

TOMASZ FRANCUZ

# JĘZYK C

DLA MIKROKONTROLERÓW AVR  
OD PODSTAW DO ZAAWANSOWANYCH APLIKACJI



Przedstawiamy przebojowy duet — język C i mikrokontroler AVR!

Poznaj budowę i podstawy programowania mikrokontrolerów

Dowiedz się, jak do swoich celów wykorzystać język C

Naucz się rozwiązywać rzeczywiste problemy i tworzyć praktyczne rozwiązania



Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentów niniejszej publikacji w jakiekolwiek postaci zabronione. Wykonywanie kopii metodą elektroniczną, fotograficzną, a także kopipowanie książki na nośniku filmowym, magnetycznym, optycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji. Niniejsza publikacja została elektronicznie zabezpieczona przed nieautoryzowanym kopipowaniem, dystrybucją i użytkowaniem. Usuwanie, omijanie lub zmiana zabezpieczeń stanowi naruszenie prawa.

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

Autor oraz Wydawnictwo HELION dołożyli 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 HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Redaktor prowadzący: Michał Mrowiec

Projekt okładki: Studio Gravite / Olsztyn

Obarek, Pokoński, Pazdrijowski, Zaprucki

Wydawnictwo HELION  
ul. Kościuszki 1c, 44-100 GLIWICE  
tel. 32 231 22 19, 32 230 98 63  
e-mail: [helion@helion.pl](mailto:helion@helion.pl)  
WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

[http://helion.pl/user/opinie?jcmikr\\_p](http://helion.pl/user/opinie?jcmikr_p)

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

ISBN: 978-83-246-3732-4

Nr katalogowy: 7042

Copyright © Helion 2011.

- [Poleć książkę na Facebook.com](#)
- [Kup w wersji papierowej](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

# Spis treści

<b>Wstęp .....</b>	<b>11</b>
Kody przykładów .....	12
Schematy .....	12
Wymagane części .....	12
<b>Rozdział 1. Instalacja środowiska i potrzebnych narzędzi .....</b>	<b>15</b>
Instalacja WinAVR .....	16
Instalacja AVR Studio .....	17
Systemy GNU/Linux .....	18
AVR Studio .....	19
Pierwsza aplikacja .....	21
Dodawanie plików do projektu .....	25
Programy narzędziowe .....	27
Linker .....	27
Program avr-size .....	31
Program avr-nm .....	32
Program avr-objcopy .....	33
Program make .....	36
Pliki wynikowe .....	43
Biblioteki .....	46
Projekt biblioteki .....	47
Tworzenie biblioteki .....	48
Dołączanie biblioteki do programu .....	49
Funkcje „przestarzale” .....	50
Nadpisywanie funkcji bibliotecznych .....	50
Usuwanie niepotrzebnych funkcji i danych .....	51
<b>Rozdział 2. Programowanie mikrokontrolera .....</b>	<b>53</b>
Podłączenie — uwagi ogólne .....	53
Problemy .....	55
Programatory ISP .....	55
Budowa programatora .....	56
Programator USBASP .....	59
Kilka procesorów w jednym układzie .....	59
Programatory JTAG .....	60
Programator JTAGICE .....	61
Programator JTAGICE mkII .....	62

Kilka procesorów w jednym układzie .....	62
AVR Dragon .....	63
Programatory HW i równoległe .....	63
Tryb TPI .....	64
Programowanie procesora w AVR Studio .....	64
Programowanie przy pomocy narzędzi dostarczonych przez firmę Atmel .....	65
Program AVRDUDE .....	67
Program PonyProg .....	70
Fusebity i lockbity w AVR-libc .....	70
Lockbity .....	71
Fusebity .....	71
Sygnatura .....	74
Lockbity w AVR-libc .....	74
Fusebity w AVR-libc .....	75
<b>Rozdział 3. Podstawy języka C na AVR .....</b>	<b>77</b>
Arytmetyka .....	77
Proste typy danych .....	77
Arytmetyka stałopozycyjna .....	81
Arytmetyka zmiennopozycyjna .....	87
Operacje bitowe .....	95
Reprezentacja binarna liczb .....	95
Operacja iloczynu bitowego .....	96
Operacja sumy bitowej .....	97
Operacja sumy wyłączającej .....	98
Operacja negacji bitowej .....	99
Operacje przesunięć bitowych .....	100
Zasięg zmiennych .....	100
Zmienne globalne .....	101
Zmienne lokalne .....	102
Modyfikator const .....	103
Wskaźniki .....	104
Tablice .....	109
Funkcje .....	112
Przekazywanie parametrów przez wartość i referencję .....	114
Wywołanie funkcji .....	114
Rekurencyjne wywoływanie funkcji .....	115
Słowa kluczowe .....	116
Operatory .....	116
Instrukcje sterujące .....	120
Preprocesor .....	123
Dyrektywa #include .....	124
Dyrektywy komplikacji warunkowej .....	124
Dyrektywa #define .....	126
Pliki nagłówkowe i źródłowe .....	127
Definicja a deklaracja .....	128
Słowo kluczowe static .....	129
Słowo kluczowe extern .....	130
Dyrektywa inline .....	132
Modyfikator register .....	136
<b>Rozdział 4. Sekcje programu .....</b>	<b>141</b>
Sekcje danych .....	142
Sekcja .text .....	142
Sekcja .data .....	142

Sekcja .bss .....	143
Sekcja .eeprom .....	143
Sekcje zawierające kod programu .....	144
Podsekcje .init[0-9] .....	144
Podsekcje .fini[0-9] .....	145
Sekcje specjalne .....	146
Sekcje tworzone przez programistę .....	146
Umieszczanie sekcji pod wskazanym adresem .....	147
<b>Rozdział 5. Kontrola rdzenia i zarządzanie poborem energii .....</b>	<b>149</b>
Źródła sygnału RESET .....	149
Power-on Reset .....	150
Zewnętrzny sygnał RESET .....	151
Brown-out Detector .....	151
Układ Watchdog .....	152
Zarządzanie poborem energii .....	156
Usypanie procesora .....	157
Wyłączanie układu BOD .....	157
Wyłączanie podsystemów procesora .....	158
Preskaler zegara .....	159
Inne sposoby minimalizowania poboru energii .....	160
<b>Rozdział 6. Dynamiczna alokacja pamięci .....</b>	<b>163</b>
Alokacja pamięci w bibliotece AVR-libc .....	164
Funkcja malloc .....	166
Funkcja calloc .....	166
Funkcja realloc .....	166
Funkcja free .....	168
Wycieki pamięci i błędne użycie pamięci alokowanej dynamicznie .....	169
Jak działa alokator .....	171
Wykrywanie kolizji sterty i stosu .....	172
Metoda I — własne funkcje alokujące pamięć .....	173
Metoda II — sprawdzanie ilości dostępnej pamięci .....	173
Metoda III — marker .....	173
Metoda IV — wzór w pamięci .....	173
Metoda V — wykorzystanie interfejsu JTAG .....	176
<b>Rozdział 7. Wbudowana pamięć EEPROM .....</b>	<b>177</b>
Zapobieganie uszkodzeniu zawartości pamięci EEPROM .....	178
Kontrola odczytu i zapisu do pamięci EEPROM .....	179
Odczyt zawartości komórki pamięci .....	180
Zapis do komórki pamięci .....	180
Dostęp do EEPROM z poziomu AVR-libc .....	181
Deklaracje danych w pamięci EEPROM .....	182
Funkcje realizujące dostęp do pamięci EEPROM .....	183
Inne funkcje operujące na EEPROM .....	185
Techniki wear leveling .....	186
<b>Rozdział 8. Dostęp do pamięci FLASH .....</b>	<b>189</b>
Typy danych związane z pamięcią FLASH .....	190
Odczyt danych z pamięci FLASH .....	191
Dostęp do pamięci FLASH >64 kB .....	192

<b>Rozdział 9. Interfejs XMEM .....</b>	<b>193</b>
Wykorzystanie zewnętrznej pamięci SRAM w programie .....	197
Konfiguracja I — w pamięci zewnętrznej jest tylko sekcja specjalna .....	198
Konfiguracja II — wszystkie sekcje w pamięci zewnętrznej, stos w pamięci wewnętrznej .....	199
Konfiguracja III — w pamięci zewnętrznej umieszczona jest tylko sterta .....	201
Konfiguracja IV — w pamięci zewnętrznej sterta i segment zdefiniowany przez programistę .....	202
Konfiguracja V — w pamięci zewnętrznej znajduje się stos .....	208
Pamięć ROM jako pamięć zewnętrzna .....	208
<b>Rozdział 10. Dostęp do 16-bitowych rejestrów IO .....</b>	<b>211</b>
Dostęp do 16-bitowego rejestru ADC .....	211
Dostęp do 16-bitowych rejestrów timerów .....	213
<b>Rozdział 11. Opóźnienia .....</b>	<b>217</b>
<b>Rozdział 12. Dostęp do portów IO procesora .....</b>	<b>221</b>
Konfiguracja pinu IO .....	221
Manipulacje stanem pinów IO .....	225
Zmiana stanu portu na przeciwny .....	225
Ustawianie linii IO .....	226
Zerowanie linii IO .....	226
Makrodefinicja _BV() .....	227
Użycie pól bitowych .....	227
Synchronizator .....	228
Przykłady praktyczne .....	230
Sterowanie wyświetlaczem 7-segmentowym .....	230
Podłączenie przycisków .....	232
Enkoder obrotowy .....	237
Klawiatura matrycowa .....	242
<b>Rozdział 13. Rejestry IO ogólnego przeznaczenia .....</b>	<b>245</b>
Wykorzystanie innych rejestrów jako GPIO .....	246
<b>Rozdział 14. Przerwania .....</b>	<b>249</b>
Obsługa przerwań .....	251
sei()/cli() .....	254
Atrybut naked i obsługa przerwań w asemblerze .....	254
Modyfikator volatile .....	257
Atomowość dostępu do danych .....	263
Funkcje reentrant .....	266
Przykłady praktyczne .....	268
Wyświetlanie multipleksowane .....	268
Wyświetlanie multipleksowane z regulacją jasności wyświetlacza .....	272
Obsługa przycisków .....	276
Obsługa enkodera .....	279
Klawiatura matrycowa .....	280
<b>Rozdział 15. Przetwornik analogowo-cyfrowy .....</b>	<b>283</b>
Wybór napięcia referencyjnego .....	284
Multipleks .....	285
Przetwornik ADC .....	285
Tryb pojedynczej konwersji .....	286
Tryb ciągłej konwersji .....	287
Wejścia pojedyncze i różnicowe .....	287

Wynik .....	288
Wyzwalacze .....	288
Blokowanie wejść cyfrowych .....	289
Przerwania ADC .....	289
Precyzyjne pomiary przy pomocy ADC .....	290
Nadpróbkowanie .....	291
Uśrednianie .....	292
Decymacja i interpolacja .....	292
Przykłady .....	292
Termometr analogowy LM35 .....	293
Klawisze .....	295
<b>Rozdział 16. Komparator analogowy .....</b>	<b>301</b>
Funkcje dodatkowe .....	302
Blokowanie pinów .....	302
Wyzwalanie zdarzeń timera .....	302
Wybór wejścia komparatora .....	302
Wyzwalanie przetwornika ADC .....	303
<b>Rozdział 17. Timery .....</b>	<b>305</b>
Sygnal taktujący .....	306
Wewnętrzny sygnał taktujący .....	306
Zewnętrzny sygnał taktujący .....	308
Licznik .....	308
Układ porównywania danych .....	309
Wpływ na piny IO .....	309
Moduł przechwytywania zdarzeń zewnętrznych .....	310
Eliminacja szumów .....	311
Komparator jako wyzwalacz zdarzenia ICP .....	311
Tryby pracy timera .....	312
Tryb prosty .....	312
Tryb CTC .....	315
Tryby PWM .....	316
Układ ochronny .....	321
Modulator sygnału wyjściowego .....	322
Miernik częstotliwości i wypełnienia .....	323
Realizacja RTC przy pomocy timera .....	326
Realizacja sprzętowa .....	327
Realizacja programowa .....	328
<b>Rozdział 18. Obsługa wyświetlaczy LCD .....</b>	<b>331</b>
Obsługa wyświetlaczy alfanumerycznych .....	332
Funkcje biblioteczne .....	337
Definiowanie własnych znaków .....	342
Przykład — menu .....	345
Obsługa wyświetlaczy graficznych .....	354
<b>Rozdział 19. Interfejs USART .....</b>	<b>367</b>
Interfejsy szeregowe .....	367
Interfejs USART .....	368
Interfejs USART mikrokontrolera AVR .....	371
Przykłady .....	375
Połączenie mikrokontroler – komputer PC .....	375
RS485 .....	383

<b>Rozdział 20. Interfejs SPI .....</b>	<b>391</b>
Inicjalizacja interfejsu .....	394
Ustawienie pinów IO .....	395
Zegar taktujący .....	396
Procesor w trybie Master SPI .....	396
Procesor w trybie slave SPI .....	397
Przykłady .....	397
Połączenie AVR-AVR .....	397
Połączenie AVR – rejestr szeregowy .....	403
Interfejs USART w trybie SPI .....	408
Taktowanie magistrali SPI .....	409
Tryb pracy SPI .....	409
Format ramki danych .....	409
Konfiguracja interfejsu .....	410
<b>Rozdział 21. Interfejs TWI .....</b>	<b>413</b>
Tryb multimaster .....	416
Inicjalizacja interfejsu .....	417
Procesor w trybie I2C master .....	417
Bity START i STOP .....	417
Podstawowe funkcje do współpracy z I2C .....	418
Współpraca z zewnętrzną pamięcią EEPROM .....	422
Współpraca z zewnętrzną pamięcią FRAM .....	427
Umieszczanie zmiennych w zewnętrznej pamięci EEPROM .....	427
Współpraca z zegarem RTC .....	431
Obsługa ekspandera IO PCF8574 .....	436
Procesor w trybie I2C slave .....	437
Przykład .....	440
<b>Rozdział 22. Interfejs USI .....</b>	<b>447</b>
4-bitowy licznik i zegar .....	447
Przerwania USI .....	448
Zmiana pozycji pinów .....	449
Wykorzystanie interfejsu USI w trybie SPI .....	449
Tryb SPI master .....	451
Tryb SPI slave .....	452
<b>Rozdział 23. Interfejs USB .....</b>	<b>453</b>
Zasilanie .....	454
Sygnały danych .....	455
VID i PID .....	456
Interfejs USB realizowany przy pomocy konwertera .....	458
Interfejs USB realizowany programowo .....	459
Połączenie elektryczne .....	460
Dostęp na PC .....	460
Programowy interfejs USB na AVR .....	461
Sprzętowy interfejs USB .....	464
<b>Rozdział 24. Interfejs 1-wire .....</b>	<b>465</b>
Realizacja master 1-wire na AVR .....	469
Realizacja master 1-wire przy pomocy pinów IO .....	469
Realizacja master 1-wire przy pomocy interfejsu USART .....	472
Wysokopoziomowe funkcje obsługi 1-wire .....	477
Termometr cyfrowy DS1820 .....	480

<b>Rozdział 25. Bootloader .....</b>	<b>483</b>
Pamięć NRWW i RWW .....	483
Bity konfiguracyjne bootloadera .....	485
Konfiguracja lockbitów z poziomu aplikacji .....	486
Programowanie pamięci FLASH .....	487
Wykorzystanie przerwań w kodzie bootloadera .....	489
Usuwanie tablicy wektorów przerwań .....	490
Skrócenie tablicy wektorów przerwań .....	491
Start bootloadera .....	496
Wykorzystanie dodatkowego przycisku/zworki .....	496
Wykorzystanie markerów w pamięci EEPROM .....	497
Oczekiwanie na specjalny znak w wybranym kanale komunikacji .....	498
Start aplikacji .....	499
Współdzielenie kodu aplikacji i bootloadera .....	499
Wywoływanie funkcji bootloadera w procesorach ATMega256x .....	501
Wywoływanie funkcji obsługi przerwań zawartych w kodzie bootloadera .....	505
Współdzielenie zmiennych pomiędzy aplikacją a bootloaderem .....	505
Mikrokontrolery AVR z wbudowanym bootloaderem .....	507
<b>Rozdział 26. Kontrola integralności programu .....</b>	<b>509</b>
Suma kontrolna .....	509
CRC .....	511
Automatyczne generowanie CRC .....	514
<b>Rozdział 27. Bezpieczeństwo kodu .....</b>	<b>517</b>
Metody łamania zabezpieczeń .....	517
Bezpieczne uaktualnianie aplikacji .....	518
Nota AVR231 — AES Bootloader .....	519
Ustawienie bitów konfiguracyjnych .....	524
Przygotowanie aplikacji .....	526
Wczytywanie uaktualnienia .....	527
<b>Rozdział 28. Łączenie kodu w C i assemblerze .....</b>	<b>529</b>
Słowo kluczowe asm .....	530
Typy operandów .....	531
Dostęp do portów IO .....	533
Dostęp do danych wielobajtowych .....	533
Dostęp do wskaźników .....	534
Lista modyfikowanych rejestrów .....	535
Wielokrotne użycie wstawki assemblerowej .....	535
Pliki .S .....	536
Wykorzystanie rejestrów w assemblerze .....	537
Przykłady .....	541
<b>Rozdział 29. Optymalizacja i debugowanie programu .....</b>	<b>543</b>
Optymalizacja programu .....	543
Opcje kompilatora związane z optymalizacją .....	545
Atrybuty optymalizacji .....	548
Debugowanie programu .....	551
Rozpoczęcie sesji debugera .....	553
Zaawansowane sterowanie przebiegiem wykonywanej aplikacji .....	556
<b>Skorowidz .....</b>	<b>559</b>



# **Wstęp**

Mikrokontrolery AVR są dynamicznie rozwijającą się rodziną układów o bardzo szerokich możliwościach. Ze względu na przemyślaną budowę, prędkość działania, bogactwo peryferii i wiele darmowych narzędzi szybko podbiły serca miłośników elektroniki, zarówno hobbystów, jak i osób profesjonalnie zajmujących się mikrokontrolerami. Dla mikrokontrolerów AVR stworzono kompilatory wielu różnych języków programowania, m.in. Basic, Pascal i C. Dla elektronika hobbysty z pewnością szczególnie atrakcyjny jest darmowy pakiet WinAVR, zawierający dostosowaną do AVR wersję jednego z najlepszych kompilatorów języka C — gcc. Do programowania w języku C dodatkowo zachęca architektura AVR i zestaw instrukcji zaimplementowany pod kątem potrzeb kompilatorów języka C. W efekcie program napisany w tym języku jest porównywany pod względem prędkości, a często także objętości do programu napisanego w asemblerze przez dobrego programistę. Jednocześnie wygoda i łatwość pisania, a przede wszystkim prostota uruchamiania i debugowania programu w C jest bez porównania większa niż analogicznego programu napisanego w asemblerze. Powoduje to, że coraz więcej osób jest zainteresowanych programowaniem AVR w języku C. Popularność języka C, szczególnie wśród polskich użytkowników AVR, jest w pewnym stopniu ograniczona z powodu braku dobrej polskojęzycznej literatury oraz krążących mitów o rzekomej trudności nauki tego języka. Poniższa książka ma na celu przybliżyć problematykę programowania mikrokontrolerów AVR w języku C, skupiając się na specyfice programowania mikrokontrolerów. Osoby znające język C z komputerów klasy PC będą mogły szczególnie łatwo „przesiąść się” na programowanie mikrokontrolerów. Należy pamiętać, że książka ta nie ma na celu nauki programowania w języku C, chociaż omówione w niej zostały podstawy języka, umożliwiające rozpoczęcie przygody z C. Jej celem jest pokazanie, jak w możliwie najefektywniejszy sposób pisać programy w języku C na mikrokontrolery, szczegółowo omawia także problematykę pisania aplikacji w C na AVR, wykraczając w tym zakresie poza oklepne kursy internetowe języka C. Omawiana tematyka zilustrowana została licznymi przykładami zawierającymi przydatne funkcje i programy, gotowe do użycia we własnych aplikacjach.

## Kody przykładów

Kody przykładów dołączonych do książki zostały napisane w AVR Studio 4.18 SP3, build 716, a następnie zostały skompilowane kompilatorem avr-gcc wersja 4.3.3, z biblioteką AVR-libc 1.6. Programy te da się skompilować także nowszymi wersjami kompilatora (przykłady z zastosowaniem arytmetyki stałopozycyjnej wymagają kompilatora avr-gcc 4.4.1 z wbudowanym wsparciem dla arytmetyki stałopozycyjnej (kompilator taki dostępny jest w *toolchainie* dostarczonym przez firmę Atmel)). Wraz z przykładami dostarczone zostały pliki projektów AVR Studio, a także w katalogach *default* projektów pliki *Makefile* umożliwiające ich budowę również w środowisku GNU/Linux. Jednak w czasie komplikacji pod tym systemem operacyjnym może być konieczne poprawienie ścieżek prowadzących do katalogów z narzędziami i plikami nagłówkowymi oraz niektórych nazw plików. W przeciwieństwie do MS Windows, w środowisku GNU/Linux rozróżniane są małe/wielkie litery w nazwach plików. Wszystkie przykłady należy rozpakować do jednego katalogu, automatycznie zostaną utworzone podkatalogi zawierające omawiane w książce przykładowe programy.

## Schematy

Realizując układy narysowane na schematach, należy uwzględnić, że przedstawiają one tylko połączenia i fragmenty układu niezbędne do realizacji pokazanego przykładu. Nie zawierają one takich niezbędnych elementów jak podłączenie złącza ISP/JTAG do programowania układu czy zasilania. Są to elementy wspólne dla każdego układu wykorzystującego procesory AVR i dla zaoszczędzenia miejsca zostały one pominięte. Należy także zwrócić uwagę na numerację wyprowadzeń procesora. Ten sam typ procesora występuje w różnych wersjach obudowy, w związku z czym przyporządkowanie sygnałów do wyprowadzeń jest zmienne. **Przed budową układu zawsze należy sprawdzić przyporządkowanie sygnałów do wyprowadzeń w posiadanej wersji procesora.**

## Wymagane części

Do realizacji przykładów i układów elektronicznych pokazanych w książce nie są wymagane żadne zestawy rozwojowe, wystarczą proste części, w większości znajdujące się w szufladzie każdego początkującego elektronika (tabela W.1). Poniżej pokazana została lista wszystkich elementów wykorzystanych w książce, co nie znaczy, że we wszystkie należy się zaopatrzyć. Wszystkie prezentowane układy zostały zmontowane na płytce stykowej. Taka płytka z pewnością przyda się także na dalszych etapach rozwoju przygody z mikrokontrolerami do testowania układów przed zbudowaniem ich w postaci finalnej. W przykładach zostały wykorzystane trzy typy procesorów AVR:

- ◆ ATTiny44 — jako przedstawiciel prostych układów AVR, zawierający uproszczone wersje interfejsów, np. USI.
- ◆ ATMega88 — jako przedstawiciel typowo wykorzystywanych układów AVR, zawierający wszystkie bloki funkcyjne opisane w książce.
- ◆ ATMega128 — na tym procesorze zademonstrowane zostały przykłady związane z wykorzystaniem *bootloadera* — nie jest on niezbędny i można go zastąpić ATTiny88. Omówiony na jego przykładzie został także interfejs pamięci zewnętrznej.

**Tabela W.1.** Wykaz podstawowych części używanych do budowy układów pokazanych w książce. Wartości elementów dyskretnych, szczególnie rezystorów, są przybliżone i można użyć dowolnych rezystorów o wartości zblżonej do podanej w poniżej tabeli

Typ	Liczba	Typ	Liczba
ATMega88	1	BC557 lub inny tranzystor PNP	4
ATMega32U2	1	Wyświetlacz graficzny 128×64 punkty, kompatybilny z KS0108	1
ATTiny461	1	Rezystory 330 om	8
ATMega128	1	Rezystory 1 kOm	4
Wyświetlacz 7-segmentowy, 4 cyfry	1	Rezystory 2,2 kOm	2
Wyświetlacz LCD 16×2 z kontrolerem HD44780 lub kompatybilnym	1	Potencjometr montażowy 10kOm	1
PCF8563	1	Kabel USB	1
PCF8574	1	Kwarc 32 768 Hz	1
Klawiatura membranowa 4×3	1	Trymer 8 – 15pF	1
Płytki stykowe >700 punktów	1	Pamięć szeregowa I2C	1
Przewody połączeniowe do płytki stykowej	1 komplet	74HC595	1

Wybór procesorów jest nieprzypadkowy — należą one do różnych rodzin AVR, posiadając szczególne cechy każdej z nich.



## Rozdział 1.

# Instalacja środowiska i potrzebnych narzędzi

Chyba najlepszą cechą całego środowiska i aplikacji związanych z tworzeniem oprogramowania na mikrokontrolery AVR jest to, że są absolutnie darmowe i dostępne dla wszystkich. Co więcej, w wielu przypadkach dostępny jest także kod źródłowy, w efekcie każdy użytkownik może dodać swoją „cegielkę” w ich rozwoju. Dzięki dostępności gotowych pakietów instalacyjnych instalacja całego środowiska jest szybka i prosta. A przynajmniej taka była...

Ten sielankowy obraz instalacji oprogramowania wspierającego tworzenie programów w języku C został nieco zmącony zapowiedziami firmy Atmel dotyczącymi wprowadzenia nowego środowiska AVR Studio 5, mającego w jednym pakiecie instalacyjnym integrować zarówno kompilator, jak i całe środowisko zintegrowane (IDE). Zapowiedzi te spowodowały porzucenie prac nad projektem WinAVR, który ma się stać integralną częścią nowego środowiska. W efekcie jesteśmy w okresie przejściowym, czyli nie ma nowego AVR Studio 5, a WinAVR nie jest dalej rozwijane. Nie jest to jednak problem, gdyż ostatnia wersja pakietu WinAVR zawierającego kompilator, narzędzia i bibliotekę AVR-libc została wydana stosunkowo niedawno — w styczniu 2010 roku. W międzyczasie, dla osób pragnących „wrażeń”, firma Atmel wydaje kolejne wersje beta pakietu będącego odpowiednikiem WinAVR, które można pobrać ze strony [http://www.atmel.no/beta\\_ware/](http://www.atmel.no/beta_ware/). Pakiet ten zawiera nową wersję kompilatora avr-gcc, najaktualniejszą wersję biblioteki AVR-libc oraz narzędzi. Ciekawą ich cechą, dającą przedsmak przyszłych możliwości, jest wsparcie dla arytmetyki stałopozycyjnej, którego brakuje w wersji gcc dostępnej w ostatnim pakiecie WinAVR oraz w wielu wersjach avr-gcc dostępnych dla środowiska GNU/Linux. Niestety, jak to bywa w przypadku wydań beta różnych programów, w tym pakiecie kryją się także różne bardziej i mniej uciążliwe błędy. Stąd też na chwilę obecną rozsądniej jest raczej bazować na ostatnim wydaniu pakietu WinAVR. Wszystkie przykłady prezentowane w dalszej części książki zostały skompilowane i przetestowane przy pomocy tego pakietu.

# Instalacja WinAVR

Pakiet WinAVR zawiera w sobie wszystkie niezbędne narzędzia, wymagane do skompilowania programu napisanego w języku C, a następnie wgrania go do procesora. Pakiet instalacyjny dla systemu MS Windows można pobrać ze strony domowej projektu — <http://winavr.sourceforge.net/>; w zakładce *Download* znajduje się odnośnik do najnowszej wersji tego pakietu. Cały pakiet instalacyjny to jeden plik o długości ok. 28 MB. Po jego pobraniu należy zainstalować go w komputerze, poprzez uruchomienie pobranego pliku (plik nie jest podpisany cyfrowo, stąd przed jego uruchomieniem zapewne pojawi się ostrzeżenie systemu Windows).

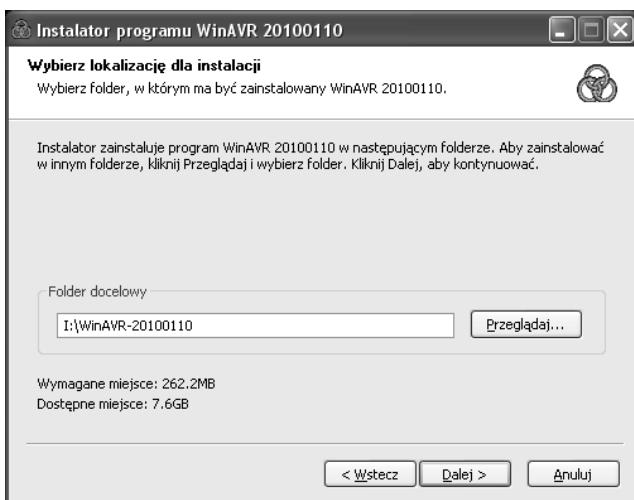


**Uwaga!** Aby poprawnie zainstalować pakiet WinAVR i uaktualnić wskazania zmiennej środowiskowej PATH, wymagane są uprawnienia administracyjne.

Program do swojej pełnej instalacji wymaga 262 MB wolnej pamięci na dysku. Po przejściu pierwszych ekranów informacyjnych pojawia się ekran z możliwością wyboru katalogu docelowego, w którym zostanie zainstalowany pakiet (rysunek 1.1).

**Rysunek 1.1.**

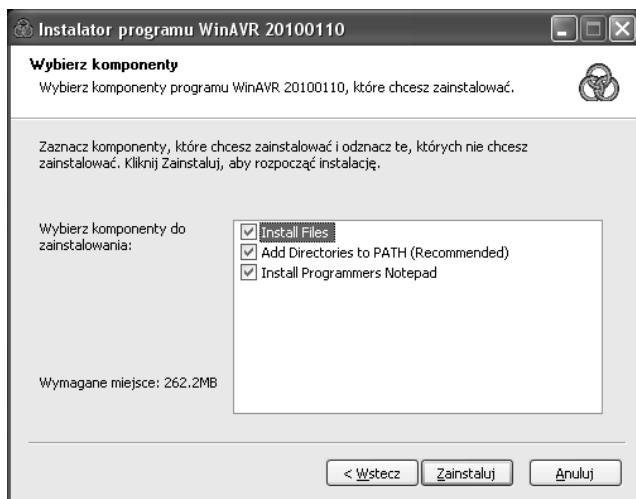
Wybór miejsca docelowej instalacji pakietu WinAVR.  
Może to być dowolny katalog lub katalog proponowany przez instalator. Domyślnie pakiet zostanie zainstalowany na dysku systemowym



Po wybraniu katalogu docelowego pojawiają się kolejne opcje instalacji (rysunek 1.2). Można pozostawić domyślne opcje, ważne, aby zaznaczone pozostały opcje *Install Files* oraz *Add Directories to PATH*. Oznaczenie tej drugiej opcji znacznie utrudni korzystanie z pakietu. Natomiast jeśli będziemy korzystać wyłącznie z AVR Studio, to można nie instalować składnika *Programmers Notepad*. Jest to edytor tekstu zawierający kilka ułatwień dla programistów, lecz kolejne wersje środowiska, wydawane już przez firmę Atmel, nie będą go zawierały. Nie ma więc większego sensu przyzwyczajać się do niego.

Po wybraniu przycisku *Zainstaluj* następuje zainstalowanie pakietu. Po zakończeniu procesu instalacji dysponujemy już wszystkimi niezbędnymi narzędziami, lecz nadal

**Rysunek 1.2.**  
*Opcje instalacji pakietu*



brakuje nam programu, który „spina” wszystko razem, czyli zintegrowanego środowiska programistycznego oraz symulatora. Zarówno symulator (SimulAVR), jak i debugger (GDB) dostarczone razem z pakietem WinAVR nie zasługują na jakąkolwiek uwagę i zostaną pominiete.

## Instalacja AVR Studio

Drugim komponentem ułatwiającym tworzenie oprogramowania dla mikrokontrolerów AVR jest dostarczony przez ich producenta pakiet AVR Studio. Jest to zintegrowane środowisko programistyczne (IDE), zawierające edytor tekstowy (niezwykle prosty, wręcz siermiężny), symulator wszystkich procesorów AVR8, asembler oraz narzędzia umożliwiające programowanie procesorów przy pomocy programatorów opracowanych przez firmę Atmel oraz ich klonów. Środowisko to wspiera także tworzenie i debugowanie aplikacji w języku C, lecz do tego wymaga wcześniej zainstalowanego pakietu WinAVR lub jego odpowiednika wydanego przez firmę Atmel.

Całe środowisko można pobrać po darmowej rejestracji ze strony firmy Atmel (<http://www.atmel.com/>), klikając na zakładki *AVR® 8- and 32-bit*, następnie *Tools & Software* i dalej *AVR Studio 4*. W efekcie pojawi się lista dostępnego do ściągnięcia oprogramowania. Powinniśmy rozpocząć od ściągnięcia i zainstalowania najnowszej wersji *AVR Studio* (w chwili pisania tej książki była to wersja 4.18), a następnie pobrać i zainstalować dostępne uaktualnienia (*service pack*). Po ich zainstalowaniu całe środowisko jest gotowe do pracy.

W tym samym miejscu znajduje się pakiet będący odpowiednikiem WinAVR — AVR Toolchain Installer. Można go pobrać i zainstalować razem z pakietem WinAVR lub jako jedyny pakiet zawierający kompilator i bibliotekę AVR-libc.



Zazwyczaj najnowsza wersja tego pakietu znajduje się do pobrania (i to bez rejestracji) na podanej wcześniej stronie [http://www.atmel.no/beta\\_ware/](http://www.atmel.no/beta_ware/).

Po zainstalowaniu WinAVR/AVR Toolchain oraz AVR Studio nasz komputer gotowy jest do pracy i tworzenia oprogramowania dla mikrokontrolerów AVR.

## Systemy GNU/Linux

Instalacja avr-gcc w systemach GNU/Linux jest równie prosta i zazwyczaj ogranicza się do wydania polecenia zainstalowania odpowiedniego pakietu zawierającego kompilator i niezbędne narzędzia z repozytorium danej dystrybucji. W zależności od dystrybucji dokonuje się tego różnymi poleceniami. O ile instalacja środowiska zawierającego kros-kompilator avr-gcc, biblioteki i podstawowe narzędzia (program make oraz linker) jest prosta, to niestety pod systemem GNU/Linux nie jest dostępny pakiet AVR Studio (ma się to zmienić wraz z wydaniem AVR Studio 5). W efekcie nie mamy do dyspozycji edytora, co akurat nie jest problemem, gdyż w tej roli świetnie spisują się takie środowiska jak np. CodeBlocks ([www.codeblocks.org](http://www.codeblocks.org)). Niestety, pod Linuksem nie dysponujemy także symulatorem procesorów AVR.

Tu z pomocą przychodzi nam emulator środowiska MS Windows — program wine. Emulator ten umożliwia uruchomienie w systemach linuksowych programów przeznaczonych dla MS Windows. Po zainstalowaniu najnowszej wersji wine należyściągnąć dodatkowy plik zawierający elementy niezbędne do poprawnej pracy AVR Studio:

```
$wget http://www.kegel.com/wine/winetricks  
$bash winetricks
```

Po uruchomieniu programu winetricks należy zaznaczyć następujące opcje:

- ◆ corefonts
- ◆ dcom98
- ◆ gdiplus
- ◆ gecko
- ◆ mdac28
- ◆ msxml3
- ◆ vcrun2005
- ◆ allfonts
- ◆ fakeie6

Następnie można przystąpić do instalowania środowiska:

```
$wine AvrStudio4Setup.exe
```

Zainstalowaną aplikację można uruchomić poleceniem:

```
$wine("~/wine/drive_c/Program Files/Atmel/AVR Tools/AvrStudio4/AVRStudio.exe")
```

zwracając uwagę na poprawność ścieżki prowadzącej do pliku *AVRStudio.exe*.

Dodatkowo, jeśli używany programator wymaga portu szeregowego (dotyczy to wielu programatorów z wbudowanym układem FTDI232), należy stworzyć odpowiedni link symboliczny, dzięki któremu możliwe będzie automatyczne znalezienie urządzenia przez program AVR Studio:

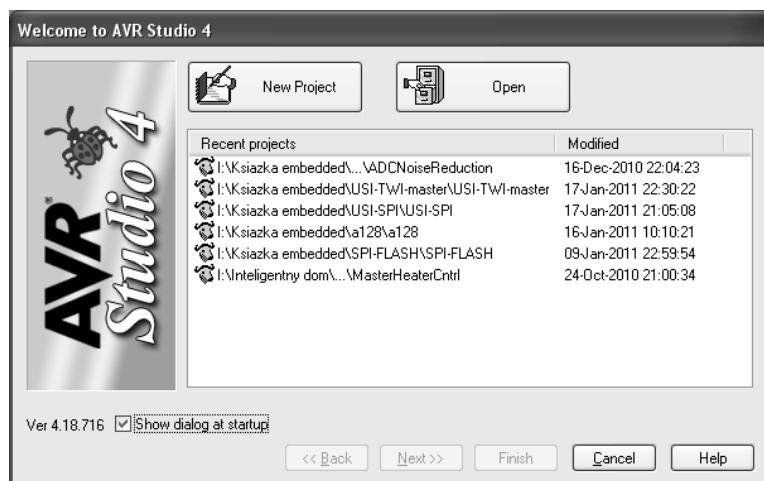
```
#ln -s /dev/ttyUSB0 <home_dir>/.wine/dosdevices/com1
```

## AVR Studio

Po zainstalowaniu kompilatora, narzędzi oraz IDE jesteśmy gotowi do pracy. Aby rozpocząć pisanie pierwszej aplikacji przeznaczonej na mikrokontroler AVR, wystarczy uruchomić program AVR Studio. Powinien nam ukazać się widok jak na rysunku 1.3.

Rysunek 1.3.

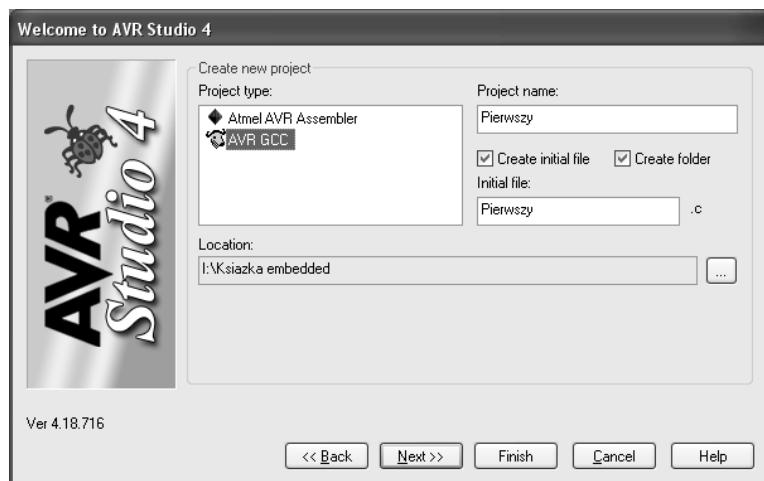
Opcje tworzenia projektu w AVR Studio



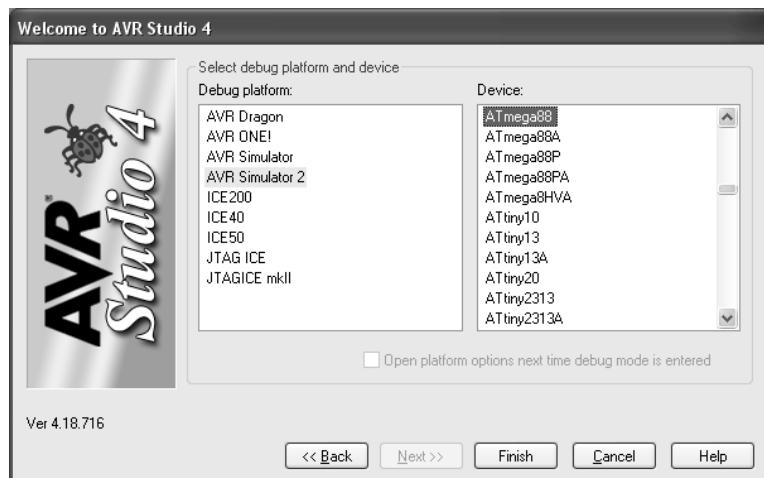
Na liście *Recent projects* oczywiście nie będzie żadnych projektów. Klikamy na przycisk *New Project* i wybieramy tworzenie projektu *AVR GCC*, a w *Project Name* podajemy nazwę projektu. Dzięki opcjom znajdującym się poniżej automatycznie utworzony zostanie główny plik projektu (*Pierwszy.c*) oraz katalog, w którym znajdować będą się wszystkie pliki projektu — rysunek 1.4.

Po określaniu nazwy projektu należy ustalić, na jakim typie mikrokontrolera będzie on wykonywany (rysunek 1.5). Oprócz typu mikrokontrolera (lista po prawej stronie), można wybrać platformę służącą do debugowania pisanej programu. Do wyboru są dwie możliwości: platforma sprzętowa lub programowy symulator.

**Rysunek 1.4.**  
Tworzenie  
nowego projektu  
w AVR Studio



**Rysunek 1.5.**  
Określenie  
typu procesora  
oraz platformy  
symulującej



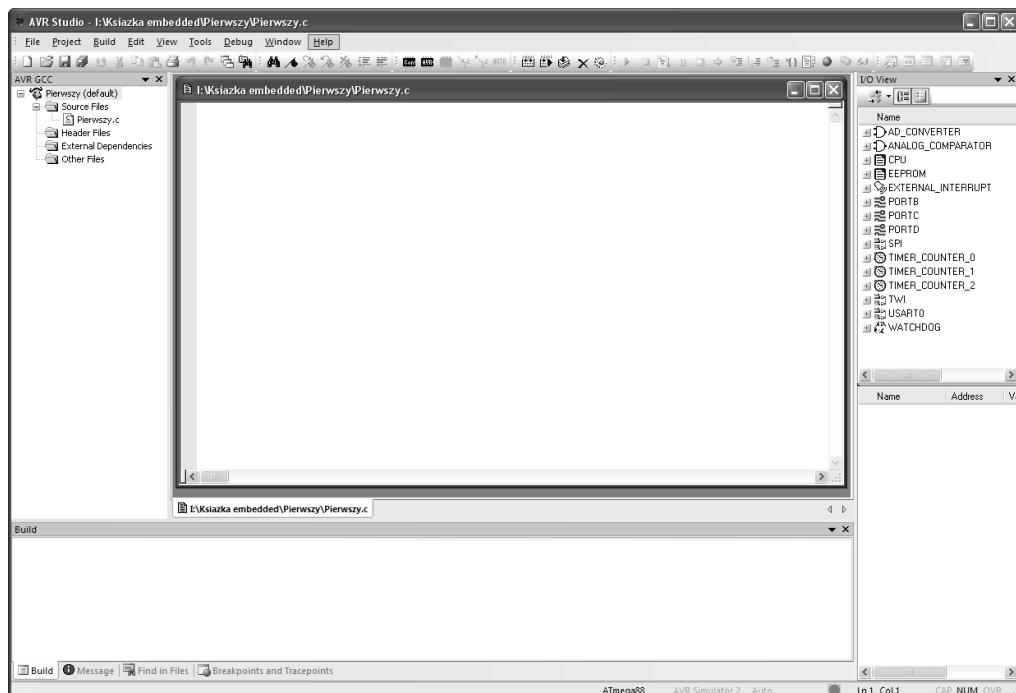
Każda z tych możliwości ma swoje wady i zalety. Zaczniemy od symulacji. Symulator AVR Studio umożliwia symulację dowolnego procesora AVR. Obecnie do dyspozycji mamy dwa symulatory — AVR Simulator oraz AVR Simulator2. AVR Simulator2 jest ciągle rozwijany przez firmę Atmel i na dzień dzisiejszy nie wspiera jeszcze wszystkich opcji symulacji (m.in. bardzo ograniczone jest wsparcie dla plików zawierających przebiegi wejściowe i wyjściowe procesora). Jeżeli jednak jest to możliwe, należy używać go do symulacji układów AVR. Symulator ten wykorzystuje w procesie symulacji pliki opisu sprzętu, z których syntetyzowany jest rdzeń i peryferia mikrokontrolerów AVR, w związku z tym działa on dokładnie tak samo jak prawdziwy procesor AVR. Użycie symulatora umożliwia w dużym stopniu przetestowanie programu, bez jakiegokolwiek dostępu do prawdziwego procesora. Dzięki temu można rozpocząć swoją przygodę z AVR, nie inwestując ani złotówki. Symulator ten nie umożliwia symulacji, poza procesorem, innych układów znajdujących się na płytce. Stanowi to pewien kłopot, lecz w wielu przypadkach nie jest to tak duży problem, jak mogłoby się wydawać.

Drugą kategorią są platformy umożliwiające debugowanie sprzętowe. Umożliwiają one podłączenie się do prawdziwego układu elektronicznego, w którym działa mikrokontroler, a następnie śledzenie wykonywania programu. Stąd też stanowią one największe ułatwienie dla programisty — dzięki nim można na bieżąco śledzić przebieg wykonywania programu oraz sterować wszystkimi wyprowadzeniami procesora. Wadą tego rozwiązania są koszty — nie tylko potrzebny jest gotowy, działający układ docelowy, do którego się podpinamy, ale także, często drogi, programator umożliwiający jednocześnie debugowanie. Tego typu programatory mają znacznie wyższe ceny, rozpoczynające się, dla najprostszych z nich, od ok. 240 zł.

Po wybraniu typu procesora oraz platformy (na razie dla potrzeb dalszych przykładów zostanie wybrany AVR Simulator2) możemy przystąpić do tworzenia aplikacji.

## Pierwsza aplikacja

Po utworzeniu nowego projektu AVR Studio jest gotowe do dalszej pracy. Na początku projekt jest pusty i składa się tylko z jednego, pustego pliku (rysunek 1.6).



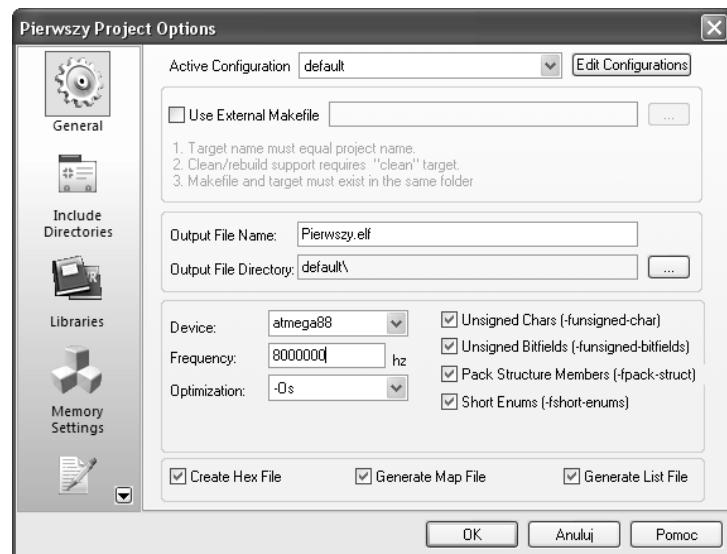
Rysunek 1.6. Początkowy etap tworzenia aplikacji w AVR Studio. Po lewej stronie znajduje się lista plików źródłowych i nagłówkowych tworzących projekt. Po prawej lista dostępnych zasobów sprzętowych procesora. Na środku otwarte jest okno zawierające aktualnie edytowany plik projektu

Możemy już rozpocząć tworzenie pierwszej aplikacji. Zanim jednak do tego przejdziemy, wróćmy jeszcze na chwilę do konfiguracji projektu. Jedną z podstawowych informacji, oprócz typu procesora, dla którego tworzymy aplikację, jest częstotliwość

jego taktowania. Informacja ta służy do precyzyjnego wyliczenia opóźnień czy np. parametrów pracy timerów. Aby ją określić, należy wybrać z menu opcje *Project/Configuration Options*. W efekcie pojawi się okno pokazane na rysunku 1.7.

**Rysunek 1.7.**

Konfiguracja projektu. W polu *Frequency*, które domyślnie jest puste, należy wpisać częstotliwość taktowania procesora



W polu *Frequency* wpisuje się częstotliwość taktowania rdzenia procesora. Domyślnie procesory AVR taktowane są z wewnętrznego generatora RC o częstotliwości typowo 1 lub 8 MHz. Domyślną częstotliwość taktowania można wyczytać z noty katalogowej procesora. W przypadku użycia zewnętrznego rezonatora kwarcowego, podłączonego do wyprowadzeń *XTAL1* i *XTAL2* procesora, zwykle częstotliwość taktowania rdzenia odpowiada częstotliwości zastosowanego rezonatora.

#### Wskaźówka

 W przypadku procesorów posiadających fusebit CKDIV8 domyślnie częstotliwość ta jest dzielona przez 8 i taką wartość należy wpisać w polu *Frequency*, chyba że fusebit CKDIV8 został skasowany (jego wartość wynosi 1).

Pozostałe opcje projektu należy pozostawić bez zmian, ich wartości domyślne są właściwe praktycznie dla wszystkich projektów. Szczegółowo znaczenie pozostałych opcji tego okna zostanie stopniowo wyjaśnione w dalszych rozdziałach książki.

Nadszedł wielki moment — napisanie pierwszej aplikacji. Tak jak dla komputerów PC naukę programowania zwykle rozpoczyna się od już kultowego programu, którego celem jest wyświetlenie napisu „hello, world”, tak w świecie mikrokontrolerów odpowiednikiem jest program powodujący mruganie diody LED. W oknie *Pierwszy.c* wpiszmy więc następujący kod:

```
#include <avr/io.h>
#include <util/delay.h>

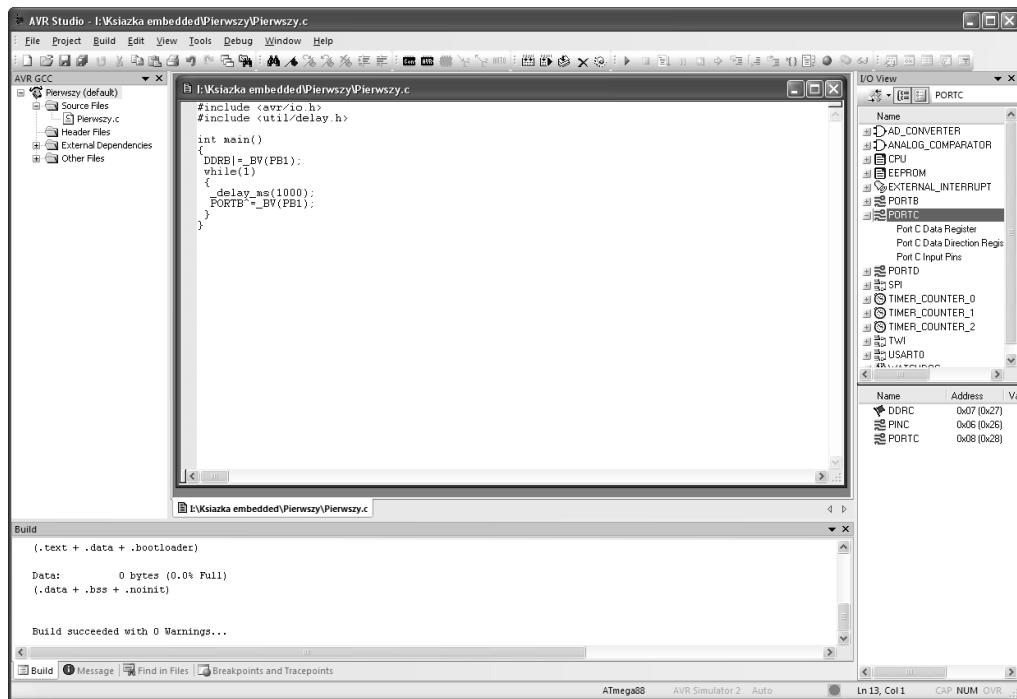
int main()
{
```

```

DDRB|=BV(PB1);
while(1)
{
    _delay_ms(1000);
    PORTB^=BV(PB1);
}
}

```

Na razie pominiemy znaczenie poszczególnych poleceń, zostaną one wyjaśnione w kolejnych rozdziałach. Mamy już kod pierwszego programu, aby jednak móc zobaczyć efekty jego działania, należy go skompilować. W tym celu należy wybrać z menu opcję *Build/Build* lub po prostu naciśnąć klawisz *F7*. Zakładając, że nie popełniliśmy błędu przy przepisywaniu powyższego kodu, powinniśmy uzyskać efekt jak na rysunku 1.8.



**Rysunek 1.8.** Efekt komplikacji pierwszego programu...

Jak widzimy, w dolnej części ekranu, w oknie *Build*, pojawiły się informacje o skompilowanym programie. Końcowy napis:

Build succeeded with 0 Warnings...

powinien nas szczególnie ucieszyć, gdyż świadczy on o prawidłowej komplikacji programu. Przewijając zawartość tego okna, powinniśmy zobaczyć taką oto treść:

```

Build started 19.1.2011 at 20:44:07
avr-gcc -mmcu=atmega88 -Wall -gdwarf-2 -std=gnu99 -DF_CPU=8000000UL -Os -
→funsigned-char -funsigned-bitfields -fpack-struct -fshort-enums -MD -MP -MT
→Pierwszy.o -MF dep/Pierwszy.o.d -c ..../Pierwszy.c
avr-gcc -mmcu=atmega88 -Wl,-Map=Pierwszy.map Pierwszy.o      -o Pierwszy.elf

```

```
avr-objcopy -O ihex -R .eeprom -R .fuse -R .lock -R .signature Pierwszy.elf  
→Pierwszy.hex  
avr-objcopy -j .eeprom --set-section-flags=.eeprom="alloc,load" --change-section-  
→lma .eeprom=0 --no-change-warnings -O ihex Pierwszy.elf Pierwszy.eep || exit 0  
avr-objdump -h -S Pierwszy.elf > Pierwszy.lss
```

```
AVR Memory Usage  
-----  
Device: atmega88  
  
Program:      116 bytes (1.4% Full)  
(.text + .data + .bootloader)  
  
Data:          0 bytes (0.0% Full)  
(.data + .bss + .noinit)
```

Build succeeded with 0 Warnings...

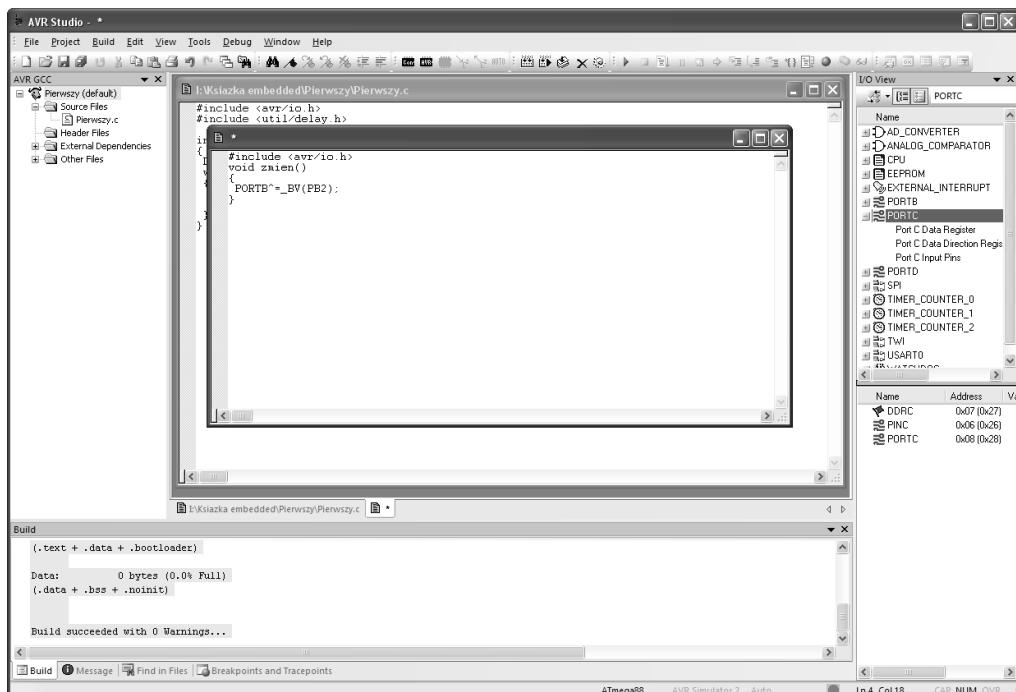
W pierwszej części widać kolejne wywołania kompilatora `gcc` i innych programów narzędziowych, których celem jest komplikacja programu i utworzenie plików, które zostaną wykorzystane do zaprogramowania procesora. Poniżej znajdują się niezwykle ważne informacje o kodzie wynikowym. W sekcji *AVR Memory Usage* podana jest ilość pamięci zajętej przez program (w tym przypadku jest to 116 bajtów) oraz ilość pamięci SRAM zajętej przez zmienne globalne, statyczne oraz prealokowane w pamięci SRAM struktury danych (w naszym przypadku jest to 0 bajtów). Obok wartości bezwzględnych w nawiasach podane są wartości określające, jaki stanowią one procent całej pamięci danego typu mikrokontrolera. Powody do zmartwienia pojawiają się w momencie, kiedy procent zajętej pamięci FLASH staje się duży, natomiast jeśli przekroczy on wartość 100%, to znaczy, że program jest zbyt duży i nie da się go zmieścić w pamięci FLASH wybranego mikrokontrolera. W takiej sytuacji mamy dwie możliwości — albo lepiej napisać nasz program i zoptymalizować go tak, aby kod wynikowy był krótszy (praktycznie zawsze jest to możliwe), albo zmienić procesor, na taki, który dysponuje większą ilością pamięci. Interpretacja pola *Data*, pokazującego zużycie pamięci SRAM, nie jest taka prosta i zostanie przedstawiona w kolejnych rozdziałach. Warto zapamiętać, że jeśli wartość ta oscyluje w granicach 80 i więcej %, to program z pewnością nie będzie działał poprawnie. Jeśli jest niższa, to najprawdopodobniej będzie działał, ale w zależności od tego, jak alokowane są zmienne, nie daje to absolutnych gwarancji, że programowi nie zabraknie pamięci SRAM.

Po skompilowaniu programu należy wczytać go do procesora, co umożliwi jego wykonanie. Jednak aby przekonać się, że program działa, możemy wykorzystać wbudowany w AVR Studio symulator. Na etapie tworzenia projektu do symulacji wybrałyśmy AVR Simulator2 (lecz w każdej chwili można ten wybór zmienić, wybierając opcję *Debug/Select Platform and Device*). Informacje o debugowaniu i symulacji programu znajdują się w rozdziale 28. Osoby niecierpliwe (i nieposiadające pod ręką programatora i procesora) mogą do niego na chwilę zaglądać, aby dowiedzieć się, jak przy pomocy AVR Studio przetestować działanie powyższego programu.

W tym momencie nasz pierwszy program jest skończony i skompilowany. W efekcie uzyskaliśmy w katalogu projektu i jego podkatalogu *Default* pliki *Pierwszy.elf* oraz *Pierwszy.hex*, zawierające kod aplikacji. Pliki te posłużą programatorowi do wczytania aplikacji do pamięci FLASH procesora.

## Dodawanie plików do projektu

Przed przystąpieniem do pisania bardziej skomplikowanych programów musimy opanować jeszcze jedną umiejętność — dodawanie do projektu kolejnych plików. Większość aplikacji pisanych w języku C składa się z wielu plików źródłowych i nagłówkowych, w których umieszczone są funkcje rozwiązuujące jakiś cząstkowy problem. W AVR Studio można dodawać istniejące pliki lub tworzyć nowe, a następnie je dodawać do tworzonego projektu. Aby utworzyć nowy plik, należy wybrać opcję menu *File/New File*. Po jej wybraniu pojawi się nowe, puste okno, pozbawione tytułu (rysunek 1.9).



Rysunek 1.9. Utworzenie nowego okna. Gwiazdka w jego nazwie oznacza, że jego zawartość nie została zapisana na dysku

Po utworzeniu nowego okna należy zapisać je na dysku przy pomocy opcji *File/Save As*. W tym momencie można wybrać lokalizację zapisywanej pliku oraz jego rozszerzenie. Pliki najlepiej jest zapisywać w katalogu, w którym znajduje się tworzony projekt, lub jego podkatalogach. Bardzo ważne jest rozszerzenie nazwy pliku. Jest ono rozpoznawane przez kompilator, który na jego podstawie określa typ pliku:

- ◆ *.h* — pliki nagłówkowe języka C,
- ◆ *.c* — pliki źródłowe języka C,
- ◆ *.C* lub *.cpp* — pliki źródłowe języka C++,
- ◆ *.S* — pliki źródłowe asemblera.

Po zapisaniu pliku na dysku możemy go dodać do projektu. W tym celu klikamy prawym przyciskiem myszy na rozwijalną listę po lewej stronie ekranu. W zależności od typu pliku klikamy na *Source Files* w przypadku plików źródłowych lub *Header Files* w przypadku plików nagłówkowych. Wybieramy opcję *Add Existing Source Files* i dodajemy wcześniej zapisany plik. Od tego momentu plik znajdować się będzie na rozwijalnej liście. Plik taki w każdej chwili możemy otworzyć w celu dalszej edycji, klikając dwukrotnie na jego nazwę.

Po dodaniu/usunięciu pliku z listy należy zapisać projekt na dysku, wybierając opcję *Project/Save Project*.

Teraz możemy ponownie skompilować nasz projekt. W tym celu naciskamy klawisz *F7*, w oknie *Build* wśród różnych komunikatów znajdujemy:

```
avr-gcc -mmcu=atmega88 -Wall -gdwarf-2 -std=gnu99 -DF_CPU=8000000UL -Os -
→ funsigned-char -funsigned-bitfields -fpack-struct -fshort-enums -MD -MP -MT
→ Pierwszy.o -MF dep/Pierwszy.o.d -c ..../Pierwszy.c
avr-gcc -mmcu=atmega88 -Wall -gdwarf-2 -std=gnu99 -DF_CPU=8000000UL -Os -
→ funsigned-char -funsigned-bitfields -fpack-struct -fshort-enums -MD -MP -MT
→ drugi.o -MF dep/drugi.o.d -c ..../drugi.c
avr-gcc -mmcu=atmega88 -Wl,-Map=Pierwszy.map Pierwszy.o drugi.o      -o Pierwszy.elf
```

Jak widać, dodany do projektu plik *drugi.c* został skompilowany, a następnie zlinkowany z innymi plikami projektu.

### Wskaźówka

 Dodanie pliku do projektu powoduje automatyczne wygenerowanie nowego pliku *Makefile*, zawierającego instrukcje umożliwiające zbudowanie pliku wynikowego, zawierającego skompilowany program.

Dzięki temu w prosty sposób można tworzyć skomplikowane projekty, składające się z dowolnej liczby plików źródłowych i nagłówkowych. AVR Studio samo zadba o właściwą komplikację plików, a następnie ich konsolidację (linkowanie).

Pliki nagłówkowe nie muszą być dodawane do projektu, są one włączane do plików źródłowych dzięki opcji preprocesora `#include`. Jednak ich dodanie do sekcji *Header Files* umożliwia ich łatwą edycję, poprzez podwójne kliknięcie nazwy takiego pliku na liście plików projektu.

# Programy narzędziowe

Wraz z pakietem WinAVR zainstalowane zostały liczne programy narzędziowe umożliwiające komplikację, konsolidację i obróbkę plików wynikowych projektu. AVR Studio, budując projekt, ogranicza się do utworzenia specjalnego skryptu Makefile, zawierającego instrukcje dla programów narzędziowych z pakietu WinAVR (m.in. dla programu make), określające, w jaki sposób zbudować dany projekt. Jeśli z jakiegoś powodu tworzony przez AVR Studio skrypt nam nie odpowiada, możemy w każdej chwili zastąpić go własnym. Efekt działania programów narzędziowych prezentowany jest w oknie *Build*. Wszystkie prezentowane w nim komunikaty pochodzą od wywoływanych programów zewnętrznych, a AVR Studio służy tylko do ich przejrzystej prezentacji, umożliwia także automatyczne przejście do miejsca pliku, którego dany komunikat dotyczy. Aby przejść do fragmentu pliku, którego dotyczy dany komunikat, wystarczy dwukrotnie kliknąć w oknie *Build* na komunikat błędu lub ostrzeżenie. W efekcie otwarty zostanie plik, którego dany komunikat dotyczy, a kurSOR przeniesiony zostanie w miejsce wystąpienia błędu lub ostrzeżenia.

Programy narzędziowe umożliwiają wykonanie znacznie większej liczby czynności niż te, do których wykorzystuje je AVR Studio. Poniżej zostaną omówione najważniejsze z tych programów, wraz z ich najczęstszymi sposobami wykorzystania.



## Wskazówka

Szczególnie na początku pisania programów przeznaczonych dla mikrokontrolerów AVR nie ma żadnej potrzeby, aby ingerować w proces wywoływanego programów narzędziowych. Jak widać z poprzedniego przykładu, cały proces tworzenia kodu wynikowego aplikacji jest zautomatyzowany, w efekcie nie musimy sobie zaprzątać głowy jego szczegółami.

Natomiast nawet podstawowa znajomość programów narzędziowych przydaje się w pewnych sytuacjach, umożliwiając bardziej efektywne i zautomatyzowane wykonywanie pewnych czynności.

## Linker

W wyniku działania kompilatora otrzymuje się jeden lub więcej plików obiektowych (z rozszerzeniem *.o*). Pliki te nie nadają się bezpośrednio do zaprogramowania mikrokontrolera. Zawierają one skompilowany kod, lecz wszystkie zawarte w tych plikach odwołania do pamięci występują w formie symboli, a nie bezwzględnych adresów. Poza tym, oprócz skompilowanego kodu pliki obiektowe mogą zawierać specjalne fragmenty metajęzyka, wymagające dalszej obróbki przez program linkera. Aby otrzymać kod wykonywalny, pliki obiektowe należy skonsolidować (zlinkować), w efekcie uzyskując finalny kod w postaci pliku *elf*. Zadanie to wykonuje program linkera (program *ld.exe*). Oprócz zamiany symboli na ich bezwzględne adresy i relokacji kodu linker ma także możliwość wprowadzania pewnych optymalizacji, np. wymiany rozkazów skoków długich na krótkie (*JMP* vs. *RJMP*) lub usuwania niewykorzystywanych fragmentów kodu. W tym zakresie jego działania są jednak mocno ograniczone — usuwany może być tylko cały plik obiektowy, ale nie jego fragmenty. Stąd też przy pisaniu bibliotek

poszczególne funkcje powinny być umieszczone w oddzielnych plikach źródłowych, w efekcie wyładują one w oddzielnych plikach obiektowych. Istnieją pewne wyjątki od tej zasady, opisane w rozdziale poświęconym tworzeniu bibliotek. Jak widać, nie jest możliwe stworzenie kodu wynikowego bez udziału linkera, stąd też wywołanie tego programu jest zwykle ostatnim etapem komplikacji programu — wyjątkiem jest tworzenie biblioteki, gdzie zamiast linkera wywołuje się program `ar`, który z plików obiektowych tworzy bibliotekę. Linker jest programem bardzo uniwersalnym, stąd też posiada wiele opcji konfiguracyjnych. Na szczęście większość z nich wykorzystywana jest tylko w specjalnych sytuacjach, które zostaną omówione w dalszej części książki. Na razie pokazane zostaną tylko podstawowe opcje związane z generowaniem plików wynikowych na platformie AVR.

Linker możemy wywoływać bezpośrednio z linii poleceń, podając mu jako parametry listę plików obiektowych, które należy zlinkować, oraz nazwę pliku wynikowego:

```
ld -o output /lib/crt0.o helloworld.o -lc
```

Powyższa prosta forma wywołania powoduje zlinkowanie pliku `helloworld.o` z plikiem `crt0.o` oraz biblioteką `libc.a`, a wynik, czyli program wykonywalny, zostanie umieszczony w pliku o nazwie `output`. Linker może być także wywoływany pośrednio przez kompilator `gcc`:

```
gcc -Wl,--startgroup foo.o bar.o -Wl,--endgroup
```

W tym przypadku `gcc` wywołuje linker z parametrem `-startgroup` i dwoma plikami obiektowymi do zlinkowania, `foo.o` i `bar.o`.



Aby dany parametr wywołania został przez `gcc` przekazany linkerowi, należy poprzedzić go prefiksem `-Wl`.

Jest to niezwykle ważne, gdyż bez tego prefiksu dany parametr zostanie po prostu zignorowany, co w efekcie może doprowadzić do niewłaściwego linkowania.

Trzecią możliwością, najczęściej stosowaną, jest wywołanie linkera ze specjalnego skryptu programu `make`. Program `make` szerzej zostanie omówiony w dalszej części rozdziału.

W celu konsolidacji programu program linkera zawsze używa tzw. **skryptu linkera**. Jest to plik tekstowy napisany w specjalnym języku skryptowym. Określa się w nim, w jaki sposób różne sekcje znajdujące się w plikach obiektowych mają zostać rozmiędzcone w pamięci. Oprócz tego skrypty mogą zawierać różne opcje komplikacji, można w nich także definiować symbole, które potem będzie można wykorzystać w programie. Symbole te mogą określać np. adresy w pamięci poszczególnych sekcji. Środowisko kompilatora `gcc` dostarczone dla platformy AVR zawiera domyślne skrypty linkera w katalogu `avr\lib\ldscripts`. Zwykle nie zachodzi potrzeba ich modyfikacji, chyba że chcemy zmienić domyślny układ segmentów pamięci. Jeśli wywołując linker, nie podamy, jakiego ma użyć skryptu, to użycie on skryptu domyślnego. Użyty skrypt zostanie wybrany na podstawie innych opcji wywołania linkera, określających m.in. architekturę procesora, dla której przeprowadzany jest proces linkowania. Przeglądając katalog

z domyślnymi skryptami linkera, można zauważyc, że skrypty te mają w nazwie *avr* oraz cyfry określające numer architektury. Na szczęście, domyślnie linker potrafi sobie sam wybrać właściwy skrypt, w efekcie w większości przypadków o czymś takim jak skrypty linkera możemy zapomnieć. Jednak domyślnie wykorzystywany skrypt można zmienić, przekazując linkerowi opcję *-T* i nazwę nowego skryptu. Konieczność taka zachodzi czasami w sytuacji, kiedy do procesora dołączona jest zewnętrzna pamięć SRAM/ROM. Przykłady takie zostaną pokazane w rozdziale 9, poświęconym interfejsowi pamięci zewnętrznej.

Jak wspomniano, najczęściej linker wywoływany jest pośrednio poprzez *gcc* w skrypcie programu *make*. Przyjęło się definiować w tym skrypcie kilka zmiennych, przechowujących elementy wywołania linkera. Pierwszą zmienną jest *LDFLAGS*, która zawiera opcje wywołania linkera. Ponieważ linker jest wywoływany pośrednio, poprzez *gcc*, należy pamiętać, aby każdą opcję poprzedzić prefiksem *-Wl*, w przeciwnym przypadku zostanie ona zignorowana, np.:

```
LDFLAGS = -Wl,-Map=ADCNoiseReduction.map
```

powoduje przekazanie linkerowi opcji nakazującej mu tworzenie pliku *map*, zawierającego mapę pamięci programu wynikowego.

Kolejną ważną zmienną związaną z linkerem jest *LIBDIRS*. Wskazuje ona katalogi, w których znajdują się biblioteki, np.:

```
LIBDIRS = -L"C:\test" -L"C:\libdir"
```

mówi linkerowi, że ma szukać bibliotek w katalogu *C:\test* i *C:\libdir*. Katalogi przeszukiwane są w kolejności ich podania, co w pewnych sytuacjach ma znaczenie.



Należy pamiętać, aby ścieżki dostępu podawać w znakach cudzysłowu, dzięki temu spacje znajdujące się w nazwie będą mogły być poprawnie zinterpretowane.

Kolejną ważną zmienną jest zmienna *LIBS*, określająca nazwy bibliotek, z którymi ma zostać zlinkowany tworzony kod:

```
LIBS = -lm -lprintf_flt
```

Powyższe powoduje zlinkowanie aplikacji z bibliotekami *libm.a* i *libprintf\_flt.a* — więcej o zasadach związanych z nazewnictwem bibliotek można dowiedzieć się z podrozdziału „Tworzenie bibliotek”. Warto tylko wspomnieć, że linker sam dodaje do nazwy biblioteki prefiks *lib* i sufiks *.a*, w efekcie nie należy ich podawać przy określaniu nazwy biblioteki. Gdyby powyższa linia wyglądała następująco:

```
LIBS = -llibm.a -llibprintf_flt.a
```

to linker szukałby bibliotek o nazwach *llibm.a.a* i *llibprintf\_flt.a.a*, co oczywiście zakończyłoby się błędem, gdyż takie biblioteki nie istnieją.

Kolejną zmienną jest zmienna *OBJECTS*, która zawiera nazwy wszystkich plików obiektowych powstałych w wyniku komplikacji programu, które należy zlinkować, abytrzymać plik wynikowy:

```
OBJECTS = ADCNoiseReduction.o test.o
```

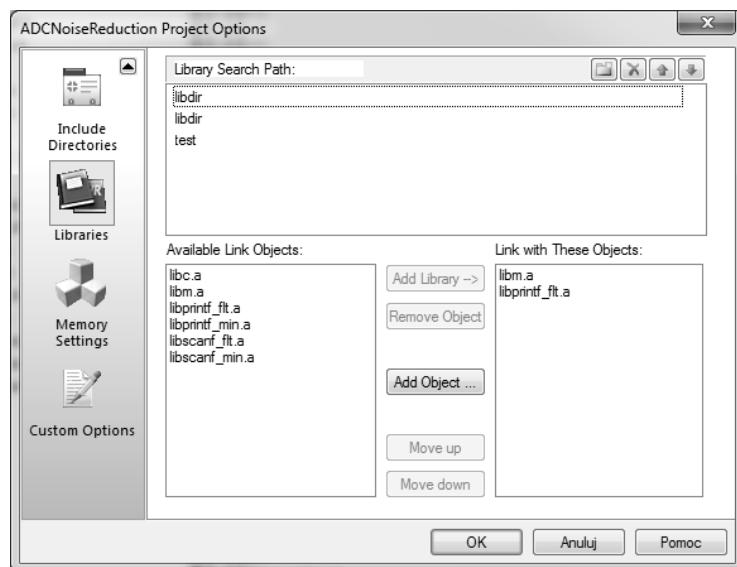
W powyższym przykładzie do linkowania wykorzystane zostaną pliki *ADCNoiseReduction.o* i *test.o*. Brak któregokolwiek z powyższych plików spowoduje błąd na etapie linkowania programu.

Czasami wykorzystujemy pliki obiektowe, które nie pochodzą z komplikacji programu, np. pliki obiektowe zawierające dane aplikacji. Tego typu pliki obiektowe dodajemy jako parametry zmiennej `LINKONLYOBJECTS`:

```
LINKONLYOBJECTS = bitmapa.o
```

Te same parametry, zamiast modyfikować bezpośrednio w skrypcie programu make, możemy określić przy pomocy graficznych narzędzi konfiguracyjnych programu AVR Studio. Opcje związane z linkowaniem programu dostępne są po wybraniu *Project/Configuration Options*. Znajdują się one w zakładkach *Libraries*, *Memory Settings* oraz *Custom Options*. W zakładce *Memory Settings* wspomniane będzie szerzej w rozdziale poświęconym interfejsowi pamięci zewnętrznej. W zakładce *Libraries* możemy określić ścieżki do katalogów zawierających biblioteki oraz nazwy bibliotek, które mają zostać dołączone do programu (rysunek 1.10).

**Rysunek 1.10.**  
Dodawanie  
do projektu bibliotek



Z kolei w zakładce *Custom Options* po wybraniu w okienku *Custom Compilation Options* opcji *[Linker Options]* można dodawać opcje linkera (rysunek 1.11).

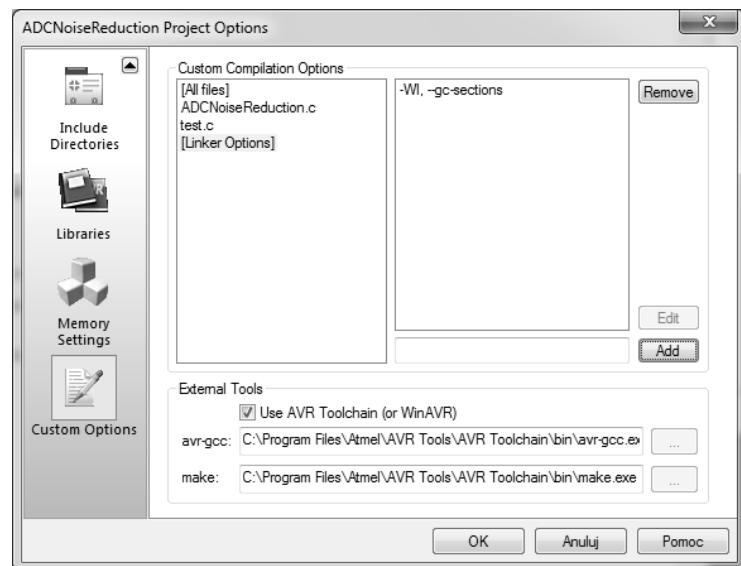
Pamiętać jednak należy, aby opcje linkera poprzedzać opcję `-Wl`, w przeciwnym przypadku nie zostaną przesłane do programu linkera. Niestety, AVR Studio samo nie dba o dodanie tego przełącznika.



**Wskazówka** W większości projektów nie zachodzi konieczność modyfikacji domyślnych opcji wywołania linkera ani tym bardziej modyfikacji skryptów linkera.

**Rysunek 1.11.**

*Ustalanie opcji linkera*



Najbezpieczniej jest więc opcji związanych z linkowaniem programu nie ruszać. Sytuacje, w których zachodzi konieczność ich modyfikacji, są nieliczne. Odpowiednie przykłady pokazane zostaną w dalszych rozdziałach książki.

## Program avr-size

Program ten jest domyślnie wywoływany przez AVR Studio w celu podania objętości wygenerowanego kodu i danych programu. Potrafi on podawać dane o długości zarówno dla plików w formacie *Intel HEX*, jak i dla plików *elf*. Ponieważ pliki *HEX* nie zawierają podziału na sekcje, w efekcie program *avr-size* dla takiego pliku zwraca tylko informację o całkowitej długości danych programu (które w tym przypadku oznaczają długość kodu):

```
avr-size ADCNoiseReduction.hex
text    data    bss    dec      hex filename
      0     252     0     252      fc ADCNoiseReduction.hex
```

W przypadku plików *elf* *avr-size* podaje informacje zarówno o długości kodu, jak i danych programu:

```
avr-size ADCNoiseReduction.elf
text    data    bss    dec      hex filename
      252     0     7     259      103 ADCNoiseReduction.elf
```

W przypadku danych należy pamiętać, że podana długość dotyczy wyłącznie danych alokowanych w trakcie komplikacji programu, a więc zmiennych globalnych i statycznych. Nie jest możliwe, w prosty sposób, określenie ilości pamięci zajętej przez dane alokowane dynamicznie oraz lokalnie na stosie. Stąd też niewielka ilość pamięci SRAM zajęta przez dane w raporcie programu *avr-size* nie oznacza, że dostępna w procesorze ilość pamięci jest wystarczająca.



Wskazówka

W obu przypadkach (to jest zwracania długości kodu programu i danych) w sytuacji, kiedy zwrócone wartości są większe niż ilości dostępnej pamięci w procesorze, możemy być pewni, że program nie będzie działał poprawnie.

Stąd też zwracanie uwagi na wyniki raportu generowanego przez `avr-size` jest bardzo ważne i powinno być automatycznym odruchem uruchamianym po każdej komplikacji.

## Program `avr-nm`

Prosty program `avr-nm` umożliwia uzyskanie wielu informacji o skompilowanym programie, których wartość trudno przecenić. Program ten wyświetla listę symboli użytych w programie. Aby uzyskać bardziej szczegółowe informacje, musimy posiadać plik *elf*. Pliki *hex* nie posiadają informacji o symbolach ani innych metadanych, nie nadają się więc do analizy. Program ten w nieco bardziej „strawnym” sposobie pokazuje informacje, które możemy znaleźć w wygenerowanym po komplikacji pliku *map*. Posiada on liczne opcje konfiguracyjne, z których poniżej pokazane zostaną tylko najważniejsze i najbardziej przydatne:

- ◆ Opcja `--size-sort` — powoduje posortowanie elementów względem ich długości od najmniejszego do największego.
- ◆ Opcja `--reverse-sort` — powoduje odwrócenie kolejności posortowanych obiektów.
- ◆ Opcja `--numeric-sort` — powoduje posortowanie elementów według adresów ich występowania. Umożliwia łatwe zorientowanie się, gdzie dany obiekt jest umieszczony w pamięci.
- ◆ Opcja `--demangle` — dla programów napisanych w C jest bez znaczenia. W przypadku programów napisanych w C++ zamienia wygenerowane przez kompilator symbole na postać czytelną dla człowieka.
- ◆ Opcja `-S` — powoduje, oprócz wyświetlenia adresu symbolu, wyświetlenie także informacji o ilości zajmowanej przez niego pamięci.
- ◆ Opcja `-l` — oprócz nazwy symbolu wyświetcone zostanie także miejsce, w którym jest on zdefiniowany (nazwa pliku, linia).

Powyższe opcje powodują wyświetlenia symboli i podstawowych informacji o nich znajdujących się w pliku *elf*. Przed każdym symbolem znajduje się adres, pod jakim jest on zdefiniowany, oraz typ symbolu. Typ symbolu jest zazwyczaj literą. Mała litera oznacza, że dany symbol jest symbolem lokalnym, wielka litera oznacza symbol globalny. Znaczenie poszczególnych symboli jest następujące:

- ◆ Symbol A — wartość symbolu jest bezwzględna i nie ulegnie zmianie na skutek linkowania czy relokacji.
- ◆ Symbol B — symbol znajduje się w sekcji niezainicjowanej (*BSS*).
- ◆ Symbol D — symbol znajduje się w zainicjowanej sekcji pamięci.
- ◆ Symbol N — dany symbol używany jest tylko do debugowania.

- ♦ Symbol R — symbol znajduje się w sekcji pamięci tylko do odczytu.
- ♦ Symbol W — symbol jest tzw. słabym symbolem, zdefiniowanym przy pomocy atrybutu weak.
- ♦ Symbol T — symbol znajduje się w sekcji *text* zawierającej kod programu.

Jako przykład przeanalizowany zostanie wynik następującego polecenia:

```
avr-nm -C --size-sort ADCNoiseReduction.elf -S
00800100 00000001 b no.1271
000000b8 00000002 T test
00800101 00000002 b tmp.1270
00800105 00000002 B wyn
00800103 00000002 B wynik
00000ee 00000008 T main
00000ba 0000000e T initADC
00000c8 00000026 T GetADC
0000056 00000062 T __vector_21
```

Jak widać, w pliku *ADCNoiseReduction.elf* zdefiniowano zmienne lokalne no i tmp oraz zmienne globalne wyn i wynik. Oprócz tego zdefiniowano funkcje test, main, initADC, GetADC oraz \_\_vector\_21, będącą funkcją obsługi przerwania ADC. Dla każdego symbolu podany został jego adres w pamięci oraz długość. Adres zmiennych może wydawać się nieco dziwny — adresy te zaczynają się od 0x00800000. Dzieje się tak, ponieważ w AVR istnieje kilka przestrzeni adresowych, czego nie wspiera kompilator gcc; aby ominąć tę wadę kompilatora, dla adresów w pamięci SRAM mikrokontrolera dodaje się adres bazowy 0x00800000, dzięki czemu programy narzędziowe, w tym linker, wiedzą, o jaką sekcję pamięci chodzi.

Program avr-nm umożliwia łatwe sprawdzenie, ile pamięci zajmują pisane funkcje, co np. ułatwia wyszukiwanie kandydatów do potencjalnej optymalizacji.

## Program avr-objcopy

Program avr-objcopy wykorzystywany jest do kopowania zawartości plików obiektowych do nowych plików, o formacie określonym przez opcje wywołania programu. Jest on wykorzystywany praktycznie w każdym projekcie, w celu zamiany wynikowego pliku *elf*, powstałego jako wynik działania linkera, na plik w formacie *Intel HEX*, będący gotowym plikiem wykorzystywanym przez programatory do zaprogramowania mikrokontrolera.



Wiele programatorów nie potrafi się posługiwać plikami w formacie *elf*, stąd wymagana jest ich konwersja do formatu *Intel HEX*.

Zwykle program avr-objcopy wywoływany jest przez standardowy skrypt programu make i nie ma potrzeby ani zmieniać opcji jego wywołania, ani w ogóle wiedzieć, że taki program istnieje. Z punktu widzenia programisty ma on jednak istotną cechę, która czyni go przydatnym w trakcie tworzenia aplikacji. Potrafi on konwertować pliki z i do różnych formatów. Ma to zastosowanie w sytuacji, kiedy dysponujemy plikiem binarnym,

z którego dane chcemy wykorzystać w aplikacji — np. pliki graficzne, konfiguracyjne lub z danymi. Pliki takie można przekonwertować np. na tablice z danymi, które potem są dołączane jako zwykłe pliki źródłowe C, lecz jest to sposób mało efektywny — powstają niepotrzebnie ogromne pliki, w których trudno jest coś zmienić, wydłużają one także czas komplikacji programu. Stąd też lepszym i prostszym rozwiążaniem jest wykorzystanie programu avr-objcopy. Jako argumenty wywołania podaje się formaty pliku wejściowego, wyjściowego oraz nazwę pliku przed konwersją i po konwersji. Plik wejściowy ma zazwyczaj postać binarną, o czym informuje parametr -I binary. Plik wyjściowy ma być plikiem obiektowym, co określa parametr -O elf32-avr. Polecenie konwersji pliku binarnego do postaci nadającej się do zlinkowania z resztą programu wygląda więc następująco:

```
avr-objcopy -I binary -O elf32-avr test.bin test.o
```

Powyższe polecenie powoduje przekształcenie pliku *test.bin* do formatu obiektowego i utworzenie pliku *test.o*. W pliku tym zostaną zdefiniowane trzy nowe symbole: `_binary__test_bin_start`, `_binary__test_bin_end` i `_binary__test_bin_size`, zawierające odpowiednio adres początkowy, końcowy i długość dołączonych danych w pamięci. Do tak utworzonych symboli można odwoływać się w programie, definiując jako extern odpowiednie zmienne:

```
extern void * _binary__test_bin_start;
extern void * _binary__test_bin_end;
extern void * _binary__test_bin_size;
```



Nazwy zdefiniowanych symboli tworzone są poprzez dodanie prefiku `_binary__` do nazwy pliku oraz jednego z trzech sufiksów `_start`, `_end` lub `_size`.

Domyślnie dane zostaną umieszczone w sekcji `.data`, zajmą więc cenną pamięć SRAM. Aby dane nie trafiały do pamięci SRAM, należy umieścić je w sekcji o nazwie `.progmem`.  
→`data`. Można tego dokonać przy pomocy opcji `-rename-section`:

```
avr-objcopy --rename-section .data=.progmem.data,contents,alloc,load,readonly,data
-I binary -O elf32-avr test.bin test.o
```

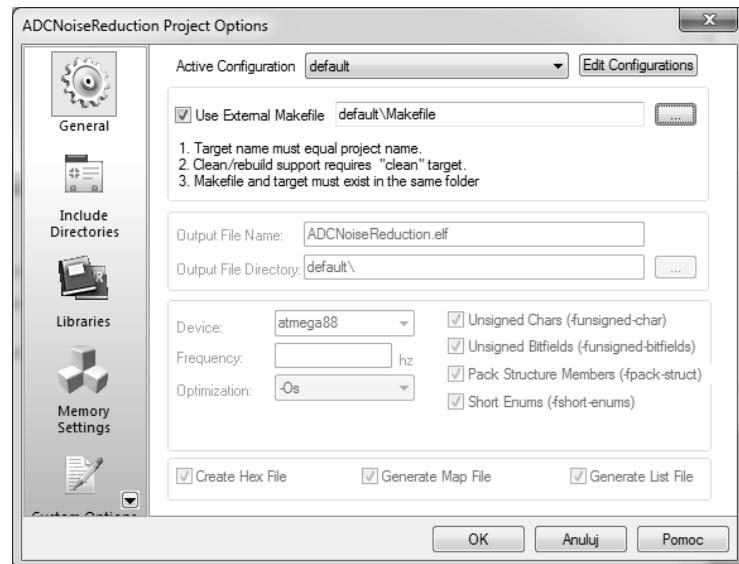
Opcja ta przyjmuje jako parametry nazwę starej sekcji (w tym przypadku `.data`) oraz nazwę nowej sekcji (`.progmem.data`) i jej atrybuty. Uzyskane w wyniku powyższych poleceń pliki obiektowe można zlinkować z programem. Proces ten można zautomatyzować, pisząc własny skrypt programu make. Niestety, AVR Studio nie wspiera dodawania własnych sekcji do skryptu programu make, nie umożliwia także dodawania do projektu gotowych plików obiektowych. Problem ten można ominąć, pisząc własny skrypt *Makefile*, a następnie zaznaczając w opcjach projektu opcję *Use External Makefile* (rysunek 1.12).

Dzięki zaznaczeniu tej opcji AVR Studio zamiast generować własny skrypt, będzie korzystać z podanego pliku *Makefile*.

Do automatyzacji zostanie wykorzystany plik *Makefile* wygenerowany przez AVR Studio. Po jego wygenerowaniu zaznaczamy pokazaną wcześniej opcję *Use External*

**Rysunek 1.12.**

Dzięki opcji *Use External Makefile* można określić zewnętrzny plik *Makefile*, który będzie używany do budowania projektu



*Makefile* i jako skrypt wskazujemy utworzony wcześniej przez AVR Studio plik *Makefile* projektu. W pliku tym odnajdujemy sekcję *OBJECTS* i dodajemy do niej pliki obiektowe, które powstaną w wyniku przekształcenia plików binarnych zawierających dane.



Wskazówka

Ponieważ przy konwersji automatycznie powstaną odpowiednie pliki nagłówkowe, które można zainkludować w pisany programie, muszą one powstać przed komplikacją wykorzystującą je programu. Stąd też najlepiej, jeśli powstałe z konwersji plików binarnych pliki obiektowe dopiszemy na początek sekcji *OBJECTS*, dzięki temu odpowiednie pliki nagłówkowe zostaną wygenerowane przed komplikacją właściwego kodu programu.

Odpowiednia linia skryptu wygląda następująco:

```
## Objects that must be built in order to link
OBJECTS = wykres.o ADCNoiseReduction.o test.o
```

Plik *wykres.o* powstanie w wyniku przekształcenia pliku *wykres.png*, zawierającego dane graficzne. Pozostałe pliki powstaną w wyniku komplikacji źródeł programu.

Kolejnym etapem jest dodanie reguły mówiącej, w jaki sposób otrzymać plik *wykres.o*:

```
wykres.o: ../wykres.png
avr-objcopy -I binary -O elf32-avr \
--rename-section .data=.progmem.data.contents.alloc.load,readonly,data \
$(<) $(@)

@echo "#include <avr\pgmspace.h>" >> ../$(*).h
@echo "extern const char" _binary____$(*)_png_start"[] PROGMEM;" >> ../$(*).h
@echo "extern const char" _binary____$(*)_png_end"[] PROGMEM;" >> ../$(*).h
@echo "extern const int" _binary____$(*)_png_size"[];" >> ../$(*).h
```

Reguła ta składa się z dwóch części. W pierwszej generujemy plik obiektowy, zawierający linkowaną wersję pliku *wykres.png*. Jak pamiętamy, utworzenie pliku obiektowego z danych binarnych wiąże się z utworzeniem symboli `_binary_wykres_png_start`, `_binary_wykres_png_end` i `_binary_wykres_png_size`, zawierających odpowiednio adres początku, końca i długość danych w pamięci FLASH mikrokontrolera. Aby te symbole móc wykorzystać w programie, musimy je zadeklarować jako symbole zewnętrzne (`extern`). W tym celu, aby ułatwić sobie życie, w drugiej części sekcji przy pomocy poleceń `echo` tworzony jest odpowiedni plik nagłówkowy zawierający stosowne deklaracje. Dla naszego przykładu utworzony plik będzie wyglądał następująco (*wykres.h*):

```
#include <avr\pgmspace.h>
extern const char _binary_wykres_png_start[] PROGMEM;
extern const char _binary_wykres_png_end[] PROGMEM;
extern const int _binary_wykres_png_size[];
```

Po jego włączeniu do programu dyrektywą `#include` można korzystać z powyższych symboli, uzyskując w ten sposób dostęp do włączonych do programu danych binarnych.

## Program make

Proces otrzymywania kodu wynikowego w postaci plików zawierających dane dla programatora jest wieloetapowy. Składa się on z kilku etapów:

- ◆ Kompilacji kodu źródłowego do plików obiektowych.
- ◆ Ewentualnego przekształcenia innych plików w pliki obiektowe.
- ◆ Linkowania powstałych plików, w wyniku czego uzyskuje się plik *elf*, który może służyć jako plik źródłowy dla programatora.
- ◆ Opcjonalnej generacji plików *Intel HEX* zrozumiałych dla większości prostych programów sterujących programatorem.

Wszystkie te etapy można wykonywać ręcznie, co jest raczej mało wygodne; można także cały proces zautomatyzować. Dla automatyzacji tworzenia kodu wynikowego stworzono program `make`. Program ten posługuje się specjalnym językiem skryptowym, umożliwiającym opisanie kolejnych etapów tworzenia aplikacji. Program `make` odczytuje dane z pliku zawierającego polecenia, jakie ma wykonać, a następnie zgodnie z zawartymi w nim instrukcjami buduje aplikację. Domyslnie, po wywołaniu programu `make` bez jakichkolwiek parametrów, szuka on pliku o nazwie *makefile*, a po jego znalezieniu zaczyna wykonywać zawarte w nim polecenia.



### Wskaźówka

Pisanie skryptów programu `make` na pierwszy rzut oka wydaje się być zadaniem nietrywialnym. Na szczęście, w znakomitej większości przypadków nie jest to konieczne. Zarówno cały *toolchain* dostarczany wraz z kompilatorem, jak i AVR Studio posiadają szablony plików *makefile* oraz wygodne w użyciu konfiguratory. Właściwie z nich korzystając, naprawdę nigdy nie będziesz musiał pisać własnego pliku *Makefile*.

Osoby lubiące wiedzieć „co siedzi w środku”, poniżej znajdą skrótnowy opis struktury skryptów programu make. Warto się z nim zapoznać, aby chociaż częściowo zrozumieć, jak program ten działa.



Pliki skryptowe programu make należy pisać w prostych edytorach tekstu, typu notepad, programmers notepad lub AVR Studio.

## Komentarze

Ważnym elementem skryptu są komentarze. Umożliwiają one wprowadzenie dowolnych opisów do pliku. Opisy te mają za zadanie ułatwić zrozumienie funkcji danych elementów pliku i są kompletnie ignorowane przez program make. Aby wprowadzić komentarz, należy użyć znaku #. Może on wystąpić w dowolnej części linii skrypu:

```
# to jest komentarz  
CC=avr-gcc #wszystko po znaku #jest komentarzem
```



Pamiętaj! Stosowanie komentarzy nic nie kosztuje, a ułatwia analizę skryptu.

## Kontynuacja linii

Przy pisaniu danego polecenia może się zdarzyć, że utworzona linia jest bardzo dłuża, co utrudnia czytanie. Możemy ją podzielić na kilka krótszych linii, mieszczących się na ekranie. Nie możemy jednak po prostu w celu podzielenia linii nacisnąć klawisza *Enter*; wiele poleceń spodziewa się parametrów znajdujących się w jednej linii i po takim podziale nie działałoby prawidłowo. Stąd też aby podzielić linię, wprowadzono znak \. Jego umieszczenie na końcu linii informuje program make, że następna linia jest kontynuacją poprzedniej, np.

```
To jest pierwsza linia \  
A to jej kontynuacja.
```

W takie podzielone linie można także wpleść komentarze:

```
CC= #tu zaczyna się komentarz \  
avr-gcc #i tu też
```

Program make zignoruje powyższe komentarze, scal i dwie linie i w efekcie uzyska:

```
CC=avr-gcc
```

## Cele

Zanim bardziej szczegółowo zostanie omówiona struktura skryptów make, zastanówmy się, po co je tworzymy. Oprócz tego, że automatyzują one pewne czynności, wywołując skrypt, mamy jakiś cel. Skrypt ma wykonać określoną przez nas czynność, czyli ma nam pomóc osiągnąć wybrany cel (ang. *Target*). Przyjęło się określać kilka podstawowych celów, które obsługuje praktycznie każdy skrypt:

- ◆ **clean** — cel ten ma za zadanie „oczyścić” katalog, w którym znajduje się program, ze wszystkich zbędnych plików, np. powstały w wyniku komplikacji. Domyślnie po wywołaniu tego celu wszystkie utworzone przez inne cele skryptu pliki powinny zostać usunięte (czyli np. wygenerowane pliki obiektowe, pliki wynikowe, tymczasowe).
- ◆ **all** — cel ten domyślnie powoduje wykonanie wszystkich kroków niezbędnych do otrzymania kodu wynikowego, a więc gotowego programu. Domyślnie po wywołaniu programu `make` realizowany jest właśnie cel `all`.
- ◆ **install** — cel ten ma za zadanie zainstalować skompilowany wcześniej program w systemie. W przypadku programowania mikrokontrolerów cel ten zwykle nie jest implementowany (bo nie ma czego instalować).

W zależności od potrzeb można implementować inne cele, np. program mający za zadanie zaprogramowanie procesora wygenerowanym wsadem. Części skryptu realizujące poszczególne cele wywołuje się poprzez przekazanie nazwy celu jako parametru wywołania programu `make`:

```
make all
```

powoduje wywołanie celu `all` z pliku *Makefile* znajdującego się w bieżącym katalogu. W skrypcie cele określa się, wpisując nazwę celu zakończoną dwukropkiem (:), np.:

```
all:
```

Po tak zdefiniowanym celu występuje zestaw reguł opisujących kolejne etapy prowadzące do jego osiągnięcia.

## Reguły

Cele definiowały uzyskany efekt, natomiast reguły określają sposób, w jaki dany cel uzyskać. Zwykle reguła prowadzi w efekcie do uzyskania pliku będącego efektem działania reguły, np.:

```
main.o : main.c  
gcc main.c
```

Powyższa reguła określa, w jaki sposób otrzymać plik `main.o` — należy wywołać kompilator `gcc`, przy pomocy którego skompilowany zostanie plik `main.c`, w efekcie uzyskamy plik `main.o`. Takie reguły jak powyższa nazywane są regułami bezpośrednimi — odnoszą się one do konkretnego pliku. W związku z tym są średnio przydatne — aby skompilować kolejny plik, należałoby taką regułę powielić dla każdego kompilowanego pliku. Reguły możemy uogólnić:

```
% .o: %.c  
gcc -c -o $@
```

Powyższa reguła określa, w jaki sposób otrzymać plik obiektowy — w tym celu każdy plik z rozszerzeniem `.c` (a więc plik źródłowy języka C) należy skompilować przy pomocy kompilatora `gcc`. Opcja `-c` jest opcją kompilatora mówiąca, że ma stworzyć plik obiektowy (`.o`). Bez niej pliki obiektowe tworzone byłyby tylko tymczasowo, w trakcie wywołania `gcc`. Opcja `-o $@` mówi, że utworzony plik ma mieć nazwę taką jak plik

źródłowy — np. plik *main.c* zostanie skompilowany do pliku *main.o* itd. Znak % odpowiada dowolnemu łańcuchowi tekstowemu zbudowanemu z zera lub większej liczby znaków. Zwykle pliki źródłowe mają takie same nazwy jak pliki wynikowe, różnią się tylko rozszerzeniem. W takiej sytuacji regułę uogólniającą możemy uproszczyć do postaci:

.c.o:

Jest ona równoważna zapisowi %.o: %.c.

Podobne reguły określają sposób tworzenia wszystkich plików.

## Makra

Makra umożliwiają przyporządkowanie zmiennym określonych wartości. Użycie w dalszej części skryptu programu make danego makra jest równoważne wstawieniu w danym miejscu zawartości makra. Zwiększa to czytelność skryptu, a także czyni go bardziej uniwersalnym — wystarczy poprawić definicję makra, aby zmiany automatycznie zostały wprowadzone w całym skrypcie. Wartość reprezentowaną przez makro umieszcza się poprzez umieszczenie znaku \$ i w nawiasach nazwy makra, np. \$(CC) wstawia w miejscu wystąpienia zawartość makra CC. Makra definiuje się zazwyczaj na początku skryptu:

```
CC=avr-gcc  
CFLAGS = $(COMMON)  
CFLAGS += -Wall -gdwarf-2 -Os -std=gnu99 -funsigned-char -funsigned-bitfields -  
    ↪fpack-struct -fshort-enums
```

Pierwsze makro przyporządkowuje zmiennej CC wartość avr-gcc, w efekcie w dalszej części skryptu zamiast pisać avr-gcc, można użyć \$(CC). Definicja makra CFLAGS jest nieco bardziej skomplikowana. Początkowo zmiennej CFLAGS przyporządkowana zostaje wartość innego makra — COMMON. W kolejnej linii do CFLAGS dodawane są kolejne opcje.



Gdyby zamiast znaku += użyć =, poprzednia wartość CFLAGS zostałaby zastąpiona nową wartością.

W skryptach zwykle definiuje się kilka standardowych makr:

- ◆ CC — zawiera nazwę i ścieżkę do kompilatora języka C.
- ◆ CPP — zawiera nazwę i ścieżkę do kompilatora C++.
- ◆ CFLAGS — zawiera opcję kompilacji kodu źródłowego w C/C++.
- ◆ LDFLAGS — zawiera parametry wywołania linkera.
- ◆ LIBS — zawiera nazwy dołączonych bibliotek.
- ◆ LIBDIRS — zawiera ścieżki do bibliotek.
- ◆ INCLUDE — zawiera ścieżki do plików nagłówkowych.

## Modyfikatory makr

Wartości przechowywane w makrach mogą być modyfikowane, co umożliwia ich przekształcenie, np.:

```
SRCS=$(OBJS .o=.c)
```

Powyższa reguła pokazuje, jak z listy nazw plików obiektów wygenerować listę nazw plików źródłowych. Powyższy modyfikator powoduje wymianę wszystkich rozszerzeń *.o* na *.c*. Jeśli *OBJS* będzie zdefiniowane następująco:

```
OBJS=test1.o test2.o test3.o
```

to w wyniku działania powyższej reguły otrzymamy makro *SRCS* zawierające wartości *test1.c test2.c test3.c*. Poszczególne modyfikatory można łączyć w jednej linii, rozdzielając je przecinkiem.

Szczególne znaczenie mają modyfikatory umożliwiające ekstrakcję z makra części nazw plików. Założymy, że zdefiniowaliśmy makro o nazwie *PLIKI*:

```
PLIKI=c:\instal\main.c ISR.S
```

Poszczególne części tak zdefiniowanego makra możemy wyekstrahować przy pomocy modyfikatorów pokazanych w tabeli 1.1.

**Tabela 1.1.** Modyfikatory zwracające części nazw plików

Modyfikator	Opis	Przykład	Wynik
D	Zwraca ścieżkę dostępu do pliku	<code>\$(PLIKI.D)</code>	<code>C:\instal</code>
E	Zwraca rozszerzenie nazwy pliku	<code>\$(PLIKI.E)</code>	<code>.c.S</code>
F	Zwraca nazwę pliku	<code>\$(PLIKI.F)</code>	<code>main.c ISR.S</code>

## Dyrektywy warunkowe

Niezwykle przydatną możliwością jest warunkowe wykonanie fragmentu skryptu. Umożliwia to tworzenie skryptów, które np. w zależności od wybranego procesora dołączają inne biblioteki. Oczywiście, potencjalnych zastosowań jest wiele. Do dyrektyw warunkowych zaliczamy dyrektywy `%if`, `%elif`, `%else` i `%endif`. Każdy blok otwarty dyrektywą `%if` musi zostać zamknięty dyrektywą `%endif`, pomiędzy nimi może zostać użyta dowolna liczba dyrektyw `%elif` i `%else`. Zastosowanie tych dyrektyw zostanie pokazane na przykładzie warunkowego dołączania do programu biblioteki:

```
MCU=ATMega88

%if $(MCU) == ATMega88
LIBS+=testATMega88
%elif $(MCU) == ATMega128
LIBS+=testATMega128
%else
%abort Nieznany procesor $(MCU)
%endif
```

Powyższy skrypt testuje typ użytego procesora, który znajduje się w makrzu MCU. W zależności od tego, czy jest to ATMega88 czy ATMega128, dołącza odpowiednio bibliotekę *libtestATMega88.a* lub *libtestATMega128.a*. Jeśli MCU nie zawiera któregoś z wyżej wymienionych procesorów, to skrypt przerywany jest z komunikatem błędu (%abort).

## Zależności

Niezwyczajnym elementem skryptu są zależności. Określają one, jakie pliki muszą zostać zbudowane, aby otrzymać plik wynikowy. Linie te zawierają nazwę pliku wynikowego, dwukropek i listę plików, które muszą zostać zbudowane, aby otrzymać plik wynikowy.



Listy zależności nic nie mówią o sposobie, w jaki dane pliki mają zostać zbudowane. Do tego celu służą reguły.

W poniższym przykładzie:

```
projekt.elf : main.o test.o
```

aby zbudować plik *projekt.elf*, musimy dysponować dwoma plikami: *main.o* oraz *test.o*. Brak któregoś z tych plików uniemożliwia zbudowanie pliku *projekt.elf*, w efekcie skrypt zostanie zakończony komunikatem błędu, mówiącym o niespełnieniu zależności. Pliki występujące na liście zależności zostaną zbudowane w kolejności, w jakiej zostały wymienione, czyli najpierw plik *main.o*, a następnie plik *test.o*. W celu ich zbudowania program *make* wyszuka w skrypcie reguły określające sposób ich budowy, a następnie je wykona. Jeśli *make* nie znajdzie reguły określającej sposób zbudowania danego pliku, wyświetli komunikat błędu:

```
make: *** Brak reguł do zrobienia obiektu `test.o'. wymaganego przez `projekt.elf'.
→Stop.
```

## Przykłady

Po tej sporej ilości informacji przeanalizujmy prosty przykład.

Zacznijmy od podstawowych definicji, które będą zawierały informacje przydatne podczas komilacji:

```
MCU = atmega88
TARGET = przyklad.elf
CC = avr-gcc
CPP = avr-g++
COMMON = -mmcu=$(MCU)
CFLAGS = $(COMMON)
CFLAGS += -Wall -gdwarf-2 -Os -std=gnu99 -funsigned-char -funsigned-bitfields -
→fpck-struct -fshort-enums
LDFLAGS = $(COMMON)
```

Powyższy fragment definiuje pewne podstawowe makra, określające typ procesora, na który komplujemy program (MCU), użyte kompilatory (CC i CPP), opcje komplikacji (FLAGS) i linkowania (LDFLAGS).

Zdefiniujmy jeszcze listę plików wymaganych do zbudowania aplikacji:

```
OBJECTS = wykres.o main.o test.o
```

Lista ta jest zwykłym makrem, które zostanie wykorzystane do zbudowania listy zależności:

```
$(TARGET): $(OBJECTS)
```

Powyższe makra zostaną rozwinięte przez program make do postaci:

```
Przyklad.elf : wykres.o main.o test.o
```

Tak więc aby zbudować program *przyklad.elf*, make najpierw będzie musiał zbudować pliki *wykres.o*, *main.o* i *test.o*. Oprócz zależności wymaganych do zbudowania programu należy jeszcze określić, co zrobić w sytuacji, kiedy wszystkie zależności są spełnione (a więc mamy już wymagane pliki). W tym celu poniżej listy zależności należy wpisać regułę określającą, co dalej z takimi plikami zrobić:

```
$(CC) $(LDFLAGS) $(OBJECTS) -o $(TARGET)
```

Powyższa reguła mówi, że wszystkie pliki występujące w zależnościach należy połączyć poprzez wywołanie kompilatora avr-gcc, otrzymując w efekcie plik wynikowy — *przyklad.elf*. Tutaj w dosyć nietypowej roli występuje kompilator. Jego zadaniem będzie nie skompilowanie powyższych plików (gdyż są to już skompilowane pliki obiektowe), a wywołanie programu linkera, który dokona konsolidacji powyższych plików; w efekcie uzyskamy plik wynikowy *projekt.elf*.

Aby jednak taka konsolidacja była możliwa, musimy określić, w jaki sposób otrzymać wymagane pliki obiektowe. W tym celu możemy dodać reguły budujące każdy z plików albo dodać jedną regułę, określającą, jak otrzymać dowolny plik obiektowy:

```
% .o : %.c  
$(CC) $(CFLAGS) -c $<
```

Powyższa reguła mówi, że aby zbudować plik obiektowy (*.o*), należy wziąć odpowiedni plik źródłowy (*.c*) i skompilować go przy pomocy programu określonego przez makro CC (a więc avr-gcc), przekazując mu jako opcje komplikacji makro CFLAGS, opcję -c (jak pamiętamy, powoduje ona zachowanie tworzonego pliku obiektowego) oraz nazwę kompilowanego pliku (makro \$<).

Na koniec dla porządku dodajmy jeszcze cel, mający za zadanie usunąć wszystkie pliki powstałe podczas komplikacji:

```
clean:  
-rm -rf $(OBJECTS) $(TARGET)
```

Polecenie rm kasuje wskazane pliki.



Uwaga

Powyższa postać wywołania polecenia `rm` powoduje usunięcie bez pytania wskazanych plików. Jeśli powyższa reguła zostanie zapisana błędnie, możemy usunąć wiele plików, co doprowadzi do katastrofy na użytym dysku twardym.

## Pliki wynikowe

Efektem działania powyższych programów są pliki, które można wykorzystać do zaprogramowania mikrokontrolera. Bezpośrednio w wyniku komplikacji plików źródłowych, a potem linkowania otrzymujemy plik wykonywalny w formacie *elf*. Może on posłużyć do zaprogramowania mikrokontrolera, ale większość programów programujących mikrokontrolery wymaga plików w formacie *Intel HEX*. Poniżej zostaną pokazane podstawowe różnice pomiędzy tymi formatami plików oraz sposoby konwersji pomiędzy nimi.

### Pliki elf

Pliki *elf* (ang. *Executable and Linkable Format*) służą do przechowywania danych, takich jak kod aplikacji czy biblioteki współdzielone. Sam format pliku jest bardzo uniwersalny i, co ważniejsze, otwarty. Plik *elf* składa się z kilku części:

- ◆ nagłówka opisującego zero lub więcej segmentów,
- ◆ nagłówka opisującego zero lub więcej sekcji,
- ◆ danych opisywanych przez wcześniejsze struktury (segmenty i nagłówki).

