy bajt
\$24(\$44)	ICR1L	Rejestr przechwytywania Timera/Licznika1 – młodszy bajt
\$21(\$41)	WDTCR	Rejestr sterujący watchdoga
\$1E(\$3E)	EEAR	Rejestr adresowy pamięci EEPROM
\$1D(\$3D)	EEDR	Rejestr danych pamięci EEPROM
\$1C(\$3C)	EECR	Rejestr sterujący pamięci EEPROM
\$18(\$38)	PORTB	Rejestr danych portu B
\$17(\$37)	DDRB	Rejestr kierunku portu B
\$16(\$36)	PINB	Rejestr linii wejściowych portu B
\$12(\$32)	PORTD	Rejestr danych portu D
\$11(\$31)	DDRD	Rejestr kierunku portu D
\$10(\$30)	PIND	Rejestr linii wejściowych portu D
\$0C(\$2C)	UDR	Rejestr danych UART-a
\$0B(\$2B)	USR	Rejestr statusu UART-a
\$0A(\$2A)	UCR	Rejestr sterujący UART-a
\$09(\$29)	UBRR	Rejestr szybkości transmisji UART-a
\$08(\$28)	ACSR	Rejestr sterujący i statusu komparatora analogowego

4.9.1. Funkcje bitów w rejestrach funkcyjnych

SREG (Status Register) – rejestr statusu – \$3F



B7 – I (Global Interrupt Enable): bit globalnego zezwolenia na przerwanie.

Bit ten musi być ustawiony („1”) w celu włączenia wszystkich przerw. Jeśli jest wyzerowany, to niezależnie od ustawienia poszczególnych bitów zezwoleń żadne z przerw nie będzie przyjmowane do obsługi. Indywidualne przerwy są kontrolowane przez wydzielone rejestry. Bit I jest zerowany sprzętowo w momencie przyjęcia obsługi przerwy i ustawiany w chwili zakończenia obsługi przerwy, podczas wykonania rozkazu RETI. Bit ten jest dostępny programowo do odczytu i zapisu.

B6 – T (Copy Storage): bit zachowania kopii.

Rozkazy kopiowania bitów BLD (*Bit Load*) i BST (*Bit Store*) wykorzystują bit T jako źródło i przeznaczenie operacji bitowych. Dany bit rejestru roboczego może być kopiowany do T rozkazem BST i odwrotnie – bit T może być kopiowany do rejestru roboczego rozkazem BLD.

B5 – H (Half-Carry Flag): wskaźnik przeniesienia połówkowego.

Wskaźnik H jest to bit przeniesienia połówkowego w niektórych operacjach arytmetycznych. Jego ustawienie następuje w momencie przeniesienia z jednej połówki bajtu do drugiej. Szczegóły w liście rozkazów. Wskaźnik H jest przydatny podczas wykonywania arytmetyki na liczbach BCD.

Przykład 4.12. Ustawienie flagi H podczas wystąpienia przeniesienia z młodszej półbajtu na starszy

```
LDI    DANA1, $0F
LDI    DANA2, $01      ; H=0
ADD    DANA1, DANA2     ; DANA1=$10, H=1 (przeniesienie połówkowe)
ADD    DANA1, DANA2     ; DANA1=$11, H=0
```

B4 – S (Sign Bit): bit znaku, $S = N \oplus V$

Bit S jest zawsze wyliczany przez funkcję Ex-OR pomiędzy bitami N (wskaźnik wartości ujemnej) i V (przepelnienie uzupełnienia do dwóch). Szczegóły w liście rozkazów.

B3 – V (Two's Complement Overflow Flag): wskaźnik przepełnienia uzupełnienia do dwóch.

Wskaźnik V wspomaga wykonywanie operacji na liczbach w kodzie uzupełnień do dwóch (U2). Jest ustawiany w momencie przekroczenia dopuszczalnej wartości ujemnej w rejestrze.

Przykład 4.13. Ustawienie flagi V po przekroczeniu wartości -128 w rejestrze R16

```
LDI    R16, -127 ;V=0
DEC    R16       ;R16=-128, V=0
DEC    R16       ;R16=-129, ale nie może przyjąć takiej wartości,
                ;więc V=1
```

B2 – N (Negative Flag): wskaźnik wartości ujemnej.

Wskaźnik N jest ustawiany, zawsze gdy wynik operacji arytmetycznej przyjmuje wartość ujemną. Znak liczby zapisanej w kodzie U2 jest zapisany na najstarszym bicie, tak więc wskaźnik N zawsze będzie kopią tego bitu. Nie dotyczy to sytuacji, w których zostanie on w sposób jawny zmieniony.

B1 – Z (Zero Flag): wskaźnik zera.

Wskaźnik Z jest ustawiany wtedy, gdy w wyniku wykonania operacji arytmetycznej lub logicznej wynik jest równy 0.

B0 – C (Carry Flag): wskaźnik przeniesienia.

Wskaźnik C sygnalizuje przeniesienie podczas operacji arytmetycznych.

SPL (Stack Pointer) - wskaźnik stosu – \$3D

Bit	7	6	5	4	3	2	1	0	
\$3D (\$5D)	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0	SPL
Odczyt/Zapis	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Wartość początkowa	0	0	0	0	0	0	0	0	

Jest to 8-bitowy rejestr pełniący funkcję wskaźnika stosu. Za jego pomocą można zaadresować 128 bajtów pamięci SRAM o adresach od \$60 do \$DF. Programista powinien zapewnić odpowiednią wielkość pamięci przeznaczoną na stos. Zależy ona od liczby zagnieżdżeń podprogramów i argumentów przekazywanych przez stos (co jest często praktykowane przez kompilatory języków wysokiego poziomu). Trzeba także pamiętać, że pojawiające się asynchronicznie przerwania również wykorzystują stos. Wskaźnik stosu powinien być zawsze ustawiany na początku programu, w przeciwnym przypadku nieuchronnie dojdzie do błędnego działania mikrokontrolera lub całkowitego zawieszenia programu. SPL wskazuje na aktualnie zapisywany adres stosu. Po odłożeniu danej na stosie rozkazem PUSH, zawartość wskaźnika

SPL zostaje zmniejszona o 1. W przypadku odkładania adresu powrotu dla rozkazów wywołań podprogramów i procedur obsługi przerwania zawartość rejestru SPL jest zmniejszana o 2. Po zdjęciu danej ze stosu rozkazem POP zawartość rejestru SPL jest zwiększana o 1, a po wykonaniu powrotu z podprogramu (RET) lub obsługi przerwania (RETI) jest zwiększana o 2.

W „większych” odmianach mikrokontrolerów AVR wskaźnik stosu jest rejestrem 16-bitowym, złożonym z dwóch rejestrów 8-bitowych: SPH i SPL.

4.10. Zerowanie i wektory przerwania

Jednym z podstawowych zadań mikrokontrolera jest sterowanie różnymi urządzeniami peryferyjnymi. Najczęściej – można powiedzieć, że prawie zawsze – pracują one asynchronicznie w stosunku do programu realizowanego przez CPU. Mikrokontroler nie potrafi w dosłownym znaczeniu obsługiwać wszystkich zdarzeń jednocześnie, chociaż obserwator przyglądający się jego pracy z zewnątrz może mieć takie wrażenie. Jedną z metod „panowania” nad systemem jest np. sekwencyjne przeglądanie stanu poszczególnych urządzeń i w razie wystąpienia konieczności jego obsługi wykonanie odpowiedniej procedury. Takie rozwiązanie powoduje jednak bezproduktywne marnowanie czasu przez ciągłe sprawdzanie urządzeń, które w danym momencie nie wymagają obsługi. Sytuacja staje się dramatyczna, gdy urządzenia są uszeregowane według określonych priorytetów. Aby było możliwe zapewnienie warunków obsługi priorytetowej należałoby wówczas bardzo rozbudować program. Rozwiązaniem powyższych problemów jest wykorzystanie systemu przerwania. Program mikrokontrolera w takim przypadku składa się z pętli, w której wykonuje zadania wynikające z wymagań konkretnej aplikacji. W chwili, gdy dane urządzenie peryferyjne wymaga obsługi, zgłasza ten fakt do mikrokontrolera poprzez przerwanie. W efekcie następuje zawieszenie wykonywania aktualnej operacji i przejście do procedury obsługi tego przerwania. Jeśli teraz nastąpi żądanie od innego urządzenia, to gdy ma ono wyższy priorytet od bieżącego może nastąpić kolejne zawieszenie wykonywanej operacji i przejście do obsługi przerwania o wyższym priorytecie. Jeśli priorytet drugiego urządzenia jest niższy, to musi ono czekać do momentu zakończenia obsługi urządzenia o wyższym priorytecie. Skok do procedur obsługi odbywa się zawsze ze śladem (adres powrotu odkładany na stosie). Po zakończeniu działań sterowanie jest przekazywane do miejsca, w którym wystąpiło żądanie obsługi. W ten sposób uzyskuje się wrażenie jednoczesnego, niezależnego funkcjonowania wielu asynchronicznych urządzeń.

Tab. 4.3. Wektory obsługi przerwań

Numer wektora	Adres	Źródło	Opis
1	\$000	RESET	Zewnętrzne wejście zerujące, sygnału power-on reset i watchdoga
2	\$001	INT0	Zewnętrzne przerwanie INT0
3	\$002	INT1	Zewnętrzne przerwanie INT1
4	\$003	TC1 CAPT	Przerwanie od TC1 – wystąpiło przechwycenie
5	\$004	TC1 COMP	Przerwanie od TC1 – wykryto równość
6	\$005	TC1 OVF	Przerwanie od TC1 – przepełnienie Timera/Licznika1
7	\$006	TC0 OVF	Przerwanie od TC0 – przepełnienie Timera/Licznika0
8	\$007	UART RX	Przerwanie od układu UART – odebrano znak
9	\$008	UART UDRE	Przerwanie od układu UART – rejestr nadajnika pusty
10	\$009	UART TX	Przerwanie od układu UART – wysłano znak
11	\$00A	ANA_COMP	Przerwanie od komparatora analogowego

W praktyce najczęściej nie jest tak „różowo” jak podano w opisie. Pisząc program trzeba na ogół dbać o to, aby procedury obsługi przerwań były bardzo starannie optymalizowane pod względem czasowym.

Wszystkie urządzenia obsługiwane przez system przerwań mają przydzielone tzw. wektory przerwań, czyli ściśle określone adresy pamięci programu, pod którymi są umieszczane procedury obsługi. Dokładniej pod adresem tym musi się zaczynać taka procedura, najczęściej znajduje się tam jeden rozkaz skoku do innego adresu pamięci programu, gdzie znajdują się dalsze instrukcje. Specjalny wektor przeznaczono dla zerowania. W mikrokontrolerze AT90S2313 przewidziano 11 różnych źródeł przerwań. Są one opisane w tabeli 4.3. Wszystkie przerwania mogą być indywidualnie odblokowywane lub blokowane poprzez ustawienie („1”) lub zerowanie („0”) odpowiedniego bitu zezwolenia znajdującego się w przydzielonym do danego urządzenia rejestrze sterującym. Można także włączyć lub wyłączyć cały system przerwań ustawiając lub zerując bit I w rejestrze statusu SREG.

Typowy fragment inicjujący program przedstawiono w poniższym przykładzie.

Przykład 4.14. Typowy fragment programu inicjujący pracę mikrokontrolera

```

$000 RJMP RESET      ;uchwyt dla resetu
$001 RJMP EXT_INT0   ;uchwyt dla INT0
$002 RJMP EXT_INT1   ;uchwyt dla INT1
$003 RJMP TIM_CAPT1  ;uchwyt dla przechwycenia Timer1
$004 RJMP TIM_COMP1  ;uchwyt dla równości Timer1
$005 RJMP TIM_OVF1   ;uchwyt dla przepełnienia Timer1
$006 RJMP TIM_OVF0   ;uchwyt dla przepełnienia Timer0
$007 RJMP UART_RXC   ;uchwyt dla odebranego znaku przez UART
$008 RJMP UART_DRE   ;uchwyt dla pustego nadajnika UART-a
$009 RJMP UART_TXC   ;uchwyt dla wysłanego znaku przez UART
$00A RJMP ANA_COMP    ;uchwyt dla komparatora analogowego
$00B MAIN: LDI R16,LOW(RAMEND);start programu głównego
$00C      OUT SPL,R16      ;ustawienie wskaźnika stosu
$00D      ...

```

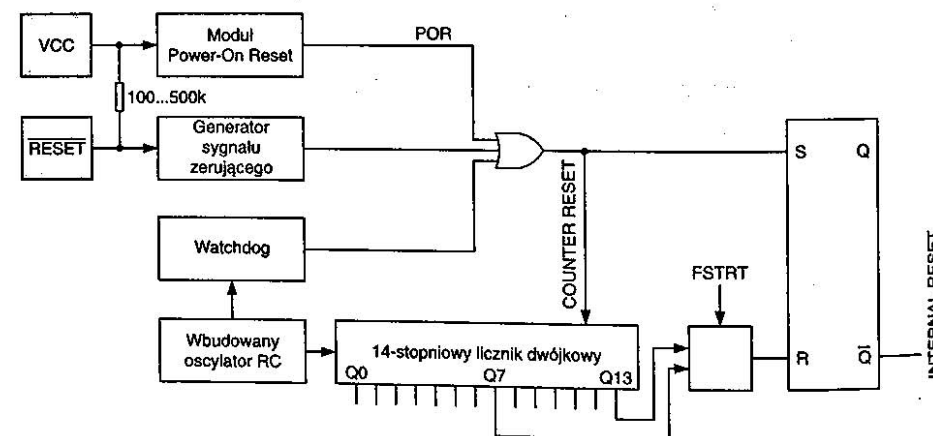
4.10.1. Źródła sygnału zerującego

W mikrokontrolerze AT90S2313 przewidziano trzy źródła sygnału zerowania:

- *Power-on Reset* – mikrokontroler jest zerowany, gdy napięcie zasilające spadnie poniżej wartości progowej V_{POT} (rysunek 4.23).
- *External Reset* – mikrokontroler jest zerowany, gdy na wejściu \overline{RESET} wystąpi niski stan przez co najmniej 50 ns.
- *Watchdog Reset* – mikrokontroler jest zerowany, gdy licznik watchdog nie zostanie w porę wyzerowany (w sytuacji gdy watchdog jest uaktywniony).

Podczas zerowania mikrokontrolera wszystkie rejestry we/wy są ustawiane zgodnie z przyjętymi dla nich stanami początkowymi. Licznik programu jest zerowany, tak więc pierwszy rozkaz jest wykonany spod adresu \$000. Jeśli planowane jest wykorzystywanie przerwań, musi tu być umieszczony skok RJMP do adresu, pod którym znajdują się instrukcje inicjujące pozostałe elementy systemu. Jeśli nie jest przewidywane wykorzystywanie przerwań kolejne adresy pamięci programu odpowiadające wektorom przerwań nie będą potrzebne. W takim przypadku program może być liniowo kontynuowany od adresu \$000. Na rysunku 4.22 przedstawiono schemat układu sprzętowego zerowania, natomiast w tabeli 4.4 zestawiono parametry elektryczne i czasowe układu zerowania.

Użytkownik może wybrać odpowiedni czas startu w zależności od typowego czasu wzbudzenia oscylatora. Liczbę cykli oscylatora watchdoga powodującą przekroczenie limitu czasu (*time-outu*) podano w tabeli 4.5. Należy pamiętać, że częstotliwość impulsów zliczanych przez watchdog jest zależna od napięcia.



Rys. 4.22. Schemat układu zerowania zastosowanego w mikrokontrolerach AVR

Tab. 4.4. Parametry elektryczne i czasowe układu zerującego

Symbol	Parametr	Min.	Typ.	Maks.	Jednostki
V_{POT}	Napięcie progowe wyzwolenia układu zerującego po włączeniu zasilania (napięcie narasta)	1,0	1,4	1,8	V
	Napięcie progowe wyzwolenia układu zerującego po wyłączeniu zasilania (napięcie opada)	0,4	0,6	0,8	V
V_{RST}	Napięcie progowe na wejściu RESET			$0,85 V_{CC}$	V
t_{TOUT}	Opóźnienie od wyzwolenia do końca zerowania FSTRT nie zaprogramowany	11	16	21	ms
t_{TOUT}	Opóźnienie od wyzwolenia do końca zerowania FSTRT zaprogramowany	0,25	0,28	0,31	ms

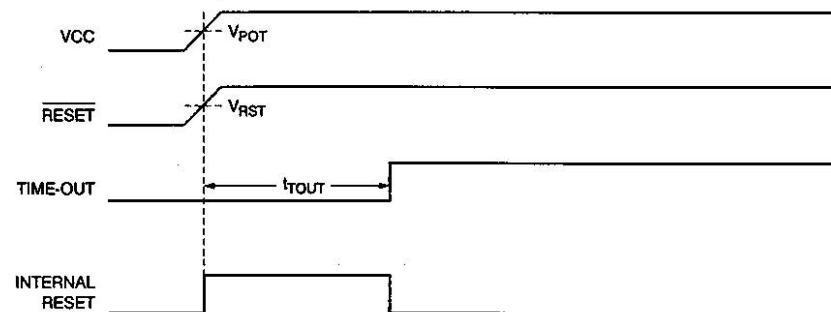
Uwaga! Sygnał Power-on Reset nie będzie generowany dopóki napięcie zasilające spadnie poniżej V_{POT} .

Tab. 4.5. Liczba cykli oscylatora watchdoga

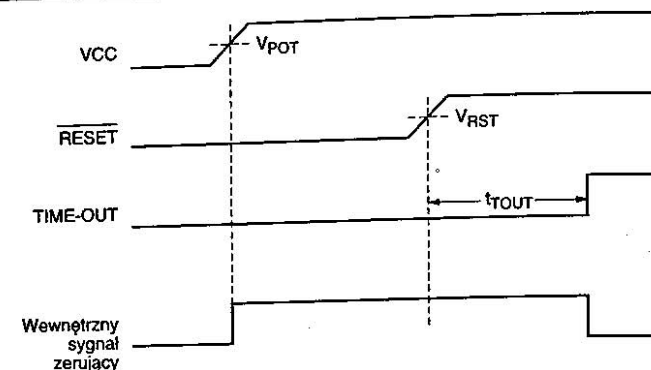
Bit konfiguracyjny FSTRT	Time-out dla $V_{CC}=5V$	Liczba cykli WDT
zaprogramowany	0,28 ms	256
nie zaprogramowany	16,0 ms	16 k

Zerowanie automatyczne Power-on Reset

W układzie zerowania występuje kilka bloków odpowiedzialnych za generowanie sygnału zerującego w różnych sytuacjach. Układ *Power-on Reset* (POR) wytwarza sygnał zerowania w momencie załączenia zasilania. Jak widać na rysunku 4.22, w układzie zerującym znajduje się wewnętrzny licznik taktowany z oscylatora watchdoga. Zapobiega on rozpoczęciu pracy przez mikrokontroler przed osiągnięciem przez napięcie zasilające wartości progowej V_{POT} (rysunek 4.23).

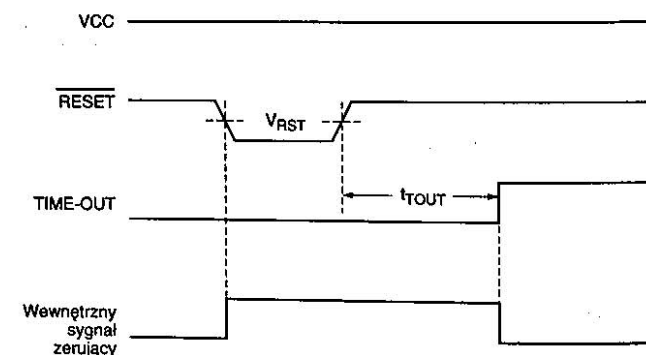


Rys. 4.23. Generowanie wewnętrznego sygnału zerującego po włączeniu napięcia zasilającego, gdy wejście RESET jest dołączone do VCC



Rys. 4.24. Generowanie wewnętrznego sygnału zerującego po włączeniu napięcia zasilającego, gdy linia RESET jest sterowana zewnętrznym

Użytkownik ma możliwość zdecydowania jak szybko powinien wystartować mikrokontroler po włączeniu zasilania. Czyni to przez odpowiednie zaprogramowanie bitu konfiguracyjnego (*fuse bit*) FSTRF w pamięci Flash. Jego zaprogramowanie jest wskazane, gdy generator taktujący współpracuje z rezonatorem kwarcowym lub gdy inne źródło przebiegu zegarowego gwarantuje odpowiednio szybkie wzbudzenie po włączeniu zasilania. Jeśli „wbudowane” opóźnienie startu jest wystarczające, wejście RESET może być podłączone bezpośrednio lub poprzez rezystor podciągający do V_{CC} . Niski stan logiczny na wejściu RESET powinien być utrzymany przez czas, w którym napięcie zasilające osiągnie stan ustalony (rysunek 4.24). W praktyce bywa z tym różnie. Z tego względu powszechne jest stosowanie specjalizowanych układów generujących sygnał zerowania w sposób prawidłowy.



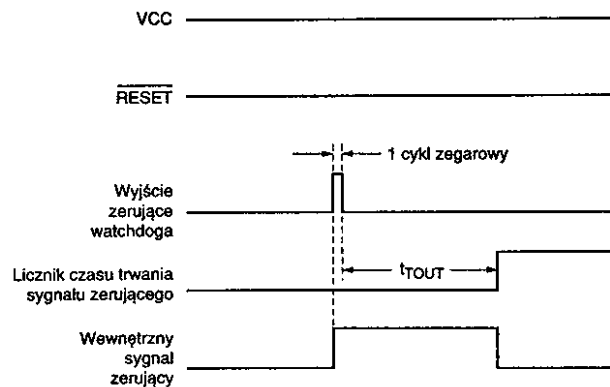
Rys. 4.25. Zerowanie mikrokontrolera za pomocą zewnętrznego sygnału podawanego na wejście RESET

Zerowanie zewnętrzne

Zewnętrzne zerowanie jest generowane przez podanie sygnału niskiego na wejście $\overline{\text{RESET}}$. Powinno być utrzymane w tym stanie przez czas nie krótszy niż 50 ns nawet, gdy zegar systemowy nie chodzi. Krótsze impulsy nie gwarantują prawidłowego zerowania. Wykrycie stanu poniżej wartości progowej V_{RST} powoduje wygenerowanie wewnętrznego sygnału zerowania (rysunek 4.25). Ponowne przekroczenie tego progu na narastającym zboczu sygnału $\overline{\text{RESET}}$ inicjuje pracę timera opóźnienia startu. Po przekroczeniu limitu czasu t_{TOUT} mikrokontroler zostaje wyzerowany i rozpoczyna pracę od początku.

Zerowanie wywołane przez watchdoga

Przekroczenie limitu czasu określonego przez watchdog powoduje wygenerowanie krótkiego impulsu zerującego, trwającego jeden cykl oscylatora XTAL (rysunek 4.26). Jednocześnie z narastającym zboczem tego impulsu pojawia się wewnętrzny sygnał zerowania. Na opadającym zboczu inicjowany jest timer opóźnienia startu. Po przekroczeniu przez niego *time-out*'u t_{TOUT} mikrokontroler zostaje wyzerowany i rozpoczyna pracę od początku.



Rys. 4.26. Sygnał zerowania od watchdoga

4.10.2. Uchwyty przerwań

Mikrokontroler AT90S2313 ma dwa 8-bitowe rejestry wykorzystywane przez system przerwań. Są to: GIMSK (*General Interrupt Mask Register*) i TIMSK (*Timer/Counter Interrupt Mask Register*). Po wystąpieniu przerwania bit glo-

balnego zezwolenia na przerwanie jest zerowany, w wyniku czego wszystkie przerwanie zostają automatycznie zablokowane. Jeśli użytkownikowi zależy, aby w trakcie obsługi tego przerwania było możliwe zgłaszanie innych (o wyższym priorytecie), musi ręcznie ustawić bit I. Jeśli tego nie zrobi, wszystkie przerwanie, które ewentualnie nadejdą w międzyczasie, zostaną ustawione w kolejce według priorytetów obsługi.



Niektóre flagi przerwań są zerowane poprzez wpisywanie logicznej „1” na pozycję zerowanego bitu.

Wyjście z procedury obsługi każdego przerwania realizuje rozkaz RETI, który przywraca stan bitu I (ustawia go na „1”). Zgłoszenie przerwania powoduje ustawienie odpowiedniej dla niego flagi. W przypadku, gdy zgłoszenie przerwania nastąpi w chwili, gdy bit I jest wyzerowany, przerwanie nie zostanie obsłużone. Jeśli teraz flaga zgłoszonego przerwania zostanie wyzerowana (bez wchodzenia do procedury obsługi), to ponowne jej ustawienie będzie możliwe dopiero w momencie nadejścia następnego żądania obsługi. W ten sposób jedno zgłoszenie może zostać utracone. Trzeba jednak pamiętać o tym, że nieco inaczej wygląda obsługa przerwań zgłaszanych poziomem, kiedy to żądanie obsługi pozostaje ważne przez cały czas utrzymywania się niskiego poziomu na wejściu INTx. Flaga danego przerwania jest automatycznie zerowana, gdy do licznika programu zostanie wpisany (automatycznie) adres skoku odpowiadający temu przerwaniu.



Przerwanie zewnętrzne zgłaszane poziomem nie ma przypisanej flagi przerwania. Żądanie przerwania pozostaje aktywne tak długo, jak długo występuje fizycznie sygnał przerwania.

Niektóre flagi przerwań są zerowane poprzez wpisywanie logicznej „1” na pozycję zerowanego bitu. Jeśli wystąpi zgłoszenie przerwania, gdy odpowiadający mu bit zezwolenia jest wyzerowany, flaga przerwania zostanie ustawiona i zapamiętana do czasu ponownego odblokowania przerwania lub ręcznego jej wyzerowania. W sytuacji, gdy bit globalnego zezwolenia na przerwanie jest wyzerowany i następują kolejne zgłoszenia przerwań, odpowiednie flagi zostają ustawione i zapamiętane. Po ustawieniu bitu I wszystkie przerwanie zostają obsługiwane kolejno według priorytetów.

B7 – TOIE1 (Timer/Counter1 Overflow Interrupt Enable): zezwolenie na przerwanie od układu timera/licznika TC1 spowodowane przepełnieniem.

Przerwanie od przepełnienia timera/licznika TC1 jest odblokowane, gdy bit TOIE1 jest ustawiony („1”) i bit I (SREG) jest ustawiony („1”). W chwili, gdy nastąpi przepełnienie timera/licznika TC1 zostanie wygenerowane przerwanie, po przyjęciu którego nastąpi skok do procedury obsługi (\$005). Żądanie obsługi przerwania jest sygnalizowane ustawieniem bitu TOV1 w rejestrze TIFR.

B6 – OCIE1A (Timer/Counter1 Output Compare Match Interrupt Enable): zezwolenie na przerwanie od układu timera/licznika TC1 spowodowane osiągnięciem wartości Timera/Licznika1 zgodnej z porównywaną wartością.

Przerwanie od osiągnięcia zadanego stanu timera/licznika TC1 jest odblokowane, gdy bit OCIE1A jest ustawiony („1”) i bit I (SREG) jest ustawiony („1”). W momencie, gdy timer/licznik TC1 osiągnie zadaną wartość zostanie wygenerowane przerwanie, po przyjęciu którego nastąpi skok do procedury obsługi (\$004). Żądanie obsługi przerwania jest sygnalizowane ustawieniem bitu OCF1A w rejestrze TIFR.

B5...4 – zarezerwowane.

Te bity nie są wykorzystywane w AT90S2313 i zawsze odczytywane jako zero.

B3 – TICIE1 (Timer/Counter1 Input Capture Interrupt Enable): zezwolenie na przerwanie w wyniku wystąpienia przechwycenia.

Gdy bit TICIE1 jest ustawiony („1”) i bit I (SREG) jest ustawiony („1”), przerwanie od przechwycenia zawartości licznika są odblokowane. W chwili, gdy nastąpi wyzwolenie przechwycenia na wejściu PD6 (ICP) nastąpi skok do procedury obsługi (\$003). Żądanie obsługi przerwania jest sygnalizowane ustawieniem bitu ICF1 w rejestrze TIFR.

B2 – zarezerwowany.

Ten bit nie jest wykorzystywany w AT90S2313 i zawsze odczytywany jako zero.

B1 – TOIE0 (Timer/Counter0 Overflow Interrupt Enable): zezwolenie na przerwanie od układu timera/licznika TC0 spowodowane przepełnieniem.

Przerwanie od przepełnienia timera/licznika TC0 jest odblokowane, gdy bit TOIE0 jest ustawiony („1”) i bit I (SREG) jest ustawiony („1”). W chwili,

gdy nastąpi przepełnienie timera/licznika TC0, zostanie wygenerowane przerwanie, po przyjęciu którego nastąpi skok do procedury obsługi (\$006). Żądanie obsługi przerwania jest sygnalizowane ustawieniem bitu TOV0 w rejestrze TIFR.

B0 – zarezerwowany.

Ten bit nie jest wykorzystywany w AT90S2313 i jest zawsze odczytywany jako zero.

TIFR (Timer/Counter Interrupt Flag Register) – rejestr wskaźników zgłoszenia przerw od timerów/liczników – \$38

Bit	7	6	5	4	3	2	1	0	
\$38 (\$58)	TOV1	OCF1A	–	–	ICF1	–	TOV0	–	TIFR
Odczyt/Zapis	R/W	R/W	R	R	R/W	R	R/W	R	
Wartość początkowa	0	0	0	0	0	0	0	0	

B7 – TOV1 (Timer/Counter1 Overflow Flag): flaga przepełnienia timera/licznika TC1.

Flaga TOV1 jest ustawiana („1”), gdy nastąpi przepełnienie timera/licznika TC1. Jest ona sprzętowo zerowana w momencie przeniesienia sterowania do określonego wektora przerw. Alternatywnie bit ten może być również zerowany programowo po wpisaniu logicznej „1” na pozycji 7 rejestru TIFR. Jeśli ustawione są bity I (w rejestrze SREG), TOIE1 (w rejestrze TIMSK) i TOV1, to wykonywana jest procedura obsługi przerwania od przepełnienia timera/licznika TC1. W trybie PWM bit TOV1 jest ustawiany, gdy timer/licznik TC1 zmieni kierunek zliczania po osiągnięciu stanu \$0000.

B6 – OCF1A (Output Compare Flag 1A): flaga porównania 1A (osiągnięcia zadanej wartości przez licznik TC1).

Flaga OCF1A zostaje automatycznie ustawiona („1”) w chwili zrównania stanów timera/licznika TC1 i rejestru porównania OCR1A (Output Compare Register1 A). OCF1A jest zerowana sprzętowo w momencie przeniesienia sterowania do określonego wektora przerw. Alternatywnie bit ten może być również zerowany programowo po wpisaniu logicznej „1” na pozycji 6 rejestru TIFR. Jeśli ustawione są bity I (w rejestrze SREG), OCIE1A (w rejestrze TIMSK) i OCF1A, to wykonywana jest procedura obsługi przerwania od porównania timera/licznika TC1.

B5...4 – zarezerwowane.

Te bity nie są wykorzystywane w AT90S2313 i zawsze odczytywane jako zero.

Bit	7	6	5	4	3	2	1	0
\$35 (\$55)	-	-	SE	SM	ISC11	ISC10	ISC01	ISC00
Odczyt/Zapis	R	R	R/W	R/W	R/W	R/W	R/W	R/W
Wartość początkowa	0	0	0	0	0	0	0	0

MCUCR

B7...6 – zarezerwowane.

Te bity nie są wykorzystywane w AT90S2313 i zawsze odczytywane jako zero.

B5 – SE (Sleep Enable): zezwolenie na wejście w tryb obniżonego poboru mocy (SLEEP).

Bit SE musi być ustawiony („1”) jeśli mikrokontroler ma wchodzić w stan uśpienia po wykonaniu rozkazu SLEEP. Aby uniknąć niepożądanego uśpienia mikrokontrolera, zalecane jest ustawianie bitu SE bezpośrednio przed wykonaniem rozkazu SLEEP.

B4 – SM (Sleep Mode): tryb uśpienia.

Za pomocą bitu SM ustawia się jeden z dwóch trybów uśpienia mikrokontrolera. Jeśli SM jest wyzerowany („0”), to wybrany jest tryb *Idle*. Jeśli SM jest ustawiony („1”), to wybrany jest tryb *Power-Down* (tryby uśpienia są dokładnie opisane w rozdziale 4.11).

B3...B2 – ISC11, ISC10 (Interrupt Sense Control 1 bit 1 i 0): wybór rodzaju sygnału wyzwalania przerwania INT1.

Przerwanie zewnętrzne 1 jest aktywowane poprzez wyprowadzenie INT1, jeśli bit 1 w rejestrze SREG jest ustawiony i odpowiednia maska przerwania w rejestrze GIMSK jest ustawiona. Rodzaj sygnału wyzwalającego przerwanie INT1 zależy od ustawienia bitów ISC11 i ISC10 (tablica 4.6).

Tab. 4.6. Wybór typu sygnału zgłaszającego przerwanie INT1

ISC11	ISC10	Opis
0	0	Wyzwalanie przerwania poziomem niskim na wejściu INT1
0	1	Zarezerwowane
1	0	Wyzwalanie przerwania zboczem opadającym
1	1	Wyzwalanie przerwania zboczem narastającym

B1...B0 – ISC01, ISC00 (Interrupt Sense Control 0 bit 1 i 0): wybór rodzaju sygnału wyzwalania przerwania INT0.

Przerwanie zewnętrzne 0 jest aktywowane poprzez wyprowadzenie INT0, jeśli bit 1 w rejestrze SREG jest ustawiony i odpowiednia maska przerwania

Tab. 4.7. Wybór typu sygnału zgłaszającego przerwanie INT0

ISC01	ISC00	Opis
0	0	Wyzwalanie przerwania poziomem niskim na wejściu INT0
0	1	Zarezerwowane
1	0	Wyzwalanie przerwania zboczem opadającym
1	1	Wyzwalanie przerwania zboczem narastającym

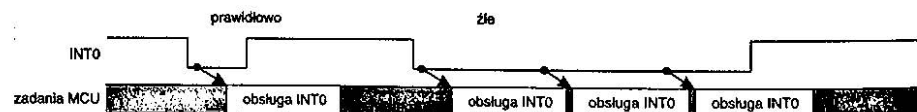
w rejestrze GIMSK jest ustawiona. Rodzaj sygnału wyzwalającego przerwanie INT0 zależy od ustawienia bitów ISC01 i ISC00 (tablica 4.7)

UWAGA

W trybie wyzwalania przerwania poziomem, żądanie obsługi przerwania pozostaje aktywne tak długo, jak długo utrzymuje się niski poziom na odpowiednim wejściu. Jeśli nie zadba się o to, by w procedurze obsługi zablokować zgłoszenia przerwania, to wystąpi cykliczne generowanie przerwania, mogące zawiesić pracę mikrokontrolera.

Stan końcówki INTn jest próbkowany w celu wykrycia zbocza sygnału żądania przerwania. Jeśli wybrano wyzwalanie zboczem, to impuls na tej końcówce nie powinien być krótszy od okresu zegara systemowego. Tylko spełnienie tego warunku gwarantuje przyjęcie zgłoszenia przerwania. W przypadku wybrania wyzwalania poziomem, sygnał żądania przerwania powinien utrzymać się w stanie niskim co najmniej przez czas wykonywania bieżącego rozkazu.

W trybie wyzwalania przerwania poziomem, żądanie obsługi przerwania pozostaje aktywne tak długo, jak długo utrzymuje się niski poziom na odpowiednim wejściu. Jeśli nie zadba się o to, by w procedurze obsługi zdjąć na czas sygnał zgłoszenia przerwania, to wystąpi najprawdopodobniej niepożądane, cykliczne generowanie przerwania, mogące zawiesić pracę mikrokontrolera (rysunek 4.27).



Rys. 4.27. Zakłócenie pracy MCU w przypadku zbyt długiego sygnału zgłoszenia przerwania (przerwanie zgłaszane poziomem)

4.11. Tryby oszczędzania energii

Mikrokontroler AT90S2313 w normalnym trybie pracy, przy zasilaniu napięciem 4 V i z dołączonym rezonatorem 4 MHz pobiera z zasilania prąd ok. 2,8 mA. Niby nie jest to dużo, ale oprócz niego w systemie znajduje się jeszcze najczęściej kilka innych odbiorników energii. Po wykonaniu bilansu okazuje się, że łączne zużycie energii przez urządzenie nie kwalifikuje go do za-

silania baterijnego. Nawet jeśli tak będzie, to trzeba pamiętać o niemalym problemie ekologicznym związanym z utylizacją zużytych ogniw. Zawsze, jeśli tylko to możliwe trzeba więc dbać o minimalizację prądu pobieranego ze źródła zasilania. Mikrokontrolery AVR wyposażono w specjalne tryby pracy z obniżonym poborem mocy, tzw. tryby uśpienia. Okazuje się, że w praktyce bardzo często występują sytuacje, w których mikrokontroler przez długi czas nie wykonuje żadnych zadań, czekając na nie w martwej pętli. Pomysł nasuwa się więc sam: w chwilach oczekiwania trzeba wyłączyć rdzeń mikrokontrolera, w taki jednak sposób, by mógł on w każdej chwili automatycznie wznowić pracę. Służą do tego tryby uśpienia – *Sleep Modes*. W celu wprowadzenia mikrokontrolera w stan uśpienia bit SE rejestru MCUR musi być ustawiony („1”) i niezbędne jest wykonanie rozkazu SLEEP. Jeśli w tym stanie do mikrokontrolera dotrze zgłoszenie odblokowanego przerwania, to mikrokontroler zostanie obudzony, wykonana procedura obsługi przerwania i przejdzie do rozkazu występującego bezpośrednio za rozkazem SLEEP. Wszystkie zasoby MCU (rejstry, pamięć SRAM, układy we/wy) pozostają nienaruszone. Przebudzenie mikrokontrolera może również nastąpić po sygnale $\overline{\text{RESET}}$. W tym przypadku program będzie wykonywany od wektora resetu, czyli od adresu \$000.

4.11.1. Tryb *Idle*

Wyzerowanie bitu SM („0”) powoduje, że po wykonaniu rozkazu SLEEP mikrokontroler przechodzi do trybu *Idle*, wstrzymując pracę CPU, zachowując natomiast aktywne timery/liczniki, watchdoga i system przerwań. Przebudzenie mikrokontrolera jest możliwe zarówno po nadejściu przerwania zewnętrznego, jaki i wystąpieniu któregoś z przerwań wewnętrznych, np. po przepełnieniu timera/licznika lub wyzerowaniu systemu przez watchdoga. Jeśli nie jest wskazane przebudzenie od przerwania komparatora analogowego, powinien on być programowo wprowadzony w stan *Power-Down* poprzez ustawienie bitu ACD w rejestrze sterującym komparatora – ASCR (*Analog Comparator Control and Status*). Po wyprowadzeniu mikrokontrolera ze stanu *Idle* wznowia on swoją pracę natychmiast.

4.11.2. Tryb *Power-Down*

Ustawienie bitu SM („1”) powoduje, że po wykonaniu rozkazu SLEEP mikrokontroler przechodzi do trybu *Power-Down*. W tym stanie jest zatrzymywany generator taktujący, lecz przerwania zewnętrzne i watchdog są nadal

aktywne (jeśli nie są programowo zablokowane). Wybudzić mikrokontroler z tego stanu mogą jedynie: zewnętrzny sygnał zerowania, sygnał zerowania od watchdoga (jeśli nie jest on zablokowany programowo) oraz zewnętrzne przerwania na wejściu INT1 lub INT0 wyzwalane poziomem.

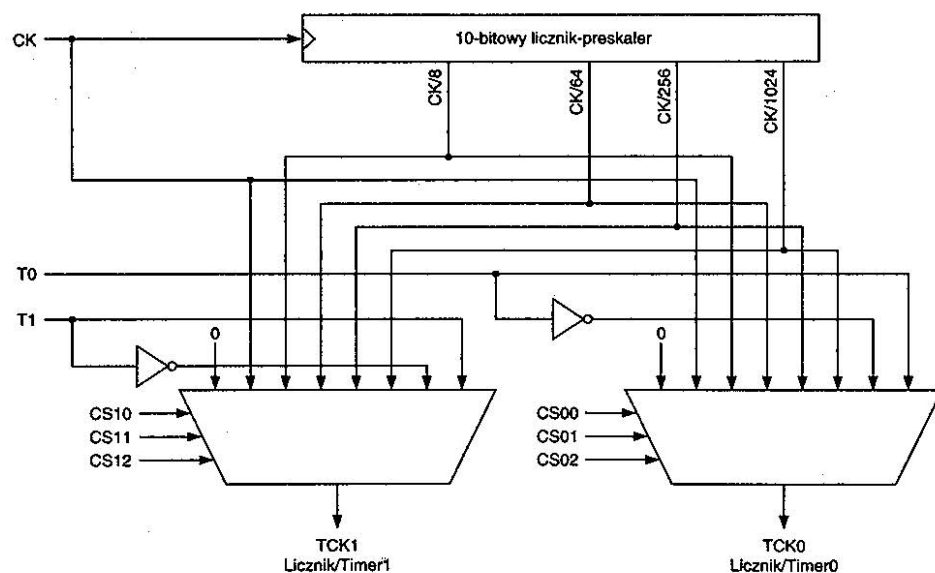


Jeśli do obudzenia mikrokontrolera ze stanu *Power-Down* stosuje się przerwania wyzwalane poziomem, to wejście zgłaszające przerwanie powinno być aktywne przez czas dłuższy niż wynosi czas opóźnienia zerowania t_{ROUT} . W przeciwnym przypadku mikrokontroler może nie obudzić się ze stanu *Power-Down*.

5. Timery/liczniki

Mikrokontroler AT90S2313 wyposażono w dwa timery/liczniki ogólnego przeznaczenia. Jeden z nich jest 8-, drugi 16-bitowy. W celu rozszerzenia zakresu ich zliczania wprowadzono dodatkowy licznik 10-bitowy, pełniący funkcję wstępnego dzielnika impulsów taktujących. Jest to tzw. preskaler. Jego konfigurację pokazano na **rysunku 5.1**.

Jeśli jest wykorzystywany sygnał CK aktywnego generatora (taktującego również mikrokontroler), to mamy do czynienia z timerem, natomiast gdy jest wykorzystywany sygnał zewnętrzny T0 (dla układu TC0) lub T1 (dla układu TC1), mówimy o funkcji licznika. Stopień podziału preskalera może być ustawiony na jedną z czterech standardowych wartości, niezależnie dla każdego timera/licznika: CK/8, CK/64, CK/256, CK/1024. Liczniki/timery mogą być taktowane również sygnałem zegarowym bez podziału (CK/1).

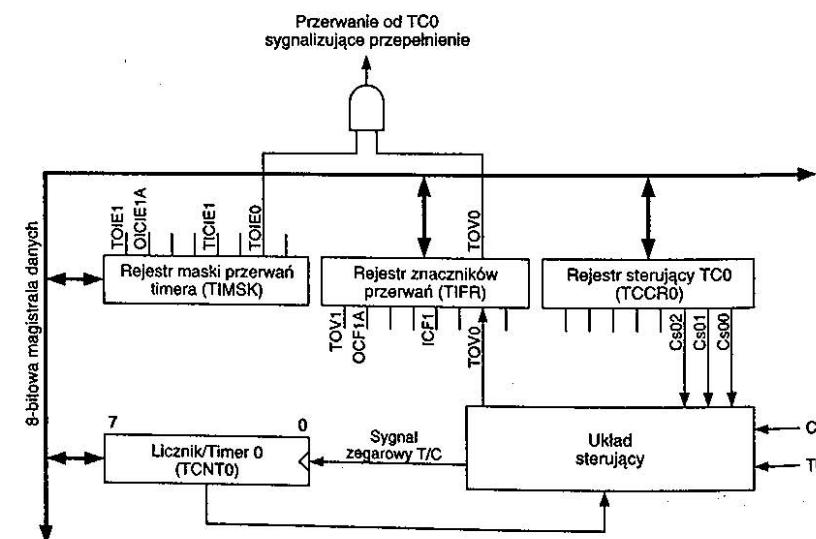


Rys. 5.1. Konfiguracja 10-bitowego preskalera służącego do wytwarzania sygnałów zegarowych dla Liczników/Timerów 0 i 1

5.1. 8-bitowy Timer/Licznik0

Schemat blokowy układu TC0 przedstawiono na **rysunku 5.2**. Jest to 8-bitowy timer/licznik, który może być taktowany bezpośrednio sygnałem CK (z oscylatora wewnętrznego), sygnałem CK' podzielonym wstępnie przez preskaler oraz sygnałem z wejścia zewnętrznego T0. Układ TC0 może być zatrzymany w dowolnym momencie poprzez wyzerowanie bitów sterujących CS02, CS01, CS00 znajdujących się w rejestrze sterującym tego timera/licznika – TCCR0.

Jeśli układ TC0 jest skonfigurowany jako licznik, to zewnętrzny przebieg zegarowy nie taktuje go bezpośrednio, lecz jest synchronizowany sygnałem wewnętrznego oscylatora. W tym trybie sygnał zegarowy T0 jest próbkowany podczas narastającego zbocza wewnętrznego sygnału zegarowego. Odpowiednie warunki próbkowania będą spełnione tylko wtedy, gdy czas pomiędzy kolejnymi zboczami zewnętrznego sygnału zegarowego będzie dłuższy niż okres wewnętrznego przebiegu taktującego. Maksymalna mierzona częstotliwość sygnału na wejściu T0 może być więc równa $f_{XTAL}/2$. Układ TC0 dla małych stopni podziału preskalera cechuje się dużą rozdzielczością i dokładnością. Duże stopnie podziału przydają się natomiast do generowania długich impulsów lub obsługi wolnozmiennych zdarzeń. W przypadkach,



Rys. 5.2. Schemat blokowy Timera/Licznika0 (TC0)

w których jest wymagany precyzyjny pomiar długich czasów, niezbędne staje się wprowadzenie dodatkowych mechanizmów programowych. Najczęściej będzie to dodatkowa zmienna programu zliczająca przerwania od timera/licznika. Zliczanie kończy się dopiero po osiągnięciu wymaganej liczby wejść do procedury obsługującej przerwanie.

Timer/Licznik0 jest konfigurowany poprzez rejestr sterujący TCCR0. Flaga przepełnienia dla tego układu (TOV0) znajduje się w rejestrze TIFR (*Timer/Counter Interrupt Flag Register*).

TCCR0 (Timer/Counter0 Control Register) – rejestr sterujący Timera/Licznika0 – \$33

Bit	7	6	5	4	3	2	1	0	
\$33 (\$53)	–	–	–	–	–	CS02	CS01	CS00	TCCR0
Odczyt/Zapis	R	R	R	R	R	R/W	R/W	R/W	
Wartość początkowa	0	0	0	0	0	0	0	0	
R – oznacza odczyt, R/W – oznacza odczyt/zapis									

B7...3 – zarezerwowane.

Te bity są zarezerwowane w układzie AT90S2313 i zawsze odczytywane jako zero.

B2...B0 – CS02, CS01, CS00 (*Clock Select0*, bity: 2, 1, 0): bity wyboru stopnia podziału preskalera oraz źródła sygnału taktującego i jego zbocza aktywnego dla licznika TC0.

Wszystkie kombinacje bitów wyboru dla Timera/Licznika0 przedstawiono w tablicy 5.1.

Występujący w tablicy 5.1 sygnał CK pochodzi z wewnętrznego generatora i jest to ten sam sygnał, który taktuje CPU. Skonfigurowanie układu TC0 jako licznik powoduje – jak już wiadomo – zliczanie impulsów z wejścia T0. Dzieje się tak nawet wtedy, gdy wyprowadzenie mikrokontrolera PD4/T0 jest ustawione jako wyjście. Rozwiązanie takie umożliwia programową kontrolę

Tab. 5.1. Konfiguracja układu Timera/Licznika0

CS02	CS01	CS00	Opis
0	0	0	Stop – układ TC0 jest zatrzymany
0	0	1	Sygnał taktujący CK
0	1	0	Sygnał taktujący CK/8
0	1	1	Sygnał taktujący CK/64
1	0	0	Sygnał taktujący CK/256
1	0	1	Sygnał taktujący CK/1024
1	1	0	Sygnał taktujący: zewnętrzny sygnał T0, zbocze opadające
1	1	1	Sygnał taktujący: zewnętrzny sygnał T0, zbocze narastające

zliczania. Na skutek przepełnienia Timera/Licznika0 może być wygenerowane przerwanie, którego procedura obsługi jest umieszczona w pamięci programu pod adresem \$006.

TCNT0 (Timer/Counter0) – rejestr Timera/Licznika0 – \$32

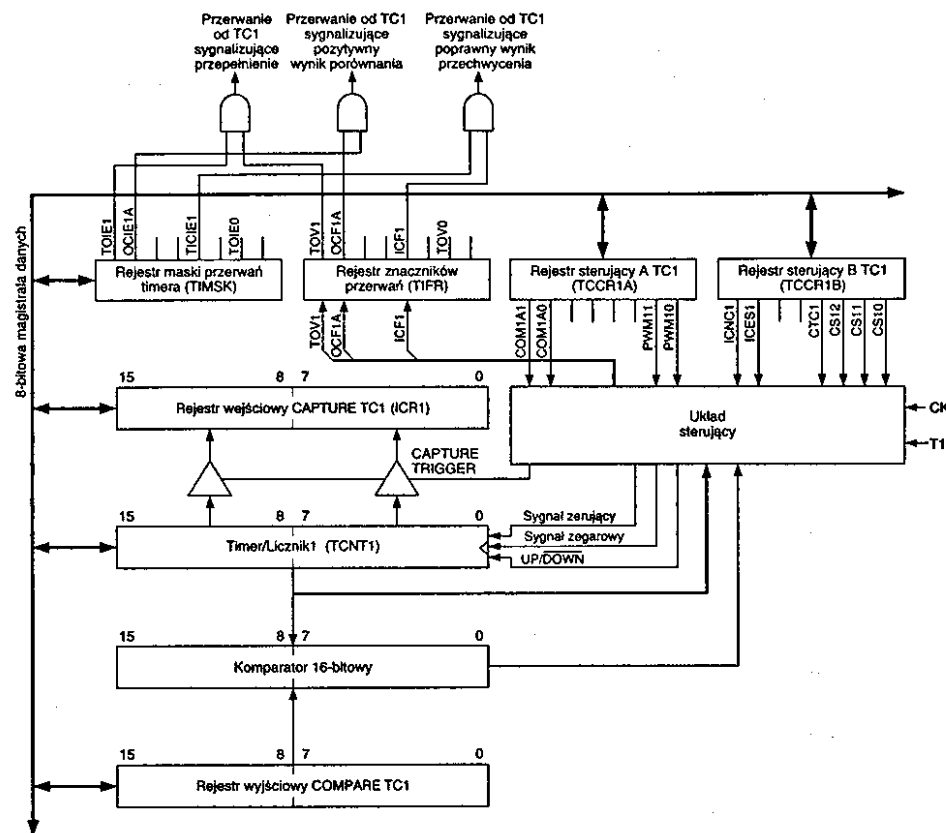
Bit	7	6	5	4	3	2	1	0	
\$32 (\$52)	MSB							LSB	TCNT0
Odczyt/Zapis	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Wartość początkowa	0	0	0	0	0	0	0	0	
R/W – oznacza odczyt/zapis									

Timer/Licznik0 zaprojektowano jako układ zliczający w górę. Rejestr TCNT0 przechowujący aktualny stan zliczania jest dostępny zarówno do zapisu jak i odczytu. Jeśli zapis do tego rejestru nastąpi jednocześnie z impulsem zegarowym, najpierw zostanie zwiększony stan licznika, następnie zostanie dokonany wpis nowej wartości do rejestru. Osiągnięcie stanu 0 jest traktowane jako przepełnienie. W tym momencie jest ustawiana flaga przepełnienia TOV0 sygnalizująca żądanie obsługi przerwania. To czy zostanie ono zauważone przez CPU zależy od ustawienia bitu TOIE0 w rejestrze TIMSK (*Timer/Counter Interrupt Mask Register*) i bitu globalnego zezwolenia na przerwanie I w rejestrze SREG.

5.2. 16-bitowy Timer/Licznik1

Drugi timer/licznik wbudowany w mikrokontroler AT90S2313 jest znacznie bardziej rozbudowany niż omawiany poprzednio, może w związku z tym spełniać dodatkowe funkcje. Schemat blokowy Timera/Licznika1 przedstawiono na rysunku 5.3. Jest to 16-bitowy timer/licznik, który może być taktowany bezpośrednio sygnałem CK (z oscylatora wewnętrznego), sygnałem CK podzielonym wstępnie przez preskaler lub sygnałem z wejścia zewnętrznego T1. Układ TC1 może być zatrzymany w dowolnym momencie poprzez wyzerowanie bitów sterujących CS12, CS11, CS10 znajdujących się w rejestrze sterującym tego timera/licznika – TCCR1B (*Timer/Counter1 Control Register B*).

Jeśli układ TC1 jest skonfigurowany jako licznik, zewnętrzny przebieg zegarowy nie steruje nim bezpośrednio, lecz jest synchronizowany sygnałem wewnętrznego oscylatora. W tym trybie sygnał zegarowy T1 jest próbkowany na narastającym zboczu zegara wewnętrznego. Odpowiednie warunki próbkowania będą spełnione tylko wtedy, gdy czas pomiędzy kolejnymi zbocza-



Rys. 5.3. Schemat blokowy Timera/Licznika1 (TC1)

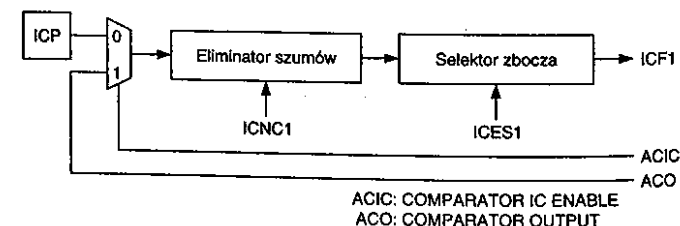
mi zewnętrznego sygnału zegarowego będzie dłuższy niż okres wewnętrzne-go oscylatora. Maksymalna mierzona częstotliwość sygnału na wejściu T1 może być więc równa $f_{XTAL}/2$. Układ TC1 dla małych stopni podziału pre-skalera cechuje się dużą rozdzielczością i dokładnością, duże stopnie podziału przydają się natomiast do generowania długich impulsów lub obsługi wolnozmennych zdarzeń. W przypadkach, w których jest wymagany precyzyjny pomiar długich czasów, niezbędne staje się wprowadzenie dodatkowych mechanizmów programowych. Najczęściej będzie to dodatkowa zmienna programu zliczająca przerwy timera/licznika. Właściwy proces zliczania kończy się dopiero po osiągnięciu wymaganej liczby wejść do procedury obsługującej przerwanie.



Jedną z interesujących funkcji Licznika/Timera1 jest możliwość przechwytywania (*capture*). Jeśli TC1 pracuje w tym trybie, to podanie impulsu wyzwalającego na wejście ICP powoduje przepisanie aktualnego stanu licznika TC1 do rejestrów [ICR1H][ICR1L] (czyli [ICR1H][ICR1L]=[TCNT1H][TCNT1L]). Tak więc, po wyzwoleniu licznik zlicza w ustalonym cyklu, a jego stan w chwili wyzwolenia został zatrzaśnięty w 16-bitowym rejestrze ICR1. Impuls wyzwalania może być pobierany także z wyjścia komparatora analogowego.

Układ TC1 oprócz typowych trybów pracy jako zwykły timer lub licznik, umożliwia dodatkowo realizację funkcji porównywania wyjścia (*Output Compare*), modulacji PWM (*Pulse Width Modulation* – modulacja szerokości impulsu) i przechwytywania wejścia (*Input Capture*). Funkcja *Output Compare* wykorzystuje rejestry OCR1AH i OCR1AL (*Output Compare Register 1A*) jako źródło danej porównywanej ze stanem Timera/Licznika1. W przypadku wystąpienia równości rejestrów: OCR1AH=TCNT1H i OCR1AL=TCNT1L może nastąpić opcjonalne zerowanie Timera/Licznika1 oraz akcja na wyjściu (OC1) *Output Compare1*. Sposób reakcji zależy od ustawień bitów COM1A1 i COM1A0 w rejestrze TCCR1A (*Timer/Counter1 Control Register A*). Szczegółowy opis zamieszczono w dalszej części rozdziału.

Rejestry OCR1AH i OCR1AL są również wykorzystywane podczas pracy Timera/Licznika1 jako 8-, 9- lub 10-bitowy modulator PWM. Funkcja przechwytywania jest związana z rejestrem ICR1 (*Input Capture Register*). Jest to *de facto* rejestr 16-bitowy składający się z ICR1H (starszy bajt) i ICR1L (młodszy bajt). Przechwytywanie jest wyzwalane zewnętrznym sygnałem pojawiającym się na wyprowadzeniu ICP (*Input Capture Pin*) mikrokontrolera. Funkcję przechwytywania konfiguruje się poprzez ustawienia rejestru TCCR1B (*Timer/Counter1 Control Register*). Do wyzwalania przechwytywania może być wykorzystany ponadto komparator analogowy zawarty



Rys. 5.4. Schemat układu obrabiającego sygnał z wejścia ICP

w strukturze mikrokontrolera. Na rysunku 5.4 przedstawiono budowę układu obrabiającego sygnał z wyprowadzenia ICP i wyprowadzeń związanych z komparatorem analogowym.

Z funkcją przechwytywania związany jest ponadto układ eliminacji szumów (*Noise Canceler*). Jeśli jest włączony, to warunek wyzwolenia dla funkcji przechwytywania zostanie przyjęty dopiero po wykryciu czterech kolejnych próbek sygnału wyzwalającego o jednakowej wartości.

Timer/Licznik1 jest konfigurowany poprzez rejestry sterujące TCCR1A i TCCR1B. Flagi zdarzeń związanych z Timerem/Licznikiem1 znajdują się w rejestrze TIFR (*Timer/Counter Interrupt Flag Register*). Są to: flaga przepełnienia TOV1 (*Timer/Counter1 Overflow Flag*), flaga porównania OCF1A (*Output Compare Flag*) i flaga przechwycenia ICF1 (*Input Capture Flag*).

TCCR1A (*Timer/Counter1 Control Register A*) – rejestr sterujący A Timera/Licznika1 – \$2F

Bit	7	6	5	4	3	2	1	0	
\$2F (\$4F)	COM1A1	COM1A0	–	–	–	–	PWM11	PWM10	TCCR1A
Odczyt/Zapis	R/W	R/W	R	R	R	R	R/W	R/W	
Wartość początkowa	0	0	0	0	0	0	0	0	
R – oznacza odczyt, R/W – oznacza odczyt/zapis									

B7, B6 – COM1A1, COM1A0 (*Compare Output Mode1*, bity: 1,0): bity konfiguruje Timer/Licznik1 w trybie porównania.

Bity COM1A1 i COM1A0 określają zachowanie się wyprowadzenia mikrokontrolera OC1 (*Output Compare 1* – alternatywna funkcja wyprowadzenia PB3), następującego po wykryciu pozytywnego porównania w układzie Timera/Licznika1. Wyjście to może w takich przypadkach zachowywać się zgodnie z opisem z tablicy 5.2. W przypadku wykorzystywania funkcji porównania wyprowadzenie OC1/PB3 powinno być skonfigurowane jako wyjście.

Tab. 5.2 Konfiguracja Timera/Licznika1 w trybie porównania⁽¹⁾⁽²⁾

COM1A1	COM1A0	Opis
0	0	Wyjście OC1 jest odłączone od układu Timera/Licznika1
0	1	Zmiana stanu na wyjściu OC1 w wyniku pozytywnego porównania
1	0	Wyzierowanie („0”) wyjścia OC1 w wyniku pozytywnego porównania
1	1	Ustawienie („1”) wyjścia OC1 w wyniku pozytywnego porównania

Uwagi:

1. W trybie PWM powyższe bity zmieniają znaczenie (patrz tablica 5.6).
2. Początkowy stan wyjścia OC1 jest nieokreślony.

Tab. 5.3. Konfiguracja modulatora PWM

PWM11	PWM10	Opis
0	0	Funkcja PWM dla Timera/Licznika1 zablokowana
0	1	Timer/Licznik1 pracuje jako 8-bitowy PWM
1	0	Timer/Licznik1 pracuje jako 9-bitowy PWM
1	1	Timer/Licznik1 pracuje jako 10-bitowy PWM

B5...2 – zarezerwowane.

Te bity są zarezerwowane w układzie AT90S2313 i zawsze odczytywane jako zero.

B1, B0 – PWM11, PWM10 (*Pulse Width Modulator Select Bits*): bity wyboru trybu pracy modulatora PWM.

Bity te ustalają tryb pracy modulatora PWM wykorzystującego Timer/Licznik1 zgodnie z tablicą 5.3.

TCCR1B (*Timer/Counter1 Control Register B*) – rejestr sterujący B Timera/Licznika1 – \$2E

Bit	7	6	5	4	3	2	1	0	
\$2E (\$4E)	ICNC1	ICES1	–	–	CTC1	CS12	CS11	CS10	TCCR1B
Odczyt/Zapis	R/W	R/W	R	R	R/W	R/W	R/W	R/W	
Wartość początkowa	0	0	0	0	0	0	0	0	
R – oznacza odczyt, R/W – oznacza odczyt/zapis									

B7 – ICNC1 (*Input Capture 1 Noise Canceler*): bit włączający/wyłączający układ eliminacji szumu dla Timera/Licznika1 pracującego w trybie porównania.

Jeśli bit ICNC1 jest wyzerowany, układ eliminacji szumu jest wyłączony. Przechwytywanie jest wówczas wyzwolone na pierwszym narastającym lub opadającym zboczu (w zależności od konfiguracji) spróbkowanym na wejściu ICP przez CPU. Gdy bit ICNC1 jest ustawiony (wpisano „1”), wyzwolenie przechwycenia nastąpi dopiero po wykryciu czterech kolejnych, jednakowych próbek (o wartości określonej konfiguracją) na wejściu ICP. Konfigurację wyzwalania określa bit ICES1 rejestru TCCR1B. Częstotliwość próbkowania jest równa f_{XTAL} . Funkcja eliminacji szumu przydaje się, gdy wejście wyzwalające ICP współpracuje np. z wyjściem komparatora analogowego. Na skutek niestabilności źródeł referencyjnych i samego sygnału mierzonego w stanach równowagi na wejściach komparatora może dochodzić do wielokrotnych przerzutów na wyjściu komparatora. Sytuacja taka mogłaby zakłócić pracę Timera/Licznika1 z włączoną funkcją porównania.

B6 – ICES1 (Input Capture1 Edge Select): bit określający zbocze sygnału wyzwalającego przechwytywanie.

Jeśli bit ICES1 jest wyzerowany („0”), to zawartość rejestrów Timera/Licznika1 jest przepisywana do rejestru ICR1 (Input Capture Register) na opadającym zboczu wejściowego sygnału ICP. Jeśli bit ICES1 jest ustawiony („1”), to zawartość rejestrów Timera/Licznika1 jest przepisywana do rejestru ICR1 (Input Capture Register) na narastającym zboczu wejściowego sygnału ICP.

B5, B4 – zarezerwowane.

Te bity są zarezerwowane w układzie AT90S2313 i zawsze odczytywane jako zero.

B3 – CTC1 (Clear Timer/Counter1 on Compare Match): bit zerowania Timera/Licznika1 po spełnieniu warunku porównania.

Jeśli bit CTC1 jest ustawiony („1”), to Timer/Licznika1 jest zerowany (TCNT1H=\$00 i TCNT1L=\$00) w najbliższym cyklu zegarowym po spełnieniu warunku porównania. Warunek porównania następuje, gdy TCNT1H=OCR1AH i TCNT1L=OCR1AL. Jeśli bit CTC1 jest wyzerowany („0”), to wystąpienie warunku porównania nie wpływa na stan Timera/Licznika1. Kontynuuje on zliczanie. Warunek porównania jest wykrywany przez CPU w najbliższym cyklu zegarowym po jego wystąpieniu. Dzieje się tak dla wartości podziału preskalera równej 1. Dla większych wartości stopnia podziału funkcja porównania będzie działała inaczej. Najlepiej zilustruje to poniższy przykład.

Jeśli stopień podziału preskalera jest równy 1, do rejestru porównania wpisano wartość C, a bit CTC1 jest ustawiony, to Timer/Licznika1 będzie liczył w cyklu:

....IC-2|C-1|C|0|1|....

Gdy stopień podziału preskalera będzie ustawiony np. na 8, Timer/Licznika1 będzie liczył w cyklu:

....|C-2, C-2, C-2, C-2, C-2, C-2, C-2|C-1, C-1, C-1, C-1, C-1, C-1, C-1, C-1|C, 0, 0, 0, 0, 0, 0, 0|....

Bit CTC1 w trybie PWM nie ma znaczenia.



Warunek porównania jest wykrywany przez CPU w najbliższym cyklu zegarowym po jego wystąpieniu. Dzieje się tak dla wartości podziału preskalera równej 1. Dla większych wartości stopnia podziału funkcja porównania będzie działała inaczej.

Tab. 5.4. Konfiguracja układu Timera/Licznika1

CS12	CS11	CS10	Opis
0	0	0	Stop – układ TC1 jest zatrzymany
0	0	1	Sygnał taktujący CK
0	1	0	Sygnał taktujący CK/8
0	1	1	Sygnał taktujący CK/64
1	0	0	Sygnał taktujący CK/256
1	0	1	Sygnał taktujący CK/1024
1	1	0	Sygnał taktujący: zewnętrzny sygnał T1, zbocze opadające
1	1	1	Sygnał taktujący: zewnętrzny sygnał T1, zbocze narastające

B2...B0 – CS12, CS11, CS10 (Clock Select1, bity: 2, 1, 0): bity wyboru stopnia podziału preskalera oraz źródła sygnału taktującego i jego zbocza aktywnego dla licznika TC1.

Wszystkie możliwe kombinacje bitów wyboru dla Timera/Licznika1 przedstawiono w tabelicy 5.4.

Występujący w tabelicy 5.4 sygnał CK pochodzi z wewnętrznego oscylatora i jest to ten sam sygnał, który taktuje CPU. Skonfigurowanie układu TC1 jako licznik powoduje – jak już wiadomo – zliczanie impulsów z wejścia T1. Dzieje się tak nawet wtedy, gdy wyprowadzenie mikrokontrolera PD5/T1 jest ustawione jako wyjście. Rozwiązanie takie umożliwia programową kontrolę zliczania. Na skutek przepełnienia Timera/Licznika1 może być wygenerowane przerwanie, którego procedura obsługi jest umieszczona w pamięci programu pod adresem \$005.

TCNT1H i TCNT1L (Timer/Counter1) – rejestr Timera/Licznika1 – \$2D/\$2C

Bit	15	14	13	12	11	10	9	8	
\$2D (\$4D)	MSB								TCNT1H TCNT1L
\$2C (\$4C)								LSB	
	7	6	5	4	3	2	1	0	
Odczyt/Zapis	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Wartość początkowa	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

R/W – oznacza odczyt/zapis

16-bitowy rejestr TCNT1 przechowuje aktualny stan Timera/Licznika1 – TCNT1H starszy bajt, TCNT1L młodszy bajt. Jest dostępny zarówno do zapisu jak i odczytu, jednak w pewnej specyficznej sytuacji mogłoby dochodzić do niejednoznaczności operacji odczytu. Przypadek taki mógłby wystąpić, gdyby moment odczytu nastąpił, gdy rejestr przechowywał wartość \$FFFF i stopień podziału preskalera był równy 1. Po odczytaniu np.

młodsze bajtu z TCNT1L nastąpiłoby zwiększenie stanu rejestru, w wyniku czego odczytana następnie wartość TCNT1H byłaby równa \$00, zamiast spodziewanej \$FF. Z problemem tym muszą sobie radzić specjalnymi zabiegami programowymi użytkownicy mikrokontrolerów rodziny '51, w AVR-ach problem został rozwiązany przez wprowadzenie dodatkowego 8-bitowego rejestru tymczasowego TEMP (nie jest on dostępny dla użytkownika bezpośrednio), wspomagającego operacje dostępu do rejestru TCNT1. Zapewnia on jednoczesny dostęp do rejestrów TCNT1H i TCNT1L i jest również wykorzystywany przy dostępie do rejestrów OCR1A i ICR1. Gdy CPU dokonuje zapisu do bardziej znaczącego rejestru TCNT1H, zapisywana dana jest umieszczana początkowo w rejestrze tymczasowym TEMP. Następnie, w tym samym momencie, w którym odbywa się zapis mniej znaczącego rejestru TCNT1L, do rejestru TCNT1H jest przepisywany rejestr TEMP. Konsekwencją takiego rozwiązania jest konieczność wpisywania najpierw starszej (TCNT1H), później młodszej (TCNT1L) części rejestru podczas operacji 16-bitowego zapisu. Analogicznie wygląda operacja odczytu rejestru TCNT1. Gdy CPU dokonuje odczytu mniej znaczącego rejestru TCNT1L, pobierana dana jest przesyłana bezpośrednio do CPU i jednocześnie zawartość bardziej znaczącego rejestru (TCNT1H) jest umieszczana w rejestrze tymczasowym TEMP. Kiedy teraz CPU odczytuje rejestr TCNT1H, dana jest pobierana z rejestru TEMP. Konsekwencją takiego rozwiązania jest konieczność odczytywania najpierw młodszej (TCNT1L), później starszej (TCNT1H) części rejestru podczas operacji 16-bitowego odczytu.

Gdy zapis do rejestru TCNT1 następuje w chwili wystąpienia impulsu zegarowego, najpierw będzie wykonane zliczenie, później zaś rejestr zostanie ustawiony zgodnie z zapisaną daną.



Jeśli program główny i procedury obsługi przerwań wykorzystują operacje z udziałem rejestru TEMP, to przerwania powinny być zablokowane na czas dostępu do tego rejestru.

OCR1AH i OCR1AL (Timer/Counter1 Output Compare Register A) – starszy i młodszy rejestr wartości porównywanej – \$2B/\$2A

Bit	15	14	13	12	11	10	9	8	
\$2B (\$4B)	MSB								OCR1AH OCR1AL
\$2A (\$4A)								LSB	
	7	6	5	4	3	2	1	0	
Odczyt/Zapis	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Wartość początkowa	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

R/W – oznacza odczyt/zapis

Wartość porównywana przez układ TC1 jest przechowywana w 16-bitowym rejestrze OCR1A składającym się z rejestru OCR1AH (starszy bajt) i OCR1AL (młodszy bajt). Wartość zapisana w nim jest bezustannie porównywana z wartością Timera/Licznika1. W przypadku wystąpienia warunku równości obu rejestrów, tzn. gdy OCR1AH=TCNT1H i OCR1AL=TCNT1L, wykonywana jest akcja określona w rejestrze sterującym Timera/Licznika1 (TCCR1A) i rejestrze statusu (SREG). Aby zapewnić jednoczesność odczytu dwóch 8-bitowych rejestrów składających się na rejestr 16-bitowy, zastosowano podobny mechanizm jak w przypadku rejestru TCNT1. W operacjach dostępu do OCR1A korzysta się z rejestru tymczasowego TEMP. Gdy CPU dokonuje zapisu do bardziej znaczącego rejestru OCR1AH, zapisywana dana jest umieszczana początkowo w rejestrze tymczasowym TEMP. Następnie, w tym samym momencie, w którym odbywa się zapis mniej znaczącego rejestru OCR1AL, do rejestru OCR1AH przepisywany jest rejestr TEMP. Konsekwencją takiego rozwiązania jest konieczność wpisywania najpierw starszej (OCR1AH), później młodszej (OCR1AL) części rejestru podczas operacji 16-bitowego zapisu. Analogicznie wygląda operacja odczytu rejestru OCR1A. Gdy CPU dokonuje odczytu mniej znaczącego rejestru OCR1AL, pobierana dana jest przesyłana bezpośrednio do CPU i jednocześnie zawartość bardziej znaczącego rejestru (OCR1AH) jest umieszczana w rejestrze tymczasowym TEMP. Kiedy teraz CPU odczytuje rejestr OCR1AH, dana jest pobierana z rejestru TEMP. Konsekwencją takiego rozwiązania jest konieczność odczytywania najpierw młodszej (OCR1AL), później starszej (OCR1AH) części rejestru podczas operacji 16-bitowego odczytu. Jeśli program główny i procedury obsługi przerwań wykorzystują operacje z udziałem rejestru TEMP, przerwania powinny być zablokowane na czas dostępu do tego rejestru.

ICR1H i ICR1L (Timer/Counter1 Input Capture Register) – starszy i młodszy rejestr przechwytywania – \$25/\$24

Bit	15	14	13	12	11	10	9	8	
\$25 (\$45)	MSB								ICR1H ICR1L
\$24 (\$44)								LSB	
	7	6	5	4	3	2	1	0	
Odczyt/Zapis	R	R	R	R	R	R	R	R	
	R	R	R	R	R	R	R	R	
Wartość początkowa	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

R – oznacza odczyt

16-bitowy rejestr przechwytywania jest przeznaczony tylko do odczytu. W chwili, w której na wejściu ICP zostanie wykryte narastające lub opadające zbocze sygnału (w zależności od ustawień bitu ICES1 w rejestrze TCC1B), aktualna wartość Timera/Licznika1 jest przepisywana do rejestru ICR1A (Input Capture Register1). W tym samym momencie flaga ICF1 (Input Capture Flag) jest ustawiana. I w tym przypadku, aby zapewnić jednocześnie odczytu dwóch 8-bitowych rejestrów składających się na rejestr 16-bitowy ICR1A, zastosowano rejestr tymczasowy TEMP. Gdy CPU dokonuje odczytu mniej znaczącego rejestru ICR1L, pobierana dana jest przesyłana bezpośrednio do CPU i jednocześnie zawartość bardziej znaczącego rejestru (ICR1AH) jest umieszczana w rejestrze tymczasowym TEMP. Kiedy teraz CPU odczytuje rejestr OCR1AH, dana jest pobierana z rejestru TEMP. Konsekwencją takiego rozwiązania jest konieczność odczytywania najpierw młodszej (ICR1AL), później starszej (ICR1AH) części rejestru podczas operacji 16-bitowego odczytu. Jeśli program główny i procedury obsługi przerwań wykorzystują operacje z udziałem rejestru TEMP, to przerwania powinny być zablokowane na czas dostępu do tego rejestru.

5.3. Timer/Licznik1 w trybie PWM

Do realizacji 8-, 9- lub 10-bitowego modulatora PWM (Pulse Width Modulation) wykorzystywany jest Timer/Licznik1 oraz rejestr OCR1A (Output Compare Register 1). Wyjściem modulatora jest wyprowadzenie OC1/PB3 mikrokontrolera. Modulator jest układem samodzielnym, co oznacza, że po załadowaniu parametrów generowanego sygnału, dalsza praca odbywa się bez ingerencji programu użytkowego. Parametr odpowiadający współczynnikowi wypełnienia może być oczywiście w dowolnym momencie zmieniany bez generowania zakłóceń typu *glitch* (niepożądane impulsy). Zmiany

Tab. 5.5. Wartość zliczania licznika TC1 w trybie PWM (TOP) i częstotliwość generowanego przebiegu

Rozdzielczość PWM	Wartość zliczania (TOP)	Częstotliwość generowanego sygnału PWM
8 bitów	\$00FF (255)	$f_{TC1}/510$
9 bitów	\$01FF (511)	$f_{TC1}/1022$
10 bitów	\$03FF (1023)	$f_{TC1}/2046$

Tab. 5.6. Znaczenie bitów COM1A1 i COM1A0 rejestru TCCR1A w trybie PWM(1)

COM1A1	COM1A0	Akcja podejmowana na wyjściu OC1
0	0	Nie występuje
0	1	Nie występuje
1	0	Wyjście OC1 jest zerowane („0”) po osiągnięciu warunku równości podczas zliczania w górę i ustawiane („1”) po osiągnięciu warunku równości podczas zliczania w dół (normalny tryb PWM)
1	1	Wyjście OC1 jest zerowane („0”) po osiągnięciu warunku równości podczas zliczania w dół i ustawiane („1”) po osiągnięciu warunku równości podczas zliczania w górę (odwrócony tryb PWM)

Uwaga:

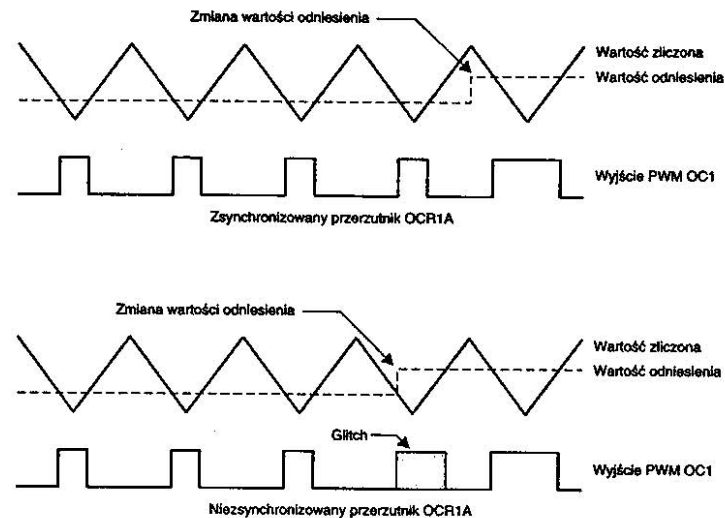
1. Stan początkowy wyjścia OC1 jest nieokreślony.

współczynnika wypełnienia przebiegają bez zakłócenia fazy generowanego sygnału.

W trybie PWM Timer/Licznik1 pracuje jako licznik rewersyjny zliczający od \$0000 do wartości maksymalnej (TOP – patrz **tablica 5.5**), po czym kierunek zliczania zostaje zmieniony i licznik odlicza z powrotem do zera. W tym momencie następuje ponowna zmiana kierunku liczenia i cykl powtarza się. Gdy licznik osiągnie stan, w którym 8, 9 lub 10 najmniej znaczących bitów będzie miało taką samą wartość jak odpowiadające im bity w rejestrze OCR1A, wyprowadzenie OC1/PB3 jest ustawiane lub zerowane, zgodnie z ustawieniami bitów COM1A1 i COM1A0 znajdujących się w rejestrze TCCR1A. Szczegóły podano w **tablicy 5.6**.

Zapis wartości porównywania do rejestru OCR1A w trybie PWM przebiega pośrednio – poprzez rejestr TEMP. Operacja ta jest wykonywana na 10 najmłodszych bitach rejestru OCR1A, które są zatraskiwane, kiedy Timer/Licznik1 osiągnie stan TOP. Zapobiega to powstawaniu szkodliwych impulsów (*glitches*) w przypadku niesynchronicznego zapisu rejestru OCR1A. Sytuację tę zilustrowano na **rysunku 5.5**.

W przedziale czasu pomiędzy zapisem i zatrzaśnięciem danej, odczyt z rejestru OCR1A powoduje pobieranie zawartości rejestru TEMP. W ten sposób dane zapisywane ostatnio do rejestru OCR1A są zawsze czytane spoza niego. Jeśli OCR1A zawiera wartość \$0000 lub TOP, to wyjście OC1 jest aktualizowane do stanu niskiego lub wysokiego (w zależności od ustawień bitów



Rys. 5.5. Przyczyna powstawania impulsów glitches na wyjściu generatora PWM w mikrokontrolerach AVR

COM1A1 i COM1A0 w rejestrze TCCR1A) w chwili spełnienia następnego warunku porównania. Zilustrowano to w tablicy 5.7.

UWAGA

Jeśli rejestr porównania (OCR1) zawiera wartość TOP i preskaler nie jest używany (CS12...CS10=001), na wyjściu PWM nie jest generowany żaden przebieg. Dzieje się tak, gdyż wartości porównania podczas zliczania w górę i w dół są osiągane jednocześnie. Zastosowanie preskalera (CS12...CS10 = 001 lub 000) powoduje uaktywnienie wyjścia PWM kiedy licznik osiągnie wartość TOP, ale podczas odliczania w dół warunek porównania nie jest interpretowany. W tym przypadku zostanie wygenerowany jedynie pojedynczy impuls.

Tab. 5.7. Zachowanie się wyjścia OC1 w trybie PWM, gdy rejestr porównywania jest równy \$0000 lub TOP

COM1A1	COM1A0	OCR1A	Wyjście OC1
1	0	\$0000	L
1	0	TOP	H
1	1	\$0000	H
1	1	TOP	L

Flaga przepełnienia Timera/Licznika – TOV1 w trybie PWM jest ustawiana, gdy licznik znajdzie się w stanie \$0000 podczas zliczania w górę. Nie dotyczy to jednak pierwszego cyklu po jego włączeniu.

Przykład 5.1. Zachowanie się flagi TOV1 i OCF1A w trybie PWM

;W pętli LOOP są wykonywane kolejne rozkazy programu (wykonywane ;w jednym cyklu), w tym przypadku nie jest istotne jakie, gdyż ;interesują nas tylko stany licznika i flag ;definicja rejestru tmp

```
def tmp=r17 ;definicja rejestru tmp
cseg org $0000

rjmp RESET

RESET: ... ;inicjowanie zmiennych
;programu ...
;w tym SPL
;globalne włączenie przerwań

sei
ldi tmp,0x81
out tccr1a,tmp ;8-bitowy PWM normalny
ldi tmp,0x32
ldi tmp,0
out ocr1ah,tmp
out ocr1al,tmp ;OCR1AL=$32, przykładowa wartość
ldi tmp,0x01
out tccr1b,tmp ;preskaler=1, start timera
;TCNT1=0, TOV1=0, OCF1A=0
;kolejne rozkazy (nieistotne)

LOOP: ....
....
....
.... ;TCNT1=$32
.... ;TCNT1=$33, OCF1A=1
.... ;(wystąpił pozytywny wynik porównania)
....
.... ;TCNT1=$FE
.... ;TCNT1=$FF
.... ;TCNT1=$FE
.... ;TCNT1=$FD
....
.... ;TCNT1=$01
.... ;TCNT1=$00, TOV1=1 (przepełnienie)
.... ;TCNT1=$01
```

Przerwania od przepełnienia licznika mogą w trybie PWM funkcjonować normalnie, muszą być oczywiście ustawione odpowiednie bity sterujące: TOIE1 w rejestrze TIMSK i I w rejestrze SREG. Analogicznie można korzystać z przerwania od spełnienia warunku porównania, gdy tylko będzie ustawiony bit OCIE1A w rejestrze TIMSK i I w rejestrze SREG.

6. Watchdog

Stale rosnące wymagania niezawodnościowe stawiane przez użytkowników sprzętu elektronicznego – i to nie tylko profesjonalnego – powodują, że konstruktorzy projektując urządzenia muszą wykazać dużą staranność w tym zakresie. Niezawodność sprzętu zależy od wielu czynników, np. odporności na zakłócenia elektryczne czy poprawności oprogramowania mikrokontrolera. Na skutek przypadkowego zakłócenia wykonywanie programu może zostać przeniesione w przypadkowe miejsce w pamięci programu, powodując nieprzewidywalne działanie systemu. Niedostateczne przetestowanie programu może również doprowadzić np. do powstawania martwych pętli, powodując w efekcie zawieszenie się systemu. Często sytuacje takie są trudne do wykrycia, gdyż na pierwszy rzut oka program wygląda na pozbawiony błędów. Na przykład poniższa pętla napisana w języku C wydaje się być poprawna i po sześciu iteracjach powinna zostać zakończona. Jednakże na skutek zaokrąglania wartości zmiennej sterującej typu *float*, warunek $x=0$ nigdy nie zostanie w niej osiągnięty i pętla będzie nieprzerwanie wykonywana.

Przykład 6.1. Z pozoru poprawna pętla programowa, okazuje się pętlą nieskończoną

```
#include <wdt.h>
#include <io4433.h>          //Uwaga! Z uwagi na zbyt duże biblioteki
                             //operacji zmiennoprzecinkowych, ten program
                             //nie zmieści się do układu AT90S2313

....
float x;                     //deklaracja zmiennej typu float
....
wdt_enable(2);               //
for(x=5./3.;x!=0.;x-=1./3.) //zmienna x jest zmniejszana od wartości
                             //5/3 do 0 co 1/3
{
    ....
    wdt_reset();             //umieszczenie rozkazu zerowania rejestrów
                             //watchdoga w tym miejscu jest błędem, nie
                             //spełni on tu swej funkcji. Mikrokontroler
                             //powinien być wyzerowany, jeśli będzie za
                             //długo przebywał w tej pętli
    ....
}
```

Aby zapobiec podobnym przypadkom opracowano dość prosty mechanizm sprzętowo-programowy. W wielu współczesnych mikrokontrolerach, m. in. w AT90S2313, zawarto w strukturze układu wydzielony timer-watchdog, którego zadaniem jest odmierzanie specjalnych interwałów czasowych (*time-out*), po przekroczeniu których mikrokontroler jest zerowany. Programista pisząc program nie może dopuścić do tego, aby przekroczyć czas *ti-*

me-out, gdyż spowoduje to wygenerowanie sygnału zerującego przez układ watchdoga. W tym celu w różnych punktach programu umieszcza rozkazy zerujące rejestry timera i rozpoczynające tym samym cykl liczenia od początku. Punkty takie muszą być starannie dobrane. Czas przejścia pomiędzy dwoma sąsiednimi (ze względu na kolejność przechodzenia, a nie miejsce w programie) nie może być dłuższy niż *time-out* watchdoga. Jednocześnie umieszczenie rozkazu zerującego watchdog w pętli programu podejrzanej o to, że może być pętlą nieskończoną (jak w **przykładzie 6.1**), nie spełni swojego zadania.

W mikrokontrolerze AT90S2313 watchdog jest timerem taktowanym z wydzielonego generatora, zawartego w strukturze układu i pracującego z częstotliwością 1 MHz. Jest to typowa wartość dla napięcia zasilającego $V_{CC} = 5\text{ V}$. Watchdog ma swój własny prescaler, dzięki czemu programista może ustawiać odpowiednią dla siebie wartość *time-outu* (tablica 6.1). Na liście rozkazów mikrokontrolera AT90S2313 znajduje się specjalny rozkaz – WDR (*Watchdog Reset*) – zerujący rejestr timera watchdoga. Długość cyklu odmierzanego przez watchdog może przybierać jedną z ośmiu różnych wartości. Doliczenie do końca powoduje wygenerowanie sygnału zerującego (rysunek 4.26) i skok pod adres wektora *Reset Vector*. Programista, który świadomie chce zablokować układ watchdoga musi wykonać specjalną sekwencję wyłączającą. Zabezpieczenie takie ma na celu uchronienie się od przypadkowego zablokowania watchdoga.

WDTCR (*Watchdog Timer Control Register*) – rejestr sterujący watchdoga – \$21

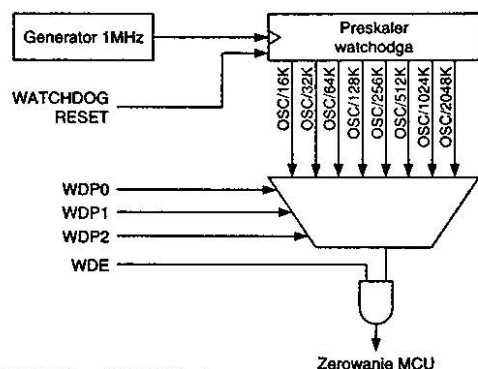
Bit	7	6	5	4	3	2	1	0	
\$21 (\$41)	–	–	–	WDTOE	WDE	WDP2	WDP1	WDP0	WDTCR
Odczyt/Zapis	R	R	R	R/W	R/W	R/W	R/W	R/W	
Wartość początkowa	0	0	0	0	0	0	0	0	
R – oznacza odczyt, R/W – oznacza odczyt/zapis									

B7...5 – Zarezerwowane.

Te bity są zarezerwowane w układzie AT90S2313 i zawsze odczytywane jako zero.

B4 – WDT0E (*Watchdog Turn-off Enable*): bit zezwolenia na wyłączenie watchdoga.

Bit ten musi być ustawiony („1”) przed wyzerowaniem bitu WDE. Jeśli się tego nie robi, zablokowanie watchdoga nie zostanie wykonane. Po ustawieniu bitu WDT0E zostaje on sprzętowo wyzerowany automatycznie po 4 cyklach zegarowych. Wyłączenie watchdoga musi więc nastąpić w tym czasie.



Rys. 6.1. Schemat blokowy watchdoga

B3 – WDE (Watchdog Enable) – bit włączający/wyłączający układ watchdoga.

Watchdog jest włączony, jeśli bit WDE jest ustawiony („1”). Jeśli WDE jest wyzerowany („0”), to watchdog jest wyłączony. WDE może być zerowany tylko wtedy, gdy jednocześnie WDTOE jest ustawiony („1”). W celu zablokowania pracującego watchdoga musi być wykonana poniższa procedura:

1. Zapisać jednocześnie „1” do bitów WDTOE i WDE. Logiczna „1” musi być zapisana do WDE podczas procedury blokowania watchdoga nawet wtedy, gdy bit ten był ustawiany wcześniej.
2. W ciągu następnych czterech cykli zegarowych zapisać „0” do bitu WDE. W tym momencie watchdog zostanie zablokowany.

B2...0 – WDP2, WDP1, WDP0 (Watchdog Timer Prescaler) – bity konfiguruje preskaler watchdoga.

Bity WDP2, WDP1, WDP0 określają parametry preskalera watchdoga, gdy jest on aktywny. Ustawienie preskalera wpływa na długość *time-outu* watchdoga (tablica 6.1).

Tab. 6.1. Ustawienie preskalera watchdoga⁽¹⁾

WDP2	WDP1	WDP0	Liczba cykli oscylatora WDT	Typowa długość <i>time-outu</i> dla $V_{CC}=3\text{ V}$	Typowa długość <i>time-outu</i> dla $V_{CC}=5\text{ V}$
0	0	0	16 kcykli	47 ms	15 ms
0	0	1	32 kcykli	94 ms	30 ms
0	1	0	64 kcykli	0,19 s	60 ms
0	1	1	128 kcykli	0,38 s	0,12 s
1	0	0	256 kcykli	0,75 s	0,24 s
1	0	1	512 kcykli	1,5 s	0,49 s
1	1	0	1024 kcykli	3,0 s	0,97 s
1	1	1	2048 kcykli	6,0 s	1,9 s

Uwaga:

⁽¹⁾ Częstotliwość oscylatora watchdoga jest uzależniona od wartości napięcia zasilającego.

Rozkaz WDR (Watchdog Reset) powinien być zawsze wykonany przed włączeniem watchdoga. Tym samym mamy pewność, że okres zerowania będzie zgodny z ustawieniami preskalera. Jeśli watchdog jest włączony bez wcześniejszego zerowania, to jego stan początkowy może być różny od zera. Dla uniknięcia niezamierzonego zerowania MCU, watchdog powinien być blokowany lub zerowany przed zmianą parametrów jego preskalera.

Przykład 6.2. Obsługa watchdoga (wersja programu w asemblerze)

```
.def temp=r20
.def temp1=r21

...
ldi temp,$08      ;przygotowanie wpisu do WDTCSR
in temp1,sreg      ;zapamiętanie statusu CPU
cli               ;zablokowanie przerwań
wdr               ;reset watchdoga
out wdtcr,temp     ;start watchdoga z minimalnym timeoutem
out sreg,temp1     ;odtworzenie statusu CPU

...

wdr               ;zerowanie watchdoga w takim miejscu programu,
                  ;przez który licznik rozkazu musi przechodzić
                  ;w odstępach czasu nie dłuższych niż wartość
                  ;time-outu

...

ldi temp,$18      ;procedura wyłączenia watchdoga
ldi temp1,sreg     ;zapamiętanie statusu CPU
cli               ;zablokowanie przerwań
wdr               ;
out wdtcr,temp     ;WDTOE=1 i WDE=1
ldi temp,$10      ;
out wdtcr,temp     ;WDTOE=1 i WDE=0 - stop watchdoga, gdyby nie
                  ;było powyższego wpisu do wdtcr, watchdog nie
                  ;zatrzymałby się
out sreg,temp1     ;odtworzenie statusu CPU
...
```

Przykład 6.3. Obsługa watchdoga (wersja programu w języku C)

```
...
wdt_reset();      //zerowanie licznika watchdoga
wdt_enable(0);    //start watchdoga z minimalnym timeoutem
...

wdt_reset();      //zerowanie licznika watchdoga w takim miejscu
                  //programu, przez który licznik rozkazu musi
                  //przechodzić w odstępach czasu nie dłuższych
                  //niż wartość time-outu

...

wdt_disable();    //wyłączenie watchdoga, procedura biblioteczna
                  //realizuje ustawienie odpowiedniej sekwencji
                  //bitów WDTOE=1 i WDE=1
...
```

7. Pamięć danych EEPROM

Większość konstruowanych współcześnie urządzeń wykorzystujących mikrokontrolery wymaga wielokrotnego zachowywania specyficznych informacji, np. o ostatnich ustawieniach elementów regulacyjnych, trybach pracy, zakresach pomiarowych lub też innych parametrach pracy sterownika. Dane te powinny być zachowywane także po wyłączeniu zasilania. Użyta do tego celu pamięć powinna charakteryzować się długim czasem przechowywania danych i dostatecznie dużą liczbą cykli zapisu/kasowania. Jej pojemność w większości przypadków nie musi być bardzo duża. Zazwyczaj wystarcza kilkadziesiąt do kilkuset bajtów. Nie są też bardzo krytyczne wymagania dotyczące czasu dostępu do takiej pamięci, gdyż nie pełni ona funkcji pamięci danych, w której przechowuje się dynamicznie modyfikowane zmienne programu. Powyższe wymagania spełnia pamięć EEPROM. Występuje ona w prawie wszystkich odmianach mikrokontrolerów AVR, nie ma jej tylko w kilku wersjach ATtiny (patrz dodatek A).

7.1. Zapis i odczyt pamięci

W strukturach mikrokontrolerów AVR zawarto wszystkie elementy niezbędne do programowania wewnętrznej pamięci EEPROM, takie jak logika sterująca i pompy ładunkowe wytwarzające napięcia programujące. Korzystanie z pamięci EEPROM może się więc odbywać w typowej aplikacji mikrokontrolera bez żadnych elementów zewnętrznych. Programista musi jednakże zachować pewne środki ostrożności w tych miejscach programu, w których odwołuje się do pamięci EEPROM. Powodem tego mogą być np. nieoczekiwane zachowania mikrokontrolera podczas włączania lub wyłączania systemu. Zbyt wolne pojawianie się lub zanik napięcia zasilającego może spowodować nieoczekiwany skok do rozkazów korzystających z pamięci EEPROM i wykonanie rozkazu zapisu do niej. Między innymi z tego powodu wskazane jest stosowanie zewnętrznego układu zerowania mikrokontrolera. Zapis do pamięci odbywa się poprzez specjalną procedurę minimalizującą prawdopodobieństwo niekontrolowanego dostępu do EEPROM-u. Będzie ona opisana w dalszej części rozdziału.

Operacje zapisu i odczytu pamięci EEPROM są realizowane poprzez wydzielone rejestry specjalne. Czas dostępu podczas zapisu waha się w przedziale od 2,5 do 4 ms, w zależności od wartości napięcia zasilającego V_{CC} .

UWAGA

Podczas zapisywania EEPROM-u, praca CPU zostaje wstrzymana na dwa cykle zegarowe, w przypadku odczytu, CPU jest zatrzymana na cztery cykle zegarowe. Dopiero po tym czasie jest wykonywany następny rozkaz.

EEAR (Address Register) – rejestr adresowy pamięci EEPROM – \$1E

Bit	7	6	5	4	3	2	1	0	
\$1E (\$3E)	–	EEAR6	EEAR5	EEAR4	EEAR3	EEAR2	EEAR1	EEAR0	EEAR
Odczyt/Zapis	R	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Wartość początkowa	0	0	0	0	0	0	0	0	

R – oznacza odczyt, R/W – oznacza odczyt/zapis

B7 – zarezerwowany. Ten bit jest zarezerwowany w układzie AT90S2313 i zawsze odczytywany jako zero.

B6...0 – EEAR6...0 (EEPROM Address): adres zapisu/odczytu pamięci EEPROM.

Bity te zawierają adres dostępu do pamięci EEPROM. Określają pojedynczą komórkę EEPROM-u dla operacji zapisu lub odczytu z całego 128-bajtowego obszaru jakim dysponuje mikrokontroler. Adresowanie pamięci jest liniowe pomiędzy adresami 0 i 127 (\$7F).

EEDR (Data Register) – rejestr danych pamięci EEPROM – \$1D

Bit	7	6	5	4	3	2	1	0	
\$1D (\$3D)	MSB							LSB	EEDR
Odczyt/Zapis	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Wartość początkowa	0	0	0	0	0	0	0	0	

R/W – oznacza odczyt/zapis

B7...0 – EEDR7...0 (EEPROM Data): dana zapisywana/odczytywana do/z pamięci EEPROM.

Podczas operacji zapisu pamięci EEPROM, dana umieszczona wcześniej w rejestrze EEDR jest wpisywana pod adres określony zawartością rejestru EEAR. Podczas operacji odczytu, w rejestrze EEDR zostaje umieszczona dana z pamięci EEPROM spod adresu określonego w rejestrze EEAR.

EECR (Control Register) – rejestr sterujący pamięci EEPROM – \$1C

Bit	7	6	5	4	3	2	1	0	
\$1C (\$3C)	–	–	–	–	–	EEMWE	EWE	EERE	EECR
Odczyt/Zapis	R	R	R	R	R	R/W	R/W	R/W	
Wartość początkowa	0	0	0	0	0	0	0	0	

R – oznacza odczyt, R/W – oznacza odczyt/zapis

B7...3 – zarezerwowane. Te bity są zarezerwowane w układzie AT90S2313 i zawsze odczytywane jako zero.

B2 – EEMWE (EEPROM Master Write Enable): bit zezwolenia na zapis do pamięci EEPROM.

Bit EEMWE określa, czy zapis do pamięci EEPROM może nastąpić, czy nie. Jeśli EEMWE jest ustawiony („1”), ustawienie bitu EEWB spowoduje zapisanie danej pod wskazany adres. Jeśli EEMWE ma wartość zero, ustawienie bitu EEWB nie powoduje zapisu. Zapis do pamięci powinien być wykonany najdalej w ciągu czterech cykli po ustawieniu bitu EEMWE. Jeśli to nie nastąpi, to bit ten zostanie sprzętowo wyzerowany (bez ingerencji programu).

B1 – EEWB (EEPROM Write Enable): strob zapisu do pamięci EEPROM.

Bit EEWB stanowi strob zapisu do pamięci EEPROM. Po ustawieniu odpowiednich wartości rejestrów EEAR (adres) i EEDR (dana) ustawienie bitu EEWB powoduje wykonanie zapisu do pamięci EEPROM. Bezpośrednio przed wystawieniem strobu musi być jeszcze ustawiony bit EEMWE. Procedura zapisu powinna wyglądać następująco:

1. Czekaj do momentu, gdy EEWB osiągnie wartość zero.
2. Zapisz nowy adres do rejestru EEAR (opcjonalnie).
3. Zapisz nową daną do rejestru EEDR (opcjonalnie).
4. Ustaw bit EEMWE w rejestrze EECR. Aby było możliwe ustawienie tego bitu, bit EEWB musi być w tym samym cyklu zapisywany wartością zero.
5. Najdalej w ciągu czterech cykli zegara po ustawieniu bitu EEMWE, zapisać logiczną „1” do bitu EEWB.

Po chwili równej czasowi dostępu (typowo 2,5 ms dla $V_{CC} = 5\text{ V}$ lub 4 ms dla $V_{CC} = 2,7\text{ V}$) bit EEMWE jest sprzętowo zerowany. Programista powinien zadbać, aby bit EEWB był zawsze sprawdzany przed wykonaniem operacji zapisu do pamięci EEPROM, aż osiągnie wartość zero. Po ustawieniu bitu EEWB praca CPU jest wstrzymywana na czas trwania dwóch cykli zegarowych i dopiero po tym czasie jest wykonywany następny rozkaz.



Jeśli między krokiem 4. a 5. powyższego algorytmu wystąpi jakiegokolwiek przerwanie, operacja zapisu może się nie powieść z powodu przekroczenia *time-outu* zapisu. Może się też zdarzyć, że procedura obsługi przerwania wykorzystująca dostęp do pamięci EEPROM zmodyfikuje rejestry EEAR i EEDR powodując błędne działanie programu. Zaleca się więc globalne blokowanie przerw w czasie wykonywania kroków 2. do 4.

B0 – EERE (EEPROM Read Enable): strob odczytu pamięci EEPROM.

Bit EERE jest strobem odczytu pamięci EEPROM. Aby odczytać dane z pamięci EEPROM, po wpisaniu odpowiedniego adresu w rejestrze EEAR, należy wstawić bit EERE. W chwili, gdy EERE zostanie sprzętowo wyzerowany, odczytywana dana zostanie umieszczona w rejestrze EEDR, jednakże nie jest konieczne cykliczne sprawdzanie bitu EERE. Po ustawieniu bitu EERE praca CPU zostaje wstrzymana na czas równy czterem cyklom zegarowym przed wykonaniem następnego rozkazu. Programista powinien jednak sprawdzić stan bitu EEWB przed rozpoczęciem czytania pamięci EEPROM. Nie może on być równy „1”. Jeśli tak będzie (co oznacza, że trwa operacja zapisu), to zmiana rejestru danej lub adresu EEPROM-u spowoduje, że zapis zostanie przerwany, a jego rezultat będzie niezdefiniowany.

Przykład 7.1. Obsługa wewnętrznej pamięci EEPROM w asemblerze. Dwie dane będą kolejno zapisywane pod adresy \$01 i \$02, a następnie zostaną odczytane

```
.def temp=r16
.def danal=r17
.def dana2=r18
.def temp1=r19
...
;Zapis EEPROM-u
L1: sbic EECR,EWE ;czekaj na gotowość
    rjmp L1
    ldi temp,1
    out eear,temp ;ustaw adres zapisu = 1
    ldi temp,$7f
    out eedr,temp ;ustaw daną do zapisu np.=$7f
    sbi eecr,eemwe ;przygotuj EEPROM do zapisu
    sbi eecr,eewb ;zapisz pod adres 1 daną $7f
...
L2: sbic EECR,EWE ;czekaj na gotowość
    rjmp L2
    ldi temp,2
    out eear,temp ;ustaw adres zapisu = $02
    ldi temp,0xbf
    out eedr,temp ;ustaw daną do zapisu np.=$bf
    in temp1,sreg ;zachowaj status CPU
    cli ;zablokuj globalne przerwania
    sbi eecr,eemwe ;przygotuj EEPROM do zapisu
    sbi eecr,eewb ;zapisz pod adres 2 daną 0xbf
    out sreg,temp1 ;odtwórz status CPU
...
;Odczyt EEPROM-u
L3: sbic EECR,EWE ;czekaj na gotowość
    rjmp L3
    ldi temp,0x1
    out eear,temp ;ustaw adres odczytu = 1
    sbi eecr,eere ;czytaj eeprom[1]
    in danal,EEDR ;i umieść w zmiennej danal
    inc temp
    out eear,temp ;ustaw adres odczytu = 2
    sbi eecr,eere ;czytaj eeprom[2]
    in dana2,EEDR ;i umieść w zmiennej dana2
...
...
```

Przykład 7.2. Ten sam problem co w przykładzie 7.1 zrealizowany w języku C, wzbogacony o wyświetlanie stanu odczytanej pamięci na diodach LED sterowanych z PORTB

```
#include <io.h>
#include <io2313.h>
#include <eeprom.h>

void czekaj(unsigned long zt) //procedura pomocnicza
{
    unsigned char zt1;
    for(;zt>0;zt--)
    {
        for(zt1=255;zt1!=0;zt1--);
    }
}

int main(void)
{
    unsigned char dana;
    PORTB=0xff; //gaś LED-y
    DDRB=0xff; //PORTB w całości jako wyjściowy
    czekaj(10000L); //czekaj ok. 1 s
    eeprom_wb(1,0xf); //zapisz daną 0xf do EEPROM-u pod adres = 1
    eeprom_wb(2,0xbf); //zapisz daną 0xbf do EEPROM-u pod adres = 2
    dana=eeprom_rb(1); //czytaj EEPROM[1]
    PORTB=dana; //i wyświetl na LED-ach
    czekaj(10000L); //czekaj ok. 1 s
    dana=eeprom_rb(2); //czytaj EEPROM[2]
    PORTB=dana; //i wyświetl na LED-ach
    czekaj(10000L); //czekaj ok. 1 s
    while (1); //zapętł program
}
```

Jak widać, rozwiązanie tego samego problem w języku C jest znacznie prostsze. Odpada żmudne operowanie na pojedynczych bitach rejestrów, sprawdzanie warunków gotowości EEPROM-u do zapisu itp. Oczywiście wszystkie te operacje są wykonywane w procedurach bibliotecznych języka C.



Mikrokontrolery AVR nie są niestety pozbawione pewnych przykrych dla użytkownika wad. Jedną z nich jest brak zabezpieczenia przed sytuacją, w której w trakcie wykonywania zapisu do EEPROM-u nastąpi zerowanie mikrokontrolera. Cykl zapisu zostanie dokończony normalnie, ale rejestr adresu zostanie wyzerowany. Dana zostanie zapisana pod adres wskazany i pod adres 0, co oczywiście jest zachowaniem niepoprawnym. Z tego powodu zaleca się niewykorzystywanie zerowego adresu pamięci EEPROM, mimo że opisywana sytuacja wydaje się być mało prawdopodobna.