Taka budowa czyni ten format bardzo uniwersalnym. Przy programowaniu mikrokontrolerów najważniejszą jest możliwość zawarcia w jednym pliku *elf* wielu sekcji. Mogą to być sekcje zawierające kod i dane programu, ale także tzw. sekcje specjalne, które co prawda nie zawierają kodu programu, ale mogą zawierać różne metadane, np. informacje o procesorze, dla którego dany program został skompilowany, konfigurację jego *fuse*- i *lockbitów*. Dzięki temu plik w formacie *elf* zawiera wszystkie niezbędne dane, aby programator mógł przy jego pomocy zaprogramować mikrokontroler. Niestety, ceną, jaką za to płacimy, jest stosunkowo skomplikowana budowa tego pliku, w efekcie nie wszystkie programy potrafią się nim posługiwać. Plikami *elf* potrafi posługiwać się AVR Studio. Niestety, większość innych programów nie rozpoznaje takich plików, w efekcie przed programowaniem musimy z pliku *elf* wyekstrahować zawarte w nim dane (dane dla pamięci FLASH, EEPROM, bity konfiguracyjne procesora). Dane zawarte w plikach *elf* możemy podejrzeć przy pomocy programu narzędziowego `avr-objdump`. Wśród jego licznych opcji przydatna jest opcja `-x`, umożliwiająca wyświetlenie informacji na temat sekcji i symboli zawartych w pliku *elf*:

```
avr-objdump -x ADCNoiseReduction.elf
```

```
ADCNoiseReduction.elf:      file format elf32-avr
ADCNoiseReduction.elf
architecture: avr:4, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x00800100
```

## Program Header:

```

LOAD off 0x00000094 vaddr 0x00000000 paddr 0x00000000 align 2**0
    filesz 0x00000af0 memsz 0x00000af0 flags r-x
LOAD off 0x00000b84 vaddr 0x00800100 paddr 0x00000af0 align 2**0
    filesz 0x00000006 memsz 0x00000006 flags rw-
LOAD off 0x00000b8a vaddr 0x00800106 paddr 0x00800106 align 2**0
    filesz 0x00000000 memsz 0x00000007 flags rw-

```

## Sections:

Idx	Name	Size	VMA	LMA	File off	Algn
0	.data	00000006	00800100	00000af0	00000b84	2**0
			CONTENTS, ALLOC, LOAD, DATA			
1	.text	00000af0	00000000	00000000	00000094	2**1
			CONTENTS, ALLOC, LOAD, READONLY, CODE			
2	.bss	00000007	00800106	00800106	00000b8a	2**0
			ALLOC			
3	.stab	000006cc	00000000	00000000	00000b8c	2**2
			CONTENTS, READONLY, DEBUGGING			
4	.stabstr	00000085	00000000	00000000	00001258	2**0
			CONTENTS, READONLY, DEBUGGING			
5	.debug_aranges	00000040	00000000	00000000	000012dd	2**0
			CONTENTS, READONLY, DEBUGGING			
6	.debug_pubnames	00000093	00000000	00000000	0000131d	2**0
			CONTENTS, READONLY, DEBUGGING			
7	.debug_info	000001fa	00000000	00000000	000013b0	2**0
			CONTENTS, READONLY, DEBUGGING			
8	.debug_abbrev	0000012e	00000000	00000000	000015aa	2**0
			CONTENTS, READONLY, DEBUGGING			
9	.debug_line	000001fb	00000000	00000000	000016d8	2**0
			CONTENTS, READONLY, DEBUGGING			
10	.debug_frame	00000080	00000000	00000000	000018d4	2**2
			CONTENTS, READONLY, DEBUGGING			
11	.debug_str	00000115	00000000	00000000	00001954	2**0
			CONTENTS, READONLY, DEBUGGING			

## SYMBOL TABLE:

00800100	1	d	.data	00000000	.data
00000000	1	d	.text	00000000	.text
00800106	1	d	.bss	00000000	.bss
00000000	1	d	.stab	00000000	.stab
00000000	1	d	.stabstr	00000000	.stabstr
00000000	1	d	.debug_aranges	00000000	.debug_aranges
00000000	1	d	.debug_pubnames	00000000	.debug_pubnames
00000000	1	d	.debug_info	00000000	.debug_info
00000000	1	d	.debug_abbrev	00000000	.debug_abbrev
00000000	1	d	.debug_line	00000000	.debug_line
00000000	1	d	.debug_frame	00000000	.debug_frame
00000000	1	d	.debug_str	00000000	.debug_str
00000000	1	df	*ABS*	00000000	ADCNoiseReduction.c
000003f	1		*ABS*	00000000	_SREG_
000003e	1		*ABS*	00000000	_SP_H_
000003d	1		*ABS*	00000000	_SP_L_
0000034	1		*ABS*	00000000	_CCP_

Wśród licznych wyświetlonych informacji z pewnością użyteczne są informacje na temat architektury, na jaką skompilowano program (w powyższym przykładzie jest to AVR4),

oraz sekcji użytych w programie i ich położenia. Informacja o położeniu sekcji jest szczególnie istotna w przypadku pisania aplikacji *bootloadera*; dzięki temu możemy szybko się zorientować, czy kod programu trafi pod wyznaczony przez nas adres.

## Pliki Intel HEX

Z pliku w formacie *elf* możemy uzyskać pliki w formacie *Intel HEX*. Mają one stosunkowo prostą budowę. Zawierają informacje binarne zakodowane w postaci heksadecymalnej zapisanej w postaci znaków ASCII, dzięki czemu łatwo można podejrzeć zawartość takiego pliku przy pomocy standardowych edytorów tekstu. Plik *hex* jest prostym formatem przechowującym dane binarne, nie zawiera on żadnych informacji o typie zapisanego w takim pliku kodu ani instrukcji, co z nim zrobić. W związku z tym przy posługiwaniu się tego typu plikami na programiście ciąży odpowiedzialność, aby użyć ich do zaprogramowania właściwego typu procesora. Plik ten nie zawiera też żadnych informacji o swoim przeznaczeniu, w efekcie można pliku z kodem programu, przeznaczonego do zaprogramowania pamięci FLASH, użyć do zaprogramowania pamięci EEPROM i vice versa. Podobnie, plik przeznaczony np. dla mikrokontrolera ATMega88 może zostać użyty do zaprogramowania np. mikrokontrolera ATMega128 — nie trzeba dodawać, że w obu przypadkach raczej nie uzyskamy działającej aplikacji.



Wskazówka

Ze względu na swoją budowę długość pliku *hex* nie odpowiada długości zawartej w nim aplikacji. Aby dowiedzieć się, jaka jest długość zawartej w pliku aplikacji, należy użyć polecenia `avr-size`.

Dzięki swojej prostocie format ten jest obsługiwany przez praktycznie wszystkie programatory. Poszczególne linie pliku *hex* rozpoczynają się od znaku :, następnie występuje ciąg cyfr w kodzie szesnastkowym. Dwie pierwsze cyfry określają ilość zawartych w danej linii danych. Następne cztery cyfry oznaczają 16-bitowy adres, pod który dane mają trafić. Aby rozszerzyć możliwy do uzyskania adres, wprowadzono specjalny typ rekordu, określający adres segmentu, pod który mają trafić dane. W efekcie dane zostają umieszczone pod adresem *adres\_segmentu+adres\_rekordu*. Kolejne dwie cyfry określają tzw. typ rekordu. Determinuje on sposób interpretacji danych zawartych w kolejnym polu. Po nim następuje sekwencja danych, zakończona dwoma cyframi sumy kontrolnej — tabela 1.2.

**Tabela 1.2.** Przykładowy wygląd pliku w formacie Intel HEX. BB — ilość bajtów w polu danych, AAAA — adres rekordu, TT — typ rekordu, DD — bajt danych, CC — suma kontrolna. Poniżej fragment pliku hex

:BBAAAATTTDDDDDDDDDDDDDDDDDDCC

:100AD00083B78E7F83BF2091780030917900C90160  
:100AE0000895789480E090E008950895F894FFCFF9  
:060AF0003400120ADE09C9  
:040000050080010076  
:00000001FF

**Ponieważ każdy rekord opatrzony jest sumą kontrolną, plików hex nie da się w prosty sposób edytować — zamiana nawet jednej cyfry powoduje niezgodność**

**sumy kontrolnej.** W przeciwnieństwie do plików zawierających po prostu obrazy pamięci dane w pliku *hex* nie muszą tworzyć spójnego obszaru. W pliku tym mogą występować „dziury”, dzięki czemu w pliku *hex* znajdują się tylko informacje o zawartym w nim kodzie.

Stąd do manipulacji tego typu plikami wykorzystuje się dostarczone z kompilatorem programy narzędziowe *srec\_cat*, *srec\_cmp* i *srec\_info*. Przykłady ich wykorzystania zostaną pokazane w rozdziale 26., poświęconym zabezpieczaniu aplikacji kodem CRC.

Aby z pliku *elf* otrzymać plik *hex*, należy posłużyć się znanim już nam poleceniem *avr-objcopy*:

```
avr-objcopy -O ihex -R .eeprom -R .fuse -R .lock -R .signature plik.elf plik.hex
```

Powyższe polecenie powoduje usunięcie z pliku *elf* wszystkich zbędnych sekcji (pozostaje sekcja *text*, zawierająca kod programu), pozostałe dane zapisane zostają do pliku *plik.hex*. Ponieważ plik w formacie *hex* nie umożliwia jednoczesnego przechowywania danych mających trafić do różnych obszarów pamięci mikrokontrolera (EEPROM, FLASH), dane mające trafić do pamięci EEPROM musimy umieścić w osobnym pliku:

```
avr-objcopy -O ihex -j .eeprom --set-section-flags=.eeprom="alloc,load" --change-section-lma .eeprom=0 --no-change-warnings plik.elf plik.eep
```

Tym razem wywołanie jest nieco bardziej skomplikowane. Pozostawiamy wyłącznie sekcję *eeprom*, zawierającą dane przeznaczone dla tej pamięci, jednocześnie zmieniając jej adres początku na 0 — adresy w pamięci EEPROM zaczynają się właśnie od adresu 0.



Przyjęło się do plików zawierających dane przeznaczone dla pamięci EEPROM dodać rozszerzenie *eep*. Jednak pomimo zmiany rozszerzenia ciągle są to zwykłe pliki w formacie *Intel HEX*.

W podobny sposób możemy uzyskać z pliku *elf* dane pochodzące z innych sekcji, w tym z sekcji specjalnych. Na szczęście, normalnie wywołanie *avr-objcopy* występuje w standardowych skryptach programu *make*, także tych generowanych automatycznie przez AVR Studio, stąd też zazwyczaj konwersją plików z formatu *elf* do *hex* nie musimy się zajmować.

## Biblioteki

Po pewnym czasie programowania każdy programista dysponuje pewnym zestawem gotowych do wykorzystania funkcji. Zwykle dołącza się je w postaci kodów źródłowych do projektu, co jest proste, ale ma jedną wadę — często niepotrzebnie taki kod jest rekompilowany, w efekcie tracimy czas. Ale nie to jest największym problemem. Zwykle w skład takiej „biblioteki” wchodzi wiele plików źródłowych, w efekcie powstaje niezły bałagan, dodatkowo potęgowany przez różne wersje tej samej biblioteki i liczne kopie dołączone do każdego z projektów ją wykorzystujących. Wiele z tych problemów możemy ominąć, wykorzystując biblioteki.

Opisując działanie linkera, wspomniano, że jego zadaniem jest łączenie symboli generowanych przez kompilator, znajdujących się w różnych plikach powstały w wyniku kompilacji. Tę jego funkcję wykorzystuje się właśnie w celu stworzenia biblioteki. Jak pamiętamy, po skompilowaniu pliku źródłowego powstaje z niego plik obiektowy, z rozszerzeniem *.o*. Plik ten zawiera skompilowany kod, który następnie zostanie skonsolidowany z innymi plikami obiektowymi przez linker. Wynika z tego, że zamiast dołączać do projektu kolejne pliki zawierające kod źródłowy, równie dobrze można dołączyć skompilowane pliki obiektowe. Dzięki temu nie musimy powtarzać ich kompilacji — są one w postaci gotowej do wykorzystania przez linker. Ciągle pozostaje jednak problem ich ilości — z każdego pliku źródłowego powstaje jeden plik obiektowy. W efekcie do projektu czasami należałoby dołączyć kilkadziesiąt, a nawet kilkaset takich plików. Aby tę niedogodność ominąć, wprowadzono właśnie biblioteki. Pliki obiektowe zawierające funkcje podobne do siebie albo realizujące powiązane ze sobą zadania warto połączyć razem w większą strukturę, nazywaną biblioteką.



Biblioteki to nic innego jak kolekcje plików obiektowych (*.o*), połączone przy pomocy programu archiwizującego w jeden plik.

## Projekt biblioteki

Biblioteka składa się z dwóch części:

- ◆ pliku nagłówkowego, zawierającego udostępnione przez bibliotekę funkcje (tzw. funkcje biblioteczne), zmienne i inne symbole;
- ◆ pliku *lib* zawierającego prekompilowany kod biblioteki.



Ponieważ funkcje są prekompilowane na konkretnej platformie sprzętowej, skompilowaną bibliotekę można wykorzystać tylko na platformie, dla której została ona skompilowana.

Szczególnie istotny jest plik nagłówkowy i udostępnione w nim prototypy funkcji. Prototypów funkcji wewnętrznych biblioteki, które nie powinny być dostępne poza biblioteką, nie należy umieszczać w pliku nagłówkowym biblioteki. Zwykle do tego celu stosuje się inne pliki nagłówkowe, które nie są dostępne dla użytkowników. Udostępnione prototypy funkcji powinny być dobrze przemyślane i uniwersalne, tak aby nie zachodziła konieczność ich modyfikacji w kolejnych wersjach biblioteki. **Pamiętajmy, że jakakolwiek modyfikacja udostępnionego pliku nagłówkowego wiąże się z koniecznością dostosowania wszystkich zależnych od danej biblioteki programów. Z kolei modyfikacja „wewnętrznych” plików nagłówkowych i źródeł biblioteki wiąże się tylko z prostą rekompilacją wykorzystującego ją programu.**

Ponieważ biblioteki zawierają duże zbiory funkcji, ich dołączenie do programu może wiązać się ze znacznym zwiększeniem objętości kodu źródłowego. Nie ma problemu, jeśli wszystkie funkcje są wykorzystywane. Co jednak jeśli wykorzystujemy tylko jedną funkcję biblioteczną? Aby uniknąć sytuacji, w której wykorzystanie jednej lub kilku funkcji bibliotecznych wiąże się z koniecznością dołączenia wszystkich funkcji danej

biblioteki, należy wykorzystać pewien trick. Otóż kod źródłowy każdej funkcji bibliotecznej należy umieścić w osobnym pliku źródłowym. W efekcie każda funkcja znajdzie się w odrębnym pliku *.o*. O ile bibliotekę do projektu dołączają się jako całość, to linker może zidentyfikować funkcje biblioteczne, do których nie odwołujemy się w kodzie pisanej programu. Jeśli funkcje te znajdują się w oddzielnych plikach *.o*, to brak odwołania się do takiej funkcji oznacza brak odwołania się do zawierającego ją pliku *.o*, w efekcie plik ten zostanie na etapie konsolidacji programu usunięty. Dzięki temu w ostatecznym programie znajdzie się wyłącznie kod funkcji wykorzystanych w programie. Kody pozostałych funkcji nie zostaną dołączone. Jak widać, źródła biblioteki należy dobrze zaprojektować przy jej tworzeniu. Oczywiście, w przypadku funkcji, które zawsze występują razem, nie ma sensu umieszczać ich w osobnych plikach — takie funkcje można umieścić w jednym pliku źródłowym.

Czasami spotykamy się z sytuacją odwrotną — do danej funkcji bibliotecznej nie występuje odwołanie w programie, lecz z różnych powodów muszą one znaleźć się w finalnym kodzie programu. Przykładem takich funkcji są funkcje obsługi przerwań — nigdzie ich jawnie nie wywołuje się w programie, lecz muszą one być dołączone, aby wektory tabeli wektorów przerwań mogły na nie wskazywać. Tu z pomocą przychodzą rozszerzenia języka C wspierane przez gcc:

```
void test() __attribute__ ((used));
```

Atrybut used mówi linkerowi, że nawet jeśli do danej funkcji nie ma odniesień w kodzie, musi się ona znaleźć w kodzie wynikowym. W efekcie funkcja taka nie zostanie usunięta z kodu wynikowego na etapie linkowania programu.

## Tworzenie biblioteki

Jak wyżej wspomniano, biblioteka jest czymś w rodzaju archiwum plików obiektowych. Archiwum to tworzy się przy pomocy dołączonego do kompilatora programu narzędzia ar, który dla platformy AVR nosi nazwę avr-ar. Ogólna postać wywołania tego programu wygląda następująco:

```
avr-ar opcje <nazwa_biblioteki> <lista_plików_obj>
```

Wśród licznych opcji programu ar interesujące są trzy:

- ◆ c — powoduje utworzenie biblioteki.
- ◆ s — powoduje utworzenie lub uaktualnienie indeksu symboli zgromadzonych w bibliotece, co ułatwia pracę linkera.
- ◆ r — powoduje podmianę pliku znajdującego się w archiwum plikiem podanym na liście plików do dodania.

Nazwa biblioteki musi zaczynać się prefiksem *lib*, a kończyć sufiksem *.a* i, co ważne, zarówno prefiks, jak i sufiks muszą być pisane małymi literami. Jest to szczególnie istotne w środowiskach, w których wielkość liter w nazwach plików jest rozróżnialna. Jako wynik działania programu ar dostajemy gotową do użycia bibliotekę.

Dla przykładu, chcąc stworzyć bibliotekę *libtest.a* z plików obiektowych *test1.o* i *test2.o*, należy wydać polecenie:

```
avr-ar rcs libtest.a test1.o test2.o
```

Listę plików wchodzących w skład archiwum można podejrzeć poleceniem:

```
avr-ar t <nazwa_archiwum>
```

## Dołączanie biblioteki do programu

Aby dołączyć do programu bibliotekę, należy wykorzystać opcję linkera **-l**, podając nazwę dołączanej biblioteki. Nazwa dołączonej biblioteki powinna zawierać tylko unikalny dla biblioteki rdzeń, opcja **-l** sama dołączy do niej prefiks *lib* i sufiks *.a*. W efekcie wywołanie **-lm** spowoduje dołączenie biblioteki o nazwie *libm.a*, podobnie opcja **-lprintf\_flt** spowoduje dołączenie biblioteki *libprintf\_flt*. Stąd też wszystkie biblioteki powinny mieć nazwy rozpoczynające się od *lib*, a kończyć się rozszerzeniem *.a*.

W opcji wywołania może wystąpić wiele razy opcja **-l**, w efekcie biblioteki są przeszukiwane w kolejności, w jakiej zostały przekazane linkerowi. Po odnalezieniu symbolu przeszukiwanie jest kończone. Stąd też w różnych bibliotekach może być zdefiniowany ten sam symbol, w dodatku zdefiniowany w różny sposób. W takiej sytuacji kolejność dołączania bibliotek będzie decydowała o sposobie, w jaki działa program. Tu warto wrócić do rozdziału poświęconego arytmetyce zmiennej pozycyjnej i funkcjom *printf*/*scanf* — ich działanie zależy od tego, czy dołączono bibliotekę matematyczną (*libm*), czy tylko bibliotekę standardową.

Oprócz kolejności bibliotek przekazywanej linkerowi ważna jest także kolejność ich przeszukiwania. Linker szuka bibliotek w katalogach określonych przy pomocy parametru **-L**:

```
-L"C:\libdir"
```

Podobnie jak opcja **-l**, także opcja **-L** może występować wielokrotnie. Katalogi przeszukiwane są w kolejności ich występowania w opcjach wywołania linkera.



Należy pamiętać, że jeśli katalogi w nazwie zawierają spacje, należy całą ścieżkę umieścić w cudzysłowach.

Podsumowując, bibliotekę dodajemy w programie poprzez umieszczenie w pliku *Makefile* następującej linii:

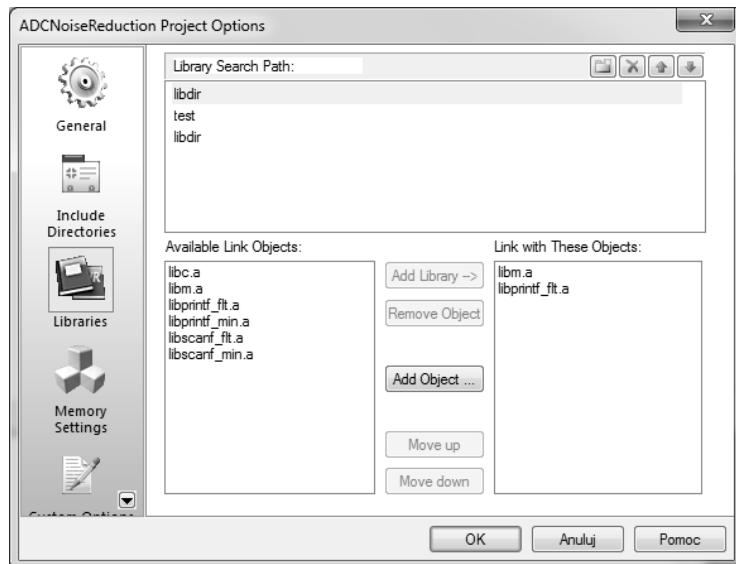
```
LIBS = -lm -lprintf_flt
```

Powyższa linia powoduje dołączenie do programu bibliotek *libm* i *libprintf\_flt*. Dodatkowo możemy określić katalogi, w których znajdują się biblioteki:

```
LIBDIRS = -L"C:\test"
```

To samo możemy osiągnąć w AVR Studio, wybierając z menu *Project/Configuration Options/Libraries* (rysunek 1.13).

**Rysunek 1.13.**  
*Dodawanie bibliotek w programie AVR Studio. W górnej części okna możemy dodawać ścieżki do bibliotek. Na podstawie wybranej konfiguracji automatycznie zostaną dodane odpowiednie opcje do skryptu Makefile*



## Funkcje „przestarzałe”

Czasami, pisząc bibliotekę, prototypy pewnych funkcji zmieniamy, lecz ich stare wersje zostawiamy dla zachowania kompatybilności. Takie funkcje w przyszłości mogą zostać z biblioteki usunięte, w związku z tym nie chcemy zachęcać programistów wykorzystujących daną bibliotekę do ich używania. W tym celu, definiując prototyp funkcji, należy dodać do niego atrybut deprecated:

```
void test() __attribute__ ((deprecated));
```

Użycie w kodzie takiej funkcji spowoduje wygenerowanie ostrzeżenia:

```
warning: 'test' is deprecated (declared at .../ADCNoiseReduction.c:23)
```

Dzięki temu programista wie o naszych zamiarach i może użyć innej funkcji.

## Nadpisywanie funkcji bibliotecznych

W pewnych sytuacjach w bibliotece definiuje się funkcje, które mają zostać użyte jako funkcje domyślne, lecz użytkownik powinien mieć możliwość ich zmiany, poprzez zdefiniowanie własnych. Dzięki temu możemy zmienić domyślne działanie biblioteki. Normalnie nie jesteśmy w stanie ponownie zdefiniować funkcji, która została wcześniej (w tym przypadku w bibliotece) zdefiniowana. Aby dopuścić taką możliwość, należy posłużyć się atrybutem weak:

```
void test() __attribute__ ((weak));
```

Atrybut ten powoduje, że zamiast symbolu globalnego zdefiniowany będzie słaby symbol, który można będzie nadpisać, redefiniując funkcję. W przypadku kiedy funkcja `test()` zostanie ponownie zdefiniowana, kompilator nie wygeneruje błędu — po prostu użyje nowej definicji, a poprzednia zostanie usunięta z programu.

## Usuwanie niepotrzebnych funkcji i danych

Jak wspomniano, dzięki zdefiniowaniu każdej funkcji bibliotecznej w osobnym pliku źródłowym linker ma możliwość usunięcia kodu funkcji nieużywanych poprzez usunięcie zawierającego ten kod pliku obiektowego. Rozwiążanie to jest zdecydowanie rozwiązaniem polecanym, aczkolwiek niezbyt wygodnym. W pewnych sytuacjach możemy skorzystać z opcji kompilatora, które umożliwiają osiągnięcie tego samego efektu bez rozdziału funkcji do oddzielnych plików. Efekt ten możemy uzyskać, przekazując kompilatorowi następujące opcje:

- ◆ `-ffunction-sections` — powoduje, że każda funkcja użyta w programie znajdzie się w osobnej sekcji — efekt końcowy jest więc taki, jakby każda funkcja znajdowała się w osobnym pliku źródłowym.
- ◆ `-fdata-sections` — opcja ta daje efekt podobny do wcześniejszej, z tym że dotyczy on danych programu. Każda zmienna znajdzie się we własnej sekcji.

Powyższe dwie opcje powodują jedynie umieszczenie każdej struktury (funkcji lub danych) we własnych sekcjach, co stwarza możliwości optymalizacji poprzez usunięcie niewykorzystanych obiektów, lecz aby to było możliwe, należy poinstruować linker, aby sekcje, do których nie ma referencji w programie, zostały usunięte. Za tą funkcję odpowiada opcja linkera `-gc-sections`. Ponieważ jest to opcja linkera, przekazujemy ją przy wywołaniu gcc w następujący sposób:

```
gcc -Os -fdata-sections -ffunction-sections test.c -o test.elf -Wl,--gc-sections
```



## Rozdział 2.

# Programowanie mikrokontrolera

Po wygenerowaniu plików wynikowych należy ich zawartość umieścić w pamięci mikrokontrolera. Dzięki temu po restarcie procesor będzie mógł rozpoczęć wykonywanie programu. Procesory AVR dysponują możliwością programowania „w układzie” przy pomocy interfejsu ISP, część może być programowana poprzez interfejsy JTAG, debugWire, PDI, TPI, a w przypadku procesorów posiadających interfejs USB można także programować procesor poprzez wbudowany *bootloader*. W tym ostatnim przypadku nie da się jednak zmieniać konfiguracji *fusebitów*. Każda z metod programowania ma swoje zalety i wady.

## Podłączenie — uwagi ogólne

Każdy programator łączy się z układem docelowym przy pomocy dedykowanych wyprowadzeń. Dla programatorów szeregowych jest to zwykle 4 – 5 wyprowadzeń, dla równoległych znacznie więcej. Wykorzystanie wyprowadzeń mikrokontrolera do programowania ogranicza możliwość ich wykorzystania do innych celów. Najlepiej, jeśli takie wyprowadzenia nie będą wykorzystywane do niczego innego — w układzie będą one podłączone wyłącznie do gniazda łączącego z programatorem. Jednak w układach posiadających niewielką liczbę wyprowadzeń nie zawsze jest to możliwe. Stąd też powinniśmy pamiętać, aby podłączone do tych wyprowadzeń urządzenia nie obciążały ich zbytnio (w trybie programowania będą one obciążały wyjście programatora). Z tego powodu nie zaleca się podłączać do nich np. diod LED, nie należy na tych liniach dodawać także kondensatorów, szczególnie o większych pojemnościach ( $>1\text{ nF}$ ). Dodatkowo jeśli jakieś wyprowadzenie jest wejściem (czyli wyjściem układu programującego), nie należy łączyć do niego innych wyjść — w takiej sytuacji w trakcie programowania powstanie konfliktu pomiędzy wyjściem programatora a wyjściem układu korzystającym z tego pinu.



## Wskazówka

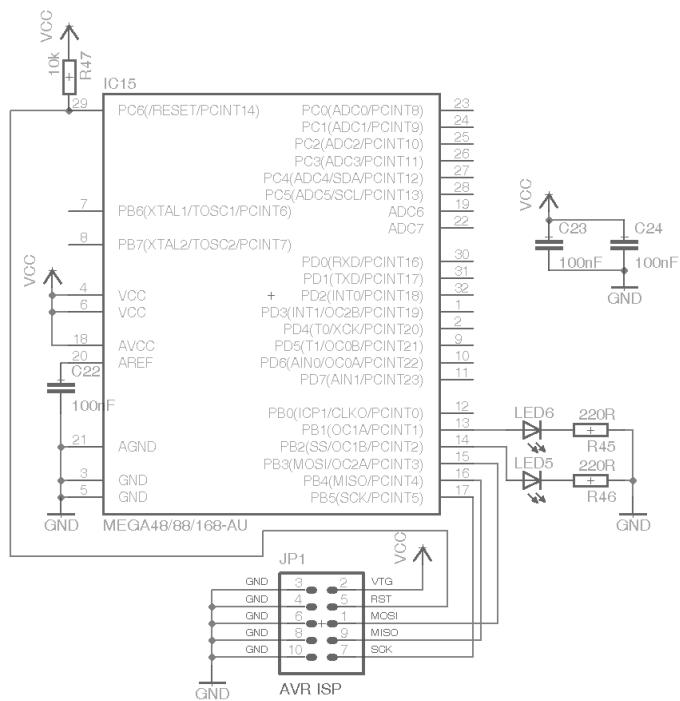
Pamiętajmy, że dla programatora nie ma znaczenia, jak w programie skonfigurowane są piny wykorzystywane do programowania.

Programator na czas programowania wprowadza procesor w stan *RESET*, co wiąże się z wprowadzeniem wszystkich pinów *IO* w stan wysokiej impedancji.

Szczególną uwagę należy zwrócić na podłączenie sygnału *RESET*. Aby wejść w tryb programowania, programator musi mieć możliwość wymuszenia na tej linii stanu niskiego (lub +12 V w przypadku programatorów wysokonapięciowych). W efekcie problem może wystąpić, jeśli w układzie używane są zewnętrzne układy generujące *RESET* lub monitorujące zasilanie. Przykładowy schemat podłączenia procesora do programatora ISP pokazano na rysunku 2.1.

## Rysunek 2.1.

Przykład podłączenia programatora ISP do mikrokontrolera ATmega88



## Wskazówka

W dalszych rozdziałach książki pokazane schematy, dla uproszczenia i większej przejrzystości, nie będą już zawierały elementów pokazanych na schemacie z rysunku 2.1, czyli gniazda ISP, kondensatorów odsprzęgających oraz połączeń z zasilaniem (Vcc i GND).

Aby pokazane w dalszej części układy działały poprawnie, należy zapewnić poprawne połączenie elementów pokazanych na rysunku 2.1. Szczególnie istotne jest podłączenie w każdym procesorze wszystkich występujących w nim wyprowadzeń zasilania (Vcc i GND). Przy braku podłączenia niektórych wyprowadzeń układ może działać

niestabilnie i stwarzać problemy. Drugim istotnym elementem są kondensatory odsprzęgające C23 i C24. Są to elementy, których zadaniem jest odsprzęganie zasilania, a ich znaczenie rośnie wraz ze wzrostem stopnia skomplikowania układu.

## Problemy

Najczęstsze problemy z zaprogramowaniem procesora:

1. W przypadku programowania w trybie ISP „zablokowanie” procesora, poprzez niewłaściwą konfigurację *fusebitów*.
2. Nieprawidłowa częstotliwość sygnału *SCK* (w przypadku programatorów ISP). Jeśli podejrzewamy taki problem, należy zmniejszyć szybkość programowania. W żadnym przypadku nie może ona przekroczyć  $\frac{1}{4}$  częstotliwości taktowania procesora.
3. Zbyt długi kabel łączący programator z układem. Im dłuższy kabel, tym większe ryzyko niepoprawnej pracy układu. Zwykle problem ten objawia się niestabilną pracą programatora.
4. Błędne podłączenie sygnałów. Zawsze warto się upewnić, że wszystkie sygnały zostały prawidłowo połączone z odpowiednimi wyprowadzeniami procesora.
5. Pomyłkowe podłączenie programatora nie do wyprowadzeń związanych z ISP, lecz do wyprowadzeń związanych z interfejsem SPI (oznaczenia linii sygnałowych są podobne). Problem ten dotyczy głównie procesorów ATMega128.
6. Wybór niewłaściwego programatora lub niewłaściwego trybu programowania.
7. Zablokowanie wykorzystywanego interfejsu (dotyczy głównie próby programowania przy wyłączonym interfejsie JTAG lub próby programowania przy pomocy ISP, z włączonym interfejsem debugWire).

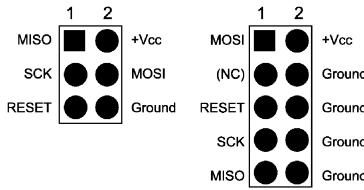
## Programatory ISP

Prawie każdy procesor AVR dysponuje możliwością programowania przy pomocy interfejsu ISP (ang. *In-system Programming Interface*). Interfejs ten wykorzystuje do programowania piny *RESET*, *MISO*, *MOSI* oraz *SCK*. Zwykle piny te pokrywają się z analogicznymi wyprowadzeniami interfejsu SPI, lecz nie zawsze tak jest. Jednym z takich wyjątków jest procesor ATMega128. Stąd też zawsze należy dokładnie sprawdzić, jakie wyprowadzenia procesora wykorzystywane są do programowania przy pomocy interfejsu ISP. Informacje o wykorzystanych wyprowadzeniach znajdują się w nocy katalogowej procesora, w sekcji *Memory Programming/Serial Downloading*. Programatory ISP mają znormalizowany układ sygnałów wyprowadzonych na złącze programujące, pokazany na rysunku 2.2.

Istnieją dwa typy złącza ISP — jedno mniejsze, 6-pinowe, oraz większe 10-pinowe. Odstęp pomiędzy pinami wynosi 2,54 mm, chociaż w nowszych konstrukcjach Atmela

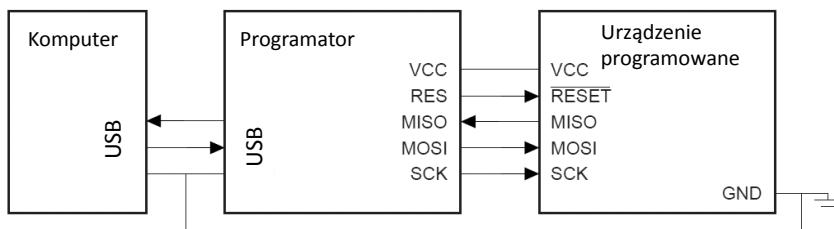
**Rysunek 2.2.**

Rozkład sygnałów standardowych programatorów.  
Kwadratem oznaczono wyprowadzenie o numerze 1



ISP Connectors: 6-pin &amp; 10-pin

spotyka się rozstaw pinów 1,27 mm. Umożliwia to zmniejszenie rozmiarów złącza programującego na płytce. Pewnego omówienia wymaga przeznaczenie pinu oznaczonego jako +Vcc. Do tego pinu należy podłączyć napięcie zasilające układ. Napięcie z tego pinu wykorzystywane jest przez programator do zasilania buforów wyjściowych, dzięki czemu programator dostosowuje poziomy napięć na pozostałych pinach programatora do napięć panujących w programowanym układzie. Część programatorów posiada także specjalną zworkę przełączającą napięcie. W takiej sytuacji jedna z pozycji powoduje zasilenie programatora z programowanego układu, w drugiej pozycji to programator zasila programowany układ. Programator łączy się z programowanym układem tak, jak pokazano na rysunku 2.3.



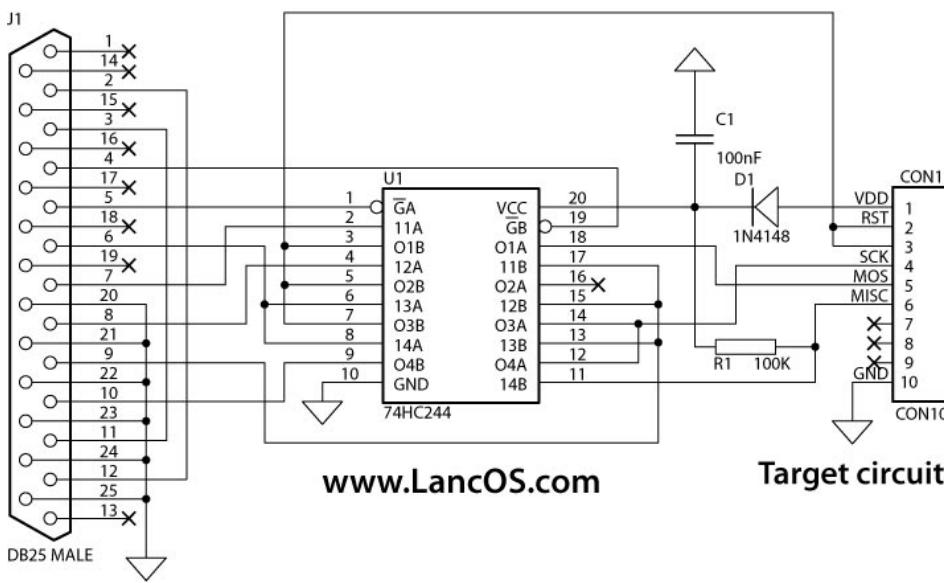
**Rysunek 2.3.** Połączenie programowanego układu z programatorem i komputerem PC. Masy wszystkich urządzeń muszą zostać połączone razem

Łącząc programator z komputerem i programowanym układem, należy zwracać uwagę na potencjał masy. W komputerach klasy PC, ze względu na budowę zasilacza, masa (obudowa komputera) przy braku zerowania ma potencjał ok. 115 V względem ziemi (wynika to z istnienia w zasilaczu układu filtrującego). W efekcie przy braku zerowania komputera lub niepoprawnym zerowaniu może dojść do uszkodzenia programatora lub programowanego układu. Aby uniknąć takich przykrych niespodzianek, można zaopatrzyć się w programator z optoizolowanymi wyjściami, lecz jest to dodatkowy, spory wydatek.

## Budowa programatora

Programatory ISP są jednymi z najprostszych w budowie, w związku z tym każdy może poskładać sobie taki programator, dosłownie z niczego. W szczególnie dobrej sytuacji są tu użytkownicy posiadający komputery z wyprowadzonym portem równoległym. W takiej sytuacji programator może być zwykłą przejściówką pomiędzy portem komputera a gniazdem ISP. Układ taki jest niezwykle prosty, lecz niezalecany. Jakkolwiek błąd w połączeniach może bardzo łatwo doprowadzić do uszkodzenia portu równoległego,

dodatkowo jego niewielka wydajność prądowa powoduje znaczne ograniczenie maksymalnej długości przewodu łączącego komputer z programowanym układem (w praktyce do kilkunastu cm). Stąd znacznie lepszym rozwiązaniem jest to pokazane na rysunku 2.4.



Rysunek 2.4. Schemat prostego programatora ISP podłączanego do portu równoległego komputera

Jak widać, programator taki składa się z bufora 74XX244 (nie musi to być układ serii HC). Jego wprowadzenie umożliwia znaczne wydłużenie przewodu łączącego komputer z programatorem, nawet do 1 m i więcej (należy mieć na uwadze, że zbyt długi przewód łączący nie jest zalecany i w pewnych okolicznościach może prowadzić do problemów z programowaniem). Wprowadzenie tego układu chroni także port równoległy. W przypadku błędnego podłączenia zasilania np. do pinów programatora uszkodzeniu ulegnie tylko tani układ buforujący, ochraniając port równoległy. Wykonanie takiego programatora to koszt rzędu kilku złotych, lecz już za kilkanaście złotych można kupić programatory bardziej rozbudowane, których zaletami są:

- ◆ bezpośrednia współpraca z AVR Studio;
- ◆ możliwość programowania układów zasilanych innym napięciem niż 5 V;
- ◆ współpraca z portem USB mikrokontrolera.

Szczególnie ta ostatnia cecha jest pożądana. Porty USB, w przeciwieństwie do równoległych, występują praktycznie w każdym urządzeniu, lecz ich największą zaletą jest możliwość czerpania energii z takiego portu. Stwarza to możliwość nie tylko zasilania samego programatora, ale także zasilania programowanego układu (tu jednak trzeba mieć na uwadze ograniczoną do ok. 0,5 A wydajność prądową portu USB).

## Programator AVRISP

Jest to prosty programator z możliwością podłączenia poprzez port szeregowy RS232 lub USB. Na rynku dostępne są liczne klony tego układu, w efekcie można go kupić już za kilkanaście złotych, co czyni go szczególnie interesującym dla amatora. W stosunku do prostych programatorów, posiadających tylko bufor, jego zaletą jest możliwość programowania układów zasilanych napięciem w granicach 2,7 – 5,5 V. Klasyczny programator AVRISP zasilany jest z urządzenia programowanego, lecz jego wersje na USB często posiadają zworki, umożliwiające wybór źródła zasilania. Przy jego pomocy można programować szeroką gamę modeli procesorów AVR. Wyjątkiem jest tu tylko rodzina AVR Xmega oraz AVR32, wymagające programatora AVRISP mkII. Programator ten posiada wbudowany procesor, którego firmware kontroluje proces programowania. Wraz z uaktualnianiem AVR Studio firmware ten też może zostać uaktualniony, w efekcie poszerza się lista obsługiwanych procesorów.

Programator ten jest rozwiązaniem tanim, lecz warto mieć na uwadze, że obecnie nie jest on już praktycznie rozwijany przez firmę Atmel. Stąd potencjalnie mogą być problemy z jego wykorzystaniem z najnowszymi modelami procesorów.

## Programator AVRISP mkII

Rozwiązaniem dla bardziej zaawansowanych amatorów i osób, które półprofesjonalnie chcą się zajmować mikrokontrolerami AVR, jest programator AVRISP mkII. Również ten programator dostępny jest w postaci klonów, w efekcie można go kupić za cenę ok. 100 – 150 zł. Jest to programator aktywnie wspierany przez firmę Atmel, wspierający wszystkie rodziny procesorów AVR (łącznie z procesorami Xmega oraz AVR32). Wspiera także procesory ATTiny, nieposiadające interfejsu ISP, dzięki możliwości wykorzystania interfejsu TPI.

Programator ten może programować układy zasilane napięciem od 1,8 do 5,5 V, dodatkowo można regulować częstotliwość sygnału zegarowego taktującego transmisję w zakresie 50 Hz – 8 MHz. Ma to istotną zaletę w przypadku programowania układów niskonapięciowych, taktowanych z wolnych zegarów, np. kwarców zegarkowych o częstotliwości 32 768 Hz.



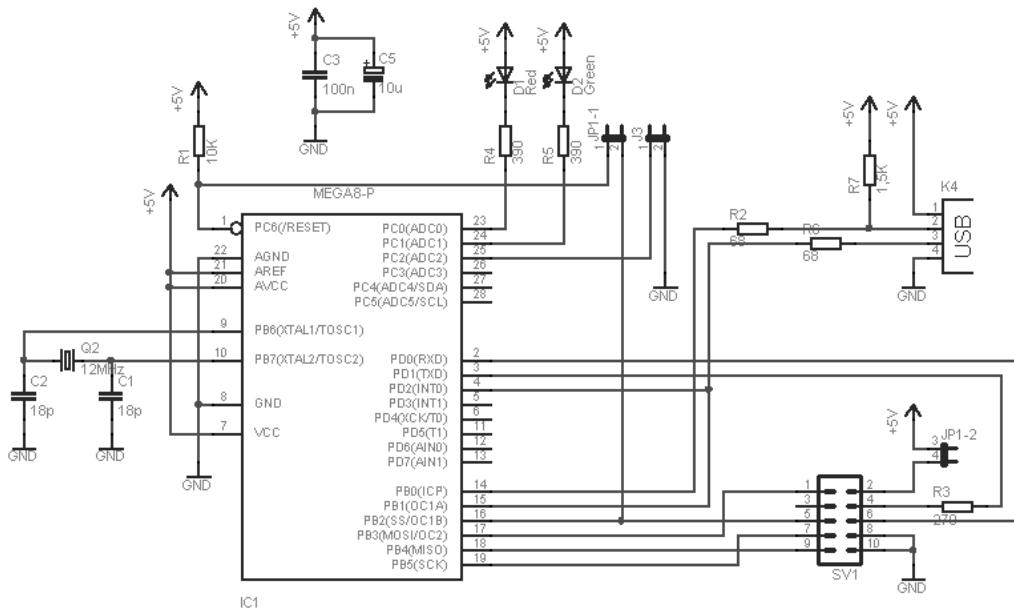
Maksymalna prędkość programowania wynika z ograniczeń interfejsów szeregowych — częstotliwość linii SCK nie może być większa niż czterokrotność częstotliwości taktującej rdzeń procesora.

Tak więc wykorzystanie programatora AVRISP mkII umożliwia programowanie procesorów taktowanych zegarem od 200 Hz wzwyż.

Programator ten współpracuje z interfejsem USB, posiada także wyjścia zabezpieczone przez zwarcie.

# Programator USBASP

Programator ten jest niezwykle popularny ze względu na jego prostotę oraz niską cenę. Dodatkową zaletą tego programatora jest wsparcie ze strony WinAVR oraz bardzo popularnego programu AVRDUDE. Schemat tego programatora pokazano na rysunku 2.5.



**Rysunek 2.5.** Schemat programatora USBASP. Programator ten zawiera mikrokontroler sterujący procesem programowania układu docelowego. Dzięki temu programator ten jest niezależny od przebiegów czasowych generowanych przez komputer. Łączność z komputerem następuje poprzez złącze USB, stąd też programator czerpie zasilanie. Po zwarciu zworki JP1-2 możliwe jest także zasilanie z portu USB układu programowanego

Programator ten umożliwia także programowanie procesorów taktowanych zegarami o niskiej częstotliwości. Przy pomocy zworki JP3 można przełączać częstotliwość linii *SCK* z 375 kHz na 8 kHz, co umożliwia programowanie układów taktowanych kwarcem zegarkowym o częstotliwości 32 768 Hz. Zwarcie zworki JP1-1 umożliwia zaprogramowanie lub uaktualnienie oprogramowania programatora poprzez jego złącze ISP.

## **Kilka procesorów w jednym układzie**

Sporadycznie zdarza się, że na jednej płytce znajduje się więcej niż jeden procesor AVR i każdy powinien mieć zapewnioną możliwość programowania. Najchętniej w takiej sytuacji chcielibyśmy móc korzystać tylko z jednego gniazda programującego. Konfiguracja taka jest możliwa, musimy tylko pamiętać o spełnieniu pewnych dodatkowych założeń. W takiej sytuacji sygnały z programatora (*RESET*, *MISO*, *MOSI*) powinny być rozprowadzone do wszystkich mikrokontrolerów. Natomiast sygnał *SCK* musi pozostać

rozdzielony. Przy takiej konfiguracji wyboru programowanego mikrokontrolera dokonuje się poprzez wybranie mikrokontrolera, do którego doprowadzony zostanie sygnał *SCK*. Układ taki może działać, ponieważ aby procesor wszedł w tryb programowania (a co za tym idzie, linie *MOSI* i *MISO* stały się aktywne), musi być spełnionych kilka założeń. Po pierwsze, programator musi zapewnić aktywność sygnału *RESET*. Dzięki utrzymywaniu go w stanie aktywnym wyprowadzenia wszystkich mikrokontrolerów przechodzą w stan wysokiej impedancji. Dzięki temu nie zakłócają one transmisji. Uaktywnienie trybu programowania wymaga w takiej sytuacji doprowadzenia do wejścia *SCK* odpowiedniego przebiegu. Ponieważ przebieg taki zostanie doprowadzony wyłącznie do wybranego procesora, inne procesory pozostaną nieaktywne, z wyprowadzeniami w stanie wysokiej impedancji.

W przypadku gdy na płytce znajduje się jeden większy procesor i jeden lub więcej procesorów ze stosunkowo niewielką ilością pamięci FLASH, można rozważyć jeszcze jedną możliwość. Funkcję programatora może przejąć procesor „większy”, odpowiednio sterując wyprowadzeniami odpowiedzialnymi za programowanie innych procesorów. W takiej sytuacji ich przeprogramowanie wymaga wczytania do procesora kontrolującego pozostałe odpowiedniego programu oraz zawartości pamięci FLASH pozostałych procesorów. Rozwiążanie takie jest stosunkowo proste, lecz wymaga takiego podłączenia wszystkich mikrokontrolerów, aby ich wyprowadzenia programujące były dostępne dla procesora nadzawanego.

## Programatory JTAG

Programatory wykorzystujące interfejs JTAG są o wiele droższe, ale oprócz możliwości programowania przy ich pomocy procesora oferują także możliwość debugowania. Obecnie na rynku występują dwie wersje programatora JTAG dla mikrokontrolerów AVR — JTAGICE oraz JTAGE II. Ten drugi cechuje się bardzo wysoką ceną (ok. 700 – 1200 zł), ale oferuje możliwość programowania wszystkich mikrokontrolerów AVR wyposażonych w interfejs JTAG. Za jego pomocą można także programować mikrokontrolery AVR32. Możliwości programatora JTAGICE są skromniejsze, ale za to jego cena jest niewiele wyższa niż programatora ISP. Programator ten łączy się z programowanym układem przy pomocy gniazda o innym rozkładzie sygnałów niż w przypadku programatora ISP — rysunek 2.6.

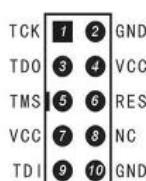
**Rysunek 2.6.**

Rozkład sygnałów

na złączu JTAG.

Pin 1 oznaczono

kwadratem



| Atmel Standard 10-pin JTAG layout

Interfejs JTAG wykorzystuje pięć sygnałów: *RESET*, *TCK*, *TMS*, *TDI* oraz *TDO*. Do pinu 7 (VCC) należy doprowadzić napięcie zasilające tylko w sytuacji, w której programator ma być zasilany z układu. Jeśli programator ma własne zasilanie, pin 7 można pozostawić

niepodłączony. Z kolei pin 4 dostarcza napięcia umożliwiającego programatorowi dostosowanie poziomu napięć na liniach *RESET*, *TCK*, *TMS*, *TDI* i *TDO* do napięć panujących w układzie. Na podstawie napięcia na tej linii programator wykrywa także podłączenie do układu programowanego. Wyrowadzenia oznaczone jako *NC* należy pozostawić niepodłączone.



Wskazówka

Aby móc korzystać z tego trybu, procesor musi obsługiwać interfejs JTAG, a *fusebit JTAGEN* musi być zaprogramowany (mieć wartość 0).

Programowanie przy użyciu interfejsu JTAG ma liczne zalety:

- ◆ Jest 3 – 4 razy szybsze w stosunku do programowania przy użyciu interfejsu ISP.
- ◆ Podobnie, znacznie szybsze jest także programowanie pamięci EEPROM.
- ◆ Umożliwia zmianę *fusebitów* określających źródło sygnału zegarowego, niezależnie od ich poprzednich wartości. Interfejs JTAG sam generuje zegar dla układu docelowego, stąd wybranie nawet błędnych wartości nie blokuje możliwości dalszego programowania (odmiennie niż w przypadku interfejsu ISP).
- ◆ Istnieje możliwość łączenia urządzeń w konfigurację *daisy-chain*, umożliwiającą programowanie wielu urządzeń przy pomocy jednego złącza JTAG.
- ◆ Istnieje możliwość programowania nie tylko mikrokontrolerów AVR, ale także innych układów kompatybilnych ze standardem JTAG (np. FPGA).



Wskazówka

Niezwykle istotną zaletą interfejsu JTAG jest możliwość debugowania przy jego pomocy programu w trakcie jego działania w docelowym układzie elektronicznym.

Możliwość taka jest wprost trudna do przecenienia, szerzej zostanie opisana w rozdziale 29.

## Programator JTAGICE

Cena tego układu porównywalna jest z ceną dobrego programatora ISP. Jest to więc propozycja dla hobbystów zdecydowanie poważniej myślących o zajęciu się budowaniem układów w oparciu o mikrokontrolery AVR. Zastosowania tego programatora ogranicza stosunkowo niewielka liczba wspieranych układów [ATmega16(L), ATmega162(L), ATmega169(L or V), ATmega32(L), ATmega323(L), ATmega64(L), ATmega128(L)]. Lecz nawet pomimo tej wady warto rozważyć jego zakup, szczególnie jeśli jesteśmy w posiadaniu płytki rozwojowej zawierającej jeden z wyżej wymienionych procesorów. Programowanie przy jego pomocy jest nie tylko szybsze, lecz przede wszystkim udostępnia szerokie możliwości debugowania układu w systemie. Dzięki temu nawet jeśli pisany program będzie docelowo działał na innym typie procesora, łatwiej jest napisać aplikację na jednym ze wspieranych przez JTAGICE procesorów, a następnie ją tylko zmodyfikować dla potrzeb procesora docelowego. Użycie interfejsu JTAG umożliwia nie tylko debugowanie samego programu, ale także sprawdzenie stanu wszystkich bloków

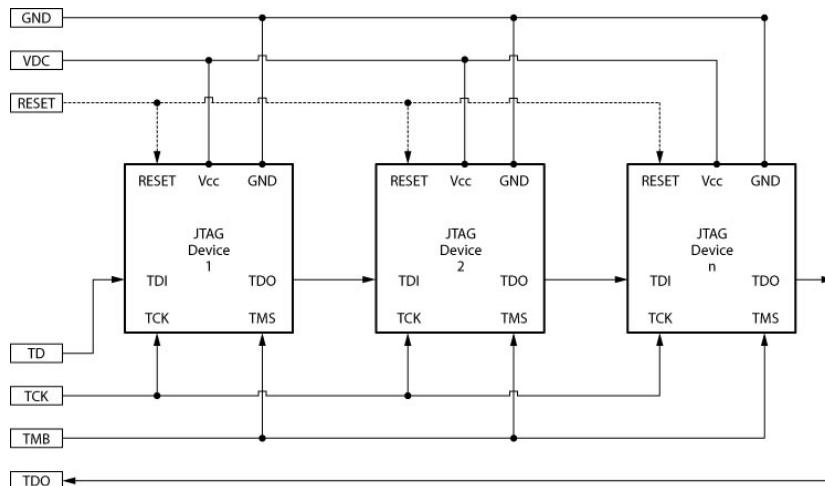
procesora, a także jego portów *IO*. Oprócz możliwości sprawdzenia stanu można ich stan także modyfikować „w locie”. Ułatwia to testowanie poprawności połączeń elektrycznych na płytce i poprawności montażu.

## Programator JTAGICE mkII

Programator JTAGICE mkII jest rozwinięciem układu JTAGICE. Umożliwia on programowanie wszystkich procesorów AVR wyposażonych w interfejs JTAG, w tym także procesorów z rodziny AVR32. Ze względu na cenę tego programatora (przekraczającą 1000 zł) jest to raczej propozycja dla osób chcących bardziej profesjonalnie zająć się programowaniem i budowaniem układów w oparciu o mikrokontrolery. Funkcjonalnie programator ten nie różni się od swojego poprzednika, udostępnia podobne możliwości. Oprócz programowania przez interfejs JTAG udostępnia także możliwość programowania z wykorzystaniem interfejsów PDI, debugWire, SPI oraz aWire. W efekcie za jego pomocą można zaprogramować praktycznie wszystkie procesory AVR.

## Kilka procesorów w jednym układzie

Podobnie jak w przypadku ISP, także JTAG umożliwia wykorzystanie jednego złącza do programowania kilku układów. Funkcja taka jest wpisana w specyfikację protokołu JTAG, więc teoretycznie taka konfiguracja powinna być nawet łatwiejsza w realizacji. Tu, niestety, jak to zwykle bywa, napotykamy na problemy natury programowej. Większość dostępnego oprogramowania nie wspiera możliwości wybierania procesora w konfiguracji łańcuchowej JTAG (ang. *Daisy-chain JTAG mode*). Sytuacja ta stopniowo się zmienia i część oprogramowania dostarczonego przez firmę Atmel wspiera taką konfigurację dla programatora AVRICE mkII. Schemat podłączenia interfejsów JTAG w konfiguracji *daisy-chain* pokazano na rysunku 2.7. Linie *TDI* i *TDO* kolejnych procesorów są połączone szeregowo.



Rysunek 2.7. Połączenie kilku układów AVR, wykorzystujących jedno złącze JTAG

Inną możliwością jest rozwiązywanie analogiczne do pokazanego przy okazji programowania ISP — połączenie równolegle odpowiednich linii JTAG, z wyjątkiem linii *SCK*. Wybór aktywnej linii *SCK* umożliwia wybór programowanego/debugowanego układu.



Wskazówka

Korzystając z możliwości konfiguracji *daisy-chain*, należy mieć na uwadze jeszcze jeden problem — niektóre mikrokontrolery AVR mają błędna implementację obsługi JTAG, uniemożliwiającą zastosowanie konfiguracji *daisy-chain*. Stąd przed jej użyciem należy zawsze sprawdzić erratę do noty katalogowej procesora, zgodną z jego modelem oraz wersją układu.

## AVR Dragon

Alternatywą dla wcześniej wymienionych programatorów, w tym dla drogiego JTAGICE mkII, jest układ AVR Dragon. W przeciwieństwie do wcześniejszych układów jest on sprzedawany bez obudowy, złącz i kabli. Potrzebne złącza należy wlutować samemu. Dzięki temu jego cena jest niezwykle atrakcyjna — można go kupić w cenie ok. 200 – 240 zł. **Niestety, brak wielu wbudowanych zabezpieczeń czyni go niezwykle podatnym na uszkodzenie.** Aby taką możliwość znacznie zmniejszyć, należy samemu dodać odpowiednie układy zabezpieczające — np. układy buforujące wyjścia programatora. Układ AVR Dragon umożliwia programowanie wszystkich mikrokontrolerów AVR dzięki wyposażeniu go w interfejsy HVPP, HVSP, ISP, JTAG, PDI. Umożliwia także debugowanie układu docelowego dzięki interfejsom JTAG i debugWire. Programowane układy mogą być zasilane napięciem z zakresu 1,8 – 5,5 V. Układ AVR Dragon może także dostarczać dla nich napięcia zasilającego o natężeniu maksymalnie 300 mA.

## Programatory HW i równoległe

Programatory wysokonapięciowe (HW, ang. *High Voltage*) oraz równoległe są niezwykle rzadko wykorzystywane. Programator wysokonapięciowy wykorzystuje podobne sygnały co programator ISP, lecz podczas programowania na wejściu *RESET* procesora zamiast stanu niskiego doprowadzane jest napięcie +12V. Dzięki temu można programować procesory, w których przy pomocy *fusebitu RSTDSBL* wejście *RESET* zostało zablokowane. Nie wszystkie procesory dysponują możliwością programowania wysokonapięciowego. W tym trybie programowania procesor wymaga doprowadzenia sygnałów pokazanych w tabeli 2.1.

**Tabela 2.1.** Sygnały wykorzystywane do programowania w trybie wysokonapięciowym

Sygnał	Kierunek	Opis
SDI	Wejście	Wejście danych
SII	Wejście	Wejście instrukcji
SDO	Wyjście	Wyjście danych
SCI	Wejście	Wejście zegarowe

Programatory równolegle wykorzystywane są jeszcze rzadziej. Ich potencjalną zaletą jest większa szybkość działania, lecz do poprawnej pracy wymagają podłączenia kilku-nastu różnych sygnałów. Zaletą tego typu programatorów jest możliwość programowania procesora zablokowanego w wyniku przeprogramowania *fusebitów* odpowiedzialnych za wybór zegara. Jest to możliwe, ponieważ w tym trybie programator generuje przebieg zegarowy taktujący procesor, który jest doprowadzony do wejścia *XTAL1*.

## Tryb TPI

Jest to uproszczony interfejs umożliwiający programowanie najmniejszych procesorów Atmel z serii ATTiny. Używa on linii *RESET* oraz linii danych *TPIDATA* i zegara *TPICLK*. W przypadku kiedy pin *RESET* jest wykorzystywany jako zwykły pin *IO*, wejście w tryb TPI jest wymuszane poprzez podanie na ten pin napięcia +12 V. Protokół ten wspierany jest przez najnowsze programatory, m.in. AVRISP mkII, AVR Dragon.

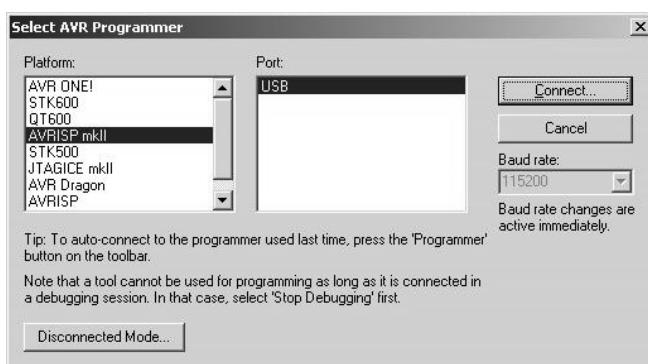
## Programowanie procesora w AVR Studio

Zdecydowanie najłatwiejszą opcją jest wykorzystanie do programowania zintegrowanego środowiska, jakim jest AVR Studio. Dzięki temu mamy możliwość, przy pomocy jednego programu, pisać program, kompilować go, debugować (za pomocą interfejsów sprzętowych lub wbudowanego w AVR Studio symulatora), a efekt finalny przy pomocy jednego przycisku wgrywać do pamięci procesora. Bezpośrednio AVR Studio wspiera narzędzia dostarczane przez firmę Atmel — programatory AVRISP, JTAGICE, Dragon. Po pewnych zabiegach można także korzystać z innych programatorów.

Rozpoczęcie procesu programowania w AVR Studio wymaga najpierw skonfigurowania interfejsu programatora — czyli wybrania z listy programatora, który posiadamy, oraz podania sposobu komunikacji z nim (rysunek 2.8). Opcję konfiguracji wybiera się z menu *Tools/Program AVR/Connect*.

**Rysunek 2.8.**

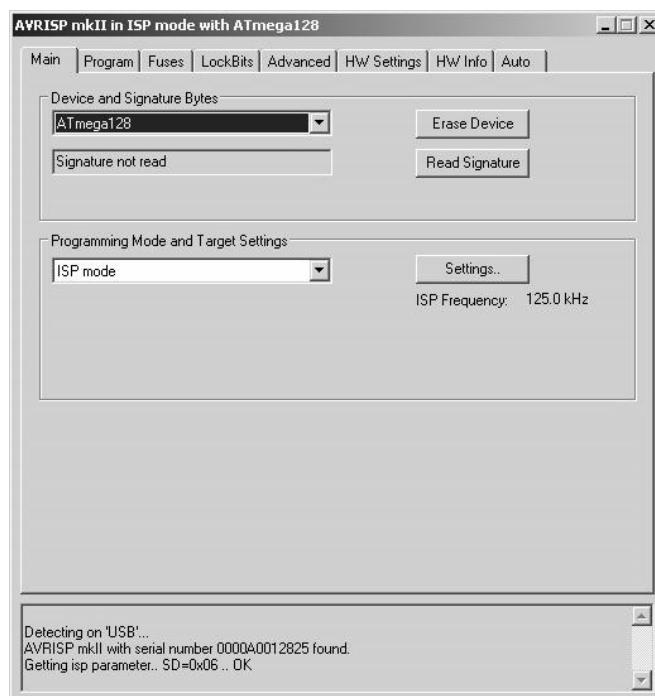
Konfiguracja programatora w AVR Studio — w powyższym przykładzie został wybrany programator AVRISP mkII, podłączony przez port USB



Po udanej próbie nawiązania połączenia z programatorem wyświetcone zostanie kolejne okno, z opcjami, jakie możemy wybrać. Opcje niewspierane przez dany typ programatora nie będą dostępne (rysunek 2.9).

Rysunek 2.9.

Opcje dostępne dla programatora AVRISP MkII. Ponieważ programator ten można łączyć z układem programowanym przy pomocy różnych interfejsów, właściwy tryb połączenia należy wybrać w menu Programming Mode and Target Settings. Dla tego programatora należy także określić częstotliwość zegara taktującego transmisję, pamiętając, że nie może ona być wyższa niż ¼ częstotliwości zegara taktującego rdzenia procesora



W oknie konfiguracji można także ustawić konfigurację fusebitów oraz lockbitów (zakładki *Fuse* oraz *LockBits*) — rysunek 2.10.

W zakładce *Program* określa się ścieżki do plików zawierających program, który chcemy wczytać do mikrokontrolera (pliki muszą być w formacie *elf* lub *Intel HEX*) — rysunek 2.11.

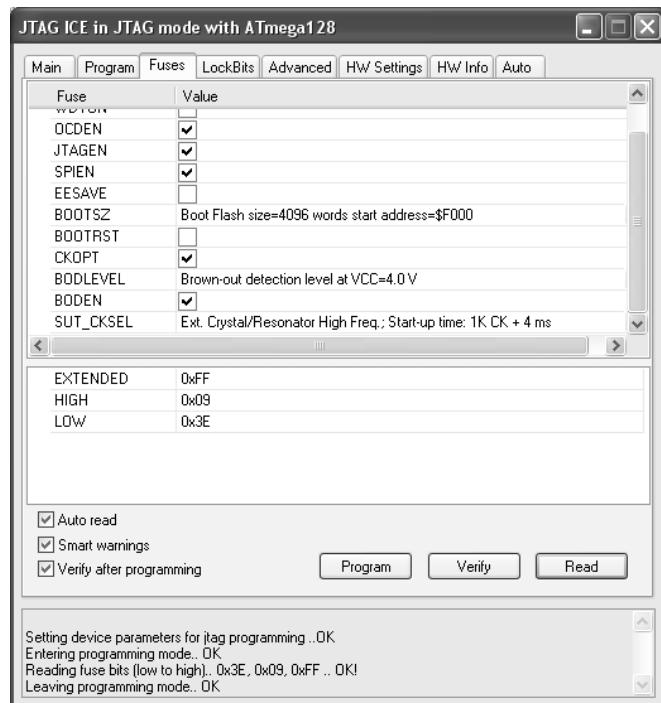
## Programowanie przy pomocy narzędzi dostarczonych przez firmę Atmel

Firma Atmel wraz z AVR Studio dostarcza wielu różnych programów umożliwiających programowanie z linii poleceń przy pomocy programatorów kompatybilnych z protokołami firmy Atmel. Służą one generalnie do automatyzacji procesu programowania w przypadku programowania dłuższych serii procesorów. Wśród licznych programów na szczególną uwagę zasługuje program FLIP. Nie jest on dostarczany razem z AVR Studio, lecz wymaga osobnego pobrania ze strony [www.atmel.com](http://www.atmel.com) i instalacji. Wśród licznych

**Rysunek 2.10.**

Konfiguracja fusebitów w AVR Studio.

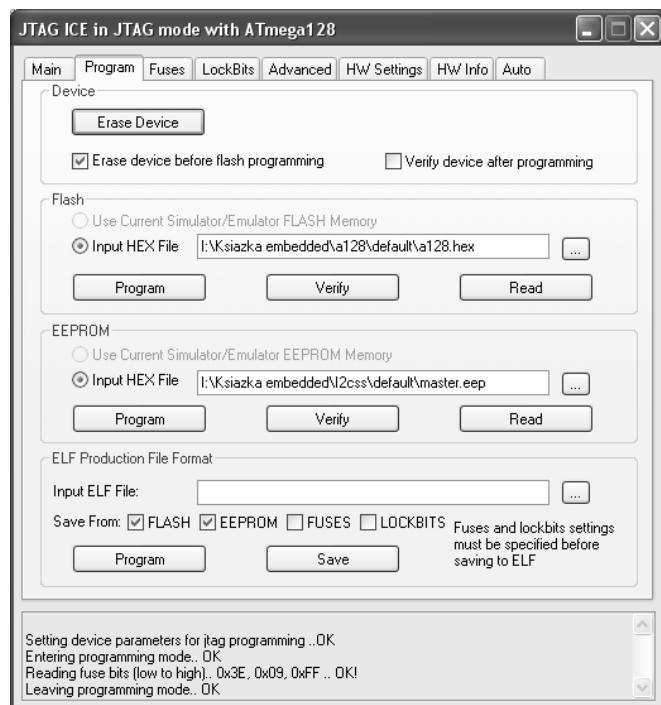
W przypadku złożonych operacji (np. wyboru zegara) zamiast wybierać konfiguracje poszczególnych fusebitów, możemy posłużyć się rozwijalnymi listami z możliwymi do wybrania opcjami. W znacznym stopniu ogranicza to możliwość pomyłki. Wybraną konfigurację fusebitów wprowadza się do procesora po naciśnięciu przycisku Program

**Rysunek 2.11.**

W zakładce Program określa się ścieżki dostępu do plików wykorzystywanych w trakcie programowania.

Najwygodniej jest użyć pliku w formacie elf, gdyż zawiera on wszystkie niezbędne do zaprogramowania procesora dane.

Alternatywnie można podać ścieżki do plików hex i eep, zawierających wsad do zaprogramowania pamięci FLASH i EEPROM



jego możliwości jest także możliwość programowania urządzeń wyposażonych w *bootloader* kompatybilny ze specyfikacją Atmela dla urządzeń klasy DFU (ang. *Device Firmware Update*). Do tej klasy urządzeń zalicza się m.in. procesory AVR wyposażone w sprzętowy interfejs USB. Są one sprzedawane z firmowym wgranym *bootloaderem*, umożliwiającym wczytanie oprogramowania do pamięci FLASH i EEPROM mikrokontrolera.



Tryb DFU nie umożliwia zmiany *fusebitów*. W tym celu należy posłużyć się innym programatorem.

Aby uruchomić wbudowany w urządzenie *bootloader*, podczas wyprowadzenia urządzenia ze stanu *RESET* należy zewrzeć do masy pin *HWB*. Dzięki temu zamiast programu zostanie uruchomiony *bootloader* umożliwiający wczytanie nowego oprogramowania.

Po podłączeniu programowanego układu do komputera przy pomocy USB i uruchomieniu *bootloader* przy pomocy pinu *HWB* urządzenie jest gotowe do programowania.



Aby klasa DFU była rozpoznawana przez komputer, należy zainstalować sterowniki DFU dostarczone przez firmę Atmel.

Po uruchomieniu programu FLIP wybieramy z menu *Device>Select*; w efekcie ukazuje się okno wyboru procesora (rysunek 2.12).

**Rysunek 2.12.**

Okno wyboru procesora. Będzie on programowany przy pomocy bootloadera w trybie DFU



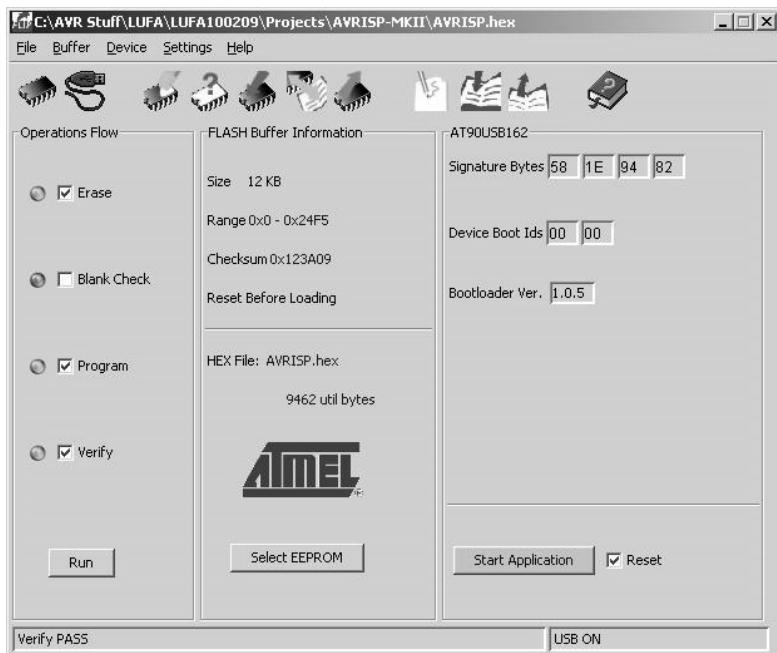
Po wyborze procesora klikamy na *Open*, co powoduje nawiązanie połączenia z programowanym układem. Następnie wczytujemy pliki do zaprogramowania (*File/Load HEX*), wybieramy opcje programowania i weryfikacji układu i klikamy na przycisk *Run*, co inicjuje proces aktualniania oprogramowania (rysunek 2.13).

## Program AVRDUDE

Jest to jeden z najpopularniejszych programów używanych do programowania mikrokontrolerów AVR. Jest on dostarczany wraz z pakietem WinAVR. Sam program AVRDUDE jest aplikacją uruchamianą z wiersza poleceń, parametry podaje się jako opcje wywołania.

**Rysunek 2.13.**

Proces uaktualniania oprogramowania przy pomocy programu FLIP



Aby uczynić go nieco bardziej przyjaznym, w Internecie dostępnych jest wiele graficznych nakładek, umożliwiających uzyskanie tych samych efektów przy pomocy prostego interfejsu graficznego. Program AVRDUDE obsługuje następujące programatory:

- ◆ STK500, STK600,
- ◆ AVRISP i AVRISP mkII,
- ◆ AVRICE i AVRICE mkII,
- ◆ proste programatory podłączane do wyjścia równoległego i szeregowego komputera.

Program ten wspiera wszystkie protokoły transmisji używane przez firmę Atmel. Niestety, do programowania można posłużyć się wyłącznie plikami *IntelHEX*, gdyż nie wspiera on formatu *elf*. W efekcie musimy dysponować oddzielnymi plikami zawierającymi obrazy pamięci FLASH, EEPROM, a także wartościami numerycznymi *fuse*-i *lockbitów*. Stwarza to pewne dodatkowe możliwości pomyłki.

Program ten może pracować w dwóch trybach — terminalowym oraz wywoływany z wiersza poleceń. Poniżej krótko pokazane zostaną podstawowe opcje wywołania, umożliwiające zaprogramowanie przy jego pomocy mikrokontrolera.

Parametry wywołania:

- ◆ `-p procesor` — jest to obowiązkowy parametr wywołania programu. Określa on typ procesora podłączonego do programatora. Listę dostępnych typów można wyświetlić, wydając polecenie `avrdude -p ?`. W efekcie powinna wyświetlić się lista wspieranych typów procesorów.

- ◆ -B *okres* — parametr ten jest używany przy programowaniu za pomocą interfejsu JTAG w trybie ISP. Umożliwia on określenie prędkości programowania poprzez podanie okresu (w mikrosekundach) sygnału *SCK*. Np. avrdude -B 1 powoduje, że linia *SCK* będzie taktowana sygnałem o częstotliwości 1 MHz.
- ◆ -c *programator* — określa typ programatora, który ma zostać użyty do programowania mikrokontrolera. Listę dostępnych typów można wyświetlić polecienniem avrdude -c ?. Na liście tej należy odnaleźć używany programator.



Wskazówka

Czasami dany programator wspiera różne protokoły programowania. W takiej sytuacji będzie występował na liście wiele razy z sufiksami określającymi wybrany tryb programowania.

- ◆ -F — powoduje, że program nie weryfikuje sygnatury układu z typem podanym jako parametr -p. W nielicznych sytuacjach umożliwia to obejście pewnych problemów związanych z uszkodzeniem sygnatury procesora, lecz normalnie opcja ta nie powinna być używana.
- ◆ -n — wykonuje wszystkie operacje, ale bez fizycznego zapisu do układu. Jest to przydatne do testowania różnych skryptów automatyzujących proces programowania.
- ◆ -0 — przeprowadza kalibrację wewnętrznego generatora RC zgodnie z opisem z noty AVR053. Uzyskany w wyniku kalibracji bajt kalibracyjny jest zapisywany do komórki pamięci EEPROM o adresie 0, skąd może zostać odczytany przez program i użyty do kalibracji rejestru OSCCAL mikrokontrolera. Co prawda operacja ta nie poprawia stabilności wewnętrznego generatora RC, ale określa dokładniej jego częstotliwość.
- ◆ -U *obszar:typ:plik[:format]* — opcja ta przeprowadza operację na wskazanym obszarze (może to być operacja odczytu lub zapisu). Parametr *obszar* może być jednym z symboli: eeprom, flash, fuse, hfuse, lfuse, efuse, lock. Określa on obszar podlegający danej operacji, zgodnie z nazwą podanych symboli. Parametr *typ* określa typ operacji: r — odczyt, w — zapis, v — weryfikacja. Parametr *plik* określa nazwę pliku, z którego będą odczytywane dane w przypadku operacji zapisu lub do którego będą zapisywane dane w przypadku operacji odczytu. Ostatni parametr, *format*, określa format pliku. Z licznych formatów istotne są i — określający, że plik jest w formacie IntelHEX, i m — określający, że parametr będzie wartością bezpośrednią, podaną w linii wywołania (najczęściej używane do programowania *fuse-* i *lockbitów*).

Wywołanie:

```
avrdude -p m88 -u -U flash:w:test.hex -U eeprom:w:test.eep -U efuse:w:0xff:m -U  
→hfuse:w:0x89:m -U lfuse:w:0x2e:m
```

spowoduje zaprogramowanie procesora ATmega88 plikami *test.hex* i *test.eep*, których zawartość zostanie umieszczona odpowiednio w pamięci FLASH i EEPROM mikrokontrolera. Dodatkowo wartość *fusebitów* zostanie ustawiona na 0x2E89FF. Z kolejnym wywołaniem:

```
avrdude -c avrisp2 -P usb -p t26 -U flash:w:main.hex:i
```

powoduje zaprogramowanie procesora typu ATTiny26 plikiem *main.hex*, przy pomocy programatora AVRISP mkII, podłączonego przez port USB, w trybie ISP.

## Program PonyProg

Program PonyProg jest bardzo prostym programem umożliwiającym programowanie różnych układów przy pomocy prostych interfejsów podłączanych do portu równoległego lub szeregowego komputera. Obsługuje on pliki *Intel Hex*. Program można pobrać ze strony <http://www.lancos.com/prog.html>. Oprócz programowania procesorów przy jego pomocy można także programować różnego typu pamięci szeregowe. Po skonfigurowaniu typu posiadanej programatora (opcja *Setup/Interface Setup*) należy wybrać typ programowanego układu. Następnie z menu *File* wybieramy *Open Program File* oraz *Open Data File* i wczytujemy uzyskane w trakcie komplikacji pliki z rozszerzeniem *hex* i *eep*. Ostatnią czynnością jest zaprogramowanie procesora poleceniem *Command/Write All*. PonyProg umożliwia także konfigurację *fuse-* i *lockbitów*. W tym względzie jego prostota prowadzi często do błędów. Do dyspozycji mamy tylko pojedyncze *fusebit*, których odpowiednią wartość należy ustalić po przejrzeniu sekcji noty katalogowej procesora poświęconej konfiguracji *fusebitów*.



Wskazówka

Należy pamiętać, że podobnie jak w przypadku innych programów, *fusebit* zaprogramowany oznacza *fusebit* o wartości 0.

Po skonfigurowaniu *fusebitów* wybieramy opcję *Write*, co powoduje ich zapisanie do procesora.

## Fusebit i lockbity w AVR-libc

Biblioteka AVR-libc udostępnia wygodny sposób modyfikacji bitów konfiguracyjnych procesora. Ponieważ bity te nie mogą być zmieniane programowo, aby taka konfiguracja była możliwa, potrzebne jest specjalne oprogramowanie wspierające funkcje biblioteki AVR-libc. Zwykle wspierają taką możliwość programy, które jako źródło danych do programowania procesora wykorzystują pliki w formacie *elf*. W plikach w formacie *Intel HEX* nie ma możliwości umieszczenia informacji o konfiguracji *fuse-* i *lockbitów*; w efekcie programatory wykorzystujące ten format nie wspierają funkcji AVR-libc. W takiej sytuacji pozostaje ręczna konfiguracja tych bitów poprzez wybranie odpowiednich opcji programatora.



Wskazówka

Programując *lock*- i *fusebit*, należy pamiętać, że wartości jeden odpowiada *fusebit* niezaprogramowany, natomiast wartości 0 — zaprogramowany.

Aby *fusebitы* i *lockbitы* zostały poprawnie skonfigurowane i umieszczone w wynikowym pliku *elf*, należy wybrać właściwy typ procesora. Błędne ustawienie typu procesora może spowodować jego zablokowanie na skutek próby wpisania nieprawidłowej konfiguracji *fuse-* i *lockbitów*.

## Lockbitы

*Lockbitы* zostały dokładnie omówione w rozdziale 25. Ich funkcją jest ochrona pamięci mikrokontrolera przed możliwością jej odczytania przy pomocy programatora. Dzięki temu umieszczony w pamięci FLASH program po zaprogramowaniu *lockbitów* nie daje się odczytać. Ich zaprogramowanie nie blokuje możliwości komunikacji z procesorem, przy próbie odczytu zwracane są wartości będące kolejnymi adresami komórek pamięci FLASH. Natomiast w żaden sposób nie da się odczytać ich zawartości, mimo że programator nie zasygnalizuje żadnego błędu. Raz zaprogramowane *lockbitы* można skasować wyłącznie razem z kasowaniem pamięci FLASH i EEPROM poleceniem *Chip Erase*. W ten sposób odzyskujemy możliwość programowania i odczytywania zawartości pamięci FLASH procesora, lecz jednocześnie tracimy zawarty w niej poprzednio program. Zwykle właściwa konfiguracja *lockbitów* określana jest na końcowym etapie tworzenia urządzenia. Nie ma sensu ich używać w trakcie pisania aplikacji.

## Fusebitы

Wszystkie procesory AVR posiadają tzw. *fusebitы*, umożliwiające określenie konfiguracji początkowej procesora po włączeniu zasilania. W zależności od modelu procesora dostępne *fusebitы* mogą się nieznacznie różnić, oferując więcej lub mniej opcji konfiguracyjnych. Poniżej przedstawiona zostanie krótko charakterystyka poszczególnych *fusebitów*. Należy pamiętać, że nowa konfiguracja *fusebitów* zaczyna obowiązywać dopiero po wyjściu z trybu programowania procesora. Dzięki temu jeśli przypadkowo wprowadzono nieprawidłową konfigurację *fusebitów*, to można ją poprawić, o ile procesor nadal znajduje się w trybie programowania.

### Fusebitы BODLEVEL

Są one odpowiedzialne za konfigurację układu odpowiedzialnego za detekcję awarii zasilania. Jeśli napięcie zasilające procesor będzie poniżej progu wyznaczonego wartością *fusebitów BODLEVEL*, procesor utrzymywany będzie w stanie resetu do czasu, aż napięcie wróci do wartości prawidłowych. Domyślnie ich konfiguracja odpowiada zablokowanemu układowi detekcji awarii zasilania. **W gotowym układzie właściwie w każdej sytuacji należy włączyć ten układ ochronny.** Zapobiega to pracy procesora przy napięciach spoza specyfikacji, co może doprowadzić do nieprawidłowej pracy rdzenia procesora i układów peryferyjnych. Często spotykanym problemem przy wyłącznie układzie BOD jest uszkodzenie komórek pamięci EEPROM. Włączenie układu BOD praktycznie eliminuje ten problem. Układ BOD można wyłączać okresowo w systemach, w których pobór mocy jest szczególnie istotny. Jego wyłączenie nieznacznie zmniejsza pobór energii przez procesor. Szczegółowo zostało to omówione w rozdziale poświęconym trybom oszczędzania energii.

## Fusebit WDTON

Jego zaprogramowanie powoduje włączenie układu *watchdoga*. W takiej sytuacji układ ten nie może zostać wyłączony. O konsekwencjach włączenia układu *watchdoga* szerzej napisano w rozdziale 5.

## Fusebit EESAVE

Ma on znaczenie tylko podczas kasowania pamięci procesora przed programowaniem. Jego zaprogramowanie powoduje zachowanie zawartości pamięci EEPROM, zawartość pamięci FLASH nadal będzie mogła być normalnie kasowana. *Fusebit* ten ma zastosowanie w sytuacjach, w których wgrywana jest przy pomocy programatora nowa zawartość pamięci FLASH, a jednocześnie nowy program ma mieć dostęp do danych umieszczonych w pamięci EEPROM przez program poprzedni. Poza sytuacją programowania procesora przez programator, *fusebit* EESAVE nie ma znaczenia.

## Fusebity BOOTSZ i BOOTRST

Ich znaczenie zostało szerzej omówione w rozdziale 25.

## Fusebit JTAGEN

*Fusebit* ten umożliwia wyłączenie interfejsu JTAG. Domyślnie procesory sprzedawane są z zaprogramowanym *fusebitem* JTAGEN (o ile posiadają interfejs JTAG). **Odblokowanie układu JTAG powoduje przejęcie kontroli nad pinami IO wspólnymi z tym interfejsem, co jest częstą przyczyną pomyłek.** Nad takimi pinami nie ma żadnej kontroli ze strony programu. Jeśli interfejs JTAG nie jest używany, można go wyłączyć, dzięki czemu wykorzystywane przez niego piny IO zostaną zwolnione i będą mogły zostać wykorzystane w programie.

## Fusebit SPIEN

*Fusebit* SPIEN odblokowuje interfejs ISP procesora. Domyślnie procesory są sprzedawane z zaprogramowanym *fusebitem* SPIEN, dzięki czemu można je programować przy pomocy programatorów szeregowych. Tego *fusebitu* nie można skasować w trybie programowania szeregowego — można to uczynić np. w trybie programowania poprzez interfejs JTAG lub w trybie wysokonapięciowym.

## Fusebit CKDIV8

Określa on częstotliwość taktowania procesora. Domyślnie procesory sprzedawane są z zaprogramowanym *fusebitem* CKDIV8, w efekcie zegar taktujący jest dzielony przez 8, co prowadzi do częstych pomyłek — np. wyliczone pętle opóźniające są 8-krotnie dłuższe. Jego zaprogramowanie powoduje wpisanie po *resecie* do rejestru preskalera zegara (CLKPR) wartości odpowiadającej podziałowi przez 8. Zamiast kasować ten *fusebit*, można programowo zmienić wartość preskalera.



Wskazówka

Przykładowe programy przedstawione w dalszej części książki zakładają, że *fusebit CKDIV8* ma wartość 1 — ponieważ nie jest to domyślna wartość tego *fusebitu*, należy przed uruchomieniem aplikacji go przeprogramować.

## Fusebity SUT

Określają one liczbę cykli zegara połączenia zasilania, po których procesor będzie wyprowadzony ze stanu resetu. W większości przypadków ich konfiguracja nie ma znaczenia, tym bardziej że ich wartością domyślną jest najdłuższy możliwy czas wyjścia z resetu. W przypadku kiedy zasilanie procesora szybko ulega połączeniu stabilizacji, czas ten można skrócić.

## Fusebit CKOUT

Powoduje on wyprowadzenie na wyjście procesora *CKOUT* zbuforowanego zegara taktującego rdzeń. Dzięki temu inne układy mogą korzystać z zegara procesora, umożliwia to także synchroniczną pracę innych układów z procesorem. Domyślnie ten *fusebit* nie jest zaprogramowany, w efekcie wyjście *CKOUT* zachowuje się jak normalny pin portu *I/O*.

## Fusebity CKSEL

Określają one sposób generowania sygnału zegarowego dla procesora. Prawdopodobnie są to *fusebity* sprawiające najwięcej kłopotów, gdyż ich nieprawidłowe ustawienie może doprowadzić do zablokowania procesora (niemożność dalszego programowania w trybie ISP). Domyślną ich wartością jest wartość wybierająca jako źródło zegara wewnętrzny generator RC. W wielu przypadkach to domyślne ustawienie jest wystarczające i nie ma potrzeby go zmieniać. Potrzeba taka zachodzi, jeśli chcemy taktować procesor przy pomocy zewnętrznego układu generującego zegar lub przy pomocy kwarca. **Konfiguracja tych bitów jest zależna od procesora i przed ich zmianą należy skonsultować się z notą katalogową używanego układu.**

Jeśli zdarzy się nam wybrać niewłaściwe źródło zegara, procesor można „uratować”, doprowadzając zewnętrzny przebieg zegarowy do wejścia *XTAL1*. Dzięki temu możliwe będzie nawiązanie komunikacji z programatorem i ponowne przeprogramowanie *fusebitów*.

Mikrokontrolery AVR mogą być taktowane z trzech źródeł zegara:

1. Zegara zewnętrznego doprowadzonego do wejścia *XTAL1*. Jest to rzadko wykorzystywana możliwość. W tym trybie potrzebny jest zewnętrzny generator zegara, którym może być np. inny procesor AVR. Umożliwia to synchronizację pracy obu kontrolerów.
2. Zegara wewnętrznego (układ generatora RC). W tym trybie (jest on trybem domyślnym) procesor jest taktowany z własnego generatora, w efekcie nie trzeba doprowadzać zewnętrznego przebiegu zegarowego. Wadą generatora wewnętrznego jest jego stosunkowo niewielka stabilność. W efekcie nie może on być wykorzystywany np. w programach, w których wymagana jest duża stabilność zegara.

3. Zewnętrznego rezonatora kwarcowego. W tym celu do pinów *XTAL1* i *XTAL2* należy podłączyć rezonator kwarcowy o pożądanej częstotliwości. Tryb ten umożliwia taktowanie procesora ze stabilnego źródła zegara.

## Fusebit RSTDISBL

Zwykle pin RESET procesora współdzieli wyprowadzenie w pinem IO. Aby móc wykorzystać pin RESET jako normalny pin IO, należy zaprogramować *fusebit RSTDISBL*. Po jego zaprogramowaniu dalsze programowanie procesora w trybie szeregowym jest niemożliwe. Procesor można nadal programować programatorem wysokonapięciowym lub równoległy.

## Fusebit DWEN

Jego zaprogramowanie uruchamia możliwość wykorzystania interfejsu debugWire, przy pomocy którego można programować procesor oraz w ograniczonym stopniu debugować program. Tylko nieliczne procesory dysponują tą funkcją, w dodatku wymaga ona posiadania specjalnego programatora obsługującego debugWire.

## Sygnatura

Wszystkie procesory AVR dysponują unikalną sygnaturą nadawaną im w czasie procesu produkcji. Sygnatura ta określa typ procesora i ilość pamięci. Dzięki temu programator, odczytując sygnaturę, może weryfikować, czy podłączony procesor odpowiada procesorowi wybranemu przez użytkownika. **Sygnatura nie zawiera unikalnego dla konkretnego chipu numeru seryjnego.**

## Lockbity w AVR-libc

Definacje wspieranych przez dany model procesora lockbitów zawiera plik nagłówkowy *<avr/io.h>*. Po jego włączeniu można włączyć plik *<avr/io.h>* zawierający definicję różnych makrodefinicji związanych z konfiguracją lockbitów dla danego procesora. Po włączeniu do programu dostajemy do dyspozycji makrodefinicję **LOCKBITS**, której można przypisać pożadaną kombinację lockbitów. Domyślną wartością tego makra jest 0xFF; w efekcie wszystkie *lockbity* pozostają niezaprogramowane. Jeśli chcemy to zmienić, należy symbolowi **LOCKBITS** przypisać nową wartość, najlepiej posługując się predefiniowanymi w pliku *<avr/io.h>* symbolami. **Poszczególne symbole można łączyć ze sobą przy pomocy iloczynu bitowego.** Przykładowo:

```
LOCKBITS=(BLB1_MODE_3 & LB_MODE_3);
```

spowoduje wybranie trybu 3 dla kodu aplikacji oraz *bootloadera*. Więcej o trybach ochrony pamięci przy pomocy lockbitów znajdziesz w rozdziale 25.

Makrodefinicję `LOCKBITS` można zainicjować tylko raz, przyporządkowanie jej innej wartości w dalszych fragmentach programu nie odnosi żadnego skutku.

Wartość ustawionych *lockbitów* dla skompilowanego programu można odczytać z pliku `elf` przy pomocy polecenia:

```
avr-objdump -s -j .lock <plik ELF>
```

W efekcie dla wartości *lockbitów* określonej powyżej dla procesora ATMega88 powiniśmy uzyskać taki rezultat:

```
test.elf:      file format elf32-avr
```

```
Contents of section .lock:  
830000 cc
```

Wartość *lockbitów* wynosi więc 0xCC.

## Fusebity w AVR-libc

Podobnie jak w przypadku *lockbitów*, *fusebity* określone w programie zostaną umieszczone w specjalnej sekcji pliku `elf`, skąd będą mogły zostać odczytane przez program sterujący programatorem i użyte do konfiguracji procesora. Plik `<avr/fuse.h>` zawiera definicje przydatne przy niskopoziomowym manipulowaniu *fusebitami*. Plik nagłówkowy `<avr/io.h>` zawiera definicje *fusebitów* używanych przez wybrany model procesora. W zależności od typu procesora *fusebity* mieszą się w jednym, dwóch lub trzech bajtach. Ilość bajtów przeznaczonych na *fusebity* zwraca makro `FUSE_MEMORY_SIZE`. Definiowanie *fusebitów* następuje poprzez przypisanie wartości makra `FUSES`, np.:

```
FUSES =  
{  
    .low=LFUSE_DEFAULT,  
    .high=(FUSE_BOOTSZ0 & FUSE_BOOTSZ1 & FUSE_EESAVE & FUSE_SPIEN),  
    .extended=EFUSE_DEFAULT,  
};
```

Jak widzimy, makro to ma trzy pola odpowiadające poszczególnym bajtom przechowującym *fusebity*. Dla procesorów, w których makro `FUSE_MEMORY_SIZE` zwraca wartość jeden, dostępne jest tylko pole `.low`, dla procesorów, w których powyższe makro zwraca 2, dostępne są pola `.low` i `.high`, w pozostałych procesorach dostępne jest także pole `.extended`. Niestety, wykorzystując powyższe definicje, należy pamiętać, w którym bajcie jakie *fusebity* są przechowywane. Ich pomylenie spowoduje nieprawidłowe zaprogramowanie procesora, co może się nawet skończyć jego zablokowaniem. Podobnie jak w przypadku *lockbitów*, ważne jest tylko pierwsze przypisanie wartości do makra, kolejne są ignorowane. Stąd też najlepiej taką definicję umieścić raz, na początku programu.

Podobnie jak w przypadku *lockbitów*, wyliczone wartości *fusebitów* można odczytać z pliku *elf*:

```
avr-objdump -s -j .fuse test.elf  
ADCNoiseReduction.elf:      file format elf32-avr  
  
Contents of section .fuse:  
820000 62d1f9          b..
```

Wyświetlona wartość odpowiada poszczególnym bajtom przechowującym *fusebit*. Dla procesorów, w których mieszą się one w mniejszej ilości bajtów, pokazanych zostanie ich mniej. **Co ważne, najmłodszy bajt wyświetlany jest z lewej, najstarszy z prawej strony.**

## Rozdział 3.

# Podstawy języka C na AVR

Język C dla mikrokontrolerów AVR cechuje się pewnymi odmiennościami związanymi ze specyfiką sprzętu, niewielką ilością pamięci FLASH i SRAM oraz inną organizacją przestrzeni adresowych. W przeciwieństwie do komputerów klasy PC mikrokontrolery AVR dysponują oddzielnymi przestrzeniami adresowymi (dla pamięci FLASH, SRAM i EEPROM), co wymusza stosowanie pewnych rozszerzeń języka. Ze względu na ograniczone zasoby i moc obliczeniową większą wagę należy przykładać do używanych typów danych i przeprowadzanych na nich obliczeń. Celem tego rozdziału nie jest zastąpienie podręcznika nauki programowania w języku C, lecz pokazanie pewnych istotnych różnic pomiędzy implementacją języka C na komputerach klasy PC i na mikrokontrolerach AVR. Niejako przy okazji zostaną omówione w stopniu bardzo podstawowym elementarne słowa kluczowe i składnia języka. Szerzej omówione zostaną aspekty związane z efektywnym programowaniem mikrokontrolerów.

## Arytmetyka

### Proste typy danych

W języku C możemy wykorzystywać tzw. proste typy danych (ang. *Primitive Data Types*) oraz złożone typy danych (ang. *Complex Data Types*). Do typów prostych zaliczamy typy całkowite i zmiennopozycyjne. Poniżej zostaną przedstawione podstawowe informacje o różnych rodzajach typów danych.

#### Typy całkowite

W języku C istnieje szereg typów mogących przechowywać liczby całkowite. W zależności od wielkości przechowywanej liczby możemy wybrać odpowiedni do jej reprezentacji typ. Dzięki temu zmienna zajmuje w pamięci operacyjnej mniej miejsca, a dostęp

do niej i przeprowadzane na niej operacje są szybsze. Ma to szczególne znaczenie w przypadku mikrokontrolerów, które z natury mają ograniczone zasoby.

### Typ bool

Najprostszą wartością, jaką możemy przechować, jest *prawda* (1) lub *falsz* (0). Taką wartość możemy przechować w 1 bicie pamięci. Standard języka C99 definiuje specjalny typ `bool`, jednak na AVR typ ten zajmuje aż 8 bitów — czyli cały bajt. Jest to duże marnotrawstwo pamięci, ale dzięki temu dostęp do zmiennej o typie `bool` odbywa się znacznie szybciej. W tej sytuacji *falsz* jest reprezentowany przez 0, a *prawda* przez wartość różną od 0.



Definicja tego typu znajduje się w pliku nagłówkowym `<stdbool.h>`.

Wskazówka

Jeśli zależy nam na zmniejszeniu zużycia pamięci, a czas dostępu do zmiennej nie jest krytyczny i korzystamy z kilku zmiennych typu `bool`, możemy zadeklarować strukturę umożliwiającą dostęp do poszczególnych bitów bajtu:

```
typedef struct
{
    bool b0 : 1;
    bool b1 : 1;
    bool b2 : 1;
    bool b3 : 1;
    bool b4 : 1;
    bool b5 : 1;
    bool b6 : 1;
    bool b7 : 1;
} flags;
```

Jeśli zadeklarujemy taką strukturę w zakresie adresów I/O dostępnych instrukcjom CBI/SBI, dostęp do jej poszczególnych pól będzie bardzo szybki. Szczegółowo zostanie to pokazane w rozdziale 13.

### Typy znakowe

Do typów znakowych zaliczamy trzy typy: `signed char`, `unsigned char` oraz `char` (bez specyfikowania, czy jest to typ ze znakiem [`signed`] czy bez znaku [`unsigned`]). Implementacja gcc na mikroprocesory AVR zakłada, że typ `char` i pochodne zajmują 8 bitów (1 bajt).



Pamiętaj, że podane długości wszystkich typów dotyczą procesorów AVR. Na innych platformach (np. na komputerach klasy PC) mogą one mieć (i zazwyczaj mają) inne długości. Stąd też przenoszenie danych binarnych pomiędzy platformami jest utrudnione. Problem ten zostanie omówiony szerzej w dalszej części książki.

## Typ int

Język C wyposażony jest w cały szereg różnych typów całkowitych (ang. *Integer*). Różnią się one długością, a więc maksymalnym zakresem liczb, które mogą być przez nie reprezentowane. Podobnie jak typy znakowe, typy całkowite możemy podzielić na typy ze znakiem (ang. *Signed*) i typy bez znaku (ang. *Unsigned*). **Jeżeli jawnie nie określmy, czy typ jest ze znakiem czy bez znaku, kompilator przyjmuje, że typ jest ze znakiem.** Oprócz nazwy typu (`int`) stosujemy prefiksy:

- ◆ `short int` — określający typ „krótki” — 8-bitowy,
- ◆ `int` — normalny typ całkowity — 16-bitowy,
- ◆ `long int` — typ wydłużony — 32-bitowy,
- ◆ `long long int` — typ 64-bitowy.

Szczegółowo typy i zakres reprezentowanych wartości przedstawiono w tabeli 3.1.



Dokładne zakresy, jakie reprezentują poszczególne typy, określone są poprzez stałe zdefiniowane w pliku `limits.h`. Np. zakres liczb mieszczących się w typie `int` określany jest stałymi `INT_MIN` oraz `INT_MAX`.

**Tabela 3.1.** Typy całkowite i ich długość

Nazwa typu	Nazwa skrócona	Długość	Zakres
<code>char</code>	<code>char</code>	8 bitów	-128 – 127
<code>unsigned char</code>	<code>unsigned char</code>	8 bitów	0 – 255
<code>signed short int</code>	<code>short</code>	8 bitów	-128 – 127
<code>unsigned short int</code>	<code>unsigned short</code>	8 bitów	0 – 255
<code>signed int</code>	<code>int</code>	16 bitów	-32 768 – 32 767
<code>unsigned int</code>	<code>unsigned int</code>	16 bitów	0 – 65 535
<code>signed long int</code>	<code>long</code>	32 bity	-2 147 483 648 – 2 147 483 647
<code>unsigned long int</code>	<code>unsigned long</code>	32 bity	0 – 4 294 967 295
<code>signed long long int</code>	<code>long long</code>	64 bity	-9 223 372 036 854 775 808 – -9 223 372 036 854 775 807
<code>unsigned long long int</code>	<code>unsigned long long</code>	64 bity	0 – 18 446 744 073 709 551 615

W przypadku typów ze znakiem najstarszy bit odpowiedzialny jest za przechowywanie znaku liczby. W przypadku liczb ujemnych jego wartość wynosi 1.



Ponieważ zmienna długość typów na różnych platformach stwarza problemy, w pliku nagłówkowym `<stdint.h>` zdefiniowano pochodne typów całkowitych o stałej, niezależnej od platformy długości.

Dzięki temu mamy do dyspozycji typy bez znaku: `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`, i typy ze znakiem: `int8_t`, `int16_t`, `int32_t`, `int64_t`, o długościach odpowiednio 8,

16, 32 i 64 bity. Ich użycie jest zalecane w sytuacjach, w których potrzebna jest znajomość dokładnej długości danego typu — np. przy przenoszeniu danych binarnych pomiędzy platformami.

## Typy wyliczeniowe

Typy wyliczeniowe reprezentują kolejne wartości jakiegoś zbioru. Zazwyczaj ich reprezentacja odpowiada reprezentacji typu `int`. Ponieważ na mikrokontrolerach AVR rzadko spotykamy się z koniecznością tworzenia typów wyliczeniowych o długości 16 bitów, co wiąże się z niepotrzebnym zwiększeniem objętości kodu wynikowego, możemy skrócić reprezentację tego typu do 8 bitów, stosując opcję komplikacji `-fshort-enums`.

Wykorzystanie typów wyliczeniowych ilustruje poniższy przykład:

```
enum dni {Poniedzialek, Wtorek, Sroda, Czwartek, Sobota=6, Niedziela};  
  
prog_char *dzien(enum dni day)  
{  
    switch(day)  
    {  
        case Poniedzialek : return PSTR("Poniedziałek");  
        case Wtorek       : return PSTR("Wtorek");  
        case Sroda        : return PSTR("Środa");  
        case Czwartek    : return PSTR("Czwartek");  
        default          : return PSTR("Piątek/Sobota/Niedziela");  
    }  
}
```

Powyżej zdefiniowano typ wyliczeniowy `dni`, zawierający kolejne elementy zbioru. Są im automatycznie przyporządkowane kolejne liczby, począwszy od zera. W przypadku elementu `Sobota` nadajemy mu jawnie wartość 6, element `Niedziela` będzie miał kolejną dostępną wartość, czyli 7. Zdefiniowany typ wykorzystujemy w funkcji `dzien`, która na podstawie nazwy elementu typu `dni` zwraca wskaźnik do jego tekstowego opisu.

## Operacje niezdefiniowane

W języku C istnieje możliwość poprawnego z punktu widzenia semantyki języka zapisania pewnych operacji, których wynik jednak jest niemożliwy do przewidzenia. A dokładniej, wynik ten jest zależny od użytego kompilatora lub innych „magicznych” opcji, ale nie daje się przewidzieć na podstawie standardów języka. Operacji takich powinniśmy unikać, nawet jeśli w konkretnym przypadku prowadzą one do wygenerowania poprawnego (tzn. działającego zgodnie z intencjami programisty) kodu. Pierwszą tego typu operacją jest:

```
int a, b=1;  
a=b++ + b++;
```

W powyższym przykładzie problem wynika z tego, że operacja dodawania może zostać przeprowadzona przed operacjami inkrementacji zmiennej `b` lub po niej, dając w efekcie różne wyniki. Poprawny zapis wyglądałby następująco:

```
a=b++;  
a+=b++;
```

## Arytmetyka stałopozycyjna

W pewnych sytuacjach operacje na liczbach całkowitych są niewystarczające, nie znaczy to jednak, że od razu musimy sięgać po arytmetykę zmiennopozycyjną. Zwykle dokładnie wiemy, jaką chcemy uzyskać dokładność, wystarczy więc odpowiednio przesunąć przecinek, aby uzyskać dokładność 1-, 2-, 3- itd. miejsc po przecinku.



Wskazówka

Tak naprawdę arytmetyka stałopozycyjna to zwykła arytmetyka liczb całkowitych, jedyna różnica polega na wprowadzeniu wyimaginowanego przecinka.

Wyobraźmy sobie ceny towarów w sklepie. Coś kosztuje np. 20,19 zł. Czy do zapisu musimy użyć zmiennej zmiennopozycyjnej? Zdecydowanie nie. 20 zł i 19 gr to inaczej 2019 groszy, a więc zwykła liczba całkowita. To tylko my zmieniliśmy jej interpretację. Aby więc poprawnie wyświetlić cenę towaru, wystarczy w odpowiednim miejscu umieścić przecinek. Na podobnej zasadzie opiera się właśnie arytmetyka stałopozycyjna. Jeśli potrzebujemy np. reprezentować liczbę z dokładnością do 0,001, to wystarczy przyjąć, że najmniejszą wartością jest liczba 1 odpowiadająca 0,001. W takiej sytuacji zapis 1,002 będzie reprezentowany jako liczba całkowita 1002 — musimy tylko pamiętać, że po pierwszej cyfrze od lewej znajduje się przecinek.

W arytmetyce stałopozycyjnej liczbę dzielimy na dwie części — część całkowitą (to, co jest przed przecinkiem) i część ułamkową (która jest po przecinku). Obie części możemy traktować jako dwie liczby całkowite. Podejście takie ma kilka zalet — przede wszystkim te same procedury, które wykorzystywaliśmy przy obliczeniach na zmiennych całkowitych, możemy wykorzystać do obliczeń stałopozycyjnych (liczba całkowita różni się od liczby stałopozycyjnej tylko tym, że część ułamkowa wynosi 0). Takie podejście powoduje, niestety, ciągłą konieczność korekcji położenia miejsca dziesiętnego. Np. mnożąc  $10,23 * 11,21$ , uzyskamy liczbę 114,6783, posiadającą aż 4 miejsca dziesiętne. Zakładając, że interesują nas wyłącznie 2 miejsca dziesiętne, uzyskany wynik musimy podzielić przez 100. Konieczność takich korekcyj nie jest jednak dużym utrudnieniem. Kolejną zaletą takiego zapisu jest szybkość prowadzonych obliczeń. Jak widzimy, są one tylko nieznacznie wolniejsze niż obliczenia na liczbach całkowitych — zwolnienie wynika z konieczności korekcji po niektórych operacjach położenia miejsca dziesiętnego. Jednak takie obliczenia są ciągle ponad 20 razy szybsze niż podobne na zmiennych zmiennopozycyjnych.

Aby skorzystać z dobrodziejstw arytmetyki stałopozycyjnej, nie musimy nic szczególnego robić — te same typy, które wykorzystywaliśmy dla liczb całkowitych, możemy wykorzystać także do reprezentacji liczb stałopozycyjnych. Np. to, czy zmienna o typie `int` będzie miała wartość 3276 czy 32,76, a może 3,276, nie zmienia jej reprezentacji binarnej. Położenie przecinka możemy sobie ustalić w dowolny sposób, bez wpływu na zapis liczby.

Widzimy więc, że aby skorzystać z dobrodziejstw arytmetyki stałopozycyjnej, najpierw należy określić przedział wartości, jakie przyjmuje zmienna, oraz maksymalną rozdzielczość (dokładność), z jaką chcemy reprezentować wartości. Np. wracając do wcześniejszego przykładu z cenami — założymy, że przedział cen to 0 do 1 000 000 zł, z rozdzielczością 1 grosza, a więc  $0,01$ . Zmienna może więc przyjmować  $10^6 * 10^2$  różnych

wartości, a więc bez problemu zmieści się w typie `unsigned long`. Operacje dodawania i odejmowania liczb stałopozycyjnych nie różnią się od analogicznych operacji prowadzonych na liczbach całkowitych. Jedynie operacje mnożenia i dzielenia wymagają korekcji położenia miejsca dziesiętnego. Możemy go za każdym razem „ręcznie” poprawiać lub zdefiniować własne operacje mnożenia i dzielenia<sup>1</sup>:

```
unsigned long fixmul(unsigned long x, unsigned long y)
{
    return x*y/100;
}

unsigned long fixdiv(unsigned long x, unsigned long y)
{
    return x/y*100;
}
```

Od tej chwili zamiast używać zwykłych operacji mnożenia lub dzielenia, będziemy wykorzystywać powyższe, które automatycznie korygują położenie miejsca dziesiętnego:

```
unsigned long x=fixmul(1001, 2002);
x=fixdiv(x,2002);
```



**Wskazówka** Operacje na typach stałopozycyjnych możemy jeszcze bardziej przyśpieszyć, jeśli zamiast określać rozdzielcość typu w miejscach dziesiętnych, zrobimy to w miejscach binarnych. W takiej sytuacji adjustacja liczby miejsc po przecinku ogranicza się tylko do wykonania paru szybkich operacji przesunięć bitowych.

## Wsparcie dla arytmetyki zmiennopozycyjnej w kompilatorze avr-gcc

Kompilator avr-gcc może wspierać automatycznie operacje na liczbach stałopozycyjnych. Dostępne wtedy stają się nowe typy umożliwiające ich łatwe wykorzystanie. Jednak aby takie wsparcie było możliwe, kompilator musi zostać skompilowany z odblokowaną opcją `--enable-fixed-point`. Domyslnie kompilator avr-gcc dostępny w pakiecie WinAVR ma tę opcję niedostępną, natomiast wersja kompilatora dostarczona przez firmę Atmel ma ją od blokowaną. O tym, czy użyty kompilator wspiera arytmetykę stałopozycyjną, możemy się przekonać, wydając polecenie `avr-gcc -v`. W efekcie uzyskamy wynik zbliżony do poniższego:

```
Using built-in specs.
Target: avr
Configured with: /home/tools/hudson/workspace/avr8-gnu-toolchain/src/gcc/configure --target=avr --host=i686-mingw32 --build=x86_64-pc-linux-gnu --prefix=/home/tools/hudson/workspace/avr8-gnu-toolchain/avr8-gnu-toolchain-win32_x86 --enable-languages=c,c++ --with-dwarf2 --enable-doc --disable-shared --disable-libada --disnable-libssp --disable-nls --with-mpfr=/home/tools/hudson/workspace/avr8-gnu-toolchain/avr8-gnu-toolchain-win32_x86 --with-gmp=/home/tools/hudson/workspace/avr8-gnu-toolchain/avr8-gnu-toolchain-win32_x86 --enable-win32-registry=avrtoolchain --enable-fixed-point --with-pkgversion=AVR_Toolchain_3.0_149 --with-bugurl=http://www.atmel.com
Thread model: single
gcc version 4.4.3 (AVR_Toolchain_3.0_149)
```

<sup>1</sup> Język C++ umożliwia wykonanie tego w bardziej elegancki sposób przy pomocy przeciążania operatorów.

Jeśli na liście znajduje się opcja `--enable-fixed-point`, to znaczy, że używany kompilator wspiera operacje na zmiennych stałopozycyjnych.



Obecne wsparcie dla typów stałopozycyjnych w avr-gcc jest tylko częściowe, ale sytuacja ta może się szybko zmienić.



Deklaracja typów stałopozycyjnych jest wbudowana w kompilator gcc. Dodatkowo w pliku `<stdfix.h>` zdefiniowane są wygodne w użyciu aliasy dla typów wbudowanych.

Typy stałopozycyjne dzielą się na typy przechowujące tylko część ułamkową liczby (typ `_Fract` i pochodne) oraz typy zawierające część całkowitą i ułamkową (typ `_Accum`).

### Pochodne typu `_Fract`

Typ ten przechowuje część ułamkową liczby. Jego zakres obejmuje liczby  $<-1;1>$  dla typu ze znakiem lub  $<0;1>$  dla typu bez znaku (`unsigned _Fract`). Ilość przechowywanych cyfr znaczących zależy od długości typu i jest pokazana w tabeli 3.2.

**Tabela 3.2.** Długości i zakres typów `_Fract`

Typ	Format	Długość	Rozdzielcość
<code>signed short _Fract</code>	<code>S.7</code>	1 bajt	0,0078125 (1/2 <sup>7</sup> )
<code>signed _Fract</code>	<code>S.15</code>	2 bajty	ok. 0,00003 (1/2 <sup>15</sup> )
<code>signed long _Fract</code>	<code>S.31</code>	4 bajty	ok. 4,6E-10 (1/2 <sup>31</sup> )
<code>signed long long _Fract</code>	<code>S.63</code>	8 bajtów	ok. 4E-19 (1/2 <sup>63</sup> )
<code>unsigned short _Fract</code>	<code>.8</code>	1 bajt	0,00390625 (1/2 <sup>8</sup> )
<code>unsigned _Fract</code>	<code>.16</code>	2 bajty	ok. 0,000015 (1/2 <sup>16</sup> )
<code>unsigned long _Fract</code>	<code>.31</code>	4 bajty	ok. 2,3E-10 (1/2 <sup>32</sup> )
<code>unsigned long long _Fract</code>	<code>.64</code>	8 bajtów	ok. 5,4E-20 (1/2 <sup>64</sup> )

### Pochodne typu `_Accum`

Typ ten i jego pochodne reprezentuje liczbę stałopozycyjną składającą się z części całkowitej oraz ułamkowej. Podstawowe parametry poszczególnych podtypów pokazane są w tabeli 3.3. Podobnie jak typ `_Fract`, typ `_Accum` występuje w dwóch odmianach — bez znaku (`unsigned`) i ze znakiem (`signed`). Najstarszy bit typu wykorzystywany jest do przechowywania znaku liczby, stąd też typy ze znakiem mają o połowę mniejszą pojemność w stosunku do typów bez znaku.

**Tabela 3.3.** Długości i zakresy typów \_Accum

Typ	Format	Długość	Rozdzielcość	Zakres
signed short _Accum	S7.8	2 bajty	$1/2^8$	$<-2^7; 2^8 - 1/2^8>$
signed _Accum	S15.16	4 bajty	$1/2^{16}$	$<-2^{15}; 2^{16} - 1/2^{16}>$
signed long _Accum	S31.32	8 bajtów	$1/2^{32}$	$<-2^{31}; 2^{32} - 1/2^{32}>$
signed long long _Accum	S63.64	16 bajtów	$1/2^{64}$	$<-2^{63}; 2^{64} - 1/2^{64}>$
unsigned short _Accum	8.8	2 bajty	$1/2^8$	$<0; 2^8 - 1/2^8>$
unsigned _Accum	16.16	4 bajty	$1/2^{16}$	$<0; 2^{16} - 1/2^{16}>$
unsigned long _Accum	32.32	8 bajtów	$1/2^{32}$	$<0; 2^{32} - 1/2^{32}>$
unsigned long long _Accum	64.64	16 bajtów	$1/2^{64}$	$<0; 2^{64} - 1/2^{64}>$

### Typy z saturacją

Wcześniej omówione typy należały do tzw. typów bez saturacji — znaczy to tyle, że zachowują się one tak jak zwykłe typy — po przekroczeniu zakresu wartość „zawija się”, w efekcie po przepełnieniu uzyskujemy najmniejszą wartość, a po niedopełnieniu największą. Np. w wyniku poniższej operacji:

```
_Accum a=32767;
a+=1;
```

zmienna a zamiast spodziewanej wartości 32768 będzie miała wartość -32768. Dzieje się tak na skutek powstania przepełnienia i w efekcie przeniesienia na kolejny bit, który nie jest reprezentowany przez typ, w rezultacie jest on „obcięty”. Podobny efekt możemy zaobserwować w przypadku typu \_Fract:

```
Fract f = 0.25r;
int i = 4;
f = f * i;
```

Spodziewalibyśmy się, że  $4 * 0.25$  da nam 1. Nic bardziej mylnego. Jako wynik mnożenia otrzymujemy wartość -1 — bierze się to stąd, że liczby 1 nie da się poprawnie przedstawić w typie \_Fract, kolejną wartością jest -1. Takie zachowanie czasami bywa niepożądane, stąd wprowadzono tzw. typy z saturacją. Różnią się one tym, że po przepełnieniu (ang. *Overflow*) wartość zmiennej jest równa maksymalnej wartości dla danego typu, a przy niedopełnieniu (ang. *Underflow*) wartości minimalnej. Typy te tworzymy, dodając do nazwy prefiks \_Sat. Poprzedni przykład po poprawieniu wygląda następująco:

```
Sat _Fract f = 0.25r;
int i = 4;
f = f * i;
```

Tym razem wyliczona wartość zmiennej f równa się 1. Gdybyśmy ją pomnożyli przez więcej niż 4, np. 5, też otrzymalibyśmy jako wynik wartość 1 — jest to maksymalna wartość, jaką przyjmuje zmienna typu \_Fract, niezależnie od użytej precyzji.



Operacje na zmiennych typu \_Sat przebiegają nieznacznie wolniej — potrzebne są dodatkowe instrukcje sprawdzające wystąpienie nadmiaru/niedomiaru.

## Definicje specjalne

Oprócz definicji typów stałopozycyjnych język C udostępnia także definicje symboli określających wartości minimalną i maksymalną dla danego typu, rozdzielcość oraz liczbę bitów przeznaczonych do reprezentacji części całkowitej i ułamkowej liczby:

- ◆ `_FRACT_FBIT_` — zwraca liczbę bitów wykorzystanych do reprezentacji części ułamkowej typu.
- ◆ `_FRACT_MIN_` — zwraca najmniejszą możliwą wartość przyjmowaną przez zmienną danego typu (w tym wypadku typu `_Fract`).
- ◆ `_FRACT_MAX_` — zwraca największą możliwą wartość przyjmowaną przez zmienną danego typu (w tym wypadku typu `_Fract`).
- ◆ `_FRACT_EPSILON_` — zwraca rozdzielcość danego typu, czyli minimalną wartość, o jaką mogą się różnić zmienne o danym typie.
- ◆ Dla typu `_Accum` i pochodnych dodatkowo zdefiniowany jest symbol `_ACCUM_IBIT_`, określający liczbę bitów w podanym typie wykorzystanych do reprezentacji jego części całkowitej. Symbol ten jest niezdefiniowany dla typu `_Fract` — liczba bitów reprezentujących jego część całkowitą zawsze wynosi 0.

Do powyższych symboli można dodać prefiks precyzujący typ:

- ◆ S oznacza typ `short`.
- ◆ U oznacza typ bez znaku (ang. *Unsigned*).
- ◆ L oznacza typ `long`.
- ◆ LL oznacza typ `long long`.

Np. `_LLACCUM_MIN_` zwraca minimalną wartość zmiennej o typie `long long _Accum`, a `_USFRAC_FBIT_` zwraca liczbę bitów użytych do reprezentacji części ułamkowej typu `unsigned short _Fract`.

## Zapis liczb stałopozycyjnych

Podobnie jak w przypadku liczb całkowitych dla poinformowania kompilatora, jaki zapis stosujemy, do liczby dodajemy odpowiedni sufiks (tabela 3.4). Litera h oznacza typ `_Fract`, litera k typ `_Accum`, odpowiednie wielkie litery oznaczają typy z saturacją. Pozostałe znaczą dokładnie to samo co w zapisie liczb całkowitych.

Zgodnie z powyższą tabelą zapis 0.25r oznacza liczbę o typie `_Fract`, bez saturacji; zapis 1000.23ULK oznacza liczbę o typie `unsigned long _Accum` z saturacją.

## Konwersja z i do typów łańcuchowych

Pomimo że kompilator `gcc` może wspierać arytmetykę na liczbach stałopozycyjnych, to robi to tylko w ograniczony sposób. W chwili pisania tej książki nie były dostępne m.in. funkcje zapewniające konwersję z typu stałopozycyjnego na łańcuch znakowy

**Tabela 3.4.** Sufiksy określające typ liczby stałopozycyjnej

Sufiks	Typ	Sufiks	Typ
hr	short _Fract	HR	_Sat short _Fract
r	_Fract	R	_Sat _Fract
lr	long _Fract	LR	_Sat long _Fract
llr	long long _Fract	LLR	_Sat long long _Fract
uhr	unsigned short _Fract	UHR	_Sat unsigned short _Fract
ur	unsigned _Fract	UR	_Sat unsigned _Fract
ulr	unsigned long _Fract	ULR	_Sat unsigned long _Fract
ullr	unsigned long long _Fract	ULLR	_Sat unsigned long long _Fract
hk	short _Accum	HK	_Sat short _Accum
k	_Accum	K	_Sat _Accum
lk	long _Accum	LK	_Sat long _Accum
llk	long long _Accum	LLK	_Sat long long _Accum
uhk	unsigned short _Accum	UHK	_Sat unsigned short _Accum
uk	unsigned _Accum	UK	_Sat unsigned _Accum
ulk	unsigned long _Accum	ULK	_Sat unsigned long _Accum
ullk	unsigned long long _Accum	ULLK	_Sat unsigned long long _Accum

i odwrotnie. Nie stanowi to jednak istotnej przeszkody — możemy je łatwo zastąpić własnymi funkcjami. Np. konwersji liczby typu stałopozycyjnego na odpowiadający jej łańcuch znakowy możemy dokonać za pomocą prostej funkcji:

```
void AccumToStr(_Accum z, char *Bufor)
{
    itoa(z.Bufor,10); //Dokonaj konwersji części całkowitej liczby
    strcat(Bufor,".");
    z=z-(int)z; //Usuń część całkowitą, zostaw resztę
    itoa(z*10000,&Bufor[strlen(Bufor)], 10); //Dokonaj konwersji reszty z dokładnością
                                                //do 5 miejsc dziesiętnych
}
```

Powyższa funkcja konwertuje zmienną o typie \_Accum na odpowiadający jej łańcuch tekstowy, który znajdzie się w miejscu wskazywanym przez zmienną Bufor. Jak zwykle, Bufor musi mieć odpowiednią wielkość, aby pomieścić powstały łańcuch. Powyższa funkcja ma pewne ograniczenia — dokonuje konwersji tylko 5 miejsc dziesiętnych (stała 10000); jeśli potrzebujemy więcej, to należy zamiast typu int zastosować typ bardziej pojemy, np. long, i zmienić mnożnik.

Podobnie łatwo możemy przeprowadzić konwersję odwrotną:

```
int pow10i(unsigned short n) //Oblicz 10^n
{
    if(n>0) return 10*pow10i(n-1);
    else return 1;
}

_Accum StrToAccum(char *Bufor)
```

```
{  
    _Accum a=atoi(Bufor); //Konwersja części całkowej  
    Bufor=strchr(Bufor, '.'); //Czy jest coś po przecinku?  
    if(Bufor)  
    {  
        Bufor++;  
        int tmp=atoi(Bufor); //Konwersja części ułamkowej  
        _Accum r=(_Accum)tmp/pow10i(strlen(Bufor)); //Dopisanie do wyniku  
        if(a<0) a-=r; else a+=r; //W celu poprawnej konwersji liczb ujemnych  
    }  
    return a;  
}
```

Najpierw zdefiniowaliśmy pomocniczą funkcję pow10i, której zadaniem jest obliczenie wartości  $10^n$ . Ponieważ obliczane wartości są krótkie, wykorzystano rekurencję. Można wykorzystać standardowe funkcje pow i pow10, których prototypy znajdują się w <math.h>, lecz dla nich argumentem są zmienne typu double — co wymaga dołączenia biblioteki realizującej obliczenia zmiennopozycyjne, a tego właśnie chcieliśmy uniknąć.

Właściwa funkcja konwersji to StrToAccum. Dokonuje ona konwersji łańcucha znakowego dwuetapowo — najpierw konwertuje część całkowitą, po czym konwertuje część ułamkową — w tym celu potrzebna jest właśnie wcześniej zdefiniowana funkcja pow10i.

## Arytmetyka zmiennopozycyjna

**Na wstępie zapamiętaj być może szokującą przestrogę — arytmetyka zmiennopozycyjna to зло.** W świecie mikrokontrolerów to зло jest nawet jeszcze gorsze. I to z kilku powodów — po pierwsze, obliczenia zmiennopozycyjne są wolne. Na mikrokontrolerach AVR są one ekstremalnie wolne w porównaniu z obliczeniami na liczbach całkowitych, gdyż mikrokontrolery te nie mają koprocesora arytmetycznego. Drugi powód łatwo zauważyc, komplikując dowolny program zawierający zmienne zmiennopozycyjne — objętość wynikowego kodu od razu rośnie o kilka kB (w przypadku małych procesorów typu ATMega88 powoduje to od razu zajęcie 50% dostępnej pamięci FLASH). Jeśli te dwa powody nie zdolały Cię zniechęcić, to jest jeszcze trzeci — liczby zmiennopozycyjne są reprezentowane ze skońzoną precyzją — w efekcie stosowanie dla nich operatorów relacji, a szczególnie operatora równości (==), nie ma większego sensu.



Tak naprawdę w 99% przypadków wcale liczb zmiennopozycyjnych nie potrzebujesz.

Wskaźówka

Jeśli jednak z jakichś powodów zdecydujesz się użyć typów zmiennopozycyjnych, to koniecznie przeczytaj ten podrozdział.



W przypadku avr-gcc istnieje tylko jeden typ zmiennopozycyjny — float. Typ o pojedynczej precyzyji (double) istnieje, ale funkcjonalnie odpowiada typowi float.

Tak więc zmienne typu float i double są sobie równe i obie zajmują 32 bity (4 bajty). Ponieważ oba typy są wspierane przez sam kompilator, wydaje się, że ich użycie nie wymaga żadnych specjalnych zabiegów:

```
float mul(float x, float y)
{
    return x*y;
}

int main()
{
    float z=mul(2, 10);
}
```

Okazuje się, że tak prosty program zajmuje aż 1346 bajtów<sup>2</sup>. Szybki rzut oka na wygenerowany plik *lss* ujawnia przyczynę — do programu zostały dołączone procedury obsługi liczb zmiennopozycyjnych. Dodanie kolejnej funkcji:

```
float div(float x, float y)
{
    return x/y;
}
```

zwiększa rozmiar programu do 1688 bajtów. Program zajął już sporo pamięci, a wykryliśmy dopiero proste operacje mnożenia i dzielenia, a do dyspozycji mamy jeszcze wiele innych operacji zmiennopozycyjnych. Jeśli w takim tempie będzie nam ubywać pamięci, to prawdopodobnie nie napiszemy naszego programu. Niestety, aby operacje na typach zmiennopozycyjnych były możliwe, muszą zostać dołączone odpowiednie funkcje. Na szczęście, możemy nieco zminimalizować ilość zajętej przez nie pamięci. Użycie operacji na typach zmiennopozycyjnych wymusza na kompilatorze dołączenie kodu, który je realizuje — mikrokontrolery AVR nie posiadają koprocesora arytmetycznego, operacje te muszą więc być symulowane programowo. Jednak kompilator *gcc* nie powstał z myślą o 8-bitowych procesorach — stąd też kod tych procedur nie jest optymalizowany dla mikrokontrolerów AVR i jak się przekonaliśmy, zajmuje sporo miejsca. Na szczęście, twórcy *avr-gcc* przyszli nam z pomocą, pisząc specjalne wersje procedur realizujących obliczenia zmiennopozycyjne dostosowanych do mikrokontrolerów AVR. Aby z nich skorzystać, musimy poinformować linker, żeby zamiast oryginalnych procedur wbudowanych w *gcc* wykorzystał procedury z zewnętrznej biblioteki.



Dostosowane do mikrokontrolerów AVR procedury realizujące obliczenia zmiennopozycyjne znajdują się w bibliotece *libm.a*.

Bibliotekę *libm.a* dołączamy, umieszczając w skrypcie linkera polecenie:

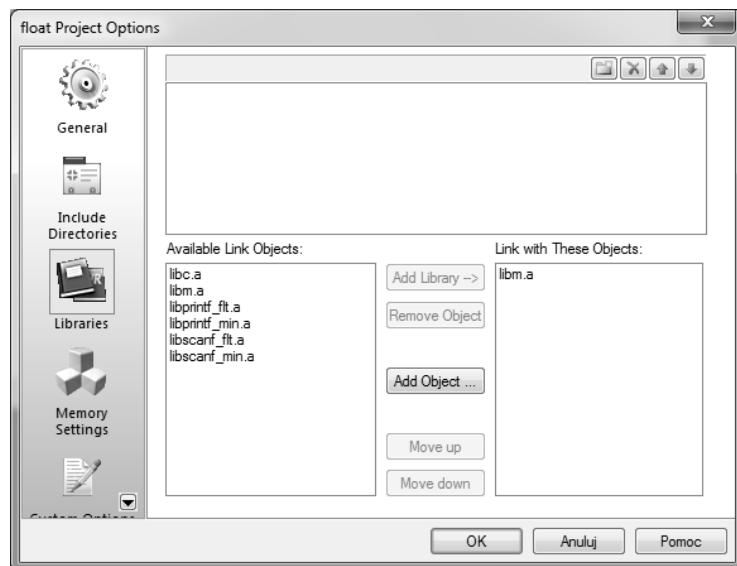
`LIBS = -lm`

lub w AVR Studio wybierając ją w opcjach projektu (rysunek 3.1).

Po dołączeniu *libm.a* okazuje się, że program uległ skróceniu z 1688 bajtów do 656. Oczywiście, przekłada się to nie tylko na wielkość kodu, ale także na czas obliczeń.

<sup>2</sup> Dokładne rezultaty mogą się nieznacznie różnić w zależności od wersji kompilatora i opcji komplikacji.

**Rysunek 3.1.**  
Dodawanie  
zewnętrznych  
bibliotek,  
które zostaną  
zlinkowane  
z tworzonym  
programem



Wykorzystanie bardziej zaawansowanych operacji na zmiennych typu float (m.in. funkcji trygonometrycznych, pierwiastków, potęg oraz stałych matematycznych) wymaga dołączenia pliku nagłówkowego `<math.h>`. Zadeklarowane są w nim prototypy powyższych funkcji.

## Operatory relacji

Jak na wstępie wspomniano, w stosunku do typu float powinno się unikać stosowania operatorów relacji, a szczególnie operatora równości (`==`) i operatora nierówności (`!=`). Poniższy kod uświadomi nam problemy związane z tymi operatorami:

```
float y=0;
do
{
    y+=0.1;
} while(y!=10);
```

Powyższa pętla powinna zostać wykonana 100 razy, po czym kiedy zmienna y będzie równa 10, pętla powinna się zakończyć. Kiedy uruchomimy powyższy kod np. w symulatorze AVR Studio, okaże się, że pętla nigdy się nie kończy! Podglądając zawartość zmiennej y, zauważymy, że przybiera ona „dziwne” wartości, wcale nie będące wielokrotnością 0,1. W efekcie zamiast osiągnąć wartość 10, kończącą pętlę, przyjmuje wartość 10,000002. Jest to spowodowane skońzoną reprezentacją liczb zmiennopozycyjnych w pamięci komputera. Powoduje to, że w większości przypadków niemożliwe jest przechowanie dokładnej wartości zmiennej i w efekcie operator równości traci sens.

 **Wskazówka**

Porównując cokolwiek ze zmienną typu float (lub innym typem zmiennopozycyjnym), musimy uwzględnić możliwy błąd reprezentacji — wartość tego błędu oznaczamy symbolem e).

Tak więc zamiast porównania:

```
if(y==10) ...
```

powinniśmy zastosować:

```
if(((y-e)<10) && ((y+e)>10)) ... //y==10
```

gdzie e to stała określająca, jak bardzo y może się różnić od 10, aby obie wartości uznać za równe.

Ten sam problem powstaje przy stosowaniu operatorów większy (**>**) i mniejszy (**<**) — w przypadku gdy dwie porównywane wartości są do siebie zbliżone. Prześledźmy nieco zmodyfikowaną pętlę z poprzedniego przykładu:

```
volatile float y=0;
do
{
    y+=0.1;
} while(y<3.0);
```

Powyższa pętla powinna się wykonać 30 razy, niestety, wykona się 31 razy. Zmienna y po 30. iteracji zamiast przyjąć wartość 3, kończącą pętlę, przyjmie wartość 2,9999993, która to liczba jest mniejsza niż 3, i wykonana zostanie dodatkowa iteracja. Poprawiona pętla wygląda następująco:

```
const float e=0.000001;
volatile float y=0;
do
{
    y+=0.1;
} while((y+e)<3);
```

W powyższej pętli porównanie następuje z uwzględnieniem błędu, którego maksymalna wartość definiowana jest stałą e.

## Konwersja typu float do łańcucha znakowego

W sytuacji kiedy chcemy wyświetlić zawartość zmiennej na wyświetlaczu, zachodzi konieczność konwersji typu **float** na łańcuch znaków. Możemy tego dokonać na kilka sposobów.



Prototypy funkcji **dtostre** oraz **dtosstrf** znajdują się w pliku nagłówkowym **<stdlib.h>**.

### Funkcja **dtostre**

Funkcja ta konwertuje podaną liczbę typu **double** (czyli **float**, pamiętamy, że na AVR oba typy są identyczne) na łańcuch znakowy w formacie ***-/-zzze±zz***. Prototyp funkcji **dtostre** wygląda następująco:

```
char* dtostre (double __val, char *__s, unsigned char __prec, unsigned char __flags);
```

Pierwszym argumentem funkcji jest konwertowana liczba, drugim wskaźnik do bufora, w którym zostanie umieszczony wynik konwersji, następny określa liczbę cyfr po przecinku, która zostanie przekonwertowana, a ostatni określa parametry konwersji. Funkcja zwraca adres bufora, w którym znajduje się wynik konwersji (jest on równy adresowi podanemu jako parametr `_s`).



Wskazówka

W tej i pozostałych funkcjach niezwykle istotne jest, aby bufor zawierający wynik konwersji miał wystarczającą wielkość. Jeśli będzie za mały, nadpisane zostaną dane leżące za buforem, co doprowadzi do nieprzewidywalnych efektów.

Argument `_flags` może przyjmować następujące wartości:

- ◆ `DTOSTRE_UPPERCASE` — zamiast litery *e* jako symbolu wykładnika użyta zostanie litera *E*.
- ◆ `DTOSTRE_ALWAYS_SIGN` — w przypadku liczb dodatnich przed liczbą zostanie umieszczona spacja — jest to przydatne do utrzymania odpowiedniego wyrównania łańcucha.
- ◆ `DTOSTRE_PLUS_SIGN` — powoduje, że w przypadku liczb dodatnich zostanie dodany znak +.

W wyniku działania poniższego kodu:

```
char bufor[20];
float z=12.345678;
dtostre(z, bufor, 2, 0);
```

zmienna bufor będzie zawierała łańcuch '1.23e+01'.

### Funkcja `dtostrf`

Funkcja ta jest podobna do wcześniej omówionej, z tym że zamiast zapisu naukowego stosuje się zapis w formacie `[-]x.y`. Prototyp funkcji wygląda następująco:

```
char* dtostrf (double __val, signed char __width, unsigned char __prec, char *__s);
```

Tak jak poprzednio `__val` zawiera konwertowaną liczbę, a `__s` wskaźnik do bufora, w którym znajdzie się wynik. `__prec` określa liczbę miejsc dziesiętnych po przecinku. Poniższy kod:

```
char bufor[20];
float z=12.345678;
dtostrf(z, 0, 2, bufor);
```

spowoduje, że zmienna bufor będzie zawierać '12.35'.

### Funkcja `atof`

W pliku nagłówkowym `<stdlib.h>` znajduje się prototyp funkcji `atof`, konwertującej łańcuch znakowy na wartość typu `double` — `double atof (const char *nptr)`, gdzie `nptr` to wskaźnik do łańcucha znakowego. Jej wywołanie jest równoważne `strtod(nptr, (char **)0)`.

## Funkcja strtod

W <stdlib.h> znajduje się jeszcze jeden prototyp funkcji konwertującej — double strtod(const char \*nptr, char \*\*endptr). Funkcja ta różni się od poprzedniej tylko tym, że jako parametr przyjmuje dodatkowy parametr endptr — jeśli jest on różny od NULL, to zapisany w nim zostanie wskaźnik do znaku leżącego tuż po ostatnim znaku konwertowanej liczby.

## Funkcje umożliwiające manipulację sformatowanymi łańcuchami

W przeciwieństwie do wcześniej omówionych funkcji konwertujących zmienne typu float/double na łańcuchy tekstowe i odwrotnie poniższe funkcje umożliwiają bardziej złożone manipulacje. Dzięki nim można przeprowadzać praktycznie dowolne operacje konwersji, jednak cena, jaką płacimy za tę uniwersalność, jest spora — zajmują one o wiele więcej miejsca w pamięci FLASH mikrokontrolera, ich wykonanie jest też znacznie wolniejsze.



Prototypy tych funkcji są zadeklarowane w pliku nagłówkowym <stdio.h>.

## Funkcja printf

Funkcja ta, a raczej jej różne pochodne (sprintf, fprintf itd.) umożliwiają dosyć uniwersalne formatowanie łańcucha znaków, przy okazji zapewniając konwersję liczb typu float na odpowiadający jej łańcuch znakowy. Ze względu na swoją uniwersalność (funkcja umożliwia konwersję nie tylko z typu float, ale z wielu innych) jest ona stosunkowo wolna, a jej implementacja zajmuje dużo miejsca w pamięci. Niemniej warto z niej korzystać, jeśli dokonujemy konwersji z różnych typów lub mamy dosyć szczególne oczekiwania co do łańcucha wyjściowego.

Jej działanie najłatwiej będzie prześledzić na przykładzie:

```
char bufor[20];
double z=12.345678;
sprintf(bufor, "%+12.3e", z);
```

W efekcie zmienna bufor będzie zawierać '<spacja><spacja>+1.235e+01'. Poszczególne znaki łańcucha formatującego oznaczają:

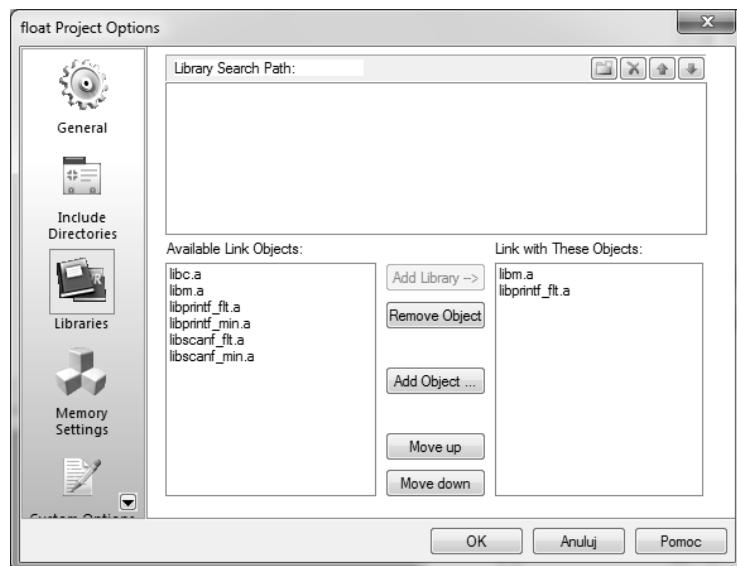
- ◆ Znak + nakazuje umieścić w wynikowym łańcuchu znak liczby (domyślnie umieszczany jest tylko znak -).
- ◆ 12 mówi, że łańcuch wynikowy musi mieć co najmniej 12 znaków. Jeśli będzie miał mniej, z lewej dodane zostaną znaki <spacja>.
- ◆ 3 po kropce oznacza, że wynik będzie zawierał 3 miejsca dziesiętne po przecinku.
- ◆ Litera e oznacza, że konwersja dotyczy typu double.

Jest tylko jeden problem — jeśli np. w symulatorze uruchomisz powyższy fragment kodu, to okaże się, że on nie działa! Zmienna bufor będzie zawierała przypadkowe wartości.

Jest tak, ponieważ wersja sprintf z biblioteki standardowej nie wspiera formatów zmiennopozycyjnych. Dołączając w AVR Studio bibliotekę matematyczną, być może zauważysz w oknie opcji projektu inne biblioteki, m.in. *libprintf\_flt* i *libprintf\_min*. Zawierają one wersję sprintf obsługującą liczby zmiennopozycyjne. Do projektu wystarczy dodać jedną z nich, np. *libprintf\_flt*. Biblioteka *libprintf\_min* zawiera tylko pewien ograniczony podzbiór funkcji — m.in. nie zawiera kodu odpowiedzialnego za współpracę z liczbami zmiennopozycyjnymi. Dołączyć je możemy analogicznie jak *libm* w opcjach projektu (rysunek 3.2).

**Rysunek 3.2.**

Dolaczanie bibliotek odpowiedzialnych za poprawne działanie funkcji printf



Po rekompilacji programu i próbie w symulatorze niestety ponownie okazuje się, że *sprintf* nie działa. Tym razem jednak źródło problemu jest nieco bardziej ukryte. Ale zacznijmy od rozwiązania — do opcji linkowania programu należy dodać *-Wl,-u,vfprintf* (rysunek 3.3).

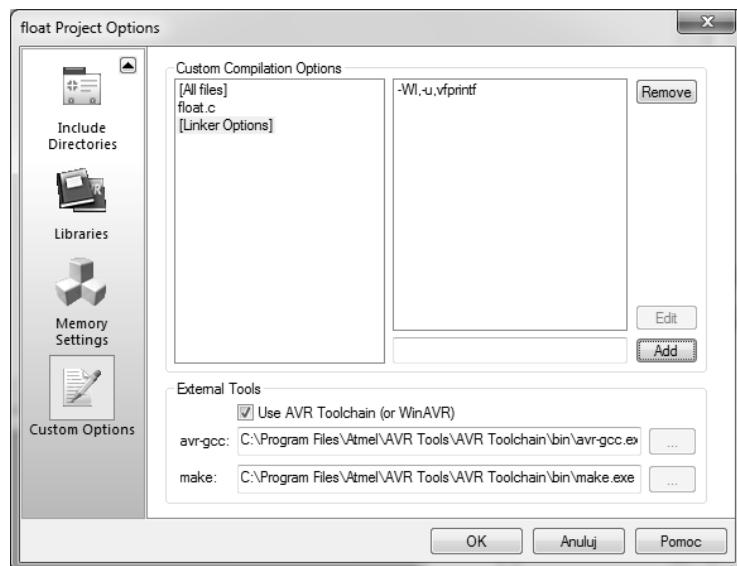
Przyczyna problemów leżała w sposobie, w jaki działa linker. Po natrafieniu na nieznany symbol (*sprintf*) linker pomija bibliotekę *libprintf\_flt*, gdyż nie znajduje w niej definicji tego symbolu. Definicja tego symbolu znajduje się w bibliotece *libc*, tam też z kolei znajduje się odwołanie do symbolu wykorzystywanego przez *sprintf* — *vfprintf*. Symbol ten jest zdefiniowany w *libc*, jednak wskazuje na standardową implementację, która przecież nie obsługuje formatów zmiennopozycyjnych. W efekcie dołączenie odpowiedniej biblioteki niczego nie zmieniło. Opcja linkera *-u vfprintf* powoduje, że linker po natrafieniu na ten symbol rozpoczyna przeszukiwanie bibliotek od początku — zanim dojdzie do *libc*, natrafi na jego definicję w bibliotece *libprintf\_flt*, w efekcie użyta zostanie prawidłowa wersja *vfprintf*.

Jeśli nie korzystamy z graficznego kreatora plików *Makefile* z AVR Studio, musimy zadbać, aby w skrypcie *Makefile* znalazły się linie:

```
LDFLAGS += -Wl,-u,vfprintf  
LIBS = -lm -lprintf_flt
```

**Rysunek 3.3.**

Dodanie opcji linkera odpowiedzialnych za prawidłowe linkowanie funkcji printf



Zapewne niezbyt często będzie zachodziła konieczność zmiany łańcucha formatującego. W takiej sytuacji możemy skorzystać ze zmodyfikowanej funkcji `sprintf_P`. Jej łańcuch formatujący zamiast w pamięci SRAM mikrokontrolera znajduje się w pamięci FLASH. Jej prototyp jest podobny do `sprintf`:

```
int sprintf_P(char * __s, const char * __fmt, ...);
```

Łańcuch formatujący `__fmt` musi znajdować się w pamięci FLASH, umieścić go możemy tam za pomocą makra `PSTR` (jego definicja znajduje się w pliku `<avr\pgmspace.h>`), np. `sprintf_P(bufor, PSTR("%12.4e"), z)`.

Pierwszym zauważalnym efektem zastosowania tej wersji funkcji jest zmniejszenie zużycia pamięci SRAM. To zaskakujące zjawisko zostanie wyjaśnione w rozdziale 8.

## Funkcja `sscanf`

Podobnie jak `sprintf` umożliwia tworzenie sformatowanych łańcuchów znakowych, tak `sscanf` umożliwia konwersję sformatowanego ciągu znaków np. na liczby. Aby skorzystać z oferowanej przez nią możliwości konwersji łańcucha do zmiennej typu `float`, należy dołączyć podobnie jak w przypadku funkcji `sprintf` odpowiednią bibliotekę — `libscanf_flt.a` — oraz przekazać linkerowi polecenie `-Wl,-u,vfscanf`. Możemy to zrobić w AVR Studio lub umieścić w skrypcie *Makefile* polecenia:

```
LDFLAGS += -Wl,-u,vfscanf  
LIBS = -lscanf_flt
```

Po tych zabiegach możemy wykorzystać nową funkcję:

```
float y;  
char buf[] = "-12.456";  
sscanf(buf, "%e", &y);
```

Zmienna `y` będzie miała wartość `-12,456`. Zauważmy, że funkcji `sscanf` przekazujemy adres zmiennej `y`. Analogicznie do funkcji `sprintf_P` istnieje także funkcja `sscanf_P`, dla której ciąg formatujący znajduje się w pamięci FLASH mikrokontrolera.

Oprócz wspomnianych funkcji `sprintf` i `sscanf` istnieją funkcje pokrewne `fprintf`, `fscanf` operujące na plikach, `printf` wysyłająca dane do standardowego strumienia wyjściowego (`stdout`), `scanf` odczytująca dane ze standardowego strumienia wejściowego (`stdin`).

Warto pamiętać o funkcji `snprintf`, o prototypie `int snprintf(char * __s, size_t __n, const char * __fmt, ...)`, która zapisuje do buforu maksymalnie `__n` znaków, co uniemożliwia przepelenienie bufora i w efekcie błędne działanie aplikacji.

## Operacje bitowe

Operacje bitowe odgrywają kluczową rolę w trakcie programowania mikrokontrolerów. Stąd też ich dobre opanowanie jest jednym z czynników warunkujących sukcesy przy pisaniu programów.

Język C dysponuje wszystkimi podstawowymi operacjami bitowymi:

- ♦ operacją iloczynu bitowego (`&`),
- ♦ operacją sumy bitowej (`|`),
- ♦ operacją negacji (`~`),
- ♦ operacją sumy wyłączającej (`^`),
- ♦ operacjami przesunięcia bitowego w lewo (`<<`) i w prawo (`>>`).



Należy zwrócić uwagę na różnice w zapisie operacji bitowych i logicznych. Często zdarzają się pomyłki polegające na błędnym użyciu zamiast operacji bitowej operacji logicznej, np. zamiast iloczynu bitowego (`&`) wykonywana jest operacja iloczynu logicznego (`&&`).

Zanim jednak przejdziemy do szczegółów operacji bitowych, przypomniana zostanie reprezentacja binarna podstawowych typów danych.

## Reprezentacja binarna liczb

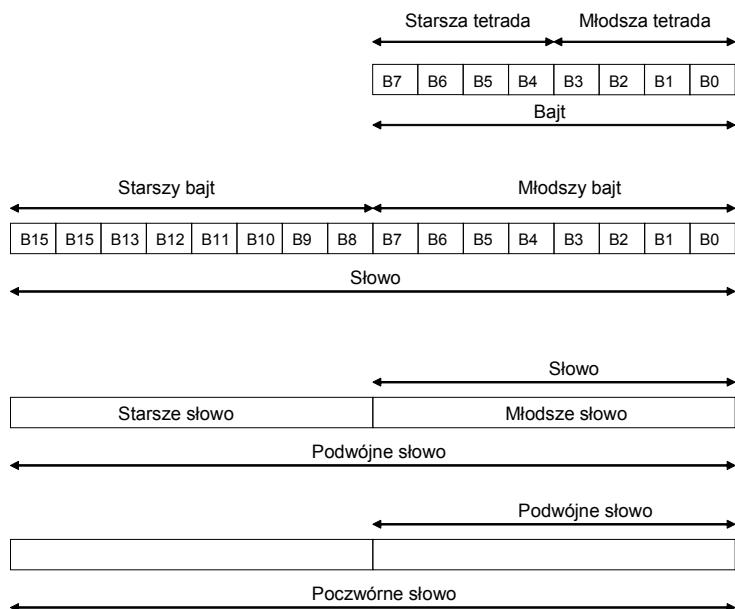
Przywykliśmy w życiu codziennym używać liczb zapisanych w systemie dziesiętnym, lecz dla komputera, posługującego się tylko dwoma stanami, zapis dziesiętny nie jest zapisem naturalnym, stąd też liczby w pamięci komputera zapisywane są w postaci binarnej. Dzięki temu przeprowadzane na nich operacje są znacznie szybsze. Inne formy zapisu stosuje się praktycznie wyłącznie w celu przedstawienia wyników w postaci

czytelnej dla człowieka. Taka zmiana podstawy zapisu liczb wymaga zastosowania odpowiednich funkcji konwersji. Język C dysponuje wbudowanymi funkcjami realizującymi łatwo proces konwersji pomiędzy systemami.

Warto przyswoić sobie podstawową arytmetykę związaną z zapisem binarnym liczb, gdyż wszystkie operacje bitowe przeprowadzane są na tej postaci ich zapisu. Jak zaznaczono w poprzedniej części rozdziału, liczby możemy podzielić na trzy główne typy: liczby całkowite, stałopozycyjne i zmiennopozycyjne. Zazwyczaj nie ma wielkiego sensu przeprowadzanie operacji bitowych na dwóch ostatnich typach liczb, przyjrzyjmy się więc, jak wyglądają te operacje w stosunku do liczb całkowitych.

Liczby całkowite można podzielić na liczby bez znaku i liczby ze znakiem. Podział ten jest niezwykle istotny — w przypadku liczb ze znakiem **najstarszy bit zapisu liczby wykorzystywany jest do określenia jej znaku**. Jeśli jest on równy 1, to liczba jest ujemna. Zmienne o typach całkowitych możemy także podzielić ze względu na wielkość możliwej do zapisania w nich liczby, czyli inaczej mówiąc, ilość zajmowanych przez nie bajtów pamięci. Mogą one w przypadku mikrokontrolerów AVR zajmować 8, 16, 32 lub 64 bity — rysunek 3.4.

**Rysunek 3.4.**  
Oznaczenia typów danych przechowywanych w pamięci komputera



Operacje bitowe przeprowadzane są zawsze na odpowiadających sobie bitach dwóch argumentów. Wyjątkiem jest jednoargumentowa operacja negacji. Poniżej przedstawiono podstawowe typy operacji bitowych.

## Operacja iloczynu bitowego

Operacja iloczynu bitowego daje na odpowiednim biecie wyniku wartość 1 wyłącznie, gdy na tej samej pozycji bitu w obu argumentach znajdowała się wartość 1 — tabela 3.5.

**Tabela 3.5.** Tabela prawdy dla operacji iloczynu bitowego

<b>Argument 1</b>	<b>Argument 2</b>	<b>Wynik</b>
0	0	0
0	1	0
1	0	0
1	1	1

Poniżej przedstawiono wynik operacji iloczynu bitowego dla dwóch liczb:

$$\begin{array}{cccccccc} 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ \hline 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \end{array}$$

Operację iloczynu bitowego wykorzystuje się często do zerowania pewnych bitów. Założymy, że chcemy wyzerować najmłodsze 4 bity bajtu, nie zmieniając stanu 4 najstarszych bitów. Wiedząc, że iloczyn logiczny dowolnego bitu z 1 daje niezmienioną wartość bitu, a z 0 daje zawsze 0, możemy stworzyć odpowiednią maskę:

$x = x \& 0b11110000;$

Warto zapamiętać, że dla 8-bitowej zmiennej  $x$ :

$$\begin{aligned} x \& 0 &= 0 \\ x \& 0xFF &= x \end{aligned}$$

## Operacja sumy bitowej

W przeciwnieństwie do operacji iloczynu bitowego suma bitowa daje w wyniku 1, jeśli dowolny z dwóch bitów argumentów miał wartość 1 — tabela 3.6.

**Tabela 3.6.** Tabela prawdy dla operacji sumy bitowej

<b>Argument 1</b>	<b>Argument 2</b>	<b>Wynik</b>
0	0	0
0	1	1
1	0	1
1	1	1

Zobaczmy, jak wygląda wynik operacji sumy bitowej dla wcześniej pokazanej pary liczb:

$$\begin{array}{cccccccc} 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ \hline 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 \end{array}$$

Operację sumy bitowej wykorzystuje się najczęściej do ustawiania wybranych bitów. Analogicznie jak w poprzednim przykładzie, jeśli chcemy, aby 4 najmłodsze bity zmiennej  $x$  miały wartość 1, przeprowadzamy operację:

$x = x | 0b00001111;$

W efekcie najstarsze 4 bity zmiennej  $x$  pozostaną bez zmian, a cztery najmłodsze będą miały wartość 1.

Warto zapamiętać, że dla 8-bitowej zmiennej  $x$ :

$$\begin{array}{l} x \mid 0 = x \\ x \mid 0xFF = 0xFF \end{array}$$

Łącznie operację sumy i iloczynu bitowego można wykorzystać do przeniesienia części bitów z jednej zmiennej do innej. Np. założmy, że zmienna  $y$  ma zawierać 4 najstarsze bity zmiennej  $x$  i 4 najmłodsze bity zmiennej  $z$  (przyjmujemy, że wszystkie zmienne są 8-bitowe):

```
y=(x & 0b11110000) | (z & 0b00001111);
```

W wyniku powyższej operacji przy pomocy iloczynu logicznego maskowana jest starsza tetrada zmiennej  $x$  i młodsza zmiennej  $z$ , następnie obie tetrydy są „składane” razem przy pomocy operacji sumy logicznej.

## Operacja sumy wyłączającej

Operacja ta daje w wyniku 1, jeśli odpowiadające sobie bity argumentów się różnią — tabela 3.7.

**Tabela 3.7.** Tabela prawdy dla operacji sumy wyłączającej

Argument 1	Argument 2	Wynik
0	0	0
0	1	1
1	0	1
1	1	0

Zobaczmy, jak wygląda wynik operacji sumy wyłączającej dla wcześniej pokazanej pary liczb:

$$\begin{array}{cccccccc} 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ \hline 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \end{array}$$

Operację tę wykorzystuje się do badania, czy dany bit się zmienił. Jeśli jego stan od poprzedniego razu uległ zmianie, to w wyniku dostajemy 1, jeśli nie, to 0.

Warto zapamiętać, że:

$$\begin{array}{l} x \wedge 0 = x \\ x \wedge 0xFF = \sim x \end{array}$$

Z powyższego wynika, że operacja sumy wyłączającej, w której jednym z argumentów jest 0, nic nie zmienia, natomiast z liczbą 0xFF powoduje zmianę na przeciwny wszystkich bitów. Operacja ta ma szerokie zastosowanie. Jednym z ciekawszych przykładów

jest grafika. Zwykle w przypadku wyświetlaczy monochromatycznych piksele zgaszone reprezentowane są przez 0, a włączone przez 1. Aby dokonać negacji obrazu (np. w celu realizacji podświetlenia lub też migania jego fragmentu), przeprowadza się operację sumy wyłączającej, w której jednym z argumentów jest bajt obrazu, a drugi ma wartość 0xFF. **Zauważmy, że ponowne przeprowadzenie takiej operacji na fragmencie obrazu przywraca jego początkowy stan.**



W języku C operacja sumy wyłączającej nie ma odpowiednika wśród operacji logicznych.

Operacja sumy wyłączającej jest czasami używana do wymiany zawartości dwóch rejestrów, bez konieczności użycia trzeciej zmiennej. Klasycznie wymiana zawartości zmiennych a i b wygląda następująco:

```
c=a;  
a=b;  
b=c;
```

Dzięki operacji sumy wyłączającej możemy się obyć bez pomocniczej zmiennej c:

```
a^=b;  
b^=a;  
x^=b;
```

Jak widać, liczba koniecznych do wykonania operacji jest taka sama, lecz w drugim przypadku nie wykorzystujemy żadnych pomocniczych zmiennych, co może zaoszczędzić operacji związanych z zachowaniem i następnie odtworzeniem wartości rejestrów.

## Operacja negacji bitowej

Operacja negacji jest operacją jednoargumentową. Powoduje ona zmianę stanu wszystkich bitów na przeciwnie, np.

1	0	1	1	0	0	1	1
0	1	0	0	1	1	0	0

Operację tę wykorzystuje się w celu przeprowadzenia operacji arytmetycznych dodawania i odejmowania. Zauważmy, że w związku ze sposobem zapisu liczb całkowitych operacja dodawania jest równoważna operacji odejmowania zanegowanego argumentu, np.:

$x=a + b$ , to to samo co:  $x=a - (\sim b)$ .

Oczywiście przy pisaniu w języku C nie musimy stosować takich sztuczek, lecz kompilator często, generując kod assemblerowy, korzysta z powyższej zależności. Jest to spowodowane tym, że lista rozkazów procesorów AVR zawiera operację dodawania stałej do słowa (ADIW), lecz nie zawiera operacji odejmowania stałej od słowa.

## Operacje przesunięć bitowych

Istnieją dwie operacje przesunięcia bitowego: w lewo (`<<`) i w prawo (`>>`).



Na miejsce zwalnianych bitów wstawiane są bity o wartości 0.

Operacji przesunięć bitowych nie należy mylić ze znymi z asemblera operacjami rotacji, w których bit wysuwany z rejestru jest wsuwany z drugiej jego strony (czasami w tej operacji pośredniczy specjalna flaga *Carry* procesora). **Język C nie dysponuje tego typu operacjami.**

Przesunięcie o 0 nie zmienia argumentu. Jeśli przesuwamy argument o więcej pozycji, niż jest używanych do jego reprezentacji, w efekcie uzyskamy 0. I tak np. dla liczby 8-bitowej przesunięcie o więcej niż 7 pozycji daje w efekcie 0.

Operacje te są często wykorzystywane do dzielenia lub mnożenia liczby przez potęgi liczby 2. Zauważmy, że każde przesunięcie w lewo o jeden bit wiąże się z pomnożeniem argumentu 2 razy, natomiast każde przesunięcie o jeden bit w prawo wiąże się z podzieleniem argumentu przez 2.



Obecnie nie ma potrzeby stosowania tego typu operacji jako operacji arytmetycznych. Optymalizator kompilatora sam rozpoznaje takie sytuacje i zamienia mnożenia i dzielenia przez potęgi liczby 2 na odpowiednie operacje przesunięć bitowych.

Dla operacji przesunięcia bitowego w lewo nie ma znaczenia, czy dana zmienna jest ze znakiem, czy bez znaku. Inaczej ma się sytuacja w przypadku operacji przesunięcia w prawo. Dla zmiennych bez znaku operacja przesunięcia w prawo przebiega tak, jak to zostało wcześniej przedstawione. Dla zmiennej ze znakiem przy przesunięciu w prawo na pozycję najstarszego bitu wstawiana jest wartość 1, a nie 0. Dzięki temu po przesunięciu w prawo nie zmienia się znak liczby i jest ona nadal prawidłową liczbą ujemną, zgodnie z oczekiwaniami będącą wynikiem dzielenia przez 2, np. poniższa liczba przesunięta o jeden bit w prawo:

$$\begin{array}{r} 1 \quad 0 \quad 1 \quad 1 \quad 0 \quad 0 \quad 1 \quad 1 \\ \hline 1 \quad 1 \quad 0 \quad 1 \quad 1 \quad 0 \quad 0 \quad 1 \end{array}$$

O ile więc w przypadku innych operacji bitowych znak liczby jest bez znaczenia, traktowany jest jak zwykły bit, o tyle w przypadku operacji przesunięcia w prawo ma on specjalne znaczenie, stąd też należy zwracać uwagę na typ użytej zmiennej.

## Zasięg zmiennych

Zmienne programu możemy podzielić na zmienne globalne i zmienne lokalne. Podział ten jest nie bez znaczenia. Pisząc program, staramy się podzielić złożony problem na wiele problemów prostszych, które możemy rozwiązać np. przy pomocy jednej funkcji.

Funkcje rozwiązuające proste problemy z kolei można łączyć, w efekcie uzyskując rozwiązanie problemu złożonego. Na każdym etapie pojawia się jednak konieczność operowania na zmiennych. Czasami są to zmienne wspólne dla wielu funkcji, a czasami są to zmienne specyficznie związane z daną funkcją, do których dostęp poza nią nie jest konieczny. Dzięki możliwości definicji zmiennych lokalnych można ograniczyć dostęp do takich zmiennych, co ma wiele zalet:

- ♦ W zespołach złożonych z wielu programistów każdy może pracować nad swoją częścią kodu, gdyż kod pisany przez innych nie ma na nią wpływu.
- ♦ Nie trzeba się martwić o konflikty pomiędzy zmiennymi o tej samej nazwie.
- ♦ Zmienne zostają powiązane z kodem, który na nich operuje.

## Zmienne globalne

Do zmiennych globalnych można uzyskać dostęp z każdego punktu programu. Aby zadeklarować taką zmienną, należy jej deklarację umieścić poza funkcją i poza blokiem programu, np.:

```
int x;  
float y;  
int main()  
{  
}
```

W powyższym przykładzie zmienne `x` i `y` są zadeklarowane poza funkcją i poza blokiem programu, stąd dostęp do nich można uzyskać z dowolnego miejsca programu. Dokładniej mówiąc, dostęp do takiej zmiennej można uzyskać od fragmentu programu znajdującego się po jej deklaracji.

### Wskaźówka

O ile to możliwe, należy unikać deklarowania zmiennych globalnych. Ponieważ dostęp do takich zmiennych może odbywać się z różnych części programu, nawet z różnych jednostek komplikacji, utrudnia to debugowanie takiego programu oraz zwiększa jego złożoność.

W pewnym sensie stosowanie zmiennych globalnych przeczy stylowi programowania dzielącemu rozwiązywanie złożonego problemu na wiele problemów prostych. Stosowania zmiennych globalnych możemy uniknąć, przekazując argumenty do funkcji poprzez wartość (jeśli zmienna jest tylko do odczytu) lub poprzez wskazanie (dla zmiennych, których wartość dana funkcja może modyfikować).

Właściwie jedyną sytuacją, w której stosowanie zmiennych globalnych jest niezbędne, jest sytuacja, w której przekazywane są wartości pomiędzy zwykłą funkcją programu a funkcją obsługi przerwania. Ponieważ funkcje obsługujące przerwania nie mogą przyjmować argumentów, jedyną możliwością ich interakcji z resztą programu jest zastosowanie zmiennych globalnych.

## Zmienne lokalne

Zmienne lokalne są deklarowane wewnątrz funkcji lub wewnątrz bloku programu. Wynikają z tego dwie poważne konsekwencje:

- ◆ Dostęp do tak zadeklarowanej zmiennej możliwy jest wyłącznie w funkcji lub w bloku programu, w którym taka zmienna została zadeklarowana.
- ◆ Czas życia takiej zmiennej jest ograniczony — po wyjściu z bloku, w którym zmienna została zadeklarowana, jest ona niszczona.



Wyjątkiem są zmienne statyczne, których zawartość pomiędzy kolejnymi wywołaniami bloku jest zachowana.

Zobaczmy, jak wygląda deklaracja zmiennych lokalnych:

```
void test()
{
    int x;
    while(1)
    {
        int y;
        y=1+x; //Ok, x i y są zadeklarowane
    }
    y=y+1; //Błąd — y tu już nie istnieje
}
```

Zmienna x widoczna jest tylko w obrębie funkcji test(), próba odwołania się do niej poza tą funkcją spowoduje błąd. Z kolei zmienna y zadeklarowana jest w obrębie bloku programu związanego ze słowem kluczowym while. W ramach tego bloku można się do niej odwoływać, lecz próba odwołania się do tej zmiennej poza blokiem spowoduje błąd — zmienna poza blokiem nie istnieje.

Z zasięgiem zmiennej związany jest jeszcze jeden aspekt — przesłanianie. Zobaczmy przykład:

```
void test()
{
    int a=20;
    while(a>0)
    {
        for(int a=0;a<10;a++) asm volatile ("nop");
        a--;
    }
}
```

Zmienna a o typie int została zadeklarowana w funkcji test(), ale oprócz tego została zadeklarowana jako zmienna sterująca pętli for. Nie jest to błędem — w takiej sytuacji tworzone są dwie różne zmienne, zmienna wewnątrz pętli przesyłania zmienną zadeklarowaną na zewnątrz pętli. W efekcie wewnątrz pętli for tworzona jest lokalna wersja zmiennej a, niemającą nic wspólnego ze zmienną a zadeklarowaną na zewnątrz pętli.



Wielokrotne definiowanie zmiennych o takiej samej nazwie nie jest zalecane. Bardzo łatwo prowadzi to do trudnych do wykrycia pomyłek.

## Modyfikator const

Język C dysponuje dwoma modyfikatorami określającymi sposób dostępu do zmiennej: `const` i `volatile`. Ten drugi zostanie omówiony w rozdziale poświęconym przerwaniom. Modyfikator `const` powoduje, że wartość tak zdeklarowanej zmiennej nie może ulec zmianie — staje się ona stałą, np.:

```
const float pi=3.1415;
```

definiuje stałą o nazwie `pi` i typie `float`. Próba przypisania innej wartości tak zdefiniowanej stałej kończy się błędem:

```
error: assignment of read-only variable 'pi'
```

Wynika z tego podstawowe zastosowanie stałych — są one zabezpieczone przed przypadkową modyfikacją.

Modyfikator `const` często wykorzystuje się w połączeniu ze zmiennymi łańcuchowymi:

```
const char txt[] PROGMEM = "Test";
```

Powyższy łańcuch tekstowy rezyduje w pamięci FLASH mikrokontrolera, nie ma więc możliwości jego modyfikacji. Zastosowanie modyfikatora `const` powoduje, że jakakolwiek próba jego modyfikacji zakończy się błędem. Bez tego modyfikatora operacja:

```
txt[1]=0;
```

byłaby poprawna, lecz ponieważ `txt` leży w obszarze pamięci, której nie można zapisywać, byłaby bez efektu<sup>3</sup>.

Podobnie możemy zdefiniować wskaźnik na łańcuch tekstowy:

```
const char *txt1=PSTR("Test1");
```

Wskazywanej wartości łańcucha nie można modyfikować. Można jednak modyfikować sam wskaźnik, np. poprzez przypisanie mu nowej wartości:

```
txt1=PSTR("Test2");
```

Aby nie można było modyfikować wskazywanej wartości oraz samego wskaźnika, potrzebna jest deklaracja:

```
const char * const txt1=PSTR("Test1");
```

Jak widać, modyfikator `const` może w wielu sytuacjach zastępować dyrektywę preprocesora `#define`. Jakie są więc zalety korzystania z `const`? Oto kilka z nich:

<sup>3</sup> W rzeczywistości ze względu na architekturę AVR spowodowałaby zapis do pamięci SRAM nieprzydzielonej zmiennej `txt`.

- ◆ Zasięg zmiennych const jest taki sam jak normalnych zmiennych, zasięg makrodefinicji jest globalny.
- ◆ Stałe const są widoczne w trakcie debugowania programu.
- ◆ Zastosowanie ich umożliwia przeprowadzenie kontroli typów, co zmniejsza podatność na błędy.
- ◆ Można pobrać ich adres.
- ◆ Można przekazać je przez referencję.

Z dwóch ostatnich właściwości wynika jedna wada: stałe const mogą zajmować pamięć — skoro można pobrać ich adres, to znaczy, że muszą być gdzieś fizyczne przechowywane. Odróżnia je to od stałych definiowanych przez #define. Aby pozbyć się tej wady, należy użyć słowa kluczowego static:

```
static const float pi=3.1415;
```

Tak zdefiniowana stała nie będzie zajmowała pamięci i będzie przypominała stałą zdefiniowaną przy pomocy dyrektywy #define, przynajmniej do czasu, kiedy nie będziemy próbować pobrać jej adresu. W takiej sytuacji kompilator będzie zmuszony potraktować ją jako zmienną, tyle że o niezmiennej wartości.

## Wskaźniki

Do tej pory poznaliśmy zmienne, które mogły przechowywać dowolny typ danych. W języku C istnieje jeszcze jeden, bardzo szczególny typ zmiennych. Są to zmienne przechowujące adresy innych zmiennych, czyli tzw. wskaźniki (ang. *Pointers*). Ponieważ wskaźniki przechowują adres wskazywanego obiektu (którym mogą być zmienne, struktury lub funkcje), ich długość (a więc ilość zajmowanego przez nie miejsca w pamięci) zależy od długości adresu w danym typie procesora. Mikrokontrolery AVR posługują się 16-bitowym adresem, a więc wskaźniki zajmują na tej platformie dwa bajty<sup>4</sup>, niezależnie od obiektu, na który wskazują.

Do zwykłych zmiennych odwołujemy się przez ich nazwy, w efekcie uzyskując związaną z nimi wartość. Jednak jak wiemy, zmienne przechowywane są w pamięci, więc nazwa zmiennej tak naprawdę jest adresem komórki pamięci zawierającej daną zmienną. Można więc powiedzieć, że nazwa zmiennej jest wskaźnikiem do miejsca w pamięci, w którym jest przechowywana jej wartość. Typ zmiennej potrzebny jest tylko po to, aby kompilator wiedział, jak interpretować wskazywany przez nazwę zmiennej adres pamięci. Widzimy więc, że deklaracja zmiennej wiąże się z utworzeniem etykiety będącej nazwą zmiennej i przypisaniem jej adresu w pamięci przeznaczonego do przechowywania wartości zmiennej. Podobnie rzecz się ma przy wskaźnikach, z tym że wartość wskaźnika określa adres w pamięci, pod którym przechowywana jest zmienna o typie określonym przez typ wskaźnika.

---

<sup>4</sup> Co prawda większe mikrokontrolery rodziny AVR mają dodatkowe rejestrze adresowe i operują na 24-bitowym adresie, lecz nie jest to wspierane przez gcc.



Wskazówka

Wskaźnik tworzy się poprzez dodanie przed nazwą wskaźnika gwiazdki (\*).

Przyjrzyjmy się przykładowi:

```
int x=10;  
int *ptr=&x;  
*ptr=12;
```

W powyższym przykładzie zdefiniowano zmienną `x`, o wartości początkowej 10. Następnie utworzono wskaźnik na typ `int` i zainicjowano go adresem zmiennej `x` w pamięci.



Wskazówka

Operator `*` & pobiera adres etykiety (nazwy zmiennej lub funkcji).

Następnie pod adres wskazywany przez wskaźnik `ptr` (a więc adres zmiennej `x`) wpisano wartość 12.



Wskazówka

Operator `*` to tzw. operator wyłuskania (dereferencji). Powoduje on zwrócenie wartości przechowywanej pod adresem wskazywanym przez wskaźnik.

Ponieważ pod adresem wskazywanym przez `ptr` przechowywana jest wartość zmiennej `x`, zmiana powoduje zmianę wartości tej zmiennej; w efekcie `x` będzie zawierało wartość 12.



Wskazówka

Podobnie jak zmienne, przed użyciem wskaźnik musi zostać zainicjowany. Użycie niezainicjowanego wskaźnika prowadzi do nieokreślonych efektów. Specjalną wartością wskaźnika jest wartość 0 (`NULL` — symbol zdefiniowany m.in. w pliku nagłówkowym `stdlib.h`).

Wartość `NULL` przypisana wskaźnikowi oznacza, że nie wskazuje on na żaden adres pamięci.

Przypisując wartość wskaźnikowi, należy pamiętać, że przypisanie typu:

```
int *ptr=12;
```

przypisuje wartość 12 wskaźnikowi, a nie miejscu przez niego wskazywanemu. Aby przypisać wartość wskazywanej przez wskaźnik komórce pamięci, należy użyć operatora wyłuskania.

Wskaźniki nie muszą wskazywać tylko na typy proste. Wśród typów złożonych interesujące są wskaźniki na struktury oraz wskaźniki na funkcje. Zaczniemy od wskaźników na struktury:

```
struct point  
{  
    int x;  
    int y;
```

```
};

struct point *pptr;
```

W powyższym przykładzie zdefiniowano wskaźnik na typ `point`. Do poszczególnych pól takiej struktury można się odwoływać przy pomocy operatora dereferencji:

```
(*pptr).y=11;
```

lub operatora `->`:

```
pptr->y=11;
```

Oba odwołania są równoznaczne.

Do czego wskaźniki można wykorzystać? Ich zastosowania są różnorakie, w tym miejscu pokazane zostanie jedno — przekazywanie argumentów przez referencję. W języku C, w przeciwieństwie do C++, jest to możliwe wyłącznie przy użyciu wskaźników. Klasyczne przekazywanie argumentów następuje przez wartość:

```
void test(int x)
{
    x=x+1; //Po powrocie wartość x jest tracona
}
```

Zmiana wartości zmiennej `x` taka jak pokazana w powyższym przykładzie nie ma żadnego wpływu na wartość argumentu, a więc po wywołaniu:

```
int x=10;
test(x);
```

wartość `x` ciągle będzie wynosić 10. Czasami jednak chcemy, aby funkcja miała możliwość modyfikacji argumentów, w takiej sytuacji wykorzystuje się przekazanie przez referencję:

```
void test(int *x)
{
    *x=(*x)+1; //Po powrocie wartość x nie jest tracona
}
```

Po takim zdefiniowaniu funkcji `test` po wywołaniu `test(&x)` wartość zmiennej `x` zwiększy się o 1.

## Wskaźniki o typie void

Specjalnym typem wskaźników są wskaźniki `void`. `void` nie jest żadnym typem, oznacza raczej brak typu. Dzięki temu wskaźniki `void` mogą wskazywać na dowolny typ danych — przypisanie im wskaźników innych typów nie powoduje błędów ani generowania ostrzeżeń. Dzięki temu można konstruować uniwersalne funkcje przyjmujące jako argumenty wskaźniki o dowolnym typie. Niestety, jak zawsze płacimy za to pewną cenę. Ponieważ `void` oznacza brak typu, nie można określić, jak wyglądają dane reprezentowane przez taki wskaźnik, w efekcie nie można dla nich używać operatorów dereferencji `*` i `->`. Aby móc ich użyć na wskaźniku `void`, należy wcześniej dokonać rzutowania typów i zrzutować taki wskaźnik na typ, dla którego dereferencja jest możliwa.

## Wskaźniki na wskaźniki

Wskaźniki na wskaźniki nie są żadnym nowym typem danych. Jest to raczej konsekwencja istnienia wskaźników — nic nie stoi na przeszkodzie, aby zadeklarować wskaźnik na typ wskaźnikowy:

```
int **ptr;
```

Powyższa instrukcja deklaruje wskaźnik na wskaźnik na typ `int`. Oczywiście, sytuację można dowolnie komplikować, deklarując wskaźniki na wskaźniki na wskaźniki itd. Jednak zazwyczaj taka komplikacja nie ma wielu zastosowań, natomiast wskaźniki na wskaźniki są stosunkowo szeroko stosowane, szczególnie przy manipulacjach łańcuchami tekstowymi. W pliku `string.h` znajdują się prototypy dwóch funkcji wykorzystujących tę technikę: `strsep` oraz `strtok_r`. Wskaźniki na wskaźniki używane są w sytuacji, kiedy wywołując funkcję, chcemy mieć możliwość modyfikacji zwracanego wskaźnika:

```
int allocstr(char **ret, int len)
{
    char *p=malloc(len+1);
    if(p==NULL) return 0;
    *ret=p;
    return 1;
}
```

Powyższa funkcja rezerwuje `(len+1)` bajtów pamięci na stercie; jeśli rezerwacja nie powiedzie się, to zwracana jest wartość 0. Jeśli powiedzie się, to zmiennej `ret` nadawana jest wartość wskaźnika do przydzielonej pamięci i zwracana jest wartość 1.



Wskazówka

Ponieważ funkcja może zwracać tylko jedną wartość, nie jest możliwe jednoczesne zwrócenie stanu (wartości 0 lub 1) i wskaźnika do zaallokowanego bloku. Aby to było możliwe, należałoby zwrócić wskaźnik na strukturę zawierającą dwa pola, co jest bardziej skomplikowane niż użycie wskaźnika na wskaźnik.

Powyższą funkcję możemy wykorzystać w następujący sposób:

```
char *ptr;
if(allocstr(&ptr, 10)
{ //Przydzielono pamięć, ptr zawiera wskaźnik do niej
}
```

Zmienna `ptr` w wyniku wywołania funkcji `allocstr` będzie zawierała wskaźnik do zaallokowanej pamięci.

## Arytmetyka wskaźników

Arytmetyka wskaźników jest nieco odmienna niż innych typów, stąd warto o niej w paru zdaniach wspomnieć. Na wskaźnikach można przeprowadzać wszystkie operacje arytmetyczne, ale w praktyce stosuje się operacje inkrementacji/dekrementacji oraz dodawania i odejmowania. W przypadku operacji arytmetycznych niezwykle ważny jest typ wskaźnika. Dodając lub odejmując wartość `n` od wskaźnika, jego adres zmienia się o `n*sizeof(typ_wskaznika)`. Zobaczmy, jak to działa, na poniższym przykładzie:

```
int *w1;
long *w2;

w1=w1+1;
w2=w2+1;
```

W powyższym przykładzie zadeklarowano dwa wskaźniki — wskaźnik `w1` na typ `int` oraz `w2` na typ `long`. Typ `int` jest reprezentowany przez 2 bajty, typ `long` przez 4. Stąd też dodanie wartości jeden do obu wskaźników spowoduje, że `w1` będzie wskazywać na adres o 2 większy, a `w2` na adres o 4 większy niż poprzednia wartość tych wskaźników. Specjalną sytuacją jest sytuacja, w której wskaźnik wskazuje na typ `void`. Operacja `sizeof(void)` jest operacją niedozwoloną, typ `void` nie ma zdefiniowanej długości. Na wielu kompilatorach operacja:

```
void *w3;
w3++;
```

jest więc operacją nieprawidłową. Jednak kompilator `gcc` jest tu mniej restrykcyjny i taka operacja powoduje zwiększenie wskaźnika `w3` o 1; wskazuje on w efekcie na kolejny bajt pamięci.

## Wskaźniki na funkcje

Wskaźniki mogą wskazywać nie tylko na struktury danych, ale także na funkcje. Definicja takich wskaźników może wydawać się nieco dziwna. Spróbujmy utworzyć wskaźnik na funkcję; normalna deklaracja funkcji wygląda następująco:

```
char *strcpy(char *dst, const char *src);
```

Definicja wskaźnika na taką funkcję jest podobna:

```
char *(*strcpy_ptr)(char *dst, const char *src);
```

Jak widać, w nawiasach zawarta jest nazwa wskaźnika poprzedzona gwiazdką (\*). W ten sposób zmienna `strcpy_ptr` stała się wskaźnikiem na funkcję przyjmującą dwa argumenty o typie `char*` i zwracającą wynik o typie `char*`. W efekcie zdeklarowaliśmy wskaźnik na funkcję, ale przed jego użyciem należy go zainicjować. Inicjacja odbywa się tak jak dla klasycznych wskaźników:

```
str_ptr=&strcpy
```

lub jeśli chcemy jednocześnie wskaźnik zadeklarować i zdefiniować:

```
char *(*strcpy_ptr_noparams)(char *, const char *) = strcpy_ptr;
```

Jak widać, aby pobrać adres funkcji, należy jej nazwę poprzedzić znany już operatorem &. Co ważne, po nazwie funkcji nie występują jej argumenty.



**Wskazówka** Próba przypisania tak zdeklarowanemu wskaźnikowi funkcji o innych argumentach lub innym typie zwracanego wyniku zakończy się błędem.

Istnienie wskaźników na funkcje jest ciekawe z jednego powodu — wskaźniki takie mogą zostać wykorzystane do wywołania funkcji:

```
(*strcpy_ptr)(dst, src);
```

Powyższe polecenie powoduje wywołanie funkcji, której adres znajduje się we wskaźniku strcpy\_ptr z argumentami dst i src.

Poniższa funkcja dodaje do siebie dwa argumenty i zwraca wynik:

```
int dodaj(int a, int b) { return a+b; }
```

Adres tej funkcji zostanie przypisany zmiennej wskaźnikowej fadd:

```
int (*fadd)(int,int) = &dodaj;
```

Powyższą funkcję można wywołać przy pomocy wskaźnika na dwa sposoby — bezpośrednio i przy użyciu dereferencji:

```
int suma_5_7 = fadd(5,7);
int suma_6_7 = (*fadd)(6,7);
```

Zaleca się wywoływanie takich funkcji przez dereferencję, dzięki czemu od razu wiadomo, że użyty został wskaźnik.

## Tablice

Tablice są złożonymi strukturami danych, składającymi się z kolekcji elementów o takim samym typie. Tablice zostaną omówione dopiero w tym miejscu, gdyż ich zachowanie bardziej przypomina zachowanie wskaźników niż innych zmiennych. Ogólna postać deklaracji tablicy wygląda następująco:

*Typ nazwa\_tablicy[rozmiar]:*

*Typ* określa typ poszczególnych elementów tablicy, a *rozmiar* liczbę elementów danego typu. Do elementów tablicy można się odwoływać przy pomocy indeksów, elementy indeksowane są od 0 do *rozmiar-1*. Analogicznie do innych zmiennych, tablicę możemy zadeklarować, np.:

```
int tablica[10];
```

albo zadeklarować i zdefiniować:

```
int tablica[3]={1, 2, 3};
```

W tym drugim przypadku nie musimy podawać rozmiaru tablicy, zostanie on nadany automatycznie przez kompilator:

```
int tablica[]={1, 2, 3};
```

W tym przypadku tablica będzie miała tyle elementów, ile znajduje się na liście. Definicja poszczególnych elementów tablicy może być niekompletna:

```
int tablica[10] = {1, 2, 3,};
```

W powyższym przykładzie po trzecim elemencie inicjującym tablicę występuje przecinek. W efekcie kolejne elementy, aż do ostatniego, zostaną zainicjowane wartością 0.



Wskazówka

Rozmiar tablicy określany jest stałą, lecz począwszy od standardu C99, do określania rozmiaru tablicy można użyć zmiennej, przy czym raz określony rozmiar tablicy nie może ulec zmianie.

## Tablice wielowymiarowe

W powyższych przykładach stworzono tablice jednowymiarowe, nic nie stoi jednak na przeszkodzie, aby stworzyć tablice o dowolnej liczbie wymiarów. Dodatkowe wymiary określa się, dopisując je w oddzielnych nawiasach kwadratowych, np.:

```
int tablica[10][10];
```

deklaruje tablicę dwuwymiarową o dziesięciu kolumnach i dziesięciu rzędach. Łącznie z deklaracją można zdefiniować elementy takiej tablicy:

```
int tablica[2][2]={{1,1},{1,2},{2,1},{2,2}};
```

Poszczególne elementy definicji rozdzielone są nawiasami klamrowymi.

## Dostęp do elementów tablicy

Do poszczególnych elementów tablicy można uzyskać dostęp przez indeksy:

```
tablica[1]=10;
tablica2[1][1]=3;
```



Wskazówka

Kompilator nie sprawdza, czy indeks tablicy jest prawidłowy.

Prowadzi to do częstych błędów polegających na przekroczeniu indeksu tablicy, w efekcie nadpisywane są obszary pamięci nieprzydzielone tablicy.



Wskazówka

Jeśli w pisany programie wartość niektórych zmiennych wydaje się przypadkowo zmieniać, jedną z najprawdopodobniejszych przyczyn jest nadpisywanie sąsiednich obszarów pamięci przez nieprawidłowo indeksowane tablice.

Błędy tego typu często popełniane są przy konstrukcji pętli, np.:

```
int tablica[10];
for(int a=0:a<15:a++) tablica[a]=a;
```

W powyższym przykładzie tablica ma tylko 10 elementów, ale w pętli indeksujemy ich aż 15. Aby się ustrzec tego błędu, dobrze jest definiować rozmiary tablic przez stałe:

```
#define rozmiar 10
int tablica[rozmiar];
for(int a=0:a<rozmiar:a++) tablica[a]=a;
```

W tym przypadku maksymalny indeks definiowany jest przez symbol rozmiar, w efekcie zmiana rozmiaru tablicy zawsze pociągnie ze sobą zmianę pętli indeksujących.

Inną ciekawą możliwością jest wykorzystanie operatora `sizeof` do uzyskania liczby elementów tablicy:

```
int tablica[10];
for(int a=0;a<sizeof(tablica)/sizeof(*tablica);a++) tablica[a]=a;
```

Operator `sizeof` zwraca wielkość argumentu, w tym przypadku jest to wielkość całej tablicy i wielkość jej pojedynczego elementu. Stosunek tych wielkości jest równy liczbie elementów tablicy. Odpowiednie wielkości wyliczane są na etapie kompilacji programu, stąd też w kodzie wynikowym umieszczona zostanie tylko odpowiednia stała.

## Tablice jako wskaźniki

Jak wspomniano na wstępie, tablice są bardzo podobne do wskaźników. Łatwo się o tym przekonać, definiując funkcję, która przyjmuje jako argument tablicę:

```
void wyzeruj(int t[])
{
    for(uint8_t a=0;a<10;a++) t[a]=0;
}
```

Powyższa funkcja przyjmuje jako argument tablicę o elementach typu `int`, a następnie zeruje jej 10 pierwszych elementów. Spróbujmy wykorzystać tę funkcję:

```
int tablica[10];
for(uint8_t a=0;a<10;a++) tablica[a]=a;
wyzeruj(tablica);
```

Ponieważ zmienna `tablica` przekazywana jest funkcji `wyzeruj` przez wartość, wydawałoby się, że zerowanie elementów tablicy odbywające się w funkcji `wyzeruj` nie ma wpływu na zmienną `tablica`. Nic bardziej błędnego.



Tablice przekazywane są do funkcji poprzez wskazanie, a nie przez wartość.

Wytlumaczenie tego jest bardzo proste. Tablice zajmują bardzo dużo pamięci. Aby przekazać tablicę przez wartość, należałoby stworzyć drugą tablicę, a następnie skopiować całą pierwszą tablicę do drugiej. Byłby to proces czasochłonny i pamięciochłonny. W tym miejscu właśnie zbliżamy się do wskaźników. Tak naprawdę nazwa tablicy to nic innego jak wskaźnik na obszar pamięci o wielkości definiowanej przez rozmiar tablicy i rozmiar elementu tablicy. Stąd też do tablicy:

```
int tablica[5];
```

można odwoływać się tak, jak pokazano wcześniej, lub:

```
*tablica=1; //odpowiada tablica[0]=1
*(tablica+1)=2; //odpowiada tablica[1]=2
```

Podobnie w przypadku tablic wielowymiarowych:

```
int tablica[2][2];
```

ich deklaracja odpowiada deklaracji wskaźnika na wskaźnik. Do poszczególnych wierszy i kolumn można się więc odwoływać, stosując odpowiednią dereferencję wskaźnika:

```
**tablica = 12; //odpowiada tablica[0][0]=12  
*(tablica[3]+2)=8; //Odpowiada tablica[3][2]=8
```

W przypadku tablic o większej liczbie wymiarów mamy do czynienia ze wskaźnikami na wskaźniki na wskaźniki...

Na wskaźnikach będących nazwami tablic, jak widać, poprawnie działa także arytmetyka wskaźników, dzięki czemu łatwo można odwoływać się do poszczególnych elementów tablicy.

## Ograniczenia tablic

W avr-gcc maksymalna ilość pamięci zajmowanej przez jedną tablicę nie może przekroczyć 32 767 bajtów pamięci. W praktyce to ograniczenie staje się dokuczliwe wyłącznie w sytuacji, w której używamy zewnętrznej pamięci RAM. Stąd też maksymalne rozmiary tablic zależą od typu elementu (jego rozmiaru) oraz liczby wymiarów, np.:

```
uint8_t t[32767];  
uint16_t t[16383];  
uint16_t t[127][128];
```

Drugie ograniczenie nie wynika z ograniczeń platformy AVR, lecz ze standardu języka. Tablice są strukturami statycznymi, o rozmiarze określonym w momencie ich deklaracji. W związku z tym nie można zmieniać rozmiaru tablicy, co zazwyczaj prowadzi do marnotrawstwa pamięci.

# Funkcje

W przeciwieństwie do innych języków programowania, które dysponują procedurami i funkcjami, w języku C występują wyłącznie funkcje. Nie jest to żadnym ograniczeniem, gdyż odpowiednikiem procedur znanych z innych języków są funkcje zwracające void (czyli nic). Funkcje umożliwiają podzielenie złożonego problemu na wiele mniejszych, prostszych do rozwiązania. Stąd też funkcja powinna rozwiązywać jeden prosty problem, wykorzystując do jego rozwiązania zasoby lokalne — powinniśmy jak najmniej korzystać ze zmiennych globalnych. Dzięki temu funkcja jest niezależna od reszty programu, co z kolei czyni program czytelniejszym. Funkcje realizujące podobne cele można grupować w jednym pliku źródłowym, a ich prototypy umieścić w pliku nagłówkowym, dzięki czemu takie funkcje stają się dostępne w innych jednostkach kompilacji programu (innych plikach źródłowych).

Podobnie jak w matematyce, funkcje w języku C przyjmują argumenty i zwracają wynik. Szczególnym typem argumentu i wyniku jest void — oznaczający odpowiednio brak argumentu lub brak wyniku, np.:

```
void nic(void);
```

oznacza funkcję nieprzyjmującą żadnych argumentów i niezwracającą żadnych wyników.

Co ważne, funkcja może mieć dowolną liczbę argumentów, ale może zwracać tylko jeden wynik. Jeśli funkcja musi zwrócić więcej wartości, to trzeba zastosować wskaźniki lub typy złożone (struktury, których pola zawierają zwracane wartości).

Podobnie jak w przypadku zmiennych, funkcje możemy deklarować i definiować. Deklaracja funkcji tworzy jej prototyp, który informuje kompilator, jakie argumenty funkcja przyjmuje i jakie zwraca wyniki. Do skompilowania programu prototyp jest wystarczający, jednak do jego zlinkowania niezbędna jest także definicja funkcji. Stosowanie prototypów nie jest konieczne, ale zalecane. Zwykle prototypy funkcji umieszcza się w plikach nagłówkowych, co umożliwia ich wykorzystanie w różnych plikach źródłowych. Stwórzmy prototyp prostej funkcji:

```
int add(int a, int b);
```

Dzięki takiemu prototypowi w programie możemy odwoływać się do funkcji add. Aby jednak zlinkowanie takiego programu było możliwe, musimy funkcję zdefiniować:

```
int add(int a, int b)
{
    return a+b;
}
```

#### Wskazówka

Definicja funkcji musi dokładnie odpowiadać jej deklaracji. Jakakolwiek niezgodność argumentów lub typu zwracanego wyniku jest niedopuszczalna.

W przeciwieństwie do języka C++, język C nie dopuszcza możliwości przeciążania funkcji — tzn. tworzenia funkcji o takich samych nazwach, lecz różniących się argumentami lub typami argumentów.

Wynik funkcji jest zwracany dzięki słowu kluczowemu `return`. W przypadku funkcji zwracających typ inny niż `void`, `return` musi wystąpić przynajmniej raz w kodzie funkcji. Jeśli funkcja zwraca `void` (czyli nic nie zwraca), słowo `return` może zostać pominięte lub można użyć `return;` — bez jakiegokolwiek argumentu.

Słowo kluczowe `return` może w ciele funkcji wystąpić wielokrotnie:

```
int max(int a, int b)
{
    if(a>=b) return a;
    return b;
}
```

Po natrafieniu na słowo kluczowe `return` dalsze wykonanie funkcji jest przerywane. Stąd w powyższym przykładzie jeśli  $a \geq b$ , zwracana jest wartość `a` i funkcja na tym się kończy. W przeciwnym przypadku zwracana jest wartość `b`.

## Przekazywanie parametrów przez wartość i referencję

Parametry funkcji możemy przekazywać na dwa sposoby — przez wartość lub przez referencję. Argument przekazywany przez wartość staje się zmienną lokalną funkcji, w efekcie jego modyfikacja nie zmienia wartości takiego argumentu poza funkcją. Jest to domyślny sposób przekazywania argumentów, np.

```
void t(int a)
{
    a+1;
}
int x=10;
t(x);
```

Funkcji t przekazana zostaje wartość zmiennej x. Wewnątrz funkcji wartość przypisana zostaje zmiennej a. Jakiekolwiek modyfikacje tej zmiennej nie wpłyną jednak na zmiennej x. Przekazywanie przez wartość ma jeszcze jedną cechę — argumentem funkcji nie musi być zmienna, może nim być także literal, np. wywołanie t(10) jest również poprawne.

W przypadku przekazywania przez referencję do funkcji przekazywana jest nie wartość zmiennej, a jej adres (wskaźnik):

```
void t(int *a)
{
    (*a)+1;
}
int x=10;
t(&x);
```

W wyniku powyższego wywołania do funkcji t zostanie przekazany adres zmiennej x, w efekcie wewnątrz funkcji a będzie wskazywało na obszar pamięci zajmowany przez zmiennej x. Jakiekolwiek modyfikacja pamięci wskazywanej przez zmienną a będzie więc wpływała na wartość zmiennej x. W rezultacie powyższego wywołania wartość zmiennej x wyniesie 11.

Przekazywanie argumentów przez referencję ma więc zastosowanie w sytuacji, kiedy funkcja ma mieć możliwość modyfikacji danego argumentu, a także przy przekazywaniu argumentów wskazujących na duże struktury danych. W takiej sytuacji oszczędza się czas potrzebny na stworzenie kopii takich struktur, zmniejsza się także zapotrzebowanie na pamięć. W ten sposób domyślnie przekazywane są także tablice. Ponieważ do funkcji przekazywany jest adres zmiennej, a nie wartość, w wywołaniu takiej funkcji nie możemy użyć literalu, np. wywołanie funkcji t(10) jest błędne, gdyż nie można pobrać adresu literalu o wartości 10.

## Wywołanie funkcji

Funkcję możemy wywołać, podając jej nazwę i w nawiasach argumenty. W przypadku braku argumentów stosuje się puste nawiasy (), np.:

```
int x=add(10, 2);
```

wywołuje funkcję add z argumentami 10 i 2, a jej wynik przypisuje zmiennej x.

## Rekurencyjne wywołania funkcji

Rekurencja jest techniką programowania polegającą na definiowaniu funkcji poprzez wywołanie jej samej. W wielu przypadkach użycie rekurencji prowadzi do bardziej czytelnego kodu niż użycie iteracji. Najlepiej pokazać, czym jest rekurencja, na prostym przykładzie:

```
unsigned long long silnia(uint8_t a)
{
    if(a<=1) return 1;
    return a*silnia(a-1);
}
```

Powyższa funkcja oblicza silnię z liczby a. W tym celu wywołuje samą siebie, z parametrem a-1, wywoływana funkcja znowu wywołuje siebie z parametrem a-1 (czyli pierwotną wartością a-2) itd., aż do momentu, kiedy a<=1, co kończy rekurencję. Jak widać, aby rekurencja nie doprowadziła do nieskończonej pętli, należy zastosować warunek kończący rekurencję (w tym przypadku a<=1 kończy rekurencyjne wywołania). Bez takiego warunku rekurencja spowodowałaby powstanie nieskończonej pętli, co z kolei doprowadziłoby do nieprawidłowego działania programu. Porównajmy powyższe rekurencyjne obliczanie silni z obliczaniem metodą iteracyjną:

```
unsigned long long silnia(uint8_t a)
{
    unsigned long long x=1;
    while(a>1) x *= a--;
    return x;
}
```

Jak widać, wywołanie rekurencyjne jest prostsze. Ale, niestety, ma poważne wady. Pierwszą jest prędkość działania tak zdefiniowanej funkcji. W każdym wywołaniu rekurencyjnym kompilator musi stworzyć wywołanie funkcji, jej zmienne lokalne i odłożyć adres powrotu. Są to prace czasochłonne, w związku z tym rekurencja jest zwykle wolniejsza niż analogiczna iteracja. Druga wada jest o wiele poważniejsza i wynika bezpośrednio z pierwszej. Rekurencja bardzo obciąża pamięć. W powyższym przykładzie obliczenie silni z liczby n metodą rekurencyjną prowadzi do zagnieżdżenia n wywołań funkcji silnia. Dla funkcji silnia każde wywołanie wiąże się z rezerwacją co najmniej 7 bajtów pamięci SRAM (4 na wynik, 1 na wartość a i 2 na adres powrotu). W dużych komputerach klasy PC pamięciożerność wywołań rekurencyjnych nie ma większego znaczenia, lecz może doprowadzić do poważnych problemów w systemach z ograniczoną ilością pamięci. **Stąd też programując mikrokontrolery, rekurencję należy stosować z dużą ostrożnością.**

# Słowa kluczowe

Poniżej przedstawione zostaną wybrane słowa kluczowe i operatory języka C. Operatory bitowe zostały przedstawione wcześniej i tu już do nich nie będziemy wracać. Poniższe opisy są bardzo skrócone i mają na celu tylko ogólne zorientowanie się w słowach kluczowych języka C, które zostaną wykorzystane w przykładach występujących w kolejnych rozdziałach książki.

## Operatory

**Operator to symbol o dokładnie określonym znaczeniu, pozwalający wykonać określoną operację na jednym argumencie lub większej ich liczbie.** Operatory mają takie samo znaczenie jak funkcje — jednoznacznie odwzorowują zbiór argumentów na zbiór wyników.

Najczęściej wykorzystywane operatory można podzielić na pięć grup:

1. Operatory arytmetyczne, do których zaliczamy: =, \*, /, +, -, %, ++, --.
2. Operatory porównań, do których zaliczamy: ==, >, <, >=, <=, !=.
3. Operatory logiczne, do których zaliczamy: !a, &&, ||.
4. Operatory umożliwiające operacje na bitach: ~, &, |, ^, <<, >>.
5. Inne operatory, m.in. sizeof(), new, delete.

**Na wynik wyrażenia, oprócz zastosowanego operatora arytmetycznego, ma wpływ typ zmiennych będących jego argumentami.** Na przykład program:

```
int n=1;
int i=2;
n:=n/i;
printf(“%d”, n);
```

wyswietli nam liczbę 0. Jest to spowodowane tym, że w typie int nie da się prawidłowo zapisać wartości ½, która powinna być wynikiem powyższych operacji. Przeanalizujmy kolejny przykład:

```
int x=10;
float y=x/3;
printf(“%f”, y);
```

Spodziewalibyśmy się, że skoro zmienna y jest typu zmiennopozycyjnego (float), to poprawnie będzie mogła przechować wynik powyższego dzielenia wynoszący 3,(3). Jednak jako wynik otrzymamy wartość 3. Jest to spowodowane tym, że oba argumenty operacji dzielenia są typu int (zarówno zmienna x, jak i literał 3). Wynik takiej operacji też jest typu int, co powoduje przypisanie do zmiennej y tylko części całkowitej wyniku. Aby temu zapobiec, musimy wymusić wykonanie operacji dzielenia na typie zmiennopozycyjnym, co możemy osiągnąć przez konwersję typów jednego z argumentów tej operacji. W tym celu poinformujemy kompilator, że nasza stała nie jest liczbą typu int, a liczbą o typie float:

```
int x=10;
float y=x/3.0;
printf(..%f", y);
```

Zapis 3.0 informuje kompilator, że stała jest stałą zmiennopozycyjną.



Pamiętajmy, że domyślnie w języku C stałe liczbowe traktowane są tak, jakby były typu int.

Stąd też, jeśli stała przekracza zakres typu int, musimy dodać do niej sufiks określający, jakiego jest typu, np. zapis *12345676890UL* określa, że stała jest typu unsigned long.

## Operatory arytmetyczne

Operatory arytmetyczne umożliwiają wykonanie podstawowych operacji matematycznych. Ich znaczenie jest takie samo jak w matematyce. Tylko zapis trzech z nich jest specyficznie interpretowany przez kompilator. Operator % oznacza wykonanie operacji modulo, w wyniku której otrzymujemy resztę z dzielenia całkowitego dwóch liczb, np. 5 % 2 wyniesie 1. Operatory ++ i -- mogą występować jako prefiks lub sufiks zmiennej bądź obiektu. Dla zmiennych liczbowych określają one, odpowiednio: zwiększenie lub zmniejszenie o 1 wartości zmiennej. Zapisy ++zmienna i zmienna++ pozornie wydają się być podobne — oba spowodują zwiększenie o 1 wartości zmiennej, jednak w niektórych operacjach spostrzeżemy różnice:

```
int zmienna=0;
int x=++zmienna;
int y=zmienna++;
```

W wyniku działania powyższego programu zmienne x i y będą się różniły. Zmienna x przyjmie wynik operacji zwiększenia o 1 zmiennej zmienna, w efekcie czego będzie miała wartość 2. Zmiennej y z kolei zostanie najpierw przypisana wartość zmiennej zmienna, dopiero później wartość zmiennej zmienna ulegnie zwiększeniu o 1. W efekcie zmienna y będzie miała wartość 1. Podobne różnice zaobserwujemy w przypadku zastosowania operatora --.



W przypadku kiedy użyte zmienne są różnych typów, kompilator będzie starał się automatycznie dokonać konwersji typów.

## Operatory porównania

Operatory porównań wykorzystywane są do testowania różnych warunków. Ich zapis jest intuicyjny, z wyjątkiem dwóch operatorów — operatora równości (==) i operatora nierówności (!=). Operator równości to dwa znaki =, co często prowadzi do błędów polegających na tym, że programista zamiast operatora == stosuje operator przypisania =. Jest to błąd szczególnie trudny do wychwycenia, gdyż powstały kod jest poprawny semantycznie według reguł języka C/C++, czasami tylko możemy zaobserwować, że po natrafieniu na taki kod kompilator generuje ostrzeżenie. Poniższe kody:

```
if(x==1) printf(„x jest równy 1”);  
i  
if(x=1) printf(„x jest równy 1”);
```

są semantycznie poprawne, jednak efekt ich działania jest różny. Pierwszy sprawdza, czy zmienna  $x$  jest równa 1, i jeśli tak, wyświetla napis. Drugi kod przypisuje zmiennej  $x$  wartość 1, a następnie sprawdzany jest warunek `if(x)`, który zawsze jest prawdziwy, bo  $x$  jest równy 1, a jak pamiętamy, wartość różna od 0 w języku C odpowiada prawdzie. W efekcie odpowiedni napis jest zawsze wyświetlany.



Pamiętaj, aby nigdy nie wykorzystywać operatora równości `==` do porównywania ze sobą typów zmiennopozycyjnych (`float`, `double`).

Jeśli chcemy sprawdzić, czy dwa argumenty różnią się od siebie, możemy zastosować operator `!=`. Znak `!` jest operatorem negacji, stąd też ten zapis można odczytać jako sprawdzenie negacji równości.

## Operatory logiczne

W C/C++ istnieją operatory realizujące logiczną negację (`!`), iloczyn logiczny (`&&`) oraz sumę logiczną (`||`).



Nie należy mylić tych operatorów z odpowiadającymi im operatorami bitowymi `~`, `&` i `|`.

Operatory te służą do testowania warunków logicznych i mogą być dowolnie łączone. W efekcie ich działania otrzymujemy tylko dwie wartości — *prawdę* (wartość różna od 0, *true*) lub *fałsz* (zero, *false*). Warunek:

```
int a=7;  
int b=5;  
bool w=(a>5) && (b<4);
```

jest fałszywy (wartość zmiennej `w` wyniesie *false*), gdyż co prawda  $a > 5$ , ale  $b$  nie jest mniejsze od 4.

## Kolejność operatorów

Znajomość kolejności wykonywania operatorów jest niezwykle ważna — tak jak w matematyce, aby poprawnie rozwiązać zadanie, musimy wiedzieć, że np. mnożenie ma wyższy priorytet niż dodawanie, również w programowaniu musimy zawsze pamiętać o właściwym priorytecie operatorów.



Pamiętaj, że jeśli masz wątpliwości co do kolejności użytych operatorów, zawsze możesz wymusić kolejność, stosując nawiasy okrągłe.

W tabeli 3.8 pokazano kolejność wykonywania operatorów w C/C++. Zostały one przedstawione w kolejności od operatorów o najwyższym priorytecie do operatorów o najniższym priorytecie. W ramach grupy operatorów o tym samym priorytecie w kolumnie „Ewaluacja” znalazły się informacje o tym, czy operatory są rozwiązywane od lewej czy od prawej.

**Tabela 3.8.** Priorytety wybranych operatorów w języku C/C++

Kolejność	Operator	Opis	Ewaluacja
1.	<code>++, --</code>	Postinkrementacja i postdekrementacja	Z lewej do prawej
	<code>[]</code>	Indeks tablicy	
	<code>.</code>	Wybór elementu przez referencję	
	<code>-&gt;</code>	Wybór elementu przez wskaźnik	
2.	<code>++ --</code>	Preinkrementacja i predekrementacja	Z prawej do lewej
	<code>+ -</code>	Operator jednoargumentowy (znak liczby)	
	<code>! ~</code>	Negacja logiczna i bitowa	
	<code>(typ)</code>	Rzutowanie typów	
	<code>*</code>	Wyluskanie	
	<code>&amp;</code>	Adres elementu	
3.	<code>/ % *</code>	Dzielenie, modulo, mnożenie	Z lewej do prawej
4.	<code>+ -</code>	Dodawanie i odejmowanie	
5.	<code>&lt;&lt; &gt;&gt;</code>	Operacje przesunięcia bitowego	
6.	<code>&lt;= &lt;</code>	Operatory relacji mniejsze lub równe i mniejsze	
7.	<code>&gt;= &lt;</code>	Operatory relacji większe lub równe i większe	
8.	<code>== !=</code>	Operatory relacji równa się i różne	
9.	<code>&amp;</code>	Iloczyn bitowy	
10.	<code>^</code>	Bitowa suma wyłączająca	
11.	<code> </code>	Suma bitowa	
12.	<code>&amp;&amp;</code>	Iloczyn logiczny	
13.	<code>  </code>	Suma logiczna	
14.	<code>=</code>	Przypisanie	Z prawej do lewej
	<code>+= -=</code>	Przypisanie sumy lub różnicy	
	<code>*= /= %=</code>	Przypisanie mnożenia lub wartości modulo	
	<code>&lt;&lt;= &gt;&gt;=</code>	Przypisanie wyniku operacji przesunięcia bitowego	
	<code>&amp;= ^=  =</code>	Przypisanie wyniku operacji bitowej	

Wykorzystując informacje o priorytetach operatorów, spróbujemy określić kolejność operacji na kilku przykładach. Wyrażenie `int x=++y*x;` będzie interpretowane następująco: operator predekrementacji w tym wyrażeniu ma najwyższy priorytet, dlatego najpierw wartość `x` zostanie zwiększena o jeden; kolejnym operatorem o najwyższym priorytecie jest mnożenie, stąd wynik poprzedniej operacji zostanie pomnożony przez `y`; operatorem o najniższym priorytecie w tym wyrażeniu jest operator przypisania, stąd

też wynik poprzednich operacji zostanie przypisany zmiennej y. Kolejne wyrażenie, `y=(int)x<<4;`, będzie realizowane następująco: najwyższy priorytet ma operator rzutowania typu `(int)`, dopiero kolejny jest operator przesunięcia bitowego `<<`, a na końcu operator przypisania `=`.

## Instrukcje sterujące

Instrukcje sterujące umożliwiają sterowanie kolejnością wykonywania instrukcji programu.

### Instrukcja if

Instrukcja `if` sprawdza podany warunek. Jeśli jest on prawdziwy, wykonuje podany blok instrukcji, w przeciwnym przypadku go pomija. Ogólna postać tej instrukcji jest następująca:

```
if(warunek) instrukcja; else inne_instrukcje;
```

Opcjonalnie w warunku może pojawić się słowo kluczowe `else`, wskazujące na instrukcje, które zostaną wykonane, jeśli podany warunek nie będzie prawdziwy (zwróci `falsz`). W związku z tym bloki instrukcja i inne\_instrukcje mogą być wykonane alternatywnie, ale mamy gwarancję, że nigdy nie zostaną wykonane łącznie. Instrukcje warunkowe można dowolnie zagnieżdżać, np.:

```
if(warunek1) instrukcja1;
else if(warunek2) instrukcja2;
else if(warunek3) instrukcja3;
```

Jak pamiętamy, w języku C/C++ wartość różna od zera jest interpretowana jako *prawda*, a wartość równa 0 jest interpretowana jako *falsz*, stąd pewne warunki można uprościć. Aby sprawdzić, czy zmienna `x` jest różna od zera, możemy napisać:

```
if(x!=0) instrukcja;
```

lub po prostu:

```
if(x) instrukcja;
```

### Instrukcja switch

Jeśli chcemy sprawdzać kolejne wartości lub przedziały wartości zmiennej i w zależności od tego wykonywać pewne bloki instrukcji, to zamiast stosować wielokrotne porównania (instrukcja `if`), możemy zastosować instrukcję `switch`. Jej ogólna postać jest następująca:

```
switch (wyrażenie)
{
    case wartość1 : instrukcja; break;
    case wartość2: instrukcja; break;
    default: instrukcja;
};
```

Wartością wyrażenia użytego w switch musi być jakikolwiek typ porządkowy (np. int, char), nie może być nią wartość float lub double. Kolejne wartości podane przy instrukcji case porównywane są z wartością wyrażenia i jeśli porównanie jest prawdziwe, wykonywany jest ciąg instrukcji za case. Opcjonalnie taki ciąg może kończyć się słowem kluczowym break, co przerwie wykonywanie całej instrukcji switch (nie będą wykonywane kolejne porównania). Jeśli instrukcję break pominiemy, po zakończeniu danego bloku instrukcji wykonana zostanie kolejna instrukcja case, i tak do końca bloku switch. Specjalne znaczenie ma słowo kluczowe default — powoduje ono, że jeśli żadna z wartości przy instrukcjach case nie pasuje do wartości wyrażenia, to wykonany zostanie blok instrukcji położony za default. Oczywiście, w tym przypadku nie ma sensu kończyć tego bloku instrukcją break (chociaż nie jest to niedozwolone), gdyż po zakończeniu tego bloku program będzie automatycznie kontynuowany od bloku po instrukcji switch (nie będzie kolejnych instrukcji case).

Instrukcja switch może przybrać jeszcze jedną postać:

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

W tym przypadku blok instrukcje\_case zostanie wykonany, jeśli którykolwiek z wyrażeń przy case (wartosc1, wartosc2 lub wartosc3) da pozytywny wynik porównania z wartością wyrażenia.

## Pętla while

Pętla while wykonuje znajdująca się za nią instrukcję (blok instrukcji) dopóty, dopóki podany w wyrażeniu warunek jest prawdziwy (różny od zera). Wynika z tego, że blok instrukcji po while może się wykonywać nieskończoną liczbę razy (jeśli wartość wyrażenia zawsze będzie prawdziwa) lub nigdy (jeśli wartość wyrażenia zwróci fałsz). Ogólna postać tej pętli wygląda następująco:

```
while(wyrażenie) instrukcja;
```

Przeanalizujmy przykład:

```
int y=15;
while(y>0)
{
    y--;
    printf(„%d”, y);
};
```

Powyższa pętla wykona się 15 razy.

Szczególnym przykładem pętli while jest pętla nieskończona, która ma postać:

```
while(1) {blok_instrukcji};
```

Powyższa pętla będzie zawsze wykonywana, w efekcie program nigdy nie wykona następnej instrukcji. Jedyną możliwością opuszczenia takiej pętli jest wykorzystanie instrukcji break lub goto.

## Pętla do..while

Podstawowa różnica pomiędzy pętlą do..while a pętlą while sprowadza się do tego, że warunek zakończenia pętli sprawdzany jest na końcu bloku instrukcji tworzących pętlę, a nie na początku. W związku z tym wskazany blok instrukcji wykona się przy najmniej raz — nawet jeśli warunek kończący pętlę będzie nieprawdziwy. Pętla do..while powtarzana będzie aż do momentu, kiedy warunek kończący pętlę zwróci prawdę. Pętla do..while ma postać:

```
do
{
    Blok_instrukcji;
} while(warunek);
```

## Pętla for

Pętla for podobna jest do wcześniej omówionych form pętli. Analogicznie w każdym obiegu pętli testowany jest warunek, a pętla powtarzana jest dopóty, dopóki warunek jest prawdziwy — warunek sprawdzany jest na początku każdej iteracji. Opcjonalnie w tej pętli możemy wykorzystać **zmienną sterującą pętlą**. Na zmiennej tej automatycznie w każdym obiegu pętli mogą być przeprowadzone wskazane operacje, np. zwiększenie zmiennej o zadaną wartość. Często taką zmienną sterującą testuje się w warunku zakończenia pętli. Pętla for ma następującą postać:

```
for(zmienna=wartosc_poczatkowa; warunek_dalszego_wykonywania_petli;
    ↴operacja_na_zmiennej)
```

Zmiennej sterującej zostaje przypisana wartość początkowa przed pierwszym obiegiem pętli. Każdą z części pętli for można pominąć, np. instrukcja:

```
for(;;);
```

spowoduje powstanie nieskończonej pętli. Z kolei:

```
for(int x=0;x<5;x++) instrukcja;
```

spowoduje pięciokrotne powtórzenie bloku o nazwie instrukcja.

## Instrukcja break

Instrukcja break stosowana jest do przerywania każdej z wyżej wymienionych form pętli. Jej działanie jest analogiczne do wcześniej omówionej instrukcji switch — po napotkaniu instrukcji break pętla jest przerywana, a sterowanie wraca do kolejnej instrukcji po pętli:

```
for(int x=0; x<5;x++)
{
    printf("%d", x);
    if(x==2) break;
};
```

W powyższym programie wykonane zostaną tylko trzy iteracje pętli — kiedy zmienna  $x$  osiągnie wartość 2, pętla zostanie przerwana. Podobnie wcześniej pokazana nieskończona pętla (warunek jest zawsze prawdziwy):

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

może zostać przerwana, jeśli *warunek* będzie prawdziwy.

## Instrukcja `continue`

Instrukcja `continue` powoduje natychmiastowe przejście do kolejnej iteracji pętli, z pominięciem dalszych instrukcji. Poniższy przykład:

```
for(int x=0; x<5; x++)
{
    if(x==3) continue;
    printf(“%d\n”, x);
}
```

spowoduje wyświetlenie na ekranie kolejnych liczb 0 – 4 z pominięciem liczby 3 — warunek `if(x==3)` stanie się wtedy prawdziwy i nastąpi przejście do kolejnej iteracji pętli, z pominięciem instrukcji odpowiedzialnej za wyświetlenie na ekranie wartości zmiennej  $x$ .

## Instrukcja skoku `goto`

Instrukcja `goto` jest bardzo rzadko używana. Powoduje ona skok do wskazanego miejsca programu oznakowanego tzw. etykietą. Umożliwia ona natychmiastowe opuszczenie dowolnie zagnieżdzonej pętli lub warunku, jednak jej użycie znacznie pogarsza czytelność kodu — z tego powodu większość programistów stara się jej unikać. W poniższym przykładzie:

```
int x=0;
skok:
x++;
if(x<5) goto skok;
```

została utworzona za pomocą instrukcji `goto` pętla, która składa się z 4 iteracji.

# Preprocesor

Preprocesor jest oddzielnym w stosunku do kompilatora programem. Jest on wywoływany przez kompilator przed procesem komplikacji jako pierwszy etap translacji pliku. Preprocesor dysponuje swoimi własnymi słowami kluczowymi i dyrektywami. Odróżniają się one od kodu w języku C tym, że rozpoczynają się od znaku `#`. Dyrektywy preprocesora możemy podzielić na:

- ◆ dyrektywy związane z włączaniem pliku (`#include`),
- ◆ dyrektywy związane z definiowaniem makr i symboli (`#define`),
- ◆ dyrektywy związane z komplikacją warunkową (`#if`, `#else`, `#endif`, `#ifndef`, `#ifdef`, `#elif`).

## Dyrektyna `#include`

Dyrektyna ta włącza w miejscu swojego wystąpienia zawartość wskazanego pliku. Zwykle wykorzystuje się ją do włączania plików nagłówkowych (.h).



Jakkolwiek jest to możliwe, nie należy wykorzystywać tej dyrektywy do włączania plików źródłowych. Pliki takie powinny być osobno kompilowane i linkowane, a udostępniane przez nie obiekty powinny znajdować się w stosownym pliku nagłówkowym.

Plik nagłówkowy możemy włączać na dwa sposoby: umieszczać nazwę w nawiasach ostrych (<>) lub w cudzysłowach (""). W przypadku umieszczenia nazwy włączanego pliku nagłówkowego w nawiasach ostrych, np.:

```
#include <stdlib.h>
```

kompilator szuka takiego pliku w katalogach wymienionych w parametrze `include` wywołania kompilatora (-I). W sytuacji, w której włączany plik jest w cudzysłowach:

```
#include "test.h"
```

przeszukiwanie rozpoczyna się od katalogu lokalnego, dopiero w następnej kolejności przeszukiwane są katalogi wymienione w opcji kompilatora -I.

## Dyrektwy komplikacji warunkowej

Dyrektwy te umożliwiają warunkową komplikację części kodu programu. W ten sposób w zależności od wartości symbolu pewne fragmenty kodu źródłowego mogą zostać wyeliminowane z komplikacji, a inne bloki mogą zostać włączone.

Zaczniemy od dyrektyw `#ifndef` i `#ifdef` — sprawdzają one, czy dany symbol jest zdefiniowany. Pierwsza zwraca prawdę, jeśli podany symbol nie jest zdefiniowany, a druga jeśli jest, np.:

```
#ifndef _TEST
//Ten fragment zostanie skompilowany, jeśli _TEST nie jest zdefiniowany
#endif
#ifndef _TEST
//A ten fragment jest komplikowany, jeśli symbol _TEST jest zdefiniowany
#endif
```

Przy pomocy dyrektywy `#if` można testować prawdziwość warunków i wyniki wyrażeń arytmetycznych, np.:

```
#if X==10
//To się skompiluje, jeśli symbol X będzie miał wartość 10
#endif
```

Warunek dla dyrektywy `#if` może być dowolnym prawidłowym wyrażeniem.

Jak widać na poprzednich przykładach, każda dyrektywa `#if`, `#ifdef` lub `#ifndef` musi kończyć się odpowiadającą jej dyrektywą `#endif`. Oprócz niej można stosować dyrektywy przypominające słowa kluczowe języka C — `#else` i `#elif`. `#else` wykonywane jest, jeśli warunek z poprzedzającej dyrektywy nie jest prawdziwy, np.:

```
#ifndef _TEST
//Ten fragment zostanie skompilowany, jeśli _TEST nie jest zdefiniowany
#else
//A ten fragment jest kompliowany, jeśli symbol _TEST jest zdefiniowany
#endif
```

Z kolei dyrektywa `#elif` służy do zagnieźdzania dyrektyw `#if`, np.:

```
#if X==1
//Ten fragment zostanie skompilowany, jeśli X==1
#elif X==2
//Ten fragment zostanie skompilowany, jeśli X==2
#else
//Ten fragment jest kompliowany, jeśli X!=1 i X!=2
#endif
```

Uzupełnieniem powyższych dyrektyw jest dyrektywa `defined`. Sprawdza ona, czy podany symbol jest zdefiniowany:

```
#if defined(_TEST)
//Ten fragment zostanie skompilowany, jeśli _TEST nie jest zdefiniowany
#endif
```

Powyższy test jest równoważny użyciu dyrektywy `#ifdef`. Jednak tylko w pokazanym przykładzie. Dyrektywa `defined` umożliwia tworzenie złożonych warunków logicznych, czego nie umożliwiają proste dyrektywy `#ifdef` i `#ifndef`, np.:

```
#if defined (_AVR_) && defined (_AVRLIBC_)
//Ten fragment zostanie skompilowany, jeśli _AVR_ i _AVRLIBC_ są zdefiniowane
#endif
```

## Usuwanie kodu

Powyższe dyrektywy kompilacji warunkowej stosuje się czasami do usuwania fragmentów kodu. Jeśli dany fragment programu chcemy usunąć (spowodować, aby nie był kompliowany), ale zostawić go w kodzie, zwykle nie wystarczy go zakomentować — komentarzy blokowych nie można zagnieździć. W takiej sytuacji stosuje się dyrektywę warunkową, z warunkiem dającym zawsze fałsz, np.:

```
#if 0==1
//Ten kod nie będzie kompliowany
#endif
```

## Dyrektywa #define

Dyrektywa ta przypisuje określonemu symbolowi podaną wartość. W efekcie wykorzystanie w kodzie programu zdefiniowanego symbolu powoduje wstawienie w jego miejsce wartości symbolu, np.:

```
#define pi 3.14.15  
float x=pi*r*r;
```

powoduje zastąpienie symbolu `pi` jego wartością, w efekcie uzyskujemy:

```
float x=3.14.15*r*r;
```

Definicja symbolu jest bardzo podobna do definicji stałej w języku C. Jednak w przypadku użycia symbolu kompilator nie ma możliwości przeprowadzenia kontroli typu, co jest poważną wadą i potencjalnym źródłem błędów. Stąd lepiej jest używać do definicji stałych konstrukcji języka C:

```
static const float pi=3.1415;
```

Oprócz prostych podstawień przy pomocy `#define` można podstawać złożone bloki tekstu.

Przy pomocy tej dyrektywy można tworzyć tzw. makrodefinicje, które mogą mieć także postać funkcji, np.:

```
#define Pole(r) (pi*r*r)
```

Jednak z takim wykorzystaniem makr wiążą się poważne problemy. Spróbujmy wywołać makrodefinicję `Pole`:

```
p=Pole(x+r);
```

Spodziewalibyśmy się, że makro `Pole` zostanie wywołane z argumentem będącym sumą wartości `x` i `r`. Nic bardziej błędnego. Jak wspomniano na wstępie, makrodefinicje powodują po prostu wstawienie w miejscu wywołania związanego z nimi tekstu, stąd powyższy przykład dla kompilatora będzie wyglądał następująco:

```
p=3.1415*x+r*x+r;
```

Jak widać, z pewnością nie jest to to, o co nam chodziło. Sytuację można ratować, poprawiając makro `Pole`:

```
#define Pole(r) (pi*(r)*(r))
```

co doprowadzi do powstania poprawnego wyrażenia:

```
p=3.1415*(x+r)*(x+r);
```

Jak wspomniano na wstępie, kompilator, wstawiając symbole, nie ma możliwości sprawdzenia ich typów. Ma to wiele wad, ale także zaletę, uzasadniającą ich stosowanie. Przeanalizujmy poniższe makro:

```
#define MAX(a,b) ((a)<(b)?(b):(a))
```

Zwraca ono wartość większego wyrażenia (a lub b). Ponieważ kompilator po prostu wstawia wartość makra w miejscu jego wywołania, powyższe makro nadaje się do sprawdzania zmiennych dowolnych typów. W pewnym sensie takie zastosowanie makra jest odpowiednikiem szablonów (ang. *Templates*) znanych z języka C++. Lecz ma także wady. Jak widać, wyeliminowana została wada makra `Pole i argumenty w wyrażenia` występują w nawiasach, co gwarantuje ich poprawną ewaluację. Niestety, pojawia się tu kolejny problem. Przeanalizujmy wywołanie:

```
a=MAX(a++,b);
```

Zostanie ono rozwinięte do postaci:

```
a=((a++)<(b)?(b):(a++));
```

w sytuacji, kiedy `a>b` wartość a zostanie zinkrementowana dwukrotnie. Rozwiążanie tego problemu, niestety, nie jest banalne — wymaga wprowadzenia zmiennych pomocniczych.

Stąd też jeśli nie potrzebujemy makr niezależnych od typów parametrów, powinniśmy wykorzystywać funkcje, w tym funkcje `static inline`, które są równoważne makrom, lecz nie posiadają ich wad.

## Pliki nagłówkowe i źródłowe

Pisząc jakikolwiek większy program w języku C, warto od początku dbać o jego przejrzystość. Nic nie stoi na przeszkodzie, aby cały skomplikowany program umieścić w jednym pliku, lecz takie rozwiązanie ma same wady. Po pierwsze, jakakolwiek zmiana w takim pliku wiąże się z koniecznością przekompilowania całego programu, co może być operacją dosyć czasochłonną. Drugim powodem jest czytelność — znalezienie konkretnej funkcji czy fragmentu programu w pliku liczącym kilkadziesiąt tysięcy linii kodu nie jest łatwe. Stąd naturalnym rozwiązaniem jest podzielenie kodu programu na odrębne pliki, w których znajdują się fragmenty kodu odpowiedzialne za realizację powiązanych ze sobą funkcji. Np. program dokonujący pomiarów przy pomocy przetwornika ADC, a następnie wyświetlający je na wyświetlaczu LCD, dodatkowo umożliwiający interakcję z użytkownikiem przy pomocy menu można podzielić na trzy w miarę niezależne części:

- ◆ kod związany z obsługą LCD,
- ◆ kod związany z obsługą przetwornika ADC,
- ◆ kod związany z obsługą menu.

Dzięki temu zdecydowanie łatwiej jest znaleźć fragment kodu odpowiedzialny za konkretne zadanie. Co więcej, pewne fragmenty kodu mogą być wykorzystywane przez różne aplikacje — np. kod odpowiedzialny za obsługę LCD można wykorzystać także w innej aplikacji, która korzysta z wyświetlacza. Jednak rozbicie aplikacji na wiele plików wiąże się z koniecznością wywoływanego funkcji zdefiniowanych w innych plikach. Aby móc tego dokonać, potrzebny jest nowy typ plików — pliki nagłówkowe.

Pliki te, w przeciwieństwie do plików źródłowych, posiadają rozszerzenie *h* i zazwyczaj zawierają tylko deklaracje użytych funkcji i zmiennych. W efekcie plik źródłowy chcący wykorzystać funkcje, których deklaracja znajduje się w innym pliku źródłowym, wczytuje tylko odpowiedni plik nagłówkowy, zawierający prototypy funkcji i deklaracje zmiennych oraz struktur danych, udostępnionych przez związany z nim plik źródłowy. Takie pliki nagłówkowe możnałączyć wielokrotnie, w różnych plikach źródłowych, przy pomocy dyrektywy preprocesora `#include`. Pliki źródłowe zwykle komplilowane są niezależnie od pozostałych plików źródłowych tworzących projekt. W efekcie powstaje zbiór plików obiektowych, które są łączone razem przez linker. Oprócz poprawy czytelności kodu i możliwości jego sprawnej edycji ma to jeszcze inne zalety. Wprowadzane zmiany w projekcie ograniczają się zazwyczaj do pojedynczych plików źródłowych. Pliki, w których nie wprowadzono zmian od ostatniej kompilacji projektu, nie muszą być komplilowane ponownie (o ile nie zmieniły się łączane przez nie pliki nagłówkowe). Dzięki temu skracą się czas kompilacji. Druga zaleta takiego podziału ujawnia się w sytuacji, w której nad programem pracuje niezależnie kilku programistów. Każdy z nich może pracować na pewnych plikach źródłowych, udostępniając efekty swojej pracy dla innych poprzez funkcje, których prototypy znajdują się w pliku nagłówkowym. Dopóki udostępnione prototypy funkcji nie ulegną zmianie, zmiany wprowadzone w kodzie tych funkcji nie mają najmniejszego wpływu na kod źródłowy reszty programu.