7.2. Zapewnienie prawidłowych warunków pracy pamięci EEPROM

Pamięć EEPROM do prawidłowego działania wymaga stabilnego napięcia zasilającego o wartości mieszczącej się w dopuszczalnym przedziale. Z jednej strony zależy od tego prawidłowe wytworzenie napięć programujących, z drugiej zaś obniżone napięcie zasilające może mieć szkodliwy wpływ na działanie mikrokontrolera. W efekcie mogą powstawać nieprawidłowości pracy, w szczególności związane z dostępem do pamięci EEPROM. Aby uchronić się od opisanych problemów zaleca się stosowanie jednej z następujących zasad:

1. Utrzymywać aktywny stan linii **RESET** mikrokontrolera przez czas ustalania się napięcia zasilającego. Najpopularniejszą metodą jest wykorzystywanie w aplikacjach specjalizowanych układów zerowania (np. DS1813). Przy ich doborze trzeba pamiętać, aby miały odpowiedni poziom aktywny sygnału zerującego. Mikrokontrolery AVR są zerowane poziomem niskim. Układy takie są nazywane *Brown-out Detector* (BOD) i zapewniają prawidłowe generowanie sygnału zerującego oraz kontrolę obniżenia napięcia zasilającego. Wiele dodatkowych informacji można znaleźć w nocie AVR180 dostępnej na internetowej stronie Atmela (www.atmel.com).

2. Gdy napięcie zasilające jest obniżone, należy utrzymywać mikrokontroler w stanie uśpienia. Rozwiązanie takie zapobiegnie nieprawidłowemu wykonaniu rozkazów, skutecznie zabezpieczając komórki pamięci EEPROM przed niekontrolowanymi zmianami.

3. Stałe wykorzystywane przez program należy przechowywać w pamięci Flash (w pamięci programu), chroniąc je tym samym przed jakimikolwiek zmianami wywołanymi przez źle działający program. Pamięć Flash nie może być bowiem modyfikowana przez CPU (nie dotyczy mikrokontrolerów ATmega).

8. Układ transmisji szeregowej (UART)

W wielu przypadkach system oparty na mikrokontrolerze musi komunikować się z jakimś urządzeniem zewnętrznym. Aktualnie coraz powszechniej wykorzystuje się do tego celu interfejs USB, ale w dalszym ciągu klasyczny interfejs – jakim jest RS232 – cieszy się ogromną popularnością wśród użytkowników. Dlatego też, wiele współczesnych mikrokontrolerów ma wbudowane odpowiednie bloki funkcjonalne do realizacji asynchronicznej (niekiedy również synchronicznej) transmisji szeregowej. Określane są one mianem UART (*Universal Asynchronous Receiver and Transmitter*). Stanowią one kompletne rozwiązanie interfejsu pod względem logicznym, jednak gdy transmisja ma być prowadzona na większe odległości wymagają zastosowania zewnętrznych układów dopasowujących poziom sygnałów do standardu RS232. Można z nich zrezygnować w przypadku wieloprotokolowych aplikacji, w których poszczególne jednostki pracują w swoim bezpośrednim sąsiedztwie. Większość urządzeń obsługujących transmisję szeregową wykorzystuje interfejs RS232, w którym logicznej jedynce przypisano wartość napięcia z przedziału od -3 do -12 V, zaś logicznemu zeru przypisano wartość napięcia z przedziału od 3 do 12 V. Sygnały na liniach transmisyjnych nie powinny osiągać wartości od -3 do +3 V. Najpopularniejszym układem do realizacji takiego interfejsu jest MAX232 lub jego odmiany. W niektórych przypadkach zadowalające może być wykonanie konwertera poziomów na tranzystorze. Mimo popularności tego typu transmisji nie wszystkie mikrokontrolery AVR wyposażono w UART (patrz porównanie parametrów w dodatku A). W takich przypadkach konieczna jest niestety całkowicie programowa jego implementacja. Stanowi to pewne utrudnienie, ale nie jest niemożliwe do zrobienia. Opisany tu układ AT90S2313 na szczęście ma w swojej architekturze duplexowy blok UART-u, do którego dostęp zapewniają wydzielone rejestry nadajnika i odbiornika.

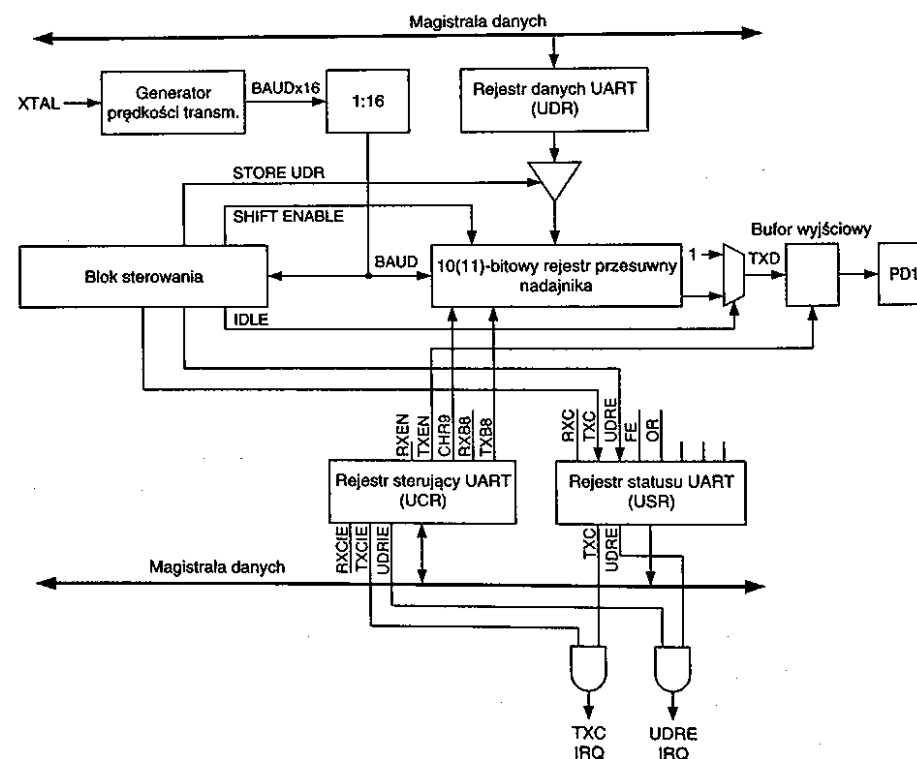
Podstawowe cechy UART-u wbudowanego w mikrokontrolery AVR to:

- wbudowany generator podstawy czasu dla transmisji szeregowej,
- możliwość uzyskiwania wysokich prędkości transmisji dla niskich częstotliwości rezonatora kwarcowego,
- dana o długości 8 lub 9 bitów,
- układ redukcji szumów zakłócających transmisję,
- detekcja nadpisania zawartości bufora odbiornika,
- detekcja błędu ramki,
- detekcja błędnego bitu startu,
- trzy wydzielone przerwy do obsługi zdarzeń: zakończenia transmisji danej, opróżnienia bufora nadawczego, zapelnienia bufora odbiorczego.

8.1. Budowa i działanie nadajnika UART

Schemat blokowy układu nadajnika przedstawiono na rysunku 8.1. Nadawanie jest inicjowane przez zapisanie wysyłanej danej do rejestru nadajnika UDR (*UART I/O Register*). Dana ta jest następnie przenoszona do wyjściowego rejestru przesuwającego. Jeśli zapis do UDR nastąpi po tym, gdy bit stopu poprzedniego znaku opuści rejestr przesuwający, to rejestr ten jest ładowany bezpośrednio nową daną.

Jeśli zapis do UDR nastąpi przed wysłaniem bitu stopu poprzedniego znaku, to nowa dana czeka w rejestrze UDR na przepisanie do wyjściowego rejestru przesuwającego do momentu, gdy bit stopu poprzedniego znaku zostanie wysłany. Rejestr przesuwający ma długość 10 (lub 11) bitów (8 lub 9 bitów danej + znak startu + znak stopu). W chwili, gdy rejestr przesuwający nadajnika jest pusty, nowa dana jest przenoszona do niego z rejestru UDR i jednocześnie zostaje ustawiony bit UDRE (*UART Data Register Empty*) znajdujący się w rejestrzeUSR (*UART Status Register*), co jest sygnałem gotowości nadaj-



Rys. 8.1. Schemat blokowy nadajnika układu UART

nika do przyjęcia nowej danej. Jednocześnie z przenoszeniem znaku z rejestru UDR do rejestru przesuwającego bit 0 tego rejestru jest zerowany (będzie to bit startu), a bit 9 (lub 10) jest ustawiany (będzie to bit stopu). Jeśli wybrano 9-bitową długość słowa, to bit TXB8 z rejestru UCR (*UART Control Register*) jest przenoszony do bitu 9 rejestru przesuwającego. Długość słowa określa się nadając odpowiednią wartość bitowi CHR9 znajdującemu się w rejestrze UCR. Dla słowa 9-bitowego bit ten powinien być ustawiony („1”). Przesuwanie bitów w rejestrze wyjściowym jest taktowane zegarem transmisyjnym (*Baud Rate Clock*). Jest on niezależny od zegara taktującego CPU. Bity nadawanego znaku pojawiają się na wyprowadzeniu TXD mikrokontrolera. Najpierw bit startu, potem bity danej począwszy od najmniej znaczącego (LSB). Na końcu jest wysyłany bit stopu. Kiedy bit stopu opuści rejestr przesuwający, zostaje do niego załadowana nowa dana z rejestru UDR (jeśli wcześniej została tam umieszczona). Podczas ładowania bit UDRE jest ustawiony. Jeśli w rejestrze UDR nie ma kolejnego znaku do wysłania, to bit UDRE zostaje ustawiony w chwili wyjścia poprzedniego bitu stopu z rejestru przesuwającego i pozostaje w tym stanie do momentu kolejnego wpisu do rejestru UDR. Jeśli nie nastąpi zapisanie nowej danej, gdy bit stopu występuje na wyjściu TXD przez czas trwania transmisji jednego bitu, to zostanie ustawiona flaga *TX Complete Flag* (TXC) znajdująca się w rejestrze USR. Nadajnik może być włączany lub wyłączny za pomocą bitu TXEN z rejestru UCR. Jego ustawienie („1”) uaktywnia nadajnik. Jeśli bit ten będzie wyzerowany („0”), wyprowadzenie PD1 może być wykorzystywane jako wejście/wyjście ogólnego przeznaczenia.



Ustawienie TXEN powoduje całkowite przechwycenie funkcji wyprowadzenia PD1 przez nadajnik, niezależnie od ustawienia bitu DDD1 w rejestrze DDRD.

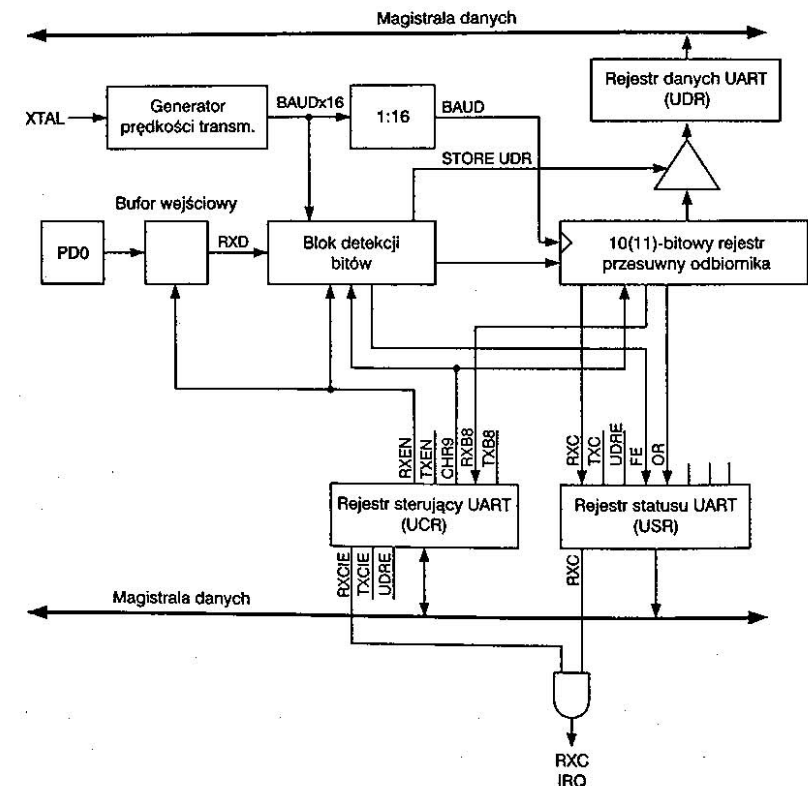
Ustawienie bitu CHR9 w rejestrze UCR powoduje, że nadawane i odbierane znaki mają długość 9 bitów, plus bit startu i bit stopu. Musi mu być nadana odpowiednia wartość przed zainicjowaniem transmisji. Dziewiąty bit do wysłania, to TXB8 znajdujący się w rejestrze UCR.

8.2. Budowa i działanie odbiornika UART

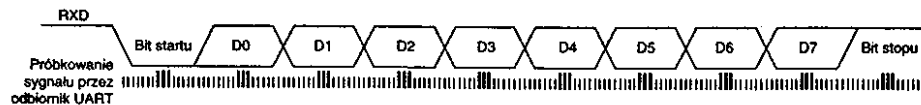
Schemat blokowy układu odbiornika przedstawiono na rysunku 8.2. Wejście odbiornika transmisji szeregowej jest połączone za pomocą odpowiednich układów przełączających z wyprowadzeniem PD0/RXD. Wewnętrzna logika próbkuje sygnał 16 razy w ciągu czasu trwania transmisji jednego bitu.

W czasie, gdy linia znajduje się w stanie *idle* (oczekiwanie na bit startu), pojedyncza próbka o wartości „0” jest interpretowana jako opadające zbocze sygnału wejściowego. Powoduje to zainicjowanie sekwencji działań związanych z wykryciem bitu startu. Przyjmijmy, że próbka ta ma numer 1.

Tak więc po zmianie stanu linii odbiorczej (RXD) z 1 na 0 odbiornik analizuje 8, 9 i 10 próbkę. Jeśli co najmniej dwie z nich będą miały wartość „1”, wykrycie bitu startu jest anulowane. Sytuację taką uznaje się za zakłócenie impulsowe i cała powyższa procedura jest wznowiana od początku. W przeciwnym razie uznaje się, że odebrano bit startu. Próbkowanie kolejnych bitów odbywa się w podobny sposób. Wartości co najmniej dwóch jednakowych próbek spośród 8, 9 i 10 decydują o wartości odebranego bitu. Jednakowe próbki nie muszą przy tym występować kolejno po sobie. Sekwencja np. 101 jest interpretowana jako bit o wartości „1”. Odebrany bit „wchodzi” do rejestru przesuwającego. Próbkowanie wejścia RXD zilustrowano na rysunku 8.3.



Rys. 8.2. Schemat blokowy odbiornika układu UART



Rys. 8.3. Próbkowanie danych odbieranych przez odbiornik UART-u

Układ odbiornika liczy odbierane bity. Ostatnim bitem ramki jest bit stopu, który powinien mieć wartość „1”. Jest on rozpoznawany w znany już sposób. Jeśli okaże się, że dwie lub trzy próbki (spośród 8, 9 i 10) bitu stopu są równe „0”, podejmowana jest decyzja o wystąpieniu błędu ramki i ustawiana jest flaga *Framing Error* (FE) w rejestrze statusu USR (*UART Status Register*). Przed odczytem danej z rejestru UDR programista powinien zawsze sprawdzić, stan tej flagi, aby upewnić się, czy nie wystąpił błąd transmisji. Po prawidłowym lub błędnym odebraniu bitu stopu dana jest przepisywana z rejestru przesuwającego do rejestru UDR. Jednocześnie jest ustawiana flaga RXC w rejestrze USR. Rejestr UDR to *de facto* fizycznie dwa oddzielne rejestry dla nadajnika i odbiornika. Podczas odczytu UDR jest udostępniany odbiorczy rejestr danych, podczas zapisywania natomiast jest udostępniany nadawczy rejestr danych. Jeśli wybrano 9-bitowe słowo danych (bit CHR9 w rejestrze UCR ma wartość „1”), bit RXB8 w rejestrze UCR jest ładowany 9 bitem z rejestru przesuwającego. Dzieje się to w tej samej chwili, co przepisywanie danej z rejestru przesuwającego do rejestru UDR. Tak skompletowana dana powinna być odczytana z rejestru UDR przed odebraniem następnej. Jeśli tak się nie stanie, to w rejestrze USR jest ustawiana flaga *OverRun* (OR), sygnalizując błąd nadpisania danej. W takiej sytuacji ostatnia dana skompletowana w rejestrze przesuwającym nie powinna być (teoretycznie) przenoszona do UDR. Flagą OR jest buforowana i zawsze aktualizowana po odczytaniu prawidłowej danej z rejestru UDR. Programista powinien zawsze sprawdzać stan flagi OR po odczytaniu rejestru UDR. Występowanie błędów nadpisania może być konsekwencją zbyt dużej prędkości transmisji lub nadmiernym obciążeniem jednostki centralnej innymi zadaniami o wyższym priorytecie. Wyzerowanie („0”) bitu RXEN w rejestrze UCR powoduje zablokowanie odbiornika. Wyprowadzenie PD0/RXD może być wówczas wykorzystywane jako wejście/wyjście ogólnego przeznaczenia.

W przypadku ustawienia („1”) bitu CHR9 w rejestrze UCR, dziewiąty bit odebranego znaku to RXB8 znajdujący się w rejestrze UCR.



Włączenie odbiornika powoduje całkowite przechwycenie funkcji wyprowadzenia PD0 przez odbiornik, niezależnie od ustawienia bitu DDD0 w rejestrze DDRD.

8.3. Sterowanie transmisją

Do obsługi transmisji szeregowej poprzez UART służą cztery rejestry specjalne.

UDR (*UART I/O Data Register*) – rejestr danych nadawanych/odbieranych przez UART – \$0C

Bit	7	6	5	4	3	2	1	0	
\$0C (\$2C)	MSB							LSB	UDR
Odczyt/Zapis	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Wartość początkowa	0	0	0	0	0	0	0	0	
R/W – oznacza odczyt/zapis									

Rejestr UDR to fizycznie dwa oddzielne rejestry występujące pod tym samym adresem w przestrzeni I/O. Podczas zapisu dane są kierowane do rejestru danych wysyłanych (*UART Transmit Data Register*). Odczyt następuje z rejestru danych odbieranych (*UART Receive Data Register*).

USR (*UART Status Register*) – rejestr stanu UART-u – \$0B

Bit	7	6	5	4	3	2	1	0	
\$0B (\$2B)	RXC	TXC	UDRE	FE	OR	–	–	–	USR
Odczyt/Zapis	R	R/W	R	R	R	R	R	R	
Wartość początkowa	0	0	1	0	0	0	0	0	
R – oznacza odczyt, R/W – oznacza odczyt/zapis									

Rejestr USR może być tylko odczytywany. Sygnalizuje stan układu UART.

B7 – RXC (*UART Receive Complete*): flaga sygnalizująca odebranie znaku przez odbiornik.

Znacznik ten jest ustawiany („1”) w chwili, gdy odebrana dana jest przenoszona z odbiorczego rejestru przesuwającego do rejestru UDR. Ustawienie bitu RXC odbywa się niezależnie od ewentualnego wykrycia błędów transmisji. Jeśli bit RXCIE z rejestru UCR jest ustawiony („1”), to ustawienie flagi RXC będzie oznaczało zgłoszenie przerwania *UART Receive Complete*, oznaczającego, że w rejestrze UDR znajduje się dana do odczytu. Bit RXC jest automatycznie kasowany w chwili odczytu rejestru UDR. Jeśli transmisję prowadzi się wykorzystując system przerwań, to procedura obsługi przerwania *UART Receive Complete* powinna odczytać rejestr UDR w celu skasowania flagi RXC. W przeciwnym razie, zanim zostanie zakończona aktualna procedura obsługi wystąpi kolejne przerwanie.

B6 – TXC (*UART Transmit Complete*): flaga sygnalizująca zakończenie wysyłania danej przez nadajnik.

Znacznik ten jest ustawiany („1”) w chwili, gdy cała nadawana dana (włączając bit stopu) zostanie wysunięta z nadawczego rejestru przesuwającego i nie ma

nowej danej zapisanej w rejestrze UDR. Znacznik ten jest szczególnie przydatny podczas obsługi transmisji półdupleksowej, w której bezpośrednio po wysłaniu znaku trzeba przechodzić „na odbiór” i zwalniać linię transmisyjną. Jeśli bit TXCIE z rejestru UCR jest ustawiony („1”), to ustawienie flagi TXC będzie oznaczało zgłoszenie przerwania *UART Transmit Complete*, oznaczającego, że w rejestrze UDR nie ma danej do wysłania. Bit TXC jest automatycznie zerowany w chwili wykonania skoku do odpowiedniego wektora przerwania. Można też go zerować poprzez wpisanie logicznej jedynki („1”) do tego bitu:

```
sbi    ucr,txc    ;zeruj flagę TXC w rejestrze UCR
```

B5 – UDRE (UART Data Register Empty): flaga sygnalizująca opróżnienie bufora UART.

Bit UDRE jest ustawiany („1”) w chwili przenoszenia danej z rejestru UDR do nadawczego rejestru przesuwającego. Sygnalizowana jest tym samym gotowość nadajnika do wczytania nowej danej przeznaczonej do wysłania. Jeśli bit UDRIE w rejestrze UCR jest ustawiony („1”), to przerwanie *UART Transmit Complete* będzie mogło być obsługiwane dopóty, dopóki bit UDRE jest ustawiony. Flaga UDRE jest zerowana przez zapisanie rejestru UDR. Jeśli transmisję prowadzi się wykorzystując system przerwań, to procedura obsługi przerwania *UART Data Register Empty* musi zapisywać rejestr UDR w celu skasowania flagi UDRE. W przeciwnym razie, zanim zostanie zakończona aktualna procedura obsługi wystąpi kolejne przerwanie. Po restarcie mikrokontrolera flaga UDRE jest ustawiana („1”) sygnalizując gotowość nadajnika.

B4 – FE (Framming Error): flaga sygnalizująca błąd ramki.

Flaga FE jest ustawiana, gdy zostanie wykryty błąd ramki (np. w przypadku rozpoznania zerowej wartości bitu stopu), zerowana zaś automatycznie po odebraniu prawidłowego bitu stopu.

B3 – OR (OverRun): flaga sygnalizująca nadpisanie danych w rejestrze odbiornika.

Flaga OR jest ustawiana, jeśli zostanie wykryte nadpisanie danych w rejestrze odbiornika (np. gdy dana znajdująca się aktualnie w rejestrze UDR nie zostanie odczytana przed wprowadzeniem następnego znaku do odbiorczego rejestru przesuwającego). Znacznik OR jest zerowany po przesłaniu odebranej danej do rejestru UDR.

B2...0 – zarezerwowane.

Te bity są zarezerwowane w układzie AT90S2313 i zawsze odczytywane jako zero.

UCR (UART Control Register) – rejestr sterujący UART-u – \$0A

Bit	7	6	5	4	3	2	1	0	
\$0A (\$2A)	RXCIE	TXCIE	UDRIE	RXEN	TXEN	CHRS	RXB8	TXB8	UCR
Odczyt/Zapis	R/W	R/W	R/W	R/W	R/W	R/W	R	W	
Wartość początkowa	0	0	0	0	0	0	1	0	

R – oznacza odczyt, W – oznacza zapis, R/W – oznacza odczyt/zapis

B7 – RXCIE (RX Complete Interrupt Enable): bit zezwolenia na przerwanie od odbiornika (po odebraniu danej).

Ustawienie („1”) bitu RXCIE zezwala na przyjęcie przerwania *Receive Complete Interrupt*, zgłaszanego automatycznie przez UART po odebraniu danej i sygnalizowanego ustawieniem flagi RXC w rejestrze USR. Przerwanie zostanie obsłużone pod warunkiem globalnego zezwolenia na przerwanie.

B6 – TXCIE (TX Complete Interrupt Enable): bit zezwolenia na przerwanie od nadajnika (po wysłaniu danej).

Ustawienie („1”) bitu TXCIE zezwala na przyjęcie przerwania *Transmit Complete Interrupt*, zgłaszanego automatycznie przez UART po zakończeniu nadawania danej i sygnalizowanego ustawieniem flagi TXC w rejestrze USR. Przerwanie zostanie obsłużone pod warunkiem globalnego zezwolenia na przerwanie.

B5 – UDRIE (UART Data Register Empty Interrupt Enable): bit zezwolenia na przerwanie od nadajnika (po opróżnieniu bufora nadajnika UDR).

Ustawienie („1”) bitu UDRIE zezwala na przyjęcie przerwania *UART Data Register Empty Interrupt*, zgłaszanego automatycznie przez UART po opróżnieniu bufora nadajnika i sygnalizowanego ustawieniem flagi UDRE w rejestrze USR. Przerwanie zostanie obsłużone pod warunkiem globalnego zezwolenia na przerwanie.

B4 – RXEN (Receiver Enable): bit włączający/wyłączający odbiornik UART.

Ustawienie („1”) bitu RXEN powoduje włączenie odbiornika UART-u. Jeśli bit RXEN ma wartość „0” odbiór jest niemożliwy, a flagi RXC, OR i FE nie mogą stawać się aktywne (są ustawiane automatycznie po wystąpieniu odpowiednich zdarzeń). Jeśli flagi te zostały w czasie pracy ustawione, to wyzerowanie bitu RXEN nie wpływa na nie (są nadal ustawione).

B3 – TXEN (Transmitter Enable): bit włączający/wyłączający nadajnik UART.

Ustawienie („1”) bitu TXEN powoduje włączenie nadajnika UART. Jeśli bit TXEN zostanie wyzerowany w chwili, gdy nie została jeszcze zakończona transmisja, to nadajnik nie zostanie wyłączony natychmiast, lecz dopiero po całkowitym zakończeniu wysyłania bieżącej danej (wychodzącej z szeregowego rejestru przesuwającego nadajnika) i danej znajdującej się w rejestrze UDR (oczekującej na wysłanie).

B2 – CHR9 (9 Bit Characters): bit ustalający długość danej nadawanej/odbieranej.

Ustawienie („1”) bitu CHR9 powoduje przyjęcie 9-bitowej długości danej nadawanej i odbieranej plus bit stopu i bit startu. Dziewiąty bit odebranego znaku jest odczytywany z rejestru UCR (bit RXB8). Dziewiąty bit transmitowanego znaku jest zapisywany w rejestrze UCR (bit TXB8).

B1 – RXB8 (Receive Data Bit 8): ósmy (najbardziej znaczący) bit odebranej danej.

Kiedy bit CHR9 jest ustawiony („1”), RXB8 zawiera najstarszy bit odebranej danej.

B0 – TXB8 (Transmit Data Bit 8): ósmy (najbardziej znaczący) bit nadawanej danej.

Kiedy bit CHR9 jest ustawiony („1”), TXB8 zawiera najstarszy bit nadawanej danej.

8.4. Generator podstawy czasu transmisji (Baud Rate Generator)

Zegar synchronizujący pracę UART jest wytwarzany przez odpowiednie podzielenie częstotliwości oscylatora systemowego. Do obliczenia jego parametrów stosuje się poniższą formułę:

$$\text{BAUD} = f_{\text{CK}} / 16 \cdot (\text{UBRR} + 1)$$

gdzie: BAUD – prędkość transmisji [b/s],

f_{CK} – częstotliwość oscylatora taktującego CPU,

UBRR – wartość rejestru UBRR zapewniająca transmisję z ustaloną prędkością (może przyjmować wartości od 0 do 255).

Jak widać, ustawianie prędkości tylko poprzez zapisanie rejestru UBRR pociąga za sobą pewne niedogodności. Taka dyskretna forma regulacji z 256 (tylko) możliwymi krokami może budzić obawy co do jej skuteczności. Istotnie, nie da się w ten sposób uzyskać dowolnej prędkości transmisji dla dowolnej wartości zastosowanego kwarcu. Pamiętajmy jednak, że najczęściej jest stosowana jedna z kilku standardowych prędkości transmisji przewidzianych dla interfejsu RS232. Zasada prowadzenia transmisji asynchronicznej dopuszcza również pewną tolerancję częstotliwości zegara transmisyjnego. Zakłada się, że 1% błąd określenia tej prędkości nie powinien powodować praktycznych problemów. Gdyby pomimo tego nie udało się obliczyć para-

metru UBRR, należy próbować dobrać częstotliwość rezonatora kwarcowego. Wiąże się to niestety z ewentualnymi późniejszymi problemami z obliczeniami czasu w programie, jeśli z obliczeń wyniknie „nierówna” wartość kwarcu. Zestawienie z tablicy 8.1 ułatwi odpowiedni dobór wszystkich parametrów określających prędkość transmisji. Wytłuszczono wszystkie pozycje, w których obliczony błąd jest mniejszy niż 2%.

Tab. 8.1. Dobór parametru UBRR dla standardowych prędkości transmisji i kilku typowych rezonatorów kwarcowych

Prędkość transmisji	1 MHz	Błąd [%]	1,8432 MHz	Błąd [%]	2 MHz	Błąd [%]	2,4576 MHz	Błąd [%]
2400	UBRR=25	0,2	UBRR=47	0,0	UBRR=51	0,2	UBRR=63	0,0
4800	UBRR=12	0,2	UBRR=23	0,0	UBRR=25	0,2	UBRR=31	0,0
9600	UBRR=6	7,5	UBRR=11	0,0	UBRR=12	0,2	UBRR=15	0,0
14400	UBRR=3	7,8	UBRR=7	0,0	UBRR=8	3,7	UBRR=10	3,1
19200	UBRR=2	7,8	UBRR=5	0,0	UBRR=6	7,5	UBRR=7	0,0
28800	UBRR=1	7,8	UBRR=3	0,0	UBRR=3	7,8	UBRR=4	6,3
38400	UBRR=1	22,9	UBRR=2	0,0	UBRR=2	7,8	UBRR=3	0,0
57600	UBRR=0	7,8	UBRR=1	0,0	UBRR=1	7,8	UBRR=2	12,5
76800	UBRR=0	22,9	UBRR=1	33,3	UBRR=1	22,9	UBRR=1	0,0
115200	UBRR=0	84,3	UBRR=0	0,0	UBRR=0	7,8	UBRR=0	25,0
Prędkość transmisji	3,2768 MHz	Błąd [%]	3,6864 MHz	Błąd [%]	4 MHz	Błąd [%]	4,608 MHz	Błąd [%]
2400	UBRR=84	0,4	UBRR=95	0,0	UBRR=103	0,2	UBRR=119	0,0
4800	UBRR=42	0,8	UBRR=47	0,0	UBRR=51	0,2	UBRR=59	0,0
9600	UBRR=20	1,6	UBRR=23	0,0	UBRR=25	0,2	UBRR=29	0,0
14400	UBRR=13	1,6	UBRR=15	0,0	UBRR=16	2,1	UBRR=19	0,0
19200	UBRR=10	3,1	UBRR=11	0,0	UBRR=12	0,2	UBRR=14	0,0
28800	UBRR=6	1,6	UBRR=7	0,0	UBRR=8	3,7	UBRR=9	0,0
38400	UBRR=4	6,3	UBRR=5	0,0	UBRR=6	7,5	UBRR=7	6,7
57600	UBRR=3	12,5	UBRR=3	0,0	UBRR=3	7,8	UBRR=4	0,0
76800	UBRR=2	12,5	UBRR=2	0,0	UBRR=2	7,8	UBRR=3	6,7
115200	UBRR=1	12,5	UBRR=1	0,0	UBRR=1	7,8	UBRR=2	20,0
Prędkość transmisji	7,3728 MHz	Błąd [%]	8 MHz	Błąd [%]	9,216 MHz	Błąd [%]	11,059 MHz	Błąd [%]
2400	UBRR=191	0,0	UBRR=207	0,2	UBRR=239	0,0	UBRR=287	–
4800	UBRR=95	0,0	UBRR=103	0,2	UBRR=119	0,0	UBRR=143	0,0
9600	UBRR=47	0,0	UBRR=51	0,2	UBRR=59	0,0	UBRR=71	0,0
14400	UBRR=31	0,0	UBRR=34	0,8	UBRR=39	0,0	UBRR=47	0,0
19200	UBRR=23	0,0	UBRR=25	0,2	UBRR=29	0,0	UBRR=35	0,0
28800	UBRR=15	0,0	UBRR=16	2,1	UBRR=19	0,0	UBRR=23	0,0
38400	UBRR=11	0,0	UBRR=12	0,2	UBRR=14	0,0	UBRR=17	0,0
57600	UBRR=7	0,0	UBRR=8	3,7	UBRR=9	0,0	UBRR=11	0,0
76800	UBRR=5	0,0	UBRR=6	7,5	UBRR=7	6,7	UBRR=8	0,0
115200	UBRR=3	0,0	UBRR=3	7,8	UBRR=4	0,0	UBRR=5	0,0

UBRR (UART Baud Rate Register) – rejestr wyboru prędkości transmisji
UART-u – \$09

Bit	7	6	5	4	3	2	1	0	
\$09 (\$29)	MSB							LSB	UBRR
Odczyt/Zapis	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Wartość początkowa	0	0	0	0	0	0	0	0	
R/W – oznacza odczyt/zapis									

Rejestr UBRR zawiera 8-bitowy parametr określający prędkość transmisji szeregowej, zgodnie z przedstawionymi wyżej obliczeniami.

UWAGA

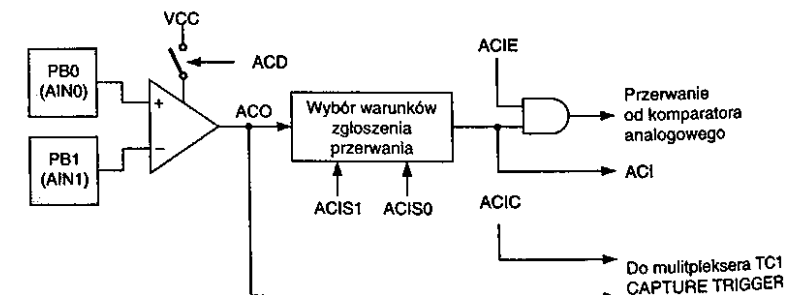
Układ UART mikrokontrolera AT90S2313 może utracić synchronizację, jeśli podczas wyłączenia odbiornika na linii RXD będzie występował stan niski. Dzieje się tak, gdyż wewnętrzna logika bloku UART wykrywa rozpoczęcie transmisji (bit startu) nawet wówczas, gdy UART nie jest włączony. W takiej sytuacji pierwszy odebrany bajt po włączeniu UART będzie błędny. Aby uchronić się od takiego przypadku należy zapewnić utrzymywanie linii RXD w stanie wysokim podczas wyłączenia interfejsu UART.

9. Komparator analogowy

Marzeniem każdego konstruktora jest mikrokontroler wyposażony w przetwornik analogowo-cyfrowy. Niestety, często okazuje się, że wykonanie takiego układu chociaż możliwe, nie jest proste. Zaimplementowanie dobrego przetwornika A/C w strukturze mikrokontrolera jest na ogół okupione dość znacznym wzrostem kosztu wykonania układu.

Minimalizacja wzajemnego oddziaływania na siebie bloków analogowych i cyfrowych w jednym układzie stanowi niemałe wyzwanie dla biur konstrukcyjnych producentów. Rezygnacja z tego komponentu znajduje uzasadnienie, gdyż w wielu przypadkach problem braku przetwornika można obejść pewnymi metodami. Wprawdzie specjalizowane, „analogowe” aplikacje nie obędą się bez prawdziwego przetwornika A/C, lecz w mniej wymagających systemach można do celów przetwarzania analogowo-cyfrowego zastosować komparator analogowy. Okazuje się to na tyle prostym i tanim rozwiązaniem, że jest stosowane w wielu bardzo popularnych mikrokontrolerach. Komparator taki może być oczywiście również wykorzystywany do innych zadań.

Większość mikrokontrolerów AVR – w tym AT90S2313 – ma w swojej strukturze komparator analogowy (patrz zestawienie w dodatku A). Jego zadaniem jest porównywanie dwóch napięć doprowadzonych do wejścia nieodwracającego AIN0 (PB0) i wejścia odwracającego AIN1 (PB1) (rysunek 9.1). Wyjście komparatora ACO (Analog Comparator Output) jest w stanie wysokim („1”), jeśli napięcie na wejściu nieodwracającym AIN0 jest większe od napięcia na wejściu odwracającym AIN1. Wyjście komparatora może być wykorzystywane do wyzwalania funkcji przechwytywania Timera/Licznika. W systemie przerwań mikrokontrolera AT90S2313 przewidziano specjalny wektor przerwania przydzielony dla komparatora analogowego, równy \$00A.



Rys. 9.1. Schemat blokowy komparatora analogowego

Przerwanie jest generowane na wyjściu komparatora, przy czym zgłoszenie przerwania może następować po zmianie stanu na jego wyjściu z niskiego na wysoki, z wysokiego na niski, lub dowolną zmianą stanu (przerzutem). Komparator może być programowo wyłączany w celu zmniejszenia poboru prądu zasilającego przez mikrokontroler.

ACSR (Analog Comparator Control and Status Register) – rejestr sterujący i statusu komparatora analogowego – \$08

Bit	7	6	5	4	3	2	1	0	
\$08 (\$28)	ACD	—	ACO	ACI	ACIE	ACIC	ACIS1	ACIS0	ACSR
Odczyt/Zapis	R/W	R	R	R/W	R/W	R/W	R/W	R/W	
Wartość początkowa	0	0	N/A	0	0	0	0	0	

R – oznacza odczyt, R/W – oznacza odczyt/zapis, N/A – nieustalona

B7 – ACD (Analog Comparator Disable): bit włączający/wyłączający komparator analogowy.

Ustawienie („1”) bitu ACD powoduje wyłączenie komparatora. Bit ten może być ustawiony w dowolnym momencie. Wyłączenie komparatora wyraźnie zmniejsza pobór prądu zasilającego w trybie normalnym i *Idle*. W chwili zmiany bitu ACD musi być wyłączone przerwanie komparatora (*Analog Comparator Interrupt*) przez wyzerowanie bitu ACIE w rejestrze ACSR. Jeśli się nie zapewni tego warunku, może wystąpić niekontrolowane wygenerowanie przerwania.

B6 – zarezerwowany.

Ten bit jest zarezerwowany w układzie AT90S2313 i zawsze odczytywany jako zero.

B5 – ACO (Analog Comparator Output): wyjście komparatora analogowego.

Bit ACO jest bezpośrednim odzwierciedleniem stanu wyjścia komparatora analogowego.

B4 – ACI (Analog Comparator Interrupt): flaga przerwania od komparatora.

Bit ACI jest ustawiany („1”) w momencie wystąpienia zdarzenia na wyjściu komparatora analogowego zdefiniowanego za pomocą bitów ACIS1 i ACIS0 jako zgłoszenie przerwania. Procedura obsługi przerwania od komparatora (*Analog Comparator Interrupt*) jest wykonywana, jeśli bit ACIE jest ustawiony („1”) i włączone jest globalne zezwolenie na przerwanie (bit I w rejestrze SREG jest ustawiony). Flaga ACI jest zerowana sprzętowo w chwili skoku do wektora przerwania od komparatora. Może być także zerowana poprzez wpisanie jedynki („1”) do tego bitu. Jeśli w programie pozostałe bity

rejestru UCSR są modyfikowane rozkazami SBI lub CBI, to bit ACI powinien być wyzerowany przed wykonaniem tych modyfikacji.

B3 – ACIE (Analog Comparator Interrupt Enable): flaga zezwolenia na przerwanie od komparatora.

Ustawienie bitu ACIE („1”) przy włączonym globalnym zezwoleniu na przerwanie (bit I w rejestrze SREG jest ustawiony), zezwala na generowanie przerwania od komparatora. Jeśli ACIE = 0, to przerwania komparatora są zablokowane.

B2 – ACIC (Analog Comparator Input Capture Enable): bit zezwolenia na wyzwalanie funkcji przechwytywania Timera/Licznika1 przez komparator analogowy.

Ustawienie („1”) bitu ACIC powoduje, że funkcja przechwytywania Timera/Licznika1 jest wyzwalana przez komparator analogowy. Wyjście komparatora zostaje wówczas dołączone bezpośrednio do układu sterującego funkcją przechwytywania. W tym trybie można normalnie wykorzystywać eliminator szumu i wybierać zboczne generujące przerwanie *Timer/Counter1 Input Capture Interrupt* (patrz opis Licznika/Timera1). Jeśli bit ACIC jest wyzerowany („0”), opisanego wyżej połączenia nie ma. Aby wyzwalane komparetorem przerwanie *Timer/Counter1 Input Capture Interrupt* mogło być wygenerowane, bit TICIE1 w rejestrze TIMSK musi być ustawiony („1”).

B1, B0 – ACIS1, ACIS0 (Analog Comparator Interrupt Mode Select): bity wyboru warunku zgłaszania przerwania przez komparator.

Powyższe bity określają które zdarzenie na wyjściu komparatora analogowego powoduje zgłoszenie przerwania. Możliwości są przedstawione w tabelicy 9.1.

Przykład wykorzystania komparatora analogowego jako źródła wyzwalania funkcji przechwytywania Timera/Licznika1 zamieszczono w rozdziale 14.1 (ćwiczenie 6).

Tab. 9.1 Tryby wyzwalania przerwania komparatora ustawiane bitami ACIS1, ACIS0⁽¹⁾

ACIS1	ACIS0	Tryb przerwania
0	0	Przerwanie komparatora wyzwalane zmianą stanu na jego wyjściu
0	1	Zarezerwowane
1	0	Przerwanie komparatora wyzwalane opadającym zboczem sygnału wyjściowego komparatora
1	1	Przerwanie komparatora wyzwalane narastającym zboczem sygnału wyjściowego komparatora

Uwaga:

⁽¹⁾ Podczas zmian bitów ACIS1 i ACIS0 przerwanie od komparatora analogowego powinno być zablokowane przez wyzerowanie bitu ACIE w rejestrze ACSR. W przeciwnym razie może wystąpić niekontrolowane zgłoszenie przerwania komparatora.

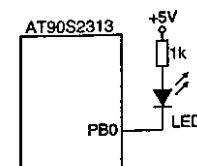
10. Porty wejścia-wyjścia (I/O)

Do czego może być wykorzystany mikrokontroler? Jak sama nazwa wskazuje do sterowania, a jeśli tak, to powinien być wyposażony w odpowiednią liczbę uniwersalnych wejść i wyjść. Uwzględnienie tego postulatu może niekorzystnie wpływać na wielkość obudowy mikrokontrolera. Aby zminimalizować liczbę wyprowadzeń układu scalonego, wymyślono wielofunkcyjne porty we/wy. W zależności od ustawień bitów konfiguracyjnych mogą one pełnić rolę portów we/wy ogólnego przeznaczenia wykorzystywanych do sterowania urządzeniami zewnętrznymi lub mogą współpracować z wewnętrznymi blokami funkcjonalnymi. Ze względu na budowę wewnętrzną CPU, a w szczególności długość rejestrów i szerokość szyny danych porty te mają najczęściej budowę 8-bitową. Niektóre z nich mogą nawet pracować jako analogowe lub cyfrowe. W zależności od typu mikrokontrolera jest w nim zaimplementowana mniejsza lub większa liczba portów (patrz zestawienie parametrów w dodatku A). W układzie AT90S2313 użytkownik ma do dyspozycji 8-bitowy port we/wy – PORTB i 7-bitowy PORTD. Kierunek każdego wyprowadzenia dowolnego portu może ustalany niezależnie od pozostałych i zmieniany w dowolnym momencie bez wpływu na pozostałe linie portów. Stosowną konfigurację można zrealizować przy użyciu rozkazów SBI lub CBI. Te same rozkazy mogą służyć do ustawienia odpowiedniego poziomu na porcie wyjściowym lub włączenia/wyłączenia podciągania *pull-up* linii portu wejściowego.

10.1. Budowa portu B

Jak już wiemy, PORTB jest 8-bitowym, dwukierunkowym portem we/wy. Jest on dostępny poprzez 3 lokacje w przestrzeni we/wy mikrokontrolera. Dlaczego jeden port zajmuje aż trzy adresy? Praktycznie jeden adres wystarczy, ale rozwiązania zastosowane w AVR-ach pozwalają na bardziej efektywne konfigurowanie i wykorzystywanie portów.

Jak się to zatem odbywa? Pierwsza lokacja to rejestr danych (*Data Register*) PORTB – adres \$18 w przestrzeni we/wy mikrokontrolera lub \$38 w obszarze pamięci wewnętrznej. Następna, to rejestr konfiguracji kierunku (*Data Direction Register*) DDRB mający odpowiednie adresy: \$17 (\$37) i ostatni – rejestr wejściowy (*Input Pins*) PINB o adresach \$16 (\$36). Rejestr PINB może być tylko odczytywany, PORTB i DDRB mogą być zarówno czytane jak



Rys. 10.1. Przykład bezpośredniego sterowania diodą LED przez mikrokontroler AVR

i zapisywane. Ważną cechą portu B jest to, że każda jego linia ma indywidualnie wybierany rezystor podciągający do „plusa” zasilania (*pull-up*). Do wszystkich wyprowadzeń PORTB można dołączać diody LED jedynie poprzez rezystor ograniczający. Dopuszczalna obciążalność każdego wyprowadzenia PORTB wynosi 20 mA w stanie niskim (dla $V_{CC} = 5\text{ V}$), trzeba jednak pamiętać o tym, że suma wszystkich prądów z portów wyjściowych nie może przekroczyć wartości 200 mA. Bezpośrednio sterowane diody LED powinny być włączone poprzez rezystor ograniczający pomiędzy plusem zasilania a portem wyjściowym (przykładowe sterowanie diodą LED przedstawiono na rysunku 10.1). Zaświecenie diody następuje po ustawieniu na wyjściu portu niskiego poziomu logicznego.



Jeśli wyprowadzenia PB0 do PB7 są wykorzystywane jako wejścia i są zewnętrznie podciągane do dołu, to stanowią źródło prądu wypływającego z portu, jeśli tylko są uaktywnione wewnętrzne rezystory podciągające.

Jak już wiemy większość linii portów ogólnego przeznaczenia pełni dodatkowe funkcje wykorzystywane do obsługi urządzeń peryferyjnych. Znaczenie funkcji alternatywnych portu B przedstawiono w tabelicy 10.1.

Tab. 10.1. Alternatywne funkcje portu B

Wyprowadzenie portu	Funkcja alternatywna
PB0	AIN0 (wejście nieodwracające komparatora analogowego)
PB1	AIN1 (wejście odwracające komparatora analogowego)
PB3	OC1 (wyjście porównania dla Timera/Licznika1)
PB5	MOSI (wejście danych dla programowania szeregowego)
PB6	MISO (wyjście danych dla programowania szeregowego)
PB7	SCK (wejście zegarowe dla programowania szeregowego)

PORTB (Port B Data Register) – rejestr danych portu B – \$18

Bit	7	6	5	4	3	2	1	0	
\$18 (\$38)	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0	PORTB
Odczyt/Zapis	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Wartość początkowa	0	0	0	0	0	0	0	0	
R/W – oznacza odczyt/zapis									

Rejestr PORTB może być zapisywany i odczytywany. Zawiera dane we/wy.

DDRB (Port B Data Direction Register) – rejestr kierunku portu B – \$17

Bit	7	6	5	4	3	2	1	0	
\$17 (\$37)	DDB7	DDB6	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0	DDRB
Odczyt/Zapis	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Wartość początkowa	0	0	0	0	0	0	0	0	
R/W – oznacza odczyt/zapis									

Rejestr DDRB określa typ każdej linii portu B. Linie mogą być skonfigurowane jako wejściowe lub wyjściowe. Dodatkowo ustawia się opcje związane z podciąganiem. Szczegóły są podane w dalszej części niniejszego rozdziału.

PINB (Port B Input Pin Address) – rejestr wejściowy portu B – \$16

Bit	7	6	5	4	3	2	1	0	
\$16 (\$36)	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0	PINB
Odczyt/Zapis	R	R	R	R	R	R	R	R	
Wartość początkowa	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	
R – oznacza odczyt									

Rejestr PINB może być tylko odczytywany. Jego zawartość odzwierciedla bezpośrednio stan wyprowadzeń mikrokontrolera przypisanych do portu B. Fizycznie nie jest to więc rejestr. Inaczej jest podczas bezpośredniego odczytu PORTB. Wówczas czytany jest stan zatrząsków portu B (*Port B Data Latch*).

10.1.1. Port B jako cyfrowy port we/wy ogólnego przeznaczenia

Jeśli poszczególne linie portu B są wykorzystywane jako wyprowadzenia cyfrowe ogólnego przeznaczenia, to są wzajemnie równoważne. Przyjmijmy oznaczenie P_{Bn} do określenia *n*-tego wyprowadzenia portu B. Stosując dalej tę konwencję, bit DDB_n w rejestrze DDRB będzie określał kierunek wyprowadzenia P_{Bn}. Jeśli DDB_n będzie ustawiony („1”), to wyprowadzenie P_{Bn} będzie skonfigurowane jako wyjściowe. Jeśli DDB_n będzie wyzerowany („0”), wyprowadzenie P_{Bn} będzie skonfigurowane jako wejściowe. Wyprowadzenie P_{Bn} skonfigurowane jako wejściowe będzie się zachowywało różnie, w zależności od stanu PORTB_n. Jeśli bit PORTB_n będzie ustawiony („1”), to linia wejściowa P_{Bn} bę-

Tab. 10.2. Wpływ ustawienia bitów DDB_n i PORTB_n na wyprowadzenia portu B

DDB _n	PORTB _n	I/O	Podciąganie pull-up	Uwagi
0	0	Wejście	Brak	Wysoka impedancja (<i>high-Z</i>)
0	1	Wejście	Jest	P _{Bn} jest źródłem prądu, jeśli jest zewnętrznie podciągany do masy zasilania (<i>pull-low</i>)
1	0	Wyjście	Brak	Wyjście typu <i>push-pull</i> stan niski
1	1	Wyjście	Brak	Wyjście typu <i>push-pull</i> stan wysoki

dzie miała uaktywniony MOS-owy rezystor podciągający (*pull-up*). Taka konfiguracja świetnie nadaje się do obsługi np. zewnętrznej klawiatury, różnego rodzaju przełączników. Trzeba jednak pamiętać o ograniczonych możliwościach wewnętrznego podciągania. Do wyłączenia wewnętrznego rezystora podciągającego należy wyzerować („0”) bit PORTB_n lub skonfigurować linię P_{Bn} jako wyjściową. Po zerowaniu mikrokontrolera port B jest ustawiany w stan wysokiej impedancji, nawet wówczas, gdy nie jest aktywny zegar systemowy.

Konfigurowanie portów w języku C było przedstawione „przy okazji” w przykładzie 9.1. Poniżej można zobaczyć, jak robi się to w assemblerze.

Przykład 10.1. Konfigurowanie portów mikrokontrolera w assemblerze. W charakterze ćwiczenia, dla lepszego zrozumienia programu warto rozpiszać poszczególne bity w rejestrach PORTB i DDRB

```
.include "2313def.inc"
.def temp=r17

...
ldi temp,$af
out ddrb,temp ;PB0 do PB3 oraz PB5 i PB7 - wyjścia
;pozostałe - wejścia

ldi temp,0xcf
out portb,temp ;PB0 do PB3 i PB7 wyjście w stanie wysokim
;PB5 wyjście w stanie niskim
;PB4 wejście w stanie High-Z
;PB6 wejście podciągnięte do plusa

...
```

10.1.2. Funkcje alternatywne portu B

Wyprowadzenia portu B mogą spełniać także następujące funkcje alternatywne:

B7 – SCK: wejście zegarowe dla programowania szeregowego.

Do tego wejścia jest doprowadzany zewnętrzny przebieg zegarowy taktujący transferem danych podczas szeregowego programowania pamięci mikrokontrolera.

B6 – MISO: wyjście danych dla programowania szeregowego.

Z tego wyjścia są przekazywane informacje z mikrokontrolera do programatora podczas szeregowego programowania pamięci.

B5 – MOSI: wejście danych dla programowania szeregowego.

Do tego wejścia są przekazywane informacje z programatora do mikrokontrolera podczas szeregowego programowania pamięci.

B3 – OC1: wyjście przechwytywania.

Wyjście OC1 jest wykorzystywane, gdy jest aktywna funkcja porównywania Timera/Licznika1. Pojawia się na nim stan wysoki, jeśli stan Timera/Licznika1 zrówna się z rejestrem OCR1A (tzn. gdy TCNT1H=OCR1AH

i $TCNT1L=OCR1AL$). Wyprowadzenie PB3 powinno być w takim przypadku skonfigurowane jako wyjście. Więcej informacji podano w rozdziale 5.2.

B1 – AIN1: wejście odwracające komparatora analogowego.

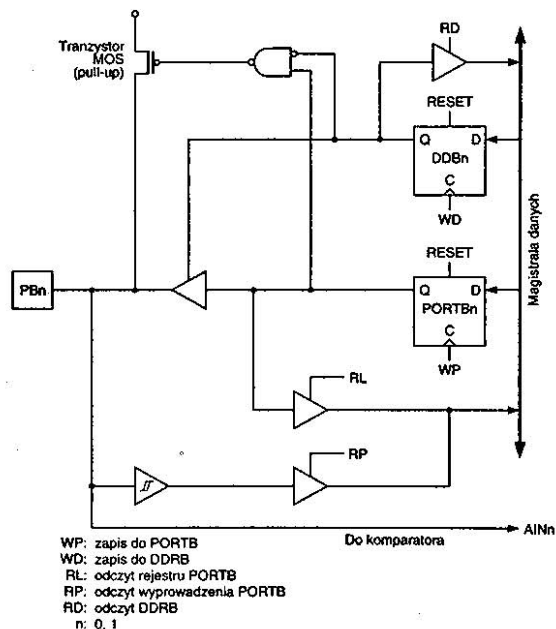
Jeśli wyprowadzenie to jest skonfigurowane jako wejście z wyłączonym podciąganiem (DDB1 jest wyzerowane („0”) i PB1 jest wyzerowane („0”), wyprowadzenie AIN1 jest bezpośrednio dołączone do odwracającego wejścia komparatora analogowego zawartego w strukturze mikrokontrolera.

B0 – AIN0: wejście nieodwracające komparatora analogowego.

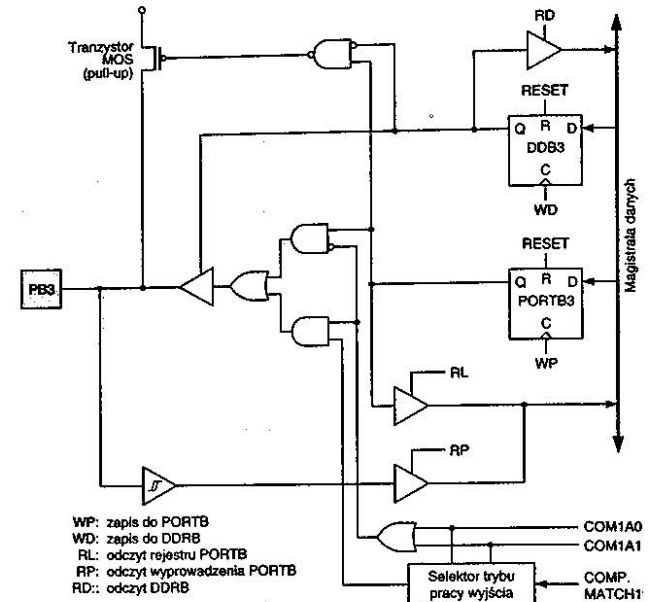
Jeśli wyprowadzenie to jest skonfigurowane jako wejście z wyłączonym podciąganiem (DDB0 jest wyzerowane („0”) i PB0 jest wyzerowane („0”), to wyprowadzenie AIN0 jest bezpośrednio dołączone do nieodwracającego wejścia komparatora analogowego zawartego w strukturze mikrokontrolera.

10.1.3. Budowa linii portu B

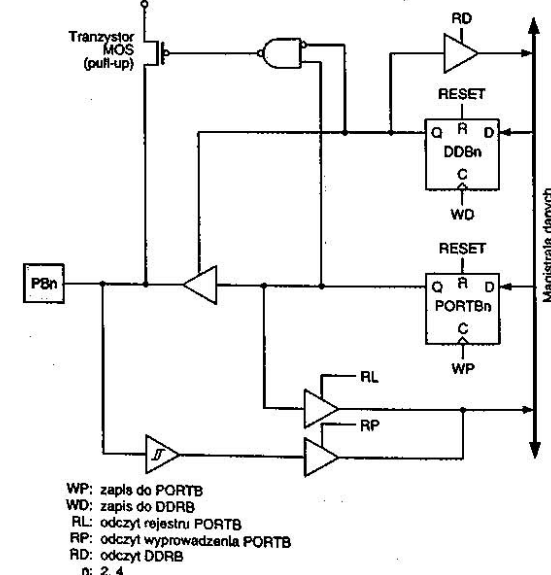
Na rysunkach 10.2 do 10.7 przedstawiono schematy poszczególnych linii portu B. Ich budowa wewnętrzna różni się ze względu na funkcje alternatywne. Wszystkie wyprowadzenia portu są synchronizowane, jednakże na rysunkach nie pokazano zatrząsków synchronizujących.



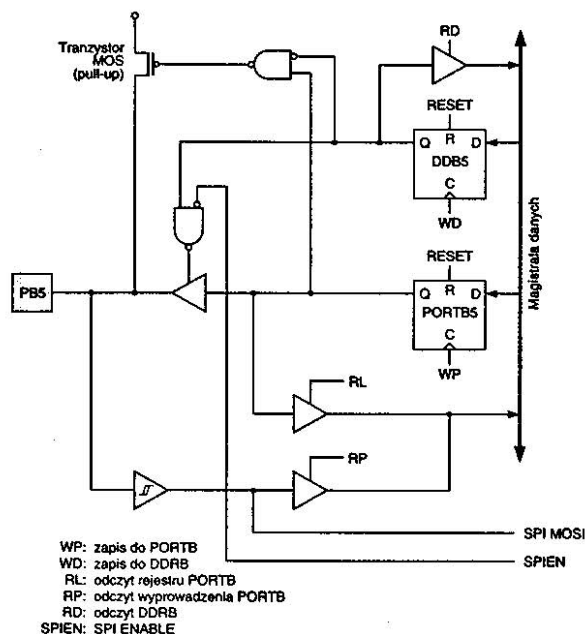
Rys. 10.2. Schemat linii 0 i 1 portu B



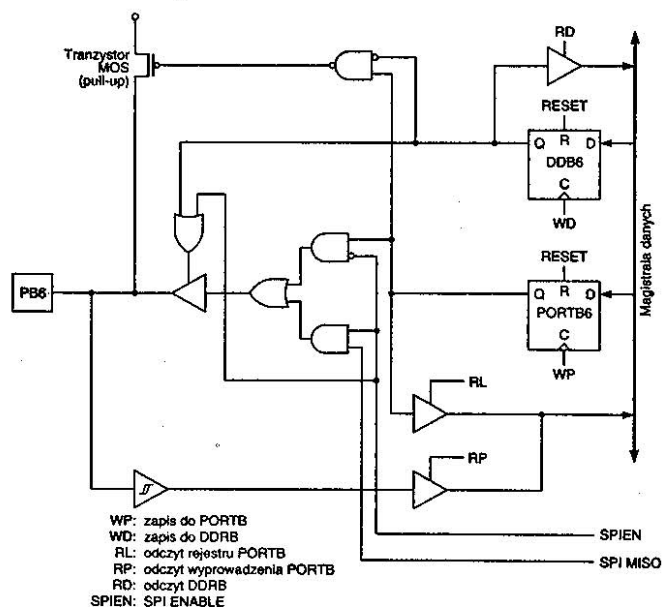
Rys. 10.3. Schemat linii 3 portu B



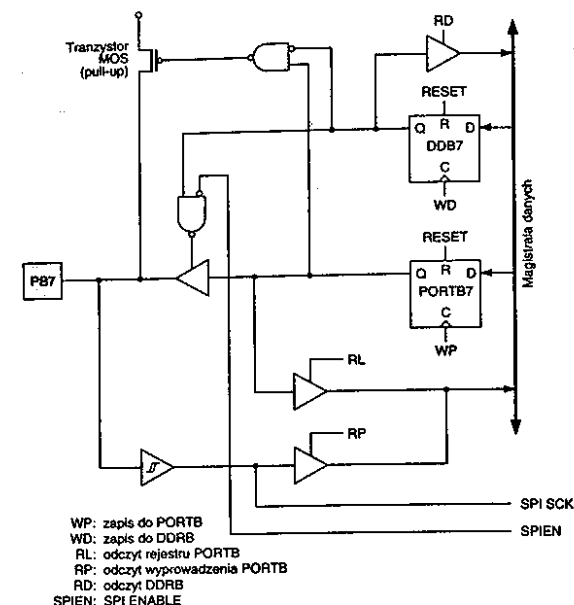
Rys. 10.4. Schemat linii 2 i 4 portu B



Rys. 10.5. Schemat linii 5 portu B



Rys. 10.6. Schemat linii 6 portu B



Rys. 10.7. Schemat linii 7 portu B

10.2. Budowa portu D

PORTD zawiera 7 dwukierunkowych linii we/wy. Jest on dostępny poprzez 3 lokacje w przestrzeni we/wy mikrokontrolera. Pierwsza lokacja to rejestr danych (Data Register) PORTD – adres \$12 w przestrzeni we/wy mikrokontrolera lub \$32 w obszarze pamięci wewnętrznej. Następna, to rejestr konfiguracji kierunku (Data Direction Register) DDRD mający odpowiednie adresy: \$11 (\$31) i ostatni – rejestr wejściowy (Input Pins) PIND o adresach \$10 (\$30). Rejestr PIND może być tylko odczytywany, PORTD i DDRD mogą być zarówno czytane jak i zapisywane. Każda linia portu D ma indywidualnie wybierany rezystor podciągający do góry (pull-up). Do wszystkich wyprowadzeń PORTD można dołączać diody LED, jedynie poprzez rezystor ograniczający. Dopuszczalna obciążalność każdego wyprowadzenia PORTD wynosi 20 mA w stanie niskim (dla $V_{CC} = 5\text{ V}$), trzeba jednak pamiętać o tym, że suma wszystkich prądów z portów wyjściowych nie może przekroczyć wartości 200 mA. Bezpośrednio sterowane diody LED powinny być włączone poprzez rezystor ograni-



Jeśli wyprowadzenia PD0 do PD6 są wykorzystywane jako wejścia i są zewnętrznie podciągane do dołu, to stanowią źródło prądu wypływającego z portu, jeśli tylko są uaktywnione wewnętrzne rezystory podciągające.

Tab. 10.3. Funkcje alternatywne portu D

Wyprowadzenie portu	Funkcja alternatywna
PD0	RXD (wejście danych odbiornika UART)
PD1	TXD (wyjście danych nadajnika UART)
PD2	INT0 (wejście zgłoszenia przerwania zewnętrznego INT0)
PD3	INT1 (wejście zgłoszenia przerwania zewnętrznego INT1)
PD4	TO (wejście zewnętrznego sygnału dla licznika TC0)
PD6	T1 (wejście zewnętrznego sygnału dla licznika TC1)
PD7	ICP (wejście przechwytywania dla timera/licznika TC1)

czający pomiędzy plusem zasilania a portem wyjściowym (przykład sterowania diodami LED przedstawiono na rysunku 10.1). Dioda świeci po wystereowaniu linii portu niskim poziomem logicznym.

Wszystkie linie portu D pełnią dodatkowe funkcje wykorzystywane do obsługi urządzeń peryferyjnych, które zestawiono w tablicy 10.3.

W przypadku wykorzystywania poszczególnych linii portu D do obsługi funkcji alternatywnych, odpowiadające im bity konfiguracyjne w rejestrach DDRD i PORTD powinny być ustawione zgodnie z opisem tych funkcji.

PORTD (Port D Data Register) – rejestr danych portu D – \$12

Bit	7	6	5	4	3	2	1	0	
\$12 (\$32)	–	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0	PORTD
Odczyt/Zapis	R	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Wartość początkowa	0	0	0	0	0	0	0	0	

R – oznacza odczyt, R/W – oznacza odczyt/zapis

Rejestr PORTD może być zapisywany i odczytywany. Zawiera dane we/wy.

DDRD (Port D Data Direction Register) – rejestr kierunku portu D – \$11

Bit	7	6	5	4	3	2	1	0	
\$11 (\$31)	–	DD6	DD5	DD4	DD3	DD2	DD1	DD0	DDRD
Odczyt/Zapis	R	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Wartość początkowa	0	0	0	0	0	0	0	0	

R – oznacza odczyt, R/W – oznacza odczyt/zapis

Rejestr DDRD określa typ każdej linii portu D. Linie mogą być skonfigurowane jako wejściowe lub wyjściowe. Dodatkowo ustawia się opcje związane z podciąganiem. Szczegóły są podane w dalszej części rozdziału.

PIND (Port D Input Pins Address) – rejestr wejściowy portu D – \$10

Bit	7	6	5	4	3	2	1	0	
\$10 (\$30)	–	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0	PIND
Odczyt/Zapis	R	R	R	R	R	R	R	R	
Wartość początkowa	0	N/A	N/A	N/A	N/A	N/A	N/A	N/A	

R – oznacza odczyt, N/A – nieustalona

Rejestr PIND może być tylko odczytywany. Jego zawartość odzwierciedla bezpośrednio stan wyprowadzeń mikrokontrolera przypisanych do portu D. Fizycznie nie jest to więc rejestr. Inaczej jest podczas bezpośredniego odczytu PORTD. Wówczas czytany jest stan zatrząsków portu D (*Port D Data Latch*).

10.2.1. Port D jako cyfrowy port we/wy ogólnego przeznaczenia

Jeśli poszczególne linie portu D są wykorzystywane jako wyprowadzenia cyfrowe ogólnego przeznaczenia, to są wzajemnie równoważne. Przyjmijmy oznaczenie PDn do określenia n-tego wyprowadzenia portu D. Stosując dalej tę konwencję, bit DDn w rejestrze DDRD będzie określał kierunek wyprowadzenia PDn. Jeśli DDn będzie ustawiony („1”), to wyprowadzenie PDn będzie skonfigurowane jako wyjściowe. Jeśli DDn będzie wyzerowany („0”), to wyprowadzenie PDn będzie skonfigurowane jako wejściowe. Wyprowadzenie PDn skonfigurowane jako wejściowe będzie się zachowywało różnie, w zależności od stanu PORTDn. Jeśli bit PORTDn będzie ustawiony („1”), to linia wejściowa PDn będzie miała uaktywniony MOS-owy rezystor podciągający (*pull-up*). Taka konfiguracja świetnie nadaje się do obsługi np. zewnętrznej klawiatury, różnego rodzaju przełączników. Trzeba jednak pamiętać o ograniczonych możliwościach wewnętrznego podciągania. Do wyłączenia wewnętrznego rezystora podciągającego należy wyzerować („0”) bit PORTDn lub skonfigurować linię PDn jako wyjściową. Po zerowaniu mikrokontrolera port D jest ustawiany w stan wysokiej impedancji, nawet wówczas, gdy nie jest aktywny zegar systemowy.

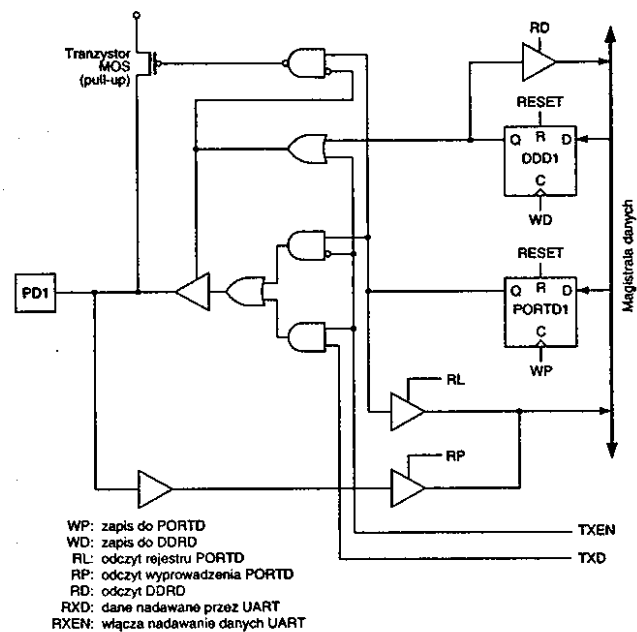
10.2.2. Funkcje alternatywne portu D

Wyprowadzenie portu D mogą spełniać także następujące funkcje alternatywne:

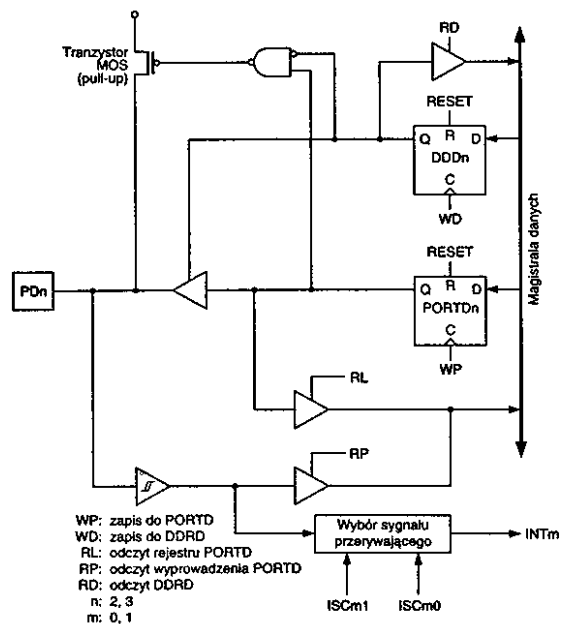
B6 – ICP: wejście przechwytywania dla Timera/Licznika1.

Wejście ICP jest wykorzystywane, gdy jest aktywna funkcja przechwytywania Timera/Licznika1. Pojawienie się na tym wyprowadzeniu zbocza określonego ustawieniem bitu ICES1 w rejestrze TCCR1B powoduje przepisanie stanu Timera/Licznika1 do rejestru ICR1 (tzn. ICR1H=TCNT1H i ICR1L=TCNT1L). Wyprowadzenie PD6 powinno być w takim przypadku skonfigurowane jako wejście. Więcej informacji podano w rozdziale 5.2.

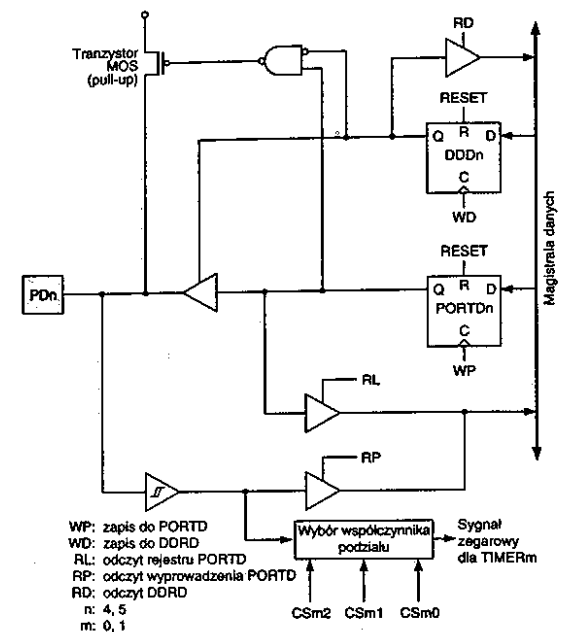
Rys. 10.8. Schemat linii 0 portu D



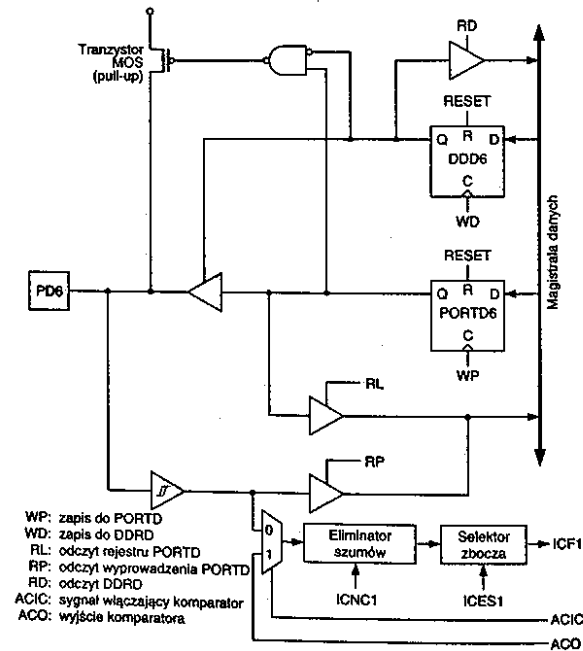
Rys. 10.9. Schemat linii 1 portu D



Rys. 10.10. Schemat linii 2 i 3 portu D



Rys. 10.11. Schemat linii 4 i PD5 portu D

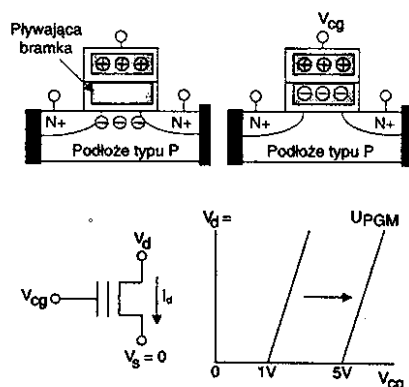


Rys. 10.12. Schemat linii 6 portu D

11. Pamięci nieulotne w mikrokontrolerach AVR

Zapewne część Czytelników pamięta pierwsze wersje mikrokontrolerów i pamięci stosowanych w systemach mikroprocesorowych. Najpowszechniejszym rodzajem pamięci były wówczas układy typu UVEPROM (*Ultra Violet EPROM*). Można je było wielokrotnie zapisywać w specjalnym programatorze i kasować za pomocą promieni ultrafioletowych. Były więc w użyciu niezbyt wygodne, a i ich cena nie była niska. Pomijając pamięci OTP (*One Time Programmable* – programowane jednorazowo), nie było jednak specjalnego wyboru. Cena pamięci OTP była na ogół znacząco niższa od UVEPROM, lecz ze względu na możliwość jednorazowego zapisu zupełnie nie nadawały się one do prac konstrukcyjnych. Wadą UVEPROM-ów była konieczność stosowania niełatwego do zdobycia kasownika (zawierającego lampę ultrafioletową emitującą promieniowanie o odpowiedniej długości fali) i długi czas kasowania, na ogół nie krótszy niż kilkanaście do kilkudziesięciu minut.

Dzięki spopularyzowaniu pamięci Flash wszystkie te zmartwienia praktycznie poszły w zapomnienie. Zdominowały one większość obecnie produkowanych mikrokontrolerów, a także nieulotnych pamięci zewnętrznych. Technologia ta umożliwia wielokrotny zapis i kasowanie na drodze elektronicznej. Informacja zapisana w pojedynczej komórce pamięci jest reprezentowana przez ilość ładunku wstrzykniętego do obszaru tzw. pływającej bramki zmodyfikowanego nieco – w stosunku do rozwiązań typowych – tranzystora MOS (rysunek 11.1). Wstrzyknięty ładunek jest zatrzymywany



Rys. 11.1. Zasada działania komórki pamięciowej w technologii Flash

na stałe wskutek pułapki potencjałowej. Mówiąc prościej „nie ma siły”, aby uwolnić się od otaczającej go bariery potencjału. Doprowadzając dostatecznie wysokie i odpowiednio spolaryzowane napięcie do poszczególnych elektrod tranzystora można jednak spowodować odpływ ładunku z obszaru pływającej bramki, tym samym skasować pamięć (lub zaprogramować, zależy od przyjętej konwencji). Na świecie – jak wiemy – nie ma rzeczy idealnych. Zawsze podczas kasowania może w obszarze pływającej bramki pozostawać pewien szczątkowy ładunek. Z drugiej strony, mimo bariery potencjałów pojedyncze ładunki mogą w pewnych okolicznościach opuszczać obszar pływającej bramki. Skumulowanie tych efektów spowoduje, że po pierwsze kolejny cykl kasowania/programowania pamięci Flash może się nie udać, po drugie zaprogramowana pamięć rozprogramuje się sama po pewnym czasie. Liczba cykli kasowania/zapisu oraz czas utrzymywania wartości pamięci są na szczęście wystarczające do praktycznych zastosowań.

Trzeba tu wspomnieć o jeszcze innym rodzaju pamięci nieulotnej jakim jest pamięć EEPROM. Ten rodzaj pamięci również pozwala na zachowanie danych po wyłączeniu zasilania i nie wymaga dodatkowych urządzeń do kasowania. W mikrokontrolerach, w tym w AVR-ach, pamięć EEPROM jest wykorzystywana jako pamięć danych, które powinny być zachowane po wyłączeniu zasilania. Typowe jej zastosowania to przechowywanie parametrów konfiguracyjnych, opcji programu itp.

Reasumując: w mikrokontrolerach AVR zastosowane są jednocześnie dwa rodzaje pamięci nieulotnych. Pamięć Flash – pełniąca rolę pamięci programu i pamięć EEPROM – służąca do przechowywania parametrów programu, a także pewnych informacji dotyczących konfiguracji samego mikrokontrolera. Programowanie obu rodzajów pamięci może się odbywać bezpośrednio w systemie (bez konieczności przenoszenia mikrokontrolera do programatora).

11.1. Bity zabezpieczające pamięć programu i danych

W mikrokontrolerze AT90S2313 zastosowano dwa bity zabezpieczające (*Lock Bits*), za pomocą których ustawia się stopień ochrony danych zapisanych w pamięci nieulotnej mikrokontrolera, czyli w pamięci Flash oraz EEPROM. Podczas programowania pamięci mikrokontrolera bity zabezpieczające można pozostawić niezaprogramowane („1”) lub zaprogramować je

Tab. 11.1. Ochrona pamięci mikrokontrolera za pomocą bitów zabezpieczających

Tryb	Bity zabezpieczające		Typ zabezpieczenia
	LB1	LB2	
1	1	1	Pamięć niezabezpieczona
2	0	1	Zapis pamięci Flash i EEPROM zablokowany ⁽¹⁾
3	0	0	Zapis i weryfikacja pamięci Flash i EEPROM zablokowana

Uwaga:

⁽¹⁾ Podczas programowania równoległego zostaje zablokowana również możliwość późniejszego programowania bitów konfiguracyjnych. Jeśli zachodzi taka konieczność, to bity konfiguracyjne (*Fuse Bits*) powinny być zaprogramowane przed bitami zabezpieczającymi (*Lock Bits*).

(„0”). Znaczenie poszczególnych kombinacji zestawiono w **tablicy 11.1**. Kasowanie bitów zabezpieczających może być przeprowadzone tylko podczas kasowania całej pamięci. Oprócz bitów zabezpieczających pamięć programu i nieulotną pamięć danych występują jeszcze bity nazywane w dokumentacji źródłowej *Fuse Bits*, które z racji pełnionych przez nie funkcji powinno się raczej nazywać bitami konfiguracyjnymi.

11.2. Bity konfiguracyjne

W mikrokontrolerze AT90S2313 zastosowano dwa bity konfiguracyjne (*Fuse Bits*): SPIEN i FSTRT. Ich funkcje są następujące:

- Jeśli bit konfiguracyjny SPIEN jest zaprogramowany („0”), to możliwe jest szeregowe programowanie układu. Bit konfiguracyjny SPIEN jest domyślnie programowany („0”).
- Zaprogramowanie bitu konfiguracyjnego FSTRT („0”) powoduje ustawienie krótkiego czasu restartu (*start-up*). Na **rysunku 4.22** przedstawiono blok zerowania mikrokontrolera. Jak na nim widać, sygnał zerowania przerzutnika generującego impuls zerujący jest pobierany z wyjścia Q7 lub Q13 licznika. Zaprogramowanie tego bitu konfiguracyjnego powoduje uaktywnienie wyjścia Q7, tym samym skrócenie czasu restartu z 16 ms do ok. 0,28 ms (typowo). Domyślną wartością bitu konfiguracyjnego FSTRT jest „1” (bit konfiguracyjny niezaprogramowany), jednakże układ może być dostarczany na żądanie z wartością domyślną „0”.



Ustawianie bitów konfiguracyjnych nie jest możliwe podczas programowania szeregowego.

11.3. Sygnatury

Wszystkie mikrokontrolery z rodziny AVR mają 3-bajtową sygnaturę identyfikującą typ układu. Sygnatura może być odczytywana zarówno podczas programowania równoległego, jak i szeregowego. Dzięki temu, podczas programowania można w wielu sytuacjach uniknąć błędu mogącego uszkodzić układ. Rozmieszczenie sygnatur w mikrokontrolerze AT90S2313 wygląda następująco:

1. \$000: - \$1E (identyfikator produktu Atmela),
2. \$001: - \$91 (identyfikator 2 kB pamięci Flash),
3. \$002: - \$01 (identyfikator układu AT90S2313, jeśli sygnatura \$001 ma wartość \$91).

Jeśli bity zabezpieczające są ustawione w trybie 3 (oba zaprogramowane), to odczyt sygnatur jest zablokowany w trybie szeregowym. Zwracane będą w tym przypadku wartości \$00, \$01 i \$02.

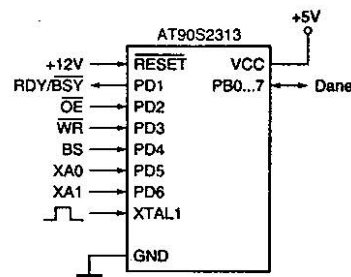
11.4. Programowanie pamięci Flash i EEPROM

W układzie AT90S2313 zastosowano 2 kB pamięć Flash i 128 B pamięć danych typu EEPROM (obie programowane w systemie). Układy są dostarczane z pamięciami fabrycznie wykasowanymi (zawartość każdej komórki = \$FF) i gotowymi do zaprogramowania. W strukturze mikrokontrolera znajdują się wszystkie wymagane do programowania bloki funkcjonalne, w tym pompy ładunkowe wytwarzające niezbędne napięcia programujące o wartości większej od napięcia zasilającego. Są one wykorzystywane podczas programowania ISP, natomiast programowanie równoległe wymaga dostarczenia z zewnątrz napięcia +12 V. Zarówno pamięć programu, jak i EEPROM są programowane bajt po bajcie, niezależnie od sposobu programowania. Wartości napięcia zasilającego mikrokontrolera AT90S2313 podczas programowania powinna zawierać się przedziałach: 2,7...6,0 V (podczas programowania szeregowego) lub 4,5...5,5 V (podczas programowania równoległego).

11.4.1. Programowanie równoległe

Programowanie równoległe jest tym trybem programowania, który udostępnia wszystkie możliwe opcje, w tym kasowanie bitów konfiguracyjnych i bitów zabezpieczających. Niestety konieczne jest doprowadzenie napięcia zewnętrznego +12 V i sterowanie niemal wszystkich wyprowadzeń układu.

Z tego względu programowanie równoległe odbywa się w specjalnie skonstruowanym programatorze. Pewne standardowe funkcje i nazwy wyprowadzeń mikrokontrolera ulegają zmianie podczas programowania równoległego. Opisano je w **tablicy 11.2**, a schemat ilustrujący sposób dołączenia sygnałów wykorzystywanych podczas programowania przedstawiono na **rysunku 11.2**. Wyprowadzenia nie ujęte w tej tablicy (i na rysunku) nie są wykorzystywane.



Rys. 11.2. Funkcje wyprowadzeń mikrokontrolera AT90S2313 podczas programowania równoległego

Tab. 11.2. Opis funkcji wyprowadzeń mikrokontrolera AT90S2313 podczas programowania równoległego

Nazwa sygnału	Nazwa wyprowadzenia	Typ wyprowadzenia	Funkcja
RDY/BSY	PD1	Wy	0: układ zajęty (programowanie) 1: układ gotowy do przyjęcia nowego polecenia
OE	PD2	We	Odblokowanie buforów wyjściowych (aktywny stan niski)
WR	PD3	We	Impuls zapisu (aktywny stan niski)
BS	PD4	We	Wybór bajtu („0” – wybierz młodszy bajt; „1” – wybierz starszy bajt)
XA0	PD5	We	XTAL Bit 0 (patrz tablica 11.3)
XA1	PD6	We	XTAL Bit 1
DATA	PB7...0	We/Wy	Dwukierunkowa szyna danych (wyjściowa dla OE w stanie niskim)

Tab. 11.3. Kodowanie operacji za pomocą bitów XA1 i XA0

XA1	XA0	Operacja wykonana po wystąpieniu impulsu na wejściu XTAL1
0	0	Ładuj adres pamięci Flash lub (EEPROM) (starszy lub młodszy bajt, zależnie od ustawienia BS)
0	1	Ładuj daną (starszy lub młodszy bajt, zależnie od ustawienia BS)
1	0	Ładuj polecenie (patrz tablica 11.4)
1	1	Bez akcji, stan Idle

Tab. 11.4. Kody poleceń wykorzystywanych podczas programowania pamięci nieulotnych

Bajt polecenia (postać binarna)	Akcja
1000 0000	Kasowanie pamięci
0100 0000	Zapis bitów konfiguracyjnych
0010 0000	Zapis bitów zabezpieczających
0001 0000	Zapis pamięci Flash
0001 0001	Zapis pamięci EEPROM
0000 1000	Czytanie bajtów sygnatury
0000 0100	Czytanie bitów konfiguracyjnych i bitów zabezpieczających
0000 0010	Czytanie pamięci Flash
0000 0011	Czytanie pamięci EEPROM

Wprowadzenie układu w tryb programowania

Każde rozpoczęcie programowania równoległego musi być poprzedzone wykonaniem poniższej sekwencji działań:

1. Doprowadź napięcie zasilające do wyprowadzeń VCC i GND mikrokontrolera.
2. Ustaw wyprowadzenia $\overline{\text{RESET}}$ i BS w stan niski („0”) i odczekać co najmniej 100 ns.
3. Doprowadź napięcie 11,5...12,5 V do wyprowadzenia $\overline{\text{RESET}}$. Jakakolwiek zmiana stanu na wejściu BS w ciągu 100 ns po ustaleniu się napięcia +12 V na wejściu $\overline{\text{RESET}}$ spowoduje błąd wprowadzenia w stan programowania.

Kasowanie pamięci (Chip Erase)

Polecenie *Chip Erase* powoduje skasowanie pamięci Flash i EEPROM, a także bitów zabezpieczających. Bity zabezpieczające nie ulegają zmianie dopóki pamięci Flash i EEPROM nie zostaną całkowicie skasowane. Bity konfiguracyjne nie ulegają zmianie. Przed wykonaniem reprogramowania pamięci Flash i EEPROM musi być wykonane polecenie *Chip Erase*.

Ładowanie polecenia Chip Erase

1. Ustaw XA1 na „1”, XA0 na „0” w celu umożliwienia załadowania polecenia.
2. Ustaw BS na „0”.
3. Ustaw DATA na „1000 0000” (kod polecenia *Chip Erase*).
4. Doprowadź impuls $\overline{\text{XTAL1}}$ na wejście XTAL1. Nastąpi ładowanie polecenia.
5. Doprowadź impuls $\overline{\text{WR}}$ do wejścia WR o czasie trwania równym 5...15 ms w celu wykasowania pamięci. Fakt zakończenia kasowania nie jest sygnalizowany sygnałem na wyjściu RDY/BUSY.

Programowanie pamięci Flash

A: Ładuj polecenie Write Flash

1. Ustaw XA1 na „1”, XA0 na „0” w celu umożliwienia załadowania polecenia.
2. Ustaw BS na „0”.
3. Ustaw DATA na „0001 0000” (kod polecenia *Write Flash*).
4. Doprowadź impuls $\overline{\text{XTAL1}}$ na wejście XTAL1. Nastąpi załadowanie polecenia.

B: Ładuj starszy bajt adresu

1. Ustaw XA1 na „0”, XA0 na „0” w celu umożliwienia załadowania adresu.
2. Ustaw BS na „1” (wybór starszego bajtu).
3. Ustaw na liniach DATA starszy bajt adresu (\$00...\$03).
4. Doprowadź impuls \square na wejście XTAL1. Nastąpi ładowanie adresu.

C: Ładuj młodszy bajt adresu

1. Ustaw XA1 na „0”, XA0 na „0” w celu umożliwienia załadowania adresu.
2. Ustaw BS na „0” (wybór młodsze bajtu).
3. Ustaw na liniach DATA młodszy bajt adresu (\$00...\$FF).
4. Doprowadź impuls \square na wejście XTAL1. Nastąpi ładowanie adresu.

D: Ładuj młodszy bajt danej

1. Ustaw XA1 na „0”, XA0 na „1” w celu umożliwienia załadowania danej.
2. Ustaw na liniach DATA młodszy bajt danej (\$00...\$FF).
3. Doprowadź impuls \square na wejście XTAL1. Nastąpi ładowanie młodsze bajtu danej.

E: Zapisz młodszy bajt danej

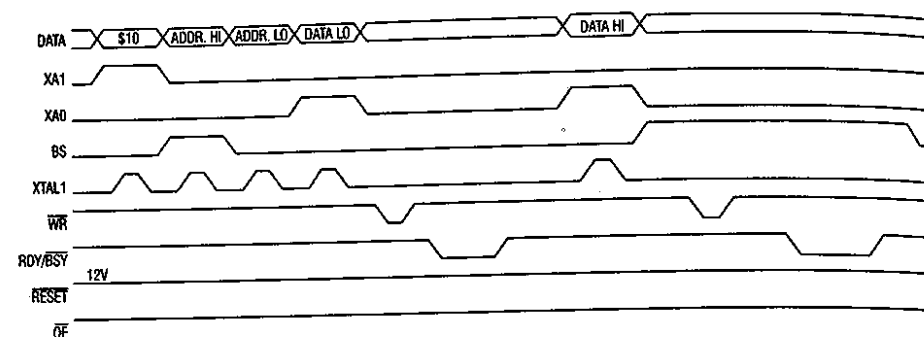
1. Ustaw BS na „0” (wybór młodsze bajtu).
2. Doprowadź impuls \square na wejście XTAL1. Nastąpi rozpoczęcie programowania bajtu danej. Wyjście RDY/BUSY wejdzie w stan niski („0”).
3. Czekaj dopóki na wyjściu RDY/BUSY będzie panował stan niski. Zmiana stanu na wysoki oznacza gotowość do programowania następnego bajtu (rysunek 11.2).

F: Zapisz starszy bajt danej

1. Ustaw XA1 na „0”, XA0 na „1” w celu umożliwienia załadowania danej.
2. Ustaw na liniach DATA starszy bajt danej (\$00...\$FF).
3. Doprowadź impuls \square na wejście XTAL1. Nastąpi ładowanie starsze bajtu danej.

G: Zapisz starszy bajt danej

1. Ustaw BS na „1” (wybór starsze bajtu).
2. Doprowadź impuls \square na wejście XTAL1. Nastąpi rozpoczęcie programowania bajtu danej. Wyjście RDY/BUSY wejdzie w stan niski („0”).

**Rys. 11.3. Przebiegi podczas programowania pamięci Flash**

3. Czekaj dopóki na wyjściu RDY/BUSY będzie panował stan niski. Zmiana stanu na wysoki oznacza gotowość do programowania następnego bajtu (rysunek 11.3).

Załadowane polecenia i adresy pozostają zachowane podczas wykonywania operacji programowania. Efektywne programowanie powinno więc uwzględniać kilka następujących zasad (dotyczą one programowania pamięci Flash i EEPROM oraz czytania pamięci Flash, EEPROM i bajtów sygnatur):

- Kod polecenia powinien być ładowany tylko raz, gdy mamy do czynienia z kolejnym zapisem lub odczytem wielu lokacji pamięci.
- Starszy bajt adresu powinien być ładowany tylko raz przed zmianą 256-słowej strony pamięci Flash.
- Należy unikać zapisu danej o wartości \$FF. Wartość taka jest wprowadzana do pamięci Flash podczas operacji kasowania.

Odczyt pamięci Flash

Algorytm odczytu pamięci Flash jest następujący (zasady ładowania poleceń i adresów są takie same jak w punkcie *Programowanie pamięci Flash*):

1. **A:** Ładuj polecenie „0000 0010” (*Read Flash*).
2. **B:** Ładuj starszy bajt adresu (\$00...\$03).
3. **C:** Ładuj młodszy bajt adresu (\$00...\$FF).
4. Ustaw OE na „0” i BS na „0”. Młodszy bajt słowa danych może być teraz odczytany z linii DATA.
5. Ustaw BS na „1”. Starszy bajt słowa danych może być teraz odczytany z linii DATA.
6. Ustaw OE na „1”.

Programowanie pamięci EEPROM

Algorytm programowania pamięci EEPROM jest następujący (zasady ładowania poleceń i adresów są takie same jak w punkcie *Programowanie pamięci Flash*):

1. **A:** Ładuj polecenie „0001 0001” (*Write EEPROM*).
2. **C:** Ładuj młodszy bajt adresu (\$00...\$7F).
3. **D:** Ładuj młodszy bajt danej (\$00...\$FF).
4. **E:** Zapisz młodszy bajt danej.

Odczyt pamięci EEPROM

Algorytm odczytu pamięci EEPROM jest następujący (zasady ładowania poleceń i adresów są takie same jak w punkcie *Programowanie pamięci Flash*):

1. **A:** Ładuj polecenie „0000 0011” (*Read EEPROM*).
2. **C:** Ładuj młodszy bajt adresu (\$00...\$7F).
3. Ustaw \overline{OE} na „0” i BS na „0”. Dana z pamięci EEPROM może być teraz odczytana z linii DATA.
4. Ustaw \overline{OE} na „1”.

Programowanie bitów konfiguracyjnych (Fuse Bits)

Algorytm programowania bitów konfiguracyjnych jest następujący (zasady ładowania poleceń są takie same jak w punkcie *Programowanie pamięci Flash*):

1. **A:** Ładuj polecenie o kodzie „0100 0000” (*Write Fuse Bits*).
2. **D:** Ładuj młodszy bajt danej (zawierającej informacje o poszczególnych bitach konfiguracyjnych). Wartości są zakodowane na bitach:
 - bit 5 – bit konfiguracyjny SPIEN,
 - bit 0 – bit konfiguracyjny FSTRT,
 - bity 7...6 i 4...1 = „1” (bity te w układzie AT90S2313 stanowią rezerwę i powinny pozostać w stanie „1”).

Uwaga! Wartość „0” bitu oznacza, że jest on zaprogramowany, „1” zaś, że jest niezaprogramowany.

3. Doprowadź impuls \square do wejścia \overline{WR} o czasie trwania równym 1...1,8 ms w celu zaprogramowania bitu konfiguracyjnego. Fakt zakończenia programowania nie jest sygnalizowany stanem „1” na wyjściu RDY/BUSY.

Programowanie bitów zabezpieczających (Lock Bits)

Algorytm programowania bitów zabezpieczających jest następujący (zasady ładowania poleceń są takie same jak w punkcie *Programowanie pamięci Flash*):

1. **A:** Ładuj polecenie o kodzie „0010 0000” (*Write Lock Bits*).
2. **D:** Ładuj młodszy bajt danej (zawierającej informacje o poszczególnych bitach zabezpieczających). Bity zabezpieczające są programowane wartością „0” na pozycjach:
 - bit 2 – Lock Bit2,
 - bit 1 – Lock Bit1,
 - bity 7...3 i 0 = „1” (bity te stanowią rezerwę i powinny pozostać w stanie „1”).
3. **E:** Zapisz młodszy bajt danej.

Bity zabezpieczające mogą być kasowane tylko poprzez kasowanie pamięci (*Chip Erase*).

Odczyt bitów konfiguracyjnych i bitów zabezpieczających

Algorytm odczytu bitów konfiguracyjnych i bitów zabezpieczających jest następujący (zasady ładowania poleceń są takie same jak w punkcie *Programowanie pamięci Flash*):

1. **A:** Ładuj polecenie o kodzie „0000 0100” (*Read Fuse and Lock Bits*).
2. Ustaw \overline{OE} na „0” i BS na „1”. Stan bitów zabezpieczających i bitów zabezpieczających może być teraz odczytany z linii DATA („0” oznacza stan zaprogramowany):
 - bit 7 – Lock Bit1,
 - bit 6 – Lock Bit2,
 - bit 5 – bit konfiguracyjny SPIEN,
 - bit 0 – bit konfiguracyjny FSTRT.
3. Ustaw \overline{OE} na „1”.

Odczyt bajtów sygnatury

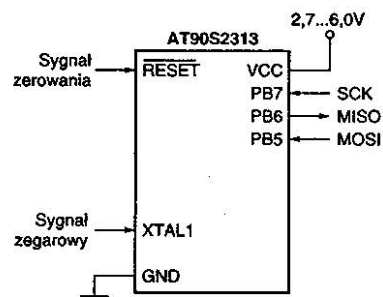
Algorytm odczytu bajtów sygnatury jest następujący (zasady ładowania poleceń i adresu są takie same jak w punkcie *Programowanie pamięci Flash*):

1. A: Ładuj polecenie o kodzie „0000 1000” (*Read Signature Bytes*).
2. C: Ładuj młodszy bajt adresu (\$00...\$7F).
3. Ustaw \overline{OE} na „0” i BS na „0”. Stan wybranego bajtu sygnatury może być teraz odczytany z linii DATA.
4. Ustaw \overline{OE} na „1”.

11.4.2. Programowanie szeregowe

Pamięć programu (Flash) i nieulotna pamięć danych (EEPROM) mikrokontrolera mają rozdzielne przestrzenie adresowe: \$0000...\$03FF (Flash) i \$000...\$07F (EEPROM). Obydwa obszary mogą być programowane przy wykorzystaniu interfejsu SPI (rysunek 11.4), który zawiera linie: SCK, MOSI (wejście), MISO (wyjście). Aby rozpocząć kasowanie/programowanie pamięci, linia \overline{RESET} musi znajdować się w stanie niskim. Po spełnieniu tego warunku niezbędne jest wykonanie polecenia *Programming Enable*

(zezwolenie na programowanie), zapobiegające ewentualnemu przypadkowemu wejściu w tryb programowania. Przed rozpoczęciem zapisu danych do pamięci EEPROM jest wykonywany automatyczny cykl kasowania, wykorzystujący układ wewnętrznego taktowania. Nie jest więc konieczne wywołanie polecenia *Chip Erase*. Operacja kasowania powoduje wypełnienie wszystkich komórek pamięci wartością \$FF. Zewnętrzny przebieg zegarowy może być doprowadzony do wejścia XTAL1. Można również wykorzystywać wbudowany w mikrokontroler oscylator, współpracujący z rezonatorem kwarcowym dołączonym do końcówek XTAL1 i XTAL2. Na przebieg taktujący operację zapisu doprowadzony poprzez wejście SCK, nałożone są pewne wymagania. Czasy trwania tak stanu niskiego, jak i wysokiego tego przebiegu powinny być dłuższe od dwóch okresów XTAL1.



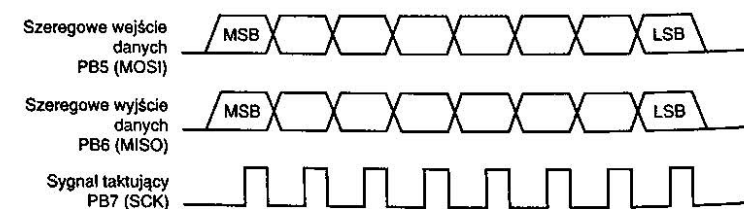
Rys. 11.4. Funkcje wyprowadzeń mikrokontrolera AT90S2313 podczas programowania szeregowego (w systemie)

Algorytm programowania szeregowego

Podczas szeregowego przesyłania danych do mikrokontrolera AT90S2313, dane są odczytywane na narastającym zboczach zegara SCK. Odbiór danych z mikrokontrolera jest natomiast taktowany zboczem opadającym. Szczegóły przedstawiono na rysunku 11.5 oraz w tabelicy 11.5.

Do programowania i weryfikacji pamięci wykorzystywane są 4-bajtowe polecenia, których format przedstawiono w tabelicy 11.5. Zalecana jest następująca procedura:

1. Po włączeniu zasilania należy doprowadzić napięcie zasilające do wyprowadzeń V_{CC} i GND, utrzymując przez ten czas linie \overline{RESET} i SCK w stanie niskim. Jeśli rezonator kwarcowy nie jest dołączony do końcówek XTAL1 i XTAL2, należy zastosować zewnętrzny przebieg zegarowy doprowadzony do wejścia XTAL1. W wielu sytuacjach nie można zagwarantować utrzymania linii SCK w stanie niskim, zanim napięcie zasilające osiągnie odpowiednią wartość. W takich przypadkach należy wygenerować dodatni impuls na linii \overline{RESET} przez co najmniej dwa okresy zegara XTAL1, po których linia SCK powinna zostać wyzerowana („0”).
2. Po odczekaniu 20 ms należy przesłać do mikrokontrolera za pomocą linii MOSI (PB5) polecenie *Programming Enable*.
3. Programowanie szeregowe nie będzie działać, jeśli nie nastąpi synchronizacja transmisji pomiędzy programatorem i mikrokontrolerem. W stanie synchronizmu, drugi bajt polecenia *Programming Enable* (\$53) spowoduje zwrócenie trzeciego bajtu do programatora (echo). Jednak bez względu na to, czy odebrane echo będzie poprawne, czy nie, wszystkie cztery bajty polecenia muszą być przesłane do mikrokontrolera. Jeśli odebrane echo będzie niepoprawne, należy podać dodatni impuls SCK i ponownie wysłać polecenie *Programming Enable*. Brak poprawnej odpowiedzi po kodzie \$53 utrzymujący się w 32 próbach nawiązania synchronizacji oznacza, że występują problemy z programowanym mikrokontrolerem, bądź z połączeniem pomiędzy nim a programatorem.



Rys. 11.5. Przebiegi charakteryzujące programowanie szeregowe mikrokontrolera

Tab. 11.5. Formaty poleceń wykorzystywanych podczas programowania szeregowego pamięci Flash i EEPROM

Polecenie	Format polecenia				Operacja
	bajt 1	bajt 2	bajt 3	bajt 4	
Zezwolenie na programowanie	1010 1100	0101 0011	xxxx xxxx	xxxx xxxx	Zezwolenie na programowanie szeregowo, jeśli linia RESET jest w stanie niskim
Kasowanie pamięci	1010 1100	100x xxxx	xxxx xxxx	xxxx xxxx	Kasowanie pamięci Flash i EEPROM
Czytanie pamięci Programu	0010 H000	xxxx xaa	bbbb bbbb	0000 0000	Czytanie H bajtu pamięci spod adresu a:b
Zapis pamięci Programu	0100 H000	xxxx xaa	bbbb bbbb	iiii iiii	Zapis H bajtu pamięci pod adres a:b
Czytaj pamięć EEPROM	1010 0000	xxxx xxxx	xbbb bbbb	0000 0000	Czytanie danej z pamięci EEPROM spod adresu b
Zapis pamięci EEPROM	1100 0000	xxxx xxxx	xbbb bbbb	iiii iiii	Zapis danej do pamięci EEPROM pod adres b
Zapis bitów zabezpieczających	1010 1100	111x x21x	xxxx xxxx	xxxx xxxx	Zapis bitów zabezpieczających. W celu zaprogramowania danego bitu zabezpieczającego należy wyzerować odpowiadający mu bit 1 lub 2 (1,2="0")
Czytaj bajty sygnatury	0011 0000	xxxx xxxx	xxxx xbbb	0000 0000	Czytaj bajt sygnatury spod adresu bb ⁽¹⁾

Uwagi:

⁽¹⁾ Odczyt bajtów sygnatury nie jest możliwy, gdy ustawiono tryb 3 zabezpieczenia pamięci nieulotnej (gdy obydwa bity są zaprogramowane). Odblokowanie tej opcji można wykonać jedynie poprzez programowanie równoległe.

a=starsze bity adresu

b=mlodsze bity adresu

H=0 – mlodszy bajt, H=1 – starszy bajt

o=dane wyjściowe

i=dane wejściowe

x=wartość bitu bez znaczenia

1=bit zabezpieczający 1

2=bit zabezpieczający 2

4. Jeśli wykonano polecenie *Chip Erase* (jest obowiązkowe podczas programowania pamięci Flash), należy odczekać 8...18 ms (czas zależy m.in. od napięcia zasilania), po czym wygenerować dodatni impuls na wejściu RESET i rozpocząć działania od kroku 2.

5. Do zaprogramowania każdego bajtu pamięci Flash i EEPROM niezbędne jest przesłanie jego adresu oraz zawartości. Każda komórka w pamięci EEPROM jest przed zapisaniem nowej wartości kasowana. Przejście do programowania kolejnego adresu może być realizowane metodą *Data Polling* opisaną dalej. Przed zmianą adresu należy odczekać 4...9 ms (ten czas zależy m.in. od napięcia zasilania). Operację programowania można przyspieszyć omijając adresy, które zawierają dane o wartości \$FF.

6. Poprawność zapisu można zweryfikować wykorzystując polecenie *Read*. Powoduje ono przesłanie zapisanej w pamięci danej z mikrokontrolera do programatora poprzez linię MISO (PB6).

7. Zakończenie programowania następuje po ustawieniu linii RESET w stan wysoki, rozpoczynając tym samym normalną pracę mikrokontrolera.

8. Jeśli potrzebna jest sekwencja wyłączenia zasilania (*power-off*), to powinna wyglądać następująco:

- ustawić wyprowadzenie XTAL1 w stanie „0” (przy braku rezonatora kwarcowego),
- ustawić linię RESET w stan „1”,
- wyłączyć zasilanie.