Rozdzielenie deklaracji od definicji i tworzenie plików nagłówkowych stwarza także inne możliwości optymalizacji pisaneego programu i wpływanie na sposób jego kompilacji, co zostanie poniżej pokazane na wybranych przykładach. Jest to szczególnie istotne w systemach opartych na mikrokontrolerach, w których trzeba bardzo racjonalnie gospodarować dostępnymi zasobami sprzętowymi.

## Definicja a deklaracja

W języku C obiekt może być niezależnie deklarowany lub definiowany. Deklaracja obiektu (zmiennej lub funkcji) powoduje tylko zarezerwowanie związanej z nim nazwy, dodatkowo informuje kompilator, jaki jest typ obiektu, a w przypadku funkcji jakie przyjmuje parametry i jakie zwraca wyniki. W języku C dany obiekt może być deklarowany wielokrotnie — oczywiście wszystkie deklaracje muszą być takie same. Ponieważ deklaracja nie wiąże się z przydzieleniem obiektowi jakichkolwiek zasobów, nie można zadeklarowanemu obiektowi przypisywać wartości ani się do niego odwoływać. Aby to było możliwe, obiekt musi zostać zdefiniowany. Definicja obiektu jest pojęciem szerszym, każda definicja jest jednocześnie deklaracją obiektu. W przeciwieństwie do deklaracji definicja wiąże się z przydzieleniem obiektowi zasobów (np. adresu pamięci, pod którym się znajduje). Każdy obiekt może być zdefiniowany tylko jeden raz. Poniżej przedstawione zostały przykłady deklaracji obiektów:

```
extern int z;  
void foo();
```

Znaczenie słowa kluczowego `extern` zostanie wyjaśnione w dalszej części książki. Z kolei:

```
int z;
void foo()
{//Ciało funkcji
}
```

to przykłady definicji obiektów.

## Słowo kluczowe static

Słowo static może być użyte zarówno w połączeniu z funkcjami, jak i zmiennymi. W obu przypadkach jego użycie ogranicza zasięg symbolu do jednostki komplikacji, w której dana funkcja/zmienna zostały zadeklarowane. Dzięki temu nie są one dostępne w innych jednostkach komplikacji. Stąd też funkcje statyczne są zwykle wewnętrznymi funkcjami danego modułu, a użycie słowa kluczowego static zapobiega próbce ich wykorzystania w innych jednostkach komplikacji. Mówiąc prościej — jeśli nie chcemy, aby dana funkcja lub zmienna była dostępna poza modelem, w którym jest zdefiniowana, należy jej nazwę poprzedzić słowem static. W praktyce jest to więc wykorzystywane przy pisaniu wewnętrznych funkcji biblioteki, które nie mają poza nią być dostępne.

O wiele ciekawsze zastosowanie ma powyższe słowo kluczowe w odniesieniu do zmiennych. Zdefiniowanie zmiennej jako static powoduje, że czas jej życia rozciąga się na cały czas wykonywania aplikacji. Zmienna taka jest tworzona jednorazowo, podczas startu aplikacji, i istnieje aż do jej zakończenia. W przeciwnieństwie do innych zmiennych zmienne statyczne inicjalizowane są na etapie komplikacji, a ich wartość jest zachowywana pomiędzy kolejnymi odwołaniami do zmiennej, niezależnie od jej zasięgu. Najprościej będzie to pokazać na przykładzie:

```
int licznik()
{
    static int x;
    x++;
    return x;
}
```

Zmienna x została zadeklarowana jako statyczna, zgodnie ze standardem języka C jest ona inicjalizowana na etapie komplikacji programu wartością 0. Stąd też przy pierwszym wywołaniu funkcji licznik zostanie zwrocona wartość 1, jednocześnie wartość zmiennej x zostanie zwiększena o jeden. Ponieważ czas życia zmiennych statycznych rozciąga się na cały czas wykonywania programu, ponowne wywołanie funkcji licznikwróci wartość 2. Tym razem zmienna x, w wyniku poprzedniego wywołania funkcji, będzie miała wartość początkową 1. Oczywiście, wartość początkowa zmiennej statycznej może być dowolna — wystarczy ją zainicjować inną wartością.

Specyficzną sytuacją jest sytuacja, w której zmienna statyczna występuje w pliku nagłówkowym. Zdefiniujmy w pliku nagłówkowym *zmienna.h* zmienną:

```
static int g;
```

Taka deklaracja spowoduje, że każde dołączenie takiego nagłówka spowoduje utworzenie lokalnej w danej jednostce komplikacji kopii zmiennej g. **W większości przypadków**

**taka definicja zmiennej w nagłówku jest błędem**, natomiast bywa pożądana w przypadku pliku źródłowego — ogranicza widoczność zdefiniowanego symbolu do pliku, w którym taka definicja wystąpiła.

## Słowo kluczowe extern

Słowo kluczowe `extern` jest w dużej mierze przeciwieństwem słowa kluczowego `static`. Może być użyte w odniesieniu do funkcji lub do zmiennych. Aby zrozumieć działanie `extern`, musimy przypomnieć sobie informacje na temat deklaracji i definicji zmiennych oraz funkcji. Deklaracja informuje kompilator, jakiego typu jest dana zmienna, a w przypadku funkcji jakie przyjmuje argumenty oraz jakie zwraca wyniki. Jednak sama deklaracja nie wiąże się z zarezerwowaniem pamięci przewidzianej dla danej zmiennej lub funkcji. Co ważne, w programie dana zmienna lub funkcja mogą być zadeklarowane wielokrotnie. Ale oczywiście wszystkie deklaracje muszą być takie same. W przeciwieństwie do deklaracji zdefiniowanie zmiennej lub funkcji wiąże się z przydzieleniem dla niej pamięci. Dodatkowo w programie dana zmienna lub funkcja mogą być zdefiniowane wyłącznie jeden raz. Użycie słowa kluczowego `extern` w stosunku do funkcji absolutnie nic nie zmienia<sup>5</sup>. Dzieje się tak, ponieważ prototyp funkcji jest wyłącznie jej deklaracją, a nie definicją. W efekcie wystarczy zadeklarować prototyp, a następnie taką deklarację (np. w postaci pliku nagłówkowego) dołączać w plikach źródłowych, w których daną funkcję chcemy wykorzystać. Zobaczmy, jak wygląda użycie `extern` w stosunku do zmiennych. Otóż `extern` powoduje, że dana zmienna jest wyłącznie deklarowana, a nie definiowana. W efekcie analogicznie jak w przypadku funkcji deklarację zmiennej możemy powtórzyć wielokrotnie. Przeanalizujmy poniższy przykład:

```
extern int d;
void addtod()
{
    d++;
}
```

Zadeklarowaliśmy zmienną `d`, informując kompilator, że jej definicja występuje w innym module. Dzięki deklaracji kompilator wie, jakiego typu jest zmienna `d`, może więc wygenerować odpowiedni kod. Zostanie on powiązany z definicją zmiennej `d` na etapie linkowania programu. Aby powyższy kod mógł zostać skompilowany, musimy w innym module zdefiniować zmienną `d`. Jeśli tego nie zrobimy, zgłoszony zostanie błąd:

```
C:\ float.c:16: undefined reference to `d'
```

Błąd dotyczy miejsca programu, w którym następuje odwołanie do zmiennej `d`, a więc linii `d++;`.

Jak wcześniej wspomniano, `extern` powoduje, że dana zmienna jest deklarowana, ale nie jest definiowana (nie jest jej przydzielane miejsce w pamięci). Jednak istnieją wyjątki od tej reguły. Jeśli w poprzednim przykładzie rozszerzymy dyrektywę `extern`:

```
extern int d=10;
```

---

<sup>5</sup> Nie jest to prawdą w przypadku łączenia funkcji napisanych w C z programami napisanymi w C++.

to zmienna d będzie jednocześnie deklarowana i definiowana; w efekcie powyższy przykład skompiluje się poprawnie. Ponieważ nie jest to jednak typowy przykład wykorzystania słowa kluczowego extern (aczkolwiek całkowicie zgodny ze standardem języka C), kompilator wygeneruje ostrzeżenie dotyczące takiej definicji:

```
.../float.c:13: warning: 'd' initialized and declared 'extern'
```



Wskazówka

Możliwość jednoczesnej deklaracji i definicji zmiennej typu extern jest dostępna wyłącznie w przypadku zmiennych globalnych. Zmiennych lokalnych nie da się w ten sposób definiować.

W praktyce extern używa się do deklarowania zmiennych globalnych, które mogą być używane pomiędzy różnymi jednostkami komplikacji. Najczęściej dotyczy to plików nagłówkowych, w których umieszczamy deklaracje takich zmiennych, natomiast w jednym pliku źródłowym musi znaleźć się definicja takiej zmiennej.



Wskazówka

Pamiętajmy, aby w plikach nagłówkowych nigdy nie definiować zmiennych. W przypadku wielokrotnego włączenia pliku nagłówkowego zawierającego definicje zmiennych kompilator zgłosi błąd informujący o próbie wielokrotnego zdefiniowania zmiennej.

Zasady wykorzystania extern wyglądają więc następująco:

- ♦ W plikach nagłówkowych deklaracje wszystkich zmiennych powinny być poprzedzone słowem extern. Jego pominięcie zazwyczaj nie prowadzi do błędów, ale jest naruszeniem standardu języka i nie powinno być stosowane.
- ♦ extern nie powinno być wykorzystywane w plikach źródłowych — zamiast tego plik powinien wczytywać odpowiedni nagłówek zawierający deklarację potrzebnej zmiennej (koniecznie poprzedzoną słowem extern).
- ♦ Zmienne extern powinny być definiowane wyłącznie w jednym pliku źródłowym, przy czym plik ten powinien włączać nagłówek z deklaracją danej zmiennej — umożliwia to wykrycie sytuacji, w której przypadkowo doszłoby do niezgodności typów.
- ♦ Prototypy funkcji nie muszą być poprzedzone słowem extern, chociaż nie jest to błędem. extern jest konieczne, jeśli dany nagłówek może być wykorzystany także w języku C++ — w takiej sytuacji extern określa sposób linkowania takich funkcji.



Wskazówka

O ile w języku C użycie extern bywa opcjonalne, to jest absolutnie wymagane w języku C++. Stąd też jeśli dany nagłówek może być używany w obu językach, należy nazwy zmiennych i funkcji poprzedzić słowem kluczowym extern.

Wynika to z faktu, że o ile w języku C deklaracja:

```
int z;  
int z;
```

jest poprawna, to w języku C++ jest niezgodna ze standardem i prowadzi do powstania błędu na etapie komplikacji.

Tak więc poprawna definicja i deklaracja globalnej zmiennej współdzielonej pomiędzy modułami programu powinna wyglądać następująco:

*Plik1.h:*

```
extern int zmienna_globalna; //Deklaracja zmiennej
```

*Plik1.c:*

```
#include "Plik1.h"
int zmienna_globalna=10; //Definicja zmiennej
void zwięks()
{
    zmienna_globalna++;
}
```

*Plik2.c:*

```
#include "Plik1.h"
void drukuj()
{
    printf("Wartość %d\n", zmienna_globalna);
}
```

W *Plik1.h* została zadeklarowana zmienna o nazwie `zmienna_globalna`. Deklaracja ta może zostać włączona do wszystkich plików, w których ta zmienna będzie wykorzystywana. Jej deklaracja następuje w *Plik1.c*; dzięki włączeniu w nim *Plik1.h* mamy pewność, że w każdej sytuacji deklaracja odpowiada definicji.

## Dyrektywa `inline`

Dyrektywą `inline` poświęcone zostanie w tej książce relatywnie więcej miejsca z powodu ważnych funkcji, jakie pełni w programowaniu mikrokontrolerów. Wywołując funkcję, kompilator musi wygenerować kod umożliwiający odpowiednie przygotowanie parametrów, a po zakończeniu wykonywania funkcji powrót z niej. Powoduje to pewną niewielką stratę czasu, która w przypadku krótkich funkcji, składających się z pojedynczych instrukcji, jest jednak znacząca — szczególnie w przypadku funkcji często wywoływanych — np. w pętli. Kompilator C dysponuje specjalną dyrektywą `inline`, informującą go, że kod funkcji ma być agresywnie zoptymalizowany — zazwyczaj oznacza to, że zamiast generować rozkazy wywołania funkcji, w miejscu wywołania zostanie umieszczony jej kod. Dzięki temu oszczędzamy czas, który trzeba by poświęcić na wywołanie funkcji i powrót. Poza tym kompilator ma możliwość przeprowadzenia dodatkowych operacji w zależności np. od argumentów wywołania funkcji `inline`. Oczywiście, każdy medal ma dwie strony. Mroczną stroną dyrektywy `inline` jest zazwyczaj wywoływanie przez nią wydłużenie kodu wynikowego — zamiast krótkiego rozkazu skoku do funkcji, w programie, w każdym miejscu, w którym jest wywoływana, umieszczany jest jej kompletny kod. Jednak w pewnych sytuacjach dzięki dyrektywie `inline` kod wynikowy może ulec skróceniu. Dzieje się tak w sytuacji, kiedy wywołanie funkcji (instrukcje skoku, powrotu i przygotowujące argumenty) są podobnej długości jak kod wynikowy powstający z komplikacji ciała funkcji. Czasami też dane funkcje są wykorzystywane w programie tylko raz — w takiej sytuacji, jeśli funkcja zostaje włączona

czona w miejscu wywołania, kod wynikowy będzie krótszy, niż jeśli zostaną wygenerowane instrukcje jawie wywołujące zdefiniowaną funkcję. Wykorzystanie dyrektywy `inline` ma szereg zalet:

- ♦ Zwiększenie wydajności programu dzięki brakowi konieczności przekazywania parametrów do funkcji przez stos.
- ♦ Większe bloki programu, w skład których wchodzą funkcje `inline`, mogą być lepiej optymalizowane; w efekcie tworzony jest krótszy i bardziej wydajny kod.
- ♦ Utworzone większe bloki podlegają lepszej optymalizacji także z powodu propagacji stałych — w przypadku niezmiennych parametrów kompilator umieszcza gotowy wynik funkcji, a nie kod, który go generuje.

Funkcja ma także pewne wady:

- ♦ Czasami prowadzi do wydłużenia kodu.
- ♦ Zwiększa się czas komplikacji programu (to w przypadku programów na mikrokontrolery może być praktycznie niezauważalne).

Stąd też dobrymi kandydatami do funkcji `inline` są:

- ♦ Funkcje wywoływanie często. W efekcie uzyskujemy znaczne przyśpieszenie wykonywania kodu.
- ♦ Funkcje krótkie — w tym przypadku kod wywołania funkcji może być dłuższy niż sama funkcja.
- ♦ Funkcje w miarę często wywoływanie, ale tylko z kilku miejsc w programie, preferencyjnie jednego miejsca — w takim przypadku zyskujemy na prędkości oraz skróceniu kodu wynikowego.

Pomimo umieszczenia słowa kluczowego `inline` w deklaracji funkcji czasami kompilator jest jednak zmuszony wygenerować jej kod jako osobny kod funkcji. Dzieje się tak m.in. w przypadku, kiedy:

- ♦ W programie pobieramy adres funkcji `inline` — w tej sytuacji musi ona istnieć również jako osobna funkcja.
- ♦ Nie włączono optymalizacji kodu, chyba że funkcja jest zdefiniowana z atrybutem `always_inline` — `__attribute__((always_inline))`.
- ♦ Funkcja jest rekurencyjna — w tym przypadku tylko czasami kompilator jest w stanie określić liczbę rekurencji i podjąć próbę rozwinięcia jej kodu.



Pamiętajmy, że dyrektywa `inline` jest tylko podpowiedź dla kompilatora. Kompilator w zależności od sytuacji może ją całkowicie zignorować.

Jak wspomniano, atrybut `__attribute__((always_inline))` powoduje, że niezależnie od stanu opcji optymalizacji kod funkcji jest zawsze włączany w miejscu jej wywołania; kompilator nigdy nie generuje instrukcji skoku do takiej funkcji. Dokładnym przeciwieństwem jest atrybut `noinline` — jego umieszczenie w nazwie funkcji powoduje,

że zawsze będzie ona wywoływana, jej kod nigdy nie zostanie włączony w miejscu wywołania funkcji. Atrybut ten ma szczególne znaczenie w przypadku wywołania kompilatora z opcją `-finline-functions` — powoduje ona, że wszystkie wystarczająco proste funkcje są traktowane tak, jakby w ich deklaracji występowało słowo kluczowe `inline`. To, jak duże funkcje traktowane są jako `inline`, możemy określić, przekazując kompilatorowi parametr `-finline-limit=X`, gdzie  $X$  określa maksymalną długość funkcji, która może być automatycznie traktowana jako `inline`. Po przekroczeniu podanej długości kod funkcji nie jest włączany w miejscu jej wywołania, lecz generowana jest instrukcja skoku do niej. Wartość  $X$  w praktyce należy dobrać eksperymentalnie — musi ona równoważyć ewentualne wydłużenie kodu wynikowego i zwiększenie prędkości wykonywania programu.

Istnieją trzy typy deklaracji funkcji `inline`. Każdy z nich różni się pewnymi niuansami.

## inline funkcja();

Najprostszą postacią deklaracji funkcji `inline` jest deklaracja:

```
inline funkcja() {ciało_funkcji;};
```

W takim przypadku deklaracja musi wystąpić jednocześnie z definicją. W ramach jednostki komplikacji (np. w pliku źródłowym) w której taka definicja występuje, w miejscu jej wywoływanego będzie umieszczony kod funkcji, a nie jej wywołanie. Jednak jeśli taka funkcja stanie się dostępna w innych jednostkach komplikacji, zostanie w nich wygenerowane wywołanie funkcji, tak jakby nie była ona zdefiniowana ze słowem kluczowym `inline`. Stąd też w ten sposób najczęściej deklaruje się funkcje w plikach nagłówkowych.

Plik `inline.h`:

```
inline void foo(int *z)
{
    (*z)++;
}
```



**Wskaźówka** W przypadku umieszczenia definicji funkcji w pliku nagłówkowym nie można pobrać jej adresu.

Próba pobrania adresu takiej funkcji kończy się błędem:

```
C:\ \ float.c:18: undefined reference to `foo'
```

Jeśli jednak w pliku `inline.h` umieścimy tylko deklarację funkcji:

```
void foo(int *z);
```

a w pliku `inline.c` jej definicję:

```
#include "inline.h"
inline void foo(int *z)
{
    (*z)++;
}
```

to w ramach pliku *include.c* funkcja *foo* będzie się zachowywała jak typowa funkcja *inline*, lecz kiedy zostanie wywołana z innych jednostek komplikacji, będzie się zachowywać tak, jakby była zadeklarowana bez słowa kluczowego *inline* (co jest logiczne, gdyż w innych modułach widoczna jest deklaracja funkcji bez słowa kluczowego *inline*); w efekcie będą generowane instrukcje wywołania funkcji. Dodatkowo w takiej sytuacji komplikator będzie zmuszony wygenerować kod funkcji *foo*. Skutkiem ubocznym tego będzie możliwość pobrania jej adresu. W większości sytuacji spowoduje to jednak niepotrzebne zwiększenie objętości kodu wynikowego.

### **static inline funkcja();**

Jak pamiętamy, funkcje statyczne są funkcjami o zasięgu lokalnym, to znaczy wykorzystywane są tylko w jednej jednostce komplikacji, np. w ramach jednego pliku źródłowego. Dzięki temu komplikator wie, że do danej funkcji nie może wystąpić odwołanie z innego modułu, co umożliwia przyjęcie pewnych bardziej agresywnych założeń co do optymalizacji. W przypadku funkcji zadeklarowanych z modyfikatorem *static inline* nie jest generowany kod funkcji — jej definicja zawsze jest włączana w miejscu jej wywołania. Działanie tej dyrektywy możemy sobie wyobrazić jako wklejanie definicji funkcji w miejscu, w którym jest ona wywoływana. Stąd też w przypadku dłuższych funkcji, często wywoływanych w różnych miejscach kodu, może to prowadzić do znacznego wzrostu objętości kodu wynikowego. Tak więc największy sens ma deklarowanie jako *static inline* krótkich, często wykonywanych funkcji. Ponieważ dodanie słowa kluczowego *static* powoduje, że dany symbol staje się symbolem lokalnym, deklaracja i definicja funkcji *static inline* muszą wystąpić albo w obrębie jednego pliku źródłowego, w którym taka funkcja jest wykorzystywana, albo w pliku nagłówkowym. Jeżeli taka funkcja zostanie zdefiniowana w innej jednostce komplikacji, to nie będzie dostępna w innych. Np. zdefiniowanie w pliku *inline.h* funkcji:

```
static inline void foo(int *z)
{
    (*z)++;
}
```

spowoduje, że może ona zostać wykorzystana w dowolnym pliku źródłowym, który ten plik zainkluduje. W każdym miejscu, w którym zostanie ona użyta, zostanie ona po prostu wstawiona. Funkcja taka nie będzie istniała jako osobna funkcja — chyba że w programie będziemy pobierali jej adres. W takiej sytuacji zostanie wygenerowany kodu funkcji *foo*. Jeśli z jakiegoś powodu chcemy, aby komplikator mimo wszystko wygenerował kod tak zadeklarowanej funkcji, musimy dodać opcję komplikacji *-fkeep-inline-functions*.

### **extern inline funkcja();**

Modyfikator *extern inline* ma przeciwnie znaczenie do *static inline*. Dyrektywa *extern* informuje komplikator, że dany symbol jest zdefiniowany w innym module. W przypadku funkcji *inline* wprowadza on jednak pewne zamieszanie. Dzięki użyciu *extern* funkcja może być zdefiniowana wielokrotnie. Prześledźmy przykład:

```
inline.h:
#include<stdio.h>

extern inline void test()
```

```

{ // Definicja ta jest używana wyłącznie jako funkcja inline
    printf("Jesteśmy w pliku inline.h\n");
}

main.c:
#include "inline.h"

int main()
{
    void (*pf)() = test;
    test();
    (*pf)();
}

source.c:
#include<stdio.h>

void test()
{
    printf("Jesteśmy w pliku source.c\n");
}

```

W pliku *inline.h* zdefiniowana została funkcja `test()`, jednak została ona zdefiniowana z modyfikatorem `extern`, który powoduje, że `test()` jest symbolem globalnym, który może być zredefiniowany w innym miejscu programu. Tak też uczyniliśmy, definiując ponownie funkcję `test()` w pliku *source.c*. Teraz w pliku *main.c* wywołujemy naszą funkcję na dwa sposoby — po prostu wywołujemy funkcję `test()`, a następnie pobieramy jej adres i wywołujemy przez wskaźnik `pf`. Pewnie ku naszemu niemałemu zaśkoczeniu stwierdzimy, że w obu przypadkach wywołane zostały różne wersje funkcji `test()` — na standardowe wyjście zostanie wysłany ciąg znaków:

```

Jesteśmy w pliku inline.h
Jesteśmy w pliku source.c

```

Widzimy, że tylko w przypadku bezpośredniego wywołania funkcja `test()` została włączona w generowany kod, zgodnie z działaniem `inline`. W przypadku pobrania adresu funkcji został zwrócony adres do funkcji zadeklarowanej w pliku *source.c*. Jest to zachowanie zgodne z tym, co wcześniej zostało napisane o funkcjach `inline` — ponieważ znajdują się one włączone w kod w miejscu wywołania, nie istnieją jako odrębne funkcje, nie można więc pobrać ich adresu. Stąd w przypadku próby pobrania adresu kompilator generuje informację o błędzie (w przypadku funkcji zadeklarowanej jako `static inline`) bądź też generuje osobny kod funkcji. Dlatego w powyższym przypadku używa definicji funkcji z pliku *source.c*. Takie działanie może prowadzić do trudnych do wykrycia błędów, więc konstrukcji `extern inline` należy używać z dużą ostrożnością.

## Modyfikator register

Użycie modyfikatora `register` jest co najmniej kontrowersyjne, zapewne pominięcie tego paragrafu może okazać się dobrym pomysłem. Modyfikator ten powoduje związanie danej zmiennej z konkretnym rejestrem procesora. Dzięki temu teoretycznie dostęp do takiej zmiennej będzie się odbywał najszybciej, jak to tylko możliwe. Teoretycznie

więc warto używać tego modyfikatora razem ze zmiennymi, do których często się odwołujemy lub do których odwołania są krytyczne czasowo. Niestety, w praktyce różnie z tym bywa, i to z kilku powodów. Po pierwsze, optymalizator kompilatora wykonuje świetną pracę, alokując dynamicznie rejesty w sposób optymalny lub prawie optymalny. Jest mała szansa, że człowiek wykona tę pracę lepiej. Drugim powodem jest ograniczona liczba rejestrów procesora, które można wykorzystać. Jeśli z tego skońzonego zbioru usuniemy kompilatorowi kolejne rejesty, może okazać się, że reszta generowanego kodu mocno na tym ucierpi. W efekcie co prawda dostęp do wybranej zmiennej może okazać się szybki, ale ta szybkość nie zrekompensuje znacznego spowolnienia i wydłużenia reszty programu. Trzeci powód dla nieużywania modyfikatora register wynika z faktu, że aby z niego bezpiecznie korzystać, cały program musi być przekompilowany. W praktyce uniemożliwia to korzystanie z prekompilowanych bibliotek dołączanych do programu, a taką biblioteką jest AVR-libc. Jak widać, użycie tego modyfikatora nie jest proste i w wielu przypadkach stwarza problemy. Jeśli jednak to nas nie zniechęca, poniżej zostaną pokazane przykłady użycia modyfikatora register.

Związanie zmiennej z wybranym rejestrem następuje po deklaracji:

```
register char regval asm("r2");
```

W powyższym przykładzie zmienna regval będzie przechowywana w rejestrze r2 procesora. Zmienne zajmujące 1 bajt pamięci możemy przechowywać w dowolnym rejestrze procesora. Pamiętać jednak należy, że rejesty r0 – r15 nie są w pełni „funkcjonalne” — to znaczy nie wszystkie operacje mogą być na nich bezpośrednio przeprowadzane. W efekcie użycie któregoś z tych rejestrów zmusi kompilator do wygenerowania dodatkowych instrukcji przesyłań, np. kod:

```
register char regval asm("r2");
regval=10;
```

zostanie skompilowany do kodu:

```
regval=10;
6c: 8a e0      ldi r24, 0xA ; 10
6e: 28 2e      mov r2, r24
```

podczas gdy zdefiniowanie zmiennej regval w rejestrze r16 wygeneruje prostszy kod:

```
regval=10;
6c: 0a e0      ldi r16, 0xA ; 10
```

Jest to możliwe, ponieważ do rejestrów r16 – r31 można bezpośrednio ładować stałą.



Jak widzimy, w przypadku wykorzystania modyfikatora register w celu optymalizacji kodu niezbędna jest dobra znajomość architektury danego procesora.

Jeśli w rejestrze chcemy umieścić zmienną, której reprezentacja zajmuje więcej niż jeden bajt, stosowna deklaracja wygląda podobnie. Różnica sprowadza się do tego, że jako rejestr musimy podać rejestr o parzystym numerze; kolejne rejesty o wyższych numerach zostaną użyte do przechowywania pozostałych bajtów zmiennej, np.:

```
register int regval asm("r16");
```

spowoduje zarezerwowanie do przechowywania zmiennej regval rejestrów r16 i r17. Jeśli podalibyśmy rejestr o nieparzystym numerze, kompilator wygeneruje błąd:

```
error: register specified for 'regval' isn't suitable for data type
```

Jest to znowu związane z architekturą mikrokontrolerów AVR.

Jak wspomniano wcześniej, przy korzystaniu z modyfikatora register cały program musi zostać przekompilowany, co prowadzi do problemów w przypadku użycia pre-kompilowanych bibliotek. W przypadku biblioteki AVR-libc stosunkowo bezpieczne jest użycie rejestrów r2 – r7. Dodatkowo, jeśli zmienna jest deklarowana lokalnie, a funkcja, w której deklaracja występuje, nie przyjmuje zbyt dużo argumentów, to można użyć rejestrów r8 – r15.



**Wskazówka** W żadnym przypadku nie możemy użyć rejestrów Y, nie zaleca się także rezerwacji rejestrów Z i X.

Jest to związane z tym, że gcc wykorzystuje rejesty indeksowe do adresowania złożonych struktur danych oraz generacji skoków pośrednich. Lokalne wykorzystanie rejestrów do przechowywania zmiennych jest bezpieczne, o ile w bloku, w którym zostały zadeklarowane, nie korzystamy z odwołań do zewnętrznych bibliotek. Jednak w tym przypadku użycie register jest tylko luźną wskazówką dla kompilatora. Np. w kodzie:

```
ISR(ADC_vect)
{
    register int adcval asm("r18");
    adcval=ADC;
    adcval++;
    regval=adcval;
}
```

pomimo nakazania użycia rejestrów r18:r19 do przechowywania wartości zmiennej adcval, wygenerowany kod tych rejestrów nie używa:

```
register int adcval asm("r18");
adcval=ADC;
64: 80 91 78 00 lds r24, 0x0078
68: 90 91 79 00 lds r25, 0x0079
adcval++;
regval=adcval;
6c: 8c 01        movw r16, r24
6e: 0f 5f        subi r16, 0xFF : 255
70: 1f 4f        sbci r17, 0xFF : 255
```

**Ze zmiennymi zadeklarowanymi z modyfikatorem register nie działa modyifikator volatile, co może prowadzić do poważnych błędów w programie.** Dodanie do modyfikatora register modyfikatora volatile generuje ostrzeżenie:

```
Warning: optimization may eliminate reads and/or writes to register variables
```

Jest to związane z tym, że wielokrotne zapisy lub odczyty z/do rejestrów z punktu widzenia optymalizatora nie mają sensu — inaczej sprawa wygląda w przypadku zmiennych

zależnych od jakiegoś zasobu sprzętowego. W przypadku rejestrów optymalizator wie, że dwa kolejne odczyty muszą zwrócić tę samą wartość, więc niepotrzebne odczyty/zapisy są eliminowane. Stąd też w następującym kodzie:

```
regval=10;  
regval=11;
```

zapis do zmiennej regval wartości 10 zostanie zignorowany:

```
regval=10;  
regval=11;  
b6: 0b e0      ldi r16, 0x0B : 11  
b8: 10 e0      ldi r17, 0x00 : 0
```

W przypadku gdyby regval było zwykłą zmienną zadeklarowaną z modyfikatorem volatile, sytuacja wyglądałaby inaczej:

```
regval=10;  
ba: 8a e0      ldi r24, 0x0A : 10  
bc: 90 e0      ldi r25, 0x00 : 0  
be: 90 93 01 01 sts 0x0101, r25  
c2: 80 93 00 01 sts 0x0100, r24  
regval=11;  
c6: 8b e0      ldi r24, 0x0B : 11  
c8: 90 e0      ldi r25, 0x00 : 0  
ca: 90 93 01 01 sts 0x0101, r25  
ce: 80 93 00 01 sts 0x0100, r24
```

Jak widać, w tym przypadku wygenerowany został kod realizujący dokładnie to, co napisaliśmy, a optymalizator nie usunął wcześniejszego wpisania wartości 10.



# Rozdział 4.

# Sekcje programu

Kompilator gcc, generując kod programu, umieszcza wyniki swojej pracy w tzw. sekcjach. Każda z predefiniowanych sekcji zawiera charakterystyczne dla siebie dane. Dzięki istnieniu sekcji linker może fragmenty programu odpowiednio „poskładać”, umieszczając je w odpowiednich rejonach pamięci. Sekcje mogą być także wykorzystywane przez programistę w celu umieszczenia fragmentów kodu/danych w ścisłe określonej lokalizacji. Umożliwia to lepszą gospodarkę pamięcią, a także określenie pewnych dodatkowych założeń co do lokalizacji danych — np. umieszczenie ich pod wskazanym adresem. Zmiana rozmieszczenia sekcji pamięci może też być niezbędna w przypadku wykorzystania zewnętrznej pamięci (w przypadku mikrokontrolerów dysponujących interfejsem XMEM — ang. *External Memory Interface*). Standardowo biblioteka AVR-libc definiuje następujące sekcje danych:

- ◆ `.data`
- ◆ `.bss` i jej specjalna podsekcja `.noinit`
- ◆ `.eeprom`

oraz sekcję danych/kodu programu `.text`. W skład tej sekcji wchodzą podsekcje `.init[0-9]` oraz `.fini[0-9]`. Istnieją także specjalne sekcje, których nazwa zaczyna się od `.debug_`, zawierające informacje dla debugera. Sekcje te nie generują danych, które znajdują się w finalnym pliku wykorzystywanym do zaprogramowania mikrokontrolera. Wykorzystywane są one wyłącznie podczas debugowania programu, dzięki czemu możliwe jest powiązanie zmiennych i kodu asemblerowego programu z kodem źródłowym. Modyfikując skrypt linkera, programista ma pełną kontrolę nad położeniem sekcji w pamięci mikrokontrolera. Zwykle taka kontrola nie jest nam potrzebna i możemy zadanie właściwego umieszczenia sekcji w pełni powierzyć linkerowi. Umieszczenie fragmentu programu lub zmiennych w poszczególnych sekcjach może być bardzo użyteczne, stąd w dalszej części szerzej zostanie przedstawione zastosowanie poszczególnych sekcji.

# Sekcje danych

Do sekcji danych trafiają wszystkie zmienne i stałe wykorzystane w programie. W zależności od sposobu deklaracji zmiennej może ona trafić do jednej z kilku sekcji: `.data`, `.bss`, `.eeprom`, `.text` lub do sekcji zdefiniowanej przez użytkownika.

## Sekcja `.text`

Do sekcji `.text` trafiają zmienne zadeklarowane ze specjalnym atrybutem PROGMEM, którego dokładna funkcja zostanie omówiona w rozdziale 8. — „Dostęp do pamięci FLASH”. Sekcja ta składa się z kilku podsekcji, z których istotna jest dla nas podsekcja `.progmem.data` — znajdują się w niej stałe zdefiniowane z atrybutem PROGMEM. Domyślnie sekcja ta znajduje się na początku sekcji `.text`, przed nią znajduje się tylko podsekcja `.vectors`, zawierająca wektory przerwań. Dzięki takiemu rozmieszczeniu dostęp do stałych zdefiniowanych w pamięci FLASH odbywa się szybciej — dane umieszczone są poniżej granicy 64 kB, co umożliwia realizację dostępu do nich za pomocą instrukcji asemblera LPM, a nie dłuższej i wolniejszej ELPML. Problem pojawia się w przypadku procesorów z więcej niż 64 kB pamięci FLASH, kiedy dane przekroczą granicę 64 kB. Szerzej zostanie to wyjaśnione w rozdziale poświęconym umieszczaniu danych w pamięci FLASH mikrokontrolera. Praktycznie nigdy nie zachodzi potrzeba, aby taki domyślny układ sekcji zmieniać. Wszystkie dane znajdujące się w sekcji `.text` są danymi tylko do odczytu, warto więc w tej sekcji umieszczać stałe wykorzystywane w programie, dzięki czemu nie będą one zajmowały cennej pamięci RAM.

## Sekcja `.data`

W sekcji tej znajdują się zmienne statyczne wykorzystane w programie. Zawartość takich zmiennych jest podczas inicjalizacji programu kopiwana do obszarów pamięci RAM przeznaczonych dla sekcji `.data`. Np. definicja zmiennej globalnej:

```
char txt[] = "To się znajdzie w sekcji .data";
```

spowoduje umieszczenie w sekcji `.data` zawartości zmiennej `txt[]`. Po komplikacji ilość wolnej pamięci RAM zmniejszy się stosownie do długości tej zmiennej. Ponieważ mikrokontrolery AVR mają rozdzieloną adresację różnych typów pamięci, co nie do końca wspiera kompilator gcc, twórcy wersji narzędzi na platformę AVR przyjęli pewne założenia co do oznaczania sekcji, które mają być umieszczone w pamięci RAM. **Adresy takich sekcji zaczynają się od wartości 0x800000**, a więc aby sekcja była umieszczona, począwszy od pierwszej komórki pamięci RAM (o adresie 0), adres takiej sekcji musi wynosić 0x800000; aby sekcja była umieszczona od komórki o adresie 0x00FF, sekcja musi się rozpoczynać od adresu 0x8000FF. **Konwencja ta dotyczy wszystkich sekcji, które docelowo mają znaleźć się w pamięci RAM.**

## Sekcja .bss

W sekcji tej umieszczone zostaną wszystkie niezainicjowane zmienne globalne oraz zmienne statyczne. Domyślnie zmienne w tej sekcji inicjalizowane są wartością 0 przez kod inicjalizacyjny programu. Stąd też zmienne:

```
char xbss[10];  
static char sbss[10];
```

zostaną umieszczone w sekcji *.bss*, a domyślna wartość poszczególnych pozycji tablic po uruchomieniu programu będzie wynosiła 0. Zauważmy, że zgodnie ze standardem języka C zmienne statyczne i globalne z pewnością będą zainicjalizowane wartością 0. Czasami jednak zależy nam, aby zmienna nie była zainicjalizowana podczas uruchomienia programu. W takim wypadku możemy umieścić ją w specjalnej podsekcji sekcji *.bss* o nazwie *.noinit*. Dokonujemy tego poprzez zdefiniowanie zmiennej z odpowiednim atrybutem, będącym nazwą sekcji, w której zmienna ta ma zostać umieszczona:

```
int noinitvar __attribute__ ((section (".noinit")));
```

Zmienne umieszczone w sekcji *.noinit* nie mogą być zainicjalizowane; próba skompilowania poniższego kodu:

```
int noinitvar __attribute__ ((section (".noinit"))) = 0xAA;
```

spowoduje wygenerowanie przez kompilator ostrzeżenia:

```
warning: only uninitialized variables can be placed in the .noinit section
```

Zmienna taka będzie jednak umieszczona poprawnie w sekcji *.noinit*, a więc jej wartość początkowa będzie nieokreślona (nie będzie zainicjalizowana podaną przez nas wartością 0x00AA).

## Sekcja .eeprom

Do sekcji tej trafiają zmienne, które docelowo mają znaleźć się w pamięci EEPROM mikrokontrolera. Zmienne takie deklarujemy z atrybutem określającym nazwę sekcji, w której zmienna ma się znaleźć, w tym przypadku *.eeprom*:

```
int eepromvar __attribute__ ((section (".eeprom"))) = 0xAA;
```

Powyższy kod spowoduje zadeklarowanie zmiennej o nazwie *eepromvar*, która zostanie umieszczona w pamięci EEPROM mikrokontrolera, i jej jednoczesne zainicjalizowanie wartością 0x00AA. Taką definicję możemy uprościć, wykorzystując zdefiniowaną w pliku nagłówkowym *<avr/eeprom.h>* definicję *EEMEM*:

```
int eepromvar1 EEMEM = 0xBB;
```

Powyższy kod spowoduje zadeklarowanie zmiennej *eepromvar1* w pamięci EEPROM mikrokontrolera i jej zainicjalizowanie wartością 0x00BB.



Wskazówka

Zmienne umieszczone w pamięci EEPROM nie muszą mieć zdefiniowanej wartości początkowej (choć w większości przypadków brak definicji nie ma wielkiego sensu). Musimy pamiętać, że aby takie zmienne miały zdefiniowaną przez nas wartość początkową, podczas programowania procesora musimy zaprogramować także pamięć EEPROM zawartością wygenerowanego pliku zawierającego wartości początkowe komórek pamięci EEPROM.

Niezaprogramowanie pamięci EEPROM spowoduje, że wartości początkowe wszystkich zmiennych umieszczonych w tej pamięci będą wynosiły 0xFF (domyślna wartość skasowanej pamięci EEPROM). Więcej na ten temat dowiesz się w rozdziale 7 — „Wbudowana pamięć EEPROM”.

Analogicznie do sekcji umieszczonych w pamięci RAM mikrokontrolera, aby linker poprawnie rozpoznał sekcje, których docelowa lokalizacja znajduje się w pamięci EEPROM mikrokontrolera, **adres takich sekcji musi być większy lub równy 0x00810000**. Tak więc sekcja zaczynająca się od komórki pamięci EEPROM o adresie 0 ma adres 0x00810000, od komórki pamięci EEPROM o adresie 0x100 ma adres 0x00810100.

## Sekcje zawierające kod programu

Cały kod programu zawarty jest w sekcji `.text`, w skład której wchodzi kilka podsekcji, z których do tej pory poznaliśmy podsekcję `.vectors` i podsekcję `.progmem.data`. Zaraz za nimi znajdują się dwie podsekcje (`.trampolines` i `.jumptables`). Podsekcja `.trampolines` wykorzystywana jest przez kompilator do generowania kodu umożliwiającego wywołanie funkcji umieszczonych w całej pamięci mikrokontrolerów, których wielkość pamięci FLASH jest większa niż 128 kB. Dla mikrokontrolerów o mniejszej ilości pamięci podsekcja ta zawsze jest pusta. Podsekcja `.jumptables` zawiera adresy funkcji w postaci tabeli skoków, co jest wykorzystywane do optymalizacji m.in. funkcji `switch/case`. Obie podsekcje dla programisty nie są zbyt interesujące. Interesujące mogą natomiast być sekcje `.init[0-9]` i w mniejszym stopniu `.fini[0-9]`.

### Podsekcje `.init[0-9]`

Kod umieszczony w tych podsekcjach wykonywany jest przed wykonaniem kodu umieszczonego w funkcji `main`. Stwarza to programistę możliwość umieszczenia kodu inicjalizacyjnego, który będzie wykonany na wczesnym etapie inicjalizacji programu. Kod z tych sekcji jest wykonywany zgodnie z numerem sekcji (0-9), najpierw jest więc wykonywany kod z sekcji `.init0`, następnie `.init1` itd. W praktyce wykorzystuje się to do zainicjalizowania interfejsu pamięci zewnętrznej (*XMEM*) i *Watchdoga*. Pisząc jednak kod umieszczony w tych sekcjach, należy sobie zdawać sprawę, że w zależności od numeru sekcji zmienne globalne mogą być niezainicjalizowane, nie powinniśmy polegać więc na ich zawartości. Ich modyfikacja nie odniesie skutku (z wyjątkiem zmiennych umieszczonych w sekcji `.noinit`), gdyż zostaną one nadpisane przez kod inicjalizacyjny programu. Nie powinniśmy się także odwoływać do innych funkcji ani

tworzyć zmiennych lokalnych, gdyż rejestr stosu mikrokontrolera nie jest zainicjalizowany, podobnie jak niezainicjalizowane będą rejesty procesora, w tym rejestr R1, który powinien zawsze zawierać 0. Stąd też kod w tych sekcjach powinien być krótki, bardziej skomplikowany kod powinien być napisany w asemblerze. Stos i rejestr R1 są inicjalizowane w kodzie umieszczonym w sekcji `.init2`, zmienne programu inicjalizowane są w sekcji `.init4`. Stąd też funkcje znajdujące się w sekcjach `.init5` i wyższych mogą bezpiecznie korzystać ze zmiennych oraz stosu. Biblioteka AVR-libc definiuje także specjalny symbol `_init()`. Jeśli użytkownik zdefiniuje funkcję o takiej nazwie, to zostanie ona wykonana przed jakimkolwiek kodem wstawionym przez linker — wektor RESET będzie wskazywał na jej adres. Funkcje możemy umieszczać w dowolnej sekcji za pomocą atrybutu `section`.

Poniższy przykład:

```
void init_XMEM() __attribute__ ((naked)) __attribute__ ((section (".init1")));
void init_XMEM()
{
    MCUCR |= _BV(SRE);
}
```

na mikrokontrolerze ATMega128 powoduje odblokowanie interfejsu XMEM, zanim jeszcze wykonane zostaną funkcje odpowiedzialne za inicjalizację zmiennych. Jest to możliwe dzięki zdefiniowaniu funkcji `init_XMEM` w sekcji `.init1`, dzięki czemu jej kod będzie wykonany przed jakimkolwiek kodem inicjalizacyjnym dodanym przez linker. Zauważmy, że powyższa funkcja jest dodatkowo zadeklarowana z atrybutem `naked`, dzięki czemu nie zostanie wygenerowany prolog i epilog funkcji, które w tym przypadku są zbędne. Jednakże tak zadeklarowana funkcja nie może zostać bezpośrednio wywołana w żadnym miejscu programu. Jeśli w danej sekcji zdefiniujemy więcej funkcji, to zostaną one umieszczone po kolej. Do powyższego przykładu możemy dodać funkcję umożliwiającą wczesną inicjalizację *Watchdoga*:

```
void init_WD() __attribute__ ((naked)) __attribute__ ((section (".init1")));
void init_WD()
{
    wdt_enable(WDP0);      //Włącz WD z 32K cyklami
}
```

Definicja funkcji `wdt_enable` znajduje się w pliku nagłówkowym `<avr\wdt.h>`. Zauważmy, że jej wywołanie jest bezpieczne, gdyż jej implementacja jest napisana całkowicie w asemblerze i nie zależy od wcześniejszej inicjalizacji żadnych struktur programu.

## Podsekcje `.fini[0-9]`

Podsekcje te są wykonywane po opuszczeniu przez kod programu funkcji `main` w kolejności `.fini9 – .fini0`. Domyslnie kod w podsekcji `.fini0` zawiera nieskończoną pętlę. Ponieważ programy pisane na mikrokontrolery zwykle nie kończą się, przydatność umieszczania własnego kodu w tych podsekcjach jest wątpliwa.

## Sekcje specjalne

Oprócz wcześniej omówionych sekcji w wynikowym pliku *elf* znajduje się wiele innych sekcji, w tym pewne sekcje specjalne, które nie są w żaden sposób powiązane z kodem programu, ale zawierają informacje przydatne dla programatora. Należą do nich sekcje *.lock*, *.fuse* oraz *.signature*. Zawierają one konfigurację tzw. *lockbitów*, *fusebitów* oraz sygnaturę procesora, dla którego został wygenerowany kod. Umożliwia to programatorowi weryfikację, czy kod odpowiada programowanemu procesorowi, a także dodatkową konfigurację jego lock- i fusebitów. Szerzej temat ten zostanie omówiony w rozdziale poświęconym bitom konfiguracyjnym procesora.

## Sekcje tworzone przez programistę

Oprócz predefiniowanych sekcji programista może tworzyć także swoje własne sekcje. Jest to stosunkowo rzadko wykorzystywana możliwość. Mechanizm ten wykorzystujemy do umieszczenia danych lub kodu programu pod ściśle określonymi adresami pamięci. Normalnie za rozmieszczenie sekcji w pamięci odpowiedzialny jest linker i programista nie powinien w ten proces ingerować. Wyjątki od tej ogólnej zasady są nieliczne — obejmują one głównie pisanie kodu tzw. *bootloadera*, który musi być umieszczony w pamięci mikrokontrolera pod ściśle określonym adresem, oraz zmiennych wspólnie dzielonych przez kod *bootloadera* i program główny. Aby umieścić kod lub dane pod określonym adresem, musimy najpierw umieścić je w nowej sekcji, co użyskujemy za pomocą znanego już nam atrybutu *section*: `_attribute_ ((section (.bootloader)))`. Funkcja zadeklarowana w poniższy sposób:

```
void boot(void) __attribute__ ((section (.bootloader)));
```

zostanie umieszczona w sekcji o nazwie *.bootloader*. W drugim etapie musimy poinformować linker, gdzie taką sekcję ma umieścić. Możemy w tym celu zmodyfikować skrypt wywołujący linker lub przekazać mu dodatkowy parametr:

```
-Wl,--section-start=.bootloader=0x1E000
```

Spowoduje to, że sekcja o nazwie *.bootloader* zostanie umieszczona w pamięci FLASH mikrokontrolera, począwszy od adresu 0x1E000. W podobny sposób możemy umieścić zmienne pod wskazanym adresem:

```
int xxx __attribute__ ((section (.XMEM)));
```

Spowoduje to, że zmienna *xxx* zostanie umieszczona w sekcji o nazwie *XMEM*. Wywołując linker, musimy podać, gdzie ta sekcja powinna się znaleźć, pamiętając, że adresy pamięci RAM zaczynają się od wartości 0x800000, co odpowiada komórce pamięci RAM o adresie 0.

# Umieszczanie sekcji pod wskazanym adresem

Powyżej pokazano, jak umieścić pod wskazanym adresem sekcje dodane przez programistę. W rzadkich przypadkach istnieje potrzeba relokacji sekcji standardowych, zdefiniowanych przez AVR-libc. Zdarza się tak np. w sytuacji, kiedy wykorzystujemy zewnętrzną pamięć dołączoną do mikrokontrolera. Relokacji dokonujemy również na etapie linkowania programu, przekazując do linkera polecenie `section-start` z nazwą relokowanej sekcji i jej nowym adresem początkowym:

```
-Wl,--section-start=.noinit=0x8000FF
```

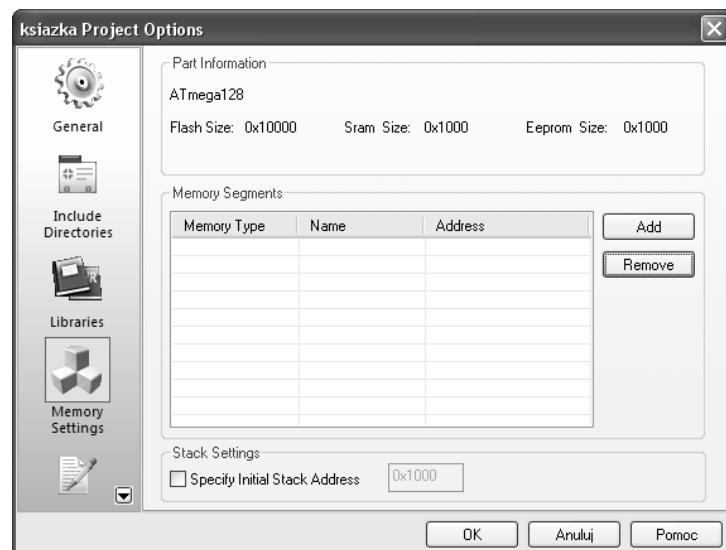
Powyższa linia spowoduje relokację sekcji `.noinit` pod adres 0xFF (pamiętamy, że do adresów w pamięci RAM musimy dodać offset wynoszący 0x800000, informujący linker, że sekcja znajduje się w pamięci RAM).

Relokację lub umieszczenie sekcji pod wskazanym adresem możemy uzyskać, modyfikując skrypt *Makefile* służący do wygenerowania naszego programu. W tym celu dodajemy do niego linię określającą dodatkowe parametry przekazywane do linkera po komplikacji programu:

```
LDFLAGS += -Wl,-section-start=FMEM=0x2000
```

Dodanie do *Makefile* powyższej linii spowoduje utworzenie nowej sekcji o nazwie `FMEM`, znajdującej się w pamięci FLASH, począwszy od adresu 0x2000. Jeśli korzystamy z programu AVR Studio, ten sam efekt możemy uzyskać za pomocą graficznych narzędzi konfiguracji projektu (rysunek 4.1). W tym celu wybieramy z menu opcje *Project/Configuration Options* i wybieramy zakładkę *Memory Settings*.

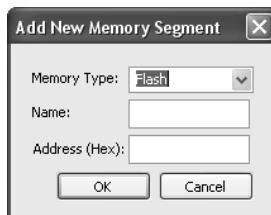
**Rysunek 4.1.**  
Okno konfiguracji sekcji programu w AVR Studio



Następnie klikamy przycisk *Add*, co spowoduje pojawienie się okienka konfiguracji wybranej sekcji (rysunek 4.2).

**Rysunek 4.2.**

Konfiguracja typu,  
nazwy i adresu  
początkowego nowej  
sekcji programu



Wybieramy typ pamięci, w której ma znaleźć się nowa sekcja (do wyboru mamy pamięć FLASH, SRAM i EEPROM), podajemy nazwę nowej sekcji (*Name*) i jej adres początkowy (*Address*) i klikamy przycisk *OK*.



Wskazówka

Adres sekcji umieszczonej w pamięci FLASH określa adres słowa, a nie bajtu. Dlatego jeżeli adres początkowy zostanie określony na np. 0x1000, sekcja zostanie umieszczona pod adresem słowa o tym adresie; adres bajtu będzie dwukrotnie większy, w tym przypadku wynosił będzie 0x2000.

Adresy zewnętrznej pamięci RAM zaczynają się tuż po adresach wewnętrznej pamięci RAM mikrokontrolera, nie mają więc żadnego specjalnego wyróżnika. **Korzystając z tego narzędzia, nie podajemy offsetów dla poszczególnych typów pamięci, gdyż zostaną one dodane automatycznie przez AVR Studio (offset 0x800000 dla pamięci RAM, 0x810000 dla pamięci EEPROM).**

## Rozdział 5.

# Kontrola rdzenia i zarządzanie poborem energii

Mikrokontrolery AVR dysponują wieloma mechanizmami kontroli egzekucji programu. Możemy wyłączać poszczególne bloki funkcyjne mikrokontrolera, zmniejszając w ten sposób pobór energii, zmieniać źródła zegara oraz jego częstotliwość, możemy także określać różne źródła mogące zresetować procesor, a także wywoływać reset po zajściu pewnych okoliczności. Wszystkie te mechanizmy zapewne nie są niezbędne i pisząc programy, w większości wypadków możemy o nich nie pamiętać, lecz ich właściwe wykorzystanie może otworzyć nowe perspektywy wykorzystania mikrokontrolera. Poniżej przedstawione zostaną mechanizmy umożliwiające sterowanie mikrokontrolerem.

## Źródła sygnału RESET

Dla poprawnego rozpoczęcia wykonywania programu niezwykle istotne jest przywrócenie domyślnych ustawień procesora i jego układów peryferyjnych. Zapewnia to sygnał reset procesora. Podanie sygnału reset powoduje przywrócenie domyślnych ustawień wszystkim podsystemom procesora, rejestrów I/O, a program jest wykonywany, począwszy od adresu 0.



W czasie resetu nie zmienia się zawartość pamięci SRAM oraz rejestrów R0 – R31 procesora — w efekcie część danych może pozostać niezmieniona.

Mikrokontrolery AVR dysponują co najmniej czterema źródłami sygnału RESET:

- ◆ sygnał generowany przez układ kontroli zasilania (ang. *Brown-out Reset Circuit*);
- ◆ sygnał generowany przez układ startu po włączeniu zasilania (ang. *Power-on Reset Circuit*);
- ◆ zewnętrzny sygnał RESET doprowadzony do pinu o tej samej nazwie;
- ◆ sygnał generowany przez układ tzw. *Watchdoga*.

Wystąpienie sygnału RESET wygenerowanego przez którykolwiek z powyższych systemów generuje sygnał RESET dla całego procesora. Przyczynę restartu procesora można odczytać z rejestru MCUSR (ang. *MCU Status Register*) — tabela 5.1.

**Tabela 5.1.** Znaczenie bitów rejestru MCUSR

Bit	Znaczenie
4 — JTRF	ang. <i>JTAG Reset Flag</i> — ustawiony wskazuje, że źródłem resetu był interfejs JTAG
3 — WDRF	ang. <i>Watchdog Reset Flag</i> — źródłem sygnału RESET był <i>Watchdog</i>
2 — BORF	ang. <i>Brown-out Reset Flag</i> — źródłem sygnału RESET był układ detekcji niskiego napięcia zasilającego
1 — EXTRF	ang. <i>External Reset Flag</i> — źródłem sygnału RESET był sygnał podany na końcówkę RESET procesora
0 — PORF	ang. <i>Power-on Reset Flag</i> — źródłem sygnału RESET był sygnał wygenerowany przez układ startu procesora

W rejestrze MCUSR odpowiednie bity są ustawione, jeśli opisywany przez nie podsysteem był źródłem sygnału RESET dla procesora. Bity te można wyzerować; automatycznie są one zerowane, gdy procesor resetuje układ PORF. Tak więc po odczytaniu zawartości rejestru MCUSR należy go jak najszybciej wyzerować.



**Wskazówka** W danym modelu mikrokontrolera nie wszystkie podsystemy muszą być zaimplementowane, np. w mikrokontrolerach, które nie posiadają interfejsu JTAG, bit JTRF zawsze będzie miał wartość 0.

## Power-on Reset

Układ ten odpowiada za resetowanie mikrokontrolera w sytuacji przywrócenia napięcia zasilającego. Kiedy napięcie przekroczy zależną od modelu procesora wartość, układ ten utrzymuje sygnał RESET w stanie aktywnym przez zaprogramowany przy pomocy fusebitów odcinek czasu. Daje to możliwość np. ustabilizowania się oscylatora i innych układów mikrokontrolera, a także osiągnięcia przez zasilanie napięć nominalnych. Nad tym sygnałem nie ma żadnej programowej kontroli.

## Zewnętrzny sygnał RESET

Procesory AVR mają specjalny pin oznaczony jako RESET — podanie na niego stanu niskiego aktywuje proces restartu mikrokontrolera. W procesorach ATTiny pin ten jest współdzielony z innymi podsystemami mikrokontrolera, lecz domyślnie są one wyłączone, a sygnał RESET ma priorytet. Możemy to zmienić, programując *fusebit* RSTDISBL. Po jego zaprogramowaniu pin RESET traci swoją funkcję, natomiast można go wykorzystać jako normalny pin *IO*.



W takiej sytuacji tracimy także możliwość programowania procesora w systemie (ISP). Przywrócić funkcję pinu RESET można wyłącznie przy pomocy programatora wysokącięciowego.

Czasami spotyka się układy, w których projektant jeden z pinów *IO* łączy z sygnałem RESET. W ten sposób, zmieniając stan takiego pinu na niski, można zresetować programowo procesor. W czasie resetu piny ustawiane są jako wejścia, znika więc sygnał sterujący sygnałem RESET i procesor może rozpocząć normalną pracę<sup>1</sup>.

## Brown-out Detector

Jest to układ monitorujący napięcie zasilające mikrokontroler. Jeśli napięcie to obniży się poniżej ustalonej wartości, układ ten generuje sygnał RESET i utrzymuje go do chwili, kiedy napięcie nie wróci do wartości prawidłowych. Próg zadziałania układu BOD ustawia się przy pomocy *fusebitów* BODLEVEL0-2.



Pamiętaj, że jeśli ustawisz minimalne napięcie zadziałania układu BOD powyżej nominalnego napięcia zasilającego, to mikrokontroler będzie znajdował się w stanie ciągłego resetu, nie będzie nawet możliwe jego zaprogramowanie — w tym zaprogramowanie nowych wartości *fusebitów* BODLEVEL.

Układ BOD warto włączać w każdej sytuacji. Jest on szczególnie istotny w sytuacji, kiedy procesor może być przejściowo zasilany niskim napięciem. Po obniżeniu napięcia poniżej dopuszczalnego procesor w jakimś stopniu ciągle działa — tyle że wykonywanie rozkazów może przebiegać nieprawidłowo. Co ważniejsze, w takich warunkach niemożliwy jest poprawny zapis do pamięci EEPROM. Może to doprowadzić do uszkodzenia jej zawartości, co jest często spotykany problemem w układach zasilanych z niestabilnych źródeł. Włączenie układu BOD praktycznie całkowicie eliminuje ten problem.

<sup>1</sup> Podobny efekt można uzyskać przy pomocy Watchdoga, co zostanie pokazane w dalszej części rozdziału.

## Układ Watchdog

Z punktu widzenia programisty pod wieloma względami jest to najciekawszy, a jednocześnie najbardziej skomplikowany podsystem mogący generować sygnał reset. Jest to niezależnie działający podsystem procesora, posiadający własny, niezależny zegar. Po jego skonfigurowaniu i uaktywnieniu działa jako licznik, po zliczeniu do określonej wartości generuje on sygnał RESET. Aby temu zapobiec, program okresowo, w odstępach nie większych niż określonych w konfiguracji *Watchdoga*, musi wykonać instrukcję WDR, resetującą licznik *Watchdoga*. Proces zerowania *Watchdoga* bywa z angielskiego nazywany „kopaniem psa” (*dog* to po angielsku pies, *kicking the dog*). Powiedzenie to jest zabawne (choćż niektórym może wydawać się brutalne), ale dobrze oddaje istotę działania tego układu. Atakujący pies wartowniczy (czyli nasz *Watchdog*) chce nas ugryźć (czyli zresetować procesor), a zapobiec temu możemy, kopiąc go, uniemożliwiając mu ugryzienie (czyli zresetowanie układu). Aby układ taki był skuteczny, musi być maksymalnie niezależny od programu i reszty procesora. Stąd też układ *Watchdoga* posiada własny zegar, którego częstotliwość zależna jest od modelu mikrokontrolera, oraz własny preskaler i rejesty kontrolne. Zmieniając wartość preskalera, można zmieniać wymagany okres, po jakim *Watchdog* zresetuje mikrokontroler.



O ile modyfikacja rejestru kontrolnego wymaga specjalnego sposobu dostępu, to resetowanie jest pojedynczą instrukcją asemblera.

W efekcie resetowanie *Watchdoga* w procesorach AVR jest niezbyt przemyślane — nawet błędnie działający program ma dużą szansę na cykliczne wykonywanie instrukcji resetującej licznik, w efekcie układ *Watchdoga* może nie zadziałać. Stąd też, aby zmniejszyć to ryzyko, należy ograniczyć liczbę wystąpień w kodzie instrukcji resetujących.

Z układu *Watchdoga* należy korzystać w każdym urządzeniu, jednak jego wykorzystanie staje się krytyczne w urządzeniach, które mają pracować bez nadzoru i interwencji człowieka. W takiej sytuacji nieprawidłowe działanie programu lub po prostu jego zawieszenie się trwale blokuje urządzenie. Jedną z możliwości wyjścia z takiej opresji jest automatyczny reset generowany przez układ *Watchdoga*.



AVR-libc udostępnia funkcje i makrodefinicje służące do obsługi *Watchdoga* w pliku nagłówkowym `<avr\wdt.h>`.

W starszych mikrokontrolerach AVR układ *Watchdoga* po zadziałaniu mógł jedynie być źródłem sygnału RESET dla procesora. W nowszych układ ten jest bardziej elastyczny i po zadziałaniu może także generować przerwanie lub przerwanie i sygnał RESET. Układ *Watchdoga* można odblokować programowo, może on być także zawsze włączony poprzez zaprogramowanie fusebitu WDTON. Po jego zaprogramowaniu układ *Watchdoga* zawsze jest włączony, a stan bitów WDE i WDIE jest bez znaczenia (są one przez procesor traktowane tak, jakby WDE miał wartość 1, a WDIE 0). W efekcie procesor, niezależnie od konfiguracji *Watchdoga*, po upływie określonego czasu wchodzi w stan RESET. W tym trybie nie jest możliwe wykorzystanie rozszerzonych funkcji układu *Watchdoga*.



Wskazówka

Jeśli planujemy wykorzystać rozszerzoną funkcjonalność układu, należy *fusebit WDTON* pozostawić niezaprogramowany.

W takiej sytuacji funkcje układu *Watchdoga* kontroluje rejestr *WDTCSR* (ang. *Watchdog Timer Control Register*). Jego bity *WDE* (ang. *Watchdog System Reset Enable*) i *WDIE* (ang. *Watchdog Interrupt Enable*) określają funkcję układu zgodnie z tabelą 5.2.

**Tabela 5.2. Funkcje układu Watchdoga**

<b>Bit WDE</b>	<b>Bit WDIE</b>	<b>Tryb i działanie po upływie zaprogramowanego czasu</b>
0	0	Tryb 1. Zatrzymany, brak działań
0	1	Tryb 2. Generowanie przerwania
1	0	Tryb 3. Generowanie sygnału RESET
1	1	Tryb 4. Generowanie przerwania i sygnału RESET

W trybie pierwszym układ *Watchdoga* nie działa; jest to tryb domyślny. Jeśli jawnie go nie zmienimy, układ *Watchdoga* pozostało zablokowany.

W trybie drugim układ *Watchdoga* po osiągnięciu zaprogramowanego limitu czasu generuje przerwanie o wektorze *WDT\_vect*<sup>2</sup>. Programista jest całkowicie odpowiedzialny za to, co z takim przerwaniem zrobi. Umożliwia to wykorzystanie *Watchdoga* jako zwykłego *timera* lub, co jest bardziej praktyczne, wybudzenie procesora z trybów uśpienia po określonym czasie. Czas, po jakim jest generowane przerwanie, zależy od ustawienia preskalera. Jego upłynięcie sygnalizowane jest poprzez ustawienie bitu *WDIF* (ang. *Watchdog Interrupt Flag*) rejestru *WDTCSR*. Bit ten jest kasowany automatycznie po wejściu w procedurę obsługi przerwania *Watchdoga*, można go także skasować programowo poprzez wpisanie do niego wartości jeden:

```
WDTCSR |= _BV(WDIF);
```

Tryb trzeci nie różni się niczym od sytuacji, w której zaprogramowany jest *fusebit WDTON*. Po upływie określonego czasu generowany jest sygnał *RESET* dla procesora.

Tryb czwarty łączy ze sobą dwa poprzednie tryby działania. Po pierwszym upływie czasu generowane jest przerwanie *Watchdoga*. W procedurze jego obsługi automatycznie zerowane są bity *WDIF* oraz *WDIE*. Powoduje to przejście *Watchdoga* w tryb 3, czyli generowanie sygnału *RESET*. W efekcie po ponownym upływie czasu *Watchdoga* generowane jest nie kolejne przerwanie, lecz sygnał *RESET*. Stan ten można zmienić, ustawiając w międzyczasie ponownie bit *WDIE*.



Wskazówka

Ustawienia tego bitu nie należy jednak wykonywać w procedurze obsługi przerwania *Watchdoga*. Doprowadziłoby to w sytuacji awaryjnej tylko do generowania przerwań, bez generowania sygnału *RESET*, co czyni układ *Watchdoga* bezużytecznym.

<sup>2</sup> W niektórych procesorach wektor ten nosi nazwę *WDT\_OVERFLOW\_vect* lub *WATCHDOG\_vect*.

## Zmiana stanu rejestru WDTCSR

Aby zapobiec przypadkowym zmianom stanu rejestru WDTCSR, kontrolującego układ *Watchdoga*, wszelkie wpisy do tego rejestru wymagają specjalnej sekwencji rozkazów. Aby zmiany zostały wprowadzone, wymagana jest następująca sekwencja rozkazów:

- ◆ W jednej instrukcji należy ustawić flagę WDCE (ang. *Watchdog change enable*) i bit WDE (nawet jeśli wcześniej jego wartość wynosiła 1).
- ◆ W ciągu 4 taktów zegara należy wpisać nową wartość do rejestru WDTCSR, przy czym tym razem bit WDCE musi mieć wartość 0.

Zmiana stanu bitu WDE wymaga wcześniejszego wyzerowania bitu WDRF rejestru MCUSR. W efekcie, jeśli *Watchdog* zostanie odblokowany, a następnie nastąpi reset procesora, układ *Watchdoga* po restarcie procesora pozostaje odblokowany, z najkrótszym możliwym czasem. Aby go zablokować, należy najpierw skasować bit WDRF. Stąd też inicjalizacja *Watchdoga* powinna być jedną z pierwszych rzeczy, jakie przeprowadza program po uruchomieniu. Taki tryb działania *Watchdoga* ma także pewną wadę — jeśli aplikacja nie korzysta z *Watchdoga*, a jej kod przypadkowo go uruchomi, np. na skutek błędnego działania, procesor będzie permanentnie resetowany. Z takiego stanu może go wyprowadzić wyłącznie wyłączenie zasilania. W efekcie, nawet jeśli aplikacja użytkownika nie korzysta z układu *Watchdoga*, w pierwszych instrukcjach powinna ten układ jawnie zablokować, wcześniej zerując flagę WDRF:

```
#include <stdint.h>
#include <avr/wdt.h>

uint8_t mcusr_mirror __attribute__ ((section (.noinit)));
void get_mcusr(void) __attribute__((naked)) __attribute__((section(".init3")));
void get_mcusr(void)
{
    mcusr_mirror=MCUSR;
    MCUSR=0;
    wdt_disable();
}
```

Powyższy kod jest wykonywany automatycznie w trakcie startu aplikacji. Powoduje on wyzerowanie rejestru MCUSR i zablokowanie układu *Watchdoga*. Wcześniej jednak zapisuje on wartość rejestru MCUSR do zmiennej mcusr\_mirror, która dzięki umieszczeniu w sekcji *.noinit* nie jest zamazywana w trakcie wykonywania dalszych procedur inicjalizacyjnych aplikacji. Powoduje to, że w dowolnym punkcie aplikacji można sprawdzić przyczynę zresetowania procesora poprzez testowanie wartości tej zmiennej, analogicznie do testowania zawartości rejestru MCUSR. Gdy aplikacja nie sprawdza przyczyny restartu, fragment odpowiedzialny za zachowanie kopii rejestru można pominąć.

Powyższa funkcja wykorzystuje w celu wyłączenia *Watchdoga* makrodefinicję `wdt_disable()`. Aby wyłączenie układu zakończyło się sukcesem, bit WDRF musi być wyzerowany, a fusebit WDTON nie może być zaprogramowany. Jeśli oba warunki nie będą spełnione, wyłączenie *Watchdoga* nie zakończy się sukcesem.

Drugą funkcją jest makrodefinicja włączająca *Watchdoga* `wdt_enable(wartość)`, gdzie *wartość* to jedna z predefiniowanych stałych `WDTO_15MS`, `WDTO_30MS`, `WDTO_60MS`, `WDTO_120MS`, `WDTO_250MS`, `WDTO_500MS`, `WDTO_1S`, `WDTO_2S`, `WDTO_4S`, `WDTO_8S` określających okres *Watchdoga*, wynoszący odpowiednio 15 ms, 30 ms, 60 ms, 120 ms, 250 ms, 500 ms, 1 s, 2 s, 4 s i 8 s. **Należy pamiętać, że nie wszystkie stałe dostępne są dla wszystkich procesorów.**



Wskazówka

Na czas realizacji makr `wdt_enable` i `wdt_disable` automatycznie blokowane są przerwania.

Makrodefinicja `wdt_enable` nie umożliwia konfigurowania innych funkcji układu *Watchdoga*, w tym trybu jego działania. Jeśli zachodzi konieczność zmiany trybu jego działania, należy tego dokonać, bezpośrednio modyfikując rejestr `WDTCSR`. Dla przykładu, chcąc wykorzystać *Watchdog* w trybie generowania przerwań, można posłużyć się następującym kodem:

```
ISR(WDT_vect)
{
    //Obsługa przerwania Watchdoga
}
void init_WDT()
{
    wdt_reset();
    WDTCSR=_BV(WDCE) | _BV(WDE);
    WDTCSR=_BV(WDIE);
}
```

Ostatnią makrodefinicją zdefiniowaną w pliku `wdt.h` jest polecenie zerujące licznik *Watchdoga* `wdt_reset()`. Polecenie to jest komplikowane do pojedynczej instrukcji asemblera `wdr`. Instrukcję `wdt_reset()` należy umieścić przed zmianą preskalera *Watchdoga*, w przeciwnym przypadku, szczególnie zmniejszając wartość preskalera, można doprowadzić do natychmiastowego zadziałania układu i resetu procesora.

Układ *Watchdoga* bywa wykorzystywany do celowego zerowania procesora. Poniżej kod:

```
wdt_enable(WDTO_15MS);
cli();
while(1);
```

powoduje odblokowanie układu *Watchdoga*, z najmniejszym możliwym czasem oczekiwania, wyłączenie przerwań i zapłateienie w pustej pętli `while`. Procesor pozostaje w takim stanie aż do zadziałania *Watchdoga*, następuje jego zresetowanie i rozpoczęcie wykonywania programu od początku. Technikę tę często wykorzystuje się do przywracania procesora i jego periferii do stanu takiego jak po włączeniu zasilania, np. podczas końca aplikacji *bootloadera*.



Wskazówka

Ze względu na to, że *Watchdog* działa niezależnie od reszty programu i procesora, na czas debugowania programu warto układ ten zablokować.

## Resetowanie i okres Watchdoga

Konfiguracja *Watchdoga* oraz punkty programu, w których powinien on być resetowany, są przedmiotem ciągłych sporów i w dużej mierze zalecenia zależą od indywidualnych preferencji i praktyk programistycznych. Okres *Watchdoga* musi być co najmniej tak długi jak czas wykonywania najdłużej wykonywanej funkcji w programie.



Istotne jest, aby obliczając maksymalny czas wykonywania danej ścieżki programu, uwzględnić także możliwe przerwania, które ten czas wydłużają.

Nieuwzględnienie czasu, jaki mogą zajmować przerwania, szczególnie w sytuacji, kiedy pojawiają się one z niezbyt ściśle określona częstotliwością, może doprowadzić do resetowania programu w losowych okolicznościach.

Jeśli taki czas wykonania funkcji staje się bardzo długi, możemy taką funkcję podzielić na kilka krótszych albo w jej ciele umieścić instrukcje resetujące *Watchdoga*. I tu pojawiają się pewne rozbieżności. Tradycyjaliści uważają, że w programie resetowanie *Watchdoga* powinno następować wyłącznie w jednym miejscu. Programiści nieco bardziej liberalni dopuszczają pewne nieliczne wyjątki od tej zasady. Z pewnością można powiedzieć, że im więcej instrukcji resetujących *Watchdoga* w programie, tym bardziej pogarsza się jego zdolność do wykrywania i w efekcie resetowania procesora w sytuacji, w której aplikacja nie funkcjonuje prawidłowo. **Drugą pewną rzeczą jest absolutny zakaz używania instrukcji resetującej Watchdog w jakiejkolwiek procedurze obsługi przerwania.** Procedury te są wywoływanie niezależnie od programu głównego, w efekcie jego nieprawidłowe działanie ma mały wpływ na procedury obsługi przerwań.

Powstaje więc pytanie, gdzie takie instrukcje umieścić? Dobrym miejscem może być pętla główna programu, jednak pod warunkiem, że wszystkie wywoływane z niej funkcje wracają do tej pętli w sensownie krótkim czasie. Jeśli tak nie jest, należy zmodyfikować program, co zwykle jest zalecanym rozwiązaniem. Alternatywą jest umieszczenie instrukcji resetujących w takich długo się wykonujących funkcjach. Pamiętamy jednak, że rozwiązanie to zmniejsza skuteczność *Watchdoga*.

## Zarządzanie poborem energii

Większość mikrokontrolerów z rodziny AVR umożliwia zarządzanie poborem energii dzięki umożliwieniu przejścia w tryby, w których tylko część podsystemów mikrokontrolera jest aktywna, a pozostałe są wyłączone. Dzięki temu w sytuacji, kiedy procesor jest bezczynny, można znacznie zredukować jego pobór energii.



Wszystkie definicje związane z zarządzaniem poborem energii znajdują się w plikach nagłówkowych `<avr\sleep.h>` i `<avr\power.h>`.

W pliku nagłówkowym `<avr\sleep.h>` znajdują się funkcje związane z różnymi stanami uśpienia procesora. Oprócz możliwości usypiania niektóre mikrokontrolery z rodziną AVR dysponują specjalnymi rejestrami umożliwiającymi włączanie i wyłączanie w zależności od potrzeb poszczególnych podsystemów procesora. Definicje związane z tymi funkcjami znajdują się w pliku nagłówkowym `<avr\power.h>`.

## Usypianie procesora

Usypianie procesora w sposób znaczący redukuje pobór prądu. Mikrokontrolery AVR dysponują różnymi trybami uśpienia, które wybiera się przy pomocy bitów SM0 – SM2 rejestru SMCR (ang. *Sleep Mode Control Register*). W zależności od typu procesora nie wszystkie tryby uśpienia są dostępne. Aby wejść w stan uśpienia, oprócz konfiguracji bitów SM0 – SM2, należy ustawić bit SE (ang. *Sleep Enable*) rejestru SMCR. Po jego ustaleniu wykonanie instrukcji `sleep` uaktywnia wybrany tryb.



Wskazówka

Procesor ze stanu uśpienia może zostać wybudzony wyłącznie przez przerwanie. Jeśli przerwania zostaną zablokowane, jedyną możliwością wybudzenia jest podanie sygnału RESET.

Po wykonaniu procedury obsługi przerwania, która wybudziła procesor, program jest kontynuowany od instrukcji następnej po instrukcji `sleep`. Biblioteka AVR-libc wspiera funkcje związane z usypianiem mikrokontrolera, dostarczając wygodnych w użyciu makrodefinicji, uniezależniających programistę od konkretnych sposobów realizacji usypiania w danym modelu procesora.

Pierwszym etapem konfiguracji jest wybór pożądanego trybu uśpienia. W tym celu należy posłużyć się makrodefinicją `set_sleep_mode(tryb)`, gdzie `tryb` określa wybrany tryb uśpienia — tabela 5.3.

Po wybraniu odpowiedniego trybu procesor można wprowadzić w uśpienie przy pomocy makrodefinicji `sleep_mode()`. Powoduje ona odblokowanie bitu SE umożliwiającego wejście w tryb uśpienia, uśpienie procesora, a po wybudzeniu zablokowanie bitu SE.

Zamiast makra `sleep_mode()` można bezpośrednio sterować procesem usypiania procesora przy pomocy makr `sleep_enable()` i `sleep_disable()`, umożliwiających odpowiednio odblokowanie lub zablokowanie możliwości wchodzenia w tryb uśpienia, oraz makra `sleep_cpu()`, realizującego proces uśpienia procesora.

## Wyłączanie układu BOD

Niektóre procesory AVR oferują możliwość wyłączenia w trybie uśpienia układu BOD. W takim przypadku dostępna jest makrodefinicja `sleep_bod_disable()`, która wyłącza ten układ. Aby jednak układ BOD został całkowicie wyłączony, w określonym czasie procesor musi wejść w stan uśpienia. Przypadkowe przerwanie pomiędzy wyłączeniem układu BOD a uśpieniem procesora uniemożliwiłoby wyłączenie układu BOD. Stąd też wymagana jest specyficzna sekwencja instrukcji:

**Tabela 5.3.** Tryby uśpienia obsługiwane przez mikrokontrolery AVR

Tryb uśpienia	Opis
SLEEP_MODE_IDLE	Najprostszym trybem jest SLEEP_MODE_IDLE. Wszystkie układy peryferyjne działają normalnie, wyłączany jest tylko rdzeń procesora (brak generowania sygnału taktowania CPU i FLASH).
SLEEP_MODE_PWR_DOWN	W tym trybie wyłączany jest zewnętrzny oscylator i większość podsystemów procesora. Działa tylko BOD, Watchdog, TWI, możliwe są także przerwania zewnętrzne (INT). W przypadku przerwań wyzwalanych zboczem sygnał musi trwać odpowiednio długo.
SLEEP_MODE_PWR_SAVE	W tym trybie działają tylko timer'y. Źródło zegara, z którego nie korzysta dany timer, też jest wyłączane.
SLEEP_MODE_ADC	Wszystkie podsystemy działają normalnie, wyłączany jest tylko rdzeń procesora, pamięć FLASH i układy portów IO, co zmniejsza zakłócenia, ułatwiając pomiar ADC. Po wejściu w ten tryb automatycznie ustawiana jest flaga ADSC rejestru ADCSRA (rozpoczynany jest pomiar ADC).
SLEEP_MODE_STANDBY	W przypadku wykorzystania zewnętrznego oscylatora tryb ten różni się od trybu SLEEP_MODE_POWER_DOWN tylko tym, że oscylator pozostaje włączony, co przyśpiesza wybudzenie procesora (tylko 6 cykli zegara).
SLEEP_MODE_EXT_STANDBY	Tryb podobny do trybu SLEEP_MODE_PWR_SAVE, z tym że zewnętrzny oscylator pozostaje włączony, co przyśpiesza wybudzenie procesora.

```
set_sleep_mode(tryb);
cli();
sleep_enable();
sleep_bod_disable();
sei();
sleep_cpu();
sleep_disable();
```

Pomiędzy zablokowaniem układu BOD a wejściem w uśpienie przerwania są blokowane, dzięki czemu instrukcja sleep\_cpu() wykonana się w wymaganym limicie czasowym. Instrukcja sei(), powodująca odblokowanie przerwań, odblokowuje je po wykonaniu przez procesor następnej instrukcji, czyli instrukcji wywołującej uśpienie. Dzięki temu od razu po wejściu w stan uśpienia procesor będzie mógł zostać wybudzony.

## Wyłączanie podsystemów procesora

Nowsze procesory AVR wyposażone są w rejestr PRR (ang. *Power Reduction Register*), którego poszczególne bity kontrolują taktowanie bloków funkcyjnych procesora. Dzięki temu nieużywane części można selektywnie wyłączyć, oszczędzając w ten sposób energię. Oszczędzanie energii odbywa się poprzez wyłączenie zegara taktującego wybrany blok, co powoduje, że stan danego bloku zostaje „zamrożony”. Ponieważ stan bloku nie zmienia się w momencie wyłączenia, po ponownym włączeniu zegara taktującego dany podsystem powraca do stanu, w jakim go „zamrożono”. Ma to pewną wadę — zajęte przez dany podsystem linie IO lub interfejsy nie ulegają zmianie, stąd też przed wyłączeniem danego układu należy wyłączyć obsługiwane przez niego funkcje.



Wskaźówka

Po wyłączeniu danego podsystemu związane z nim rejestryst IO stają się niedostępne i nie można się do nich odwoływać.

Biblioteka AVR-libc w pliku nagłówkowym `<avr\power.h>` definiuje makra umożliwiające włączanie (makra z sufiksem `_enable()`) lub wyłączanie (makra z sufiksem `_disable()`) wybranego podsystemu. Dzięki temu nie ma potrzeby bezpośredniego operowania na rejestrze PRR, co z kolei zwiększa kompatybilność programu z różnymi procesorami. Wybrane makrodefinicje umożliwiające określenie stanu podsystemów procesora pokazano w tabeli 5.4.

**Tabela 5.4.** Wybrane makrodefinicje określające stan podsystemów procesora. Do podanego rdzenia należy dodać sufiks `_enable()` lub `_disable()`, w zależności od tego, czy chcemy dany podsystem odblokować, czy wyłączyć

Makrodefinicja	Znaczenie
<code>power_all</code>	Włącza/wyłącza wszystkie podsystemy
<code>power_adc</code>	Włącza/wyłącza ADC
<code>power_spi</code>	Włącza/wyłącza interfejs SPI
<code>power_timerX</code>	Włącza/wyłącza timer o numerze X
<code>power_twi</code>	Włącza/wyłącza interfejs TWI
<code>power_usartX</code>	Włącza/wyłącza interfejs USART o numerze X

## Preskaler zegara

Ponieważ ilość zużywanej przez układy cyfrowe energii zależy od częstotliwości ich przełączania, zmniejszając tę częstotliwość, można znacznie zredukować zapotrzebowanie na energię. Stąd też w nowszych procesorach AVR występuje rejestr CLKPR (ang. *Clock Prescale Register*), umożliwiający podzielenie zegara taktującego CPU przez wybraną wartość. W rezultacie rdzeń procesora taktowany jest wolniej, a co za tym idzie, zużywane jest mniej energii. Zmian preskalera możemy dokonywać dynamicznie, w zależności od chwilowego zapotrzebowania na moc obliczeniową procesora. W procesorach obsługujących taką możliwość po włączeniu do programu pliku `<avr\power.h>` dostępne są makrodefinicje `clock_prescale_set(x)` i `clock_prescale_get()`, odpowiednio ustawiające i odczytujące wartość preskalera. Jako parametru x można użyć jednej z następujących wartości: 1, 2, 4, 8, 16, 32, 64, 128 lub 256. Określają one, przez ile należy podzielić zegar CLK, aby otrzymać zegar taktujący rdzeń procesora. Jak widać, maksymalne zwolnienie zegara wynosi 256 razy. Zamiast wartości liczbowych możemy używać także predefiniowanych stałych `clock_div_1` do `clock_div_256`.



Wskaźówka

Początkowy stan rejestrów CLKPR zależy od fusebitu CKDIV8. Jeśli jest on zaprogramowany, to początkowy stan rejestrów CLKPR odpowiada podziałowi zegara przez 8, jeśli CKDIV8 nie jest zaprogramowany, CLKPR zawiera wartość 0, powodującą brak podziału zegara taktującego.

## Inne sposoby minimalizowania poboru energii

Oprócz powyższych możliwości sterowania pracą procesora należy także pamiętać o innych możliwościach oszczędzania energii. Bez poprawnego skonfigurowania podsystemów wymienionych w tym punkcie oszczędności wynikające z wprowadzenia procesora w tryb uśpienia mogą być niewielkie.

### Porty IO

Jest niezwykle istotne, aby stan wszystkich portów *IO* był ustalony. Bezpiecznie jest nieużywane porty skonfigurować jako wejścia, przy czym nie mogą one pozostać niepodłączone. Ich stan (1 lub 0) należy wymusić, przyłączając je do masy lub zasilania, najlepiej przez rezystor, co uchroni nas przez zwarciem w sytuacji, w której port przypadkowo zostały przez program przestawione na wyjście. Pozostawienie niepodłączonych wejść powoduje, że potencjał na nich „pływą”, co z kolei generuje ciągłe przełączanie ich stanu i zwiększyony pobór prądu. Niepodłączone wejścia są także punktem wejścia zakłóceń. Alternatywnie można niewykorzystane wejścia zostawić niepodłączone, lecz w takiej sytuacji ich stan należy wymusić, włączając wewnętrzny rezystor podciągający. W trybach uśpienia bufory cyfrowe związane z pinami *IO* są wyłączone, z wyjątkiem pinów używanych do wybudzenia procesora. Ważną rzeczą jest też odpowiednia konfiguracja pinów, do których podłączony jest sygnał analogowy. Jeśli część cyfrowa takiego portu nie jest wykorzystywana, to port taki należy wyłączyć poprzez odpowiednią konfigurację rejestru DIDR<sub>x</sub> (nie we wszystkich procesorach ten rejestr występuje). **Sygnal analogowy o potencjalnie zbliżonym do  $V_{cc}/2$  powoduje stale przewodzenie obu tranzystorów bufora wejściowego i znaczne zwiększenie pobieranego prądu.**

### Przetwornik ADC

We wszystkich trybach uśpienia przetwornik ADC jest włączony, niepotrzebnie pobierając prąd. Przed wejściem w uśpienie należy go wyłączyć, zerując bit ADEN rejestru ADCSRA.

### Wewnętrzne napięcie referencyjne

Źródło wewnętrznego napięcia referencyjnego jest wykorzystywane przez ADC, komparator analogowy i układ kontroli zasilania BOD. Jeśli te podsystemy są wyłączone, to można także wyłączyć źródło napięcia referencyjnego przez wyzerowanie bitów REFS0 i REFS1 rejestru ADMUX. Należy pamiętać, że po ponownym włączeniu napięcia referencyjnego należy odczekać pewien czas na jego ustabilizowanie.

### Komparator analogowy

Jeśli go nie używamy, powinien zostać wyłączony (jest to jego stan domyślny). W większości trybów uśpienia komparator jest wyłączany automatycznie, wyjątkiem jest tryb *Idle* (*SLEEP\_MODE\_IDLE*) i tryb redukcji szumów (*SLEEP\_MODE\_ADC*). Jeśli nie wykorzystujemy komparatora, to przed wejściem w te tryby należy go wyłączyć. Jego wyłączenie

jest także istotne w sytuacji, w której wykorzystuje on wewnętrzne napięcie referencyjne. Bez jego wcześniejszego wyłączenia nie jest możliwe wyłączenie napięcia referencyjnego.

## Układ kontroli zasilania BOD

Jeśli układ BOD jest włączony (jest to zalecane), to automatycznie włączone jest także źródło napięcia referencyjnego. Układ BOD pozostaje włączony we wszystkich trybach uśpienia. Wyłączyć można go tylko poprzez konfigurację fusebitów BODLEVEL.

## Układ Watchdoga

Podobnie jak BOD, jeśli jest włączony, to pozostaje włączony we wszystkich trybach uśpienia. Można go wyłączyć albo poprzez konfigurację fusebitu WDTON, albo programowo. Wydaje się, że wyłączenie Watchdoga w trybie uśpienia jest bezpieczne, pod warunkiem że zaraz po wyjściu z tego trybu Watchdog jest ponownie włączany. **Niemniej, podobnie jak w przypadku układu BOD, zawsze należy dokładnie rozważyć potencjalne korzyści (zmniejszenie poboru prądu) i straty (brak detekcji不稳定nego zasilania i brak ochrony Watchdoga).**

## System uruchomieniowy procesora

Układ odpowiedzialny za możliwość debugowania systemu (ang. *On-chip debug system*) w trakcie normalnej pracy jest niepotrzebny. Możemy go zablokować poprzez konfigurację fusebitu DWEN. Warto to uczynić, gdyż system ten jest włączony niezależnie od trybu uśpienia procesora i wymusza włączenie taktowania rdzenia, co dodatkowo zwiększa pobór energii.



## Rozdział 6.

# **Dynamiczna alokacja pamięci**

Dynamiczną alokację pamięci nazywamy proces przydzielania pamięci dla struktur danych w trakcie wykonywania programu. Znaczy to, że pamięć dla danych nie jest przydzielana na etapie kompilacji programu, ale zmienia się w miarę potrzeb wykonywanego programu. Ma to kilka zalet, m.in. umożliwia lepsze wykorzystanie pamięci pomiędzy różnymi podprogramami. Wyobraźmy sobie, że mamy dwa podprogramy, które czasowo wymagają do swojego działania stosunkowo dużych bloków pamięci, ale po zakończeniu pamięć przydzielona dla tych danych nie jest wykorzystywana. Dzięki alokacji dynamicznej pamięć zajmowana jest tylko wtedy, kiedy jest potrzebna. Alokacja dynamiczna umożliwia także implementację struktur danych o dynamicznych rozmiarach. Przykładem może być magistrala 1-Wire. Na tej magistrali każde urządzenie ma swój unikalny, 8-bajtowy identyfikator. Ponieważ do magistrali może być przyłączona nieokreślona liczba urządzeń, więc zastosowanie tablicy do przechowywania identyfikatorów jest rozwiązaniem mało efektywnym — albo zarezerwujemy zbyt dużo miejsca, w efekcie marnując pamięć, albo tablica będzie zbyt mała, aby pomieścić wszystkie identyfikatory. Dynamiczna alokacja umożliwia rozwiązanie tego problemu poprzez łatwą implementację tablic dynamicznych i list. Jednak oprócz zalet ma ona także wady:

- ◆ Fragmentacja pamięci — jest to problem związany ze sposobem, w jaki działa alokator. W efekcie możemy dysponować wystarczającą ilością wolnej pamięci, która jednak ze względu na podzielenie na małe bloki nie będzie możliwa do wykorzystania.
- ◆ Alokator, przydzielając pamięć, musi zapisywać pewne dane dodatkowe, co powoduje, że ilość alokowanej pamięci jest ciut większa niż ilość żądana przez program. W przypadku kiedy mamy zaalokowanych dużo małych obszarów pamięci, informacje dla alokatora mogą zajmować sporą jej część.
- ◆ Wykorzystanie pamięci dynamicznej stwarza możliwość doprowadzenia do sytuacji, w której występują tzw. wycieki pamięci, spowodowane niezwolnieniem wcześniej przydzielonego bloku z jednoczesną utratą referencji do niego.

- ◆ Dynamiczny przydział pamięci jest wolniejszy. Alokator musi znaleźć odpowiedni blok pamięci, wydzielić z niego fragment i zwrócić aplikacji. W przypadku zmiennych sprawa jest prostsza: albo pamięć dla nich jest z góry przydzielana, albo przydział wymaga tylko prostej zmiany wartości wskaźnika stosu (w przypadku zmiennych lokalnych).



Wskazówka

Szczególnie z powodu możliwości fragmentacji pamięci i w efekcie niemożności przydzielenia żądanego przez aplikację bloku, niektórzy programiści uważają, że dynamiczna alokacja pamięci nie powinna być stosowana w systemach z małą ilością pamięci RAM, a więc np. w mikrokontrolerach AVR.

Trudno jednak zgodzić się z takim radykalnym poglądem. Korzyści płynące ze stosowania struktur dynamicznych są ogromne, a przy uważnej alokacji nie powstaje także problem fragmentacji. Niewątpliwie pewnym problemem przy alokacji dynamicznej jest konieczność obsłużenia zdarzenia polegającego na niemożności przydzielenia żądanego bloku pamięci RAM. Jednak nie jest to wielkim problemem — aplikacja na mikrokontroler powinna być napisana tak, aby do sytuacji braku pamięci nigdy nie doszło. Wykrywanie braku pamięci na etapie wykonywania programu nie ma wielkiego sensu, gdyż w przypadku braku wystarczającej jej ilości i tak nie za bardzo wiadomo, co aplikacja powinna zrobić. Stąd programy wykorzystujące dynamiczną alokację pamięci wymagają nieco większej uwagi na etapie debugowania kodu oraz określenia pewnego marginesu ilości wolnej pamięci, który zapewnia bezproblemową pracę aplikacji. Z pewnością prawdą jest, że zmienne, które ze swojej natury powinny być alokowane statycznie, nie powinny w sposób sztuczny być zamieniane na zmienne dynamiczne.



Wskazówka

Zauważmy, że na etapie komplikacji i konsolidacji nie ma możliwości łatwego określenia, ile pamięci jest zajęte przez dynamiczne struktury danych, zmienne lokalne i stos. Na tym etapie wyświetlana jest tylko informacja o pamięci zajętej przez zmienne globalne i statyczne, dla których pamięć alokowana jest na etapie konsolidacji programu.

Prototypy wszystkich przedstawionych poniżej funkcji znajdują się w pliku nagłówkowym `<stdlib.h>`.

## Alokacja pamięci w bibliotece AVR-libc

Pamięć alokowana dynamicznie przydzielana jest z obszaru pamięci zwanego stertą (ang. *Heap*). Sterta zajmuje obszar od końca segmentu `.bss` do końca dostępnej pamięci RAM minus pewien margines przydzielony na stos. Dokładne położenie sterty określone jest przez symbole zdefiniowane w skryptach linkera: `_heap_start` i `_heap_end`. Na ich podstawie biblioteka AVR-libc inicjalizuje dwie zmienne: `_malloc_heap_start` i `_malloc_heap_end`, używane przez program alokatora pamięci dynamicznej. Wartość symboli `_heap_start` i `_heap_end` możemy zmienić na etapie linkowania programu, natomiast wartość zmiennych `_malloc_heap_start` i `_malloc_heap_end` można zmienić w trakcie wykonywania programu.



Wskazówka

Ważne jest, aby takiej zmiany dokonać przed pierwszym użyciem funkcji alokującej pamięć.

Alokator używa jeszcze jednej stałej — `_malloc_margin`. W przypadku kiedy wartość zmiennej `_malloc_heap_end` wynosi zero, alokator ustawia ją na adres stosu pomniejszony o `_malloc_margin`. Dzięki temu odpowiednia ilość pamięci zarezerwowana jest wyłącznie na potrzeby stosu. Domyślna wartość tej stałej to 32 bajty, co w wielu programach jest wartością niewystarczającą. Jednak jej zmiana nie ma sensu — powiększenie jej co prawda zmniejszy ryzyko kolizji ze stosem, ale zwiększy ryzyko niepowodzenia przy próbie alokacji pamięci. W efekcie program będzie i tak działał nieprawidłowo, tyle że z innych przyczyn.

Obszar sterty rozrasta się w góre, tzn. kolejne bloki pamięci alokowane są pod kolejnymi, coraz wyższymi adresami. Z kolei dane lokalne i adresy powrotów z procedur oraz przerwań umieszczone są na stosie, który położony jest na końcu dostępnego obszaru pamięci i rozrasta się w dół — w kierunku sterty. **Wynika z tego, że w przypadku niewystarczającej ilości pamięci RAM może dojść do kolizji pomiędzy tymi dwiema strukturami danych.** Do takiej kolizji dochodzi w kilku sytuacjach:

- ◆ Alokujemy duże obszary pamięci na stercie lub wiele mniejszych obszarów.
- ◆ Wykorzystujemy dużo zmiennych lokalnych, szczególnie o typach tablicowych, które zużywają ogromne ilości miejsca na stosie.
- ◆ Rekurencyjnie wywołujemy funkcje o dużym stopniu zagnieżdżenia — każdy krok rekurencji wiąże się z koniecznością odłożenia na stosie adresu powrotu oraz utworzenia zmiennych lokalnych.
- ◆ Dochodzi do wycieku pamięci na skutek utraty referencji do przydzielonego bloku i w efekcie niemożliwości jego zwolnienia.

Detekcja takiej kolizji jest niezwykle trudna. Metody takiej detekcji zostaną szczegółowo omówione w dalszej części rozdziału.

Zaalokowany fragment pamięci pozostaje do dyspozycji od chwili jego przydzielenia za pomocą funkcji `malloc` lub podobnych do chwili jego zwolnienia przy pomocy funkcji `free`. **W przeciwieństwie do innych zmiennych zmienne dynamiczne nie są zwalniane automatycznie, np. po opuszczeniu danego bloku programu.** Powoduje to, że programista musi uważnie śledzić takie zmienne, aby z jednej strony nie dopuścić do sytuacji, w której traciona jest referencja do bloku pamięci, a w efekcie niemożliwe będzie jego zwolnienie. Z drugiej strony, nie można zwolnić pamięci, do której później chce się odwołać.



Wskazówka

Pamiętaj, aby zwalniać wyłącznie pamięć, która wcześniej została zaalokowana za pomocą funkcji `malloc` i pochodnych.

Nigdy nie należy zwalniać pamięci, której nie zaalokowaliśmy, np.

```
char mem[]="Tego nie alokowaliśmy";
free(mem);      //Błąd
```

W powyższym przykładzie zmienna `mem` jest wskaźnikiem na tekst. Ponieważ wskaźnik ten nie wskazuje na strukturę w obrębie sterty, tylko na stałą, nie jest możliwe jego zwolnienie. Jednak kompilator nie ma szans na wykrycie takiej sytuacji, w efekcie powyższy kod poprawnie się kompliuje. Jednak zwolnienie wskaźnika spowoduje niezdefiniowane efekty.

## Funkcja malloc

Prototyp funkcji `malloc` jest zadeklarowany następująco:

```
void *malloc(size_t ilość_bajtów)
```

Funkcja ta umożliwia przydzielenie bloku pamięci o podanej liczbie bajtów i zwrócenie do niego referencji. W przypadku kiedy alokacja bloku nie powiodła się, zwracana jest wartość `NULL`. Alokacja może zakończyć się niepowodzeniem z dwóch powodów:

- ◆ Nie istnieje wystarczająca ilość pamięci RAM.
- ◆ Co prawda ilość pamięci jest wystarczająca, ale nie istnieje pojedynczy, wystarczająco duży blok wolnej pamięci.

Ta druga sytuacja może się zdarzyć, kiedy dojdzie do fragmentacji sterty na skutek wielu żądań przydziału i zwolnienia pamięci. Blok pamięci, do którego zwracany jest wskaźnik, nie jest zainicjalizowany — tzn. wartości komórek pamięci należących do tego bloku są przypadkowe.

## Funkcja calloc

Jeśli chcemy, aby przydzielony blok pamięci był wypełniony zerami, to możemy użyć funkcji `calloc`, której prototyp wygląda następująco:

```
void *calloc(size_t ilość_elementów, size_t rozmiar_elementu);
```

W przeciwieństwie do `malloc` funkcja `calloc` przydziela blok pamięci o rozmiarze w bajtach równym `ilość_elementów*rozmiar_elementu`, np.:

```
int *i=calloc(100, sizeof(int));
```

przydziela blok pamięci o długości umożliwiającej pomieszczenie 100 elementów o typie `int`. Analogicznie do `malloc`, w przypadku kiedy przydział jest niemożliwy, funkcja zwraca `NULL`. Typy przyjmowanych argumentów powodują, że funkcja `calloc` jest szczególnie przydatna przy tworzeniu tablic o dynamicznych rozmiarach.

## Funkcja realloc

W przypadku kiedy chcemy zmienić rozmiar przydzielonego bloku pamięci (chcemy go powiększyć lub pomniejszyć), możemy wykorzystać funkcję `realloc` o prototypie:

```
void *realloc(void *wskaźnik, size_t rozmiar);
```

Działanie tej funkcji jest dosyć zawiłe. Jeżeli żądamy zmniejszenia bloku pamięci, powstała wolna pamięć jest zwracana do puli wolnej pamięci na stercie. **Żądanie zmniejszenia bloku pamięci zawsze kończy się sukcesem.** Inaczej sprawa wygląda w sytuacji, kiedy chcemy powiększyć pulę zaalokowanej pamięci. Jeżeli wskazany blok jest ostatni i za nim znajduje się wystarczająca ilość wolnej pamięci sterty, to jest on po prostu rozszerzany, zwracany jest wskaźnik na cały blok, który w tym wypadku jest równy wskaźnikowi na oryginalny blok pamięci. W innych sytuacjach najpierw dokonywana jest próba przydzielenia nowego bloku pamięci o długości podanej jako parametr rozmiar, następnie kopowane są dane ze starego bloku do nowego i zwracany jest wskaźnik do nowego bloku pamięci. Widzimy więc, że taka operacja jest bardzo kosztowna. Dodatkowo tymczasowo musimy dysponować ilością wolnej pamięci na stercie równej co najmniej rozmiarowi bloku pamięci + nowy żądany rozmiar. Stąd też takie żądanie czasami nie może być spełnione i funkcja zwróci wartość `NULL`, informującą o niepowodzeniu operacji. Tu jednak kryje się poważne niebezpieczeństwo. Przeanalizujmy program:

```
int *x=malloc(100);
x=realloc(x, 200);
```

W pierwszej linii przydzieliśmy blok pamięci o długości 100 bajtów. Następnie podjęliśmy próbę jego rozszerzenia do 200 bajtów. Jeżeli operacja ta się powiedzie, `x` będzie wskazywać na nowy obszar pamięci, pamięć wcześniej wskazywana przez `x` zostanie automatycznie zwolniona. Co jednak w sytuacji, kiedy rozszerzenie się nie powiedzie? Wskaźnik `x` będzie zawierał wartość `NULL`, jednocześnie jego poprzednia wartość zostanie utracona. W takiej sytuacji (działanie `realloc` zakończyło się niepowodzeniem) poprzedni blok pamięci nie zostaje jednak zwolniony. W efekcie utraciliśmy referencję do przydzielonego wcześniej bloku pamięci, niemożliwe będzie więc jego zwolnienie! Jest to jedna z przyczyn powstawania tzw. **wycieków pamięci**. Aby zapobiec takiej sytuacji, powyższy kod musimy zmodyfikować:

```
int *x=malloc(100);
int *tmpx=realloc(x, 200);
if(tmpx) x=tmpx;
else
{
    //Tu musimy obsłużyć sytuację, w której rozszerzenie bloku nie było możliwe
}
```

Zauważmy, że w powyższym kodzie nigdy nie tracimy referencji do przydzielonego przez `malloc` bloku pamięci. W przypadku kiedy `realloc` zakończy się niepowodzeniem, ciągle mamy referencję do oryginalnego bloku pamięci.

Funkcji `realloc` jako wskaźnik do pamięci możemy przekazać także wartość `NULL`:

```
int *x=realloc(NULL, 100);
```

Działa dokładnie tak samo jak wywołanie:

```
int *x=malloc(100);
```

Inną specjalną postacią wywołania `realloc` jest sytuacja, w której długość bloku wynosi 0:

```
int *x=malloc(100);  
x=realloc(x, 0);
```

W takiej sytuacji nowy blok ma długość 0 bajtów, efektywnie więc funkcja zwalnia blok pamięci (jest równoważna funkcji `free(x)`). Ponieważ zmniejszenie bloku pamięci zawsze kończy się sukcesem, nie istnieje obawa wycieku pamięci spowodowanego utratą referencji do przydzielonego bloku.

Wszystkie wyżej wymienione funkcje zwracają wskaźnik o typie `void`, który jest kompatybilny (w języku C) z dowolnym typem danych. Stąd też możliwe jest przypisanie wyniku działania powyższych funkcji do wskaźnika o dowolnym typie. Możemy jednak jawnie wymusić konwersję typów. Takie podejście ma kilka zalet:

- ◆ Czyni program kompatybilnym z językiem C++, w którym konwersja typów musi być jawną.
- ◆ Jeśli przypadkowo zmienimy typ wskaźnika, do którego przypisujemy wynik działania powyższych funkcji, to kompilator zgłosi ostrzeżenie.

Jawne rzutowanie typów ma także wadę — jeżeli nie zostanie włączony nagłówek `<stdlib.h>` zawierający prototypy powyższych funkcji, kompilator przyjmie, że zwracają one wartość `int`, i wyświetli ostrzeżenie o konwersji z typu `int` na typ wskaźnikowy. W przypadku jawnej konwersji typów nie nastąpi zgłoszenie tego problemu.

Powyższe funkcje w przypadku niepowodzenia zwracają wartość `NULL`. Programując na komputerach klasy PC, powinniśmy sprawdzać zwracaną wartość i w przypadku zwrócenia `NULL` podjąć odpowiednie działanie, np. zamknięcie aplikacji i wygenerowanie ostrzeżenia o braku pamięci. Natomiast nie bardzo wiadomo, co należy zrobić w przypadku, gdy powyższe funkcje zwrócią `NULL` w programie wykonywanym przez mikrokontroler. Brak systemu operacyjnego i niemożność jakiegokolwiek rozwiązania zaistniałego problemu powoduje, że sprawdzanie wyniku tych funkcji nie ma wielkiego sensu. W efekcie możemy założyć, że każda operacja alokacji pamięci zakończy się sukcesem, jednocześnie tak pisząc program i dobierając sprzęt, aby to założenie zawsze było słusze. Ze względu na brak mechanizmów kontroli pamięci w mikrokontrolerach AVR użycie wskaźnika o wartości `NULL` nie generuje żadnych błędów. Jednak użycie takiego wskaźnika spowoduje modyfikację obszarów pamięci zarezerwowanych dla rejestrów procesora i w rezultacie program nie będzie działał zgodnie z naszymi oczekiwaniami.

## Funkcja free

Przydzielony blok pamięci możemy zwolnić, korzystając z funkcji `free`:

```
void free(void *wskaźnik);
```

Funkcja ta zwalnia pamięć wskazywaną przez wskaźnik; kiedy wskaźnik jest równy `NULL`, funkcja nic nie robi. Stąd też wywołanie:

```
free(NULL);
```

jest zupełnie poprawne. Zwalniana pamięć wraca do puli wolnej pamięci na stercie i może zostać ponownie przydzielona.



Pamiętaj, aby zawsze zwracać pamięć przydzieloną za pomocą funkcji malloc, calloc, realloc.

## Wycieki pamięci i błędne użycie pamięci alokowanej dynamicznie

Powyżej przedstawiliśmy dwie typowe sytuacje, w których doszło do nieprawidłowego użycia alokacji dynamicznej. W pierwszym przykładzie dokonaliśmy próby zwolnienia pamięci, która nie została przydzielona funkcjami malloc/calloc/realloc. Druga polegała na wycieku pamięci podczas wywołania funkcji realloc. Poniżej przedstawione zostaną inne typowe błędy. Pierwszym z nich jest próba dostępu do pamięci poprzez wskaźnik wskazujący na blok pamięci nieprzydzielony wcześniej aplikacji lub zajęty przez inne struktury, np.:

```
int *x;  
*x=10;
```

W powyższym przykładzie zadeklarowaliśmy wskaźnik na typ int, następnie komórce pamięci przez niego wskazywanej przypisaliśmy wartość 10. Problem w tym, że nigdzie nie określiliśmy, na co ten wskaźnik wskazuje, w efekcie możemy otrzymać nieprzewidywalne skutki. Poprawny kod powinien wyglądać następująco:

```
int *x=malloc(sizeof(int));  
*x=10;
```

Kolejny częsty błąd to próba dostępu do pamięci leżącej poza przydzielonym blokiem:

```
int *x=malloc(10*sizeof(int));  
x[10]=10;
```

W tym wypadku x wskazuje na blok 10 elementów o typie int, natomiast zapisujemy do 11. elementu, więc nadpisana zostanie pamięć należąca do innej struktury. Tego typu błędy są trudne do wykrycia, gdyż jeśli po przydzielonym bloku jest wolna pamięć, to nic zlego się nie stanie. Dopiero w sytuacji, w której pamięć ta zostanie wykorzystana przez inną zmienną, program zacznie działać nieprawidłowo. Jeśli w pisany programie obserwujesz nieprzewidywalne, niewynikające bezpośrednio z programu zmiany wartości zmiennych, to najprawdopodobniej dochodzi do nadpisywania okolicznych struktur. W takiej sytuacji przede wszystkim należy sprawdzić, czy używane indeksy tablic są prawidłowe; kolejnym podejrzaniem stają się zmienne alokowane dynamicznie i wskaźniki.

Kolejny błąd polega na użyciu wskaźnika wskazującego na blok pamięci, który wcześniej został zwolniony:

```
int *x=malloc(10*sizeof(int));  
free (x);  
x[1]=10;
```

Po wywołaniu funkcji free x może mieć dowolną wartość, ale wskaźnik taki nie wskazuje już na blok pamięci przydzielony aplikacji. Nie można więc takiego wskaźnika używać.



Niektórzy programiści, aby łatwiej wykrywać tego typu błędy, jawnie nieużywanym wskaźnikom przypisują wartość NULL.

Zapobiega to także błędom polegającym na próbie wielokrotnego zwolnienia tego samego bloku pamięci:

```
int *x=malloc(100);  
free(x);  
free(x);
```

W przypadku jawnego przypisania wartości NULL po pierwszym wywołaniu funkcji free drugie jest bezpieczne, gdyż free(NULL) jest poprawną konstrukcją:

```
int *x=malloc(100);  
free(x);  
x=NULL;  
free(x);
```

Wyjście z funkcji bez zwolnienia pamięci to kolejny często popełniany błąd:

```
void test()  
{  
    int *x=malloc(10);  
}
```

Po wyjściu z funkcji test() zmienna lokalna jest tracona, w efekcie tracona jest referencja do przydzielonego bloku pamięci, niemożliwe jest więc jego zwolnienie. **Pamiętajmy, że w przeciwieństwie do innych zmiennych zmienne alokowane dynamicznie musimy jawnie zwolnić w chwili, kiedy przestajemy z nich korzystać:**

```
void test()  
{  
    int *x=malloc(10);  
    free (x);  
}
```



Pamiętaj, że wszystkie wyżej wymienione funkcje nie są funkcjami reentry. Nie wolno ich więc wywoływać w procedurach obsługi przerwań i innych częściach programu wykonywanych asynchronicznie.

## Jak działa alokator

Alokator odpowiedzialny za zarządzanie pamięcią sterty jest zrealizowany niezwykle prosto. Sterta składa się z bloków pamięci (wolnej lub przydzielonej aplikacji) oznaczonych specjalnymi metadanymi. W pliku `<stdlib_private.h>` (plik ten występuje tylko w źródłach AVR-libc) znajduje się definicja struktury odpowiedzialnej za przechowywanie metadanych:

```
struct __freelist {
    size_t sz;
    struct __freelist *nx;
};
```

Jak widzimy, składa się ona z pola `sz` przechowującego długość bloku pamięci oraz wskaźnika `nx` przechowującego referencję do kolejnego wolnego bloku. Adres pierwszego elementu listy przechowywany jest w specjalnej zmiennej `_flp`:

```
extern struct __freelist * __flp;
```

W przypadku przydzielenia bloku pamięci alokator potrzebuje dodatkowych 2 bajtów na przechowanie długości przydzielonego bloku (rozmiar pola `sz` struktury `__freelist`). Podczas żądania przydziału pamięci alokator przeszukuje listę wolnych bloków i po znalezieniu pierwszego wystarczająco dużego bloku dzieli go na dwie części — pierwsza będzie przydzielona aplikacji, natomiast reszta wolnej pamięci w bloku zostanie zwrócona do listy wolnych bloków. Jedynie w przypadku, kiedy znaleziony wolny blok jest tylko o 2 bajty dłuższy niż długość żądania przydziału przez aplikację, żądanie to w sposób ukryty jest zmieniane na o 2 bajty dłuższe.



Wskazówka

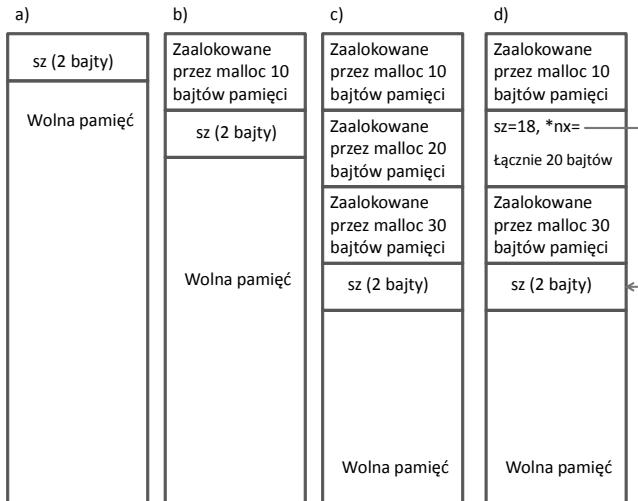
Jednak w języku C nie istnieje żaden mechanizm umożliwiający sprawdzenie rzeczywistej długości zaalokowanego bloku pamięci. Stąd też nigdy nie należy zapisywać/odczytywać pamięci poza ostatnim adresem, którego alokacji jawnie zażądaliśmy.