Data Polling EEPROM

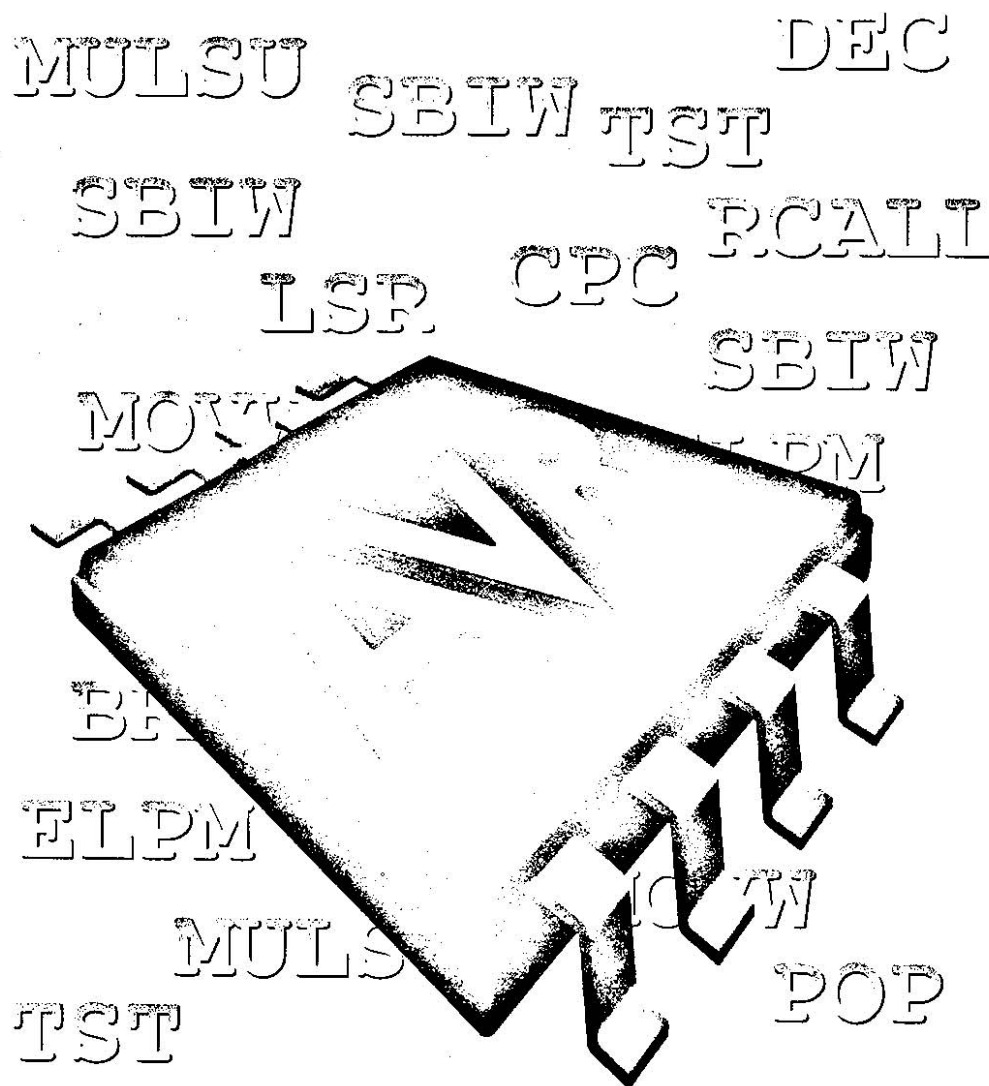
Przechodzenie pomiędzy poszczególnymi programowanymi komórkami pamięci EEPROM może się odbywać na zasadzie odpytywania pamięci: „czy już można dalej?” (*Data Polling*). Jeśli proces autokasowania nie zostanie zakończony, w wyniku cyklicznego odczytywania programowanej lokacji otrzymuje się wartość \$80. Po zakończeniu kasowania będzie odczytywana wartość \$7F. Gdy układ jest gotowy do przyjęcia kolejnej danej, w wyniku odczytania tej lokacji otrzymuje się zaprogramowaną daną. Jeśli zdarzy się, że do pamięci mają być zapisane dane \$80 i \$7F, po zaprogramowaniu pierwszej wartości należy odczekać co najmniej 4...9 ms, a następnie przejść do programowania drugiej. Wszystkie komórki wykasowanej pamięci mają wartość \$FF. Zapis danych o wartości \$FF można więc pomijać. W ten sposób udaje się niekiedy skrócić czas programowania pamięci mikrokontrolera. Metoda *Data Polling* nie może być stosowana, jeśli EEPROM jest reprogramowany bez wcześniejszego kasowania pamięci (*Chip Erase*).

Data Polling Flash

W przypadku pamięci Flash, w wyniku cyklicznego odczytywania programowanej lokacji otrzymywana będzie wartość \$7F przez czas, gdy pamięć nie będzie gotowa do przyjęcia kolejnej danej. Gdy gotowość taka nastąpi, odczytana dana będzie odpowiadać ostatnio zapisywanej do pamięci i w tym momencie można przystąpić do programowania kolejnej lokacji. Jeśli do pamięci Flash ma być zapisana dana o wartości \$7F, to zanim rozpocznie się jej programowanie należy odczekać min. 4...9 ms po zaprogramowaniu poprzedniej. Na skutek kasowania pamięci wszystkie jej komórki przybierają wartość \$FF. Zapisywane danej o wartości \$FF można więc pomijać.

Część 3

Lista rozkazów



12. Zestawienie rozkazów mikrokontrolera AT90S2313

Zastosowane w mikrokontrolerach AVR mnemoniki są niestety trudne do zapamiętania, mimo że w ich doborze widać pewną logikę i konsekwencję. Problem polega m.in. na tym, że w liście rozkazów mikrokontrolerów AVR spotykamy zbliżone „brzmieniowo” mnemoniki zdecydowanie różniących się rozkazów. Przykładem mogą być choćby: SBI – ustaw bit w rejestrze we/wy i SBIC – skocz, jeśli jest ustawiony bit w rejestrze we/wy.

Czy listy rozkazów trzeba koniecznie się uczyć? I tak i nie. Tak, bo wpływa to korzystnie na płynne i wydajne pisanie programów, pozwalając niejednokrotnie na optymalne wykorzystanie zasobów mikrokontrolera. Nie, bo po pierwsze: zawsze można przecież zajrzeć do pomocy, choćby takiej jak zawarta w tej książce. Po drugie: „czysty” assembler coraz rzadziej jest wykorzystywany do pisania programów, oddając prymat językom wysokiego poziomu (Bascom, C). Wszystkim tym, którzy rozpoczynają przygodę z mikrokontrolerami AVR gorąco polecam zapoznanie się z ich listą rozkazów, bez względu na to, czy będą pisać programy w assemblerze, czy nie. Znajomość ta ułatwia zrozumienie architektury mikrokontrolera. Czasami zachodzi potrzeba napisania fragmentu programu zoptymalizowanego w szczególny sposób, np. pod względem zależności czasowych. Pisząc go w języku C, nie mówiąc już o Bascomie, nie zawsze programista może zapanować nad kodem wynikowym. Nie zawsze też jest on optymalny. W takich sytuacjach przydają się lub są wręcz niezbędne wstawki assemblerowe. I tu często okazuje się, że część nabytej kiedyś wiedzy „poszła w zapomnienie”. Wystarczy jednak „rzut oka” na zestawienie mnemoników i wszystko staje się jasne. Takie zestawienie znajduje się w dalszej części książki i przyda się ono wszystkim tym, którzy listę rozkazów już znają, a w konkretnych sytuacjach potrzebują szybkiego odświeżenia wiadomości. Pełny opis rozkazów znajduje się w rozdziale 12.1.

Do opisu rozkazów użyto symboli, których znaczenie wyjaśniono poniżej. Szczególną uwagę należy zwrócić na przyjęte oznaczenia rejestrów. Wynikają one z ich ograniczonego zasięgu działania:

- Rd – rejestr źródła lub rejestr przeznaczenia (R0...R31),
- Rs – rejestr źródła (R0...R31),
- Rh – górne rejestry mikrokontrolera (R16...R31), Rhd oznacza rejestr przeznaczenia mieszczący się w zakresie rejestrów górnych

- (R16...R31), analogicznie Rhs – rejestr źródła należący do rejestrów górnych,
- Rm – rejestry środkowe mikrokontrolera (R16...R23), Rmd oznacza rejestr przeznaczenia mieszczący się w zakresie rejestrów środkowych (R16...R31), analogicznie Rms – rejestr źródła należący do rejestrów środkowych,
- RR – para rejestrów (r24=R25:R24, r26=R27:R26, r28=R29:R28, r30=R31:R30),
- Rxyz – rejestry indeksowe (X=R27:R26, Y=R29:R28, Z=R31:R30),
- Ry – rejestry indeksowe z przemieszczeniem (Y=R29:R28, Z=R31:R30),
- c63 – stała wskaźnikowa (0...63),
- c127 – stała używana jako parametr określający zakres skoku warunkowego (-64...63),
- c255 – stała 8-bitowa (0...255),
- c1024 – stała używana jako parametr określający zakres skoku względnego (-512...511),
- adr – stała określająca 16-bitowy adres (0...65535),
- adr2k – stała określająca względne przemieszczenie w obrębie ± 2 k,
- adr64k – stała określająca 16-bitowy adres (0...\$FFFF),
- adr65535 – stała określająca 16-bitowy adres (0...\$FFFF) w obrębie bieżącego segmentu,
- adr4M – stała określająca 22-bitowy adres (0...\$3FFFFFF),
- b – pozycja bitu (0...7),
- P – numer portu (rejestru) należącego do obszaru we/wy (0...63 = \$00...\$3F),
- Pl – adres portu (rejestru) należącego do dolnej strony obszaru we/wy (0...31 = \$00...\$1F),
- +
- dodawanie arytmetyczne,
-
- odejmowanie arytmetyczne,
- ∨
- suma logiczna,
- ^
- iloczyn logiczny,
- ⊕
- suma modulo 2 (Ex-OR),

Ponadto w opisie rozkazów przyjęto następujące zapisy symboliczne:

- Ri – zawartość 8-bitowego rejestru Ri,
- Rj:Ri – zawartość 16-bitowego rejestru złożonego z rejestrów Rj (starszy) i Ri (młodszy),
- (Ri) – komórka pamięci adresowana przez rejestr Ri,

- Ri(b) – bit b rejestru Ri,
- Rk(j...i) – bity od i do j rejestru Rk, np. R1(3...0) – młodsze 4 bity rejestru R1,
- $a \Rightarrow b$ – koniunkcja logiczna (jeżeli a, to b),
- $a \leftarrow b$ – przypisanie b do a (np. wpisanie wartości b do rejestru a).
- Zachowanie się znaczników (flag) po wykonaniu rozkazu jest przedstawione za pomocą oznaczeń:
- ↔ – flaga ustawiana zgodnie z wynikiem operacji,
- 0 – flaga zerowana („0”),
- 1 – flaga ustawiana („1”),
- – flaga pozostaje bez zmiany.

Tab. 12.1. Zestawienie listy rozkazów mikrokontrolera AT90S2313

Mnemonik	Operandy	Opis	Operacja	Modyfikowane flagi	Liczba taktów zegara
Operacje arytmetyczne i logiczne					
ADD	Rd, Rs	Dodaj zawartość dwóch rejestrów	$Rd \leftarrow Rd + Rs$	Z,C,N,V,H,S	1
ADC	Rd, Rs	Dodaj zawartość dwóch rejestrów z przeniesieniem	$Rd \leftarrow Rd + Rs + C$	Z,C,N,V,H,S	1
ADIW	RR, c63	Dodaj bezpośrednio stałą do słowa	$RRh:RRI \leftarrow RRh:RRI + c63$	Z,C,N,V,S	2
SUB	Rd, Rs	Odejmij zawartość dwóch rejestrów	$Rd \leftarrow Rd - Rs$	Z,C,N,V,H,S	1
SUBI	Rh, c255	Odejmij stałą od rejestru	$Rh \leftarrow Rh - c255$	Z,C,N,V,H,S	1
SBIW	RR, c63	Odejmij bezpośrednio stałą od słowa	$RRh:RRI \leftarrow RRh:RRI - c63$	Z,C,N,V,S	2
SBC	Rd, Rs	Odejmij zawartość dwóch rejestrów z przeniesieniem	$Rd \leftarrow Rd - Rs - C$	Z,C,N,V,H,S	1
SBCI	Rh, c255	Odejmij stałą z przeniesieniem od rejestru	$Rh \leftarrow Rh - c255 - C$	Z,C,N,V,H,S	1
AND	Rd, Rs	Iloczyn logiczny rejestrów	$Rd \leftarrow Rd \wedge Rs$	Z,N,V,S	1
ANDI	Rh, c255	Iloczyn logiczny rejestru i stałej	$Rh \leftarrow Rh \wedge c255$	Z,N,V,S	1
OR	Rd, Rs	Suma logiczna rejestrów	$Rd \leftarrow Rd \vee Rs$	Z,N,V,S	1
ORI	Rh, c255	Suma logiczna rejestru i stałej	$Rh \leftarrow Rh \vee c255$	Z,N,V,S	1
EOR	Rd, Rs	Suma Exclusive OR rejestrów	$Rd \leftarrow Rd \oplus Rs$	Z,N,V,S	1
COM	Rd	Uzupełnienie do jedności (negacja bitów)	$Rd \leftarrow \text{SFF} - Rd$	Z,C,N,V,S	1
NEG	Rd	Uzupełnienie do dwóch	$Rd \leftarrow \$00 - Rd$	Z,C,N,V,H,S	1
SBR	Rh, c255	Ustaw bit(y) w rejestrze	$Rh \leftarrow Rh \vee c255$	Z,N,V,S	1
CBR	Rh, c255	Zeruj bit(y) w rejestrze	$Rh \leftarrow Rh \wedge c255$	Z,N,V,S	1
INC	Rd	Inkrementuj rejestr	$Rd \leftarrow Rd + 1$	Z,N,V,S	1
DEC	Rd	Dekrementuj rejestr	$Rd \leftarrow Rd - 1$	Z,N,V,S	1
TST	Rd	Sprawdź zero lub minus	$Rd \leftarrow Rd \wedge Rd$	Z,N,V,S	1
CLR	Rd	Zeruj rejestr	$Rd \leftarrow Rd \oplus Rd$	Z,N,V,S	1
SER	Rh	Ustaw rejestr	$Rh \leftarrow \text{SFF}$	-	1
MUL	Rd, Rs	Mnożenie liczb bez znaku	$R1:R0 \leftarrow Rd \cdot Rs$	Z,C	2
MULS	Rhd, Rhs	Mnożenie liczb ze znakiem	$R1:R0 \leftarrow Rhd \cdot Rhs$	Z,C	2
MULSU	Rhd, Rhs	Mnożenie liczby ze znakiem z liczbą bez znaku	$R1:R0 \leftarrow Rhd \cdot Rhs$	Z,C	2

Mnemonik	Operandy	Opis	Operacja	Modyfikowane flagi	Liczba taktów zegara
FMUL	Rd, Rs	Mnożenie liczb ułamkowych bez znaku	$R1:R0 \leftarrow (Rd \cdot Rs) << 1$	Z, C	2
FMULS	Rd, Rs	Mnożenie liczb ułamkowych ze znakiem	$R1:R0 \leftarrow (Rd \cdot Rs) << 1$	Z, C	2
FMULSU	Rd, Rs	Mnożenie liczby ułamkowej ze znakiem z liczbą ułamkową bez znaku	$R1:R0 \leftarrow (Rd \cdot Rs) << 1$	Z, C	2
Rozkazy skoków					
RJMP	c1024	Skok względny	$PC \leftarrow PC + c1024 + 1$	-	2
IJMP		Skok pośredni określony zawartością rejestru Z	$PC \leftarrow Z$	-	2
EIJMP		Rozszerzony skok pośredni określony zawartością rejestru Z	$PC(15..0) \leftarrow Z$ $PC(21..16) \leftarrow EIND$	-	2
JMP	adr4M	Skok bezpośredni	$PC \leftarrow \text{adr4M}$	-	3
RCALL	c1024	Względne wywołanie podprogramu	$(SPL) \leftarrow PC + 1$ $SPL \leftarrow SPL - 2$ $PC \leftarrow PC + c1024 + 1$	-	3
ICALL	-	Pośrednie wywołanie podprogramu, określone zawartością rejestru Z	$(SPL) \leftarrow PC + 1$ $SPL \leftarrow SPL - 2$ $PC \leftarrow Z$	-	3
HCALL	-	Rozszerzone, pośrednie wywołanie podprogramu, określone zawartością rejestrów Z i EIND	$(SP) \leftarrow PC + 1$ $SPL \leftarrow SPL - 3$ $PC(15..0) \leftarrow Z$ $PC(21..16) \leftarrow EIND$	-	4
CALL	adr4M	Wywołanie podprogramu	$(SP) \leftarrow PC + 1$ $(SP) \leftarrow SP - 2$ $PC \leftarrow \text{adr4M}$	-	4
RET	-	Powrót z podprogramu	$PC \leftarrow (SPL)$ $SPL \leftarrow SPL + 2$	-	4
RETI	-	Powrót z przerwania	$PC \leftarrow (SPL)$ $SPL \leftarrow SPL + 2$	I	4
CPSE	Rd, Rs	Porównaj i przeskocz, jeśli równe	$(Rd = Rs) \Rightarrow PC \leftarrow PC + 2$ lub 3	-	1 lub 2
CP	Rd, Rs	Porównaj rejestry	$Rd - Rs$	Z, C, N, V, H, S	1
CPC	Rd, Rs	Porównaj rejestry z przeniesieniem	$Rd - Rs - C$	Z, C, N, V, H, S	1
CPI	Rh, c255	Porównaj rejestr ze stałą	$Rh - c255$	Z, C, N, V, H, S	1
SBRC	Rs, b	Przeskocz, jeśli bit w rejestrze jest wyzerowany	$Rs(b) = 0 \Rightarrow PC \leftarrow PC + 2$ $Rs(b) = 1 \Rightarrow PC \leftarrow PC + 1$	-	1, 2 lub 3
SBRS	Rs, b	Przeskocz, jeśli bit w rejestrze jest ustawiony	$Rs(b) = 1 \Rightarrow PC \leftarrow PC + 2$ $Rs(b) = 0 \Rightarrow PC \leftarrow PC + 1$	-	1, 2 lub 3
SBIC	Pl, b	Przeskocz, jeśli bit w rejestrze we/wy jest wyzerowany	$Pl(b) = 0 \Rightarrow PC \leftarrow PC + 2$ $Pl(b) = 1 \Rightarrow PC \leftarrow PC + 1$	-	1, 2 lub 3
SBIS	Pl, b	Przeskocz, jeśli bit w rejestrze we/wy jest ustawiony	$Pl(b) = 1 \Rightarrow PC \leftarrow PC + 2$ $Pl(b) = 0 \Rightarrow PC \leftarrow PC + 1$	-	1, 2 lub 3
BRBS	b, c127	Skok względny, jeśli flaga w rejestrze SREG jest ustawiona	$SREG(b) = 1 \Rightarrow PC \leftarrow PC + c127 + 1$ $SREG(b) = 0 \Rightarrow PC \leftarrow PC + 1$	-	1 lub 2
BRBC	b, c127	Skok względny, jeśli flaga w rejestrze SREG jest wyzerowana	$SREG(b) = 0 \Rightarrow PC \leftarrow PC + c127 + 1$ $SREG(b) = 1 \Rightarrow PC \leftarrow PC + 1$	-	1 lub 2
BREQ	c127	Skok względny, jeśli równe	$Z = 1 \Rightarrow PC \leftarrow PC + c127 + 1$ $Z = 0 \Rightarrow PC \leftarrow PC + 1$	-	1 lub 2

Mnemonik	Operandy	Opis	Operacja	Modyfikowane flagi	Liczba taktów zegara
BRNE	c127	Skok względny, jeśli nierówne	$Z = 0 \Rightarrow PC \leftarrow PC + c127 + 1$ $Z = 1 \Rightarrow PC \leftarrow PC + 1$	-	1 lub 2
BRCS	c127	Skok względny, jeśli flaga przeniesienia jest ustawiona	$C = 1 \Rightarrow PC \leftarrow PC + c127 + 1$ $C = 0 \Rightarrow PC \leftarrow PC + 1$	-	1 lub 2
BRCC	c127	Skok względny, jeśli flaga przeniesienia jest wyzerowana	$C = 0 \Rightarrow PC \leftarrow PC + c127 + 1$ $C = 1 \Rightarrow PC \leftarrow PC + 1$	-	1 lub 2
BRSH	c127	Skok względny, jeśli większy lub równy (dotyczy liczb bez znaku)	$C = 0 \Rightarrow PC \leftarrow PC + c127 + 1$ $C = 1 \Rightarrow PC \leftarrow PC + 1$	-	1 lub 2
BRLO	c127	Skok względny, jeśli mniejszy (dotyczy liczb bez znaku)	$C = 1 \Rightarrow PC \leftarrow PC + c127 + 1$ $C = 0 \Rightarrow PC \leftarrow PC + 1$	-	1 lub 2
BRMI	c127	Skok względny, jeśli ujemny	$N = 1 \Rightarrow PC \leftarrow PC + c127 + 1$ $N = 0 \Rightarrow PC \leftarrow PC + 1$	-	1 lub 2
BRPL	c127	Skok względny, jeśli dodatni	$N = 0 \Rightarrow PC \leftarrow PC + c127 + 1$ $N = 1 \Rightarrow PC \leftarrow PC + 1$	-	1 lub 2
BRGE	c127	Skok względny, jeśli większy lub równy (dotyczy liczb ze znakiem)	$S = 0 \Rightarrow PC \leftarrow PC + c127 + 1$ $S = 1 \Rightarrow PC \leftarrow PC + 1$	-	1 lub 2
BRLT	c127	Skok względny, jeśli mniejszy niż zero (dotyczy liczb ze znakiem)	$S = 1 \Rightarrow PC \leftarrow PC + c127 + 1$ $S = 0 \Rightarrow PC \leftarrow PC + 1$	-	1 lub 2
BRHS	c127	Skok względny, jeśli flaga przeniesienia pomocniczego ustawiona	$H = 1 \Rightarrow PC \leftarrow PC + c127 + 1$ $H = 0 \Rightarrow PC \leftarrow PC + 1$	-	1 lub 2
BRHC	c127	Skok względny, jeśli flaga przeniesienia pomocniczego wyzerowana	$H = 0 \Rightarrow PC \leftarrow PC + c127 + 1$ $H = 1 \Rightarrow PC \leftarrow PC + 1$	-	1 lub 2
BRTS	c127	Skok względny, jeśli znacznik T jest ustawiony	$T = 1 \Rightarrow PC \leftarrow PC + c127 + 1$ $T = 0 \Rightarrow PC \leftarrow PC + 1$	-	1 lub 2
BRTC	c127	Skok względny, jeśli znacznik T jest wyzerowany	$T = 0 \Rightarrow PC \leftarrow PC + c127 + 1$ $T = 1 \Rightarrow PC \leftarrow PC + 1$	-	1 lub 2
BRVS	c127	Skok względny, jeśli flaga przepełnienia ustawiona	$V = 1 \Rightarrow PC \leftarrow PC + c127 + 1$ $V = 0 \Rightarrow PC \leftarrow PC + 1$	-	1 lub 2
BRVC	c127	Skok względny, jeśli flaga przepełnienia wyzerowana	$V = 0 \Rightarrow PC \leftarrow PC + c127 + 1$ $V = 1 \Rightarrow PC \leftarrow PC + 1$	-	1 lub 2
BRIE	c127	Skok względny, jeśli przerwania odblokowane	$I = 1 \Rightarrow PC \leftarrow PC + c127 + 1$ $I = 0 \Rightarrow PC \leftarrow PC + 1$	-	1 lub 2
BRID	c127	Skok względny, jeśli przerwania zablokowane	$I = 0 \Rightarrow PC \leftarrow PC + c127 + 1$ $I = 1 \Rightarrow PC \leftarrow PC + 1$	-	1 lub 2
Rozkazy transferu danych					
MOV	Rd, Rs	Kopiuje zawartość rejestru Rs do rejestru Rd	$Rd \leftarrow Rs$	-	1
MOVW	Rd+1:Rd, Rs+1:Rs	Kopiuje zawartość słowa z rejestrów Rs+1:Rs do rejestrów Rd+1:Rd	$Rd+1:Rd \leftarrow Rs+1:Rs$	-	1
LDI	Rh, c255	Ładuj rejestr bezpośrednio stałą	$Rh \leftarrow c255$	-	1
LD	Rd, X	Ładuj rejestr pośrednio daną z pamięci SRAM spod adresu wskazywanego przez rejestr X	$Rd \leftarrow (X)$	-	2
LD	Rd, X+	Ładuj rejestr pośrednio daną z pamięci SRAM spod adresu wskazywanego przez rejestr X z postinkrementacją	$Rd \leftarrow (X)$ $X \leftarrow X + 1$	-	2
LD	Rd, -X	Ładuj rejestr pośrednio daną z pamięci SRAM spod adresu wskazywanego przez rejestr X z predekrementacją	$X \leftarrow X - 1$ $Rd \leftarrow (X)$	-	2
LD	Rd, Y	Ładuj rejestr pośrednio daną z pamięci SRAM spod adresu wskazywanego przez rejestr Y	$Rd \leftarrow (Y)$	-	2

Mnemonic	Operandy	Opis	Operacja	Modyfikowane flagi	Liczba taktów zegara
LD	Rd,Y+	Ładuj rejestr pośrednio daną z pamięci SRAM spod adresu wskazywanego przez rejestr Y z postinkrementacją	$Rd \leftarrow (Y)$ $Y \leftarrow Y + 1$	-	2
LD	Rd,-Y	Ładuj rejestr pośrednio daną z pamięci SRAM spod adresu wskazywanego przez rejestr Y z predekrementacją	$Y \leftarrow Y - 1$ $Rd \leftarrow (Y)$	-	2
LDD	Rd,Y+c63	Ładuj rejestr pośrednio daną z pamięci SRAM spod adresu wskazywanego przez rejestr Y z przemieszczeniem	$Rd \leftarrow (Y + c63)$	-	2
LD	Rd,Z	Ładuj rejestr pośrednio daną z pamięci SRAM spod adresu wskazywanego przez rejestr Z	$Rd \leftarrow (Z)$	-	2
LD	Rd,Z+	Ładuj rejestr pośrednio daną z pamięci SRAM spod adresu wskazywanego przez rejestr Z z postinkrementacją	$Rd \leftarrow (Z)$ $Z \leftarrow Z + 1$	-	2
LD	Rd,-Z	Ładuj rejestr pośrednio daną z pamięci SRAM spod adresu wskazywanego przez rejestr Z z predekrementacją	$Z \leftarrow Z - 1$ $Rd \leftarrow (Z)$	-	2
LDD	Rd,Z+c63	Ładuj rejestr pośrednio daną z pamięci SRAM spod adresu wskazywanego przez rejestr Z z przemieszczeniem	$Rd \leftarrow (Z + c63)$	-	2
LDS	Rd,adr65535	Ładuj rejestr bezpośrednio daną z pamięci SRAM spod adresu adr	$Rd \leftarrow (adr65535)$	-	2
ST	X,Rs	Zachowaj pośrednio rejestr w pamięci SRAM pod adres wskazywany przez rejestr X	$(X) \leftarrow Rs$	-	2
ST	X+,Rs	Zachowaj pośrednio rejestr w pamięci SRAM pod adres wskazywany przez rejestr X z postinkrementacją	$(X) \leftarrow Rs$ $X \leftarrow X + 1$	-	2
ST	-X,Rs	Zachowaj pośrednio rejestr w pamięci SRAM pod adres wskazywany przez rejestr X z predekrementacją	$X \leftarrow X - 1$ $(X) \leftarrow Rs$	-	2
ST	Y,Rs	Zachowaj pośrednio rejestr w pamięci SRAM pod adres wskazywany przez rejestr Y	$(Y) \leftarrow Rs$	-	2
ST	Y+,Rs	Zachowaj pośrednio rejestr w pamięci SRAM pod adres wskazywany przez rejestr Y z postinkrementacją	$(Y) \leftarrow Rs$ $Y \leftarrow Y + 1$	-	2
ST	-Y,Rs	Zachowaj pośrednio rejestr w pamięci SRAM pod adres wskazywany przez rejestr Y z predekrementacją	$Y \leftarrow Y - 1$ $(Y) \leftarrow Rs$	-	2
STD	Y+c63,Rs	Zachowaj pośrednio rejestr w pamięci SRAM pod adres wskazywany przez rejestr Y z przemieszczeniem	$(Y+c63) \leftarrow Rs$	-	2
ST	Z,Rs	Zachowaj pośrednio rejestr w pamięci SRAM pod adres wskazywany przez rejestr Z	$(Z) \leftarrow Rs$	-	2
ST	Z+,Rs	Zachowaj pośrednio rejestr w pamięci SRAM pod adres wskazywany przez rejestr Z z postinkrementacją	$(Z) \leftarrow Rs$ $Z \leftarrow Z + 1$	-	2
ST	-Z,Rs	Zachowaj pośrednio rejestr w pamięci SRAM pod adres wskazywany przez rejestr Z z predekrementacją	$Z \leftarrow Z - 1$ $(Z) \leftarrow Rs$	-	2

Mnemonic	Operandy	Opis	Operacja	Modyfikowane flagi	Liczba taktów zegara
STD	Z+c63,Rs	Zachowaj pośrednio rejestr w pamięci SRAM pod adres wskazywany przez rejestr Z z przemieszczeniem	$(Z+c63) \leftarrow Rs$	-	2
STS	adr,Rs	Zachowaj bezpośrednio rejestr w pamięci SRAM pod adres adr	$(adr) \leftarrow Rs$	-	2
LPM	-	Ładuj bajt pamięci programu do rejestru R0	$R0 \leftarrow (Z)$	-	3
LPM	Rd,Z	Ładuj bajt pamięci programu do rejestru Rd	$Rd \leftarrow (Z)$	-	3
LPM	Rd,Z+	Ładuj bajt pamięci programu do rejestru Rd z postinkrementacją	$Rd \leftarrow (Z)$ $Z \leftarrow Z + 1$	-	3
ELPM	-	Rozszerzone ładowanie bajtu pamięci programu do rejestru R0	$R0 \leftarrow (RAMPZ:Z)$	-	3
ELPM	Rd,Z	Rozszerzone ładowanie bajtu pamięci programu do rejestru Rd	$Rd \leftarrow (RAMPZ:Z)$	-	3
ELPM	Rd,Z+	Rozszerzone ładowanie bajtu pamięci programu do rejestru Rd z postinkrementacją	$Rd \leftarrow (RAMPZ:Z)$ $Z \leftarrow Z + 1$	-	3
SPM	-	Zapisz pamięć programu	$(Z) \leftarrow R1:R0$	-	-
IN	Rd,P	Czytaj port	$Rd \leftarrow P$	-	1
OUT	P,Rs	Zapisz port	$P \leftarrow Rs$	-	1
PUSH	Rs	Odlóż rejestr na stos	$(SPL) \leftarrow Rs$ $SPL \leftarrow SPL - 1$	-	2
POP	Rd	Pobierz ze stosu	$SPL \leftarrow SPL + 1$ $Rd \leftarrow (SPL)$	-	2
Rozkazy bitowe i testujące bity					
SBI	Pl,b	Ustaw bit w rejestrze we/wy	$Pl(b) \leftarrow 1$	-	2
CBI	Pl,b	Zeruj bit w rejestrze we/wy	$Pl(b) \leftarrow 0$	-	2
LSL	Rd	Przesuń logicznie w lewo zawartość rejestru	$Rd(n+1) \leftarrow Rd(n), Rd(0) \leftarrow 0$	Z,C,N,V,H	1
LSR	Rd	Przesuń logicznie w prawo zawartość rejestru	$Rd(n) \leftarrow Rd(n+1), Rd(7) \leftarrow 0$	Z,C,N,V	1
ROL	Rd	Obróć w lewo przez przeniesienie	$Rd(0) \leftarrow C, Rd(n+1) \leftarrow Rd(n), C \leftarrow Rd(7)$	Z,C,N,V,H	1
ROR	Rd	Obróć w prawo przez przeniesienie	$Rd(7) \leftarrow C, Rd(n) \leftarrow Rd(n+1), C \leftarrow Rd(0)$	Z,C,N,V	1
ASR	Rd	Przesuń arytmetycznie w prawo	$Rd(n) \leftarrow Rd(n+1), n=0..6$	Z,C,N,V	1
SWAP	Rd	Zamień półbajty rejestru	$Rd(3..0) \leftarrow Rd(7..4), Rd(7..4) \leftarrow Rd(3..0)$	-	1
BSET	b	Ustaw flagę	$SREG(b) \leftarrow 1$	SREG(b)	1
BCLR	b	Zeruj flagę	$SREG(b) \leftarrow 0$	SREG(b)	1
BST	Rs,b	Zachowaj bit rejestru w znaczniku T	$T \leftarrow Rs(b)$	T	1
BLD	Rd,b	Ładuj znacznik T do bitu rejestru	$Rd(b) \leftarrow T$	-	1
SEC	-	Ustaw flagę przeniesienia	$C \leftarrow 1$	C	1
CLC	-	Zeruj flagę przeniesienia	$C \leftarrow 0$	C	1
SEN	-	Ustaw flagę wartości ujemnej	$N \leftarrow 1$	N	1
CLN	-	Zeruj flagę wartości ujemnej	$N \leftarrow 0$	N	1
SEZ	-	Ustaw flagę zera	$Z \leftarrow 1$	Z	1
CLZ	-	Zeruj flagę zera	$Z \leftarrow 0$	Z	1
SEI	-	Odblokuj przerwania	$I \leftarrow 1$	I	1
CLI	-	Zablokuj przerwania	$I \leftarrow 0$	I	1
SES	-	Ustaw flagę znaku	$S \leftarrow 1$	S	1

Mnemonik	Operandy	Opis	Operacja	Modyfikowane flagi	Liczba taktów zegara
CLS	–	Zeruj flagę znaku	$S \leftarrow 0$	S	1
SEV	–	Ustaw flagę przepełnienia uzupełnienia do dwóch	$V \leftarrow 1$	V	1
CLV	–	Zeruj flagę przepełnienia uzupełnienia do dwóch	$V \leftarrow 0$	V	1
SET	–	Ustaw flagę T w rejestrze SREG	$T \leftarrow 1$	T	1
CLT	–	Zeruj flagę T w rejestrze SREG	$T \leftarrow 0$	T	1
SEH	–	Ustaw flagę przeniesienia pomocniczego	$H \leftarrow 1$	H	1
CLH	–	Zeruj flagę przeniesienia pomocniczego	$H \leftarrow 0$	H	1
Rozkazy sterujące pracą MCU					
NOP	–	Nic nie rób		–	1
SLEEP	–	Przejdź w tryb uśpienia	Działanie opisano w dalszej części rozdziału	–	1
WDR	–	Zeruj rejestr watchdoga	Działanie opisano w dalszej części rozdziału	–	1
BREAK	–	Przerwij wykonywanie programu	Działanie opisano w dalszej części rozdziału	–	1

12.1. Opis działania rozkazów

W opisie działania rozkazów obowiązują oznaczenia skrótowe przedstawione na początku rozdziału 12. Ze względu na to, że działanie rozkazów mikrokontrolera jest zazwyczaj silnie związane z jego architekturą, wszystkie ewentualne wątpliwości należy wyjaśniać w odpowiednim wcześniejszym rozdziale książki. W mikrokontrolerze można wyróżnić trzy rejestry, które mają wpływ na przebieg programu, są to:

- 16-bitowy rejestr PC – licznik programu (*Program Counter*) zawierający adres pamięci programu, spod którego jest pobierany kod rozkazu wykonywanego w kolejnym kroku,
- 8-bitowy (w przypadku mikrokontrolera AT90S2313) rejestr wskaźnika stosu – SP (*Stack Pointer*), zawierający adres pamięci SRAM, pod który będzie zapisywany np. adres powrotu z podprogramu (adres następnego rozkazu po rozkazie wywołania podprogramu),
- 8-bitowy rejestr znaczników (flag) – SREG (*Status Register*), zawierający znaczniki mogące mieć wpływ na przebieg programu (wykorzystywane np. w rozkazach skoków warunkowych).

Ze względu na szczególne znaczenie tych rejestrów, w dalszym opisie uwzględniono sposób ich modyfikacji przez poszczególne rozkazy.

ADC – dodaj zawartość rejestrów z przeniesieniem

Dodanie zawartości dwóch rejestrów oraz znacznika C. Wynik jest umieszczany w rejestrze Rd.

Operacja: $Rd \leftarrow Rd + Rs + C$

$PC \leftarrow PC + 1$

Składnia: ADC Rd, Rs

Kod:

0	0	0	1	1	1	s	d	d	d	d	s	s	s	s
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:

I	T	H	S	V	N	Z	C
—	—	↔	↔	↔	↔	↔	↔

H: ustawiana, jeśli nastąpiło przeniesienie z bitu 3, w przeciwnym przypadku zerowana.

S: wykorzystywana do sprawdzania znaku wyniku.

V: ustawiana, jeśli nastąpiło przepełnienie operacji uzupełnienia do dwóch, w przeciwnym przypadku zerowana.

N: ustawiana, jeśli MSB został ustawiony, w przeciwnym przypadku zerowana.

Z: ustawiana, jeśli wynik jest równy zero, w przeciwnym przypadku zerowana.

C: ustawiana, jeśli nastąpiło przeniesienie z bitu MSB, w przeciwnym przypadku zerowana.

Przykład:

```

                ;dodaj dwie liczby 16-bitowe r1:r0 + r3:r2
add    r2,r0    ;dodaj młodsze bajty
adc    r3,r1    ;dodaj starsze bajty z przeniesieniem

```

ADD – dodaj zawartość dwóch rejestrów

Dodanie zawartości dwóch rejestrów. Wynik jest umieszczany w rejestrze Rd.

Operacja: $Rd \leftarrow Rd + Rs$

$PC \leftarrow PC + 1$

Składnia: ADD Rd, Rs

Kod:

0	0	0	0	1	1	s	d	d	d	d	s	s	s	s
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:

I	T	H	S	V	N	Z	C
—	—	↔	↔	↔	↔	↔	↔

H: ustawiana, jeśli nastąpiło przeniesienie z bitu 3, w przeciwnym przypadku zerowana.

S: wykorzystywana do sprawdzania znaku wyniku.

V: ustawiana, jeśli nastąpiło przepełnienie operacji uzupełnienia do dwóch, w przeciwnym przypadku zerowana.

N: ustawiana, jeśli MSB został ustawiony, w przeciwnym przypadku zerowana.

Z: ustawiana, jeśli wynik jest równy zero, w przeciwnym przypadku zerowana.

C: ustawiana, jeśli nastąpiło przeniesienie z bitu MSB, w przeciwnym przypadku zerowana.

Przykład:

```

add    r1,r2    ;dodaj r2 do r1
add    r28,r28  ;dodaj r28 do r28

```

ADIW – dodaj bezpośrednio stałą do słowa

Dodanie bezpośrednio stałej z zakresu 0...63 do zawartości jednej pary rejestrów środkowych i umieszczenie wyniku w tych rejestrach.

Operacja: $RRh:RRI \leftarrow RRh:RRI + c63$
 $PC \leftarrow PC + 1$

Składnia: ADIW $RRh:RRI, c63$

Kod:

1	0	0	1	0	1	1	0	c	c	h	h	c	c	c	c
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 2

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	↔	↔	↔	↔	↔

S: wykorzystywana do sprawdzania znaku wyniku.

V: ustawiana, jeśli nastąpiło przepełnienie operacji uzupełnienia do dwóch, w przeciwnym przypadku zerowana.

N: ustawiana, jeśli MSB został ustawiony, w przeciwnym przypadku zerowana.

Z: ustawiana, jeśli wynik jest równy zeru, w przeciwnym przypadku zerowana.

C: ustawiana, jeśli nastąpiło przeniesienie z bitu MSB, w przeciwnym przypadku zerowana.

Przykład:

```
adiw r25:r24,15    ;dodaj 15 do pary rejestrów r25:r24
adiw ZH:ZL,63      ;dodaj 63 do rejestru wskaźnikowego Z
```

AND – iloczyn logiczny rejestrów

Obliczenie iloczynu logicznego zawartości rejestrów Rd i Rs. Wynik jest umieszczany w rejestrze Rd.

Operacja: $Rd \leftarrow Rd \wedge Rs$
 $PC \leftarrow PC + 1$

Składnia: AND Rd, Rs

Kod:

0	0	1	0	0	0	s	d	d	d	d	s	s	s	s
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	↔	0	↔	↔	↔

S: wykorzystywana do sprawdzania znaku wyniku.

V: 0.

N: ustawiana, jeśli MSB został ustawiony, w przeciwnym przypadku zerowana.

Z: ustawiana, jeśli wynik jest równy zeru, w przeciwnym przypadku zerowana.

Przykład:

```
and r2,r3           ;oblicz iloczyn logiczny r2 i r3, wynik w r2
ldi r16,8           ;ustaw maskę 0000 1000 w rejestrze r16
and r2,r16          ;pozostaw tylko bit 3 w rejestrze r16
```

ANDI – iloczyn logiczny rejestru i stałej

Obliczenie iloczynu logicznego zawartości górnego rejestru Rh i stałej. Wynik jest umieszczany w rejestrze Rh.

Operacja: $Rh \leftarrow Rh \wedge c255$

$PC \leftarrow PC + 1$

Składnia: ANDI Rh,c255

Kod:

0	1	1	1	c	c	c	c	h	h	h	h	c	c	c	c
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	↔	0	↔	↔	—

S: wykorzystywana do sprawdzania znaku wyniku.

V: 0.

N: ustawiana, jeśli MSB został ustawiony, w przeciwnym przypadku zerowana.

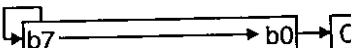
Z: ustawiana, jeśli wynik jest równy zero, w przeciwnym przypadku zerowana.

Przykład:

```
andi r17,$0f ;zeruj górny półbajt r17
andi r18,$10 ;izoluj 4 bit w r18
andi r19,$aa ;zeruj nieparzyste bity w r19
```

ASR – przesun arytmetycznie w prawo

Przesunięcie bitów rejestru Rd w prawo. Zawartość bitu 7 nie zmienia się, natomiast zawartość bitu 0 jest wpisywana do znacznika C w rejestrze SREG. Rozkaz umożliwia wykonywanie operacji dzielenia przez 2 liczby jednobajtowej ze znakiem. Wynik jest umieszczany w rejestrze Rd.

Operacja: 

$PC \leftarrow PC + 1$

Składnia: ASR Rd

Kod:

1	0	0	1	0	1	0	d	d	d	d	0	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	↔	↔	↔	↔	↔

S: wykorzystywana do sprawdzania znaku wyniku.

V: $N \oplus C$ (wartości N i C po przesunięciu).

N: ustawiana, jeśli MSB został ustawiony, w przeciwnym przypadku zerowana.

Z: ustawiana, jeśli wynik jest równy zero, w przeciwnym przypadku zerowana.

C: ustawiana, jeśli bit LSB rejestru przed przesunięciem był ustawiony, w przeciwnym przypadku zerowana.

Przykład:

```
ldi r16,16 ;załaduj r16 stałą równą 16
asr r16 ;r16=r16/2
ldi r17,-4 ;załaduj do r17 stałą równą -4 ($FC)
asr r17 ;r17=r17/2
```

BCLR – zeruj flagę w rejestrze SREG

Rozkaz zeruje wybraną flagę w rejestrze statusowym SREG.

Operacja: $SREG(b) \leftarrow 0$
 $PC \leftarrow PC + 1$

Składnia: BCLR b

Kod:

1	0	0	1	0	1	0	0	1	b	b	b	1	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:

I	T	H	S	V	N	Z	C
↔	↔	↔	↔	↔	↔	↔	↔

I: 0, jeśli b=7, w przeciwnym przypadku bez zmiany.

T: 0, jeśli b=6, w przeciwnym przypadku bez zmiany.

H: 0, jeśli b=5, w przeciwnym przypadku bez zmiany.

S: 0, jeśli b=4, w przeciwnym przypadku bez zmiany.

V: 0, jeśli b=3, w przeciwnym przypadku bez zmiany.

N: 0, jeśli b=2, w przeciwnym przypadku bez zmiany.

Z: 0, jeśli b=1, w przeciwnym przypadku bez zmiany.

C: 0, jeśli b=0, w przeciwnym przypadku bez zmiany.

Przykład:

```
bclr 0      ;zeruj flagę przeniesienia
bclr 7      ;zablokuj przerwania
```

BLD – ładuj znacznik T do bitu rejestru

Rozkaz kopiuje flagę T z rejestru statusowego SREG do bitu b rejestru Rd.

Operacja: $Rd(b) \leftarrow T$
 $PC \leftarrow PC + 1$

Składnia: BLD Rd,b

Kod:

1	1	1	1	1	0	0	d	d	d	d	0	b	b	b
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

```
bst  r1,2      ;umieść bit 2 rejestru r1 w znaczniku T
bld  r0,4      ;kopiuj flagę T do bitu 4 rejestru r0
```

BRBC – skok względny, jeśli flaga w rejestrze SREG jest wyzerowana

Warunkowe rozgałęzienie programu. Sprawdza pojedynczy bit w rejestrze SREG. Jeśli jest wyzerowany, to nastąpi skok względem bieżącego adresu wskazywanego przez PC. Zasięg skoku wynosi: $PC-63 \leq \text{adres_skoku} \leq PC+64$. Parametr c127 jest przesunięciem zapisanym w kodzie uzupełnienia do dwóch.

Operacja: Jeśli $SREG(b)=0$, to $PC \leftarrow PC + c127 + 1$
w przeciwnym przypadku $PC \leftarrow PC + 1$

Składnia: BRBC b,c127

Kod:

1	1	1	1	0	1	c	c	c	c	c	c	c	c	b	b	b
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1, jeśli warunek niespełniony

2, jeśli warunek spełniony

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

```

    cpi    r20,5      ;porównaj r20 z liczbą 5
    brbc   1,nie      ;skocz, jeśli flaga Z jest wyzerowana
    ..
    ..
nie:    nop           ;tu skocz
  
```

BRBS – skok względny, jeśli flaga w rejestrze SREG jest ustawiona

Warunkowe rozgałęzienie programu. Sprawdza pojedynczy bit w rejestrze SREG. Jeśli jest ustawiony, nastąpi skok względem bieżącego adresu wskazywanego przez PC. Zasięg skoku wynosi: $PC-63 \leq \text{adres_skoku} \leq PC+64$. Parametr c127 jest przesunięciem zapisanym w kodzie uzupełnienia do dwóch.

Operacja: Jeśli $SREG(b)=1$, to $PC \leftarrow PC + c127 + 1$
w przeciwnym przypadku $PC \leftarrow PC + 1$

Składnia: BRBS b,c127

Kod:

1	1	1	1	0	0	c	c	c	c	c	c	c	c	b	b	b
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1, jeśli warunek niespełniony

2, jeśli warunek spełniony

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

```

    bst    r0,3      ;ładuj flagę T bitem 3 z rejestru r0
    brbs   6,jest1   ;skocz, jeśli flaga T została ustawiona
    ..
    ..
jest1:  nop           ;tu skocz
  
```

BRCC – skok względny, jeśli flaga przeniesienia jest wyzerowana

Warunkowe rozgałęzienie programu. Sprawdza flagę przeniesienia (C). Jeśli jest ona wyzerowana, to nastąpi skok względem adresu wskazywanego przez PC. Zasięg skoku wynosi: $PC-63 \leq \text{adres_skoku} \leq PC+64$. Parametr c127 jest przesunięciem zapisanym w kodzie uzupełnienia do dwóch. Rozkaz ten jest odpowiednikiem BRBC 0, c127.

Operacja: Jeśli $C=0$, to $PC \leftarrow PC + c127 + 1$
w przeciwnym przypadku $PC \leftarrow PC + 1$

Składnia: BRCC c127

Kod:

1	1	1	1	0	1	c	c	c	c	c	c	c	c	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1, jeśli warunek niespełniony
2, jeśli warunek spełniony

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

```

        add  r22,r23    ;dodaj r23 do r22
        brcc niec       ;skocz, jeśli C=0
        ..
        ..
nec:    nop             ;tu skocz

```

BRCS – skok względny, jeśli flaga przeniesienia jest ustawiona

Warunkowe rozgałęzienie programu. Sprawdza flagę przeniesienia (C). Jeśli jest ustawiona, nastąpi skok względem bieżącego adresu wskazywanego przez PC. Zasięg skoku wynosi: $PC-63 \leq \text{adres_skoku} \leq PC+64$. Parametr c127 jest przesunięciem zapisanym w kodzie uzupełnienia do dwóch. Rozkaz ten jest odpowiednikiem BRBS 0, c127.

Operacja: Jeśli $C=1$, to $PC \leftarrow PC + c127 + 1$
w przeciwnym przypadku $PC \leftarrow PC + 1$

Składnia: BRCS c127

Kod:

1	1	1	1	0	0	c	c	c	c	c	c	c	c	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1, jeśli warunek niespełniony
2, jeśli warunek spełniony

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

```

        cpi  r26,$56    ;porównaj r26 z liczbą $56
        brcs jestc      ;skocz, jeśli C=1
        ..
        ..
jestc:  nop             ;tu skocz

```

BREAK – zatrzymaj CPU

Rozkaz BREAK jest wykorzystywany przez systemy uruchomieniowe *On-chip Debug*. W oprogramowaniu aplikacyjnym nie jest stosowany. Po wykonaniu rozkazu BREAK jednostka centralna mikrokontrolera AVR zostaje wprowadzona w tryb *Stopped Mode*, co stwarza możliwość dostępu do wewnętrznych zasobów mikrokontrolera przez debugger. Jeśli ustawiony jest któryś z bitów zabezpieczających albo bit konfiguracyjny (*fuse bit*) JTAGEN lub OCDEN jest niezaprogramowany, CPU traktuje rozkaz BREAK jako NOP i nie wchodzi w tryb *Stopped Mode*.



Rozkaz BREAK nie został zaimplementowany we wszystkich mikrokontrolerach AVR, w tym w AT90S2313. Szczegóły w notach katalogowych.

Operacja: Zatrzymanie CPU, $PC \leftarrow PC + 1$

Składnia: BREAK

Kod:

1	0	0	1	0	1	0	1	1	0	0	1	1	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

BREQ – skok względny, jeśli równe

Warunkowe rozgałęzienie programu. Sprawdza flagę zera (Z). Jeśli jest ustawiona, to nastąpi skok względem adresu wskazywanego przez PC. Gdy rozkaz ten jest wykonywany bezpośrednio po CP, CPI, SUB lub SUBI, skok zostanie wykonany wtedy i tylko wtedy, gdy binarna liczba bez znaku lub ze znakiem umieszczona w rejestrze Rd jest równa tak samo reprezentowanej liczbie zapisanej w rejestrze Rs. Zasięg skoku wynosi: $PC-63 \leq \text{adres_skoku} \leq PC+64$. Parametr c127 jest przesunięciem zapisanym w kodzie uzupełnienia do dwóch. Rozkaz ten jest odpowiednikiem BRBS 1, c127.

Operacja: Jeśli $Z=1$, czyli $Rd=Rs$, to $PC \leftarrow PC + c127 + 1$
w przeciwnym przypadku $PC \leftarrow PC + 1$

Składnia: BREQ c127

Kod:

1	1	1	1	0	1	c	c	c	c	c	c	c	0	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1, jeśli warunek niespełniony

2, jeśli warunek spełniony

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

```

cp    r1,r0    ;porównaj r1 z r0
breq  rowne    ;skocz, jeśli są równe
...
rowne: nop          ;tu skocz

```

BRGE – skok względny, jeśli większy lub równy (rozpatrywane są liczby ze znakiem)

Warunkowe rozgałęzienie programu. Sprawdza flagę znaku (S). Jeśli jest wyzerowana, to nastąpi skok względem bieżącego adresu wskazywanego przez PC. Gdy rozkaz ten jest wykonywany bezpośrednio po CP, CPI, SUB lub SUBI, skok zostanie wykonany wtedy i tylko wtedy, gdy binarna liczba ze znakiem umieszczona w rejestrze Rd jest większa lub równa od tak samo reprezentowanej liczby zapisanej w rejestrze Rs. Zasięg skoku wynosi: $PC-63 \leq \text{adres_skoku} \leq PC+64$. Parametr c127 jest przesunięciem zapisanym w kodzie uzupełnienia do dwóch. Rozkaz ten jest odpowiednikiem BRBC 4, c127.

Operacja: Jeśli $S=N \oplus V=0$, czyli $Rd \geq Rs$, to $PC \leftarrow PC + c127 + 1$
w przeciwnym przypadku $PC \leftarrow PC + 1$

Składnia: BRGE c127

Kod:

1	1	1	1	0	1	c	c	c	c	c	c	c	c	1	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1, jeśli warunek niespełniony
2, jeśli warunek spełniony

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

```

cp    r11,r12    ;porównaj r11 z r12
brge  niemn      ;skocz, jeśli r11 większy lub równy od r12
..
..
niemn: nop        ;tu skocz
```

BRHC – skok względny, jeśli flaga przeniesienia pomocniczego jest wyzerowana

Warunkowe rozgałęzienie programu. Sprawdza flagę przeniesienia pomocniczego (H). Jeśli jest wyzerowana, to nastąpi skok względem bieżącego adresu wskazywanego przez PC. Zasięg skoku wynosi: $PC-63 \leq \text{adres_skoku} \leq PC+64$. Parametr c127 jest przesunięciem zapisanym w kodzie uzupełnienia do dwóch. Rozkaz ten jest odpowiednikiem BRBC 5, c127.

Operacja: Jeśli $H=0$, to $PC \leftarrow PC + c127 + 1$
w przeciwnym przypadku $PC \leftarrow PC + 1$

Składnia: BRHC c127

Kod:

1	1	1	1	0	1	c	c	c	c	c	c	c	c	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1, jeśli warunek niespełniony
2, jeśli warunek spełniony

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

```

brhc  hzero      ;skocz, jeśli H=0
..
..
hzero: nop        ;tu skocz
```

BRHS – skok względny, jeśli flaga przeniesienia pomocniczego jest ustawiona

Warunkowe rozgałęzienie programu. Sprawdza flagę przeniesienia pomocniczego (H). Jeśli jest ustawiona, to nastąpi skok względem bieżącego adresu wskazywanego przez PC. Zasięg skoku wynosi: $PC-63 \leq \text{adres_skoku} \leq PC+64$. Parametr c127 jest przesunięciem zapisanym w kodzie uzupełnienia do dwóch. Rozkaz ten jest odpowiednikiem BRBS 5, c127.

Operacja: Jeśli $H=1$, to $PC \leftarrow PC + c127 + 1$
w przeciwnym przypadku $PC \leftarrow PC + 1$

Składnia: BRHS c127

Kod:

1	1	1	1	0	0	c	c	c	c	c	c	c	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1, jeśli warunek niespełniony

2, jeśli warunek spełniony

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

```

    brhs  hust      ;skocz, jeśli H=1
    ..
    ..
hust:  nop          ;tu skocz

```

BRID – skok względny, jeśli przerwania zablokowane

Warunkowe rozgałęzienie programu. Sprawdza flagę globalnego sterowania przerwaniami (I). Jeśli jest wyzerowana, to nastąpi skok względem bieżącego adresu wskazywanego przez PC. Zasięg skoku wynosi: $PC-63 \leq \text{adres_skoku} \leq PC+64$. Parametr c127 jest przesunięciem zapisanym w kodzie uzupełnienia do dwóch. Rozkaz ten jest odpowiednikiem BRBC 7, c127.

Operacja: Jeśli $I=0$, to $PC \leftarrow PC + c127 + 1$
w przeciwnym przypadku $PC \leftarrow PC + 1$

Składnia: BRID c127

Kod:

1	1	1	1	0	1	c	c	c	c	c	c	c	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1, jeśli warunek niespełniony

2, jeśli warunek spełniony

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

```

    brid  bezp      ;skocz, jeśli przerwania zablokowane
    ..
    ..
bezp:  nop          ;tu skocz

```

BRIE – skok względny, jeśli przerwania odblokowane

Warunkowe rozgałęzienie programu. Sprawdza flagę globalnego sterowania przerwaniami (I). Jeśli jest ustawiona, to nastąpi skok względem bieżącego adresu wskazywanego przez PC. Zasięg skoku wynosi: $PC-63 \leq \text{adres_skoku} \leq PC+64$. Parametr c127 jest przesunięciem zapisanym w kodzie uzupełnienia do dwóch. Rozkaz ten jest odpowiednikiem BRBS 7, c127.

Operacja: Jeśli $I=1$, to $PC \leftarrow PC + c127 + 1$
w przeciwnym przypadku $PC \leftarrow PC + 1$

Składnia: BRIE c127

Kod:

1	1	1	1	0	0	c	c	c	c	c	c	c	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1, jeśli warunek niespełniony
2, jeśli warunek spełniony

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

```

    brie zprz      ;skocz, jeśli przerwania odblokowane
    ..
    ..
zprz: nop          ;tu skocz

```

BRLO – skok względny, jeśli mniejszy (rozpatrywane są liczby bez znaku)

Warunkowe rozgałęzienie programu. Sprawdza flagę przeniesienia (C). Jeśli jest ona ustawiona, nastąpi skok względem bieżącego adresu wskazywanego przez PC. Gdy rozkaz ten jest wykonywany bezpośrednio po CP, CPI, SUB lub SUBI, skok zostanie wykonany wtedy i tylko wtedy, gdy binarna liczba bez znaku umieszczona w rejestrze Rd jest mniejsza od tak samo reprezentowanej liczby zapisanej w rejestrze Rs. Zasięg skoku wynosi: $PC-63 \leq \text{adres_skoku} \leq PC+64$. Parametr c127 jest przesunięciem zapisanym w kodzie uzupełnienia do dwóch. Rozkaz ten jest odpowiednikiem BRBS 0, c127.

Operacja: Jeśli $C=1$, czyli $Rd < Rs$, to $PC \leftarrow PC + c127 + 1$
w przeciwnym przypadku $PC \leftarrow PC + 1$

Składnia: BRLO c127

Kod:

1	1	1	1	0	0	c	c	c	c	c	c	c	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1, jeśli warunek niespełniony
2, jeśli warunek spełniony

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

```

    eor   r19,r19    ;zeruj r19
petla:  inc   r19     ;inkrementuj r19
    ..
    ..
    cpi   r19,$10    ;porównaj r19 z liczbą $10
    brlo  petla      ;skacz, jeśli r19<$10 (bez znaku)
    nop                   ;wyjście z petli

```

BRLT – skok względny, jeśli mniejszy niż zero (rozpatrywane są liczby ze znakiem)

Warunkowe rozgałęzienie programu. Sprawdza flagę znaku (S). Jeśli jest ona ustawiona, to nastąpi skok względem bieżącego adresu wskazywanego przez PC. Gdy rozkaz ten jest wykonywany bezpośrednio po CP, CPI, SUB lub SUBI, skok zostanie wykonany wtedy i tylko wtedy, gdy binarna liczba ze znakiem umieszczona w rejestrze Rd jest mniejsza od tak samo reprezentowanej liczby zapisanej w rejestrze Rs. Zasięg skoku wynosi: $PC-63 \leq \text{adres_skoku} \leq PC+64$. Parametr c127 jest przesunięciem zapisanym w kodzie uzupełnienia do dwóch. Rozkaz ten jest odpowiednikiem BRBS 4, c127.

Operacja: Jeśli $S=N \oplus V=1$, czyli $Rd < Rs$, to $PC \leftarrow PC + c127 + 1$
w przeciwnym przypadku $PC \leftarrow PC + 1$

Składnia: BRLT c127

Kod:

1	1	1	1	0	0	c	c	c	c	c	c	c	c	1	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1, jeśli warunek niespełniony

2, jeśli warunek spełniony

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

```
cp   r16,r1      ;porównaj r16 z r1
brlt mniejsze    ;skok, jeśli r16<r1 (ze znakiem)
..
..
mniejsze: nop      ;tu skocz
```

BRMI – skok względny, jeśli wartość ujemna

Warunkowe rozgałęzienie programu. Sprawdza flagę wartości ujemnej (N). Jeśli jest ona ustawiona, to nastąpi skok względem bieżącego adresu wskazywanego przez PC. Zasięg skoku wynosi: $PC-63 \leq \text{adres_skoku} \leq PC+64$. Parametr c127 jest przesunięciem zapisanym w kodzie uzupełnienia do dwóch. Rozkaz ten jest odpowiednikiem BRBS 2, c127.

Operacja: Jeśli $N=1$, to $PC \leftarrow PC + c127 + 1$
w przeciwnym przypadku $PC \leftarrow PC + 1$

Składnia: BRMI c127

Kod:

1	1	1	1	0	0	c	c	c	c	c	c	c	c	c	0	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1, jeśli warunek niespełniony

2, jeśli warunek spełniony

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

```
subi  r18,4      ;odejmij liczbę 4 od r18
brmi  ujemny     ;skocz, jeśli wynik ujemny
..
..
ujemny: nop      ;tu skocz
```

BRNE – skok względny, jeśli nie równe

Warunkowe rozgałęzienie programu. Sprawdza flagę znaku (Z). Jeśli jest ona wyzerowana, to nastąpi skok względem bieżącego adresu wskazywanego przez PC. Gdy rozkaz ten jest wykonywany bezpośrednio po CP, CPI, SUB lub SUBI, skok zostanie wykonany wtedy i tylko wtedy, gdy binarna liczba ze znakiem lub bez znaku umieszczona w rejestrze Rd jest różna od tak samo reprezentowanej liczby zapisanej w rejestrze Rs. Zasięg skoku wynosi: $PC-63 \leq \text{adres_skoku} \leq PC+64$. Parametr c127 jest przesunięciem zapisanym w kodzie uzupełnienia do dwóch. Rozkaz ten jest odpowiednikiem BRBC 1, c127.

Operacja: Jeśli $Z=0$, czyli $Rd \neq Rs$, to $PC \leftarrow PC + c127 + 1$
w przeciwnym przypadku $PC \leftarrow PC + 1$

Składnia: BRNE c127

Kod:

1	1	1	1	0	1	c	c	c	c	c	c	c	0	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1, jeśli warunek niespełniony
2, jeśli warunek spełniony

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

```

eor   r27,r27    ;zeruj r27
petla: inc  r27    ;inkrementuj r27
      ..
      ..
      cpi   r27,5  ;porównaj r27 z liczbą 5
      brne petla  ;skocz, jeśli r27 < 5
      nop      ;wyjście z pętli

```

BRPL – skok względny, jeśli wartość dodatnia

Warunkowe rozgałęzienie programu. Sprawdza flagę wartości ujemnej (N). Jeśli jest ona wyzerowana, to nastąpi skok względem bieżącego adresu wskazywanego przez PC. Zasięg skoku wynosi: $PC-63 \leq \text{adres_skoku} \leq PC+64$. Parametr c127 jest przesunięciem zapisanym w kodzie uzupełnienia do dwóch. Rozkaz ten jest odpowiednikiem BRBC 2, c127.

Operacja: Jeśli $N=0$, to $PC \leftarrow PC + c127 + 1$
w przeciwnym przypadku $PC \leftarrow PC + 1$

Składnia: BRPL c127

Kod:

1	1	1	1	0	1	c	c	c	c	c	c	c	0	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1, jeśli warunek niespełniony
2, jeśli warunek spełniony

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

```

      subi  r26,$50 ;odejmij liczbę $50 od r26
      brpl dodatni ;skocz, jeśli wynik dodatni (r26 >= 0)
      ..
      ..
dodatni: nop      ;tu skocz

```

BRSH – skok względny jeśli większy lub równy (rozpatrywane są liczby bez znaku)

Warunkowe rozgałęzienie programu. Sprawdza flagę przeniesienia (C). Jeśli jest wyzerowana, to nastąpi skok względem bieżącego adresu wskazywanego przez PC. Gdy rozkaz ten jest wykonywany bezpośrednio po CP, CPI, SUB lub SUBI, skok zostanie wykonany wtedy i tylko wtedy, gdy binarna liczba bez znaku umieszczona w rejestrze Rd jest większa lub równa od tak samo reprezentowanej liczby zapisanej w rejestrze Rs. Zasięg skoku wynosi: $PC-63 \leq \text{adres_skoku} \leq PC+64$. Parametr c127 jest przesunięciem zapisanym w kodzie uzupełnienia do dwóch. Rozkaz ten jest odpowiednikiem BRBC 0, c127.

Operacja: Jeśli $C=0$, czyli $Rd \geq Rs$, to $PC \leftarrow PC + c127 + 1$
w przeciwnym przypadku $PC \leftarrow PC + 1$

Składnia: BRSH c127

Kod:

1	1	1	1	0	1	c	c	c	c	c	c	c	c	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1, jeśli warunek niespełniony

2, jeśli warunek spełniony

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

```

    subi   r19,4    ;odejmij liczbę 4 od r19
    brsh   wiekszy  ;skocz, jeśli r19>=4
    ..
    ..
wieszy: nop          ;tu skocz

```

BRTC – skok względny, jeśli flaga T jest wyzerowana

Warunkowe rozgałęzienie programu. Sprawdza flagę pomocniczą (T). Jeśli jest ona wyzerowana, to nastąpi skok względem bieżącego adresu wskazywanego przez PC. Zasięg skoku wynosi: $PC-63 \leq \text{adres_skoku} \leq PC+64$. Parametr c127 jest przesunięciem zapisanym w kodzie uzupełnienia do dwóch. Rozkaz ten jest odpowiednikiem BRBC 6, c127.

Operacja: Jeśli $T=0$, to $PC \leftarrow PC + c127 + 1$
w przeciwnym przypadku $PC \leftarrow PC + 1$

Składnia: BRTC c127

Kod:

1	1	1	1	0	1	c	c	c	c	c	c	c	c	1	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1, jeśli warunek niespełniony

2, jeśli warunek spełniony

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

```

    bst    r3,5      ;umieść bit 5 rejestru r3 w znaczniku T
    brtc   tzero     ;skocz, jeśli ten bit jest wyzerowany
    ..
    ..
tzero: nop          ;tu skocz

```

BRTS – skok względny, jeśli flaga T jest ustawiona

Warunkowe rozgałęzienie programu. Sprawdza flagę pomocniczą (T). Jeśli jest ona ustawiona, to nastąpi skok względem bieżącego adresu wskazywanego przez PC. Zasięg skoku wynosi: $PC-63 \leq \text{adres_skoku} \leq PC+64$. Parametr c127 jest przesunięciem zapisanym w kodzie uzupełnienia do dwóch. Rozkaz ten jest odpowiednikiem BRBS 6, c127.

Operacja: Jeśli $T=1$, to $PC \leftarrow PC + c127 + 1$
w przeciwnym przypadku $PC \leftarrow PC + 1$

Składnia: BRTS c127

Kod:

1	1	1	1	0	0	c	c	c	c	c	c	c	1	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1, jeśli warunek niespełniony
2, jeśli warunek spełniony

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

```

    bst  r3,5      ;umieść bit 5 rejestru r3 w znaczniku T
    brts tust      ;skocz, jeśli bit T jest ustawiony
    ..
    ..
tust:  nop         ;tu skocz

```

BRVC – skok względny, jeśli flaga przepełnienia jest wyzerowana

Warunkowe rozgałęzienie programu. Sprawdza flagę przepełnienia (V). Jeśli jest ona wyzerowana, to nastąpi skok względem bieżącego adresu wskazywanego przez PC. Zasięg skoku wynosi: $PC-63 \leq \text{adres_skoku} \leq PC+64$. Parametr c127 jest przesunięciem zapisanym w kodzie uzupełnienia do dwóch. Rozkaz ten jest odpowiednikiem BRBC 3, c127.

Operacja: Jeśli $V=0$, to $PC \leftarrow PC + c127 + 1$
w przeciwnym przypadku $PC \leftarrow PC + 1$

Składnia: BRVC c127

Kod:

1	1	1	1	0	1	c	c	c	c	c	c	c	0	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1, jeśli warunek niespełniony
2, jeśli warunek spełniony

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

```

    add  r3,r4      ;dodaj r4 do r3
    brvc niev       ;skocz, jeśli nie było przepełnienia
    ..
    ..
niev:  nop          ;tu skocz

```

BRVS – skok względny, jeśli flaga przepełnienia jest ustawiona

Warunkowe rozgałęzienie programu. Sprawdza flagę przepełnienia (V). Jeśli jest ona ustawiona, to nastąpi skok względem bieżącego adresu wskazywanego przez PC. Zasięg skoku wynosi: $PC-63 \leq \text{adres_skoku} \leq PC+64$. Parametr c127 jest przesunięciem zapisanym w kodzie uzupełnienia do dwóch. Rozkaz ten jest odpowiednikiem BRBS 3, c127.

Operacja: Jeśli $V=1$, to $PC \leftarrow PC + c127 + 1$
w przeciwnym przypadku $PC \leftarrow PC + 1$

Składnia: BRVC c127

Kod:

1	1	1	1	0	0	c	c	c	c	c	c	c	0	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1, jeśli warunek niespełniony
2, jeśli warunek spełniony

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

```
add r3,r4      ;dodaj r4 do r3
brvs przep     ;skocz, jeśli nastąpiło przepełnienie
..
..
przep: nop      ;tu skocz
```

BSET – ustaw flagę w rejestrze SREG

Rozkaz ustawia wybrany bit (flagę) w rejestrze statusowym SREG.

Operacja: $SREG(b) \leftarrow 1$
 $PC \leftarrow PC + 1$

Składnia: BSET b

Kod:

1	0	0	1	0	1	0	0	0	b	b	b	1	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:

I	T	H	S	V	N	Z	C
↔	↔	↔	↔	↔	↔	↔	↔

I: 1, jeśli b=7, w przeciwnym przypadku bez zmiany.
T: 1, jeśli b=6, w przeciwnym przypadku bez zmiany.
H: 1, jeśli b=5, w przeciwnym przypadku bez zmiany.
S: 1, jeśli b=4, w przeciwnym przypadku bez zmiany.
V: 1, jeśli b=3, w przeciwnym przypadku bez zmiany.
N: 1, jeśli b=2, w przeciwnym przypadku bez zmiany.
Z: 1, jeśli b=1, w przeciwnym przypadku bez zmiany.
C: 1, jeśli b=0, w przeciwnym przypadku bez zmiany.

Przykład:

```
bset 6          ;ustaw flagę T
bset 7          ;odblokuj przerwania
```

BST – zachowaj bit rejestru w znaczniku T

Rozkaz umieszcza bit b rejestru Rs w znaczniku T rejestru statusowego SREG.

Operacja: $T \leftarrow Rs(b)$
 $PC \leftarrow PC + 1$

Składnia: BST Rs,b

Kod:

1	1	1	1	1	0	1	s	s	s	s	s	0	b	b	b
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:

I	T	H	S	V	N	Z	C
—	↔	—	—	—	—	—	—

T: 0, jeśli bit b w rejestrze Rs jest wyzerowany, 1 w przeciwnym przypadku.

Przykład:

```
bst  r1,2      ;umieść bit 2 rejestru r1 w znaczniku T
bld  r0,4      ;kopiuje flagę T do bitu 4 rejestru r0
```

CALL – dalekie wywołanie podprogramu

Wywołanie podprogramu umieszczonego w dowolnym miejscu pamięci programu. Adres powrotu (adres następnego rozkazu po rozkazie wywołania podprogramu) jest zawsze odkładany na stos. Każdorazowo po odłożeniu jednego bajtu, wskaźnik stosu (SPH:SPL) jest dekrementowany (zmniejszany o jeden).

UWAGA

Rozkaz CALL nie został zaimplementowany we wszystkich mikrokontrolerach AVR, w tym w AT90S2313 (szczegóły w notach katalogowych). W innych mikrokontrolerach AVR jego działanie może być różne w zależności od wielkości obsługiwanej pamięci programu. Możliwe są wersje 16-bitowego adresu (128 kB pamięci) lub adresu 22-bitowego (8 MB pamięci). Trzeba pamiętać, że słowo zawierające kod rozkazu w mikrokontrolerach AVR ma długość 16-bitów (dwa bajty). Stąd np. 16-bitowa przestrzeń adresowa wymaga 128 kB pamięci.

Operacja: $(SPH:SPL) \leftarrow PC+1$ (odłóż adres powrotu na stos)
 $SPH:SPL \leftarrow SPH:SPL - 2$ (w przypadku adresu 16-bitowego (2-bajtowego))
 lub
 $SPH:SPL \leftarrow SPH:SPL - 3$ (w przypadku adresu 22-bitowego (3-bajtowego))
 $PC \leftarrow \text{adr64k}$ lub $PC \leftarrow \text{adr4M}$

Składnia: CALL adr64k

lub

CALL adr4M

Kod:

1	0	0	1	0	1	0	a	a	a	a	a	1	1	1	a
a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a

Liczba słów: 2 (4 bajty)

Liczba cykli: 4 dla adresu 16-bitowego

5 dla adresu 22-bitowego

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

```

mov r16,r0 ;kopiuj r0 do r16
call spr ;wywołaj podprogram
..
..
spr: cpi r16,$42 ;sprawdź, czy r16 ma wartość $42
breq error ;skocz, jeśli tak
ret
..
..
error: rjmp error ;zapętl program

```

CBI – zeruj bit w rejestrze we/wy

Zerowanie wyspecyfikowanego bitu w rejestrze we/wy. Rozkaz działa tylko na „niskich” rejestrach tego obszaru (o adresach z przedziału 0...31 (\$00...\$1F)).

Operacja: $Pl(b) \leftarrow 0$
 $PC \leftarrow PC + 1$

Składnia: CBI Pl,b

Kod:

1	0	0	1	1	0	0	0	p	p	p	p	p	b	b	b	b
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 2

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

```
cbi $12,7 ;zeruj bit 7 w porcie D
```

CBR – zeruj bity w rejestrze

Zerowanie wyspecyfikowanych bitów w „górnym” rejestrze Rh. Rozkaz realizuje operację iloczynu logicznego AND pomiędzy zawartością rejestru Rh i uzupełnieniem maski będącej parametrem rozkazu. Oznacza to, że zostaną wyzerowane te bity rejestru, którym odpowiadają bity o wartości „1” w masce.

Operacja: $Rh \leftarrow Rh \wedge (\$FF - c255)$
 $PC \leftarrow PC + 1$

Składnia: CBR Rh, c255

Kod:

0	1	1	1	c	c	c	c	h	h	h	h	c	c	c	c
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Kod rozkazu CBR jest podobny jak ANDI, przy czym bity c odpowiadają logicznemu uzupełnieniu (complement) stałej c255 ($\$FF - c255$)

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	↔	0	↔	↔	—

S: wykorzystywana do sprawdzania znaku wyniku.

V: 0.

N: ustawiana, jeśli MSB został ustawiony, w przeciwnym przypadku zerowana.

Z: ustawiana, jeśli wynik jest równy zero, w przeciwnym przypadku zerowana.

Przykład:

```
cbr r16,$f0 ;zeruj starszy półbajt rejestru r16
cbr r18,1    ;zeruj bit 0 w r18
cbr r17,0    ;bez żadnej akcji, r17 bez zmian!
```

CLC – zeruj flagę przeniesienia

Rozkaz zeruje flagę przeniesienia (C) w rejestrze statusowym SREG.

Operacja: $C \leftarrow 0$
 $PC \leftarrow PC + 1$

Składnia: CLC

Kod:

1	0	0	1	0	1	0	0	1	0	0	0	1	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	0

C: 0

Przykład:

```
add r0,r0 ;dodaj do siebie r0
clc       ;zeruj flagę przeniesienia
```

CLH – zeruj flagę przeniesienia pomocniczego

Rozkaz zeruje flagę przeniesienia pomocniczego (H) w rejestrze statusowym SREG.

Operacja: $H \leftarrow 0$
 $PC \leftarrow PC + 1$

Składnia: CLH

Kod:

1	0	0	1	0	1	0	0	1	1	0	1	1	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:

I	T	H	S	V	N	Z	C
—	—	0	—	—	—	—	—

H: 0

Przykład:

```
clh          ;zeruj flagę przeniesienia pomocniczego
```

CLI – zeruj flagę przerwań

Rozkaz zeruje flagę globalnego sterowania przerwaniami (I) w rejestrze statusowym SREG. Po wyzerowaniu flagi I natychmiast zostaną zablokowane wszelkie przerwania mikrokontrolera, nawet jeśli nastąpią w chwili wykonywania rozkazu CLI.

Operacja: $I \leftarrow 0$
 $PC \leftarrow PC + 1$

Składnia: CLI

Kod:

1	0	0	1	0	1	0	0	1	1	1	1	1	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:

I	T	H	S	V	N	Z	C
0	—	—	—	—	—	—	—

I: 0

Przykład:

```
.def    temp=r16
..
in      temp,sreg ;zachowaj rejestr statusowy (temp musi być
                  ;zdefiniowany przez użytkownika)
cli     ;zablokuj przerwania
sbi     eecr,eemwe ;start zapisu pamięci EEPROM
sbi     eecr,eewe
out     sreg,temp ;odtwórz stan procesora (w tym przerwania)
```

CLN – zeruj flagę wartości ujemnej

Rozkaz zeruje flagę wartości ujemnej (N) w rejestrze statusowym SREG.

Operacja: $N \leftarrow 0$
 $PC \leftarrow PC + 1$

Składnia: CLN

Kod:

1	0	0	1	0	1	0	0	1	0	1	0	1	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	0	—	—

N: 0

Przykład:

```
add  r2,r3    ;dodaj r3 do r2
cln                     ;zeruj flagę wartości ujemnej
```

CLR – zeruj rejestr

Zerowanie zawartości rejestru Rd. Rozkaz wykonuje operację Ex-OR (*Exclusive OR*) której argumentem jest ten sam rejestr Rd. Wynikiem takiego działania jest wyzerowanie wszystkich bitów rejestru Rd.

Operacja: $Rd \leftarrow Rd \oplus Rd$
 $PC \leftarrow PC + 1$

Składnia: CLR Rd

Kod:

0	0	1	0	0	1	d	d	d	d	d	d	d	d	d	d
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

patrz też rozkaz EOR Rd, Rs

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	0	0	0	1	—

S: 0.

V: 0.

N: 0.

Z: 1.

Przykład:

```
clr  r18      ;zeruj r18
petla: inc r18 ;inkrementuj r18
      ..
      cpi r18,$50 ;porównaj r18 z liczbą $50
      brne petla ;zapętł, jeśli nierówne
```

CLS – zeruj flagę znaku

Rozkaz zeruje flagę znaku (S) w rejestrze statusowym SREG.

Operacja: $S \leftarrow 0$
 $PC \leftarrow PC + 1$

Składnia: CLS

Kod:

1	0	0	1	0	1	0	0	1	1	0	0	1	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	0	—	—	—	—

S: 0

Przykład:

```
add  r2,r3    ;dodaj r3 do r2
cls                ;zeruj flagę znaku
```

CLT – zeruj flagę pomocniczą T

Rozkaz zeruje flagę pomocniczą (T) w rejestrze statusowym SREG.

Operacja: $T \leftarrow 0$
 $PC \leftarrow PC + 1$

Składnia: CLT

Kod:

1	0	0	1	0	1	0	0	1	1	1	0	1	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:

I	T	H	S	V	N	Z	C
—	0	—	—	—	—	—	—

T: 0

Przykład:

```
clt                ;zeruj flagę T
```

CLV – zeruj flagę przepełnienia uzupełnienia do dwóch

Rozkaz zeruje flagę przepełnienia (V) w rejestrze statusowym SREG.

Operacja: $V \leftarrow 0$
 $PC \leftarrow PC + 1$

Składnia: CLV

Kod:

1	0	0	1	0	1	0	0	1	0	1	1	1	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	0	—	—	—

V: 0

Przykład:

```
add  r2,r3    ;dodaj r3 do r2
clv                     ;zeruj flagę przepełnienia
```

CLZ – zeruj flagę zera

Rozkaz zeruje flagę Z w rejestrze statusowym SREG.

Operacja: $Z \leftarrow 0$
 $PC \leftarrow PC + 1$

Składnia: CLZ

Kod:

1	0	0	1	0	1	0	0	1	0	0	1	1	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	0	—

Z: 0

Przykład:

```
add  r2,r3    ;dodaj r3 do r2
clz                     ;zeruj flagę Z
```

COM – oblicz uzupełnienie do jedności

Rozkaz oblicza uzupełnienie do jedności Zawartości rejestru Rd, czyli neguje wszystkie jego bity. Wynik jest umieszczany w rejestrze Rd.

Operacja: $Rd \leftarrow \$FF - Rd$
 $PC \leftarrow PC + 1$

Składnia: COM Rd

Kod:

1	0	0	1	0	1	0	d	d	d	d	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	↔	0	↔	↔	1

S: wykorzystywana do sprawdzania znaku wyniku.

V: 0.

N: ustawiana, jeśli MSB został ustawiony, w przeciwnym przypadku zerowana.

Z: ustawiana, jeśli wynik jest równy zero, w przeciwnym przypadku zerowana.

C: ustawiana.

Przykład:

```
ldi  r16,$aa    ;r16=10101010b
com  r16        ;oblicz uzupełnienie do jedności w r16
                ;w tym przypadku r16=01010101b
breq zero       ;skocz, jeśli zero
..
..
zero: nop       ;dalsze obliczenia
```

CP – porównaj zawartość rejestrów

Porównanie zawartości rejestrów Rd i Rs bez zmiany ich stanu. Rozkaz ten może być wykorzystywany do realizacji skoków warunkowych.

Operacja: $Rd - Rs$
 $PC \leftarrow PC + 1$

Składnia: CP Rd, Rs

Kod:

0	0	0	1	0	1	s	d	d	d	d	s	s	s	s
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:

I	T	H	S	V	N	Z	C
—	—	↔	↔	↔	↔	↔	↔

H: ustawiana, jeśli nastąpiło przeniesienie z bitu 3, w przeciwnym przypadku zerowana.

S: wykorzystywana do sprawdzania znaku wyniku.

V: ustawiana, jeśli nastąpiło przepełnienie operacji uzupełnienia do dwóch, w przeciwnym przypadku zerowana.

N: ustawiana, jeśli MSB został ustawiony, w przeciwnym przypadku zerowana.

Z: ustawiana, jeśli wynik jest równy zero, w przeciwnym przypadku zerowana.

C: ustawiana, jeśli wartość bezwzględna zawartości rejestru Rs jest większa niż wartość bezwzględna zawartości rejestru Rd, w przeciwnym przypadku zerowana.

Przykład:

```
cp   r4,r19      ;porównaj r4 z r19
brne rozne       ;skocz, jeśli r4 <> r19
..
..
rozne: nop       ;dalsze obliczenia
```

CPC – porównaj zawartość rejestrów z uwzględnieniem flagi przeniesienia

Porównanie zawartości rejestrów Rd i Rs (bez zmiany ich stanu) uwzględniając przy tym stan flagi przeniesienia (C) przed operacją. Rozkaz ten może być wykorzystywany do realizacji skoków warunkowych. Jest to odpowiednik rozkazu SBC, przy czym wynik operacji nie jest nigdzie zapisywany. Typowym zastosowaniem może być porównanie dwóch rejestrów 16-bitowych złożonych z par rejestrów.

Operacja: Rd – Rs – C
PC ← PC + 1

Składnia: CPC Rd, Rs

Kod:

0	0	0	0	0	1	s	d	d	d	d	d	s	s	s	s
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:

I	T	H	S	V	N	Z	C
—	—	↔	↔	↔	↔	↔	↔

H: ustawiana, jeśli nastąpiło przeniesienie z bitu 3, w przeciwnym przypadku zerowana.

S: wykorzystywana do sprawdzania znaku wyniku.

V: ustawiana, jeśli nastąpiło przepełnienie operacji uzupełnienia do dwóch, w przeciwnym przypadku zerowana.

N: ustawiana, jeśli MSB został ustawiony, w przeciwnym przypadku zerowana.

Z: ustawiana, jeśli wynik jest równy zeru, w przeciwnym przypadku zerowana.

C: ustawiana, jeśli wartość bezwzględna zawartości rejestru Rs plus flaga przeniesienia sprzed operacji jest większa niż wartość bezwzględna zawartości rejestru Rd, w przeciwnym przypadku zerowana.

Przykład:

```

;porównaj rejestry 16-bitowe r3:r2 z r1:r0
cp    r2,r0    ;porównaj młodsze rejestry
cpc   r3,r1    ;porównaj starsze rejestry
brne  r0,r1    ;skocz, jeśli r4 <> r19
;
;
rozne: nop     ;dalsze obliczenia
```

CPI – porównaj rejestr ze stałą

Porównanie zawartości rejestru Rh ze stałą c255. Stan rejestru Rh nie ulega zmianie. Rozkaz ten może być wykorzystywany do realizacji skoków warunkowych.

Operacja: Rh – c255
PC ← PC + 1

Składnia: CPI Rh, c255

Kod:

0	0	1	1	c	c	c	c	h	h	h	h	c	c	c	c
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:

I	T	H	S	V	N	Z	C
—	—	↔	↔	↔	↔	↔	↔

H: ustawiana, jeśli nastąpiło przeniesienie z bitu 3, w przeciwnym przypadku zerowana.

S: wykorzystywana do sprawdzania znaku wyniku.

V: ustawiana, jeśli nastąpiło przepełnienie operacji uzupełnienia do dwóch, w przeciwnym przypadku zerowana.

N: ustawiana, jeśli MSB został ustawiony, w przeciwnym przypadku zerowana.

Z: ustawiana, jeśli wynik jest równy zeru, w przeciwnym przypadku zerowana.

C: ustawiana, jeśli wartość bezwzględna stałej c255 jest większa niż wartość bezwzględna zawartości rejestru Rd, w przeciwnym przypadku zerowana.

Przykład:

```

cpi   r19,3    ;porównaj r19 z liczbą 3
brne  r19,3    ;skocz, jeśli r19 <> 3
;
;
r19nie3: nop    ;dalsze obliczenia
```

CPSE – porównaj i skocz, jeśli równe

Rozkaz porównuje zawartość rejestrów Rd i Rs, po czym kolejny rozkaz nie jest wykonywany, jeśli zawartości rejestrów są jednakowe. Stan rejestrów nie ulega zmianie.

Operacja: Jeśli $Rd=Rs$, to $PC \leftarrow PC + 2$ (lub $PC + 3$)
w przeciwnym przypadku $PC \leftarrow PC + 1$

Składnia: CPSE Rd, Rs

Kod:

0	0	0	1	0	0	s	d	d	d	d	s	s	s	s
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1, jeśli warunek jest niespełniony (bez przeskoku)

2, jeśli warunek jest spełniony i następuje przeskok rozkazu złożonego z 1 słowa

3, jeśli warunek jest spełniony i następuje przeskok rozkazu złożonego z 2 słów

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

```
inc  r4      ;powiększ r4
cpse r4,r0   ;porównaj r4 z r0
neg  r4      ;wykonuj tylko wtedy, gdy r4 <> r0
nop         ;następne operacje
```

DEC – dekrementuj rejestr

Zmniejszenie o jeden zawartości rejestru Rd. Wynik jest umieszczany w Rd. Dekrementowanie rejestru nie powoduje modyfikowania flagi przeniesienia (C). Dzięki temu możliwe jest wykorzystanie rozkazu DEC do realizacji pętli, w których licznik jest wykorzystywany w obliczeniach o wielokrotnej precyzji. Jeśli w obliczeniach będą występowały liczby bez znaku, to skoki powinny być wykonywane tylko za pomocą rozkazów BREQ i BRNE. Jeśli zawartość dekrementowanych rejestrów będzie traktowana jako liczba zapisana w kodzie uzupełnień do dwóch, to skoki mogą być wykonywane za pomocą wszystkich rozkazów skoków (uwzględniających znak).

Operacja: $Rd \leftarrow Rd - 1$
 $PC \leftarrow PC + 1$

Składnia: DEC Rd

Kod:

1	0	0	1	0	1	0	d	d	d	d	1	0	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	↔	↔	↔	↔	—

S: wykorzystywana do sprawdzania znaku wyniku.

V: ustawiana, jeśli nastąpiło przepełnienie operacji uzupełnienia do dwóch, w przeciwnym przypadku zerowana. Przepełnienie operacji uzupełnienia do dwóch nastąpi tylko wówczas, gdy rejestr Rd po wykonaniu rozkazu będzie miał wartość równą \$80.

N: ustawiana, jeśli MSB został ustawiony, w przeciwnym przypadku zerowana.

Z: ustawiana, jeśli wynik jest równy zero, w przeciwnym przypadku zerowana.

Przykład:

```
ldi  r17,$10 ;ładuj wartość $10 do rejestru r17
petla: add r1,r2 ;dodaj r2 do r1
      dec r17 ;dekrementuj r17
      brne petla ;skocz, jeśli r17 <> r0
      nop ;dalsze operacje
      ..
      ..
```

EICALL – rozszerzone, pośrednie wywołanie podprogramu

Pośrednie wywołanie podprogramu. Adres podprogramu jest wskazywany przez rejestr indeksowy Z (16 młodszych bitów adresu) i rejestr EIND znajdujący się w obszarze we/wy (6 starszych bitów adresu). Rozkaz umożliwia wywołanie procedury ulokowanej w dowolnym miejscu pamięci programu. Adres powrotu (adres następnego rozkazu po rozkazie wywołania podprogramu) jest zawsze odkładany na stosie. Każdorazowo po odłożeniu jednego bajtu, wskaźnik stosu (SPH:SPL) jest dekrementowany (zmniejszany o jeden).



Rozkaz EICALL występuje jedynie w mikrokontrolerach z rdzeniem AVR3Core, np. AT90SC646C-USB.

Operacja: (SPH:SPL) \leftarrow PC+1 (odłóż adres powrotu na stos (3 bajty))
 SPH:SPL \leftarrow SPH:SPL - 3
 PC(15..0) \leftarrow Z(15..0), PC(21..16) \leftarrow EIND

Składnia: EICALL

Kod:

1	0	0	1	0	1	0	1	0	0	0	1	1	0	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 4 tylko dla mikrokontrolerów z 22-bitowym PC

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

```
ldi r16,$05 ;ustaw adres w rejestrze EIND
out eind,r16
ldi r30,$00 ;oraz w Z
ldi r31,$10
eicall ;wywołaj podprogram spod adresu $051000
..
..
```

EIJMP – rozszerzony skok pośredni

Skok pośredni pod adres wskazywany przez rejestr indeksowy Z (16 młodszych bitów adresu) i rejestr EIND znajdujący się w obszarze we/wy (6 starszych bitów adresu). Rozkaz umożliwia wykonanie skoku do dowolnego miejsca w pamięci programu.



Rozkaz EIJMP występuje jedynie w mikrokontrolerach z rdzeniem AVR3Core, np. AT90SC646C-USB.

Operacja: PC(15..0) \leftarrow Z(15..0), PC(21..16) \leftarrow EIND

Składnia: EIJMP

Kod:

1	0	0	1	0	1	0	0	0	0	0	1	1	0	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 2

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

```
ldi r16,$05 ;ustaw adres w rejestrze EIND
out eind,r16
ldi r30,$00 ;oraz w Z
ldi r31,$10
eijmp ;skocz do adresu $051000
..
..
```

ELPM – rozszerzone ładowanie bajtu z pamięci programu do rejestru

Ładowanie do rejestru Rd jednego bajtu pamięci programu wskazywanego przez rejestr indeksowy Z oraz rejestr RAMPZ znajdujący się w obszarze we/wy. Rozkaz jest wykorzystywany do inicjowania stałych i ich pobierania do dalszych obliczeń przez program. Pamięć programu w mikrokontrolerach AVR jest zorganizowana w 16-bitowe słowa, lecz rejestr Z wskazuje kolejne bajty (nie słowa). Jeśli najmłodszy bit rejestru Z będzie miał wartość „0”, to rejestr ten będzie wskazywał młodszy bajt słowa pamięci programu. Analogicznie wartość „1” będzie informowała, że chodzi o starszy bajt. Rozkaz ELPM może adresować całą pamięć programu bez żadnych ograniczeń. W wyniku jego działania, rejestr indeksowy może pozostać bez zmiany lub może być inkrementowany. Inkrementacja obejmuje zarówno rejestr Z, jak i rejestr RAMPZ (razem 24 bity).



Rozkaz ELPM nie został zaimplementowany we wszystkich mikrokontrolerach AVR, w tym w AT90S2313. Szczegóły w notach katalogowych.

Układy zezwalające na autoprogramowanie mogą wykorzystywać rozkaz ELPM do odczytywania stanu bitów konfiguracyjnych i bitów zabezpieczających. Szczegółowych informacji należy szukać w notach katalogowych.



Rezultat poniższych operacji jest nieokreślony:

```
elpm    r30,z+
elpm    r31,z+
```

Operacja: (*) $R0 \leftarrow (RAMPZ:Z)$ (R0 jest rejestrem domyślnym)
 $PC \leftarrow PC + 1$

(**) $Rd \leftarrow (RAMPZ:Z)$
 $PC \leftarrow PC + 1$

(***) $Rd \leftarrow (RAMPZ:Z)$
 $RAMPZ:Z \leftarrow RAMPZ:Z + 1$
 $PC \leftarrow PC + 1$

Składnia: (*) ELPM
 (**) ELPM Rd,Z
 (***) LPM Rd,Z+

Kod: (*)

1	0	0	1	0	1	0	1	1	1	0	1	1	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

(**)

1	0	0	1	0	0	0	d	d	d	d	d	0	1	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

(***)

1	0	0	1	0	0	0	d	d	d	d	d	0	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 3

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

```
ldi    z1,byte3(tabl<<1) ;inicjuj rejestr indeksowy Z
out     rampz,z1
ldi     zh,byte2(tabl<<1)
ldi     z1,byte1(tabl<<1)
elpm    r16,z+             ;ładuj stałą z pamięci programu
                               ;wskazywaną przez parę
                               ;rejestrów RAMPZ:Z

::
::
tabl:   .dw    $3738        ;bajt $38 jest wskazywany,
                               ;gdy ZLSB=0
                               ;bajt $37 jest wskazywany,
                               ;gdy ZLSB=1
```

EOR – suma Exclusive Or

Obliczenie sumy Exclusive Or rejestrów Rd i Rs. Wynik jest umieszczany w rejestrze Rd.

Operacja: $Rd \leftarrow Rd \oplus Rs$
 $PC \leftarrow PC + 1$

Składnia: EOR Rd, Rs

Kod:

0	0	1	0	0	1	s	d	d	d	d	s	s	s	s
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	↔	0	↔	↔	—

S: wykorzystywana do sprawdzania znaku wyniku.

V: zerowana.

N: ustawiana, jeśli MSB został ustawiony, w przeciwnym przypadku zerowana.

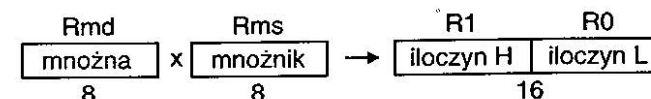
Z: ustawiana, jeśli wynik jest równy zero, w przeciwnym przypadku zerowana.

Przykład:

```
eor    r4,r4    ;zeruj r4
eor    r0,r22   ;oblicz Ex-OR z r0 i r22
```

FMUL – mnożenie ułamkowe bez znaku

Mnożenie dwóch liczb 8-bitowych. W wyniku operacji otrzymuje się liczbę 16-bitową bez znaku, która dodatkowo zostaje przesunięta w lewo o jeden bit.



Niech (N.Q) oznacza liczbę ułamkową zawierającą N binarnych cyfr w części całkowitej i Q binarnych cyfr w części ułamkowej. Wymnożenie dwóch liczb (N1.Q1) i (N2.Q2) da w wyniku liczbę ((N1+N2).(Q1+Q2)). W aplikacjach obróbki sygnałów szeroko jest stosowany format (1.7) dla wartości wejściowych i (2.14) dla wyniku mnożenia. Przesunięcie w lewo jest wymagane ze względu na zapewnienie zgodności formatów liczb wejściowych i wyniku mnożenia. Rozkaz FMUL wykonuje całą operację, włącznie z przesunięciem w tej samej liczbie cykli, co rozkaz MUL.

Format (1.7) jest bardzo często stosowany do obliczeń na liczbach ze znakiem, lecz rozkaz FMUL nadaje się tylko dla liczb bez znaku. Jest jednak przydatny do wykonywania obliczeń cząstkowych iloczynów podczas mnożenia 16-bitowych liczb ze znakiem zapisanych w formacie (1.15), dających w wyniku liczbę formatu (1.31). Najstarszy bit mnożenia przed przesunięciem musi być uwzględniany w obliczeniach i jest wpisywany do znacznika przeniesienia.

Mnożna wpisana do rejestru Rmd i mnożnik w Rms są traktowane jako liczby ułamkowe bez znaku, w których punkt dziesiętny znajduje się pomiędzy 6, a 7 bitem. 16-bitowy wynik jest liczbą ułamkową bez znaku, w której punkt dziesiętny jest położony pomiędzy 14, a 15 bitem. Wynik mnożenia jest umieszczony w rejestrze R1 (starszy bajt) i R0 (młodszy bajt).



Rozkazu FMUL nie zaimplementowano we wszystkich mikrokontrolerach AVR, w tym w AT90S2313. Szczegóły w notach katalogowych.

Operacja: $R1:R0 \leftarrow Rmd \cdot Rms$
 (liczba bez znaku (1.15) \leftarrow liczba bez znaku (1.7) \cdot liczba bez znaku (1.7))
 $PC \leftarrow PC + 1$

Składnia: FMUL Rmd, Rms (Rmd=R16 do R23, Rms=R16 do R23)

Kod:

0	0	0	0	0	0	1	1	0	d	d	d	1	s	s	s
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 2

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	↔	↔

Z: ustawiana, jeśli wynik jest równy \$0000, w przeciwnym przypadku zerowana.

C: ustawiana, jeśli bit 15 iloczynu przed przesunięciem był ustawiony, w przeciwnym przypadku zerowana.

Przykład:

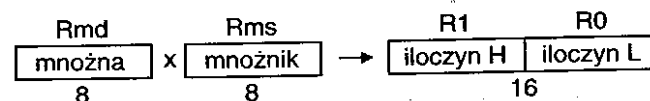
```

;*****
; Procedura mnożenia dwóch 16-bitowych liczb ułamkowych ze
; znakiem, dająca w wyniku liczbę 32-bitową
;
; r19:r18:r17:r16 = (r23:r22 * r21:r20) << 1
;*****
fmuls16x16_32:
    clr     r2
    fmuls   r23,r21      ;((signed)ah * (signed)bh) << 1
    movw    r19:r18,r1:r0 ;przepisz wynik do r19:r18
    fmul    r22,r20      ; (a1 * b1) << 1
    adc     r18,r2
    movw    r17:r16,r1:r0 ;przepisz wynik do r17:r16
    fmulsu  r23,r20      ;((signed)ah * b1) << 1
    sbc     r19,r2
    add     r17,r0
    adc     r18,r1
    adc     r19,r2
    fmulsu  r21,r22      ;((signed)bh * a1) << 1
    sbc     r19,r2
    add     r17,r0
    adc     r18,r1
    adc     r19,r2
    ret

```

FMULS – mnożenie ułamkowe ze znakiem

Mnożenie dwóch liczb 8-bitowych. W wyniku operacji otrzymuje się liczbę 16-bitową ze znakiem, która dodatkowo zostaje przesunięta w lewo o jeden bit.



Niech (N.Q) oznacza liczbę ułamkową zawierającą N binarnych cyfr w części całkowitej i Q binarnych cyfr w części ułamkowej. Wymnożenie dwóch liczb (N1.Q1) i (N2.Q2) da w wyniku liczbę ((N1+N2).(Q1+Q2)). W aplikacjach obróbki sygnałów szeroko jest stosowany format (1.7) dla wartości wejściowych i (2.14) dla wyniku mnożenia. Przesunięcie w lewo jest wymagane ze względu na zapewnienie zgodności formatów liczb wejściowych i wyniku mnożenia. Rozkaz FMULS wykonuje całą operację, włącznie z przesunięciem w tej samej liczbie cykli, co rozkaz MULS.

Mnożna wpisana do rejestru Rmd i mnożnik w Rms są traktowane jako liczby ułamkowe ze znakiem, w których punkt dziesiętny znajduje się pomiędzy 6 a 7 bitem. 16-bitowy wynik jest liczbą ułamkową ze znakiem, w której punkt dziesiętny jest położony pomiędzy 14 a 15 bitem. Wynik mnożenia jest umieszczony w rejestrze R1 (starszy bajt) i R0 (młodszy bajt).

Należy pamiętać, że podczas mnożenia liczby 0x80 (−1) przez liczbę 0x80 (−1) rezultatem przesunięcia będzie liczba 0x8000 (−1). W wyniku przesunięcia nastąpi przepełnienie uzupełnienia do dwóch. Sytuacja taka musi być rozpoznawana i odpowiednio obsługiwana przez program.



Rozkazu FMUL nie zaimplementowano we wszystkich mikrokontrolerach AVR, w tym w AT90S2313. Szczegóły w notach katalogowych.

Operacja: $R1:R0 \leftarrow Rmd \cdot Rms$
 (liczba ze znakiem (1.15) \leftarrow liczba ze znakiem (1.7) \cdot liczba ze znakiem (1.7))
 $PC \leftarrow PC + 1$

Składnia: FMULS Rmd,Rms (Rmd=R16 do R23, Rms=R16 do R23)

Kod:

0	0	0	0	0	0	1	1	1	d	d	d	0	s	s	s
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 2

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	↔	↔

Z: ustawiana, jeśli wynik jest równy \$0000, w przeciwnym przypadku zerowana.

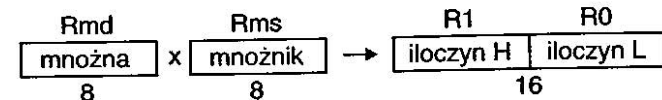
C: ustawiana, jeśli bit 15 iloczynu przed przesunięciem był ustawiony, w przeciwnym przypadku zerowana.

Przykład:

```
fmuls r23,r22      ;mnożenie liczb ze znakiem w formacie
                   ;(1.7) umieszczonych w rejestrach
                   ;r23 i r22
                   ;iloczyn w formacie (1.15)
movw r23:r22,r1:r0 ;przepisz wynik do r23:r22
```

FMULSU – mnożenie ułamkowe liczby ze znakiem przez liczbę bez znaku

Mnożenie dwóch liczb 8-bitowych. W wyniku operacji otrzymuje się liczbę 16-bitową ze znakiem, która dodatkowo zostaje przesunięta w lewo o jeden bit.



Niech (N.Q) oznacza liczbę ułamkową zawierającą N binarnych cyfr w części całkowitej i Q binarnych cyfr w części ułamkowej. Wymnożenie dwóch liczb (N1.Q1) i (N2.Q2) da w wyniku liczbę ((N1+N2).(Q1+Q2)). W aplikacjach obróbki sygnałów szeroko jest stosowany format (1.7) dla wartości wejściowych i (2.14) dla wyniku mnożenia. Przesunięcie w lewo jest wymagane ze względu na zapewnienie zgodności formatów liczb wejściowych i wyniku mnożenia. Rozkaz FMULSU wykonuje całą operację, włącznie z przesunięciem w tej samej liczbie cykli, co rozkaz MULSU.

Format (1.7) jest bardzo często stosowany do obliczeń na liczbach ze znakiem, lecz rozkaz FMULSU nadaje się tylko do mnożenia liczby ze znakiem przez liczbę bez znaku. Jest jednak przydatny do wykonywania obliczeń częściowych iloczynów podczas mnożenia 16-bitowych liczb ze znakiem zapisanych w formacie (1.15), dających w wyniku liczbę formatu (1.31). Najstarszy bit mnożenia przed przesunięciem musi być uwzględniany w obliczeniach i jest wpisywany do znacznika przeniesienia.

Mnożna wpisana do rejestru Rmd i mnożnik w Rms stanowią liczby ułamkowe w których punkt dziesiętny znajduje się pomiędzy 6, a 7 bitem. Mnożna Rmd jest liczbą ułamkową ze znakiem, a mnożnik Rms jest liczbą ułamkową bez znaku. 16-bitowy wynik jest liczbą ułamkową ze znakiem, w której punkt dziesiętny jest położony pomiędzy 14, a 15 bitem. Wynik mnożenia jest umieszczony w rejestrze R1 (starszy bajt) i R0 (młodszy bajt).



Rozkazu FMULSU nie zaimplementowano we wszystkich mikrokontrolerach AVR, w tym w AT90S2313. Szczegóły w notach katalogowych.

Operacja: $R1:R0 \leftarrow Rmd \cdot Rms$
 (liczba ze znakiem (1.15) \leftarrow liczba ze znakiem (1.7) \cdot liczba bez znaku (1.7))
 $PC \leftarrow PC + 1$

Składnia: FMULSU Rmd,Rms (Rmd=R16 do R23, Rms=R16 do R23)

Kod:

0	0	0	0	0	0	1	1	d	d	d	1	s	s	s
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 2

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	↔	↔

Z: ustawiana, jeśli wynik jest równy \$0000, w przeciwnym przypadku zerowana.

C: ustawiana, jeśli bit 15 iloczynu przed przesunięciem był ustawiony, w przeciwnym przypadku zerowana.