W przypadku wywołania funkcji `free` alokator zwraca dany blok pamięci poprzez utworzenie do niego referencji w strukturze `__freelist`. Dodatkowo, jeśli sąsiednie bloki pamięci też są wolne, to nastąpi ich scalenie w jeden większy blok pamięci. Ten prosty mechanizm znacznie ogranicza fragmentację pamięci. W pewnych sytuacjach do fragmentacji musi dojść, co pokazano na rysunku 6.1.

Zwolniony został blok pamięci leżący pomiędzy dwoma innymi. W efekcie pula wolnej pamięci na stercie powiększyła się, lecz rozmiar maksymalnego możliwego do zaalokowania bloku się nie zmienił. Jeśli kolejne żądanie przydziału pamięci będzie dotyczyło bloku o długości  $\leq 20$  bajtów, to zostanie on przydzielony w obszarze pamięci wcześniej zwolnionym. Jeśli żądanie będzie dotyczyło dłuższego bloku, to zostanie ono zrealizowane poprzez pominięcie kolejnego wolnego bloku pamięci. Na skutek takich naprzemiennych przydziałów i zwolnień sterata może zawierać wiele wolnych bloków o niewielkiej długości i w rezultacie kolejne żądania przydziału nie będą mogły zakończyć się sukcesem. Na szczęście, taka sytuacja nie jest częsta i raczej nie wystąpi w normalnym programie, w którym żądaniom przydziału pamięci towarzyszą odpowiadające

**Rysunek 6.1.**

*Mechanizm alokacji pamięci na stercie:  
 a) sterta jest pusta,  
 b) sterta po przydzieleniu 10 bajtów pamięci za pomocą funkcji malloc, c) ta sama sterta po przydzieleniu dwóch bloków o długości 20 i 30 bajtów,  
 d) poprzednia sterta po zwolnieniu bloku pamięci o długości 20 bajtów*



im instrukcje zwalniające pamięć. Widzimy także, że alokacja każdego bloku pamięci pochłania dodatkowe 2 bajty na przechowanie jego długości. W przypadku alokacji dużej liczby małych bloków pamięci to dodatkowe zapotrzebowanie może być znaczące.

## Wykrywanie kolizji stertry i stosu

Jak wcześniej wspomniano, w pewnych sytuacjach może dojść do kolizji stertry i stosu. Wielkość stertry jest ograniczona (jej koniec określa zmienna `_malloc_heap_end`), więc niezależnie od żądań przydziału pamięci stertry nigdy nie „wejdzie” na stos. Taki limit nie dotyczy jednak stosu. Na stosie tworzone są zmienne lokalne programu, przechowywane są wartości rejestrów (głównie ma to znaczenie w przypadku procedur obsługi przerwań), odkładane są także adresy powrotów z podprogramów. W efekcie stos, rozrastając się „w dół”, może zająć obszary pamięci wykorzystywane przez stertry, a następnie obszary zajmowane przez zmienne umieszczone w sekcjach `.data` i `.bss`. Taka sytuacja prowadzi oczywiście do błędного działania programu. Objawy mogą być różne, najczęściej pisany program, który do tej pory działał dobrze, nagle zaczyna działać w sposób nieprzewidywalny. Nie od razu musi dojść do dużych zaburzeń funkcjonowania programu, najpierw możemy zauważyc, że wartości niektórych zmiennych (w pierwszej kolejności alokowanych dynamicznie) zmieniają się w sposób nieprzewidywalny. Niestety, nie istnieje prosta metoda detekcji takich sytuacji. Poniżej zostaną omówione niektóre sposoby. Jednak aby unikać takich problemów, zawsze należy starać się oszacować ilość pamięci, jaka może być potrzebna aplikacji, i dobrać procesor tak, aby ilość dostępnej pamięci była wystarczająca do realizacji zadania. Im większym nadmiarem pamięci RAM będziemy dysponować w stosunku do minimalnej koniecznej do realizacji programu ilości, tym mniejsze jest prawdopodobieństwo wystąpienia konfliktu.

## Metoda I — własne funkcje alokujące pamięć

Metoda ta polega na napisaniu własnych funkcji alokujących pamięć (`malloc`, `calloc`, `realloc` i `free`). Dzięki temu podczas każdego wywołania którejś z powyższych funkcji możemy porównać adres końca sterty ze wskaźnikiem stosu. Zaletą tej metody jest prostota i brak konieczności jakichkolwiek zmian w programie. Metoda ta ma pewne wady — sprawdzanie dokonywane jest wyłącznie w chwili wykonywania operacji na stercie, natomiast stos może rozrastać się niezależnie od alokacji/dealokacji pamięci. W efekcie nie możemy wyłapać wszystkich konfliktów.

## Metoda II — sprawdzanie ilości dostępnej pamięci

Metoda ta, podobnie jak poprzednia, polega na okresowym sprawdzeniu adresu ostatniego zajętego bajtu sterty i porównaniu go ze wskaźnikiem stosu. Sprawdzenie dokonywane jest okresowo, poprzez wywołanie w krytycznych miejscach programu funkcji sprawdzającej. Inny wariant, wygodniejszy, polega na okresowym, w miarę częstym wywoływaniu funkcji sprawdzającej w procedurze obsługi przerwania, np. *timera*. Dzięki temu asynchronicznie w stosunku do wykonywanego programu sprawdzamy ilość wolnej pamięci, mamy więc spore szanse na wykrycie konfliktu. Oczywiście i ta metoda nie zapewnia 100-procentowej skuteczności.

## Metoda III — marker

W metodzie tej na końcu obszaru pamięci zajmowanego przez stertę umieszczamy charakterystyczny marker — zmienną o unikalnej wartości. Następnie okresowo sprawdzamy, czy wartość naszego markera nie uległa zmianie. Jeśli uległa, to znaczy, że coś (najpewniej stos), rozrastając się, zajęło obszar pamięci przydzielony zmiennej markerowej. W celu wykrycia zmiany markera możemy posłużyć się technikami wykorzystywanymi w metodach I i II. Metoda ta jest bardzo skuteczna. Jakakolwiek modyfikacja markera jest łatwo wykrywalna. Szansa, że dane nadpisujące marker mają taką samą wartość jak marker, jest mała. Jeśli marker będzie 32-bitowy, to szansa, że nadpisywane dane będą takie same, jest jak  $1:2^{32}$ .

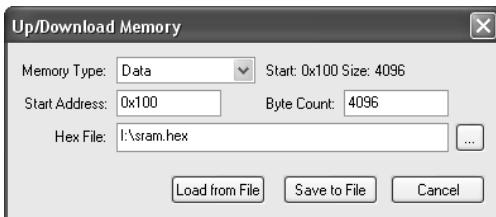
## Metoda IV — wzór w pamięci

Metoda ta jest podobna do metody III, z tym że zamiast jednej konkretnej zmiennej markerowej jako marker wykorzystujemy całą pamięć RAM mikrokontrolera. W tym celu wypełniamy ją znany wzorcem. Szansa, że zapisywane dane będą miały identyczną wartość jak wzorzec, jest mała, w związku z tym łatwo zidentyfikować komórki pamięci zajęte przez dane. W efekcie jeśli pomiędzy stertą a stosem nie widzimy naszego wzorca, to świadczy to o konflikcie. Problemem w tej metodzie jest detekcja zmiany wzoru. Najprościej w tym celu wykorzystać interfejs JTAG. Uruchamiamy program z pamięcią RAM wypełnioną wcześniej naszym wzorcem, po pewnym czasie przerywamy działanie programu i na podglądzie pamięci sprawdzamy, gdzie nasz wzór został zaburzony. Metoda ta nie za bardzo nadaje się do detekcji konfliktów w przypadku procesorów

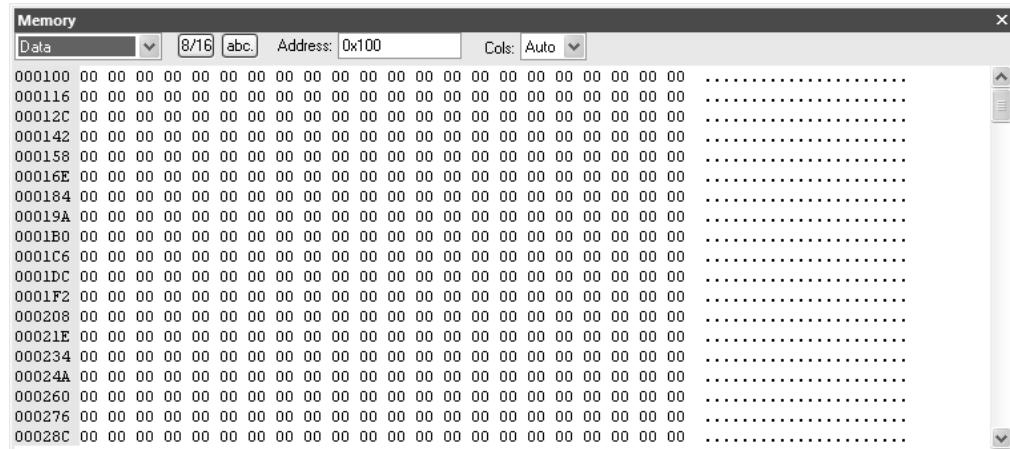
nieposiadających interfejsu JTAG. Wypełnienie pamięci wzorcem możemy uzyskać programowo, ale skoro planujemy użyć interfejsu JTAG, to również ten etap można zrobić za pomocą narzędzi AVR Studio, co oszczędzi nam konieczności modyfikacji programu. W tym celu rozpoczynamy sesję debugera (*Debug/Start Debugging*). Po starcie wybieramy opcję *Debug/Up/Download Memory*; powinniśmy zobaczyć okno pokazane na rysunku 6.2.

## Rysunek 6.2.

*Wczytywanie danych do wybranej sekcji pamięci*



W polu *Hex File* wpisujemy nazwę pliku hex zawierającego obraz pamięci RAM. Możemy go wygenerować w AVR Studio lub dowolnym innym programie mającym możliwość edytowania plików hex. Następnie klikamy na przycisk *Load from File*, co powoduje załadowanie przy użyciu interfejsu JTAG nowej zawartości pamięci RAM do mikrokontrolera. W naszym przykładzie cała pamięć RAM zostanie zainicjowana zerami. Po tej operacji wygląda ona jak na rysunku 6.3 (*View/Memory*).



**Rysunek 6.3.** Obraz pamięci SRAM po wczytaniu pliku zawierającego wzorzec

Uruchamiamy program (*F5*) i po chwili go przerywamy, ponownie oglądając obraz pamięci (rysunek 6.4).

Po uruchomieniu aplikacji pamięć została częściowo wypełniona różnymi wartościami, które nadpisały nasz wzorzec (0x00). W powyższym przykładzie widzimy, że stos maksymalnie doszedł do adresów 0x001081, poniżej mamy zera, co świadczy o tym, że obszary o niższych adresach nie były używane, nie nastąpiła więc najprawdopodobniej kolizja ze stertą. Pozwalamy programowi wykonywać się przez dłuższy czas i ponownie go przerywamy (rysunek 6.5).

Memory		8/16	abc.	Address: 0xF2E	Cols: Auto	
000F2E	00	00	00	00	00	.....
000F44	00	00	00	00	00	.....
000F5A	00	00	00	00	00	.....
000F70	00	00	00	00	00	.....
000F86	00	00	00	00	00	.....
000F9C	00	00	00	00	00	.....
000FB2	00	00	00	00	00	.....
000FC8	00	00	00	00	00	.....
000FDE	00	00	00	00	00	.....
000FF4	00	00	00	00	00	.....
00100A	00	00	00	00	00	.....
001020	00	00	00	00	00	.....
001036	00	00	00	00	00	.....
00104C	00	00	00	00	00	.....
001062	00	00	00	00	00	.....
001078	00	00	00	00	00	.....
00108E	04	05	79	05	29	.....
0010A4	10	B3	06	BA	05	77
0010B0	AA	9F	00	23	05	81
0010D0	10	DE	1E	01	00	00
0010E6	6B	06	05	EA	05	ZE
0010FC	2B	A5	17	A4		

Rysunek 6.4. Obraz pamięci po wczytaniu wzorca i uruchomieniu aplikacji. Nadpisane komórki pamięci mają wartość inną niż wzorzec (w tym przypadku inną niż 0x00)

Memory		8/16	abc.	Address: 0xF2E	Cols: Auto	
000F2E	00	00	00	00	00	.....
000F44	00	00	00	00	00	.....
000F5A	00	00	00	00	00	.....
000F70	00	00	00	00	00	.....
000F86	00	00	00	00	00	.....
000F9C	00	00	00	00	00	.....
000FB2	00	00	00	00	00	.....
000FC8	00	00	00	00	00	.....
000FDE	00	00	00	00	00	.....
000FF4	00	00	00	00	00	.....
00100A	00	00	00	00	00	.....
001020	00	00	00	00	00	.....
001036	00	00	00	00	00	.....
00104C	3E	04	3E	04	52	.....
001062	64	24	3E	04	83	10
001078	3E	10	7C	04	04	10
00108E	30	10	B1	05	30	00
0010A4	9F	32	10	A9	34	71
0010BA	10	CF	10	D0	00	03
0010D0	04	10	D4	24	6B	3E
0010E6	03	00	83	EA	05	05
0010FC	2B	B0	17	A4		

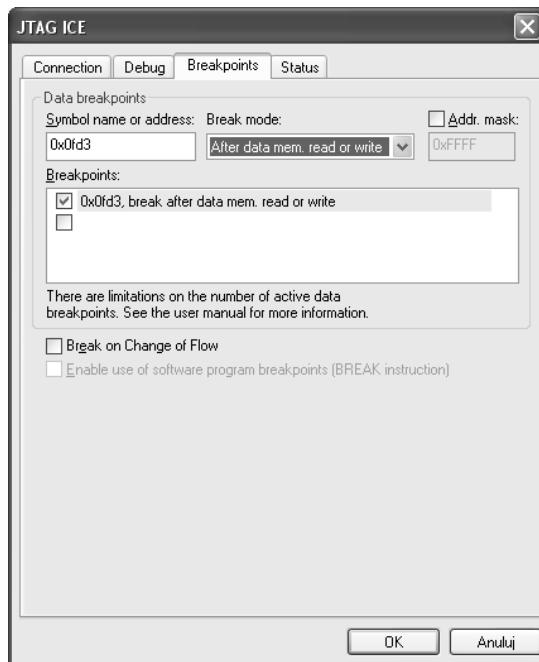
Rysunek 6.5. Obraz pamięci po dłuższym czasie wykonywania programu. W dalszym ciągu większa część pamięci wypełniona jest wzorcem

Widzimy, że tym razem stos „zszedł” aż do adresu 0x00104C, ale poniżej ciągle mamy nasz wzorzec, a więc najprawdopodobniej wszystko jest ok. Gdyby zamiast zer cała pamięć wypełniona była różnymi wartościami, najprawdopodobniej mielibyśmy do czynienia z kolizją stosu i innych obszarów danych (sterta, zmienne programu).

## Metoda V — wykorzystanie interfejsu JTAG

W procesorach wyposażonych w interfejs JTAG możemy skorzystać z możliwości, jakie on oferuje, m.in. stawiania pułapek. W przypadku kiedy program próbuje dokonać odczytu lub zapisu wskazanego miejsca pamięci, następuje jego przerwanie. Poniżej pokazane zostanie wykorzystanie tego mechanizmu w AVR Studio z podłączonym JTAGICE do procesora ATMega128. W pierwszym etapie musimy określić, do jakiego adresu może rozrastać się stos. Pamiętamy, że standardowo wskaźnik stosu ustawiony jest na koniec pamięci RAM i maleje w kierunku niższych adresów. Założymy, że w pisanej aplikacji na stos przeznaczamy 300 bajtów. Wskaźnik stosu SP nigdy nie powinien być niższy niż ostatnia komórka pamięci minus 300 bajtów, czyli  $0x10ff - 0x012c = 0x0fd3$ . Ustawiamy więc pod tym adresem pułapkę sprzętową. W tym celu należy podłączyć interfejs JTAG do układu i rozpoczęć sesję debugera (*Debug/Start Debugging*). Następnie wybieramy opcję *Debug/JTAGICE Options* i w zakładce *Breakpoints* ustawiamy typ pułapki i adres (rysunek 6.6).

**Rysunek 6.6.**  
Ustawienie pułapki  
sprzętowej  
na wybrany adres  
pamięci SRAM



Klikamy *OK* i klawiszem *F5* lub poprzez wybór *Debug/Run* uruchamiamy aplikację. Jeśli procesor wykona operację odczytu bądź zapisu wybranej komórki, to wykonywanie programu zostanie przerwane i wyświetlony zostanie fragment kodu odpowiedzialny za próbę dostępu do określonej przez nas komórki pamięci. Dodatkowo możemy ustawić pułapkę na pewien obszar pamięci wybrany za pomocą maski. Adres komórki pamięci, do której odbywa się dostęp, jest mnożony bitowo przez maskę, a następnie porównywany z podanym adresem. Jeśli porównanie wypada pozytywnie, to wykonywanie programu jest przerywane. Dzięki temu mechanizmowi możemy ustawić pułapkę na kilka kolejnych adresów, zwiększaając w ten sposób szansę detekcji potencjalnej kolizji.

# Rozdział 7.

# Wbudowana pamięć EEPROM

Procesory AVR mają oddzielne przestrzenie adresowe dla różnych typów pamięci. Dostęp do każdej z nich odbywa się za pomocą specyficznych dla niej instrukcji. Podobnie rzeczą się ma w przypadku pamięci EEPROM. Co prawda w niektórych mikrokontrolerach z rodziny AVR (np. ATXMega) pamięć ta jest zmapowana do odczytu w obszarze pamięci SRAM, jednak w większości przypadków dostęp do niej odbywa się za pomocą specjalnych rejestrów:

- ◆ *EEAR* (ang. *EEPROM Address Register*) — zawiera adres adresowanej komórki pamięci EEPROM.
- ◆ *EEDR* (ang. *EEPROM Data Register*) — zawiera 8-bitową daną do zapisu lub odczytaną z pamięci EEPROM.
- ◆ *EECR* (ang. *EEPROM Control Register*) — rejestr kontrolny, za pomocą którego odbywają się operacje kasowania, zapisu i odczytu pamięci EEPROM.

Pamięć EEPROM nie traci swojej zawartości po wyłączeniu mikrokontrolera i w przeciwieństwie do pamięci FLASH, w której zawarty jest program, umożliwia łatwe przeprogramowanie swej zawartości. Jednak w przeciwieństwie do pamięci SRAM czas potrzebny na zapis nowej wartości do pamięci jest bardzo długi i wynosi ok. 3,3 ms/komórkę. Co gorsze, liczba możliwych cykli zapisu jest ograniczona do ok. 100 tys./komórkę. Po przekroczeniu tej wartości producent nie gwarantuje parametrów dotyczących czasów przechowywania zawartości. Każda operacja kasowania komórki pamięci powoduje stopniowe pogarszanie właściwości izolatora, w efekcie dochodzi do ucieczki ładunku elektrycznego i powolnego kasowania zawartości komórki. Im więcej cykli kasowania wykonano, tym proces ten przebiega szybciej.



Uszkadzanie komórki pamięci EEPROM powodują wyłącznie operacje kasowania. Zapis oraz odczyt nie wywołuje takich efektów i może być wykonany nieskończoną ilością razy.

W niektórych mikrokontrolerach AVR zapis do komórki pamięci EEPROM zawsze wymusza wykonanie operacji kasowania. W nowszych te dwie operacje można przeprowadzić niezależnie od siebie. Kasowanie zawartości komórki pamięci EEPROM powoduje ustawienie wszystkich bitów, w efekcie taka komórka zawiera wartość 0xFF. Zapis polega więc wyłącznie na zerowaniu niektórych bitów. Tak więc komórki niezaprogramowanej pamięci EEPROM mają wartość 0xFF.



Należy pamiętać, że programowanie pamięci EEPROM odbywa się niezależnie od programowania pamięci FLASH mikrokontrolera.

Jeśli końcowym etapem komplikacji programu są pliki w formacie Intel HEX, programując, musimy wybrać opcję programowania zarówno pamięci FLASH, jak i pamięci EEPROM. Zwykle w takim przypadku otrzymujemy dwa oddzielne pliki — jeden z rozszerzeniem *hex*, zawierający dane do zaprogramowania pamięci FLASH, drugi z rozszerzeniem *eep*, zawierający dane niezbędne do zaprogramowania pamięci EEPROM. Jeśli do programowania używamy pliku *elf* (ang. *Executable and Linkable Format*), to programator powinien sam stwierdzić, czy istnieje w nim sekcja definiująca zawartość pamięci EEPROM, i ją ewentualnie zaprogramować. Warto jednak przy korzystaniu z nowego oprogramowania upewnić się, w jaki sposób radzi ono sobie z programowaniem pamięci EEPROM. **Mikrokontrolery AVR posiadają także specjalny fusebit EESAVE, określający, czy podczas kasowania pamięci programu ma być także kasowana zawartość pamięci EEPROM.** Jeśli bit ten jest niezaprogramowany (ma wartość 1), to podczas kasowania pamięci FLASH w trakcie programowania kasowana jest także zawartość pamięci EEPROM. Jeśli bit ten jest zaprogramowany (ma wartość 0), to zawartość pamięci EEPROM jest zachowywana. Dzięki temu jeśli zawiera ona prawidłowe dane, przy kolejnych programowaniach wystarczy tylko wczytać zawartość pamięci FLASH.

## Zapobieganie uszkodzeniu zawartości pamięci EEPROM

Do przypadkowego zapisu komórki pamięci EEPROM może dojść w sytuacji, kiedy procesor lub poszczególne jego bloki nie funkcjonują prawidłowo. Najczęściej z takim stanem mamy do czynienia w momencie, kiedy napięcie zasilające procesor znajduje się poza zakresem dopuszczonym przez producenta układu. Stan taki występuje w chwili włączenia urządzenia do prądu oraz w chwili jego wyłączenia. Dodatkowo problem z zapisem może pojawić się w sytuacji obniżenia napięcia zasilającego poniżej dolnej granicy w trakcie wykonywania zapisu do pamięci EEPROM. Potencjalnie istnieje jeszcze jedna możliwość uszkodzenia zawartości pamięci — kiedy procesor taktowany jest zegarem o częstotliwości przekraczającej maksymalną częstotliwość pracy układu. Oczywiście, ten ostatni powód jest całkowicie pod kontrolą projektanta urządzenia. Układ zawsze powinien być tak zaprojektowany, aby w sytuacji kiedy napięcie z różnych przyczyn jest zbyt niskie, procesor był w stanie resetu. Efekt ten możemy uzyskać, stosując zewnętrzne układy nadzorujące napięcie zasilania lub wykorzystując

układ nadzorujący zasilanie wbudowany w procesor (BOD — ang. *Brown-out Detektor*). Ten ostatni możemy uaktywnić, stosując odpowiednie ustawienia tzw. *fusebitów*. Oprócz aktywowania układu BOD musimy pamiętać także o ustawieniu właściwego napięcia, poniżej którego BOD wywoła reset procesora.



Wskazówka

Pamiętaj! Jeśli ustawisz próg zadziałania BOD na napięcie wyższe niż napięcie zasilające procesor, to mikrokontroler będzie utrzymywany w stanie permanentnego resetu.

W tabeli 7.1 pokazano ustawienia *fusebitów* i progi zadziałania BOD dla mikrokontrolera ATMega88. Jeżeli mikrokontroler pracuje z napięciem zasilającym, dla którego nie da się wybrać odpowiedniego ustawienia progu zadziałania wewnętrznego układu BOD, jedyną możliwością jest zastosowanie zewnętrznego układu.

**Tabela 7.1.** Konfiguracja *fusebitów* i progi zadziałania układu BOD. W tabeli podano minimalne, typowe i maksymalne napięcie zadziałania układu BOD dla mikrokontrolera ATMega88

Fusebity BODLEVEL2:0	Vmin	Vtyp	Vmax
111	Układ BOD jest nieaktywny		
110	1,7 V	1,8 V	2,0 V
101	2,5 V	2,7 V	2,9 V
100	4,1 V	4,3 V	4,5 V

Włączenie układu BOD właściwie powinno być obowiązkowym etapem tworzenia każdego urządzenia opartego o mikrokontrolery AVR. Co prawda BOD dramatycznie zmniejsza ryzyko uszkodzenia zawartości komórki pamięci EEPROM, jednak go całkowicie nie eliminuje. Stąd też powinniśmy stosować także inne metody. Jedną z zalecanych jest zapisanie do rejestru adresowego EEPROM (*EEAR*) adresu nieużywanej komórki pamięci EEPROM. Dzięki temu nawet jeśli dojdzie do przypadkowego zapisu do komórki pamięci, to nie spowoduje on uszkodzenia istotnych danych. Stosunkowo bezpieczne jest ustawienie go tak, aby wskazywał na ostatnią komórkę pamięci EEPROM.

## Kontrola odczytu i zapisu do pamięci EEPROM

Zarówno odczyt, jak i zapis do pamięci EEPROM są operacjami wieloetapowymi. Ze względu na uniemożliwienie przypadkowych modyfikacji tej pamięci na skutek np. błędного działania programu operacje te wymagają wykonania kilku etapów w ścisłe określonym czasie. Każdy dostęp do pamięci niespełniający wszystkich wymagań jest przez procesor ignorowany.

## Odczyt zawartości komórki pamięci

Jest stosunkowo prostą operacją. W pierwszym etapie do rejestru *EEAR* musimy wpisać adres komórki pamięci, której zawartość chcemy odczytać. W przypadku kontrolerów dysponujących >256 bajtów pamięci EEPROM rejestr ten jest 16-bitowy, lecz kolejność dostępu do jego młodszej i starszej połowy jest bez znaczenia. Po wpisaniu adresu musimy ustawić bit zezwolenia na odczyt (*EERE*, ang. *EEPROM Read Enable*) znajdujący się w rejestrze *EECR*. Powoduje to wstrzymanie wykonywania dalszych instrukcji procesora na 4 cykle zegara, po czym odczytana wartość dostępna jest w rejestrze *EEDR*.



Wskazówka

W trakcie zapisu dowolnej komórki pamięci EEPROM niemożliwy jest odczyt danych z tej pamięci. Jeśli taka sytuacja może wystąpić, to przed odczytem musimy upewnić się, że bit *EEPE* (ang. *EEPROM Write Enable*) jest wyzerowany.

## Zapis do komórki pamięci

Operacja zapisu do pamięci EEPROM jest nieco bardziej skomplikowana. W zależności od procesora przebiega ona zawsze w połączeniu z operacją kasowania zawartości komórki, a w nowszych mikrokontrolerach AVR może przebiegać rozdzielnie. Rozdzielenie operacji kasowania od zapisu ma kilka zalet: czas zapisu ulega dwukrotnemu skróceniu, jeśli nie towarzyszy mu operacja kasowania, jest to także wykorzystywane do zwiększenia liczby możliwych zapisów do pamięci EEPROM. Jak pamiętamy, sam zapis nie powoduje skrócenia żywotności komórki. Oczywiście, operacja kasowania jest niezbędna w sytuacji, w której bit pamięci EEPROM o wartości 0 chcemy przeprogramować na 1. Jak pamiętamy, w trakcie zapisu możliwa jest wyłącznie operacja odwrotna, tj. zmiana bitu o wartości 1 na 0. Zapis, podobnie jak odczyt, przebiega w kilku etapach:

- ◆ Pierwszym krokiem jest upewnienie się, że bit *EEPE* rejestru *EECR* jest wyzerowany. Wartość 1 świadczy o aktualnie toczącej się operacji zapisu do pamięci, w efekcie niemożliwe jest przeprowadzenie kolejnego zapisu do czasu, aż poprzedni nie ulegnie zakończeniu.
- ◆ Na kolejnym etapie do rejestru *EEAR* wpisujemy adres komórki pamięci, do której chcemy przeprowadzić operację zapisu.
- ◆ Następnie do rejestru *EEDR* wpisujemy 8-bitową liczbę, która ma znaleźć się w pamięci.
- ◆ W kolejnym etapie do bitu *EEMPE* (ang. *EEPROM Master Write Enable*) rejestru *EECR* wpisujemy wartość 1.
- ◆ W ciągu maksymalnie 4 cykli zegara od zapisania 1 do *EEMPE* do bitu *EEPE* (ang. *EEPROM Write Enable*) rejestru *EECR* wpisujemy 1. Koniec operacji zapisu sygnalizowany jest wyzerowaniem tego bitu lub wygenerowaniem przerwania *EE READY* (o ile na nie zezwoliliśmy, ustawiając bit *EERIE* [ang. *EEPROM Ready Interrupt Enable*]).

Przerwanie wywołane pomiędzy zapisami do *EEMPE* i *EEPE* wydłużałoby czas operacji ponad wymagane 4 cykle, w efekcie zapis do pamięci EEPROM nie powiodłby się.



Wskazówka

Ponieważ od chwili ustawienia bitu *EEMPE* do chwili ustawienia bitu *EEPE* nie może minąć więcej niż 4 cykle zegara, to w systemie, w którym mamy odblokowane przerwania, musimy na czas zmiany tych bitów je zablokować.



Wskazówka

W niektórych procesorach opisane powyżej bity *EEMPE* i *EEPE* nazywają się odpowiednio *EEMWE* i *EEWE*.

W niektórych (nowszych) procesorach AVR w rejestrze *EECR* istnieją dwa opcjonalne bity, których stan precyzuje charakter dokonywanej operacji zapisu. Są to bity *EEPMS* i *EEPMD* (ang. *EEPROM Programming Mode Bits*). Typ wykonywanej operacji zapisu w zależności od stanu tych bitów pokazano w tabeli 7.2.

Tabela 7.2. Tryby zapisu do pamięci EEPROM

EEPMS	EEPMD	Czas programowania	Operacja
0	0	3,4 ms	Kasowanie i zapis w jednej operacji. Domyślna wartość po resecie
0	1	1,8 ms	Tylko kasowanie
1	0	1,8 ms	Tylko zapis
1	1	Kombinacja zarezerwowana	

Jeśli nie zmienimy zawartości tych bitów, to zapis do pamięci będzie się odbywał tak jak w starszych modelach mikrokontrolerów AVR, tj. nastąpi jednoczesne kasowanie i zapis. W przypadku kiedy przeprowadzamy tylko operację kasowania, zawartość rejestru *EEDR* jest bez znaczenia, stąd też w tym trybie możemy pominąć zapis do tego rejestru.

## Dostęp do EEPROM z poziomu AVR-libc

AVR-libc definiuje funkcje umożliwiające wygodny dostęp do wbudowanej w procesor pamięci EEPROM. Prototypy tych funkcji znajdują się w pliku nagłówkowym `<avr/eeprom.h>`. Udostępnione funkcje bazują na metodzie *poolingu* — przed dokonaniem zapisu sprawdzany jest bit *EEPE*. Jeśli nie jest on wyzerowany, to znaczy, że aktualnie trwa zapis do pamięci. W efekcie wywołanie funkcji umożliwiających zapis do pamięci EEPROM jest bardzo czasochłonne i może wynieść nawet kilka ms. Funkcje operujące na pamięci EEPROM modyfikują także rejesty *IO* związane z tą pamięcią.



Wskazówka

Pamiętaj! Wszystkie funkcje zdefiniowane w `<avr/eeprom.h>` nie są funkcjami *reentrant*.

**Biblioteka AVR-libc w żaden sposób nie wspiera operacji na pamięci EEPROM realizowanych za pomocą przerwań.** Jeśli takowych potrzebujesz, trzeba bazować na innych rozwiązańach. Dodatkowo wszystkie funkcje realizujące zapis do pamięci EEPROM dokonują tego łącznie z operacją kasowania. Jeśli wymagane są inne tryby

dokonywania zapisu, np. w celu zrealizowania *wear leveling*, musimy wykorzystać własne funkcje. Jednakże w znakomitej większości przypadków możemy korzystać z funkcji zdefiniowanych w bibliotece AVR-libc. Planując wykorzystanie wbudowanej pamięci EEPROM, natykamy dwa problemy. Pierwszym jest umieszczenie interesujących nas danych w tej pamięci, drugim — sposób dostępu do nich.

## Deklaracje danych w pamięci EEPROM

Dane w pamięci EEPROM możemy umieszczać na kilka sposobów. Możemy przygotować plik zawierający obraz pamięci i wykorzystać go do zaprogramowania procesora. Sposób ten jest mało wygodny i niezbyt uniwersalny. Lepszym rozwiązaniem jest wykorzystanie możliwości definiowania sekcji linkera. Biblioteka AVR-libc definiuje specjalną sekcję o nazwie *eeprom*, wszelkie dane do niej należące zostaną umieszczone w pamięci EEPROM. Jeśli chcemy więc, aby dana zmienna została umieszczona w pamięci EEPROM, to należy do jej deklaracji dodać odpowiedni atrybut:

```
_attribute_((section(".eeprom")))
```

W pliku nagłówkowym *eeprom.h* zdefiniowano dla uproszczenia definicję EEMEM, która jest równoważna powyższemu zapisowi. Aby zawartość zmiennej znalazła się w pamięci EEPROM, musimy ją zadeklarować następująco:

```
int zmienna EEMEM = 100;
```

lub:

```
EEMEM int zmienna = 100;
```



Zmienne odwołujące się do pamięci EEPROM muszą być zadeklarowane jako zmienne globalne.

Próba zadeklarowania takiej zmiennej lokalnie zakończy się błędem:

```
error: section attribute cannot be specified for local variables
```

Zauważmy, że w przypadku zmiennych umieszczanych w pamięci EEPROM deklaracja zmiennej musi być powiązana z jej inicjalizacją. Poniższy ciąg instrukcji:

```
EEMEM int zmienna;
zmienna=10;
```

zostanie skompilowany poprawnie, lecz jeśli spojrzymy na wygenerowany kod assemblerowy:

```
zmienna=255:
188: 8f ef        ldi    r24, 0xFF      : 255
18a: 90 e0        ldi    r25, 0x00      : 0
18c: 90 93 05 00   sts    0x0005, r25
190: 80 93 04 00   sts    0x0004, r24
```

okoże się, że jest on nieprawidłowy. Kompilator nie wie, że EEPROM jest specjalną przestrzenią adresową procesora i dostęp do niego wymaga wygenerowania innych instrukcji, niż gdyby zmienna znalazła się w pamięci SRAM. W efekcie powyższy kod

nie tylko nie przypisuje zmiennej zmiennej wartości 255, lecz powoduje nadpisanie danych w obszarze pamięci SRAM niezarezerwowanym dla zmiennej. W efekcie działanie programu będzie nieprzewidywalne.



Dostęp do zmiennych przechowywanych w pamięci EEPROM wymaga specjalnych funkcji. Nigdy nie odwołuj się do takich zmiennych za pomocą standardowych operatorów języka C.

W pamięci EEPROM możemy przechowywać także złożone typy danych, łańcuchy:

```
EEMEM char tekst[]="Tekst w EEPROM";
```

oraz struktury:

```
struct xy  
{  
    int x;  
    int y;  
};  
  
struct xy EEMEM punkt={12,20};
```

## Funkcje realizujące dostęp do pamięci EEPROM

Standardowe operatory języka C są nieprzydatne do operowania na zmiennych umieszczonych w pamięci EEPROM. Aby uzyskać dostęp do danych zawartych w tych zmiennych, musimy posłużyć się funkcjami zdefiniowanymi w bibliotece AVR-libc. Umożliwiają one odczyt/zapis pojedynczych bajtów i bloków pamięci. Dla wygody zdefiniowano także funkcje umożliwiające odczyt/zapis prostych typów danych języka C, takich jak: word (typy signed i unsigned int), dword (typy signed i unsigned long) oraz float. Zawsze jednym z parametrów wywołania tych funkcji jest adres komórki pamięci EEPROM, z której ma nastąpić odczyt lub do której chcemy zapisać dane. Adres ten możemy uzyskać dzięki operatorowi &. Funkcje operujące na pamięci EEPROM możemy podzielić na trzy kategorie:

- ◆ funkcje odczytujące dane z pamięci EEPROM: eeprom\_read\_byte, eeprom\_read\_word, eeprom\_read\_dword, eeprom\_read\_float i eeprom\_read\_block;
- ◆ funkcje zapisujące dane do pamięci EEPROM: eeprom\_write\_byte, eeprom\_write\_word, eeprom\_write\_dword, eeprom\_write\_float i eeprom\_write\_block;
- ◆ funkcje modyfikujące dane w pamięci EEPROM: eeprom\_update\_byte, eeprom\_update\_word, eeprom\_update\_dword, eeprom\_update\_float i eeprom\_update\_block.

### Odczyt z pamięci EEPROM

Odczyt z pamięci EEPROM odbywa się za pomocą funkcji, których ogólna postać wygląda następująco:

```
typ eeprom_read_typ(const typ *adres);
```

gdzie *typ* jest jednym z wyrażeń: byte, word, dword lub float. Istnieje także funkcja umożliwiająca odczyt bloku pamięci EEPROM, w tym wypadku dodatkowo należy podać wskaźnik do pamięci SRAM, gdzie znajdą się odczytane dane, oraz liczbę odczytywanych bajtów:

```
void read_eeprom_block(void *miejsce_przeznaczenia, const void *adres_w_EEPROM,
→size_t Ilość_bajtów);
```

Funkcja ta umożliwia odczytanie dowolnej liczby bajtów z pamięci EEPROM, jest więc przydatna przy operowaniu na niestandardowych typach danych. Wykorzystanie tych funkcji ilustruje przykład:

```
EEMEM int x=1234;
int a=eprom_read_word(&x);
```

Zdefiniowaliśmy zmienną *x*, która znajduje się w pamięci EEPROM, a jej wartość wynosi 1234. Ponieważ standardowe operatory języka C na takich zmiennych nie działają prawidłowo, nie można więc wykonać przypisania *int a=x*: (jakkolwiek komplator nie zgłosi błędu, to z przyczyn już wymienionych operacja taka nie zadziała prawidłowo). Stąd do odczytania wartości zmiennej *x* użyto funkcji *eeprom\_read\_word*, jako argument podając adres zmiennej *x*. Typ *int* na mikrokontrolerach AVR jest 16-bitowy, stąd użycie funkcji *eeprom\_read\_word*.



**Pamiętajmy,** że jako argument powyższych funkcji zawsze podajemy adres zmiennej umieszczonej w pamięci EEPROM, nigdy nie wartość.

Wcześniej zdefiniowaliśmy zmienną punkt o typie *xy*. Do odczytania zmiennych o typach złożonych najwygodniej jest posłużyć się funkcją *eeprom\_read\_block*:

```
struct xy pkt;
eprom_read_block(&pkt, &punkt, sizeof(punkt));
```

Zadeklarowaliśmy zmienną *pkt* o typie *xy*, która będzie zawierała odczytaną wartość ze zmiennej *punkt* znajdującej się w pamięci EEPROM. Co prawda w naszym przykładzie dokładnie wiemy, że długość struktury *xy* wynosi 4 bajty (dwie zmienne o typie *int*), lecz bezpieczniej jest używać operatora *sizeof()* zwracającego długość typu lub zmiennej. Dzięki temu jeśli kiedyś zmienimy definicję typu *xy*, nie będziemy musieli się martwić korektą długości odczytywanego bloku bajtów.

## Zapis danych do pamięci EEPROM

Ogólna postać funkcji dokonujących zapisu do pamięci EEPROM wygląda następująco:

```
void eeprom_write_typ(typ *zmienna_w_EEPROM, typ dane);
```

gdzie, podobnie jak w przypadku funkcji odczytujących dane, *typ* może być wyrażeniem byte, word, dword lub float. Istnieje także funkcja umożliwiająca zapisywanie bloku danych:

```
void eeprom_write_block(const void *źródło, void *adres_w_EEPROM, size_t ilość);
```

Jej zastosowanie jest podobne do analogicznej funkcji umożliwiającej odczyt danych z EEPROM. Zupełnie podobnie wygląda także operowanie na tych funkcjach:

```
eeprom_write_word(&x, a);  
eeprom_write_block(&punkt, &pkt, sizeof(punkt));
```

Pierwsze polecenie zapisuje do zmiennej znajdującej się w pamięci EEPROM wartość zmiennej *a*, drugie z kolei zapisuje dane ze zmiennej *pkt* do zmiennej w EEPROM o nazwie punkt.

## Modyfikowanie danych w pamięci EEPROM

Funkcje umożliwiające zapis do pamięci EEPROM dokonują zapisu przy każdym wywołaniu. Pamiętamy jednak, że każdy zapis wiąże się z przeprowadzeniem operacji kasowania, co nie tylko wydłuża zapis, ale przede wszystkim skraca żywotność tej pamięci. Jeśli może zdarzyć się sytuacja, w której zapisywana zmienna i zmienna w pamięci EEPROM mogą mieć taką samą wartość, to zapis nie jest konieczny. W takiej sytuacji warto wykorzystać grupę instrukcji warunkowo dokonujących zapisu. Działają one w ten sposób, że przed zapisem nowego bajtu najpierw odczytywana jest jego poprzednia wartość z pamięci EEPROM i porównywana z nowo zapisywaną. Jeśli obie są takie same, to zapis nie jest wykonywany. Dzięki temu wydłużamy żywotność pamięci oraz przyśpieszamy wykonywanie funkcji. Ogólna postać tych funkcji podobna jest do poprzednich:

```
void eeprom_update_typ(typ *zmienna_w_EEPROM, typ dane);  
void eeprom_update_block(const void *źródło, void *adres_w_EEPROM, size_t ilość);
```

Z tych funkcji możemy korzystać w każdej sytuacji, czas poświęcony na dodatkowy odczyt danych z EEPROM i ich porównanie jest bez znaczenia w porównaniu do czasu poświęconego na zapis (ten drugi jest zwykle tysiące razy dłuższy).

## Inne funkcje operujące na EEPROM

Funkcje zapisu/odczytu pamięci EEPROM na wstępie sprawdzają, czy pamięć nie jest zajęta operacją zapisu. Jeśli tak, to najpierw czekają na jej zakończenie. Ponieważ proces zapisu do pamięci EEPROM jest niezwykle powolny, może to wprowadzać długie opóźnienia. W każdej chwili możemy sprawdzić, czy aktualnie nie odbywa się zapis do pamięci za pomocą funkcji `eeprom_is_ready()`. Funkcja ta zwraca true, jeżeli aktualnie pamięć EEPROM nie jest zapisywana. Możemy także wstrzymać dalsze wykonywanie programu do czasu skończenia bieżącego zapisu za pomocą funkcji `eeprom_busy_wait()`.

### Wskazówka

 Warto pamiętać, że wszystkie funkcje zadeklarowane w `<avr\EEPROM.h>` nie sprawdzają, czy aktualnie nie jest zapisywana pamięć FLASH — należy więc z nich ostrożnie korzystać, pisząc *bootloader* i programy, w których pamięć FLASH może być modyfikowana.

W mikrokontrolerach AVR nie jest możliwe jednoczesne programowanie pamięci FLASH i EEPROM.

# Techniki wear leveling

Producent gwarantuje, że każdą komórkę pamięci EEPROM możemy zapisać co najmniej 100 000 razy. Jednak czasami wartość ta może okazać się za mała. Możemy ją zwiększyć, poświęcając dodatkowe komórki pamięci EEPROM do przechowywania danych. Ogólnie, zwiększając liczbę komórek przeznaczonych na zapis zmiennej n-razy, zwiększamy także n-razy liczbę możliwych zapisów. Niestety, techniki *wear leveling* nie są zaimplementowane w AVR-libc. W tym podrozdziale zostaną pokazane dwie z nich, co przy okazji zilustruje zasady posługiwania się pamięcią EEPROM. Najpierw rozważmy prosty przypadek. Założymy, że chcemy przechowywać zmienną o typie `unsigned int`. Standardowe gwarantowane 100 000 zapisów nam nie wystarcza, ale obliczyliśmy, że w trakcie okresu eksploatacji projektowanego urządzenia liczba zapisów do pamięci EEPROM naszej zmiennej nie przekroczy 1 000 000. Co więcej, wiemy, że nasza zmienna nigdy nie przyjmuje wartości 0xFFFF. Wszystkie inne wartości dla zmiennej są dozwolone. Z tą wiedzą możemy rozwiązać nasz problem, implementując bufor cykliczny. Ponieważ nasza zmienna z pewnością nie przyjmie wartości 0xFFFF, możemy tę liczbę wykorzystać jako wskaźnik wolnej pozycji w buforze (tak się dobrze składa, że 0xFFFF będą miały skasowane komórki pamięci EEPROM). Przykładowa implementacja przedstawionego algorytmu jest następująca:

```
#define ILE_WYDLUZYC 10
EEMEM unsigned int bufor[ILE_WYDLUZYC];

void skasuj_bufor()
{
    int i=0;
    while(i<ILE_WYDLUZYC) eeprom_write_word(&bufor[i], 0xFFFF);
}

void zapisz_bufor(unsigned int wartosc)
{
    int i=0;
    while((i<ILE_WYDLUZYC) && (eeprom_read_word(&bufor[i])!=0xFFFF)) i++;
    if(i==ILE_WYDLUZYC) skasuj_bufor();
    eeprom_write_word(&bufor[i%ILE_WYDLUZYC], wartosc);
}

unsigned int odczytaj_bufor()
{
    int i=0;
    unsigned int wartosc=eeprom_read_word(&bufor[i]);
    while((i<ILE_WYDLUZYC) && (eeprom_read_word(&bufor[i])!=0xFFFF))
    {
        wartosc=eeprom_read_word(&bufor[i]);
        i++;
    }
    return wartosc;
}
```

Na początku zadeklarowaliśmy tablicę przechowującą wartości zapisywanej zmiennej. Ponieważ pamięć EEPROM domyślnie zawiera wartości 0xFF, nie musimy jej inicjalizować. Stworzyliśmy dwie funkcje umożliwiające wygodny dostęp do danych: pierwsza,

zapisz\_bufor, zapisuje w pamięci EEPROM podaną przez nas wartość, druga, odczytaj\_bufor, zwraca wartość wcześniej zapisaną. Przyjrzyjmy się dokładniej, jak wygląda procedura zapisu. W pierwszym etapie szukamy wolnej pozycji w tablicy bufor — wolna pozycja sygnalizowana jest odczytaniem wartości 0xFFFF. Po jej napotkaniu dokonujemy operacji zapisu za pomocą poznanej już funkcji eeprom\_write\_word. Jeśli cała tablica bufor jest już wypełniona danymi, to wywołujemy procedurę skasuj\_bufor, której celem jest wpisanie wartości 0xFFFF do wszystkich pozycji bufora. Dopiero potem na pozycję o indeksie 0 wpisywana jest nowa wartość (zmienna i w przypadku zapełnienia bufora będzie miała wartość równą ILE\_WYDLUZYC, co po operacji modulo da 0). Widzimy więc, że każda pozycja tablicy bufor będzie zapisywana ILE\_WYDLUZYC razy mniej niż liczba wywołań funkcji zapisz\_bufor. Spowoduje to odpowiednie zwiększenie liczby możliwych zapisów. Odczytanie wcześniej zapisanych danych nie stanowi problemu. Poszukujemy ostatniej zapisanej pozycji (następna po niej ma wartość 0xFFFF lub i==ILE\_WYDLUZYC) i zwracamy ją jako wynik funkcji.

Powyższy przykład był prosty, gdyż jedną z wartości zapisywanej zmiennej mogliśmy wykorzystać jako wskaźnik wolnej pozycji w buforze. Nie zawsze jednak mamy do czynienia z tak komfortową sytuacją. Jeśli żadna wartość zmiennej nie może posłużyć jako wyróżnik wolnej pozycji w buforze, to możemy stworzyć drugi bufor, który będzie zawierał wskaźnik wolnej pozycji w buforze, w którym przechowujemy wartość interesującą nas zmiennej:

```
#define ILE_WYDLUZYC    10
EEMEM unsigned int bufor[ILE_WYDLUZYC];
EEMEM unsigned char wskaznik[ILE_WYDLUZYC];

void skasuj_wskaznik()
{
    int i=0;
    while(i<ILE_WYDLUZYC) eeprom_write_byte(&wskaznik[i], 0xFF);
}

void zapisz_bufor(unsigned int wartosc)
{
    int i=0;
    while((i<ILE_WYDLUZYC) && (eeprom_read_byte(&wskaznik[i])!=0xFF)) i++;
    if(i==ILE_WYDLUZYC) skasuj_wskaznik();
    eeprom_write_byte(&wskaznik[i%ILE_WYDLUZYC], 0);
    eeprom_write_word(&bufor[i%ILE_WYDLUZYC], wartosc);
}

unsigned int odczytaj_bufor()
{
    int i=0;
    unsigned int wartosc=eeprom_read_word(&bufor[i]);
    while((i<ILE_WYDLUZYC) && (eeprom_read_byte(&wskaznik[i])!=0xFF))
    {
        wartosc=eeprom_read_word(&bufor[i]);
        i++;
    }
    return wartosc;
}
```

W przedstawionym przykładzie stworzyliśmy bufor pomocniczy, w którym przechowujemy informację o zajętych pozycjach tablicy bufor. Wolna pozycja oznaczona została wartością 0xFF, a zajęta wartością 0. Wydawać by się mogło, że poświęcanie całego bajtu na przechowywanie tylko dwóch wartości jest marnotrawstwem. Jednak musimy zapewnić, że liczba zapisów do komórek pamięci przechowujących tablicę wskaznik będzie taka sama jak liczba zapisów do tablicy bufor. W przeciwnym wypadku komórki te zużywałyby się szybciej i nie moglibyśmy uzyskać wiarygodnych wskazań. Jednak w nowszych mikrokontrolerach AVR zapis do pamięci EEPROM może być operacją rozdzielnią z kasowaniem komórki. Ponieważ tylko operacja kasowania uszkadza komórkę, możemy więc jako znacznik zajętości w tablicy bufor wykorzystać pojedyncze bity z tablicy wskaznik. Niestety, w takiej sytuacji musimy napisać własną procedurę zapisu do pamięci EEPROM, gdyż wszystkie procedury biblioteczne kasują bity *EEMPI* i *EEMPO* (o ile takie w posiadanym procesorze występują), wymuszając w ten sposób wykonanie operacji jednoczesnego kasowania i zapisu.

## Rozdział 8.

# Dostęp do pamięci FLASH

Podczas pisania dowolnego programu na mikrokontroler często zachodzi potrzeba przechowywania pewnych stałych, których program nigdy nie modyfikuje, a są one potrzebne do jego działania. Np. łącząc mikrokontroler z wyświetlaczem LCD, chcemy wyświetlić menu albo pewne komunikaty. Naturalnym sposobem, aby to osiągnąć, wydaje się być zadeklarowanie odpowiednich zmiennych jako stałych, np.:

```
const char tekst[]="Menu";
```

definiuje stałą tekst będącą tablicą o typie char, zawierającą łańcuch. Jednak po skompilowaniu programu z tak umieszczonymi w nim stałymi pewnie z niewielkim zaskoczeniem odkryjemy, że spora (o ile nie cała) pamięć RAM mikrokontrolera jest zajęta.



Pamiętajmy, że mikrokontrolery AVR posiadają kilka rozłącznych przestrzeni adresowych, a kompilator gcc nie potrafi ich prawidłowo obsługiwać.

Pamiętajmy też, że modyfikator `const` informuje kompilator, że tak zdefiniowana stała nie może być modyfikowana, nie mówi nic o miejscu, gdzie ma się ona znaleźć. Stąd też skrypty inicjalizacyjne programu dodawane automatycznie na etapie linkowania kopią tak zadeklarowane dane z pamięci FLASH do pamięci SRAM, gdzie jako stałe mogą być używane w programie. Taki sposób działania wydaje się być marnotrawstwem pamięci, lecz obecne wersje kompilatora gcc nie potrafią operować na zmiennych umieszczonych w pamięci FLASH. To domyślne zachowanie kompilatora i linkera możemy jednak zmienić. W tym celu musimy poinformować linker, gdzie ma umieścić nasze stałe. Dokonujemy tego dwuetapowo — najpierw musimy stałą zadeklarować tak, aby znalazła się w odpowiedniej sekcji, następnie musimy poinformować linker, gdzie ta sekcja ma się znaleźć. Jeśli korzystamy z narzędzi z pakietu WinAVR oraz dostarczonych z nim standardowych skryptów linkera, to o prawidłowe umieszczenie sekcji nie musimy się martwić. Jedyne, co musimy zrobić, to zadeklarować zmienną z atrybutem `_progmem`:

```
const char __attribute__((__progmem__)) xx[]="xxx";
```

deklaruje zmienną łańcuchową `xx`, informując linker, że ma się ona znaleźć w pamięci FLASH. Niestety, do tak zadeklarowanej zmiennej nie możemy odwoływać się za

**pomocą standardowych operatorów języka C.** Przy próbie wykonania jakichkolwiek operacji na takiej zmiennej kompilator co prawda nie zgłosi żadnych problemów, lecz zostaną one przeprowadzone nieprawidłowo. Zmienna zostanie potraktowana jak zwykła zmienna znajdująca się w pamięci SRAM mikrokontrolera, zostaną więc odczytane przypadkowe dane. Aby „dostać się” do danych zgromadzonych w tak zadeklarowanych zmiennych, musimy posłużyć się specjalnymi funkcjami. Tu z pomocą ponownie przychodzi nam AVR-libc; w pliku nagłówkowym `<avr/pgmspace.h>` znajdziemy deklaracje różnych funkcji i typów umożliwiające w wygodny sposób deklarowanie i dostęp do zmiennych umieszczonych w pamięci FLASH.

## Typy danych związane z pamięcią FLASH

W pliku nagłówkowym `<avr/pgmspace.h>` zdefiniowano makrodefinicję `PROGMEM`. Jej umieszczenie przed dowolnym typem danych powoduje, że reprezentowana przez niego zmienna zostanie umieszczona w pamięci FLASH. Definicja ta jest rozwijana do znanego nam już atrybutu `_attribute__((__progmem))`, jest tylko krótsza i przez to wygodniejsza w użyciu. Jeśli chcemy, aby dana stała umieszczona została w pamięci FLASH, to wystarczy jej deklarację poprzedzić makrem `PROGMEM`:

```
const PROGMEM char menu[]="Menu";
const int x=10;
```

Zdefiniowano także odpowiedniki standardowych typów zmiennych dotyczące pamięci FLASH. Aby otrzymać typ, którego wartość przechowywana jest w pamięci FLASH, wystarczy typ poprzedzić prefiksem `prog_`, np. odpowiednikiem typu `char` będzie `prog_char`, `int` — `prog_int` itd. W przypadku typów złożonych sytuacja nie wygląda tak prosto. Dla przykładu stwórzmy tablicę zawierającą elementy menu:

```
char *Menu[3]= {"Otworz", "Zamknij", "Zakoncz"};
```

Tablica ta zostanie utworzona w pamięci SRAM, dla zaoszczędzenia miejsca spróbujmy umieścić ją w pamięci FLASH. Wydawałoby się, że najprościej będzie zmienić typ elementów z `char*` na `prog_char*`:

```
prog_char *Menu[3]= {"Otworz", "Zamknij", "Zakoncz"};
```

Jednak ku naszemu zaskoczeniu nie przynosi to pożądanych rezultatów. Elementy tablicy ciągle kopiowane są do pamięci SRAM, gdyż atrybut `_progmem` powoduje, że co najwyżej wskaźnik, a nie wskazywany przez niego obiekt znajduje się w pamięci FLASH. Możemy poinstruować kompilator, że sama tablica zawierająca wskaźniki znajduje się w pamięci FLASH:

```
prog_char *Menu[3] PROGMEM = {"Otworz", "Zamknij", "Zakoncz"};
```

W tym wypadku zużycie pamięci istotnie się zmniejszyło (tablica `Menu` znajduje się teraz w pamięci FLASH mikrokontrolera), lecz wskazywane w niej obiekty ciągle znajdują się w pamięci SRAM. Aby temu zapobiec, musimy najpierw stworzyć w pamięci FLASH elementy tablicy `Menu`:

```
const prog_char m1[]="Otworz";
const prog_char m2[]="Zamknij";
const prog_char m3[]="Zakończ";
```

a następnie stworzyć samą tablicę, jednocześnie inicjalizując ją odpowiednimi wartościami:

```
const prog_char *Menu[3] PROGMEM ={m1, m2, m3};
```

W ten sposób cała tablica zostanie utworzona w pamięci FLASH. Niestety, nie ma innej możliwości, aby poinstruować kompilator, gdzie powinny znaleźć się podawane literały.

## Odczyt danych z pamięci FLASH

Jak wcześniej wspomniano, do danych znajdujących się w pamięci FLASH mikrokontrolera nie mamy dostępu za pomocą standardowych operatorów języka C. Jakakolwiek próba odwołania się do wcześniej zdefiniowanych struktur danych spowoduje próbę odczytania ich tak, jakby znajdowały się w pamięci SRAM, a nie w pamięci FLASH, w efekcie uzyskamy błędne dane. Aby umożliwić dostęp do danych znajdujących się w pamięci FLASH, zdefiniowano makrodefinicje, zwracające odpowiednio bajt, słowo, podwójne słowo lub zmienną o typie float:

- ◆ pgm\_read\_byte(adres) — zwraca bajt danych spod podanego adresu.
- ◆ pgm\_read\_word(adres) — zwraca dwa bajty (słowo).
- ◆ pgm\_read\_dword(adres) — zwraca 4 bajty (dwa słowa).
- ◆ pgm\_read\_float(adres) — zwraca wartość o typie float.

adres jest 16-bitowym wskaźnikiem do pamięci FLASH. Stąd przedstawione makrodefinicje mogą zwracać dane znajdujące się w obszarze pierwszych 64 kB pamięci FLASH.



W mikrokontrolerach posiadających większą ilość pamięci makra te nie umożliwiają dostępu do danych leżących powyżej granicy 64 kB.

Standardowe skrypty linkera umieszczają sekcję danych tuż za obszarem wektorów przerwań mikrokontrolera, a więc w początkowym obszarze pamięci FLASH. Stąd też jeśli ilość danych zapamiętyanych w pamięci FLASH nie przekracza ok. 64 kB, powyższe makrodefinicje są w zupełności wystarczające i będą działały prawidłowo niezależnie od ilości dostępnej pamięci FLASH w użytkowym mikrokontrolerze.



Pewną ostrożność należy zachować, wykorzystując wyżej wymienione funkcje w programie bootloadera. Jeśli kod bootloadera wraz z danymi, do których chcemy uzyskać dostęp, znajduje się powyżej granicy 64 kB, powyższe funkcje nie będą działać poprawnie.

## Dostęp do pamięci FLASH >64 kB

W sytuacji kiedy mikrokontroler posiada >64 kB pamięci FLASH i jednocześnie dane, do których chcemy uzyskać dostęp, mogą znajdować się powyżej tej granicy, musimy użyć podobnych makr, tyle że z sufiksem \_far:

- ◆ pgm\_read\_byte\_far(adres)
- ◆ pgm\_read\_word\_far (adres)
- ◆ pgm\_read\_dword\_far (adres)
- ◆ pgm\_read\_float\_far (adres)

W tym wypadku *adres* jest 32-bitową zmienną o typie `uint32_t`. Makra te nie rozwiązują, niestety, wszystkich problemów. Operator pobrania adresu w języku C na AVR (&) zwraca 16-bitowy wskaźnik. Nie jest więc możliwe pobranie w ten sposób adresu danych znajdujących się powyżej granicy 64 kB. Tutaj z pomocą przychodzi nam biblioteka o nazwie *morepgmspace.h*<sup>1</sup>, napisana przez Carlosa Lamasa. Do tej pory nie jest ona dołączona do oficjalnej dystrybucji AVR-libc. Zawiera ona deklaracje funkcji umożliwiających dostęp do pamięci FLASH leżącej poza granicą 64 kB. Dla nas najbardziej jednak interesująca jest makrodefinicja `GET_FAR_ADDRESS`:

```
#define GET_FAR_ADDRESS(var)
({
    uint_farptr_t tmp;
    __asm__ __volatile__(
        "ldi    %A0, lo8(%1)"      "\n\t"
        "ldi    %B0, hi8(%1)"      "\n\t"
        "ldi    %C0, hh8(%1)"      "\n\t"
        "clr    %D0"              "\n\t"
        :
        "=d"  (tmp)               \
        :
        "p"   (&(var))            \
    );
    tmp;
})
```

Makro to umożliwia pobranie adresu danych leżących w dowolnym miejscu pamięci FLASH, adres przypisywany jest zmiennej typu `int32_t`. Posiadając 32-bitowy adres zmiennej, łatwo możemy odczytać jej zawartość przy pomocy wymienionych wcześniej funkcji z sufiksem `_far`, podobnie jak to robiliśmy w przypadku zwykłych funkcji realizujących dostęp do pamięci FLASH.

---

<sup>1</sup> Całą bibliotekę można pobrać pod adresem <https://savannah.nongnu.org/patch/?6352>.

## Rozdział 9.

# Interfejs XMEM

W mikrokontrolerach ATMega64 i wyższych istnieje możliwość podłączenia za pomocą sprzętowego interfejsu XMEM (ang. *External Memory Interface*) zewnętrznej pamięci SRAM. Daje to możliwość rozszerzenia dostępnej ilości pamięci RAM. Interfejs ten umożliwia bezpośrednie Interfejs XMEM podłączenie pamięci SRAM o pojemności 0-64 kB. Pamięci o większej pojemności muszą być podzielone na banki, których obsługa nie jest bezpośrednio zaimplementowana przez interfejs XMEM i pozostaje w gestii programisty. Interfejs ten nie generuje cykli odświeżania pamięci, nie jest więc możliwe w prosty sposób podłączenie pamięci dynamicznej RAM. Interfejs ten umożliwia:

- ◆ podłączenie zewnętrznej pamięci statycznej RAM (SRAM);
- ◆ określenie czasu dostępu do pamięci poprzez wprowadzenie cykli oczekiwania (ang. *wait states*);
- ◆ niezależne określanie opóźnień dostępu dla różnych sekcji pamięci zewnętrznej, co umożliwia wykorzystanie pamięci o różnych czasach dostępu;
- ◆ zastosowanie niewykorzystanych linii adresowych interfejsu XMEM jako normalnych portów IO procesora;
- ◆ użycie tzw. *bus keepers* dla minimalizacji poboru prądu.

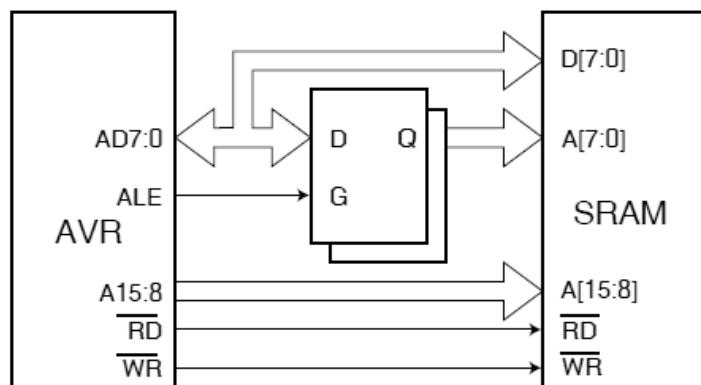
Do podłączenia pamięci wykorzystywanych jest kilka linii kontrolnych:

- ◆ linie adresowe AD0 – AD7, które są multipleksowane z liniami danych D0 – D7,
- ◆ linie adresowe AD8 – AD15,
- ◆ linia strobu odczytu RD (aktywna w stanie niskim),
- ◆ linia strobu zapisu WR (aktywna w stanie niskim),
- ◆ linia strobu zapisu do rejestru zatrzaskowego (ALE) — zmiana stanu z wysokiego na niski powoduje zatrzaśnięcie stanu linii adresowych AD0 – AD7, jej stan niski wskazuje, że na tych liniach znajdują się dane D0 – D7.

Sposób podłączenia zewnętrznej pamięci pokazany jest na rysunku 9.1. Ze względu na prędkość procesora ważny jest dobór odpowiednio szybkiego rejestru zatrzaskowego. Atmel zaleca, aby był to układ serii 74AHC, np. 74AHC573. Ma to szczególne znaczenie

podczas zasilania procesora niższym napięciem. Zastosowanie zbyt wolnego rejestru zatrzaskowego spowoduje nieprawidłowe działanie interfejsu, co będzie się objawało niestabilną pracą pamięci i częstymi przekłamaniami zapisywanych i odczytywanych danych.

**Rysunek 9.1.**  
Sposób podłączenia  
zewnętrznej  
pamięci SRAM  
do mikrokontrolera  
AVR



Aby linie AD0 – AD7 zawsze miały ustalony stan logiczny, możemy aktywować na nich rezystory podciągające poprzez wpisanie 1 na pozycjach portu IO odpowiadających tym liniom. Możemy także uaktywnić funkcję *bus keeper*, która utrzymuje na tych liniach ostatni występujący na nich stan logiczny. Ma to na celu minimalizację poboru prądu dzięki wyeliminowaniu stanów przejściowych.

Wszystkie funkcje interfejsu XMEM kontrolowane są za pomocą trzech rejestrów kontrolnych — MCUCR (ang. *MCU Control Register*) oraz XMCRA i XMCRB (ang. *External Memory Control Register A & B*). W MCUCR znajdują się bity kontrolne umożliwiające włączenie interfejsu (SRE — ang. *External SRAM/XMEM Enable*) i wybór liczby cykli oczekiwania. Odblokowanie interfejsu następuje poprzez wpisanie 1 do bitu SRE rejestru MCUCR. Spowoduje to uaktywnienie alternatywnych funkcji portów IO wykorzystywanych przez ten interfejs. Od tego momentu o funkcji i kierunku tych portów decyduje interfejs XMEM. Jedyne, co możemy zrobić, to aktywować przypisane do nich rezystory podciągające poprzez wpisanie wartości 1 do rejestru PORT, na którym znajdują się linie adresowe AD0 – AD7.

Interfejs XMEM możemy aktywować i dezaktywować w dowolnym momencie wykonywania programu. Jeśli jednak chcemy, aby kompilator mógł umieścić w zewnętrznej pamięci SRAM stos lub segmenty danych, to aktywacja interfejsu XMEM musi nastąpić przed wykonaniem kodu inicjalizacyjnego programu, a więc w sekcji *.init1* lub *.init0*. Kod aktywujący ten interfejs wygląda następująco:

```
void init_XMEM() __attribute__ ((naked)) __attribute__ ((section (.init1)));
void init_XMEM()
{
    MCUCR |= _BV(SRE);
}
```

Opcjonalnie możemy go wzbogacić o ustalenie liczby cykli oczekiwania wykorzystywanych do wydłużenia czasu dostępu do pamięci, podziału pamięci na obszary o różnych

czasach dostępu oraz aktywacji funkcji *bus keeper* i dezaktywacji niektórych linii adresowych. Ta ostatnia możliwość jest cenna, kiedy podłączamy niewielką pamięć zewnętrzną, niewykorzystującą wszystkich linii adresowych AD8 – AD15. W tym przypadku niewykorzystane linie adresowe mogą być wykorzystane jako zwykłe linie IO procesora (będą one wtedy pod pełną kontrolą rejestrów PORT i DDR). Funkcja:

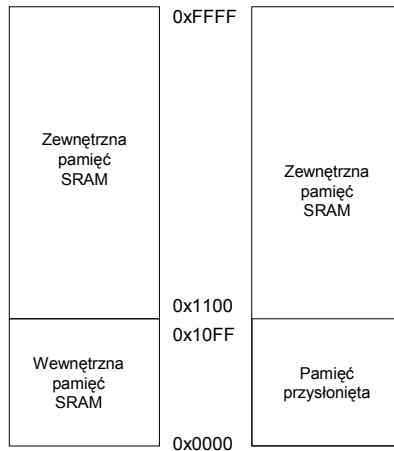
```
void init_XMEM() __attribute__ ((naked)) __attribute__ ((section ("._init1")));
void init_XMEM()
{
    XMCRB |= _BV(XMBK) | _BV(XMM2) | _BV(XMM1) | _BV(XMM0);
    MCUCR |= _BV(SRE);
}
```

spowoduje odblokowanie interfejsu pamięci zewnętrznej, uruchomienie funkcji *bus keeper*, dodatkowo zablokowane zostaną linie adresowe AD8 – AD15, dzięki czemu przypisane im piny portu IO będzie można wykorzystać w programie. Ograniczy to ilość zewnętrznej pamięci SRAM do 256 bajtów.

Korzystając z zewnętrznej pamięci SRAM, musimy pamiętać o kilku rzeczach. Po pierwsze, zawsze pamięć wewnętrzna procesora przesyłania adresy pamięci zewnętrznej, co pokazano na rysunku 9.2. Stąd też np. w ATMega128, która dysponuje 4 kB wewnętrznej pamięci RAM, podłączenie 64 kB zewnętrznej pamięci spowoduje, że łącznie będziemy mieli tylko 64 kB, a nie, jak byśmy się spodziewali, 68 kB. Pierwsze 4 kB pamięci zewnętrznej zostaną przysłonięte przez pamięć wewnętrzna procesora. Można dostać się do tej przesłoniętej pamięci, maskując odpowiednio bity adresowe pamięci dzięki rejestrowi XMCRB i bitom XMM0 – 2, określającym liczbę wykorzystanych najstarszych bitów adresu (linii AD8 – AD15). Wymaga to jednak specjalnych funkcji umożliwiających dostęp do tego bloku pamięci i nie jest w żaden sposób wspierane przez bibliotekę AVR-libc i kompilator.

**Rysunek 9.2.**

Mapa pamięci i przesyłanie pamięci zewnętrznej przez wewnętrzną pamięć procesora



Jeżeli podłączymy mniejszą pamięć, np. 32 kB, możemy zmapować ją w przestrzeń adresową 0x8000 – 0xFFFF, co da nam dostęp do całości pamięci. W takiej sytuacji powstanie jednak „dziura” w adresacji — adresy pamięci wewnętrznej w przypadku procesora ATMega128 kończą się na 0x10FF, a adresy komórek pamięci zewnętrznej rozpoczynają się dopiero od 0x8000. Nie stanowi to jednak istotnego utrudnienia przy pisaniu programu.

W przypadku gdy wykorzystujemy zewnętrzną pamięć o pojemności 64 kB, dostęp do „utraconych” 4 kB pamięci zewnętrznej możemy łatwo uzyskać poprzez proste funkcje napisane w języku C, wykorzystujące możliwość maskowania linii adresowych AD8 – AD15:

```
char Read_4kB(size_t adres)
{
    DDRC=0xFF;
    PORTC=0x00;           //Pin odpowiadający linii adresowej AD15 będzie wyjściem w stanie 0
    XMCRB|= _BV(XMM0);   //Zamaskuj linię adresową AD15 - jej stan będzie równy 0
    char byte=*((volatile char*)(adres | 0x8000)); //Odczytaj bajt
    XMCRB&=~_BV(XMM0);   //Przywróć normalną funkcję linii AD15
    return byte;
}

void Write_4kB(size_t adres, char byte)
{
    DDRC=0xFF;
    PORTC=0x00;           //Pin odpowiadający linii adresowej AD15 będzie wyjściem w stanie 0
    XMCRB|= _BV(XMM0);   //Zamaskuj linię adresową AD15 - jej stan będzie równy 0
    *((volatile char*)(adres | 0x8000))=byte; //Zapisz bajt
    XMCRB&=~_BV(XMM0);   //Przywróć normalną funkcję linii AD15
}
```

Powyższe funkcje w celu dostępu do przesłoniętego bloku zewnętrznej pamięci SRAM wykorzystują możliwość przejęcia kontroli nad liniami adresowymi AD8 – AD15 za pomocą bitów XMM0 – XMM2 rejestru XMCRB. Na pierwszym etapie linie IO, na których znajdują się linie adresowe AD8 – AD15, są ustawiane na 0. Następnie blokowana jest kontrola interfejsu XMEM nad linią adresową AD15, dzięki czemu zamienia się ona w zwykły pin portu IO, który wcześniej ustawiony został na 0. Dzięki temu najstarszy bit adresu zewnętrznej pamięci SRAM ma wartość 0 niezależnie od adresu generowanego przez interfejs XMEM procesora. Wykorzystujemy ten fakt, generując dla procesora adres leżący poza przestrzenią pamięci SRAM (ustawiamy najstarszy bit adresu), ale dla zewnętrznej pamięci SRAM bit AD15 ciągle ma wartość 0 — umożliwia to odczytanie wcześniej przesłoniętego przez wewnętrzną pamięć SRAM bloku pamięci. Po odczycie/zapisie przywracamy kontrolę interfejsu XMEM nad linią adresową AD15. W powyższych przykładach warto zastanowić się nad dokonanym rzutowaniem typów:

```
char byte=*((volatile char*)(adres | 0x8000));
```

Jak widzimy, zmienną adres traktujemy jak wskaźnik na typ char (który na AVR jest typem 8-bitowym). Co ciekawe, zastosowano modyfikator volatile. Jest to w tym przypadku niezbędne, gdyż jeśli wskaźnik nie byłby volatile, to kompilator miałby swobodę przemieszczenia operacji odczytu/zapisu w dowolne miejsce funkcji, co z kolei wiążałoby się z odczytem niewłaściwej komórki pamięci (odczyt/zapis musi być wykonany w chwili, kiedy mamy kontrolę nad linią AD15). Warto także wspomnieć o pewnych ograniczeniach powyższych funkcji. Jeśli w wykorzystującym je programie jednocześnie wykorzystujemy przerwania, w których korzystamy ze zmiennych globalnych, a w przypadku kiedy stos procesora przenieśliśmy do pamięci zewnętrznej, także w wypadku wykorzystywania zmiennych lokalnych, procedura obsługi przerwania nie będzie działać prawidłowo. Wynika to z faktu, że w trakcie wykonywania funkcji Read\_4kB i Write\_4kB nie możemy uzyskać dostępu do starszych 32 kB zewnętrznej pamięci SRAM

(linia adresowa AD15 na stałe jest ustawiona na zero). W efekcie niemożliwy jest dostęp do zmiennych przechowywanych w tej pamięci. Dodatkowo, jeśli w tym obszarze pamięci znajdzie się stos, to jakiekolwiek przerwanie lub instrukcje języka C związane z odłożeniem adresu na stosie spowodują nieprawidłowe działanie programu. Zapobiec temu możemy stosunkowo łatwo, zapewniając wykonanie całej operacji od chwili modyfikacji rejestru XMCRB do chwili przywrócenia jego poprzedniej wartości atomowo. Szerzej o atomowości kodu będzie wspomniane w rozdziale 13., poświęconym przerwaniom. Oczywiście, powyższe problemy nie wystąpią, jeśli stos i standardowe sekcje (.data, .bss) oraz ich podsekcje znajdują się w wewnętrznej pamięci SRAM mikrokontrolera. W takiej jednak sytuacji zewnętrzna pamięć SRAM nie może być bezpośrednio wykorzystywana w programie — uzyskamy do niej dostęp wyłącznie poprzez jawnie zdefiniowane wskaźniki, co jest sytuacją niezbyt wygodną.

## Wykorzystanie zewnętrznej pamięci SRAM w programie

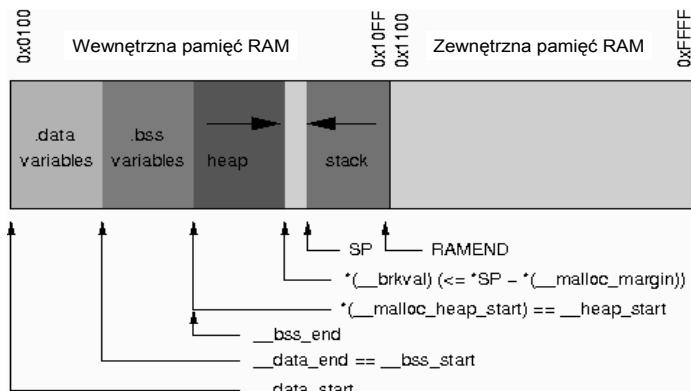
Powyżej przedstawiony został sposób dostępu do zewnętrznej pamięci SRAM poprzez specjalne funkcje. Nie jest on zbyt wygodny, a co gorsze, nie zwiększa on ilości dostępnej pamięci dla zmiennych, stosu i sterty. Domyślnie niezależnie od ilości dołączonej pamięci zewnętrznej wszystkie te struktury są umieszczone w wewnętrznej pamięci RAM procesora. Wynika to z faktu, że kompilator nic nie wie o szczegółach sprzętowych budowy naszego układu elektronicznego, w szczególności nie wie o tym, że dołączyliśmy zewnętrzną pamięć. Aby więc ją wykorzystać, musimy kompilator poinstruować o jej istnieniu i sami zadbać o odpowiednie rozmieszczenie sekcji pamięci. Aby tego dokonać, musimy wiedzieć, jak zorganizowana jest struktura pamięci w AVR-gcc i AVR-libc. Standardowo na początku pamięci (od adresu zależnego od liczby rejestrów IO procesora) umieszczona jest sekcja .data, zawierająca zmienne zainicjowane, po niej znajduje się sekcja .bss, zawierająca zmienne niezainicjowane (są one automatycznie zerowane), po niej znajduje się sterta (ang. *Heap*), wykorzystywana do dynamicznej alokacji pamięci. Sterta w miarę alokacji kolejnych bloków pamięci rozrasta się w kierunku wyższych adresów. Na końcu pamięci znajduje się stos, który z kolei rośnie w kierunku mniejszych adresów. Wynika z tego, że nigdy nie może dojść do kolizji zmiennych z sekcji .data i .bss ze zmiennymi alokowanymi dynamicznie na sterce. Może natomiast zdarzyć się sytuacja kolizji sterty ze stosem — w wyniku próby alokacji dużej ilości pamięci na sterce lub w wyniku dużego zapotrzebowania na stos (wykorzystanie rekurencji lub zmiennych lokalnych).



Mikrokontrolery AVR nie posiadają żadnych sprzętowych mechanizmów służących do detekcji i zapobiegania takim sytuacjom.

Informacje o metodach detekcji kolizji stosu ze zmiennymi i stertą znajdują się w rozdziale 6., poświęconym dynamicznej alokacji pamięci. Mapa pamięci została szczegółowo przedstawiona na rysunku 9.3. Poniżej przedstawione zostaną różne konfiguracje pamięci zewnętrznej i rozmieszczenia danych.

**Rysunek 9.3.**  
Mapa pamięci  
w środowisku  
AVR-libc



## Konfiguracja I — w pamięci zewnętrznej jest tylko sekcja specjalna

Dokładając zewnętrzną pamięć SRAM, możemy wykorzystać ją do przechowywania danych zawartych w poszczególnych sekcjach lub możemy stworzyć nową sekcję, do której trafią określone przez nas zmienne. Ta ostatnia sytuacja jest najprostsza. Jak pamiętamy, programista może tworzyć własne sekcje, a następnie przekazując odpowiednie parametry linkerowi, decydować o ich położeniu w pamięci. Nie zmieniając więc położenia standardowych sekcji (będą się one ciągle znajdowały wewnętrznej pamięci SRAM mikrokontrolera), możemy utworzyć nową sekcję, zawierającą zmienne, które powinny znaleźć się w zewnętrznej pamięci SRAM:

```
int xxx __attribute__ ((section (.XMEM)));
```

Powyższy kod spowoduje utworzenie zmiennej o nazwie `xxx` w sekcji o nazwie `.XMEM`. Pozostaje nam tylko zdefiniować położenie sekcji `.XMEM` w pamięci. W tym celu możemy posłużyć się konfiguratorem zawartym w AVR Studio (*Project/Configuration Options/Memory Settings*) lub dodać odpowiednie parametry do skryptu *Makefile*:

```
LDFLAGS += -Wl,-section-start=.XMEM=0x808000
```

Powyższa linia spowoduje utworzenie sekcji `.XMEM` w pamięci SRAM, począwszy od adresu `0x8000`. Pamiętajmy, że adresy pamięci SRAM zaczynają się od `0x800000`, tak więc do tego adresu bazowego musimy dodać adres, pod którym chcemy umieścić sekcję `.XMEM` (`0x8000`), co da nam w efekcie adres `0x808000`.

Aby za każdym razem nie używać `__attribute__`, co zmniejsza czytelność programu, możemy — podobnie jak twórcy AVR-libc dla innych sekcji — zdefiniować sobie wygodny alias:

```
#define XMEM __attribute__ ((section (.XMEM)))
```

Dzięki temu zmienne, które chcemy umieścić w sekcji `.XMEM`, możemy zadeklarować następująco:

```
int xx1 XMEM;  
XMEM int xx2;  
int XMEM xx3;
```

co znakomicie poprawia czytelność programu.

Powyższy sposób przypisania zmiennych do sekcji znajdującej się w pamięci zewnętrznej jest prosty i daje nam pełną kontrolę nad lokalizacją zmiennej.



**Pamiętajmy, że w takiej sekcji możemy deklarować wyłącznie zmienne globalne i statyczne.**

Próba zadeklarowania w ten sposób zmiennej lokalnej zakończy się komunikatem błędu:

```
error: section attribute cannot be specified for local variables
```

W ten sposób nie możemy powiększyć przestrzeni sterty, co uniemożliwia wykorzystanie zewnętrznej pamięci SRAM do przechowywania zmiennych alokowanych dynamicznie. Jesteśmy także limitowani ograniczoną pamięcią wewnętrzną mikrokontrolera w przypadku zmiennych lokalnych oraz faktem, że stos również znajduje się wewnętrznej pamięci mikrokontrolera, przez co ograniczone są możliwości rekurencji i wielokrotnego wywoływanego funkcji. Aby pokonać te problemy, możemy zmienić domyślną lokalizację sekcji.

## Konfiguracja II — wszystkie sekcje w pamięci zewnętrznej, stos w pamięci wewnętrznej



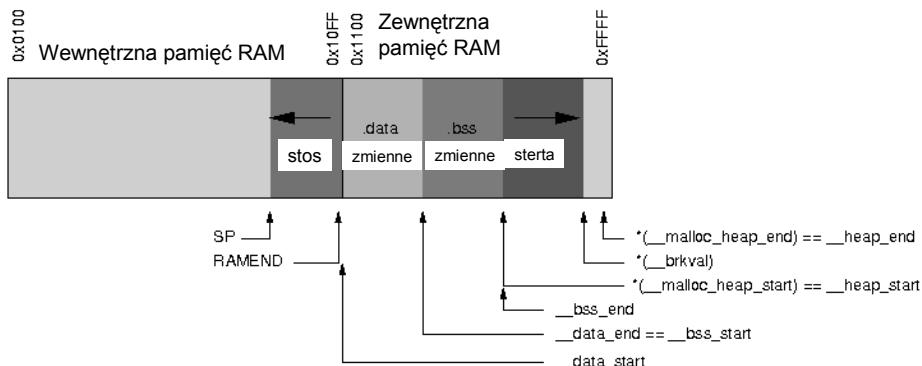
**Pamiętajmy jednak, że dostęp do zewnętrznej pamięci SRAM jest wolniejszy niż do wewnętrznej — przy każdej operacji odczytu/zapisu wprowadzany jest co najmniej jeden takt zegarowy opóźnienia. W efekcie raczej nie należy w pamięci zewnętrznej umieszczać stosu procesora. Pamiętać także należy, że w niektórych mikrokontrolerach rodzin AVR umieszczenie stosu w pamięci zewnętrznej nie jest możliwe.**

Spróbujmy rozmieścić sekcje tak, jak to przedstawiono na rysunku 9.4.

W tym celu bez zmian pozostawiamy inicjalizację stosu — domyślnie znajduje się on na końcu wewnętrznej pamięci SRAM i rośnie w dół. Zmienić musimy tylko położenie sekcji `.data`, `.bss` oraz sterty. Możemy to osiągnąć, przekazując linkerowi nowe położenia poszczególnych segmentów:

```
LDFLAGS += -Wl,-section-start,.data=0x801100,--defsym=__heap_end=0x80ffff
```

Powyższa linia powoduje, że sekcja `.data` zostanie umieszczona, począwszy od adresu 0x1100 (a więc tuż za końcem pamięci SRAM w mikrokontrolerze ATMega128), za nią znajdzie się sekcja `.bss`. Standardowo następny bajt po sekcji `.bss` należy już do sterty, nie musimy więc ustawać jej początku — zrobi to za nas automatycznie linker.



**Rysunek 9.4.** Alternatywne rozmieszczenie sekcji pamięci ze stosem znajdującym się w obszarze wewnętrznej pamięci SRAM mikrokontrolera

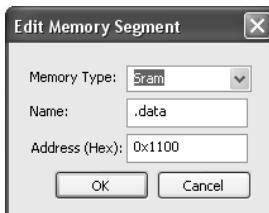
Jedynie, co musimy zmienić, to adres końca sterty. Możemy go zmienić w samym programie, zmieniając zmienną `_malloc_heap_end`, lub też definiując w trakcie linkowania symbol `_heap_end`. W naszym przypadku sterta będzie mogła się rozrastać aż do końca zewnętrznej pamięci SRAM (do adresu 0xFFFF).



Ponieważ definiujemy symbol, a nie sekcję, do jego adresu nie dodajemy offsetu określającego lokalizację sekcji (0x800000).

To samo możemy uzyskać za pomocą graficznych narzędzi zawartych w AVR Studio. W tym celu otwieramy znane nam już okno *Configuration Options* (z menu *Project*), najpierw klikamy na zakładkę *Memory Settings* i definiujemy segment `.data`. Wybieramy jego typ na SRAM i jako adres początku podajemy 0x1100 (AVR Studio samo doda odpowiedni offset informujący linker, że segment leży w pamięci RAM) — rysunek 9.5.

**Rysunek 9.5.**  
Wybór typu i adresu  
początkowego  
segmentu pamięci

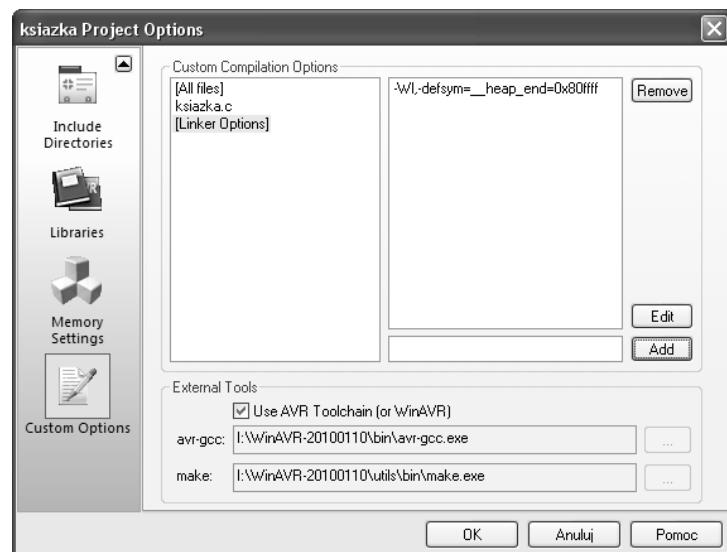


Następnie przechodzimy do zakładki *Custom Options* i z listy *Custom Compilation Options* wybieramy *[Linker Options]*, dodajemy parametr `-Wl,-defsym=_heap_end=0x80ffff`; w efekcie powinniśmy uzyskać obraz jak na rysunku 9.6.

Wygenerowany automatycznie plik *Makefile* będzie zawierał informacje o pożądanym przez nas rozmieszczeniu sekcji.

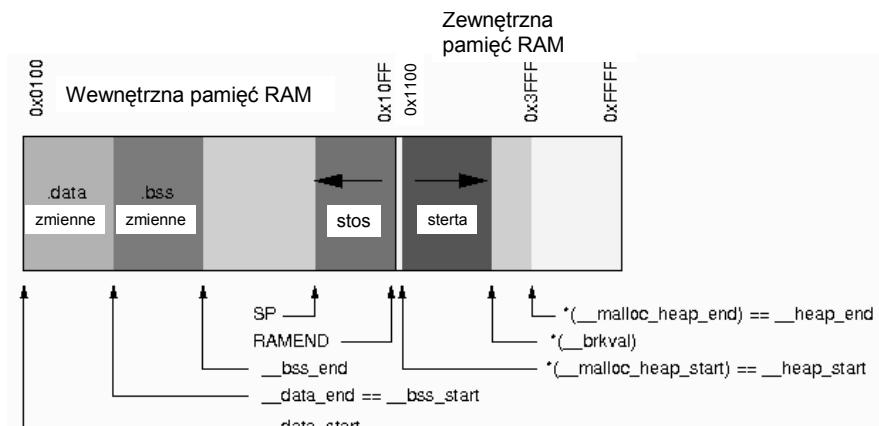
**Rysunek 9.6.**

Konfiguracja adresu końcowego sterty.  
Jej adres określany jest przez redefinicję symbolu `_heap_end`



### Konfiguracja III — w pamięci zewnętrznej umieszczona jest tylko sterta

Jak wspomniano powyżej, dostęp do zewnętrznej pamięci SRAM jest wolniejszy niż do wewnętrznej. Z tego powodu czasami lepiej jest umieścić wszystkie standardowe sekcje pamięci w wewnętrznej pamięci RAM, a w zewnętrznej trzymać tylko zmienne alokowane dynamicznie — zwykle są to większe struktury danych, na których prowadzamy stosunkowo rzadziej operacje i czas dostępu do nich nie jest aż tak krytyczny jak np. do zmiennej globalnej wykorzystywanej w procedurze obsługi przerwania. Nową konfigurację segmentów pamięci ilustruje rysunek 9.7.

**Rysunek 9.7.** Konfiguracja segmentów pamięci ze stertą znajdującej się w pamięci zewnętrznej

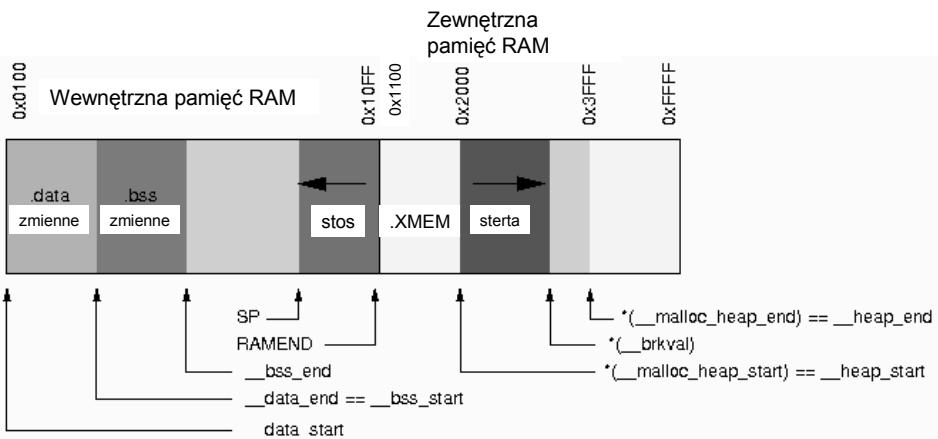
Uzyskanie takiej konfiguracji jest niezwykle proste. Wszystkie segmenty mają standarde pozycje, jedynie sterta jest przemieszczana do pamięci zewnętrznej. Najprościej to osiągnąć, przekazując linkerowi definicję symbolu `_heap_start` zawierającą adres początkowy sterty (czyli adres pierwszej komórki zewnętrznej pamięci SRAM). Musimy także zdefiniować symbol określający ostatni dostępny dla sterty adres (`_heap_end`) — w tym przypadku będzie to adres końca zewnętrznej pamięci SRAM:

```
LDFLAGS += -Wl,--defsym=_heap_start=0x801100,--defsym=_heap_end=0x80ffff
```

lub też przekazać parametr do linkera poprzez automatycznie generowany plik *Makefile* w AVR Studio w sposób analogiczny do opisanego powyżej.

## Konfiguracja IV — w pamięci zewnętrznej sterta i segment zdefiniowany przez programistę

Warto przemyśleć jeszcze jeden układ segmentów w pamięci. Zmienne programu możemy podzielić na takie, do których dostęp jest krytyczny czasowo (np. wspomniane zmienne wykorzystywane w procedurach obsługi przerwań), i takie, do których dostęp może odbywać się stosunkowo wolniej. W poprzednim przykładzie tylko zmienne alokowane dynamicznie (na stercie) były umieszczane w pamięci zewnętrznej, a wszystkie inne w pamięci wewnętrznej. W tym przykładzie także zmienne globalne i statyczne (zarówno globalne, jak i lokalne) będą mogły być umieszczone w pamięci zewnętrznej. W tym celu zdefiniujemy własny segment, nazwany `.XMEM`. Zmienna, której nadamy atrybut `.XMEM`, znajdzie się w efekcie w pamięci zewnętrznej, zwalniając w ten sposób cenne miejsce w pamięci wewnętrznej procesora. Nową mapę pamięci ilustruje rysunek 9.8.



**Rysunek 9.8.** Mapa pamięci, w której część zmiennych trafi do specjalnego segmentu pamięci zewnętrznej o nazwie `.XMEM`

Niestety, realizacja tego celu nie jest tak prosta jak poprzednio. Napotykamy tu na kilka problemów. Po pierwsze, symbol `_heap_start` domyślnie wskazuje na koniec segmentu `.bss`. Co prawda wiemy już, jak zmienić wartość tego symbolu, ale nie wiemy, jaką wartością go zainicjalizować — aby to zrobić, musielibyśmy wiedzieć, jaka jest długość segmentu `.XMEM`. Możemy co prawda umieścić stertę, począwszy od adresu,

co do którego istnieje pewność, że nie jest zajęty przez segment `.XMEM`, ale wiązałoby się to albo z marnotrawstwem pamięci, albo pewnymi niewygodami. Musielibyśmy skompilować program, sprawdzić, ile miejsca zajmuje segment `.XMEM`, a następnie poprawić wartość symbolu `_heap_start` i ponownie zlinkować program. W dodatku po każdej zmianie wielkości segmentu `.XMEM` (dodanie lub usunięcie z niego zmiennej) cały proces należałoby powtórzyć. Na szczęście, istnieje jednak inne rozwiązanie.



Pokazane w dalszej części rozdziału operacje wymagają modyfikacji plików konfiguracyjnych linkera.

Analogicznie jak w przypadku standardowych sekcji, dla nowo utworzonej sekcji `.XMEM` stworzymy symbol określający adres końca tej sekcji. Wartość tego symbolu przypiszemy symbolowi `_heap_start`, dzięki czemu sterta będzie rozpoczęta się tuż po segmencie `.XMEM`. Niestety, to z pozoru proste działanie wymaga modyfikacji domyślnych skryptów linkera. Użyte skrypty linkera zależą od architektury procesora. Do tej pory w przypadku rodziny AVR istnieje kilkanaście różnych architektur, które przedstawiono w tabeli 9.1. Pierwszym etapem będzie więc identyfikacja, do której architektury należy użyty przez nas procesor. Architekturę możemy określić pośrednio na podstawie budowy procesora:

- ◆ `avr1` — najprostszy rdzeń, nie jest wspierana przez avr-gcc;
- ◆ `avr2` — prosty rdzeń, z maksymalnie 8 kB pamięci;
- ◆ `avr25` — rdzeń z maksymalnie 8 kB pamięci FLASH, dodatkowo obsługa instrukcji `MOVW` i `LPM Rx,Z[+]`;
- ◆ `avr3` — prosty rdzeń z FLASH o pojemności 16 – 64 kB;
- ◆ `avr31` — prosty rdzeń z maksymalnie 128 kB FLASH;
- ◆ `avr35` — rdzeń z 16 – 64 kB FLASH, obsługa instrukcji `MOVW` i `LPM Rx,Z[+]`;
- ◆ `avr4` — rozszerzony rdzeń z maksymalnie 8 kB FLASH;
- ◆ `avr5` — rozszerzony rdzeń z maksymalnie 16 kB FLASH;
- ◆ `avr51` — rozszerzony rdzeń z maksymalnie 128 kB FLASH;
- ◆ `avr6` — rozszerzony rdzeń z maksymalnie 256 kB FLASH.

Architektury avrxmega związane są z różnymi wersjami procesorów rodzinny XMega.

Po określeniu używanej przez nasz procesor architektury musimy odnaleźć domyślnie używany dla niej skrypt linkera. Wszystkie skrypty znajdują się w katalogu `WinAVR\avr\lib\ldscripts`. Jak widać, dla każdej architektury stworzonych jest pięć różnych skryptów linkera. Wywoływanie są one w zależności od pewnych parametrów linkowania programu, jednak śmiało możemy założyć, że domyślnie wywoływany jest skrypt z rozszerzeniem `.x`. Do tej pory dla ilustracji wykorzystania zewnętrznej pamięci SRAM wykorzystywaliśmy procesor ATMega128. Jak widać z tabeli 9.1, należy on do architektury `avr5/avr51`, znajdujemy więc plik o nazwie `avr51.x` i kopujemy go do katalogu projektu (do katalogu, w którym znajduje się skrypt `Makefile`) jako `avr51xmemp.x`.

**Tabela 9.1.** Architektury procesorów AVR

<b>Architektura</b>	<b>Typy procesorów</b>
avr1	at90s1200, attiny11, attiny12, attiny15, attiny28
avr2	at90s2313, at90s2323, at90s2333, at90s2343, attiny22, attiny26, at90s4414, at90s4433, at90s4434, at90s8515, at90c8534, at90s8535
avr2/avr25	at86rf401, ata6289, attiny13, attiny13a, attiny2313, attiny2313a, attiny24, attiny24a, attiny25, attiny261, attiny261a, attiny4313, attiny43u, attiny44, attiny44a, attiny45, attiny461, attiny461a, attiny48, attiny84, attiny84a, attiny85, attiny861, attiny861a, attiny87, attiny88
avr3	atmega603, at43usb355
avr3/avr31	atmega103, at43usb320
avr3/avr35	at90usb82, at90usb162, atmega8u2, atmega16u2, atmega32u2, attiny167
avr3	at76c711
avr4	atmega48, atmega48a, atmega48p, atmega8, atmega8515, atmega8535, atmega88, atmega88a, atmega88p, atmega88pa, atmega8hva, at90pwm1, at90pwm2, at90pwm2b, at90pwm3, at90pwm3b, at90pwm81
avr5	at90can32, at90can64, at90pwm216, at90pwm316, at90scr100, at90usb646, at90usb647, at94k, atmega16, atmega161, atmega162, atmega163, atmega164a, atmega164p, atmega165, atmega165a, atmega165p, atmega168, atmega168a, atmega168p, atmega169, atmega169a, atmega169p, atmega169pa, atmega16a, atmega16hva, atmega16hva2, atmega16hbv, atmega16m1, atmega16u4, atmega32, atmega323, atmega324a, atmega324p, atmega324pa, atmega325, atmega325a, atmega325p, atmega3250, atmega3250a, atmega3250p, atmega328, atmega328p, atmega329, atmega329a, atmega329p, atmega329pa, atmega3290, atmega3290a, atmega3290p, atmega32c1, atmega32hbv, atmega32m1, atmega32u4, atmega32u6, atmega406, atmega64, atmega640, atmega644, atmega644a, atmega644p, atmega644pa, atmega645, atmega645a, atmega645p, atmega6450, atmega6450a, atmega6450p, atmega649, atmega649a, atmega6490, atmega6490a, atmega6490p, atmega649p, atmega64c1, atmega64hve, atmega64m1, m3000
avr5/avr51	at90can128, at90usb1286, at90usb1287, atmega128, atmega1280, atmega1281, atmega1284p
avr6	atmega2560, atmega2561
avrxmega2	atxmega16a4, atxmega16d4, atxmega32a4, atxmega32d4
avrxmega4	atxmega64a3, atxmega64d3
avrxmega5	atxmega64a1, atxmega64a1u
avrxmega6	atxmega128a3, atxmega128d3, atxmega192a3, atxmega192d3, atxmega256a3, atxmega256a3b, atxmega256d3
avrxmega7	atxmega128a1, atxmega128a1u
avrtiny10	attiny4, attiny5, attiny9, attiny10, attiny20, attiny40



Pamiętaj, aby zawsze pracować na kopii pliku skryptu linkera. Zmodyfikowanie oryginału doprowadzi do zmian we wszystkich tworzonych przez Ciebie projektach.

Ponieważ pliki te są zwykłymi plikami tekstowymi, możemy edytować je w dowolnym edytorze tekstu, np. w notatniku Windows. Nie powinniśmy używać do tego celu

programu MS Word, gdyż może on wprowadzić wiele własnych poprawek, w efekcie otrzymany skrypt nie będzie działał prawidłowo. Skrypt zapisany jest w standardzie linuksowym zaznaczania końca linii — znakiem końca jest kod 0x0A. Windows natomiast używa znaków 0x0D, 0x0A, w efekcie wszystkie wiersze będą ze sobą „posklejane”, co utrudni edycję. Z pomocą przyjdzie nam program Programmer's Notepad, dostarczany wraz z pakietem WinAVR. Otworzymy w nim plik skryptu w celu znalezienia następującego fragmentu:

```
.noinit :
{
    PROVIDE (__noinit_start = .) ;
*(.noinit*)
    PROVIDE (__noinit_end = .) ;
    _end = . ;
    PROVIDE (__heap_start = .) ;
} > data
```

Definiuje on sekcję `.noinit`, po której rozpoczyna się sterta — jej początek jest dostępny poprzez symbol `__heap_start`, którego wartością inicjalizowana jest zmienna `__malloc_heap_start`, określająca początek sterty programu. Ponieważ chcemy stertę zainicjalizować w innym miejscu, należy tę linię zakomentować:

```
.noinit :
{
    PROVIDE (__noinit_start = .) ;
*(.noinit*)
    PROVIDE (__noinit_end = .) ;
    _end = . ;
    /*PROVIDE (__heap_start = .) ;*/
} > data
```

Teraz pozostaje nam stworzenie nowej sekcji reprezentującej segment w pamięci rozszerzonej. Ponieważ pamięć ta leży w regionie pamięci RAM o nazwie `data`, nasz segment również umieścimy w tym regionie pamięci. Będzie się on zaczynał od adresu pierwszej komórki pamięci poza pamięcią wewnętrzną mikrokontrolera:

```
.XMEM    0x801100:
{
*(.XMEM*)
    PROVIDE (__heap_start = .);
} > data
```

Jednocześnie adres końca tego segmentu będzie początkowym adresem sterty, stąd też w ramach segmentu zdefiniowano globalny symbol `__heap_start`. Teraz pozostaje nam jeszcze poinformować linker, że zamiast domyślnych skryptów ma używać naszego. Uzyskamy to poleceniem `-Tnazwa_skryptu`. Stosowne polecenie możemy dodać do pliku `Makefile`:

```
LDFLAGS += -Tavr51xmem.x
```

Jeśli piszemy aplikację w AVR, możemy też przekazać linkerowi nowy parametr w znany nam już sposób — za pomocą narzędzi graficznych programu AVR Studio — *Project/Configuration Options/Custom Options/[Linker Options]* i dodajemy:

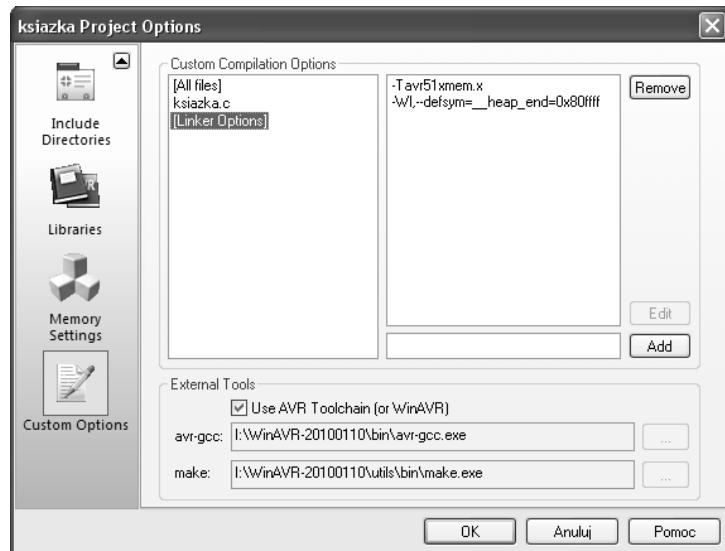
```
-Tavr51xmem.x
```

Na koniec musimy jeszcze określić ostatni adres pamięci dostępny dla sterty. Robimy to już w znany nam sposób, dodając odpowiedni parametr do skryptu *Makefile*, lub za pomocą narzędzi programu AVR Studio.

Wynik powinien wyglądać jak na rysunku 9.9.

**Rysunek 9.9.**

Parametry linkera w AVR Studio dla segmentu *.XMEM* i sterty leżących w pamięci zewnętrznej mikrokontrolera



Początek sekcji *.XMEM* jest ustawiony w skrypcie linkera, nie ma jednak przeskódek, aby go w razie potrzeby zmienić, modyfikując skrypt albo korzystając ze znanego już parametru linkera *-section-start*. Od tej chwili każda zmienna zdefiniowana z nazwą sekcji *.XMEM* trafi do pamięci zewnętrznej:

```
int xxx __attribute__ ((section (".XMEM")));
#define XMEM __attribute__ ((section (".XMEM")))
int xx1 XMEM;
XMEM int xx2;
int XMEM xx3;
```

Tak jak poprzednio, powyżej wprowadzone zostało makro *XMEM*, którego dodanie do zmiennej spowoduje jej umieszczenie w pamięci zewnętrznej.

Na koniec pozostaje nam tylko sprawdzenie wyników komplikacji programu wykorzystującego nowy rozkład segmentów. Skompilujmy poniższy program:

```
#include <avr\io.h>
#include <stdio.h>

void init_XMEM() __attribute__ ((naked)) __attribute__ ((section (".init1")));
void init_XMEM()
{
    MCUCR |= _BV(SRE);
    XMCRB |= _BV(XMBK) | _BV(XMM2) | _BV(XMM1) | _BV(XMM0);
```

```

}

char Read_4kB(size_t adres)
{
    DDRC=0xFF;
    PORTC=0x00;           //Pin odpowiadający linii adresowej AD15 będzie wyjściem w stanie 0
    XMCRB|= _BV(XMM0);   //Zamaskuj linię adresową AD15 - jej stan będzie równy 0
    char byte=*((volatile char*)(adres | 0x8000));      //Odczytaj bajt
    XMCRB&=~ _BV(XMM0); //Przywróć normalną funkcję linii AD15
    return byte;
}

void Write_4kB(size_t adres, char byte)
{
    DDRC=0xFF;
    PORTC=0x00;           //Pin odpowiadający linii adresowej AD15 będzie wyjściem w stanie 0
    XMCRB|= _BV(XMM0);   //Zamaskuj linię adresową AD15 - jej stan będzie równy 0
    *((volatile char*)(adres | 0x8000))=byte;      //Zapisz bajt
    XMCRB&=~ _BV(XMM0); //Przywróć normalną funkcję linii AD15
}

#define XMEM __attribute__ ((section (".XMEM")))

int xx1 XMEM;
XMEM int xx2;
int XMEM xx3;

extern char *_malloc_heap_start;
extern char *_malloc_heap_end;

int main()
{
    volatile char *tmp=_malloc_heap_start;
    char byte=Read_4kB(*tmp);
    tmp=_malloc_heap_end;
    byte=Read_4kB(*tmp);
}

```

Ma on za zadanie zainicjalizowanie pamięci zewnętrznej, dodatkowo umieściliśmy w nim wcześniej omówione funkcje umożliwiające dostęp do przesłoniętej pamięci SRAM oraz zmienne xx1, xx2 i xx3, które powinny znaleźć się na początku pamięci zewnętrznej. Dla testu zdefiniowaliśmy zmienną tymczasową tmp, której zostanie przypisana wartość zmiennej wskazującej na początek sterty. Dzięki temu będziemy wiedzieć, czy sterta została prawidłowo zainicjalizowana. Dodatkowe wywołanie funkcji Read\_4kB z tmp jako argumentem ma na celu tylko oszukanie optymalizatora — bez tego mógłby stwierdzić, że tmp nie jest w programie nigdzie używana, więc jej definicję można wyrzucić. Po skompilowaniu powyższego programu spójrzmy na wygenerowany plik map — powinniśmy odnaleźć w nim następujący fragment:

.XMEM	0x00801100	0x6
*(.XMEM*)		
.XMEM	0x00801100	0x6 ksiazka.o
	0x00801102	xx2
	0x00801100	xx1
	0x00801104	xx3
	0x00801106	PROVIDE (_heap_start, .)

Widzimy, że sekcja `.XMEM`, tak jak chcieliśmy, została utworzona od adresu 0x1100 (pamiętajmy, że 0x00800000 to tylko offset informujący linker, że dany segment należy do pamięci RAM), definiuje także globalny symbol o nazwie `_heap_start`, który powinien wskazywać na początek sterty, umieszczonej tuż po segmencie `.XMEM`. Aby się przekonać, że rzeczywiście tak jest, możemy w symulatorze AVR Studio uruchomić powyższy program i sprawdzić, czy zmienna `tmp` zawiera adres początku sterty (powinien on wynosić 0x1106), a po drugim przypisaniu powinna ona zawierać adres końca sterty (0xFFFF). Widzimy także, że zadeklarowane zmienne `xx1`, `xx2` i `xx3` zostały umieszczone w pamięci zewnętrznej (adresy 0x1100 – 0x1105).

## Konfiguracja V — w pamięci zewnętrznej znajdują się stos

Rozważmy jeszcze jedną sytuację, w której w pamięci zewnętrznej umieszczony został stos. Jak wspomniano, nie jest to dobre rozwiązanie, gdyż dostęp do pamięci zewnętrznej trwa dłużej (do każdej operacji odczytu lub zapisu dodawany jest co najmniej jeden takt opóźnienia), stąd umieszczenie stosu w pamięci zewnętrznej może mieć znaczący wpływ na prędkość działania programu. Należy także upewnić się, czy użyty procesor ma możliwość umieszczenia stosu w pamięci zewnętrznej. Aby zmienić lokalizację stosu, możemy zmienić wartość przypisaną specjalnemu symbolowi `_stack` — jest on wykorzystywany przez skrypty inicjalizacyjne do inicjalizacji wskaźnika stosu. Najwygodniej umieścić stos na końcu pamięci RAM, w ten sposób będzie on rosnąć w dół.



Pamiętajmy, aby na stos przeznaczyć wystarczająco dużo miejsca, aby nigdy nie doszło do jego kolizji z innymi strukturami danych umieszczonymi w pamięci RAM.

Początkowy adres wskaźnika stosu możemy zmienić, dodając w pliku `Makefile` odpowiedni parametr dla linkera:

```
LDFLAGS += -Wl,--defsym=_stack=0x80FFFF
```

bądź też w AVR Studio w znany nam już sposób dodając w okienku *Project Options* w zakładce *[Linker options]* wartość:

```
-Wl,--defsym=_stack=0x80FFFF
```

## Pamięć ROM jako pamięć zewnętrzna

Jako pamięć zewnętrzna możemy także zastosować pamięć tylko do odczytu (ang. *Read Only Memory*). Sposób jej podłączenia jest podobny do sposobu podłączenia pamięci SRAM, możemy w tym przypadku pozostawić niepodłączoną linię strobu zapisu WR. Pamięć taka będzie widoczna w przestrzeni adresowej procesora jako zwykła pamięć, niemożliwy będzie jednak jej zapis. Stąd też nie ma sensu umieszczać w niej sekcji pamięci, w których znajdują się zmienne o dostępie swobodnym — a więc sekcji `.data` i `.bss` oraz sterty i stosu programu. Możemy w niej umieścić struktury danych tylko do odczytu. W tym celu możemy zdefiniować własny segment, w którym umieścimy tego typu dane. Za pomocą opcji wywołania linkera ani za pomocą narzędzi dostępnych

w AVR Studio nie ma jednak możliwości zdefiniowania segmentu pamięci tylko do odczytu. Nie stanowi to jednak przeszkodey, dopóki nie będziemy do takiej sekcji próbowały dokonać zapisu. Próba zapisu nie spowoduje w tym przypadku wygenerowania jakiegokolwiek ostrzeżenia, procesor takie polecenia wykona poprawnie, z tym że sam zapis się nie powiedzie, a więc dane nie zostaną zmodyfikowane. Stąd też zmienne, które znajdują się w zewnętrznej pamięci ROM, należy deklarować z modyfikatorem const, np.:

```
const int x=10;
```

Dzięki temu kompilator będzie mógł wykryć próby modyfikacji takiej stałej i wygenerować informację o błędzie.



Należy także pamiętać, że zmienne umieszczone w zewnętrznej pamięci ROM będą miały nadaną im w programie wartość wyłącznie w sytuacji, w której pamięć ta zostanie odpowiednio zaprogramowana. Standardowe oprogramowanie służące do programowania procesorów AVR nie oferuje takiej możliwości.



## Rozdział 10.

# Dostęp do 16-bitowych rejestrów IO

Ponieważ mikrokontrolery AVR8 są 8-bitowe, dostęp do danych przekraczających długość jednego bajta odbywa się wieloetapowo. W przestrzeni adresowej mikrokontrolerów z tej rodziny znajduje się wiele rejestrów o długości większej niż 8 bitów, np. rejesty liczników *timerów* 16-bitowych są również 16-bitowe, podobnie 16-bitowy jest rejestr przetwornika ADC. Dostęp do takich rejestrów wymaga wykonania pewnej sekwencji rozkazów i przebiega co najmniej dwuetapowo. Wykorzystując kompilator gcc i bibliotekę AVR-libc, do każdego rejestru 16-bitowego możemy się odwoływać na dwa sposoby — albo oddziennie adresując jego 8-bitowe połówki, albo odwołując się do całego 16-bitowego rejestru. Adresowanie poszczególnych bajtów następuje poprzez dodanie do nazwy rejestru sufiksów L dla dostępu do młodszego bajtu lub sufiksów H dla uzyskania dostępu do starszego bajtu rejestru.

## Dostęp do 16-bitowego rejestru ADC

Zaczniemy od prostszego przypadku — rejestru danych ADC. Ponieważ przetwornik ADC w mikrokontrolerach AVR8 jest zazwyczaj tylko 10-bitowy, rejestr przechowujący wynik przetwarzania został podzielony na dwie 8-bitowe połówki, które można odczytywać niezależnie. Każdy z powstałych rejestrów (ADCH i ADCL) ma swój własny adres w przestrzeni adresowej procesora. **Wpisanie nowego wyniku konwersji do rejestru wymaga odczytania jego starszej polowy — rejestru ADCH.** Bez tego wyniki kolejnych konwersji nie są przepisywane do rejestru ADC, w efekcie odczyt młodszej połówki — rejestru ADCL — za każdym razem zwróci nam taką samą wartość.



Nawet jeśli interesuje Cię tylko 8 bitów wyniku, to obowiązkowo musisz odczytywać rejestr ADCH.

W takim wypadku warto wykorzystać bit ADLAR rejestru ADMUX. Jego ustawienie powoduje wyrównanie wyniku do lewej, w efekcie dla uzyskania 8-bitowej dokładności wystarczy odczytywać tylko sam rejestr ADCH. W języku C 10-bitowy wynik konwersji przy założeniu, że bit ADLAR jest wyzerowany, możemy uzyskać w następujący sposób:

```
int a=(ADCH<<8) | ADCL;
```

co spowoduje wygenerowanie kodu zbliżonego do przedstawionego poniżej:

c0:	20 91 79 00	lds	r18, 0x0079
c4:	30 91 78 00	lds	r19, 0x0078
c8:	92 2f	mov	r25, r18
ca:	80 e0	ldi	r24, 0x00 : 0
cc:	23 2f	mov	r18, r19
ce:	30 e0	ldi	r19, 0x00 : 0
d0:	28 2b	or	r18, r24
d2:	39 2b	or	r19, r25

Jak widzimy, prosta operacja została przez kompilator absurdalnie skomplikowana, w dodatku powstały kod działa błędnie. Wygenerowanie tak długiego kodu jest spowodowane tym, że domyślnie w języku C operacje logiczne i arytmetyczne są przeprowadzane na zmiennych o typie `int`. Stąd też nasze 8-bitowe połówki rejestru ADC są najpierw rzutowane na 16-bitowy typ `int`, a następnie jest przeprowadzana pomiędzy nimi operacja sumy bitowej. Widzimy więc, że prosty kod w języku C nie prowadzi do optymalnych rezultatów. W dodatku kompilator nic nie wie o właściwej kolejności odczytu połówek rejestru ADC, stąd też odczytał je w kolejności podanej przez nas — czyli najpierw starszą połówkę, a następnie młodszą. Może to spowodować, że młodsza połówka rejestru ADC będzie zawierała wynik innej (kolejnej) konwersji niż połówka starsza, co doprowadzi do błędnego wyniku. Po odwróceniu kolejności odczytów w wyrażeniu:

```
int a=ADCL | (ADCH<<8);
```

wygenerowany kod będzie prawidłowy (chociaż równie zawiły jak poprzedni).



Zgodnie ze standardem języka C kompilator może dowolnie zmienić kolejność podwrażeń, stąd nie ma gwarancji, że odczyt rejestrów nastąpi w takiej samej kolejności jak w podanym przykładzie.

Wybrnąć z tej sytuacji możemy na dwa sposoby. Możemy nasze wyrażenie rozbić na dwa oddzielne wyrażenia:

```
char l=ADCL;
int a= l | (ADCH<<8);
```

Dzięki temu, że ADCL i ADCH mają atrybut `volatile`, w powyższej sytuacji kolejność odczytów części rejestrów ADC zawsze będzie prawidłowa. Inną możliwością jest odczyt całego 16-bitowego rejestru ADC. Wystarczy więc napisać:

```
int a=ADC;
```

a kompilator wygeneruje optymalny kod, dbając także o właściwą kolejność dostępu do połówek rejestrów ADC:

c0:	20 91 78 00	lds	r18, 0x0078
c4:	30 91 79 00	lds	r19, 0x0079

# Dostęp do 16-bitowych rejestrów timerów

W przypadku timerów 16-bitowych rejesty TCNTn, ICRn i OCRnA/B/C są również 16-bitowe, stąd dostęp do nich odbywa się w specjalny sposób, za pomocą ukrytego dla programisty rejestr tymczasowego TEMP. Zapewnia to jednoczasową modyfikację 16-bitowego rejestru w trakcie zapisu oraz jego prawidłowy odczyt. W trakcie tych operacji dostęp do starszej połówki wyżej wymienionych rejestrów nie prowadzi do ich modyfikacji, zamiast tego dane są zapisywane/odczytywane z ukrytego rejestru TEMP. Rejestr ten jest wspólny dla wszystkich rejestrów 16-bitowych danego timera (ale jest różny dla różnych timerów).



Jedynie odczyt 16-bitowych rejestrów OCRnA/B/C nie odbywa się z użyciem rejestru tymczasowego TEMP.

Rejestr TEMP przechowuje najstarsze 8 bitów 16-bitowego rejestru. Wykonując operację zapisu do 16-bitowego rejestru timera, najpierw należy dokonać zapisu do jego starszej połówki, a następnie do młodszej. Zapis młodszej połówki rejestru 16-bitowego powoduje jednoczasowe zapisanie wartości rejestru TEMP i młodszej części adresowanego rejestru w pełnym 16-bitowym rejestrze timera. Podobnie podczas operacji odczytu najpierw odczytujemy młodszą część interesującego nas rejestru, co powoduje przepisanie do rejestr TEMP jego starszej połówki, a dopiero w następnym etapie odczytujemy starszą połówkę rejestru. Takie pozornie skomplikowane działanie procesora zapewnia jednoczasowe uaktualnienie całego 16-bitowego rejestru.



Zapamiętaj. Odczytując 16-bitowy rejestr, zawsze najpierw odczytuj jego młodszą połówkę, a następnie starszą. Podczas operacji zapisu postępuj dokładnie odwrotnie, a więc najpierw zapisuj starszą połówkę rejestru, a następnie młodszą.

Jeśli powyższe reguły wydadzą Ci się niejasne, to możesz zrzucić całą pracę na kompilator, używając jako argumentu operacji całego 16-bitowego rejestru. Kompilator sam zadba o wygenerowanie właściwego kodu:

```
TCNT1=1024;  
OCR1A=512;  
int a=ICR1;
```

Powyższy kod wpisze do rejestru TCNT1 wartość 1024, następnie do OCR1A 512, a na końcu odczyta wartość 16-bitowego rejestru ICR1. Kompilator powinien go zamienić na następujący kod assemblerowy:

```
TCNT1=1024;  
c0: 80 e0 ldi r24, 0x00 : 0  
c2: 94 e0 ldi r25, 0x04 : 4  
c4: 90 93 85 00 sts 0x0085, r25  
c8: 80 93 84 00 sts 0x0084, r24  
OCR1A=512;
```

```

cc:    80 e0      ldi    r24, 0x00      : 0
ce:    92 e0      ldi    r25, 0x02      : 2
d0:    90 93 89 00 sts   0x0089, r25
d4:    80 93 88 00 sts   0x0088, r24
int a=ICR1;
d8:    20 91 86 00 lds   r18, 0x0086
dc:    30 91 87 00 lds   r19, 0x0087

```

Widzimy, że zapis i odczyt 16-bitowych rejestrów został dokonany we właściwej kolejności. Oczywiście, nic nie stoi na przeszkodzie, aby używać tylko połówek powyższych rejestrów, własnoręcznie dbając o kolejności zapisu i odczytu.



Wskaźówka

Ponieważ operacje dostępu do 16-bitowych rejestrów są operacjami co najmniej dwuetapowymi, pamiętajmy, że jeśli do jakiegokolwiek rejestrów timera uzyskujemy dostęp zarówno z programu głównego, jak i z procedury obsługi przerwania, to musimy zadbać o atomowość przeprowadzonej operacji.

Szerzej problem ten zostanie poruszony w rozdziale 13., poświęconym przerwaniom.

Jeśli w trakcie zapisu starsza połówka 16-bitowego rejestru nie ulega zmianie, to możemy zaoszczędzić nieco miejsca i skrócić czas wykonywania programu, wykonując operację zapisu tylko do młodszej połówki rejestru. W poniższym przykładzie:

```

OCR1A=0x01AA;
OCR1B=0x01FF;

```

zapisujemy dwa 16-bitowe rejesty tego samego *timera*; w obu przypadkach starsza połówka rejestru jest taka sama (jej wartość wynosi 0x01). Spowoduje to wygenerowanie następującego kodu:

```

OCR1A=0x01AA;
c8:    8a ea      ldi    r24, 0xAA      : 170
ca:    91 e0      ldi    r25, 0x01      : 1
cc:    90 93 89 00 sts   0x0089, r25
d0:    80 93 88 00 sts   0x0088, r24
OCR1B=0x01FF;
d4:    8f ef      ldi    r24, 0xFF      : 255
d6:    91 e0      ldi    r25, 0x01      : 1
d8:    90 93 8b 00 sts   0x008B, r25
dc:    80 93 8a 00 sts   0x008A, r24

```

Widzimy, że niepotrzebnie dwukrotnie wpisujemy taką samą wartość do rejestru TEMP. Możemy więc nieco zmodyfikować kod:

```

OCR1A=0x01AA;
OCR1BL=0xFF;

```

W tym wypadku wpisujemy nową wartość tylko do młodszej połówki rejestru OCR1B, odpowiednia wartość starszej połówki już znajduje się w rejestrze TEMP. Wygenerowany kod będzie krótszy o 2 instrukcje:

```

OCR1A=0x01AA;
c8:    8a ea      ldi    r24, 0xAA      : 170
ca:    91 e0      ldi    r25, 0x01      : 1

```

```
cc:    90 93 89 00    sts    0x0089, r25
d0:    80 93 88 00    sts    0x0088, r24
OCR1BL=0xFF;
d4:    8f ef          ldi    r24, 0xFF      : 255
d6:    80 93 8a 00    sts    0x008A, r24
```

Oczywiście, musimy sobie zadać pytanie, czy zysk 2 instrukcji asemblera uzasadnia pewne „zaciemnienie” kodu. Zauważmy, że jeśli w powyższym przykładzie zmienimy wartość wpisywaną do rejestru 0C1A tak, że zmieni się jej starszy bajt, to automatycznie taka sama zmiana nastąpi w 0C1B. Może to prowadzić do trudnych do wychwycenia błędów.



# Rozdział 11.

# Opóźnienia

Pisząc program, stosunkowo często stojmy przed koniecznością wprowadzania opóźnień — np. musimy odczekać na gotowość urządzenia IO lub też wygenerować sekwencję o określonym czasie trwania. Opóźnienia możemy realizować, wprowadzając pętle, np.:

```
for(int c=0;c<100;c++);
```

Problem polega jednak na tym, że w języku C nie mamy kontroli nad wygenerowanym przez kompilator kodem asemblerowym, nie możemy więc ustalić, ile dokładnie taka pętla będzie się wykonywać. Co więcej, w zależności od opcji komplikacji (np. opcji optymalizacji -O0, -Os) wygenerowany kod będzie zupełnie różny. Co gorsze, przy włączonej optymalizacji zauważymy, że żadne opóźnienie nie jest generowane — kompilator stwierdza, że powyższa pętla nic nie robi, i ją usuwa. Możemy się przed tym zabezpieczyć, umieszczając w niej instrukcję, której kompilator nie może usunąć:

```
for(int c=0;c<100;c++) asm volatile ("nop");
```

Dzięki użyciu modyfikatora `volatile` kompilator nie może usunąć instrukcji `nop`, w efekcie realizowana jest cała pętla. Rozwiązaliśmy jeden problem, lecz ciągle pisane w ten sposób funkcje opóźniające nie będą generowały ściśle określonych opóźnień. W tym miejscu przychodzi nam z pomocą biblioteka AVR-libc, a dokładniej umieszczone w pliku nagłówkowym `<util\delay.h>` prototypy funkcji opóźniających.

W pliku tym znajdują się deklaracje dwóch pokrewnych funkcji: `_delay_ms` oraz `_delay_us`. Obie funkcje przyjmują jako parametr wartość opóźnienia — pierwsza podanego w milisekundach, druga w mikrosekundach.



Aby powyższe funkcje działały poprawnie, w czasie komplikacji musi być znana częstotliwość taktowania procesora.

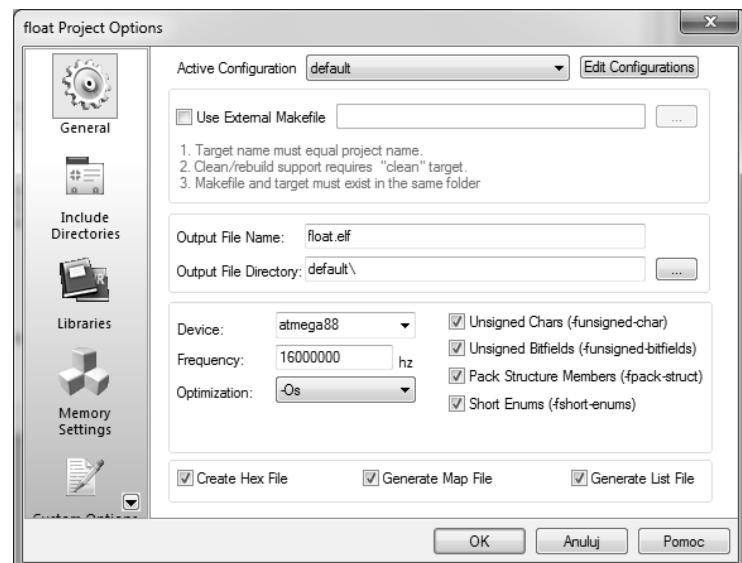
W tym celu funkcje te wymagają wcześniejszego zdefiniowania symbolu `F_CPU`, który musi zawierać częstotliwość taktowania procesora wyrażoną w hercach. Symbol ten możemy zdefiniować przed dyrektywą włączającą plik nagłówkowy:

```
#define F_CPU 16000000UL  
#include <util\delay.h>
```

Nie jest to jednak sposób zalecany — jeżeli plik nagłówkowy `<util\delay.h>` włączamy w kilku miejscach, istnieje spora szansa, że gdzieś zapomnimy zdefiniować `F_CPU` lub podamy jego nieprawidłową wartość. Poprawiane takiego kodu nie należy też do przyjemności. Dlatego lepszym rozwiązaniem jest zdefiniowanie dostępnego globalnie symbolu na poziomie pliku *Makefile*. Możemy to zrobić w AVR Studio, wybierając z menu *Project\Configuration Options\Project Options\General*. W polu *Frequency* wpisujemy częstotliwość taktowania procesora (rysunek 11.1).

**Rysunek 11.1.**

Ustalenie wartości symbolu `F_CPU` w AVR Studio



W powyższym przykładzie wpisano częstotliwość taktowania równą 16 MHz. Jeśli nie korzystamy z AVR Studio, definicję `F_CPU` należy umieścić w pliku *Makefile* za pomocą następującej linii:

```
CFLAGS += -DF_CPU=16000000UL
```

Korzystając z funkcji, których prototypy znajdują się w `<util\delay.h>`, powinniśmy pamiętać, aby przekazywać im wartość opóźnienia jako stałą, nigdy zmienną. Jest to spowodowane sposobem, w jaki obie funkcje są zadeklarowane:

```
void _delay_ms(double __ms);
void _delay_us(double __us);
```

Jak widać, obie funkcje jako argument przyjmują zmienną typu `double` — liczba iteracji pętli wymagana do uzyskania pożądanego opóźnienia liczona jest więc za pomocą typu zmiennopozycyjnego. **Jeśli więc jako argument wywołania podamy zmienną, użycie obu funkcji będzie wymagało dodania całej biblioteki matematycznej.** Spowoduje to wzrost objętości programu o kilka kB! Jeżeli podamy stałą, wszystkie obliczenia zostaną wykonane na etapie komplikacji programu.

Jeśli z jakichś powodów musimy uzyskać zmienne opóźnienie, to należy funkcje `delay_xx` „opakować” we własną pętlę, np. tak:

```
void delay_ms(int ile)
{
    do
    {
        delay_ms(1);
        ile--;
    } while(ile);
}
```

Powyższa funkcja realizuje zmienne opóźnienie — jego czas trwania w milisekundach podawany jest jako argument wywołania, rozdzielcość tej funkcji wynosi 1 ms.



Pamiętaj, że opóźnienia realizowane przez te funkcje nie uwzględniają czasu, jaki poświęca procesor na obsługę przerwań.

Jeśli w trakcie realizowania opóźnienia przerwania nie są zablokowane, to łączny czas trwania opóźnienia przedłuża się o czas trwania obsługi przerwań. **Jeśli chcemy realizować precyzyjne opóźnienia, należy posłużyć się timerami.**



Ostatnią rzeczą, o której musimy pamiętać, korzystając z powyższych funkcji, jest włączenie optymalizacji. Funkcje te nie działają poprawnie, jeśli optymalizacja jest wyłączona (opcja -O0).

Maksymalne opóźnienie możliwe do uzyskania za pomocą funkcji `_delay_us` wynosi **768 µs/F\_CPU[MHz]**. Po przekroczeniu tej wartości w celu uzyskania dłuższego opóźnienia automatycznie wywoływana jest funkcja `_delay_ms`, co jednak wiąże się ze zmniejszeniem rozdzielcości. Podobnie w przypadku wywołania funkcji `_delay_ms`, jeśli opóźnienie przekracza wartość **262,14 ms/F\_CPU[MHz]**, rozdzielcość funkcji zmniejsza się do ok. 0,1 ms. Maksymalne opóźnienie możliwe do uzyskania wynosi **6,5535 s**, niezależnie od częstotliwości taktowania procesora. W obu przypadkach programista nie jest informowany o zmniejszeniu rozdzielcości pomiaru czasu.



Pamiętajmy, aby z funkcji opóźniających korzystać rozsądnie. Umieszczanie zbyt długich opóźnień w programie nie jest zazwyczaj pożąданie. Nie należy też umieszczać tych funkcji w procedurach obsługi przerwań.

Proste opóźnienia możemy realizować za pomocą funkcji, których prototypy znajdują się w pliku nagłówkowym `<util\delay_basic.h>`. Są to dwie funkcje: `_delay_loop_1` i `_delay_loop_2`, które implementowane są jako proste pętle wykonujące zadaną liczbę iteracji. `_delay_loop_1` jako argument przyjmuje wartość typu `uint8_t`, można więc wykonać maksymalnie 256 iteracji pętli, `_delay_loop_2` jako argument przyjmuje zmienną typu `uint16_t`, można więc wykonać maksymalnie 65536 iteracji. Argument wywołania obu funkcji może być zmienną. Czas iteracji pierwszej funkcji to 3 cykle, ostatnia pętla wykonywana jest 2 cykle, zatem łączny czas wykonania funkcji wynosi **(n-1)\*3+2 cykle**, gdzie **n** to liczba iteracji. W przypadku `_delay_loop_2` opóźnienie wynosi **(n-1)\*4+3 cykle**. Czas opóźnienia całkowicie zależy od częstotliwości taktowania procesora.



## Rozdział 12.

# Dostęp do portów IO procesora

Programując mikrokontrolery, praktycznie w każdym projekcie korzystamy z portów wejścia/wyjścia (ang. *Input/Output Ports*). Mikrokontrolery AVR zwykle mają od kilku do kilkudziesięciu (3 – 86) niezależnie programowalnych pinów IO pogrupowanych w 8-bitowe porty. Dostęp do nich na poziomie sprzętowym odbywa się za pomocą 3 specjalnych rejestrów: DDRx, PORTx i PINx. Oprócz tego w innych rejestrach kontrolnych mikrokontrolera znajdują się pewne bity o specjalnym znaczeniu (np. bit PUD) — blokują lub odblokowują one rezystory podciągające piny portu do zasilania. **Wykorzystując porty IO procesora, musimy pamiętać, że piny obudowy, przez które są one wyprowadzone, są współdzielone przez inne urządzenia peryferyjne procesora. Odblokowanie ich (np. interfejsu UART, I2C, interfejsu pamięci zewnętrznej itd.) powoduje blokowanie ich funkcji jako portów IO procesora.**



Wskazówka

Musimy także pamiętać, że w mikrokontrolerach wyposażonych w interfejs JTAG jest on standardowo włączony, uniemożliwiając wykorzystanie pinów IO współdzielonych z tym interfejsem.

Wyłączenie interfejsu JTAG umożliwia wykorzystanie zajętych przez niego pinów IO procesora.



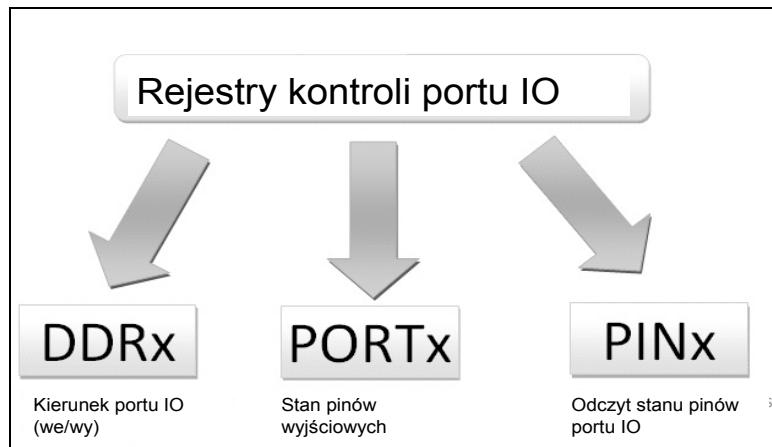
Wskazówka

Wszystkie definicje umożliwiające dostęp do portów IO procesora znajdują się w pliku nagłówkowym `<avr\io.h>`. Nie zapomnij go dołączyć we własnym programie.

## Konfiguracja pinu IO

Piny IO pogrupowane są w porty, liczące od 3 do 8 pinów. Każdy port IO składa się z 3 rejestrów — PORTx, DDRx i PINx. Ich znaczenie pokazane jest na rysunku 12.1. Port DDRx określa kierunek poszczególnych pinów IO (wejście, wyjście), PORTx zawiera

**Rysunek 12.1.**  
Struktura portu IO  
procesora. X oznacza  
wybrany port (A, B,  
C, D, E, F, G, H)



stan pinów ustawionych jako wyjście lub aktywuje rezystor podciągający przypisany pinowi (ang. *Pull Up*), natomiast rejestr PINx służy do odczytu stanu portu (w niektórych mikrokontrolerach AVR możemy także do tego rejestru zapisywać).



Domyślnie wszystkie piny IO ustawione są jako wejście bez włączonego rezystora podciągającego.

Każdy pin portu może być niezależnie skonfigurowany jako wyjście, wejście lub wejście z podciąganiem zgodnie z tabelą 12.1.

**Tabela 12.1.** Stan pinu portu IO w zależności od stanu rejestrów PORT<sub>x</sub>, DDR<sub>x</sub> i bitu PUD w rejestrze MCUCR

DDR <sub>xn</sub>	PORT <sub>xn</sub>	PUD	Pin wejściowy/ wyjściowy	Podciąganie do Vcc	Komentarz
0	0	X	wejście	nie	wejście trójstanowe
0	1	0	wejście	tak	wejście z podciąganiem do Vcc poprzez wewnętrzny rezystor
0	1	1	wejście	nie	wejście trójstanowe
1	0	X	wyjście	nie	wyjście w stanie niskim
1	1	X	wyjście	nie	wyjście w stanie wysokim

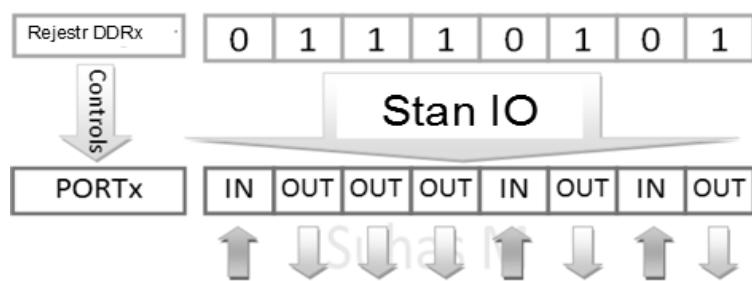
Dany pin portu jest pinem wyjściowym, jeśli odpowiadający mu bit w rejestrze DDR<sub>x</sub> ma wartość 1 — rysunek 12.2.

W takiej sytuacji stan poszczególnych linii IO portu określa rejestr PORT<sub>x</sub> — rysunek 12.3.

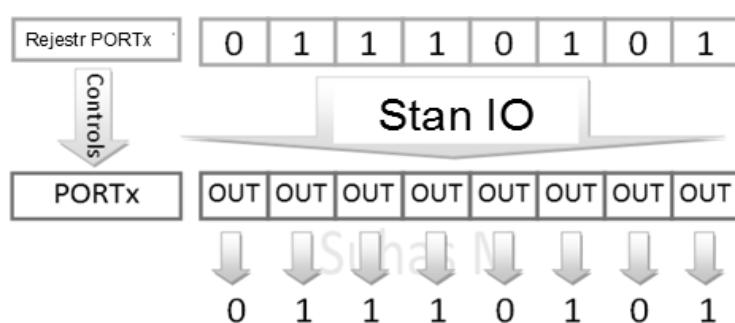
Jeśli dany bit rejestrów DDR<sub>x</sub> ma wartość 0, to pin jest skonfigurowany jako wejście. W tej sytuacji, jeśli odpowiadający mu bit rejestrów PORT<sub>x</sub> ma wartość 1, to dodatkowo włączany jest wbudowany w procesor rezystor podciągający, dzięki czemu na danym pinie panuje stan wysoki — rysunek 12.4.

**Rysunek 12.2.**

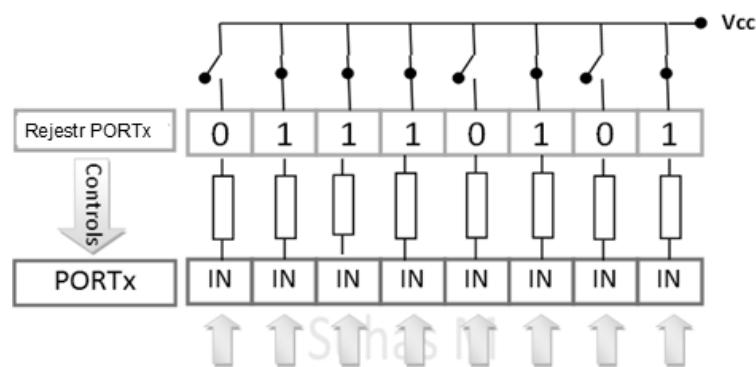
Kontrola kierunku poszczególnych linii portu IO. Jeśli bit rejestru DDRx ma wartość 1, to odpowiadająca mu linia portu IO jest wyjściem, którego stan określany jest przez odpowiedni bit rejestru PORTx

**Rysunek 12.3.**

W przypadku kiedy rejestr DDRx określa port IO jako wyjściowy, stan poszczególnych linii odpowiada wartości rejestru PORTx

**Rysunek 12.4.**

W sytuacji, w której port lub poszczególne piny są ustawione jako wejścia, możemy poprzez rejestr PORTx kontrolować włączenie rezystorów podciągających. Przy ich włączeniu dana linia portu IO jest podciągana przez rezistor 40-60 kΩ do zasilania



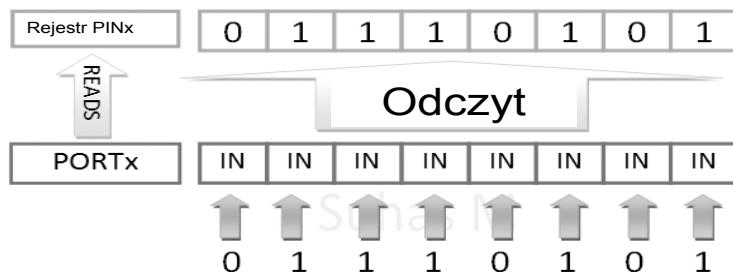
W niektórych mikrokontrolerach AVR dodatkowo występuje w rejestrze MCUCR specjalny bit PUD (ang. *Pull-up Disable*), którego ustawienie powoduje wyłączenie wszystkich rezystorów podciągających niezależnie od stanu PORTx. Trzeci rejestr kontrolny związany z danym portem to rejestr PINx. Odzwierciedla on stan pinów danego portu — rysunek 12.5.



Wskaźówka

Pamiętajmy, że w rejestrze PORTx zawsze znajduje się wartość, którą tam wpisaliśmy, a nie aktualny stan pinów portu IO. Aby odczytać stan pinów portu IO, musimy odczytać rejestr PINx.

**Rysunek 12.5.**  
Odczyt stanu portu  
przy pomocy rejestru  
PINx



Jeżeli dany pin portu ustawiony jest jako wyjście, jego stan w rejestrze PINx odpowiada wartości odpowiedniego bitu rejestru PORTx.



W przypadku kiedy funkcja danego pinu jest przejęta przez inny układ (np. port USART), stan pinu ciągle możemy określić, odczytując stan odpowiedniego bitu rejestru PINx.

Z poziomu asemblera mamy możliwość indywidualnej zmiany poszczególnych bitów wyżej wymienionych rejestrów za pomocą instrukcji CBI/SBI. W starych wersjach biblioteki AVR-libc były zdefiniowane makra CBI/SBI o identycznej funkcjonalności, lecz zostały one wycofane. Obecnie taką samą funkcjonalność można uzyskać, stosując standardowe operacje bitowe języka C.

Konfigurując port jako wyjście, wystarczy ustawić odpowiednie bity rejestru DDRx:

```
DDRB=0xFF;
PORTB=0x00;
```

Po wykonaniu powyższych instrukcji cały port B skonfigurowany jest jako wyjście, a stan wszystkich jego bitów jest niski. Możemy skonfigurować tylko część pinów portu jako wyjście, a pozostałe jako wejścia:

```
DDRB=0xF0;
PORTB=0x00;
```

W tym przypadku najstarsze 4 bity portu B (PB4 – PB7) będą wyjściami w stanie niskim, a najmłodsze 4 bity (PB0 – PB3) będą wejściami bez podciągania. Ich stan możemy odczytać za pomocą rejestru PINB:

```
char a=PINB;
```

Rejestr PINB będzie zawierał stan wszystkich bitów rejestru PORTB. Ponieważ 4 najmłodsze są wejściami, więc ich stan będzie zależał od stanu linii podłączonych do tych pinów portu. Natomiast stan 4 najstarszych będzie zawsze wynosił 0 — bo taką wartość wpisaliśmy do rejestru PORTB. Po zmianie powyższych przykładów na:

```
DDRB=0xF0;
PORTB=0xFF;
char a=PINB;
```

odczytana wartość zmiennej a będzie wynosiła 0xFF. Powyższy ciąg instrukcji ustawia 4 najstarsze bity portu B jako wyjścia, ich stan jest więc określany przez odpowiednie bity rejestru PORTB. Natomiast 4 najmłodsze bity są wejściami, ale ponieważ odpowia-

dające im bity rejestru PORTB są ustawione, włączone są wewnętrzne rezystory podciągające. Stąd jeśli port zostawimy niepodłączony, to zawsze z PINB odczytamy wartość 0xFF. Zauważmy, że gdyby rezystory podciągające na pinach skonfigurowanych jako wejściowe nie były włączone, to ich stan byłby nieustalony.

## Manipulacje stanem pinów IO

Stosunkowo często zachodzi potrzeba zmiany poszczególnych pinów portu IO. W języku C osiągniemy to za pomocą operacji bitowych. Niektóre z nich (ustawianie lub zerowanie stanu pinu IO) mogą być realizowane atomowo, inne niestety nie. Do portów odwołujemy się jak do zwykłych 8-bitowych zmiennych, w których stan linii IO jest odzwierciedlony stanem poszczególnych bitów 8-bitowej zmiennej.

### Zmiana stanu portu na przeciwny

Jedną z prostych, ale często potrzebnych operacji jest zmiana stanu linii IO na przeciwny. Przeanalizujmy funkcję:

```
void CLK()
{
    DDRB|=0x01;
    while(1)
    {
        PORTB=PORTB ^ 0x01;
    }
}
```

Pin PB0 portu B zostanie skonfigurowany jako wyjście, funkcja pozostałych pinów portu B nie ulegnie zmianie. Następnie stan pinu PB0 będzie się naprzemiennie zmieniał, dzięki wykorzystaniu operacji sumy wyłączającej. Stan pozostałych pinów portu B nie ulegnie zmianie.

#### Wskazówka

Operacja zmiany stanu portu na przeciwny nie jest operacją atomową. O związkach z tym problemach przeczytasz w rozdziale poświęconym przerwaniom.

Nowsze procesory AVR, m.in. ATMega88, umożliwiają zmianę dowolnego bitu portu na przeciwny po zapisaniu 1 na odpowiadającym mu bicie rejestru PINx. W starszych procesorach AVR zapis do rejestru PINx nie pociągał za sobą żadnych skutków.

Stąd powyższą funkcję możemy zapisać następująco:

```
void CLK_atomово()
{
    DDRB|=0x01;
    while(1)
    {
```

```
PINB|=0x01;
}
}
```

Zaletą tego, poza uproszczeniem, jest zmiana bitu portu w sposób atomowy. Wygenerowany kod asemblerowy wygląda następująco:

```
void CLK_atomowo()
{
    DDRB|=0x01;
    120:   20 9a      sbi     0x04, 0 : 4
    while(1)
    {
        PINB|=0x01;
        122:   18 9a      sbi     0x03, 0 : 3
        124:   fe cf      rjmp   .-4      ; 0x122 <CLK_atomowo+0x2>

        00000126 <main>:
    }
}
```

Widzimy, że kompilator wygenerował optymalny kod.



**Wskazówka** Pamiętajmy jednak, że jakiekolwiek odwołania do rejestrów PORT $x$  i DDR $x$  zmieniające jednocześnie więcej niż jeden bit zostaną przetłumaczone na ciąg co najmniej 4 instrukcji asemblera. Stąd w przypadku odwoływanego się do wyżej wymienionych rejestrów z programu głównego i procedury obsługi przerwania istnieje konieczność zapewnienia atomowości dostępu do rejestrów. W przeciwnym przypadku działanie programu może być nieprzewidywalne.

## Ustawianie linii IO

Stan linii IO możemy zmienić na 1, wykonując operację sumy bitowej. Np.:

```
PORTB|=0x01;
```

spowoduje ustawienie najmłodszego bitu (PB0) portu B, bez zmiany stanu innych bitów tego rejestru. Jeśli wybrany port mieści się w przestrzeni IO procesora, to w efekcie zostanie wygenerowana jedna instrukcja:

```
PORTB|=0x01;
1d0:   28 9a      sbi     0x05, 0 : 5
```

W przeciwnym wypadku kompilator będzie zmuszony wygenerować dłuższy kod.

## Zerowanie linii IO

Operację tę możemy wykonać za pomocą iloczynu bitowego. Np.:

```
PORTB&=0xFE;
```

spowoduje wyzerowanie najmłodszego bitu (PB0) rejestru PORTB. Analogicznie jak poprzednio, jeśli wybrany rejestr leży w przestrzeni IO procesora, zostanie wygenerowana jedna instrukcja:

```
PORTB&=0xFE:  
1d0:    28 98      cbi   0x05, 0    ; 5
```

## Makrodefinicja \_BV()

Zmieniając stan danego pinu portu IO, musimy stworzyć odpowiadającą mu maskę bitową. Zapisując maski w postaci heksadecymalnej lub binarnej, bardzo łatwo o pomyłkę, a powstały błąd jest bardzo trudny do wykrycia. Stąd też twórcy biblioteki AVR-libc stworzyli makrodefinicję `_BV()`, której argumentem jest numer bitu, a wynikiem odpowiadająca mu maska bitowa. Np.

```
PORTB=_BV(7);
```

powoduje wpisanie do rejestru PORTB wartości 128 (7. bit rejestru jest ustawiony). Chcąc ustawić np. piny PB1 i PB4 PORTB, zamiast mozolnie wyliczać maski bitowe, możemy napisać:

```
PORTB=_BV(PB4) | _BV(PB1);
```

Klasycznie powyższy kod wyglądałby tak:

```
PORTB=(1<<PB4) | (1<<PB1);
```

lub:

```
PORTB=18;
```

Stosowanie makra, oprócz dyskusyjnej poprawy czytelności kodu, ma niezaprzeczalną przewagę nad jawnym wykorzystywaniem operacji przesunięć bitowych w celu wygenerowania maski — eliminujemy ryzyko popełnienia błędu polegającego na pomyłkowym zastosowaniu operatora relacji (< lub >) zamiast operatora przesunięcia bitowego (<< lub >>). Tego typu błąd jest skrajnie trudny do wykrycia. Przeciwnicy makrodefinicji `_BV()` argumentują z kolei, że jej stosowanie utrudnia przenoszenie programu na inne platformy, gdzie taka makrodefinicja nie występuje. Nie wydaje się być to mocnym argumentem przeciwko stosowaniu powyższej makrodefinicji — jej zdefiniowanie na innej platformie to... jedna linia kodu.

## Użycie pól bitowych

Stosowanie masek w celu wykonywania operacji na poszczególnych liniach IO bywa niewygodne, zmniejsza czytelność kodu, a także zwiększa możliwość popełnienia błędu. Nawet makrodefinicja `_BV()` niewiele poprawia czytelność kodu. Język C oferuje mechanizmy pozwalające na wygodne odwoływanie się do poszczególnych bitów. W tym przypadku wyliczenia odpowiednich masek i dobór właściwych operacji bitowych możemy pozostawić kompilatorowi. Niektóre biblioteki oferują wprost taką możliwość, niestety AVR-libc do nich nie należy. Stąd też musimy sami zdefiniować struktury umożliwiające dostęp do poszczególnych bitów:

```
typedef struct
{
    unsigned int b0:1;
    unsigned int b1:1;
    unsigned int b2:1;
    unsigned int b3:1;
    unsigned int b4:1;
    unsigned int b5:1;
    unsigned int b6:1;
    unsigned int b7:1;
} IO;
```

Powyżej zdefiniowano strukturę o nazwie IO, która umożliwia dostęp do poszczególnych bitów poprzez zdefiniowane pola b0-b7. Pozostaje nam zdefiniować zmienną, która umożliwi dostęp do portu:

```
#define _PORTB  (*( volatile IO*)&PORTB)
```

Dzięki powyższej definicji poprzez symbol \_PORTB będziemy mogli odwoływać się do prawdziwego portu B procesora. Zwróćmy uwagę na obecność modyfikatora volatile — jest on niezbędny, gdyż stan portów IO może zmieniać się w sposób niemożliwy do przewidzenia przez kompilator.



Więcej o modyfikatorze volatile dowiesz się w rozdziale 13., poświęconym przerwaniom.

Od tej chwili możemy się w wygodny sposób odwoływać do poszczególnych linii IO portu B procesora:

```
_PORTB.b1=1;  
_PORTB.b4=1;
```

Wygenerowany kod nie różni się od kodu otrzymanego w wyniku komplikacji instrukcji realizujących dostęp za pomocą operacji bitowych:

```
_PORTB.b1=1:  
  1d0: 29 9a      sbi    0x05, 1      ; 5  
_PORTB.b4=1:  
  1d2: 2c 9a      sbi    0x05, 4      ; 5
```

## Synchronizator

Mikrokontroler jest urządzeniem sekwencyjnym, w którym wszystkie zmiany stanów synchronizowane są zegarem taktującym. Niemniej zmiany stanu pinów wejściowych mogą przebiegać asynchronicznie w stosunku do zegara. Dla uniknięcia problemów z określeniem stanu pinu i metastabilności wszystkie piny IO posiadają tzw. **synchronizator**. Składa się on ze sterowanego zegarem procesora zatrzasku. Umożliwia to co prawda uniknięcie stanów metastabilnych, ale ceną, jaką za to płacimy, jest opóźnienie pomiędzy rzeczywistym stanem pinu IO a jego odczytaną z rejestru PINx wartością.

Opróżnienie to wynosi od  $\frac{1}{2}$  do  $1\frac{1}{2}$  cykli zegara. Zazwyczaj nie stanowi to problemu, lecz czasami istnienie tego układu prowadzi do bardzo subtelnych błędów. Przeanalizujmy poniższą funkcję:

```
void synchro_demo()
{
    char a;
    DDRB=0x01;
    PORTB=0x01;
    PORTB=0x00;
    a=PINB;
    a=PINB;
}
```

Powyższy kod oczywiście niewiele robi, ale podobna sekwencja zmian portu IO często jest wykorzystywana np. do skanowania klawiatury matrycowej. Jak widzimy, ustawiamy pin PB0 jako wyjście, jego stan najpierw ustalamy na 1, a następnie zmieniamy na 0 i odczytujemy stan portu B. Spodziewalibyśmy się, że najmłodszy bit portu B (PB0) będzie miał wartość 0 — bo taką wpisaliśmy w poprzedniej instrukcji. Lecz z powodu istnienia synchronizatora stan PB0 wyniesie 1! Dzieje się tak z powodu opóźnienia wprowadzanego przez synchronizator. Dopiero kolejny odczyt PINB da nam spodziewaną wartość PB0 wynoszącą 1.



Jeśli zmieniamy stan rejestru DDRx lub PORTx, po czym odczytujemy rejestr PINx, zawsze musimy wprowadzić opóźnienie jednej instrukcji.

Ponieważ język C nie oferuje instrukcji opóźniającej będącej odpowiednikiem instrukcji NOP w asemblerze, musimy się posilić wbudowanym asemblerem<sup>1</sup>. Stąd poprawna wersja powyższej funkcji wygląda następująco:

```
void synchro_demo1()
{
    char a;

    DDRB=0x01;
    PORTB=0x01;
    PORTB=0x00;
    asm volatile("NOP": :);      //Opóźnienie
    a=PINB;
    a=PINB;
}
```

Do realizacji opóźnienia została wykorzystana konstrukcja `asm volatile(„NOP”::)`, której zadaniem jest umieszczenie w wygenerowanym kodzie instrukcji opóźniającej NOP. Modyfikator `volatile` jest w tej sytuacji niezbędny — uniemożliwia on kompilatorowi zoptymalizowanie kodu, poprzez wyrzucenie instrukcji NOP (która z punktu widzenia kompilatora nic nie robi).

<sup>1</sup> Więcej na temat wbudowanego asemblera dowiesz się z rozdziału 28., „Łączenie kodu w C i asemblerze”.

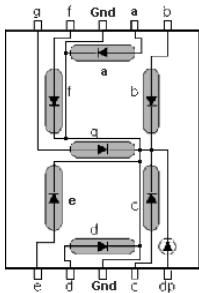
## Przykłady praktyczne

Po tej porcji teorii pokazane zostaną praktyczne przykłady wykorzystania portów I/O procesora. W różnych opisach i instrukcjach zwykle pierwszym przykładem wykorzystania portu I/O i w ogóle mikrokontrolera jest realizacja migającej diody LED. Jednak po tak dużej porcji informacji na temat działania mikrokontrolera spróbujmy na początek zrobić coś bardziej ambitnego — wyświetlimy cyfry przy pomocy wyświetlacza 7-segmentowego LED, a następnie spróbujmy podłączyć proste przyciski, enkoder obrotowy i klawiaturę matrycową, co da nam możliwość sterowania wyświetlana cyfrą. Przedstawione poniżej funkcje obsługi przycisku, enkodera i klawiatury matrycowej wykorzystują w celu wykrycia zmiany stanu technikę częstego próbkowania (ang. *Polling*). W prostych programach technika ta daje niezłe rezultaty, lecz ma swoje ograniczenia. W innych częściach książki, m.in. w rozdziałach poświęconych przerwaniom i interfejsom komunikacyjnym, pokazane zostaną inne, bardziej efektywne techniki. Pamiętaj jednak, że nie ma rozwiązań idealnych — są tylko rozwiązania najlepiej sprawdzające się w ścisłe określonych warunkach. Stąd też poniższe funkcje można z powodzeniem wykorzystywać w swoich programach — ich niewątpliwą zaletą jest prostota.

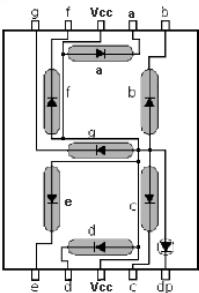
## Sterowanie wyświetlaczem 7-segmentowym

Wyświetlacz 7-segmentowe zbudowane są z diod LED ułożonych tak, aby tworzyć zarysy cyfr 8, co daje nam możliwość, poprzez odpowiednie nimi sterowanie, wyświetlania wszystkich cyfr systemu dziesiętnego, a także kilku dodatkowych symboli. W tego typu wyświetlaczach często występuje dodatkowa dioda LED, realizująca funkcję przecinka dziesiętnego (DP, ang. *Decimal Point*) — rysunek 12.6.

Wspólna katoda



Wspólna anoda



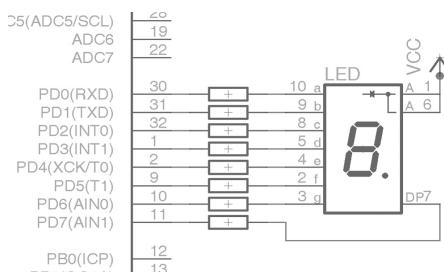
Cyfra	Segmenty (1 – włączony)						
	a	b	c	d	e	f	g
0	1	1	1	1	1	1	0
1	0	1	0	0	0	0	0
2	1	1	0	1	1	0	1
3	1	1	1	1	0	0	1
4	0	1	1	0	0	1	1
5	1	0	1	1	0	1	1
6	1	0	1	1	1	1	1
7	1	1	1	0	0	0	0
8	1	1	1	1	1	1	1
9	1	1	1	1	0	1	1

Rysunek 12.6. Budowa wyświetlacza 7-segmentowego LED

**Wyświetlacz te występują w dwóch podstawowych odmianach — ze wspólną katodą lub ze wspólną anodą.** W wyświetlaczach ze wspólną katodą katody wszystkich segmentów są ze sobą połączone, natomiast sterowanie następuje poprzez odpowiednie wsterowanie anod. W wyświetlaczach ze wspólną anodą jest dokładnie odwrotnie. Ponieważ wydajność prądowa portów I/O procesorów AVR nie zależy od stanu logicznego — jest taka sama dla stanu niskiego i wysokiego — nie ma znaczenia, jaki wyświet-

tlacz podłączymy. W naszym przykładzie wykorzystany zostanie wyświetlacz LED ze wspólną anodą, który podłączymy do portów procesora zgodnie ze schematem przedstawionym na rysunku 12.7.

**Rysunek 12.7.**  
Schemat podłączenia  
7-segmentowego  
wyświetlacza LED  
do portu IO  
procesora AVR



Jak widać, do sterowania wyświetlaczem wykorzystany został cały 8-bitowy port IO mikrokontrolera (w tym przypadku jest to port D). Anoda wyświetlacza została na stałe podłączona do dodatniego napięcia zasilającego. Dzięki temu jeśli na danym pinie portu procesora będzie panował stan niski, podłączony do niego segment wyświetlacza będzie świecił.



Pamiętaj, aby diody LED łączyć zawsze przez rezystor ograniczający prąd.

Wartość tego rezystora można dobrać ze wzoru  $R = (Vcc - Vf)/If$ , gdzie  $Vcc$  to napięcie zasilające układ (typowo 5 V),  $Vf$  to napięcie przewodzenia diody (zależy od typu diody i koloru — jeżeli nie dysponujemy dokładnymi danymi, możemy przyjąć, że wynosi ono ok. 1,5 V), a  $If$  to prąd przewodzenia diody (dla średnich wyświetlaczy typowo wynosi on ok. 20 mA, dla małych może wynosić 1 – 5 mA). Dosyć bezpieczną wartością będzie więc wartość tych rezystorów wynosząca 330 omów. Ważne jest też, aby każdy segment miał własny rezystor ograniczający prąd — nie można zastosować po prostu jednego rezystora podłączonego do wspólnej anody/katody, gdyż w takim przypadku jasność świecenia wyświetlacza zależałaby od liczby świecących segmentów, a takiego efektu z pewnością byśmy nie chcieli.

Spróbujmy więc wykorzystać nowo zdobytą wiedzę do wyświetlania cyfr na takim wyświetlaczu. Przede wszystkim potrzebować będziemy tablicy, której poszczególne elementy reprezentować będą kolejne cyfry dziesiętne:

```
//Cyfry 0,1,2,3,4,5,6,7,8,9 i symbol -
static uint8_t PROGMEM DIGITS[11]={0xC0, 0xF9, 0xA4, 0xB0, 0x99, 0x92, 0x82, 0xF8,
→0x80, 0x90, 0xBF};
const uint8_t DP=0x80;
```

Kolejne elementy tablicy DIGITS zawierają definicję poszczególnych cyfr dziesiętnych, dodatkowo zdefiniowaliśmy stałą DP zawierającą definicję kropki dziesiętnej oraz symbole specjalne: – (minus) i całkowicie wyłączony wyświetlacz. Bity o wartościach 0 odpowiadają zapalonym segmentom, a bity o wartościach 1 odpowiadają segmentom zgaszonym (dla wyświetlacza ze wspólną katodą byłoby odwrotnie). Dodatkowo dla zaoszczędzenia pamięci RAM tablicę DIGITS umieszczono przy pomocy atrybutu PROGMEM w pamięci FLASH mikrokontrolera.

Zdefiniujmy jeszcze funkcję odpowiedzialną za wyświetlanie cyfry:

```
void ShowOnLED(uint8_t val)
{
    uint8_t tmp=0xFF;
    if((val & 0x7F)<11) tmp=pgm_read_byte(&DIGITS[val & 0x7F]);
    if((val & DP)==1) tmp&= ~(DP);
    LEDPORT=tmp;
}
```

Funkcja ta jako argument przyjmuje wartość określającą cyfrę (0 – 9); jeśli podamy wartość większą niż 10, to cały wyświetlacz pozostanie wyłączony. Dodatkowo najstarszy bit argumentu określa stan przecinka dziesiętnego. Ponieważ tablica cyfr (DIGITS) znajduje się w pamięci FLASH, dostęp do niej realizujemy przy pomocy funkcji pgm\_>read\_byte, odczytującej 1 bajt spod podanego adresu pamięci. W funkcji tej wykorzystaliśmy makrodefinicję LEDPORT; razem z makrodefinicją LEDDDR umożliwiają one łatwą zmianę portu, do którego podłączony jest wyświetlacz:

```
#define LED    D
#define GLUE(a, b)      a##b
#define LEDPORT1(s)  GLUE(PORT,s)
#define LEDPORT  LEDPORT1(LED)
#define LEDDDR1(s)   GLUE(DDR,s)
#define LEDDDR  LEDDDR1(LED)
```

Jeśli wyświetlacz podłączymy do innego portu IO mikrokontrolera, wystarczy zmienić definicję symbolu LED, tak aby zawierał on literę wykorzystywanego portu IO.

Na koniec pozostaje nam zaprezentować działanie powyższych funkcji:

```
int main()
{
    LEDDDR=0xFF;           //Wszystkie piny portu są wyjściem
    uint8_t x=0;
    while(1)
    {
        ShowOnLED(x);
        x=(x+1)%12;
        _delay_ms(1000);
    }
}
```

Funkcja main najpierw ustawia port wykorzystywany do podłączenia wyświetlacza jako wyjście, po czym cyklicznie w odstępach 1-sekundowych zmieniana zostaje wyświetlana kolejna cyfra.

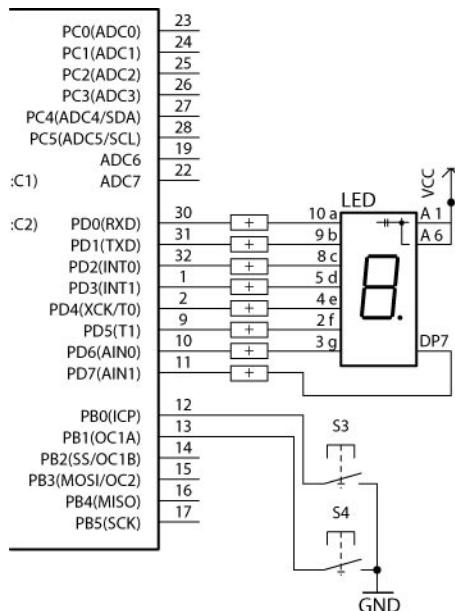
## Podłączenie przycisków

Kolejnym ważnym krokiem w okiełznaniu mikrokontrolera jest dodanie możliwości interakcji z użytkownikiem, poprzez wprowadzenie przycisków dających możliwość wpływu na przebieg wykonywanego programu. Spróbujmy do układu pokazanego na

rysunku 7 dodać dwa przyciski (rysunek 12.8), których naciskanie będzie powodowało zwiększenie lub zmniejszenie o 1 wyświetlanej cyfry.

**Rysunek 12.8.**

Podłączenie  
przycisków  
do mikrokontrolera



Przyciski podłączone zostały do pinów PB0 i PB1 tak, że ich wciśnięcie będzie powodowało, że odpowiedni pin znajdzie się w stanie niskim (będzie zwierany do masy). W sytuacji, w której przycisk nie będzie naciśnięty, stan pinu wymusi wbudowany w procesor rezystor podciągający. Jego wartość wynosi  $20 - 50\text{ k}\Omega$ , co w środowisku o dużych zakłóceniach może się okazać wartością zbyt dużą. W takiej sytuacji możemy zastosować zewnętrzny rezystor podłączony pomiędzy pin IO a Vcc o wartości ok.  $4,7\text{ k}\Omega$ . Jednak w wielu przypadkach w zupełności wystarcza wbudowany rezystor.



Jeśli nie korzystasz z wbudowanego rezystora, to nie zapomnij podłączyć pomiędzy przycisk a zasilanie zewnętrzny rezystor, którego funkcją jest wymuszanie wysokiego stanu logicznego w sytuacji, w której przycisk jest rozwarty.

Do obsługi przycisków nieco zmodyfikujemy funkcję `main`, pozostały kod pozostanie niezmieniony:

```
int main()
{
    LEDDDR=0xFF; //Wszystkie piny portu są wyjściem
    PORTB|= _BV(PB0) | _BV(PB1); //Włącz pull up na pinach PB4 i PB5
    uint8_t x=0;
    while(1)
    {
        ShowOnLED(x);
        if((PINB & _BV(PB4))==0) x--;
        if((PINB & _BV(PB5))==0) x++;
        x=x%10;
```

```

        _delay_ms(100);
    }
}

```

Zauważmy, że w powyższym kodzie zostały włączone rezystory podciągające poprzez wpisanie 1 na odpowiednich bitach portu B, ale nigdzie nie został zainicjowany rejestr DDRB. Stało się tak dlatego, że domyślnie po zresetowaniu układu wszystkie linie IO są ustawione jako wejścia, nie musimy więc tego jawnie deklarować.



#### Wskazówka

W większych programach, szczególnie wykorzystujących *bootloader*, dobrze jest jednak jawnie ustawać stan wszystkich portów i rejestrów *IO*. Dzięki temu można uniknąć wielu trudnych do wyłopienia błędów, kosztem naprawdę tylko nieznacznego wzrostu wielkości programu.

Powyższy kod działa zgodnie z naszymi oczekiwaniami i pozornie wszystko jest w porządku. Teraz spróbujmy go nieco zmodyfikować — wcisnięcie przycisku spowoduje zmianę stanu wyświetlacza (lub diody LED) na przeciwny, np. z 0 na 1 i z 1 na 0 przy każdym wcisnięciu przycisku:

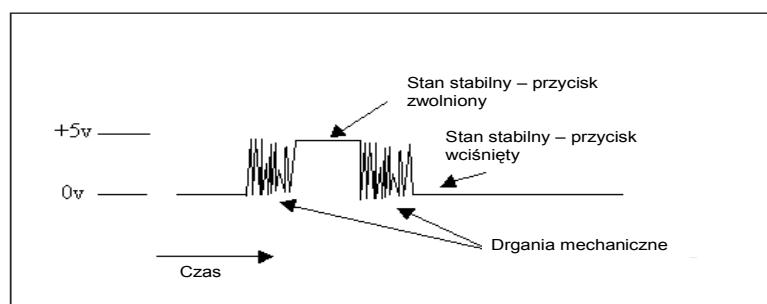
```

int main()
{
    LEDDDR=0xFF;           //Wszystkie piny portu są wyjściem
    PORTB|= _BV(PB4);      //Włącz pull up na pinie PB4
    uint8_t oldkey=PINB & _BV(PB4);
    uint8_t key;
    uint8_t x=0;
    while(1)
    {
        ShowOnLED(x);
        key=PINB & _BV(PB4);          //Odczytaj stan klawisza
        if((key^oldkey) && (key==0)) x=(x+1)%2; //Jeśli się zmienił i jest wcisnięty, to zmień x
        oldkey=key;
    }
}

```

Zgodnie z powyższym programem każde zwarcie do masy portu PB4 powinno zmienić stan zmiennej *x* na przeciwny, w efekcie na wyświetlaczu na przemian powinno się pojawiać 0 i 1. I zasadniczo tak jest, ale nie zawsze. Czasami wcisnięcie przycisku powoduje zmianę stanu zgodnie z oczekiwaniem, a czasami wyświetlacz mrugnie, ale stan się nie zmienia. Wyjaśnieniem tego zagadkowego i nieprzewidywalnego działania programu jest zjawisko drgań styków przycisku (ang. *Bouncing*) — rysunek 12.9.

**Rysunek 12.9.**  
Zjawisko drgań  
styków przełączników  
mechanicznych  
przy zmianie stanu



Jak widzimy, przy każdej zmianie stanu przez pewien czas przyciski drgają, generując naprzemiennie stan niski i wysoki, co jest nieprawidłowo odczytywane przez program jako kolejne wciśnięcia przycisku. Takie zjawisko dotyczy wszystkich styków mechanicznych — niezależnie, czy jest to przycisk, enkoder mechaniczny, czy przekaźnik. W zależności od konstrukcji styków drgania te mają różną długość, typowo od ok. 1 do kilku ms. Możemy częściowo im zapobiec, stosując różne rozwiązania analogowe (filtry RC), przekaźniki RS, lub programowo. Ponieważ interesuje nas programowanie, zastanówmy się, w jaki sposób programowo można rozwiązać problem drgających styków przycisków (ang. *Debouncing*).

Jednym z najprostszych programowych sposobów na eliminację drgań styków jest wprowadzenie pewnego czasu opóźnienia. Jeśli dojdzie do zmiany stanu przycisku wykrytej przez program, następna zmiana będzie zaakceptowana dopiero po upływie pewnego czasu, który musi być dłuższy niż czas trwania eliminowanych drgań:

```
int main()
{
    LEDDDR=0xFF;           //Wszystkie piny portu są wyjściem
    PORTB|= _BV(PB4);      //Włącz pull up na pinie PB4
    uint8_t oldkey=PINB & _BV(PB4);
    uint8_t key;
    uint8_t x=0;
    uint8_t counter=0;
    while(1)
    {
        ShowOnLED(x);
        if(counter==0)
        {
            key=PINB & _BV(PB4);      //Odczytaj stan klawisza
            if((key^oldkey) && (key==0))
            {
                x=(x+1)%2;          //Jeśli się zmienił i jest wciśnięty, to zmień x
                counter=200;          //Czas, przez jaki stan przycisku będzie ignorowany
            }
            oldkey=key;
        } else
        {
            _delay_ms(1);
            counter--;
        }
    }
}
```

W powyższym programie wprowadzono zmienną `counter`, stanowiącą zezwolenie na odczytanie klawisza. Jeśli jej wartość wynosi 0, to można odczytać stan klawisza, w przypadku wykrycia zmiany jego stanu (wciśnięcia lub zwolnienia) zmiennej tej nadawana jest wartość różna od 0, będąca opóźnieniem w ms, przez jakie program będzie ignorował kolejne zmiany stanu przycisku. W powyższym przykładzie opóźnienie to wyniesie ok. 200 ms; jeśli w tym czasie przycisk zmieni stan, to program to zignoruje. Po uruchomieniu powyższego kodu okazuje się, że problem drgających styków zniknął i program działa zgodnie z oczekiwaniami.



**Wskazówka** W dalszych rozdziałach przedstawione zostaną inne, lepsze sposoby eliminacji drgań styków.

W podobny sposób możemy wyeliminować drgania wielu przycisków. Aby pokazać, jak to zrobić, wróćmy do wcześniejszego przykładu. Tym razem użyte zostaną dwa przyciski podłączone do portów PB4 i PB5 mikrokontrolera, których naciśnięcie będzie zwiększać/zmniejszać o 1 wyświetlaną cyfrę. Każdy przycisk będzie posiadał przyporządkowaną mu zmienną counter, najwygodniej więc będzie stworzyć odpowiednią tablicę. Ponieważ mamy dwa przyciski, nasza tablica będzie 2-elementowa:

```
uint8_t counters[2]; //Tablica zawierająca liczniki
```

Tablicy tej nie inicjujemy, gdyż każda zmienna globalna podczas startu programu jest zerowana. Teraz kolej na program główny:

```
int main()
{
    LEDDDR=0xFF; //Wszystkie piny portu są wyjściem
    PORTB|=_BV(PB4) | _BV(PB5); //Włącz pull up na pinie PB4 i PB5
    uint8_t oldkey=PINB & (_BV(PB4) | _BV(PB5));
    uint8_t key;
    uint8_t x=0;
    uint8_t przerwa;

    while(1)
    {
        ShowOnLED(x);
        if(counters[0]==0)
        {
            key=PINB & _BV(PB4); //Odczytaj stan klawisza
            if(((key^oldkey) & _BV(PB4)) && ((key & _BV(PB4))==0))
            {
                if(x>0) x--;
                counters[0]=200; //Jeśli się zmienił i jest wciśnięty, to zmień x
                //Czas, przez jaki stan przycisku będzie ignorowany
            }
            oldkey&=~_BV(PB4);
            oldkey|=key;
        }

        if(counters[1]==0)
        {
            key=PINB & _BV(PB5); //Odczytaj stan klawisza
            if(((key^oldkey) & _BV(PB5)) && ((key & _BV(PB5))==0))
            {
                if(x<9) x++;
                counters[1]=200; //Jeśli się zmienił i jest wciśnięty, to zmień x
                //Czas, przez jaki stan przycisku będzie ignorowany
            }
            oldkey&=~_BV(PB5);
            oldkey|=key;
        }

        przerwa=0;
        for(uint8_t c=0;c<2;c++)
        if(counters[c])
        {

```

```
counters[c]--;  
przerwa=1; //Robimy opóźnienie tylko, jeśli któryś z liczników był !=0  
}  
if(przerwa) _delay_ms(1);  
}  
}
```

Tak jak poprzednio zostały zainicjowane porty, a na pinach, do których podłączone są przyciski, włączone zostały wewnętrzne rezystory podciagające. Stworzyliśmy dwa bloki, sprawdzające, czy stan konkretnego przycisku uległ zmianie, i jeśli tak ustawiające związaną z nim pozycję tablicy counters. Na końcu w petli for tablica ta jest przeglądana, a jeśli któraś z pozycji jest różna od 0, to jej wartość jest dekrementowana, a zmiennej przerwa nadawana jest wartość 1. W efekcie po opuszczeniu pętli zmienna przerwa ma wartość 1 tylko w sytuacji, w której któryś z liczników był różny od 0 — w takim przypadku realizowane jest 1 ms opóźnienie. Zauważmy, że opóźnienie zawsze wynosi 1 ms, niezależnie od liczby pozycji tablicy counters, różnych od 0.

Powyższy kod łatwo jest rozbudować o obsługę kolejnych przycisków, wystarczy zwiększyć liczbę pozycji tablicy counters i dodać kolejne warunki odpowiadające kolejnym przyciskom. Nieco skomplikowane mogą wydawać się warunki typu:

```
if(((key^oldkey) & _BV(PB5)) && ((key & _BV(PB5))==0))
```

Pierwsza część warunku sprawdza, czy stan badanego przycisku uległ zmianie; jeśli tak, zwracana jest prawda. Druga część sprawdza, czy przycisk jest wcisnięty. Zmiana stanu zmiennej x następuje tylko w sytuacji, w której oba warunki są prawdziwe.

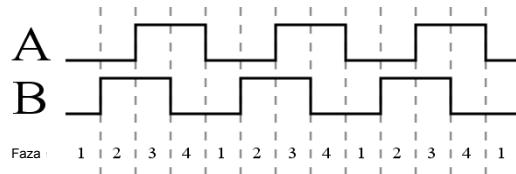


Pamiętaj o istotnej różnicy pomiędzy operacją iloczynu bitowego (&) a operacją iloczynu logicznego (&&).

## Enkoder obrotowy

W wielu sytuacjach zamiast przycisków bardziej intuicyjne sterowanie zapewnia enkoder obrotowy. Jest to urządzenie przekształcające ruch obrotowy na ciąg impulsów, na podstawie których można określić kierunek ruchu (lewo, prawo) oraz jego prędkość. W elektrotechnice stosuje się różne typy enkoderów, w dalszej części przedstawiony zostanie przykład obsługi mechanicznego enkodera obrotowego. Kręcząc gałkę enkodera, możemy np. przesuwać się po różnych pozycjach menu, a wciskając ją, wybierać daną pozycję. Na razie jednak pokazany zostanie przykład bazujący na poprzednich — tym razem wyświetlana cyfra będziemy zmieniać, kręcząc gałkę enkodera. Zazwyczaj tego typu enkodery mają dwa wyjścia A i B, na których w trakcie obracania pojawia się przesunięty w fazie sygnał (rysunek 12.10).

Obracanie gałką enkodera generuje na wyjściu naprzemiennie sygnał niski i wysoki, co umożliwia detekcję obrotu. Im jest on szybszy, tym wyższa jest częstotliwość powstających impulsów. Enkodery mają ściśle określona liczbę impulsów przypadających na jeden pełny obrót. Wartość ta mieści się w szerokich granicach, od kilku do kilkunastu tysięcy impulsów/obrót. Im więcej impulsów przypada na jeden pełny obrót



**Rysunek 12.10.** Sygnał na wyjściach A i B enkodera obrotowego. Sygnał A względem B jest przesunięty w fazie o 90 stopni, co umożliwia rozpoznanie kierunku obrotu

enkodera, z tym większą precyją możemy zmierzyć kąt obrotu. Jednak w naszym przykładzie, gdzie chcemy wykorzystać enkoder tylko do ustawienia pożąanej wartości na wyświetlaczu, nie potrzebujemy bardzo precyjnych i, co za tym idzie, bardzo drogich enkoderów. Prędkość obrotu enkodera możemy wyliczyć z częstotliwości sygnału A lub B, oba sygnały są natomiast potrzebne dla określenia kierunku obrotu.

Zazwyczaj enkodery dysponują wyjściami dającymi sygnał w tzw. kodzie Graya. Jest to odmiana kodu binarnego, w którym jednorazowo zmianie ulega wyłącznie 1 bit. Np. kolejne liczby 0 – 3 w kodzie Graya wyglądają następująco: 00, 01, 11, 10. Dzięki temu jesteśmy w stanie na podstawie sygnałów A i B określić kierunek obrotu (tabela 12.2).

**Tabela 12.2.** Tablica przejść dla enkodera obrotowego w zależności od kierunku obrotu

<b>Obrót w lewo</b>			<b>Obrót w prawo</b>		
Cykl	A	B	Cykl	A	B
1	0	0	1	1	0
2	0	1	2	1	1
3	1	1	3	0	1
4	1	0	4	0	0

Znając stan wyjściowy enkodera i stan po wykonaniu jednego kroku obrotu, możemy na podstawie tabeli przejść określić, w którym kierunku nastąpił obrót. Dla przykładu założymy, że wyjściowo odczytana wartość linii A i B wynosiła 01, kolejna odczytana wartość wyniosła 00. Z tabeli 2 wynika, że jedyną możliwością, aby taką zmianę zaszła, jest wykonanie obrotu w prawo. Przy obrocie w lewo nie jest możliwe przejście ze stanów 01 do 00, gdyż następnym stanem po 01 jest dla obrotu w lewo stan 11. Z kolei dla obrotu w prawo stanem po 01 jest właśnie 00. Na skutek zbyt wolnego próbkowania linii A i B przez program lub zbyt szybkiego obrotu możemy „zgubić” pewne stany enkodera; w efekcie niemożliwe będzie określenie kierunku obrotu. Podobnie przy bardzo szybkim obrocie, jeśli zgubionych zostanie kilka stanów pośrednich, program taki ruch może błędnie zinterpretować jako ruch w przeciwnym kierunku. Np. założmy wyjściowy stan enkodera równy 01. Kręcząc odpowiednio szybko enkoderem w prawo, można spowodować, że program „nie zauważy” przejść 00 i 10, a następnym odczytanym stanem będzie stan 11. Jak widać z tabeli 2, dla obrotu w prawo przejście 01→11 jest niemożliwe, natomiast jest ono prawidłowym przejściem dla obrotu w lewo. **Program taki szybki ruch w prawo błędnie zinterpretuje jako przeskok o jedną pozycję w lewo.**

Kolejny problem, jaki możemy napotkać, związany jest z mechaniczną budową enkodera. Podobnie jak zwykły przycisk składa się on z elementów mechanicznych, które naprzemiennie są zwierane i rozwierane, generując sygnał. Mają więc takie same wady jak przycisk — podczas zmiany stanu generują drgania, prowadzące do powstania wie-lokrotnych przejść.



Wskazówka

Istnieją inne typy enkoderów, m.in. enkodery optyczne, które pozbawione są tej wady. Jednak ich cena zniechęca do stosowania ich w zastosowaniach amatorskich.

Większość enkoderów, podobnie jak zwykłe przyciski, podczas obrotu zwiera i rozwiera wyjścia A i B do specjalnego wejścia oznaczonego często jako COM. Jeśli wejście COM podłączymy do masy, to obrót enkodera spowoduje, że na wyjściach A i B będziemy obserwować naprzemiennie stan niski i stan wysokiej impedancji (rozwarcia). **Stąd też, podobnie jak w przypadku zwykłych przycisków, niezbędne jest stosowanie rezystorów podciągających, wymuszających stan wysoki w chwili rozwarcia wyjścia enkodera.**

Przejdzmy więc do praktycznej realizacji współpracy z enkoderem obrotowym. Poniższa funkcja odczytuje stan pinów podłączonych do wyjść A i B enkodera, zamieniając uzyskaną wartość na informację o kierunku obrotów:

```
int8_t enc_delta;
void ReadEncoder()
{
    static int8_t last;
    int8_t newpos, diff;

    newpos=0;
    if((PINB & _BV(PB4))==0) newpos=3;
    if((PINB & _BV(PB5))==0) newpos^=1; // konwersja kodu Graya na binarny
    diff=last-newpos;
    if(diff & 1)
    {
        last=newpos; // bit 0 = krok
        enc_delta+=(diff & 2)-1; //bit 1 - kierunek
    }
}
```

Powyższa funkcja dokonuje konwersji z kodu Graya na zwykły kod binarny, w efekcie uzyskujemy stan enkodera (0 – 3). Aby możliwe było ustalenie kierunku obrotu, od poprzedniego stanu enkodera odejmowany jest stan aktualny. Poprzedni stan enkodera zapisany jest w zmiennej last — jej stan, dzięki zadeklarowaniu jej jako zmiennej statycznej, nie ulega zmianie pomiędzy kolejnymi wywołaniami funkcji. Zachowuje się ona jak zmienna globalna, tyle że jej zasięg ograniczony jest wyłącznie do funkcji ReadEncoder(). Wartości mniejsze od 0 świadczą o obrocie w lewo, a większe od 0 o obrocie w prawo. Uzyskana informacja zapisywana jest w globalnej zmiennej enc\_delta. Jak widać, współpraca z enkoderem jest banalna w realizacji. Zdefiniujmy jeszcze pomocniczą funkcję:

```
int8_t encode_read1()
{
    ReadEncoder();
```

```
int8_t val=enc_delta;
enc_delta=0;
return val;
}
```

Odczytuje ona aktualny stan enkodera, co powoduje uaktualnienie zawartości zmiennej `enc_delta`, a następnie zwraca jej wartość, jednocześnie ją zerując. Teraz możemy spróbować wykorzystać enkoder do zmiany cyfry wyświetlanej na wyświetlaczu 7-segmentowym (w tym celu wykorzystane zostaną wcześniej omówione funkcje obsługi tego wyświetlacza):

```
int main()
{
    LEDDDR=0xFF;           //Wszystkie piny portu są wyjściem
    PORTB|=BV(PB4) | _BV(PB5); //Włącz pull up na pinie PB4 i PB5
    uint8_t x=0;

    while(1)
    {
        ShowOnLED(x);
        switch(Read1StepEncoder())
        {
            case -1 : if(x>0) x-=1; break;
            case 0 : break;
            case 1 : if(x<9) x+=1; break;
        };
    }
}
```

Powyższa funkcja na podstawie wartości zwracanej przez `Read1StepEncoder()` zmienia wartość zmiennej `x` w zależności od kierunku obrotu. W ten sposób, kręcząc enkoderem, możemy ustawić na wyświetlaczu dowolną cyfrę z zakresu 0 – 9. Program ten działa doskonale, ale jest duże prawdopodobieństwo, że posiadany przez Ciebie enkoder będzie się zachowywał nieco „dziwnie”. Być może pomiędzy osiągnięciem stanów stabilnych pokrętła (można je wyczuć, delikatnie kręcząc pokrętłem, wyczuwając charakterystyczny moment przeskoczenia galki) cyfra wyświetlana na wyświetlaczu zmienia się wielokrotnie. W efekcie delikatny ruch pokrętłem będzie powodował zmianę cyfry — a tak nie powinno być. Takie zachowanie nie jest błędem. Po prostu na rynku istnieje wiele typów enkoderów — są takie, które na jeden „przeskok” generują tylko jeden impuls (jak w przykładzie powyżej), ale są także takie, które na jeden przeskok generują 2, a nawet 4 impulsy (te ostatnie są dosyć często spotykane). Z tego właśnie powodu w powyższym programie wprowadzono funkcję `Read1StepEncoder()`. Poniżej pokazane zostały funkcje umożliwiające poprawną współpracę z enkoderami 2- i 4-stopniowymi:

```
int8_t Read2StepEncoder()
{
    ReadEncoder();
    int8_t val=enc_delta;
    enc_delta=val & 1;
    return val>>1;
}

int8_t Read4StepEncoder()
{
```

```
ReadEncoder();
int8_t val=enc_delta;
enc_delta=val & 3;
return val>>2;
}
```

W funkcji ReadEncoder() nie wprowadzono także procedur eliminujących drgań styków. W powyższym przykładzie drgań styków nie przeszkadzają w prawidłowym działaniu programu. Dzieje się tak dlatego, że przy obracaniu enkoderem jeden z sygnałów (A lub B) zawsze jest sygnałem stabilnym, zmianie ulega tylko jedna z dwóch linii (wynika to z zasady działania kodu Graya). W efekcie drgań występują tylko na jednej linii, co powoduje, że odczytywany stan enkodera oscyluje chwilowo pomiędzy jego poprzednią wartością a nową. W efekcie na wyświetlaczu może przez bardzo krótki czas być wyświetlana poprzednia lub nowa cyfra. Lecz czas ten jest tak krótki, że go z pewnością nie dostrzeżemy. Stąd też w przypadku enkoderów eliminacja drgań styków nie zawsze jest konieczna. Jednak w sytuacji, w której każdy ruch enkodera miałby generować jakieś zdarzenie, eliminacja drgań byłaby konieczna. Możemy jej dokonać w pokazany wcześniej sposób:

```
void ReadEncoder()
{
    static int8_t last;
    static uint8_t laststate;
    static uint8_t counters[2];      //Tablica zawierająca liczniki
    int8_t newpos, diff;

    uint8_t state=PINB;
    if((state^laststate) & _BV(PB4)) && (counters[0]==0))
    {
        counters[0]=200;
        laststate&=(~_BV(PB4));
        laststate|= (state & _BV(PB4));
    }

    if((state^laststate) & _BV(PB5)) && (counters[1]==0))
    {
        counters[1]=200;
        laststate&= (~_BV(PB5));
        laststate|= (state & _BV(PB5));
    }

    uint8_t przerwa=0;
    for(uint8_t c=0;c<2;c++)
        if(counters[c])
    {
        counters[c]--;
        przerwa=1;                      //Robimy opóźnienie tylko, jeśli któryś z liczników był !=0
    }
    if(przerwa) _delay_ms(1);

    newpos=0;
    if((PINB & _BV(PB4))==0) newpos=3;
    if((PINB & _BV(PB5))==0) newpos^=1; //konwersja kodu Graya na binarny
    diff=last-newpos;
    if(diff & 1)
```

```

{
    last=newpos;                                // bit 0 = krok
    enc_delta+=(diff & 2)-1;                  //bit 1 - kierunek
}
}

```

Pozostałe funkcje pozostają niezmienione.



Wskazówka

Bardzo ciekawą opcją są enkodery obrotowe, które oprócz obrotu oferują także funkcję przycisku. Gałkę takich enkoderów można pokręcać, a dodatkowo naciskając ją, uzyskuje się funkcję przycisku. Obsługa tej funkcji jest analogiczna do obsługi zwykłego przycisku.

Powyższy przykład oparty został na metodzie okresowego sprawdzania stanu linii A i B enkodera. Często można się spotkać z przykładami wykorzystującymi metodę detekcji zboczy sygnałów A i B. Nie jest to dobre rozwiązanie, ponieważ:

- ◆ Drgania styków powodują ciągłą reakcję procesora, a program staje się wrażliwy na zakłócenia.
- ◆ Często odczytywane są nieprawidłowe przejścia w kodzie Graya, w efekcie mogą być gubione przejścia enkodera lub odczytywane przejścia, które są efektem drgań, a nie działania użytkownika.
- ◆ Szybkie obracanie enkoderem powoduje ciągłe generowanie przerwań, w efekcie działanie programu staje się mało deterministyczne.

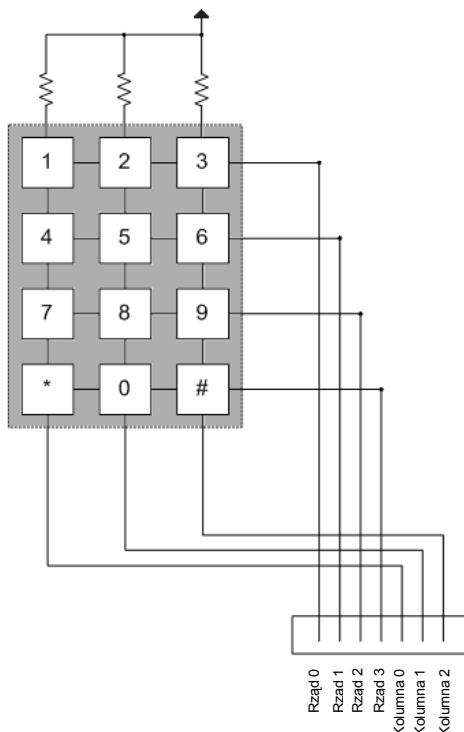
## Klawiatura matrycowa

Przedstawione w poprzednich przykładach sposoby podłączenia przycisków do mikrokontrolera sprawdzają się w sytuacji, w której wykorzystujemy 1 – 4 przyciski. Przy większej ich liczbie zajmujemy kolejne linie portu IO, w efekcie może nam ich po prostu zabraknąć<sup>2</sup>. Drugą niedogodnością jest rosnąca liczba połączeń. Oba problemy rozwiązuje podłączenie matrycowe. Jest ono wykorzystywane m.in. w celu podłączenia różnorodnych klawiatur (rysunek 12.11). Klawiatura zawierająca 12 przycisków jest łączona z mikrokontrolerem zaledwie za pomocą 7 linii IO, w efekcie zyskujemy aż 5 wolnych linii. Liczbę tą możemy dodatkowo zmniejszyć do 5 linii, jeśli zastosujemy dekoder z 2 na 4, a nawet do 3 linii IO w przypadku zastosowania dekodera oraz wolnego kanału ADC. Te dwa ostatnie rozwiązania wymagają jednak zastosowania zewnętrznych elementów — układu scalonego dekodera oraz rezystorów.

**Klawiatura matrycowa podzielona jest na rzędy oraz kolumny.** Na przecięciu rzędów i kolumn umieszczony jest przycisk, który kiedy jest naciśnięty, zwiera daną kolumnę z rzędem. Aby wykryć naciśnięcie określonego przycisku, na dany rzad musimy podać stan niski; jeśli któryś z przycisków przyporządkowanych wybranemu rzędowi będzie naciśnięty, to stan odpowiedniej kolumny zmieni się na niski. Oczywiście,

<sup>2</sup> Jak podłączyć kilka przycisków do jednego pinu IO, zostanie pokazane w rozdziale poświęconym przetwornikowi analogowo-cyfrowemu (ADC).

**Rysunek 12.11.**  
Schemat budowy  
klawiatury  
matrycowej  
o organizacji  $3 \times 4$



zamiana kolumn z rzędami nic nie zmienia, w efekcie możemy stosować dowolne konfiguracje. Przyjmijmy jednak, że w celu odczytania stanu klawiatury wybierać będziemy kolejne rzędy, a odczytywać będziemy stan kolejnych kolumn. Jak widać, do podłączenia klawiatury z rysunku 11 musimy wykorzystać 7 linii IO mikrokontrolera. Zdefiniujmy więc funkcję, która będzie przeszukiwać matrycę i zwracać numer naciśniętego przycisku:

```
uint8_t scankbd()
{
    uint8_t y=0;
    uint8_t x;
    while(y<4) //Sq 4 rzędy
    {
        x=PORTB;
        x=(x | 0x0F) ^ (1<<y); //Wybierz aktywny rząd, nie zmieniając stanu innych pinów portu
        PORTB=x;
        asm volatile ("nop"); //Synchronizator
        x=PINB;
        if((x & 0x70)!=0x70) //Jakiś klawisz był wcisnięty?
        {
            if((x & 0x10)==0) x=0; //Jeśli tak, to z której kolumny?
            else if((x & 0x20)==0) x=1;
            else x=2;
            x=x+y*3; //Oblicz numer klawisza
            return x;
        }
        y++;
    }
}
```

```

    }
    return 255;
}

```

Funkcja scankbd() zwraca numer wciśniętego klawisza (jego numer kolejny w kolumnie i rzędzie) lub 255, jeśli żaden klawisz nie został wciśnięty. Cztery najmłodsze bity portu B wybierają rząd matrycy — wybór następuje przez wystawienie wartości 0 na pinie podłączonym do wybranego rzędu. Piny połączone z pozostałymi rzędami mają w tym czasie wartość 1. Jeśli użytkownik wciśniął jakikolwiek klawisz w wybranym rzędzie, to na odpowiadającej mu kolumnie będzie też panował stan logiczny 0, który zostanie odczytany przez program. Jeśli żaden klawisz w wybranym rzędzie nie był wciśnięty, to dana kolumna będzie w stanie 1, wymuszonym przez wewnętrzny rezystor podciagający portu. Pomiędzy ustawieniem stanu portu, do którego przyłączona jest klawiatura, a odczytaniem wartości kolumn wstawiona jest jedna instrukcja NOP — jest ona niezbędna ze względu na opóźnienie, jakie wprowadza synchronizator.

Numer klawisza uzyskany z funkcji scankbd() możemy wyświetlić na wyświetlaczu 7-segmentowym (numerowi 10 odpowiadają będzie symbol minus (-), a numerowi 12 wygaszony wyświetlacz):

```

int main()
{
    LEDDDR=0xFF;      //Wszystkie piny portu są wyjściem
    DDRB=0x0F;        //4 rzędy są wyjściami, 3 kolumny są wejściami
    PORTB=0x7F;       //Włącz pull upy na kolumnach, ustaw rzędy w stanie 1

    uint8_t kbd;
    while(1)
    {
        kbd=scankbd();
        ShowOnLED(kbd);
    }
}

```

W podobny sposób możemy zrealizować skanowanie matryc o dowolnej liczbie wierszy i kolumn.



**Pamiętajmy,** że styki klawiatur matrycowych w chwili włączenia i wyłączenia również drgają, w efekcie może być konieczne zapewnienie eliminacji drgań, podobnie jak we wcześniejszych przykładach.

Realizacja tego zadania jest jednak prosta — w klawiaturze matrycowej zwykle zakłada się, że naraz zwarty może być wyłącznie jeden przycisk. Upraszczą to znacznie funkcje realizujące eliminację drgań styków.

## Rozdział 13.

# Rejestry IO ogólnego przeznaczenia

Część procesorów AVR posiada tzw. rejesty IO ogólnego przeznaczenia GPIOREG (ang. *General Purpose IO Registers*). Nie pełnią one żadnych funkcji i mogą być swobodnie wykorzystywane przez programistę. Znajdują się w przestrzeni IO procesora dostępnej dla instrukcji CBI/SBI/SBIC/SBIS, dzięki czemu doskonale nadają się do przechowywania informacji, do których wymagany jest atomowy dostęp — np. flag zmienianych w procedurach obsługi przerwań czy też wskaźników do globalnych struktur danych. Nowsze procesory ATMega i ATTiny posiadają zwykle 3 8-bitowe rejesty GPIOREG — GPIOREG0, GPIOREG1 i GPIOREG2. Ze względu na lokalizację w pamięci GPIOREG0 może być wykorzystany do przechowywania wartości 8-bitowej, natomiast GPIOREG1 i GPIOREG2 łącznie do przechowywania wartości 16-bitowej (oba rejesty leżą pod kolejnymi po sobie adresami pamięci):

```
volatile uint8_t * const byteIO=&GPIOREG0;  
volatile uint16_t * const wordIO=(uint16_t*)&GPIOREG1;  
  
*byteIO=100;  
*wordIO=0xaabb;
```

Powyższy kod definiuje dwa wskaźniki — jeden wskazujący na typ `uint8_t` (rejestr GPIOREG0), a drugi wskazujący na typ `uint16_t` (rejesty GPIOREG1 i GPIOREG2). Lokacjom wskazywanym przez zdefiniowane wskaźniki przypisujemy następnie wartość 100 i 0xaabb. W powyższym przykładzie zdefiniowaliśmy wskaźniki z modyfikatorem `volatile`, dzięki czemu kompilator nie próbuje przechowywać danych tymczasowych w rejestrach procesora. Ma to znaczenie, jeśli do powyższych rejestrów odwołujemy się także w przerwaniach. W przeciwnym wypadku możemy modyfikator `volatile` pominąć, co dodatkowo przyśpieszy operacje wykonywane na wskaźnikach `byteIO` i `wordIO`. Dodatkowy modyfikator `const` informuje kompilator, że wartość zdefiniowanych wskaźników nie może się zmieniać — w razie przypadkowej modyfikacji wskaźnika kompilator zgłosi błąd. Dostęp do zdefiniowanych w powyższy sposób zmiennych nie jest atomowy — jeśli `wordIO` może zmieniać się i w procedurze obsługi przerwania, i w innym miejscu programu, to musimy zapewnić dostęp atomowy, stosując makrodefinicję `ATOMIC_BLOCK`.

Przechowywanie zwykłych zmiennych w tych rejestrach wydaje się być pewnym mar-notrawstwem. Prawdopodobnie o wiele lepiej jest je wykorzystać do przechowywania flag bitowych:

```
typedef struct
{
    uint8_t b0:1;
    uint8_t b1:1;
    uint8_t b2:1;
    uint8_t b3:1;
    uint8_t b4:1;
    uint8_t b5:1;
    uint8_t b6:1;
    uint8_t b7:1;
} volatile IO;

IO * const flagi=(IO*)&GPIO0;

flagi->b0=1;
flagi->b7=1;
```

W powyższym przykładzie zdefiniowaliśmy strukturę o nazwie `IO`, zawierającą pola odpowiadające poszczególnym bitom rejestru. Następnie stworzyliśmy wskaźnik o typie `IO`, wskazujący na rejestr `GPIO0`. W kolejnym etapie, wykorzystując wskaźnik `flagi`, ustawiliśmy bity `b0` i `b7`, które fizycznie znajdują się w rejestrze `GPIO0`.



**Dostęp do takich flag odbywa się atomowo — możemy więc bezpiecznie modyfikować je i w procedurach obsługi przerwań, i w innych częściach programu.**

Oczywiście, możemy się odwoływać do tych rejestrów bezpośrednio, stosując operacje przypisania czy maski bitowe, bez pośrednictwa zdefiniowanych powyżej struktur danych czy wskaźników. W tym przypadku jednak tracimy oferowaną przez język C kontrolę typów, musimy także sami wyliczać odpowiednie maski.

## Wykorzystanie innych rejestrów jako GPIO

Nie wszystkie mikrokontrolery AVR posiadają rejestyre GPIO. Jednakże jako rejestyre przechowujące zmienne programu możemy wykorzystać także nieużywane rejestyre IO procesora. Dobrymi kandydatami do tego celu są rejestyre, które można odczytywać/zapisywać, przy czym zmiana stanu rejestrów przy wyłączonym podsystemie IO, w skład którego wchodzi, nie zmienia działania procesora. Dobrymi kandydatami są rejestyre:

- ◆ `TWAR` podsystemu TWI; cały rejestr może służyć jako 8-bitowy rejestr IO.
- ◆ `UBRRL` interfejsu USART — podobnie jak wyżej, cały rejestr przy wyłączonym porcie USART jest do dyspozycji programisty. Zauważ, że `UBRRH` się nie nadaje, gdyż dostęp do niego wymaga wcześniejszej selekcji przy pomocy bitu `URSEL`,

podobnie jak rejestr UDR — w tym przypadku są to dwa oddzielne rejesty, w efekcie nie odczytalibyśmy tego, co wcześniej zostało zapisane.

- ♦ Rejestry PORT i DDR niewykorzystanych portów IO procesora — jeśli piny danego portu nie są do niczego podłączone, to zmiana ich stanu nie stanowi problemu, a stosowne rejesty можемy wykorzystać do przechowywania 8-bitowych danych.
- ♦ Rejestry timerów, np. OCRxA, OCRxB, OCRxC — nawet jeśli timer jest wykorzystywany, to przy zablokowaniu przerwań od danego zdarzenia (*Output Compare Match*) i przy zablokowanych pinach OCR korzystanie z wyżej wymienionych rejestrów jest bezpieczne.

W przypadku innych rejestrów zwykle mamy do dyspozycji poszczególne bity, можemy je wykorzystać np. do przechowywania bitów statusu.



# Rozdział 14.

# Przerwania

Mikroprocesory AVR dysponują bardzo bogatą listą zdarzeń mogących wywoływać przerwania. Przykładową listę obsługiwanych przerwań wraz z krótkim opisem dla mikrokontrolera ATMega88 przedstawiono w tabeli 14.1.

Przerwania umożliwiają przerwanie normalnego ciągu wykonywanych instrukcji i wywołanie specjalnych funkcji — tak zwanych **procedur obsługi przerwań**. Adresy tych procedur (wektory) znajdują się w pamięci FLASH, począwszy od adresu 0 (o ile nie przenieśliśmy wektorów w inne miejsce, o czym przeczytasz w rozdziale poświęconym *bootloaderowi*). W momencie kiedy pojawi się przerwanie, procesor wstrzymuje wykonywanie instrukcji, odkłada na stosie zawartość rejestru SP (wskaźnika stosu) i zaczyna wykonywać procedurę obsługi przerwania. Jednocześnie przyjmowanie dalszych przerwań jest blokowane.



Mamy więc gwarancję, że o ile tego nie zmienimy, procedura obsługi przerwania zawsze wykonywana jest z zablokowanymi przerwaniem.

Zwykle tego domyślnego zachowania nie chcemy zmieniać. W dalszej części rozdziału zostanie pokazane, jak i w jakim celu czasami to domyślne zachowanie się zmienia. Ponieważ język C nie ma żadnych specjalnych mechanizmów odpowiedzialnych za obsługę przerwań, ta część kodu zwykle jest specyficzna dla platformy sprzętowej, na której pracujemy, i konkretnej implementacji języka. Różne kompilatory dostępne dla mikrokontrolerów AVR wykazują w tym miejscu daleko idące różnice. Nawet dla AVR-gcc i AVR-libc implementacja mechanizmu przerwań w ostatnich latach uległa zmianie, co powoduje, że starszy kod jest niekompatybilny z nowymi wersjami kompilatora i biblioteki standardowej dla AVR. Wykorzystanie przerwań rodzi pewne specyficzne problemy, takie jak problem atomowości kodu, funkcji reentrant i współdzielenia zasobów pomiędzy asynchronicznymi wywoływanymi funkcjami. Szerzej ten temat zostanie poruszony w dalszych częściach tego rozdziału.



Wszystkie przydatne definicje związane z obsługą przerwań znajdują się w pliku nagłówkowym `<avr\interrupt.h>`.

**Tabela 14.1.** Lista przerwań obsługiwanych przez mikrokontroler ATMega88

Nazwa wektora	Opis
INT0_vect	Zewnętrzne żądanie przerwania
INT1_vect	Zewnętrzne żądanie przerwania
PCINT0_vect	Przerwanie zmiany stanu pinu
PCINT1_vect	Przerwanie zmiany stanu pinu
PCINT2_vect	Przerwanie zmiany stanu pinu
WDT_vect	Przerwanie <i>Watchdoga</i>
TIMER2_COMPA_vect	Przerwanie <i>Compare Match</i>
TIMER2_COMPB_vect	Przerwanie <i>Compare Match</i>
TIMER2_OVF_vect	Przerwanie nadmiaru licznika <i>timera</i>
TIMER1_CAPT_vect	Przerwanie <i>Capture Event</i>
TIMER1_COMPA_vect	Przerwanie <i>Compare Match</i>
TIMER1_COMPB_vect	Przerwanie <i>Compare Match</i>
TIMER1_OVF_vect	Przerwanie nadmiaru licznika <i>timera</i>
TIMERO_COMPA_vect	Przerwanie <i>Compare Match</i>
TIMERO_COMPB_vect	Przerwanie <i>Compare Match</i>
TIMERO_OVF_vect	Przerwanie nadmiaru licznika <i>timera</i>
SPI_STC_vect	Przerwanie zakończenia transferu SPI
USART_RX_vect	Zakończenie odbioru znaku na interfejsie USART
USART_UDRE_vect	Rejestr danych interfejsu USART jest pusty
USART_TX_vect	Zakończenie wysyłania znaku przez interfejs USART
ADC_vect	Zakończenie przetwarzania ADC
EE_READY_vect	Pamięć EEPROM jest gotowa do następnej operacji
ANALOG_COMP_vect	Przerwanie komparatora
TWI_vect	Przerwanie interfejsu TWI
SPM_READY_vect	Zakończenie operacji zapisu do pamięci FLASH

Dobrze napisana procedura obsługi przerwania powinna spełniać pewne ogólne kryteria:

1. Powinna być jak najkrótsza, tak aby możliwie szybko sterowanie wróciło do przerwanej części programu. Stąd też w procedurach obsługi przerwań nie powinno się stosować opóźnień ani innych czasochłonnych operacji.
2. Nie wolno wywoływać funkcji, które nie są napisane w specjalny sposób, umożliwiający ich wielokrotne wywołanie z różnych wątków programu (nie są *reentrant*).
3. Nie należy korzystać z zasobów, które z definicji nie są współdzielone pomiędzy różnymi wątkami programu — takich jak np. magistrale, porty UART, pamięć EEPROM. Od tej reguły istnieją pewne wyjątki, ale zawsze w takiej sytuacji musimy napisać specjalne procedury umożliwiające bezkonfliktowe współdzielenie tych zasobów.

Aby procedura obsługi danego przerwania została wywołana, muszą być spełnione następujące warunki:

1. Globalna flaga zezwolenia na przerwanie musi być odblokowana.
2. Flaga zezwolenia na przerwanie, które chcemy obsłużyć, musi być odblokowana.
3. Musi istnieć kod procedury obsługi przerwania.
4. Musi wystąpić zdarzenie wywołujące przerwanie.

Procedury obsługi przerwań w AVR domyślnie wykonywane są z zablokowanymi przerwaniami, co zapobiega sytuacji wielokrotnego wchodzenia w to samo przerwanie. Jeśli jednak chcielibyśmy, aby przerwania w czasie ich obsługi były odblokowane, musimy to jawnie zadeklarować. Nie należy do tego wykorzystywać makrodefinicji `sei()`, gdyż jej wykonanie nastąpi stosunkowo późno — najpierw wykonany zostanie cały tzw. **prolog funkcji obsługi przerwania**. Jeśli zależy nam na jak najszybszym ich odblokowaniu, powinniśmy użyć specjalnego atrybutu.

## Obsługa przerwań

Obsługę każdego przerwania powinniśmy rozpocząć od napisania procedury obsługi przerwania (ang. *ISR* — *Interrupt Service Routine*). Jest to specjalna procedura, na którą wskazuje wektor w tabeli wektorów przerwań. W przypadku zaistnienia zdarzenia wywołującego przerwanie wykonywanie normalnego kodu zostaje zawieszone, a procesor zaczyna wykonywać tę procedurę. W języku C funkcja obsługi przerwania jest zwykłą funkcją, musi ona jednak spełniać pewne warunki. Po pierwsze, nie może zwracać żadnego wyniku, a więc musi być zadeklarowana jako `void`. Funkcja ta musi zwracać także specjalny prolog i epilog zapewniające przywrócenie stanu wszystkich zmodyfikowanych rejestrów mikrokontrolera do wartości, jakie miały one w chwili wystąpienia przerwania. Kolejną różnicą jest inny sposób powrotu z takiej procedury. Zwykle wywołanie procedury kończy się wygenerowaniem specjalnej instrukcji asemblera `RET`, powodującej załadowanie do rejestru PC wartości wcześniej odłożonej na stosie, a więc powrót do instrukcji następnej po instrukcji wywołującej daną funkcję. Procedura obsługi przerwania przed zakończeniem musi dodatkowo odblokować globalną flagę zezwolenia na przerwanie, stąd kompilator generuje specjalną instrukcję powrotu `RETI` — instrukcja ta zachowuje się jak instrukcja `RET`, z tym że dodatkowo ustawia flagę I w rejestrze `SREG`, co umożliwia przyjmowanie kolejnych przerwań. Ponieważ standard języka C nie przewiduje żadnych rozwiązań umożliwiających implementację takich specjalnych funkcji, są one realizowane jako rozszerzenie języka, zależne od kompilatora. Kompilator `gcc` wykorzystuje w tym celu specjalny atrybut `SIGNAL`, a sama definicja funkcji wygląda następująco:

```
extern "C" void vector (void) __attribute__ ((signal, __INTR_ATTRS))
```

Aby ułatwić pisanie funkcji obsługi przerwań, twórcy AVR-libc zadeklarowali specjalną makrodefinicję `ISR(vector, ...)`. Przyjmuje ona dwa parametry: `vector`, określający

numer wektora przerwania, którego dotyczy, oraz opcjonalny parametr definiujący sposób, w jaki zostanie wygenerowany prolog i epilog funkcji. Parametr ten może przyjmować następujące wartości:

1. **ISR\_BLOCK** — jest to wartość domyślna. Funkcja tak zadeklarowana będzie wykonywana z zablokowanymi przerwaniami, przerwania zostaną odblokowane po jej zakończeniu. Definiując taką funkcję, możemy pominąć atrybut `ISR_BLOCK`.
2. **ISR\_NOBLOCK** — funkcja odblokuje przerwania najszybciej, jak to tylko jest możliwe. W efekcie procedura obsługi przerwania będzie mogła być przerywana przez inne przerwania, być może także przez kolejne przerwanie, które ją wywołało. Opcję tę należy stosować bardzo ostrożnie, gdyż może ona doprowadzić do przepełnienia stosu i nieprawidłowego funkcjonowania programu. Najczęściej wykorzystuje się ją dla krótkich, niekrytycznych czasowo procedur, dzięki temu nie blokują one innych przerwań.
3. **ISR\_NAKED** — kompilator nie będzie generował prologu i epilogu funkcji. Za zachowanie wartości rejestrów odpowiedzialny jest programista. W praktyce opcja ta stosowana jest wyłącznie w sytuacji, kiedy procedurę obsługi przerwania piszemy całkowicie w asemblerze.
4. **ISR\_ALIASOF(vect)** — tworzy alias wektora przerwań. Dzięki temu kilka wektorów przerwań może wskazywać na tę samą procedurę obsługi. Jest to wygodne dla niektórych rodzajów przerwań, np. PCINT. Deklaracja `ISR(PCINT1_vect, ISR_ALIASOF(PCINT0_vect))`: spowoduje, że wektor przerwania PCINT1 będzie wskazywał na procedurę obsługi przerwania PCINT0. Procedura obsługi PCINT0 musi zostać wcześniej zadeklarowana, w przeciwnym wypadku kompilator zgłosi błąd mówiący o niezdefiniowanym symbolu dotyczącym przerwania, do którego tworzymy alias.



We wcześniejszych wersjach AVR-libc funkcje obsługi przerwań były deklarowane za pomocą makrodefinicji `SIGNAL()`, będącej odpowiednikiem `ISR(vector)`, oraz `INTER_UPRUPT()`, będącej odpowiednikiem `ISR(vector, ISR_NOBLOCK)`. Obecnie te makrodefinicje nie powinny być używane, a w przyszłości ich implementacja zostanie usunięta.

Obowiązkowym argumentem makra `ISR` jest numer wektora obsługi przerwania. Możemy go podawać bezpośrednio jako `_VECTOR(nr_przerwania)` lub też możemy użyć wygodnej do zapamiętania nazwy przerwania. Np. dla mikrokontrolera ATMega88 wektor 1 wskazuje na przerwanie INT1, możemy więc napisać `ISR(_VECTOR(1))` lub czytelniej `ISR(INT1_vect)`. Drugi sposób jest sposobem zalecanym — w przypadku zmiany procesora i w konsekwencji zmiany kolejności wektorów przerwań stosowanie nazwy zamiast numeru przerwania jest bardziej uniwersalne. Jeśli na użytym procesorze dane przerwanie nie występuje, to zostanie wygenerowany błąd. Np. na mikrokontrolerze ATMega88 nie występuje przerwanie o nazwie `PSC0_CAPT_vect`. Poniższy kod:

```
ISR(PSC0_CAPT_vect)
{
}
```

spowoduje wygenerowanie ostrzeżenia i błędu:

```
warning: 'PSCO_CAPT_vect' appears to be a misspelled signal handler
make: *** [ksiazka.lss] Error 5
Build failed with 1 errors and 1 warnings...
```

**Sposób sygnalizowania błędu zależy od użytego skryptu programu make, stąd powinniśmy zwrócić uwagę na towarzyszące mu ostrzeżenie o nieprawidłowo zapisanej nazwie przerwania.**

Biblioteka AVR-libc zawiera dwie specjalne makrodefinicje procedur obsługi przerwań. Pierwszą jest EMPTY\_INTERRUPT(vector), definiująca funkcję obsługi przerwania, która składa się tylko z jednej instrukcji RETI, a więc powrotu z obsługi przerwania, a kompilator nie generuje dla niej prologu ani epilogu.



Wskazówka

Używając tej makrodefinicji, nie definiujemy ciała funkcji.

EMPTY\_INTERRUPT(vector) jest bardzo rzadko używana, głównie w sytuacji, kiedy jakieś przerwanie może wystąpić, ale nie chcemy dla niego definiować żadnej procedury obsługi. Przerwanie może też zostać przypadkowo wygenerowane na skutek błędnego kodu programu — kiedy myłkowo zezwolimy na dane przerwanie. Jednak normalnie sytuacja taka nie powinna mieć miejsca. Makro to jest użyteczne na etapie debugowania programu do wyłapywania powyższych błędów.

Druga makrodefinicja, BADISR\_vect, jest aliasem \_\_vector\_default. W efekcie wszystkie wektory przerwań, dla których jawnie nie zadeklarujemy funkcji obsługi, będą wskazywać na funkcję BADISR\_vect. Domyślnie funkcja ta powoduje resetowanie procesora poprzez skok do instrukcji znajdującej się pod adresem 0.



Wskazówka

Pamiętaj, że taki reset nie przywraca domyślnych ustawień rejestrów IO procesora.

Makro to może być wykorzystane do debugowania programu. Podstawiając pod nie własną funkcję obsługi przerwania, możemy przechwycić wszystkie przerwania, dla których jawnie nie zdefiniowaliśmy procedur obsługi. W gotowym programie użyteczność tej makrodefinicji jest niewielka, gdyż sytuacja, w której dochodzi do wywołania przerwania, dla którego nie zdefiniowaliśmy procedury obsługi, nie powinna mieć miejsca. Umieszczenie w kodzie programu definicji:

```
ISR(BADISR_vect)
{
}
```

spowoduje, że wektory wszystkich niewykorzystanych przerwań będą wskazywać na zadeklarowaną powyżej procedurę obsługi.

## sei() / cli()

Makrodefinicje te powodują odpowiednio odblokowanie i zablokowanie przerwań po przez modyfikację flagi I w rejestrze SREG. W programie wykorzystującym przerwania sei() powinno wystąpić po inicjalizacji programu, co umożliwia zezwolenie na przerwania.



Makrodefinicji cli() nigdy nie powinniśmy używać w programie.

Przyczyny, dla których w normalnym programie cli() nie powinno być wykorzystywane, zostaną omówione w podrozdziale poświęconym atomowemu dostępowi do zasobów. Ponieważ w programie nie powinniśmy wykorzystywać makra cli(), nie zachodzi także potrzeba użycia makra sei(), z wyjątkiem początku programu. Zamiast tego powinniśmy używać makrodefinicji ATOMIC\_BLOCK, co umożliwia bezpieczne i przewidywalne tworzenie sekcji krytycznych w programach, w których przerwania powinny być zablokowane.

## Atrybut naked i obsługa przerwań w asemblerze

Procedura obsługi przerwania powinna być możliwie najkrótsza. Zapewnia to bezproblemową obsługę przerwań. Aby mieć pełną kontrolę nad kodem wygenerowanym przez kompilator, czasami zachodzi potrzeba napisania procedury obsługi przerwania w asemblerze. Kod asemblerowy możemy dodać jako krótką wstawkę w procedurze obsługi przerwania napisanej w języku C lub możemy całą procedurę napisać w asemblerze. Najprościej w tym celu wykorzystać wbudowany w kompilator języka C asembler. Napisanie ciała funkcji w asemblerze nie daje nam jednak kontroli nad tworzonym automatycznie prologiem i epilogiem funkcji, którego celem jest zachowanie stanu wszystkich rejestrów procesora, w tym szczególnie rejestru statusu (ang. *Flags*). Możemy zabronić kompilatorowi tworzenia prologu i epilogu, stosując atrybut ISR\_NAKED, który jest zdefiniowany następująco:

```
# define ISR_NAKED      __attribute__((naked))
```

Widzimy więc, że taki sam efekt uzyskamy, stosując słowo kluczowe \_\_attribute\_\_ z argumentem naked. Po napotkaniu funkcji z atrybutem naked kompilator przestaje generować dla niej prolog i epilog, o zachowanie stanu rejestrów i ich odtworzenie musi więc zadbać programista. Ponieważ pisząc w języku C, nie mamy kontroli nad tym, jakie rejesty wykorzysta kompilator po zdefiniowaniu tego atrybutu, mamy dwie opcje prawidłowego napisania funkcji obsługi przerwania:

1. Możemy w prologu zachować stan wszystkich rejestrów i w epilogu je odtworzyć. Nie daje nam to jednak żadnej przewagi nad pozostawieniem tego zadania kompilatorowi, a więc rozwiązania tego nigdy nie stosujemy.
2. Całe ciało funkcji będzie napisane w asemblerze, na stos odkładamy wyłącznie te rejesty, których stan zmodyfikowaliśmy.

Widzimy więc, że stosując funkcję z atrybutem naked, musimy całe ciało funkcji napisać w asemblerze<sup>1</sup>. Generalnie warto pisać w asemblerze bardzo krótkie i proste funkcje, które są krytyczne czasowo. Dla dłuższych zysk pod postacią skrócenia kodu i skrócenia czasu wykonania funkcji zwykle jest nieistotny. Dla zilustrowania użycia atrybutu naked przeanalizujemy następujący kod:

```
static char licznik;

ISR(PCINT0_vect, ISR_NAKED)
{
    asm volatile("PUSH R24\n\t"
                "IN R24, 0x3f\n\t"
                "LDS R24, licznik\n\t"
                "INC R24\n\t"
                "STS licznik, R24\n\t"
                "OUT 0x3f, R24\n\t"
                "POP R24\n\t"
                "RETI\n\t"
                ::);
}

ISR(INT0_vect)
{
    licznik++;
}
```

Stworzyliśmy dwie funkcje obsługujące różne przerwania (co dla naszego przykładu jest bez znaczenia). Jedna została napisana w asemblerze (`ISR(PCINT0_vect, ISR_NAKED)`), druga (`ISR(INT0_vect)`) w języku C. Obie robią dokładnie to samo — inkrementują zawartość zmiennej o nazwie *licznik*. Funkcja napisana w asemblerze wydaje się być nieco bardziej skomplikowana, co jest jednak związane z koniecznością zachowania zawartości używanych rejestrów. Normalnie tą częścią pracy zająłby się kompilator. Porównajmy, jak wygląda kod asemblerowy obu funkcji. Najpierw nasza funkcja, której ciało napisaliśmy w asemblerze:

```
ISR(PCINT0_vect, ISR_NAKED)
{
    asm volatile("PUSH R24\n\t"
                "7e: 8f 93      push r24
                80: 8f b7      in r24, 0x3f ;63
                82: 80 91 00 01 lds r24, 0x0100
                86: 83 95      inc r24
                88: 80 93 00 01 sts 0x0100, r24
                8c: 8f bf      out 0x3f, r24 ;63
                8e: 8f 91      pop r24
                90: 18 95      reti

    00000092 <_vector_1>:
        "RETI\n\t"
        ::);
}
```

<sup>1</sup> Jak od każdej reguły, także od tej istnieją wyjątki, przykład takiego wyjątku zostanie pokazany pod koniec tego rozdziału.

Jak widać, kompilator dokonał tylko niewielkich zmian — zamiast nazwy zmiennej *licznik* pojawił się jej adres w pamięci SRAM. Zgodnie z oczekiwaniami nie został wygenerowany prolog ani epilog funkcji. Gdybyśmy sami nie nakazali zachować rejestru statusu oraz rejestru R24, ich zawartość uległaby zniszczeniu, powodując nieprawidłowe działanie reszty programu.



**Wskazówka** Jeśli procedura obsługi przerwania nie modyfikuje rejestru statusu, to nie ma potrzeby zachowywać jego zawartości.

A teraz czas na kod wygenerowany przez kompilator na podstawie kodu w języku C:

```
ISR(INT0_vect)
{
    92: 1f 92      push r1
    94: 0f 92      push r0
    96: 0f b6      in r0, 0x3f ; 63
    98: 0f 92      push r0
    9a: 11 24      eor r1, r1
    9c: 8f 93      push r24
    licznik++;
    9e: 80 91 00 01 lds r24, 0x0100
    a2: 8f 5f      subi r24, 0xFF ; 255
    a4: 80 93 00 01 sts 0x0100, r24
}
    a8: 8f 91      pop r24
    aa: 0f 90      pop r0
    ac: 0f be      out 0x3f, r0 ; 63
    ae: 0f 90      pop r0
    b0: 1f 90      pop r1
    b2: 18 95      reti
```

Widzimy, że trzon funkcji, czyli inkrementacja zmiennej *licznik*, jest zrealizowany tak samo, za pomocą ciągu instrukcji:

```
9e: 80 91 00 01 lds r24, 0x0100
a2: 8f 5f         subi r24, 0xFF ; 255
a4: 80 93 00 01 sts 0x0100, r24
```

Kompilator zamiast instrukcji INC użył instrukcji odejmowania stałej, SUBI, jednak efekt ich działania jest taki sam — zmienna *licznik* jest inkrementowana o jeden. To, co zwraca uwagę, to bardzo długi prolog i epilog funkcji. Modyfikowana jest zawartość aż 3 rejestrów (R0, R1 i R24), co jest spowodowane koniecznością spełnienia pewnych założeń ABI (ang. *Application Binary Interface*) kompilatora. Kompilator gcc gwarantuje, że register R1 zawsze będzie zawierał 0, ponieważ w procedurze obsługi przerwania nie można domniemywać, że R1 istotnie zawiera 0, jest on więc odpowiednio modyfikowany. Podobnie, ponieważ niektóre instrukcje asemblera AVR (np. MUL) używają domyślnie pary rejestrów R0 i R1, również zachowywany jest register R0. Z kolei ładowanie zawartości wskazanej komórki pamięci możliwe jest tylko do rejestrów R16 – R31, stąd konieczność użycia trzeciego rejestrów. Widzimy więc, że kompilator wygenerował znacznie dłuższy kod niż napisany przez nas w asemblerze. Co więcej, kod ten wykonuje się ponad dwukrotnie dłużej (13 vs. 28 taktów procesora). Jednakże gdyby

procedura obsługi przerwania była bardziej skomplikowana, to różnica długości kodów byłaby pomijalna.