Przykład:

```

;*****
; Procedura mnożenia dwóch 16-bitowych liczb ułamkowych ze
; znakiem, dająca w wyniku liczbę 32-bitową
;
; r19:r18:r17:r16 = (r23:r22 * r21:r20) << 1
;*****
fmuls16x16_32:
    clr     r2
    fmul    r23,r21      ;((signed)ah * (signed)bh) << 1
    movw    r19:r18,r1:r0 ;przepisz wynik do r19:r18
    fmul    r22,r20      ; (a1 * b1) << 1
    adc     r18,r2
    movw    r17:r16,r1:r0 ;przepisz wynik do r17:r16
    fmulsu  r23,r20      ;((signed)ah * b1) << 1
    sbc     r19,r2
    add     r17,r0
    adc     r18,r1
    adc     r19,r2
    fmulsu  r21,r22      ;((signed)bh * a1) << 1
    sbc     r19,r2
    add     r17,r0
    adc     r18,r1
    adc     r19,r2
    ret

```

ICALL – pośrednie wywołanie podprogramu

Wywołanie pośrednie podprogramu. Adres podprogramu jest wskazywany przez rejestr indeksowy Z (16-bitowy), należący do rejestrów roboczych procesora. Podprogram może być więc ulokowany w najniższym obszarze adresowym (64 kslów (128 kB)) pamięci programu. Pamięć programu mikrokontrolera AT90S2313 ma wielkość 1 kslów. Adres powrotu (adres następnego rozkazu po rozkazie wywołania podprogramu) jest zawsze odkładany na stosie. Każdorazowo po odłożeniu jednego bajtu, wskaźnik stosu jest dekrementowany (zmniejszany o jeden).



Rozkazu ICALL nie zaimplementowano we wszystkich mikrokontrolerach AVR. Szczegóły w notach katalogowych.

Operacja: dla AT90S2313

(SPL) ← PC+1 (odłóż adres powrotu na stos)

SPL ← SPL – 2

PC ← Z

W mikrokontrolerach z 16-bitowym wskaźnikiem stosu.

(SPH:SPL) ← PC+1 (odłóż adres powrotu na stos)

SPH:SPL ← SPH:SPL – 2 (w przypadku adresu 16-bitowego (2-bajtowego))

lub

SPH:SPL ← SPH:SPL – 3 (w przypadku adresu 22-bitowego (3-bajtowego))

PC(15...0) ← Z(15.0), PC(21...16) ← 0

Składnia: ICALL (zarówno w przypadku adresu 16-bitowego, jak i 22-bitowego)

Kod:

1	0	0	1	0	1	0	1	0	0	0	0	1	0	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 3 dla 16-bitowego adresu

4 dla 22-bitowego adresu

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

```

mov r30,r0 ;ustaw adres procedury
icall      ;wywołaj podprogram

```

```

..

```

IJMP – skok pośredni określony zawartością rejestru Z

Skok pośredni. Adres skoku jest wskazywany przez rejestr indeksowy Z (16-bitowy), należący do rejestrów roboczych procesora. Długość rejestru zezwala na wykonywanie skoków w obrębie najniższego obszaru adresowego (64 kslów (128 kB)) pamięci programu. Pamięć programu mikrokontrolera AT90S2313 ma wielkość 1 kslów.



Rozkazu IJMP nie zaimplementowano we wszystkich mikrokontrolerach AVR. Szczegóły w notach katalogowych.

Operacja: dla AT90S2313

$PC \leftarrow Z$

w mikrokontrolerach z 22-bitowym licznikiem programu

$PC(15...0) \leftarrow Z(15...0)$, $PC(21...16) \leftarrow 0$

Składnia: IJMP

Kod:

1	0	0	1	0	1	0	0	0	0	0	1	0	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 2

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

```

mov r30,r0 ;ustaw adres skoku
ijmp      ;skok pod adres określony przez rejestr Z
..

```

IN – czytaj port

Załadowanie danej dostępnej w rejestrze należącym do obszaru we/wy (porty, timery/liczniki, rejestry konfiguracyjne itp.) do jednego z rejestrów roboczych.

Operacja: $Rd \leftarrow P$
 $PC \leftarrow PC + 1$

Składnia: IN Rd,P

Kod:

1	0	1	1	0	p	p	d	d	d	d	p	p	p	p
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

```
in    r25,$16    ;ładuj port B do rejestru r25
cpi   r25,4      ;porównaj odczytaną daną z liczbą 4
breq  wyjscie    ;skok, jeśli r25=4
..
wyjscie: nop      ;dalsze operacje
```

INC – inkrementuj zawartość rejestru

Zwiększenie o jeden zawartości rejestru Rd. Wynik jest umieszczany w Rd. Inkrementowanie rejestru nie powoduje modyfikowania flagi przeniesienia (C). Dzięki temu możliwe jest wykorzystanie rozkazu DEC do realizacji pętli, w których licznik jest wykorzystywany w obliczeniach o wielokrotnej precyzji. Jeśli w obliczeniach będą występowały liczby bez znaku, to skoki powinny być wykonywane tylko za pomocą rozkazów BREQ i BRNE. Jeśli zawartość dekrementowanych rejestrów będzie traktowana jako liczba zapisana w kodzie uzupełnień do dwóch, to skoki mogą być wykonywane za pomocą wszystkich rozkazów skoków (uwzględniających znak).

Operacja: $Rd \leftarrow Rd + 1$
 $PC \leftarrow PC + 1$

Składnia: INC Rd

Kod:

1	0	0	1	0	1	0	d	d	d	d	0	0	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	↔	↔	↔	↔	—

S: wykorzystywana do sprawdzania znaku wyniku.

V: ustawiana, jeśli nastąpiło przepełnienie operacji uzupełnienia do dwóch, w przeciwnym przypadku zerowana. Przepełnienie operacji uzupełnienia do dwóch nastąpi tylko wówczas, gdy Rd przed operacją był równy \$7F.

N: ustawiana, jeśli MSB został ustawiony, w przeciwnym przypadku zerowana.

Z: ustawiana, jeśli wynik jest równy zero, w przeciwnym przypadku zerowana.

Przykład:

```
clr   r22        ;zeruj r22
petla: inc r22     ;inkrementuj r22
..
cpi   r22,$4f    ;porównaj r22 ze stałą $4F
brne  petla      ;skocz, jeśli różne
nop    ;dalsze operacje
..
```

JMP – skok bezpośredni

Skok bezpośredni w obrębie przestrzeni 4 Mslów pamięci programu. Patrz również RJMP.



Rozkazu JMP nie zaimplementowano we wszystkich mikrokontrolerach AVR, w tym w AT90S2313. Szczegóły w notach katalogowych.

Operacja: $PC \leftarrow \text{adr4M}$ (stos pozostaje bez zmiany)

Składnia: **JMP** adr4M

Kod:

1	0	0	1	0	1	0	a	a	a	a	1	1	0	a
a	a	a	a	a	a	a	a	a	a	a	a	a	a	a

Liczba słów: 2 (4 bajty)

Liczba cykli: 3

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

```

mov    r1,r0    ;kopiuj r0 do r1
jmp     dlugiskok ;długi skok
;
;
dlugiskok: nop    ;dalsze operacje
;
;

```

LD – ładuj rejestr pośrednio daną z pamięci SRAM spod adresu wskazywanego przez rejestr indeksowy X

Ładowanie pośrednie jednego bajtu z obszaru danych do rejestru. W mikrokontrolerach zawierających pamięć SRAM (należy do nich AT90S2313) obszar danych (jest to obszar ciągły) rozciąga się na obszar rejestrów roboczych (adresy od 0 do \$1F), obszar we/wy (adresy od \$20 do \$5F) i wewnętrzną (roboczą) pamięć SRAM (adresy od \$60 do adresu wynikającego z wielkości tej pamięci, w przypadku AT90S2313 do \$DF), a także zewnętrzną pamięć SRAM, jeśli jest obsługiwana przez mikrokontroler. W układach bez pamięci SRAM obszarem danych będą jedynie rejestry robocze. Pamięć EEPROM ma wydzieloną przestrzeń adresową. Dodatkowe informacje można znaleźć w rozdziale 4. Adres danej, która ma być załadowana do rejestru jest wskazywany przez 16-bitowy rejestr indeksowy X. Dana ta może więc być ulokowana w najniższym obszarze adresowym (64 kB). Jeśli zachodzi potrzeba pobrania danej spoza tego obszaru (dotyczy tylko tych układów, które to umożliwiają, nie dotyczy AT90S2313), należy posłużyć się rejestrem RAMPX z obszaru we/wy.

W wyniku działania rozkazu LD, rejestr indeksowy może pozostać bez zmiany, może być inkrementowany po załadowaniu danej (*post-increment*) lub może być dekrementowany przed załadowaniem danej (*pre-decrement*). Stwarza to możliwość wygodnego manipulowania tablicami danych, a także implementacji stosu użytkownika (niezależnego od stosu mikrokontrolera), dla którego wskaźnikiem będzie rejestr X. Chociaż rejestr X jest rejestrem 16-bitowym, to w układach mających nie więcej niż 256 bajtów pamięci SRAM (np. AT90S2313) w wyniku operacji indeksowych jest modyfikowana tylko młodsza jego część. W takich układach starsza część rejestru indeksowego może być używana jako rejestr ogólnego przeznaczenia. Rejestr RAMPX jest modyfikowany w przypadku mikrokontrolerów mających przestrzeń danych lub przestrzeń pamięci programu większą niż 64 kB.



Nie wszystkie warianty rozkazu LD zaimplementowano we wszystkich mikrokontrolerach AVR.



Rezultat poniższych operacji jest nieokreślony:

```

ld      r26,x+
ld      r27,x+
ld      r26,-x
ld      r27,-x

```

Operacja: (*) $Rd \leftarrow (X)$
 $PC \leftarrow PC + 1$

(**) $Rd \leftarrow (X)$
 $X \leftarrow X + 1$
 $PC \leftarrow PC + 1$

(***) $X \leftarrow X - 1$
 $Rd \leftarrow (X)$
 $PC \leftarrow PC + 1$

Składnia: (*) LD Rd,X
 (**) LD Rd,X+
 (***) LD Rd,-X

Kod: (*)

1	0	0	1	0	0	0	d	d	d	d	1	1	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

 (**)

1	0	0	1	0	0	0	d	d	d	d	1	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

 (***)

1	0	0	1	0	0	0	d	d	d	d	1	1	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 2

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

```
clr r27      ;zeruj starszą część rejestru X
ldi r26,$60  ;młodsza część rejestru X = $60
ld r0,x+     ;ładuj r0 daną spod adresu $60
             ;i przygotuj następny adres w rejestrze X
ld r1,x      ;ładuj r1 daną spod adresu $61
ldi r26,$63  ;młodsza część rejestru X = $63
ld r2,x      ;ładuj r2 daną spod adresu $63
ld r3,-x     ;przygotuj wcześniejszy adres
             ;w rejestrze X i ładuj r1 daną
             ;spod tego adresu ($62)
```

LD (LDD) – ładuj rejestr pośrednio daną z pamięci SRAM spod adresu wskazywanego przez rejestr indeksowy Y

Ładowanie pośrednie jednego bajtu z obszaru danych do rejestru. W mikrokontrolerach zawierających pamięć SRAM (należy do nich AT90S2313) obszar danych (jest to obszar ciągły) rozciąga się na obszar rejestrów roboczych (adresy od 0 do \$1F), obszar we/wy (adresy od \$20 do \$5F) i wewnętrzną (roboczą) pamięć SRAM (adresy od \$60 do adresu wynikającego z wielkości tej pamięci, w przypadku AT90S2313 do \$DF), a także zewnętrzną pamięć SRAM, jeśli jest obsługiwana przez mikrokontroler. W układach bez pamięci SRAM obszarem danych będą jedynie rejestry robocze. Pamięć EEPROM ma wydzieloną przestrzeń adresową. Dodatkowe informacje można znaleźć w rozdziale 4. Adres danej, która ma być załadowana do rejestru jest wskazywany przez 16-bitowy rejestr indeksowy Y. Dana ta może więc być ulokowana w najniższym obszarze adresowym (64 kB). Jeśli zachodzi potrzeba pobrania danej spoza tego obszaru (dotyczy tylko tych układów, które to umożliwiają, nie dotyczy AT90S2313), należy posłużyć się rejestrem RAMPY z obszaru we/wy.

W wyniku działania rozkazu LD, rejestr indeksowy może pozostać bez zmiany, może być inkrementowany po załadowaniu danej (*post-increment*) lub może być dekrementowany przed załadowaniem danej (*pre-decrement*). W przypadku wykorzystania rejestru Y jako rejestru indeksowego, można realizować pośrednie ładowanie danej z przemieszczeniem – rozkaz LDD. Adres danej jest określony wtedy zawartością rejestru Y i stałym przemieszczeniem, będącym parametrem rozkazu. Ładowanie pośrednie stwarza możliwość wygodnego manipulowania tablicami danych, a także implementacji stosu użytkownika (niezależnego od stosu mikrokontrolera), dla którego wskaźnikiem będzie rejestr Y. Chociaż rejestr Y jest rejestrem 16-bitowym, to w układach mających nie więcej niż 256 bajtów pamięci SRAM (np. AT90S2313) w wyniku operacji indeksowych jest modyfikowana tylko młodsza jego część. W takich układach starsza część rejestru indeksowego może być używana jako rejestr ogólnego przeznaczenia. Rejestr RAMPY jest modyfikowany w przypadku mikrokontrolerów mających przestrzeń danych lub przestrzeń pamięci programu większą niż 64 kB.



Nie wszystkie warianty rozkazu LD zaimplementowano we wszystkich mikrokontrolerach AVR.

UWAGA

Rezultat poniższych operacji jest nieokreślony:

```
ld r28,y+
ld r29,y+
ld r28,-y
ld r29,-y
```

Operacja: (*) $Rd \leftarrow (Y)$
 $PC \leftarrow PC + 1$

(**) $Rd \leftarrow (Y)$
 $Y \leftarrow Y + 1$
 $PC \leftarrow PC + 1$

(***) $Y \leftarrow Y - 1$
 $Rd \leftarrow (Y)$
 $PC \leftarrow PC + 1$

(****) $Rd \leftarrow (Y+c63)$
 $PC \leftarrow PC + 1$

Składnia: (*) LD Rd,Y

(**) LD $Rd,Y+$

(***) LD $Rd,-Y$

(****) LDD $Rd,Y+c63$

Kod: (*)

1	0	0	0	0	0	0	d	d	d	d	1	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

(**)

1	0	0	1	0	0	0	d	d	d	d	1	0	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

(***)

1	0	0	1	0	0	0	d	d	d	d	1	0	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

(****)

1	0	c	0	c	c	0	d	d	d	d	1	c	c	c
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 2

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

```
clr r29 ;zeruj starszą część rejestru Y
ldi r28,$60 ;młodsza część rejestru Y = $60
ld r0,y+ ;ładuj r0 daną spod adresu $60
; i przygotuj następny adres w rejestrze Y
ld r1,y ;ładuj r1 daną spod adresu $61
ldi r28,$63 ;młodsza część rejestru Y = $63
ld r2,y ;ładuj r2 daną spod adresu $63
ld r3,-y ;przygotuj wcześniejszy adres
;w rejestrze Y i ładuj r1 daną
;spod tego adresu ($62)
ldd r4,y+2 ;ładuj r4 daną spod adresu $64
```

LD (LDD) – ładuj rejestr pośrednio daną z pamięci SRAM spod adresu wskazywanego przez rejestr indeksowy Z

Ładowanie pośrednie jednego bajtu z obszaru danych do rejestru. W mikrokontrolerach zawierających pamięć SRAM (należy do nich AT90S2313) obszar danych (jest to obszar ciągły) rozciąga się na obszar rejestrów roboczych (adresy od 0 do \$1F), obszar we/wy (adresy od \$20 do \$5F) i wewnętrzną (roboczą) pamięć SRAM (adresy od \$60 do adresu wynikającego z wielkości tej pamięci, w przypadku AT90S2313 do \$DF), a także zewnętrzną pamięć SRAM, jeśli jest obsługiwana przez mikrokontroler. W układach bez pamięci SRAM obszarem danych będą jedynie rejestry robocze. Pamięć EEPROM ma wydzieloną przestrzeń adresową. Dodatkowe informacje można znaleźć w rozdziale 4. Adres danej, która ma być załadowana do rejestru jest wskazywany przez 16-bitowy rejestr indeksowy Z. Dana ta może więc być ulokowana w najniższym obszarze adresowym (64 kB). Jeśli zachodzi potrzeba pobrania danej spoza tego obszaru (dotyczy tylko tych układów, które to umożliwiają, nie dotyczy AT90S2313), należy posłużyć się rejestrem RAMPZ z obszaru we/wy.

W wyniku działania rozkazu LD, rejestr indeksowy może pozostać bez zmiany, może być inkrementowany po załadowaniu danej (*post-increment*) lub może być dekrementowany przed załadowaniem danej (*pre-decrement*). W przypadku wykorzystania rejestru Z jako rejestru indeksowego, można realizować pośrednie ładowanie danej z przemieszczeniem – rozkaz LDD. Adres danej jest określony wtedy zawartością rejestru Z i stałym przemieszczeniem, będącym parametrem rozkazu. Ładowanie pośrednie stwarza możliwość wygodnego manipulowania tablicami danych, parametrycznego wywoływania podprogramów, a także implementacji stosu użytkownika (niezależnego od stosu mikrokontrolera), dla którego wskaźnikiem będzie rejestr Z. Chociaż rejestr Z jest rejestrem 16-bitowym, to w układach mających nie więcej niż 256 bajtów pamięci SRAM (np. AT90S2313) w wyniku operacji indeksowych jest modyfikowana tylko młodsza jego część. W takich układach starsza część rejestru indeksowego może być używana jako rejestr ogólnego przeznaczenia. Rejestr RAMPZ jest modyfikowany w przypadku mikrokontrolerów mających przestrzeń danych lub przestrzeń pamięci programu większą niż 64 kB.



Nie wszystkie warianty rozkazu LD zaimplementowano we wszystkich mikrokontrolerach AVR.



Rezultat poniższych operacji jest nieokreślony:

ld r30, z+
ld r31, z+
ld r30, -z
ld r31, -z

- Operacja: (*) $Rd \leftarrow (Z)$
 $PC \leftarrow PC + 1$
- (**) $Rd \leftarrow (Z)$
 $Z \leftarrow Z + 1$
 $PC \leftarrow PC + 1$
- (***) $Z \leftarrow Z - 1$
 $Rd \leftarrow (Z)$
 $PC \leftarrow PC + 1$
- (****) $Rd \leftarrow (Z+c63)$
 $PC \leftarrow PC + 1$
- Składnia: (*) LD Rd,Z
- (**) LD Rd,Z+
- (***) LD Rd,-Z
- (****) LDD Rd,Z+c63

Kod:

(*)	1	0	0	0	0	0	0	d	d	d	d	0	0	0	0
(**)	1	0	0	1	0	0	0	d	d	d	d	0	0	0	1
(***)	1	0	0	1	0	0	0	d	d	d	d	0	0	1	0
(****)	1	0	c	0	c	c	0	d	d	d	d	0	c	c	c

Liczba słów: 1 (2 bajty)

Liczba cykli: 2

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

```

clr  r31      ;zeruj starszą część rejestru Z
ldi  r30,$60  ;młodsza część rejestru Z = $60
ld   r0,z+    ;ładuj r0 daną spod adresu $60
      ;i przygotuj następny adres w rejestrze Z
ld   r1,z     ;ładuj r1 daną spod adresu $61
ldi  r30,$63  ;młodsza część rejestru Z = $63
ld   r2,z     ;ładuj r2 daną spod adresu $63
ld   r3,-z    ;przygotuj wcześniejszy adres
      ;w rejestrze Z i ładuj r1 daną
      ;spod tego adresu ($62)
ldd  r4,z+2   ;ładuj r4 daną spod adresu $64

```

LDI – ładuj rejestr bezpośrednio stałą

Ładowanie bezpośrednio 8-bitowej danej do górnego rejestru Rh (R16...R31).

Operacja: $Rh \leftarrow c255$

$PC \leftarrow PC + 1$

Składnia: LDI Rh,c255

Kod:

1	1	1	0	c	c	c	c	h	h	h	h	c	c	c	c
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

```

clr  r31      ;zeruj starszą część rejestru Z
ldi  r30,$f0  ;młodsza część rejestru Z = $f0
lpm  ;pobranie stałej z pamięci programu spod
      ;adresu wskazywanego
      ;zawartością rejestru Z

```

LDS – ładuj rejestr bezpośrednio daną z pamięci SRAM

Bezpośrednie ładowanie jednego bajtu z obszaru danych (pamięć SRAM) do rejestru. W mikrokontrolerach zawierających pamięć SRAM (należy do nich AT90S2313) obszar danych (jest to obszar ciągły) rozciąga się na obszar rejestrów roboczych (adresy od 0 do \$1F), obszar we/wy (adresy od \$20 do \$5F) i wewnętrzną (roboczą) pamięć SRAM (adresy od \$60 do adresu wynikającego z wielkości tej pamięci, w przypadku AT90S2313 do \$DF), a także zewnętrzną pamięć SRAM, jeśli jest obsługiwana przez mikrokontroler. W układach bez pamięci SRAM obszarem danych będą jedynie rejestry robocze. Pamięć EEPROM ma wydzieloną przestrzeń adresową. Dodatkowe informacje można znaleźć w rozdziale 4. Adres danej, która ma być załadowana do rejestru jest wskazywany przez 16-bitową stałą, podawaną jako parametr rozkazu. Zasięg działania rozkazu LDS jest więc ograniczony do aktualnie dostępnego segmentu danych o wielkości 64 kB. Jeśli zachodzi potrzeba pobrania danej spoza tego obszaru (dotyczy tylko tych układów, które to umożliwiają, nie dotyczy AT90S2313), należy posłużyć się rejestrem RAMPD należącym do obszaru we/wy.



Rozkazu LDS nie zaimplementowano we wszystkich mikrokontrolerach AVR. Szczegóły w notach katalogowych.

Operacja: $Rd \leftarrow (adr65535)$

$PC \leftarrow PC + 2$

Składnia: LDS $Rd, adr65535$

Kod:

1	0	0	1	0	0	0	d	d	d	d	0	0	0	0
a	a	a	a	a	a	a	a	a	a	a	a	a	a	a

Liczba słów: 2 (4 bajty)

Liczba cykli: 2

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

```
lds r2,$ff00 ;ładuj r2 daną z obszaru danych,
               ;spod adresu $FF00
add r2,r1     ;dodaj r1 do r2
sts $ff00,r2  ;i zapisz ponownie zmodyfikowaną wartość
```

LPM – ładuj bajt pamięci programu do rejestru

Ładowanie do rejestru Rd jednego bajtu pamięci programu wskazywanego przez rejestr Z. Rozkaz jest wykorzystywany do inicjowania stałych i ich pobierania do dalszych obliczeń przez program. Pamięć programu w mikrokontrolerach AVR jest zorganizowana w 16-bitowe słowa, lecz rejestr Z wskazuje kolejne bajty (nie słowa). Jeśli najmłodszy bit rejestru Z będzie miał wartość „0”, to rejestr ten będzie wskazywał młodszy bajt słowa pamięci programu. Analogicznie wartość „1” będzie informowała, że chodzi o starszy bajt. Zasięg działania rozkazu LPM jest ograniczony do pierwszych 64 kB pamięci programu (32 kslów). W wyniku działania rozkazu LPM, rejestr indeksowy programu może pozostać bez zmiany lub może być inkrementowany. Inkrementacja nie dotyczy rejestru RAMPZ.

Układy zezwalające na autoprogramowanie mogą wykorzystywać rozkaz LPM do odczytywania stanu bitów konfiguracyjnych i bitów zabezpieczających. Szczegółowych informacji należy szukać w notach katalogowych.



Nie wszystkie warianty rozkazu LPM zaimplementowano w każdym mikrokontrolerze AVR. Szczegóły w notach katalogowych (listy rozkazów).



Rezultat poniższych operacji jest nieokreślony:

lpm $r30, z+$
lpm $r31, z+$

Operacja: (*) $R0 \leftarrow (Z)$ (R0 jest rejestrem domyślnym)
 $PC \leftarrow PC + 1$

(**) $Rd \leftarrow (Z)$
 $PC \leftarrow PC + 1$

(***) $Rd \leftarrow (Z)$
 $Z \leftarrow Z + 1$
 $PC \leftarrow PC + 1$

Składnia: (*) LPM
(**) LPM Rd, Z (nie występuje w AT90S2313)
(***) LPM $Rd, Z+$ (nie występuje w AT90S2313)

Kod:	(*)	1 0 0 1 0 1 0 1 1 1 0 0 1 0 0 0
	(**)	1 0 0 1 0 0 0 d d d d 0 1 0 0
	(***)	1 0 0 1 0 0 0 d d d d 0 1 0 1

Liczba słów: 1 (2 bajty)

Liczba cykli: 3

Flagi:	I	T	H	S	V	N	Z	C
	—	—	—	—	—	—	—	—

Przykład:

```
ldi zh,high(tabl<<1) ;inicjuj rejestr indeksowy Z
ldi zl,low(tabl<<1)
lpm r16,z             ;ładuj stałą z pamięci programu
                     ;wskazywaną przez rejestr Z
                     ;do rejestru r16
..
..
tabl: .dw $5876       ;bajt $76 jest wskazywany, gdy ZLSB=0
                     ;bajt $58 jest wskazywany, gdy ZLSB=1
```

LSL – przesun logicznie w lewo

Przesunięcie wszystkich bitów rejestru Rd w lewo. Bit 0 jest zerowany, bit 7 jest przesuwany do znacznika C rejestru SREG. Rozkaz jest przeznaczony do wykonywania operacji mnożenia liczby bajtowej bez znaku przez 2.

Operacja: $C \leftarrow b7 \leftarrow \dots \leftarrow b0 \leftarrow 0$ $PC \leftarrow PC + 1$

Składnia: LSL Rd

Kod: 0 0 0 0 1 1 d d d d d d d d

kod rozkazu LSL Rd jest taki sam jak dla ADD Rd, Rd

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:	I	T	H	S	V	N	Z	C
	—	—	↔	↔	↔	↔	↔	↔

H: przyjmuje wartość bitu Rd(3).

S: wykorzystywana do sprawdzania znaku wyniku.

V: $N \oplus C$ (N i C po przesunięciu).

N: ustawiana, jeśli MSB został ustawiony, w przeciwnym przypadku zerowana.

Z: ustawiana, jeśli wynik jest równy zero, w przeciwnym przypadku zerowana.

C: ustawiana, jeśli bit MSB rejestru przed przesunięciem był ustawiony, w przeciwnym przypadku zerowana.

Przykład:

```
add r0,r4 ;dodaj r4 do r0
lsl r0    ;pomnóż zawartość r0 przez 2
```

LSR – przesun logicznie w prawo

Przesunięcie bitów rejestru Rd w prawo. Bit 7 jest zerowany, bit 0 jest przesuwany do znacznika C rejestru SREG. Rozkaz jest przeznaczony do wykonywania operacji dzielenia przez 2 liczby bajtowej bez znaku.

Operacja: $0 \rightarrow b7 \rightarrow b0 \rightarrow C$

$PC \leftarrow PC + 1$

Składnia: LSR Rd

Kod:

1	0	0	1	0	1	d	d	d	d	d	d	0	1	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	↔	↔	0	↔	↔

S: wykorzystywana do sprawdzania znaku wyniku.

V: $N \oplus C$ (N i C po przesunięciu).

N: 0.

Z: ustawiana, jeśli wynik jest równy zeru, w przeciwnym przypadku zerowana.

C: ustawiana, jeśli bit LSB rejestru przed przesunięciem był ustawiony, w przeciwnym przypadku zerowana.

Przykład:

```
add r0,r4    ;dodaj r4 do r0
lsr r0       ;podziel wartość wpisaną do r0 przez 2
```

MOV – kopiuj zawartość rejestru do rejestru

Rozkaz kopiuje zawartość jednego rejestru do drugiego. Rejestr źródłowy pozostaje niezmienny.

Operacja: $Rd \leftarrow Rs$
 $PC \leftarrow PC + 1$

Składnia: MOV Rd, Rs

Kod:

0	0	1	0	1	1	s	d	d	d	d	s	s	s	s
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

```
mov r16,r0   ;kopiuj r0 do r16
rcall spr    ;wywołaj podprogram
..
spr: cpi r16,$11 ;porównaj zawartość r16 z liczbą $11
ret          ;powrót z podprogramu
```

MOVW – kopiuj zawartość słowa zawartego w parze rejestrów Rs+1:Rs do pary rejestrów Rd+1:Rd

Rozkaz kopiuje zawartość 2-bajтового słowa zapisanego w parze rejestrów Rs+1:Rs do pary rejestrów Rd+1:Rd. Rejestry źródłowe pozostają niezmienione.



Jako rejestry Rd i Rs mogą być użyte tylko rejestry o parzystych adresach (0, 2, ..., 30).



Rozkazu MOVW nie zaimplementowano we wszystkich mikrokontrolerach AVR, w tym w AT90S2313. Szczegóły w notach katalogowych.

Operacja: $Rd+1:Rd \leftarrow Rs+1:Rs$
 $PC \leftarrow PC + 1$

Składnia: MOVW Rd+1:Rd, Rs+1:Rs

Kod:

0	0	0	0	0	0	0	1	d	d	d	d	s	s	s	s
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:

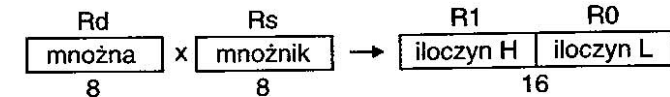
I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

```
movw r17:r16, r1:r0 ;kopiuj r1:r0 do r17:r16
rcall spr           ;wywołaj podprogram
..
spr: cpi r16, $11    ;porównaj zawartość r16 z liczbą $11
..
cpi r17, $32        ;porównaj zawartość r17 z liczbą $32
..
ret                 ;powrót z podprogramu
```

MUL – mnożenie liczb bez znaku

Mnożenie dwóch 8-bitowych liczb bez znaku. W wyniku operacji otrzymuje się liczbę 16-bitową bez znaku.



Przed wykonaniem operacji MUL do rejestru Rd należy wpisać mnożną, do rejestru Rs mnożnik. Obie liczby są traktowane jako 8-bitowe liczby bez znaku. W rezultacie wykonania rozkazu MUL, w rejestrze R1 zostanie zapisany bardziej znaczący bajt 16-bitowego wyniku, a w R0 mniej znaczący bajt wyniku.



Jeśli jako Rd i Rs wybrano rejestry R1 i R0, to ich zawartość zostanie zmodyfikowana.



Rozkazu MUL nie zaimplementowano we wszystkich mikrokontrolerach AVR, w tym w AT90S2313. Szczegóły w notach katalogowych.

Operacja: $R1:R0 \leftarrow Rd \cdot Rs$ (wszystkie liczby bez znaku)
 $PC \leftarrow PC + 1$

Składnia: MUL Rd, Rs

Kod:

1	0	0	1	1	1	s	d	d	d	d	s	s	s	s
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 2

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	↔	↔

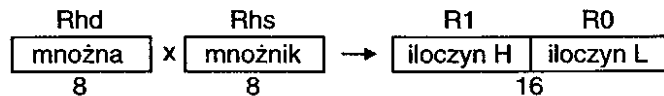
Z: ustawiana, jeśli wynik jest równy \$0000, w przeciwnym przypadku zerowana.
 C: ustawiana, jeśli bit MSB iloczynu (R1(7)) jest ustawiony, w przeciwnym przypadku zerowana.

Przykład:

```
mul r5, r4          ;pomnóż r5 przez r4
movw r4, r0          ;kopiuj iloczyn do pary r5:r4
```

MULS – mnożenie liczb ze znakiem

Mnożenie dwóch 8-bitowych liczb ze znakiem. W wyniku operacji otrzymuje się liczbę 16-bitową ze znakiem.



Przed wykonaniem operacji MULS do rejestru Rh_d należy wpisać mnożną, do rejestru Rh_s mnożnik. Obie liczby są traktowane jako 8-bitowe liczby ze znakiem. W rezultacie wykonania rozkazu MULS, w rejestrze R₁ zostanie zapisany bardziej znaczący bajt 16-bitowego wyniku, a w R₀ mniej znaczący bajt wyniku.



Rozkazu MULS nie zaimplementowano we wszystkich mikrokontrolerach AVR, w tym w AT90S2313. Szczegóły w notach katalogowych.

Operacja: R₁:R₀ ← Rh_d · Rh_s (wszystkie liczby ze znakiem)

PC ← PC + 1

Składnia: MULS Rh_d, Rh_s

Kod:

0	0	0	0	0	0	1	0	d	d	d	d	s	s	s	s
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 2

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	↔	↔

Z: ustawiana, jeśli wynik jest równy \$0000, w przeciwnym przypadku zerowana.

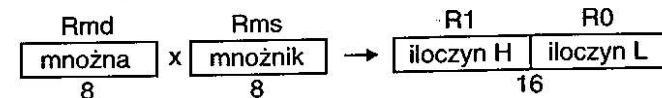
C: ustawiana, jeśli bit MSB iloczynu (R₁(7)) jest ustawiony, w przeciwnym przypadku zerowana.

Przykład:

```
muls r21,r20 ;pomnóż r21 przez r20
movw r20,r0  ;kopiuj iloczyn do pary r21:r20
```

MULSU – mnożenie liczby ze znakiem z liczbą bez znaku

Mnożenie liczby 8-bitowej ze znakiem z liczbą 8-bitową bez znaku. W wyniku operacji otrzymuje się liczbę 16-bitową ze znakiem.



Przed wykonaniem operacji MULSU do rejestru Rm_d należy wpisać mnożną (liczbę 8-bitową ze znakiem), do rejestru Rm_s mnożnik (liczbę 8-bitową bez znaku). W rezultacie wykonania rozkazu MULSU, w rejestrze R₁ zostanie zapisany bardziej znaczący bajt 16-bitowego wyniku (liczba ze znakiem), a w R₀ mniej znaczący bajt wyniku.



Rozkazu MULSU nie zaimplementowano we wszystkich mikrokontrolerach AVR, w tym w AT90S2313. Szczegóły w notach katalogowych.

Operacja: R₁:R₀ ← Rm_d · Rm_s (Rm_d = R₁₆ do R₂₃, Rm_s = R₁₆ do R₂₃)

PC ← PC + 1

Składnia: MULSU Rm_d, Rm_s

Kod:

0	0	0	0	0	0	1	1	0	d	d	d	0	s	s	s
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 2

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	↔	↔

Z: ustawiana, jeśli wynik jest równy \$0000, w przeciwnym przypadku zerowana.

C: ustawiana, jeśli bit MSB iloczynu (R₁(7)) jest ustawiony, w przeciwnym przypadku zerowana.

Przykład:

```

;*****
; Procedura mnożenia dwóch liczb 16-bitowych ze znakiem
; iloczyn jest 32-bitową liczbą ze znakiem
;
; r19:r18:r17:r16 = r23:r22 * r21:r20
;*****
mulsl6x16_32:
    clr    r2
    muls   r23,r21      ;mnóż starsze bajty
    movw   r19:r18,r1:r0 ;przepisz wynik do r19:r18
    mul    r22,r20      ;mnóż młodsze bajty
    movw   r17:r16,r1:r0 ;przepisz wynik do r17:r16
    mulsu  r23,r20
    sbc    r19,r2
    add    r17,r0
    adc    r18,r1
    adc    r19,r2
    mulsu  r21,r22
    sbc    r19,r2
    add    r17,r0
    adc    r18,r1
    adc    r19,r2
    ret

```

NEG – uzupełnienie do dwóch

Zamiana zawartości rejestru Rd na jego uzupełnienie do dwóch. Wartość \$80 pozostaje niezmienną. Zapis liczb w kodzie uzupełnienia do dwóch pozwala interpretować je jako liczby ze znakiem. Wyjaśnia to poniższy przykład:

liczba przed wykonaniem rozkazu NEG	liczba po wykonaniu rozkazu NEG
-3 = \$11111101	-> \$00000011 = 3
-2 = \$11111110	-> \$00000010 = 2
-1 = \$11111111	-> \$00000001 = 1
0 = \$00000000	-> \$00000000 = 0

Operacja: $Rd \leftarrow \$00 - Rd$
 $PC \leftarrow PC + 1$

Składnia: NEG Rd

Kod:

1	0	0	1	0	1	0	d	d	d	d	0	0	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:

I	T	H	S	V	N	Z	C
—	—	↔	↔	↔	↔	↔	↔

- H: ustawiana, jeśli nastąpiło przeniesienie z bitu 3, w przeciwnym przypadku zerowana.
- S: wykorzystywana do sprawdzania znaku wyniku.
- V: ustawiana, jeśli nastąpiło przepełnienie operacji uzupełnienia do dwóch, w przeciwnym przypadku zerowana. Przepełnienie nastąpi tylko wówczas, gdy rejestr Rd po wykonaniu rozkazu będzie miał wartość równą \$80.
- N: ustawiana, jeśli MSB został ustawiony, w przeciwnym przypadku zerowana.
- Z: ustawiana, jeśli wynik jest równy zeru, w przeciwnym przypadku zerowana.
- C: ustawiana, jeśli podczas operacji odejmowania liczby od zera była potrzebna pożyczka, w przeciwnym przypadku zerowana. Flaga C będzie ustawiana zawsze, z wyjątkiem przypadku, gdy zawartość rejestru Rd po wykonaniu rozkazu NEG będzie równa \$00.

Przykład:

```

sub    r11,r0;    ;odejmij r0 od r11
brpl   dod        ;skocz, jeśli wynik dodatni
neg    r11        ;oblicz uzupełnienie do dwóch
dod:   nop        ;dalsze operacje
      ..
      ..

```

NOP – nic nie rób

W wyniku działania rozkazu NOP nie jest podejmowana żadna akcja (oprócz zwiększenia stanu licznika programu umożliwiającego pobranie kolejnego rozkazu). Jest on najczęściej wykorzystywany do uzyskiwania opóźnienia. Zajmuje jeden cykl.

Operacja: $PC \leftarrow PC + 1$

Składnia: NOP

Kod:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

```

clr    r16        ;zeruj r16
ser    r17        ;ustaw r17
out    $18,r16    ;zapisz 0 do portu B
nop    ;nic nie rób
out    $18,r17    ;zapisz jedynki do portu B

```

OR – suma logiczna rejestrów

Obliczenie sumy logicznej rejestrów Rd i Rs. Wynik jest umieszczany w rejestrze Rd.

Operacja: $Rd \leftarrow Rd \vee Rs$
 $PC \leftarrow PC + 1$

Składnia: OR Rd, Rs

Kod:

0	0	1	0	1	0	s	d	d	d	d	s	s	s	s
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	↔	0	↔	↔	—

S: wykorzystywana do sprawdzania znaku wyniku.

V: 0.

N: ustawiana, jeśli MSB został ustawiony, w przeciwnym przypadku zerowana.

Z: ustawiana, jeśli wynik jest równy zeru, w przeciwnym przypadku zerowana.

Przykład:

```

or    r15,r16    ;oblicz sumę logiczną r15 i r16, wynik w r15
bst   r15,6      ;skopiuj bit 6 rejestru r16 do znacznika T
brts  ok         ;skocz, jeśli znacznik T ustawiony
...
ok:   nop        ;dalsze działania

```

ORI – suma logiczna rejestru i stałej

Obliczenie sumy logicznej górnego rejestru Rh i stałej. Wynik jest umieszczany w rejestrze Rh.

Operacja: $Rh \leftarrow Rh \vee c255$
 $PC \leftarrow PC + 1$

Składnia: ORI Rh, c255

Kod:

0	1	1	0	c	c	c	c	d	d	d	d	c	c	c	c
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	↔	0	↔	↔	—

S: wykorzystywana do sprawdzania znaku wyniku.

V: 0.

N: ustawiana, jeśli MSB został ustawiony, w przeciwnym przypadku zerowana.

Z: ustawiana, jeśli wynik jest równy zeru, w przeciwnym przypadku zerowana.

Przykład:

```

ori   r16,$f0    ;ustaw górny półbajt r16
ori   r17,1      ;ustaw bit 0 w r17

```

OUT – zapisz port

Umieszczenie danej z rejestru roboczego Rs do rejestru należącego do obszaru we/wy (porty, timery/liczniki, rejestry konfiguracyjne itp.).

Operacja: $P \leftarrow R_s$
 $PC \leftarrow PC + 1$

Składnia: OUT P, Rs

Kod:

1	0	1	1	1	p	p	s	s	s	s	p	p	p	p
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

```
clr r16 ;zeruj rejestr r16
ser r17 ;ustaw r17
out $18,r16 ;zapisz zera do portu B
nop ;czekaj
out $18,r17 ;zapisz jedynki do portu B
```

POP – pobierz rejestr ze stosu

Rozkaz ładuje bajt ze szczytu stosu mikrokontrolera do rejestru Rd. Przed pobraniem danej ze stosu inkrementowany jest wskaźnik stosu (preinkrementacja).



Rozkazu POP nie zaimplementowano we wszystkich mikrokontrolerach AVR. W mikrokontrolerze AT90S2313 wskaźnikiem stosu jest 8-bitowy rejestr SPL (SPH nie występuje). Szczegóły w notach katalogowych.

Rozkaz POP jest na ogół wykorzystywany z rozkazem PUSH. Stos mikrokontrolera można porównać do stosu talerzy poukładanych jeden na drugim. Operacja na stosie może się odbywać tylko poprzez jego górny element (nie ma możliwości wyciągnięcia talerza ze środka). Trzeba również pamiętać o tym, że elementy są pobierane ze stosu w odwrotnej kolejności niż w tej, w której były na nim odkładane.

Operacja: $SPH:SPL \leftarrow SPH:SPL + 1$
 $Rd \leftarrow (SPH:SPL)$

dla mikrokontrolera AT90S2313:

$SPL \leftarrow SPL + 1$

$Rd \leftarrow (SPL)$

$PC \leftarrow PC + 1$

Składnia: POP Rd

Kod:

1	0	0	1	0	0	0	d	d	d	d	1	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 2

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

```

call proc      ;wywołaj podprogram
..
proc: push r14  ;odłóż na stos r14
push r13       ;odłóż na stos r13
..
..
clr r13        ;zeruj r13
clr r14        ;zeruj r14
..
pop r13        ;odtwórz r13 (pobierając ze stosu)
pop r14        ;odtwórz r14 (pobierając ze stosu)
ret

```

PUSH – odłóż rejestr na stos

Rozkaz zachowuje zawartość rejestru Rd na szczycie stosu mikrokontrolera. Po zapisaniu danej na stosie dekrementowany jest wskaźnik stosu (postdekrementacja).



Rozkazu PUSH nie zaimplementowano we wszystkich mikrokontrolerach AVR. W mikrokontrolerze AT90S2313 wskaźnikiem stosu jest 8-bitowy rejestr SPL (SPH nie występuje). Szczegóły w notach katalogowych.

Rozkaz PUSH jest na ogół wykorzystywany z rozkazem POP. Stos mikrokontrolera można porównać do stosu talerzy poukładanych jeden na drugim. Operacja na stosie może się odbywać tylko poprzez jego górny element (nie ma możliwości wyciągnięcia talerza ze środka). Trzeba również pamiętać o tym, że elementy są pobierane ze stosu w odwrotnej kolejności niż w tej, w której były na nim odkładane.

Operacja: $(SPH:SPL) \leftarrow Rd$
 $SPH:SPL \leftarrow SPH:SPL - 1$

dla mikrokontrolera AT90S2313:

$(SPL) \leftarrow Rd$
 $SPL \leftarrow SPL - 1$
 $PC \leftarrow PC + 1$

Składnia: PUSH Rd

Kod:

1	0	0	1	0	0	1	d	d	d	d	1	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 2

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

```

call proc      ;wywołaj podprogram
..
..
proc: push r14   ;odłóż na stos r14
push r13        ;odłóż na stos r13
..
..
clr r13         ;zeruj r13
clr r14         ;zeruj r14
..
..
pop r13         ;odtwórz r13 (pobierając ze stosu)
pop r14         ;odtwórz r14 (pobierając ze stosu)
ret

```

RCALL – wywołanie podprogramu o adresie względnym

Wywołanie podprogramu o adresie względnym. Adres ulokowania podprogramu w pamięci programu nie może być bardziej odległy od rozkazu wywołującego niż PC-2 kół+1 do PC+2 kół. W mikrokontrolerach, w których pamięć programu jest mniejsza niż dopuszczalny zasięg działania rozkazu RCALL (np. AT90S2313) powyższe ograniczenie oczywiście nie ma znaczenia. Adres powrotu (adres następnego rozkazu po rozkazie wywołania podprogramu) jest zawsze odkładany na stosie. Każdorazowo po odłożeniu jednego bajtu, wskaźnik stosu (SPH:SPL) jest dekrementowany (zmniejszany o jeden). W mikrokontrolerze AT90S2313 wskaźnik stosu jest rejestrem 8-bitowym – SPL (tylko). Argumentem rozkazu RCALL jest adres wywoływane podprogramu, a właściwie przemieszczenia względem adresu bieżącego (stąd wywołanie względne). „Ręczne” obliczanie przemieszczeń byłoby jednak bardzo niewygodne w praktyce. Najczęściej więc stosuje się symboliczny zapis adresu podprogramu, w postaci etykiety. Konkretna wartość przemieszczenia jest później obliczana na etapie asemblacji.

Operacja: (SPH:SPL) ← PC+1 (odłóż adres powrotu na stos)
 SPH:SPL ← SPH:SPL – 2 (w przypadku adresu 16-bitowego (2-bajtowego))

lub

SPH:SPL ← SPH:SPL – 3 (w przypadku adresu 22-bitowego (3-bajtowego))

dla mikrokontrolera AT90S2313:

(SPL) ← PC + 1

SPL ← SPL – 2

PC ← PC + adr2k +1

Składnia: RCALL adr2k

Kod:

1	1	0	1	a	a	a	a	a	a	a	a	a	a	a	a
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 3 dla 16-bitowego adresu wywoływane podprogramu

4 dla 22-bitowego adresu wywoływane podprogramu

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

```

rcall proc    ;wywołaj podprogram
               ;(adres w postaci symbolicznej)
..
..
proc: push r14 ;zachowaj r14 na stosie
..
..
pop r14       ;odtwórz rejestr r14
ret           ;powrót z podprogramu

```

RET – powrót z podprogramu

Rozkaz powoduje inkrementację wskaźnika stosu, a następnie zdjęcie ze stosu danej będącej adresem powrotu z podprogramu i załadowanie jej do licznika programu PC.



W mikrokontrolerze AT90S2313 wskaźnik stosu jest rejestrem 8-bitowym – SPL. Rejestru SPH nie zaimplementowano.

Operacja: dla 16-bitowego PC:
 $SPL:SPL \leftarrow SPL:SPL + 2$
 $PC(15...0) \leftarrow (SPL:SPL)$ dla 16-bitowego PC

dla 22-bitowego PC:
 $SPL:SPL \leftarrow SPL:SPL + 3$
 $PC(21...0) \leftarrow (SPL:SPL)$ dla 22-bitowego PC

dla mikrokontrolera AT90S2313:
 $SPL \leftarrow SPL + 2$
 $PC(15...0) \leftarrow (SPL)$

Składnia: RET

Kod:

1	0	0	1	0	1	0	1	0	0	0	0	1	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 4 dla 16-bitowego adresu
 5 dla 22-bitowego adresu

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

```

call proc     ;wywołaj podprogram
..
..
proc: push r14 ;odłóż na stos r14
..
..
pop r14       ;odtwórz r14 (pobierając ze stosu)
ret

```

RETI – powrót z procedury obsługi przerwania

Powrót z procedury obsługi przerwania wymaga wykonania pewnych czynności, których nie realizuje rozkaz RET. Z tego powodu każda procedura obsługi przerwania powinna być zakończona rozkazem RETI. Powoduje on inkrementację wskaźnika stosu, a następnie zdjęcie ze stosu danej będącej adresem powrotu z podprogramu i załadowanie jej do licznika programu PC. Dodatkowo, automatycznie jest ustawiana flaga globalnego sterowania przerwaniem, powodując ich odblokowanie. Programista musi pamiętać, że tak jak wywołanie procedury obsługi przerwania nie powoduje automatycznego zachowania używanych rejestrów na stosie, tak powrót z przerwania za pomocą rozkazu RETI nie powoduje ich automatycznego zdjęcia ze stosu. Najczęściej jednak pewne rejestry są używane. W szczególności, w wyniku wykonywanych w procedurze operacji może ulec zmianie rejestr statusowy SREG. Jego użycie może być więc niejawnie. Trzeba mieć tego świadomość i bezpieczniej jest zawsze zachowywać go. Pewien kłopot z tym związany polega na tym, że rejestr SREG nie może być argumentem rozkazu PUSH, a tym samym nie może być bezpośrednio zachowany na stosie. Niezbędny jest zabieg pokazany w przykładzie dla rozkazu RETI.



W mikrokontrolerze AT90S2313 wskaźnik stosu jest rejestrem 8-bitowym – SPL. Rejestru SPH nie zaimplementowano.

Operacja: dla 16-bitowego PC:
 $SPL:SPH \leftarrow SPL:SPH + 2$
 $I \leftarrow 1$
 $PC(15...0) \leftarrow (SPL:SPH)$ dla 16-bitowego PC

dla 22-bitowego PC:
 $SPL:SPH \leftarrow SPL:SPH + 3$
 $I \leftarrow 1$
 $PC(21...0) \leftarrow (SPL:SPH)$ dla 22-bitowego PC

dla mikrokontrolera AT90S2313:
 $SPL \leftarrow SPL + 2$
 $I \leftarrow 1$
 $PC(15...0) \leftarrow (SPL)$

Składnia: RETI

Kod:

1	0	0	1	0	1	0	1	0	0	0	1	1	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 4 dla 16-bitowego adresu

5 dla 22-bitowego adresu

Flagi:	I	T	H	S	V	N	Z	C
	1	—	—	—	—	—	—	—

I: 1.

Przykład:

```

ovr2:                ;procedura obsługi przerwania
    push r24          ;zachowaj r24 na stosie
    push r0           ;zachowaj r0 na stosie
    in r0,sreg         ;pobierz rejestr statusowy do r0
    push r0           ;zachowaj SREG na stosie
    ..
    lds r24,$60
    subi r24,$FF       ;niejawna zmiana stanu rejestru SREG
    sts $60,r24
    ..
    pop r0             ;stara wartość SREG ze stosu do r0
    out sreg,r0        ;odtwórz rejestr statusowy
    pop r0             ;odtwórz pozostałe rejestry
    pop r24
    reti              ;powrót z przerwania
  
```

RJMP – skok względny

Skok względny w obrębie 2 kslów od bieżącego adresu (PC-2 kslów+1 do PC+2 kslów). W mikrokontrolerach, w których pamięć programu jest mniejsza niż dopuszczalny zasięg działania rozkazu RJMP (np. AT90S2313) powyższe ograniczenie oczywiście nie ma znaczenia. Argumentem rozkazu RJMP jest adres skoku, a właściwie przemieszczenie względem adresu bieżącego. „Ręczne” obliczanie przemieszczeń byłoby jednak bardzo niewygodne w praktyce. Najczęściej więc stosuje się symboliczny zapis adresu procedury, w postaci etykiety. Konkretna wartość przemieszczenia jest później obliczana na etapie asemblacji.

Rozkaz RJMP może adresować całą pamięć mikrokontrolera AT90S2313.

Operacja: $PC \leftarrow \text{adr2k}$ (stos pozostaje bez zmiany)

Składnia: RJMP adr2k

Kod:

1	1	0	0	a	a	a	a	a	a	a	a	a	a	a	a
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 2

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

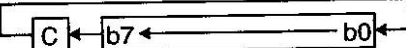
```

    cpi    r16,$42    ;sprawdź, czy r16 ma wartość $42
    breq   error      ;skocz, jeśli tak
    rjmp   ok          ;skok bezwarunkowy
error:  add    r16,r17  ;dodaj r17 do r16
        inc    r16      ;inkrementuj r16
ok:     nop           ;adres skoku rjmp
        ..
        ..

```

ROL – obróć w lewo przez flagę przeniesienia

Przesunięcie wszystkich bitów rejestru Rd o jedną pozycję w lewo. Wskaźnik C rejestru SREG jest przesuwany do bitu 0 rejestru Rd, bit 7 jest przesuwany do znacznika C. Rozkaz ROL w połączeniu z LSL jest wykorzystywany do mnożenia przez 2 wielobajtowych liczb ze znakiem lub bez znaku.

Operacja: 

$PC \leftarrow PC + 1$

Składnia: ROL Rd

Kod:

0	0	0	1	1	1	d	d	d	d	d	d	d	d	d	d
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

kod rozkazu ROL Rd jest taki sam jak dla ADC Rd, Rd

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:

I	T	H	S	V	N	Z	C
—	—	↔	↔	↔	↔	↔	↔

H: Rd(3).

S: wykorzystywana do sprawdzania znaku wyniku.

V: $N \oplus C$ (N i C po przesunięciu).

N: ustawiana, jeśli MSB został ustawiony, w przeciwnym przypadku zerowana.

Z: ustawiana, jeśli wynik jest równy zeru, w przeciwnym przypadku zerowana.

C: ustawiana, jeśli bit MSB rejestru Rd przed przesunięciem był ustawiony, w przeciwnym przypadku zerowana.

Przykład:

```

    lsl    r18          ;pomnóż r19:r18 przez 2
    rol    r19          ;r19:r18 jest liczbą integer ze znakiem
                    ;lub bez znaku
    brcs   cjed          ;skok jeśli flaga C ustawiona
    ..
    ..
cjed:  nop              ;dalsze operacje

```

ROR – obróć w prawo przez flagę przeniesienia

Przesunięcie wszystkich bitów rejestru Rd o jedną pozycję w prawo. Wskaźnik C rejestru SREG jest przesuwany do bitu 7 rejestru Rd, bit 0 jest przesuwany do znacznika C. Rozkaz ROR w połączeniu z rozkazem ASR jest wykorzystywany do dzielenia przez 2 wielobajtowych liczb ze znakiem, a w połączeniu z rozkazem LSR do dzielenia przez 2 wielobajtowych liczb bez znaku.

Operacja: 

$PC \leftarrow PC + 1$

Składnia: ROR Rd

Kod:

1	0	0	1	0	1	0	d	d	d	d	0	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	↔	↔	↔	↔	↔

S: wykorzystywana do sprawdzania znaku wyniku.

V: $N \oplus C$ (N i C po przesunięciu).

N: ustawiana, jeśli MSB został ustawiony, w przeciwnym przypadku zerowana.

Z: ustawiana, jeśli wynik jest równy zero, w przeciwnym przypadku zerowana.

C: ustawiana, jeśli bit LSB rejestru Rd przed przesunięciem był ustawiony, w przeciwnym przypadku zerowana.

Przykład:

```

lsr r19      ;podziel r19:r18 przez 2
ror r18      ;r19:r18 jest liczbą integer bez znaku
brcc czero1  ;skok jeśli flaga C wyzerowana
asr r17      ;podziel r17:r16 przez 2
ror r16      ;r17:r16 jest liczbą integer ze znakiem
brcc czero2  ;skok jeśli flaga C wyzerowana
..
..
czero1: nop   ;dalsze operacje
..
..
czero2: nop   ;dalsze operacje
..
..

```

SBC – odejmij zawartość rejestrów z przeniesieniem

Odejęcie zawartości dwóch rejestrów oraz znacznika C. Wynik jest umieszczany w rejestrze Rd.

Operacja: $Rd \leftarrow Rd - Rs - C$
 $PC \leftarrow PC + 1$

Składnia: SBC Rd, Rs

Kod:

0	0	0	0	1	0	s	d	d	d	d	s	s	s	s
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:

I	T	H	S	V	N	Z	C
—	—	↔	↔	↔	↔	↔	↔

H: ustawiana, jeśli nastąpiło przeniesienie z bitu 3, w przeciwnym przypadku zerowana.

S: wykorzystywana do sprawdzania znaku wyniku.

V: ustawiana, jeśli nastąpiło przepełnienie operacji uzupełnienia do dwóch, w przeciwnym przypadku zerowana.

N: ustawiana, jeśli MSB został ustawiony, w przeciwnym przypadku zerowana.

Z: ustawiana, jeśli wynik jest równy zero, w przeciwnym przypadku zerowana.

C: ustawiana, jeśli wartość bezwzględna zawartości rejestru Rs plus pierwotny stan flagi C jest większa niż wartość bezwzględna zawartości rejestru Rd.

Przykład:

```

                                ;odejmij r1:r0 od r3:r2
sub r2,r0                      ;odejmij młodsze bajty
sbc r3,r1                      ;odejmij starsze bajty z przeniesieniem

```

SBCI – odejmij bezpośrednio stałą od rejestru

Odjęcie bezpośrednie stałej z zakresu 0...255 i przeniesienia od górnego rejestru. Wynik jest umieszczany w tym samym górnym rejestrze Rh.

Operacja: $Rh \leftarrow Rh - c255 - C$
 $PC \leftarrow PC + 1$

Składnia: SBCI Rh,c255

Kod:

0	1	0	0	c	c	c	c	h	h	h	h	c	c	c	c
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:

I	T	H	S	V	N	Z	C
—	—	↔	↔	↔	↔	↔	↔

H: ustawiana, jeśli nastąpiło przeniesienie z bitu 3, w przeciwnym przypadku zerowana.

S: wykorzystywana do sprawdzania znaku wyniku.

V: ustawiana, jeśli nastąpiło przepełnienie operacji uzupełnienia do dwóch, w przeciwnym przypadku zerowana.

N: ustawiana, jeśli MSB został ustawiony, w przeciwnym przypadku zerowana.

Z: ustawiana, jeśli wynik jest równy zero, w przeciwnym przypadku zerowana.

C: ustawiana, jeśli wartość bezwzględna stałej c255 plus pierwotny stan flagi C jest większa niż wartość bezwzględna zawartości rejestru Rh.

Przykład:

```

;odejmij liczbę $4F23 od r17:r16
subi r16,$23 ;odejmij młodszy bajt
sbci r17,$4f ;odejmij starszy bajty z przeniesieniem

```

SBI – ustaw bit w rejestrze we/wy

Ustawienie wyspecyfikowanego bitu w rejestrze we/wy. Rozkaz działa tylko na niskich rejestrach tego obszaru (o adresach z przedziału 0...31 (\$00...\$1F)).

Operacja: $Pl(b) \leftarrow 1$
 $PC \leftarrow PC + 1$

Składnia: SBI Pl,b

Kod:

1	0	0	1	1	0	1	0	p	p	p	p	p	b	b	b
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 2

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

```

#include "2313def.inc"
...
out eear,r0 ;zapisz adres EEPROM-u
sbi eecr,0 ;ustaw bit EERE rejestru EECR
in r1,eedr ;czytaj pamięć EEPROM

```

SBIC – przeskocz, jeśli bit w rejestrze we/wy jest wyzerowany

Rozkaz sprawdza wyspecyfikowany bit w dolnym rejestrze obszaru we/wy i jeśli bit ten ma wartość zero, to następuje przeskoczenie kolejnego rozkazu. Stan rejestru nie ulega zmianie. Rozkaz operuje tylko na dolnych rejestrach przestrzeni we/wy.

Operacja: Jeśli $PI(b)=0$, to $PC \leftarrow PC + 2$ (lub $PC + 3$)
w przeciwnym przypadku $PC \leftarrow PC + 1$

Składnia: SBIC PI,b

Kod:

1	0	0	1	1	0	0	1	p	p	p	p	p	b	b	b
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1, jeśli warunek jest niespełniony (bez przeskoku)

2, jeśli warunek jest spełniony i następuje przeskoczenie rozkazu złożonego z 1 słowa

3, jeśli warunek jest spełniony i następuje przeskoczenie rozkazu złożonego z 2 słów

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

```
.include "2313def.inc"
..
e2zap: sbic  eecr,eeew ;przeskocz następny rozkaz, jeśli EEWE=0
        rjmp  e2zap   ;zapis do EEPROM-u nie zakończony
        nop
        ..
        ..
```

SBIS – przeskocz, jeśli bit w rejestrze we/wy jest ustawiony

Rozkaz testuje wyspecyfikowany bit w dolnym rejestrze obszaru we/wy i jeśli bit ten jest ustawiony, to następuje przeskoczenie kolejnego rozkazu. Stan rejestru nie ulega zmianie. Rozkaz operuje tylko na dolnych rejestrach przestrzeni we/wy.

Operacja: Jeśli $PI(b)=1$, to $PC \leftarrow PC + 2$ (lub $PC + 3$)
w przeciwnym przypadku $PC \leftarrow PC + 1$

Składnia: SBIS PI,b

Kod:

1	0	0	1	1	0	1	1	p	p	p	p	p	b	b	b
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1, jeśli warunek jest niespełniony (bez przeskoku)

2, jeśli warunek jest spełniony i następuje przeskoczenie rozkazu złożonego z 1 słowa

3, jeśli warunek jest spełniony i następuje przeskoczenie rozkazu złożonego z 2 słów

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

```
.include "2313def.inc"
..
czekaj: sbis  pind,0   ;przeskocz następny rozkaz,
                        ;jeśli bit 0 portu D nie jest ustawiony
        rjmp  czekaj  ;czekaj na zmianę stanu
        nop           ;następne operacje
        ..
        ..
```

SBIW – odejmij bezpośrednio stałą od słowa

Bezpośrednie odjęcie stałej z zakresu 0...63 od pary rejestrów RR i umieszczenie wyniku w tych samych rejestrach RR. Możliwe pary rejestrów to: R25:R24, R27:R26, R29:R28, R31:R30.

Operacja: $RRh:RRl \leftarrow RRh:RRl - c63$
 $PC \leftarrow PC + 1$

Składnia: SBIW RR,c63

Kod:

1	0	0	1	0	1	1	1	c	c	r	r	c	c	c	c
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	↔	↔	↔	↔	↔

S: wykorzystywana do sprawdzania znaku wyniku.

V: ustawiana, jeśli nastąpiło przepełnienie operacji uzupełnienia do dwóch, w przeciwnym przypadku zerowana.

N: ustawiana, jeśli MSB został ustawiony, w przeciwnym przypadku zerowana.

Z: ustawiana, jeśli wynik jest równy \$0000, w przeciwnym przypadku zerowana.

C: ustawiana, jeśli wartość bezwzględna stałej c63 jest większa niż wartość bezwzględna zawartości rejestru RR.

Przykład:

```
sbiw r25:r24,1 ;odejmij 1 od r25:r24
sbiw yh:y1,63 ;odejmij 63 od rejestru indeksowego Y=r29:r28
```

SBR – ustaw bity w rejestrze

Ustawienie wyspecyfikowanych bitów w górnym rejestrze Rh. Rozkaz realizuje operację sumy logicznej OR pomiędzy zawartością rejestru Rh i maską będącą parametrem rozkazu. Oznacza to, że zostaną ustawione te bity rejestru, którym odpowiadają bity o wartości „1” w masce.

Operacja: $Rh \leftarrow Rh \vee c255$
 $PC \leftarrow PC + 1$

Składnia: SBR Rh,c255

Kod:

0	1	1	0	c	c	c	c	h	h	h	h	c	c	c	c
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	↔	0	↔	↔	—

S: wykorzystywana do sprawdzania znaku wyniku.

V: 0.

N: ustawiana, jeśli MSB został ustawiony, w przeciwnym przypadku zerowana.

Z: ustawiana, jeśli wynik jest równy zero, w przeciwnym przypadku zerowana.

Przykład:

```
sbr r16,3 ;ustaw bit 0 i 1 w r16
sbr r17,$f0 ;ustaw 4 najstarsze bity w r17
```

SBRC – przeskocz, jeśli bit w rejestrze jest wyzerowany

Rozkaz testuje wyspecyfikowany bit w rejestrze Rs i jeśli bit ten jest wyzerowany, to następuje przeskoczenie kolejnego rozkazu. Stan rejestru nie ulega zmianie.

Operacja: Jeśli $Rs(b)=0$, to $PC \leftarrow PC + 2$ (lub $PC + 3$)
w przeciwnym przypadku $PC \leftarrow PC + 1$

Składnia: SBRC Rs,b

Kod:

1	1	1	1	1	1	0	s	s	s	s	0	b	b	b
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1, jeśli warunek jest niespełniony (bez przeskoku)

- 2, jeśli warunek jest spełniony i następuje przeskoczenie rozkazu złożonego z 1 słowa
- 3, jeśli warunek jest spełniony i następuje przeskoczenie rozkazu złożonego z 2 słów

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

```
sub  r0,r1    ;odejmij r1 od r0
sbrc r0,7     ;przeskocz, jeśli bit 7 w r0 jest wyzerowany
sub  r0,r1     ;wykonuj tylko wtedy,
               ;gdy bit 7 w r0 nie jest wyzerowany
nop          ;dalsze operacje
..
..
```

SBRs – przeskocz, jeśli bit w rejestrze jest ustawiony

Rozkaz testuje wyspecyfikowany bit w rejestrze Rs i jeśli bit ten jest ustawiony, to następuje przeskoczenie kolejnego rozkazu. Stan rejestru nie ulega zmianie.

Operacja: Jeśli $Rs(b)=1$, to $PC \leftarrow PC + 2$ (lub $PC + 3$)
w przeciwnym przypadku $PC \leftarrow PC + 1$

Składnia: SBRs Rs,b

Kod:

1	1	1	1	1	1	1	s	s	s	s	0	b	b	b
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1, jeśli warunek jest niespełniony (bez przeskoku)

- 2, jeśli warunek jest spełniony i następuje przeskoczenie rozkazu złożonego z 1 słowa
- 3, jeśli warunek jest spełniony i następuje przeskoczenie rozkazu złożonego z 2 słów

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

```
sub  r0,r1    ;odejmij r1 od r0
sbrs r0,7     ;przeskocz, jeśli bit 7 w r0 jest ustawiony
neg  r0        ;wykonuj tylko wtedy,
               ;gdy bit 7 w r0 nie jest ustawiony
nop          ;dalsze operacje
..
..
```

SEC – ustaw flagę przeniesienia

Rozkaz ustawia flagę przeniesienia (C) w rejestrze statusowym SREG.

Operacja: $C \leftarrow 1$

$PC \leftarrow PC + 1$

Składnia: SEC

Kod:

1	0	0	1	0	1	0	0	0	0	0	0	1	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	1

C: 1

Przykład:

```
sec          ;ustaw flagę C
adc  r0,r1    ;r0=r0+r1+1
```

SEH – ustaw flagę przeniesienia pomocniczego

Rozkaz ustawia flagę przeniesienia pomocniczego (H) w rejestrze statusowym SREG.

Operacja: $H \leftarrow 1$

$PC \leftarrow PC + 1$

Składnia: SEH

Kod:

1	0	0	1	0	1	0	0	0	1	0	1	1	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:

I	T	H	S	V	N	Z	C
—	—	1	—	—	—	—	—

H: 1

Przykład:

```
seh          ;ustaw flagę przeniesienia pomocniczego
```

SEI – ustaw flagę globalnego sterowania przerwaniami

Rozkaz ustawia flagę globalnego sterowania przerwaniami (I) w rejestrze statusowym SREG. Po wykonaniu rozkazu SEI, przed faktycznym odblokowaniem przerw, będzie wykonany następny po SEI rozkaz.

Operacja: $I \leftarrow 1$
 $PC \leftarrow PC + 1$

Składnia: SEI

Kod:

1	0	0	1	0	1	0	0	0	1	1	1	1	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:

I	T	H	S	V	N	Z	C
1	—	—	—	—	—	—	—

I: 1

Przykład:

```
sei      ;odblokuj przerwania
sleep   ;uśpij procesor, czekaj na przerwanie lub reset
..
..
```

SEN – ustaw flagę wartości ujemnej

Rozkaz ustawia flagę wartości ujemnej (N) w rejestrze statusowym SREG.

Operacja: $N \leftarrow 1$
 $PC \leftarrow PC + 1$

Składnia: SEN

Kod:

1	0	0	1	0	1	0	0	0	0	1	0	1	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	1	—	—

N: 1

Przykład:

```
add     r2,r19    ;dodaj r19 do r2
sen     ;ustaw flagę wartości ujemnej
```

SER – ustaw rejestr

Rozkaz ustawia wszystkie bity w górnym rejestrze Rh.

Operacja: $Rh \leftarrow \$FF$
 $PC \leftarrow PC + 1$

Składnia: SER

Kod:

1	1	1	0	1	1	1	1	s	s	s	s	1	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

```
.include "2313def.inc"
..
clr  r16      ;zeruj r16
ser  r17      ;ustaw r17
out  portb,r16 ;zapisz zera do portu B
nop
out  portb,r17 ;zapisz jedynki do portu B
```

SES – ustaw flagę znaku

Rozkaz ustawia flagę znaku (S) w rejestrze statusowym SREG.

Operacja: $S \leftarrow 1$
 $PC \leftarrow PC + 1$

Składnia: SES

Kod:

1	0	0	1	0	1	0	0	0	1	0	0	1	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	1	—	—	—	—

S: 1

Przykład:

```
add  r2,r19    ;dodaj r19 do r2
ses                      ;ustaw flagę znaku
```

SET – ustaw flagę pomocniczą T

Rozkaz ustawia flagę pomocniczą (T) w rejestrze statusowym SREG.

Operacja: $T \leftarrow 1$
 $PC \leftarrow PC + 1$

Składnia: SET

Kod:

1	0	0	1	0	1	0	0	0	1	1	0	1	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:

I	T	H	S	V	N	Z	C
—	1	—	—	—	—	—	—

T: 1

Przykład:

```
set          ;ustaw flagę T
```

SEV – ustaw flagę przepełnienia uzupełnienia do dwóch

Rozkaz ustawia flagę przepełnienia uzupełnienia do dwóch (V) w rejestrze statusowym SREG.

Operacja: $V \leftarrow 1$
 $PC \leftarrow PC + 1$

Składnia: SEV

Kod:

1	0	0	1	0	1	0	0	0	0	1	1	1	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	1	—	—	—

V: 1

Przykład:

```
add  r2,r19    ;dodaj r19 do r2
sev                ;ustaw flagę V
```

SEZ – ustaw flagę zera

Rozkaz ustawia flagę zera (Z) w rejestrze statusowym SREG.

Operacja: $Z \leftarrow 1$
 $PC \leftarrow PC + 1$

Składnia: SEZ

Kod:

1	0	0	1	0	1	0	0	0	0	0	1	1	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	1	—

Z: 1

Przykład:

```
add r2,r19 ;dodaj r19 do r2
sez        ;ustaw flagę zera
```

SLEEP – przejdź w tryb uśpienia

Wprowadzenie mikrokontrolera w tryb obniżonego poboru mocy zdefiniowanego ustawieniem bitu SM w rejestrze MCUCR (patrz rozdział 4.11). Aby wejście w tryb uśpienia było możliwe, musi być ustawiony bit SE w rejestrze MCUCR. Zalecane jest jego ustawianie bezpośrednio przed wykonaniem rozkazu SLEEP. Zabezpiecza to przed możliwością niepożądanego uśpienia mikrokontrolera. Wyprowadzenie układu z tego stanu jest możliwe tylko poprzez przerwanie lub zerowanie. Programista musi pamiętać, aby przed wykonaniem rozkazu SLEEP odpowiednie przerwanie oraz zezwolenie globalne na przerwanie były odblokowane.

Operacja: wprowadzenie układu w tryb uśpienia

$PC \leftarrow PC + 1$

Składnia: SLEEP

Kod:

1	0	0	1	0	1	0	1	1	0	0	0	1	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

```
.include "2313def.inc"
.def temp=r16
..
ldi temp,$02
out mcucr,temp ;przerwanie od opadającego zbocza INT0
sei           ;odblokuj globalnie przerwanie
..
in temp,mcucr ;przepisz stan rejestru MCUR do temp
ori temp,(1<<SE) ;przygotuj możliwość wprowadzenia trybu
                ;uśpienia
out mcucr,temp
sleep          ;uśpienie procesora
out portb,0    ;zapal LED-y dołączone do portu B po
                ;obudzeniu
..            ;dalsze operacje
..
```

SPM – zapisz pamięć programu

Rozkaz SPM może być używany do kasowania strony pamięci programu, zapisywania strony pamięci programu (która jest już wykasowana) oraz do ustawiania bitu zabezpieczającego *boot loadera*. W niektórych mikrokontrolerach można zapisywać pojedyncze słowo pamięci programu, w innych zaś, przed ostatecznym zapisem należy umieścić dane w specjalnym buforze strony. Zawsze jednak operacja zapisu pamięci programu będzie wiązała się z wykasowaniem całej strony pamięci. W czasie kasowania rejestry RAMPZ i Z służą do adresowania strony. Podczas zapisu, rejestry RAMPZ i Z służą do adresowania strony lub słowa, a w parze rejestrów R1:R0 jest przechowywana dana do zapisu (R1 – starszy bajt, R0 – młodszy bajt).



Rozkazu SPM nie zaimplementowano we wszystkich mikrokontrolerach AVR, w tym w AT90S2313. Szczegóły w notach katalogowych.

- Operacja: (*) (RAMPZ:Z) ← \$FFFF kasuj pamięć programu
PC ← PC + 1
- (**) (RAMPZ:Z) ← R1:R0 zapisz słowo pamięci programu
PC ← PC + 1
- (***) (RAMPZ:Z) ← R1:R0 zapisz bufor pamięci programu
PC ← PC + 1
- (****) (RAMPZ:Z) ← TEMP zapisz dane z bufora do pamięci programu
PC ← PC + 1
- (*****) BLBITS ← R1:R0 ustaw bit zabezpieczający *boot loadera*
PC ← PC + 1

Składnia: SPM

Kod:

1	0	0	1	0	1	0	1	1	1	0	1	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: zależy od operacji

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

```
;Przykład pokazuje zapis strony pamięci
;- procedura zapisuje jedną stronę z pamięci RAM do Flash-a
;- adres pierwszej danej w RAM-ie jest wskazywany przez rejestr Y
;- adres pierwszej danej we Flash-u jest wskazywany przez rejestr Z
;- nie ma obsługi błędów
;- procedura musi być umieszczona w obszarze boot (przynajmniej
;- podprogram do_spm
;- używane rejestry: r0, r1, temp1, temp2, looplo, loophi, spmcval
;- (temp1, temp2, looplo, loophi, spmcval muszą być zadeklarowane
;- przez użytkownika

.equ PAGESIZEB = PAGESIZE * 2      ;PAGESIZE jest wielkością strony
                                   ;w bajtach, nie w słowach

.org SMALLBOOTSTART
zap_str:                            ;kasuj stronę
    ldi    spmcval,(1<<PGERS) + (1<<SPMEN)
    call   do_spm

;przepisanie danych z RAM-u do Flash-a
    ldi    looplo,low(PAGESIZEB)    ;inicjuj zmienną sterującą petli
    ldi    loophi,high(PAGESIZEB)   ;zbędne, jeśli PAGESIZEB<=256
wrloop: ld    r0,y+
    ld     r1,y+
    ldi    spmcval,(1<<SPMEN)
    call   do_spm
    adiw   zh,zl,2
    sbiw   loophi:looplo,2          ;dla PAGESIZEB<=256 użyj subi
    brne   wrloop

;zapis strony
    subi   zl,low(PAGESIZEB)        ;odtwórz wskaźnik
    subi   zh,high(PAGESIZEB)       ;zbędne, jeśli PAGESIZEB<=256
    ldi    spmcval,(1<<PGWRT) + (1<<SPMEN)
    call   do_spm

;odczytaj ponownie i sprawdź
    ldi    looplo,low(PAGESIZEB)    ;inicjuj zmienną sterującą petli
    ldi    loophi,high(PAGESIZEB)   ;zbędne, jeśli PAGESIZEB<=256
    subi   yl,low(PAGESIZEB)        ;odtwórz wskaźnik
    subi   yh,high(PAGESIZEB)
rdloop: lpm    r0,Z+
    ld     r1,y+
    cpse   r0,r1
    jmp     error
    sbiw   loophi:looplo,2          ;dla PAGESIZEB<=256 użyj subi
    brne   rdloop
    ret

do_spm:                            ;wejście: spmcval określa akcję
                                   ;SPM
                                   ;zablokuj przerwania, zachowaj
                                   ;status

    in     temp2,sreg
    cli

    wait:  in     temp1,spmcval
    sbrc   temp1,spmen
    wait

    ;sekwencja SPM

    out    spmcval,spmcval
    out    sreg,temp2              ;odtwórz status
    ret

error: ret
```

ST – zachowaj pośrednio rejestr w pamięci SRAM pod adresem wskazywanym przez rejestr indeksowy X

Pośrednie zachowanie zawartości pojedynczego rejestru w obszarze danych. W mikrokontrolerach z pamięcią SRAM (należy do nich AT90S2313) obszar danych (jest to obszar ciągły) rozciąga się na obszar rejestrów roboczych (adresy od 0 do \$1F), obszar we/wy (adresy od \$20 do \$5F) i wewnętrzną (roboczą) pamięć SRAM (adresy od \$60 do adresu wynikającego z wielkości tej pamięci, w przypadku AT90S2313 do \$DF), a także zewnętrzną pamięć SRAM, jeśli jest obsługiwana przez mikrokontroler. W układach bez pamięci SRAM obszarem danych są jedynie rejestry robocze. Pamięć EEPROM ma wydzieloną przestrzeń adresową. Dodatkowe informacje można znaleźć w rozdziale 4. Adres, pod który ma być zapisana dana z rejestru wskazywany jest przez 16-bitowy rejestr indeksowy X. Dana ta może więc być ulokowana w najniższym obszarze adresowym (64 kB). Jeśli zachodzi potrzeba zachowania danej poza tym obszarem (dotyczy tylko tych układów, które to umożliwiają, nie dotyczy AT90S2313), należy posłużyć się rejestrem RAMPX należącym do obszaru we/wy.

W wyniku działania rozkazu ST, rejestr indeksowy może pozostać bez zmiany, może być inkrementowany po załadowaniu danej (*post-increment*) lub może być dekrementowany przed załadowaniem danej (*pre-decrement*). Stwarza to możliwość wygodnego manipulowania tablicami danych, a także implementacji stosu użytkownika (niezależnego od stosu mikrokontrolera), dla którego wskaźnikiem będzie rejestr X. Chociaż rejestr X jest rejestrem 16-bitowym, to w układach mających nie więcej niż 256 bajtów pamięci SRAM (np. AT90S2313) w wyniku operacji indeksowych jest modyfikowana tylko młodsza jego część. W takich układach starsza część rejestru indeksowego może być używana jako rejestr ogólnego przeznaczenia. Rejestr RAMPX jest modyfikowany w przypadku mikrokontrolerów mających przestrzeń danych lub przestrzeń pamięci programu większą niż 64 kB.



Nie wszystkie warianty rozkazu LD są zaimplementowane we wszystkich mikrokontrolerach AVR.



Rezultat poniższych operacji jest nieokreślony:

```
st x+,r26
st x+,r27
st -x,r26
st -x,r27
```

Operacja: (*) $(X) \leftarrow R_s$
 $PC \leftarrow PC + 1$

(**) $(X) \leftarrow R_s$
 $X \leftarrow X + 1$
 $PC \leftarrow PC + 1$

(***) $X \leftarrow X - 1$
 $(X) \leftarrow R_s$
 $PC \leftarrow PC + 1$

Składnia: (*) ST X,Rs

(**) ST X+,Rs

(***) ST -X,Rs

Kod: (*)

1	0	0	1	0	0	1	s	s	s	s	s	1	1	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

(**)

1	0	0	1	0	0	1	s	s	s	s	s	1	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

(***)

1	0	0	1	0	0	1	s	s	s	s	s	1	1	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 2

Flags:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

```
clr r27 ;zeruj starszą część rejestru X
ldi r26,$60 ;młodsza część rejestru X = $60
st x+,r0 ;zachowaj r0 pod adres $60
;i przygotuj następny adres w rejestrze X
st x,r1 ;zachowaj r1 pod adres $61
ldi r26,$63 ;młodsza część rejestru X = $63
st x,r2 ;zachowaj r2 pod adres $63
st -x,r3 ;przygotuj wcześniejszy adres
;w rejestrze X i zachowaj r3
;pod ten adres ($62)
```

ST (STD) – zachowaj pośrednio rejestr w pamięci SRAM pod adres wskazywany przez rejestr indeksowy Y

Pośrednie zachowanie zawartości pojedynczego rejestru do obszaru danych. W mikrokontrolerach zawierających pamięć SRAM (należy do nich AT90S2313) obszar danych (jest to obszar ciągły) rozciąga się na obszar rejestrów roboczych (adresy od 0 do \$1F), obszar we/wy (adresy od \$20 do \$5F) i wewnętrzną (roboczą) pamięć SRAM (adresy od \$60 do adresu wynikającego z wielkości tej pamięci, w przypadku AT90S2313 do \$DF), a także zewnętrzną pamięć SRAM, jeśli jest obsługiwana przez mikrokontroler. W układach bez pamięci SRAM obszarem danych są jedynie rejestry robocze. Pamięć EEPROM ma wydzieloną przestrzeń adresową. Dodatkowe informacje można znaleźć w rozdziale 4. Adres, pod który ma być zapisana dana z rejestru jest wskazywany przez 16-bitowy rejestr indeksowy Y. Dana ta może więc być ulokowana w najniższym obszarze adresowym (64 kB). Jeśli zachodzi potrzeba zachowania danej poza tym obszarem (dotyczy tylko tych układów, które to umożliwiają, nie dotyczy AT90S2313), należy posłużyć się rejestrem RAMPY należącym do obszaru we/wy.

W wyniku działania rozkazu ST, rejestr indeksowy może pozostać bez zmiany, może być inkrementowany po załadowaniu danej (*post-increment*) lub może być dekrementowany przed załadowaniem danej (*pre-decrement*). W przypadku wykorzystania rejestru Y jako rejestru indeksowego, można realizować zachowywanie rejestru pośrednie z przemieszczeniem – rozkaz STD. Adres danej jest określony wtedy zawartością rejestru Y i stałym przemieszczeniem, będącym parametrem rozkazu. Zachowywanie pośrednie rejestru stwarza możliwość wygodnego manipulowania tablicami danych, a także implementacji stosu użytkownika (niezależnego od stosu mikrokontrolera), dla którego wskaźnikiem będzie rejestr Y. Chociaż rejestr Y jest rejestrem 16-bitowym, to w układach mających nie więcej niż 256 bajtów pamięci SRAM (np. AT90S2313) w wyniku operacji indeksowych modyfikowana jest tylko młodsza jego część. W takich układach starsza część rejestru indeksowego może być używana jako rejestr ogólnego przeznaczenia. Rejestr RAMPY jest modyfikowany w przypadku mikrokontrolerów mających przestrzeń danych lub przestrzeń pamięci programu większą niż 64 kB.



Nie wszystkie warianty rozkazu LD są zaimplementowane we wszystkich mikrokontrolerach AVR.



Rezultat poniższych operacji jest nieokreślony:

```
st y+, r28
st y+, r29
st -y, r28
st -y, r29
```

Operacja: (*) $(Y) \leftarrow R_s$
 $PC \leftarrow PC + 1$

(**) $(Y) \leftarrow R_s$
 $Y \leftarrow Y + 1$
 $PC \leftarrow PC + 1$

(***) $Y \leftarrow Y - 1$
 $(Y) \leftarrow R_s$
 $PC \leftarrow PC + 1$

(****) $(Y) \leftarrow (Y + c63)$
 $PC \leftarrow PC + 1$

Składnia: (*) ST Y, R_s

(**) ST Y+, R_s

(***) ST -Y, R_s

(****) ST Y+c63, R_s

Kod: (*)

1	0	0	0	0	0	1	s	s	s	s	s	1	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

(**)

1	0	0	1	0	0	1	s	s	s	s	s	1	0	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

(***)

1	0	0	1	0	0	1	s	s	s	s	s	1	0	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

(****)

1	0	c	0	c	c	1	s	s	s	s	s	1	c	c	c
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 2

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

```

clr    r29      ;zeruj starszą część rejestru Y
ldi    r28,$60  ;młodsza część rejestru Y = $60
st     y+,r0    ;zachowaj r0 pod adres $60
        ;i przygotuj następny adres w rejestrze Y
st     y,r1     ;zachowaj r1 pod adres $61
ldi    r28,$63  ;młodsza część rejestru Y = $63
st     y,r2     ;zachowaj r2 pod adres $63
st     -y,r3    ;przygotuj wcześniejszy adres
        ;w rejestrze Y i zachowaj r3
        ;pod ten adres ($62)
std    y+2,r4   ;zachowaj r4 pod adres $64

```

ST (STD) – zachowaj pośrednio rejestr w pamięci SRAM pod adres wskazywany przez rejestr indeksowy Z

Pośrednie zachowanie zawartości pojedynczego rejestru do obszaru danych. W mikrokontrolerach zawierających pamięć SRAM (należy do nich AT90S2313) obszar danych (jest to obszar ciągły) rozciąga się na obszar rejestrów roboczych (adresy od 0 do \$1F), obszar we/wy (adresy od \$20 do \$5F) i wewnętrzną (roboczą) pamięć SRAM (adresy od \$60 do adresu wynikającego z wielkości tej pamięci, w przypadku AT90S2313 do \$DF), a także zewnętrzną pamięć SRAM, jeśli jest obsługiwana przez mikrokontroler. W układach bez pamięci SRAM obszarem danych są jedynie rejestry robocze. Pamięć EEPROM ma wydzieloną przestrzeń adresową. Dodatkowe informacje można znaleźć w rozdziale 4. Adres, pod który ma być zapisana dana z rejestru jest wskazywany przez 16-bitowy rejestr indeksowy Z. Dana ta może więc być ulokowana w najniższym obszarze adresowym (64 kB). Jeśli zachodzi potrzeba zachowania danej poza tym obszarem (dotyczy tylko tych układów, które to umożliwiają, nie dotyczy AT90S2313), należy posłużyć się rejestrem RAMPZ należącym do obszaru we/wy.

W wyniku działania rozkazu ST, rejestr indeksowy może pozostać bez zmiany, może być inkrementowany po załadowaniu danej (*post-increment*) lub może być dekrementowany przed załadowaniem danej (*pre-decrement*). W przypadku wykorzystania rejestru Z jako rejestru indeksowego, można realizować zachowywanie rejestru pośrednie z przemieszczeniem – rozkaz STD. Adres danej jest określony wtedy zawartością rejestru Z i stałym przemieszczeniem, będącym parametrem rozkazu. Zachowywanie pośrednie rejestru stwarza możliwość wygodnego manipulowania tablicami danych, a także implementacji stosu użytkownika (niezależnego od stosu mikrokontrolera), dla którego wskaźnikiem będzie rejestr Z. Chociaż rejestr Z jest rejestrem 16-bitowym, to w układach mających nie więcej niż 256 bajtów pamięci SRAM (np. AT90S2313) w wyniku operacji indeksowych modyfikowana jest tylko młodsza jego część. W takich układach starsza część rejestru indeksowego może być używana jako rejestr ogólnego przeznaczenia. Rejestr RAMPZ jest modyfikowany w przypadku mikrokontrolerów mających przestrzeń danych lub przestrzeń pamięci programu większą niż 64 kB.



Nie wszystkie warianty rozkazu ST są zaimplementowane we wszystkich mikrokontrolerach AVR.

UWAGA

Rezultat poniższych operacji jest nieokreślony:

```

st    z+,r30
st    z+,r31
st    -z,r30
st    -z,r31

```

Operacja: (*) $(Z) \leftarrow R_s$
 $PC \leftarrow PC + 1$

(**) $(Z) \leftarrow R_s$
 $Z \leftarrow Z + 1$
 $PC \leftarrow PC + 1$

(***) $Z \leftarrow Z - 1$
 $(Z) \leftarrow R_s$
 $PC \leftarrow PC + 1$

(****) $(Z) \leftarrow (Z+c63)$
 $PC \leftarrow PC + 1$

Składnia: (*) ST Z,Rs

(**) ST Z+,Rs

(***) ST -Z,Rs

(****) ST Z+c63,Rs

Kod: (*)

1	0	0	0	0	0	1	s	s	s	s	s	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

(**)

1	0	0	1	0	0	1	s	s	s	s	s	0	0	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

(***)

1	0	0	1	0	0	1	s	s	s	s	s	0	0	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

(****)

1	0	c	0	c	c	1	s	s	s	s	s	0	c	c	c
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 2

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

```

clr   r31      ;zeruj starszą część rejestru Z
ldi   r30,$60  ;młodsza część rejestru Z = $60
st    z+,r0    ;zachowaj r0 pod adres $60
      ;i przygotuj następny adres w rejestrze Z
st    z,r1     ;zachowaj r1 pod adres $61
ldi   r30,$63  ;młodsza część rejestru Z = $63
st    z,r2     ;zachowaj r2 pod adres $63
st    -z,r3    ;przygotuj wcześniejszy adres
      ;w rejestrze Z i zachowaj r3
      ;pod ten adres ($62)
std   z+2,r4   ;zachowaj r4 pod adres $64

```

STS – zachowaj rejestr bezpośrednio w pamięci SRAM

Bezpośrednie zachowanie zawartości pojedynczego rejestru w obszarze danych (pamięć SRAM). W mikrokontrolerach z pamięcią SRAM (należy do nich AT90S2313) obszar danych (jest to obszar ciągły) rozciąga się na obszar rejestrów roboczych (adresy od 0 do \$1F), obszar we/wy (adresy od \$20 do \$5F) i wewnętrzną (roboczą) pamięć SRAM (adresy od \$60 do adresu wynikającego z wielkości tej pamięci, w przypadku AT90S2313 do \$DF), a także zewnętrzną pamięć SRAM, jeśli jest obsługiwana przez mikrokontroler. W układach bez pamięci SRAM obszarem danych będą jedynie rejestry robocze. Pamięć EEPROM ma wydzieloną przestrzeń adresową. Dodatkowe informacje można znaleźć w rozdziale 4. Adres danej, która ma być załadowana do rejestru jest wskazywany przez 16-bitową stałą, podawaną jako parametr rozkazu. Zasięg działania rozkazu STS jest więc ograniczony do aktualnie dostępnego segmentu danych o wielkości 64 kB. Jeśli zachodzi potrzeba zachowania danej poza tym obszarem (dotyczy tylko tych układów, które to umożliwiają, nie dotyczy AT90S2313), należy posłużyć się rejestrem RAMPD należącym do obszaru we/wy.



Rozkazu STS nie zaimplementowano we wszystkich mikrokontrolerach AVR. Szczegóły w notach katalogowych.

Operacja: $(\text{adr65535}) \leftarrow R_s$
 $PC \leftarrow PC + 2$

Składnia: STS adr65535, Rs

Kod:

1	0	0	1	0	0	1	s	s	s	s	0	0	0	0
a	a	a	a	a	a	a	a	a	a	a	a	a	a	a

Liczba słów: 2 (4 bajty)

Liczba cykli: 2

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

```
lds   r2,$ff00 ;ładuj r2 daną z obszaru danych,
              ;spod adresu $FF00
add   r2,r1    ;dodaj r1 do r2
sts   $ff00,r2 ;i zapisz ponownie zmodyfikowaną wartość
```

SUB – odejmij zawartość rejestrów bez przeniesienia

Od zawartości rejestru Rd jest odejmowana zawartość rejestru Rs. Wynik jest umieszczany w rejestrze Rd.

Operacja: $R_d \leftarrow R_d - R_s$
 $PC \leftarrow PC + 1$

Składnia: SUB Rd, Rs

Kod:

0	0	0	1	1	0	s	d	d	d	d	d	s	s	s	s
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:

I	T	H	S	V	N	Z	C
—	—	↔	↔	↔	↔	↔	↔

H: ustawiana, jeśli nastąpiła pożyczka z bitu 3, w przeciwnym przypadku zerowana.

S: wykorzystywana do sprawdzania znaku wyniku.

V: ustawiana, jeśli nastąpiło przepełnienie operacji uzupełnienia do dwóch, w przeciwnym przypadku zerowana.

N: ustawiana, jeśli MSB został ustawiony, w przeciwnym przypadku zerowana.

Z: ustawiana, jeśli wynik jest równy \$00, w przeciwnym przypadku zerowana.

C: ustawiana, jeśli wartość bezwzględna zawartości rejestru Rs jest większa niż bezwzględna wartość rejestru Rd, w przeciwnym przypadku zerowana.

Przykład:

```
sub   r13,r12   ;odejmij r12 od r13
brne  rozne     ;skocz, jeśli r12 > r13
...
...
rozne: nop      ;dalsze operacje
```

SUBI – odejmij bezpośrednio stałą od rejestru

Od zawartości górnego rejestru Rh jest odejmowana stała. Wynik jest zapisywany w tym samym rejestrze.

Operacja: $Rh \leftarrow Rh - c255$

$PC \leftarrow PC + 1$

Składnia: SUBI Rh,c255

Kod:

0	1	0	1	c	c	c	c	h	h	h	h	c	c	c	c
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:

I	T	H	S	V	N	Z	C
—	—	↔	↔	↔	↔	↔	↔

H: ustawiana, jeśli nastąpiła pożyczka z bitu 3, w przeciwnym przypadku zerowana.

S: wykorzystywana do sprawdzania znaku wyniku.

V: ustawiana, jeśli nastąpiło przepełnienie operacji uzupełnienia do dwóch, w przeciwnym przypadku zerowana.

N: ustawiana, jeśli MSB został ustawiony, w przeciwnym przypadku zerowana.

Z: ustawiana, jeśli wynik jest równy \$00, w przeciwnym przypadku zerowana.

C: ustawiana, jeśli wartość bezwzględna zawartości rejestru Rh jest większa niż bezwzględna wartość stałej c255, w przeciwnym przypadku zerowana.

Przykład:

```
subi r22,$11 ;odejmij liczbę $11 od rejestru r22
brne rozne   ;skocz, jeśli r22 < 11
::
::
rozne: nop    ;dalsze operacje
```

SWAP – zamień półbajty w rejestrze

Zamiana miejscami starszego i młodszego półbajtu rejestru Rd.

Operacja: $Rd(7...4) \leftarrow Rd(3...0)$, $Rd(3...0) \leftarrow Rd(7...4)$

$PC \leftarrow PC + 1$

Składnia: SWAP Rd

Kod:

1	0	0	1	0	1	0	d	d	d	d	0	0	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

```
ldi r16,$a5 ;załaduj rejestr r16 wartością $a5
swap r16    ;zamień półbajty r16=$5a
```

TST – sprawdź zero lub minus

Sprawdzenie, czy zawartość rejestru jest ujemna lub równa zero. Operacja polega na obliczeniu iloczynu logicznego AND rejestru ze sobą. Zawartość rejestru nie ulega jednak zmianie.

Operacja: $Rd \leftarrow Rd \wedge Rd$

$PC \leftarrow PC + 1$

Składnia: TST Rd

Kod:

0	0	1	0	0	0	d	d	d	d	d	d	d	d	d	d
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

kod rozkazu TST jest taki sam jak dla AND Rd, Rd

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:

I	T	H	S	V	N	Z	C
—	—	—	↔	0	↔	↔	—

S: wykorzystywana do sprawdzania znaku wyniku.

V: 0.

N: ustawiana, jeśli MSB został ustawiony, w przeciwnym przypadku zerowana.

Z: ustawiana, jeśli wynik jest równy zero, w przeciwnym przypadku zerowana.

Przykład:

```
tst  r0      ;testuj r0
breq zero    ;skok, jeśli r0=0
...
zero: nop     ;dalsze operacje
```

WDR – zeruj rejestr watchdoga

Wyzerowanie licznika watchdoga. Rozkaz ten powinien być wykonywany systematycznie w czasie określonym przez ustawienie preskalera watchdoga. Szczegóły można znaleźć w rozdziale 6.

Operacja: Restart watchdoga

$PC \leftarrow PC + 1$

Składnia: WDR

Kod:

1	0	0	1	0	1	0	1	0	1	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---

Liczba słów: 1 (2 bajty)

Liczba cykli: 1

Flagi:

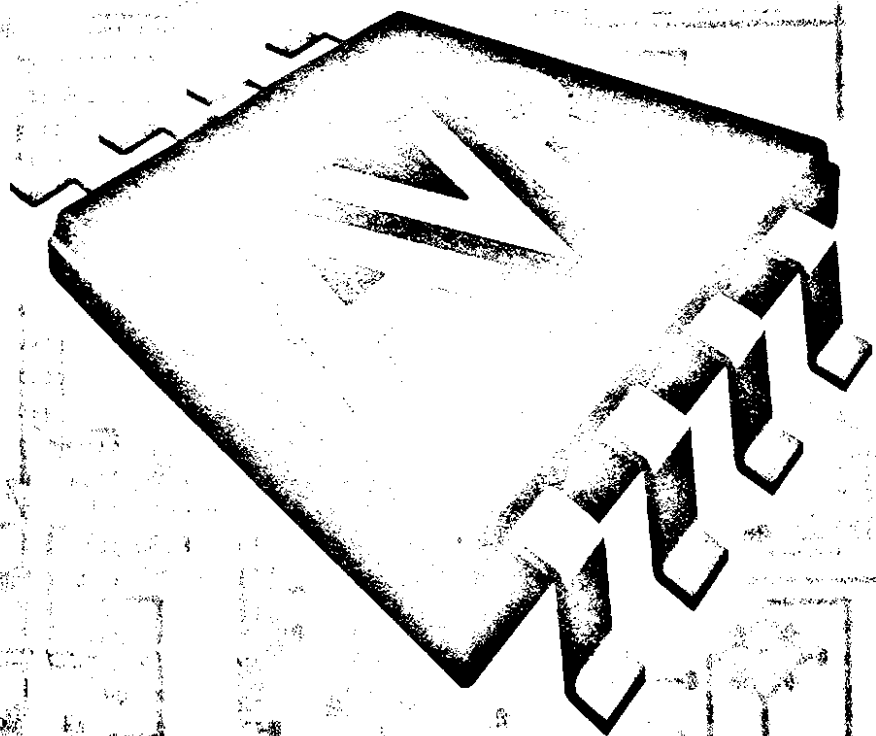
I	T	H	S	V	N	Z	C
—	—	—	—	—	—	—	—

Przykład:

```
.def  temp=r20
.def  temp1=r21
...
ldi   temp,$08      ;przygotowanie wpisu do WDTCSR
in     temp1,sreg    ;zapamiętanie statusu CPU
cli    ;zablokowanie przerwań
wdr    ;reset watchdoga
out    wdtcr,temp    ;start watchdoga z minimalnym timeoutem
out    sreg,temp1    ;odtworzenie statusu CPU
...
wdr    ;zerowanie watchdoga w takim miejscu
;programu, przez który licznik rozkazu musi
;przebiegać w odstępach czasu nie dłuższych
;niż wartość timeoutu
...
ldi   temp,$18      ;procedura wyłączenia watchdoga
in     temp1,sreg    ;zapamiętanie statusu CPU
cli    ;zablokowanie przerwań
wdr    ;WDTOE=1 i WDE=1
out    wdtcr,temp    ;WDTOE=1 i WDE=0 - stop watchdoga, gdyby
;nie było powyższego wpisu do wdtcr,
;watchdog nie zatrzymałby się
out    sreg,temp1    ;odtworzenie statusu CPU
...
...
```

Część 4

Narzędzia i projekty przykładowe



13. Narzędzia projektowe

Do uruchamiania urządzeń zbudowanych z wykorzystaniem mikrokontrolerów nie wystarczy sama lutownica. Wyjątkiem może być sytuacja, w której ograniczymy swoją rolę jedynie do zmontowania układu z użyciem zaprogramowanego mikrokontrolera. Jednak większość Czytelników książki chce zapewne aktywnie uczestniczyć w tworzeniu aplikacji, tzn. projektować sprzęt i pisać oprogramowanie.

W co projektant powinien wyposażać swoją pracownię? Możemy tu wyróżnić trzy rodzaje narzędzi:

- Oprogramowanie niezbędne (lub co najmniej przydatne) do pisania własnych programów, a więc wszelkiego rodzaju edytory tekstów.
- Kompilatory języków programowania. W przypadku mikrokontrolerów AVR największą popularnością cieszą się języki: assembler (programowanie na poziomie kodu maszynowego), Basic (a właściwie jego implementacja znana jako Bascom) i język C.
- Symulatory stosowane do uruchamiania programów przeznaczonych dla mikrokontrolerów, nie wymagające przy tym „kontaktu” z układami fizycznymi. Wykorzystują jedynie ich wirtualne implementacje. Niestety, symulatory rzadko potrafią w pełni oddać faktyczne działanie mikrokontrolera i dlatego – podczas realizacji bardziej złożonych systemów – często konieczne jest użycie emulatora sprzętowego, w którym można uruchomić program z dokładnością do fizycznych sygnałów występujących na wyprowadzeniach układu.

Pierwsza grupa narzędzi jest licznie reprezentowana i z pewnością każdy programista znajdzie odpowiednie narzędzie, spełniające indywidualne preferencje. Powszechnie dostępne – bo występujące w pakiecie oprogramowania systemowego Windows – edytory tekstów takie jak Notepad, WordPad lub popularny edytor Word mogą służyć do pisania źródłowych wersji oprogramowania. Niestety, nawet mimo bardzo zaawansowanych możliwości, jakie ma np. Word, nie zawsze będą one narzędziami optymalnymi z punktu widzenia programisty. Przykładowym edytorem, mającym szansę zaspokoić wymagania praktycznie każdego programisty, jest Kedit (dostępny zarówno w wersji DOS-owej, jak i nowocześniejszej – dla Windows). Jest on dostępny na stronie <http://www.kedit.com/>.

Coraz większa popularność zintegrowanych narzędzi IDE (*Integrated Development Environment*) przyczynia się do spadku zainteresowania samodzielnymi edytorami. W przypadku IDE mamy bowiem do czynienia z komple-

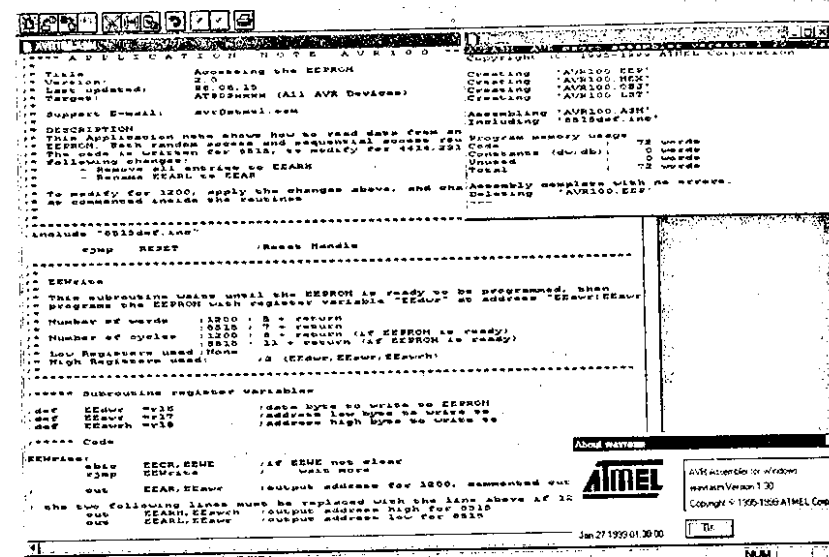
nym, zintegrowanym środowiskiem uruchomieniowym, w którym w jednym pakiecie jest dostępny edytor, kompilator (lub polecenia menu wywołujące kompilator zewnętrzny) i – zazwyczaj – symulator. Najczęściej zintegrowane środowiska uruchomieniowe umożliwiają również bezpośrednie sterowanie popularnych programatorów. Przykładowymi programami tego rodzaju, opisanymi w dalszej części rozdziału, są: AVR Studio i Visual Micro Lab.

13.1. AVR Assembler for Windows

Pierwsze kroki z mikrokontrolerem AVR warto z pewnością wykonać pisząc programy w assemblerze. W dalszych pracach, nawet mimo stosowania języków wysokiego poziomu, bardzo często będzie zachodziła konieczność analizy kodu generowanego przez kompilatory. Czasami wstawki assemblerowe będą pomocne w rozwiązaniu niektórych problemów optymalizacyjnych, np. podczas pisania procedur wrażliwych na zależności czasowe, a także wymagających szczególnie starannego wykorzystywania pamięci mikrokontrolera lub innych jego zasobów. Umiejętność pisania i uruchamiania procedur assemblerowych da tu nieocenione usługi. Firma Atmel bezpłatnie udostępnia użytkownikom mikrokontrolerów AVR assembler *wavras* V 1.30, który jest umieszczony na stronie: http://www.atmel.com/dyn/products/tools.asp?family_id=607. Po kliknięciu na powyższy odsyłacz przechodzimy do działu *Software files* i tam z pozycji *AVR Family Assembler* można pobrać plik *asmpack.exe*. Po jego uruchomieniu zostaną w aktualnym katalogu rozpakowane pliki instalacyjne, wśród których znajduje się *Setup.exe*. Uruchomienie tego programu rozpoczyna instalację, która przebiega niemal automatycznie, w sposób typowy dla systemu Windows. Po zakończeniu instalacji i zrestartowaniu komputera program jest gotowy do pracy. W menu startowym, w grupie *ATMELAVR Tools* znajduje się skrót *AVR Assembler 1.30*. Kliknięcie na niego spowoduje uruchomienie programu, którego główne okno przedstawiono na rysunku 13.1.

Po uruchomieniu programu można przystąpić do edycji nowego programu lub skorzystać z gotowych przykładów. W pierwszym przypadku wybieramy z menu opcję *File>New*, w wyniku czego na ekranie zostaje wyświetlone puste okno edycyjne. Wpisujemy w nim własny program assemblerowy. W drugim przypadku korzystamy z opcji *File>Open*, a następnie wybieramy interesujący nas plik z rozszerzeniem *.asm*. Jest to źródłowa postać programu, który będziemy kompilować. Linie programu pisanego w assemblerze powinny mieć następującą budowę (nawiasy kwadratowe oznaczają opcjonalne występowanie danego elementu):

```
[etykieta] dyrektywa [parametry] [;komentarz]
[etykieta] mnemonik_rozkazu [parametry] [;komentarz]
;komentarz
pusta_linia
```



Rys. 13.1. Główne okno programu AVR Assembler 1.30

Poniżej przedstawiono przykładowy fragment programu napisanego w assemblerze.

Przykład 13.1. Przykład programu napisanego w assemblerze mikrokontrolera AVR

```
etykieta: .EQU Var1=100 ;dyrektywa definiująca stałą Var1 o wartości 100
          .EQU Var2=200 ;analogicznie nadanie Var2 wartości 200
test:     nop
          rjmp test      ;rozkaz skoku (realizacja martwej pętli)
                          ;Linia komentarza, poniżej będzie linia pusta

                          ;Kolejna linia komentarza. Ten komentarz jest
                          ;bardzo długi i dlatego musiał być przełamany
```

Komentarze zawsze rozpoczynają się od znaku średnika. Jeśli nie mieszczą się w jednej linii, to trzeba je przełamywać i nową linię również rozpoczynać od znaku średnika (patrz przykład 13.1). Etykiety mogą być używane jako adresy symboliczne. W wyniku assembleracji zostaną one zastąpione adresami rzeczywistymi. Do wpisywania rozkazów mikrokontrolera korzystamy z ich tzw. mnemoników. Są to symboliczne nazwy operacji mikrokontrolera. Do-

Tab. 13.1. Najważniejsze dyrektywy asemblera AVR Assembler 1.30

Dyrektywa	Opis
BYTE	Rezerwuj bajt dla zmiennej
CSEG	Segment Code
DB	Definiuj stałą – bajt (bajty)
DEF	Definiuj symboliczną nazwę rejestru
DEVICE	Definiuj typ mikrokontrolera
DSEG	Segment Data
DW	Definiuj stałą – słowo (słowa)
ENDM, ENDMACRO	Koniec makrodefinicji
EQU	Przypisz etykietce wartość wyrażenia
ESEG	Segment EEPROM
EXIT	Dyrektywa zatrzymania asemblacji
INCLUDE	Dołącz do pliku źródłowe dane z innego pliku
LIST	Generuj listing po asemblacji
LISTMAC	Włącz rozwijanie makrodefinicji w pliku listingu
NOLIST	Nie generuj listingu po asemblacji
ORG	Ustaw licznik rozkazów
SET	Przypisz etykietce wartość

kładny ich opis zamieszczono w rozdziale 12. W tabelicy 13.1 przedstawiono znaczenie dyrektyw asemblera. Dokładny ich opis można znaleźć w pomocy programu.

Podczas edycji programu można korzystać z typowych udogodnień, jak na przykład: wyszukiwanie, wyszukiwanie z zamianą, kopiowanie lub przenoszenie fragmentów tekstu. Z wybranych, powtarzających się wielokrotnie fragmentów programu mogą być tworzone makrodefinicje. Umożliwiają one zastępowanie tekstu jedną, symboliczną nazwą przypisaną danej makrodefinicji. Nie ma więc potrzeby powtarzania czasami nawet dość długich fragmentów programu. Jakkolwiek wątpliwości nasuwające się podczas prac edycyjnych powinna skutecznie wyjaśnić pomoc zawarta w programie.

Ostatnią czynnością przed przystąpieniem do symulacji programu będzie jego asemblacja. W tym celu wybieramy z menu komendę *Assemble*. W przypadku jednoczesnej edycji kilku plików trzeba pamiętać o wcześniejszym uaktywnieniu tego okna, które zawiera moduł główny programu, zaś przed przystąpieniem do asemblacji wszystkie moduły powinny być wcześniej zapisane na dysku. W wyniku asemblacji i konsolidacji tworzony jest plik relokowalny *.obj*, plik wynikowy *.hex* oraz opcjonalnie plik *.lst* zawierający listing programu z rzeczywistymi adresami umieszczonymi obok rozkazów. Generowanie pliku listingu może być wyłączone dyrektywą *.NOLIST*. W przypadku jej zastosowania plik *.lst* będzie

zawierał jedynie komunikaty o przebiegu asemblacji. Podczas pisania wielomodułowych programów należy pamiętać o tym, że dyrektywa *.include* powoduje dołączenie zawartości osobnego pliku w miejscu jej wystąpienia. Nieumiejętne korzystanie z niej może powodować trudne do zlokalizowania błędy, gdyż nie są one wykrywane na etapie asemblacji (przykład 13.2).

Przykład 13.2. Przykład nieprawidłowego zastosowania dyrektywy *.include*

```
;Zawartość pliku modul1.asm:
.include "2313def.inc"           ;dołączono definicje rejestrów
.def temp=r16                    ;definicja rejestru temp
.cseg                             ;definicja adresu bezwzględnego
.org 0                           ;skok do początku programu pod adres $10

        rjmp    reset

.include "modul2.asm"            ;w tym miejscu jest dołączony fragment
                                ;programu zawarty w pliku modul2.asm

.org 10                           ;definicja adresu bezwzględnego
reset:   ldi     temp,low(RAMEND) ;temp=adres stosu
        out     SPL,temp         ;ustaw wskaźnik stosu
        ldi     temp,$a5
tu:      rcall   pprog            ;wywołaj podprogram (jego treść jest
                                ;zamieszczona w pliku modul2.asm)
        inc     temp             ;inkrementuj zmienną temp
        nop
        rjmp    tu

;Zawartość pliku modul2.asm:
pprog:   nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        out     portb,temp
        ret
```

Umieszczenie podprogramu *pprog* w miejscu wystąpienia dyrektywy *.include "modul2.asm"* jest w powyższym przykładzie błędem. Długość tego podprogramu jest zbyt duża, w wyniku czego następujący po nim rozkaz *ldi temp,low(RAMEND)* umiejscowiony na sztywno pod adresem \$10 za pomocą dyrektywy *.org 10*, tym samym nadpisze ostatnie dwa rozkazy podprogramu *pprog*. W konsekwencji nie wystąpią one w ogóle w programie. Aby uniknąć takiej sytuacji należałoby albo przenieść wspomnianą dyrektywę *.include* na koniec modułu 1 lub nie ustalać bezwzględnego adresu dyrektywą *.org 10*.

W oknie komunikatów (*Message*) są sygnalizowane wszystkie wykryte błędy kompilacji. Ustawienie kursora na wybrany komunikat powoduje podświetlenie w oknie edycyjnym linii programu podejrzanej o błąd. Należy się jej uważnie przyjrzeć, usunąć nieprawidłowość i powtórnie wykonać asemblację programu. Dopiero po uzyskaniu wyniku *Assembly complete with no errors* można przystąpić do dalszych czynności związanych z uruchamianiem programu. Będą one opisane w dalszej części rozdziału (w podrozdziale 13.3.1).

13.2. Kompilator języka C – AVR-GCC wersja 3.2

Wyższość języków wysokiego poziomu nad językami niskopoziomowymi jest bezdyskusyjna. Gdyby tak nie było najprawdopodobniej w ogóle by nie powstały, a programiści tworzyliby programy na poziomie pojedynczych bitów. Program napisany w języku wysokiego poziomu jest czytelniejszy od asemblerowego, umożliwia skrócenie czasu potrzebnego na zakodowanie algorytmu, zdecydowanie ułatwia programowanie złożonych operacji arytmetycznych. Mimo, że w językach wysokiego poziomu problemy są traktowane w sposób bardziej abstrakcyjny niż w asemblerze, możliwe jest precyzyjne sterowanie urządzeniami peryferyjnymi dołączonymi do mikrokontrolera i wykorzystywanie jego zasobów niemal tak precyzyjne, jak za pomocą języków niskiego poziomu. Języki wysokiego poziomu nie nadają się jednak do realizacji wszystkich zadań. Największą bolączką programistów jest problem z optymalizacją kodu wynikowego przejawiający się zbyt dużą wielkością programu lub zbyt długim czasem wykonywania się poszczególnych procedur. W krytycznych sytuacjach programista musi skorzystać z możliwości umieszczania w programie pisanym w języku wysokiego poziomu wstawek asemblerowych. Metoda taka na ogół rozwiązuje większość problemów, a jeśli nie, oznacza to zazwyczaj, że trzeba sięgnąć po silniejszy mikrokontroler. We wcześniejszych rozdziałach książki przedstawiono przykłady programów pisanych zarówno w asemblerze, jak i w języku C. O wyborze tego języka zadecydowało m.in. to, że jest dostępny jego bezpłatny kompilator w wersji bez ograniczeń. Mowa tu o AVR-GCC. Jest to dość interesujący – z punktu widzenia sposobu jego tworzenia – produkt. Nie ma on jednego autora, każdy kto czuje się na siłach może zostać jednym z jego wielu twórców. Nad synchronizacją prac czuwa specjalna 14-osobowa grupa zwana *GCC Steering Committee*. Proponuje ona tematy do realizacji i zbiera wyniki prac programistów z całego świata. Szczegóły można znaleźć na stronie

<http://gcc.gnu.org/>. Taka formuła tworzenia programu powoduje, że ewoluuje on dość szybko. Niestety, nie zawsze udaje się zapewnić pełną kompatybilność nowych wersji ze starszymi. Niedogodności te z pewnością rekompensuje fakt, że AVR-GCC jest produktem darmowym.

13.2.1. Instalacja kompilatora

Kompilator AVR-GCC można pobrać ze strony <http://www.avrfreaks.net/AVRGCC>. Użytkownicy modemów nie będą zachwyceni, gdyż do ściągnięcia jest plik o objętości ok. 12 MB. Trzeba za to przyznać, że geograficznie najbliższy Polsce serwer mieszczący się w Dublinie działa bardzo sprawnie i zwykle gwarantuje dobry transfer. Pobrany plik ma nazwę np. *WinAVR-2002-06-25_FREAKS.exe*, w której – jak widać – jest ukryta data powstania pobieranej wersji kompilatora. Najprawdopodobniej, w chwili ukazania się książki na rynku, będą już dostępne nowsze wersje kompilatora. Uruchomienie tego programu spowoduje rozpoczęcie procedury instalowania kompilatora. Po zaznajomieniu się i zaakceptowaniu warunków licencji (naciskamy klawisz *I Agree*) ukazuje się okno, w którym jest proponowany katalog docelowy – domyślnie *C:\WINAVR*. W razie konieczności można go zmienić. Naciskając klawisz *Install* rozpoczynamy właściwą instalację. Po jej zakończeniu wyświetlana jest zawartość pliku *readme.txt*. Program nie wymaga żadnych dodatkowych czynności konfiguracyjnych, jest gotowy do pracy zaraz po restarcie komputera. Będzie on wykorzystywany najczęściej w połączeniu z jakimś symulatorem, np. AVR Studio, który musimy zainstalować na swoim komputerze.

13.3. AVR Studio wersja 3.56

AVR Studio to – jak już wiemy – programowy symulator mikrokontrolerów AVR. Integruje on w sobie również asembler, umożliwiając tym samym uruchamianie niskopoziomowych programów bez konieczności instalowania dodatkowych programów narzędziowych. Jest on udostępniony bezpłatnie przez firmę Atmel – na stronie http://www.atmel.com/dyn/products/tools.asp?family_id=607. W chwili pisania książki dostępne były dwie wersje programu: 3.56 i 4.0. Z uwagi na planowane wykorzystywanie AVR Studio wraz z kompilatorem AVR-GCC zalecane jest zainstalowanie wersji 3.56, przystosowanej do łatwej integracji obu programów. W tym celu na wymienionej stronie należy wskazać odnośnik *AVR Studio 3.5*, a na-

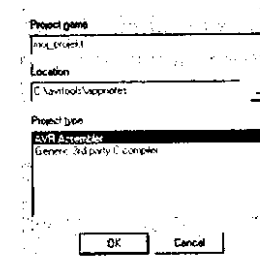
stępnie odnośnik powodujący pobranie pliku *astudio3.exe*. Jest to samorozpakowujące się archiwum. Jego uruchomienie powoduje umieszczenie we wskazanym przez użytkownika katalogu kompletu plików instalacyjnych. Właściwa instalacja rozpoczyna się po uruchomieniu programu *setup.exe* i przebiega automatycznie. Program jest gotowy do pracy natychmiast po zakończeniu instalacji.

13.3.1. Przygotowanie programów pisanych w asemblerze do symulacji w AVR Studio 3.56

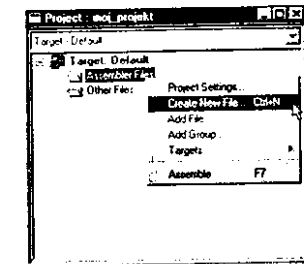
Program AVR Studio może być wykorzystywany do testowania programów przygotowanych za pomocą zewnętrznego asemblera, np. opisanego na początku niniejszego rozdziału. Można pisać własny program od podstaw, korzystając z wbudowanego edytora i asemblera, można również symulować programy pisane w językach wysokiego poziomu, np. AVR-GCC. W pierwszym przypadku wystarczy za pomocą polecenia *File>Open* otworzyć jeden z plików wynikowych asemblera. Korzystając z przykładu 13.2 może to być np. *modul1.hex* lub *modul1.obj*. Otwierając plik *.hex* w oknie roboczym uzyskujemy podgląd jego zdisasemblowanej postaci, niestety bez nazw symbolicznych (jedynie mnemoniki mikrokontrolera i rzeczywiste adresy). Praca w takim przypadku jest bardzo niewygodna, nadaje się w zasadzie jedynie do podglądania tego, co robi mikrokontroler w obcych programach, gdzie mamy do dyspozycji jedynie odczytaną zawartość pamięci Flash mikrokontrolera. Otwierając plik *.obj* uzyskujemy już pełen komfort pracy, gdyż w oknie roboczym dostajemy pełną postać źródłową programu wraz z adresami symbolicznymi. Jeśli w trakcie pracy z AVR Studio za pomocą zewnętrznego asemblera zostanie zmodyfikowany program wynikowy mikrokontrolera, to zmiana ta zostanie wykryta i pojawi się informacja, której zaakceptowanie powoduje przeładowanie (odświeżenie) plików roboczych. Opisany wyżej tryb pracy, choć możliwy, nie znajduje większego zastosowania wobec faktu wyposażenia AVR Studio we własny asembler. Bardziej naturalnym sposobem pracy wydaje się więc pisanie i uruchamianie programu od początku do końca w tym systemie.

Na początku konieczne jest utworzenie projektu. W tym celu należy wybrać polecenie *Project>New*, a następnie w oknie dialogowym podać nazwę, np. *moj_projekt* i wybrać typ AVR Assembler (rysunek 13.2).

Kolejną czynnością jest utworzenie nowego pliku źródłowego lub dodanie do projektu pliku już istniejącego. Wywołanie odpowiedniego okna dialogowe-



Rys. 13.2. Okno wstępnej konfiguracji projektu



Rys. 13.3. Dołączanie plików źródłowych do projektu

go następuje po kliknięciu prawym przyciskiem myszki, gdy podświetlony jest folder *Assembler files* (rysunek 13.3) i wybraniu opcji odpowiednio: *Create New File...* lub *Add File...*. W wyniku przeprowadzenia tych czynności w folderze tym pojawiają się wskazane pliki lub – w przypadku nowego pliku – jest otwierane okno edycyjne, w którym można wpisywać źródło programu.

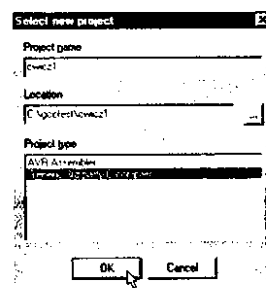
Po zdefiniowaniu projektu i przed rozpoczęciem dalszych prac warto wykonać polecenie *Project>Save* powodujące zachowanie projektu na dysku. Po zakończeniu wszystkich prac edycyjnych można podjąć próbę wygenerowania wersji wykonywalnej *.hex*. Służy do tego celu polecenie *Project>Build and run*.

W pierwszej fazie prac nad projektem, gdy spodziewane są liczne błędy formalne, wygodniej będzie skorzystać z polecenia *Project>Assemble*, powodującego jedynie przeprowadzenie asemblacji bez tworzenia pliku wynikowego *.hex*. Wynik tej operacji jest wyświetlany w oknie komunikatów, które pojawia się po zakończeniu asemblacji. Warunkiem koniecznym do rozpoczęcia następnego etapu prac uruchomieniowych jest wyeliminowanie wszystkich błędów, o czym informuje komunikat: *Assembly complete with no errors*. Jeśli tak się nie stanie, to zostanie wyświetlona lista błędów. Klikając na poszczególne pozycje tej listy powodujemy przeniesienie kursora do linii programu związanej ze wskazanym błędem, gdzie można wprowadzić odpowiednie poprawki. Po „wyczyszczeniu” programu można przystąpić do symulacji. Opis tego etapu prac będzie zamieszczony w dalszej części rozdziału.

13.3.2. Integracja programu AVR Studio 3.56 z kompilatorem AVR-GCC

O ile praca z symulatorem AVR Studio podczas uruchamiania programów assemblerowych przebiega w sposób naturalny i raczej bez większych problemów, to wykorzystywanie go w połączeniu z kompilatorem AVR-GCC nie jest – niestety – już takie proste. Uwaga ta dotyczy głównie zasad tworzenia projektów wykorzystujących pliki źródłowe pisane w języku AVR-GCC, późniejsza symulacja przebiega już normalnie. Najlepiej zilustruje to poniższy przykład.

W celu utworzenia nowego projektu należy wybrać z menu polecenie *Project>New*. Na ekranie zostanie wyświetlone okno, w którym trzeba podać nazwę projektu np. *cwicz1*, wskazać lokalizację jego plików, a następnie podświetlić opcję *Generic 3rd party Compiler*. Wymagane jest, aby wszystkie pliki projektu znajdowały się w jednym katalogu. Niech to będzie np. *c:\gcc\test\cwicz1* (jest to przykład dokładnie opisany w rozdziale 14.). Taką ścieżkę dostępu należy więc wpisać w polu *Location* okna pokazanego na rysunku 13.4.



Rys. 13.4. Okno konfiguracji projektu

Po zaakceptowaniu ustawień klawiszem *OK* wskazane jest zapisanie projektu na dysku, np. przez naciśnięcie trzeciej od lewej strony ikony na pasku narzędziowym. Dołączanie lub tworzenie plików źródłowych przebiega tak samo jak w przypadku programów assemblerowych. Pliki programu **.c* należy umieszczać w folderze *Source files*, pliki nagłówkowe **.h* w folderze *Header files*, natomiast plik *makefile* w folderze *Other files*. Wskazanie dowolnego pliku i dwukrotne kliknięcie spowoduje otwarcie okna edycyjnego, w którym można dokonywać zmian zawartości edytowanego pliku. Po kliknięciu prawym przyciskiem myszki na podświetlonym folderze *Source Files* wybieramy opcję *Create New File*. W oknie, które teraz zostanie wyświetlone należy wpisać nazwę z rozszerzeniem pliku źródłowego programu. W naszym przykładzie jest to *cwicz1.c*. Pole *Location* pozostaje bez zmian. W otwartym oknie edycyjnym można teraz wprowadzić tekst programu. Zamknięcie okna spowoduje automatyczne zachowanie efektów pracy na dysku. Zakończenie czynności edycyjnych nie spowoduje automatycznego dołączenia pliku do projektu, trzeba to zrobić ręcznie. Do kompilacji programu niezbędny będzie jeszcze plik *makefile*, pełniący szczególną rolę w projektach AVR-GCC. Przykład takiego pliku jest umieszczany podczas instalacji kompilatora w ka-

talogu *c:\avrgcc\avrfreaks*. Można go skopiować do naszego katalogu roboczego i odpowiednio do potrzeb modyfikować. Dwukrotne kliknięcie na nim spowoduje jak zwykle rozpoczęcie edycji. W omawianym przykładzie należy doprowadzić go do postaci:

```
# Simple Makefile Volker Oth (c) 1999
# edited by AVRfreaks.net nov.2001
#### change these lines according to your project ####
#put the name of the target mcu here (at90s8515, at90s8535, attiny22,
atmega603 etc.)
MCU = at90s2313
#put the name of the target file here (without extension)
TRG = cwicz1
#put your C sourcefiles here
SRC = $(TRG).c
#put additional assembler source file here
ASRC =
#additional libraries and object files to link
LIB =
#additional includes to compile
INC =
#assembler flags
ASFLAGS = -Wa, -gstabs
#compiler flags
CPFLAGS = -g -Os -Wall -Wstrict-prototypes -Wa, -ahlms=$(<:.c=.lst)
#linker flags
LDFLAGS = -Wl, -Map=$(TRG).map, -cref

#### you should not need to change the following line ####
include $(AVR)/avrfreaks/avr_make

#### dependencies, add any dependencies you need here ####
$(TRG).o : $(TRG).c
```

Linie rozpoczynające się znakiem „#” to komentarze i nie są istotne z punktu widzenia kompilatora. Pozostałe zawierają informacje, mające wpływ na przebieg kompilacji. Są to m.in. typ mikrokontrolera (MCU), nazwa pliku wynikowego bez rozszerzenia (TRG), lista plików źródłowych (SRC) **.c*, a także ewentualne informacje o dołączanych modułach assemblerowych, bibliotekach. Ich opis wykracza poza ramy tej książki, więcej wiadomości należy szukać w dokumentacji kompilatora. W tym miejscu warto jedynie wspomnieć o linii pliku *makefile*, która wpływa na nieoczekiwane czasami zachowanie się programu wynikowego. Zawiera ona m.in. opcję optymalizacji kompilatora *-O*. W sytuacjach ekstremalnych, na skutek ustawienia zbyt wysokiego poziomu optymalizacji niektóre linie programu źródłowego mogą po prostu nie znaleźć się w programie wynikowym, jeśli tylko kompilator uzna, że są zbędne. Dla przykładu krótki program przedstawiony w przykładzie 13.3 i skompilowany z opcją *-O2* pominię całkowicie pętlę *for*. Opcja *-O1* spowoduje, że pętla będzie się wprawdzie wykonywała, ale po wyjściu z niej zmienna *licznik* pozostanie bez zmiany. Spowodowane

jest to tym, że zmienna *licznik* nie jest w dalszej części programu wykorzystywana. Dopiero, gdy zostanie zastosowany parametr `-O0` program wykona się zgodnie z zamiarem autora.

Przykład 13.3. Przykładowy program ilustrujący wpływ opcji optymalizacji kompilatora

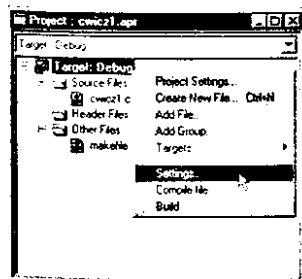
```
int main(void)
{
    unsigned char i, licznik;

    licznik=0x00;
    for(i=0; i<10; i++)
    {
        licznik++;
    }
    while(1);
}
```

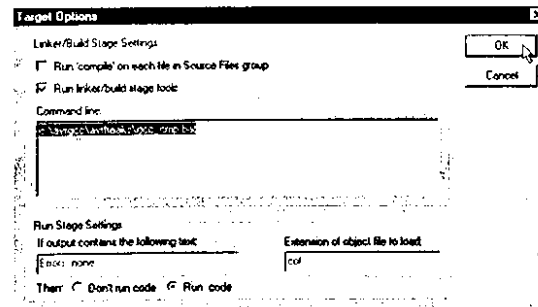
Nie jest to jeszcze koniec czynności niezbędnych do przeprowadzenia kompilacji programu. W kolejnym kroku należy ustawić parametry w zakładce *Target Debug* okna projektu. W tym celu należy kliknąć prawym klawiszem myszki na podświetlony folder *Target Debug*, w wyniku czego zostaje wyświetlone menu, z którego trzeba – tym razem lewym klawiszem – wybrać opcję *Settings* (rysunek 13.5). W oknie, które zostanie wyświetlone, należy zmienić trzy ustawienia:

- należy odznaczyć opcję *Run 'compile' on each file in Source Files group*,
- w polu *Command line* powinna być wpisana linia polecenia wywołującego kompilator. Ma ona postać: `c:\avrgcc\avr\freake\gcc_cmp.bat`,
- w polu *Extension of object file to load* wpis *obj* należy zastąpić wpisem *cof*.

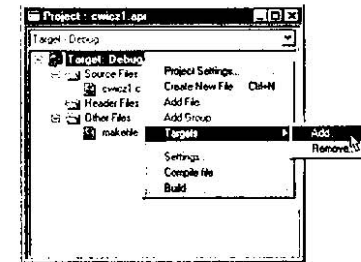
Powyższe zmiany przedstawiono na rysunku 13.6. Dopiero teraz staje się możliwe kompilowanie programów pisanych w języku AVR-GCC. Kompilator w pewnych sytuacjach wymaga jednak, aby przed rozpoczęciem pracy



Rys. 13.5. Ustawianie parametrów zakładki *Target Debug*



Rys. 13.6. Zalecane ustawienia parametrów kompilatora



Rys. 13.7. Tworzenie zakładki *Target Clean*

z katalogu roboczego były wykasowane pliki wygenerowane wcześniej. Czynności te wykonywane ręcznie są dość uciążliwe dla programisty, warto więc dokonać jeszcze jednego ustawienia programu AVR Studio uwalniającego nas od nich. Trzeba w tym celu utworzyć dodatkową zakładkę, której zachowując angielską terminologię, można nadać nazwę np. *Clean*. Postępujemy podobnie jak wcześniej, przy czym tym razem należy wybrać w menu opcję *Target>Add* (rysunek 13.7).

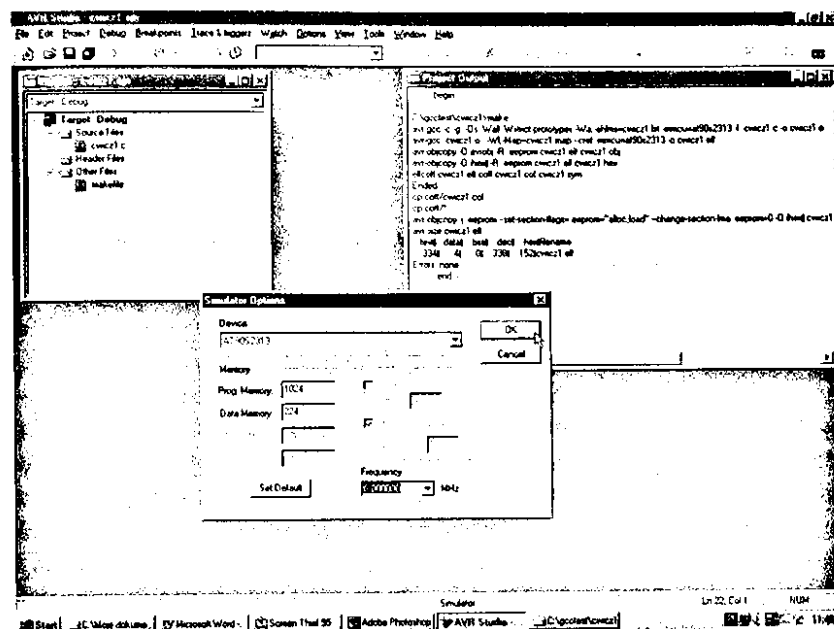
Następnie w polu *Name* wpisujemy przyjętą nazwę *Clean*, a z listy rozwijanej *Copy settings from:* wybieramy pozycję *Debug*. Pozostaje jeszcze ustawienie parametrów zakładki *Target Clean*. W oknie projektu, w górnej jego części, znajduje się lista, z której trzeba wybrać pozycję *Target Clean*. Dalej postępujemy podobnie, jak przy zakładce *Target Debug*. Tym razem linia poleceń powinna mieć postać: `c:\avrgcc\avr\freake\gcc_cmp.bat clean`, a obydwa pola *Run Stage Settings* muszą być puste.

Pracy trochę było, najwyższy już czas sprawdzić, czy to wszystko działa.

13.3.3. Symulacja programów w AVR Studio V. 3.56

Bezbledny wynik kompilacji nie dowodzi oczywiście, że program jest już gotowy do zapisania w pamięci mikrokontrolera. Oznacza jedynie, że jest pozbawiony błędów formalnych takich jak: literówki, nieprawidłowe użycie mnemoników assemblerowych lub zła składnia instrukcji języka C, niezgodność parametrów rozkazów, zdublowane adresy symboliczne itp. Błędy takie popełnia się często, ale na ogół też bardzo łatwo jest je usunąć z programu. Połowę zadań wykonuje przecież kompilator. Znacznie gorzej jest z błędami logicznymi. Niestety, musimy być w znacznej mierze zdani na własne siły, wspierając się jedynie takim programem, jak np. AVR Studio.

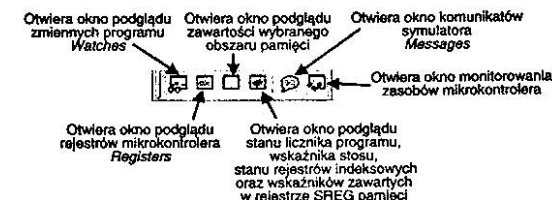
Po wykonaniu wszystkich czynności konfiguracyjnych opisanych wyżej, można przeprowadzić kompilację programu źródłowego. W tym celu należy z menu wybrać polecenie *Project>Build and run* lub jednocześnie nacisnąć klawisze `Ctrl` i `F7`. W przypadku błędów kompilacji w oknie *Project Output* będą wyświetlane numery błędów i linii, w których wykryty błąd występuje.



Rys. 13.8. Okno ustawiania parametrów uruchamiania systemu

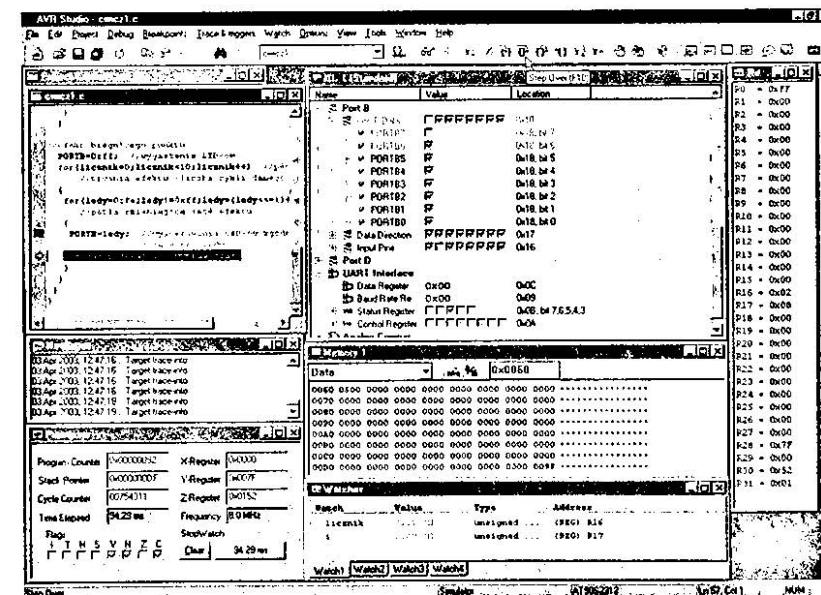
Niestety w przypadku programów pisanych w C nie działa kliknięcie na linię z komunikatem błędu powodujące przeniesienie kursora w odpowiednie miejsce tekstu źródłowego, tak jak to było w przypadku programów asemblerowych. Tym razem trzeba podejrzana o błąd linię odszukać ręcznie, korzystając z informacji wyświetlanych w prawej części paska stanu (u dołu okna głównego programu).

Po pierwszej bezbłędnej kompilacji zostanie wyświetlone okno, w którym konieczne będzie podanie pewnych informacji dotyczących parametrów docelowego systemu (rysunek 13.8). Są to: typ mikrokontrolera (w naszym przykładzie jest to AT90S2313), wielkość pamięci programu i danych (pozwalamy wartości domyślne) oraz częstotliwość rezonatora kwarcowego (ustawiamy taką wartość, jaka jest zastosowana w uruchamianym systemie, aby prawidłowo analizować zależności czasowe – w omawianym przykładzie jest to 8 MHz). Znajdują się tu jeszcze inne parametry. Są one aktywne tylko dla niektórych typów mikrokontrolerów. Po zaakceptowaniu ustawień przyciskiem **OK** można przygotować sobie okno robocze. Wybieramy te informacje, które będą potrzebne do symulacji. Na małych monitorach nie będzie to łatwe.

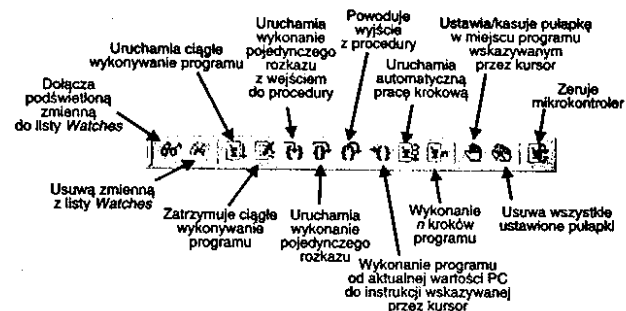


Rys. 13.9. Ikony paska narzędziowego służące do konfiguracji głównego okna roboczego programu AVR Studio

Naciśnięcie pierwszej lewej ikonki paska przedstawionego na rysunku 13.5 otwiera okno podglądu zmiennych programu (*Watches*), druga ikona powoduje podgląd rejestrów mikrokontrolera (*Registers*), trzecia ikona powoduje wyświetlenie zawartości pamięci. W zależności od wybranej pozycji z listy rozwijanej może to być obszar pamięci danych SRAM, obszar we/wy, wewnętrzny EEPROM lub pamięć programu. Możliwa jest jednoczesna obserwacja kilku obszarów. Naciśnięcie czwartej ikonki powoduje wyświetlenie informacji związanych ze stanem mikrokontrolera, a więc stanu licznika programu, wskaźnika stosu, stanu rejestrów indeksowych oraz wskaźników zawartych w rejestrze SREG. Bardzo przydatny do obliczeń czasowych będzie z pewnością licznik wykonanych cykli uzupełniony o pole, w którym wartość ta jest przeliczana na



Rys. 13.10. Okno główne programu AVR Studio



Rys. 13.11. Ikony paska narzędziowego AVR Studio służące do prowadzenia symulacji programu

jednostki czasu. Dodatkową korzyścią jest możliwość zerowania wskazań tego licznika, co umożliwia realizację precyzyjnego pomiaru czasu wykonywania poszczególnych fragmentów programu. Piąta ikona służy do otwarcia okna komunikatów symulatora (*Messages*), a ostatnia umożliwia wyświetlenie zasobów mikrokontrolera (stan CPU, ustawienia przerwań, rejestry timerów/liczników, watchdoga, EEPROM-u, UART-u, komparatora analogowego, a także stany rejestrów związanych z portami we/wy). Sposób prezentowania danych zapewnia wygodną interpretację poszczególnych bitów każdego rejestru (rysunek 13.10). Są one wyświetlane na czerwono w przypadku, gdy w wyniku wykonania pojedynczego rozkazu (praca krokowa) ulegną modyfikacji. Wszystkie zasoby mikrokontrolera mogą być ręcznie modyfikowane przez użytkownika.

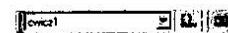
Na rysunku 13.11 pokazano ikony związane bezpośrednio z przebiegiem symulacji. Pierwsza od lewej powoduje dołączenie podświetlonej w oknie programu symulacji zmiennej do listy *Watches*, dzięki czemu stan tej zmiennej jest śledzony w trakcie wykonywania się programu. Druga ikona służy do kasowania zmiennej z listy podglądu. Musi być ona wcześniej podświetlona w oknie *Watches*. Następne ikonki uruchamiają symulację w różnych trybach. Pierwsza z nich powoduje ciągłe wykonywanie programu, w związku z czym najczęściej będzie współpracowała z wcześniej ustawionymi pułapkami. Stosuje się ją, gdy nie chcemy śledzić wykonywania programu instrukcja po instrukcji, gdyż sam przebieg nas nie interesuje, natomiast może on mieć wpływ na resztę programu – np. pętle opóźniające. Naciśnięcie tej ikony powoduje dezaktywację pozostałych. Nie są też widoczne zmiany licznika programu. Dłuższa praca w tym trybie może nasuwać wątpliwości, czy program gdzieś się nie zawiesił.

Kolejna ikona (czwarta od lewej na rysunku 13.11) powoduje przerwanie wykonywania programu. Wyświetlony w tym momencie stan mikrokontrolera

ra odpowiada takiemu, jaki wystąpił po ostatnim rozkazie. Kolejne ikony służą do: wykonania pojedynczego rozkazu z wejściem do ciała procedury (jeśli jest to np. *RCALL*), wykonania rozkazu traktując napotkaną procedurę jako pojedynczy krok programu. Siódma od lewej strony ikona z rysunku 13.11 służy do opuszczenia procedury. Jej naciśnięcie powoduje ciągłe wykonywanie programu, aż do napotkania rozkazów *RET* lub *RETI*.

Kolejna ikona powoduje ciągłe wykonanie programu od adresu wynikającego z aktualnej wartości licznika programu *PC* do miejsca wskazywanego przez kursor w oknie programu. W niektórych sytuacjach bardzo przydatne mogą być kolejne dwie ikonki. Pierwsza z nich inicjuje automatyczną pracę krokową. Między poszczególnymi krokami programu wstawiana jest przez symulator przerwa, której długość w milisekundach określa się poleceniem menu *Debug>Options>Single step delay (ms)*. Kolejna ikona powoduje ciągłe wykonanie *n* kroków programu bez podglądania zasobów. Wartość *n* jest ustawiana w poleceniu *Debug>Options>Number of Single Steps*.

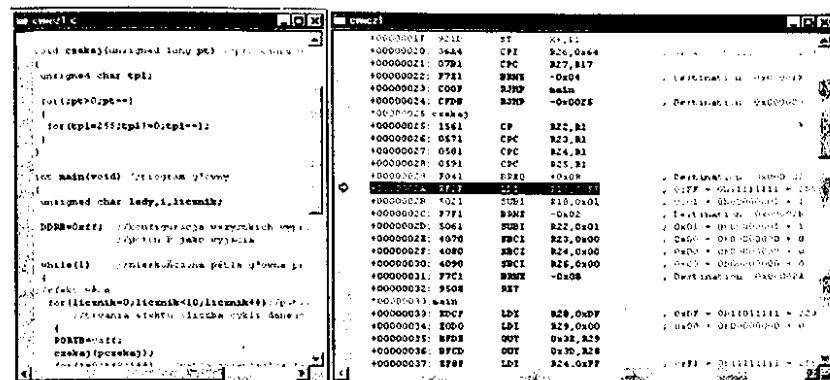
Do omówienia pozostały jeszcze trzy ostatnie ikony przedstawione na rysunku 13.11. Ikona z „łapką” ustawia/kasuje pułapkę programową w miejscu programu wskazywanym przez kursor w postaci żółtej strzałki w oknie programu. Jeśli pułapka jest ustawiona po lewej stronie instrukcji programu zostanie wyświetlony mały kwadracik. Ikona z dwiema „łapkami” usuwa wszystkie pułapki ustawione w programie. Pułapki mogą być tymczasowo blokowane. Odpowiednie do tego polecenia znajdują się w menu *Breakpoints>Show List>Enable* lub *Breakpoints>Show List>Disable*. Ostatnia ikona służy do wyzerowania symulowanego mikrokontrolera.



Rys. 13.12. Ikony paska narzędziowego informujące o aktywnym module i przełączające tryb pracy okna programu

Podczas symulacji programu wielomodulowego nazwa aktywnego modułu jest wyświetlana w oknie przedstawionym na rysunku 13.12. Krok wykonywania programu napisanego w C pokrywa się domyślnie z instrukcją tego języka. Czasami zachodzi jednak konieczność pracy na poziomie kodu maszynowego. Jest to możliwe po naciśnięciu ikonki z nawiasami klamrowymi przedstawionej na rysunku 13.12. Zostaje wówczas otwarte okno zawierające zdeasemblowaną postać programu. Od tego momentu praca krokowa odbywa się na poziomie rozkazów mikrokontrolera (rysunek 13.13).

Program AVR Studio stwarza możliwość uruchamiania programu bez konieczności wielokrotnego programowania pamięci mikrokontrolera. Wirtualna jednostka centralna nie zachowuje jednak pełnej kompatybilności z jej rzeczywistym odpowiednikiem. Przykładowo AVR Studio 3.56 nie potrafi symulować watchdoga, nie radzi sobie też dobrze z komparatorem analogowym. Nie



Rys. 13.13. Okno podglądu pracy mikrokontrolera na poziomie kodu mikrokontrolera

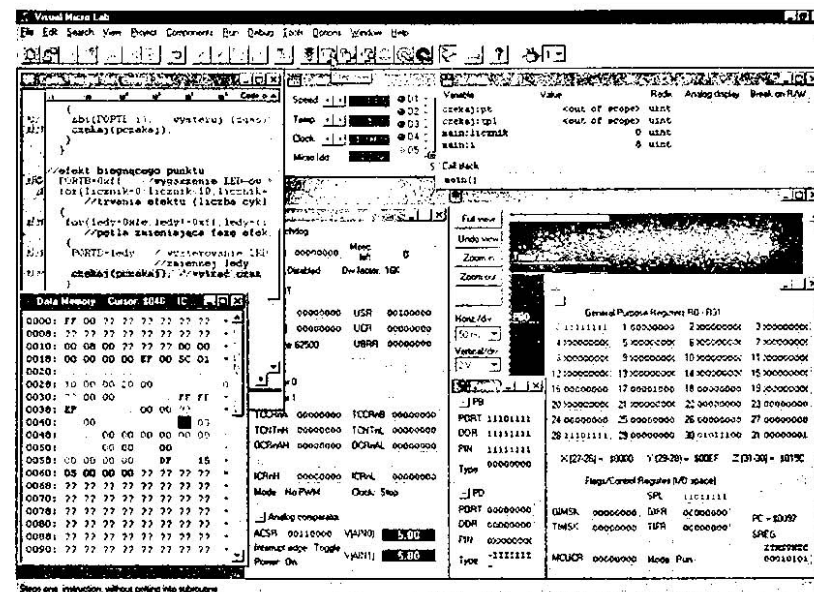
wszystkie więc tryby pracy układu rzeczywistego mogą być przetestowane w programie. Czasami trzeba się odwoływać do emulatorów sprzętowych.

Po etapie żmudnego wyszukiwania błędów logicznych przychodzi w końcu moment, w którym uznajemy, że program może być zapisany w pamięci mikrokontrolera. Teraz nastąpi weryfikacja działania symulatora w porównaniu z układem rzeczywistym. Opcje programu AVR Studio pozwalają na bezpośrednie zaprogramowanie mikrokontrolerów wykorzystując zestawy STK500/AVRISP/JTAG ICE oraz AVR Prog. W przypadku ich braku należy zastosować dowolny programator zewnętrzny wykorzystujący wygenerowany przez kompilator AVR-GCC plik *.hex.

13.4. Symulator Visual Micro Lab 3.56

Wszystkie opisywane wcześniej w książce programy występują jako tzw. *freeware*, co oznacza że można z nich korzystać zgodnie z warunkami licencji bez wnoszenia żadnych opłat. Jest jednak pewien program komercyjny, który ze względu na swoje zalety, także zasługuje na prezentację. Mowa o pakiecie Visual Micro Lab.

Symulator Visual Micro Lab znacznie lepiej oddaje rzeczywistość niż AVR Studio, ponadto umożliwia wierną symulację różnorodnych, zewnętrznych elementów i układów peryferyjnych dołączanych do mikrokontrolera takich jak: rezystor, kondensator, dioda LED, klawiatura 4x4, wzmacniacz operacyjny, komparator, przetworniki C/A i A/C, funktry logiczne, generatory różnych przebiegów, potencjometry, a także monitor I²C, alfanumeryczny



Rys. 13.14. Okno programu Visual Micro Lab

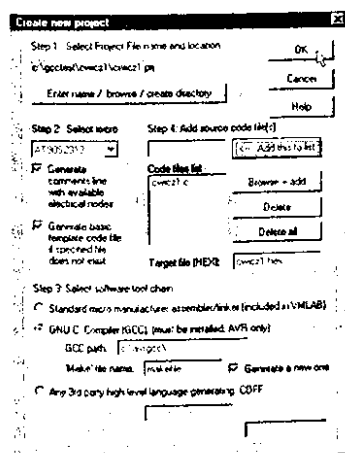
wyświetlacz LCD oraz terminal komunikacyjny TTY. VMLAB w wielu sytuacjach uwalnia użytkownika od konieczności budowania układu prototypowego i stosowania emulatorów sprzętowych. Oprócz podzespołów, z których można zbudować system, w programie VMLAB zaimplementowano również wielokanałowy oscyloskop wirtualny. Opis uruchamianego układu przypomina nieco standard, jakim jest Spice. Za pomocą Visual Micro Lab można uruchamiać urządzenia, w których wykorzystuje się zaimplementowany w mikrokontrolerze przetwornik analogowo-cyfrowy lub komparator, z czym jak pamiętamy miał problemy symulator AVR Studio.

Visual Micro Lab to program typu IDE. Integruje on assembler, zewnętrzny kompilator np. AVR-GCC, linker i debugger, przy czym integracja poszczególnych elementów jest znacznie lepsza niż w AVR Studio. Debugger wyświetla i wykonuje program zarówno na poziomie kodu maszynowego jak i języka wysokiego poziomu. Widać przy tym współzależność obu postaci programu. W VMLAB użytkownik również w każdej chwili może podejrzeć lub zmienić stan dowolnego komponentu mikrokontrolera lub zmiennej programu. Prezentacja wszystkich zasobów mikrokontrolera jest zrealizowana w sposób bardzo czytelny (rysunek 13.14), lecz rozmieszczenie wszystkich okien wymaga zastosowania dużego monitora, pracującego w wysokiej rozdzielczości.

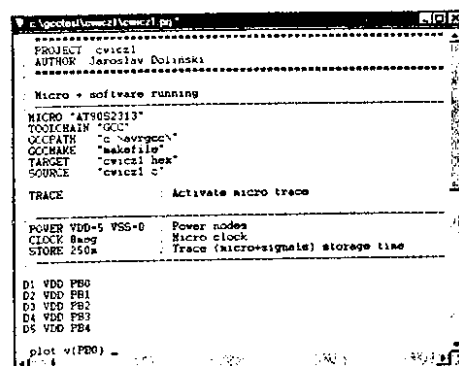
Instalacja programu jest bardzo prosta. Przed zakupem pełnej wersji można wypróbować działanie programu na 21-dniowej wersji ewaluacyjnej. Pliki można pobrać ze strony <http://www.amtools.net>. Zainstalowanie wersji pełnej wymaga wypełnienia odpowiedniego formularza i dokonania opłaty. Po wykonaniu tych czynności, drogą elektroniczną są dostarczane kody dla wersji pełnej.

Instalator zakłada na dysku domyślnym katalog `c:\VMLAB`, w którym umieszcza wszystkie niezbędne do pracy pliki. Kompilatory języka C muszą być zainstalowane na komputerze niezależnie od VMLAB-a. Nadają się jedynie te, które generują pliki `*.cof`. Symulacja uruchamianego urządzenia musi być poprzedzona zdefiniowaniem projektu. Z menu wybieramy polecenie *Project>New project*. Pojawia się okno kreowania projektu, w którym należy wypełnić odpowiednie pola. Trzymając się poprzedniego przykładu nadajemy nazwę np. `cwicz1.prj`, wybieramy jako docelowy mikrokontroler AT90S2313, decydujemy się na program w języku C. Przyjmujemy wersję GNU C i zgadzamy się na automatyczne wygenerowanie pliku `makefile`, co jest dużym udogodnieniem, gdyż zwalnia nas z konieczności kopiowania tego pliku z innych katalogów lub żmudnego, samodzielnego wpisywania. Automatyczne tworzenie pliku nie zamyka oczywiście możliwości późniejszej jego edycji. Przykładowe ustawienia projektu przedstawiono na rysunku 13.15.

Przed przystąpieniem do symulacji przydatne będzie określenie konfiguracji całego uruchamianego systemu, w tym ewentualne dołączenie wirtualnych peryferiów, zdefiniowanie parametrów śledzenia, określenie punktów syste-



Rys. 13.15. Okno parametrów projektu



Rys. 13.16. Konfiguracja uruchamianego systemu

mu, do których będzie dołączony wielokanałowy oscyloskop wirtualny. Ustawień tych dokonuje się w oknie wywołanym poleceniem *View>Project File* (rysunek 13.16). Szczegóły dotyczące użycia dostępnych elementów można znaleźć w dobrze opracowanej pomocy programu.

Zaletą VMLAB-a jest możliwość zdefiniowania pięciu różnych organizacji ekranu. Zapamiętywanie kilku widoków pozwoli szybko przechodzić pomiędzy ekranami zawierającymi istotne w danym momencie informacje. Znacznie zwiększa się tym samym czytelność, co ma znaczenie szczególnie wtedy, gdy monitor nie jest zbyt duży. Widoki zmienia się wybierając odpowiednią pozycję listy rozwijanej umieszczonej na końcu paska narzędziowego pokazanego na rysunku 13.14.

Pierwsza faza uruchamiania programu najczęściej będzie polegała na częstym sięganiu do źródła, kompilowaniu programu, przeglądaniu komunikatów kompilatora i ponownej edycji źródła. Warto więc w jednym z widoków podzielić ekran tylko na dwa okna: edycyjne i komunikatów. Zapewni to duży komfort podczas prac na tym etapie. Do przeprowadzenia kompilacji programu należy wybrać polecenie *Project>Build*. Można też użyć klawisza F9 lub ikony *Build* (środkowa ikona na pasku narzędziowym). W oknie *Messages* pojawiają się wszystkie komunikaty generowane przez kompilator. Jeśli wystąpi choćby jeden błąd, kursor edycyjny zostanie ustawiony w podejrzanym o to miejscu. Czasami jednak błąd może być ulokowany gdzie indziej. Na przykład, brak średnika kończącego linię programu C spowoduje wystąpienie błędu dopiero podczas kompilacji dalszej części programu. Wszystkie miejsca, na które trzeba zwrócić uwagę, są sygnalizowane wykrzyknikami umieszczonymi przed treścią komunikatu. Kolor żółty oznacza, że chodzi tylko o ostrzeżenie, program będzie kompilowany. Przykładowo VMLAB ostrzega w ten sposób przed zadeklarowaniem zmiennej, która nie jest używana dalej w żadnym miejscu programu. Gorzej, gdy wykrzyknik ma kolor czerwony. W takiej sytuacji kompilator nie tworzy pliku wykonywalnego, a więc uruchomienie programu nie będzie możliwe. Tego typu błędy trzeba bezwzględnie poprawić. Czasami zdarza się, że jedna nieprawidłowość w źródle powoduje wygenerowanie całej listy błędów. Żmudny etap eliminacji błędów formalnych kończy się wyświetleniem komunikatu: *Success! All ready to run*. Od tego momentu rozpoczyna się kolejna faza uruchamiania aplikacji – symulacja. Teraz wskazane będzie stworzenie sobie kolejnych ekranów roboczych wynikających ze specyfiki uruchamianego układu. Na pewno warto umieścić na nim choćby niewielkie okno źródła programu. Wykorzystując krokowy tryb pracy, będzie w nim zaznaczana aktualnie wykonywana instrukcja na poziomie odpowiadającym wersji źródłowej (język C lub asemb-

ler). Jeśli program jest napisany w C, a chcemy wykonywać go na poziomie rozkazów mikrokontrolera, konieczne będzie dołożenie okna *Program memory* i uczynienie go aktywnym. Po lewej stronie każdej instrukcji widnieją niewielkie przyciski ekranowe. Ich naciśnięcie powoduje wstawienie w danym miejscu programu pułapki (*breakpoint*). Kolor klawisza zmienia się wówczas na czerwony. Aktywnych pułapek może być wiele. Podczas prac uruchomieniowych często zachodzi konieczność pomiaru czasu wykonywania się np. określonej procedury. Tu niestety trzeba przyznać wyższość programu AVR Studio nad Visual Micro Lab. Atrakcyjna może być natomiast graficzna interpretacja czasu przebywania jednostki centralnej w określonym miejscu programu. Rolę wskaźników spełniają żółte paski nałożone na poszczególne instrukcje. Trzeba przy tym pamiętać, że skala czasu jest logarymiczna!

Poszczególne komponenty mikrokontrolera są zgrupowane w kilku niezależnych oknach, np.: rejestry, pamięć programu, pamięć danych SRAM, pamięć EEPROM, porty oraz UART, timery, watchdog. Zawartość każdego z nich może być wyświetlana w postaci binarnej, hexadecymalnej, całkowitoliczbowej, a nawet w kodach ASCII. Na wszystkie pola jest nałożony półprzezroczysty pasek graficzny, który w postaci analogowej pokazuje stan danego komponentu. Każdy z rejestrów może być w dowolnej chwili zmieniony. Niestety nie dotyczy to licznika programu, co oznacza, że program może być wykonany tylko w takiej kolejności, jak w układzie rzeczywistym. Użytkownik ma też możliwość podglądania zmiennych programu. Bywają z tym niestety problemy, zmienne te nie zawsze są dostępne. Nie można też modyfikować w prosty sposób ich wartości. W tym zakresie AVR Studio wykazuje przewagę nad Visual Micro Lab. Pewnym wyjściem z sytuacji jest podejrzenie miejsca przechowywania zmiennej (rejestry, pamięć RAM) i jej edycja na tym poziomie.

Specyficznym oknem VMLAB-a jest tzw. panel sterujący. Zawiera on zewnętrzne urządzenia peryferyjne wykorzystywane w aplikacji. Może to być klawiatura 4x4, alfanumeryczny wyświetlacz LCD, wirtualny terminal TTY, diody świecące, potencjometry. Ich parametry konfiguruje się w oknie projektu.

Uruchomienie wirtualnego systemu następuje po naciśnięciu klawisza F5 lub ikonki z semaforem. Nasz wirtualny system zaczyna w tym momencie działać. Na oscyloskopie (jeśli jest włączony) pojawiają się przebiegi generowane przez mikrokontroler, zmianie ulegają wszystkie wykorzystywane komponenty. Symulację można zatrzymać naciskając ikonę ze znakiem „STOP”. Oprócz pracy ciągłej mamy do dyspozycji – podobnie jak w AVR Studio – różne tryby pracy krokowej.

Przebieg na oscyloskopie może być dość swobodnie skalowany, umożliwiając wyświetlanie charakterystyk z dużą dokładnością (100 ns/działkę) lub wyświetlanie przebiegów zebranych w dłuższym czasie, a także przyjrzenie się detalom z rozdzielczością 100 ns/działkę. Ustawiając na oscylogramie dwa kursory można łatwo dokonać pomiaru czasu.

Program Visual Micro Lab jest bardzo wygodny w użyciu, ma wiele cech sprawiających, że chętnie się po niego sięga. W sytuacjach, w których wady stają się męczące zawsze można sięgnąć po AVR Studio.

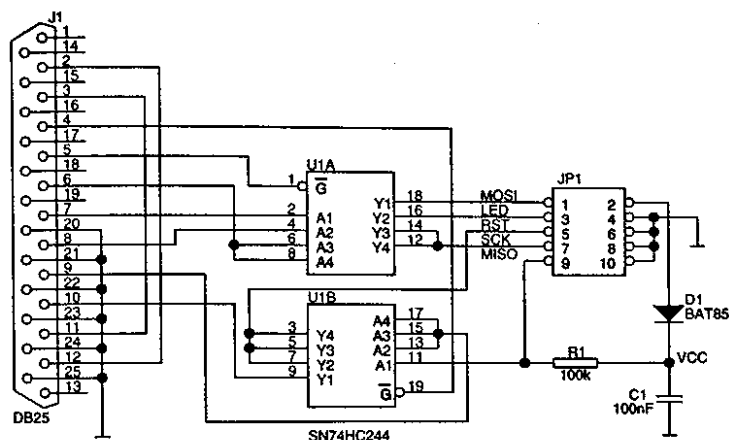
13.5. Programowanie pamięci programu w systemie (ISP)

Mikrokontrolery AVR firmy Atmel są wyposażone w interfejs umożliwiający programowanie ich pamięci już po zamontowaniu w systemie (ISP). Wykorzystanie takiego sposobu programowania powoduje, że można zrezygnować z zakupu programatora stacjonarnego, wystarczy jedynie prosty interfejs między uruchamianym systemem a komputerem PC (jak np. ZL2PRG opisany w dalszej części rozdziału). Liczne przykłady takich konstrukcji można znaleźć w Internecie. Odnośniki są zamieszczone w dodatku H.

13.5.1. Programator ZL2PRG

Schemat elektryczny programatora ZL2PRG pokazano na **rysunku 13.17**. Układ U1 pełni rolę separatora linii we/wy interfejsu drukarkowego Centronics od systemu, w którym znajduje się programowany mikrokontroler. Interfejs jest zasilany napięciem pobieranym z systemu, w związku z czym podczas korzystania z niego nie trzeba stosować dodatkowego zasilacza. Dla programatora ZL2PRG zaprojektowano dwustronną płytkę drukowaną, której mozaiki i schemat montażowy pokazano w dodatku G.

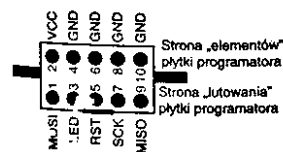
Układ U1 ma obudowę przystosowaną do montażu SMD (SO20), pozostałe elementy są montowane klasycznie. Złącza J1 i JP1 są montowane na krawędzi płytki w taki sposób, że przed ich przylutowaniem laminat jest wsuwany pomiędzy rzędy wyprowadzeń. Na **rysunku 13.18** pokazano przypisanie sygnałów do styków gniazda JP1. Sygnał LED można wykorzystać do sterowania diodą świecącą, sygnalizującą programowanie układu. Na płycie ZL1AVR dioda taka jest uwzględniona (D9 dołączona do styku 3. gniazda ISP2).



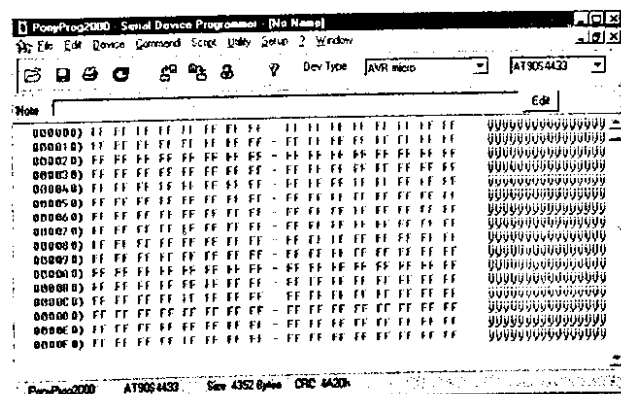
Rys. 13.17. Schemat elektryczny programatora ZL2PRG

Atutem programatora ZL2PRG jest możliwość współpracy z wieloma bezpłatnymi programami sterującymi jego pracą. Jednym z lepszych jest program PonyProg 2000 (rysunek 13.19), dostępny w Internecie pod adresem: <http://www.lancos.com/ppwin95.html>. PonyProg występuje w wersjach dla Windows (łącznie z NT/2K/XP) oraz Linuksa. Obsługuje mikrokontrolery: AT90S1200, AT90S2313, AT90S2323, AT90S2333, AT90S2343, AT90S4414, AT90S4434, AT90S8515, AT90S8534, AT90S8535, ATmega8, ATmega16, ATmega64, ATmega103, ATmega128, ATmega161, ATmega163, ATmega323, ATtiny12 i ATtiny15.

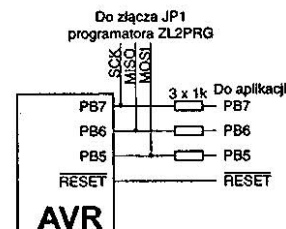
Na rysunku 13.20 pokazano zalecany sposób dołączenia programatora ZL2PRG do mikrokontrolera zainstalowanego w systemie. Taki sposób dołą-



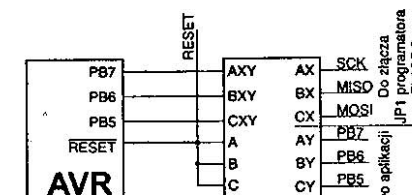
Rys. 13.18. Wyrowadzenia sygnałów na styki gniazda JP1



Rys. 13.19. Wygląd okna programu PonyProg 2000



Rys. 13.20. Zalecany sposób dołączenia programatora ZL2PRG do mikrokontrolera zainstalowanego w systemie, gdy PB5, PB6 i PB7 pracują jako wejścia



Rys. 13.21. Zalecany sposób dołączenia programatora ZL2PRG do mikrokontrolera zainstalowanego w systemie, gdy co najmniej jedno z wyprowadzeń PB5, PB6 lub PB7 pracuje jako wyjście

czenia interfejsu ISP do mikrokontrolera jest możliwy tylko w przypadku, gdy porty PB5, PB6 i PB7 są skonfigurowane jako wejścia. W przypadku, gdy któreś z tych wyprowadzeń musi pracować jako wyjście, firma Atmel zaleca inny sposób dołączenia programatora – pokazano go na rysunku 13.21.

14. Przykładowe aplikacje

Wcześniejsze rozdziały książki zawierały wiele informacji technicznych, które można znaleźć w notach katalogowych mikrokontrolerów AVR, w szczególności AT90S2313. Zapoznanie się z nimi jest nieodzowne do świadomego i efektywnego wykorzystywania możliwości mikrokontrolerów we własnych projektach, pomaga także zrozumieć zasadę działania rozwiązań spotykanych w książkach, czasopiśmie i Internecie. Sucha wiedza katalogowa z pewnością nie wystarczy do pełnego zaznajomienia się z mikrokontrolerami AVR. Najczęściej dopiero samodzielnie przeprowadzone eksperymenty praktyczne wyjaśniają wątpliwości, jakie mogły nasunąć się podczas lektury opisu technicznego.

Rozdział 14. ułożono w formie zestawu ćwiczeń praktycznych, sukcesywnie odsłaniających tajemnice mikrokontrolera, objaśniających zasady wykorzystywania jego bloków funkcjonalnych i rozwiązujących wiele problemów, na jakie natykają początkujący elektronicy. Powtarzające się bez końca pytania, z którymi można spotkać się na internetowych grupach dyskusyjnych typu: „Jak obsłużyć wyświetlacz alfanumeryczny?”, „Jak zrealizować transmisję, wykorzystując interfejs RS232?”, czy nawet „Jak sterować LED-em dołączonym do mikrokontrolera?”, skłaniają do zajęcia się tymi zagadnieniami. Mam nadzieję, że opisane eksperymenty rozwieją przynajmniej część tego typu wątpliwości Czytelników.

Przedstawiony minikurs opiera się na płytce uruchomieniowej ZL1AVR z mikrokontrolerem AVR AT90S2313. Zestaw zaprojektowano specjalnie na potrzeby książki. Wybór mikrokontrolera został podyktowany dużą popularnością tego układu, jego niską ceną i faktem, że świetnie się nadaje do wielu praktycznych aplikacji.



Płytki drukowane zestawu ZL1AVR oraz programatora ZL2PRG są dostępne w internetowym sklepie Wydawnictwa BTC (<http://www.btc.pl/index.php?id=15>).

Podstawowym dylematem rozstrzyganym podczas przygotowywania ćwiczeń był wybór języka programowania. W poprzednich rozdziałach książki większość przykładów napisano w assemblerze. Trudno było uczynić inaczej podczas omawiania listy rozkazów mikrokontrolera. Język ten nie jest jednak obecnie zbyt często stosowany w praktyce. Pisanie programów w assemblerze zajmuje dużo czasu, a ich czytelność, szczególnie po pewnym czasie staje się

niewielka. Alternatywnymi możliwościami był Bascom lub język C – mój wybór padł na język C. Argumentem przemawiającym za tym rozwiązaniem był fakt, że jest dostępny kompilator tego języka w wersji darmowej (AVR-GCC), bez żadnych ograniczeń (licencja GNU). Spotykane w Internecie opisy zagadnień technicznych również na ogół występują w postaci procedur lub całych programów pisanych w C. Nie da się ukryć, że język ten stanowi obecnie standard w zastosowaniach profesjonalnych, warto więc nauczyć się go jak najwcześniej.



Przykładowe programy przedstawione w dalszej części książki napisano w języku C i kompilowano za pomocą bezpłatnego kompilatora AVR-GCC w wersji 3.2 datowanej na 2002.06.25. Jest ona dostępna na stronie internetowej Wydawnictwa BTC (<http://www.btc.pl?id=plicki>).

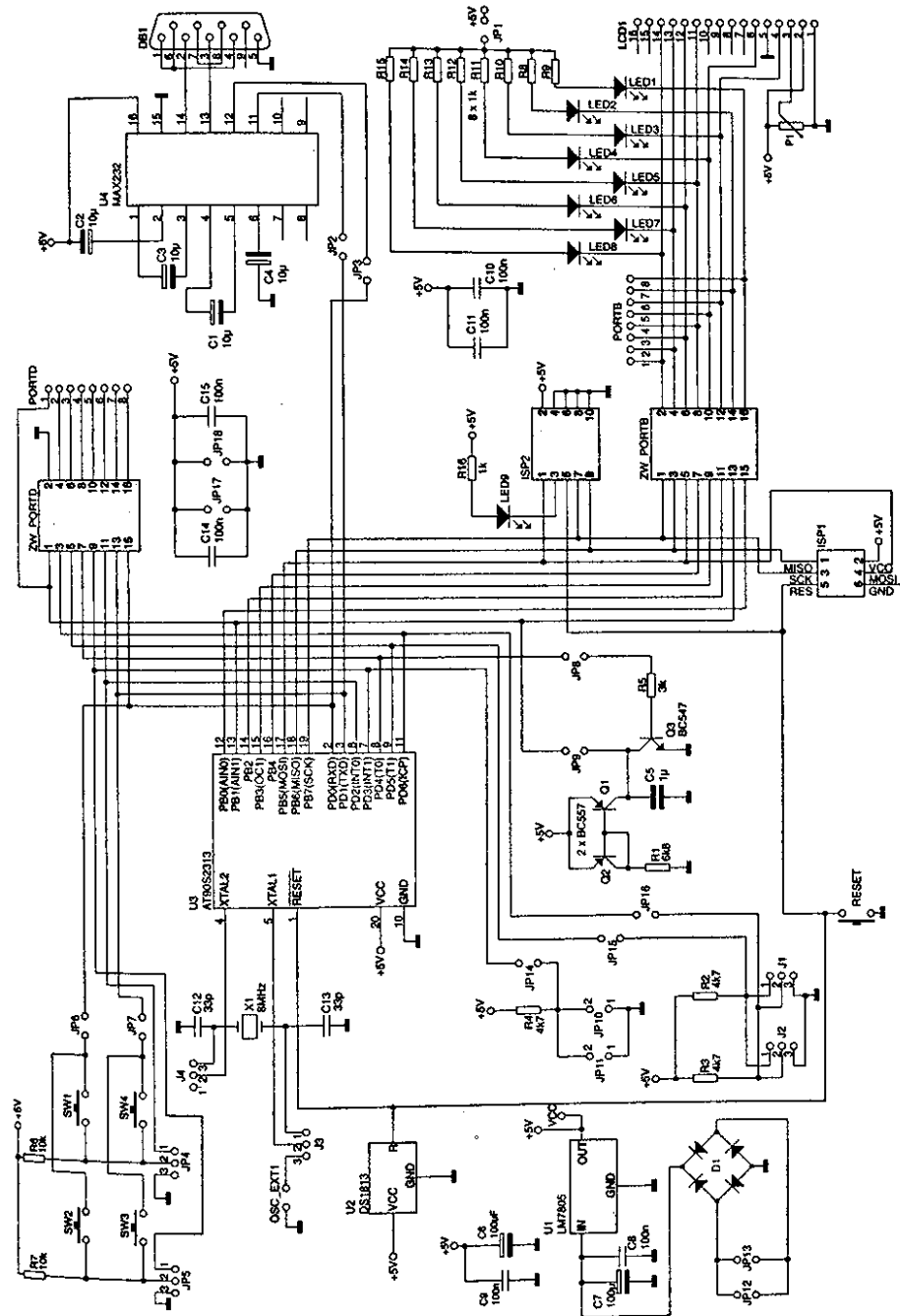
Wybranie języka wysokiego poziomu ma również taką zaletę, że ucząc się programowania w języku wysokiego poziomu, w każdej chwili można podejrzeć kod wynikowy i w pewnym sensie również w ten sposób uczyć się assemblera.

Przykłady przedstawione w książce kompilowane są kompilatorem AVR-GCC. Sposób skonfigurowania środowiska do pracy z tym kompilatorem opisano w rozdziale 13. Należy zaznaczyć, że nie było zamiarem autora pisanie podręcznika do nauki języka C. Trzeba również dodać, że zamieszczone programy reprezentują tylko jedną z wielu możliwych metod rozwiązywania poszczególnych problemów.

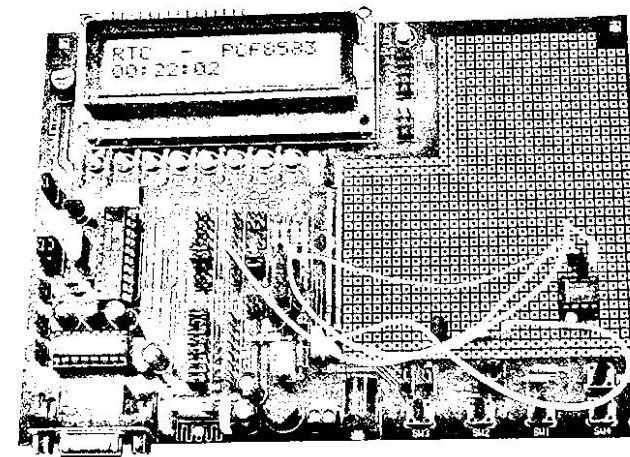
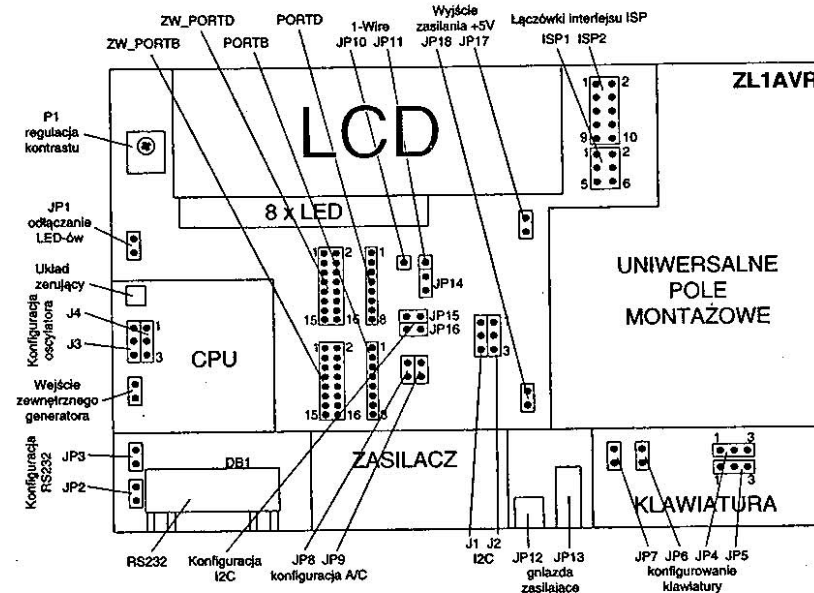
14.1. Zestaw uruchomieniowy ZL1AVR

Jak już wspomniano, specjalnie na potrzeby niniejszej książki zaprojektowano zestaw uruchomieniowy, na którym były testowane wszystkie projekty przedstawione w tym rozdziale. Jest on stosunkowo łatwy w wykonaniu, bardzo uniwersalny, a jego koszt wykonania relatywnie niski.

Na **rysunku 14.1** przedstawiono schemat elektryczny zestawu uruchomieniowego ZL1AVR, a na **rysunku 14.2** widok zmontowanej płytki i rozmieszczenie na niej najważniejszych elementów. Dla jak najpełniejszego zilustrowania możliwości mikrokontrolera przewidziano kilka bloków funkcjonalnych, skonfigurowanych za pomocą licznych zworek. Odpowiednie ich ustawienie pozwoli na przeprowadzenie różnorodnych eksperymentów praktycznych.



Rys. 14.1. Schemat elektryczny zestawu ZL1AVR



Rys. 14.2. Rozmieszczenie najważniejszych elementów na płycie zestawu ZL1AVR

Wykorzystując zestaw będzie można również uruchamiać fragmenty własnych aplikacji, a to dzięki uniwersalnemu polu montażowemu umieszczone-
mu w prawej części płytki. Rozwiązanie takie podnosi walory funkcjonalne,
a przede wszystkim edukacyjne zestawu.

14.1.1. Zasilanie zestawu

Zasilanie zestawu ZL1AVR może być zrealizowane na kilka sposobów. Pierwszy z nich, to wykorzystanie typowego zasilacza wtyczkowego dostarczającego napięcie o wartości od 8 do 12 V. Umieszczenie na płytce mostka prostowniczego D1 i stabilizatora U1 nie narzuca ostrych wymagań co do napięcia wejściowego. Nie musi być ono stabilizowane, nie jest nawet konieczne napięcie stałe. Jak więc widać, drugim sposobem zasilania będzie wykorzystanie zwykłego transformatora. W tym przypadku wartość skuteczna napięcia powinna zawierać się między 5,6 V, a 8,5 V.



Mozaiki płytki drukowanej oraz schemat montażowy płytki zestawu ZL1AVR znajdują się w dodatku G.

Napięcie zasilające doprowadza się do gniazda JP12 lub JP13. Złącze JP12 to śrubowy zacisk ARK, przydatny do podłączania przewodów. Ze względu na obecność mostka D1 nie jest istotna biegunowość dołączanego napięcia zasilającego. Złącze JP13 to typowe gniazdo zasilające, które można wykorzystywać alternatywnie z JP12.

Pobór prądu przez system nie jest duży (ok. 45 mA z włączonymi diodami LED i wyświetlaczem LCD), dzięki czemu stabilizator U1 – w większości przypadków – będzie mógł pracować bez radiatora. Dołączenie własnych układów może spowodować znaczny wzrost prądu zasilającego, a co za tym idzie wzrost temperatury stabilizatora. W takim przypadku konieczne może się okazać założenie radiatora na układ U1 lub zmniejszenie do minimum wartości napięcia zasilającego. Różnica między napięciem wejściowym, a wyjściowym nie powinna być mniejsza niż 2 V. Kondensator C7 filtruje tętnienia napięcia wyprostowanego, a C8 i C9 odsprężają układ U1 dla wielkich częstotliwości, zabezpieczając przed wzbudzeniem się. W dolnej oraz górnej części uniwersalnego pola montażowego przebiegają dwie ścieżki zasilające tak, by można było w jak najwygodniejszy sposób doprowadzić napięcie zasilające do własnej części badanego systemu. Oprócz tego napięcie to jest dostępne na łączówkach JP17 i JP18. Podczas korzystania z nich należy uważać, by nie spowodować zwarcia.

14.1.2. Taktowanie i zerowanie mikrokontrolera

Najważniejszym elementem na płytce jest oczywiście mikrokontroler U3. Zalecany jest układ AT90S2313-10, współpracujący z rezonatorem kwarcowym

wym X1 o częstotliwości 8 MHz. Dla takiej konfiguracji są przygotowane przykładowe programy. W szczególności chodzi o procedury odmierzania czasu. W przypadku korzystania z rezonatora na płytce, zworki J3 i J4 powinny być ustawione w górnym położeniu. Gdyby z jakichś powodów zaistniała konieczność dołączenia zewnętrznego sygnału zegarowego, to jest to możliwe poprzez łączówkę OSC_EXT. Zworki J3 i J4 powinny być wówczas ustawione w dolnym położeniu. Sygnał z łączówki wejściowej OSC_EXT trafia poprzez zworkę J3 na wejście XTAL1 mikrokontrolera. Możliwość użycia zewnętrznego sygnału zegarowego może być wykorzystana np. do sprawdzenia zależności prądu zasilającego w funkcji częstotliwości taktowania układu.

Z generatorem jest związany układ zerujący. Mikrokontroler jest jak wiadomo układem synchronicznym. Do jego pracy niezbędny jest odpowiedni przebieg zegarowy. W momencie włączenia zasilania generator nie wzbudza się natychmiast. Start jednostki centralnej musi więc być opóźniony o czas gwarantujący jego stabilną pracę. Do wygenerowania odpowiedniego sygnału RESET najlepiej nadają się specjalizowane układy, takie jak np. DS1813, który zastosowano w zestawie uruchomieniowym (układ U2). Zaletą takiego rozwiązania jest wygenerowanie pewnego w działaniu sygnału zerującego, bez względu na szybkość narastania napięcia zasilającego. Odpowiednia jakość zerowania bywa ignorowana przez konstruktorów, tymczasem okazuje się, że w wielu przypadkach nieprawidłowo rozwiązane zerowanie mikrokontrolera uniemożliwia jego prawidłowe działanie.

Zerowanie mikrokontrolera tylko i wyłącznie poprzez wyłączanie i włączanie zasilania byłoby na dłuższą metę uciążliwe, szczególnie w przypadku prowadzenia różnorodnych testów. Dlatego też na płytce znajduje się przycisk RESET, za pomocą którego można ręcznie zerować system.

14.1.3. Wykorzystywanie portów mikrokontrolera

Wszystkie porty mikrokontrolera są wyprowadzone na gniazda o oznaczeniach PORTB i PORTD. Zanim jednak sygnały z odpowiednich portów mikrokontrolera do nich dotrą, muszą przejść przez zworki zakładane na specjalnie do tego celu umieszczone podwójne rzędy szpilek ZW_PORTB i ZW_PORTD. Umieszczenie zworki np. w położeniu 1-2 gniazda ZW_PORTB powoduje połączenie wyprowadzenia 1 gniazda PORTB z wyprowadzeniem 19 mikrokontrolera (PB7). Analogicznie jest dla pozostałych wyprowadzeń. W ten sposób wyprowadzenia poszczególnych portów mikrokontrolera (gniazda PORTB i PORTD) mogą być na stałe dołączone, np. do własnego układu umieszczone-

go na uniwersalnym polu montażowym i w razie potrzeby odłączane od niego za pomocą zworek zakładanych na ZW_PORTB i ZW_PORTD. Na uwagę zasługują końcówki 1 i 2 gniazda ZW_PORTD oraz 1 PORTD. Mikrokontroler AT90S2313 ma jedynie 7 wyprowadzeń portu D i w związku z tym wymienione wyżej końcówki wykorzystano do dołączenia sygnału do wewnętrznego komparatora analogowego mikrokontrolera. Dla wygody wyprowadzenie 2 ZW_PORTD dołączono do masy.

UWAGA

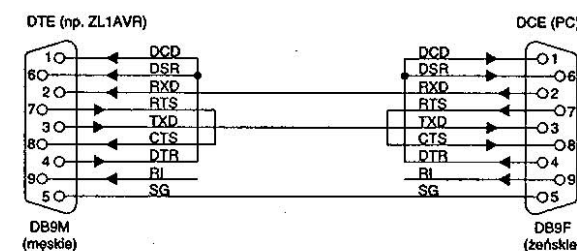
Dla wygody pracy warto przygotować sobie specjalne kabelki, które będą wykorzystywane do połączeń systemu z własnym układem umieszczonym na uniwersalnym polu montażowym. Kabelki takie mogą być wykonane np. przy użyciu pojedynczych styków tulipanowych wymontowanych ze standardowej łączówki wielostykowej. Do każdej końcówki należy przylutować kawałek drutu (linki miedzianej) i zaizolować rurką termokurczliwą, wzmacniając tym samym połączenie. Drugi koniec kabelka może być jedynie pozbawiony izolacji i pocynowany. Będzie on później dolutowywany do odpowiedniego punktu pola montażowego. Do eksperymentów przyda się też kilka kabelków zakończonych obustronnie łączówką.

14.1.4. Klawiatura

W jaki sposób człowiek może komunikować się z systemem mikroprocesorowym? Na przykład za pomocą klawiatury. Jako, że jej obsługa stanowi odwieczny problem początkujących elektroników, klawiatury nie mogło zabraknąć na płytce. Tworzą ją przyciski SW1, SW2, SW3, SW4. Zworki JP4...JP7 służą do dołączenia klawiatury do mikrokontrolera i odpowiedniego jej skonfigurowania. Przyciski SW1...SW4 mogą pracować jako dwa niezależne przyciski dołączone bezpośrednio do mikrokontrolera, przy czym muszą być obsługiwane przez dwa różne porty (można więc wykorzystywać parę SW1, SW4 lub SW2, SW3). Mogą też być skonfigurowane jako klawiatura matrycowa 2x2. W pierwszym przypadku zworki JP6 i JP7 muszą być zwarte, a JP4 i JP5 założone w pozycji 2-3 (połączenie z masą). W układzie klawiatury matrycowej zworki JP4 i JP5 powinny być założone w pozycji 1-2 (lewe położenie). Zworki JP6 i JP7 służą do całkowitego odłączenia przycisków od mikrokontrolera. Może być to konieczne np. w przypadku używania UART-u lub portów PD1 i/lub PD0. Naciśnięcie któregoś z tych przycisków mogłoby zakłócić pracę systemu.

14.1.5. Interfejs RS232

Wejście i wyjście UART-u mikrokontrolera wyprowadzono poprzez zworki JP2 i JP3 oraz układ MAX232 (U4) na gniazdo DB1. Układ U4 służy do konwersji poziomów logicznych mikrokontrolera na poziomy napięcie zgodne ze standardem RS232. Płytke uruchomieniową można łączyć poprzez gniazdo DB1 (typu DSUB-9) z dowolnym innym urządzeniem wyposażonym w interfejs RS232. Do połączenia wystarczy kabel typu *null-modem*, w którym wykorzystuje się jedynie 3 przewody (dwie linie transmisyjne i masę sygnałową). Niezbędne w takim przypadku zapętlenia sygnałów RTS i CTS oraz DSR, DTR i DCD są wykonane na płytce (rysunek 14.3).



Rys. 14.3. Połączenie typu *null-modem*. Zapętlenia sygnałów RTS i CTS oraz DSR, DTR i DCD mogą być wykonane na płytce drukowanej lub przy stykach złącz DSUB-9

14.1.6. Diody LED

Kolejnym środkiem komunikacji pomiędzy systemem a użytkownikiem jest zespół diod świecących LED1...LED8. Są one dołączane indywidualnie do portu PB mikrokontrolera poprzez zworki ZW_PORTB. Można wykorzystywać tylko wybrane, niekoniecznie wszystkie na raz. Zworka JP1 służy natomiast do jednoczesnego odłączenia wszystkich diod LED od mikrokontrolera. Sterowanie diodami odbywa się bezpośrednio z wyprowadzeń mikrokontrolera bez dodatkowych wzmacniaczy tranzystorowych. Jak pamiętamy z poprzednich rozdziałów, wydajność prądowa portów układu AT90S2313 jest wystarczająca do tego celu. Jedyne na co trzeba zwrócić uwagę, to takie dobranie rezystorów R9 do R15, aby nie przekroczyć całkowitej dopuszczalnej mocy strat układu U3. W przypadku zestawu warunek ten jest spełniony, gdyż rezystory dołączone szeregowo ograniczają prąd diod LED do ok. 4 mA/szt., co przy zapaleniu wszystkich diod daje pobór prądu o wartości ok. 32 mA. Założona wartość prądu w zupełności powinna wystarczyć do wyraźnego świecenia się diod LED, nawet gdy nie będą to diody wysokiej jasności. Dio-

dy LED są dołączone do tego samego portu co wyświetlacz alfanumeryczny LCD. Jeśli przewidywane jest jego stosowanie, to niestety należy zrezygnować z diod LED lub zgodzić się, że będą mrugały podczas pracy w czasie wymiany danych między mikrokontrolerem a wyświetlaczem. Jedynie LED1 i LED2 pozostają do ewentualnego normalnego wykorzystania, ale tym razem pod warunkiem, że nie będzie używany przetwornik analogowo-cyfrowy lub odpowiednio porty PB0 i PB1.

14.1.7. Wyświetlacz alfanumeryczny LCD

Na płycie uruchomieniowej umieszczono gniazdo szpilkowe LCD1 przeznaczone do dołączenia typowego wyświetlacza alfanumerycznego LCD. W zestawie przewidziano wyświetlacz 2×16. Obsługa jest realizowana za pomocą 4-bitowej szyny danych i dwóch sygnałów sterujących: E i RS. Typowy interfejs wyświetlacza alfanumerycznego zawiera ponadto sygnał R/ \overline{W} , za pomocą którego określa się kierunek transmisji danych. Jest to potrzebne prawie wyłącznie do sprawdzania gotowości wewnętrznego sterownika wyświetlacza. Nadesłane polecenie do wykonania przez mikrokontroler w chwili, gdy sterownik nie jest gotowy kończy się jego zignorowaniem, co najczęściej prowadzi do błędnego działania układu. Gotowość tę można jednak przewidzieć, wprowadzając do obsługi odpowiednie opóźnienia programowe (zakładając, że po tym czasie sterownik upora się z wcześniejszymi zadaniami). Obsługa nie jest wtedy tak wydajna, jak przy sprawdzaniu gotowości wyświetlacza, lecz w zastosowaniach praktycznych na ogół w niczym to nie przeszkadza. Potencjometr P1 usytuowany blisko wyświetlacza służy do regulacji kontrastu i często bywa powodem frustracji niedoświadczonych elektroników. Jeśli bowiem jego suwak będzie ustawiony w złym położeniu, to kontrast wyświetlacza może być na tyle mały, że żadne znaki nie będą widoczne na wyświetlaczu. Cała wina w takich sytuacjach jest często „rzucana” na błędy programu lub pomyłki w połączeniach. Dlatego w czasie pierwszego włączenia wyświetlacza, gdy nic na nim nie widać, warto pokręcić suwak od jednego skrajnego położenia do drugiego. Dopiero, gdy po tym sprawdzeniu wyświetlacz nadal będzie „milczał”, należy szukać innych błędów.

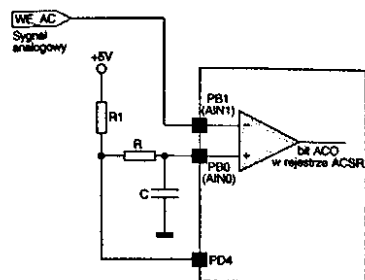
14.1.8. Interfejsy I²C i 1-Wire

Interfejsy I²C i 1-Wire są powszechnie stosowane do komunikacji różnorodnych urządzeń peryferyjnych z systemem mikroprocesorowym. Czasami można spotkać nieco zmienione ich nazwy, co bywa związane z trudnościami

w uzyskiwaniu licencji na stosowanie interfejsu przez producentów poszczególnych modułów peryferyjnych. Okazuje się, że właściciele praw autorskich bardzo surowo je egzekwują, a liczni producenci nie zawsze mogą sobie pozwolić lub po prostu nie mają zamiaru za nie płacić. Problemów takich nie uniknął nawet taki potentat, jak Analog Devices. Interfejs I²C został opracowany przez firmę Philips, natomiast 1-Wire w dawnej Dallas Semiconductor, obecnie Maxim-Dallas. Obydwa interfejsy nie wymagają stosowania żadnych dodatkowych układów oprócz rezystorów podciągających na liniach transmisyjnych. Urządzenia peryferyjne komunikujące się poprzez interfejs I²C dołącza się do systemu poprzez gniazda J1 lub J2 (zdublowano je dla wygody). Linię danych SDA dołącza się do mikrokontrolera za pomocą zworki JP15, a linię zegarową SCL za pomocą JP16. Rezystory podciągające R2 i R3 gwarantują odpowiednią jakość podciągnięcia. Analogicznie jest z interfejsem 1-Wire. Tym razem, jak sama nazwa wskazuje, jest to połączenie jedнопrzewodowe. Sygnał doprowadza się do gniazda JP10 lub JP11, a dołączenie do systemu zapewnia zworka JP14. Interfejsy I²C i 1-Wire zajmują porty odpowiednio PD5, PD6 i PD3. Należy to uwzględnić w planowaniu eksperymentów.

14.1.9. Przetwornik analogowo-cyfrowy

Z zasady działania układów cyfrowych, do jakich należą mikrokontrolery, wynika, że operują one na dyskretnym i skończonym zbiorze stanów wejściowych i wyjściowych. Tymczasem świat wokół nas jest analogowy, czyli ciągle i nieskończony (przynajmniej z matematyczno-fizycznego punktu widzenia). Można więc wysnuć wniosek, że przydatność mikrokontrolerów w dziedzinie analizy typowych, otaczających nas zjawisk będzie raczej niewielka. Z drugiej zaś strony jeśli przyjrzeć się naszym zmysłom, to okaże się, że są bardzo ułomne i stosunkowo prosto można je oszukać... oczywiście metodami cyfrowymi. Przekonujemy się o tym na co dzień słuchając płyt kompaktowych, czy nawet oglądając zdjęcia z wakacji. Jak zatem wprowadzić analogową informację do cyfrowego mikrokontrolera? W ogólnym przypadku odpowiedź brzmiałaby: stosując przetwornik analogowo-cyfrowy. Większość producentów mikrokontrolerów ma w swojej ofercie handlowej modele układów wyposażonych w wewnętrzne przetworniki ADC (*Analog-Digital Converter*). Znajdziemy je również wśród produktów Atmela, np.: ATtiny15L, ATtiny26L, AT90S4433, AT90S8535 i większość układów serii ATmega. Niestety przetwornika takiego nie zaimplementowano w układzie AT90S2313. W zamian za to znajduje się w jego strukturze komparator analogowy. Dokładając stosunkowo niewiele elementów zewnętrznych można



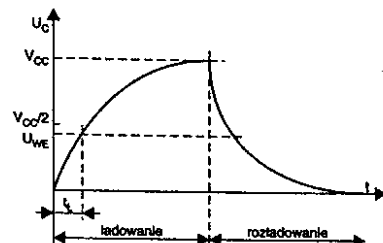
Rys. 14.4. Schemat elektryczny przetwornika analogowo-cyfrowego wykonanego na komparatorze analogowym

za jego pomocą zrealizować przetwornik analogowo-cyfrowy. Jego parametry nie są z pewnością tak dobre jak układów wyspecjalizowanych, czy przetworników zawartych w strukturach mikrokontrolerów, ale są w zupełności wystarczające w wielu zastosowaniach. Schemat elektryczny i ogólną zasadę działania takiego przetwornika przedstawiono na rysunkach 14.4 i 14.5.

Kondensator pomiarowy C dołączono do nieodwracającego wejścia komparatora analogowego oraz – poprzez opornik R – do jednego z portów mikrokontrolera (np. PD4). Do wejścia odwracającego komparatora jest doprowadzony analogowy sygnał mierzony. Cykl pomiarowy rozpoczyna się od ustawienia stanu wysokiego na wyjściu PD4, w wyniku czego kondensator C zaczyna się ładować przez opornik R do napięcia zasilającego V_{CC} . W tym samym momencie mikrokontroler powinien uruchomić pomiar czasu (po wcześniejszym wyzerowaniu rejestrów timera). Ładowanie przebiega zgodnie z zależnością:

$$V_C = V_{CC}(1 - \exp \frac{-t}{RC}) \quad (14.1)$$

W momencie, gdy napięcie na kondensatorze osiągnie wartość mierzonego napięcia wejściowego, na wyjściu komparatora nastąpi zmiana stanu. Fakt ten powinien spowodować zatrzymanie pomiaru czasu, który jak widać będzie proporcjonalny do napięcia wejściowego. W rzeczywistości zmiana stanu na wyjściu komparatora najczęściej będzie wyzwała funkcję przechwytywania Timer1, czyli zatrzaśnięcie wyniku w rejestrach ICR1H i ICR1L. Z zależności opisanej wzorem 14.1 wynika, że ładowanie kondensatora nie przebiega liniowo. Przeliczenie czasu na napięcie nie będzie więc łatwe. Wprawdzie mając do dyspozycji mikrokontroler można zależność tę zlinearyzować programowo, lecz będzie się to wiązało z koniecznością zastosowania niemałych bibliotek matematycznych. Inna metoda, to stabilizowa-



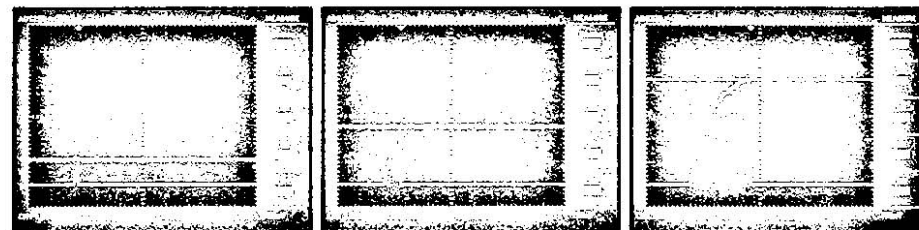
Rys. 14.5. Przebieg cyklu ładowania/rozładowania kondensatora referencyjnego C

nie funkcji ładowania. W tym przypadku trzeba będzie z kolei zająć sporo miejsca w pamięci programu. Na rysunku 14.6 przedstawiono kilka oscylogramów rzeczywistego układu mierzącego na zasadzie ładowania kondensatora przez opornik. Oglądając te przebiegi zauważamy, że mniej więcej do połowy napięcia V_{CC} , ładowanie można uznać za liniowe. Po wprowadzeniu odpowiednich ograniczeń dla napięcia wejściowego mamy więc kolejną metodę. Obliczeń dokonujemy stosując zwykle przekształcenia liniowe bez dołączania do programu bibliotek matematycznych. Zakładamy przy tym, że popełniany błąd jest do zaakceptowania. Cykl pomiarowy kończy się podaniem stanu niskiego na wyjściu PD4, w wyniku czego kondensator jest rozładowywany przez rezystor R i niewielką oporność wyjściową portu PD4 w tym stanie.

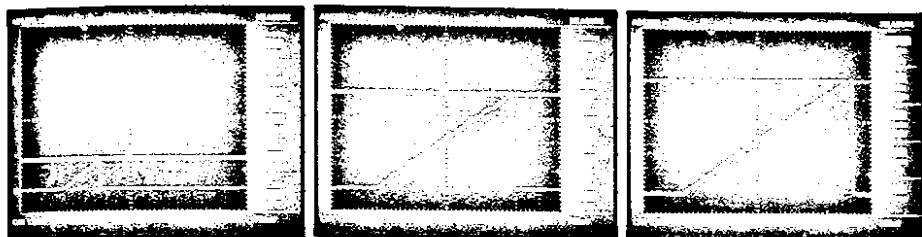
Wykorzystanie elementarnej wiedzy z fizyki pomoże nam w opracowaniu jeszcze jednego układu pomiarowego. Jak pamiętamy ze szkoły, ładunek zgromadzony w kondensatorze jest proporcjonalny do pojemności tego kondensatora i napięcia panującego na nim. Z drugiej strony ładunek jest iloczynem prądu i czasu. Wykorzystując ten fakt uzyskujemy zależność $I \cdot t = U \cdot C$, z której wynika wprost, że ładowanie kondensatora będzie miało charakter liniowy, jeśli tylko zostanie zapewniona stała wartość prądu. Na płycie ZL1AVR zastosowano właśnie takie rozwiązanie. Rolę źródła prądowego pełnią tranzystory Q1, Q2 i rezystor R1, pracujące w układzie lustra prądowego. Prąd kolektora tranzystora Q1 jest odzwierciedleniem prądu kolektora tranzystora Q2, a ten z kolei wynika z wartości rezystora R1:

$$I_{R1} = \frac{V_{CC} - U_{BEP}}{R1} \quad (\text{wzór 14.2})$$

Tranzystory powinny mieć jak najbardziej zbliżone parametry. Idealem byłoby, gdyby były wykonane w tej samej strukturze półprzewodnikowej. Odpowiednie elementy są dostępne w handlu, jednak zrezygnowano z nich na rzecz popularnych tranzystorów, które można kupić w każdym sklepie elektronicznym. Kondensator pomiarowy C5 jest dołączony do wejścia nieodwra-



Rys. 14.6. Oscylogramy układu pomiarowego wykorzystującego ładowanie kondensatora prądem płynącym przez rezystor



Rys. 14.7. Oscylogramy układu pomiarowego wykorzystującego ładowanie kondensatora przez źródło prądowe

cającego AIN0 (PB0). Jest on ładowany stałym prądem ze źródła prądowego zbudowanego z tranzystorów Q1, Q2. Zależność napięcia na kondensatorze w funkcji czasu określa wzór 14.3:

$$U_C = \frac{I \cdot t}{C} \quad (\text{wzór 14.3})$$

Jak widać jest to zależność liniowa. W praktyce oczywiście nie jest tak idealnie jak na papierze, ale błędy są do przyjęcia (patrz rozdział 14.3.6). Mierzone napięcie należy doprowadzić się do wejścia AIN1 (PB1) poprzez (uwaga!) końcówkę 1 gniazda PORTD. Tranzystor Q3 (wraz z rezystorem R5), sterowany z wyjścia PD4 umożliwia szybkie rozładowania kondensatora C5. Pracuje od nasycenia do zatkania, jak typowy klucz tranzystorowy. Tak zrealizowany przetwornik będzie pracował niemal w całym zakresie napięć wejściowych, tzn. od zera do V_{CC} . Jednak ze względu na nieznaczne pogorszenie charakterystyki dla dużych wartości napięcia wejściowego, celowe jest jego ograniczenie do ok. 4,8 V przy zasilaniu 5 V. Oscylogramy przedstawione na **rysunku 14.7** pokazują jak wygląda ładowanie kondensatora ze źródła prądowego w rzeczywistym układzie (na płytce ZL1AVR). Widać wyraźną poprawę liniowości ładowania kondensatora w porównaniu z wcześniej opisaną metodą.

14.1.10. Programowanie mikrokontrolera w systemie (ISP)

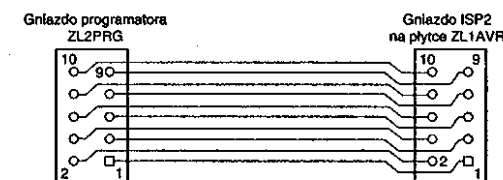
W punkcie 11.4.2 opisano szeregowe programowanie pamięci Flash mikrokontrolera. Wykorzystywany jest do tego celu interfejs SPI, zaimplementowany w większości mikrokontrolerów AVR. Dwukierunkowa transmisja szeregową jest prowadzona liniami MOSI (wejscie danych dla mikrokontrolera), MISO (wyjście danych) i SCK (zegar synchronizujący transmisję). Mikrokontroler umieszczony na płytce uruchomieniowej może być programowany za pomocą dowolnego programatora zgodnego z tym standardem (np. ZL2PRG). Przykładową konstrukcję opisano w rozdziale 13.



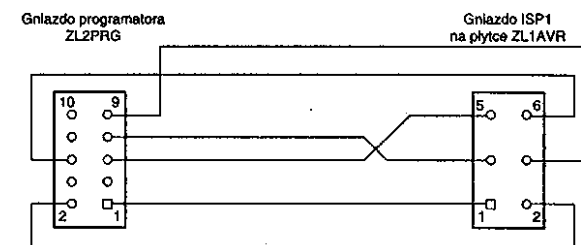
Płytki drukowane do programatora ZL2PRG oraz jego dokumentacja są dostępne w internetowym sklepie Wydawnictwa BTC (<http://www.btc.pl/index.php?id=zl2prg>).

Stosowanie programowania szeregowego jest bardzo wygodne dla użytkownika, gdyż uwalnia go od uciążliwego przekładania układu z podstawki zestawu do podstawki programatora. Ci Czytelnicy, którzy zdecydują się pisać swoje programy w Bascomie, będą mogli robić to w jednym, zintegrowanym środowisku. Trzeba jednak pamiętać, że programowanie szeregowie nie jest dokładnym odpowiednikiem programowania równoległego. Dotyczy to ustawiania bitów konfiguracyjnych (*fuse bits*). Z tego względu warto jednak zamontować na płytce podstawkę pod mikrokontroler.

Programator ZL2PRG najwygodniej jest dołączyć bezpośrednio do gniazda ISP2 przez kabel wykonany z taśmy 10-przewodowej. Połączenia powinny być wykonane 1:1 (styk 1 na styk 1, 2 na 2 itd.), zgodnie z **rysunkiem 14.8**. Na płytce ZL1AVR znajduje się ponadto gniazdo ISP1 zgodne ze specyfikacją Atmela. Zawiera ono tylko 6 styków. W przypadku wykorzystywania tego gniazda do programowania mikrokontrolera, niezbędne będzie wykonanie kabla przejściowego. Sposób jego wykonania przedstawiono na **rysunku 14.9**. W tym przypadku dioda sygnalizacyjna LED9 nie będzie działała, po-



Rys. 14.8. Kabel połączeniowy dla programatora ZL2PRG dołączonego do gniazda ISP2



Rys. 14.9. Kabel przejściowy dla programatora ZL2PRG dołączonego do gniazda ISP1

nieważ jest ona sterowana sygnałem występującym wyłącznie na jednym ze styków ISP2, sygnalizując aktywność interfejsu.

14.2. Ćwiczenia praktyczne

14.2.1. Ćwiczenie 1

Sterowanie portami mikrokontrolera w trybie wyjściowym – efekt węża świetlnego i biegnącego punktu na linijce diod LED

Poznanie mikrokontrolera rozpoczniemy od kilku przykładów obsługi jego portów. W większości praktycznych aplikacji będziemy się spotykać z podobnym zagadnieniem, wszak jeśli już w urządzeniu zastosowano mikrokontroler, to oczywiste jest, że musi wykonywać jakieś zadania. A w jaki sposób, jak nie poprzez porty we/wy, ma on przekazywać efekty swojej pracy do reszty systemu? W ćwiczeniu nauczymy się ustawiać linie portów w tryb wyjściowy i zmieniać ich stan.

Mikrokontroler AT90S2313 wyposażono w dwa porty we/wy: PORTB i PORTD. Mogą one pracować jako porty ogólnego przeznaczenia lub można wykorzystywać ich funkcje alternatywne (patrz rozdział 3.1). Do wysterowania diod świecących skorzystamy z ich podstawowych możliwości. Porty mikrokontrolerów AVR są obsługiwane przez trzy rejestry umieszczone w przestrzeni we/wy. Użycie portu powinno być poprzedzone odpowiednim skonfigurowaniem go, najlepiej we wstępnej fazie programu. Poszczególne linie portów mogą być indywidualnie ustawiane jako wejściowe lub wyjściowe. Dokonuje się tego poprzez wpis odpowiednich wartości do rejestru kierunku (np. dla portu B jest to DDRB). Wpisanie „1” na daną pozycję oznacza, że odpowiadające jej wyprowadzenie portu będzie pracowało jako wyjściowe. Analogicznie „0” konfiguruje port jako wejściowy (patrz następne ćwiczenia). Linijka diod LED na płycie uruchomieniowej jest dołączona bezpośrednio do portu B. Jak pamiętamy, porty mikrokontrolerów AVR mają wystarczającą wydajność prądową do ich wysterowania. Aby zaświecić wy-



Źródłowe i wynikowe wersje programów do ćwiczeń przedstawionych w dalszej części książki są dostępne na stronie internetowej Wydawnictwa BTC (<http://www.btc.pl?id=avr>).

braną diodę należy odpowiednie wyjście portu ustawić w stanie niskim, czyli wpisać „0” na odpowiednią pozycję rejestru PORTB. Ustawienie linii w stanie wysokim powoduje zgaszenie LED-a. Oczywiście dołączenie diod świecących w niniejszym ćwiczeniu jest podyktowane tylko chęcią pokazania „namacalnego” efektu sterowania portami.

Na wyjściach portu występują napięcia o wartościach, które umożliwiają współpracę mikrokontrolera ze zwykłymi układami cyfrowymi. Jedyne na co trzeba ewentualnie zwrócić uwagę, to upewnienie się o kompatybilności układów zasilanych różnymi napięciami, gdyż coraz powszechniejsze stają się układy zasilane napięciem np. 3,3 V. Niektóre układy występują już tylko w wersjach niskonapięciowych. W takich przypadkach może okazać się niezbędne stosowanie odpowiednich interfejsów. Przykład z diodami LED ma jeszcze jeden praktyczny walor. W wielu aplikacjach wymagane jest galwaniczne odizolowanie systemu mikroprocesorowego od współpracujących z nim urządzeń. Jeśli wymiana informacji dotyczy sygnałów cyfrowych, najprostszą metodą zrealizowania separacji jest zastosowanie transoptorów, a sterowanie nimi, to przecież nic innego jak gaszenie i zapalanie diod LED. Podobnie jest z przełączaniem obwodów dużej mocy za pomocą optotriaków lub optoprzełączników.

Sterowanie portu można zrealizować poprzez wpisanie całego bajtu o odpowiedniej wartości lub poprzez manipulowanie pojedynczymi bitami. W języku C realizuje się to instrukcjami np.:

```
PORTB=0x0f;    //wyprowadzenia PB7...PB4 w stanie "0", PB3...PB0
                //w stanie "1".
sbi(PORTB,5);   //ustaw bit 5 portu B w stanie wysokim ("1")
                //bez zmiany pozostałych
cbi(PORTB,7);   //ustaw bit 7 portu B w stanie niskim ("0")
                //bez zmiany pozostałych
```

Do sterowania indywidualnymi bitami portu można również stosować bardziej naturalny zapis. Przykład będzie podany w ćwiczeniu 9.



Aktualne wersje kompilatora AVR-GCC pozwalają, a nawet wymagają stosowania zapisu np. `PORTB=wartość`. W wersjach wcześniejszych wymagane było użycie makra `outp(wartość, port)`.