Nie używaj wstawek w asemblerze, jeśli nie masz naprawdę dobrych powodów, by to robić.

Oczywiście, zamiast używać wbudowanego w kompilator asemblera można całą procedurę napisać w osobnym pliku (koniecznie z rozszerzeniem *S*), a następnie dołączyć do projektu. Tym zagadnieniem będziemy się szerzej zajmować w rozdziale poświęconym łączeniu programów w asemblerze z programami w języku C.

## Modyfikator volatile

Słowo kluczowe *volatile* jest modyfikatorem, którego możemy użyć do poinstruowania kompilatora, jak ma traktować podaną zmienną. Jego użycie jest szczególnie szerokie w programowaniu systemów czasu rzeczywistego i mikrokontrolerów. Modyfikator ten wprowadzamy dopiero w tym miejscu książki, gdyż jest on szczególnie związany z obsługą przerwań. Przeanalizujmy poniższy kod:

```
int a=100;  
while(a) a--;
```

W założeniu kod ten przypisuje zmiennej *a* wartość 100, a następnie w pętli odejmuję od niej jeden, aż *a* osiągnie zero i w efekcie powyższa pętla zostanie zakończona. Wydawać by się mogło, że podobny kod możemy wykorzystać np. do realizacji opóźnień w programie. Efektem jego działania powinno być opóźnienie zależne od początkowej wartości zmiennej *a*. Spróbujmy go wykorzystać do napisania prostego kodu, który będzie generował falę prostokątną:

```
void fala()  
{  
    int a;  
    DDRB=1;  
    while(1)  
    {  
        PORTB=0;  
        a=255;  
        while(a) a--;  
        PORTB=1;  
        a=255;  
        while(a) a--;  
    }:  
}
```

Jeśli skompilujemy powyższy kod z opcją optymalizacji *-O0*, to powiniem on wygenerować falę prostokątną na pinie 0 portu B, której częstotliwość będzie zależna od pętli opóźniającej. Wydaje się, że kod działa poprawnie. Jednak kiedy skompilujemy go z włączoną optymalizacją (*-Os*), to zaobserwujemy coś dziwnego. Częstotliwość wygenerowanej fali znacznie się zwiększyła... Przyczyna tego stanie się jasna po sprawdzeniu wygenerowanego kodu asemblerowego:

```

void fala()
{
    int a;
    DDRB=1;
c0: 81 e0      ldi r24, 0x01 ; 1
c2: 84 b9      out 0x04, r24 ; 4
    while(1)
    {
        PORTB=0;
c4: 15 b8      out 0x05, r1 ; 5
        a=255;
        while(a) a--;
        PORTB=1;
c6: 85 b9      out 0x05, r24 ; 5
c8: fd cf      rjmp .-6       ; 0xc4 <fala+0x4>
    }
}

```

Widzimy, że pętle opóźniające „zniknęły” z programu! Stało się tak dlatego, że optymizator „zauważał”, iż pętla `while` nic nie robi i można ją usunąć. Efektem tego jest zlikwidowanie opóźnień i zwiększenie częstotliwości wygenerowanej fali.



**Pamiętaj!** Jeśli Twój program świetnie działa z wyłączoną optymalizacją, a po jej włączeniu przestaje, to jedną z najczęstszych przyczyn takiego zachowania jest brak użycia słowa kluczowego `volatile` tam, gdzie jest to niezbędne.

Przeanalizujmy kolejny przykład:

```

char a;

ISR(TIMER1_OVF_vect)
{
    a++;
}

int main()
{
    a=0;
    while(a==0) {};
}

```

Program ten po skompilowaniu z wyłąconą optymalizacją działa prawidłowo, po jej włączeniu ku naszemu zaskoczeniu program nigdy nie opuszcza pętli `while`, pomimo że zmienna `a` jest prawidłowo inkrementowana w procedurze obsługi *timera*. Czyżby błąd kompilatora? Rzut okiem na wygenerowany kod asemblerowy:

```

ce: ff cf      rjmp .-2       ; 0xce <volatile_test+0x4>

```

utwierdza nas w przekonaniu, że stało się coś dziwnego. Pamiętając wcześniejszą wskazówkę, szybko znajdujemy rozwiązanie — zmienna `a` powinna być zadeklarowana z modyfikatorem `volatile`. W efekcie prowadzi to do wygenerowania kodu zgodnie z naszymi oczekiwaniami:

```
while(a==0) {};
ce: 80 91 02 01 lds r24, 0x0102
d2: 88 23      and r24, r24
d4: e1 f3      breq .-8 ; 0xce <volatile_test+0x4>
```

Widzimy, że tym razem zgodnie z naszymi oczekiwaniami za każdym razem zmienna `a` pobierana jest z pamięci i testowany jest warunek zakończenia pętli. Czym zatem jest słowo kluczowe, od którego tyle zależy? **Najogólniej pisząc, volatile mówi kompilatorowi, że zmienna zadeklarowana z tym modyfikatorem może być modyfikowana w sposób niewynikający bezpośrednio z otaczającego kodu.** Przy odwołaniu do takiej zmiennej kompilator zawsze będzie pobierał jej wartość z pamięci, unikając tworzenia jej kopii np. w rejestrach procesora. Efektem ubocznym jest wydłużenie kodu, gdyż zmienna zadeklarowana z tym atrybutem przy każdej operacji będzie odczytywana z pamięci, a następnie nowa wartość będzie natychmiast ponownie zapisywana.

**Wskazówka**

Dodanie modyfikatora `volatile` jest niezbędne w sytuacji, kiedy zmienna globalna jest modyfikowana w procedurze obsługi przerwania i jakiekolwiek innej części kodu lub kiedy jej wartość może zmieniać się w sposób niewynikający bezpośrednio z wykonywanego kodu.

Ten drugi przypadek obejmuje głównie zmienne, które odzwierciedlają rejesty sprawdzone, np. rejesty IO procesora. Dla przykładu stan rejestrów PINx procesora zależy od części sprzętowej układu, a nie od wykonywanego programu w języku C. Każdorazowo przy odwołaniu do takiego portu kompilator powinien wygenerować kod pobierający jego nową wartość, a nie używać wartości poprzednio odczytanej. Stąd też dostęp do rejestrów procesora odbywa się za pomocą predefiniowanych makr, które są rozwijane do definicji podobnej do przedstawionej poniżej:

```
(*(volatile uint8_t *)(mem_addr))
```

Powoduje ona, że zmienna traktowana jest jako wskaźnik na zasób sprzętowy, do którego chcemy uzyskać dostęp. Modyfikator `volatile` powoduje, że kompilator, napotykając na taką konstrukcję, każdorazowo będzie pobierał nową wartość z rejestru procesora.

Zgodnie z nowo nabytą wiedzą musimy nieco zmodyfikować miniprogramy z początku tego paragrafu, aby mogły poprawnie działać. Pętla opóźniająca powinna więc wyglądać następująco:

```
volatile int a=100;
while(a) a--;
```

Ponieważ zmienną `a` zdefiniowaliśmy z modyfikatorem `volatile`, kompilator nie wie, czy nie jest ona modyfikowana poza pętlą, nie może więc tej pętli zoptymalizować. Tak więc program:

```
volatile int a;
DDRA=1;
while(1)
{
    PORTA=0;
    a=255;
```

```

while(a) a--;
PORTA=1;
a=255;
while(a) a--;
};

```

będzie działał poprawnie, niezależnie od opcji optymalizacji. Zmodyfikujmy teraz kolejny program:

```

volatile char a;

ISR(TIMER1_OVF_vect)
{
    a++;
}

int main()
{
    a=0;
    while(a==0) {};
}

```

Zauważmy, że tym razem kompilator wygenerował kod zgodnie z naszymi oczekiwaniami, a pętla `while` nie została usunięta w procesie optymalizacji. Modyfikator `volatile` możemy dodawać nie tylko do typów prostych, ale także do typów złożonych. Przeanalizujmy kolejny przykład:

```

struct UCSRA
{
    bool RXCn :1;
    bool TXCn : 1;
    bool UDREn :1;
    bool FEn : 1;
    bool DORn : 1;
    bool UPEn : 1;
    bool U2Xn : 1;
    bool MPCMn : 1;
};

```

W tym przypadku, jeśli chcielibyśmy stworzyć zmienną współdzieloną pomiędzy program główny a procedurę obsługi przerwania o typie `UCSRA`, możemy to osiągnąć na kilka sposobów. Możemy bezpośrednio zdefiniować zmienną o wskazanym typie jako `volatile`:

```
volatile UCSRA *zmienna;
```

Mogimy także zmienić definicję struktury:

```

struct UCSRA
{
    volatile bool RXCn :1;
    volatile bool TXCn : 1;
    volatile bool UDREn :1;
    volatile bool FEn : 1;
    volatile bool DORn : 1;
    volatile bool UPEn : 1;

```

```
volatile bool U2Xn : 1;  
volatile bool MPCMn : 1;  
};
```

lub prościej:

```
volatile struct UCSRA  
{  
    bool RXCn :1;  
    bool TXCn : 1;  
    bool UDREn :1;  
    bool FEn : 1;  
    bool DORn : 1;  
    bool UPEn : 1;  
    bool U2Xn : 1;  
    bool MPCMn : 1;  
};
```

W przypadku struktury zadeklarowanej z modyfikatorem `volatile` automatycznie każde jej pole zachowuje się tak, jakby było zdefiniowane z tym modyfikatorem. Zastanówmy się, która z powyższych definicji jest najlepsza. Jak to zwykle bywa, wszystko zależy od okoliczności. Jeśli wiemy, że zmienne o danym typie zawsze będą wymagały atrybutu `volatile`, to lepiej umieścić go w deklaracji struktury. Uchroni to programistę przed przypadkowym opuszczeniem słowa kluczowego `volatile`, co spowodowałoby nieprawidłowe działanie programu. Ma to jednak wadę polegającą na niemożności zadeklarowania zmiennej o podanym typie bez modyfikatora `volatile`, co — jak za chwilę zobaczymy — czasami bywa przydatne. Możemy więc zadeklarować strukturę normalnie, pamiętając, aby przy zmiennych tego wymagających dodawać modyfikator `volatile`. Możemy także zadeklarować dwie wersje struktury, jedną bez modyfikatorem `volatile`, a drugą z modyfikatorem:

```
struct UCSRA  
{  
    bool RXCn :1;  
    bool TXCn : 1;  
    bool UDREn :1;  
    bool FEn : 1;  
    bool DORn : 1;  
    bool UPEn : 1;  
    bool U2Xn : 1;  
    bool MPCMn : 1;  
};  
typedef volatile struct UCSRA UCSRA_v;  
USCRA zmienna_nievolatile;  
UCSRA_v zmienna_volatile;
```

W powyższym przykładzie zadeklarowaliśmy dwa typy: jeden, `UCSRA`, odzwierciedlający nam strukturę rejestru kontrolnego A UARTa, a drugi, `UCSRA_v`, będący taką samą strukturą, ale zmienna zadeklarowana z jej pomocą będzie miała dodatkowo modyfikator `volatile`.

Modyfikator `volatile` „wyłącza” optymalizację dotyczącą zmiennej zadeklarowanej za jego pomocą, a więc odwołania do niej stają się bardzo kosztowne, zarówno pod względem czasu egzekucji programu, jak i liczby potrzebnych do tego instrukcji (a więc

objętości generowanego kodu). W procedurze obsługi przerwania lub w bloku krytycznym, w którym przerwania są zablokowane i mamy pewność, że nic nie zmodyfikuje naszej zmiennej poza głównym ciągiem instrukcji, moglibyśmy chwilowo „wyłączyć” działanie modyfikatora `volatile`. Niestety, standard języka C nie przewiduje takiej możliwości. Jednak łatwo możemy tego dokonać, stosując sztuczkę polegającą na przypisaniu wartości zmiennej zadeklarowanej z modyfikatorem `volatile` do innej zmiennej, bez tego modyfikatora. Następnie operacje przeprowadzamy na tak utworzonej zmiennej pomocniczej, której ostateczną wartość z powrotem przepisujemy do pierwszej zmiennej. Ilustruje to przykład:

```
volatile int x;

ATOMIC_BLOCK(ATOMIC_RESTORE_STATE)
{
    int xtmp=x;
    ttmpx*=tmpx;
    tmpx<<=1;
    x=tmpx;
}
```

W powyższym przykładzie zmienna `x` jest zmienną o atrybutie `volatile`. Przeprowadzone na niej operacje są w sekcji krytycznej, w której przerwania są zablokowane. Mamy więc gwarancję, że nic poza ciągiem instrukcji w tej sekcji nie zmodyfikuje zmiennej `x`. Stąd też dla wygenerowania optymalniejszego kodu przepisaliśmy wartość zmiennej `x` do zmiennej tymczasowej, `tmpx`, na której prowadzimy dalsze operacje, których wynik z powrotem przepisujemy do zmiennej `x`. Oprócz sekcji krytycznych miejscem, gdzie możemy bezpiecznie posłużyć się powyższą sztuczką, są procedury obsługi przerwań. Domyślnie procedura obsługi przerwania wykonywana jest z zablokowanymi przerwaniami (są więc wykonywane atomowo), a więc mamy gwarancję, że nasze `x` nie zostanie zmodyfikowane w innym miejscu programu. Jeśli obsługa przerwania odbywa się z odblokowanymi przerwaniami, to powyższy trick też można bezpiecznie stosować, pod warunkiem że nasza zmienna nie jest modyfikowana w procedurach obsługi innych przerwań. Jeśli zmienna `volatile` wskazuje na zasób sprzętowy mogący ulec zmianie w trakcie wykonywania kodu programu, to oczywiście jej przypisanie do zmiennej pomocniczej powoduje „zamrożenie” stanu tego zasobu w zmiennej pomocniczej.

## Zmiana kolejności instrukcji

Modyfikator `volatile` ma jeszcze jedno zastosowanie. Normalnie optymalizator może przestawiać wyrażenia, aby zapewnić optymalny przebieg programu, np. sekwencja:

```
int x=10;
x=y+2;
x=s+2;
```

wcale nie musi zostać wykonana w kolejności, w jakiej ją zapisaliśmy. Wynik programu będzie dokładnie taki sam, niezależnie, czy najpierw wykonamy operację `x=y+2`, czy `x=s+2`. W związku z tym optymalizator przestawi kolejność tych wyrażeń tak, aby otrzymać krótszy lub szybszy kod wynikowy. W rozdziale 9., „Interfejs XMEM”, stanęliśmy przed sytuacją, w której sekwencja instrukcji musiała zostać wykonana w ścisłe określonej kolejności. W sekwencji:

```
XMCRB|= _BV(XMM0); //Zamaskuj linię adresową AD15 - jej stan będzie równy 0
char byte=*((volatile char*)(adres | 0x8000)); //Odczytaj bajt
XMCRB&=~ _BV(XMM0); //Przywróć normalną funkcję linii AD15
```

modyfikator `volatile` uniemożliwia optymalizatorowi zmianę kolejności operacji, co w efekcie spowodowałoby odczytanie komórki pamięci o nieprawidłowym adresie. W sytuacji, w której zamiast `(volatile char*)` zastosowalibyśmy samo `(char*)`, nie byłoby gwarancji, że zachowana zostałaby podana sekwencja instrukcji. Deklaracje wszystkich rejestrów IO procesora zawierają modyfikator `volatile`, dzięki czemu optymalizator nie zmienia kolejności podanych instrukcji. W rezultacie ciąg instrukcji:

```
POR TA=0;
POR TA=0xFF;
POR TA=0;
```

zawsze zostanie wykonany w podanej kolejności.

## Atomowość dostępu do danych

W paragrafie poświęconym modyfikatorowi `volatile` rozważaliśmy działanie następującego kodu:

```
volatile char a;

ISR(TIMER1_OVF_vect)
{
    a++;
}

int main()
{
    a=0;
    while(a==0) {};
}
```

Zastanówmy się, czy gdyby zmienić typ zmiennej `a` z typu `char` na np. typ `int`, to czy program również wykonywałby się prawidłowo? Pozornie tak, ale zobaczymy, jak w takim przypadku jest realizowana pętla `while(a==0) {}`. Okazuje się, że porównanie to realizowane jest „na raty”:

```
while(a==0) {};
d2: 80 91 02 01 lds r24, 0x0102
d6: 90 91 03 01 lds r25, 0x0103
da: 89 2b          or r24, r25
dc: d1 f3          breq .-12      ; 0xd2 <volatile_test+0x8>
```

Najpierw odczytywana jest starsza część 16-bitowej zmiennej, a następnie młodsza. Co jeśli przerwanie *timera* nastąpi pomiędzy tymi operacjami? W przypadku kiedy `a` zawierało 0 i pętla powinna zostać przerwana, kolejne przerwanie może zmienić jej stan na 1 i warunek nie będzie spełniony. Podobnie w poniższym przykładzie:

```
int a;

ISR(TIMER1_OVF_vect)
```

```

{
    a++;
}

int main()
{
    a=0;
    while(1)
    {
        a++;
    };
}

```

Inkrementacja zmiennej a czasami będzie prowadziła do nieprawidłowych wyników. Aby temu zapobiec, musimy zapewnić, aby operacje przeprowadzone na zmiennej a były atomowe, to znaczy aby niemożliwe było ich przerwanie. W tym celu możemy lokalnie blokować przerwania instrukcją `cli()`, a następnie je odblokowywać instrukcją `sei()`:

```

int a;

ISR(TIMER1_OVF_vect)
{
    a++;
}

int main()
{
    a=0;
    while(1)
    {
        cli();
        a++;
        sei();
    };
}

```

Jednak takie działanie nie jest zalecane. Optymalizator kompilatora jest zobowiązany generować poprawny kod, w tym celu może przestawiać kolejność operacji, jeśli jest to bez znaczenia dla działania programu. Powyższy przykład jest tak prosty, że optymalizator nie ma wielkiego pola do popisu, ale gdyby instrukcje `cli()` poprzedzały inne instrukcje albo po `sei()` następowaly kolejne, to optymalizator mógłby zmienić ich pozycję, niepotrzebnie rozszerzając zakres instrukcji znajdujących się w tak utworzonej sekcji krytycznej. Aby zwiększyć wygodę i zminimalizować ryzyko błędu, wprowadzono makrodefinicje, znajdujące się w pliku nagłówkowym `<util/atomic.h>`. W pliku tym znajduje się makrodefinicja `ATOMIC_BLOCK(typ)`, która przyjmuje dodatkowy parametr określający sposób jej działania. Parametr ten przybiera następujące postaci:

- ◆ `ATOMIC_FORCEON`
- ◆ `ATOMIC_RESTORESTATE`
- ◆ `NONATOMIC_FORCEOFF`

Najczęściej wykorzystujemy opcję `ATOMIC_RESTORESTATE`, powodującą zapamiętanie przed wejściem do sekcji krytycznej globalnej flagi zezwolenia na przerwania, zablokowanie

przerwań, wykonanie kodu z sekcji i na końcu przywrócenie stanu flagi zezwolenia na przerwanie. W związku z tym, jeśli przed wejściem do sekcji krytycznej przerwania są zablokowane, to po jej opuszczeniu również zostaną zablokowane. Jeśli były włączone, to po opuszczeniu sekcji zostaną włączone. W niektórych sytuacjach wiemy jednak, jaki jest zastany stan flagi zezwolenia na przerwania, nie musimy więc go zapamiętywać. W takiej sytuacji możemy zastosować parametr ATOMIC\_FORCEON, powodujący, że po opuszczeniu sekcji krytycznej przerwania będą odblokowane, lub rzadziej NONATOMIC\_→FORCEOFF, powodujący, że przerwania po opuszczeniu sekcji pozostaną zablokowane.

Nasz przykładowy kod wykorzystujący makrodefinicję ATOMIC\_BLOCK wygląda więc następująco:

```
int a;

ISR(TIMER1_OVF_vect)
{
    a++;
}

int main()
{
    a=0;
    while(1)
    {
        ATOMIC_BLOCK(ATOMIC_FORCEON)
        {
            a++;
        };
    };
}
```

Wykorzystaliśmy tu parametr ATOMIC\_FORCEON, gdyż stan flagi zezwolenia na przerwanie jest znany.



Makrodefinicja sei() powinna być wykorzystywana tylko raz w programie, do odblokowania przerwań. Nigdy nie powinna zajść sytuacja, w której konieczne byłoby wykorzystanie makrodefinicji cli().

W przypadku kiedy testujemy warunek logiczny pętli, tak jak w pierwszym przykładzie, nie jest możliwe wykorzystanie makrodefinicji ATOMIC\_BLOCK. Aby taki program działał poprawnie, musimy w sposób atomowy przepisać zawartość licznika do zmiennej pomocniczej, a następnie w pętli testować tę zmienną:

```
volatile char a;

ISR(TIMER1_OVF_vect)
{
    a++;
}

int main()
{
    a=0;
```

```
int tmpa=0;
while(tmpa==0)
{
    ATOMIC_BLOCK(ATOMIC_FORCEON)
    {
        tmpa=a;
    };
}
```

W tym przypadku operację porównania wykonujemy na kopii zmiennej `a`, nie zachodzi więc obawa, że zostanie ona niespodziewanie zmodyfikowana. Zauważmy, że operacja przypisania zmiennej `tmpa` wartości zmiennej `a` nie jest w mikroprocesorach AVR operacją atomową, stąd zachodzi konieczność zastosowania makrodefinicji `ATOMIC_BLOCK`.



Wskazówka

Osoby znające asembler mikroprocesorów AVR z pewnością w tym momencie zaprotestują. Niektóre architektury AVR mają zaimplementowaną wykonywaną atomowo instrukcję `MOVW Rd,Rr`. Niestety, pisząc w języku wysokiego poziomu, jakim jest język C, nie mamy gwarancji, że instrukcja `tmpa=a` zostanie zamieniona na wykonywaną atomowo instrukcję asemblera `MOVW`.

W sytuacji kiedy w dużej sekcji krytycznej, chcemy czasowo zezwolić na nieatomowe wykonywanie niektórych instrukcji, użyteczna może okazać się makrodefinicja `NONATOMIC_BLOCK(typ)`. Jej parametry są identyczne jak w przypadku makra `ATOMIC_BLOCK`.

## Funkcje reentrant

Kod *reentrant* to kod, który może być wykonywany jednocześnie z dwóch lub więcej wątków. Co prawda kojarzy się to z systemami wielozadaniowymi, ale w pewnych warunkach sytuacja taka może występować także podczas programowania na AVR bez obecności systemu operacyjnego. Funkcja może zostać ponownie wywołana przed zakończeniem jej działania w dwóch sytuacjach:

- ◆ W przypadku funkcji rekurencyjnych — jednak w tym wypadku miejsce przerwania funkcji jest dokładnie znane.
- ◆ W przypadku funkcji wywoływanych w procedurze obsługi przerwań — w tym wypadku miejsce przerwania funkcji przez przerwanie jest nieprzewidywalne, co stwarza pewne kłopoty.

Podeczas wykonywania procedury obsługi przerwania istnieje możliwość wywołania innych funkcji, może więc dojść do sytuacji, w której wykonywana funkcja jest przerwana, a następnie wywoływana ponownie z procedury obsługi przerwania. Musimy sobie zdawać sprawę z tego, że nie każda funkcja może być bezpiecznie wywoływana w sytuacji, w której jej poprzednie wywołanie nie zostało zakończone. Problem funkcji *reentrant* zwykle nie dotyczy funkcji, które operują wyłącznie na zmiennych lokalnych. Zmienne takie tworzone są na stosie i każde wywołanie funkcji operuje na swojej lokalnej kopii zmiennych. Wyjątkiem od tej reguły są zmienne lokalne zadeklarowane z modyfikatorem `static`. W tym przypadku istnieje tylko jedna kopia zmiennej, która z punktu

widzenia funkcji zachowuje się jak zmienna globalna. Zdecydowanie „niebezpieczne” są funkcje operujące na danych zewnętrznych — zmiennych globalnych lub wskaźnikach do współdzielonych zasobów lub danych statycznych. Z pewnością kłopoty sprawią funkcje, które odwołują się bezpośrednio do zasobów sprzętowych: funkcje wysyłające dane przez magistralę procesora, realizujące transmisje I2C, 1-wire, SPI itd. Aby mogły być bezpiecznie wywoływane z procedury obsługi przerwania, muszą być napisane w specjalny sposób.

Zdecydowana większość funkcji w AVR-libc jest *reentrant*, wyjątkiem są funkcje pokazane w tabeli 14.2.

**Tabela 14.2.** Funkcje biblioteki AVR-libc, które nie są reentrant

Funkcja	Problem	Rozwiążanie
rand(), random()	Używają zmiennych globalnych	Użyj wersji reentrant rand_r(), random_r()
strtod(), strtol(), strtoul()	Używają zmiennej globalnej errno	Funkcje są reentrant, jeśli aplikacja nie korzysta z kodów błędów zwracanych w errno lub są one chronione makrem ATOMIC_BLOCK
malloc(), realloc(), calloc(), free()	Używają stosu i innych struktur globalnych do zarządzania pamięcią	Ochrona za pomocą ATOMIC_BLOCK lub napisanie własnych funkcji zarządzających pamięcią
fdevopen(), fclose()	Używają funkcji calloc() i free()	jw.
eeprom_*( ), boot_*( )	Używają rejestrów IO	jw.
printf(), printf_P(), vprintf(), vprintf_P(), puts(), puts_P()	Zmieniają globalny strumień stdout	jw. Użycie innych podobnych funkcji, jak sprintf(), sprintf_P(), snprintf(), snprintf_P(), vsprintf(), vsprintf_P(), vsnprintf(), vsnprintf_P(), jest bezpieczne
fprintf(), fprintf_P(), vfprintf(), vfprintf_P(), fputs(), fputs_P()	Problem występuje tylko wtedy, gdy jako argument podany jest ten sam strumień o typie FILE	Strumień podany jako argument wywołania musi być różny dla kolejnych wywołań
assert()	Wywołuje funkcję fprintf()	jw.
clearerr()	Zmienia argument FILE	jw.
getchar(), gets()	Korzysta ze strumienia globalnego stdin	jw. Aczkolwiek czytanie z Stein w różnych wątkach nie ma sensu.
fgetc(), ungetc(), fgets(), scanf(), scanf_P(), fscanf(), fscanf_P(), vscanf(), vfscanf(), vfscanf_P(), fread()	Problem występuje tylko wtedy, gdy jako argument podany jest ten sam strumień o typie FILE	jw.

Jak widać, w wielu przypadkach rozwiązaniem problemu funkcji *reentrant* jest wywoływanie ich jako operacji atomowych, np.:

```
ATOMIC_BLOCK(ATOMIC_RESTORESTATE)
{
    void *ptr=malloc(100);
}
```

W ten sposób mamy gwarancję, że funkcja `malloc` nie zostanie przerwana i wywołana ponownie np. w procedurze obsługi przerwania. Rozwiązanie to jest efektywne, jeśli możemy pozwolić sobie na zablokowanie obsługi przerwań na czas wywołania takiej funkcji.



Najbezpieczniej jest jednak unikać wywołania w procedurze obsługi przerwania funkcji, co do których nie jesteśmy pewni, że są one funkcjami *reentrant*.

## Przykłady praktyczne

Do tej pory pokazane zostały liczne przykłady wykorzystania przerwań i związane z tym problemy. Pora wykorzystać te informacje do stworzenia kilku konkretnych przykładów, które potem będzie można z powodzeniem wykorzystać w swoich projektach. W rozdziale 12., poświęconym portom IO, pokazanych zostało kilka przykładów użycia wyświetlaczy 7-segmentowych i urządzeń umożliwiających komunikację z użytkownikiem (przyciski, enkoder, klawiatura). Pokazane tam przykłady można z powodzeniem wykorzystać w większości budowanych układów, ale pewne rzeczy można zoptymalizować w taki sposób, aby procesor był mniej obciążony pracą. Dzięki temu możemy zmniejszyć częstotliwość jego taktowania, co przekłada się na zmniejszone zużycie energii i bardziej niezawodną pracę. Pewne rzeczy można też zrobić prościej. Główną wadą przedstawionych w rozdziale 12. rozwiązań było to, że opierały się na technice *poolingu* — ciągłego odpytywania o stan urządzenia oraz oczekiwania w pętli. Teraz pokazane zostanie, jak osiągnąć to samo, lecz przy znacznie mniejszym zaangażowaniu procesora. Pokazane zostaną także rzeczy praktycznie niemożliwe do osiągnięcia bez właściwego wykorzystania przerwań.



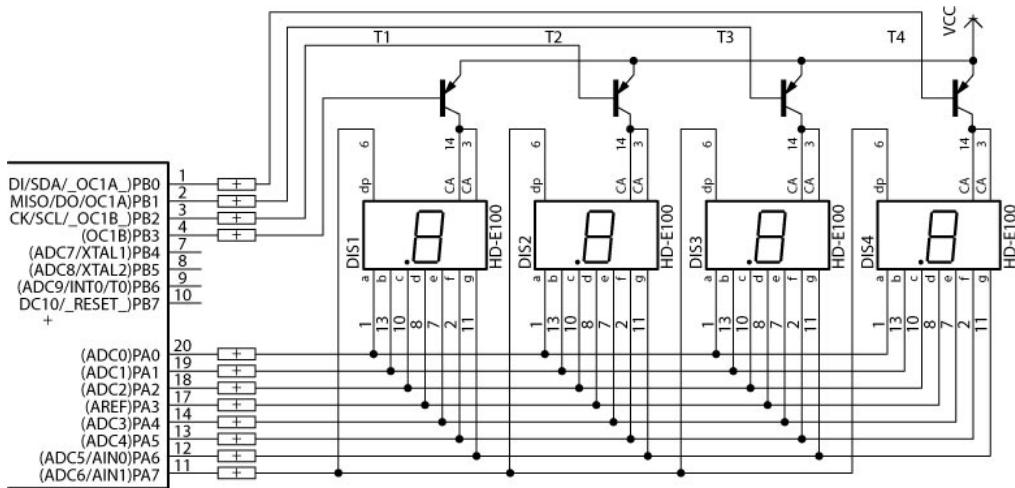
Wszystkie poniższe przykłady znajdują się w formie gotowych projektów w katalogu R14.

Zostały one przystosowane do procesora ATTINY261/461/861, jednak po niewielkich zmianach będą działać poprawnie praktycznie na każdym procesorze AVR.

## Wyświetlanie multipleksowane

Dopóki sterujemy jednym wyświetlaczem 7-segmentowym, możemy go łatwo podłączyć do dowolnego portu IO mikrokontrolera, co daje nam nad nim pełną kontrolę. Niemniej jedna cyfra dziesiętna to niewiele — zwykle, aby przedstawić stan danego urządzenia, potrzebujemy ich więcej — 2, 3, a czasami (np. do wyświetlenia daty lub czasu) nawet 4 do 6 cyfr. Bezpośrednie ich podłączenie do mikrokontrolera jest możliwe,

ale komplikuje i podrażnia układ. Musimy zastosować mikrokontroler z większą liczbą portów IO lub stosować zewnętrzne układy scalone sterujące wyświetlaczem. Istnieje jednak prostsze rozwiązanie — wykorzystanie multipleksowania. Ten obco brzmiący termin oznacza metodę polegającą na sekwencyjnym wyświetlaniu danych naraz tylko na jednym z kilku wyświetlaczy. Wyświetlacz, na którym aktualnie coś wyświetlamy, jest wybierany przy pomocy specjalnego klucza, w tym czasie pozostałe wyświetlacze są wygaszone. Jeśli aktywny wyświetlacz będzie zmieniany wystarczająco szybko, to na skutek bezwładności oka nie uda nam się zauważyc, że w danej chwili włączony jest tylko jeden wyświetlacz — będziemy mieli wrażenie, że wszystkie świecą naraz, a każdy wyświetla przeznaczoną dla niego informację. Sposób podłączenia wyświetlaczy umożliwiający ich sterowanie multipleksowe pokazany został na rysunku 14.1.

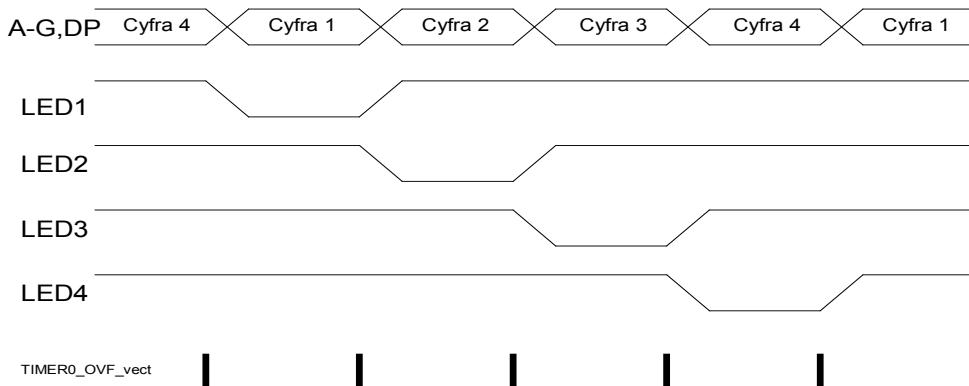


**Rysunek 14.1.** Połączenie wyświetlaczy 7-segmentowych LED z mikrokontrolerem w sposób umożliwiający sterowanie multipleksowe

Rolle klucza aktywującego dany wyświetlacz spełniają tranzystory T1-T4. Kiedy na sterującym tranzystorem pinie IO mikrokontrolera panuje stan niski, tranzystor zostaje włączony i prąd z zasilania może płynąć do połączonych anod diod tworzących wyświetlacz. Kiedy pin jest w stanie wysokim, tranzystor jest zablokowany, a sterowany przez niego wyświetlacz wyłączone. Teoretycznie wyświetlacz moglibyśmy sterować bezpośrednio z pinu IO procesora, z pominięciem klucza tranzystorowego. Jednak takie postępowanie nie jest zalecane. Dla większości procesorów AVR maksymalna wydajność prądowa pinu IO wynosi zaledwie 40 mA. Jeśli włączymy wszystkie diody, to na każdą przypadka będzie zaledwie  $40/7 = 5,7$  mA. Wartość ta może okazać się niewystarczająca do uzyskania satysfakcyjującego poziomu świecenia wyświetlacza. Na rysunku 14.1 widzimy, że wyprowadzenia A – G oraz DP wszystkich wyświetlaczów są ze sobą zwarte — dzięki temu do sterowania czterema wyświetlaczami potrzebujemy tylko 4+8 pinów IO. Przy klasycznym sterowaniu potrzebowalibyśmy aż  $4*8 = 32$  pinów IO. Co więcej, jeśli będziemy chcieli podłączyć 8 wyświetlaczów, to potrzebujemy tylko 4 dodatkowe piny. Liczbę pinów sterujących kluczami tranzystorowymi możemy dodatkowo zmniejszyć, stosując układ demultiplesera, np. 74xx138 — jest to demultiplexer

3 na 8, to znaczy, że niski stan logiczny pojawia się na wyjściu, którego numer podany został na linie adresowe A0 – A2 demultiplexera. Dzięki temu przy wyświetlaniu 8 cyfr możemy zaoszczędzić dodatkowych 5 linii IO.

Wróćmy jednak do schematu przedstawionego na rysunku 14.1. Aby wyświetlić 4 cyfry dziesiętne, musimy kolejno aktywować każdy z wyświetlaczów i na czas jego aktywacji wystawiać, na porcie łączącym mikrokontroler z segmentami A – G wyświetlacza, informację, która na wybranym wyświetlaczu ma zostać pokazana. Sposób sterowania pokazany został na rysunku 14.2.



**Rysunek 14.2.** Przebiegi elektryczne na pinach IO sterujących wyświetlaczem. LED1 – LED4 to kolejne wyświetlatce 7-segmentowe. Na czas aktywacji danego wyświetlacza (odpowiedni sygnał LEDx w stanie niskim) na porcie łączącym mikrokontroler z segmentami A – G wystawiana jest wartość, która na wybranym wyświetlaczu ma zostać pokazana. Gdy sygnał LEDx jest w stanie wysokim, wyświetlacz jest wygaszony i nie reaguje na stan sygnałów A – G

Stosując takie cykliczne wyświetlanie, uzyskujemy pożądany efekt — widzimy wyświetlane cyfry. Cały problem sterowania tak połączonymi wyświetlaczami sprawdza się więc do odpowiednio szybkiego cyklicznego wyświetlania treści przeznaczonej dla każdego z wyświetlaczów z osobna. Możemy to robić w pętli, lecz w takiej sytuacji trudno by było to sprząć z innymi funkcjami. Najlepiej, gdyby przy konieczności przełączenia wyświetlacza procesor niezależnie od wykonywanego programu wstrzymywał aktualnie wykonywaną funkcję, przełączał wyświetlacz i wznowiał przerwaną pracę. Do tego celu idealnie nadają się przerwania. Sygnałem nakazującym przełączenie aktywnego wyświetlacza będzie przerwanie generowane przez timer po osiągnięciu przez jego licznik maksymalnej wartości. W pokazanym przykładzie wykorzystany został 8-bitowy Timer0 i przerwanie TIMERO\_OVF\_vect. Każdorazowo, kiedy licznik osiągnie wartość 0xFF, generowane jest przerwanie, procedura jego obsługi przełącza aktywny wyświetlacz i odpowiednio wysterowuje piny A – G sterujące segmentami:

```
#define LEDDISPNO 4

volatile uint8_t LEDDIGITS[LEDDISPNO];

ISR(TIMERO_OVF_vect)
{
    static uint8_t LEDNO;
```

```
PORTB|=0x0F; //Wyłącz wszystkie wyświetlacze
LEDNO=(LEDNO+1)%LEDDISPNO;
ShowOnLED(LEDDIGITS[LEDNO]);
PORTB=(PORTB & 0xF0) | (~(1<<LEDNO) & 0x0F); //Wybierz kolejny wyświetlacz
}
```

Powyższa procedura wykorzystuje pokazaną w rozdziale 12. funkcję ShowOnLED, która wyświetla podaną cyfrę na wyświetlaczu 7-segmentowym. Najpierw deklarujemy tablicę LEDDIGITS, która ma tyle pozycji, ile cyfr chcemy wyświetlić. Zauważmy, że tablica ta musi zostać zdefiniowana z modyfikatorem volatile — dostęp do tej tablicy będzie się odbywał i w przerwaniu, i w programie głównym. Zapisując w programie nową wartość na danej pozycji tablicy, z pewnością nie chcielibyśmy, aby w ramach optymalizacji kompilator przechowywał nową wartość tymczasowo w którymś z rejestrów procesora — w takiej sytuacji pomimo uaktualnienia zawartości tablicy LEDDIGITS na wyświetlaczu ciągle obserwovalibyśmy jej starą zawartość. Każda pozycja tej tablicy określa stan jednego wyświetlacza. Następnie w procedurze obsługi przerwania przepelenia Timer0 zwiększamy o jeden numer aktywnego wyświetlacza (zmienna LEDNO) — ponieważ jest to zmienna statyczna, jej stan jest zachowany pomiędzy kolejnym wywołaniem funkcji obsługi przerwania. Dzięki zadeklarowaniu jej wewnątrz funkcji do zmiennej tej nie mamy dostępu z reszty programu, w efekcie nie uda nam się jej przypadkowo zmodyfikować. Trochę skomplikowany może się wydawać sposób sterowania tranzystorami sterującymi poszczególnymi wyświetlaczami. Najpierw wszystkie wyświetlacze wyłączały — jest to niezbędne, gdyż w przeciwnym wypadku aktualnie wybrany wyświetlacz przez krótki czas wyświetlałby informację nieprzeznaczoną dla niego — mogłyby to czasami powodować nieprzyjemne miganie wyświetlacza. Następnie, po wpisaniu nowych wartości do portu sterującego segmentami A – G wyświetlacza, na podstawie nowej wartości zmiennej LEDNO wybieramy wyświetlacz, który ustawiową wcześniej wartość ma wyświetlać.

Pozostaje jeszcze odpowiednio zainicjować timer:

```
void Timer0Init()
{
    TCCR0B=_BV(CS01); //Preskaler CLKIO/8
    TIMSK1=_BV(TOIE0); //Odblokuj przerwanie nadmiaru timera 0
}
```

Timer0 będzie taktowany zegarem taktującym procesor podzielonym przez 8 — na jedno zliczenie timera przypadać więc będzie 8 cykli zegara taktującego procesor. Odblokowane także zostaje przerwanie przepelnilenia timer0. Przerwanie to będzie generowane co  $8 \times 256 = 2048$  cykli zegara.

Teraz możemy napisać prosty program demonstracyjny:

```
int main()
{
    LEDDDR=0xFF; //Wszystkie piny portu są wyjściem
    DDRB|=0x0F; //Piny PB0-PB3 jako wyjścia
    Timer0Init();
    sei();

    LEDDIGITS[0]=1;
    LEDDIGITS[1]=2;
```

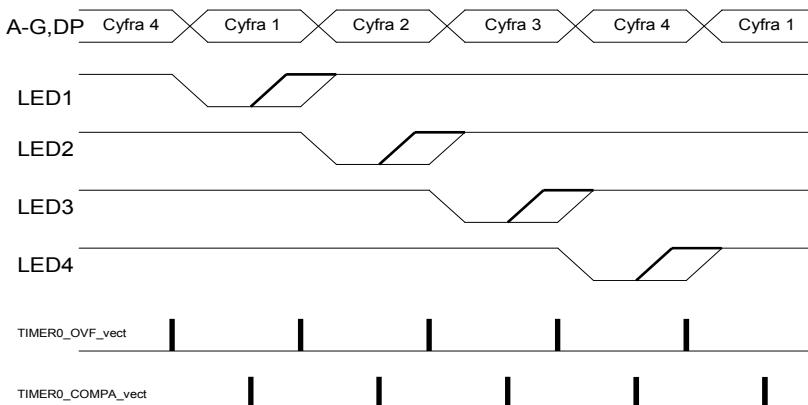
```
LEDDIGITS[2]=3;  
LEDDIGITS[3]=4;  
  
    while(1) {};  
}
```

Na wyświetlaczu powinna zostać wyświetlona liczba 4321. Widzimy więc, że korzystanie z wyświetlania multipleksowanego jest bardzo proste. Po prawidłowym skonfigurowaniu *timera* wyświetlanie odbywa się niezależnie od działania programu głównego, nie musimy więc zaprzatać sobie nim głowy.

## Wyświetlanie multipleksowane z regulacją jasności wyświetlacza

Skoro w powyższym przykładzie wykorzystany został jeden *timer* do realizacji multipleksowania, spróbujmy go wykorzystać jeszcze bardziej. Normalnie jasność wyświetlacza możemy regulować poprzez odpowiedni dobór wartości rezystorów ograniczających prąd poszczególnych segmentów. Sposób ten jest prosty, ma jednak dużą wadę: w zmontowanym układzie, jeśli okaże się, że wyświetlacz świeci zbyt jasno, to niewiele możemy na to poradzić — jedyną możliwością jest zmiana rezystorów. Często chcielibyśmy, aby wyświetlacz dostosowywał swoją jasność do otoczenia — aby cyfry były dobrze widoczne w jasny dzień, wyświetlacz musi świecić o wiele jaśniej niż w sytuacji, kiedy na niego patrzymy w ciemnym pokoju. Powstaje więc problem — jak prościej niż przez wymianę rezystorów ograniczających prąd regulować jasność świecenia wyświetlacza czy, ogólniej mówiąc, dowolnej diody LED? W tym celu stosuje się technikę PWM (ang. *Pulse Width Modulation*), czyli technikę modulacji szerokości impulsów. Normalnie diodę sterujemy, stosując stały prąd i napięcie, w efekcie uzyskujemy stały strumień świetlny, co obserwujemy jako świecenie. Co by się stało, gdyby diodę sterować nie stałe, a np. impulsami o wypełnieniu 50%? Dioda przez połowę czasu by świeciła, a przez połowę była zgaszona. Jeśli częstotliwość przebiegu sterującego byłaby odpowiednio wysoka, to oko nie zauważałoby, że dioda mruga — zamiast tego zaobserwowałibyśmy stałe świecenie. Jednak w tej sytuacji strumień świetlny byłby o połowę mniejszy, w efekcie wydawałoby nam się, że dioda świeci ciut słabiej. W ten sposób uzyskaliśmy to, co chcieliśmy — modulując szerokość impulsu sterującego diodą (percentowy stosunek czasu świecenia do okresu przebiegu sterującego), możemy zmieniać jasność świecenia diody. **Zależność obserwowanej jasności diody względem współczynnika wypełnienia impulsów ze względu na budowę oka nie jest zależnością liniową**. Nie ma to dla nas jednak większego znaczenia. Zastanówmy się, w jaki sposób można to wykorzystać do regulacji jasności wyświetlacza. Na rysunku 14.2 pokazane zostały przebiegi sterujące poszczególnymi wyświetlaczami (sygnały LED1 – LED4). Każdy wyświetlacz świeci dokładnie przez czas równy  $1/\text{liczba\_wyświetlaczy}$ . Gdyby ten czas świecenia regulować (np. skracić go), można by w ten sposób regulować jasność (rysunek 14.3).

W tym celu ponownie wykorzystamy Timer0 i zaimplementowane w nim rejestrory OCR (ang. *Output Compare Register*). Zawartość tych rejestrów jest stale porównywana z aktualną wartością licznika *timera*. Jeśli są one sobie równe, to generowane jest przerwanie TIMER0\_COMPA\_vect. Dzięki wykorzystaniu rejestrów OCR0A możemy więc uzyskać



**Rysunek 14.3.** Regulacja jasności świecenia wyświetlacza LED poprzez modulację długości impulsu sterującego

przerwanie po zaprogramowanym czasie od wystąpienia przerwania `TIMER0_OVF_vect`. Jeśli w procedurze obsługi przerwania `TIMER0_COMPA_vect` będziemy wyłączać wszystkie wyświetlacze, to możemy w ten sposób skrócić czas świecenia każdego z wyświetlaczy, a co za tym idzie, programowo regulować jego jasność. Spróbujmy to praktycznie zrealizować. W tym celu wprowadzimy do kodu z poprzedniego przykładu dodatkową funkcję obsługi przerwania `TIMER0_COMPA_vect`:

```
ISR(TIMER0_COMPA_vect)
{
    PORTB|=0x0F; //Włącz wszystkie wyświetlacze
}
```

Jej zadaniem jest wygaszenie wszystkich wyświetlaczy. Pozostaje nam jeszcze inicjalizacja timera0 tak, aby powyższe przerwanie było generowane:

```
void Timer0InitWithDimmer()
{
    TIMSK|=BV(OCIE0A); //Włącz przerwanie Compare Match A
    OCR0A=255;
    Timer0Init();
}
```

Powyższa funkcja odwołuje się do funkcji inicjującej `timer0` z poprzedniego przykładu, dodatkowo włączane jest przerwanie `TIMER0_COMPA_vect` i inicjowany jest rejestr `OCR0A`. Zobaczmy, jak działa nasza regulacja:

```
int main()
{
    LEDDDR=0xFF; //Wszystkie piny portu są wyjściem
    DDRB|=0x0F; //Piny PB0-PB3 jako wyjścia
    Timer0InitWithDimmer();
    sei();

    LEDDIGITS[0]=1;
    LEDDIGITS[1]=2;
    LEDDIGITS[2]=3;
    LEDDIGITS[3]=4;
```

```

while(1)
{
    OCR0A=200;
    _delay_ms(1000);
    OCR0A=100;
    _delay_ms(1000);
}:
}

```

W wyniku działania powyższego programu powinniśmy zaobserwować cykliczne zmiany jasności wyświetlacza LED.



Uwaga

Jeśli procesor taktowany jest zegarem o wyższej częstotliwości, to możemy odpowiednio zwiększyć *preskaler* timera0, tak aby przerwania nie były generowane zbyt często, co niepotrzebnie obciąża procesor. Aby nie dało się zaobserwować migania, wystarczy, że wyświetlacz będzie odświeżany z częstotliwością kilkuset Hz.

Zajmijmy się jeszcze pewną kosmetyką i optymalizacją powyższych funkcji. Jeśli przejrzymy wygenerowany plik *lss*, to zauważymy, że prosta procedura obsługi przerwania *TIMER0\_COMPA\_vect* po skompilowaniu rozrasta się do całkiem sporego kodu:

```

ISR(TIMER0_COMPA_vect)
{
    ee: 1f 92      push r1
    f0: 0f 92      push r0
    f2: 0f b6      in r0, 0x3f ; 63
    f4: 0f 92      push r0
    f6: 11 24      eor r1, r1
    f8: 8f 93      push r24
PORTB|=0XF; //Wyłącz wszystkie wyświetlacze
    fa: 88 b3      in r24, 0x18 ; 24
    fc: 8f 60      ori r24, 0x0F ; 15
    fe: 88 bb      out 0x18, r24 ; 24
}
100: 8f 91      pop r24
102: 0f 90      pop r0
104: 0f be      out 0x3f, r0 ; 63
106: 0f 90      pop r0
108: 1f 90      pop r1
10a: 18 95      reti

```

Możemy tu nieco pomóc kompilatorowi<sup>2</sup>:

```

ISR(TIMER0_COMPA_vect, ISR_NAKED)
{
    PORTB|=0x01; //Wyłącz wszystkie wyświetlacze
    PORTB|=0x02;
    PORTB|=0x04;
    PORTB|=0x08;
    asm volatile("RETI\n\t:::");
}

```

<sup>2</sup> Pokazana funkcja będzie działać poprawnie jedynie w sytuacji, w której włączona jest optymalizacja.

Jak pamiętamy, atrybut ISR\_NAKED powoduje, że kompilator nie generuje prologu i epilogu funkcji. Nie jest on nam potrzebny, gdyż ustawianie poszczególnych bitów rejestru PORTB jest operacją niewymagającą zaangażowania żadnych rejestrów procesora, nie wpływa ona też na rejestr stanu (stąd też w powyższym przykładzie wykonaliśmy 4 oddzielne zapisy; zapisanie od razu wartości 0x0F wymagałoby użycia dodatkowego rejestru). Teraz wygenerowany kod wygląda następująco:

```
ISR(TIMER0_COMPA_vect, ISR_NAKED)
{
    PORTB|=0x01; //Wyłącz wszystkie wyświetlacze
    ee: c0 9a      sbi 0x18, 0 : 24
    PORTB|=0x02;
    f0: c1 9a      sbi 0x18, 1 : 24
    PORTB|=0x04;
    f2: c2 9a      sbi 0x18, 2 : 24
    PORTB|=0x08;
    f4: c3 9a      sbi 0x18, 3 : 24
    asm volatile("RETI\n\t:::");
    f6: 18 95      reti
```

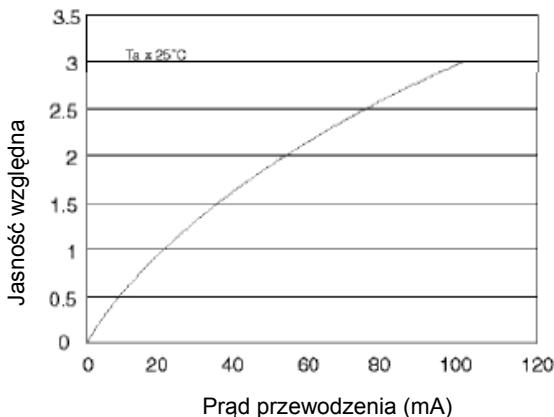
Udało nam się skrócić procedurę obsługi przerwania aż 3 razy, stosując proste rozwiązania.

## Trochę więcej o prądzie i jasności diody

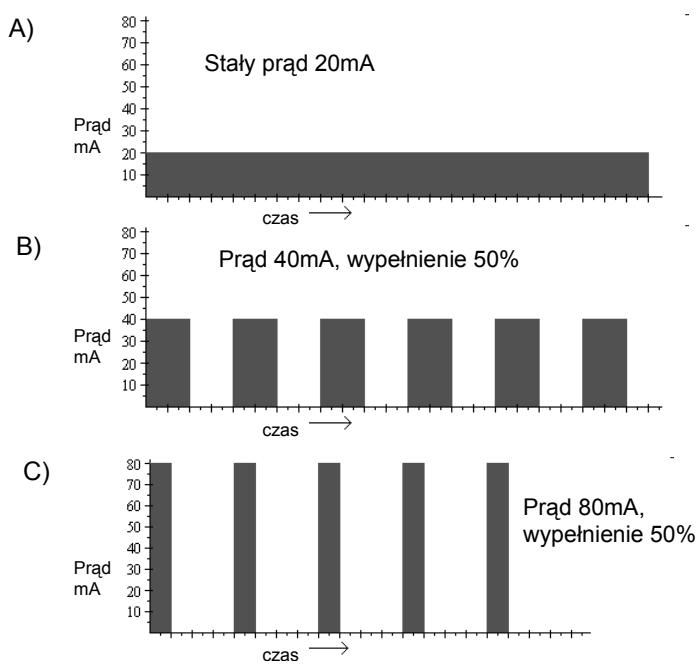
W poprzednich przykładach wykorzystano wyświetlanie multipleksowe oraz regulację jasności diody poprzez regulację współczynnika wypełnienia impulsu sterującego. Wykorzystując te techniki, musimy zwrócić uwagę na kilka potencjalnych problemów. Z pewnością zauważyleś, że po podłączeniu przez te same rezystory ograniczające 4 wyświetlacze 7-segmentowych sterowanych multipleksowo ich jasność w stosunku do jednego wyświetlacza uległa zmniejszeniu. Jest to spowodowane tym, że w takiej sytuacji wyświetlacze świeią z wypełnieniem 25% — przez 75% okresu wyświetlacz jest wyłączony. Przy stosowaniu 8 wyświetlaczów sterowanych multipleksowo każdy z nich świeciłby tylko przez  $\frac{1}{8}$  okresu, a więc z mocą 12,5% pojedynczego wyświetlacza. Uzyskana w ten sposób jasność może okazać się niewystarczająca. Tu jednak z pomocą przychodzą nam same właściwości diody LED. Producent diody podaje jej dopuszczalny prąd przewodzenia (ang. *Forward Current*) — wartość prądu, który stale płynąc przez diodę, nie spowoduje jej uszkodzenia. Oprócz tej wartości producent określa także maksymalny prąd szczytowy [ang. *Maximum (peak, surge) forward current*]. Jest to maksymalny prąd, jaki może przepływać przez diodę spolaryzowaną w kierunku przewodzenia. Jest on zwykle wielokrotnie, nawet 100-krotnie wyższy niż prąd przewodzenia diody. Jest jednak pewien haczyk — taki prąd może przez diodę przepływać przez bardzo krótki czas — jest to związane z pojemnością cieplną złącza diody i maksymalną mocą strat. **Ogólna zasada mówi, że im wyższy jest impuls prądowy, tym krótszy musi być jego czas trwania.** Co nam to daje? Jasność świecenia diody jest proporcjonalna do płynącego prądu (rysunek 14.4), sterując więc diodę prądem wyższym niż normalny prąd przewodzenia, możemy uzyskać znacznie większą jasność diody.

Ponieważ wyświetlacze sterowane multipleksowo są włączone tylko przez  $1/n$  okresu (gdzie  $n$  to liczba wyświetlaczów), bezpiecznie możemy zwiększyć ich prąd, zwiększając w ten sposób ich jasność (rysunek 14.5).

**Rysunek 14.4.**  
Zależność jasności diody od płynącego przez nią prądu



**Rysunek 14.5.**  
Sterowanie diod LED przy pomocy krótkich impulsów prądowych zwiększających jej jasność: a) dioda jest sterowana stałym prądem, b) dioda sterowana jest prądem dwukrotnie większym, lecz płynie on tylko przez 50% okresu, c) dioda sterowana jest prądem czterokrotnie wyższym, lecz płynie on tylko przez 25% czasu trwania okresu. We wszystkich przypadkach prąd średni jest taki sam i wynosi 20 mA



Ponieważ — jak pokazano na rysunku 14.4 — jasność diody nie rośnie proporcjonalnie do płynącego prądu, zwiększając prąd szczytowy, tylko do pewnego stopnia możemy skompensować utratę jasności wynikającą z multipleksowania.

## Obsługa przycisków

Obsługa przycisków, a zwłaszcza technika eliminacji drgań zrealizowana przy pomocy przerwań, wygląda podobnie do techniki zastosowanej w przypadku *poolingu*. Jest jedna drobna, lecz niezwykle istotna różnica. Do sprawdzania stanu przycisków możemy wykorzystać przerwanie pojawiające się co ścisłe określony czas, np. przerwanie pełnienia *timera*, które w poprzednim przykładzie wykorzystaliśmy do multipleksowego

sterowania wyświetlaczem LED. Ponieważ przerwanie to pojawia się okresowo, możemy je jednocześnie wykorzystać do dekrementacji licznika określającego czas eliminacji drgań. **Dzięki temu unikamy jawnnej deklaracji opóźnienia, a więc sytuacji, w której moc obliczeniowa procesora marnowana jest na wykonywanie pustych pętli.** Ta pozornie mała zaleta stanowi cały sens wykorzystania przerwań do odczytu stanu przycisków, enkodera, klawiatury i innych układów interfejsów. Spróbujmy więc w oparciu o przerwania zrealizować ten sam przykład, który został pokazany w rozdziale 12. Dysponujemy tym samym układem, z tym że tym razem mamy podłączone 4 cyfry wyświetlacza 7-segmentowego, sterowane multipleksowo; do dyspozycji mamy także dwa przyciski podłączone do pinów PB4 i PB5 mikrokontrolera, których naciśnięcie spowoduje zwiększenie/zmniejszenie o 1 wyświetlanej na wyświetlaczu liczby.

Najpierw zdefiniujmy pomocniczą funkcję, której zadaniem będzie konwersja podanej 16-bitowej liczby na elementy tablicy LEDDIGITS, zawierającej wyświetlane na kolejnych wyświetlaczach cyfry. Samo wyświetlanie zrealizowane jest dokładnie tak samo jak w poprzednim przykładzie.

```
void itoLED(uint16_t l)
{
    for(uint8_t i=0;i<LEDDISPNO;i++)
    {
        LEDDIGITS[i]=l%10;
        l/=10;
    }
}
```

Funkcja itoLED działa niezwykle prosto, kolejnym elementom tablicy LEDDIGITS przypisywane są kolejne reszty z dzielenia przez 10. Funkcja wykonuje tyle iteracji, ile mamy wyświetlaczy, a więc możliwych do wyświetlenia cyfr.



Pamiętajmy, że pokazane w poprzednim przykładzie funkcje multipleksowego sterowania wyświetlaczem LED na bieżąco wyświetlają zawartość tablicy LEDDIGITS, więc każda wpisana do niej wartość praktycznie natychmiast pojawi się na wyświetlaczu.

Procedura eliminująca drgania styków przycisków w oparciu o przerwania wygląda następująco:

```
volatile uint8_t key=0xFF;                                //Stan przycisków
void ReadButtons()
{
    static uint8_t Counters[2];

    uint8_t k=PINB;
    if(Counters[0]==0)
    {
        if(((k^key) & _BV(PB4)) && ((k & _BV(PB4))==0)) Counters[0]=100; //Czas, przez jaki
                                                                                   //stan przycisku będzie ignorowany
        key&=~_BV(PB4);
        key|= (k & _BV(PB4));
    } else Counters[0]--;
}
```

```

if(Counters[1]==0)
{
  if(((k^key) & _BV(PB5)) && ((k & _BV(PB5))==0)) Counters[1]=100; //Czas, przez jaki
                                                                           //stan przycisku będzie ignorowany
  key&=~_BV(PB5);
  key|= (k & _BV(PB5));
} else Counters[1]--;
}

```

Aby mogła ona poprawnie funkcjonować, musimy wywoływać ją cyklicznie, najlepiej przez jakąś procedurę obsługi przerwania. Ponieważ w pokazanych przykładach taką cyklicznie wywoływaną procedurą jest procedura obsługi przerwania przepełnienia timer0, możemy w tej procedurze umieścić wywołanie funkcji ReadButtons():

```

ISR(TIMER0_OVF_vect)
{
  static uint8_t LEDNO;

  PORTB|=0x0F;                                     //Wylacz wszystkie wyświetlacze
  LEDNO=(LEDNO+1)%LEDDISPNO;
  ShowOnLED(LEDDIGITS[LEDNO]);
  PORTB=(PORTB & 0xF0) | (~(1<<LEDNO) & 0x0F); //Wybierz kolejny wyświetlacz
  ReadButtons();
}

```

Ponieważ funkcja ReadButtons() będzie wywoływana wyłącznie w jednym miejscu, nie ma sensu tworzyć osobnej procedury — lepiej, żeby została ona włączona w miejscu jej wywołania, co zredukuje długość kodu. W tym celu zdefiniowano jej prototyp następująco:

```
static inline void ReadButtons();
```

Zobaczmy, jak nasze nowe funkcje sprawują się w programie:

```

int main()
{
  LEDDDR=0xFF;                                     //Wszystkie piny portu są wyjściem
  PORTB|=_BV(PB4) | _BV(PB5); //Włącz pull up na pinie PB4 i PB5
  uint16_t liczba=1234;                            //Początkowa wyświetlana wartość

  DDRB|=0x0F;                                       //Piny PB0-PB3 jako wyjścia
  Timer0Init();
  sei();

  uint8_t oldkey=0xFF;                             //Przechowuje poprzedni stan klawiszy

  while(1)
  {
    itoLED(liczba);
    if((oldkey ^ key) & _BV(PB4)) && (key & _BV(PB4))==0) liczba++;
    if((oldkey ^ key) & _BV(PB5)) && (key & _BV(PB5))==0) liczba--;
    oldkey=key;
  }
}

```

Wprowadzenie zmiennej `oldkey` jest niezbędne w celu detekcji zmiany stanu przycisku. Bez niej, dopóki dany przycisk byłby wciśnięty, zmienna `liczba` byłaby inkrementowana/dekrementowana. Ponieważ mikrokontroler w ciągu sekundy wykonuje bardzo dużo obiegów tej pętli, w efekcie wartość wyświetlana na wyświetlaczu, nawet przy krótkim naciśnięciu przycisku, zmieniałaby się bardzo szybko.



Wskazówka

Jako pouczające doświadczenie z modyfikatorem `volatile`, sprawdź, jak zadziała powyższy program, kiedy zmienią key będzie zdefiniowana bez tego modyfikatora. Spróbuj wyjaśnić przyczyny zaobserwowanego działania programu<sup>3</sup>.

Stosunkowo często można spotkać się z przykładami, w których mechaniczny element, jakim jest przycisk, podłączony jest do pinu IO procesora, który może wyzwalać przerwanie. Zdaniem autora, nie jest to dobre rozwiązanie. Drgające styki spowodują, że przy każdej zmianie stanu przycisku generowane będzie przerwanie. Jednorazowo przy zmianie stanu przycisku może zostać wygenerowanych nawet kilkadziesiąt przerwań, które trzeba będzie obsłużyć.

## Obsługa enkodera

Obsługa enkodera przy użyciu przerwań niewiele się różni od techniki pokazanej w rozdziale 12. Cała procedura jest podobna:

```
int8_t enc_delta;  
  
void ReadEncoder()  
{  
    static int8_t last;  
    static uint8_t laststate;  
    static uint8_t counters[2];           //Tablica zawierająca liczniki  
    int8_t newpos, diff;  
  
    uint8_t state=PINB;  
    if((state^laststate) & _BV(PB4)) && (counters[0]==0))  
    {  
        counters[0]=200;  
        laststate&=(~_BV(PB4));  
        laststate|= (state & _BV(PB4));  
    }  
  
    if((state^laststate) & _BV(PB5)) && (counters[1]==0))  
    {  
        counters[1]=200;  
        laststate&= (~_BV(PB5));  
        laststate|= (state & _BV(PB5));  
    }  
  
    uint8_t przerwa=0;  
    for(uint8_t c=0;c<2;c++)  
        if(counters[c])
```

<sup>3</sup> Aby różnica była widoczna, program musi zostać skompilowany z włączoną optymalizacją.

```

{
  counters[c]--;
  przerwa=1;                                //Robimy opóźnienie tylko, jeśli któryś z liczników był !=0
}

newpos=0;
if((PINB & _BV(PB4))==0) newpos=3;
if((PINB & _BV(PB5))==0) newpos=1; //konwersja kodu Graya na binarny
diff=last-newpos;
if(diff & 1)
{
  last=newpos;                                //bit 0 = krok
  enc_delta+=(diff & 2)-1;                    //bit 1 - kierunek
}
}

```

Różnica sprowadza się do usunięcia pętli opóźniającej, powyższa procedura jest wywoływana w określonych odstępach czasowych. Analogicznie jak w poprzednim przykładzie, również wywołanie funkcji ReadEncoder() należy umieścić w procedurze obsługi przerwania, np. użytego wcześniej wektora TIMER0\_OVF\_vect. Przetestujmy więc działanie enkodera opartego na przerwaniach:

```

int main()
{
  LEDDDR=0xFF;                                //Wszystkie piny portu są wyjściem
  PORTB|= _BV(PB4) | _BV(PB5);    //Włącz pull up na pinie PB4 i PB5
  uint16_t liczba=1234;

  DDRB|=0x0F;                                  //Piny PB0-PB3 jako wyjścia
  Timer0Init();
  sei();

  while(1)
  {
    liczba+=Read4StepEncoder();                //Wywołanie funkcji
    itoLED(liczba);
  }
}

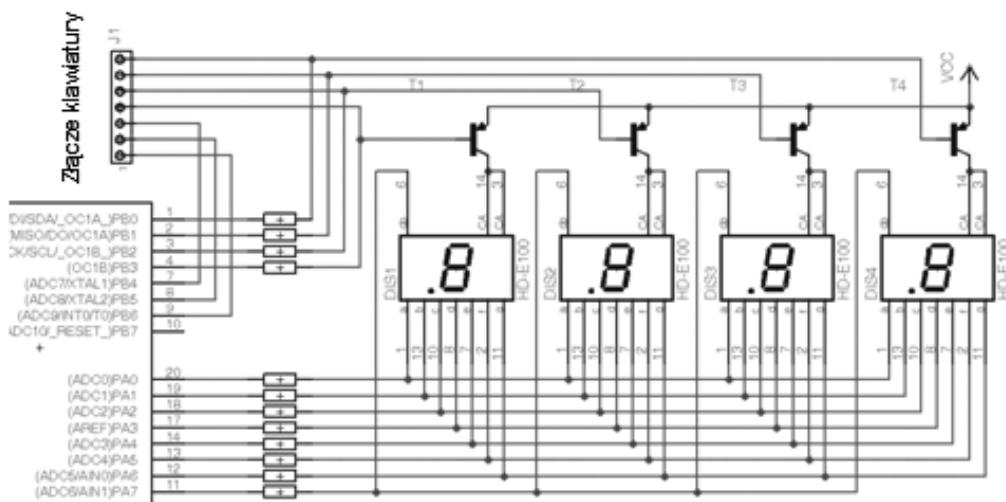
```

Dzięki powyższemu kodowi kręcenie enkoderem powinno zmieniać wyświetlaną na wyświetlaczu liczbę. Do odczytu stanu enkodera, w zależności od jego typu, należy wykorzystać jedną z procedur: Read1StepEncoder(), Read2StepEncoder() lub Read4StepEncoder(). Powyższe funkcje zwracają liczbę obrotów enkodera od czasu poprzedniego odczytu jego stanu. Liczba ta jest sumowana z wartością zmiennej liczba i wyświetlana na wyświetlaczu 7-segmentowym.

## Klawiatura matrycowa

Na koniec spróbujmy podłączyć dodatkowo klawiaturę matrycową o organizacji 4\*3. Tu na samym wstępnie pojawi się poważny problem. Do tej pory wykorzystywaliśmy procesor ATTINY461, który ma tylko 16 linii IO, z czego 12 już wykorzystaliśmy do podłączenia 4 wyświetlaczy LED sterowanych multipleksowo. Do podłączenia takiej klawiatury potrzebujemy dodatkowych 7 linii IO. Musimy więc zmienić procesor na

taki, który dysponuje większą liczbą dostępnych linii IO, lub pokombinować. To drugie rozwiązanie jest ciekawsze. Jak wcześniej wspomniano, dodatkowe linie IO można wygospodarować, stosując układy demultiplexerów, lecz wymaga to zastosowania dodatkowych układów scalonych. Lepszym rozwiązaniem wydaje się być współdzielienie niektórych linii IO pomiędzy klawiaturę a wyświetlacz LED. Zauważmy, że sterując poszczególnymi cyframi, wystawiamy na nie niski stan logiczny, podczas gdy na liniach podłączonych do pozostałych cyfr panuje stan wysoki. Takie sterowanie bardzo przypomina wybór poszczególnych segmentów/kolumn przy przeglądaniu klawiatury matrycowej. Dzięki temu do podłączenia potrzebujemy tylko 3 dodatkowe linie IO, a tyloma dysponujemy. Schemat całego układu przedstawia rysunek 14.6.



**Rysunek 14.6.** Schemat podłączenia klawiatury i 4 wyświetlaczy 7-segmentowych, tak aby niektóre linie IO były współdzielone pomiędzy nimi

Ponieważ wybór odczytywanego rzędu w tym przypadku zsynchronizowany jest z numerem aktywnego wyświetlacza, najrozsądniej będzie umieścić procedurę odczytu klawiatury razem z procedurą obsługi wyświetlaczy. Najpierw zdefiniujmy zmienną, która będzie przechowywać numer aktualnie wcisniętego klawisz:

```
volatile uint8_t key=0xFF;
```

Kiedy żaden klawisz nie jest wcisnięty, zmienna ta przyjmie wartość 0xFF. Ponieważ dostęp do niej będzie się odbywał zarówno w procedurze obsługi przerwania, jak i w innych częściach programu, musimy zadeklarować ją z modyfikatorem volatile. A oto, jak wygląda zmodyfikowana procedura odczytu klawiatury i sterowania wyświetlaczami LED:

```
ISR(TIMER0_OVF_vect)
{
    static uint8_t LEDNO;
    static uint8_t Counter;           //Debouncing
    static uint8_t row;              //Rzqd, w którym znajdował się ostatnio odczytany klawisz
```

```

PORTB=0x7F;                                //Wyłącz wszystkie wyświetlacze
LEDNO=(LEDNO+1)%LEDDISPNO;
ShowOnLED(LEDDIGITS[LEDNO]);
PORTB=0x70 | (~(1<<LEDNO) & 0x0F);    //Wybierz kolejny wyświetlacz
//Tu zaczynamy obsługę klawiatury
if(Counter==0)
{
    uint8_t x=PINB;
    if((x & 0x70)!=0x70)                  //Jakiś klawisz był wcisnięty?
    {
        if((x & 0x10)==0) x=0;            //Jeśli tak, to z której kolumny?
        else if((x & 0x20)==0) x=1;
        else x=2;
        key=x+LEDNO*3;                  //Oblicz numer klawisza
        row=LEDNO;
        Counter=100;
    } else if(row==LEDNO) key=0xFF;
    } else Counter--;
}

```

Pierwsza jej część to znana nam już procedura obsługi multipleksowej wyświetlaczy. Dodatkowo zdefiniowane zostały dwie zmienne: `Counter` — użyta do eliminacji drgań styków, oraz `row` — przechowująca rząd, w którym wykryto wcisnięty klawisz. Dzięki niej możemy wykryć sytuację zwolnienia przycisku i nadać zmiennej `key` wartość 255. Jest ona niezbędna, gdyż w jednym cyklu skanowana jest tylko jedna kolumna klawiatury (nie możemy wpływać na wybrany rząd, gdyż zakłóciłoby to działanie wyświetlaczy). Stąd detekcja sytuacji, w której żaden przycisk nie jest wcisnięty, nie jest taka prosta — wymaga informacji o tym, gdzie znajdował się ostatnio wcisnięty klawisz.

Pora zademonstrować działanie klawiatury:

```

int main()
{
    LEDDDR=0xFF;
    PORTB|= _BV(PB4) | _BV(PB5) | _BV(PB6);      //Wszystkie piny portu są wyjściem
    DDRB|=0x0F;                                     //Włącz pull up na pinie PB4, PB5 i PB6
                                                       //Piny PB0-PB3 jako wyjścia
    uint16_t liczba;
    Timer0Init();
    sei();
    while(1)
    {
        liczba=key;
        itoLED(liczba);
    }
}

```

Powyższy program wyświetla numer aktualnie wcisniętego klawisza lub 255, jeśli żaden nie jest wcisnięty.

## Rozdział 15.

# Przetwornik analogowo-cyfrowy

Mikrokontrolery AVR wyposażone są w przetwornik analogowo-cyfrowy (ADC, ang. *Analog-to-digital Converter*). Układ ADC umożliwia przetworzenie analogowego sygnału doprowadzonego do wejścia mikrokontrolera na postać cyfrową, nadającą się do wykorzystania w programie. Dzięki temu możemy do mikrokontrolera podłączyć różne źródła analogowe — np. przetworniki temperatury, ciśnienia, odległości. Napięcie przyłożone do pinu ADC mikrokontrolera zamieniane jest na liczbę z przedziału 0 – 1023 lub –512 – 511 w przypadku kanałów różnicowych.

Przetwornik oferuje 10-bitową rozdzielczość, przy czasach konwersji 13 – 260 µs, co umożliwia próbkowanie z częstotliwością 15 kSPS z maksymalną rozdzielczością lub większą przy redukcji rozdzielczości przetwornika. ADC może pracować w trybie różnicowym, w którym mierzona jest różnica napięć pomiędzy dwoma pinami procesora. Aby układ ACD mógł poprawnie pracować, niezbędne jest właściwie połączenie do zasilania masy i zasilania analogowego (w większości procesorów AVR są one wyprodukowane na osobne piny procesora) oraz wybranie napięcia referencyjnego. Mikrokontrolery AVR umożliwiają wybranie jako źródła napięcia referencyjnego napięcia zasilania ( $V_{cc}$ ), wewnętrznego źródła napięcia referencyjnego (o napięciu zależnym od konkretnego typu procesora; napięcie to mieści się zwykle w zakresie 1 – 2,54 V) lub zewnętrznego napięcia referencyjnego, podłączonego do pinu  $V_{ref}$  procesora. W przypadku precyzyjnych pomiarów należy stosować zewnętrzne stabilne źródła referencyjne, jednak w większości przypadków wystarczającą dokładność uzyskamy, stosując wewnętrzne źródło napięcia referencyjnego. Mikrokontrolery AVR dysponują tylko jednym układem ADC, lecz dzięki wbudowanemu multiplekserowi analogowemu można wybrać jedno z kilku wejść, na którym odbywa się pomiar napięcia.

Przetwornik ADC może pracować w dwóch trybach — trybie pojedynczej konwersji (ang. *Single Conversion Mode*) oraz w trybie ciągłej konwersji (ang. *Free Running Mode*). W trybie pojedynczej konwersji przetwornik ADC dokonuje jednego pomiaru, rozpoczęcie kolejnego wymaga jego ponownej aktywacji. W trybie ciągłej konwersji zaraz po zakończeniu jednego pomiaru od razu rozpoczynany jest kolejny, co umożliwia próbkowanie z maksymalną prędkością przetwornika. Po zakończeniu pomiaru wynik

odczytuje się z rejestru ADC przetwornika, dodatkowo zakończenie konwersji może być związane z wygenerowaniem przerwania. Dzięki temu można w prosty sposób gromadzić dane z przetwornika. Po tym skrótnym opisie przejdźmy do bardziej szczegółowego.



Wskazówka Wszystkie definicje związane z ADC znajdują się w pliku nagłówkowym `<avr\io.h>`.

## Wybór napięcia referencyjnego

Jak wspomniano, aby pomiar ADC mógł być wykonany, należy wybrać napięcie referencyjne, czyli napięcie, względem którego dokonywany będzie pomiar.



Wskazówka Mierzone napięcie nie może być wyższe od napięcia referencyjnego. Jeśli będzie większe, niezależnie od jego wartości z rejestru ADC zawsze odczytamy wartość maksymalną (0x3FF).

Zbyt wysokie napięcie referencyjne (pamiętajmy jednak, że nie może ono przekroczyć wartości napięcia zasilania przetwornika  $AV_{cc}$ ) ogranicza dostępną rozdzielcość przetwarzania. Jak pokazano w tabeli 15.1, źródło napięcia odniesienia można wybrać przy pomocy bitów REFS0 (ang. *Reference Selection*) i REFS1 rejestru ADMUX (ang. *ADC Multiplexer Selection Register*).

**Tabela 15.1.** Wybór napięcia referencyjnego przetwornika ADC

REFS1	REFS0	Źródło napięcia referencyjnego
0	0	Napięcie referencyjne pochodzi z pinu $V_{ref}$ , wewnętrzne źródło jest wyłączone.
0	1	Napięcie odniesienia pobierane jest z pinu $AV_{cc}$ .
1	0	Kombinacja zarezerwowana.
1	1	Wewnętrzne źródło napięcia odniesienia, o wartości zależnej od typu procesora.



Wskazówka Znajomość napięcia referencyjnego niezbędna jest do przeliczenia wyniku z rejestru ADC na wartość mierzonego napięcia.

Domyślnie bity te mają wartość 0, co umożliwia przyłączenie zewnętrznego napięcia referencyjnego. Należy pamiętać, aby niełączyć wewnętrznego napięcia referencyjnego, jeśli do pinu  $V_{ref}$  dostarczamy napięcie zewnętrzne. Dobrym pomysłem jest też, przy wykorzystaniu wewnętrznego napięcia referencyjnego, przyłączenie kondensatora 10-100nF pomiędzy pin  $V_{ref}$  a masą. Jego obecność poprawia filtrowanie szumów źródła referencyjnego. Pinu tego nie należy podłączać do napięcia zasilającego — jeśli chcemy, aby napięcie zasilające było jednocześnie napięciem referencyjnym, możemy je wybrać poprzez odpowiednią konfigurację bitów wyboru napięcia referencyjnego. Pamiętajmy też o odpowiednim połączeniu masy analogowej ( $AGND$ ) i zasilania analogo-

gowego ( $AVcc$ ). Nawet jeśli przetwornik ADC nie jest wykorzystywany, należy podłączyć te piny do zasilania — piny  $I0$  portu, który współdzieli wyprowadzenia z przetwornikiem ADC, są zasilane z tych źródeł.

## Multiplekser

Po ustaleniu napięcia referencyjnego kolejną rzeczą jest wybór wejścia mikrokontrolera, na którym odbywać się będzie pomiar. Dokonuje się tego poprzez nadanie odpowiednich wartości bitom MUX0..4 wspomnianego rejestru ADMUX. Poszczególne mikrokontrolery AVR znacznie różnią się liczbą dostępnych wejść ADC, stąd też nie wszystkie kombinacje bitów MUX są prawidłowe dla każdego mikrokontrolera. Dodatkowo niektóre kombinacje bitów umożliwiają wybór specjalnych funkcji, np. pomiaru napięcia wewnętrznej diody umożliwiającej pomiar temperatury, lub włączenie dodatkowego wzmacniacza sygnału. Są to funkcje specyficzne dla konkretnego typu mikrokontrolera, tak więc przed ustawieniem bitów MUX należy dokładnie przejrzeć notę katalogową procesora.



Wskazówka

Zmiana wartości bitów MUX odnosi skutek dopiero po zakończeniu aktualnie toczącego się pomiaru. Dopiero kolejny pomiar odbędzie się na wejściu wybranym nowymi wartościami bitów MUX.

Właściwość ta nie ma wielkiego znaczenia w przypadku pojedynczej konwersji — w takiej sytuacji najpierw nadajemy pożądaną wartość bitom MUX, a dopiero potem rozpoczynamy pomiar. Stwarza jednak problemy w przypadku trybu ciągłej konwersji — w momencie odczytywania wartości aktualnie skończonej konwersji przetwornik ADC dokonuje już kolejnej. W efekcie zmiana stanu bitów MUX powoduje zmianę odczytywanego kanału dopiero przy kolejnej konwersji.



Wskazówka

Wybranie danego pinu jako wejścia ADC nie powoduje zablokowania przypisanej mu funkcji jako pinu I0.

Taki pin ciągle znajduje się pod kontrolą rejestrów DDRx i PORTx, w związku z czym aby mierzyć napięcie analogowe doprowadzone do danego pinu, należy zadbać, aby był on ustawiony jako wejście bez podciągania. Możliwość sterowania wejściem ADC z poziomu programu ma także pewne zalety, można np. w ten sposób wykrywać niepodłączone wejścia ADC.

## Przetwornik ADC

Po zainicjowaniu rejestru ADMUX możemy przystąpić do procesu konwersji. Aby rozpoczęć proces konwersji, należy włączyć przetwornik ADC poprzez wpisanie 1 do bitu ADEN (ang. *ADC Enable*) znajdującego się w rejestrze kontrolnym ADCSRA (ang. *ADC*

*Control and Status Register A*). Równocześnie z włączeniem przetwornika można ustawić tzw. preskaler. Jest to układ generujący sygnał taktujący przetwornik ADC. Preskaler dzieli zegar taktujący procesor przez wybraną wartość, co umożliwia użycie zegara taktującego ADC. W tym celu należy odpowiednio skonfigurować bity ADPS0 – ADPS2 rejestru ADCSRA (tabela 15.2).

**Tabela 15.2.** Konfiguracja preskalera ADC

<b>ADPS2</b>	<b>ADPS1</b>	<b>ADPS0</b>	<b>Preskaler</b>
0	0	0	2
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

Dla uzyskania maksymalnej dokładności ADC częstotliwość jego zegara taktującego nie powinna przekraczać 200 kHz. Jeśli możemy się zadowolić mniejszą dokładnością przetwarzania, to możemy uzyskać większą prędkość działania przetwornika, taktując go szybszym sygnałem zegarowym, nawet do 1 MHz.

Po włączeniu ADC i ustawieniu preskalera możemy rozpoczęć pomiar.



W zależności od wybranego preskalera pomiar rozpoczyna się po 2 – 128 cyklach zegara taktującego procesor.

**W przypadku użycia zewnętrznego sygnału wyzwalającego rozpoczętą konwersję opóźnienie jest stałe i niezależne od wybranego preskalera.**

## Tryb pojedynczej konwersji

W tym trybie dokonywany jest pojedynczy pomiar. Włączamy go poprzez ustawienie bitu ADSC (ang. *ADC Start Conversion*) w rejestrze ADCSRA. Pomiar trwa 13 cykli zegara taktującego ADC, lecz jeśli jednocześnie z ustawieniem bitu ADSC został włączony przetwornik (ustawiony bit ADEN), pomiar wydłuża się do 25 cykli. Po zakończeniu konwersji bit ten jest automatycznie zerowany, co może być wykorzystane do detekcji końca konwersji.

Aby rozpocząć konwersję kolejnej próbki, należy ponownie wpisać 1 do bitu ADSC. Wyzerowanie bitu ADSC w trakcie konwersji powoduje jej natychmiastowe przerwanie.

## Tryb ciągłej konwersji

Aby uzyskać maksymalną prędkość przetwarzania, należy wykorzystać tryb ciągłej konwersji. W tym trybie proces kolejnej konwersji rozpoczyna się automatycznie tuż po zakończeniu poprzedniego przetwarzania. Aby skorzystać z tego trybu, należy ustawić bit ADFR (ang. *ADC Free Running Select*) w rejestrze ADCSRA. Po ustawieniu tego bitu pierwszą konwersję należy zainicjować poprzez wpisanie 1 do bitu ADSC. Kolejne konwersje zostaną rozpoczęte automatycznie. Proces ten można zakończyć, wpisując 0 do bitu ADFR.



Wskazówka

W tym trybie, jeśli nie nadążymy z odczytem rejestru ADC przed zakończeniem procesu kolejnej konwersji, to zostanie ona nadpisana przez nowy wynik.

W tym trybie zwykle nie wykorzystuje się możliwości przełączania pomiędzy pomiarami wykorzystywanego kanału ADC (bity MUX). Jak już wspomniano, zmiana stanu bitów MUX zaczyna obowiązywać dopiero przy kolejnym pomiarze, gdyż w trybie ciągłej konwersji ich zmiana zawsze zachodzi w sytuacji, kiedy przetwornik zajęty jest przetwarzaniem sygnału. Powoduje to istotną komplikację programu — **uzyskane wyniki ADC są „spóźnione” o jeden kanał**. Oczywiście, można to rozwiązać, lecz w przypadku konieczności zmiany kanałów pomiędzy pomiarami prościej jest zastosować tryb pojedynczej konwersji.

Po zakończeniu konwersji wynik możemy odczytać z rejestru ADC. Składa się on z dwóch 8-bitowych rejestrów: ADCL i ADCH. Jeśli interesuje nas wyłącznie 8 najstarszych bitów wyniku, to możemy ustawić bit ADLAR rejestru ADMUX. Dzięki temu wynik jest wyrównany do lewej, a nie do prawej, w efekcie 8 najstarszych bitów wyniku znajduje się w rejestrze ADCH, natomiast dwa najmłodsze bity znajdują się na pozycjach bitów 7 i 6 rejestru ADCL.



Wskazówka

Aby możliwe było wykonanie kolejnej konwersji, należy odczytać stan rejestru ADCH.

## Wejścia pojedyncze i różnicowe

Przetwornik ADC może pracować w dwóch podstawowych konfiguracjach (rysunek 15.1):

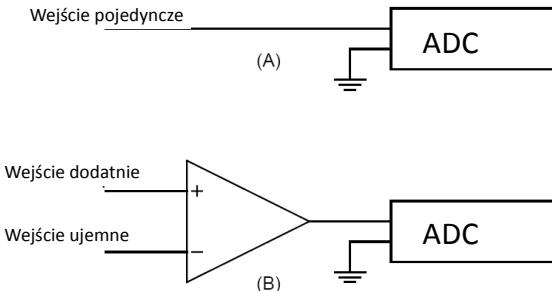
- ♦ Mierzoną jest różnica napięć pomiędzy wejściem ADC a masą analogową układu.
- ♦ Mierzoną jest różnica napięć pomiędzy dwoma różnymi wejściami ADC.

W pierwszym przypadku mierzona napięcie zawsze jest większe od potencjału masy lub mu równe, tak więc w wyniku uzyskujemy 10-bitową wartość bez znaku. W przypadku pomiarów różnicowych zmierzona napięcie może być większe lub mniejsze od napięcia panującego na drugim wejściu ADC, stąd też uzyskany wynik może być liczbą ujemną. W przypadku pomiarów różnicowych wynik jest 9-bitową liczbą ze znakiem zapisaną w kodzie uzupełnień do dwóch ( $U_2$ ). Końcówkę pomiarową, wobec

**Rysunek 15.1.**

*Podstawowe konfiguracje sprzętowe przetwornika ADC.*

- A) Wejście pojedyncze — pomiar odbywa się względem masy.
- B) wejście różnicowe — pomiar odbywa się pomiędzy wejściem dodatnim a ujemnym



której mierzone jest napięcie, wybiera się za pomocą odpowiedniej konfiguracji bitów wpisywanej do rejestru ADMUX. Ponieważ konfiguracja ta jest zależna od konkretnego procesora, należy przejrzeć jego notę katalogową, sekcję ADC *Multiplexer Selection Register — ADMUX*.



W przypadku pomiarów różnicowych oba wejścia ADC, na których odbywa się pomiar, muszą mieć potencjał w zakresie  $V_{cc} \geq V_{in} \geq GND$ .

## Wynik

Wynik konwersji można odczytać z rejestru ADC, przy czym można odczytywać cały 16-bitowy rejestr (tak jest zalecane) lub też osobno jego 8-bitowe części ADCL i ADCH. Należy pamiętać, że kolejna konwersja odbywa się dopiero po odczytaniu rejestru ADCH.

Uzyskaną wartość można przeliczyć na napięcie ze wzoru  $V_{in} = (ADC \cdot V_{ref}) / 1024$ . W przypadku pomiarów różnicowych napięcie różnicowe wynosi  $V_d = (ADC \cdot V_{ref}) / (512 \cdot wzmocnienie)$ , gdzie *wzmocnienie* to wartość wybranego wzmocnienia.

## Wyzwalacze

W nowszych procesorach AVR wejście w tryb ciągłej konwersji ADC przebiega nieco inaczej. W tym przypadku w rejestrze ADCSRA nie występuje bit ADFR, a jego miejsce zajmuje bit ADATE (ang. *ADC Auto Trigger Enable*). W przypadku jego ustawienia rozpoczęcie konwersji następuje na narastającym zboczu sygnału układu wyzwalającego. Wyzwalacz jest wybierany przy pomocy bitów ADTS0 – 2 rejestru ADCSRB — tabela 15.3.



Rejestr ADCSRB zawiera także bit ACME, który dla poprawnej pracy przetwornika ADC powinien mieć wartość 0.

Znaczenie bitu ACME zostanie opisane w rozdziale poświęconym komparatorowi analogowemu.

**Tabela 15.3.** Wybór źródła układu wyzwalającego ADC dla procesora ATMega88

<b>ADTS2</b>	<b>ADTS1</b>	<b>ADTS0</b>	<b>Wyzwalacz</b>
0	0	0	Tryb ciągłej konwersji
0	0	1	Wyjście komparatora
0	1	0	Zewnętrzna linia żądania przerwania 0
0	1	1	Zdarzenie Compare Match timera 0
1	0	0	Przepelnienie timera 0
1	0	1	Zdarzenie Compare Match B timera 1
1	1	0	Przepelnienie timera 1
1	1	1	Zdarzenie Capture Event timera 1

Dzięki możliwości wyboru wyzwalacza dla ADC można dokonywać konwersji w ścisłe określonych odcinkach czasowych lub w chwili wystąpienia jakiegoś zdarzenia zewnętrznego.

Domyślne wartości bitów ADTS0 – ADTS2 wynoszą 0, co umożliwia pracę w trybie zgodności ze starszymi mikrokontrolerami AVR — w takiej sytuacji ustawienie bitu ADATE bez zmian stanu bitów ADTS0 – 2 uruchamia tryb wielokrotnej konwersji.

## Blokowanie wejść cyfrowych

W sytuacji kiedy dane wejście jest wykorzystywane wyłącznie jako wejście analogowe, w niektórych procesorach AVR (nowsze procesory serii ATTiny i ATMega) istnieje możliwość wyłączenia związanego z nim bufora *IO*, co redukuje zapotrzebowanie na energię. W tym celu należy ustawić odpowiadający danemu wejściu bit rejestru **DIDR0** (ang. *Digital Input Disable Register*). Po zablokowaniu funkcji portu cyfrowego związanego z danym pinem odczytywanie związanego z nim bitu rejestru **PINx** da w wyniku wartość 0.

## Przerwania ADC

Jak wspomniano, po zakończeniu procesu konwersji układ ADC może wygenerować przerwanie o wektorze **ADC\_vect**. Dzięki temu program nie musi bezczynnie czekać, aż przetwornik będzie gotowy. Aby przetwornik po zakończonej konwersji mógł generować przerwanie, należy ustawić bit **ADIE** (ang. *ADC Interrupt Enable*) w rejestrze **ADCSRA**. Wykorzystanie przerwań ADC ma szczególne znaczenie w trybie ciągłej konwersji. W tym trybie w chwili zgłoszenia przerwania od razu rozpoczyna się kolejny proces konwersji. Dzięki temu przetwornik może pracować z pełną prędkością, a procedura obsługi przerwania może zapisywać kolejne wyniki w buforze, skąd są pobierane do analizy przez program główny. Taka obsługa ADC powoduje znacznie bardziej optymalne wykorzystanie zasobów procesora — procesor, zamiast czekać na wynik konwersji, może

zająć się innymi zadaniami. Podobnie, w trybie pojedynczej konwersji w procedurze obsługi przerwania można np. zmienić kanał i rozpocząć proces kolejnej konwersji (przez ustawienie bitu ADSC).

## Precyzyjne pomiary przy pomocy ADC

Układy cyfrowe nie są idealnym środowiskiem dla precyzyjnych pomiarów wartości analogowych. Praca procesora generuje zakłócenia, które wpływają na dokładność pomiarów dokonanych przy pomocy ADC. Zakłócenia te możemy zwalczać różnymi sposobami:

- ◆ Sygnały analogowe muszą być doprowadzone możliwie krótkimi ścieżkami do przetwornika, należy je prowadzić nad piaszczystymi masy.
- ◆ Pin AVcc powinien być połączony do zasilania przy pomocy filtra LC.
- ◆ Niewykorzystane piny ADC i portu, przez który są wyprowadzone, nie powinny być przełączane w trakcie dokonywania pomiaru.

Jednak mimo stosowania powyższych technik sam procesor jest sporym źródłem zakłóceń. Projektanci procesorów AVR dostrzegli ten problem, wprowadzając specjalny tryb pracy procesora, związany z redukcją szumów (ang. *ADC Noise Reduction Mode*). Aby wykorzystać ten tryb, należy:

- ◆ Skonfigurować ADC tak jak do zwykłego pomiaru (wybrać odpowiednie wartości rejestrów ADMUX, ADCSRA).
- ◆ Wybrać tryb pojedynczej konwersji, włączyć układ ADC, lecz nie rozpoczęta konwersji ADC.
- ◆ Odblokować przerwania ADC oraz globalną flagę przerwań.
- ◆ Skonfigurować bity SM rejestru MCUCR tak, aby wskazywały na tryb *ADC Noise Reduction*.
- ◆ Wykonać instrukcję `sleep()`.

W wyniku wykonania sekwencji powyższych instrukcji procesor wejdzie w tryb uśpienia i jednocześnie rozpocznie proces konwersji ADC. Po jego zakończeniu wygenerowane zostanie przerwanie ADC, co wybudzi procesor z uśpienia.



Definicje związane z zarządzaniem poborem mocy procesora znajdują się w pliku nagłówkowym `<avr\sleep.h>`

Poniższy kod pokazuje, jak korzystać z trybu redukcji szumów:

```
ISR(ADC_vect)
{
}

void initADC()
```

```
{  
    ADMUX=_BV(REFS1) | _BV(REFS0);      //Wewnętrzne nap. referencyjne, kanał 0,  
                                         //wyrównanie do prawej  
    ADCSRA=_BV(ADEN) | _BV(ADIE) | _BV(ADPS2) | _BV(ADPS1) | _BV(ADPS0);  
                                         //Włącz ADC, przerwania, preskaler 128  
}  
  
int GetADC()  
{  
    set_sleep_mode(SLEEP_MODE_ADC);      //Tryb noise canceller  
    cli();  
    sleep_enable();                    //Odblokuj możliwość wejścia w tryb sleep  
    sei();  
    sleep_cpu();                      //Wejdź w tryb uśpienia  
    sleep_disable();                  //Zablokuj możliwość wejścia w tryb sleep  
    return ADC;  
}
```

W powyższym przykładzie zdefiniowano pustą procedurę obsługi przerwania ADC. Jest to niezbędne, gdyż w celu umożliwienia wybudzenia procesora z trybu uśpienia trzeba odblokować przerwanie ADC. W przypadku ich odblokowania, bez stosowanej procedury obsługi, wywołana zostałaby procedura domyślna, która zresetowałaby procesor. Funkcja `initADC()` inicjalizuje przetwornik ADC, ustawiając wszystkie wymagane parametry konwersji. Jednak kluczową rolę odgrywa funkcja `GetADC()`, zwracająca wynik pomiaru. W funkcji tej ustalany jest tryb uśpienia (`SLEEP_MODE_ADC`), następnie procesor jest usypani, co jednocześnie aktywuje proces konwersji.



W tym trybie ustawienie bitu ADSC rejestru ADCSRA odbywa się automatycznie. Procesor nie jest w stanie go ustawić, gdyż wcześniej wszedł w tryb uśpienia.

Po zakończeniu konwersji generowane jest przerwanie, które wybudza procesor i podejmowana jest normalna praca programu. Oczywiście, wynik konwersji można odczytywać w procedurze obsługi przerwania.

## Nadpróbkowanie

W pewnych sytuacjach 10-bitowa rozdzielcość przetwornika ADC może okazać się niewystarczająca. Możemy sięgnąć po przetworniki zewnętrzne lub programowo zwiększyć rozdzielcość wbudowanego przetwornika. Technika ta nazywa się nadpróbkowaniem (ang. *Oversampling*). **Z równania Nyquista wynika, że aby móc oddać wierne kształty sygnału, częstotliwość próbkowania musi być co najmniej dwukrotnie większa niż częstotliwość próbkowanego sygnału.** Jednak dany sygnał można próbować częściej, niż jest to wymagane przez równanie Nyquista, a uzyskane dodatkowe próbki można uśredniać, co w efekcie zwiększy rozdzielcość próbkowania. Aby w ten sposób, aby zwiększyć rozdzielcość przetwornika o 1 bit, należy zwiększyć częstotliwość próbkowania 4-krotnie. Jak widać, zwiększając w ten sposób rozdzielcość ADC, zmniejsza się maksymalną częstotliwość próbkowanego sygnału. Jednak aby móc korzystać z techniki nadpróbkowania, muszą być spełnione dodatkowe warunki. Po pierwsze,

w okresie zbierania próbek, które następnie zostaną uśrednione, mierzony sygnał nie może się zmieniać. Wszelkie zmiany sygnału zostaną uśrednione i nie będzie możliwości ich wykrycia. Z drugiej strony, sygnał nie może być stały. W mierzonym sygnale musi występować pewien szum. Ze względu na błąd kwantyzacji przetwornika ADC, wynoszący ok. 0,5 LSB, szum powinien mieć nieco większą wartość, optymalnie 1 – 2 LSB. Zazwyczaj szum wnoszony przez inne komponenty znajdujące się blisko mierzonego sygnału i sam procesor jest wystarczający.



Metoda nadpróbkowania oprócz zwiększenia rozdzielczości ADC polepsza także stosunek sygnału do szumu, czyniąc pomiar bardziej stabilnym.

## Uśrednianie

Uśrednianie polega na sumowaniu  $n$  próbek i podzieleniu uzyskanej sumy przez  $n$ . Czasami do uśredniania wykorzystuje się tzw. średnią ruchomą. Jest ona niezwykle prosta w realizacji. Programowo tworzony jest bufor cykliczny, do którego z jednej strony „wsuwane są” kolejne pomiary, a najstarsze są z buforu usuwane. Wynik pomiaru ADC to średnia wartości aktualnie przebywających w buforze wyników pomiarów. **Uśrednianie nie prowadzi do zwiększenia rozdzielczości ADC, ale poprawia odstęp sygnału do szumu.**

## Decymacja i interpolacja

W przeciwnieństwie do zwykłego uśredniania celem decymacji i interpolacji jest zwiększenie rozdzielczości, z jaką próbkowany jest sygnał. Wiemy już, że aby uzyskać jeden dodatkowy bit informacji, potrzebujemy 4 próbki sygnału. W efekcie, aby uzyskać zwiększenie rozdzielczości z 10 do 12 bitów, potrzebujemy aż 16 próbek (4·4). Powoduje to 16-krotne zmniejszenie pasma próbkowanego sygnału. **Jeśli tak uzyskane pasmo zbliża się do pasma sygnału wejściowego, należy zastosować filtr dolnoprzepustowy. W przeciwnym przypadku narażamy się na niebezpieczeństwo powstawania aliasingu.**

Aby uzyskać 12-bitową rozdzielczość, należy wartości uzyskanych 16 próbek zsumować, a następnie podzielić przez 4. W wyniku operacji sumowania 16 10-bitowych wartości uzyskujemy 14-bitowy wynik, z którego dwa najmłodsze bity nie zawierają użytecznej informacji.

## Przykłady

Poniżej przedstawione zostały proste przykłady wykorzystania przetwornika ADC. W pierwszym wykorzystana została technika nadpróbkowania i uśredniania w celu zwiększenia rozdzielczości i stabilności pomiaru temperatury, drugi przykład wykorzystuje przetwornik ADC do dosyć nietypowego celu — podłączenia klawiszy.

## Termometr analogowy LM35

Układ LM35 jest termometrem z wyjściem analogowym. Różnica 1°C odpowiada różnicy 10 mV na wyjściu układu. Układ LM35 jest tani i wygodny w użyciu, a dzięki wyjściu analogowemu jego podłączenie do mikrokontrolera jest niezwykle proste i w przeciwieństwie do cyfrowych mierników temperatury nie wymaga pisania skomplikowanych protokołów obsługi. Układ ten wystarczy podłączyć do wybranego wejścia ADC, a następnie rozpocząć proces konwersji. W poniższym przykładzie wykorzystano technikę nadpróbkowania i uśredniania w celu zwiększenia rozdzielczości pomiaru.



Techniki te zwiększały rozdzielczość pomiaru, w żaden sposób nie zwiększały jego dokładności.

Dokładność pomiaru w zależności od wersji układu jest lepsza niż 1,5°C w całym zakresie temperatur pracy, a typowo jest lepsza niż 0,4°C. Na ten błąd nakładają się błędy samego przetwornika ADC oraz źródła referencyjnego. Wewnętrzne źródło referencyjne w przypadku procesora ATMega88 ma napięcie w zakresie 1,0 – 1,2 V, typowo 1,1 V. W przypadku rozdzielczości pomiaru napotykamy na barierę wynikającą z właściwości samego przetwornika ADC w mikrokontrolerach AVR. Jest to przetwornik 10-bitowy, lecz błędy przetwarzania przekraczają 4 najmniej znaczące cyfry. W efekcie, pomimo że teoretycznie dla napięcia referencyjnego 1,1 V powinno dać się osiągnąć rozdzielczość ok. 1 mV, czyli 0,1°C, to w praktyce trudno jest osiągnąć pomiar lepszy niż ok.  $\pm 3$  mV. Aby ominąć tę niedogodność, w poniższym przykładzie wykorzystano nadpróbkowanie — na jeden odczyt przypada 128 pomiarów napięcia, co powinno zapewnić co najmniej dodatkowe 3 bity rozdzielczości. Pomiar dokonywany jest przy pomocy ADC pracującego w trybie autowyzwalania:

```
void ADC_init()
{
    ADMUX=_BV(REFS0) | _BV(REFS1) | 0b0101; //Wew. referencyjne, kanał 5,
                                                //wyrównanie do prawej
    ADCSRA=_BV(ADEN) | _BV(ADIE) | _BV(ADPS2) | _BV(ADPS1) | _BV(ADPS0); //Włącz ADC,
                                                                           //przerwania, preskaler 128
    ADCSRA|=_BV(ADATE) | _BV(ADSC));
}
```

Wykorzystano preskaler 128, wewnętrzne źródło referencyjne o napięciu 1,1 V, pomiar odbywa się na kanale ADC5. W przypadku procesora ATMega88 ustawienie bitu ADATE odpowiada trybowi ciągłej konwersji (chyba że zmieni się domyślne ustawienia bitów ADTS0 – 2 rejestru ADCSRB). W tym trybie pierwszą konwersję należy wyzwolić ręcznie poprzez ustawienie bitu ADSC, kolejne konwersje wyzwalane są automatycznie po zakończeniu poprzedniej.

Zakończenie procesu konwersji wyzwala przerwanie ADC, którego obsługa odpowiedzialna jest za integrację danych:

```
#define NOOFSAMPLES 128
```

```
volatile uint32_t ADCVal;
```

```

ISR(ADC_vect)
{
    static uint32_t ADCaccum;
    static uint8_t samplenzo;

    ADCaccum+=ADC;
    samplenzo++;
    if(samplenzo==NOOFSAMPLES)
    {
        ADCVal=ADCaccum;
        ADCaccum=0;
        samplenzo=0;
    }
}

```

Symbol NOOFSAMPLES określa, z ilu próbek będzie składał się jeden pomiar. W powyższym przykładzie będzie to 128. Odczytana wartość rejestru ADC dodawana jest do zmiennej ADCaccum, przechowującej zsumowane próbki. Po zebraniu określonej przez NOOFSAMPLES liczby próbek, uaktualniana jest wartość zmiennej ADCVal. Zmienna ta zawiera odczytaną wartość temperatury. Wartość tę jednak należy przeliczyć, uwzględniając wartość napięcia źródła referencyjnego oraz liczbę próbek przypadających na jeden pomiar. Zajmuje się tym kolejna funkcja:

```

uint16_t GetTemperature()
{
    uint32_t adc;
    ATOMIC_BLOCK(ATOMIC_FORCEON)
    {
        adc=ADCVal;
    }

    return (adc*11000UL)/(1024UL*NOOFSAMPLES);
}

```

Funkcji tej warto przyjrzeć się bliżej. Najpierw wartość zmiennej ADCVal przypisywana jest lokalnej zmiennej pomocniczej adc. Przypisanie to musi wystąpić atomowo (stąd zamknięte jest w bloku ATOMIC\_BLOCK), gdyż przerwanie ADC, które wystąpiłoby pomiędzy przesaniem wartości zmiennej ADCVal do adc (przesłanie to składa się z kilku instrukcji asemblera), mogłoby zmienić wartość zmiennej ADCVal, w efekcie adc zwierałoby częściowo starą, częściowo nową wartość zmiennej ADCVal. Zmienna adc następnie jest przeliczana tak, aby funkcja mogła zwrócić temperaturę z rozdzielcością do 0,01°C. Aby zwrócić temperaturę z taką rozdzielcością bez użycia arytmetyki zmiennopozycyjnej, zastosowano sztuczkę opisaną w rozdziale 3., „Podstawy języka C na AVR”, w podrozdziale „Arytmetyka stałopozycyjna”. Stała 11000UL to napięcie referencyjne wyrażone w mV i pomnożone razy 10. Na 0,1°C przypada 1 mV, dzięki mnożeniu wartości liczne są z dokładnością do 0,1 mV, a więc 0,01°C. Sufiks UL jest niezbędny, aby poinformować kompilator, że wszystkie działania mają być przeprowadzone na typie unsigned long, a nie jak domyślnie unsigned int. Następnie uzyskana wartość jest dzielona przez liczbę próbek składających się na pomiar oraz 1024 — maksymalną wartość rejestru ADC. W efekcie zwracana jest wartość będąca wielokrotnością jednostki podstawowej, czyli 0,1 mV. Pozostaje jeszcze przedstawić wynik. W tym celu wykorzystano bibliotekę do obsługi graficznego LCD, szerzej omówioną w rozdziale 18. A oto, jak wygląda prezentacja wyników:

```

int main()
{
    char wynik[7];

    GLCD_init();
    ADC_init();
    GLCD_cls();
    GLCD_goto(0,0);

    GLCD_puttext_P(PSTR("Temperatura:"));

    while(1)
    {
        sprintf(wynik, "%5d", GetTemperature());
        uint8_t len=strlen(wynik);
        memmove(&wynik[len-1], &wynik[len-2], 3);
        wynik[len-2]='.';
        GLCD_goto(0, 10);
        GLCD_puttext(wynik);
        delay_ms(500);
    }
}

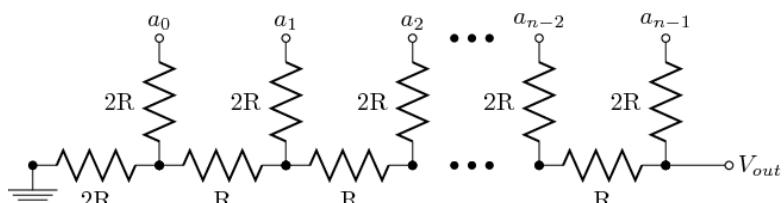
```

Ponieważ wynik zwracany przez funkcję `GetTemperature()` wyrażony jest w setnych częściach stopnia, możemy wyświetlać całą cyfrę, ale przyjemniej byłoby umieścić przecinek oddzielający setne części stopnia od całości. W tym celu wykorzystano funkcje biblioteczne z pliku `string.h`. Znajdują się tam prototypy dwóch funkcji — `strlen`, zwracającej długość łańcucha znakowego, oraz `memmove`, kopiącej fragmenty łańcucha znakowego. Funkcja `memmove` umożliwia prawidłowe kopowanie obszarów pamięci nawet w sytuacji, w której obszary te częściowo na siebie zachodzą. W powyższym przykładzie jej zadaniem jest skopiowanie dwóch ostatnich cyfr wyniku (oraz znaku kończącego łańcucha `NULL`), aby zrobić miejsce na umieszczenie przecinka dziesiętnego. Dzięki temu temperatura zawsze wyświetlana jest w formacie *ddd.dd*.

## Klawisze

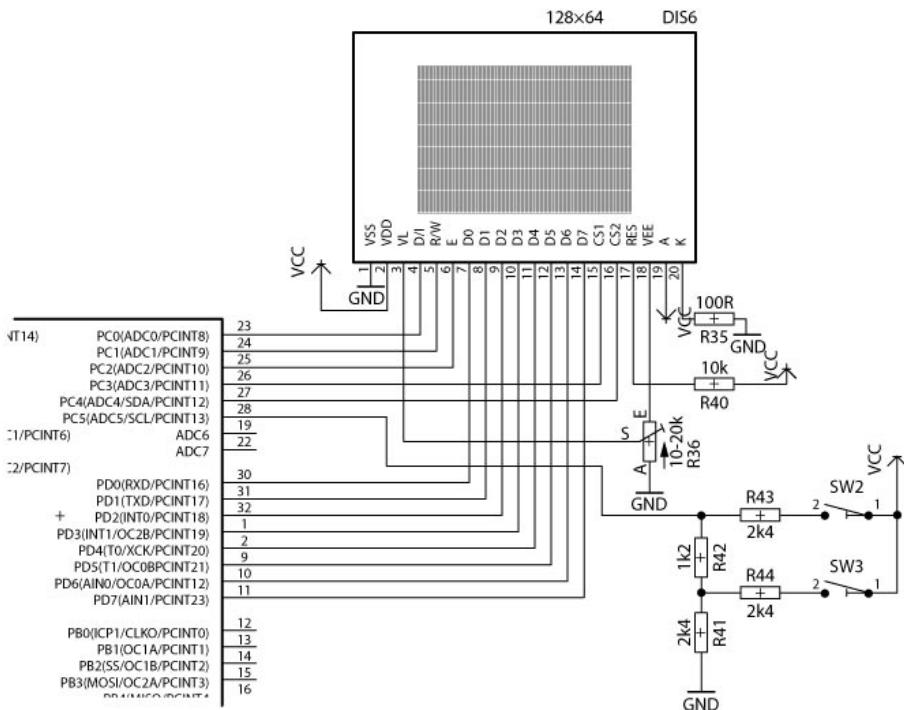
Dosyć nietypowym zastosowaniem przetwornika ADC jest... podpięcie klawiatury lub przycisków. Ma to pewną zaletę — wykorzystując jeden pin procesora, można podłączyć od kilku do nawet kilkunastu przycisków. Jest to możliwe dzięki wykorzystaniu zasady odwrotnej do zasady działania przetwornika DAC (ang. *Digital to Analog Converter*) zbudowanego przy pomocy drabinki rezystorowej R-2R (rysunek 15.2).

**Rysunek 15.2.**  
Budowa drabinki rezystorowej R-2R



Drabinka taka umożliwia konwersję sygnału cyfrowego składającego się z  $n$  bitów na napięcie. Normalnie stanem poszczególnych bitów steruje mikrokontroler poprzez

wybrany port *I0*. Jednak zamiast portu mikrokontrolera do poszczególnych linii *a0 – a7* można podłączyć przyciski, które zwierane będą zmieniały napięcie wyjściowe (*Vout*) takiej drabinki. Z kolei wartość tego napięcia można w prosty sposób zmierzyć przy pomocy przetwornika ADC. Różnica pomiędzy podłączeniem portu *I0* do konwertera DAC a podłączeniem przycisku polega na tym, że port wystawia stan wysoki i niski, natomiast przycisk na dane wejście konwertera jest w stanie podać tylko jeden ze stanów. Drugi stan jest stanem rozwarcia. Nie stanowi to jednak istotnej przeszkody, wpływa tylko nieznacznie na obliczoną wartość napięcia. Do realizacji takiej drabinki można wykorzystać dowolne rezystory. W przedstawionym przykładzie jako rezystory 2R wykorzystano rezystory o wartości 2,4 kΩ, a jako rezystor R dwa równolegle połączone rezystory 2,4 kΩ, dające w efekcie rezystor 1,2 kΩ. Wartość użytych rezystorów nie ma większego znaczenia, należy jednak pamiętać, aby ich tolerancja była możliwie mała, najlepiej poniżej 5%. Zmniejszy to zmienność napięć uzyskanych na wyjściu takiej drabinki w stosunku do wartości teoretycznie wyliczonych. Schemat układu został pokazany na rysunku 15.3.