Stany wyjść portu sterującego diodami świecącymi w tym ćwiczeniu są zmieniane cyklicznie. Czas cyklu wyznaczony jest na sztywno za pomocą funkcji `czekaj`. Jest to funkcja uniwersalna, w zależności od wartości argu-

mentu funkcji pt generowane może być opóźnienie o różnej długości. W programie z listingu 14.1 funkcja ta jest wywoływana z parametrem pczekaj. Jest to zmienna typu unsigned long, inicjowana w miejscu deklaracji wartością 1500.

Program realizujący przedstawione zadanie znajduje się na listingu 14.1. Należy skompilować program *cwicz1.c* i zaprogramować mikrokontroler z zestawu ZL1AVR plikiem wynikowym *cwicz1.hex*.

Konfiguracja płytki ZL1AVR:

- zworka J3 w pozycji 1-2, zworka J4 w pozycji 2-3 – dołączony oscylator wewnętrzny,
- zworka JPI zwarta – globalne włączenie diod LED,
- zworki ZW_PORTB zwarte (wszystkie) – włączone wszystkie diody LED,
- położenie pozostałych zwerek nieistotne (np. rozłączone).

List. 14.1. Program do ćwiczenia 1

```

/*****
/* Ćwiczenie 1 - sterowanie portami w trybie wyjściowym */
/* Efekty świetlne na linijce LED-ów */
/* J.D. '2003 */
*****/
#include <io.h>
unsigned long pczekaj=1500;

void czekaj(unsigned long pt) //funkcja opóźnienia
{
    unsigned char tpl;

    for(;pt>0;pt--)
    {
        for(tpl=255;tpl!=0;tpl--);
    }
}

int main(void)                //program główny
{
    unsigned char ledy,i,licznik;

    DDRB=0xff;                //konfiguracja wszystkich wyprowadzeń
                                //portu B w trybie wyjścia
    while(1)                   //nieskończona pętla główna programu
    {
        //efekt węża
        for(licznik=0;licznik<10;licznik++)
            //liczba powtórzeń efektu
        {
            PORTB=0xff;        //wygaś LED-y
            czekaj(pczekaj);
            for(i=0;i<8;i++)    //pętla zmieniająca fazę efektu
            {
                cbi(PORTB,i);  //wysteruj (zapal) pojedynczego LED-a
                czekaj(pczekaj);
            }
        }
    }
}

```

```

for(i=0;i<8;i++)              //pętla zmieniająca fazę efektu
{
    sbi(PORTB,i);              //wysteruj (zgaś) pojedynczego LED-a
    czekaj(pczekaj);
}

//efekt biegnącego punktu
PORTB=0xff;                   // wygaś LED-y
for(licznik=0;licznik<10;licznik++)
    //liczba powtórzeń efektu
{
    for(ledy=0xfe;ledy!=0xff;ledy=(ledy<<=1)+1)
        //pętla zmieniająca fazę efektu
    {
        PORTB=ledy;            //wysterowanie LED-ów zgodne z wartością
                                //zmiennej ledy
        czekaj(pczekaj);        //opóźnienie
    }
}
}
}

```

14.2.2. Ćwiczenie 2

Wykorzystanie timera do odmierzania czasu w trybie odpytywania (generator przebiegu prostokątnego o częstotliwości 1 kHz)

W poprzednim ćwiczeniu do odmierzania czasu trwania każdego cyklu efektu była wykorzystywana funkcja *czekaj* (pt). Uzyskiwanie opóźnienia odbywało się na drodze programowej. Czas opóźnienia jest zależny od wartości zmiennej pt, jednakże dokładne jego określenie w normalnie stosowanych do tego celu jednostkach (s, ms, μ s) w ogólnym przypadku jest dość kłopotliwe. Można oczywiście zadać sobie trud i policzyć liczbę cykli zegarowych CPU potrzebnych na wykonanie tej funkcji dla określonego parametru, jednak nietrudno zauważyć, że precyzja odmierzania czasu nie będzie zbyt wysoka. Po zastosowaniu rezonatora kwarcowego o innej częstotliwości niż przewidziana, odmierzany czas będzie miał inną wartość. Na szczęście we wszystkich mikrokontrolerach AVR jest zaimplementowany co najmniej jeden timer, który jest nieodzowny w takich sytuacjach. W przypadku AT90S2313 użytkownik ma do dyspozycji jeden timer/licznik 8-bitowy i jeden 16-bitowy (patrz rozdział 5). Obydwa timery są wyposażone w preskalery rozszerzające możliwości pomiarowe poprzez wstępny podział odmierzanych czasów (impulsów). W ćwiczeniu 2 jest wykorzystany Licznik/Timer0. Nasze zadanie polega na wygenerowaniu fali prostokątnej o częstotliwości 1 kHz. Zadanie wydaje się dość proste. Timer musi odmierzać czas równy

połowie okresu generowanego przebiegu (500 μ s). Po odmierzeniu tego czasu mikrokontroler musi zmienić stan portu wyjściowego na przeciwny, po czym uruchomić odmierzenie kolejnego opóźnienia. Aby zminimalizować błąd wynikający z niezerowego czasu reakcji na wykrycie odliczenia pół-okresu, tuż po wystąpieniu tego zdarzenia, do rejestru TCNT0 zostaje wpisana wartość odpowiadająca odmierzeniu żadanego czasu. Tym samym rozpoczyna się pomiar kolejnego półokresu. Dopiero po tym następuje zmiana stanu wyjścia (po sprawdzeniu dodatkowego warunku). Generowany przebieg będzie wyprowadzony na wyjście PB0. Port B zostanie w całości skonfigurowany jako wyjściowy. Największą trudnością w tym ćwiczeniu będzie obliczenie wartości, którą należy wpisywać do rejestru TCNT0 (nazwiemy ją stałą czasową, choć to trochę nieprecyzyjne określenie) oraz parametru preskalera wpisywanego do rejestru TCCR0. Wartości powyższe powinny być tak dobrane, by za pomocą timera możliwe było odmierzenie czasu równego dokładnie 500 μ s. Przypomnijmy, że timer/licznik TC0 zgłasza fakt przepełnienia (przejścia od stanu \$FF do stanu \$00) poprzez ustawienie flagi TOV0 w rejestrze TIFR. Inkrementacja wartości timera/licznika następuje zgodnie z ustawionym parametrem preskalera (tablica 5.1). Przykładowo, jeśli do rejestru TCCR0 zostanie wpisana wartość 2, to inkrementacja będzie następowała co 8 okresów oscylatora systemowego. Przy rezonatorze kwarcowym o częstotliwości 8 MHz, kwant odmierzanego czasu będzie więc równy 1 μ s. Zwiększenie stopnia podziału niestety doprowadzi do uzyskania niewygodnych wartości odmierzanego czasu, utrudniających lub wręcz uniemożliwiających uzyskanie założonego czasu. Przykładowo dla stałej czasowej równej 500 μ s przy kwarcu 8 MHz, liczba zliczanych odcinków czasu (L) byłaby równa:

- dla TCCR0 = 1 (CK/1), $\Delta t = 125$ ns, L = 4000,
- dla TCCR0 = 2 (CK/8), $\Delta t = 1$ μ s, L = 500,
- dla TCCR0 = 3 (CK/64), $\Delta t = 8$ μ s, L = 62,5,
- dla TCCR0 = 4 (CK/256), $\Delta t = 32$ μ s, L = 15,625,
- dla TCCR0 = 5 (CK/1024), $\Delta t = 128$ μ s, L = 3,90625.

Przyjęcie opcji z niecałkowitą wartością L będzie równoznaczne z zaakceptowaniem pomiaru czasu obciążonego pewnym błędem. W ogólnym przypadku może się okazać, że uzyskanie całkowitych wartości L nie będzie możliwe. Wówczas należy wybrać taką opcję, która da najmniejszy błąd pomiaru czasu. Zgodnie z powyższym, do naszego ćwiczenia najlepiej nadają się dwa pierwsze przypadki. Oba mają jednak tę wadę, że wymagają zliczania więcej niż 256 kwantów czasu, co przy 8-bitowym rejestrze TCNT0 jest niewykonalne. Można zastosować timer/licznik TC1, ale może on być przydatny do innych zadań.

Pozostawmy więc go w rezerwie, wybierzmy opcję drugą, a problem zliczenia 500 impulsów rozwiążemy częściowo na drodze programowej. Zliczanie zostanie podzielone na dwa etapy (2-krotne zliczenie 250 impulsów). Jak pamiętamy timery/liczniki mikrokontrolerów AVR liczą w przód. Timer/licznik TC0 może zliczać modulo 256 (czyli liczyć od 0 do 255). Jeśli liczba zliczanych impulsów ma być mniejsza, należy przed rozpoczęciem zliczania do rejestru TCNT0 wpisać odpowiednią różnicę: $TCNT0 = 256 - t$. W naszym ćwiczeniu, zgodnie z powyższymi założeniami $t = 250$. Rejestr TCNT0 powinien więc być inicjowany wartością: $TCNT0 = 256 - 250 = 6$.

W programie zdefiniowano stałą tau0 (dyrektywą kompilatora #define), która będzie wpisywana do rejestru TCNT0 na początku programu i każdorazowo, po ustawieniu flagi przepełnienia TOV0. W przypadku zastosowania rezonatora o innej częstotliwości lub zmiany częstotliwości generatora, należy tę stałą przedefiniować zgodnie z powyższymi wskazówkami. Każdorazowe przekręcenie timera TC0 powoduje predekrementację zmiennej licznik (ustawianej wstępnie na wartość 2) i sprawdzenie, czy osiągnęła ona wartość 0. Jeśli tak, to znaczy, że zostało odliczone liczba 500 kwantów czasu, w wyniku czego powinien być zmieniony stan wyjścia PB0. Odbywa się to poprzez wykonanie operacji Ex-OR na bicie 0 rejestru PORTB, a następnie zmiennej licznik nadawana jest ponownie wartość 2. Zastosowanie działania Ex-OR pozwala na selektywną zmianę jednego bitu w całym rejestrze, pozostałe mogą przecież być wykorzystywane do innych celów.

Jeśli flaga TOV0 jest kontrolowana programowo, to trzeba pamiętać o jej „ręcznym” zerowaniu po wystąpieniu przepełnienia. Pamiętajmy również, że flagę tę zeruje się poprzez wpisanie bitu o wartości „1” na pozycji TOV0, stąd w programie pojawia się instrukcja $TIFR = 1 << TOV0$.

Przy okazji warto porównać, jak kompilator przetłumaczy instrukcję `while(bit_is_clear(TIFR,TOV0))` użytą zamiast instrukcji `while((inp(TIFR)&0x02)!=0x02)`. Jak widać, czasami trud twórców kompilatora wydaje się być zupełnie zbędny. Proponowane ułatwienie (wątpliwe zresztą) powoduje zarówno zwiększenie długości kodu, jak również czasu jego wykonania. W konkretnym przypadku, w pierwszej wersji pętla wykonuje się 0,88/0,75 μ s (dla warunku spełnionego/nie spełnionego), w drugim natomiast czasy te wynoszą 0,5/0,38 μ s. Okazuje się, że nawyki nabyte podczas pisania programów w assemblerze czasami są bardzo przydatne nawet podczas pisania programów w językach wysokiego poziomu.

Program realizujący przedstawione zadanie znajduje się na listingu 14.2. Należy skompilować program *cwicz2.c* i zaprogramować mikrokontroler z zestawu ZL1AVR plikiem wynikowym *cwicz2.hex*.

Konfiguracja płytki ZL1AVR:

- zworka J3 w pozycji 1-2, zworka J4 w pozycji 2-3 – dołączony oscylator wewnętrzny,
- zworka ZW_PORTB zwarta w pozycji 15-16 – wyjście generatora na gnieździe PORTB-8,
- położenie pozostałych zwerek nieistotne (np. rozłączone).

List. 14.2. Program do ćwiczenia 2

```

/*****
/* Ćwiczenie 2 - Obsługa timera0 w trybie odpytywania */
/* Generator fali prostokątnej 1kHz */
/* J.D. '2003 */
*****/
#include <io.h>
#define tau0 6; //stała czasowa dla 1kHz @8MHz

int main(void)
{
    unsigned char licznik=2;

    DDRB=0x01; //wyjściem generatora będzie PB0
    TCNT0=tau0; //wpisz stałą czasową dla zadanego interwału
    TCCR0=2; //timer0 będzie pracował z preskalerem Fosc/8

    while(1)
    {
        while((inp(TIFR)&0x02)!=0x02);
        //czekaj na ustawienie flagi
        //TOV0 (przekreślenie licznika)

        TCNT0=tau0; //wpisz stałą czasową
        if(--licznik==0) //zmiana polaryzacji wyjścia wymaga
            //2-krotnego przekreślenia się licznika
        {
            PORTB^=0x01; //zmień stan wyjścia
            licznik=2; //odśwież stan licznika
        }
        TIFR=1<<TOV0; //kasuj flagę przepełnienia
    }
}

```

14.2.3. Ćwiczenie 3

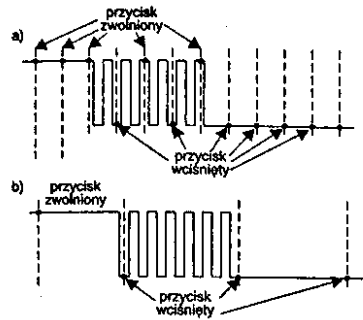
Sterowanie portami mikrokontrolera w trybie wejściowym, wykorzystanie timera do odmierzania czasu z wykorzystaniem przerwań – obsługa przycisków dołączonych do portów mikrokontrolera

W poprzednim ćwiczeniu nauczyliśmy się korzystać z timera. Kontrola przepełnienia odbywała się poprzez programowe sprawdzanie flagi TOV0. Załóżmy jednak, że jest to marnotrawstwo czasu mikrokontrolera. Zamiast

przebywać w pętli sprawdzania flagi TOV0 przez większą część programu, procesor mógłby zająć się innymi zadaniami. Ale tu z kolei całkiem prawdopodobna mogłaby być sytuacja, w której jednostka centralna, zajęta innymi zadaniami zupełnie zaniedbałaby sprawdzanie flagi TOV0, co z kolei prowadziłoby do powstawania dużych błędów pomiaru czasu. W niektórych aplikacjach jest to rozwiązanie nie do przyjęcia. Z pomocą przychodzą przerwania. W mikrokontrolerach AVR niemal każdy ich blok funkcjonalny może być źródłem przerw (patrz rozdział 4.10).

Przerwania umożliwiają obsługę asynchronicznych zdarzeń przez sekwencyjnie wykonywany program. Sygnał zgłoszenia przerwania może wystąpić w dowolnym miejscu programu. W tym momencie CPU zawiesza wykonywanie aktualnego kodu (kończąc wykonywany rozkaz) i przechodzi do specjalnej procedury obsługi przerwania. Zawsze kończy się ona rozkazem RETI, który „zacier” ślady po przerwaniu (może np. ustawiać flagę I w rejestrze SREG, jeśli zezwolono wcześniej na przerywanie wykonywania procedury obsługi). Następnie kontynuowane jest wykonywanie programu od miejsca, w którym został on zawieszony. Procedura obsługi przerwania jest ściśle związana z jego źródłem (każdy blok funkcjonalny zgłaszający przerwanie ma własną procedurę obsługi). Możliwe przypadki zostaną opisane w następnym ćwiczeniu. W tym przykładzie przerwania zostaną wykorzystane do obsługi timera odmierającego chwile, w których będą testowane stany dwóch przycisków dołączonych do portu D mikrokontrolera.

Obsługa przycisków (klawiatury) nie jest tak banalna, jak mogłoby się wydawać. Problemy są związane z drganiami styków występujących w trakcie przełączeń. W zależności od konstrukcji mechanicznej i materiału, z jakiego są zrobione styki, drgania te mogą trwać dłużej lub krócej, ale występują prawie zawsze, zwłaszcza w starszych konstrukcjach. Można przyjąć, że czas drgań wynosi kilkadziesiąt milisekund. Czasami stosowane są do ich gaszenia kondensatory. W urządzeniach, w których nie ma mikroprocesora, do likwidacji drgań używa się przerzutników RS. Jeśli jednak w urządzeniu zastosowano mikrokontroler, to najprostszą metodą eliminacji drgań jest odpowiednia obsługa programowa. Przyciski będą najczęściej dołączane do portu wejściowego tak, aby zwierały go do masy. Jest to podyktowane tym, że w mikrokontrolerach AVR możliwe jest skonfigurowanie wejścia z wewnętrznym podciąganiem „do góry” (do plusa zasilania – *pull-up*). Nie jest wtedy wymagany zewnętrzny rezystor podciągający. Aby zrealizować takie ustawienie, należy do rejestru kierunku, np. DDRD, na odpowiednim bicie (pamiętamy, że poszczególne wyprowadzenia jednego portu mogą być konfigurowane niezależnie jako wejściowe lub wyjściowe) wpisać wartość „0”, a do



Rys. 14.10. Dobranie odpowiedniej częstotliwości skanowania klawiatury pozwala na uniknięcie błędów odczytu: a) błędnie zinterpretowano trzy naciśnięcia klawisza, b) prawidłowo zinterpretowano jedno naciśnięcie klawisza

okresowym sprawdzaniu ich stanu, a następnie podejmowaniu odpowiednich działań. Czas drgań styków szacowany na kilkadziesiąt milisekund, to dla mikrokontrolera – szczególnie AVR – niemal wieczność. Jest on zdolny do o wiele częstszego próbkowania klawiatury. Jeśli jej sprawdzenie nastąpi kilkukrotnie w czasie drgań styków, to może się okazać, że zamiast jednego wciśnięcia przycisku, zostanie wykryte wielokrotne naciśnięcie. Najprostszą metodą na uniknięcie takiej sytuacji jest odpowiednio rzadkie próbkowanie klawiatury (rysunek 14.10) lub odczekiwanie określonego czasu po wykryciu zmiany stanu przycisku. Dopiero po jego upływie można będzie ponownie odczytać stan przycisku.

W ćwiczeniu przyciski będą służyły do inkrementacji (SW1) i dekrementacji (SW4) 8-bitowego licznika, którego stan w postaci binarnej będzie wyświetlany na linijce diod LED. Wyprowadzenia PD0 do PD4 portu D zostaną skonfigurowane jako wejściowe, natomiast port B i PD6 i PD5 jako wyjściowe. Stan klawiszy będzie sprawdzany w procedurze obsługi przerwania od timera TC0, odmierzającego momenty próbkowania. Zakładamy, że sprawdzanie będzie następowało co ok. 130 ms. Taki czas pozwoli skutecznie eliminować drgania styków i jednocześnie spowoduje, że możliwa będzie prosta metoda zrealizowania przycisku z funkcją autopowtarzania. Dzięki temu, nie będzie potrzebne zwalnianie klawisza w celu powtórzenia działania z nim związanego. Odliczenie 130 ms przy kwarcu 8 MHz nie jest możliwe wprost, nawet przy największym stopniu podziału preskalera. W tej sytuacji należy wprowadzić dodatkową zmienną, która będzie zliczała wejścia do przerwania i dopiero po osiągnięciu odpowiedniej liczby, będą podejmowane stosowne

rejestru PORTD wpisać wartość „1”. Niewpisanie jedynki do PORTD spowoduje, że wejście pozostanie bez podciągania i prawidłowy odczyt przełącznika nie będzie możliwy (chyba, że zostanie zastosowane podciąganie zewnętrznymi rezystorami). Czytanie portów wejściowych odbywa się poprzez rejestry PINB (dla portu B) i PIND (dla portu D). Jeśli przycisk będzie zwolniony, rezystor podciągający będzie wymuszał na wejściu stan wysoki. Zwarcie przycisku spowoduje odczytanie stanu niskiego. W ćwiczeniu zostaną wykorzystane klawisze SW1 i SW4 dołączone odpowiednio do portów PD0 i PD1.

Obsługa klawiszy będzie polegała na

działania. Biorąc pod uwagę rozważania z poprzedniego ćwiczenia zostanie wybrany preskaler CK/1024 (TCCR0 = 5), więc będzie odmierzany czas 128 μ s. Jeśli przyjmiemy wpis do rejestru TCNT0 równy 247, to przerwania od timera TC0 będą generowane co $t = (256 - 247) \cdot 128 \mu s = 1,152$ ms. Pozwoli nam to, niejako przy okazji, generować przebieg prostokątny o częstotliwości ok. 434 Hz ($f = 1/(2 \cdot t)$) na wyjściu PD6 (w celach czysto edukacyjnych, dla zilustrowania, że jedno przerwanie może pełnić kilka zadań). Obsługa klawiszy będzie wymagała przy powyższych założeniach 113-krotnego wejścia w procedurę obsługi przerwania timera. Liczba wejść będzie zliczana w zmiennej liczt0. Aby ułatwić ewentualne eksperymenty, na początku programu została zdefiniowana (dyrektywą kompilatora #define) stała vliczt0, która może być wielokrotnie użyta w programie. Stała ta będzie określała częstotliwość próbkowania klawiatury, będąc jednocześnie czasem samopowtarzania klawisza. Wartość vliczt0 oczywiście nie odpowiada żadnym jednostkom fizycznym, wynika jedynie z powyższych kalkulacji. Mierzając czas naciśnięcia klawisza (np. poprzez wprowadzenie dodatkowej zmiennej) można również zrealizować przycisk z efektem działania progresywnego. Klawisz taki mógłby zwiększać krok zliczania zmiennej licznik w zależności od czasu przyciśnięcia. Zwolnienie klawisza przywracałoby krok domyślny.

W języku AVR-GCC procedury obsługi przerwań są wyróżniane specjalnymi nazwami oraz słowem kluczowym SIGNAL lub INTERRUPT. Wykaz nazw przerwań można znaleźć w dokumentacji kompilatora. Procedury oznaczone słowem SIGNAL są wykonywane przy zablokowanych przerwaniach (I=0 w rejestrze SREG). Odwrotnie jest w przypadku procedur ze słowem kluczowym INTERRUPT. Podprogram obsługi przepełnienia timera TC0 zaczyna się od linii SIGNAL (SIG_OVERFLOW0).

Program realizujący przedstawione zadanie znajduje się na **listingu 14.3**. Należy skompilować program *cwicz3.c* i zaprogramować mikrokontroler z zestawu ZL1AVR plikiem wynikowym *cwicz3.hex*.

Konfiguracja płytki ZL1AVR:

- zworka J3 w pozycji 1-2, zworka J4 w pozycji 2-3 – dołączony oscylator wewnętrzny,
- zworka JP1 zwarta – globalne włączenie diod LED,
- zworki ZW_PORTB zwarte (wszystkie) – włączone wszystkie diody LED,
- zworka JP4 w pozycji 2-3 (włączenie klawiszy SW1 i SW4),
- zworka JP7 zwarta (włączenie klawisza SW4),
- zworka JP6 zwarta (włączenie klawisza SW1),
- położenie pozostałych zworek nieistotne (np. rozłączone).

List. 14.3. Program do ćwiczenia 3

```

/*****
/* Ćwiczenie 3 - sterowanie portami w trybie wejściowym */
/* Obsługa przycisków dołączonych do portów mikrokontrolera */
/* Licznik rewersyjny naciśnięć klawiszy */
/* J.D. '2003 */
*****/
#include <io.h>
#include <interrupt.h>
#include <signal.h>

#define tau0 247;           //stała czasowa timera0
#define vliczt0 113;       //stała wpisywana do licznika wejść do
                           //do przerwania timera0

/***** zmienne globalne *****/
unsigned char liczt0;       //Licznik wejść do przerwania timera0.
                           //Klawisz jest badany, gdy liczt0=0
unsigned char licznik;      //Licznik rewersyjny, którego stan jest
                           //wyświetlany na linijsce LED-ów

void czekaj(unsigned long zt) //funkcja opóźnienia
{
    unsigned char zt1;
    for(;zt>0;zt--)
    {
        for(zt1=255;zt1!=0;zt1--);
    }
}

SIGNAL (SIG_OVERFLOW0)      //obsługa przerwania
                           //od przepełnienia timera0
{
    TCNT0=tau0;             //odśwież stałą czasową w TCNT0
    PORTD^=1<<PD6;         //zmień stan wy generatora 434Hz
    if(--liczt0==0)         //czy już czytać klawisze?
    {
        //tak
        if(bit_is_clear(PIND,PD1)) //czytaj SW4 - <->
        {
            licznik--;
        }
        else
        {
            if(bit_is_clear(PIND,PD0)) //czytaj SW1 - <+>
            {
                licznik++;
            }
        }
    }
    PORTB=~licznik;         //wyświetl stan licznika na LED-ach
                           //negacja jest potrzebna, gdyż LEDy są
                           //zapalane niskim stanem
    liczt0=vliczt0;         //odśwież stan liczt0
}

int main(void)
{
    liczt0=vliczt0;

    DDRD=0x60;             //PD0-PD4 jako wejściowy, PD5-PD6 - wy
    PORTD=0xff;             //z podciąganiem

```

```

DDRB=0xff;                //PORTB - wy
PORTB=0xff;               //LED-y wygaszone
TIMSK=1<<TOIE0;          //zezwoleń na przerwania od TC0
TCNT0=tau0;               //wpisz stałą czasową do TCNT0
TCCR0=5;                  //preskaler XTAL/1024,
                           //mierzony czas = 128µs
                           //odblokuj globalne przerwania

sei();

while(1)
{
    PORTD^=0x20;           //generator
    czekaj(10000L);
}

```

14.2.4. Ćwiczenie 4

„Hello World!”, czyli sterowanie wyświetlaczem alfanumerycznym LCD 16×2 i 16×1. Obsługa pojedynczego przycisku

Można zaryzykować twierdzenie, że umiejętność sterowania wyświetlaczem alfanumerycznym LCD jest jedną z podstawowych dla każdego, kto planuje konstruowanie własnych urządzeń. Swą popularność elementy te zawdzięczają małym wymiarom, niewygórowanej cenie oraz możliwości wyświetlenia znaków alfanumerycznych (w tym 8 definiowanych przez użytkownika). Ostatnia cecha staje się bardzo przydatna, gdyż w handlu nie występują wersje z polskimi znakami diakrytycznymi. Podstawowy interfejs wyświetlacza jest dość rozbudowany, gdyż wymaga zastosowania 8 linii danych (dwukierunkowych), 3 linii sterujących, 2 zasilających, a także końcówki, do której dołącza się potencjometr regulacji kontrastu (rozміщення wyprowadzeń typowych wyświetlaczy przedstawiono w dodatku E).

Najczęściej jednak wyświetlacz jest dołączany do systemu mikroprocesorowego poprzez uproszczoną wersję interfejsu, w której magistrala danych jest zredukowana do 4 linii. Ponadto można zrezygnować z odczytywania statusu wyświetlacza i przyjmować, że po określonym czasie od przesłania instrukcji jest on gotowy do dalszych operacji. Linię R/\overline{W} należy wówczas na stałe dołączyć do masy. Taki wariant zastosowano również na płycie ZL1AVR. Nieznacznie bardziej złożona obsługa programowa jest zrekomensowana pozostawieniem czterech drogocennych wyprowadzeń portów mikrokontrolera do innych celów. Wyświetlacze alfanumeryczne są dostępne w wielu „rozmiarach”. Najpopularniejszy jest chyba wariant z 2 liniami po 16 znaków, który skutecznie wypiera jednolinijkową wersję z 16 znakami. Jeśli konieczne jest jednocześnie wyświetlenie większej ilości danych, to można sięgnąć po wyświetlacze z większą liczbą wierszy i znaków np. 4×10, 4×16, 2×20, 4×20,

2×24, 2×40, 4×40. Producenci z wielu krajów starają się zachować kompatybilność swoich wyrobów ze standardem, jakim stał się sterownik HD44780 (Hitachi). Sztuka ta udaje się z mniejszym lub większym powodzeniem dokonać. Bywają natomiast problemy z różnorodną obsługą wyświetlaczy jednoliniowych, wynikającą z przyjętej organizacji danych. Można spotkać na przykład wersje 1-liniowe emulujące wyświetlacze 2-liniowe. Taka właśnie wersja będzie opisana w dalszej części rozdziału. Trzeba jednak pamiętać, że przedstawione procedury mogą nie działać prawidłowo z pewnymi typami wyświetlaczy. W poniższych opisach ograniczymy się tylko do 4-bitowego trybu pracy interfejsu. W takim przypadku, dane są przesyłane do/z wyświetlacza poprzez linie DB7, DB6, DB5 i DB4, które są dołączane na płytce ZL1AVR odpowiednio do PB7, PB6, PB5, PB4. Oczywiście podane przypisanie wyprowadzeń jest tylko przykładowe.



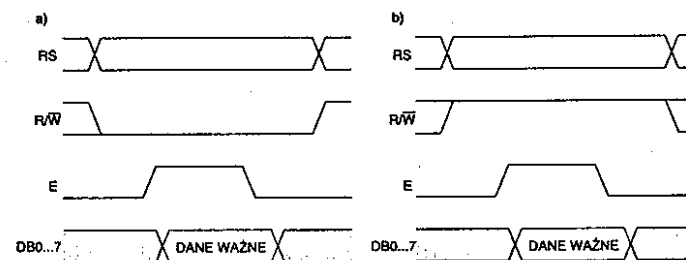
Rozmieszczenie wyprowadzeń typowych wyświetlaczy LCD 1×16 i 2×16 przedstawiono w dodatku E.

W zależności od potrzeb, wyświetlacz może być dołączany do innych portów mikrokontrolera. Niezbędne będzie wtedy odpowiednie zmodyfikowanie procedur obsługi. W przypadku interfejsu 4-bitowego, linie DB0 do DB3 szyny danych należy pozostawić nie podłączone. Informacje przekazywane do wyświetlacza można podzielić na instrukcje sterujące oraz dane do wyświetlenia lub zapisania w pamięci modułu. W drugą stronę można odbierać informacje o stanie sterownika (jego gotowości do wykonania kolejnych poleceń) lub odczytywać wewnętrzną pamięć modułu. Wyboru rodzaju informacji dokonuje się poprzez odpowiednie ustawienie linii RS, kierunkiem przepływu steruje zaś linia R/W (tablica 14.1). Dane są strobowane sygnałem podawanym na linię E (rysunek 14.11).

Informacje o kształcie wyświetlanych znaków sterownik czerpie z tzw. generatora znaków – CG-ROM (Character Generator ROM). Jest to pamięć typu ROM (Read Only Memory) programowana na etapie produkcji – jej zawartość można tylko odczytywać. Każdy znak składa się z matrycy 5×7 lub 5×10

Tab. 14.1. Możliwe stany linii sterujących RS i R/W

R/W	RS	Operacja
0	0	Zapis instrukcji lub adresu do LCD
0	1	Zapis kodu znaku do wyświetlenia lub kodu definiowanego znaku (zapis w pamięci CGRAM)
1	0	Odczyt wskaźnika zajętości wyświetlacza i bieżącego adresu wyświetlanego znaku
1	1	Odczyt pamięci DDRAM lub CGRAM



Rys. 14.11. Przebiegi czasowe interfejsu wyświetlacza LCD: podczas zapisu (a) i odczytu (b) danych do/ze sterownika

punktów. Znaki są wybierane poprzez podanie odpowiedniego kodu (w dużej części zgodnego z ASCII). Umieszczenie znaku na ekranie wyświetlacza polega na wpisaniu jego kodu pod odpowiednim adresem pamięci DDRAM (Display Data RAM).



Tablice ASCII oraz znaki wpisane do pamięci CG-ROM typowych sterowników alfanumerycznych wyświetlaczy LCD przedstawiono w dodatku I.

Sterownik wyposażono w licznik, który może być automatycznie inkrementowany lub dekrementowany po przesłaniu kodu znaku do wyświetlenia. Dzięki temu nie trzeba określać jego adresu za każdym razem. Jest to bardzo przydatna cecha, gdyż na ogół mamy do czynienia z sekwencyjnym zapisem danych do wyświetlacza. Rozszerzeniem pamięci CG-ROM jest 64-bajtowa pamięć CG-RAM. Można w niej zdefiniować 8 własnych znaków w matrycy 5×7, a ich kody są równe 0 do 7.

Z prawidłowym oprogramowaniem wyświetlacza alfanumerycznego związane są dwa zagadnienia. Jest to prawidłowe zainicjowanie sterownika po wyzerowaniu systemu mikroprocesorowego oraz odpowiednie dla zastosowanej w nim organizacji przesyłanie danych. Przykładowe adresowanie znaków

		Wyświetlacz 16×1															
Linia 1		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
		00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F

		Wyświetlacz 16×2															
Linia 1		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
		00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
Linia 2		17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
		40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F

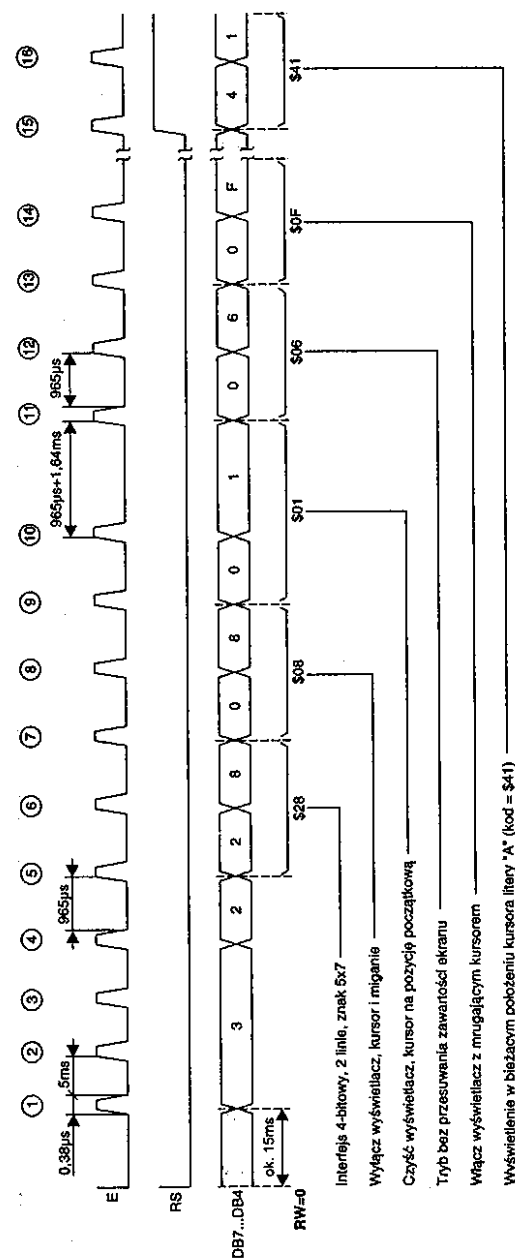
Rys. 14.12. Adresowanie znaków w wyświetlaczach alfanumerycznych LCD

w wyświetlaczach 16×1 i 16×2 przedstawiono na rysunku 14.12. Pokazana jest tu wersja wyświetlacza 16×1, który emuluje wyświetlacz 8×2. Jak widać występuje pewna niedogodność polegająca na przeskoku adresów na pozycji 9. Wyświetlacz taki powinien być inicjowany jako 2-wierszowy. Sterownik HD44780 zezwala na określenie liczby wierszy (1 lub 2). Można się spotkać również z wyświetlaczami 16×1, które powinny być konfigurowane jako 1-linijkowe. Adresowanie znaków jest w nich ciągłe, tzn. od \$00 do \$0F.

Tab. 14.2. Zestawienie instrukcji sterownika HD44780

Instrukcja	RS	RW	Kod								Opis	Czas wykonania	
			DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0			
Wyczyść wyświetlacz	0	0	0	0	0	0	0	0	0	1	Wyczyszczenie ekranu i przesunięcie kursora do pozycji początkowej (adres 0)	1,64 ms	
Przesuń kursor do pozycji początkowej	0	0	0	0	0	0	0	0	0	1	*	Przesunięcie kursora do pozycji początkowej (adres 0). Pamięć DDRAM pozostaje niezmieniona.	1,64 ms
Ustaw tryb wprowadzania znaków	0	0	0	0	0	0	0	0	1	I/D	S	Ustawienie kierunku i trybu przesuwania kursora w trakcie zapisu lub odczytu danych: I/D=0 -> zmniejszanie I/D=1 -> zwiększanie S=0 -> w prawo	40 μs
Ustaw tryb wyświetlacza włączenie/wyłączenie	0	0	0	0	0	0	0	1	D	C	B	Ustawienie trybu: D=0 -> wyłącz LCD D=1 -> włącz LCD C=0 -> wyłącz kursor C=1 -> włącz kursor B=0 -> wyłącz miganie kursora B=1 -> włącz miganie kursora	40 μs
Przesuń kursor/wskazanie	0	0	0	0	0	0	1	S/C	R/L	*	*	Przesunięcie kursora i wyświetlanego tekstu o jedną pozycję: S/C=0 -> przesuwanie kursora S/C=1 -> przesuwanie wyświetlacza R/L=0 -> w lewo R/L=1 -> w prawo	40 μs
Ustaw funkcję	0	0	0	0	1	DL	N	F	*	*	*	Konfigurowanie wyświetlacza: DL=0 -> interfejs 4-bitowy DL=1 -> interfejs 8-bitowy N=0 -> 1 linia N=1 -> 2 linie F=0 -> matryca 5×7 F=1 -> matryca 5×10	40 μs
Ustaw adres pamięci CGRAM	0	0	0	1	ACG – adres CGRAM						Ustawienie adresu CGRAM dla zapisu i odczytu tej pamięci	40 μs	
Ustaw adres pamięci DDRAM	0	0	1	ADD – adres DDRAM						Ustawienie adresu DDRAM dla zapisu i odczytu tej pamięci	40 μs		
Odczytaj flagę zajętości/adres	0	1	BF	AC – adres DDRAM i CGRAM						Czytanie flagi zajętości sterownika BF oraz licznika adresów dla pamięci DDRAM i CGRAM)		40 μs	
Zapisz dane do pamięci CGRAM/DDRAM	1	0	Zapisywana dana								Zapisanie danych do pamięci CGRAM lub DDRAM	40 μs	
Odczytaj dane z pamięci CGRAM/DDRAM	1	1	Zapisywana dana								Odczytanie danych z pamięci CGRAM lub DDRAM	40 μs	

* - wartość bez znaczenia



Rys. 14.13. Przykładowa sekwencja instrukcji inicjująca pracę sterownika wyświetlacza

W tabeli 14.2 zestawiono instrukcje sterujące dla alfanumerycznego wyświetlacza LCD. Podano w niej binarne kody rozkazów wraz z odpowiednimi dla nich stanami linii sterujących oraz czasy wykonania rozkazów dla sterowników taktowanych zegarem 250 kHz (typowo). Czas wykonania rozkazu jest szczególnie istotny dla aplikacji, w których nie sprawdza się wskaźnika gotowości sterownika.

Możliwość pracy interfejsu wyświetlacza w trybie 4- lub 8-bitowym stwarza pewne problemy z jego zainicjowaniem. Dane przekazywane do sterownika będą przecież przez niego interpretowane inaczej dla szyny 4-bitowej, inaczej dla 8-bitowej. Wyświetlacze są na ogół wyposażone w układ zerujący, który po włączeniu zasilania ustawia sterownik w trybie 8-bitowym z jedną linią i matrycą 5×7, ale po sprzętowym zerowaniu systemu stan sterownika wyświetlacza może być dla mikrokontrolera nieokreślony. Układ zerujący wymaga ponadto odpowiedniej szybkości narastania napięcia zasilającego. Jeśli warunek ten nie będzie spełniony, stan sterownika może być również nieokreślony. Z tego względu do zainicjowania pracy zalecane jest wykonanie specjalnej sekwencji instrukcji (rysunek 14.13), gwarantującej prawidłowe skonfigurowanie wyświetlacza, bez względu na jego początkowy stan. Skala czasu na wykresie nie jest jednolita.

Po włączeniu zasilania niezbędne jest odczekanie minimum 15 ms na wykonanie się procedury zerującej sterownik LCD. Następnie należy 3-krotnie

przesłać do wyświetlacza instrukcję \$3x, po czym instrukcję \$2x (takty 1, 2, 3 i 4 na **rysunku 14.13**). Zapis \$2x i \$3x oznacza, że młodsza część bajtu jest nieistotna. Kolejna instrukcja \$28 zagwarantuje ustawienie wyświetlacza w trybie 4-bitowym z dwoma liniami. Należy pamiętać, że zapis \$28 oznacza przesłanie do wyświetlacza kodu instrukcji w dwóch taktach \$2x i \$8x (**rysunek 14.13**). Założenie to będzie obowiązywało również w dalszej części opisu. Z dokumentacji wyświetlaczy wynika, że po pierwszym takcie (przed rozpoczęciem drugiego) należy odczekać minimum 4,1 ms, a po drugim ok. 100 µs. W programie z **listingu 14.4** dla uproszczenia przyjęto, że wszystkie opóźnienia są równe ok. 5 ms. Kolejnymi instrukcjami zalecanymi w procedurze inicjującej są \$08, \$01, \$06 i \$0F, przy czym ostatnie dwie zależą od tego, jak ma być skonfigurowany wyświetlacz (można je odpowiednio modyfikować). Trzeba zwrócić uwagę na dość długi czas wykonywania się instrukcji czyszczenia ekranu (1,64 ms). W programie uwzględniono pętlę opóźniającą po każdym takcie, co jest związane ze zrezygnowaniem z odczytu flagi gotowości (BF) sterownika. Gdyby jednak program i połączenia interfejsu były do tego przystosowane, to odczyt gotowości wyświetlacza jest możliwy dopiero po 13 taktach (**rysunek 14.13**).

Do ćwiczenia 4 dołączono dwa oddzielne programy: wersja „a” – dla wyświetlaczy 16×2 i wersja „b” – dla wyświetlaczy 16×1. Niektóre procedury są wspólne. Na początku zdefiniowano numery bitów, do których dołączono linie RS i E. Zmienne wiersz i kolumna ułatwiają umieszczanie znaków w dowolnym miejscu na wyświetlaczu. Funkcja czekaj jest wykorzystywana do odmierzania czasu. Stała tau pozwala w przybliżeniu przeliczać argument funkcji na czas w milisekundach. Dokładność odmierzania czasu nie jest tu zbyt duża, ale wystarczająca. Do przesłania instrukcji dla sterownika wyświetlacza służy funkcja `pisziled(instr)`, której argumentem jest kod instrukcji. W wywołaniu podaje się go w postaci bajtu, a procedura realizuje 2-taktowe przesłanie go do wyświetlacza. Podobnie działa funkcja `piszdlcd(dana)`, która przesyła sterownikowi daną do wyświetlenia. W tym przypadku dana to kod znaku, którego kształt jest zapisany w pamięci CG-ROM lub CG-RAM. W programie uwzględniono również funkcję czyszczenia ekranu – `czysc_lcd()` i ustawiania kursora w określonych współrzędnych ekranowych – `lcdxy(wiersz, kolumna)`. Za bezpośrednie umieszczenie znaku na ekranie odpowiada funkcja `piszznak(znak)`. Wywołuje ona omówioną wcześniej `piszdlcd(dana)` i odpowiednio modyfikuje zmienne wiersz i kolumna. W praktyce będziemy mieli najczęściej do czynienia z wyświetlaniem całych łańcuchów znakowych, nie pojedynczych znaków. Łańcuchy takie (*string*) są deklarowane poprzez statyczne wskaźniki (*pointery*) `*tekst1`, `*tekst2` itd. wskazujące na dane umieszczone w pamięci programu. Ich za-

stosowanie zapobiega umieszczaniu łańcuchów znakowych przez kompilator w pamięci RAM, co przy niezbyt dużej wielkości tej pamięci mogłoby prowadzić do kłopotliwych sytuacji w pewnych aplikacjach. Opisana metoda jest możliwa tylko dla tych mikrokontrolerów, które mają zaimplementowane rozkazy LPM lub ELPM. Choć zasadniczą część programu stanowią funkcje obsługi wyświetlacza, to uwzględniono w nim również krótką funkcję sprawdzającą naciśnięcie określonego klawisza – `klawisz(pozkl)`. Argument `pozkl` określa bit portu D, który będzie badany. Zauważmy, że zastosowana metoda jest w tym przypadku do przyjęcia, lecz nie zawsze będzie ona odpowiednia. Program zatrzymuje się bowiem w pętli oczekującej na naciśnięcie klawisza, zawieszając ewentualne inne zadania. Po wykryciu tego faktu wywołany jest podprogram odmierzający opóźnienie dla eliminacji drgań styków, po czym oczekuje się zwolnienia klawisza. Wykorzystywany jest klawisz SW4. Programy pokazane na **listinach 14.4** i **14.5** demonstrują kilka efektów uzyskiwanych na wyświetlaczu LCD odpowiednio 16×2 i 16×1.

Wybrany program (w przypadku wyświetlacza 16×2 – *cwicz4a.c*, w przypadku wyświetlacza 16×1 – *cwicz4b.c*) należy skompilować i następnie zaprogramować mikrokontroler z zestawu ZL1AVR odpowiednim plikiem wynikowym (*cwicz4a.hex* lub *cwicz4b.hex*).

Konfiguracja płytki ZL1AVR:

- zworka J3 w pozycji 1-2, zworka J4 w pozycji 2-3 – dołączony oscylator wewnętrzny,
- zworka JP1 rozwarta – globalne wyłączenie diod LED,
- zworka JP4 w pozycji 2-3 (włączenie klawiszy SW1 i SW4),
- zworka JP7 zwarta (włączenie klawisza SW4),
- zworka JP6 zwarta (włączenie klawisza SW1),
- zworki ZW_PORTB zwarte w pozycjach 1-2, 3-4, 5-6, 7-8, 9-10, 11-12; położenie zworek 13-14 i 15-16 jest nieistotne,
- położenie pozostałych zworek nieistotne (np. rozłączone),
- do gniazda LCD1 należy dołączyć wyświetlacz alfanumeryczny 16×2 (dla programu *cwicz4a.c*) lub 16×1 (dla programu *cwicz4b.c*).

List. 14.4. Program do ćwiczenia 4a

```

/*****
/* Ćwiczenie 4a - sterowanie alfanumerycznym wyświetlaczem LCD */
/*                               16x2 (16 znaków, 2 wiersze) */
/*                               - obsługa pojedynczego klawisza */
/* J.D. '2003 */
*****/
#include <io.h>
#include <progmem.h>

```

```
#include <stdlib.h>

#define lcd_rs 2          //definicja bitu portu dla linii RS
#define lcd_e 3           //definicja bitu portu dla linii E
#define CR 0x0a          //definicja znaku CR (przejście do nowej
                          //linii)

unsigned char wiersz=0;
unsigned char kolumna=0;

void czekaj(unsigned long pt) //funkcja opóźnienia
{
#define tau 10.38          //przybliżony przelicznik argumentu na ms
    unsigned char tpl;

    for(;pt>0;pt--)
    {
        for(tpl=255;tpl!=0;tpl--);
    }
}

void piszlcd(unsigned char instr)
{
    //zapisz instrukcję sterującą do LCD
    cbi(PORTB,lcd_rs);
    sbi(PORTB,lcd_e);
    PORTB=(PORTB&0x0f)|((instr&0xf0);

    //przygotuj starszy półbajt do LCD
    //wymagane wydłużenie impulsu
    asm("nop");
    asm("nop");
    asm("nop");
    cbi(PORTB,lcd_e);          //impuls strobujący
    czekaj(10L);              //czekaj na gotowość LCD ok. 100us
    sbi(PORTB,lcd_e);
    PORTB=(PORTB&0x0f)|((instr&0x0f)<<4);
    //przygotuj młodszy półbajt do LCD

    asm("nop");
    asm("nop");
    asm("nop");
    cbi(PORTB,lcd_e);          //impuls strobujący
    czekaj(10L);              //czekaj na gotowość LCD ok. 100us
}

void piszldcd(char dana)    //zapisz daną do LCD
{
    sbi(PORTB,lcd_rs);
    sbi(PORTB,lcd_e);
    PORTB=(PORTB&0x0f)|((dana&0xf0);
    //przygotuj starszy półbajt do LCD
    //wymagane wydłużenie impulsu
    asm("nop");
    asm("nop");
    asm("nop");
    cbi(PORTB,lcd_e);          //impuls strobujący
    czekaj(10L);              //czekaj na gotowość LCD
    sbi(PORTB,lcd_e);
    PORTB=(PORTB&0x0f)|((dana&0x0f)<<4);
    //przygotuj młodszy półbajt do LCD

    asm("nop");
    asm("nop");
    asm("nop");
    cbi(PORTB,lcd_e);          //impuls strobujący
    czekaj(10L);              //czekaj na gotowość LCD
}
```

```

}

void czysclcd(void)         //czyść ekran
{
    piszldcd(0x01);         //polecenie czyszczenia ekranu dla
                            //kontrolera LCD
    czekaj(1.64*tau);       //rozkaz 0x01 wykonuje się 1.64ms
    wiersz=0;
    kolumna=0;
}

void piszznak(char znak)    //procedura umieszcza znak na
                            //wyświetlaczu
{
    piszldcd(znak);         //wyświetl znak na LCD
    if(++kolumna==16)       //czy bieżąca kolumna mieści się na
                            //wyświetlaczu?
    {
        kolumna=0;         //jeśli nie, to ustaw początkową...
        if(++wiersz==2)    //i przejdź do nowego wiersza
        {
            wiersz=0;       //jeśli nowy wiersz jest poza
                            //wyświetlaczem, ustaw początkowy
        }
    }
}

void lcdxy(unsigned char w, unsigned char k)
//ustaw współrzędne kursora
{
    piszldcd((w*0x40+k)|0x80); //standardowy rozkaz sterownika LCD
    //ustawiający kursor w określonych
    //współrzędnych
}

void pisztekst(char *tekst) //pisz tekst na LCD wskazywany przez
//tekst
{
    char zn;
    char nr=0;

    while(1)
    {
        zn=PRG_RDB(&tekst[nr++]); //pobranie znaku z pamięci programu
        if(zn!=0)                  //czy nie ma końca tekstu?
        {
            if(zn==CR)             //czy znak nowej linii
            {
                wiersz=1?wiersz=0:++wiersz;
                                //przejdź do nowej linii
                kolumna=0;
                lcdxy(wiersz,kolumna); //ustaw obowiązujące po zmianie
                                //współrzędne na LCD
            }
            else
            {
                piszldcd(zn);       //umieść pojedynczy znak tekstu na LCD
            }
        }
        else
        {
            break;                 //zakończ pętlę, jeśli koniec tekstu
        }
    }
}
```

[illegible][illegible]

List. 14.5. Program do ćwiczenia 4b

340

```
volatile unsigned char kolumna;

void czekaj(unsigned long pt) //funkcja opóźnienia
{
#define tau 10.38 //przybliżony przelicznik argumentu na ms
    unsigned char tpl;

    for(;pt>0;pt--)
    {
        for(tpl=255;tpl!=0;tpl--);
    }
}

void piszlclcd(unsigned char instr) //zapisz instrukcję sterującą do LCD
{
    cbi(PORTB,lcd_rs);
    sbi(PORTB,lcd_e);
    PORTB=(PORTB&0xf0)|(instr&0xf0); //przygotuj starszy półbajt do LCD
    asm("nop"); //wymagane wydłużenie impulsu
    asm("nop");
    asm("nop");
    cbi(PORTB,lcd_e); //impuls strobuujący
    czekaj(10L); //czekaj na gotowość LCD ok. 100us
    sbi(PORTB,lcd_e);
    PORTB=(PORTB&0xf0)|((instr&0xf0)<<4); //przygotuj młodszy półbajt do LCD
    asm("nop");
    asm("nop");
    asm("nop");
    cbi(PORTB,lcd_e); //impuls strobuujący
    czekaj(10L); //czekaj na gotowość LCD ok. 100us
}

void pisdclcd(char dana) //zapisz daną do LCD
{
    sbi(PORTB,lcd_rs);
    sbi(PORTB,lcd_e);
    PORTB=(PORTB&0xf0)|(dana&0xf0); //przygotuj starszy półbajt do LCD
    asm("nop"); //wymagane wydłużenie impulsu
    asm("nop");
    asm("nop");
    cbi(PORTB,lcd_e); //impuls strobuujący
    czekaj(10L); //czekaj na gotowość LCD
    sbi(PORTB,lcd_e);
    PORTB=(PORTB&0xf0)|((dana&0xf0)<<4); //przygotuj młodszy półbajt do LCD
    asm("nop");
    asm("nop");
    asm("nop");
    cbi(PORTB,lcd_e); //impuls strobuujący
    czekaj(10L); //czekaj na gotowość LCD
}

void czyszcclcd(void) //czyść ekran
{
    piszlclcd(0x01); //polecenie czyszczenia ekranu dla
    //kontrolera LCD
    czekaj(1.64*tau); //rozkaz 0x01 wykonuje się 1.64ms
    kolumna=0;
}
```

```

void piszznak(char znak) //procedura umieszcza znak na
                          //wyświetlaczu
{
    unsigned char p;
    if(kolumna<8)
    {
        p=kolumna;
    }
    else
    {
        p=kolumna+56; //dodatkowe przesunięcie współrzędnych
                      //dla LCD 1x16
                      //dla drugiej połówki wyświetlacza
    }
    piszilcd(p|0x80);
    kolumna=kolumna!=15?++kolumna:0; //ustaw następną kolumnę
    piszdlcd(znak); //wyświetl znak na LCD
}

void lcdy(unsigned char k) //ustaw współrzędne kursora
{
    if(k<8)
    {
        piszilcd(k|0x80); //standardowy rozkaz sterownika LCD
                          //ustawiający kursor w określonych
                          //współrzędnych
    }
    else
    {
        piszilcd((k+56)|0x80); //dodatkowe przesunięcie współrzędnych
                                //dla LCD 1x16
    }
    kolumna=k;
}

void pisztekst(char *tekst) //pisz tekst na LCD wskazywany przez
                             //**tekst
{
    char nr=0;
    char zn;

    while(1)
    {
        zn=PRG_RDB(&tekst[nr++]); //pobranie znaku z pamięci programu
        if(zn!=0) //czy nie ma końca tekstu?
        {
            if(zn==CR) //czy znak nowej linii
            {
                kolumna=0; //jeśli tak, ustaw obowiązujące po
                           //zmianie współrzędne na LCD
            }
            else
            {
                piszznak(zn); //umieść pojedynczy znak tekstu na LCD
            }
        }
        else
        {
            break; //zakończ pętlę, jeśli koniec tekstu
        }
    }
}

```

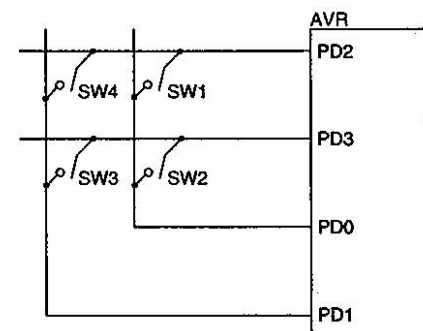
[illegible]

[illegible]

14.2.5. Ćwiczenie 5

„Łapanie muchy”, czyli obsługa klawiatury matrycowej z wykorzystaniem przerwania timera, obsługa wyświetlacza alfanumerycznego LCD 16×2

W poprzednich przykładach pokazano obsługę pojedynczych klawiszy, dołączanych bezpośrednio do portów mikrokontrolera. Rozwiązania takie stosuje się w aplikacjach, w których nie występuje duża liczba klawiszy. Jeśli będzie ich więcej niż 4, warto rozważyć możliwość połączenia ich w matrycę. Dla dużo większych klawiatur jest to właściwie jedyna sensowna metoda. Na **rysunku 14.14** pokazano przykładowy sposób dołączenia klawiatury 4-przyciskowej do mikrokontrolera AVR. Jest ona zorganizowana w układzie matrycy 2x2 (2 wiersze, 2 kolumny). Jak widać nie uzyskuje się tu właściwie żadnej



Rys. 14.14. Przykładowy sposób dołączenia klawiatury 4-przyciskowej do mikrokontrolera AVR

w sposób losowy (pseudolosowy) po ekranie wyświetlacza LCD. Zadaniem zawodnika jest podążanie za ruchem muchy za pomocą czterech klawiszy. Ich znaczenie jest następujące: SW3 – ruch w lewo, SW4 – ruch w prawo, SW2 – ruch na dół, SW1 – ruch do góry. W zależności od tego, które przyciski zostały naciśnięte, odpowiednio jest przemieszczany znak „plus” po ekranie.

Konfiguracja klawiatury nie uwalnia nas od problemów związanych z zakłóceniami wywoływanymi drganiem styków. W tym przypadku również należy podejmować odpowiednie środki do ich eliminacji. Najodpowiedniejszym rozwiązaniem jest cykliczne skanowanie stanu klawiatury z programową eliminacją zakłóceń. Momenty próbkowania najwygodniej jest wyznaczać wykorzystując timer mikrokontrolera sygnalizujący przepełnienie (a więc mierzenie określonego czasu), odpowiednim przerwaniem. W tym konkretnym przypadku będzie to Timer0. Linie portów, do których są dołączone kolumny klawiatury (PD1 i PD0) powinny być ustawione jako wyjściowe, natomiast wiersze są dołączone do portów PD3 i PD2 pracujących jako wejścia z podciąganiem. Sprawdzenie stanu klawiatury może przebiegać według następującego algorytmu:

1. Ustaw stan niski na wyjściu PD0.
2. Sprawdź stan wejścia PD2. Jeśli jest „1”, to klawisz SW1 nie jest naciśnięty. Stan wysoki jest wymuszany przez wewnętrzne podciąganie do plusa linii PD2. Na płycie ZL1AVR przewidziano ponadto rezystor R6, który będzie pełnił funkcję *pull-up* w przypadku, gdy podciąganie nie zostanie włączone (do celów doświadczalnych). Analogicznie działa R7 dla linii PD3. Naciśnięcie klawisza SW1 spowoduje podanie stanu „0” z wyjścia PD0 na wejście PD2. Odczytanie więc stanu niskiego na wejściu PD2 oznacza, że naciśnięto klawisz SW1.

oszczędności portów wejścia-wyjścia. Przy tak niewielkiej klawiaturze liczba zajętych linii jest równa 4 bez względu na konfigurację styków. Przykład omawiamy jedynie w celu pokazania zasady obsługi klawiatury matrycowej.

W programie do ćwiczenia 5 klawiatura zostanie wykorzystana w prostej grze zręcznościowej „Łapanie muchy”. „Muchą” jest znak kropki poruszający się

3. Sprawdź w analogiczny sposób pozostałe wejścia wierszy. W naszym przypadku jest to tylko linia PD3. Jeśli na odczytywanym wejściu będzie stan „1”, to klawisz leżący na przecięciu aktywnej kolumny i sprawdzanego wiersza jest zwolniony. W przeciwnym razie jest naciśnięty.
4. Powtórz analogiczną sekwencję jak w punktach 1 do 3 dla pozostałych wyjść kolumnowych (w naszym przypadku jest to tylko linia PD1).

Zauważmy, że liczba binarna odpowiadająca stanowi wszystkich linii obsługujących klawiaturę może być traktowana jako naturalny kod znaku. Na przykład naciśnięcie klawisza SW4 spowoduje zwrócenie kodu 1001 (PD3=1, PD2=0, PD1=0, PD0=1). Można także przyjąć dowolną inną metodę kodowania, lecz w tym przypadku niezbędne będą dodatkowe działania wykonane przez mikrokontroler. Przepatrywanie klawiatury odbywa się w procedurze obsługi przerwania. Akcja, jaka ma być podjęta po naciśnięciu przycisku, może czasami wymagać dużego obciążenia mikrokontrolera. Wykonanie jej z wnętrza procedury przerwania może doprowadzić do chwilowego zawieszenia systemu. Na przykład przerwanie o niższym priorytecie nie będzie się mogło doczekać swojej obsługi. Aby uniknąć takich sytuacji można zastosować specjalną flagę, dzięki której procedura obsługi będzie sygnalizować programowi głównemu naciśnięcie klawisza i konieczność wykonania związanego z tym zadania. W omawianym programie zagrożenia takiego nie ma, lecz technika taka zostanie zastosowana. Wskaźnikiem jest zmienna fklawisz, ustawiana w procedurze obsługi przerwania od Timera0 po wykryciu naciśnięcia dowolnego klawisza. Sprawdzenie, czy jest wciśnięty jakikolwiek klawisz dla aktywnej kolumny jest realizowane za pomocą operacji Ex-OR (patrz komentarze na listingu 14.6). Jeśli w głównej pętli programu okaże się, że flaga ta jest ustawiona, to wykonywana jest sekwencja instrukcji powodująca wyświetlenie znaku „+” w nowym położeniu zależnym od kodu naciśniętego klawisza. Do obsługi wyświetlacza będą wykorzystane funkcje znane z poprzedniego ćwiczenia. Na potrzeby gry nieznaczącej zmianie uległa tylko funkcja piszznak(znak). Do obliczania nowych współrzędnych ekranowych znaku „+” wykorzystane są funkcje incwiersz(w), incokolumna(k), decwiersz(w), deckolumna(k). Są one również wykorzystywane do obliczania pozycji ekranowej „muchy”. Kierunek przesunięcia jest losowany za pomocą funkcji standardowej rand(), która nadaje zmiennej r wartość losową z przedziału 0 do 255. Cały zakres podzielono na 8 przedziałów odpowiadających kierunkom: w górę, w prawo w skos do góry, w prawo itd. Przedziały ze względu na asymetrię geometryczną wyświetlacza nie są sobie równe. Kierunek ruchu „muchy” zależy od tego, w jakim przedziale znajduje się wylosowana liczba.

Program realizujący przedstawione zadanie znajduje się na listingu 14.6. Należy skompilować program *cwicz5.c* i zaprogramować mikrokontroler z zestawu ZL1AVR plikiem wynikowym *cwicz5.hex*.

Konfiguracja płytki ZL1AVR:

- zworka J3 w pozycji 1-2, zworka J4 w pozycji 2-3 – dołączony oscylator wewnętrzny,
- zworka JP1 rozwarta – globalne wyłączenie diod LED,
- zworka JP4 i JP5 w pozycji 1-2 (dołączenie klawiszy w układzie matrycy do mikrokontrolera),
- zworka JP6 i JP7 zwarta (dołączenie klawiszy do mikrokontrolera),
- zworki ZW_PORTB zwarte w pozycjach 1-2, 3-4, 5-6, 7-8, 9-10, 11-12; zworki 13-14 i 15-16 nieistotne,
- położenie pozostałych zworek nieistotne (np. rozłączone)
- podłączony do gniazda LCD1 wyświetlacz alfanumeryczny 16×2.

List. 14.6. Program do ćwiczenia 5

```

/*****
/* Ćwiczenie 5 - Obsługa klawiatury matrycowej z wykorzystaniem */
/*          przerwań timera0                                     */
/*          - wykorzystanie wyświetlacza LCD 16*2                */
/*          - prosta gra zrecznosciowa                           */
/* J.D. '2003                                                    */
*****/

#include <io.h>
#include <interrupt.h>
#include <signal.h>

#define lcd_rs 2          //definicja bitu portu dla linii RS
#define lcd_e 3           //definicja bitu portu dla linii E
#define tau0 247          //stała czasowa timera0
#define vliczt0 113       //stała wpisywana do licznika wejść do
                          //do przerwania timera0
#define vlkursor 10       //wartość wpisywana do zmiennej lkursor

/***** zmienna globalne *****/
unsigned char liczt0;      //Licznik wejść do przerwania timera0.
                          //Klawisz jest badany, gdy liczt0=0
unsigned char lkursor;    //licznik wejść do przerwania timera0
                          //dla ustawienia flagi zmiany
                          //położenia "muchy"
volatile unsigned char kodklaw; //kod naciśniętego klawisza;
volatile unsigned char fklaw;   //flaga wykrycia naciśnięcia klawisza;
volatile unsigned char fkursor; //flaga zmiany położenia "muchy"
unsigned char wiersz=0;        //pozycja umieszczenia znaku na LCD
unsigned char kolumna=0;       //pozycja umieszczenia znaku na LCD

void czekaj(unsigned long zt) //funkcja opóźnienia
{
#define tau 10.38           //przybliżony przelicznik argumentu na ms
    unsigned char zt1;

```

```

for(;zt>0;zt--)
{
    for(ztl=255;ztl!=0;ztl--);
}

SIGNAL (SIG_OVERFLOW0)      //obsługa przerwania od przepełnienia
                             //timera0
{
    unsigned char kkolumna,kwiersz;
    TCNT0=tau0;              //zmienne pomocnicze
                             //odśwież stałą czasową w TCNT0

    if(--liczt0==0)          //czy już czytać klawisze?
    {
        //tak
        for(kkolumna=0xfe;kkolumna!=0xfb;kkolumna=(kkolumna<<1)+1)
        {
            PORTD=(PORTD|0x03)&(kkolumna);
            //podaj "0" na linię sterującą aktywną
            //kolumną
            kwiersz=PIND&0x0f;
            //czytaj wiersz, pozostaw tylko linie
            //obsługujące klawiaturę
            if((kwiersz&0x0c)^0x0c) //czy jest odpowiedź na jakiejś linii
            //wierszy?
            {
                //tak, ustaw flagę gotowości klawiatury
                fklaw=1;
                break;
            }
            //i zakończ przepatrywanie
        }
    }
    if(fklaw)
    {
        kodklaw=kwiersz;
        //jeśli wykryto wciśnięcie klawisza, podaj
        //jego kod
    }
    else
    {
        kodklaw=0xff;
        //jeśli nie, ustaw kod neutralny
    }
    liczt0=vliczt0;
    //odśwież stan liczt0
    if(--lkursor==0)
    //czy można ustawić flagę zmiany "muchy"?
    {
        fkursor=1;
        //tak (minęło ok. 300ms)
        lkursor=vlkursor;
        //odśwież stan lkursor
    }
}

void piszildc(unsigned char instr)
    //zapisz instrukcję sterującą do LCD
{
    cbi(PORTB,lcd_rs);
    sbi(PORTB,lcd_e);
    PORTB=(PORTB&0x0f)|((instr&0xf0);
    //przygotuj starszy półbajt do LCD

    asm("nop");
    asm("nop");
    asm("nop");
    cbi(PORTB,lcd_e);
    //impuls strobujący
    czekaj(10L);
    //czekaj na gotowość LCD ok. 100us
    sbi(PORTB,lcd_e);
    PORTB=(PORTB&0x0f)|((instr&0xf)<<4);
    //przygotuj młodszy półbajt do LCD

```

```

asm("nop");
asm("nop");
asm("nop");
cbi(PORTB,lcd_e);
//impuls strobujący
czekaj(10L);
//czekaj na gotowość LCD ok. 100us
}

void piszldc(char dana)      //zapisz daną do LCD
{
    sbi(PORTB,lcd_rs);
    sbi(PORTB,lcd_e);
    PORTB=(PORTB&0x0f)|((dana&0xf0);
    //przygotuj starszy półbajt do LCD

    asm("nop");
    asm("nop");
    asm("nop");
    cbi(PORTB,lcd_e);
    //impuls strobujący
    czekaj(10L);
    //czekaj na gotowość LCD
    sbi(PORTB,lcd_e);
    PORTB=(PORTB&0x0f)|((dana&0xf)<<4);
    //przygotuj młodszy półbajt do LCD

    asm("nop");
    asm("nop");
    asm("nop");
    cbi(PORTB,lcd_e);
    //impuls strobujący
    czekaj(10L);
    //czekaj na gotowość LCD
}

void czysclcd(void)          //czyść ekran
{
    piszildc(0x01);
    //polecenie czyszczenia ekranu dla
    //kontrolera LCD
    czekaj(1.64*tau);
    //rozkaz 0x01 wykonuje się 1.64ms
    wiersz=0;
    kolumna=0;
}

void lcdxy(unsigned char w, unsigned char k)
    //ustaw współrzędne kursora
{
    piszildc((w*0x40+k)&0x80);
    //standardowy rozkaz sterownika LCD
    //ustawiający kursor w określonych
    //współrzędnych
}

void piszznak(char znak)     //funkcja umieszcza znak na wyświetlaczu
{
    piszldc(znak);
    //wyświetl znak na LCD
}

unsigned char incwiersz(unsigned char w)
    //inkrementuj wiersz LCD
{
    return w==1?0:1;
}

unsigned char decwiersz(unsigned char w)
    //dekrementuj wiersz LCD
{
    return w==0?1:0;
}

```

[illegible]

```

piszznak('S'); //wyświetl napis "Start"
piszznak('t');
piszznak('a');
piszznak('r');
piszznak('t');

do
{
    i++; //zmienna "i" będzie wykorzystana później
        //do zainicjowania
        //generatora pseudolosowego
}while(!fklaw); //naciśnięcie klawisza zapewnia losowy
                //start generatora
srand(i); //inicjuj generator liczb pseudolosowych
czyścld(); //czyść ekran
piszznak('+'); //umieść znak gracza na ekranie

while(1) //główna pętla programu
{
    if(fkursor) //czy można przemieścić "muchę"?
    {
        lcdxy(wiersz,kolumna); //tak...
        if((wiersz==yg)&&(kolumna==xg))
        {
            piszznak('+'); //wymaż starą pozycję pozostawiając gracza
        }
        else
        {
            piszznak(' '); //wymaż starą pozycję
        }
        r=rand(); //losuj nowe położenie
        if((r<37)&&(r>18)) //wylicz nową pozycję
        {
            wiersz=incwiersz(wiersz);
        }
        if(r<19)
        {
            wiersz=decwiersz(wiersz);
        }
        if(((r>12)&&(r<19))||((r>30)&&(r<147)))
        {
            kolumna=incolumna(kolumna);
        }
        if((r<7)||((r>25)&&(r>18))||((r>146)))
        {
            kolumna=deckolumna(kolumna);
        }
        lcdxy(wiersz,kolumna); //ustaw nowe współrzędne
        piszznak('.'); //umieść "muchę" w nowym położeniu
        fkursor=0;
    }

    if(fklaw) //wykryto naciśnięcie klawisza
    {
        fklaw=0;
        lcdxy(yg,xg); //ustaw współrzędne znaku gracza...
        piszznak(' '); //... i wymaż go
        switch (kodklaw) //reakcja na klawisz
        {
            case 0x09: xg=incolumna(xg);
                        //SW4 - w prawo z zawijaniem
                        break;

```

```

case 0x05:  xg=deckolumna(xg);
            //SW3 - w lewo z zawijaniem
            break;
case 0x06:  yg=1;
            //SW2 - na dół bez zawijania
            break;
case 0x0a:  yg=0;
            //SW1 - do góry bez zawijania
}
lcdxy(yg,xg); //ustaw nowe współrzędne znaku gracza
if((wiersz==yg)&&(kolumna==xg))
{
    piszznak('*'); //znak, jeśli trafiony
    czekaj(200);   //przytrzymaj chwilę na ekranie
}
else
{
    piszznak('+'); //znak jeśli "pułko"
}
}
}

```

14.2.6. Ćwiczenie 6

6-bitowy, binarny wskaźnik napięcia. Zastosowanie komparatora analogowego do budowy przetwornika analogowo-cyfrowego. Wyzwalanie funkcji przechwytywania Timera1 za pomocą komparatora. Przerwanie od przechwytywania Timera1. Obsługa wewnętrznej pamięci EEPROM

Opis przetwornika analogowo-cyfrowego znajdującego się na płytce ZL1AVR zamieszczono w pierwszej części tego rozdziału. W ćwiczeniu 6. przetwornik ten wykorzystamy do budowy 6-bitowego, binarnego wskaźnika napięcia. Wynik pomiaru będzie wyświetlany na sześciu diodach LED w postaci liczby binarnej. Ograniczenie wskazania do 6-bitów wynika z konieczności przydzielenia portowi B dwóch zadań: sterowania diodami LED i realizacji przetwornika A/C. Ze względu na niewykorzystane porty np. PD1 i PD0, można pokusić się ewentualnie o użycie ich do sterowania pozostałymi dwoma diodami LED. Będzie to wymagało wykonania odpowiednich połączeń na płytce oraz zmian w oprogramowaniu. W danym ćwiczeniu wariantu takiego nie uwzględniono.

W początkowej części programu następuje inicjowanie systemu. Port D jest wykorzystywany do obsługi klawiszy SW1 i SW3 oraz sterowania tranzystorem Q3. Linie PD0 i PD1 są skonfigurowane jako wejścia z podciąganiem. Wyprowadzenia PB2...PB7 sterują LED-ami, dlatego są ustawione jako wyjścia. Linie PB0 i PB1 to wejścia komparatora analogowego – pracują jako

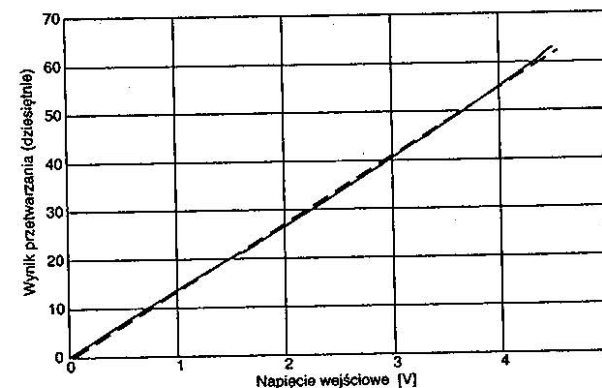
wejścia. Zastosowano podciąganie portu PB0, gdyż wpływało to korzystnie na pracę układu. Przy ustalaniu trybu pracy Timera1 zrezygnowano z preskalera, co ma na celu maksymalne skrócenie czasu konwersji. Wpis wartości \$41 do rejestru TCCR1B powoduje włączenie funkcji przechwytywania i start timera bez wstępnego podziału częstotliwości oscylatora. Ustawienie bitu ACIC w rejestrze ACSR zezwala na wyzwolenie funkcji przechwytywania Timera1 sygnałem z komparatora analogowego.

Główna pętla programu jest bardzo prosta – wynika z zasady działania przetwornika. Cykl pomiarowy rozpoczyna się od wyzerowania portu PD4, co jest równoznaczne z wyłączeniem tranzystora Q3. Płynący ze źródła Q1, Q2, R1 prąd rozpoczyna liniowe ładowanie kondensatora pomiarowego C5. Zaraz po wyłączeniu tranzystora Q3 są zerowane rejestry Timera1: TCNT1H i TCNT1L. Należy zwrócić uwagę na kolejność dostępu do tych rejestrów, wynikającą z używania przez CPU rejestru tymczasowego podczas operacji ich zapisywania (szczegóły w rozdziale 5.2 – opis rejestrów TCNT1H i TCNT1L). Wyzerowanie rejestrów jest jednoznaczne z rozpoczęciem pomiaru czasu. Timera nie trzeba startować, gdyż po zainicjowaniu pracuje on non-stop. Następnie jest zerowana flaga pomiar i w pętli while(pomiar==0) oczekuje się na zakończenie pomiaru. W chwili, gdy napięcie na kondensatorze C5 zrówna się z napięciem na gnieździe PORTD1-1 komparator analogowy mikrokontrolera zmienia stan, co powoduje wyzwolenie funkcji przechwytywania Timera1. Aktualny stan rejestrów TCNT1H i TCNT1L jest automatycznie przepisany do rejestrów ICR1H i ICR1L, po czym generowane jest przerwanie od przechwytywania Timera1. W czasie oczekiwania na to zdarzenie procesor mógłby wykonywać inne zadania, ale w tym konkretnym przypadku takich zadań nie ma, stąd pozostawanie w pętli while. Po zgłoszeniu przerwania, sterowanie jest przeniesione do procedury jego obsługi SIGNAL (SIG_INPUT_CAPTURE1), w której następuje odczyt rejestrów ICR1H i ICR1L, unormowanie wyniku do postaci 6-bitowej liczby binarnej i wyświetlenie rezultatu na diodach LED. Zapis PORTB=(PORTB&0x03)|czas8 oznacza, że najpierw są zerowane bity 2...7 portu B, a następnie są ustawiane jedynki na pozycjach wynikających z wartości zmiennej czas8, reprezentującej końcowy wynik pomiaru. Na zakończenie funkcji jest ustawiana flaga pomiar, po czym port PD4 jest przełączany w stan wysoki, powodując tym samym bardzo szybkie rozładowanie kondensatora przez niską oporność włączonego tranzystora Q3. Od tej chwili cały prąd źródła przepływa przez tranzystor Q3, a kondensator C5 może się naładować jedynie do bardzo małej wartości napięcia nasycenia U_{CES} .

Każdy „prawdziwy” przetwornik A/C ma albo wewnętrzne źródło referencyjne, albo wymaga dołączenia takiego źródła z zewnątrz. Jest ono potrzebne do odpowiedniego przetworzenia napięcia wejściowego na jego postać cyfrową. Przykładowo, jeśli 8-bitowy przetwornik będzie pracował z 2,5 V źródłem referencyjnym, to napięciu wejściowemu równemu 2,5 V będzie odpowiadał stan \$FF na wyjściu przetwornika. W przypadku przetwornika A/C opisywanego w ćwiczeniu wartością referencyjną jest czas ładowania kondensatora C5 do określonego napięcia. Chociaż czas ten można wyliczyć teoretycznie, to w praktyce – choćby ze względu na tolerancję użytych elementów – wyniki najczęściej nie będą pokrywać się z założonymi. Aby pokonać ten problem, w programie uwzględniono specjalną procedurę kalibracyjną. Jest ona uruchamiana, gdy zaraz po restarcie mikrokontrolera zostanie wykryte naciśnięcie klawisza SW4 – `if(bit_is_clear(PIND,1))`. W pętli `do...while` jest realizowany cykliczny pomiar napięcia wejściowego, przy czym w tym przypadku nie korzysta się z przerw. Tak więc po wyzerowaniu flagi zgłoszenia przerwania (wpisanie „1” na pozycję flagi) następuje oczekiwanie w pętli, aż ponownie zostanie ona ustawiona przez funkcję przechwytywania. Stan flagi jest kontrolowany programowo:

```
sbi(TIFR,ICF1);
while(bit_is_clear(TIFR,ICF1));    //czekaj na
                                   //ustawienie flagi
```

Wyjście z pętli jest możliwe po naciśnięciu klawisza SW1. Powoduje to wyliczenie współczynnika kalibracyjnego i zapisanie go w pamięci EEPROM mikrokontrolera poczynawszy od adresu \$01. Współczynnik jest liczbą typu integer, zajmuje więc dwa bajty. Do obsługi pamięci EEPROM wykorzystano funkcje biblioteczne języka AVR-GCC. Zapis odbywa się przy użyciu `eeeprom_wb(adres,dana_8_bit)`, do odczytu zaś służy `eeeprom_rw(adres,dana_16_bit)`. Jak widać, dzięki tym procedurom programista został całkowicie zwolniony z obowiązku przestrzegania wszystkich reguł dotyczących obsługi pamięci EEPROM, poza świadomą rezygnacją z zerowego adresu (patrz opis pamięci EEPROM). Wszystkie operacje, jakie wykonuje mikrokontroler podczas realizacji tych funkcji można obejrzeć dokładnie po uruchomieniu podglądu kodu na poziomie asemblera np. w AVR Studio. Jeśli po restarcie systemu nie zostanie wykryte naciśnięcie klawisza SW4, przed wejściem do pętli pomiarowej współczynnik kalibracji jest odczytywany z pamięci EEPROM i będzie później wykorzystywany do przeliczenia czasu ładowania kondensatora na liczbę binarną wyświetlaną na diodach LED. Kalibrację należy przeprowadzić po zaprogramowaniu mikrokontrolera. Później układ będzie zawsze gotowy do pomiarów po włączeniu zasilania lub zrestartowaniu mikrokontrolera.



Rys. 14.15. Wykres liniowości przetwornika A/C badanego w ćwiczeniu

Kolejne kroki postępowania są następujące:

1. Naciśnij przycisk SW4 i włącz zasilanie lub zrestartuj system (nie puszczać SW4).
2. Doprowadź do wejścia PORTD1-1 napięcie wzorcowe i ustaw za pomocą woltomierza wartość, której powinien odpowiadać stan \$3F przetwornika (np. 4,5 V). Podanie później (w trakcie pomiarów) takiego napięcia na wejście spowoduje zapalenie sześciu diod LED odpowiadających bitom najbardziej znaczącym.
3. W czasie kalibracji palą się lampki LED3 do LED8.
4. Naciśnij klawisz SW1, co spowoduje automatyczne wejście w tryb pomiarowy. Zmianom napięcia wejściowego powinno towarzyszyć zmienianie stanu diod LED.

Mierzone napięcie można pobierać z suwaka potencjometru P1, służącego normalnie do regulacji kontrastu wyświetlacza LCD (wyprowadzenie 3 łączówki LCD1). Na rysunku 14.15 przedstawiono wykres liniowości przetwornika A/C badanego w ćwiczeniu 6.

Program realizujący przedstawione zadanie znajduje się na listingu 14.7. Należy skompilować program `cwiczb.c` i zaprogramować mikrokontroler z zestawu ZL1AVR plikiem wynikowym `cwiczb.hex`.

Konfiguracja płytki ZL1AVR:

- zworka J3 w pozycji 1-2, zworka J4 w pozycji 2-3 – dołączony oscylator wewnętrzny,
- zworka JP1 zwarta – globalne włączenie diod LED,

- zworka JP4 i JP5 w pozycji 2-3 (dołączenie pojedynczych klawiszy do mikrokontrolera),
- zworka JP6 i JP7 zwarta (dołączenie klawiszy do mikrokontrolera),
- zworki ZW_PORTB zwarte w pozycjach 1-2, 3-4, 5-6, 7-8, 9-10, 11-12, zworki 13-14 i 15-16 rozwarte,
- zworki JP8 i JP9 zwarte (włączenie źródła prądowego i kondensatora pomiarowego),
- zworka ZW_PORTD 1-2 rozwarta,
- położenie pozostałych zworek nieistotne (np. rozłączone),
- wyświetlacz alfanumeryczny LCD zdemontowany,
- połączone końcówki gniazda LCD-3 z PORTD-1 (w przypadku korzystania potencjometru P1 do regulacji napięcia wejściowego),
- jeśli do pomiarów będzie wykorzystywane zewnętrzne źródło napięcia, należy je doprowadzić do gniazda PORTD-1 (połączenia z punktu wyżej nie wykonywać), końcówkę 0V źródła połączyć z masą płytki ZL1AVR.

List. 14.7. Program do ćwiczenia 6

```

/*****
 * Ćwiczenie 6 - Zastosowanie komparatora analogowego do budowy
 * przetwornika analogowo-cyfrowego.
 * Wyzwalanie funkcji przechwytywania timera
 * za pomocą komparatora.
 * Przerwanie od przechwytywania.
 * Obsługa wewnętrznej pamięci EEPROM.
 * J.D. '2003
 *****/
#include <io.h>
#include <interrupt.h>
#include <signal.h>
#include <eeprom.h>

/***** zmienne globalne *****/
unsigned char liczt0;
volatile unsigned char pomiar; //flaga dokonania pomiaru
union{
    unsigned int wspkal; //współczynnik kalibracji
    unsigned char wspkalb[2];
}uwspkal;

void czekaj(unsigned long zt) //funkcja opóźnienia
{
    #define tau 10.38
    unsigned char zt1;
    for(;zt>0;zt--)
    {
        for(zt1=255;zt1!=0;zt1--);
    }
}

SIGNAL (SIG_INPUT_CAPTURE1) //obsługa przerwania od przechwylenia
{

```

```

union{
    unsigned int czas;
    unsigned char czasb[2];
}uczas;
unsigned char czas8;

uczas.czasb[0]=ICR1L; //zatrzaśnij rejestry przechwytywania
uczas.czasb[1]=ICR1H;
czas8=-(uczas.czas/uwspkal.wspkal)<<2);
//normalizacja wyniku do postaci 6-bitowej
//liczby binarnej przesuniętej o 2 bity
//w lewo (PORTB i 0 są wykorzystywane
//przez komparator analogowy)
PORTB=(PORTB&0x03)|czas8; //wyświetl wynik na LEDach
pomiar=1; //pomiar dokonany (zapal flagę)
sbi(PORTD,4); //zaczynaj rozładowywać kondensator
//pomiarowy
}

int main(void)
{
    DDRC=0x10; //PORTD we oprócz PD4
    PORTD=0xff; //z podciąganiem
    PORTB=0x01; //PB0 z podciąganiem
    DDRB=0xfc; //PORTB7-2 - wy, PORTB1-0 - we
    TCCR1A=0; //funkcje porównania i PWM wyłączone
    TCCR1B=0x41; //preskaler XTAL/1 dla TC1, przechwyty-
    //wanie na narastającym zboczach
    TIMSK=0x08; //zezwoleń na przerwanie od
    //przechwytywania
    ACSR=1<<ACIC; //zezwoleń na wyzwalanie
    //przechwytywania komparatorem

    czekaj(10*tau);
    TIFR=0xca; //kasuj przerwanie od timerów

    if(bit_is_clear(PIND,1)) //wciśnięty SW4 - kalibracja
    {
        do
        {
            cbi(PORTD,4); //ładuj kondensator pomiarowy
            TCNT1H=0; //zeruj licznik i pomiar czasu ładowania
            TCNT1L=0;
            sbi(TIFR,ICF1); //kasuj flagę przechwytywania
            while(bit_is_clear(TIFR,ICF1));
            //czekaj aż napięcie mierzone zrówna
            //się z napięciem wejściowym

            uwspkal.wspkal=(ICR1L+256*ICR1H);
            sbi(PORTD,4); //kasuj flagę przechwytywania
            czekaj(1*tau); //opóźnienie związane z częstotliwością
            //odświeżania
        }while(bit_is_set(PIND,0));
        uwspkal.wspkal/=64;
        eeprom_wb(1,uwspkal.wspkalb[0]);
        //zapisz współczynnik kalibracji
        eeprom_wb(2,uwspkal.wspkalb[1]);
        //do pamięci EEPROM
        sbi(TIFR,ICF1); //kasuj flagę przechwytywania
    }
    else
    {
        uwspkal.wspkal=eeprom_rw(1);
        //odczytaj współczynnik kalibracji z EEPROM-u
    }
}

sei(); //odblokuj globalne przerwania

```

```

while(1)                //główna pętla pomiarowa
{
    cbi(PORTD,4);        //ładuj kondensator pomiarowy
    TCNT1H=0;            //zresetuj licznik 1 - pomiar czasu ładowania
    TCNT1L=0;
    pomiar=0;
    while(pomiar==0);    //czekaj aż napięcie mierzone zrówna się z
                        //napięciem na kondensatorze pomiarowym
    czekaj(23*tau);      //opóźnienie związane z częstotliwością
                        //odświeżania wskaźnika LED
}

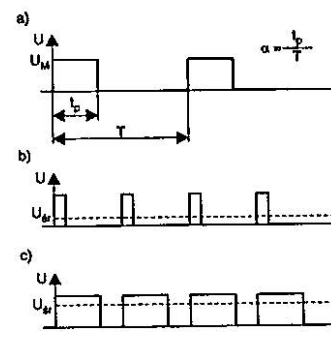
```

14.2.7. Ćwiczenie 7

Regulacja obrotów silnika DC. Wykorzystanie Timera1 jako modulatora PWM. Obsługa pojedynczych klawiszy

W życiu codziennym często spotykamy się ze sprzętem elektrycznym, którego parametry mogą być regulowane przez użytkowników, np. poprzez zmianę napięcia zasilającego. Przykładem mogą być urządzenia wykorzystujące silniki prądu stałego lub zmiennego (regulacja obrotów), sprzęt oświetleniowy (regulacja jasności), piece i grzejniki elektryczne (regulacja mocy) itp. Jedną z pierwszych metod praktycznej realizacji takich urządzeń, jaka nasuwa się w sposób naturalny, to regulacja wartości napięcia zasilającego. Nie jest to niestety zadanie łatwe do wykonania, szczególnie w przypadku urządzeń dużej mocy, gdyż może się wiązać z występowaniem dużych strat. Zauważmy jednak, że silniki, żarówki, grzałki itp. z zasady działania są urządzeniami całkującymi, czyli mówiąc prościej potrafią same uśredniać przyłożone do nich napięcie, co objawia się zmianą parametrów końcowych (obroty, jasność świecenia, moment obrotowy). Wniosek jest taki, że jeśli do zasilania takich odbiorników energii zastosowano przebieg zmienny, to wraz ze zmianą jego właściwości, będących funkcją czasu, będą zmieniały się również parametry urządzeń. Wydaje się, że mając do dyspozycji mikrokontroler najprostszym rozwiązaniem jest wykorzystanie go jako generatora przebiegu prostokątnego o zmiennym współczynniku wypełnienia. W przypadku mikrokontrolerów AVR rozumowanie takie jest tym bardziej zasadne, że są one wyposażone w timer (Timer1), który może pracować jako modulator PWM (*Pulse Width Modulation*). Zasada wykorzystania modulatora PWM będzie omówiona w tym ćwiczeniu na przykładzie prostego regulatora obrotów silnika prądu stałego. Do prób użyto małego silniczka z zestawu Lego zasilanego napięciem stałym 5 V.

Oczywiście, w podobny sposób można sterować znacznie większymi odbiornikami energii, zasilanymi nawet z sieci energetycznej. Wymagana jest wte-



Rys. 14.16. Sposób liczenia wartości współczynnika wypełnienia przebiegu prostokątnego (a), przykładowe wartości średniego napięcia w zależności od współczynnika wypełnienia (b) i (c)

$$\alpha = \frac{t_p}{T}$$

Wartość średnia takiego przebiegu (czyli całka za okres) będzie wprost proporcjonalna do wartości napięcia U_M i współczynnika wypełnienia α (rysunki 14.16b i c). Jak widać metoda ta nadaje się do regulacji parametrów w pełnym zakresie: od 0 do 100%.

Do realizacji modulatora PWM wykorzystamy Timer1. Odpowiedni tryb pracy wybiera się poprzez ustawienie dwóch najmłodszych bitów rejestru TCCR1A (PWM11 i PWM10). Układ można skonfigurować w zależności od potrzeb jako 8-, 9- lub 10-bitowy modulator PWM. Uzyskuje się w ten sposób różne rozdzielczości, a więc i precyzję regulacji. Szczegóły omówiono w rozdziale 5.3. W ćwiczeniu wybrano wariant 10-bitowy, w którym wyjście OC1 (PB3) jest ustawiane w momencie zrównania się zawartości rejestrów licznika (TCNT1H i TCNT1L) z rejestrami porównania (OCR1AH i OCR1AL) podczas zliczania w dół. Jest to tzw. prosty wariant modulacji, w którym stan wysoki wyjścia OC1 jest proporcjonalny do zawartości rejestrów OCR1A. Ten tryb jest konfigurowany bitami COM1A1 i COM1A0 rejestru TCCR1A (patrz tabela 5.6). Wydawać by się mogło, że 10-bitowa rozdzielczość, odpowiadająca 1024 krokowi regulacji, jest zbyt duża. W praktyce okazuje się jednak, że efektywny zakres regulacji obrotów silnika DC pokrywa zaledwie część możliwych wartości. W konkretnym przypadku wynosił on $\alpha = 0,75 \dots 1$. Wartość średnia napięcia zasilającego silnik odpowiadająca wypełnieniu poniżej 0,75 była na tyle mała, że silnik nie był w stanie kręcić wirnikiem. W tych warunkach efektywna rozdzielczość sterowania odpowiadała mniej więcej 8-bitowej.

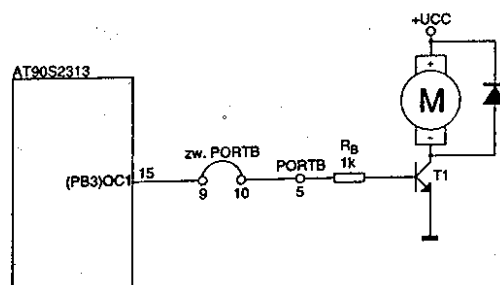
Regulacja częstotliwości generowanego przebiegu PWM jest niestety dość ograniczona. Trudno tu mówić nawet o regulacji, jest to raczej wybór jednej

z kilku możliwych wartości. Częstotliwość ta zależy jedynie od rozdzielczości PWM i wartości preskalera ustawionego dla Timer1 (patrz tablice 5.4 i 5.5). W ćwiczeniu wybrano preskaler 3, czyli podział częstotliwości przez 64 (tablica 5.4). Dla 10-bitowej PWM mamy dodatkowo podział $f_{TC1}/2046$. Ostatecznie, przy kwarcu 8 MHz, wyjściowa częstotliwość przebiegu PWM będzie równa:

$$f_{PWM} = \frac{f_{osc}}{64 \cdot 2046} = 61,1 \text{ Hz}$$

Wybranie mniejszych wartości w przypadku sterowania silnikiem nie jest wskazane, gdyż wystąpią wyraźnie odczuwalne szarpania wału. Można natomiast z powodzeniem stosować je przy sterowaniu grzałką.

Przed przystąpieniem do eksperymentów należy na uniwersalnym polu montażowym płytki ZL1AVR przygotować interfejs pomiędzy mikrokontrolerem i silnikiem. Jego schemat pokazano na rysunku 14.17. Typ użytego tranzystora zależy od rodzaju silnika. Na ogół przy zasilaniu 5 V, prąd pobierany przez niego z zasilacza będzie rzędu kilkuset mA. Użyty do prób silnik pobierał ok. 200 mA przy pełnych obrotach i bez obciążenia. Wartość rozruchowa była jednak znacznie większa. Trzeba jeszcze pamiętać o tym, że silnik jest elementem indukcyjnym i podczas impulsowego zasilania powstają na nim dość duże przepięcia, mogące uszkodzić złącze kolektor-emiter tranzystora. Szkodliwe impulsy można gasić kondensatorem o wartości kilkuset pF dołączonym równolegle do silnika lub diodą (jak na rysunku 14.17). Warto również wyposażyć stabilizator U1 w radiator, gdyż jeśli silnik będzie zasilany z napięcia +5 V dostępnego na płytce, może się on dość mocno grzać. Baza tranzystora powinna być połączona poprzez rezystor R_B z wyjściem OC1 mikrokontrolera (PB3). W tym celu najlepiej jest założyć zworkę ZW_PORTB w pozycji 9-10, a na szpiłkę PORTB-5 włożyć kabelek, którego drugi koniec należy przylutować do rezystora R_B , zgodnie z rysunkiem 14.17.



Rys. 14.17. Schemat elektryczny interfejsu służącego do sterowania silnikiem DC

Regulacji obrotów silnika dokonuje się w ćwiczeniu za pomocą przycisków SW1 (zmniejszanie) i SW4 (zwiększanie). Do zapamiętywania aktualnej wartości współczynnika wypełnienia przebiegu PWM służy zmienna upwm. Jest to unia zawierająca składową typu int (upwm.pwm), niezbędną do przechowania liczby większej niż 8-bitowa oraz tablicę dwuelementową typu char (upwm.pwmc), przydatną do wyluskiwania poszczególnych bajtów składowej upwm.pwm. Na początku programu zmiennej upwm.pwm jest nadawana wartość 0x3ff, co jest równoznaczne z wysterowaniem wyjścia OC1 w stan wysoki. Silnik po restarcie rusza więc z pełnymi obrotami. W pętli głównej programu badane jest naciśnięcie klawisza SW4 i jeśli fakt ten zostanie wykryty, wartość współczynnika wypełnienia (a ściślej długość impulsu t_p) jest zwiększana o aktualnie obowiązujący przyrost. Dodatkowo sprawdza się, czy nie przekroczono wartości TOP (0x3ff przy takich ustawieniach jak w ćwiczeniu), co byłoby interpretowane jako przeskok do wartości 0, a więc powodowało zatrzymanie silnika. Jeśli taka sytuacja nastąpi, zmiennej upwm.pwm nadaje się wartość TOP. Po tych czynnościach wywoływana jest, znana już z wcześniejszych ćwiczeń, funkcja czeka_j, generująca opóźnienie. Uzyskuje się dzięki temu efekt eliminacji trzasków a zarazem zabezpiecza się przed niepożądanym ponownym zinterpretowaniem stanu SW4 (w następnej iteracji pętli), gdy nie został jeszcze zwolniony. Zakłada się przy tym, że pojedyncze naciśnięcie klawisza nie powinno trwać dłużej niż 150 ms. Jest to wartość sprawdzona w praktyce. Podobne działania podejmowane później, służą do obsługi klawisza SW1. W tym przypadku współczynnik wypełnienia jest jednak zmniejszany. Zadbano również, aby po osiągnięciu wartości zerovej nie zmniejszać dalej tego współczynnika. Każde naciśnięcie dowolnego klawisza powoduje inkrementowanie zmiennej licznik1. Przekroczenie wartości 6 odpowiada przytrzymaniu klawisza przez ok. 0,9 s i powoduje zwiększenie zmiennej przyrost o 16. Odpowiada to przyspieszeniu regulacji współczynnika wypełnienia. Po wykryciu zwolnienia klawiszy, obie zmienne wracają do swoich wartości początkowych. Ostatnie dwie instrukcje pętli głównej służą do przepisania parametrów przebiegu PWM ze zmiennej upwm.pwm do rejestrów OCR1A, co znajduje odbicie w zmianie obrotów silnika.

Program realizujący przedstawione zadanie znajduje się na listingu 14.8. Należy skompilować program *cwicz7.c* i zaprogramować mikrokontroler z zestawu ZL1AVR plikiem wynikowym *cwicz7.hex*.

Konfiguracja płytki ZL1AVR:

- zworka J3 w pozycji 1-2, zworka J4 w pozycji 2-3 – dołączony oscylator wewnętrzny,

- zworka JP1 zwarta – globalne włączenie diod LED,
- zworka JP4 i JP5 w pozycji 2-3 (dołączenie pojedynczych klawiszy do mikrokontrolera),
- zworka JP6 i JP7 zwarta (dołączenie klawiszy do mikrokontrolera),
- połączyć kabelkiem nóżkę 9 ZW_PORTB z nóżką 16 ZW_PORTB,
- nóżkę 5 łączówki PORTB połączyć z rezystorem RB interfejsu przygotowanego wcześniej na uniwersalnym polu montażowym (rysunek 14.17),
- zworki JP8 i JP9 rozwarte,
- położenie pozostałych zwerek nieistotne (np. rozłączone).

List. 14.8. Program do ćwiczenia 7

```

/*****
/* Ćwiczenie 7 - Regulacja obrotów silnika DC          */
/*          Modulacja PWM przy użyciu timer1          */
/* J.D. '2003                                           */
*****/
#include <io.h>

void czekaj(unsigned long zt) //funkcja opóźnienia
{
    #define tau 10.38
    unsigned char zt1;
    for(;zt>0;zt--)
    {
        for(zt1=255;zt1!=0;zt1--);
    }
}

int main( void )
{
    unsigned char licznik1=0; //zmienna wykorzystywana do pomiaru
                             //czasu naciśnięcia przycisków
    char przyrost=1;         //przyrost zmiany współczynnika
                             //wypełnienia sygnału PWM
    union
    {
        unsigned int pwm;
        unsigned char pwmc[2];
    }volatile upwm;          //aktualny współczynnik wypełnienia
                             //sygn. PWM

    DDRB=0x08;              //PB3 - wy (OC1 - wyjście PWM),
                             //pozostałe we
    PORTB=0;                 //bez podciągania
    DDRD=0xfc;               //PD1 i PD0 - we (obsługa klawiszy SW1
                             //i SW4), pozostałe wy
    PORTD=0x03;              //wejścia z podciąganiem (potrzebne dla
                             //klawiatury)
    TCCR1A=0x83;             //PWM 10 bitowy
                             //zerowanie OC1 po spełnieniu warunku
                             //równości podczas liczenia w górę,
                             //ustawiane podczas liczenia w dół
    TCCR1B=0x03;             //prescaler=3, co przy 10-bit PWM daje
                             //Fwy=ok. 61Hz @8MHz
    TCNT1L=0x00;            //wstępne ustawienie licznika1

```

```

TCNT1H=0x00;
upwm.pwm=0x3ff;            //początkowo silnik włączony, wartość TOP
                             //odpowiada wysokiemu poziomowi na
                             //wyjściu OC1 (PB3)
                             //główna pętla programu
while(1)
{
    if(bit_is_clear(PIND,1)) //czy wciśnięto SW4
    {
        upwm.pwm+=przyrost; //zwiększ pwm
        if(upwm.pwm>0x3ff)
        {
            upwm.pwm=0x3ff; //jeśli przekroczono wartość TOP, to
                             //ustaw TOP
        }
        czekaj(150*tau);    //eliminacja drgań i powtórnej
                             //interpretacji naciśnięcia przycisku
        licznik1++;         //mierz długość naciśnięcia przycisku
    }
    else
    {
        if(bit_is_clear(PIND,0)) //czy wciśnięto SW1
        {
            upwm.pwm-=przyrost; //zmniejsz pwm
            if(upwm.pwm>0x3ff) //odpowiada to upwm.pwm<0
            {
                upwm.pwm=0;    //jeśli przekroczono wartość zero, to
                                //ustaw zero
            }
            czekaj(150*tau);    //eliminacja drgań i powtórnej
                                //interpretacji naciśnięcia przycisku
            licznik1++;         //mierz długość naciśnięcia przycisku
        }
        else
        {
            licznik1=0;        //zeruj licznik pomiaru czasu naciśnięcia
                                //klawisza ponieważ wszystkie przyciski
                                //są zwolnione
                                //ustaw początkową wartość przyrostu
                                //dla pwm
        }
    }
    if(licznik1>6)
    {
        przyrost=16;          //wykryto długie naciśnięcie przycisków,
                                //zwiększ krok regulacji
        licznik1=6;
        OCR1H=upwm.pwmc[1];   //wpisz aktualnie ustawiony współczynnik
                                //do rejestrów
        OCR1L=upwm.pwmc[0];   //OCR1 timer1
    }
}

```

14.2.8. Ćwiczenie 8

Sterowanie obrotami silnika DC z komputera PC. Wykorzystanie Timera1 jako modulatora PWM. Wykorzystanie UART-a mikrokontrolera do prowadzenia transmisji szeregowej pomiędzy płytką ZL1AVR a komputerem PC

W poprzednim ćwiczeniu wykonaliśmy autonomiczny sterownik silnika prądu stałego. Tym razem zbudujemy interfejs pozwalający sterować silnikiem z poziomu komputera PC. Do tego celu zostanie wykorzystany wbudowany w mikrokontroler AVR układ asynchronicznej transmisji szeregowej – UART (*Universal Asynchronous Receiver and Transmitter*). Dzięki niemu można w bardzo prosty sposób zrealizować łączność poprzez port szeregowy COM, w jaki jest wyposażona większość komputerów.

Niewątpliwą zaletą łączności poprzez łącze RS232C jest prostota protokołu komunikacyjnego i fakt, że większość mikrokontrolerów, nie tylko AVR, ma zaimplementowany układ UART. Wadą natomiast jest konieczność dobudowywania konwertera poziomów, zgodnego ze standardem RS232. Na płytce ZL1AVR rolę tę pełni układ U4.

Interfejs RS232

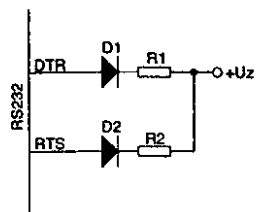
Interfejs RS232 ma dosyć długą historię, sięgającą lat 60. XX wieku. Mimo to przetrwał do dzisiaj w prawie niezmienionej postaci. Wpłynęła na to jego ogromna popularność tak w sprzęcie profesjonalnym, jak i powszechnego użytku. W roku 1991 zmieniono nazwę standardu z RS232 na EIA232, nadal jednak, chyba z przyzwyczajenia częściej jest stosowana nazwa pierwotna. Interfejs RS232C opracowano do prowadzenia łączności pomiędzy komputerem i terminalem lub pomiędzy dwoma terminalami bez komputera. W praktyce jest wykorzystywany w wielu odmianach. Najbardziej rozbudowane pozwalają np. na sprzętowe kontrolowanie przepływu danych i prowadzenie sygnalizacji, najprostsze realizują łączność za pomocą tylko trzech przewodów. Standard RS232C określa dwa rodzaje urządzeń, pomiędzy którymi jest prowadzona transmisja danych. Jest to DTE (*Data Terminal Equipment*) – najczęściej będzie nim komputer oraz DCE (*Data Circuit-terminating Equipment*) – np. modem. Norma przewiduje stosowanie odpowiednich gniazd połączeniowych, zakłada również konkretne poziomy napięć na liniach interfejsu. W praktyce, najczęściej nie ma potrzeby korzystania ze wszystkich możliwości interfejsu, wykorzystuje się jego najprostszą wersję. Schemat połączeń takiego wariantu przedstawiono na **rysunku 14.3**. Jest to połączenie nazywane *null-modem*. Opis dotyczy wersji z 9-stykowymi gniazdami DSUB, w które najczęściej są wyposażane komputery PC.

Łącząc urządzenia transmisyjne kablem *null-modem* zakłada się, że zarówno nadajniki jak i odbiorniki są zawsze gotowe do pracy. Wykorzystanie tylko trzech przewodów – dwóch transmisyjnych, potrzebnych do komunikacji dwukierunkowej oraz przewodu odniesienia (masy sygnałowej) – uniemożliwia jakąkolwiek możliwość sprzętowego kontrolowania przepływu danych. Oprogramowanie używane do prowadzenia transmisji może jednak korzystać z wybranych sygnałów interfejsu, zakładając że łączność jest prowadzona opierając się na jego pełnej wersji. W takich przypadkach są niezbędne widoczne na **rysunku 14.3** połączenia wewnętrzne po obu stronach kabla. Ich brak będzie traktowany jako zgłoszenie braku gotowości danego urządzenia, co w konsekwencji mogłoby doprowadzić do zawieszenia transmisji lub nawet niemożności jej zainicjowania. Z tego względu bezpieczniej jest uwzględniać takie zapętlenia w urządzeniach. Na płytce ZL1AVR wykonano je na obwodzie drukowanym. Standard RS232C określa dokładnie przypisanie wyprowadzeń gniazd DSUB poszczególnym sygnałom interfejsu. Rozkład wyprowadzeń dla wersji z gniazdami 9-stykowymi przedstawiono w **tablicy 14.3**.

Trzeba zwrócić uwagę na to, że w gniazdach DTE (DB9 męskie) i DCE (DB9 żeńskie) zamienione są miejscami wyprowadzenia TXD i RXD. Dzięki

Tab. 14.3. Definicje sygnałów interfejsu RS232C (wersja 9-stykowa)

Oznaczenie	Nazwa	Opis	Numer wyprowadzenia w gnieździe DSUB9
TXD	Transmitted Data	Dane nadawane z DTE do DCE. Przy braku transmisji – stan Mark	3 (DB9M), 2 (DB9F)
RXD	Received Data	Dane odbierane przez DTE z DCE. Przy braku transmisji – stan Mark	2 (DB9M), 3 (DB9F)
RTS	Request To Send	Żądanie nadawania. Sygnał informuje DCE o gotowości wysłania danych przez DTE. Aktywny stan „0” (ujemne napięcie)	7
CTS	Clear To Send	Gotowość do odbioru. Sygnał z DCE informuje o tym, że DCE jest gotowe do przyjęcia danych. Aktywny stan „0” (ujemne napięcie)	8
DSR	DCE Ready	Gotowość DCE do prowadzenia transmisji. Stan „0” oznacza, np. że urządzenie jest włączone i prawidłowo zainicjowane	6
DTR	DTE Ready	Gotowość DTE do prowadzenia transmisji. Stan „0” oznacza, że DTE chce zrealizować transmisję (otworzyć kanał komunikacyjny)	4
DCD	Carrier Detect	Wykryto nośną. Sygnał wykorzystywany we współpracy z modemami. Stan „0” oznacza, że modem nawiązał połączenie z drugim modemem	1
SG	Signal Ground	Masa sygnałowa. Linia ta stanowi poziom odniesienia dla wszystkich sygnałów elektrycznych na styku RS232	5
RI	Ring Indicator	Sygnał dzwonięcia. Wykorzystywany, gdy DTE jest modemem. Stan „0” oznacza, że modem odbiera sygnał dzwonięcia z linii telefonicznej.	9

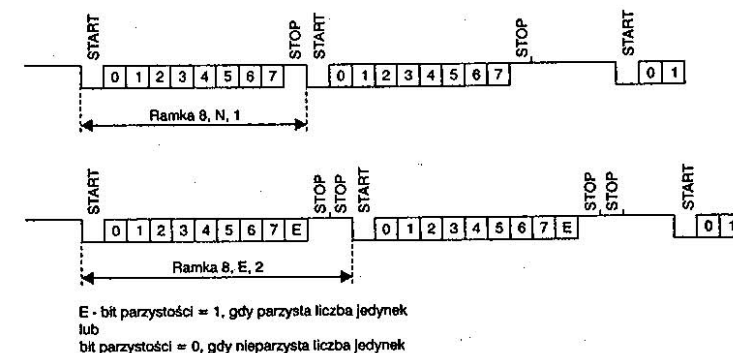


Rys. 14.18. Wykorzystanie linii DTR i RTS jako źródła zasilania

temu nadajnik jednego urządzenia jest dołączany do odbiornika drugiego i odwrotnie. Połączenie może więc być zrealizowane za pomocą kabla 1:1. Niektóre linie interfejsu RS232C mogą być w pewnych rozwiązaniach wykorzystywane w sposób niestandardowy. Na przykład umożliwiają one zasilanie urządzeń charakteryzujących się małym poborem prądu (do kilku mA) bezpośrednio z interfejsu. Program uruchomiony na komputerze powinien przy tym zadbać o ustawienie linii wyjściowych DTR i RTS w stan SPACE („0”), któremu odpowiada dodatnie napięcie na wyjściu. Schemat odpowiednich połączeń jest przedstawiony na rysunku 14.18. Rezystory wyrównujące R1 i R2 o niewielkiej wartości zapewniają w miarę równomierne obciążenie obu linii. Diody D1 i D2 zabezpieczają przed wstecznym podaniem napięcia z jednego wyjścia na drugie. Ze względu na spadek napięcia na diodach, najlepiej zastosować diody Shottky’ego. Do wyjścia $+U_z$ można dołączyć ewentualnie stabilizator charakteryzujący się małym prądem własnym, aby nie zmniejszać i tak niewielkiej wydajności prądowej takiego „zasilacza”.

Mówiąc o transmisji szeregowej nie można pominąć zagadnień związanych z formatem danych i protokołem transmisyjnym. Format danych jest związany z przyjętym sposobem przesyłania poszczególnych bitów przez kanał transmisyjny, dotyczy więc najniższego poziomu. Protokół zaś określa zasady prowadzenia transmisji na wysokim poziomie. Dotyczy metod nawiązywania łączności, sposobów organizowania danych w bloki, kontroli poprawności transmisji na poziomie blokowym (np. wyliczanie sum kontrolnych CRC) itp. Łączność wykorzystująca kanał szeregowy może być prowadzona w sposób synchroniczny lub asynchroniczny. Pierwsza z metod wymaga przesyłania zegara transmisyjnego, co może się wiązać z koniecznością dołączenia dodatkowej linii do interfejsu. Można również stosować specjalne metody kodowania danych, umożliwiające odtwarzanie zegara transmisyjnego po stronie odbiornika bez dodatkowych połączeń. Zagadnienia te wykraczają poza tematykę książki i nie będziemy się nimi zajmować. Transmisja synchroniczna jest rzadko stosowana w urządzeniach amatorskich. Druga metoda, wykorzystująca transmisję asynchroniczną jest stosowana natomiast bardzo powszechnie.

Przykładowe formaty danych (tzw. ramki) przesyłanych łączy szeregowym przedstawiono na rysunku 14.19. Najbardziej popularnym formatem jest 8,n,1. Zapis ten oznacza, że ramka składa się z 8 bitów danych, nie występuje bitu parzystości i zawiera jeden bit stopu. Niezależnie od tego, każda



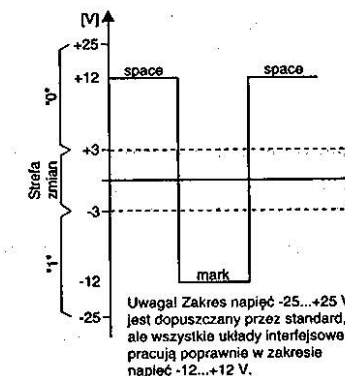
Rys. 14.19. Wygląd przykładowych ramek stosowanych w asynchronicznej transmisji szeregowej

ramka zawiera jeden bit startu. Ramka może mieć długość 5, 6, 7 lub 8 bitów danych. Czasami do kontroli poprawności transmisji używa się bitu parzystości. Bit taki osiąga wartość zależną od liczby jedynek występujących w bitach danych, np. parzystej liczbie bitów danych o wartości „1” odpowiada bit parzystości o wartości „1”. Taka kontrola poprawności transmisji jest chyba najprostszą z możliwych, ale cechuje się stosunkowo małą skutecznością. Z tego powodu najczęściej się z niej rezygnuje.

Początek ramki jest sygnalizowany pojedynczym bitem startu (rysunek 14.19). Po nim następują bity danych, wysyłane od najmniej, do najbardziej znaczącego. Następnie może występować bit parzystości (najczęściej go nie ma) i jeden lub dwa bity stopu o wartości „1”. Ramkę zawsze zaczyna bit startu o wartości „0”. Ramki mogą być wysyłane pojedynczo, niekoniecznie

muszą być formowane w większe bloki.

W chwilach przerw w transmisji linia TXD (wyjście nadajnika) przyjmuje stan MARK („1”). Interfejs RS232 wykorzystuje logikę ujemną, stanowi MARK odpowiada ujemne napięcie na linii wyjściowej (rysunek 14.20). Należy zwrócić uwagę na duże wartości napięć występujących na liniach interfejsu RS232C. Przyjęto je po to, aby zwiększyć odporność transmisji na zakłócenia w linii. Jest to oczywiście jakaś metoda, ale tak naprawdę skuteczne jest zastosowanie pętli prądowej (jak np. w interfejsie RS422).



Rys. 14.20. Stany występujące na liniach interfejsu RS232

Do prowadzenia transmisji szeregowej niezbędne jest utrzymywanie prawidłowej synchronizacji nadajnika z odbiornikiem. W transmisji synchronicznej problem ten jest rozwiązany w sposób naturalny – zegar transmisyjny jest przesyłany bezpośrednio lub jest odtwarzany innymi metodami po stronie odbiorczej. W transmisji asynchronicznej sygnał zegara nie występuje, trzeba go wytwarzać niezależnie w nadajniku i odbiorniku. Do podsynchronizowywania odbiornika służy bit startu, który powoduje zerowanie liczników odmierzających czas w układzie UART odbiornika. Gdyby taki zabieg nie był wykonany, po pewnej liczbie przesłanych danych układy „rozjechałyby” się, nawet przy wysokostabilnych generatorach. Zdolność do wstępnej synchronizacji odbiornika nie zwalnia nas jednak z konieczności jednakowego ustawiania parametrów generatorów nadajnika i odbiornika. Standard RS232 przewiduje kilka typowych szybkości transmisji. Tu niestety powstaje problem. Okazuje się, że biorąc dowolne rezonatory kwarcowe wykorzystywane przez oscylator mikrokontrolera, jego układy czasowe nie będą w stanie wytworzyć przebiegu zegarowego o częstotliwości odpowiedniej dla przyjętej szybkości transmisji. Zilustrowano to w **tablicy 8.1**. Wspomniane wyżej zdolności odbiorników tolerują pewne odchyłki w doborze parametrów. Dopuszcza się 1% błąd. Na płytce ZL1AVR zastosowano rezonator 8 MHz. Z **tablicy 8.1** wynika, że pozwoli on na osiągnięcie największej dopuszczalnej prędkości transmisji równej 38400 b/s. Efektywna prędkość transmisji będzie jednak mniejsza. Pamiętajmy, że ramka zawiera oprócz bitów danych także bit startu, bit/bity stopu i ewentualnie bit parzystości. Powyższe problemy z doбором prędkości transmisji obowiązują tylko wtedy, gdy przewidywane jest dołączanie standardowych urządzeń komunikacyjnych. Parametry interfejsu RS232 muszą być wtedy przestrzegane bezwzględnie. Jeśli planujemy łączyć ze sobą dwa systemy mikroprocesorowe, to wystarczy zadbać jedynie o to, by ich UART-y były identycznie inicjowane. To, że będą się komunikowały z niestandardowymi prędkościami nie będzie miało żadnego znaczenia.

Sterowanie silnikiem z komputera

Zadanie, jakie mamy do rozwiązania w tym ćwiczeniu, jest bardzo podobne do poprzedniego. Zasada sterowania silnikiem pozostała bez zmian. Obroty zależą od wartości średniej przebiegu PWM, jakim jest zasilany. W tym ćwiczeniu sterowanie będzie prowadzone zdalnie np. za pomocą komputera PC. Przyciski znajdujące się na płytce ZL1AVR nie będą wykorzystywane. Naciśnięcie klawisza na klawiaturze komputera powoduje przesłanie jego kodu ASCII do mikrokontrolera na płytce ZL1AVR. Po odebraniu znaku przez UART następuje jego interpretacja, po czym są podejmowane stosowne działania. Znaczenie poszczególnych klawiszy jest następujące:

◻ – zmniejszenie obrotów. Znak ◻ występuje na jednym klawiszu razem ze znakiem ◻ (dobrze kojarzącym się ze zmniejszaniem wartości parametru). Znak ◻ wymaga jednak naciśnięcia dodatkowo klawisza **[Shift]**, co mogłoby być niepotrzebnym utrudnieniem.

◻ – zwiększenie obrotów. Uzasadnienie wyboru tego klawisza jest podobne jw.,

◻ – natychmiastowe zatrzymanie silnika,

◻ – natychmiastowy start silnika z pełnymi obrotami,

◻ – żądanie przekazania aktualnych parametrów przebiegu PWM sterującego silnikiem. W odpowiedzi na odebranie tego znaku, mikrokontroler przesyła do komputera aktualną wartość rejestru OCR1A w postaci łańcucha ASCII reprezentującego liczbę szesnastkową. Liczba jest poprzedzona charakterystycznym dla zapisu liczb szesnastkowych w języku C prefiksem 0x.

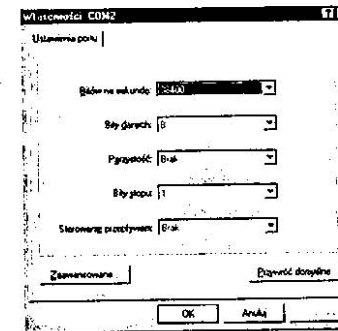
Transmisja będzie prowadzona z wykorzystaniem zaimplementowanego w mikrokontrolerze układu UART, pracującego z wykorzystaniem systemu przerwań. Dyrektywy umieszczone na początku programu służą do dołączenia niezbędnych plików w czasie kompilacji. Kolejne definicje pozwalają sparаметryzować konfigurowanie UART-u. Podana jest tam częstotliwość rezonatora kwarcowego, założona prędkość transmisji oraz wyliczony na tej podstawie parametr, jaki należy wpisać do rejestru UBRR. Pamiętajmy, że nie ma tu pełnej dowolności (**tablica 8.1**). Gdy nie jesteśmy pewni wyniku, warto obliczenia zweryfikować ręcznie. Następne linie programu, to deklaracje zmiennych globalnych, procedury obsługi przerwań i funkcje pomocnicze. Program główny rozpoczyna się, jak zwykle, od prawidłowego skonfigurowania portów mikrokontrolera, układów czasowych oraz UART-u. Port B został w całości ustawiony w trybie wyjściowym, tak by można było wykorzystywać diody LED do sygnalizowania różnych sytuacji. W praktyce dioda LED1 będzie pokazywała swoją intensywnością aktualnie ustawiony współczynnik wypełnienia przebiegu PWM (pełna jasność odpowiada wyłączeniu silnika, całkowite wygaszenie odpowiada obrotom maksymalnym), dioda LED2 będzie zapalana w chwilach, gdy na dłużej zostanie przytrzymany klawisz. Wykrycie takiego przypadku będzie skutkowało zwiększeniem kroku regulacji, który domyślnie ma wartość 1, co odpowiada największej możliwej rozdzielczości. Port D, z wyjątkiem linii PD1, skonfigurowano jako wejściowy. Linia PD1 jest wyjściem nadajnika, dlatego pracuje w trybie wyjściowym. Do rejestru UDDR jest wpisywana wartość 12, która umożliwia prowadzenie transmisji z prędkością 38400 b/s z błędem równym 0,2%, a więc dopuszczalnym. W fazie inicjującej włączane są przerwania od nadajnika

(RXCIE) i odbiornika (TXCIE), włączany jest również odbiornik (RXEN). Można powiedzieć, że odbiornik pozostanie „na nasłuchu”. Po skonfigurowaniu modulatora PWM (tak jak w ćwiczeniu 7.) jest przesyłana do komputera winietka zawierająca krótką instrukcję obsługi sterownika. Będzie ona wyświetlona na ekranie monitora komputera. Wszystkie informacje przesyłane do komputera są zapisane w tablicy `info[7]` w pamięci programu, jako pojedyncze linie tekstu. Wywołanie procedury `wyslijtekstROM(info[i])` powoduje więc wysłanie jednej linii, zakończonej znakami `\n` (LF – *Line Feed* – przejście do nowej linii) i `\r` (CR – *Carriage Return* – powrót karetki). W programie zastosowano dwie funkcje o podobnym działaniu, lecz pobierające dane z różnych typów pamięci. Wynikają one poniekąd z budowy mikrokontrolera. O ile do pobrania danej z pamięci RAM wystarczą rozkazy LD mikrokontrolera, to pobranie danej z pamięci programu wymaga użycia rozkazu LPM. W języku AVR-GCC pobieranie danych z RAM-u może być zrealizowane przy wykorzystaniu typowych wskaźników, tak jak w procedurze `wyslijtekst (*pfifosio)`. Do pobierania danych z pamięci programu służy makro `PRG_RDB` użyte w procedurze `wyslijtekstROM`.

Główna pętla programu to właściwie jedna instrukcja warunkowa sprawdzająca, czy został odebrany jakiś znak. Gdy rezultat porównania będzie pozytywny, zostaną podjęte działania polegające na zinterpretowaniu znaku. Zastosowano tu instrukcję `switch`, świetnie nadającą się do takiego celu. Porównuje ona odebrany znak, kolejno ze wszystkimi interesującymi nas przypadkami. Rozpoznanie klawiszy `[]`, `[]`, `[]` lub `[]` powoduje reakcję znaną z ćwiczenia 7. Rozpoznanie klawisza `[]` oznacza, że do komputera trzeba będzie przesłać aktualne parametry modulatora PWM. Wysyłany jest więc tekst informacyjny „Aktualne parametry PWM:” zawarty w `info[6]`, a bezpośrednio po nim prefiks `0x`. Następnie, przy wykorzystaniu standardowej funkcji `itoa` jest dokonywana konwersja aktualnego stanu rejestrów modulatora PWM, czyli liczby całkowitej (`int`) na łańcuch tekstowy. Dzięki temu liczba ta może być przesłana do komputera w postaci znaków ASCII i bezpośrednio wyświetlona na monitorze komputera. W ostatnim parametrze funkcji podaje się podstawę obliczeniową. Liczba 16 oznacza, że łańcuch tekstowy zwracany przez procedurę `itoa` będzie reprezentował liczbę szesnastkową.

Transmisja jest prowadzona przy wykorzystaniu systemu przerwań. Odbiornik zgłasza fakt odebrania znaku ustawiając flagę `RXC`. Powoduje to automatyczne wywołanie funkcji `SIG_UART_RECV`, w której jest zapamiętywany odebrany znak (w zmiennej `komenda`) oraz ustawiana jest flaga odebrania znaku. Nadajnik zgłasza przerwanie ustawiając flagę `TXC`. Robi to po wysłaniu znaku z rejestru nadajnika. Aby to jednak nastąpiło, pierwszy znak musi być wpisany

„ręcznie”. Dlatego w funkcjach `wyslijtekstROM` i `wyslijtekst` umieszczono instrukcje `UDR=`. Wcześniej jest włączany nadajnik, poprzez ustawienie bitu `TXEN` w rejestrze `UCR`. Kolejne znaki będą wysyłane z procedury obsługi przerwania `TXC – SIG_UART_TRANS`. W procedurze tej sprawdzane jest ponadto, czy wysłano już wszystkie znaki z bufora. Każdy łańcuch tekstowy kończy się znakiem o wartości „0”. Jeśli fakt ten zostanie wykryty, to nadajnik zostanie wyłączony poprzez wyzerowanie bitu `TXEN` w rejestrze `UCR`.



Rys. 14.21. Konfiguracja portu COM komputera na potrzeby ćwiczenia 8

Do prowadzenia łączności od strony komputera może być wykorzystany dowolny program terminalowy, np. HyperTerminal. Przed nawiązaniem połączenia, port komunikacyjny powinien być skonfigurowany jak na rysunku 14.21. Jeśli konfigurację przeprowadzono prawidłowo, to po połączeniu płytki ZL1AVR z komputerem i włączeniu jej zasilania lub wyzerowaniu mikrokontrolera, na ekranie powinny się pokazać kolejne linie przesłane z ZL1AVR do komputera PC. Następnie naciskając odpowiednie klawisze na klawiaturze komputera zmieniamy obroty silnika lub żądamy przesłania aktualnych parametrów regulacji.

Program realizujący przedstawione zadanie znajduje się na listingu 14.9. Należy skompilować program `cwicz8.c` i zaprogramować mikrokontroler z zestawu ZL1AVR plikiem wynikowym `cwicz8.hex`.

Konfiguracja płytki ZL1AVR:

- zworka J3 w pozycji 1-2, zworka J4 w pozycji 2-3 – dołączony oscylator wewnętrzny,
- zworka JP1 zwarta – globalne włączenie diod LED,
- zworki JP4 i JP5 całkowicie rozłączone,
- zworka ZW_PORTB zwarta w pozycji 13-14,
- połączyć kabelkiem nóżkę 9 ZW_PORTB z nóżką 16 ZW_PORTB,
- nóżkę 5 łączówki PORTB połączyć z rezystorem RB interfejsu przygotowanego wcześniej na uniwersalnym polu montażowym (rysunek 14.17),
- zworki JP8 i JP9 rozwarte,
- położenie pozostałych zwojek nieistotne (np. rozłączone),
- połączyć płytkę ZL1AVR z komputerem poprzez port szeregowy i uruchomić odpowiednio skonfigurowany program terminalowy, np. „Hyperterminal”.

List. 14.9. Program do ćwiczenia 8

```

/*****
/* Ćwiczenie 8 - Zdalna regulacja obrotów silnika DC z komputera PC */
/*      Modułacja PWM przy użyciu timer1a      */
/*      Wykorzystanie UART-u do transmisji z komputerem */
/* J.D. '2003 */
*****/

#include <io.h>
#include <progmem.h>
#include <stdlib.h>
#include <interrupt.h>
#include <signal.h>

#define FCPU 8000000 //częstotliwość oscylatora CPU
#define VUART 38400 //prędkość transmisji [b/s]
#define VUBRR FCPU/(VUART*16)-1 //wpis do UBRR dla VUART

unsigned char romram; //romram=1 => dane z pamięci programu
//romram=0 => dane z RAM-u
char *pfifosio; //wskaźnik na kolejkę UART-u
unsigned char volatile fodbznak=0; //flaga: "odebrano znak"
char komenda; //odebrana komenda z PC-ta
char *fifosio[]; //wskaźnik na kolejkę UART-u

SIGNAL(SIG_UART_RECV) //procedura obsługi odbiornika UART-u
{
    komenda=UDR; //zapamiętaj odebraną komendę
    fodbznak=1; //ustaw flagę odebrania znaku
}

SIGNAL(SIG_UART_TRANS) //procedura obsługi nadajnika UART
//wywoływana po wysłaniu znaku
{
    char znak;

    if(romram) //skąd pobierać dane?
    {
        znak=PRG_RDB(pfifosio++); //pobierz daną z pamięci programu
    }
    else
    {
        znak=*pfifosio++; //pobierz dane z pamięci RAM
    }
    if(znak!=0) //czy koniec pobierania danych?
    {
        UDR=znak; //nie, wyślij znak pobrany z kolejki
    }
    else
    {
        cbi(UCR, TXEN); //tak, wyłącz nadajnik
    }
}

void czekaj(unsigned long zt) //funkcja opóźnienia
{
    #define tau 10.38
    unsigned char zt1;
    for(;zt>0;zt--)
    {

```

```

        for(zt1=255;zt1!=0;zt1--);
    }
}

void wyslijtekstROM(char *tekst) //wysyłanie danych z pamięci programu
{
    romram=1; //dane będą z pamięci programu
    pfifosio=tekst; //ustaw wskaźnik na dane do wysłania
    sbi(UCR, TXEN); //włącz nadajnik
    UDR=PRG_RDB(pfifosio++); //wyślij pierwszy znak, pozostałe będą
    //pobierane w procedurze obsługi
    //przerwania TXC
}

void wyslijtekst(char *tekst) //wysyłanie danych z pamięci programu
{
    romram=0; //dane będą z pamięci danych
    pfifosio=tekst; //ustaw wskaźnik na dane do wysłania
    sbi(UCR, TXEN); //włącz nadajnik
    UDR=*pfifosio++; //wyślij pierwszy znak, pozostałe będą
    //pobierane w procedurze obsługi
    //przerwania TXC
}

int main(void)
{
    unsigned char i;
    unsigned char volatile licznikkl=0; //zmienna wykorzystywana do pomiaru
    //czasu naciśnięcia przycisków

    char volatile przyrost=1; //przyrost zmiany współczynnika
    //wypełnienia sygnału PWM

    //tablica komunikatów do wysłania
    char *info[7]={
        PSTR("\n\rRegulator obrotów silnika DC\n\r"),
        PSTR(" - zmniejszanie obrotów\n\r"),
        PSTR(" - zwiększanie obrotów\n\r"),
        PSTR("0 - zatrzymanie silnika\n\r"),
        PSTR("1 - start z max. obrotami\n\r"),
        PSTR("N - podaj aktualne parametry sterownika\n\r\n"),
        PSTR("\n\rAktualne parametry PWM:")
    };

    union //unia pozwala na bajtowy dostęp do
    //zmiennej int
    {
        unsigned int pwm;
        unsigned char pwmcl[2];
    }volatile upwm; //aktualny współczynnik wypełnienia
    //sygnału PWM

    DDRB=0xff; //PORTB - wy
    PORTB=0xff;
    DDRD=0x02; //PD1 - wy (RXD), pozostałe we
    PORTD=0x02; //podciągania wejścia PD1 (RXD)
    UBRR=VUBRR; //ustaw prędkość transmisji
    UCR=1<<RXCIE | 1<<TXCIE | 1<<RXEN; //zezwoleń na przerwanie od odbiornika
    //i nadajnika, zezwolenie na odbiór
    //PWM 10-bitowy

    TCCR1A=0x83;

```

```

//zerowanie OCR1 po spełnieniu warunku
//równości podczas liczenia w górę,
//ustawiane podczas liczenia w dół
TCCR1B=0x01; //preskaler=3, co przy 10-bit PWM daje
//Fwy=ok. 61Hz @8MHz
TCNT1L=0x00; //wstępne ustawienie licznika 1
TCNT1H=0x00;
upwm.pwm=0x3ff; //początkowo silnik włączony, wartość
//TOP odpowiada wysokiemu
//poziomowi na wyjściu OCR1 (PB3)
OCR1H=upwm.pwm[1]; //wpisz aktualnie ustawiony
//współczynnik do rejestrów
OCR1L=upwm.pwm[0]; //OCR1 timer1
sei(); //włącz przerwania

for(i=0;i<5;i++) //wyślij winietkę
{
    wyslijtekstROM(info[i]); //wysłanie pojedynczej linii tekstu
    while(bit_is_set(UCR, TXEN)); //trzeba zaczekać, aż zostanie wysłana
    //do końca
}

while(1) //główna pętla programu
{
    if(fodbnzak) //czy odebrano jakiś znak?
    {
        fodbnzak=0; //tak
        switch (komenda) //interpretacja komendy i wykonanie
        //odpowiedniej akcji
        {
            case '.': //odebrano "." - zwiększ prędkość
                upwm.pwm+=przyrost; //zwiększ PWM
                if(upwm.pwm>0x3ff)
                {upwm.pwm=0x3ff; //jeśli przekroczono wartość TOP, to
                //ustaw TOP
                }
                czekaj(150*tau); //eliminacja powtórnej interpretacji
                //naciśnięcia przycisku
                licznik1++; //mierz długość naciśnięcia przycisku
                break;
            case ',': //odebrano "," - zmniejsz prędkość
                upwm.pwm-=przyrost; //zmniejsz PWM
                if(upwm.pwm<0) //odpowiada upwm.pwm<0
                {upwm.pwm=0; //jeśli przekroczono wartość zero, to
                //ustaw zero
                }
                czekaj(150*tau); //eliminacja powtórnej interpretacji
                //naciśnięcia przycisku
                licznik1++; //mierz długość naciśnięcia przycisku
                break;
            case '0': //odebrano "0" - zatrzymaj silnik
                upwm.pwm=0; //silnik STOP
                break;
            case '1': //odebrano "1" - ustaw max obroty
                //silnika
                upwm.pwm=0x3ff; //silnik na MAX
                break;
            case 'n':
            case 'N':
                wyslijtekstROM(info[6]);
                //wysłanie pojedynczej linii tekstu

```

```

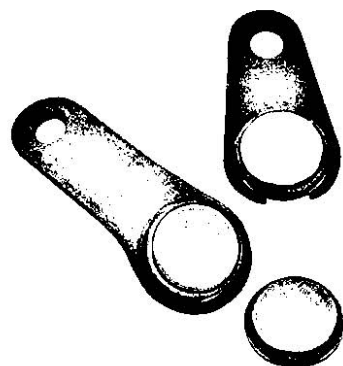
while(bit_is_set(UCR, TXEN));
//trzeba zaczekać, aż zostanie wysłana
//do końca
wyslijtekst("0x"); //wyślij prefiks dla liczb
//heksadecymalnych
while(bit_is_set(UCR, TXEN));
//trzeba zaczekać, aż zostanie wysłana
//do końca
itoa(upwm.pwm, fifosio, 16);
//konwersja liczby int (hex)
//na łańcuch znakowy
wyslijtekst(fifosio);
//wyślij aktualną wartość PWM do PC-ta
while(bit_is_set(UCR, TXEN));
//trzeba zaczekać, aż zostanie
//wysłana do końca
break;
}
if(licznik1>6)
{
    przyrost=16; //wykryto długie naciśnięcie klawisza,
    //zwiększ krok regulacji
    licznik1=6; //dalej już nie zwiększaj kroku
    cbi(PORTB, 1); //zapal diodę LED1
}
OCR1H=upwm.pwm[1]; //wpisz aktualnie ustawiony
//współczynnik do rejestrów
OCR1L=upwm.pwm[0]; //OCR1 timer1
}
else
{
    //jeśli cisza na linii, to ustaw
    //parametry spoczynkowe
    licznik1=0;
    przyrost=1;
    sbi(PORTB, 1); //zgaś diodę LED1
}
}
}

```

14.2.9. Ćwiczenie 9

Obsługa interfejsu 1-Wire. Odczyt pastylki identyfikacyjnej Dallas – DS1990A. Obsługa wyświetlacza LCD 16×2

Liczba różnorodnych interfejsów funkcjonujących w sprzęcie elektronicznym jest trudna do oszacowania. Wiele z nich zaprojektowano specjalnie na potrzeby bardzo konkretnych zastosowań i nie znajduje zastosowania w sprzęcie powszechnym. Inne, dzięki swoim walorom, stały się bardzo popularne i są chętnie wykorzystywane przez różnych producentów. Trzeba wiedzieć, że większość z nich jest chroniona patentami, co nie sprzyja ich upowszechnianiu. Być może to właśnie ten fakt stanowi motywację do podejmowania



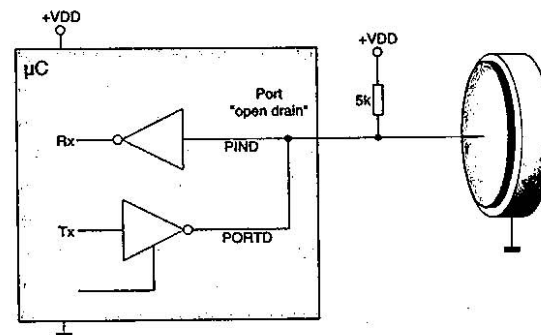
Fot. 14.22. Wygląd układów iButton

prac nad tworzeniem własnych rozwiązań przez większych potentatów światowych. Przykładem może być bardzo dynamicznie rozwijająca się jeszcze nie tak dawno temu firma Dallas Semiconductors, która może się szczycić wieloma bardzo oryginalnymi i udanymi opracowaniami. Właściwie należałoby powiedzieć, że mogła się poszczycić, gdyż – jak to często bywa w skomplikowanych współczesnych realiach – firmę tę przejął inny potentat – firma Maxim, w wyniku czego powstał prawdziwy gigant Maxim-Dallas. Jednym ze sztandarowych

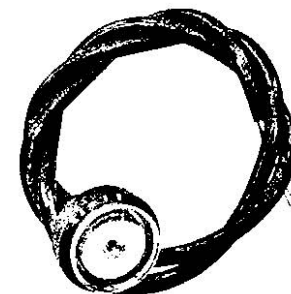
produktów opracowanych przez Dallasa są pastylki identyfikacyjne iButton, wykorzystujące jednoprzewodowy interfejs komunikacyjny 1-Wire, będący opracowaniem własnym firmy. „Jednoprzewodowy” oznacza *de facto* dwa przewody. Jeden to dwukierunkowa linia transmisyjna, drugi to oczywiście przewód odniesienia (masa). Na uwagę zasługuje fakt, że pastylki nie wymagają żadnego zasilania, czerpiąc całą energię z transmitowanych sygnałów. Mylący może więc być ich wygląd (fotografia 14.22), do złudzenia przypominający pojedyncze ogniwo akumulatora.

W ofercie Maxima-Dallasa znajduje się wiele różnych odmian pastylek. Do ćwiczenia 9. wybrano jedną z najbardziej popularnych, oznaczoną symbolem DS1990A. Jej zasadniczą część stanowi pamięć ROM, w której na etapie produkcyjnym zapisano metodami laserowymi niepowtarzalny w każdym egzemplarzu numer seryjny. Jego 48-bitowa długość ma zniechęcać potencjalnych „włamywaczy” do łamania kodu, co jest metodą skuteczną przy założeniu, że pastylka będzie pieczołowicie chroniona przez jej właściciela, tak jak dzieje się to z tradycyjnymi kluczami. Sporządzenie kopii pastylki w przypadku dostania się jej w niepowołane ręce nie stanowi jednak problemu, protokół transmisji jest bowiem jawny. Dzięki temu zresztą nie są wymagane specjalne czytniki, a każdy użytkownik może we własnym zakresie wykonać odpowiedni układ. Sposób dołączenia pastylki do mikrokontrolera jest banalny, przedstawiono go na rysunku 14.23. Dla zapewnienia komfortu pracy można zakupić specjalne gniazdo (fotografia 14.24), do którego przykładają się pastylkę. Do eksperymentów wystarczą dwa zwykłe przewody.

Aby odczytać dane z pastylki, trzeba przyłożyć ją do dwóch punktów, zgodnie z rysunkiem 14.23. Ci spośród Czytelników, którzy nie mieli wcześniej do czynienia z układami DS1990, będą zapewne zaskoczeni sposobem ich



Rys. 14.23. Sposób dołączenia pastylki iButton do portu mikrokontrolera



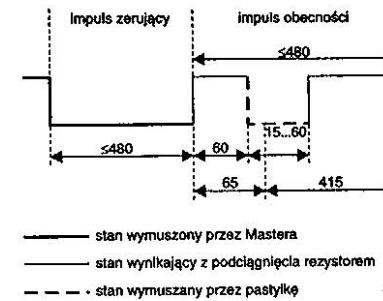
Fot. 14.24. Korzystanie z pastylek iButton ułatwiają specjalne gniazda, często wyposażone w diody LED służące do sygnalizowania stanu współpracującego sterownika

działania. Transmisja wszystkich danych trwa zaledwie około 5 ms, wystarczy więc dosłownie musnąć pastylką styki czytnika. Aby interfejs 1-Wire pracował poprawnie, port mikrokontrolera musi być dwukierunkowy, a jego wyjście powinno być typu *open drain*, co jest na ogół łatwe do spełnienia. W przypadku płytki ZL1AVR, do realizacji interfejsu 1-Wire wykorzystano linię PD3. Sterowanie tą linią będzie trochę nietypowe, co wynika ze specyfiki interfejsu. Port D będzie w całości skonfigurowany jako wejściowy bez podciągania. Do jego rejestru wyjściowego (PORTD) zostanie na stałe wpisana wartość „0” (linia Tx na rysunku 14.23). Dane będą odbierane z pastylki poprzez bezpośredni odczyt rejestru PIND. Wysłanie bitu o wartości „1” będzie się wiązało z ustawieniem portu w tryb wejściowy. Nie należy się temu dziwić – stan wysoki na linii interfejsu jest wymuszany zewnętrznym rezystorem podciągającym (R4 na płytce ZL1AVR). Wysłanie bitu o wartości „0” będzie polegało na przełączeniu portu w tryb wyjściowy. Wówczas zapisane wcześniej „0” do PORTD spowoduje ustawienie linii interfejsu w stan niski.

Dostęp do informacji bitowych z poziomu języka AVR-GCC jest dość niewygodny. Wykorzystuje się do tego celu np. makra `sbi(rejestr, bit)` lub `cbi(rejestr, bit)`. Pisząc program nie zawsze pamiętamy, w którym rejestrze jest umieszczony dany bit. Wygodniejsze byłoby ustawianie lub zerowanie bitów poprzez bezpośrednie użycie ich symbolicznej nazwy, np. `lcd_e=0`. Aby było to możliwe, na początku programu należy wykonać kilka dodatkowych czynności. Po pierwsze, można dla wygody zdefiniować specjalną strukturę, która będzie później używana do deklarowania bitów. W programie do ćwiczenia 9. jest to struktura `pole_bitowe`. Następnie zdefiniujemy makro `DAJ_BIT` przypisujące symboliczną nazwę do fizyczne-

go adresu w przestrzeni adresowej mikrokontrolera. Makro to korzysta z przestrzeni adresowej pamięci RAM, w której jak pamiętamy z wcześniejszych rozdziałów zawierają się również adresy układów wejścia-wyjścia. W standardowych plikach nagłówkowych (*.h) porty mikrokontrolera są jednak określone w przestrzeni wejścia-wyjścia, stąd konieczność użycia dodatkowej definicji używającej adresów pamięci RAM. Przykładowo dla rejestru PORTB należy w programie zamieścić definicję `#define _PORTB 0x38`. Znak podkreślenia zabezpiecza przed podwójnym zdefiniowaniem tego samego symbolu. Na koniec pozostaje już tylko przypisanie nazw symbolicznych odpowiednim bitom poszczególnych rejestrów. Na przykład zapis `#define pastylka_we DAJ_BIT(_PIND).bit3` oznacza, że od tej chwili nazwa `pastylka_we` będzie oznaczała bit 3 portu PIND. Do sprawdzania stanu linii interfejsu 1-Wire zamiast stosować zapis np. `if(bit_is_set(PIND,3))` wystarczy zapis: `if(pastylka_we)`, co w języku C jest równoznaczne z `if(pastylka_we==1)`. Nie sposób się nie zgodzić z tym, że zrozumiałość drugiej metody jest dużo lepsza. Dla podniesienia czytelności programu zastosowano jeszcze jedną definicję związaną z wysterowywaniem linii 1-Wire. Jak już było wcześniej powiedziane, wystawienie stanu niskiego, wiąże się z wpisaniem „1” do portu DDRB, analogicznie wpisanie „0” powoduje ustawienie linii w stan wysoki. Port związany z linią 1-Wire został nazwany `pastylka_wy` dyrektywą `#define pastylka_wy DAJ_BIT(DDRD).bit3`, a sterowanie nim odbywa się poprzez wpisanie wartości zdefiniowanych jako: `#define stan_0 1` oraz `#define stan_1 0`. Dzięki powyższym zabiegom ustawienie linii 1-Wire np. w stan niski odbywa się w bardzo intuicyjny sposób: `pastylka_wy=stan_0`.

No dobrze, ale jak zrealizować dwukierunkową transmisję danych za pomocą tylko jednej linii? Pomysł Dallasa jest wręcz genialny. W urządzeniu wykorzystującym interfejs 1-Wire zawsze występuje jeden układ nadrzędny (*Master*) i co najmniej jeden układ podrzędny (*Slave*). Możliwość jednoczesnego dołączenia kilku układów podrzędnych wydaje się wręcz nieprawdopodobna, ale dzięki odpowiednim zabiegom programowym ich obsługa jest jak najbardziej możliwa. Zagadnienia te nie będą jednak omawiane w tej książce. Odpowiednie materiały można znaleźć w notach aplikacyjnych Maxima-Dallasa. W stanie spoczynkowym wyjściowy port mikrokontrolera pozostaje w stanie wysokiej impedancji, co odpowiada wysokiemu poziomowi wymuszanemu przez rezystor podciągający. Transmisja jest prowadzona poprzez odpowiednio określone tzw. szczeliny czasowe (*slots*). Wyróżnia się cztery ich rodzaje. Na początku *Master* generuje szczelinę inicjującą, mającą na celu wykrycie, czy do interfejsu jest dołączony jakikolwiek układ *Slave*. Szczelina ta rozpoczyna się impulsem zerującym (wystawienie stanu niskiego na

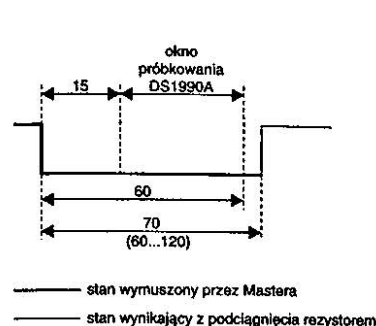


Rys. 14.25. Transmisję danych pomiędzy układem iButton i mikrokontrolerem rozpoczyna sekwencja zerowania (czasy w mikrosekundach)



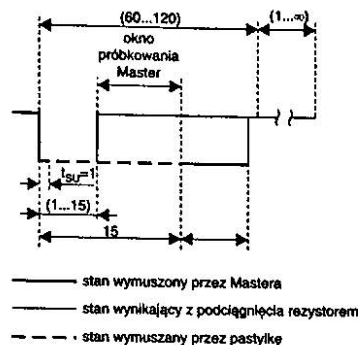
Rys. 14.26. Okno wpisu „1” do rejestru wejściowego układu iButton (czasy w mikrosekundach)

linię 1-Wire), trwającym minimum 480 µs (rysunek 14.25). Na rysunkach 14.26...14.28 stany wymuszane przez *Mastera* zaznaczono grubą linią ciągłą, stany wymuszane przez układ *Slave* grubą linią przerywaną, zaś stan wysoki będący rezultatem podciągania rezystorem zewnętrznym zaznaczony jest ciągłą linią cienką. Do uzyskiwania opóźnienia jest wykorzystywana funkcja `czekaj(czas)` znana z wcześniejszych ćwiczeń. Wartość 5 parametru `czas` odpowiada opóźnieniu równemu właśnie ok. 480 µs (dla kwarcu 8 MHz). Ewentualna zmiana częstotliwości rezonatora będzie wymagała przeliczenia wszystkich czasów. Po zakończeniu impulsu zerującego, linia interfejsu jest zwalniana poprzez wystawienie stanu „1”. Teraz jest odmierzany czas równy ok. 65 µs. Jest on potrzebny pastylce do wystawienia tzw. impulsu obecności. Niestety funkcja `czekaj` nie gwarantuje odpowiednio wysokiej dokładności, z tego względu do odmierzania krótkich interwałów będzie używana znacznie dokładniejsza funkcja `czekaj_lw(czas)`. Opóźnienie jest w tym przypadku równe $t = (5 + 5 \cdot \tau_{1w}) \cdot T$, gdzie τ_{1w} jest parametrem funkcji, a T to okres oscylatora systemowego (125 ns przy 8 MHz). Po odczekaniu 65 µs (min. 60 + 5 µs rezerwy) mikrokontroler sprawdza, czy linia 1-Wire znajduje się w stanie niskim. Jeśli tak, może to oznaczać, że zgłosiła się pastylka, ale może być też powodem zwykłego zwarcia linii. Sytuację tę trzeba będzie rozpoznać w dalszej kolejności. Tymczasem fakt wykrycia niskiego stanu na linii interfejsu jest zapamiętany przez inkrementację zmiennej pomocniczej `zp`. Po kolejnym odczekaniu ok. 415 µs do końca szczeliny (czas ten wynika z protokołu komunikacyjnego interfejsu 1-Wire) mikrokontroler ponownie bada stan linii. Przed zakończeniem szczeliny pastylka powinna zwolnić linię, co odpowiada stanowi wysokiemu będącemu wynikiem podciągania rezystorem R4. Wykrycie w tym momencie stanu nis-

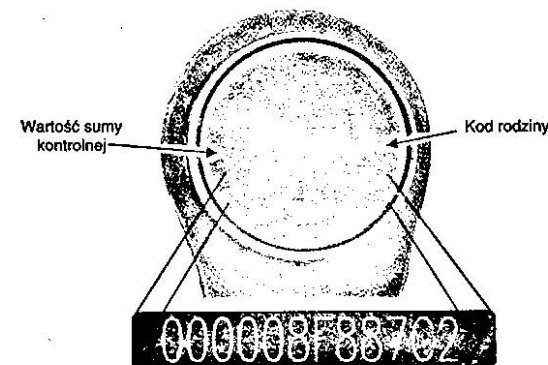


Rys. 14.27. Okno wpisu „0” do rejestru wyjściowego układu iButton (czasy w mikrosekundach)

kiego najprawdopodobniej będzie efektem zwarcia – zmienna *zp* jest po raz kolejny inkrementowana. Po zakończeniu fazy inicjującej pastylkę mikrokontroler rozpoznaje, która z powyższych sytuacji wystąpiła na linii interfejsu. Jeśli zmienna *zp* ma wartość 1, to z dużym prawdopodobieństwem można przypuszczać, że została rozpoznana obecność pastylki i nastąpi teraz próba odczytania z niej danych. Aby to uczynić, mikrokontroler musi przesłać do pastylki rozkaz *Czytaj ROM* o kodzie 0x33 (lub 0x0f). Służy do tego celu uniwersalna funkcja `zapisz1w(dana)`, której parametrem będzie w tym przypadku wartość 0x33. Funkcja ta wyluskuje kolejne bity danej, począwszy od najmniej znaczącego, i wywołuje funkcję `slot1w_zap(znak)`, której zadaniem jest wygenerowanie odpowiedniej szczeliny nadawczej. W zależności od wartości transmitowanego bitu, szczelina nadawcza przybiera odpowiednią postać. W przypadku bitu „1” mamy do czynienia ze szczeliną *Zapisz jeden* (rysunek 14.26). Mikrokontroler ustawia linię interfejsu w stan niski, odczekuje 11 μ s, po czym zwalnia ją wpisując „1”. Po 15 μ s od chwili wyzerowania, układ odbiorczy pastylki rozpoczyna próbkowanie linii, które może trwać do 60 μ s licząc od początku szczeliny. Mikrokontroler odczekuje jeszcze 60 μ s do końca slotu, po czym może kontynuować pracę. W przypadku transmisji bitu o wartości „0” mamy do czynienia ze szczeliną *Zapisz zero* (rysunek 14.27). Różni się ona od poprzedniej czasem przytrzymania linii 1-Wire w stanie niskim, który powinien zawierać się między 60 a 120 μ s. W programie odczekuje się 70 μ s. Po przesłaniu rozkazu *Czytaj ROM* mikrokontroler rozpoczyna odbieranie danych z pastylki. Służy do tego funkcja `czytaj1w()`. W wyniku jej działania do 8-elementowego bufora `bufor1w` zostaną zapisane kolejne bajty odebrane z pastylki. Funkcja `czytaj1w()` kompletuje poszczególne bity w 1-bajtowe dane, wykorzystując funkcję



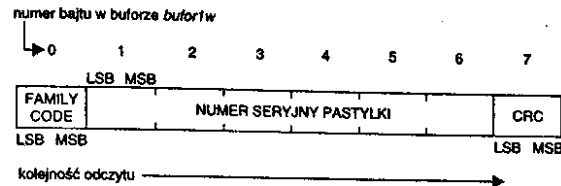
Rys. 14.28. Odczyt stanu wyjścia danych układu iButton przez mikrokontroler (czasy w mikrosekundach)



Fot. 14.29. Niepowtarzalny numer seryjny układu iButton jest wypalony laserem na obudowie układu

`slot1w_czyt()`. Szczelinę odczytu przedstawiono na rysunku 14.28. Jej początek jest sygnalizowany wyzerowaniem przez mikrokontroler linii interfejsu na czas t_{su} (1 μ s), po którym jest ona zwalniana. W tym momencie pastylka wystawia stan odpowiadający nadawanemu przez nią bitowi. Mikrokontroler odczekuje jeszcze ok. 14 μ s do momentu, w którym będzie mógł próbować linię. Po odczytaniu bitu, funkcja `slot1w_czyt()` zwraca wynik do wywołującej ją `czytaj1w()`, gdzie jak już wiadomo następuje kompletowanie ramki i zapis danej do bufora. Dane odebrane z pastylki to: identyfikator rodziny układów, tzw. *Family Code* (w przypadku układów DS1990A zawsze będzie miał wartość 1), 6-bajtowy, niepowtarzalny numer seryjny pastylki (jest on również wygrawerowany na obudowie – fotografia 14.29) oraz bajt CRC stanowiący kontrolę poprawności odczytu. Procedura rozpoznająca obecność pastylki w czytniku czasami może mylnie zinterpretować jej dołączenie, np. w wyniku zakłócenia linii interfejsu. Może się również zdarzyć, że po prawidłowym rozpoznaniu obecności pastylki, dane nie zostaną z niej odczytane bezbłędnie, np. w wyniku braku odpowiedniego kontaktu elektrycznego.

Zastosowania do jakich zostały przewidziane układy DS1990A narzucają, aby pewność odczytu danych była jak największa. W tym celu, po odebraniu kompletu danych jest obliczana suma kontrolna CRC (służy do tego celu funkcja `licz_CRC(dana, crc)`). Jak widać na rysunku 14.30, wartość sumy kontrolnej jest zapisana w pamięci ROM pastylki i przesyłana wraz z resztą danych. Do obliczeń jest wykorzystywany wielomian generujący $x^8 + x^5 + x^4 + 1$. Funkcja `licz_CRC` jest implementacją w języku C procedury assemblerowej opublikowanej w nocie katalogowej układu DS1990A. Suma kontrolna jest liczona w programie ze wszystkich 8 odebranych bajtów. Wynik obliczeń w przypadku poprawnej transmisji powinien dać war-



Rys. 14.30. Mapa pamięci układu DS1990A

tość 0. Każda inna wartość będzie oznaczała wystąpienie błędu podczas transmisji i dane takie powinny być zignorowane. Jest wówczas wyświetlany odpowiedni komunikat. Sumę kontrolną CRC można liczyć również w inny sposób. Obliczenia mogą być prowadzone dla siedmiu pierwszych bajtów, a wynik powinien odpowiadać bajtowi ósmemu. Obie metody są równoważne, wybór padł na pierwszą, nieco łatwiejszą w realizacji praktycznej.

Po otrzymaniu prawidłowego wyniku na wyświetlaczu jest wyświetlany komunikat o rozpoznaniu pastylki wraz z jej numerem seryjnym i kodem rodziny. Dane są wyświetlane od tyłu, począwszy od szóstego elementu tablicy bufor1w tak, żeby odpowiadały rzeczywistej kolejności (zgodnej z numerem wygrawerowanym na obudowie). Ostatnie dwie cyfry zawierają identyfikator rodziny. Do wyświetlania danych wykorzystano standardowe funkcje biblioteczne: `utoa` – zamieniająca liczbę całkowitą, w tym przypadku szesnastkową na znaki ASCII oraz `toupper`, która zamienia małe litery na wielkie. Obecność pastylki jest dodatkowo sygnalizowana zapaleniem diody LED1. Program wykrywa brak pastylki w czytniku, zwarcie linii interfejsu, przyłożenie pastylki i błąd odczytu.

Program realizujący przedstawione zadanie znajduje się na listingu 14.10. Należy skompilować program `cwicz9.c` i zaprogramować mikrokontroler z zestawu ZL1AVR plikiem wynikowym `cwicz9.hex`.

Konfiguracja płytki ZL1AVR:

- zworka J3 w pozycji 1-2, zworka J4 w pozycji 2-3 – dołączony oscylator wewnętrzny,
- zworka JP1 rozwarta – globalne wyłączenie diod LED,
- zworka JP5 całkowicie rozłączona,
- zworki ZW_PORTB zwarte w pozycjach 1-2, 3-4, 5-6, 7-8, 9-10, 11-12,
- zworka JP14 zwarta,
- do łączówki JP11 dołączone przewody czytnika układu DS1990A (masa na górnym styku),
- położenie pozostałych zworek nieistotne (np. rozłączone).

List. 14.10. Program do ćwiczenia 9

```

/*****
/* Ćwiczenie 9 - Obsługa interfejsu 1-Wire - odczyt pastylki DS1990A */
/* Obsługa wyświetlacza LCD 2x16 */
/* J.D. '2003 */
*****/

#include <io2313.h>
#include <progmem.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

// Poniższe definicje służą do realizacji wygodnego dostępu bitowego
typedef struct _bit_struct
{
    unsigned char bit0: 1;
    unsigned char bit1: 1;
    unsigned char bit2: 1;
    unsigned char bit3: 1;
    unsigned char bit4: 1;
    unsigned char bit5: 1;
    unsigned char bit6: 1;
    unsigned char bit7: 1;
}pole_bitowe;

#define DAJ_BIT(adr) (*((volatile pole_bitowe*) (adr)))
#define _PORTB 0x38
#define _PINE 0x36
#define _PORTD 0x32
#define _DDRD 0x31
#define _PIND 0x30
#define pastylka_we DAJ_BIT(_PIND).bit3
#define pastylka_wy DAJ_BIT(_DDRD).bit3
#define lcd_rs DAJ_BIT(_PORTB).bit2
#define lcd_e DAJ_BIT(_PORTB).bit3
#define led0 DAJ_BIT(_PORTB).bit0

#define stan_0 1 //definicja stanu niskiego na linii 1-Wire
// "1" oznacza przełączenie portu w tryb
// wyjściowy
// port jest wcześniej wysterowany w stan
// niski
#define stan_1 0 //definicja stanu wysokiego na linii 1-Wire
// "0" oznacza przełączenie portu w tryb
// wejściowy
// stan wysoki jest wymuszany przez zewnętrzny
// rezystor podciągający
#define CR 0x0a //znak CR (przejsie do nowej linii)

char buflcd[4]; //robczy bufor wyświetlacza LCD
char *pbuflcd; //wskaźnik na bufor wyświetlacza
unsigned char bufor1w[8]; //bufor interfejsu 1-Wire (dane z pastylki)
unsigned char *pbufor1w; //wskaźnik na bufor danych z pastylki
unsigned char wiersz=0; //pozycja umieszczenia znaku na LCD
unsigned char kolumna=0; //pozycja umieszczenia znaku na LCD

void czekaj(unsigned long zt) //funkcja opóźnienia
{
    #define tau 10.38
    unsigned char zt1;

```

[illegible][illegible]

```

czekaj_lw(118);           //ok. 60us
return bit1w;
}

void slot1w_zap(unsigned char znak)
//slot zapisu do pastylki
{
if(znak)
{
//slot "1"
pastylka_wy=stan_0;       //inicjuj slot
czekaj_lw(20);            //ok. 11us
pastylka_wy=stan_1;       //podciąganie do "1" rezystorem
//zewnątrznym
czekaj_lw(119);           //60us do końca slotu
}
else
{
//slot "0"
pastylka_wy=stan_0;       //inicjuj slot
czekaj_lw(139);           //ok. 70us
pastylka_wy=stan_1;       //podciąganie do "1" rezystorem
//zewnątrznym
}
}

void zapisz1w(unsigned char rozkaz)
//transmisja 8-bitowego rozkazu
//do pastylki
{
unsigned char i;
for(i=0;i<8;i++)
{
slot1w_zap(rozkaz&0x01);  //wyślij bit do pastylki
rozkaz>>=1;
}
}

void czytaj1w(void)        //odczyt bajtu z pastylki
{
unsigned char i,j;
unsigned char dana;

pbufor1w=&bufor1w[0];     //dane będą umieszczone w buforze "bufor1w"
for(i=0;i<8;i++)          //czytaj "family code" (1bajt)
{
//i "registration number" (6 bajtów)
dana=0;                  //wstępne zerowanie danej
for(j=0x01;j!=0;j<=1)    //zmienna sterująca pętlą wskazuje
//jednocześnie aktualnie zapisywany bit
{
dana|=slot1w_czyt()?j:dana;
//czytaj kolejne bity (są one sumowane
//logicznie z odczytanymi wcześniej
}
*pbufor1w++=dana;         //po skompletowaniu zapisz odebrany bajt
//do bufora
}
}

void licz_CRC(char bajt,unsigned char *CRC)
//funkcja wyliczania CRC
//wielomian generujący jest równy:
//x^8 + x^5 + x^4 + 1

```

[illegible]

```

pisziled(0x28); //interfejs 4-bitowy, 2 linie, znak 5x7
pisziled(0x08); //wyłącz LCD, wyłącz kursor, wyłącz mruganie
pisziled(0x01); //czyść LCD
czekaj(1.64*tau); //wymagane dla instrukcji czyszczenia ekranu
//opóźnienie
pisziled(0x06); //bez przesuwania w prawo
pisziled(0x0c); //włącz LCD, bez kursora, bez mrugania

while(1) //główna pętla programu
{
    zp=0;
    pastylka_wy=stan_0; //sekwencja inicjująca pastylkę
    czekaj(5); //impuls inicjujący "0" ok. 480us
    pastylka_wy=stan_1; //zwolnij linię
    czekaj_lw(128); //po odczekaniu ok. 65us czekaj na impuls obecności
    if(pastylka_we==0)
    {
        zp++; //zapamiętaj fakt wykrycia impulsu obecności
        //pastylki
    }
    czekaj(4);
    czekaj_lw(56); //czekaj ok. 416us do zakończenia slotu
    //inicjującego
    if(pastylka_we==0)
    {
        zp++; //jeśli pozostaje w stanie niskim, to oznacza
        //zwykłe zwarcie
    }
    if(zp==1) //reakcja na rozpoznane sytuacje
    {
        //jeśli zp=1, to oznacza że pastylka zgłosiła się
        zapiszlw(0x33); //wyślij do pastylki kod rozkazu przesłania
        //numeru seryjnego
        czytajlw(); //czytaj numer seryjny pastylki (+ kod rodziny +
        //CRC)
        zp=0; //w zp będzie liczone CRC - wstępne zerowanie
        for(i=0;i<8;i++)
        {
            licz_CRC(buforlw[i],&zp);
            //licz CRC ze wszystkich bajtów odebranych
        }
        lcdxy(0,0);
        if(zp==0)
        {
            //zerowa wartość CRC oznacza prawidłowość danych
            led0=0; //zapal LED-a
            pisztekst(info[0]); //wyświetl komunikat na LCD o rozpoznaniu pastylki
            lcdxy(1,0);
            pbuforlw=&buforlw[7];
            for(i=6;i!=0xff;i--)
            {
                //wyświetlenie danych z pastylki
            }
            utoa((unsigned char)buforlw[i],buflcd,16);
            //przepisz buforlw do bufora wyświetlacza
            //z jednoczesną konwersją na ASCII
            pbuflcd=&buflcd[0];
            //ustaw wskaźnik bufora LCD na początek
            if(strlen(buflcd)<2)
            {
                //jeśli odebrano pojedynczą cyfrę, trzeba dopisać 0
            }
            piszznak('0'); //wyświetlenie zera wiodącego
        }
        while(*pbuflcd) //wyświetl zawartość bufora LCD

```

```

        {
            piszznak(toupper(*pbuflcd++));
            //wyświetl dane z konwersją na duże litery
        }
    }
    else
    {
        pisztekst(info[4]); //komunikat o błędzie obliczenia CRC
    }
    czekaj(3000*tau); //przytrzymaj wynik na wyświetlaczu przez ok. 3s
}
else
{
    led0=1; //gaś LED-a
    lcdxy(1,0);
    pisztekst(info[3]); //wyczyść dolną linię wyświetlacza
    lcdxy(0,0);
    if(zp==2)
    {
        //to było tylko zwarcie
        pisztekst(info[1]); //wyświetl komunikat o zwarcu
    }
    else
    {
        pisztekst(info[2]); //wyświetl komunikat o braku pastylki
    }
}
}
}
}

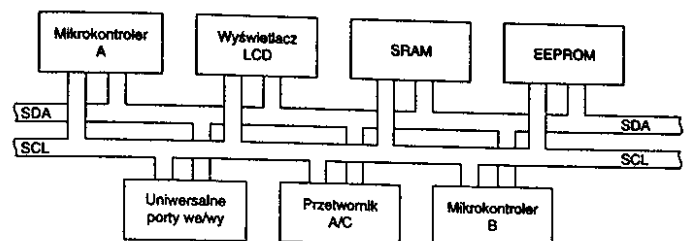
```

14.2.10. Ćwiczenie 10

Obsługa interfejsu I²C. Obsługa przerwania zewnętrznego. Wykorzystanie układu PCF8583 (RTC – Real Time Clock) do budowy zegara 24-godzinnego. Obsługa wyświetlacza LCD 16×2.

Kolejnym interfejsem powszechnie stosowanym w sprzęcie elektronicznym jest I²C (*Inter-Integrated Circuit*). Opracowano go i opatentowano w firmie Philips, a jego funkcjonalny odpowiednik występuje dość często także w wyrobach innych producentów, także pod innymi nazwami. Na przykład w układach Atmela nosi nazwę TWI (*Two Wire Interface*). W większości takich przypadków, poza bardzo niewielkimi różnicami pozwalającymi uniknąć opłat licencyjnych, poszczególne odmiany są zgodne ze standardem I²C.

Nazwa oryginału sugeruje, że interfejs I²C opracowano głównie w celu ujednolicenia lokalnej komunikacji pomiędzy układami scalonymi lub modułami funkcjonalnymi wchodzącymi w skład urządzenia. Łączy się więc za jego

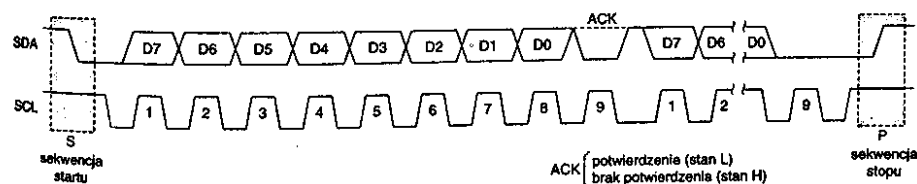


Rys. 14.31. Schemat blokowy typowego urządzenia wyposażonego w magistralę I²C

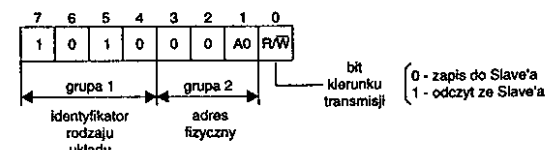
pomocą takie elementy jak: pamięci EEPROM i SRAM, potencjometry elektroniczne, zegary czasu rzeczywistego (RTC), czujniki temperatury, termostaty, przetworniki A/C i C/A itp. Interfejs I²C jest wykorzystywany do regulacji wzmocnienia wzmacniaczy, przestrajania tunerów, wyświetlania danych na wyświetlaczach LCD itp. – przykładów można mnożyć wiele. W jednej sieci I²C mogą współpracować ze sobą nadajniki, odbiorniki lub układy łączące obie te funkcje. Liczba takich elementów w jednej aplikacji z praktycznego punktu widzenia w zasadzie nie jest ograniczona. Nie można jednak mówić o pełnej swobodzie pod tym względem. Powodem tego są indywidualnie przypisywane adresy dla każdego układu. Na ogół są one – w pewnym zakresie – konfigurowalne, co daje wystarczającą swobodę konstruktorowi. Aby to zrozumieć trzeba dokładniej przyjrzeć się funkcjonowaniu interfejsu I²C. Jest on zastosowany w przykładowym urządzeniu, którego schemat blokowy jest przedstawiony na rysunku 14.31.

Ogólna zasada działania interfejsu I²C

Komunikacja poprzez interfejs I²C jest prowadzona za pomocą dwóch linii: SDA (*Serial Data*) i SCL (*Serial Clock*). Zastosowano tu synchroniczną transmisję szeregową. Kolejne bity danych są przekazywane linią SDA w takt sygnału zegarowego SCL (rysunek 14.32). Transmisję – podobnie jak to było w przypadku interfejsu 1-Wire – zarządza układ nadrzędny *Master*. To on generuje odpowiedni przebieg zegarowy synchronizujący transmisję.



Rys. 14.32. Przebieg typowej transmisji danych magistralą I²C



Rys. 14.33. Adres typowych układów z interfejsem I²C składa się z dwóch części

Konstrukcja interfejsu I²C pozwala na obecność kilku układów *Master* w jednej sieci. Jest to możliwe dzięki procedurom arbitrażu i synchronizacji przebiegów zegarowych. W większości przypadków będziemy mieli jednak do czynienia z sieciami z jednym układem pełniącym funkcję *Mastery*. Najczęściej będzie to mikrokontroler. Każdy układ wykorzystujący interfejs I²C pracujący jako *Slave* ma fabrycznie nadany adres identyfikacyjny. Nie jest to jednak adres niepowtarzalny dla każdego egzemplarza, jak w przypadku układów 1-Wire.

Zazwyczaj adres jest podzielony na dwie grupy (rysunek 14.33). Cztery najstarsze bity adresu (a7...a4) mają wartości na stałe ustalone w rejestrze adresowym. Przykładowo w układach PCF8583 mają one wartość 1010. Kolejne trzy bity (a3...a1) użytkownik może ustawiać w stan „0” lub „1” poprzez połączenie odpowiednich wejść adresowych do masy lub do plusa zasilania. To właśnie one gwarantują swobodę w konfigurowaniu adresu układu. W przypadku układów PCF8583, bity a2 i a1 są – jak starsza część adresu – zapisane na stałe. Mają one wartość „0”, a użytkownik może zmieniać dowolnie jedynie stan bitu a0. W rejestrze adresowym znajduje się ponadto bit R/W (a0), poprzez który *Master* informuje *Slave'a*, czy będzie coś z niego odczytywać (R/W=0), czy do niego zapisywać (R/W=1).

W celu zainicjowania transmisji *Master* musi wygenerować sekwencję startu – S (rysunek 14.32). Odpowiada jej opadające zbocze sygnału SDA, podczas gdy linia SCL jest utrzymywana w stanie wysokim. Od tego momentu dane na liniach SDA mogą się zmieniać tylko wtedy, gdy linia SCL będzie w stanie niskim. Dlatego też, po wygenerowaniu warunku startu linia SCL jest zerowana, a w chwilę później linia SDA przybiera stan zgodny z wartością najstarszego bitu transmitowanej danej. Po określonym czasie następuje pojedynczy impuls dodatni na linii SCL, powodujący zapisanie lub odczytanie bieżącego bitu. W analogiczny sposób są przesyłane kolejne bity. Specyficzna sytuacja występuje podczas dziewiątego impulsu zegarowego. W tym czasie zaadresowany układ, musi potwierdzić prawidłowe przyjęcie całego bajtu wyzerowaniem linii SDA (bit potwierdzenia – ACK). Odebranie bajtu może czasami powodować dłuższą zajętość odbiornika (spowodowaną np. obsługą

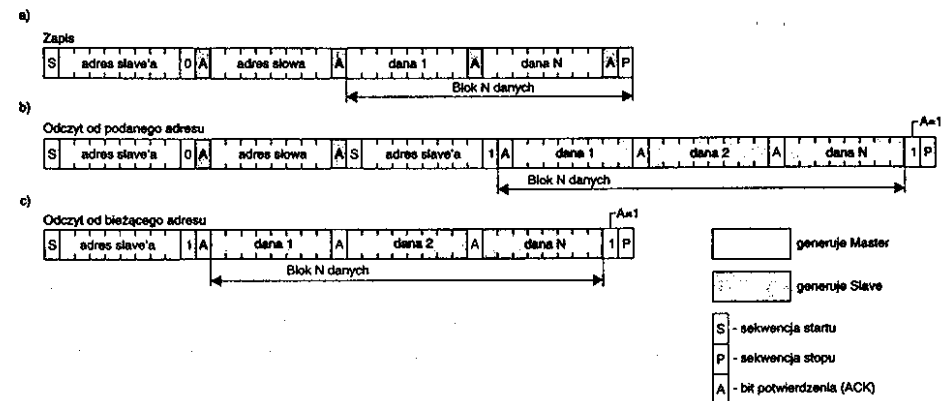
przerwania wynikającego ze skompletowania danej w odbiorniku). Z tego powodu odbiornik może po wystawieniu potwierdzenia wprowadzić *Mastera* w stan oczekiwania (*wait state*) poprzez przytrzymanie linii SCL w stanie niskim. Do momentu jej zwolnienia wszelkie operacje na magistrali I²C są zamrożone. Jeśli nie nastąpi potwierdzenie (bit ACK=1), transmisja powinna być zakończona. W podobny jak wyżej sposób przebiega przesyłanie kolejnych bajtów, aż do wystąpienia sekwencji stopu – P, czyli wygenerowania narastającego impulsu na linii SDA, podczas gdy linia SCL pozostaje w stanie wysokim (rysunek 14.32). Interwały czasowe pomiędzy poszczególnymi zboczami na liniach interfejsu wynikają z dopuszczalnej prędkości transmisji, które wynoszą: 100 kb/s w wersji standardowej i 400 kb/s w trybie *fast*.

Specyfikację I²C opracowano w latach 80. XX wieku i obecnie wersja podstawowa nie zawsze spełnia wymagania użytkowników. Wprowadzono więc 10-bitowe adresowanie, a maksymalne prędkości transmisji mogą dochodzić nawet do 3,4 Mb/s (*High-speed Mode*). Większość popularnych układów jest jednak zgodna z wersją podstawową i do niej ograniczymy się w dalszej części opisu, przyjmując ponadto, że w urządzeniu będzie pracował jeden układ *Master*.

Przesyłanie danych magistralą I²C

Specyfikacja interfejsu I²C szczegółowo określa zasady przebiegu transmisji tak na poziomie bitowym, jak i bajtowym. Generowanie poszczególnych bitów opisano wcześniej, obecnie zajmiemy się zagadnieniami związanymi z adresowaniem układów *Slave* oraz organizowaniem danych tak, aby były prawidłowo przesyłane.

Na rysunku 14.34 przedstawiono możliwe sytuacje. W pierwszym przypadku (rysunek 14.34a) mamy do czynienia z zapisem danych z układu *Master* do *Slave*. Po wygenerowaniu sekwencji startu (S) *Master* przesyła linią SDA adres układu *Slave*, do którego kieruje dane (transmisja przebiega od najstarszego do najmłodszego bitu). Po 7 bitach adresu, ósmy bit ma wartość „0”, co oznacza, że dane będą zapisywane do *Slave'a* (jest to bit R/W, ustalający kierunek transmisji danych). Bajt ten odbierają wszystkie układy dołączone do magistrali, ale tylko ten, którego adres jest zgodny z nadanym odpowiada wystawiając bit potwierdzenia ACK poprzez ściągnięcie linii SDA do stanu „0”. Kolejny bajt transmitowany przez *Mastera*, to adres rejestru *Slave'a*, który będzie zapisywany w następnym kroku. Odebranie tego bajtu powoduje odpowiednie ustawienie rejestru adresowego w układzie *Slave*. W następstwie rozpoznania bitu potwierdzenia ACK wystawionego przez *Slave'a*, *Master* rozpoczyna wysyłanie kolejnych bitów, które będą po ich skompletowa-



Rys. 14.34. Zapis danych z układu *Master* do *Slave'a* (a), blokowy odczyt danych ze *Slave'a* (b), odczyt w kilku sesjach kolejnych rejestrów *Slave'a* (c)

niu w bajt zapisane do wskazanego rejestru *Slave'a*. Po wysłaniu ósmego bitu *Slave* odpowiada potwierdzeniem oraz automatycznie inkrementuje rejestr adresowy. Jeśli *Master* będzie nadawał następne bajty, będą one zapisywane do kolejnych rejestrów *Slave'a*. Przeskoczenie adresu wymaga zakończenia transmisji wystawieniem sekwencji stop (P) i powtórzenia całej powyższej procedury z podaniem nowego adresu układu i adresu rejestru docelowego.

Sytuacja przedstawiona na rysunku 14.34b dotyczy blokowego odczytu danych ze *Slave'a*, począwszy od wskazanego adresu. Transmisja jest inicjowana wystawieniem sekwencji startu (S), a następnie zaadresowaniem odpowiedniego *Slave'a*. Bit ósmy, występujący bezpośrednio po adresie, tak jak w poprzednim przypadku ma wartość „0”. Każdy wysłany bajt kończy się jak zwykle wystawieniem potwierdzenia. Po adresie *Slave'a* i tym razem należy wskazać adres rejestru, który będzie odczytywany w dalszej części sesji. Przełączenie *Slave'a* na nadawanie następuje po ponownym przesłaniu sekwencji startu, a następnie jego adresu i bitu R/W o wartości „1”. *Slave* odpowiada bitem potwierdzenia ACK. Po jego prawidłowym rozpoznaniu *Master* zwalnia linię SDA i od tej chwili kontrolę nad nią przejmuje *Slave*. Wystawia teraz kolejne bity danych w takt zegara SCL generowanego przez *Mastera*. Po wysłaniu ósmego bitu linia SDA ponownie przechodzi pod kontrolę *Mastera*, który wystawia bit potwierdzenia. Licznik adresowy *Slave'a* jest inkrementowany co powoduje, że w następnych krokach są przesyłane kolejne rejestry. Transmisja przebiega w podobny sposób do momentu, gdy *Master* nie wystawi potwierdzenia (ACK=1). Dopiero ten warunek wyłącza nadajnik *Slave'a*. Nie można jednak zapominać o wygenerowaniu sekwencji stop (P). Szczególnym przypadkiem może być oczywiście odczytanie tylko jednego bajtu.

Czasami zachodzi potrzeba odczytywania kolejnych rejestrów *Slave'a*, ale nie w jednej sesji lecz w kilku. Taki przypadek przedstawiono na **rysunku 14.34c**. Po wygenerowaniu sekwencji startu (S) i zaadresowaniu *Slave'a* z bitem $R/\overline{W}=1$ następuje bezpośrednie odczytywanie kolejnych rejestrów. Transmisja przebiega w znany już sposób, jest kończona brakiem potwierdzenia i sekwencją stop (P).

Czytając powyższy opis można zadać sobie pytanie, jak to jest możliwe, że różne układy wyposażone zarówno w nadajniki jak i odbiorniki korzystają bezkolizyjnie z tych samych dwóch linii transmisyjnych SDA i SCL. Odpowiedzi można się domyślać po lekturze opisu interfejsu 1-Wire, metoda jest bowiem podobna. Porządku na magistrali pilnują układy *Master*, a dostęp do magistrali przez wiele układów zapewniają wyjścia typu *open drain*. Każda linia jest indywidualnie podciągana do plusa zasilania zewnętrznym rezystorem podciągającym. Wartości tych rezystorów powinny gwarantować uzyskiwanie odpowiednich stromości zboczy impulsów przy uwzględnieniu pojemności występujących na danej linii (ich rezystancja ma zazwyczaj wartość pojedynczych kΩ). Pojęcie „linia zwolniona” oznacza wyłączenie tranzystora wyjściowego, w wyniku czego na linii panuje stan wysoki wymuszany rezystorem podciągającym. Włączenie tego tranzystora oznacza natomiast „zwarcie” linii do masy, a więc wymuszenie na niej stanu niskiego.

Programowa obsługa interfejsu I²C

Mikrokontrolery AVR nie mają sprzętowego interfejsu I²C. Trzeba go obsługiwać w sposób programowy. Na szczęście zadanie nie jest zbyt trudne. Do realizacji interfejsu na płytce ZL1AVR wykorzystano port PD5 (SDA) i PD6 (SCL). Sterowanie liniami będzie realizowane podobnie, jak w interfejsie 1-Wire. Wyprowadzenia PD5 i PD6 będą skonfigurowane jako wejściowe bez podciągania. Na pozycje PORTD5 i PORTD6 rejestru wyjściowego PORTD będzie więc na stałe wpisana wartość „0”, zaś w rejestrze DDRD odpowiednie bity (DDD5 i DDD6) należy ustawić w stan „1”. Linie interfejsu będą odczytywane z rejestru PIND. Ustawienie danej linii w stan „1” jest równoważne z rekonfiguracją portu w tryb wejściowy, czemu towarzyszy – jak wiemy – wymuszenie zewnętrznym rezystorem podciągającym stanu wysokiego. Wyzerowanie linii będzie polegało na przełączeniu portu w tryb wyjściowy. Wówczas zapisane wcześniej „0” do PORTD spowoduje ustawienie linii interfejsu w stan niski. W procedurach obsługi interfejsu I²C zostaną zastosowane znane już metody definiowania bitów za pomocą struktury `pole_bitowe`, co powinno poprawić czytelność programu. Dla zwiększenia precyzji pętli opóźnienia zastosowano funkcję `czekaj_I2C(czas)`.

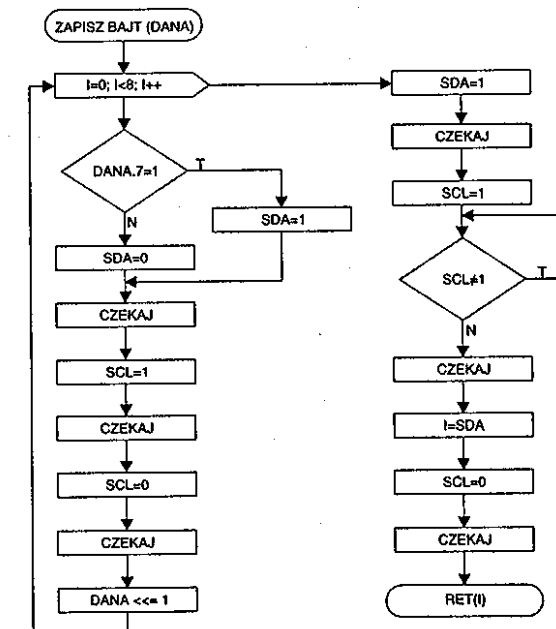
Opóźnienie jest dla niej równe:

$$t = (5 + 5 \cdot \text{tauI2C}) \cdot T,$$

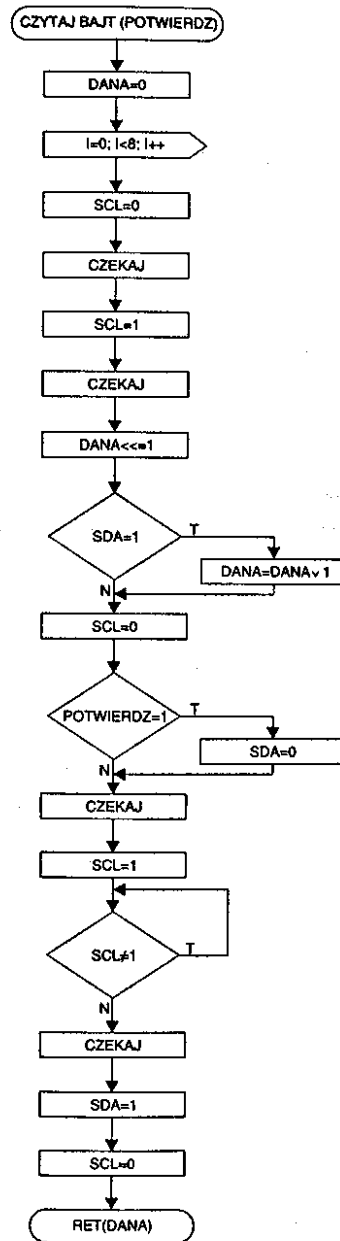
gdzie: `tauI2C` – parametr procedury, a `T` – okres oscylatora systemowego (125 ns przy częstotliwości taktowania 8 MHz). Funkcja ta jest analogiczna do `czekaj_1w` w poprzednim ćwiczeniu.

Do realizacji transmisji poprzez interfejs I²C służą w programie następujące funkcje:

- `bitstartu()` i `bitstopu()` – generują odpowiednio sekwencję start (S) i stop (P),
- `zapiszB_I2C(dana)` – wysyła poprzez I²C pojedynczy bajt przekazywany jako parametr `dana`, zwraca wartość bitu potwierdzenia ACK,
- `czytajB_I2C(ack)` – odczytuje z I²C pojedynczy bajt i w zależności od parametru `ack` generuje bit potwierdzenia lub nie generuje go, zwraca wartość odczytanego bajtu,
- `doI2C(adri2C, adrdane, lz)` – wysyła `lz` bajtów do układu o adresie `adri2C`, dane będą zapisywane w nim począwszy od adresu `adrdane`,
- `odI2C(adri2C, adrdane, lz)` – odczytuje `lz` bajtów z układu o adresie `adri2C`, dane będą odczytywane z niego począwszy od adresu `adrdane`.



Rys. 14.35. Algorytm działania funkcji `zapiszB_I2C`

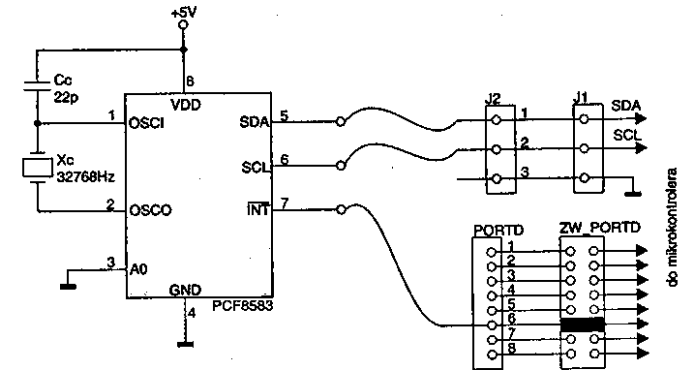


Rys. 14.36. Algorytm działania funkcji czytajB_I2C

W funkcjach doI2C i odI2C wykorzystywany jest ten sam bufor bufI2C[9], w którym umieszczane są dane do wysłania oraz dane odebrane. Algorytmy funkcji zapiszB_I2C i czytajB_I2C przedstawiono na rysunkach 14.35 i 14.36.

W ćwiczeniu 10. zastosowano układ PCF8583, będący kompletnym, autonomicznym zegarem czasu rzeczywistego. Do prawidłowej pracy wymaga on dołączenia typowego zegarkowego rezonatora kwarcowego $2^{15} = 32768$ Hz oraz kondensatora o pojemności od 15 do 35 pF. Zastąpienie zwykłego kondensatora trymerem umożliwi regulację zegara, lecz do eksperymentów nie jest to oczywiście konieczne. Możliwe jest również taktowanie układu zewnętrznym przebiegiem prostokątnym o częstotliwości 50 Hz, doprowadzonym do wejścia OSC1. Przed przystąpieniem do eksperymentów należy na uniwersalnym polu montażowym zmontować układ, którego schemat pokazano na rysunku 14.37. Do punktów połączonych z wyprowadzeniami SDA (5), SCL (6) i INT (7) można przylutować pionowe kawałki przewodów, tak żeby wystawały ok. 7 mm ponad płytkę. Zabieg ten uwolni nas to od konieczności używania lutownicy za każdym razem, gdy będziemy chcieli zmienić konfigurację. Teraz wystarczy tylko nałożyć kabelek na tak przygotowany pin i połączyć z odpowiednim pinem gniazda J2 lub PORTD (rysunek 14.37). Rezystory podciągające interfejsu I²C znajdują się na płytce ZL1AVR.

Oprócz zaznajomienia się z interfejsem I²C, w ćwiczeniu zostanie pokazany przykład użycia przerwania zewnętrznego

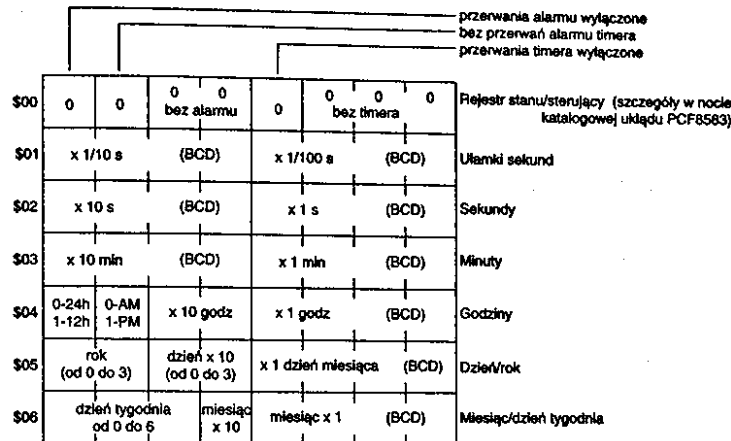


Rys. 14.37. Schemat elektryczny RTC do zmontowania na uniwersalnym polu montażowym zestawu ZL1AVR

INT0. W trybie pracy jako zegar czasu rzeczywistego na wyjściu INT układu PCF8583 jest generowany przebieg prostokątny o częstotliwości 1 Hz i współczynnika wypełnienia 0,5. Wystąpienie opadającego zbocza na tym wyjściu będzie sygnalizowało mikrokontrolerowi konieczność odczytania danych z RTC i zaktualizowanie stanu wyświetlacza. Trzeba w tym miejscu wspomnieć, że PCF8583 może pracować również jako timer lub licznik zdarzeń, zliczający impulsy z wejścia OSC1. W tych trybach wyjście INT działa trochę inaczej. W ćwiczeniu 10. zostanie wykorzystana tylko najprostsza funkcja zegara czasu rzeczywistego, bez ustawiania alarmów. Dane o aktualnym czasie umieszczone są w rejestrach układu PCF8583, począwszy od adresu \$01 do \$06. Pod adresem \$00 znajduje się rejestr stanu będący jednocześnie rejestrem sterującym. Na rysunku 14.38 przedstawiono adresowanie danych i ich format.

Program pokazany na listingu 14.11 zaczyna się od znanych już dobrze instrukcji inicjujących pracę wyświetlacza LCD (2×16). Zaraz po nich w rejestrze GIMSK jest ustawiany bit INT0, włączający przerwania zewnętrzne zgłaszane wejściem PD2 (INT0). Ze względu na długi czas trwania stanu aktywnego i praktyczny brak możliwości jego zmiany, trzeba wybrać wariant zgłaszania przerwania zboczem. Arbitralnie wybrano zbocze opadające i z tego powodu do rejestru MCUCR zostaje wpisana wartość 0x02. Przerwania będą mogły być przyjmowane po ustawieniu bitu I w rejestrze SREG. Czyny to instrukcja sei() umieszczona w dalszej części programu.

Przed wejściem w główną pętlę programu jest sprawdzany stan klawisza SW1. Rozwiązanie takie jest bardzo często stosowane, np. do uruchamiania procedur konfiguracyjnych, do których dostęp powinien być zabroniony pod-



Rys. 14.38. Rejestry wewnętrzne układu PCF8583

czas normalnego działania programu. W tym przypadku, jeśli przed restartem systemu zostanie naciśnięty i przytrzymany klawisz SW1, na wyświetlaczu pojawi się krótka informacja o płytce ZL1AVR. Ponowne jego naciśnięcie spowoduje wejście w normalny tryb pracy. Główna pętla programu składa się zaledwie z dwóch instrukcji warunkowych. Pierwsza – `if(zegar)`, to sprawdzenie czy można uaktualnić stan wyświetlacza. Flaga `fzegar` jest ustawiana w przerwaniu `INT0` wywołanym przebiegiem 1 Hz z wyjścia `INT` układu PCF8583. Zgłoszenie tego przerwania oznacza zmianę stanu rejestrów zegara, a więc konieczność ich ponownego wyświetlenia. Obsługą przerwania zajmuje się funkcja `SIGNAL(SIG_INTERRUPT0)`. Wywołuje ona funkcję `odI2C(rtc, 0x02, 3)`, która odczytuje trzy kolejne rejestry począwszy od adresu `0x02` z układu mającego symboliczny adres RTC (zdefiniowany w części deklaracyjnej programu jako `0xa0`). Te trzy rejestry zawierają informacje – jak wynika z rysunku 14.38 – o sekundach, minutach i godzinach aktualnego czasu. W celu uatrakcyjnienia działania programu, w procedurze obsługi przerwania zmienia się dodatkowo stany diod LED1 i LED2. Będą one migać co sekundę. Wyświetlaniem czasu zajmuje się funkcja `wyswietlczas()`. Pobiera ona dane bezpośrednio z bufora interfejsu I²C, dokonuje odpowiednich rozgrupowań danych i wyniki wyświetla na LCD za pośrednictwem funkcji `nalcd(zn1, zn2)`. Ze względu na to, że operujemy tylko na cyfrach, konwersja na znaki ASCII nie jest realizowana za pomocą procedur bibliotecznych, a jedynie poprzez proste dodanie liczby `0x30`.

Druga instrukcja warunkowa głównej pętli programu sprawdza, czy został naciśnięty klawisz SW4. Jeśli to nastąpi, zostaje wywołana funkcja ustawie-

nia zegara `ustawzegar()`. Na wstępie są wyłączane przerwania, aby uchronić bufor `bufI2C` przed przypadkową modyfikacją i jest wyświetlana informacja o trybie ustawiania. Na wyświetlaczu jest włączane mruganie znaku, które ułatwia orientację podczas ustawiania zegara. W pierwszej kolejności zmieniane są cyfry godzin. Służy do tego celu klawisz SW1. Jego naciśnięcie powoduje modyfikację odpowiedniej grupy cyfr. Godziny są zliczane modulo 24 (czyli od 0 do 23). Po prawidłowym ich ustawieniu należy naciśnąć klawisz SW4, w wyniku czego analogiczną operację wykonuje się teraz na cyfrach minut. Tym razem licznik liczy modulo 60 (czyli od 0 do 59). Trzeba zwrócić uwagę na to, że zmienne `godz` i `min` są bezpośrednim odzwierciedleniem danych z rejestrów układu PCF8583. Zawierają więc liczby zapisane w kodzie BCD. Jeśli więc w wyniku inkrementowania takiej zmiennej nastąpi przekroczenie wartości 9, konieczne będzie wykonanie odpowiedniej korekcji. Najprostsza metoda polega na dodaniu liczby `0x06`. Funkcja `ustawzegar()` nie ustawia sekund, przyjmując założenie, że zegar będzie startowany zawsze o pełnych minutach. Na zakończenie jest wyświetlany komunikat „STARTn/t” zachęcający do naciśnięcia któregoś z przycisków. Jeśli będzie to SW4, nowe dane zostaną zapisane do RTC i od tej chwili zegar będzie pokazywał ustawiony czas. Naciśnięcie klawisza SW1 spowoduje zignorowanie ustawień i kontynuację wyświetlania czasu, jaki obowiązywał przed wejściem do funkcji `ustawzegar()`. W trakcie eksperymentowania warto rozłączyć poszczególne linie interfejsu oraz `INT` i obserwować jak układ będzie reagował na takie sytuacje.

Do wykonania ćwiczenia konieczne jest zmontowanie układu, którego schemat pokazano na rysunku 14.37. Program realizujący przedstawione zadanie znajduje się na listingu 14.11. Należy skompilować program `cwicz10.c` i zaprogramować mikrokontroler z zestawu ZL1AVR plikiem wynikowym `cwicz10.hex`.

Konfiguracja płytki ZL1AVR:

- zworka J3 w pozycji 1-2, zworka J4 w pozycji 2-3 – dołączony oscylator wewnętrzny,
- zworka JP1 zwarta – globalne włączenie diod LED,
- zworka JP4 i JP5 w pozycji 2-3 (dołączenie pojedynczych klawiszy do mikrokontrolera),
- zworka JP6 i JP7 zwarta (dołączenie klawiszy do mikrokontrolera),
- zworki ZW_PORTB zwarte,
- zworki JP8 i JP9 rozwarne,
- zworki JP15 i JP16 zwarte,
- zworka ZW_PORTD zwarta w pozycji 11-12,

- połączyć kabelkiem nóżkę 6 PORTD z wyprowadzeniem 7. układu PCF8583,
- połączyć kabelkiem nóżkę 1 J2 z wyprowadzeniem 5. układu PCF8583,
- połączyć kabelkiem nóżkę 2 J2 z wyprowadzeniem 6. układu PCF8583,
- położenie pozostałych zwerek nieistotne (np. rozłączone),
- włożony wyświetlacz alfanumeryczny 16x2 do gniazda LCD1.

List. 14.11. Program do ćwiczenia 10

```

/* ***** */
/* Ćwiczenie 10 - Obsługa interfejsu I2C */
/* układ RTC (zegar czasu rzeczywistego) - PCF8583 */
/* Obsługa przerwania zewnętrznego INT0 */
/* Obsługa wyświetlacza LCD 2x16 */
/* J.D. '2003 */
/* ***** */

#include <io2313.h>
#include <progmem.h>
#include <interrupt.h>
#include <signal.h>

// Poniższe definicje służą do realizacji wygodnego dostępu bitowego
typedef struct _bit_struct
{
    unsigned char bit0: 1;
    unsigned char bit1: 1;
    unsigned char bit2: 1;
    unsigned char bit3: 1;
    unsigned char bit4: 1;
    unsigned char bit5: 1;
    unsigned char bit6: 1;
    unsigned char bit7: 1;
}pole_bitowe;

#define DAJ_BIT(adr) ((*({volatile pole_bitowe*} (adr)))
#define _PORTB 0x38
#define _PINB 0x36
#define _PORTD 0x32
#define _DDRD 0x31
#define _PIND 0x30
#define sda_we DAJ_BIT(_PIND).bit5
#define sda_wy DAJ_BIT(_DDRD).bit5
#define scl_wy DAJ_BIT(_DDRD).bit6
#define scl_we DAJ_BIT(_PIND).bit6
#define lcd_rs DAJ_BIT(_PORTB).bit2
#define lcd_e DAJ_BIT(_PORTB).bit3
#define sw4 DAJ_BIT(_PIND).bit1
#define sw1 DAJ_BIT(_PIND).bit0

#define stan_0 1 //definicja stanu niskiego na liniach I2C
// *1* oznacza przełączenie portu w tryb wyjściowy
// port jest wcześniejysterowany w stan niski
#define stan_1 0 //definicja stanu wysokiego na liniach I2C
// *0* oznacza przełączenie portu w tryb wejściowy
// stan wysoki jest wymuszany przez zewnętrzny
// rezystor podciągający

```

[illegible]

```
{ //umieszczenie czasu na LCD z jednoczesna  
//konwersją na ASCII  
    piszznak(zn1+0x30);  
        //dodanie 0x30 realizuje prostą konwersję  
        //liczby na ASCII  
  
    piszznak((zn2&0xf)+0x30);  
}  
  
//>>>>>>>>>>>> Procedury obsługi interfejsu I2C <<<<<<<<<<<<  
void czekaj_I2C(unsigned char tauI2C) //pętla opóźnienia dla I2C  
{  
do  
//opóźnienie jest równe t=(5+5*tauI2C)*T  
//T - cykl zegarowy MCU,  
//tauI2C - parametr procedury  
{  
asm("nop"); //wstawka assemblerowa - rozkaz NOP  
}while(--tauI2C!=0);  
}  
  
void bitstartu(void) //bit startu na magistrali I2C  
{  
SCL SDA  
sda_wy=stan_1;  
czekaj_I2C(10);  
scl_wy=stan_1;  
czekaj_I2C(10);  
sda_wy=stan_0;  
czekaj_I2C(10);  
scl_wy=stan_0;  
czekaj_I2C(10);  
}  
  
void bitstopu(void)  
{  
SCL SDA  
sda_wy=stan_0;  
czekaj_I2C(10);  
scl_wy=stan_1;  
czekaj_I2C(10);  
sda_wy=stan_1;  
czekaj_I2C(10);  
}  
  
unsigned char zapiszB_I2C(unsigned char dana)  
{  
//wysłanie pojedynczego bajtu do Slave'a  
    unsigned char i;  
  
for(i=0;i<8;i++) //będzie 8 bitów  
{  
if(dana&0x80) //badaj najstarszy bit wysyłanego znaku  
{  
sda_wy=stan_1;  
//wyślij "1"  
}  
else  
{  
sda_wy=stan_0;  
//wyślij "0"  
}  
czekaj_I2C(10);  
scl_wy=stan_1;  
czekaj_I2C(10);  
scl_wy=stan_0;  
czekaj_I2C(10);  
dana<<=1;  
//przygotuj następny bit do wysłania  
}
```

```

sda_wy=stan_1;           //zwolnij linie SDA dla Slave'a
czekaj_I2C(10);
scl_wy=stan_1;
while(!scl_we);          //czekaj, aż Slave będzie gotowy do
                          //przyjęcia następnego bitu

czekaj_I2C(10);
i=sda_we;                //czytaj potwierdzenie ACK od Slave'a
scl_wy=stan_0;
czekaj_I2C(10);
return i;                //zwróć bit potwierdzenia ACK
}

unsigned char czytajB_I2C(unsigned char ack)
{
    //przyjęcie pojedynczego bajtu od Slave'a I2C
    //parametr ack=1 oznacza, że jest wysyłany ostatni bajt
    //bloku, nie należy więc wysyłać potwierdzenia
    unsigned char dana=0,i;

    for(i=0;i<8;i++)      //będzie 8 bitów
    {
        scl_wy=stan_0;
        czekaj_I2C(10);
        scl_wy=stan_1;
        czekaj_I2C(10);
        dana<<=1;        //przygotuj miejsce na kolejny odebrany bit
        dana|=sda_we==1?1:0; //dopisz odebrany bit
    }
    scl_wy=stan_0;
    if(ack)               //czy wysyłać potwierdzenie?
    {
        sda_wy=stan_0;    //wyślij ACK
    }
    else
    {
        sda_wy=stan_1;    //brak potwierdzenia sygnalizuje zakończenie
                          //bloku
    }
    czekaj_I2C(10);
    scl_wy=stan_1;
    while(!scl_we);      //czekaj, aż Slave będzie gotowy do przyjęcia
                          //następnego bitu

    czekaj_I2C(10);
    sda_wy=stan_1;        //zwolnij linie SDA
    scl_wy=stan_0;
    return dana;
}

void bladI2C(void)
{
    //reakcja na brak potwierdzenia ACK od Slave'a
    char *tekst[2]={
        PSTR("Błąd I2C"),
        PSTR(" ")
    };

    lcdxy(0,0);
    pisztekst(tekst[0]); //wyświetl na LCD komunikat o błędzie I2C
    czekaj(1500*tau);
    lcdxy(0,0);
    pisztekst(tekst[1]); //wyczyść komunikat o błędzie
}

```

```

void doI2C(unsigned char adrI2C,unsigned char adrdane,unsigned char lz)
{
    //wysłanie bloku "lz" znaków do Slave'a I2C
    //"adrI2C" - adres fizyczny Slave'a
    //"adrdane" - adres rejestru w Slave'ie, od którego będą
    //zapisywane dane
    pbufI2C=&bufI2C;      //ustaw wskaźnik bufora I2C na jego początek
    bitstartu();           //wyślij bit startu (początek transmisji)
    if(zapiszB_I2C(adrI2C)) //wyślij adres Slave'a, to informacja dla
                          //pozostałych układów Slave, że dane nie będą
                          //dla nich
    {
        bladI2C();         //wyświetl komunikat o błędzie interfejsu I2C,
                          //jeśli nie było ACK
    }
    if(zapiszB_I2C(adrdane)) //zaadresuj rejestr w Slave'ie
    {
        bladI2C();         //wyświetl komunikat o błędzie interfejsu I2C,
                          //jeśli nie było ACK
    }
    for(;lz!=0;lz--)        //wysyłanie bloku danych (lz - liczba bajtów
                          //do wysłania)
    {
        if(zapiszB_I2C(*pbufI2C++)) //wyślij kolejny bajt do Slave'a
        {
            bladI2C();      //wyświetl komunikat o błędzie interfejsu I2C,
                          //jeśli nie było ACK
        }
    }
    bitstopu();            //wyślij bit stopu (koniec transmisji)
}

void odI2C(unsigned char adrI2C,unsigned char adrdane,unsigned char lz)
{
    //odebranie bloku "lz" znaków od Slave'a I2C
    //"adrI2C" - adres fizyczny Slave'a
    //"adrdane" - adres rejestru w Slave'ie, od którego będą
    //odczytywane dane
    bitstartu();           //wyślij bit startu (początek transmisji)
    if(zapiszB_I2C(adrI2C)) //wyślij adres Slave'a, to informacja dla
                          //wybranego Slave'a, że dane będą odbierane
                          //od niego
    {
        bladI2C();         //wyświetl komunikat o błędzie interfejsu
                          //I2C, jeśli nie było ACK
    }
    if(zapiszB_I2C(adrdane)) //zaadresuj rejestr w Slave'ie
    {
        bladI2C();         //wyświetl komunikat o błędzie interfejsu
                          //I2C, jeśli nie było ACK
    }
    bitstartu();           //ponowne wysłanie bitu startu
    if(zapiszB_I2C(adrI2C+1)) //ponowne wysłanie adresu Slave'a z bitem
                          //R/W=1 oznacza przełączenie Slave'a na
                          //nadawanie
    {
        bladI2C();         //wyświetl komunikat o błędzie interfejsu
                          //I2C, jeśli nie było ACK
    }
    pbufI2C=&bufI2C;      //ustaw wskaźnik bufora I2C na jego początek
    for(;lz>1;lz--)        //odbierz "lz" bajtów danych od Slave'a
    {

```

```

    *pbufI2C++=czytajB_I2C(1); //zapisuj odebrane bajty w buforze I2C
}
*pbufI2C++=czytajB_I2C(0); //odbierz ostatni bajt od I2C, nie wysyłaj ACK
bitstopu(); //wyslij bit stopu (koniec transmisji)
}

SIGNAL(SIG_INTERRUPT0) //procedura obsługi przerwania zewnętrznego
//INT0
{
    PORTB^=0x03; //zmień stany LED1 i LED2
    odI2C(rtc,0x02,3); //odczytaj czas z RTC (tylko godz, min i sek)
    fzegar=1; //ustaw flagę odczytania danych
}

void wyswietlczas(void) //procedura wyświetlania czasu na LCD
{
    unsigned char zp; //zmienna pomocnicza

    lcdxy(1,0);
    pbufI2C=&bufI2C[2]; //ustaw wskaźnik bufora na pozycje godzin
    zp=*pbufI2C--; //pobierz godziny
    nalcd((zp&0x30)>>4,zp); //wyświetl godziny
    pizznak(':');
    zp=*pbufI2C--; //pobierz minuty
    nalcd((zp&0xf0)>>4,zp); //wyświetl minuty
    pizznak(':');
    zp=*pbufI2C; //pobierz sekundy
    nalcd((zp&0xf0)>>4,zp); //wyświetl minuty
}

void zwolnijklaw(void)
{
    czekaj(200*tau);
    while((PIND&0x03)!=0x03); //czekaj aż wszystkie klawisze będą zwolnione
}

void ustawzegar(void) //procedura ustawiania czasu w układzie RTC
{
    char *kom[1]={PSTR("STARTn/t")};

    cli(); //wyłącz przerwania, aby nie nazytać danych
    pizilcd(0x0d); //włącz mruganie
    odI2C(rtc,0x02,3); //odczytaj czas z RTC (tylko godz, min, sek)
    wyswietlczas();
    pbufI2C=&bufI2C[1]; //ustaw wskaźnik bufora I2C na minuty
    //sekundy będą zawsze zerowane
    bufI2C[0]=0;
    min=*pbufI2C++; //pobierz minuty
    godz=*pbufI2C; //pobierz godziny
    lcdxy(1,1);
    zwolnijklaw();
    while(sw4) //czekaj na naciśnięcie SW4 wykonując
    //poniższe instrukcje ustawianie godzin
    {
        lcdxy(1,1);
        if(!sw1) //czy naciśnięto SW1?
        {
            godz++; //inkrementuj godziny
            if((godz&0x0f)>9)
            {
                godz+=0x06; //korekcja dziesiętna liczby BCD
            }
        }
        if((godz&0x3f)>0x23)
    }
}

```

```

{
    godz&=0xc0; //kasuj godziny po przekroczeniu zakresu
}
bufI2C[2]=(bufI2C[2]&0xc0)|godz;
//zapisz uaktualnione godziny w buforze I2C
//pozostawiając bity 24/12 i AM/PM
//nienaruszone

wyswietlczas();
czekaj(150*tau);
}
}
lcdxy(1,4);
zwolnijklaw();

while(sw4) //czekaj na naciśnięcie SW4 wykonując
//poniższe instrukcje
//ustawianie minut
{
    lcdxy(1,4);
    if(!sw1) //czy naciśnięto SW1?
    {
        min++; //inkrementuj minuty
        if((min&0x0f)>9)
        {
            min+=0x06; //korekcja dziesiętna liczby BCD
        }
        if(min>0x59) //korekcja przekroczenia wartości 59
        {
            min=0; //przeniesienia na godziny nie ma, bo są one
            //ustawiane niezależnie
        }
        bufI2C[1]=min; //zapisz uaktualnione minuty w buforze I2C
        wyswietlczas();
        czekaj(150*tau);
    }
}

pizilcd(0x0c); //wyłącz mruganie
lcdxy(1,8);
piztekst(kom[0]); //wyświetl komunikat "START"
zwolnijklaw();
while(sw1&sw4); //czekaj na naciśnięcie jakiegos klawisza
//po naciśnięciu klawisza SW4...
{
    doI2C(rtc,2,3); //...zapisz ustawienia zegara
    bufI2C[0]=0;
    doI2C(rtc,0,1); //start zegara
}
sei(); //włącz przerwania
}

int main(void) //program główny
{
    unsigned char i,zp;

    //tablica komunikatów do wyświetlenia
    char *info[4]={
        PSTR("Ustaw zegar   ."),
        PSTR("Błąd I2C      "),
        PSTR("RTC - PCF8583"),
        PSTR("ZL1AVR - płytka\newaluacyjna AVR")
    };
}

```

[illegible]

```

czekaj(150*tau);
lcdxy(0,0);
pisztekst(info[0]); //wyświetl komunikat o trybie ustawiania
ustawzegar();       //ustaw zegar
czysclcd();
pisztekst(info[2]); //odtwórz komunikat o układzie RTC na LCD
czekaj(300*tau);
}
}
}

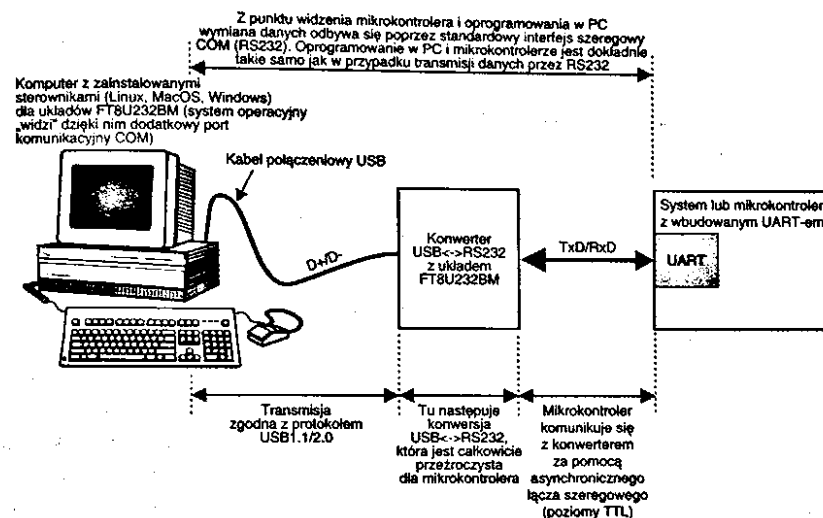
```

14211. Ćwiczenie 11

**Podłączenie mikrokontrolera AVR do komputera PC przez port USB.
Obsługa nadajnika i odbiornika UART z wykorzystaniem systemu
przerwań. Obsługa wyświetlacza LCD 16×2**

W pierwotnych zamierzeniach wykorzystywanie interfejsu USB przez mikrokontrolery AVT nie miało być opisane w niniejszej książce. Powodem jest bardzo rozbudowany protokół transmisji i wynikające z tego problemy w prawidłowym oprogramowaniu własnych aplikacji. Konieczność przedzierania się przez kilkusetstronicową dokumentację traci często sens wobec możliwości stosowania alternatywnych portów komunikacyjnych, jak chociażby nieśmiertelnego RS232, czy portu równoległego Centronics. Ten ostatni, po wprowadzeniu trybów EPP i ECP stał się bardzo atrakcyjnym rozwiązaniem, pozwalającym na stosunkowo proste oprogramowanie szybkich i wydajnych procedur komunikacyjnych. Nieco gorzej wygląda sprawa z łączem szeregowym. Tam barierę stanowi maksymalna, standardowa prędkość transmisji równa 115,2 kb/s. Zaletą zaś jest bardzo proste oprogramowanie takiego interfejsu. Jest jednak pewien problem, który staje się coraz trudniejszy do pokonania. Jest nim powszechny już brak popularnych COM-ów w komputerach, głównie typu notebook, ale wszystko wskazuje na to, że tendencja ta będzie się rozszerzała również na komputery stacjonarne. Przejście z RS-a na USB może dla wielu młodych elektroników stanowić barierę nie do przeskoczenia.

Czy to oznacza, że są bezradni? Otóż nie. Na szczęście pojawiło się rozwiązanie, które uwalnia nas od wszystkich wspomnianych problemów. Rozwiązaniem tym jest układ FT8U232BM produkowany przez Future Technology Devices Intl. Ltd. znaną jako FTDI. Jest to układ, który emuluje połączenie poprzez standardowy port szeregowy, tworząc wirtualnego COM-a z wykorzystaniem USB (**rysunek 14.39**). Zaletą takiego rozwiązania jest możliwość dołączenia do złącza USB komputera dowolnego



Rys. 14.39. Dzięki układom FT8U232BM korzystanie z USB nie różni się w praktyce od korzystania z klasycznego interfejsu RS232

urządzenia bez konieczności poznawania jego tajników i pisania specjalnych sterowników.

Układ FT8U232BM realizuje następujące funkcje:

- wzajemną konwersję transmisji USB na standardową, asynchroniczną, szeregową transmisję danych,
- zapewnia pełną kontrolę przepływu danych, wykorzystującą modemowe sygnały interfejsowe, a także kontrolę typu X-on/X-off,
- obsługuje 7- lub 8-bitowe ramki danych z 1 lub 2 bitami stopu oraz kontrolą typu *Odd/Even/Mark/Space/NoParity*,
- zapewnia wymianę danych z szybkością dochodzącą do 3 Mbd,
- zapewnia regulowany *time-out* dla bufora odbiornika,
- umożliwia zasilanie dołączanych urządzeń (np. systemu mikroprocesorowego) z gniazda USB.

Układ wyposażono w wewnętrzny układ zerowania (*Power-On-Reset*), który zapewnia stabilne działanie układu po dołączeniu do USB. Wymianę danych ułatwia wbudowany w układ wewnętrzny bufor danych o rozmiarze 384 bajtów dla odbiornika i 128 bajtów dla nadajnika. Układ FT8U232BM jest zasilany napięciem o wartości 4,4...5,25 V – w zależności od konfiguracji bezpośrednio z USB lub ze współpracującego systemu mikroprocesorowego.

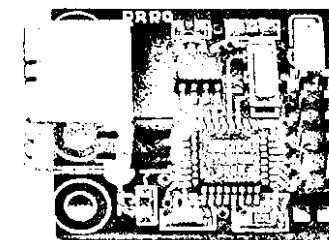
UWAGA

Wykorzystanie układu interfejsowego FT8U232BM powoduje, że komunikacja poprzez USB przebiega – z punktu widzenia mikrokontrolera i współpracującej z nim aplikacji pracującej na PC – dokładnie w taki sam sposób, jak ma to miejsce w przypadku klasycznej transmisji danych przez RS232.

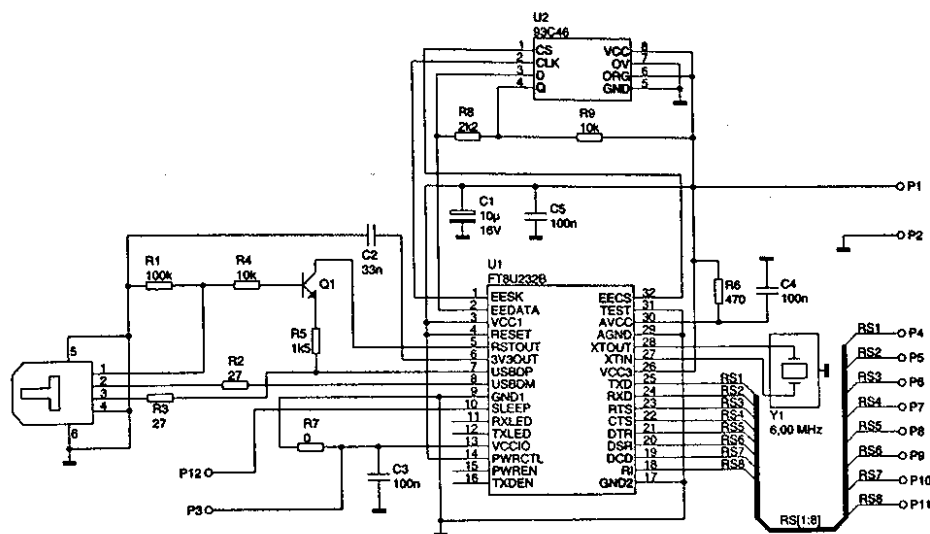
Układ FT8U232BM może być wykorzystywany w aplikacjach konwerterów USB<->RS232 oraz USB<->RS422/RS485, w interfejsach służących do łączenia systemów mikroprocesorowych z urządzeniami zewnętrznymi poprzez USB itp. Na stronie producenta są dostępne bezpłatne sterowniki dla systemów operacyjnych Windows 98, Windows 98 SE, Windows 2000/ME/XP, MAC OS-8 i OS-9, Linux 2.40 i późniejsze. Sterowniki te pozwalają na pisanie własnych procedur komunikacyjnych w sposób identyczny jak w przypadku urządzeń transmitujących dane za pośrednictwem interfejsu RS232. Do łączności od strony komputera można również stosować typowe programy terminalowe np. okienkowy HyperTerminal. Rozwiązanie takie wydaje się być najprostszą metodą realizacji transmisji z wykorzystaniem portu USB. Bardziej zaawansowani programiści mogą sięgać po sterowniki D2XX (*USB Direct Drivers + DLL S/W Interface*). Są one dostępne dla Windows 98, Windows 98 SE, Windows 2000/Me/XP. Dużą zaletą układów FT8U232 jest możliwość prawie dowolnego dobierania prędkości transmisji, byle tylko była ona taka sama po stronie nadawczej, jak i odbiorczej.

Celem ćwiczenia 11. jest pokazanie, jak w praktyce można dołączyć własne urządzenie do komputera PC poprzez port USB i zademonstrowanie przykładowego programu realizującego transmisję danych w obu kierunkach. Do realizacji tego zamierzenia wykorzystano gotowy interfejs opisany w Elektronice Praktycznej 5/2003. Wygląd tego modułu pokazano na fotografii 14.40, jego schemat elektryczny przedstawiono na rysunku 14.41, a na rysunku 14.42 pokazano sposób dołączenia go do płytki ZLI1AVR.

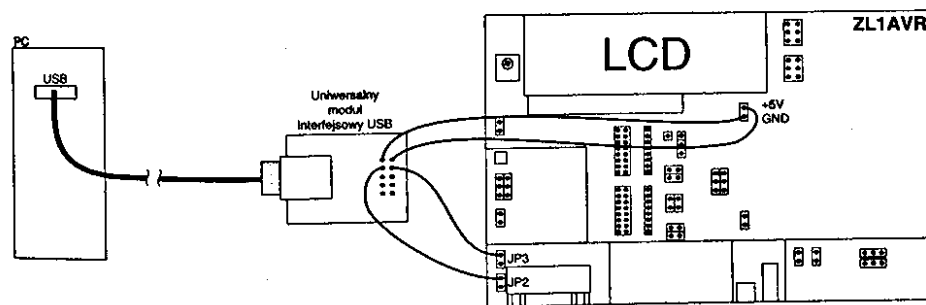
Po dołączeniu modułu USB do płytki ZLI1AVR oraz komputera PC (za pomocą typowego kabla USB) należy zasilić zestaw i włączyć komputer. Przy pierwszym uruchomieniu system Windows powinien automatycznie rozpoznać nowe urządzenie i zaproponować zainstalowanie sterowników.



Fot. 14.40. Wygląd interfejsu USB<->RS232 (TTL)



Rys. 14.41. Schemat elektryczny uniwersalnego modułu interfejsowego USB



Rys. 14.42. Sposób dołączenia uniwersalnego modułu interfejsowego USB do płytki ZL1AVR i komputera PC

Należy je wcześniej skopiować na dyskietkę lub CD-ROM, można je także wgrać na dysk twardy komputera do dowolnego katalogu. Procedura instalacji przebiega niemal automatycznie, w standardowy dla Windows sposób (został on szczegółowo opisany w Elektronice Praktycznej 10/2002, wybrane fragmenty tego artykułu znajdują się w dodatku F).

Po zainstalowaniu sterowników w windowsowym Menedżerze urządzeń pojawia się nowy port szeregowy COM z kolejnym (następnym po dotychczas widzianym przez system) numerem, który w razie konieczności można ręcznie zmienić.

Port ten można skonfigurować w identyczny sposób, jak w przypadku HyperTerminala z ćwiczenia 8. (rysunek 14.22) – czyli 38400,n,8,1. Od tej chwili urządzenia dołączone do portu USB komputera będą dostępne poprzez wirtualny port szeregowy.



Sterowniki dla układu FT8U232BM są dostępne na stronie internetowej producenta, pod adresem: <http://www.ftdichip.com/FTDriver.htm>. Opis ich instalacji znajduje się w dodatku F.

Demonstracja działania modułu będzie polegała na przesłaniu bloku danych o wielkości 128 kB z zestawu ZL1AVR do komputera PC, a następnie przejściu na odbiór danych. W tym trybie znaki wystukiwane na klawiaturze komputera będą – za pomocą programu HyperTerminal – przesyłane do mikrokontrolera na płytce ZL1AVR i wyświetlane na wyświetlaczu LCD. Do obsługi wyświetlacza zostały wykorzystane procedury z poprzednich ćwiczeń. Nieznacznej zmianie w zakresie interpretacji znaków LF i CR uległ funkcja pisztekst.

Do obsługi nadajnika i odbiornika UART-u przewidziano 32-elementowy bufor `fifosio[32]`. Rozwiązanie takie było możliwe, dzięki założeniu sympleksowej (jednokierunkowej) transmisji. Do tego bufora są więc „wkładane” znaki odebrane przez UART, a także pobierane są z niego dane do wysłania. Gdyby zaistniała konieczność prowadzenia jednoczesnego nadawania i odbioru, bufor taki powinien zostać zdublowany (oddzielny dla nadajnika, oddzielny dla odbiornika). `fifosio` to bufor cykliczny. Dane są do niego wkładane począwszy od elementu o indeksie 0, aż do elementu o indeksie 31. Po zapisaniu danej pod adresem 31 następna dana będzie ponownie zapisana na pozycję 0 itd. Podobnie dzieje się z pobieraniem danych. Operacje wkładania i pobierania elementów do/z bufora odbywają się niezależnie, w dodatku w dwóch zupełnie różnych częściach programu – w segmencie głównym i w procedurach obsługi przerwań UART-u. Nad prawidłowością zapisu i odczytu bufora czuwają zmienne `izfifosio` (indeks zapisu `fifosio`), `iofifosio` (indeks odczytu `fifosio`) oraz zmienna `ldanych` zliczająca aktualnie wpisaną liczbę danych do bufora. Wartość `ldanych` nigdy nie może przekroczyć wielkości bufora, czyli wartości 32. Przed zapisaniem danej jest sprawdzany odpowiedni warunek i program zapętla się w oczekiwaniu na zwolnienie miejsca w buforze `while(ldanych==32);`. Każdorazowa operacja na `fifosio` powoduje inkrementację odpowiedniego indeksu. Z uwagi na wielkość bufora działanie to musi być wykonywane modulo 32.

Przykładowo instrukcja inkrementowania indeksu zapisu do bufora fifo-sio wygląda następująco:

```
izfifosio==31?izfifosio=0:++izfifosio;
```

Jest to może mało czytelny zapis, ale za to bardzo zwięzły. Oznacza tyle, że sprawdzany jest warunek, czy izfifosio jest równe 31. Jeśli tak, to zmienna ta jest następnie zerowana, jeśli nie, to zwiększana o jeden. Wykorzystanie bufora pozwala poprawić płynność transmisji. Jeśli generowanie danych do wysłania przebiega nierównomiernie w czasie, to – gdyby nie było bufora – mogłyby się tworzyć przerwy w nadawaniu zmniejszające efektywną prędkość transmisji. Bufor pozwala wysyłać dane w chwilach oczekiwania na kolejne. Jeśli generowanie danych przebiega szybciej niż ich nadawanie, to dostarczanie danych jest wstrzymywane w momencie całkowitego zapełnienia bufora, czyli gdy liczba danych w buforze osiągnie wartość 32. Osiągnięcie indeksu 31 nie oznacza zapełnienia bufora. Najczęściej będziemy mieli wówczas sytuację taką, że bufor szybko zostanie zapełniony, a następnie dane będą na przemian dokładane i pobierane z szybkością na jaką pozwala transmisja.

Pierwsza część programu, to wysłanie do komputera bloku danych o rozmiarze 128 kB. Dane, to powtarzające się znaki ASCII w kolejności od spacji do litery „Z”. Przy przyjętej prędkości transmisji cały blok powinien być wysłany w ciągu ok. 34 sekund. W tym miejscu warto zauważyć, że wykorzystywanie portu USB do przesyłania niewielkich paczek danych nie tylko może nie spełnić oczekiwań co do uzyskiwanego transferu, ale w pewnych sytuacjach może dać gorsze wyniki niż np. wykorzystanie portu szeregowego. Powodem jest dość rozbudowany protokół transmisji USB, który sprawia, że np. przy wysyłaniu jednego bajtu przez łącze przechodzi dużo więcej danych (nagłówki, sporo czasu zabiera negocjacja połączenia USB itp.).

Po zakończeniu transmisji na wyświetlaczu pojawia się komunikat zachęcający tym razem do przesyłania danych z komputera PC do zestawu. Jeśli będą to znaki wysyłane z klawiatury, to zostaną natychmiast wyświetlone na wyświetlaczu. Jeśli natomiast prześlemy plik tekstowy, to niestety zauważymy, że nasz system nie nadąża z obsługą transmisji, która przebiega dużo szybciej niż prędkość wyświetlania znaków na LCD. Zaobserwujemy tu działanie bufora, objawiające się prawidłowym wyświetleniem 32 znaków, po czym kolejne dane będą nadpisywane przez następne. Powodem jest brak kontroli przepływu transmisji. Mikrokontroler na płytce ZL1AVR nie może w żaden sposób powstrzymać nadajnika komputera.

Jeśli przyjrzymy się poniższemu programowi, to zauważymy bez trudu, że nie ma tam nawet jednego rozkazu, który związany by był z protokołem USB. Cały program jest napisany tak, jakby był pisany dla połączenia za pomocą RS232. Można się o tym łatwo się przekonać odłączając moduł USB, zwierając zworki JP2 i JP3 oraz łącząc płytkę ZL1AVR z komputerem poprzez port szeregowy COM. Można też wykonać eksperyment w drugą stronę. W tym celu trzeba się cofnąć do ćwiczenia 8. Płytkę skonfigurować zgodnie z opisem do tego ćwiczenia, z tą różnicą że zamiast kabla RS232 dołączamy moduł USB. W ten sposób uzyskujemy możliwość regulacji obrotów silnika z komputera PC poprzez port USB.

Do wykonania ćwiczenia konieczne jest dołączenie do zestawu modułu interfejsowego USB. Schemat ilustrujący sposób przyłączenia pokazano na rysunku 14.42. Program realizujący przedstawione zadanie znajduje się na listingu 14.12. Program *cwicz11.c* należy skompilować i zaprogramować mikrokontroler z zestawu ZL1AVR plikiem wynikowym *cwicz11.hex*.

Konfiguracja płytki ZL1AVR:

- zworka J3 w pozycji 1-2, zworka J4 w pozycji 2-3 – dołączony oscylator wewnętrzny,
- zworki ZW_PORTB zwarte w pozycjach 1-2, 3-4, 5-6, 7-8, 9-10, 11-12,
- zworki JP6 i JP7 rozłączone,
- zworki JP2 i JP3 rozłączone, będzie do nich dołączony kabelkami moduł USB,
- położenie pozostałych zworek nieistotne (np. rozłączone),
- włożony wyświetlacz alfanumeryczny 16×2 do gniazda LCD1.

List. 14.12. Program do ćwiczenia 11

```

/*****
/* Ćwiczenie 11 - Połączenie mikrokontrolera AVR do komputera PC */
/*                                           */
/*                                           */
/* Nadawanie i odbiór poprzez UART z użyciem przerwań */
/* J.D. '2003 */
/*****
#include <io.h>
#include <progmem.h>
#include <stdlib.h>
#include <interrupt.h>
#include <signal.h>

#define FCPU      8000000          //częstotliwość oscylatora CPU
#define VUART     38400           //prędkość transmisji (b/s)
#define VUBRR     FCPU/(VUART*16)-1 //wpis do UBRR dla VUART

// Poniższe definicje służą do realizacji wygodnego dostępu bitowego
typedef struct _bit_struct
{
    unsigned char bit0: 1;
    unsigned char bit1: 1;
    unsigned char bit2: 1;

```

[illegible]

```

asm("nop");
lcd_e=0; //impuls strobujący
czekaj(10L); //czekaj na gotowość LCD
lcd_e=1;
PORTB=(PORTB&0x0f)|((dana&0x0f)<<4); //przygotuj młodszy półbajt do LCD
asm("nop");
asm("nop");
asm("nop");
lcd_e=0; //impuls strobujący
czekaj(10L); //czekaj na gotowość LCD
}

void czysclcd(void) //czyść ekran
{
    piszclcd(0x01); //polecenie czyszczenia ekranu dla
                    //kontrolera LCD
    czekaj(1.64*tau); //rozkaz 0x01 wykonuje się 1.64ms
    wiersz=0;
    kolumna=0;
}

void lcdxy(unsigned char w, unsigned char k) //ustaw współrzędne kursora
{
    piszclcd((w*0x40+k)|0x80); //standardowy rozkaz sterownika LCD
                                //ustawiający kursor w określonych
                                //współrzędnych
}

void piszznak(char znak) //procedura umieszcza znak na
{
    piszdlcd(znak); //wyświetlaczu
                    //wyświetl znak na LCD
}

void pisztekst(char *tekst, unsigned char romram) //pisz tekst na LCD wskazywany przez
{
    //**tekst
    char zn;
    char nr=0;

    while(1)
    {
        if(!romram)
        {
            zn=PRG_RDB(&tekst[nr++]); //pobranie znaku z pamięci programu
        }
        else
        {
            if(!ldanych) //czy są jeszcze jakieś dane w buforze
            {
                break;
            }
            zn=fifosio[iofifosio]; //pobierz znak z bufora
            iofifosio=31?iofifosio=0:++iofifosio;
            //inkrementacja modulo 32
            ldanych--;
        }
        if(zn!=0) //czy nie ma końca tekstu?
        {
            if((zn==LF)|| (zn==CR)) //czy znak nowej linii?
            {
                if(zn==LF)
                {
                    wiersz=1?wiersz=0:++wiersz; //przejdź do nowej linii
                }
                else //był CR
                {
                    kolumna=0; //powrót karetki
                }
            }
            else
            {
                piszdlcd(zn); //umieść pojedynczy znak tekstu na LCD
            }
        }
    }
}

```

```

    kolumna==15?kolumna=0:++kolumna;
    //inkrementacja modulo 16
    if(kolumna==0)
    {
        wiersz=1?wiersz=0:++wiersz;
        //przejdź do nowej linii
    }
    lcdxy(wiersz,kolumna);
}
else
{
    break; //zakończ pętlę, jeśli koniec tekstu
}
}

SIGNAL(SIG_UART_RECV) //procedura obsługi odbiornika UART-u
{
    led1=0;
    if(ldanych<32)
    {
        fifosio[izfifosio]=UDR; //zapamiętaj odebrany znak
        izfifosio==31?izfifosio=0:++izfifosio;
        //inkrementacja modulo 32
        ldanych++;
    }
    led1=1; //zgaś led1
}

SIGNAL(SIG_UART_TRANS) //procedura obsługi nadajnika UART
//wywoływana po wysłaniu znaku
{
    char znak;

    led0=0;
    znak=fifosio[iofifosio]; //pobierz dane z pamięci RAM
    if(ldanych!=0) //czy koniec pobierania danych?
    {
        UDR=znak; //nie, wyślij znak pobrany z kolejki
        ldanych--;
    }
    else
    {
        cbi(UCR,TXEN); //tak, wyłącz nadajnik
    }
    iofifosio==31?iofifosio=0:++iofifosio;
    //inkrementacja mod 32
    led0=1;
}

int main(void)
{
    unsigned long i;
    char znak;

    //tablica komunikatów do wysłania
    char *info[2]={
        PSTR("Wysylam dane"),
        PSTR("Przeslij z PC-ta")
    };

    DDRB=0xff; //PORTB - wy
    PORTB=0xff;
    DDRD=0x02; //PD1 - wy (RXD), pozostałe we
    PORTD=0x02; //podciągania wejścia PD1 (RXD)

    lcd_rs=0; //opóźnienie ok. 45ms dla ustabi-
    czekaj(45*tau); //lizowania się napięcia zasilania

```

```

for(i=0;i<3;i++) //LCD (katalogowo min. 15 ms)
{ //3-krotne wysłanie 3-
    lcd_e=1;
    PORTB=(PORTB&0x0f)|0x30; //wyślij 3- do LCD
    asm("nop");
    asm("nop");
    asm("nop");
    lcd_e=0;
    czekaj(5*tau); //ok. 5 ms
}
lcd_e=1;
PORTB=(PORTB&0x0f)|0x20; //wyślij 2- do LCD
asm("nop"); //wymagane wydłużenie impulsu
asm("nop");
asm("nop");
lcd_e=0; //impuls strobujący
czekaj(10L);
piszicld(0x28); //interfejs 4-bitowy, 2 linie, znak 5x7
piszicld(0x08); //wyłącz LCD, wyłącz kursor,
//wyłącz mruganie
piszicld(0x01); //czyść LCD
czekaj(1.64*tau); //wymagane dla instrukcji czyszczenia
//ekranu opóźnienie
piszicld(0x06); //bez przesuwania w prawo
piszicld(0x0c); //włącz LCD, bez kursora, bez mrugania

pisztekst(info[0],0); //komunikat na LCD
UBRR=VUBRR; //ustaw prędkość transmisji
UCR=1<<RXCIE | 1<<TXCIE | 1<<RXEN; //zezwole nie na przerwanie od odbiorni-
//ka i nadajnika, zezwolenie na odbiór
//włącz przerwania

sei();

izfifosio=0; //inicjuj zmienne
iofifosio=0;
znak=' ';
sbi(UCR,TXEN); //włącz nadajnik
UDR=znak++; //umieść pierwszy znak
for(i=0;i<0x1ffff;i++) //pętla wysyłania znaków
{
    while(ldanych==32); //jeśli bufor przepełniony, to czekaj
    fifosio[izfifosio]=znak++; //umieść znak do wysłania w buforze
    if(znak>'Z')
    {
        znak=' ';
    }
    izfifosio==31?izfifosio=0:++izfifosio;
    //inkrementuj mod 32
    ldanych++;
}
czysicld();
pisztekst(info[1],0);
czekaj(2000*tau);
czysicld();

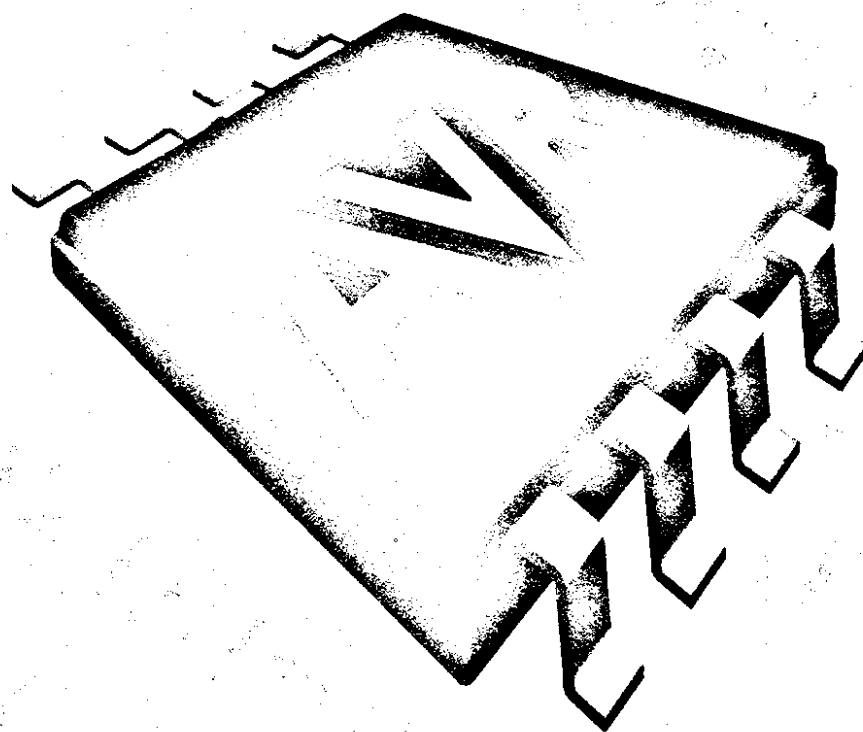
izfifosio=0; //inicjuj zmienne
iofifosio=0;
ldanych=0;
while(1) //pętla odbioru znaków
{
    if(ldanych) //czy odebrano jakiś znak?
    {
        pisztekst(fifosio[0],1); //wyświetl odebrane znaki
    }
}

```

Dodatek A

Podstawowe parametry mikrokontrolerów z rodziny AVR

	Pamięć Flash [kB]	Napięcie zasilania [V]	Pojemn. pamięci EEPROM [kB]	Pojemn. pamięci SRAM [B]	Maks. częstot. taktowania [MHz]	Liczba wyp. we/wy	Liczba przerwań	Liczba przerwań zewnętrz.	Interfejs TWI (odpow. I ² C)	Timer 16-bitowy
AT90C8534	8	3,3...6,0	0,5	256	1,5	7	6	2		1
AT90LS2323	2	2,7...6,0	0,125	128	4	3	2	1		
AT90LS2343	2	2,7...6,0	0,125	128	4	4	2	1		
AT90LS4433	4	2,7...6,0	0,25	128	4	20	13	2		1
AT90LS8535	8	2,7...6,0	0,5	512	4	32	15	2		1
AT90S1200	1	2,7...6,0	0,0625		12	15	3	1		
AT90S2313	2	2,7...6,0	0,125	128	10	15	10	2		1
AT90S2323	2	4,0...6,0	0,125	128	4	3	2	1		
AT90S2343	2	4,0...6,0	0,125	128	10	4	2	1		
AT90S4433	4	4,0...6,0	0,25	128	8	20	14	2		1
AT90S8515	8	2,7...6,0	0,5	512	8	32	11	2		1
AT90S8535	8	4,0...6,0	0,5	512	8	32	15	2		1
ATmega128	128	4,5...5,5	4	4096	16	53	34	8	+	2
ATmega128L	128	2,7...5,5	4	4096	8	53	34	8	+	2
ATmega16	16	4,5...5,5	0,5	1024	16	32	20	2	+	1
ATmega161	16	4,0...5,5	0,5	1024	8	35	20	3		1
ATmega161L	16	2,7...5,5	0,5	1024	4	35	20	3		1
ATmega162	16	4,5...5,5	0,5	1024	16	35	28	3		2
ATmega162L	16	2,7...5,5	0,5	1024	8	35	28	3		2
ATmega162V	16	1,8...3,6	0,5	1024	1	35	28	3		2
ATmega169	16	4,5...5,5V	0,5	1024	16		23	17		1
ATmega169L	16	2,7...3,6	0,5	1024	8	54	23	17		1
ATmega169V	16	1,8...5,5	0,5	1024	1	54	23	17		1
ATmega16L	16	2,7...5,5	0,5	1024	8	32	20	3	+	1
ATmega32	32	4,0...5,5	1	2048	16	32	19	3	+	1
ATmega64	64	4,5...5,5	2	4096	16	53	34	8	+	2
ATmega64L	64	2,7...5,5	2	4096	8	53	34	8	+	2
ATmega8	8	4,5...5,5	0,5	1024	16	23	18	2	+	1
ATmega8515	8	4,5...5,5	0,5	512	16	35	16	3		1
ATmega8515L	8	2,7...5,5	0,5	512	8	35	16	3		1
ATmega8535	8	4,5...5,5	0,5	512	16	32	20	3	+	1
ATmega8535L	8	2,7...5,5	0,5	512	8	32	20	3	+	1
ATmega8L	8	2,7...5,5	0,5	1024	8	23	18	2	+	1
ATtiny11	1	4...5,5			6	6	4	1		
ATtiny12	1	4...5,5	0,0625		8	6	5	1		
ATtiny15L	1	2,7...5,5	0,0625		1,6	6	8	1(+5)		
ATtiny26	2	4,5...5,5	0,125	128	16	16	11	1		
ATtiny26L	2	2,7...5,5	0,125	128	8	16	11	1		
ATtiny28L	2	2,7...5,5		32	4	11	5	2(+8)		
ATtiny28V	2	1,8...5,5		32	1	11	5	2(+8)		



[illegible]

Adres	Nazwa	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Opis na str.
\$19(\$39)	Zarezerwowane									-
\$18(\$38)	PORTB	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0	100
\$17(\$37)	DDRB	DDRB7	DDRB6	DDRB5	DDRB4	DDRB3	DDRB2	DDRB1	DDRB0	101
\$16(\$36)	PINB	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0	101
\$15(\$35)	Zarezerwowane									-
\$14(\$34)	Zarezerwowane									-
\$13(\$33)	Zarezerwowane									-
\$12(\$32)	PORTD	-	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0	107
\$11(\$31)	DDRD	DDD7	DDD6	DDD5	DDD4	DDD3	DDD2	DDD1	DDD0	107
\$10(\$30)	PIND	PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0	107
\$0F(\$2F)	Zarezerwowane									-
\$0E(\$2E)	Zarezerwowane									-
\$0D(\$2D)	Zarezerwowane									-
\$0C(\$2C)	UDR	Rejestr we/wy układu UART								90
\$0B(\$2B)	USR	RXC	TXC	UDRE	FE	OR	-	-	-	90
\$0A(\$2A)	UCR	RXCIE	TXCIE	UDRIE	RXEN	TXEN	CHR9	RXB8	TXB8	92
\$09(\$29)	UBRR	Rejestr szybkości transmisji układu UART								95
\$08(\$28)	ACSR	ACD	-	ACO	ACI	ACIE	ACIC	ACIS1	ACIS0	97
\$07(\$27)	Zarezerwowane									-
\$06(\$26)	Zarezerwowane									-
\$05(\$25)	Zarezerwowane									-
\$04(\$24)	Zarezerwowane									-
\$03(\$23)	Zarezerwowane									-
\$02(\$22)	Zarezerwowane									-
\$01(\$21)	Zarezerwowane									-
\$00(\$20)	Zarezerwowane									-

Uwagi:

1. By zapewnić kompatybilność z nowymi mikrokontrolerami, zarezerwowane bity powinny być zerowane („0”). Zarezerwowane obszary przestrzeni we/wy nie powinny być nigdy modyfikowane.

2. Niektóre flagi są zerowane poprzez wpisanie logicznej „1” na odpowiadające im pozycje. Mogą do tego służyć rozkazy CBI i SBI. Trzeba jednak pamiętać, że ich zasięg działania jest ograniczony do przestrzeni adresowej we/wy od \$00 do \$1F. O ile więc wyzerowanie flagi np. TXC może być zrealizowane przez bezpośrednie wpisanie do niej „1”:

```
sbi    usr,bxc
```

o tyle wyzerowanie flagi TOVD wymaga już bardziej złożonego działania:

```
in     temp,tifr
ori    temp,0x02
out    tifr,temp
```

3. W kolumnie „Adres” wartości podane w nawiasach oznaczają adresy rejestrów adresowanych jako pamięć SRAM.

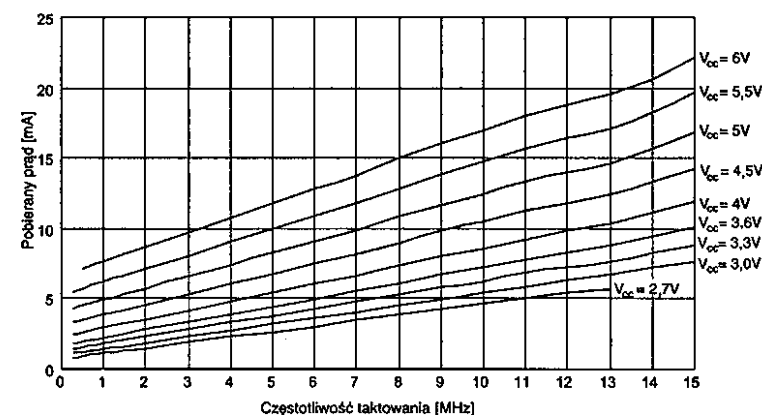
Dodatek C

Wybrane charakterystyki elektryczne i czasowe mikrokontrolera AT90S2313

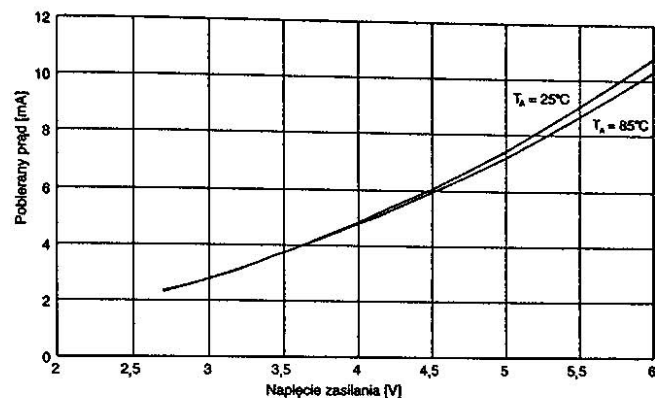
Przedstawione w tym rozdziale charakterystyki odpowiadają typowym konfiguracjom pracy mikrokontrolerów, nie są one testowane na etapie produkcji układu. Założono, że porty we/wy skonfigurowano jako wejściowe z włączonym wewnętrznym podciąganiem. Jako źródło sygnału zegarowego zastosowano generator przebiegu sinusoidalnego z wyjściem *rail-to-rail* (pracującym w całym zakresie napięcia zasilającego).

Pobór prądu jest funkcją zależną od kilku czynników, takich jak: napięcie pracy, częstotliwość pracy, obciążenie portów we/wy, szybkość przełączania portów we/wy, wykonywanego przez mikrokontroler kodu, temperatury otoczenia.

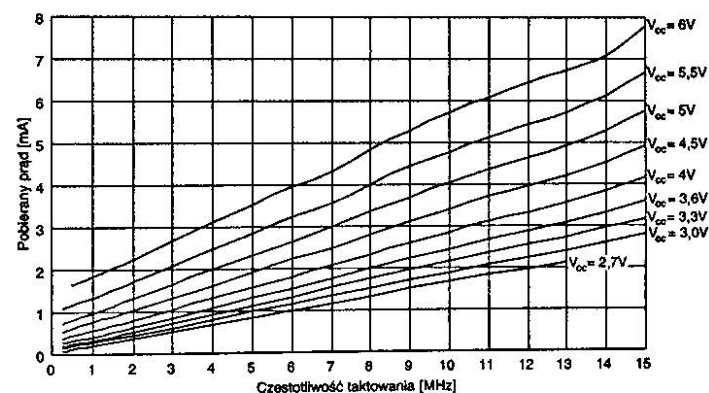
Wartość prądu pobieranego z pojedynczego wyprowadzenia portu może być oszacowana zależnością: $I = C_L \cdot V_{CC} \cdot f$, gdzie V_{CC} – napięcie pracy, f – średnia częstotliwość przełączania portów, C_L – pojemność obciążająca linie portów we/wy.



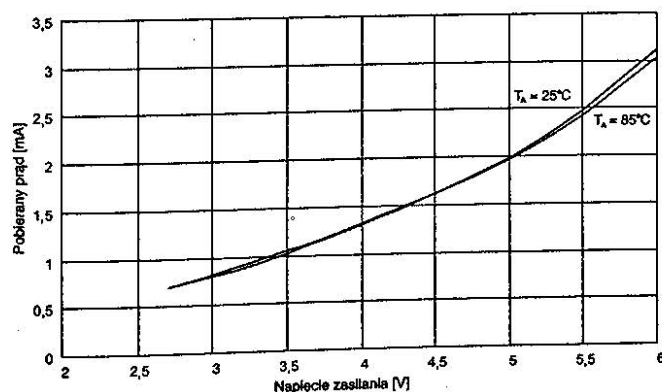
Rys. C.1. Zależność poboru prądu w funkcji częstotliwości sygnału zegarowego (tryb aktywny)



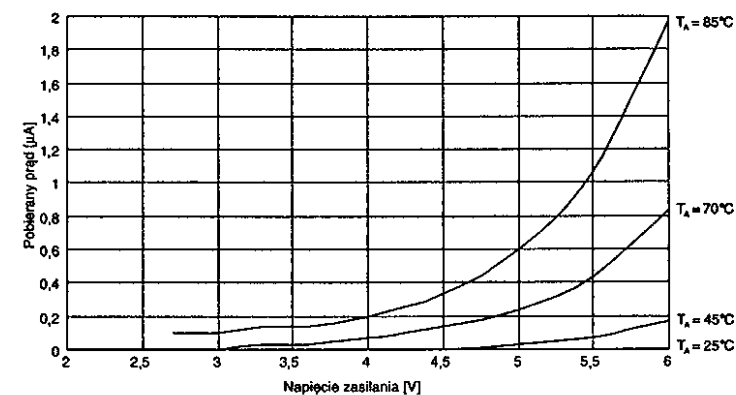
Rys. C.2. Zależność poboru prądu w funkcji napięcia zasilającego (tryb aktywny)



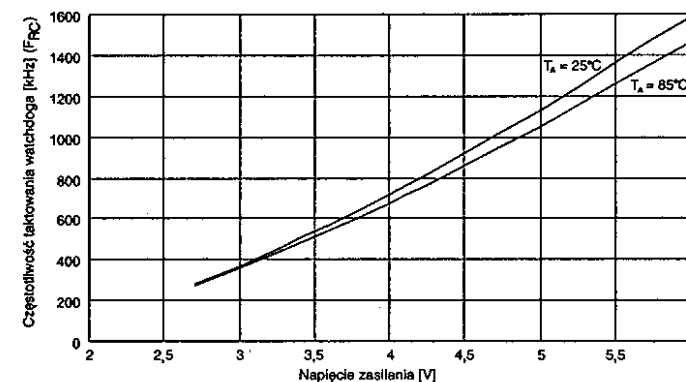
Rys. C.3. Zależność poboru prądu w funkcji częstotliwości sygnału zegarowego (tryb Idle)



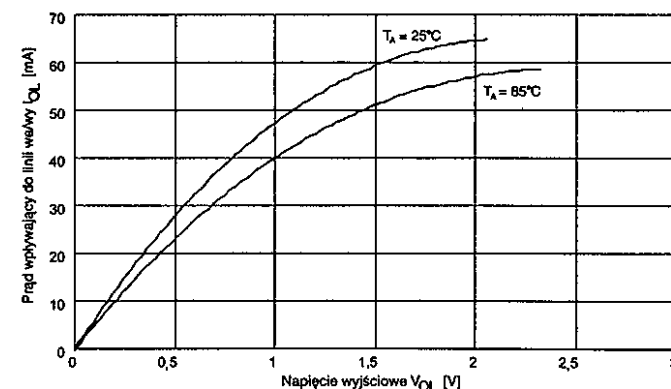
Rys. C.4. Zależność poboru prądu w funkcji napięcia zasilającego (tryb Idle)



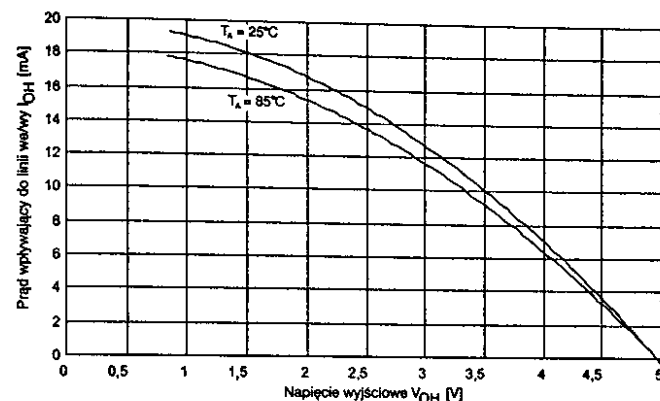
Rys. C.5. Zależność poboru prądu w funkcji napięcia zasilającego (tryb Power-down, watchdog wyłączony)



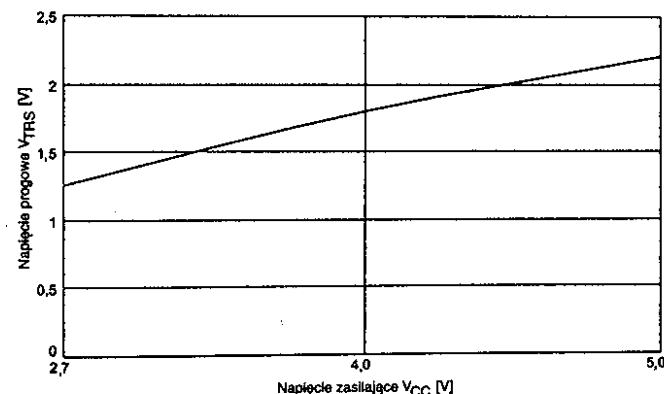
Rys. C.6. Częstotliwość oscylatora watchdoga w funkcji napięcia zasilającego



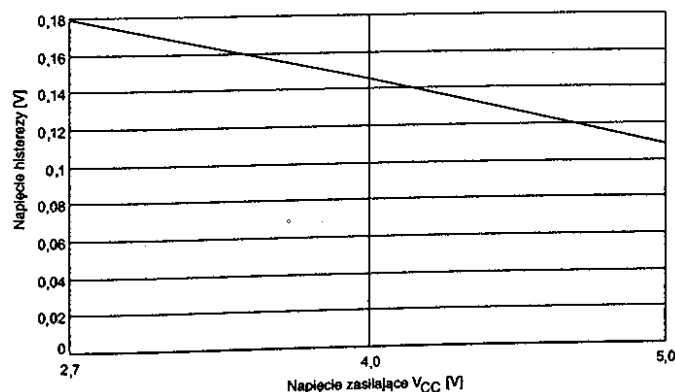
Rys. C.7. Prąd wpływający do linii portu we/wy w funkcji napięcia wyjściowego (napięcie zasilania mikrokontrolera 5 V)



Rys. C.8. Prąd pobierany z linii portu we/wy w funkcji napięcia wyjściowego (napięcie zasilania mikrokontrolera 5 V)



Rys. C.9. Wejściowe napięcie progowe przełączania portów we/wy w funkcji napięcia zasilającego



Rys. C.10. Histereza napięcia wejściowego w funkcji napięcia zasilającego

Dodatek D

D.1. Dopuszczalne parametry elektryczne mikrokontrolera AT90S2313



Parametry zawarte w tabelicy D.1 są parametrami dopuszczalnymi. Ich przekroczenie może spowodować nieodwracalne uszkodzenie mikrokontrolera.

Tab. D.1. Dopuszczalne parametry mikrokontrolera AT90S2313

Zakres temperatury pracy	-55...+125°C
Temperatura przechowywania	-65...+150°C
Napięcie na końcówkach mikrokontrolera za wyjątkiem RESET w odniesieniu do GND	-1,0...V _{CC} +0,5 V
Napięcie na końcówce RESET w odniesieniu do GND	-1,0...13,0 V
Maksymalne napięcie pracy	6,6 V
Prąd DC obciążenia wyprowadzeń I/O	40 mA
Prąd DC na wyprowadzeniach zasilania V _{CC} i GND	200 mA

Tab. D.2. Charakterystyki stałoprądowe dla T_A = -40...+85°C i V_{CC} = 2,7...6,0 V (jeśli nie zaznaczono inaczej)

Oznaczenie	Parametr	Warunek	Min.	Typ.	Maks.	Jedn.
V _{IL}	Napięcie wejściowe w stanie niskim	Z wyjątkiem XTAL1	-0,5		0,3V _{CC} ⁽¹⁾	V
V _{IL1}	Napięcie wejściowe w stanie niskim	XTAL1	-0,5		0,3V _{CC} ⁽¹⁾	V
V _{IH}	Napięcie wejściowe w stanie wysokim	Z wyjątkiem XTAL1 i RESET	0,6V _{CC} ⁽²⁾		V _{CC} +0,5	V
V _{IH1}	Napięcie wejściowe w stanie wysokim	XTAL1	0,7V _{CC} ⁽²⁾		V _{CC} +0,5	V
V _{IH2}	Napięcie wejściowe w stanie wysokim	RESET	0,85V _{CC} ⁽²⁾		V _{CC} +0,5	V
V _{OL}	Napięcie wyjściowe ⁽³⁾ (porty B i D)	I _{OL} = 20 mA, V _{CC} = 5 V I _{OL} = 10 mA, V _{CC} = 3 V			0,5 0,6	V V
V _{OH}	Napięcie wyjściowe ⁽⁴⁾ (porty B i D)	I _{OH} = -3 mA, V _{CC} = 5 V I _{OH} = -1,5 mA, V _{CC} = 3 V	4,3 2,3			V V
I _L	Prąd wejściowy I/O w stanie niskim	V _{CC} = 6 V			1,5	μA
I _H	Prąd wejściowy I/O w stanie wysokim	V _{CC} = 6 V			980	nA
RRST	Rezystancja podciągająca do góry wejścia RESET		100		500	kΩ
R _{VO}	Rezystancja podciągająca do góry wejść portów I/O		35		120	kΩ
I _{CC}	Prąd zasilania	Tryb aktywny, V _{CC} = 3 V, 4 MHz Tryb idle, V _{CC} = 3 V, 4 MHz			3,0 1,0	mA mA
I _{CC}	Tryb Power-down ⁽⁵⁾	WDT aktywny, V _{CC} = 3 V WDT zablokowany, V _{CC} = 3 V		9,0 <1,0	15,0 2,0	μA μA
V _{ACIO}	Wejściowe napięcie offsetu komparatora analogowego	V _{CC} = 5 V V _{IN} = V _{CC} /2			40	mV
I _{ACIK}	Wejściowy prąd upływu komparatora analogowego	V _{CC} = 5 V V _{IN} = V _{CC} /2	-50,0		50,0	nA
t _{ACPD}	Czas propagacji komparatora analogowego	V _{CC} = 2,7 V V _{CC} = 4 V		750,0 500,0		ns

Uwagi:

1. „Maks.” oznacza najwyższą wartość napięcia, która gwarantuje prawidłową interpretację stanu niskiego.
2. „Min.” oznacza najniższą wartość napięcia, która gwarantuje prawidłową interpretację stanu wysokiego.
3. Pomimo tego, że każdy port może być źródłem prądu większego niż podany w warunkach pomiaru (20 mA przy $V_{CC} = 5\text{ V}$ i 10 mA przy $V_{CC} = 3\text{ V}$) w stanie statycznym obowiązują poniższe warunki:

1. Suma prądów wyjściowych I_{OL} ze wszystkich wyjść nie powinna być większa niż 200 mA,
2. Suma prądów wyjściowych I_{OL} dla portów D0...D5 i XTAL2 nie powinna przekraczać 100 mA,
3. Suma prądów wyjściowych I_{OL} dla portów B0...B7 i D6 nie powinna przekraczać 100 mA.

Jeśli prąd I_{OL} przekracza założone warunki pomiarowe, to napięcie V_{OL} może przekraczać podane wartości. Wyprowadzenia nie gwarantują wydajności prądowej większej niż określają to warunki pomiaru.

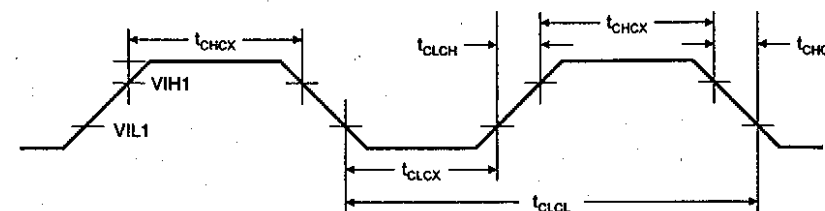
4. Pomimo tego, że każdy port może być źródłem prądu większego niż podany w warunkach pomiaru (3 mA przy $V_{CC} = 5\text{ V}$ i 1,5 mA przy $V_{CC} = 3\text{ V}$) w stanie statycznym obowiązują poniższe warunki:

1. Suma prądów wyjściowych I_{OH} ze wszystkich wyjść nie powinna być większa niż 200 mA,
2. Suma prądów wyjściowych I_{OH} dla portów D0...D5 i XTAL2 nie powinna przekraczać 100 mA,
3. Suma prądów wyjściowych I_{OH} dla portów B0...B7 i D6 nie powinna przekraczać 100 mA.

Jeśli prąd I_{OH} przekracza założone warunki pomiarowe, napięcie V_{OH} może przekraczać podane wartości. Wyprowadzenia nie gwarantują wydajności prądowej większej niż określają to warunki pomiaru.

5. Minimalna wartość napięcia V_{CC} w stanie *Power-down* wynosi 2 V.

D.2. Parametry czasowe zewnętrznego sygnału zegarowego



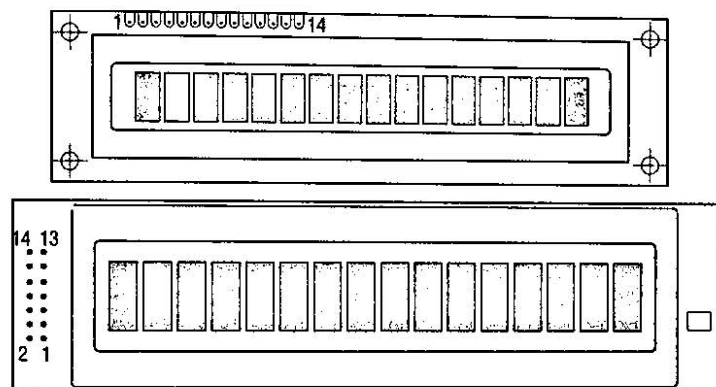
Rys. D.1. Definicje parametrów zewnętrznego sygnału zegarowego

Tab. D.3. Parametry czasowe zewnętrznego sygnału zegarowego

Oznaczenie	Parametr	$V_{CC} = 2,7 \text{ do } 6,0\text{ V}$		$V_{CC} = 4,0 \text{ do } 6,0\text{ V}$		Jedn.
		Min.	Maks.	Min.	Maks.	
$1/t_{CLCL}$	Częstotliwość oscylatora	0	4	0	10,0	MHz
t_{CLCL}	Okres zegara	250,0		100,0		ns
t_{CHCX}	Czas stanu wysokiego	100,0		40,0		ns
t_{CLCX}	Czas stanu niskiego	100,0		40,0		ns
t_{CLCL}	Czas narastania		1,6		0,5	μs
t_{CHCL}	Czas opadania		1,6		0,5	μs

Dodatek E

Wyprowadzenia typowych wyświetlaczy LCD i VFD z interfejsem równoległym



Wypr.	Symbol	Aktywny	Funkcja
1	VSS	L	Minus zasilania
2	VDD	H	Plus zasilania
3	VO/VEE	-	Regulacja kontrastu
4	RS	H/L	Wybór rejestru
5	R/W	H/L	H: odczyt/L: zapis
6	E	H	Sygnal zezwalający (<i>enable</i>)
7	D0	H/L	Linia danych D0
8	D1	H/L	Linia danych D1
9	D2	H/L	Linia danych D2
10	D3	H/L	Linia danych D3
11	D4	H/L	Linia danych D4
12	D5	H/L	Linia danych D5
13	D6	H/L	Linia danych D6
14	D7	H/L	Linia danych D7

Dodatek F

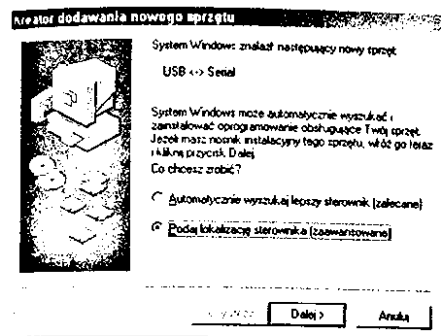
Instalacja sterowników dla układu FT8U232BM w systemie operacyjnym Windows

Fragment artykułu „Konwerter USB <-> RS232” opublikowanego w Elektronice Praktycznej 10/2002

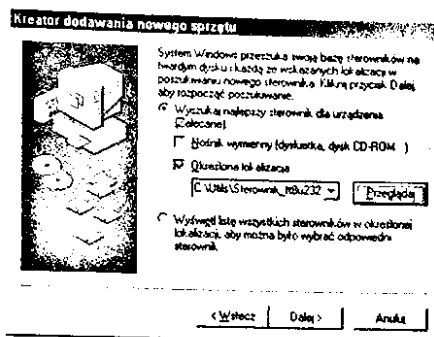
Przed pierwszym podłączeniem interfejsu do komputera musimy się wyposażyć w odpowiedni sterownik. Na stronie producenta (<http://www.ftdi-chip.com/FTDriver.htm>) mamy do dyspozycji sterowniki dla różnych systemów operacyjnych (prototyp był sprawdzany pod Windows 98 oraz Me). Wybieramy odpowiedni i zapisujemy go na dysk. Należy też zwrócić uwagę, że FT8U232 może być obsługiwany (pod Windows) za pomocą kilku sterowników:

- podstawowy sterownik portu szeregowego, który przechwytuje wywołania funkcji API i przekierowuje je do stosu,
- USB (w komunikacji korzystamy wtedy z typowych funkcji Windows obsługi portu, w Delphi bez problemu można użyć dotychczas używane komponenty obsługi COM, np. TComPort lub TRsPort ze strony EP),
- rozszerzony sterownik portu – posiada możliwości jak wyżej, ale dodatkowo obsługuje mechanizm *Plug & Play* (wykrywanie przy starcie Windows nowych urządzeń – jeśli nie budujemy urządzenia, które zareaguje na wywołanie PnP, lepiej tego sterownika nie stosować, gdyż system będzie dłużej czekać na odpowiedź, co spowolni uruchamianie komputera),
- sterownik bezpośredni (*direct*), który wymaga oddzielnych funkcji do obsługi portu (funkcje te są udostępniane w dołączonej bibliotece *dll*) – nie można zatem korzystać z gotowych komponentów, ale za to mamy dostęp do rozszerzonego zestawu operacji (ustawianie nietypowych szybkości transmisji, dostęp do zawartości EEPROM).

Dla potrzeb prezentowanego interfejsu będą nam potrzebne dwa sterowniki: podstawowy oraz *direct*. Po ściągnięciu ze strony FTDI potrzebnych plików rozpakowujemy je do oddzielnych folderów (o dowolnych nazwach) utworzonych na twardym dysku.



Rys. F.1. Windows automatycznie wykrywa dołączenie konwertera



Rys. F.2. W tym oknie podaje się lokalizację sterownika

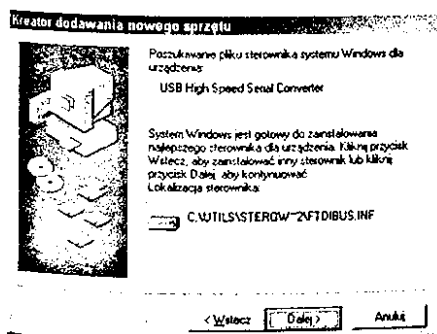
Uruchomienie układu

Teraz podłączamy kablem naszą płytkę do gniazda USB komputera (lub *huba*). Poprawnie zmontowane urządzenie jest od razu wykryte przez system, który prosi o wskazanie lokalizacji sterownika (rys. F.1) – zaznaczamy opcję *Podaj lokalizację sterownika*.

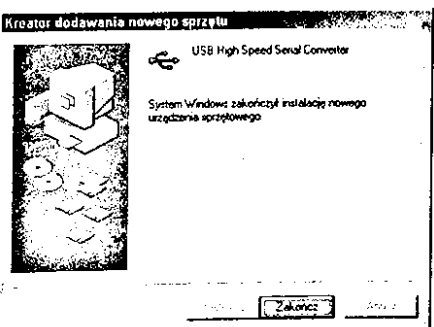
Po potwierdzeniu otwiera się okienko lokalizacji (rys. F.2) – zaznaczamy opcję *Określona lokalizacja* i wybieramy folder, do którego rozpakowaliśmy pliki podstawowego drivera wirtualnego portu COM.

Windows potwierdza znalezienie prawidłowego sterownika (rys. F.3), a następnie informuje o poprawnym zakończeniu instalacji nowego sprzętu (rys. F.4).

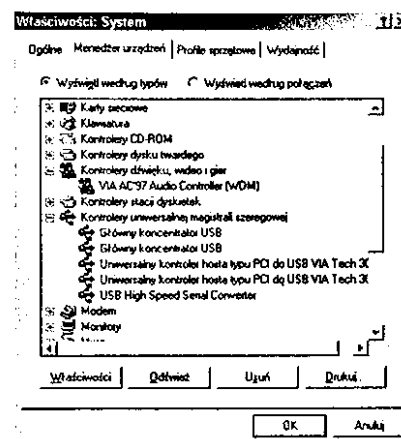
Po załadowaniu sterownika Windows umieszcza interfejs na liście sprzętu i przydziela mu numer portu szeregowego – jest to widoczne w oknie *Menedżera Urządzeń* pod pozycjami: *Kontroler uniwersalnej magistrali szeregowej* oraz *Porty* – rys. F.5 i F.6. Zwróćmy uwagę, że zapis na liście *Menedżera*



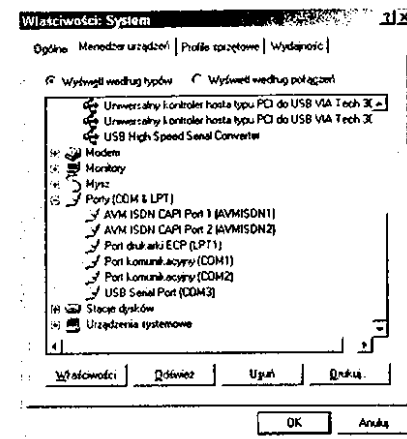
Rys. F.3. Wybór sterownika trzeba zatwierdzić



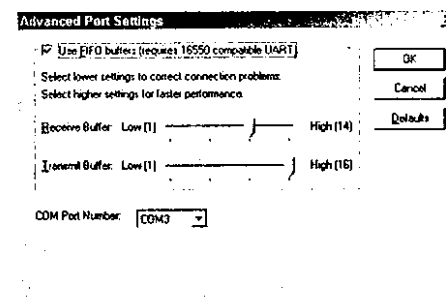
Rys. F.4. Zakończenie instalacji sterowników interfejsu



Rys. F.5. Po dołączeniu interfejsu do komputera i poprawnym zainstalowaniu sterowników kontroler interfejsu USB jest widoczny w oknie menedżera sprzętu Windows



Rys. F.6. Wirtualny port COM w oknie menedżera sprzętu Windows

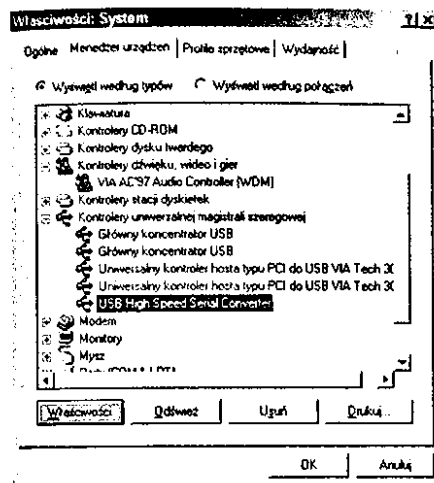


Rys. F.7. Konfiguracja wirtualnego portu szeregowego przebiega w taki sam sposób, jak innych portów COM

Programowanie pamięci EEPROM

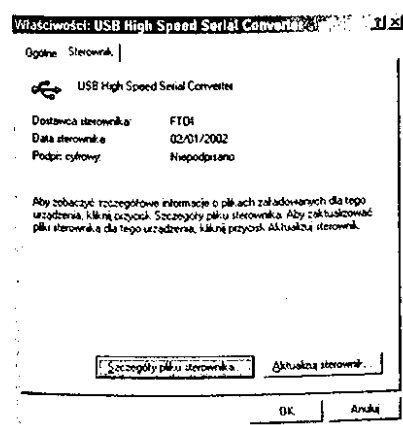
Do zaprogramowania pamięci 93C46 na płytce służy bezpłatny program *fid2xxst.exe* (pobrany ze strony FTDI wraz z opisem w formacie *pdf*). Wymaga on jednak zainstalowania wspomnianego wcześniej sterownika *direct*. Aby nie usuwać używanego do tej pory sterownika portu wirtualnego, wykorzystamy standardowy mechanizm aktualizacji sterowników w Windows. W oknie *Menedżera Sprzętu* wybieramy nasz kontroler USB (rys. F.8) i otwieramy okno jego właściwości, w których uruchamiamy opcję *Aktualizuj sterownik* (rys. F.9).

Przejdziemy wtedy przez szereg okien podobny do pierwszej instalacji – ale jako lokalizację podajemy folder z rozpakowanymi plikami sterownika *di-*



Rys. F.8. Wymianę sterownika rozpoczynamy od wskazania modyfikowanego urządzenia w menedżerze urządzeń

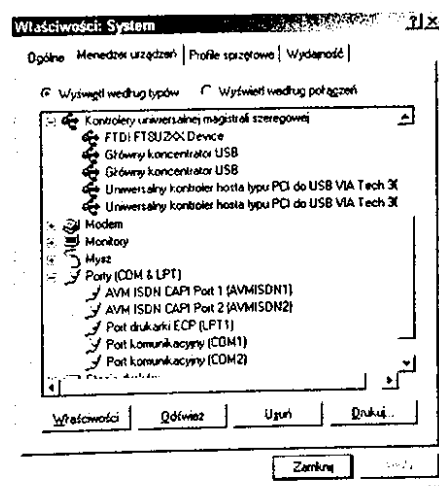
cyjno-komercyjnych) potrzeb – nie warto zmieniać żadnych identyfikatorów poza opisem. Dotyczy to zwłaszcza numerów VID i PID, na podstawie których Windows identyfikuje potrzebny sterownik. Przy zmianie VID/PID należy wyszukać i zmienić również wpisy w odpowiednich plikach *.inf – w przeciwnym razie system nie odnajdzie sterownika i konwerter pozostanie urządzeniem nieznanym dla systemu.



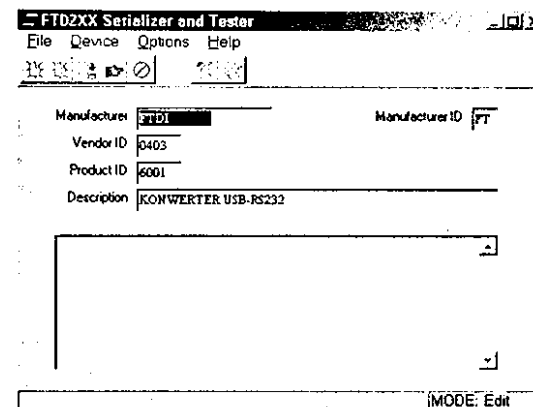
Rys. F.9. Okno aktualizacji (wymiany) sterownika

rect. Po zakończeniu aktualizacji w Menedżerze sprzętu zmienia się opis konwertera a także znika wirtualny port szeregowy (rys. F.10). Od tej chwili do obsługi UART-u musimy używać oddzielnych funkcji z dołączonej biblioteki dll (na stronie znajdziemy szereg przykładów dla popularnych środowisk programistycznych, oczywiście także dla Delphi).

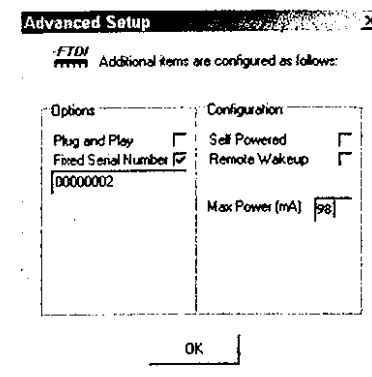
Teraz możemy uruchomić program serwisowy. Po jego uruchomieniu (rys. F.11) należy wypełnić wszystkie pola edycyjne podstawowego opisu urządzenia (dalsze opcje uaktywniają się dopiero po ponownym przejściu do pola *Manufacturer* klawiszem Tab). Tutaj jedna bardzo istotna uwaga – dla własnych warsztatowych (a nie produkcyjno-komercyjnych) potrzeb – nie warto zmieniać żadnych identyfikatorów



Rys. F.10. Menedżer urządzeń po przeinstalowaniu sterownika



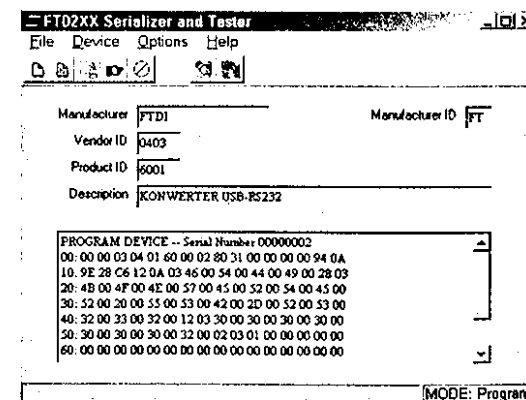
Rys. F.11. Okno główne programu serwisowego



Rys. F.12. Okno zaawansowanych ustawień układu FT8U2xxAM

Po zapamiętaniu nastaw włączamy uaktywnione wtedy ustawienia zaawansowane (*Advanced setup*) – rys. F.12. Umożliwiają one:

- Wybór opcji Plug and Play – pozostawiamy niezaznaczoną.
- Wybór ręcznego lub automatycznego przyznania numeru seryjnego (automatyczny jest wygodny, gdyż zapobiega – przy programowaniu konwerterów na danym komputerze – powtórzeniu się numeru. Dla potrzeb naszego projektu obejmującego tylko kilka płytek został jednak wybrany tryb ręczny).
- Wybór sposobu zasilania: z magistrali albo samodzielne – pozostawiamy zasilanie z magistrali (nie jest to zbyt istotne, gdyż nadrzędny dla określania sposobu zasilania jest poziom na wejściu PWRCTL kostki, co pozwala na sprzętową kontrolę aktualnie używanego źródła zasilania w bardziej rozbudowanych układach).
- Udostępnienie funkcji zdalnego budzenia hosta z trybu *stand-by*. Układ wykonuje to ustawiając parę różnicową w stan K (odwrócona polaryzacja) – dalszym przekazaniem tego sygnału zajmuje się już *hub*.
- Określenie maksymalnego poboru prądu: konwerter (jak wynika



Rys. F.13. Odczyt kontrolny zaprogramowanej pamięci

z not katalogowych użytych układów) pobiera zawsze poniżej 100 mA. Umożliwia to bezproblemowe zasilanie z magistrali USB, gdyż każdy *hub* gwarantuje 100 mA w portach *downstream* (nawet w trakcie enumeracji). W polu edycyjnym została wpisana przykładowo wartość nieco mniejsza.

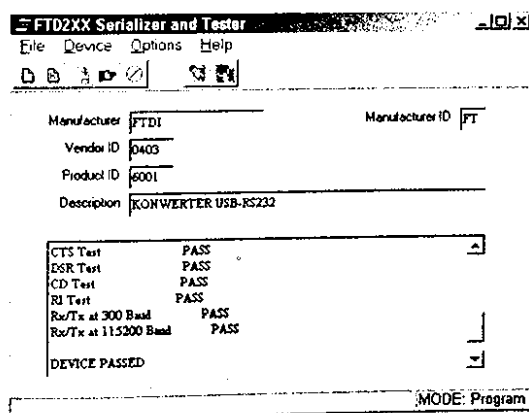
Po wprowadzeniu danych wykonujemy programowanie i kontrolny odczyt EEPROM – wynik tej operacji widzimy na rys. F.13.

Sprzętowy test konwertera

Program serwisowy umożliwia też przeprowadzenie kompletnego testu sprzętowego. Wymagane jest jednak do tego posiadanie w komputerze wolnych portów szeregowych COM1 i COM2 oraz przygotowanie odpowiedniego kabla połączeniowego (dla złącz DB9):

Pin 3 (TXD) – COM2 Pin 2 (RXD)
 Pin 2 (RXD) – COM2 Pin 3 (TXD)
 Pin 7 (RTS) – COM2 Pin 8 (CTS)
 Pin 8 (CTS) – COM2 Pin 7 (RTS)
 Pin 6 (DSR) – COM2 Pin 4 (DTR)
 Pin 5 (GND) – COM2 Pin 5 (GND)
 Pin 4 (DTR) – COM2 Pin 6 (DSR)
 Pin 1 (CD) – COM1 Pin 4 (DTR)
 Pin 9 (RI) – COM1 Pin 7 (RTS)

Uwaga! Test uruchamiamy dopiero po wykonaniu wszystkich podłączeń – w przeciwnym przypadku program się zawiesza (w przypadku Windows 98 razem z systemem). Wynik poprawnie przeprowadzonego testu pokazano



Rys. F.14. Wyniki poprawnego testu sprzętowego konwertera

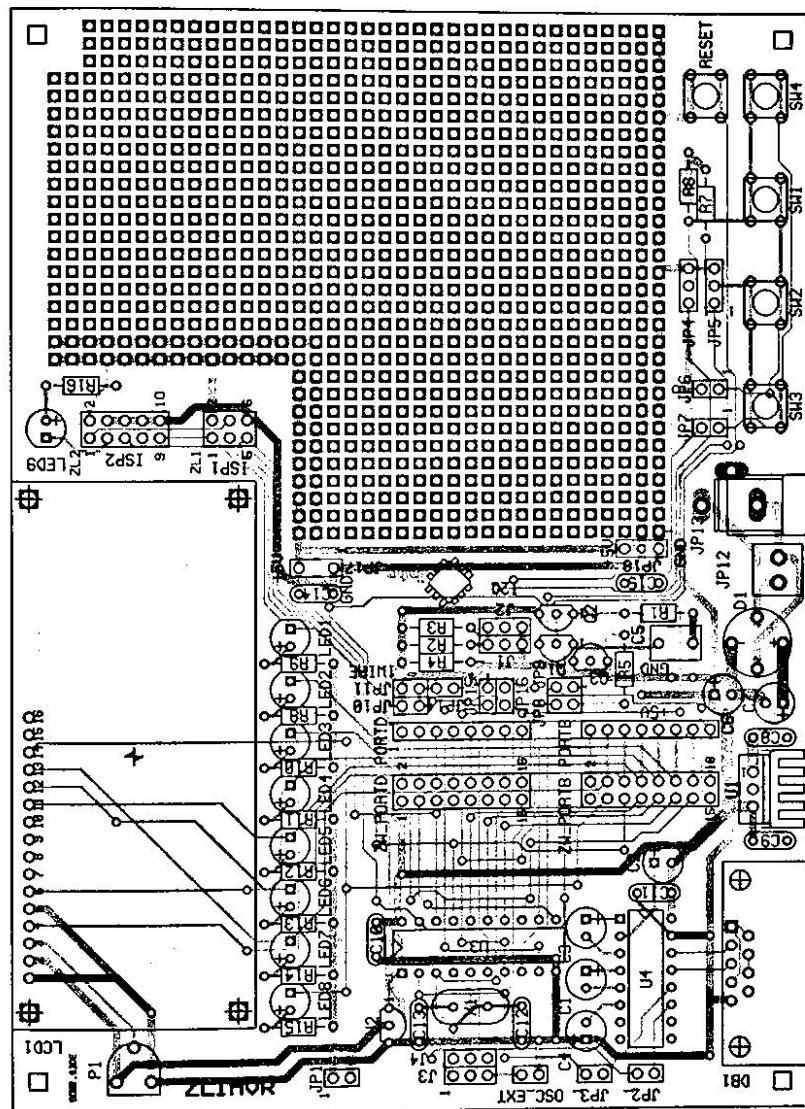
na rys. F.14. Test jest wykonywany za pomocą programu dostępnego na stronie www.ftdichip.com. Znajduje się tam też PDF z dokładnym opisem obsługi, którego tutaj nie będziemy przytaczać. Ten sam program służy do zapisywania współpracującej z FT8U232BM pamięci EEPROM – jest to łatwe dzięki załączonym szczegółowym instrukcjom.

Materiały pomocnicze FTDI zawierają wiele dodatkowych informacji na temat zastosowanego układu oraz zalecenia do programowania transmisji wynikające ze specyfiki transferów *bulk* używanych od strony magistrali USB. Warto przynajmniej częściowo zapoznać się z nimi przed pisanem własnych procedur obsługi komunikacji z użyciem konwertera oraz budową innych urządzeń korzystających z FT8U232.

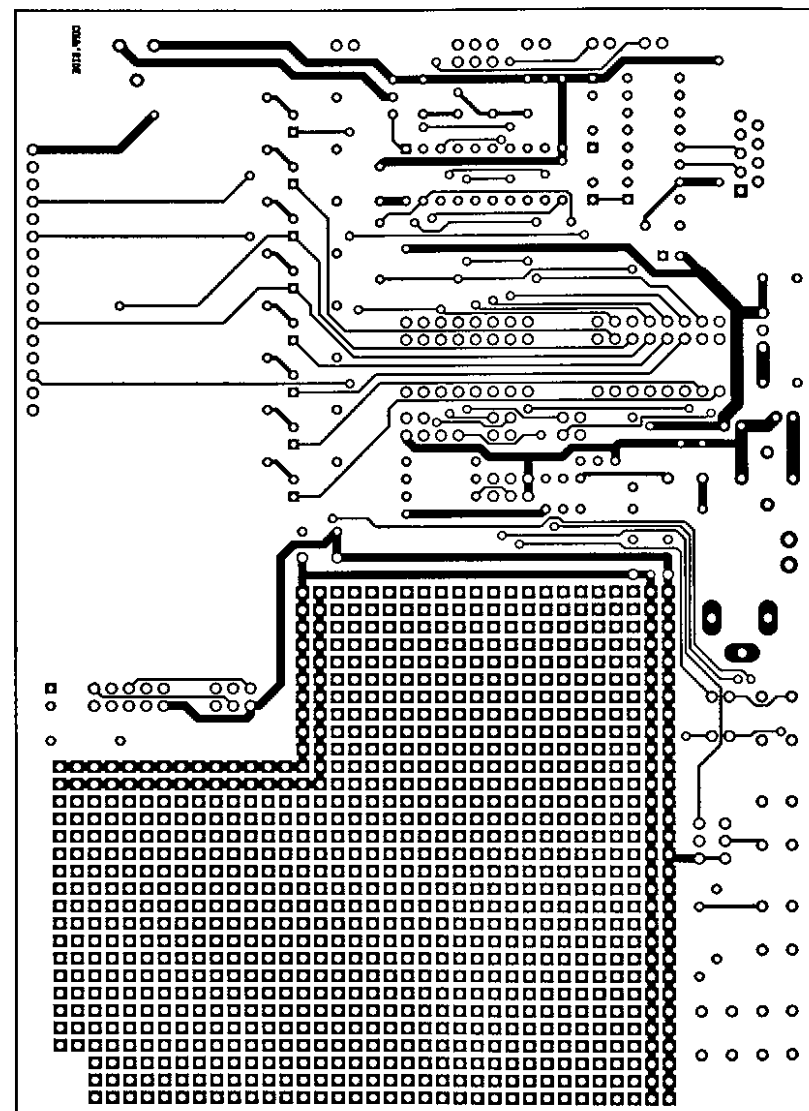
Jerzy Szczesiul, AVT
jerzy.szczesiul@ep.com.pl

Dodatek G

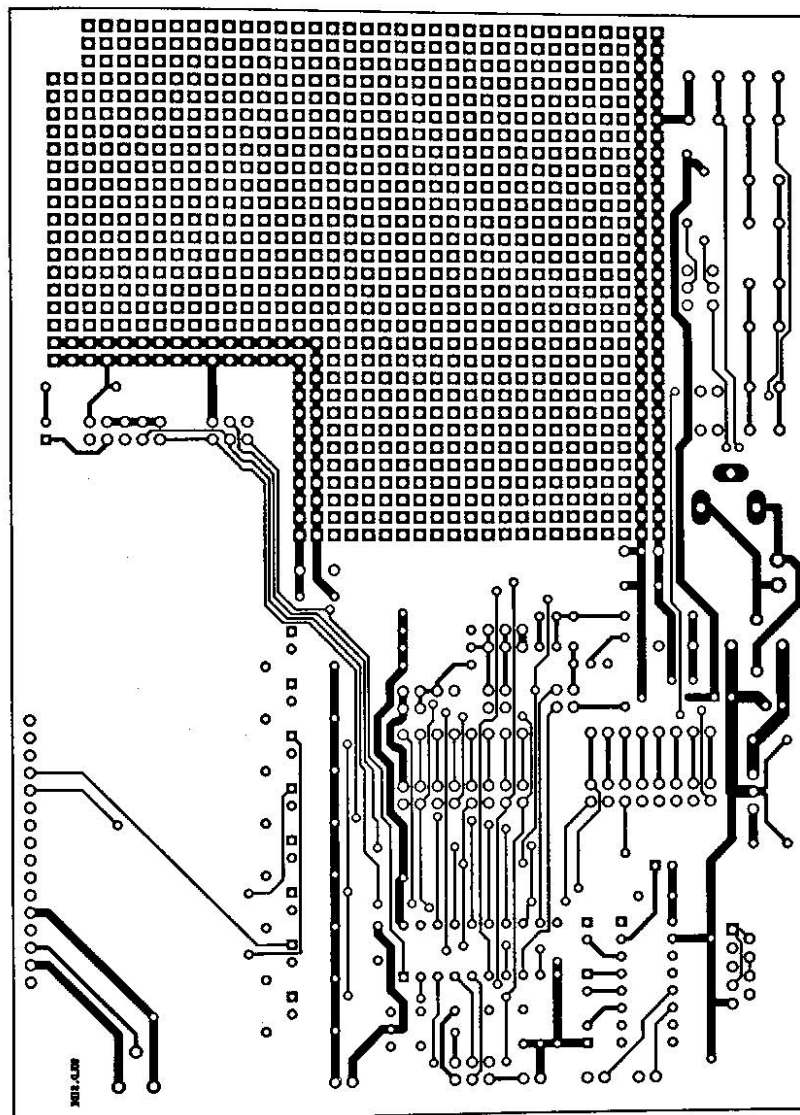
G.1. Płytką drukowaną zestawu ZL1AVR



Rys. G1. Schemat montażowy płytki zestawu ZL1AVR

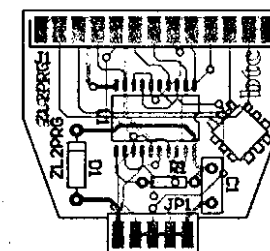


Rys. G2. Strona elementów płytki zestawu ZL1AVR

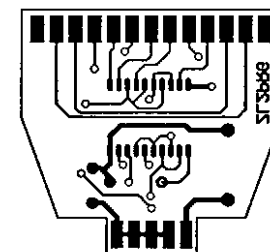


Rys. G3. Strona lutowania płytki zestawu ZL1AVR

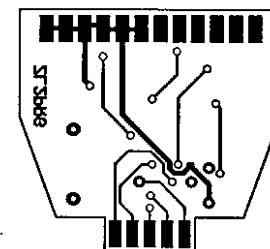
G.2. Płytki drukowane programatora ZL2PRG



Rys. G4. Schemat montażowy płytki programatora ZL2PRG



Rys. G5. Strona elementów płytki programatora ZL2PRG



Rys. G6. Strona lutowania płytki programatora ZL2PRG

Dodatek H

Wybrane adresy internetowe związane z mikrokontrolerami AVR

Internet jest nieprzebraną skarbnicą wiedzy. Nie mogło więc zabraknąć stron o mikrokontrolerach AVR. Można tu znaleźć noty katalogowe i aplikacyjne producenta, liczne strony autorskie prezentujące bardzo zróżnicowane projekty, można zapisać się na tematyczne grupy dyskusyjne, gdzie problemy techniczne rozwiązywane są przez grupowiczów. Wprawdzie rzadko kiedy udaje się uzyskać w ten sposób wyczerpującą odpowiedź, ale często poczynione sugestie otwierają nam przysłowiowe „klapki na oczach”, pozwalając zrealizować z sukcesem własny projekt. Trudno w krótkim rozdziale streścić nawet część najciekawszych stron. Poniżej zostaną wymienione jedynie wybrane adresy z krótkim przedstawieniem poruszanej tematyki:

- <http://www.atmel.com> – strona producenta mikrokontrolerów AVR – firmy Atmel.
- <http://www.avrfreaks.net> – jedna z ważniejszych stron o mikrokontrolerach AVR. Można na niej znaleźć informacje dotyczące samych układów, jak i programów narzędziowych przeznaczonych dla nich. Jest też forum podzielone na kilka grup tematycznych, np. grupa ogólna o AVR-ach, grupa o AVR-GCC, grupa o AVR Studio, grupa o projektach itp. Można tu znaleźć dane katalogowe wszystkich mikrokontrolerów AVR. Jedną z ciekawostek jest prezentowany status produkcyjny każdego elementu, co przy częstych zmianach w programie produkcyjnym Atmela może być bardzo przydatną informacją. Można też znaleźć informacje o licznych narzędziach przeznaczonych dla AVR-ów. Funkcjonująca na tej stronie *Akademia* jest źródłem wiedzy na profesjonalnym poziomie. I wreszcie w zakładce AVR GCC można znaleźć najnowsze wersje kompilatora AVR-GCC.
- <http://users.rcn.com/rneswold/avr/index.html> – strona z dokumentacją kompilatora AVR-GCC.
- <http://www.avr-asm-tutorial.net/index.html> – strona opisująca mikrokontrolery od podstaw z wieloma przykładowymi programami i projektami urządzeń.
- <http://avr.hw.cz/hw/avr/> – ciekawa strona zawierająca zarówno teorię, jak i praktykę. Można tu znaleźć m.in. porównanie poszczególnych typów mikrokontrolerów AVR, kilka schematów programatorów ISP, odnośniki do producentów narzędzi.

- <http://www.mikrocontroller.net/tutorial/index.en.htm> – tutorial mikrokontrolerów AVR.
- http://www.blitzlogic.8m.com/proj_avr.htm – przykłady różnych interfejsów wykorzystujących mikrokontroler AT90S2313, oprogramowanych w języku C.
- http://www.atmel.com/dyn/resources/prod_documents/doc0943.pdf – nota aplikacyjna Atmela AVR910 dotycząca programowania mikrokontrolerów w systemie.
- http://www.volny.cz/eremy/progfun/popis_p.htm – opisy różnych konstrukcji programatorów.
- <http://jaichi.virtualave.net/adaptor-e.htm> – przykłady różnych programatorów przeznaczonych dla mikrokontrolerów AVR.
- <http://www.avrproject.com/> – gotowe projekty na AVR-y.
- http://www.online-club.de/~burkhard-john/avrkreis/index_e.html – gotowe projekty na AVR-y.
- <http://www.myplace.nu/avr/index.htm> – strona Jespera Hansena, twórcy słynnego Yamppa. Znajdziemy tu wiele ciekawych informacji.
- http://www.opticompo.com/emb/support/docs/ieasy2_data.pdf – opis bardzo interesującego projektu modułu internetowego PPP-TCP/IP bazującego na mikrokontrolerze ATmega323L.
- <http://www.hpinfo.tech.ro/> – strona producenta kompilatora języka C Code-VisionAVR z bardzo rozbudowanymi funkcjami bibliotecznymi, wyposażonego w IDE.
- http://www.e-lab.de/AVRco/index_en.html – do pobrania na tej stronie jest darmowy kompilator Pascala dla AVR-ów.
- <http://www.mcselec.com/bascom-avr.htm> – strona producenta kompilatora BASCOM dla AVR-ów, dodatkowo opisy wielu projektów.
- <http://www.fastavr.com/> – strona o konkurencyjnym dla BASCOM-a kompilatorze Basica dla AVR-ów z procedurami obsługi LCD, RTC, I²C, DCF-77, generatora DTMF, odbiornika RC5, itp.
- <http://sourceforge.net/projects/winavr> – trochę różności nie tylko o AVR-ach.
- <http://c2.com/cgi/wiki?ExtremeProgrammingRoadmap> – strona dość luźno związana z mikrokontrolerami AVR, ale z pewnością zainteresuje każdego bardziej zaawansowanego programistę. Porusza tematykę programowania ekstremalnego.
- <http://www.elektronika.qs.pl/linki.html> – bogata lista odnośników do stron producentów elementów elektronicznych.

Dodatek I

Tablice kodów ASCII

I.1. Tablica kodów ASCII

DEC	HEX	CHAR	DEC	HEX	CHAR	DEC	HEX	CHAR
0	00	NUL	42	2A	*	84	54	T
1	01	SOH	43	2B	+	85	55	U
2	02	STX	44	2C	,	86	56	V
3	03	EXT	45	2D	-	87	57	W
4	04	EOT	46	2E	.	88	58	X
5	05	ENQ	47	2F	/	89	59	Y
6	06	ACK	48	30	0	90	5A	Z
7	07	BEL	49	31	1	91	5B	[
8	08	BS	50	32	2	92	5C	\
9	09	HT	51	33	3	93	5D]
10	0A	LF	52	34	4	94	5E	^
11	0B	VT	53	35	5	95	5F	_
12	0C	FF	54	36	6	96	60	a
13	0D	CR	55	37	7	97	61	a
14	0E	SO	56	38	8	98	62	b
15	0F	S1	57	39	9	99	63	c
16	10	DLE	58	3A	:	100	64	d
17	11	XON	59	3B	;	101	65	e
18	12	DC2	60	3C	<	102	66	f
19	13	XOFF	61	3D	=	103	67	g
20	14	DC4	62	3E	>	100	68	h
21	15	NAK	63	3F	?	105	69	i
22	16	SYN	64	40	@	106	6A	j
23	17	ETB	65	41	A	107	6B	k
24	18	CAN	66	42	B	108	6C	l
25	19	EM	67	43	C	109	6D	m
26	1A	SUB	68	44	D	110	6E	n
27	1B	ESC	69	45	E	111	6F	o
28	1C	FS	70	46	F	112	70	p
29	1D	GS	71	47	G	113	71	q
30	1E	RSt	72	48	H	114	72	r
31	1F	US	73	49	I	115	73	s
32	20	SPACE	74	4A	J	116	74	t
33	21		75	4B	K	117	75	u
34	22	•	76	4C	L	118	76	v
35	23	#	77	4D	M	119	77	w
36	24	\$	78	4E	N	120	78	x
37	25	%	79	4F	O	121	79	y
38	26	&	80	50	P	122	7A	z
39	27	'	81	51	Q	123	7B	{
40	28	(82	52	R	124	7C	
41	29)	83	53	S	125	7D	}

Rys. I.1. Znaki i odpowiadające im kody ASCII (część 1)

DEC	HEX	CHAR	DEC	HEX	CHAR	DEC	HEX	CHAR
126	7E	~	172	AC	¼	212	D4	£
127	7F	DEL	173	AD	½	213	D5	¥
128	80	Ç	174	AE	¾	214	D6	¥
129	81	à	175	AF	¸	215	D7	¸
130	82	á	176	B0	■	216	D8	¸
131	83	â	177	B1	■	217	D9	¸
132	84	ã	178	B2	■	218	DA	¸
133	85	ä	179	B3	⋮	219	DB	■
134	86	å	180	B4	⋮	220	DC	■
135	87	ç	181	B5	⋮	221	DD	■
136	88	è	182	B6	⋮	222	DE	■
137	89	é	183	B7	⋮	223	DF	■
138	8A	ê	184	B8	⋮	224	E0	α
139	8B	ë	185	B9	⋮	225	E1	β
140	8C	ì	186	BA	⋮	226	E2	Γ
141	8D	í	187	BB	⋮	227	E3	π
142	8E	Ĳ	188	BC	⋮	228	E4	Σ
143	8F	Ĳ	189	BD	⋮	229	E5	σ
144	90	É	190	BE	⋮	230	E6	α
145	91	æ	191	BF	⋮	231	E7	τ
146	92	Æ	192	C0	⋮	232	E8	Φ
147	93	ø	193	C1	⋮	233	E9	θ
148	94	ŕ	194	C2	⋮	234	EA	Ω
149	95	ð	195	C3	⋮	235	EB	δ
150	96	ó	196	C4	⋮	236	EC	∞
151	97	ù	197	C5	⋮	237	ED	∅
152	98	ÿ	198	C6	⋮	238	EE	€
153	99	Û	199	C7	⋮	239	EF	∩
154	9A	Ü	200	C8	⋮	240	F0	±
155	9B	ø	201	C9	⋮	241	F1	±
156	9C	£	202	CA	⋮	242	F2	±
157	9D	¥	203	CB	⋮	243	F3	≤
158	9E	ƒ	204	CC	⋮	244	F4	≤
159	9F	ſ	205	CD	⋮	245	F5	≤
160	A0	á	206	CE	⋮	246	F6	+
161	A1	í	207	CF	⋮	247	F7	+
162	A2	ó	208	D0	⋮	248	F8	•
163	A3	ù	209	D1	⋮	249	F9	•
164	A4	ñ	210	D2	⋮	250	FA	•
165	A5	Ñ	211	D3	⋮	251	FB	√
166	A6	•				252	FC	η
167	A7	•				253	FD	2
168	A8	•				254	FE	•
169	A9	•				255	FF	
170	AA	•						
171	AB	½						

Rys. I.2. Znaki i odpowiadające im kody ASCII (część 2)

I.2. Znaki zawarte w generatorze znaków sterownika HD44870

4 starsze bity adresu bity	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
xxxx0000	CG RAM (1)			0	1	A	Q	a	q			-	9	3	o	p
xxxx0001	(2)			!	1	A	Q	a	q			。	ア	チ	△	q
xxxx0010	(3)			"	2	B	R	b	r			「	イ	ツ	×	θ
xxxx0011	(4)			#	3	C	S	c	s			」	ウ	テ	ε	ω
xxxx0100	(5)			\$	4	D	T	d	t			、	エ	ト	†	μ
xxxx0101	(6)			%	5	E	U	e	u			・	オ	ナ	1	℃
xxxx0110	(7)			&	6	F	V	f	v			ヲ	カ	ニ	ヨ	ρ
xxxx0111	(8)			'	7	G	W	g	w			フ	キ	ヌ	ラ	q
xxxx1000	(1)			(8	H	X	h	x			イ	ク	ネ	リ	フ
xxxx1001	(2))	9	I	Y	i	y			ウ	ケ	ル	リ	ユ
xxxx1010	(3)			*	:	J	Z	j	z			エ	コ	ン	レ	j
xxxx1011	(4)			+	:	K	L	k	l			オ	サ	ヒ	ロ	*
xxxx1100	(5)			,	<	L	¥	1	l			カ	シ	フ	ワ	†
xxxx1101	(6)			-	=	M	I	m	>			ユ	ズ	△	シ	÷
xxxx1110	(7)			.	>	N	^	n	÷			ヨ	セ	ホ	°	ñ
xxxx1111	(8)			/	?	0	_	o	+			ッ	リ	マ	°	ö

Rys. I.3. Wygląd znaków zapisanych w generatorze znaków sterownika LCD HD44870 (wersja standardowa). Znaki o kodach 00...0Fh (CG RAM1....CG RAM8) użytkownik może samodzielnie zdefiniować

4 starsze bity adresu bity	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
xxxx0000	CG RAM (1)			0	1	A	Q	a	q			-	9	3	o	p
xxxx0001	(2)			!	1	A	Q	a	q			。	ア	チ	△	q
xxxx0010	(3)			"	2	B	R	b	r			「	イ	ツ	×	θ
xxxx0011	(4)			#	3	C	S	c	s			」	ウ	テ	ε	ω
xxxx0100	(5)			\$	4	D	T	d	t			、	エ	ト	†	μ
xxxx0101	(6)			%	5	E	U	e	u			・	オ	ナ	1	℃
xxxx0110	(7)			&	6	F	V	f	v			ヲ	カ	ニ	ヨ	ρ
xxxx0111	(8)			'	7	G	W	g	w			フ	キ	ヌ	ラ	q
xxxx1000	(1)			(8	H	X	h	x			イ	ク	ネ	リ	フ
xxxx1001	(2))	9	I	Y	i	y			ウ	ケ	ル	リ	ユ
xxxx1010	(3)			*	:	J	Z	j	z			エ	コ	ン	レ	j
xxxx1011	(4)			+	:	K	L	k	l			オ	サ	ヒ	ロ	*
xxxx1100	(5)			,	<	L	¥	1	l			カ	シ	フ	ワ	†
xxxx1101	(6)			-	=	M	I	m	>			ユ	ズ	△	シ	÷
xxxx1110	(7)			.	>	N	^	n	÷			ヨ	セ	ホ	°	ñ
xxxx1111	(8)			/	?	0	_	o	+			ッ	リ	マ	°	ö

Rys. I.4. Wygląd znaków zapisanych w generatorze znaków sterownika LCD HD44870 (wersja europejska). Znaki o kodach 00...0Fh (CG RAM1....CG RAM8) użytkownik może samodzielnie zdefiniować

1-wire 311, 374
 A/C 96, 351, 389
 ACD 57, 97
 ACI 97
 ACIC 98
 ACIE 98
 ACISO 98
 ACISI 98
 ACO 97
 ACSR 97
 ADC 137
 ADD 138
 ADIW 139
 Adresowanie bezpośrednie 31
 pośrednie 31
 różnicowe 25
 AINO 17, 96, 100, 103
 AINI 17, 96, 100, 103
 Aktywne zbocze 49, 50, 67, 98
 ALU 13, 20, 26
 AND 140
 ANDI 141
 Architektura 11, 21
 Arytmetyczne 26, 130
 ASCII 299, 330, 367, 397, 413, 445
 Asembler 8, 128, 278, 279, 284, 296
 ASR 142
 Asynchroniczna 85, 365, 409
 ATmega 84, 301, 312
 ATtiny 79, 420
 AVR Studio 279, 284, 290, 353
 AVR-GCC 8, 283, 304, 326, 369
 Bascom 128, 278, 304
 Baud 93
 Baud Rate 87, 93, 95
 BCLR 143
 Bit startu 87, 93, 366
 stopu 86, 93, 365
 zezwożenia 48, 50, 76, 81, 92, 98
 Bity konfiguracyjne 107, 115
 zabezpieczające 114
 BLD 40, 144
 Błąd ramki 89, 91
 BRBC 145
 BRBS 146
 BRCC 147
 BRCS 148
 BREQ 150
 BRGE 151
 BRHC 152
 BRHS 153
 BRID 154
 BRIE 155
 BRLO 156
 BRLT 157
 BRMI 158
 BRNE 159
 Brown-out 84
 BRPL 160
 BRSH 161
 BRTC 162
 BRTS 163
 BRVC 164
 BRVS 165
 BS 118
 BSET 166
 BST 167
 C/A 295, 389
 CBI 38, 98, 170
 CBR 171
 CG ROM 330
 CG RAM 330
 Chip erase 118
 CHR9 87, 93
 CLC 172
 CLH 173
 CLI 174
 CLN 175
 CLR 176
 CLS 177
 CLT 178
 CLV 179
 CLZ 180
 COM 181
 COMIA0 64, 72, 358
 COMIA1 64, 72, 358
 CP 182
 CPC 183
 CPI 184
 CPSE 185
 CRC 365, 380
 CS00 60, 61
 CS01 60, 61
 CS02 60, 61
 CS10 62, 68, 73
 CS11 62, 68, 73
 CS12 62, 68, 73
 CTC1 67

Cykl 47, 72, 76, 83, 114, 123, 313, 352
 DATA 117
 Data Polling 126
 DDRAM 329, 330
 DDRB 39, 99, 101
 DDRD 39, 87, 106, 107
 DEC 186
 DTR 310
 Dyrektywa 281
 EEAR 39, 80
 EECR 39, 80
 EEDR 39, 80
 EEMWE 39, 80
 EEPROM 26, 79, 80, 82, 84, 114, 116, 123
 EERE 82
 EEWB 81
 EIA232 363
 EICALL 187
 EIIMP 188
 ELPM 189
 EOR 191
 FE 89, 91
 Fifo 369, 413
 Flaga przerwania 48, 97
 Flash 16, 21, 26, 84, 113, 116, 123, 285
 FMUL 192
 FMULS 194
 FMULSU 196
 FSTRT 45, 115
 FTDI 408
 Funkcja porównania 65, 67
 Funkcje alternatywne 102, 108
 generator 73, 76, 93
 taktujący 23, 46, 57
 GIFR 39, 50
 GIMSK 39, 47, 49
 Glitch 71, 73
 Globalne zezwolenie 97
 GNU 284, 297, 304
 I2C 311, 388
 ICALL 34, 198
 ICES1 66
 ICF1 53
 ICNC1 66
 ICP 18, 64, 67, 108
 ICR1 39, 64, 71, 108
 ICR1H 39, 64, 71, 108, 313, 352
 ICR1L 39, 64, 71, 108, 313, 352
 Identyfikator 116, 380
 Idle 16, 55, 57, 88, 97
 JMP 34, 200
 IN 201
 INC 202
 Indeksowy 20, 25, 32
 Instalacja 284, 297, 432
 INTO 18, 49, 53, 55, 109
 INT1 18, 49, 53, 55, 109
 Interfejs 85, 300, 310, 331, 363, 374, 388
 INTERRUPT 40, 47, 92, 97
 INTF0 50
 INTF1 50
 ISCO0 55
 ISC01 55
 ISC10 55
 ISC11 55
 ISP1 316
 ISP2 316
 JMP 36, 203
 Kedit 278
 Klawiatura 295, 309
 Komparator analogowy 64, 96, 312, 352
 Kompilator 8, 278, 283, 287
 LCD 311, 328, 343, 374, 396, 408, 412
 LCD1 311
 LD 208
 LDD 206, 209
 LDI 212
 LDS 213
 LED 303, 310, 317, 351, 368
 LF 369, 412
 Licznik programu 26, 44, 136
 Lista rozkazów 128
 Lock bits 114, 122
 LPM 214
 LSL 216
 LSR 217
 Magistrala 11, 20, 328, 389
 Makefile 287, 297
 Mapa pamięci 22, 381
 Master 81, 377, 389
 MCUCR 39, 54
 MISO 17, 100, 102, 123, 315
 MOSI 17, 100, 102, 123, 315
 MOV 218
 MOVW 219
 MUL 220
 MULS 221
 MULSU 222

NEG 224
 NOP 226
 Null-modem 310
 Obszar adresowy 26
 we/wy 204, 206, 209, 213, 263, 265,
 268, 271
 OC1 17, 65, 71, 102, 358
 OCIE1A 51
 OCR1A 70, 72, 102, 358
 OCR1AH 39, 64, 70, 102, 358
 OCR1AL 39, 64, 70, 103, 358
 Odbiornik 88, 90, 110, 369
 OE 117
 Open drain 376, 393
 OR flaga 25, 89, 92
 OR 227
 ORI 228
 OSC_EXT 308
 Oscylator 15, 36
 OUT 38, 229
 Pamięć danych 11, 26, 27, 79, 114, 123, 299
 Pamięć EEPROM 26, 79, 84, 114, 116, 123
 Pamięć nieulotna 26, 123
 Pamięć programu 11, 21, 26, 28, 114, 116,
 122, 189, 198
 PINB 39, 99, 101, 325
 PIND 39, 106, 107, 325
 Pobranie rozkazu 21, 226
 Pointer 21, 41, 136
 PonyProg 301
 POP 230
 Port 17, 99, 229
 szeregowy 363, 408
 PORTB 39, 99
 PORTD 39, 106
 Postinkrementacja 27, 33
 Power-down 16, 55, 57
 Predekrementacja 28, 32, 322
 Preskaler 59, 76, 276
 Prędkość transmisji 39, 94, 95, 367, 408
 Priorytet 23, 42, 48
 Programator 300
 Programowanie równoległe 116
 szeregowe 123
 Próbkowanie 56, 89, 325, 379
 Przechwycenie 53, 66, 68
 Przerwanie 42, 48, 50, 54, 58, 81, 91, 97,
 293
 Przestrzeń adresowa 21
 Przetwornik 96, 295, 312
 Pull-low 101, 109
 Pull-up 17, 99, 101, 106, 108, 324
 PUSH 232
 Push-pull 101, 109
 PWM 52, 64, 71, 357
 PWM10 66, 358
 PWM11 66, 358
 Ramka 365
 RCALL 234
 RDY/BSY 117
 Rdzeń 13, 25
 Rejestr danych 80, 89, 99
 indeksowy 12, 20, 33
 ogólnego przeznaczenia 24
 sterujący 23, 61, 65
 tymczasowy 71
 Reset 18, 43, 44
 RET 236
 RETI 237
 Rezonator 23, 36, 56, 94, 307, 320
 RJMP 239
 ROL 240
 ROR 241
 RS232 85, 310, 363, 408
 RTC 388
 RTS 310
 RXB8 93
 RXC 90
 RXCIE 92
 RXEN 93
 SBC 242
 SBCI 243
 SBI 244
 SBIC 245
 SBIS 246
 SBIW 247
 SBR 248
 SBRC 249
 SBRS 250
 SCK 17, 102, 123, 315
 SCL 389
 SDA 389
 SE 55
 SEC 251
 SEH 252
 SEI 253
 Sekwencja startu 390
 Sekwencja stopu 390

SEN 254
 SER 255
 SES 256
 SET 257
 SEV 258
 SEZ 259
 SIGNAL 326
 Slave 377, 390
 SLEEP 260
 SM 55
 SPI 16, 123, 315
 SPIEN 115
 SPL 39, 41
 SPM 261
 SRAM 16, 23, 25, 27, 38, 57
 SREG 39, 40
 ST 263
 Start-up 115
 STD 265, 268
 Strob 81, 82
 STS 271
 SUB 272
 SUBI 273
 SWAP 274
 Sygnał taktujący 23, 36, 61
 wyzwalający 55, 66, 67
 zegarowy 15, 17, 23, 60
 Sygnatura 116
 Symulacja 287, 297
 Symulator 278, 295
 Synchroniczny 15, 365
 System przerwań 16, 23, 43, 47, 57, 91
 T0 60
 T1 68
 TCCR0 61, 321
 TCCRIA 65
 TCCRIB 66
 TCNT0 62
 TCNT1 68
 TCNT1H 68
 TCNT1L 68
 TEMP 69
 TICIE1 51
 TIFR 52
 Time-out 44, 76, 81, 409
 Timer/licznik 60, 62, 320
 Timing 36
 TIMSK 50
 TOIE0 51
 TOIE1 51
 TOP 72, 360
 TOV0 53
 TOV1 52
 Tryb adresowania 21, 34
 odpytywania 126, 320
 odwrócony 72
 pracy 56, 64, 66, 285, 298, 358, 397
 TST 275
 TWI 388
 TXB8 87
 TXC 87, 91
 TXCIE 92
 TXEN 93
 UART 85, 363, 408
 UBRR 94, 95
 UDR 86, 90
 UDRE 91
 UDRIE 92
 USB 408, 410
 USR 87, 90
 Visual Micro Lab 295
 VMLAB 295
 Watchdog 44, 57, 76, 276
 WDE 77
 WDP0 77
 WDP1 77
 WDP2 77
 WDR 76, 276
 WDTCSR 76
 WDTOE 76
 We/wy 99
 Wektor przerwania 23, 42, 52, 76, 96
 WR 117
 Współczynnik wypełnienia 360
 X 21, 25
 XA0 117
 XA1 117
 XTAL1 23
 XTAL2 23
 Y 21, 25
 Z 21, 25
 Zerowanie 42, 45
 Zestaw uruchomieniowy 304
 ZL1AVR 304
 ZL2PRG 315
 Źródło prądowe 314
 Żądanie obsługi 42, 48, 56