Rysunek 15.3. Schemat podłączenia przycisków do portu ADC

W przedstawionym układzie do wejścia ADC podłączono tylko dwa przyciski, nic nie stoi jednak na przeszkodzie, aby wykorzystać ich więcej. Włączenie danego przycisku powoduje podanie napięcia *Vcc* na odpowiedni bit drabinki R-2R. Z kolei jego rozwarcie powoduje, że dany bit jest w stanie wysokiej impedancji i nie ma wpływu na uzyskane napięcie wyjściowe.



Wskazówka

Sama wartość napięcia zasilającego drabinkę jest bez znaczenia, ważne tylko, aby napięcie zasilające drabinkę było jednocześnie napięciem referencyjnym dla przetwornika ADC.

Przy wszystkich przyciskach rozwartych na wyjściu drabinki panuje potencjał 0 V, co przetwornik odczytuje jako wartość zbliżoną do 0. Włączenie przycisku SW3 powoduje podanie napięcia  $V_{cc}$  na rezystor R44, który w tym układzie będzie stanowił część dzielnika napięcia wraz z rezystorem R41. W efekcie na wejściu ADC będzie panowało napięcie  $V_{cc}/2$ . Wartość rezystora R42 jest w tym przypadku bez znaczenia, gdyż znikomy prąd wejścia ADC nie powoduje jakiegokolwiek spadku napięcia na tym rezystorze. W sytuacji kiedy zwarty jest SW2, a rozwarty jest SW3, napięcie na wejściu ADC zależy od dzielnika utworzonego przez rezystor R43 i sumę rezystancji R41 i R42. Wartość tego dzielnika wynosi  $R43:(R41+R42+R43)$ , czyli  $2R:5R$ , napięcie wyjściowe osiągnie więc wartość 0,6 Vcc. Jeśli jednocześnie włączone zostaną przyciski SW2 i SW3, napięcie osiągnie wartość 0,75 Vcc. Wiedząc, jakie napięcia panują na wyjściu drabinki w zależności od wciśniętych przycisków, łatwo jest je przeliczyć na uzyskane wartości ADC. **Zakładając, że napięcie referencyjne wynosi  $V_{cc}$ , to rejestr ADC dla wcześniejszych wyliczonych stanów będzie miał odpowiednio wartości: 0,5·1024, 0,6·1024 i 0,75·1024, niezależnie od napięcia zasilającego i wartości użytych rezystorów (pod warunkiem że będą one miały wartości R i 2R).**



Wskazówka

Uzbrojeni w tak poważne wyliczenia możemy przejść do realizacji programu odczytującego stan przycisków. Poniższy program napisany został dla mikrokontrolera ATMega88 i wykorzystuje możliwość wyzwalania przetwornika przez sygnał *Compare Match timera 1*.

Stan klawiszy SW2 i SW3 odczytywany będzie cyklicznie w procedurze obsługi przewrana ADC. Ponieważ klawisze wykazują drgań mechanicznych, należy je wyeliminować. W celu eliminacji drgań pomiar wykonywany będzie dwukrotnie, co 5 ms. Jeśli kolejne dwa pomiary będą się różniły o mniej niż zadana wartość, to funkcja uzna odczyt za stabilny. W przeciwnym przypadku pomiary będą dokonywane do czasu, aż dwa sąsiednie wyniki będą podobne. Aby pomiary wykonywać dokładnie co 5 ms, zostanie wykorzystany *timer 1*. *Timer ten* ma możliwość ciągłego porównywania stanu rejestru licznika *timera 1* (TCNT1) z rejestrzem OCR1B. Sytuacja, w której oba rejesty są sobie równe, sygnalizowana jest poprzez ustawienie bitu OCF1B w rejestrze TIFR1. Kolejne pomiary ADC wykonywane będą co 5 ms, w efekcie dla mikrokontrolera taktowanego sygnałem o częstotliwości 8 MHz preskaler *timera 1* należy ustawić na wartość 64, a rejestr OCR1B na wartość 625 ( $8000000/(64 \cdot 625) = 200$  Hz, a więc pomiędzy kolejnymi ustawieniami flagi OCF1B mija dokładnie 5 ms). *Timer 1* będzie pracował w trybie CTC, dzięki czemu po dojściu do wartości określonej przez rejestr OCR1A będzie zliczał od 0. Procedura inicjująca *timer* wygląda następująco:

```
void TMER_init()
{
    TCCR1B= _BV(CS11) | _BV(CS10) | _BV(WGM12);           //Preskaler 64, tryb CTC
    OCR1A=625;      //Top zliczania w trybie CTC
    OCR1B=625;
}
```

Przetwornik ADC układu ATMega88 potrafi pracować w trybie, w którym rozpoczęcie pomiaru wyzwalane jest określonym zdarzeniem. W poniższym przykładzie zdarzeniem inicjującym pomiar jest ustawienie flagi OCF1B. Procedura inicjalizacji przetwornika ADC jest następująca:

```
void ADC_init()
{
    ADMUX=_BV(REFS0) | 0b0101; //Nap. referencyjne Vcc, kanał 5, wyrównanie do prawej
    ADCSRA=_BV(ADEN) | _BV(ADIE) | _BV(ADPS2) | _BV(ADPS1) | _BV(ADPS0); //Włącz ADC,
                                                                           //przerwania, preskaler 128
    ADCSRB=_BV(ADTS2) | _BV(ADTS0); //Włącz wyzwalanie zdarzeniem timer1 compare match B
    ADCSRA|= _BV(ADATE);
}
```

Jako napięcie referencyjne zostało wybrane napięcie zasilające procesor ( $V_{CC}$ ), przyciski podłączone są do wejśćia  $ADC5$  przetwornika, a układ taktujemy z częstotliwością CLK/128. Dodatkowo wybrano wyzwalanie pomiaru za pomocą sygnału *Compare Match B timera 1*, co zapewnia, że kolejne pomiary będą odbywały się w odstępach 5 ms. Pozostaje stworzyć procedurę obsługi przerwania ADC:

```
#define ADCERROR      10
volatile uint16_t ADCVal;

ISR(ADC_vect)
{
    static uint16_t lastADC;
    uint16_t tmp=ADC;
    TIFR1|= _BV(OCF1B);
    uint16_t low=0;
    if(lastADC>ADCERROR) low=lastADC-ADCERROR;
    if((tmp>(low)) && (tmp<(lastADC+ADCERROR))) ADCVal=tmp;
    lastADC=tmp;
}
```

W procedurze tej odczytywana jest wartość rejestru ADC przetwornika, zawierającego wynik konwersji. Wartość ta porównywana jest z poprzednią wartością przechowywaną w zmiennej `lastADC`. Jeśli wartości te nie różnią się o więcej niż `ADCERROR`, to procedura przyjmuje, że są takie same, i nową wartość wpisuje do zmiennej `ADCVal` zawierającej napięcie wyjściowe z drabinki, a więc stan przycisków. Dzięki tej prostej metodzie realizowana jest eliminacja drgań — drgania przycisków powodują niestabilność napięcia na wejściu przetwornika. Dopóki napięcie to będzie się zmieniać, zmienna `ADCVal` nie będzie uaktualniona, w efekcie program nie zauważa zmiany stanu przycisków. Dojdzie do tego dopiero w sytuacji, kiedy ich stan będzie stabilny. Pewnej uwagi wymaga linia `TIFR1|= _BV(OCF1B)`. Jest ona niezbędna, bez niej kolejne zdarzenie *Compare Match timera* nie wyzwoliłoby pomiaru ADC. Linia ta powoduje wyzerowanie bitu `OCF1B` (bit ten jest zerowany poprzez wpisanie 1). Normalnie zerowanie tego bitu jest wykonywane automatycznie przez procesor po wejściu w procedurę obsługi przerwania wywołanego zdarzeniem *Compare Match B*, lecz powyższy program z przerwania tego nie korzysta, bit ten należy więc wyzerować programowo.

Teraz pozostaje odpowiednio zinterpretować odczytane wartości ADC i przyporządkować im stan klawiszy. W tym celu zdefiniowane zostały symbole zawierające wyliczone wartości ADC dla poszczególnych klawiszy:

```
#define KEY_1    512
#define KEY_2    615
#define KEY_1_2   768
#define NOKEY     0
```

Należy jednak pamiętać, że ani przetwornik ADC, ani zastosowane rezystory nie są idealne, więc wyliczone wartości teoretyczne będą się nieznacznie różniły od wartości spotykanych w rzeczywistym układzie. Stąd też wprowadzono dodatkowy symbol:

```
#define KEY_ERROR      10
```

zawierający tolerancję błędu. Jeśli uzyskany wynik nie będzie odbiegał o więcej niż  $\pm$ KEY\_ERROR od wartości teoretycznej, to wynik taki zostanie uznany za równy wartości teoretycznej.



Im gorszej jakości rezystorami dysponujemy, tym większą wartość należy przyporządkować symbolowi KEY\_ERROR.

### Tolerancja użytych rezystorów i rozdzielcość przetwornika limitują liczbę jednocześnie przyłączonych przycisków do jednego wejścia ADC.

Aby ułatwić sobie porównywanie wartości uzyskanych z ADC z wartościami teoretycznie wyliczonymi, z uwzględnieniem błędu, zdefiniowana została funkcja:

```
bool InRange(uint16_t val, uint16_t range, uint8_t error)
{
    if((val>(range-error)) && (val<(range+error))) return true;
    return false;
}
```

Funkcja ta zwraca true, jeśli wartość zmiennej val mieści się w granicach tolerancji błędu, false w przeciwnym przypadku. Ostatnią funkcją jest funkcja zmieniająca wartość ADC na stan przycisków:

```
uint16_t GetKeyNo()
{
    uint16_t adc;
    ATOMIC_BLOCK(ATOMIC_FORCEON)
    {
        adc=ADCVal;
    }
    if(InRange(adc, KEY_1, KEY_ERROR)) return 1;
    if(InRange(adc, KEY_2, KEY_ERROR)) return 2;
    if(InRange(adc, KEY_1_2, KEY_ERROR)) return 3;

    return NOKEY;
}
```

Funkcja ta zwraca 0 (jeśli żaden przycisk nie został wciśnięty), 1 (jeśli wciśnięty został przycisk SW3), 2 (jeśli przycisk SW2) lub 3 (jeśli oba przyciski są wciśnięte). Powyższą funkcję można rozszerzyć o obsługę kolejnych przycisków.

Uważny czytelnik z pewnością zauważał, że w obu powyższych przykładach wykorzystywane są przerwania, ale nigdzie jawnie nie została odblokowana flaga globalnego zezwolenia na przerwania instrukcją `sei()`. A jak pamiętamy, domyślnie w mikrokontrolerach AVR przerwania są zablokowane. Nie jest to błąd — przerwania zostają odblokowane po wejściu do funkcji odczytu klawiatury (`GetKeyNo()` lub `GetTemperature()`) dzięki parametrowi z jakim zostało wywołane makro `ATOMIC_BLOCK` — `ATOMIC_FORCEON`. Parametr `ATOMIC_FORCEON` powoduje odblokowanie przerwań po wyjściu z takiego bloku. W powyższych przykładach ma to dwie zalety: po pierwszym odczycie przerwania są odblokowywane, drugą zaletą jest oszczędność pamięci — nie musimy jawnie w kodzie umieszczać instrukcji `sei()`, w samym makrzu nie zachodzi też konieczność przechowywania stanu flagi globalnego zezwolenia na przerwanie i następnie jej odtworzenia.

## Rozdział 16.

# Komparator analogowy

Mikrokontrolery AVR dysponują przynajmniej jednym komparatorem analogowym. Stwarza to możliwości porównywania ze sobą dwóch napięć i w zależności od wyniku podejmowania pewnych działań. Komparator analogowy ma wyrowadzone dwa wejścia: dodatnie (*AIN0*) i ujemne (*AIN1*). Jeśli różnica pomiędzy *AIN0* a *AIN1* jest większa od 0, na wyjściu komparatora panuje stan wysoki, jeśli jest mniejsza od zera — stan niski. Stan wyjścia komparatora można odczytać z bitu ACO (ang. *Analog Comparator Output*) rejestru ACSR (ang. *Analog Comparator Control and Status Register*). Dodatkowo zmiana stanu tego bitu może generować przerwanie o wektorze ANALOG\_→COMP\_vect lub ANA\_COMP\_vect.



Układ komparatora domyślnie jest włączony. Jeśli go nie wykorzystujemy, należy go wyłączyć, co zmniejsza zapotrzebowanie na energię.



Komparator można wyłączyć, ustawiając bit ACD (ang. *Analog Comparator Disable*) rejestru ACSR.

Włączenie komparatora nie powoduje zablokowania pinów IO związkanych z jego wejściami. Wszystkie funkcje pinów IO są nadal zachowane, co daje możliwość programowego sterowania wejściami komparatora.

Odblokowanie przerwań komparatora następuje po ustawieniu bitu ACIE (ang. *Analog Comparator Interrupt Enable*) rejestru ACSR. Równocześnie należy przy pomocy bitów ACISO i ACIS1 (ang. *Analog Comparator Interrupt Mode Select*) określić, jakie zdarzenia mają wyzwalać przerwanie — tabela 16.1.

**Tabela 16.1.** Typy zdarzeń wyzwalających przerwanie komparatora i odpowiadająca im konfiguracja bitów ACIS

ACIS1	ACISO	Zdarzenie
0	0	Przerwanie przy każdej zmianie stanu wyjścia komparatora
0	1	Zarezerwowane
1	0	Przerwanie przy zmianie stanu wyjścia z 1 na 0
1	1	Przerwanie przy zmianie stanu wyjścia z 0 na 1

## Funkcje dodatkowe

W nowszych procesorach AVR z komparatorem związane są dodatkowe funkcje i możliwości, znacznie poszerzające obszar jego zastosowań.

### Blokowanie pinów

Podobnie jak przy pinach wykorzystywanych jako wejścia analogowe przetwornika ADC także w przypadku komparatora istnieje możliwość blokowania pinów portów I/O związanych z wejściem analogowym komparatora (*AIN0* i *AIN1*). Umożliwia to zmniejszenie poboru energii. Podobnie jak w przypadku ADC funkcję tę realizuje się, ustawiając odpowiednie bity rejestru *DIDRx*.

### Wyzwalanie zdarzeń timera

Wyjście komparatora analogowego może posłużyć jako źródło sygnału dla funkcji zamrażającej stan licznika *timera* poprzez przeniesienie jego stanu do rejestrów ICR. Dzięki temu istnieje możliwość precyzyjnej lokalizacji w czasie zmiany stanu komparatora. Można to wykorzystać do określenia punktu czasowego, kiedy zadany sygnał analogowy przekroczył określona wartość. Aby skorzystać z tej funkcji, należy ustawić bit ACIC (ang. *Analog Comparator Input Capture Enable*) rejestru ACSR oraz odpowiednio skonfigurować *timer*. Więcej na temat tej funkcji dowieš się w rozdziale poświęconym *timerom*.

### Wybór wejścia komparatora

Niektóre mikrokontrolery AVR oferują także możliwość wykorzystania multipleksera ADC do wyboru pinu, który posłuży jako wejście ujemne komparatora. Aby aktywować tę funkcję, należy ustawić bit ACME (ang. *Analog Comparator Multiplexer Enable*) w rejestrze ADCSRB, przy jednoczesnym wyłączeniu układu ADC (bit ADEN rejestru ADCSRA wyzerowany). Po tej operacji bity MUX rejestru ADMUX określają, z którego pinu doprowadzany będzie sygnał na ujemne wejście komparatora. Może być to dowolny pin układu ADC. Dzięki tej funkcji można porównywać kilka sygnałów analogowych. W przy-

padku wejścia dodatniego komparatora możliwości konfiguracji są bardziej ubogie. Przy pomocy bitu ACBG (ang. *Analog Comparator Bandgap Select*) rejestru ACSR można wybierać pomiędzy wejściem *AIN0* lub wewnętrznym napięciem odniesienia (bit ACBG ustawiony).

## Wyzwalanie przetwornika ADC

Wyjście komparatora może także zostać wykorzystane do wyzwolenia procesu konwersji przetwornika ADC. W tym celu należy ustawić bit ADTS0 rejestru ADCSRB oraz bit ADATE rejestru ADCSRA. Proces konwersji zostanie wyzwolony dodatnim zboczem sygnału na wyjściu komparatora — nie ma możliwości programowej zmiany tego sposobu wyzwalania. Konfiguracja taka umożliwia wyzwalanie pmiaru ADC przy pomocy sygnału analogowego. W starszych procesorach ten sam efekt można uzyskać poprzez wyzwanie pmiaru ADC programowo w procedurze obsługi przerwania komparatora. W takiej jednak sytuacji opóźnienie pomiędzy wyzwolonym pmiarem a zmianą stanu komparatora jest jednak nieco większe (wynosi co najmniej 5 taktów zegara).



# Rozdział 17.

# Timery

Liczniki (*timery*) są elementem wchodzącym w skład każdego procesora z rodziny AVR. Procesory te zwykle wyposażone są w 1 – 5 timerów, chociaż istnieją wersje posiadające ich o wiele więcej. Trudno jest sobie wyobrazić większy program, który z nich nie korzysta. W najprostszej postaci *timer* jest licznikiem, który potrafi zliczać od 0 do pewnej maksymalnej wartości, zależnej od długości licznika. Dla liczników 8-bitowych zlicza od 0 do 255, dla liczników 16-bitowych od 0 do 65 535. Tempo zliczania można określić, wybierając źródło sygnału taktującego licznik. Po osiągnięciu przez licznik wartości maksymalnej zliczanie jest kontynuowane od zera (licznik „przekręca się”). Co ważne, zliczanie to jest niezależne od wykonywania przez procesor programu i może zostać wstrzymane tylko przez odłączenie sygnału taktującego licznik. Taki podstawowy *timer* jest niezwykle prosty, a jego zastosowania są ograniczone, stąd też projektanci AVR dodali do niego sporo ulepszeń, zwiększających funkcjonalność i uniwersalność liczników. W efekcie *timery* mogą być użyte do:

- ◆ generowania przebiegu PWM,
- ◆ działania jako 8- lub 16-bitowe liczniki,
- ◆ precyzyjnego odmierzania czasu,
- ◆ zliczania impulsów zewnętrznych,
- ◆ mierzenia czasu pomiędzy impulsami,
- ◆ generowania przebiegów o określonej częstotliwości
- ◆ pomiarów wypełnienia przebiegu zewnętrznego,
- ◆ tworzenia zegarów RTC,
- ◆ sterowania przebiegiem wykonywanego programu,
- ◆ generowania impulsów inicjujących pracę innych układów, np. ADC
- ◆ i wielu innych rzeczy.

Poniżej zostały przedstawione ogólne informacje na temat struktury i konfiguracji *timera* wykorzystywanych w procesorach rodziny AVR. W danym modelu nie wszystkie funkcje muszą być dostępne, stąd też zawsze należy sprawdzić w notach katalogowych, jakie konkretne funkcjonalności udostępnia wybrany mikrokontroler.

## Sygnal taktujący

Podstawową funkcję *timera* jest zliczanie impulsów, stąd aby *timer* mógł działać, musi do niego zostać doprowadzony sygnał zegarowy. W mikrokontrolerach AVR *timery* mogą być taktowane przebiegiem pochodzącym z różnych źródeł. Okres doprowadzonego do licznika przebiegu zegarowego określa minimalny możliwy do zmierzenia interwał czasowy. Jeśli okres zegara taktującego *timer* wynosi np. 100 µs, to będzie on pracował z taką rozdzielczością, a zwiększenie się licznika o 1 będzie sygnalizowało upływ odcinka czasu równego 100 µs. Stąd też ważne jest nie tylko doprowadzenie sygnału zegarowego, ale także zapewnienie jego okresu równego zakładanej rozdzielczości pomiaru. Z kolei doprowadzenie sygnału o dużej częstotliwości zwiększy rozdzielczość pomiaru, lecz jednocześnie zmniejszy maksymalną długość interwału czasowego, którą można zmierzyć, zanim dojdzie do przepelnienia licznika.

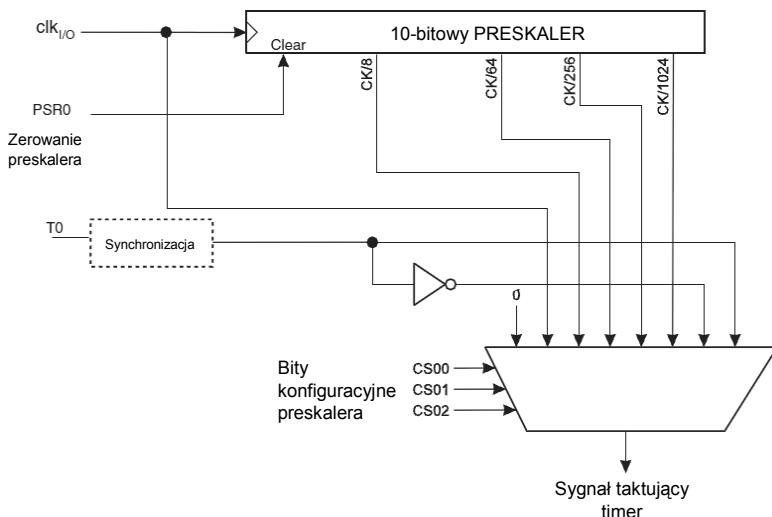
## Wewnętrzny sygnal taktujący

Najprostszą możliwością jest wykorzystanie do taktowania *timera* sygnału zegarowego taktującego procesor. Sygnał ten może zostać doprowadzony bezpośrednio do *timera* lub poprzez specjalny dzielniczak, tzw. preskaler. Umożliwia to wstępne, dosyć pobiczne określenie częstotliwości zegara taktującego *timer*. Preskaler umożliwia podzielenie sygnału taktującego poprzez jedną z wybranych wartości: 1, 8, 64, 256, 1024 (rysunek 17.1). W efekcie zmniejsza się rozdzielczość, z jaką mierzono są interwały czasowe, lecz jednocześnie wydłuża się maksymalny mierzony odcinek czasu.

Preskaler działa jak zwykły licznik, liczący do ustalonej wartości; po zliczeniu generowany jest impuls, który następnie trafia do *timera*. Preskaler działa cały czas, niezależnie od konfiguracji *timera*, co czasami sprawia problemy. Zmiana preskalera *timera* powoduje, że czas do wygenerowania pierwszego impulsu zależny jest od bieżącej wartości licznika preskalera (który jest niedostępny dla programisty). Wyobraźmy sobie, że wybrano preskaler równy 1024. W chwili włączenia *timera* preskaler zawiera pewną nieokreśloną wartość, w efekcie pierwszy impuls na wyjściu preskalera może pojawić się w dowolnym momencie, od 1 do 1024 impulsów zegara taktującego preskaler, co może zakłócić dokładność odmierzanego czasu. Dopiero kolejne impulsy będą pojawiały się zgodnie z wybraną przez preskaler liczbą taktów zegara taktującego. **Aby uniknąć problemu zniekształcenia pierwszego impulsu, wprowadzono możliwość zerowania rejestru preskalera.** Powoduje to rozpoczęcie zliczania od ustalonej wartości preskalera, równej 0. W efekcie mamy gwarancję, że czas do wygenerowania pierwszego impulsu będzie wynosił dokładnie tyle impulsów zegara taktującego, jaką wartość preskalera wybrano.

**Rysunek 17.1.**

Budowa typowego preskaleru procesora AVR. Jako sygnał taktujący można wykorzystać wewnętrzny sygnał zegarowy takiujący rdzeń lub zewnętrzny przebieg zegarowy doprowadzony do wejścia  $T_n$

**Uwaga**

W wielu przypadkach jeden preskaler dzielony jest pomiędzy więcej niż jeden timer. Wyzerowanie preskaleru będzie się więc odbijało na pracy wszystkich taktowanych z niego timerów.

Jedyną możliwością zatrzymania timera (jego wyłączenia) jest wybranie takiej konfiguracji preskaleru, w której jest on odłączony od sygnału zegarowego.

**Preskaler zaawansowany**

W niektórych procesorach preskaler (głównie ATTiny) ma bardziej skomplikowaną budowę. Licznik takiego preskaleru jest 14-, a nie 10-bitowy, co umożliwia podział sygnału zegarowego przez: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16 384, a więc kolejne potęgi liczby 2. Oprócz tego, taki preskaler może być taktowany na dwa sposoby:

- ◆ synchronicznie, gdzie sygnałem taktującym jest sygnał zegarowy procesora;
- ◆ asynchronicznie, gdzie sygnałem zegarowym jest tzw. szybki zegar takiujący peryferia o częstotliwości zwykle 32 lub 64 MHz.

Sygnal asynchroniczny otrzymywany jest w pętli synchronizacji fazowej (ang. *Phase Locked Loop*). Konfiguracja pętli synchronizacji fazowej odbywa się przy pomocy rejestru PLLCSR (ang. *PLL Control and Status Register*). Pętlę włącza się, ustawiając bit PLLE (ang. *PLL Enable*). Bit ten jest ustawiany automatycznie, jeśli procesor jest taktowany z wewnętrznego generatora RC (w takiej sytuacji PLL służy także do generowania zegara takiującego rdzeń procesora). Po włączeniu PLL należy odczekać pewien czas (ok. 100  $\mu$ s) na ustabilizowanie się częstotliwości układu. Synchronizację sygnalizuje ustawiony bit PLOCK (ang. *PLL Lock Detector*). Po jego ustawieniu można odblokować taktowanie preskaleru z PLL poprzez ustawienie bitu PCKE (ang. *PCK Enable*). Powoduje to zmianę taktowania preskaleru z sygnału synchronicznego na asynchroniczny sygnał uzyskiwany z układu PLL. Dodatkowo możemy wybrać jeden z dwóch

trybów pracy PLL przy pomocy bitu `LSM` (ang. *Low Speed Mode*). Jego ustawienie powoduje, że generowany sygnał ma częstotliwość 32 MHz; jeśli jest wyzerowany, generowany jest sygnał o częstotliwości 64 MHz.



Bitu `LSM` nie można ustawić, jeśli procesor jest taktowany z wewnętrznego generatora RC i używa PLL do generowania sygnału taktującego rdzeń.

## Zewnętrzny sygnał taktujący

*Timer* może być taktowany zewnętrznym sygnałem doprowadzonym do pinu procesora (zwyczkle oznaczonym jako `Tn`, gdzie *n* to numer *timera*). Sygnał ten musi mieć częstotliwość nie większą niż częstotliwość taktowania procesora/2, przy założeniu, że ma on współczynnik wypełnienia równy 50%. Wynika to z obecności układu synchronizującego oraz układu detekcji zboczy, który jest taktowany sygnałem zegarowym procesora. Układ ten wprowadza też opóźnienie 2,5 do 3,5 cyku pomiędzy zboczem doprowadzonego sygnału a zmianą sygnału taktującego *timer*. Zewnętrzne źródło zegarowe może być tak skonfigurowane, aby powodować zmianę stanu *timera* na zboczu narastającym lub opadającym.



Zewnętrzny sygnał taktujący nie może być dzielony przy pomocy preskalera. Jest on bezpośrednio używany do taktowania licznika.

Z niektórymi *timerami* sprzężony jest także zewnętrzny generator o niskiej częstotliwości, do którego zazwyczaj podłącza się kwarc zegarkowy (o częstotliwości 32 768 Hz) w celu realizacji zegara RTC. Pokazane zostanie to na przykładzie praktycznym na końcu rozdziału.

## Licznik

Fundamentem działania każdego *timera* jest licznik (ang. *Counter*). Licznik może być 8- lub 16-bitowy. Jego wartość zmienia się po każdym okresie zegara taktującego *timer* i w zależności od konfiguracji może ulec zwiększeniu lub zmniejszeniu o 1. Po osiągnięciu wartości maksymalnej (ang. *Top*) licznik może „zawijać się” lub może zacząć liczyć w dół. Analogicznie dzieje się po osiągnięciu wartości minimalnej (ang. *Bottom*). Sposób zachowania się licznika po osiągnięciu skrajnych pozycji określony jest przez tryb pracy *timera*. Osiągnięcie wartości skrajnych (przepełnienie licznika) sygnalizowane jest poprzez ustawienie bitu `TOVn` (ang. *Timer/Counter n Overflow Flag*) rejestru `TIFR` (ang. *Timer/Counter n Interrupt Flag Register*). Flaga ta może być traktowana jako dodatkowy bit rejestru licznika. Jest ona kasowana programowo, poprzez wpisanie do niej wartości 1, lub automatycznie po skoku do procedury obsługi przerwania przepełnienia licznika (`TIMERn_OVF_vect`). Przerwanie przepełnienia licznika można odblokować poprzez ustawienie bitu `TOIEn` (ang. *Timer/Counter n Overflow Interrupt Enable*)

rejestru TIMSK (ang. *Timer/Counter n Interrupt Mask Register*). Licznik można programowo inicjować dowolną wartością, można także w każdej chwili rejestr licznika odczytywać. Operacji tych dokonuje się poprzez rejestr TCNTn (ang. *Timer/Counter Register*).

## Układ porównywania danych

Z każdym licznikiem *timera* związany jest jeden lub więcej rejestrów, których zawartość jest stale porównywana z licznikiem. Są to rejestrory OCRn (ang. *Output Compare Register*). Kolejne rejestrory oznacza się sufiksem A, B, C itd. Pozytywny wynik porównania rejestrów TCNTn z danym rejestrarem OCRn powoduje ustawienie flagi OCFn (ang. *Timer/Counter n Output Compare Match Flag*) w rejestrze TIFRn — każdy z rejestrów OCR ma własną flagę. Jej zerowanie odbywa się programowo, poprzez wpisanie na odpowiednią pozycję rejestru wartości 1, lub automatycznie, po wykonaniu skoku do procedury obsługi przerwania TIMERn\_COMPx\_vect. Aby wywołanie takiego przerwania było możliwe, należy ustawić odpowiednią flagę zezwolenia na przerwanie OCIEn (ang. *Timer/Counter Output Compare Match Interrupt Enable*) w rejestrze TIMSKn.



Wskazówka

W przypadku kiedy nowa wartość zapisywana do rejestrów OCRn jest mniejsza niż bieżąca wartość licznika TCNTn, kolejne zdarzenie *Compare Match* nastąpi po przepeleniu się licznika i zliczeniu do nowej wartości rejestrów OCRn.

W efekcie czas do kolejnego zdarzenia ulega wydłużeniu. Podobnie jeśli nowa wartość rejestrów OCRn jest większa niż aktualna wartość rejestrów TCNTn, czas do kolejnego zdarzenia ulegnie skróceniu.

Warto także pamiętać, że wpisywanie nowej wartości do rejestrów TCNTn powoduje zablokowanie porównania z rejestrami OCRn na jeden cykl zegara *timera*. W efekcie, pomimo że TCNTn będzie równe OCRn, nie zostanie ustawiona flaga OCFn. Zdarzenie *Compare Match* można wywołać programowo poprzez ustawienie flagi FOCn (ang. *Force Output Compare*) rejestru TCCRn (ang. *Timer/Counter Control Register*). Jej ustawienie ma dokładnie takie same konsekwencje jak prawdziwe zdarzenie *Compare Match*.

## Wpływ na piny IO

Zdarzenie *Compare Match* może także służyć do generowania przebiegów na pinach IO procesora. Jego wpływ na generowany sygnał określany jest kombinacją bitów COM0:1 rejestrów kontrolnych *timera* — tabela 17.1. Znaczenie tych bitów zależne jest od trybu pracy *timera*.

W trybie FAST PWM stan pinu zależy m.in. od konfiguracji bitów WGM — tabela 17.2. Znaczenie tych bitów w trybie PWM z korekcją fazy pokazane zostało w tabeli 17.3.

Jak widać, zdarzenie *Compare Match* może wpływać także na stan licznika *timera* oraz stan wybranego portu wyjściowego, co umożliwia generowanie złożonych przebiegów zegarowych.

**Tabela 17.1.** Znaczenie bitów COM w przypadku pracy timera w trybach innych niż PWM

<b>COM1</b>	<b>COM2</b>	<b>Stan pinu IO</b>
0	0	Pin nie podlega kontroli <i>timera</i> .
0	1	Zdarzenie <i>Compare Match</i> zmienia stan pinu na przeciwny.
1	0	Zdarzenie <i>Compare Match</i> zeruje pin.
1	1	Zdarzenie <i>Compare Match</i> ustawia pin.

**Tabela 17.2.** Znaczenie bitów COM w przypadku pracy timera w trybie FAST PWM

<b>COM1</b>	<b>COM2</b>	<b>Stan pinu IO</b>
0	0	Pin nie podlega kontroli <i>timera</i> .
0	1	$\text{WGM02}=0$ — pin nie podlega kontroli timera, $\text{WGM02}=1$ — zmiana stanu pinu na przeciwny po zdarzeniu <i>Compare Match</i> .
1	0	Wyzerowanie pinu po zdarzeniu <i>Compare Match</i> , ustawienie po osiągnięciu przez licznik wartości minimalnej.
1	1	Ustawienie pinu po zdarzeniu <i>Compare Match</i> , wyzerowanie po osiągnięciu przez licznik wartości minimalnej.

**Tabela 17.3.** Znaczenie bitów COM w przypadku pracy timera w trybie PWM z korekcją fazy

<b>COM1</b>	<b>COM2</b>	<b>Stan pinu IO</b>
0	0	Pin nie podlega kontroli <i>timera</i> .
0	1	$\text{WGM02}=0$ — pin nie podlega kontroli timera, $\text{WGM02}=1$ — zmiana stanu pinu na przeciwny po zdarzeniu <i>Compare Match</i> .
1	0	Wyzerowanie pinu po zdarzeniu <i>Compare Match</i> , jeśli licznik liczy w góre, ustawienie, jeśli licznik liczy w dół.
1	1	Ustawienie pinu po zdarzeniu <i>Compare Match</i> , jeśli licznik liczy w góre, wyzerowanie, jeśli licznik liczy w dół.



Wskazówka

Ustawienie bitów COM1 lub COM2 powoduje przejęcie funkcji danego pinu portu *IO* przez *timer*. Lecz kierunkiem takiego pinu ciągle można sterować poprzez rejestr DDRx. Jeśli taki pin będzie skonfigurowany jako wejściowy, sygnału generowanego przez *timer* nie będzie można obserwować na odpowiednim wyprowadzeniu procesora, chociaż będzie można go programowo odczytać poprzez odpowiadający mu bit rejestrów PINx.

## Moduł przechwytywania zdarzeń zewnętrznych

Moduł ten potrafi nadawać zdarzeniu zewnętrznemu marker czasowy (ang. *Timestamp*), dzięki czemu w prosty sposób można mierzyć częstotliwość, współczynnik wypełnienia czy też relacje czasowe pomiędzy sygnałami. Moduł ten jest aktywowany poprzez

zmianę stanu pinu ICP (ang. *Input Capture Pin*) bądź też przez inne zdarzenie, np. zmianę stanu wyjścia komparatora analogowego. Po zajściu zdarzenia zawartość licznika TCNT<sub>n</sub> jest kopiwana do rejestru ICR<sub>n</sub> (ang. *Input Capture Register*), w ten sposób „zamrażany” jest stan licznika w momencie wystąpienia zdarzenia.



Kolejne zdarzenie powoduje zapisanie nowej wartości do rejestru ICR, nadpisując jego poprzednią wartość.

Aby zapobiec takiej sytuacji, należy odpowiednio często odczytywać wartość rejestru ICR, a najlepiej włączyć generowanie przerwania po wystąpieniu zdarzenia *input capture*.

O nowej wartości rejestru ICR informuje ustawiona flaga ICF rejestru TIFR<sub>n</sub>. Jej ustawienie sygnalizuje wpisanie nowej wartości do rejestru ICR. Flaga ta może zostać skasowana programowo lub automatycznie po wejściu do procedury obsługi przerwania TIM<sub>n</sub>\_CAPT\_vec. Aby odblokować przerwania związane ze zdarzeniem *input capture*, należy ustawić bit ICIE (ang. *Timer/Counter n Input Capture Interrupt Enable*) rejestru TIMSK<sub>n</sub>.

Za pomocą bitu ICES (ang. *Input Capture Edge Select*) rejestru TCCRnB można określić, które zbocze powoduje wpis do rejestru ICR. Jeżeli bit ten jest równy 0, jest to zbocze opadające, a jeśli wynosi 1, jest to zbocze narastające.

## Eliminacja szumów

Na wejściu ICP można włączyć eliminację szumów, co zapobiega zliczaniu przypadkowych impulsów. Układ ten działa jako filtr cyfrowy. Układ czterokrotnie próbkuje badany sygnał; jeśli wartość wszystkich próbek jest równa, układ zmienia stan wyjścia. Próbki zbierane są co jeden okres zegara taktującego procesor (a nie *timer*), a włączenie tego układu wprowadza stałe, czterotaktowe opóźnienie pomiędzy zmianą sygnału doprowadzonego do pinu ICP a zmianą stanu wyjścia tego układu. Układ ten włącza się poprzez ustawienie bitu ICNC (ang. *Input Capture Noise Canceler*) rejestru TCCRnB.

## Komparator jako wyzwalacz zdarzenia ICP

Oprócz sygnału z doprowadzonego do pinu ICP procesora można użyć także sygnału generowanego przez komparator analogowy. Sygnał ten jest doprowadzony do modułu ICP tak samo jak sygnał z zewnętrznego wyprowadzenia ICP, w efekcie również dla takiego sygnału można włączyć funkcję eliminacji szumów. Aby wykorzystać wyjście komparatora jako generator zdarzeń, należy ustawić bit ACIC (ang. *Analog Comparator Input Capture Enable*) rejestru ACSR.

# Tryby pracy timera

Każdy *timer* może pracować w różnych, predefiniowanych trybach — tabela 17.4. Aktualny tryb pracy *timera* określają bity WGM (ang. *Waveform Generation Mode*), a w trybach PWM dodatkowo bity COM, których ustawienie powoduje ewentualną inwersję sygnału wyjściowego. Rozpoczynając pracę z *timerem*, należy wybrać tryb pracy najbardziej pasujący do rozwiązywanego problemu.

**Tabela 17.4.** Konfiguracja bitów i parametry pracy timera w różnych trybach

Tryb	Bity WGM 3 – 0	Tryb działania timera/licznika	Wartość maksymalna	Uaktualnianie OCRnx	Flaga TOVn ustawiana na:
0	0000	Normalny (prosty)	0xFFFF	natychmiast	przy maksimum
1	0001	PWM, korekcja fazy, 8-bit	0x00FF	na szczytanie	na dole
2	0010	PWM, korekcja fazy, 9-bit	0x01FF	na szczytanie	na dole
3	0011	PWM, korekcja fazy, 10-bit	0x03FF	na szczytanie	na dole
4	0100	CTC	OCRnA	natychmiast	przy maksimum
5	0101	Fast PWM, 8-bit	0x00FF	na szczytanie	na dole
6	0110	Fast PWM, 9-bit	0x01FF	na szczytanie	na dole
7	0111	Fast PWM, 10-bit	0x03FF	na szczytanie	na dole
8	1000	PWM, korekcja fazy i częstotliwości	ICRn	na dole	na dole
9	1001	PWM, korekcja fazy i częstotliwości	OCRnA	na dole	na dole
10	1010	PWM, korekcja fazy	ICRn	na szczytanie	na dole
11	1011	PWM, korekcja fazy	OCRnA	na szczytanie	na dole
12	1100	CTC	ICRn	natychmiast	przy maksimum
13	1101	(zarezerwowany)	-	-	-
14	1110	Fast PWM	ICRn	na dole	na szczytanie
15	1111	Fast PWM	OCRnA	na dole	na szczytanie

## Tryb prosty

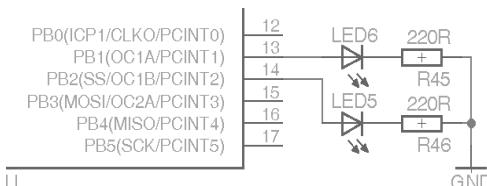
Jest to najprostszy i domyślny tryb pracy licznika. W tym trybie licznik liczy od 0 do wartości maksymalnej (255 lub 65 535), po czym rozpoczyna liczenie od nowa. Każde „przekręcenie się” licznika generuje przerwanie TIMn\_OVF\_vect (o ile oczywiście ustalono odpowiednią flagę zezwolenia na przerwanie), w każdej sytuacji ustawiany jest bit TOV. Tryb ten może być wykorzystany do odmierzania czasu (przy czym jest to zazwyczaj stały czas, określony poprzez interwał pomiędzy kolejnymi przepelnieniami licznika), jest on wykorzystywany także w połączeniu z modelem przechwytywania zdarzeń zewnętrznych. Inne odcinki czasowe można uzyskać poprzez wykorzystanie rejestrów

OCR i przerwania TIMn\_COMP\_vect. Dzięki temu na odpowiednich wyprowadzeniach OCn można uzyskać przebiegi o różnym współczynniku wypełnienia, które można wykorzystać do np. sterowania jasnością wyświetlaczy, tak jak to pokazano w rozdziale 18.

Działanie tego trybu zostanie pokazane na przykładzie dwóch diod LED podłączonych do wyjść OC1A i OC1B *timera 1*, zgodnie ze schematem pokazanym na rysunku 17.2.

### Rysunek 17.2.

Schemat podłączenia diod do układu ATMega88



Przy zastosowaniu prostego trybu pracy *timera* jako licznika wykorzystane zostanie przerwanie przepelnienia *timera* do zmiany stanu diod LED.

Najpierw wyliczmy częstotliwość, z jaką będą mrugać diody. W trybie normalnym przerwanie przepelnienia generowane jest po przepelnieniu licznika, czyli dla 16-bitowego *timera 1* co 65 536 cykli zegara taktującego *timer*. Zegar ten pochodzi z preskalera, a więc jeden takt zegara *timera* to ustawiona na preskalerze liczba cykli zegara systemowego. Stąd częstotliwość wyniesie  $F_{CLK}/(\text{preskaler} \cdot 65\,536)$ . Najpierw zainicjowany zostanie *timer*, tak aby generował przerwanie po przepelnieniu licznika:

```
void timer_init()
{
    TIMSK1=_BV(TOIE1);
    TCCR1B=_BV(CS12); //Preskaler 256
    DDRB|=_BV(PB1) | _BV(PB2);
}
```

Razem z inicjacją *timera* ustawione zostały piny portu sterującego diodami na wyjście. Kolejnym etapem będzie napisanie procedury obsługi przerwania:

```
ISR(TIMER1_OVF_vect)
{
    static uint8_t LED=0b010;
    LED^=0b110;
    uint8_t tmp=PORTB & 0b11111001;
    PORTB=(tmp | LED);
}
```

W procedurze tej diody będą naprzemiennie zmieniać stan, dzięki wykorzystaniu operacji sumy wyłączającej. Pozostaje jeszcze napisanie funkcji *main*:

```
int main()
{
    timer_init();
    sei();
    while(1);
}
```

Jak widać, w tym trybie nie mamy wielkiego wpływu na częstotliwość mrugania diod, możemy ją zmienić w pewnym zakresie poprzez zmianę preskalera.

Jednak ten sam efekt można uzyskać, znacznie upraszczając program, dzięki wykorzystaniu sprzętowych możliwości *timera*. Jak wcześniej pokazano, *timer* potrafi bezpośrednio sterować dedykowanymi wyjściami mikrokontrolera. W tym celu należy odpowiednio skonfigurować bity COM<sub>n</sub> rejestru kontrolnego *timera*. W tym trybie pojawia się jednak pewien problem — stan wyjścia może zmienić się wyłącznie w wyniku zdarzenia *Compare Match*, natomiast przepełnienie licznika nie ma na nie wpływu. Nie jest to jednak wielki problem, wystarczy odpowiednio skonfigurować rejestrów OCR<sub>n</sub>. Poniższy program wykorzystuje tę możliwość:

```
void timer_init()
{
    TCCR1A|=_BV(COM1A0) | _BV(COM1B0));
    OCR1A=32678;
    OCR1B=32768;
    TCCR1B=_BV(CS12); //Preskaler 256
    TCCR1C=_BV(FOC1A);
    DDRB|=_BV(PB1) | _BV(PB2));
}
```

Bity COM zostały ustawione tak, aby przy każdym zdarzeniu *Compare Match* stan portu zmieniał się na przeciwny. Zdarzenie to wystąpi po zliczeniu 32678 impulsów (połowa przedziału licznika). W efekcie diody będą migać dwukrotnie szybciej, ale możemy to zmienić poprzez zmianę wartości preskalera. Ponieważ tak jak w poprzednim przykładzie diody mają mrugać w przeciwfazie, należy wstępnie ustawić stan wyjść OC<sub>n</sub>. Odbywa się to poprzez ustawienie bitu FOC1A, co powoduje natychmiastowe wygenerowanie zdarzenia *Compare Match* dla pinu OC1A i zmianę jego stanu na przeciwny. Kolejne zdarzenia *Compare Match* nie zmieniają już fazy.

Ponieważ w tym przypadku nie zostały wykorzystane przerwania, upraszcza się także funkcja main:

```
int main()
{
    timer_init();
}
```

Jak widać, składa się ona tylko z wywołania inicjalizacji *timera* (timer\_init). Po zakończeniu main procesor wpada w nieskończoną pętlę z zablokowanymi przerwaniami — **normalnie byłby to błąd, lecz w powyższym przykładzie cały program jest realizowany wyłącznie sprzętowo**.

Wykorzystanie zdarzenia *Compare Match* umożliwia także łatwą zmianę współczynnika wypełnienia przebiegu. W tym celu wystarczy zmienić wartości, jakimi są zainicjowane rejestrów OCR1A i OCR1B. Stosunek wartości tych rejestrów do stałej 65 536 określa współczynnik wypełnienia (w powyższym przykładzie wynosi on  $32\ 768/65\ 536 = 0,5$ , czyli 50%).

Z trybu normalnego właściwie niewiele więcej można uzyskać. Brak możliwości dokładnej regulacji częstotliwości przebiegu znacząco ogranicza jego zastosowania. Jednak tryb ten świetnie nadaje się do odmierzania interwałów czasowych. Zobaczmy to na przykładzie:

```
void timer_init()
{
    TCCR1A=(_BV(COM1A1) | _BV(COM1A0)| _BV(COM1B1) | _BV(COM1B0));
    TCCR1C=_BV(FOC1A) | _BV(FOC1B);
    TCCR1A=_BV(COM1A1) | _BV(COM1B1));
    OCR1A=32678;
    OCR1B=50768;
    TCCR1B=_BV(CS12) | _BV(CS10); //Preskaler 1024
    GTCCR=_BV(PRSYNC); //Wyzeruj preskaler
    DDRB|=_BV(PB1) | _BV(PB2));
}
```

Tym razem *timer 1* dla lepszego uwidocznienia efektu jest taktowany z preskalera dzielącego sygnał przez 1024. Po jego ustawieniu dla dokładnego odmierzenia czasu preskaler jest zerowany, poprzez ustawienie bitu PSRSYNC rejestru GTCCR. W efekcie zwiększenie licznika o 1 następuje co 1024 taktów zegara. Rejestry OCR1A i OCR1B zawierają czas, po jakim ma nastąpić zdarzenie *Compare Match*. Jest on równy OCR1·preskaler taktów zegara. Powyższa funkcja najpierw ustawia wyjścia OCR1A i OCR1B w stan wysoki poprzez odpowiednią konfigurację bitów COM i natychmiastowe wywołanie zdarzenia *Compare Match* (ustawienie bitów FOC1A i FOC1B).

**Wskazówka**

Stanu pinów OC1A i OC1B nie można zmienić poprzez wpisanie pożądanych wartości do rejestru PORTx, gdyż funkcja tych pinów jest przejmowana przez *timer* i stan rejestru PORTx nie ma znaczenia.

Oczywiście, zamiast sterować portami przez *timer*, można wykorzystać odpowiednie przerwania (TIMER1\_COMPA\_vect i TIMER1\_COMPB\_vect). Umożliwia to realizację precyzyjnych opóźnień czasowych, bez wykorzystywania funkcji zdefiniowanych w pliku <utils\delay.h>. Przewaga wykorzystania *timerów* polega w tym przypadku na nie-wstrzymywaniu procesora, w efekcie procesor może normalnie realizować program.

## Tryb CTC

Tryb CTC (ang. *Clear Timer on Compare Match*) jest jednym z najbardziej użytecznych i najczęściej wykorzystywanych trybów pracy *timera*. W tym trybie licznik zlicza aż do uzyskania wartości określonej w rejestrze OCR lub ICR. Rejestr, z którym porównywana jest wartość licznika, wybierany jest stosowną kombinacją bitów WGM. Po doliczeniu do wartości określonej rejestrami OCR lub ICR licznik jest zerowany, a liczenie odbywa się od początku. Dzięki temu mamy lepszą kontrolę nad pracą licznika i generowanym przebiegiem niż w trybie prostym.

**Wskazówka**

W tym trybie nie jest generowane przerwanie przepełnienia licznika TIMn\_OVF\_vect. Jest to spowodowane tym, że licznik nie ulega przepełnieniu. Zamiast tego należy używać wektora TIMn\_COMP\_vect lub TIMn\_CAPT\_vect, w zależności od tego, czy używany jest do porównania rejestr OCR, czy ICR.

W trybie prostym częstotliwość występujących przerwań wynosi  $F_{CPU}/(2 \cdot N \cdot (1 + OCRnA))$ , gdzie  $N$  to wartość preskalera.



**Wskazówka**

W przypadku zmiany wartości licznika TCNT lub rejestru OCR istnieje możliwość omijnięcia zdarzenia *Compare Match*, w efekcie czas do następnego zdarzenia ulega wydłużeniu o czas potrzebny do przepełnienia się licznika. W takiej sytuacji jest generowane także przerwanie TIMn\_OVF\_vect.

Zobaczmy, jak wygląda przykład z migającymi diodami w trybie CTC:

```
void timer_init()
{
    TCCR1A=_BV(COM1A0) | _BV(COM1B0));
    TCCR1C=_BV(FOC1A);
    OCR1A=F_CPU/1024/1;
    TCCR1B=_BV(WGM12) | _BV(CS12) | _BV(CS10); //Preskaler 1024, CTC
    DDRB|=_BV(PB1) | _BV(PB2));
}
```

Jak widać, w konfiguracji *timera* niewiele się zmieniło — ale tym razem został wybrany tryb pracy CTC. W tym trybie licznik liczy do wartości maksymalnej, określonej wartością OCR1A<sup>1</sup>. Dzięki temu można precyzyjnie określić częstotliwość generowanego przebiegu. Efektem ubocznym wykorzystania rejestru OCR1A jest generowanie zdarzeń *Compare Match* każdorazowo, kiedy licznik *timera* osiągnie wartość równą rejestrowi OCR1A. Rejestr OCR1B można wykorzystać do przesuwania fazy sygnału pojawiającego się na pinie OC1B w stosunku do fazy sygnału na pinie OC1A.



**Wskazówka**

Jeśli wartość rejestru OC1B będzie większa niż OC1A, to zdarzenie *Compare Match B* nigdy nie wystąpi.

## Tryby PWM

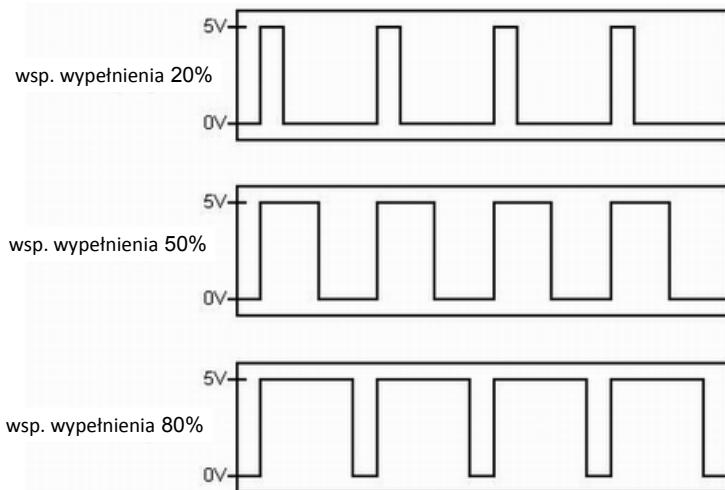
PWM (ang. *Pulse Width Modulation*) jest metodą polegającą na modulacji szerokości impulsu (zmianie współczynnika wypełnienia przebiegu) przy zazwyczaj jego stałej częstotliwości i amplitudzie (rysunek 17.3). Dla sygnału o stałej częstotliwości współczynnik wypełnienia jest stosunkiem wybranej fazy sygnału do jego okresu i waha się od 0 do 100%. Dla przykładu, jeśli okres sygnału trwa 100 ms, a stan 1 występuje przez 20 ms, to współczynnik wypełnienia wynosi  $(20/100) \cdot 100\% = 20\%$ .

Technika PWM jest wykorzystywana do kontroli układów analogowych przy pomocy układu cyfrowego. Zmieniając współczynnik wypełnienia impulsów, można regulować moc dostarczaną do układu, dzięki czemu w zależności od konfiguracji reguluje się amplitudę sygnału, prąd lub oba parametry jednocześnie. PWM znajduje także zastosowanie w regulacji obrotów silników, jasności diod LED i wielu innych.

<sup>1</sup> Istnieje także tryb CTC, w którym wartość maksymalna określona jest przez rejestr ICR1.

**Rysunek 17.3.**

Zasada działania  
PWM. Impulsy o stałej  
częstotliwości różnią  
się współczynnikiem  
wypełnienia



Wyjście PWM można wykorzystać do realizacji przetwornika cyfrowo-analogowego (DAC, ang. *Digital To Analog Converter*). Dodając na wyjściu cyfrowym odpowiednio dobrany filtr RC, można zamienić serię impulsów o zmiennym współczynniku wypełnienia na odpowiadające jej napięcie analogowe. W uproszczeniu, napięcie na wyjściu takiego filtra będzie wynosiło  $V_{cc} \cdot \text{wsp. wypełnienia}$ . Rozdzielcość, z jaką można określić wartość tego napięcia, zależy będzie od rozdzielcości, z jaką można ustawić współczynnik wypełnienia przebiegu. W precyzyjnych układach kontrolnych wyjście z takiego filtra podawane jest na wejście przetwornika ADC, dzięki czemu można ustawić szerokość impulsu tak, aby uzyskać pożądaną wartość napięcia.

PWM jest wykorzystywany także do redukcji mocy dostarczanej do układu, bez strat wynikających z regulacji. Jako przykład może posłużyć stabilizator liniowy i stabilizator impulsowy. W stabilizatorze liniowym w celu uzyskania pożądanego napięcia wyjściowego na wejściu musi panować napięcie nieco większe. Różnica tych napięć musi zostać wytracona na elemencie stabilizującym, powodując znaczne straty mocy. W efekcie część energii bezproduktywnie zamienia się na ciepło, co dodatkowo stwarza problem z jego odprowadzeniem. Inaczej rzecz ma się w stabilizatorach impulsowych. Wykorzystują one do działania przełącznik w postaci klucza tranzystorowego pracującego w dwóch stanach — zamkniętym lub otwartym. W stanie zamkniętym prąd przez klucz nie płynie, w efekcie nie ma na nim strat i nie wydziela się ciepło. Z kolei w przypadku klucza całkowicie otwartego opór źródło-dren jest zwykle bardzo mały, w efekcie straty są również niewielkie. Właściwe napięcie na wyjściu takiego elementu uzyskuje się, kluczując element przebiegiem o odpowiednim współczynniku wypełnienia. Dzięki temu, że klucz pracuje dwustanowo, straty energii są niewielkie. Tę zasadę stosuje się do sterowania wielu innych urządzeń, m.in. silników i ściemniaczy oświetlenia.

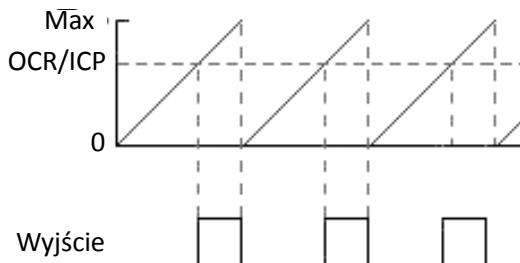
Mikrokontrolery AVR dysponują bogatym repertuarzem różnych trybów PWM dostosowanych do specyficznych potrzeb aplikacji. W trybach PWM można kontrolować zarówno częstotliwość, jak i współczynnik wypełnienia generowanego przebiegu. Zazwyczaj w tych trybach stosuje się maksymalną możliwą do uzyskania częstotliwość, jej ograniczeniem jest tylko prędkość przełączania elementów zewnętrznych. Umożliwia

to szybką reakcję na zmiany współczynnika wypełnienia. Częstotliwość przebiegu, podobnie jak w przypadku trybów normalnego i CTC, reguluje się wartością preskaleru i wartością maksymalną licznika. Współczynnik wypełnienia reguluje się poprzez zmianę wartości rejestru, z którym porównywana jest aktualna wartość licznika; osiągnięcie przez licznik wartości ustalonej powoduje inwersję sygnału wyjściowego.

## Tryb Fast PWM

W trybie szybkiego PWM (ang. *Fast PWM*) licznik liczy od zera do wartości maksymalnej, która może być ustawiona na stałe (zazwyczaj rozdzielcość wtedy wynosi 8, 9 lub 10 bitów), lub do wartości określonej przez rejestr OCRnA lub ICPn — rysunek 17.4.

**Rysunek 17.4.**  
Działanie timerów  
w trybie Fast PWM



Dla takiej konfiguracji rozdzielcość może być dowolnie określona, w przedziale od 2 do 16 bitów (w przypadku liczników 8-bitowych do 8 bitów). W trybie tym licznik zlicza tylko w góre, dzięki czemu częstotliwość uzyskanego przebiegu PWM jest dwukrotnie wyższa niż w innych trybach PWM. Stąd ten tryb nadaje się szczególnie do sterowania prostownikami, regulatorami mocy i do realizacji przetwornika DAC. W zależności od konfiguracji licznika osiągnięcie wybranego stanu sygnalizowane jest ustawieniem flagi TOV, ICF lub OCF. W przypadku odblokowania stosownych przerwa generowane są także przerwania o wektorach TIMn\_OVF\_vect, TIMn\_CAPT\_vect lub TIMn\_COMP\_vect. Jednak wraz ze zmianą współczynnika wypełnienia zmienia się faza generowanego przebiegu.

Częstotliwość generowanego przebiegu wynosi  $F_{CPU}/(2 \cdot N \cdot (1 + \text{wartość maksymalna licznika}))$ .

Dla trybów 8-, 9- i 10-bitowego wartość maksymalna jest stała i wynosi odpowiednio 255, 511 i 1023. W tych trybach częstotliwość PWM można regulować preskalerem. Jeśli taka regulacja jest niewystarczająca, to należy skorzystać z trybów, w których częstotliwość określana jest wartością rejestrów OCR lub ICP.



**Wskazówka**

W przypadku kiedy wartość rejestrów OCR jest większa niż maksymalna wartość licznika *timera*, zdarzenie *Compare Match* nigdy nie wystąpi, w efekcie na wyjściu będzie generowany przebieg o stałym poziomie logicznym.

Poniższy przykład wykorzystuje 8-bitowy tryb Fast PWM do generacji przebiegu o zmiennym wypełnieniu do sterowania jasnością diod podłączonych do wyjść OC1A i OC1B:

```
void timer_init()
{
    TCCR1A=_BV(COM1A1) | _BV(COM1B1));
    OCR1A=128;
    OCR1B=255;
    TCCR1A|=_BV(WGM10);
    TCCR1B=_BV(WGM12) | _BV(CS10); //Preskaler 1, FastPWM 8-bitowy
    DDRB|=_BV(PB1) | _BV(PB2));
}
```

Ponieważ w takich celach zwykle wykorzystuje się maksymalną częstotliwość pracy *timera*, stąd preskaler wynosi 1, a częstotliwość wyjściowa równa jest  $F_{CPU}/(2 \cdot 256)$  Hz. Zastosowane ustawienia rejestrów OCR powodują, że przebieg na wyjściu *OCR1A* będzie miał wypełnienie 50% (128/256), a na wyjściu *OCR1B* będzie miał prawie 100% (255/256). W efekcie średnie napięcie na tych pinach będzie wynosiło ok.  $Vcc/2$  i  $Vcc$ . Ten sam tryb można wykorzystać do realizacji szybkiego przetwornika cyfrowo-analogowego (DAC). Jeśli jednak taki przetwornik miałby zostać wykorzystany bezpośrednio do sterowania wejścia analogowego, to na jego wyjściu należy dodać odpowiednio dobrany do częstotliwości przebiegu filtr RC.

## Tryb PWM z korekcją fazy

Tryby PWM z korekcją fazy lub korekcją fazy i częstotliwości stosowane są w sytuacjach, w których zmiana współczynnika wypełnienia nie powinna wpływać na fazę sygnału (rysunek 17.5). Jest to kluczowe podczas sterowania silnikami. Jednak wprowadzana korekcja powoduje generowanie przebiegów o częstotliwości dwukrotnie niższej niż w trybie Fast PWM.

W przeciwieństwie do poprzedniego trybu, w tym trybie licznik *timera* odlicza od wartości najniższej do najwyższej, po jej osiągnięciu zaczyna odliczać ponownie do wartości najniższej, a następnie cały cykl się powtarza — rysunek 17.6.

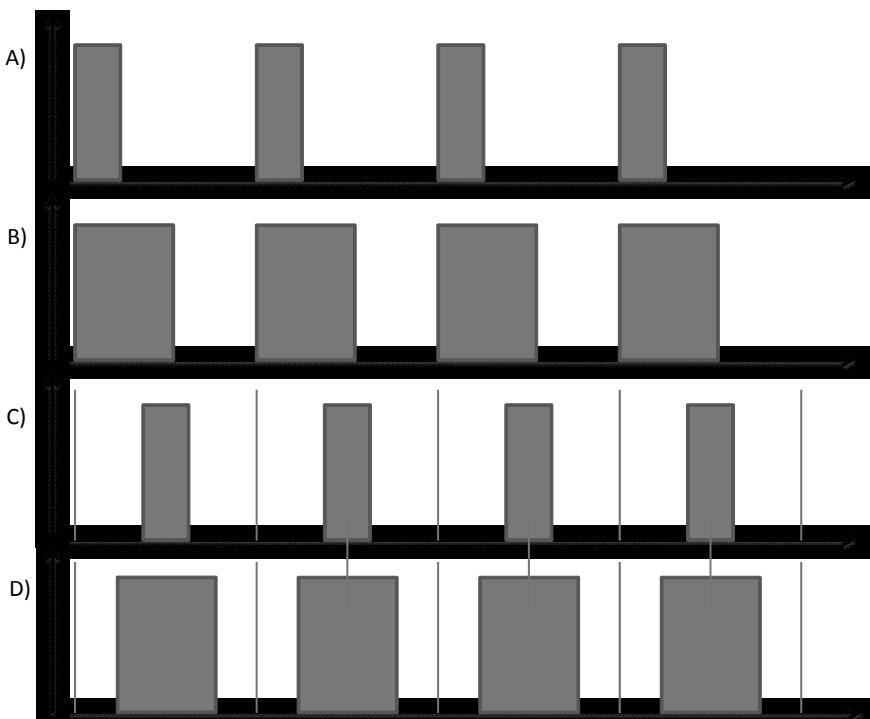
W tym trybie nieco inaczej zachowują się także piny OCn. W chwili zaistnienia zdarzenia *Compare Match* pomiędzy rejestrem TCNT a OCR, w chwili gdy licznik zlicza w góre, wyjście OCn jest zerowane. Przy ponownym zajściu tego zdarzenia, przy odliczaniu w dół, wyjście to jest ustawiane. Dodatkowo dostępny jest pin reprezentujący zanegowaną wartość stanu pinu OCn. Ze względu na symetrię tworzonego przebiegu PWM tryb ten jest preferowany przy sterowaniu silnikami. Rozdzielcość PWM jest analogiczna jak dla trybu Fast PWM, z tym że maksymalna częstotliwość przebiegu jest o połowę mniejsza.

W tym trybie flaga TOV ustawiana jest w sytuacji, kiedy licznik osiągnie wartość najniższą, odpowiednio do jego wartości ustawiane są flagi ICF i OCF.

Częstotliwość generowanego przebiegu wynosi  $F_{CPU}/(2 \cdot N \cdot \text{wartość maksymalna})$ , gdzie  $N$  to wartość preskalera.

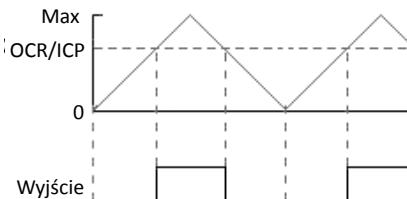
## Tryb PWM z korekcją fazy i częstotliwości

Tryb ten jest podobny do trybu poprzedniego. Różni się tylko chwilą, w której uaktualniany jest rejestr OCR (kopiwana jest do niego nowa wartość z bufora). Tryb ten w związku z tym jest zalecany w sytuacji, w której uaktualnianie rejestrów OCR następuje



**Rysunek 17.5.** Różnica pomiędzy zwykłym trybem PWM a trybem z korekcją fazy. A) i B) tryb Fast PWM z różnym współczynnikiem wypełnienia, C) i D) tryb PWM z korekcją fazy sygnału

**Rysunek 17.6.**  
Generowanie  
przebiegu PWM  
w trybach z korekcją  
fazy



w trakcie działania *timera*. Jeśli wartość, do której zlicza *timer*, nie ulega zmianie, tryb ten jest równoważny trybowi z korekcją fazy. Częstotliwość uzyskanego przebiegu jest taka sama, jak dla trybu PWM z korekcją fazy.

## Generator czasu martwego

W niektórych mikrokontrolerach z licznikami związany jest także generator czasu martwego (ang. *Dead Time Generator*). Układ ten umożliwia wprowadzenie czasu martwego pomiędzy zmianą stanu pinu OCn i zanegowanego OCn. Jest to wykorzystywane przy sterowaniu kluczów mocy, np. „górnymi” i „dolnymi” tranzystorów mostka H. Bez czasu martwego przy każdym przełączeniu przez chwilę oba tranzystory są częściowo włączone, co w efekcie prowadzi do przepływu przez nie bardzo dużego prądu, który może doprowadzić do ich uszkodzenia. Problem ten można wyeliminować poprzez wprowadzenie niewielkiego opóźnienia pomiędzy wyłączeniem jednego klucza a włączeniem

drugiego. Układ generatora czasu martwego taktowany jest przez własny preskaler, którego wejście podłączone jest do wyjścia preskalera *timera*. Dzięki temu można generować opóźnienia w dużym zakresie. Preskaler umożliwia dalsze podzielenie zegara *timera* przez 1, 2, 4 lub 8. Ustawienie preskalera kontrolują bitы DT<sub>PS</sub> (ang. *Dead Time Prescaler Bits*) znajdujące się w rejestrze TCCRnB. Uzyskany sygnał zegarowy doprowadzany jest na wejście 4-bitowego licznika związanego z danym generatorem. Licznik ten zlicza w dół od wartości określonej wartością rejestru DT<sub>N</sub> (ang. *Dead Time Value*). Czas martwy określany jest niezależnie dla obu komplementarnych wyjść OC<sub>n</sub>.

## Układ ochronny

Przy sterowaniu PWM elementami wykonawczymi w pewnych okolicznościach mogą pojawić się zdarzenia, na które program sterownika powinien niezwłocznie zareagować. Takimi zdarzeniami są wszelkiego typu awarie. Stąd też w nowszych procesorach AVR wprowadzono układ ochronny (ang. *Fault Protection Unit*), który kontroluje stan komplementarnych wyjść OC<sub>n</sub> *timera* w trybach PWM. Jego zadziałanie na skutek wyzwolenia sygnałem aktywującym powoduje natychmiastowe odłączenie pinów OC<sub>n</sub> od *timera* poprzez wyzerowanie bitów sterujących COM. W takiej sytuacji kontrolę nad pinami przejmuje układ IO mikrokontrolera (ich stan określany jest wartością odpowiednich bitów rejestru PORT<sub>X</sub>). Stwarza to możliwość błyskawicznego (w ciągu jednego okresu zegara taktującego procesor) ustawienia w sytuacji awaryjnej tych pinów w pożądanym stanie. Sygnałem aktywującym układ ochronny może być sygnał cyfrowy doprowadzony do dedykowanego wejścia procesora lub sygnał z wbudowanego komparatora analogowego.



Należy pamiętać, że zadziałanie układu ochronnego jednocześnie go dezaktywuje. Po usunięciu usterki układ ten należy ponownie włączyć.

Wskazówka

**Zmiana sygnału wyzwalającego zadziałanie układu ochronnego może go przypadkowo wyzwolić.** Stąd też najpierw należy wybrać odpowiednie źródło sygnału wyzwalającego, a dopiero później układ ten aktywować. Układ ochrony ma także własny podsystem redukcji zakłóceń, który można niezależnie włączać.



Należy pamiętać, że włączenie układu redukcji zakłóceń wprowadza dodatkowe opóźnienie (równe czterem cyklom zegara taktującego procesor) w zadziałaniu układu ochronnego.

Układ redukcji zakłóceń włącza się, ustawiając bit FPNC (ang. *Fault Protection Noise Canceler*). Wejście wyzwalające układ wybiera się poprzez ustawienie bitów FPAC (ang. *Fault Protection Analog Comparator Enable*) w przypadku wyboru komparatora analogowego. Jeśli układ ma być wyzwalany z wejścia IO procesora, bit ten powinien mieć wartość 0 przy włączonym układzie ochronnym. Po wyborze wejścia wyzwalającego należy określić zbocze, które ma wyzwolić układ. Dokonuje się tego poprzez modyfikację bitu FPES (ang. *Fault Protection Edge Select*). Jeśli ma on wartość 0, układ wyzwoli zbocze opadające, jeśli ma wartość 1 — zbocze narastające.

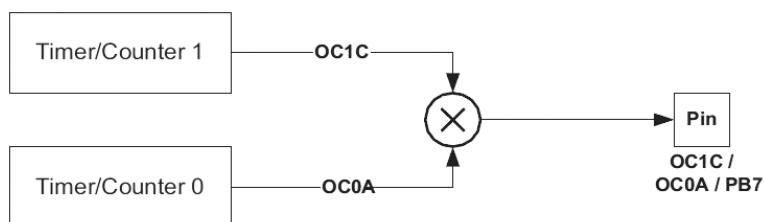
Zadziaływanie układu sygnalizuje ustawiona flaga FPF (ang. *Fault Protection Interrupt Flag*). Dodatkowo, jeśli ustawiona będzie flaga FPIE (ang. *Fault Protection Interrupt Enable*), zadziałyaniu układu ochronnego towarzyszyć będzie przerwanie o wektorze FAULT\_→PROTECTION\_vect. Układ ochronny włącza się, ustawiając bit FPEN (ang. *Fault Protection Mode Enable*).

## Modulator sygnału wyjściowego

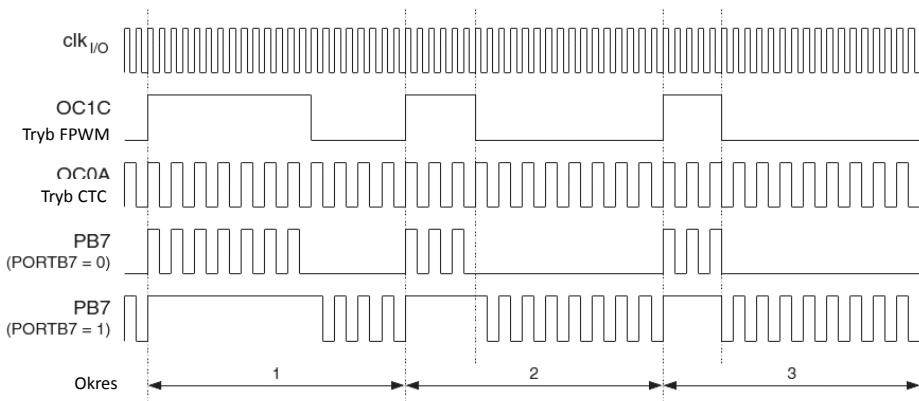
W niektórych modelach mikrokontrolerów AVR wyjścia dwóch timerów można ze sobą łączyć, uzyskując w ten sposób możliwość modulacji przebiegu generowanego przez jeden timer przebiegiem generowanym przez drugi timer — rysunek 17.7.

**Rysunek 17.7.**

Łączenie wyjść dwóch timerów w celu uzyskania przebiegu zmodulowanego przebiegiem drugiego z timerów



Aby uzyskany przebieg był dostępny na wyprowadzeniu portu I/O procesora, odpowiadający bit rejestru DDRx musi być ustawiony na wyjście (musi mieć wartość 1). Jeśli będzie ustawiony jako wejście, modulacja będzie się odbywać, lecz zmodulowany sygnał nie będzie dostępny dla programów zewnętrznych. Typ funkcji modulującej (iloczyn logiczny dwóch sygnałów lub suma logiczna) określony jest przez wartość odpowiedniego bitu rejestru PORTx. Jeśli bit ten ma wartość 0, dokonywana jest operacja iloczynu logicznego dwóch sygnałów, jeśli ma wartość 1, wykonywana jest operacja sumy logicznej — rysunek 17.8.



**Rysunek 17.8.** Różne typy modulacji sygnału uzyskane poprzez konfigurację timerów



Aby odblokować tryb modulacji, należy jednocześnie włączyć sterowanie pinem I/O współdzielonym przez dwa timerы.

# Miernik częstotliwości i wypełnienia

Jako przykład ilustrujący wykorzystanie *timerów* pokazany zostanie miernik częstotliwości i wypełnienia przebiegu. Częstotliwość można zmierzyć kilkoma sposobami. Najprościej jest zmierzyć liczbę cykli badanego przebiegu w jednostce czasu. Dzieląc liczbę cykli przez czas, uzyskuje się częstotliwość w hercach. Wymaga to zaangażowania dwóch *timerów* — jeden precyzyjnie odmierza interwały czasowe, natomiast na wejście zegarowe drugiego doprowadzany jest mierzony przebieg. Stan licznika zliczającego impulsy mierzonego sygnału w zadanych interwale czasowym daje w wyniku częstotliwość przebiegu w Hz. Sposób ten ma jednak dwie wady:

- ◆ Angażuje aż dwa *timery*.
- ◆ Aby dokładnie zmierzyć częstotliwość, należy zliczyć odpowiednią liczbę cykli mierzonego sygnału, co wymaga długich czasów próbkowania, szczególnie dla przebiegów o niskiej częstotliwości.

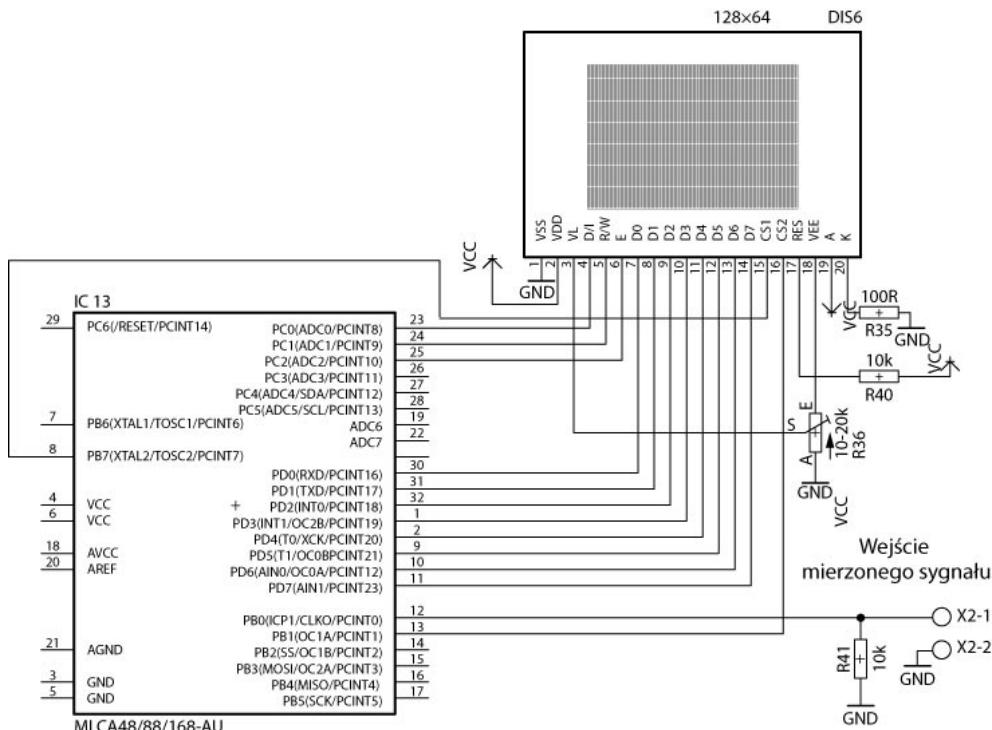
Inną metodą jest mierzenie okresu sygnału. Do realizacji tego celu wystarczy jeden *timer*. Mierzony przebieg doprowadza się do wejścia ICP *timera*, który jest taktowany zegarem o znanej częstotliwości. Czas pomiędzy kolejnymi zboczami narastającymi lub opadającymi mierzonego sygnału stanowi jego okres, z którego można wyliczyć częstotliwość. Taki sposób pomiaru oprócz zaangażowania mniejszej liczby *timerów* daje także lepszą dynamikę odczytów — do pomiaru wystarcza jeden okres mierzonego sygnału. **Wykorzystując go, należy jednak pamiętać, że maksymalny okres mierzonego sygnału nie może być dłuższy niż czas potrzebny do przepelnienia się licznika.** Dodatkowo, zmieniając konfigurację wyzwalania ICP, można na tej samej zasadzie dokonać pomiaru współczynnika wypełnienia. Wykorzystanie rejestru ICR zwiększa precyzję pomiarów — w momencie wystąpienia mierzonego zdarzenia (zmiany zbocza sygnału) stan licznika *timera* jest „zamrażany” w rejestrze ICR, skąd można go łatwo odczytać. W tym czasie *timer* w dalszym ciągu może zliczać czas. Rozdzielcość takiego pomiaru zależna jest od sygnału taktującego *timer* i równa jest jego okresowi. Można ją zwiększyć poprzez zwiększenie częstotliwości taktowania *timera*, lecz w takiej sytuacji rośnie minimalna możliwa do zmierzenia częstotliwość przebiegu.

Schemat układu został przedstawiony na rysunku 17.9.

Pisanie programu zacznijmy od konfiguracji *timera*:

```
void timer_init()
{
    TCCR1B= BV(ICNC1) | _BV(ICES1); //Włącz filtrowanie ICP, zdarzenie na zboczu narastającym
    TCCR1B|= _BV(CS11) | _BV(CS10); //Preskaler 64, tryb normalny
    TIMSK1= _BV(ICIE1) | _BV(TOIE1); //Przerwania ICP i overflow
}
```

Do pomiaru wykorzystany będzie *timer 1*, działający w trybie prostym z preskalerem 64. Mierzony sygnał doprowadzony jest do pinu ICP, a jego narastające zbocze powoduje zapamiętanie wartości rejestru TCNT1 w rejestrze ICR1. Jednocześnie takie zdarzenie oraz zdarzenie przepelnienia *timera* generować będzie przerwanie. Dzięki obsłudze zdarzenia



**Rysunek 17.9.** Schemat układu pomiaru częstotliwości i wypełnienia przy pomocy timera i wejścia ICP. Rezystor R41 ma za zadanie spolaryzować wejście ICP1 w sytuacji, w której nie doprowadzono żadnego mierzonego przebiegu

przepelenienia timera można będzie wykrywać sytuacje, w których mierzony sygnał ma zbyt niską częstotliwość. Pokazany program reaguje na taką sytuację w prosty sposób — po prostu sygnalizuje błędny pomiar:

```
volatile uint8_t is0k;

ISR(TIMER1_OVF_vect)
{
    is0k=0; //Błąd przepelenienia licznika
}
```

Lepszą reakcją byłaby zmiana preskalera, tak aby dostosować prędkość timera do mierzonego sygnału. Błąd sygnalizowany jest poprzez wyzerowanie zmiennej `is0k`.

Pozostaje jeszcze napisać procedurę obsługi zdarzenia *Input Capture*:

```
volatile uint16_t timestamp;
volatile uint16_t period;

ISR(TIMER1_CAPT_vect)
{
    static uint16_t loctimestamp;
```

```
if(TCCR1B & _BV(ICES1))
{
    GTCCR=_BV(PSRSYNC);
    TCNT1=0;
    TCCR1B=TCCR1B & (~_BV(ICES1)); //Następne zdarzenie na zboczu opadającym
    period=ICR1;
    timestamp=loctimestamp;
    isOk=255;
} else
{
    TCCR1B=TCCR1B | _BV(ICES1); //Następne zdarzenie na zboczu narastającym
    loctimestamp=ICR1;
}
```

Funkcja ta używa dwóch zmiennych: timestamp, przechowującej wartość licznika TCNT1 w chwili zmiany stanu badanego sygnału z wysokiego na niski, oraz period, zawierającej wartość licznika TCNT1 w chwili zmiany stanu badanego sygnału z niskiego na wysoki. Taka zmiana przy okazji wyzwala kolejny pomiar. Ponieważ nie ma możliwości takiego skonfigurowania *timera*, aby zdarzenie *Input Capture* występowało na obu zboczach, powyższa funkcja dokonuje zmiany konfiguracji *timera*, naprzemiennie konfiguruując wyzwalanie zdarzenia *Input Capture* zboczem narastającym lub opadającym. Na zboczu narastającym jednocześnie zerowana jest wartość licznika TCNT1 oraz resetowany jest preskaler. Dzięki temu pierwszy impuls taktujący *timer* nie jest zniekształcony (mógłby być skrócony o wartość licznika preskalera). Zeroowanie licznika oszczędza nam zapisania jego stanu w momencie rozpoczęcia pomiaru — dzięki temu wiadomo, że na początku pomiaru miał on wartość 0. Aby wartość zmiennych period i timestamp dotyczyła tego samego pomiaru, czas od zbocza narastającego do opadającego, z którego wyliczany jest współczynnik wypełnienia, przechowywany jest w zmiennej lokalnej loctimestamp. Jej wartość przepisywana jest do zmiennej timestamp dopiero w momencie rozpoczęcia kolejnego pomiaru (kiedy zmieniona period zawiera już prawidłową długość okresu, do którego odnosi się wartość zmiennej loctimestamp).

Pozostaje jeszcze przeliczyć uzyskane dane na częstotliwość i współczynnik wypełnienia oraz zaprezentować wyniki na wyświetlaczu:

```
int main()
{
    char bufor[6];
    uint16_t ts, pr;
    uint8_t ok, len;

    GLCD_init();
    GLCD_cls();
    GLCD_goto(0,0);
    GLCD_puttext_P(PSTR("Wsp. wypełnienia [%]:"));
    GLCD_goto(0,2);
    GLCD_puttext_P(PSTR("Częstotliwość [Hz]:"));
    timer_init();
    sei();
    while(1)
    {
        ATOMIC_BLOCK(ATOMIC_FORCEON)
```

```

{
ok=isOk;
ts=timestamp;
pr=period;
}
if(ok==0)
{
ts=0; period=0;
}
GLCD_goto(0,1);
if(ts) utoa(100*ts/pr, bufor, 10); else bufor[0]=0;
len=strlen(bufor);
memmove(&bufor[6-len], bufor, len+1);
for(uint8_t i=0;i<(6-len);i++) bufor[i]=' ';
GLCD_puttext(bufor);
GLCD_goto(0,3);
if(pr) utoa(F_CPU/64/pr, bufor, 10); else bufor[0]=0;
len=strlen(bufor);
memmove(&bufor[6-len], bufor, len+1);
for(uint8_t i=0;i<(6-len);i++) bufor[i]=' ';
GLCD_puttext(bufor);
}
}

```

W powyższym kodzie warto zwrócić uwagę na sposób pobierania zmiennych `timestamp`, `period` i `isOk`. Ponieważ w trakcie dostępu do tych zmiennych może wystąpić przerwanie zmieniające ich wartość, zmienne te są w sposób atomowy przepisywane do zmiennych lokalnych, na których prowadzone są dalsze obliczenia. Dzięki temu mamy gwarancję, że zmienne mają prawidłowe wyniki pomiaru (zmienna `isOk` jest różna od zera). Bez atomowego dostępu do tych zmiennych przerwanie, które by wystąpiło pomiędzy kolejnymi instrukcjami realizującymi dostęp do 16-bitowych zmiennych `timestamp` i `period`, powodowałoby ich zmianę w trakcie dostępu do nich, a w efekcie nieprawidłowe wskazania parametrów mierzonyego przebiegu.

## Realizacja RTC przy pomocy timera

Układy zegarowe RTC (ang. *Real-Time Clock*) można realizować przy pomocy specjalistycznych zewnętrznych układów scalonych. Jednak takie rozwiązanie wiąże się z kosztami i zajęciem cennego miejsca na płytce. Większość mikrokontrolerów AVR oferuje prostsze rozwiązanie — wbudowane, sprzętowe funkcje umożliwiające łatwą realizację zegara RTC. Do realizacji tego celu z elementów zewnętrznych potrzebny jest kvarc zegarkowy (32768 Hz) podłączony pod wejścia TOSC1 i TOSC2, *timer* oraz kilka linii kodu programu. W przypadku braku zewnętrznego zasilania podtrzymywanie funkcji *timera* zapewnia dodatkowa bateria lub kondensator o dużej pojemności. Dzięki wprowadzeniu procesora w tryb uśpienia i wybudzaniu go tylko na krótko po przepelenieniu licznika *timera* cały układ zużywa znikome ilości prądu.

Do realizacji funkcji RTC w mikrokontrolerach AVR zazwyczaj przeznaczony jest *timer0* lub *timer2*, ze względu na możliwość taktowania go w sposób asynchroniczny, przy pomocy zewnętrznego kwarca zegarkowego.



Wskazówka

Kwarce zegarkowe 32 768 Hz zazwyczaj przystosowane są do temperatury pracy 20 – 25°C, ich dokładność jest też zazwyczaj gorsza niż innych kwarców (ok. 20 ppm).

Taktowanie asynchroniczne timera z własnego generatora umożliwia wyłączenie wszystkich innych bloków funkcyjnych procesora, łącznie z oscylatorem, zapewniającym źródło sygnału zegarowego dla rdzenia procesora, co prowadzi do znacznego obniżenia zużycia energii, nawet do poziomu 1 – 2  $\mu$ A, czyli porównywalnego z prądem zużywanym przez dedykowane scalone układy RTC.

Realizację układu RTC należy podzielić na dwa etapy:

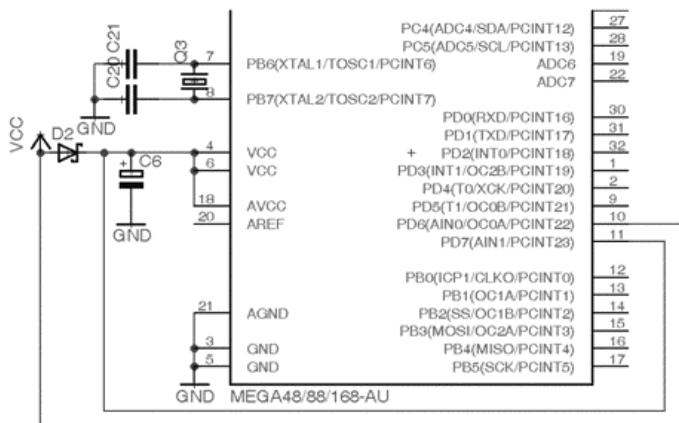
- ◆ Etap sprzętowy — musimy zapewnić podtrzymywanie zasilania procesora w sytuacji awarii głównego zasilania, co zapewnia nieprzerwane działanie *timera*. Musimy dostarczyć sygnał informujący procesor o takim stanie oraz skonstruować układ w taki sposób, aby inne komponenty, oprócz procesora, nie były zasilane ze źródła awaryjnego, co doprowadziłoby do jego szybkiego wyczerpania.
- ◆ Etap programowy — zapewnienie niskiego poboru energii przez procesor w sytuacji przejścia na zasilanie awaryjne.

## Realizacja sprzętowa

Jak wspomniano, do realizacji sprzętowej RTC wykorzystuje się możliwość asynchronicznego taktowania *timera* ze źródła sygnału zegarowego o niskiej częstotliwości. Zazwyczaj w tym celu wykorzystuje się kwarce zegarkowe o częstotliwości 32 768 Hz. Kwart należał podłączyć do wejścia TOSC1 i TOSC2 procesora. Czasami wymagane są dodatkowe kondensatory (o pojemności 8 – 18 pF). Ponieważ zegar RTC nie powinien przestać działać po zaniku napięcia zasilającego, należy układ tak zaprojektować, aby zapewnić stały dopływ energii do procesora. W tym celu można wykorzystać różne układy, lecz najprościej jest wykorzystać baterię lub specjalny kondensator o dużej pojemności, tzw. *supercap*. Jego zastosowanie zamiast baterii ma kilka istotnych zalet. Przede wszystkim taki kondensator, w przeciwieństwie do baterii lub akumulatora, bardzo powoli ulega degradacji, umożliwia wielokrotne ładowanie, a przede wszystkim nie jest wrażliwy na przeladowanie. Dzięki temu układ zasilania awaryjnego w tym przypadku sprawdza się do diody i kondensatora (rysunek 17.10).

Pozostaje jeszcze problem wykrycia spadku napięcia zasilającego. Jest to niezbędne, gdyż procesor musi wiedzieć o tym, że jest zasilany z baterii, dzięki czemu może wyłączyć zbędne podsystemy i przejść w tryb uśpienia. Układ detekcji braku napięcia zasilającego można zrobić na wiele sposobów. Jednym z prostszych jest wykorzystanie do tego celu komparatora analogowego, który znajduje się w każdym procesorze AVR. W pokazanym układzie potencjał wejścia AIN0 komparatora jest o ok. 0,2 V wyższy niż potencjał wejścia AIN1, w efekcie na wyjściu komparatora panuje stan wysoki. Odłączenie napięcia zasilającego powoduje obniżanie się potencjału na wejściu AIN0 komparatora, podczas gdy potencjał wejścia AIN1 dzięki energii z superkondensatora praktycznie się nie zmienia. Komparator przełącza się i na wyjściu panuje 0 logiczne.

**Rysunek 17.10.**  
Układ zasilania awaryjnego zrealizowany przy pomocy superkondensatora



Taka zmiana stanu komparatora może wygenerować przerwanie, którego obsługa powinna zadbać o odpowiednie zarządzanie energią. Po powrocie napięcia zasilającego komparator ponownie zmienia stan na 1, a kondensator ładuje się przez diodę D2. Jako diodę D2 należy wykorzystać dowolną diodę o możliwie małym napięciu przewodzenia, najlepsze są diody Schottkiego. Układ detekcji awarii zasilania można rozwiązać na wiele sposobów, w zależności od tego, jak zbudowany jest układ zasilacza. Obecnie wiele scalonych regulatorów napięcia dysponuje wyjściem informującym o awarii zasilania; wyjście to może zostać bezpośrednio podłączone do pinu  $\text{IO}$  procesora (najlepiej, jeśli zostanie podłączone do pinu, którego zmiana może generować przerwanie).

Należy pamiętać, że kwarce zegarkowe mają po włączeniu zasilania bardzo długi okres stabilizacji drgań, wynoszący nawet 1 sekundę. Ponieważ w tym czasie oscylacje nie są stabilne, nie należy wchodzić w tryb *power save*. Można to bezpiecznie wykonać dopiero po ustabilizowaniu się drgań кварca.

## Realizacja programowa

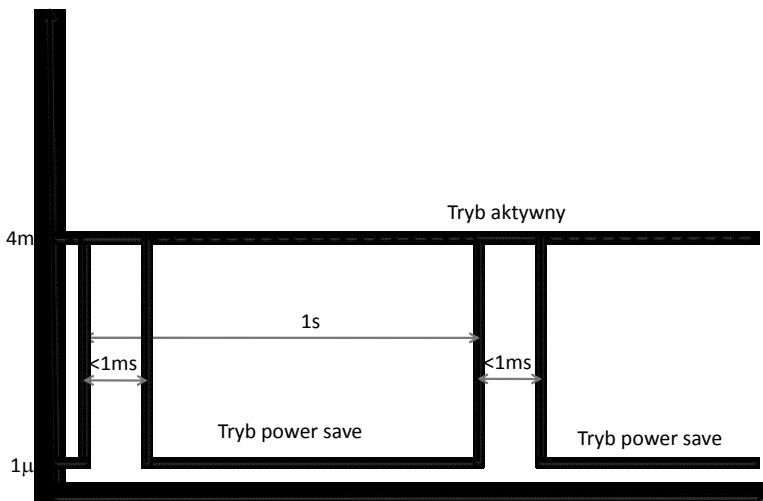
W sytuacji normalnego zasilania procesora realizacja RTC jest prosta. Wykorzystujemy *timer2* do odliczania czasu, po wystąpieniu nadmiaru generowane jest przerwanie, powodujące zwiększenie licznika sekund. Przy zasilaniu awaryjnym program wygląda podobnie, z tym że pomiędzy przerwaniami procesor wchodzi w tryb głębokiego uśpienia, co redukuje pobór mocy (rysunek 17.11).

Dzięki temu, że procesor jest wybudzany tylko na chwilę, średnie zużycie prądu dla przykłdu z rysunku 17.11 wynosi  $1\text{s} \cdot 1\mu\text{A} + 1\text{ms} \cdot 4\text{mA} = 5\mu\text{As}$ .

Asynchroniczną pracę *timera* kontroluje bit AS2 rejestru ASSR. Jego ustawienie powoduje uaktywnienie pinów TOSC1 i TOSC2, do których podłączony jest kwarc zegarkowy. Układ generatora kwarcowego dostosowany jest do kwarcu o częstotliwości 32 768 Hz. Wygenerowany przebieg zegarowy trafia do układu preskalera, który umożliwia jego podzielenie przez 1, 8, 32, 64, 128, 256, 1024 lub całkowite zatrzymanie (jednak nie zatrzymuje to samego oscylatora). Po zmianie zegara taktującego na zewnętrzny resztę konfiguracji przeprowadza się tak, jakby *timer* był taktowany wewnętrznym przebiegiem

**Rysunek 17.11.**

Procesor normalnie przebywający w stanie uśpienia pobiera znikome ilości prądu, wybudzany jest na krótko co sekundę w celu uaktualnienia licznika sekund



zegarowym. Ponieważ procedura obsługi przerwania przepelnienia *timera* powinna być wywoływaną co sekundę, należy przebieg zegarowy podzielić tak, aby dokładnie co 1 sekundę następował przepelnięcie licznika. Ponieważ licznik jest 8-bitowy, powoduje on podzielenie zegara przez 256, w efekcie uzyskujemy wartość 128 i taką właśnie wartość preskalera należy wybrać.

Po wybraniu zewnętrznego oscylatora należy w specjalny sposób operować rejestrami *timera*. Układ *timera* jest taktowany asynchronicznie, przy pomocy bardzo wolnego zegara (w stosunku do zegara procesora). Dlatego w AVR wprowadzono specjalne rejesty stanowiące bufor pomiędzy danymi zapisywanyimi przez procesor a rejestrami *timera*. Zapisanie któregokolwiek rejestru *timera* w trybie asynchronicznym powoduje ustawienie odpowiadającej mu flagi rejestru ASSR (ang. *Asynchronous Status Register*). Do czasu, aż flaga nie zostanie skasowana przez układy synchronizujące procesora, nie można wpisywać nowej wartości do danego rejestru.

Po tej poręji teorii przejdźmy do praktycznej realizacji RTC. Najpierw zostanie stworzona procedura obsługi przerwania *timera*, której zadaniem będzie inkrementacja zmiennej przechowującej liczbę sekund:

```
volatile uint32_t seconds;      //Liczni sekund

ISR(TIMER2_OVF_vect)
{
    TCCR2B=_BV(CS22) | _BV(CS20);
    seconds++;
}
```

Oprócz zwiększenia licznika sekund procedura obsługi przerwania dokonuje zapisu do rejestru TCCR2B wartości, która już wcześniej się tam znajdowała. Powód tego działania zostanie wyjaśniony za chwilę. Teraz czas na funkcję inicującą *timer* w trybie asynchronicznym:

```

void initRTC()
{
    ASSR|= BV(AS2);           //Timer/Counter2 jest taktowany asynchronicznie
    TCNT2=0x00;
    TCCR2B=_BV(CS22) | _BV(CS20); //preskaler 128, nadmiar wystąpi co 1s
    while(ASSR&0x07);          //Poczekaj na uaktualnienie T/C2
    TIMSK2|= _BV(TOIE2);       //odblokuj przerwanie Timer/Counter2 Overflow
}

```

Jak widać, pierwszym etapem jest przełączenie *timera* w tryb asynchroniczny. Trzeba to zrobić jako pierwsze, gdyż przełączenie źródła zegara wiąże się z utratą zawartości rejestrów *timera*; w efekcie wcześniejsza konfiguracja jest tracona. Po zapisaniu konfiguracji preskalera sprawdzany jest stan rejestru ASSR, odblokowanie przerwań następuje dopiero po skasowaniu flag rejestru ASSR. W celu zaoszczędzenia energii pomiędzy kolejnymi przerwaniami *timera* procesor powinien znajdować się w stanie uśpienia:

```

int main()
{
    initRTC();
    set_sleep_mode(SLEEP_MODE_PWR_SAVE);
    sei();

    while(1)
    {
        while(ASSR&0x07);
        sleep_mode();
    }
}

```

W funkcji `main` najpierw konfigurowane są bity odpowiedzialne za wybór trybu uśpienia. W trybie `SLEEP_MODE_PWR_SAVE` taktowanie prawie wszystkich podsystemów jest wyłączone, włączony pozostaje generator kwarcowy taktujący asynchronicznie *timer*. Tu pojawia się pewien kruczek związany z asynchronicznym taktowaniem *timera*. Po zgłoszeniu przerwania przepelnienia *timera* nie można od razu wprowadzić procesora w stan uśpienia, gdyż od razu zgłoszone zostały kolejne przerwanie. Przed uśpieniem procesora musi minąć przynajmniej jeden okres zegara taktującego *timer*, a więc dla częstotliwości zegara 32 768 Hz musi minąć co najmniej 30 µs. Możemy poczekać w pętli przez co najmniej taki odcinek czasu, co jednak wiąże się z wydłużeniem czasu wybudzenia procesora, a co za tym idzie, zwiększym poborem energii. Można też zastosować pewien trick — po wpisaniu jakiejś wartości do rejestru *timera* odpowiedni bit rejestru ASSR zostaje ustawiony do czasu, aż nowa wartość zostanie przepisana do właściwego rejestru. Po tym czasie mamy gwarancję, że minął co najmniej jeden okres zegara taktującego *timer*, możemy więc bezpiecznie uśpić procesor. Wartość wpisywana do rejestru *timera* powinna być neutralna, tak żeby nie zakłócić zliczania czasu. W powyższym przykładzie wykorzystano rejestr TCCR2B, wpisując do niego w procedurze obsługi przerwania przepelnienia *timera* wartość preskalera (128). W efekcie tuż po wybudzeniu procesora ustawiana jest odpowiednia flaga rejestru ASSR, procesor wykonuje kod związany z inkrementacją licznika sekund i wraca do funkcji `main`, do instrukcji następnej po instrukcji uśpienia procesora. W trakcie realizacji tych zadań powoli upływa czas, jaki musi upływać do ponownego uśpienia procesora. Upłynięcie tego czasu sygnaлизowane jest wyzerowaniem bitów rejestru ASSR, co jest testowane przed ponownym wprowadzeniem procesora w stan uśpienia.

# Rozdział 18.

## Obsługa

# wyświetlaczy LCD

Do tej pory do wyświetlania informacji dla użytkownika stosowane były wyświetlacze 7-segmentowe. Ich główną zaletami są prostota podłączenia do mikrokontrolera oraz niskie koszty. Jednak mogą one wyświetlać tylko ograniczoną ilość informacji. Ze względu na malejące ceny, zwiększącą się dostępność oraz niski pobór energii coraz większym zainteresowaniem cieszą się wyświetlacze LCD. Zawierają one zazwyczaj scalone kontrolery, stąd też procesor nie steruje bezpośrednio matrycą LCD, ale komunikuje się z wyspecjalizowanym sterownikiem, który realizuje jego polecenia. Stąd też wykorzystanie tego typu wyświetlaczy wymaga znajomości typu wykorzystanego w nich sterownika. Wyświetlacze LCD mogą być łączone z mikrokontrolerem przy pomocy różnych magistrali — najczęściej są to magistrale SPI lub równolegle (w wersji 4-, 8- i 16-bitowej). Same wyświetlacze ze względu na budowę i możliwości możemy podzielić na dwie grupy:

- ◆ Wyświetlacze alfanumeryczne — umożliwiają one wyświetlanie cyfr, liter i innych symboli. Jednak w tym typie nie mamy możliwości sterowania poszczególnymi pikselami.
- ◆ Wyświetlacze graficzne — w tym typie sterować możemy stanem każdego piksela.

Aby wyświetlana na LCD zawartość była widoczna, niezbędne jest podświetlenie. Kupując wyświetlacz, musimy zwrócić uwagę, czy jest on w jakieś podświetlenie wyposażony (można dostać wyświetlacze LCD bez podświetlenia). Oczywiście, lepiej jest wybrać wyświetlacz, który ma wbudowane podświetlenie. Samo podświetlenie może być zrealizowane na kilka sposobów. W starszych modelach spotyka się podświetlenie zrealizowane w oparciu o lampa CCFL (ang. *Cold Cathode Fluorescent Lamp*). Wyświetlacz takie są zwykle większe, cięższe, ale przede wszystkim wymagają do sterowania lampą specjalnej przetwornicy. Jej zadaniem jest generacja wysokiego napięcia niezbędnego do zaświecenia się lampy, a następnie utrzymywanie go w granicach zapewniających jej prawidłowe działanie. Wyświetlacz tego typu są więc kłopotliwe w użytkowaniu, musimy wydać czasami spore pieniądze na odpowiednią przetwornicę, w dodatku

trwałość takiej lampy jest niższa niż diod LED. Występują także problemy ze sterowaniem jasnością podświetlenia. W nowoczesnych wyświetlaczach LCD stosuje się praktycznie wyłącznie podświetlenie zrealizowane za pomocą diod LED. Nie wymagają one żadnych wyrafinowanych przetwornic do zasilania, zwykle zadowalają się napięciem w granicach 3 – 12 V, mają też znacznie większą trwałość. Jasność takiego podświetlenia z łatwością można sterować, wykorzystując technikę PWM. Podświetlenie oparte na diodach LED może być zrealizowane w różnym kolorze — białe, żółte, zielone, niebieskie, czerwone. Kolor możemy sobie więc dobrać wg własnych preferencji.



Wskazówka

Jeżeli możesz, staraj się wybierać wyświetlacze LCD z podświetleniem zrealizowanym ma diodach LED.

Kolejną istotną rzeczą jest regulacja kontrastu. Zwykle graficzne kolorowe LCD z wbudowanymi sterownikami automatycznie regulują kontrast, dodatkowo sterownik ma zaimplementowane rejesty umożliwiające wyłącznie programową zmianę kontrastu. W przypadku prostszych sterowników, występujących najczęściej w wyświetlaczach monochromatycznych (zarówno alfanumerycznych, jak i graficznych), kontrast reguluje się, podając odpowiednie napięcie na jeden z pinów wyświetlacza. Jest to niezwykle istotne, gdyż bez właściwego napięcia regulującego kontrast na wyświetlaczu nie zobaczymy. Możemy w takiej sytuacji odnieść błędne wrażenie, że wyświetlacz nie działa. Z kolei jeśli kontrast będzie zbyt duży, wszystkie piksele będą włączone, co też może sugerować błędne działanie LCD.



Wskazówka

Jeśli po prawidłowym połączeniu wyświetlacza LCD nic na nim nie widzisz albo wszystkie piksele są włączone, to jedną z częstszych przyczyn jest nieprawidłowe napięcie na pinie regulującym kontrast.

## Obsługa wyświetlaczy alfanumerycznych

Wyświetlacz alfanumeryczne zbudowane są z matryc pikseli, zwykle o wielkości  $5 \times 8$  pikseli, pogrupowanych razem. W tym typie wyświetlacza sterować możemy zawartością tak utworzonej matrycy znakowej, jednak nie ma możliwości sterowania poszczególnymi pikselami. Najczęściej spotykane wyświetlacze tego typu mogą wyświetlać  $1 \times 8$ ,  $1 \times 16$ ,  $2 \times 16$ ,  $2 \times 20$ ,  $4 \times 16$  znaków, gdzie pierwszy wymiar określa liczbę wyświetlanych wierszy, a drugi liczbę znaków w wierszu. Ponieważ nie sterujemy poszczególnymi pikselami, do kontrolera wysyłamy tylko kod znaku, który ma zostać wyświetlony na danej pozycji. Kontroler na podstawie kodu znaku generuje adres do specjalnego obszaru pamięci, tzw. tablicy znaków, w której znajduje się bajtowa reprezentacja poszczególnych pikseli tworzących znak. Dla matrycy  $5 \times 8$  ma ona 40 bitów. Tablica ta jest umieszczona w pamięci ROM kontrolera LCD, nie możemy jej więc zmieniać. Zwykle jednak kontrolery udostępniają możliwość definicji kilku własnych znaków, dzięki czemu możemy zdefiniować np. znaki charakterystyczne dla języka pol-

skiego. Takie podejście do generowania obrazu ma podstawową zaletę, jaką jest prostota i związane z tym niewielkie zapotrzebowanie na pamięć RAM. Do przechowania np. zawartości 8-znakowego wyświetlacza LCD wystarczy zaledwie 8 bajtów pamięci. Ma to szczególne znaczenie w prostych mikrokontrolerach, z niewielką ilością pamięci RAM. Najpopularniejszym kontrolerem stosowanym w wyświetlaczach alfanumerycznych jest HD44780; nawet jeśli w posiadanym wyświetlaczu jest inny kontroler, to mamy dużą szansę, że jest on z nim kompatybilny. Typowy rozkład sygnałów modułów opartych na tym kontrolerze pokazano w tabeli 18.1.

**Wskazówka**

Przed podłączeniem modułu zawsze upewnij się, czy podany rozkład odpowiada temu modułowi. W szczególności sprawdź rozmieszczenie linii zasilających. Ich nie-właściwe podłączenie grozi uszkodzeniem modułu!

**Tabela 18.1.** Typowy układ wyprowadzeń modułów alfanumerycznych LCD opartych o kontroler HD44780

Pin	Nazwa	Funkcja
1	<i>GND</i>	Masa
2	<i>Vcc</i>	Zasilanie (typowo 5 V)
3	<i>Vee</i>	Regulacja kontrastu
4	<i>RS</i>	Zapis komendy
5	<i>RW</i>	Strob odczyt/zapis
6	<i>E</i>	Wybór układu
7	<i>D0</i>	Bit danych 0
8	<i>D1</i>	Bit danych 1
9	<i>D2</i>	Bit danych 2
10	<i>D3</i>	Bit danych 3
11	<i>D4</i>	Bit danych 4
12	<i>D5</i>	Bit danych 5
13	<i>D6</i>	Bit danych 6
14	<i>D7</i>	Bit danych 7
15	<i>A</i>	Podświetlenie — anoda
16	<i>K</i>	Podświetlenie — katoda

Zwykle piny 15 i 16 sterujące podświetleniem umieszczone są osobno lub też przed pinem 1. Stąd też zawsze należy je dokładnie przed podłączeniem modułu zidentyfikować. Do pinów tych należy podłączyć napięcie wykorzystane do podświetlenia modułu. Przyłączone napięcie zależy od typu zastosowanego podświetlenia — CCFL, folia elektroluminescencyjna lub diody LED. W dalszej części zajmować się będziemy modułem z podświetleniem LED. W nocy katalogowej posiadanego modułu znajdują się szczegółowe informacje dotyczące zasilania podświetlenia. Zwykle w tym celu wymagane jest napięcie 5 V, a prąd podświetlenia wynosi 20 – 50 mA.

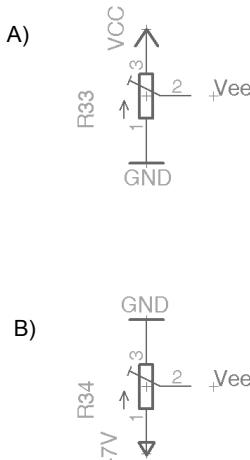
**Wskazówka**

Ponieważ zasilamy diodę LED, wymagane jest dołączenie szeregowego rezystora. Czasami rezistor taki wbudowany jest w moduł LCD.

Kolejną istotną rzeczą jest podłączenie właściwego napięcia regulującego kontrast wyświetlacza. Dla większości modułów LCD mieści się ono w przedziale  $GND - Vcc$ , dla modułów o rozszerzonym zakresie temperatury jest ono w przedziale  $0 - -7\text{ V}$  — rysunek 18.1.

**Rysunek 18.1.**

Podłączenie napięcia regulującego kontrast.  
A) moduły klasyczne,  
B) moduły o rozszerzonym zakresie temperatur



Napięcie ujemne można uzyskać z prostej pompy ładunkowej sterowanej przebiegiem PWM z procesora. Pobór prądu z tego wejścia jest znikomo mały. Alternatywnie można użyć układu ILC7660, mogącego generować napięcia ujemne.

Kolejną decyzją, jaką musimy podjąć, jest sposób podłączenia modułu LCD z mikrokontrolerem. Mamy do wyboru dwie opcje — albo podłączamy wszystkie 8 linii danych ( $D0 - D7$ ), albo tylko 4 najstarsze ( $D4 - D7$ ). Podłączając tylko 4 linie danych, zmniejszamy liczbę niezbędnych połączeń modułu z mikrokontrolerem oraz liczbę wykorzystanych linii  $I/O$ . Ze względu na znikomą ilość danych przesyłanych z/do modułu, wynikające z posiadania tylko 4 linii danych zwolnienie transmisji jest bez znaczenia. **W trybie 4-bitowym najpierw transferowana jest starsza, a potem młodsza tetrad.** Niewykorzystane linie  $D0 - D3$  można pozostawić niepodłączone, większość modułów wewnętrznie podciąga niepodłączone linie poprzez rezystor do  $Vcc$ .



Ponieważ linie  $D0 - D7$  są dwukierunkowe, musimy pamiętać, aby podczas operacji odczytu z LCD odpowiednie piny mikrokontrolera były ustawione jako wejścia. W przeciwnym przypadku dojdzie do konfliktu.

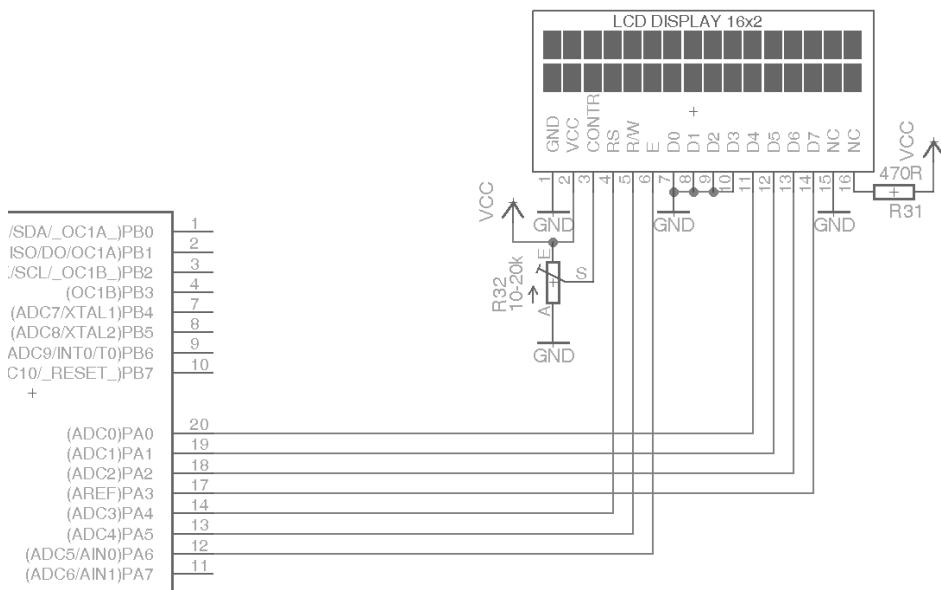
Kolejnym sygnałem, co do którego musimy podjąć decyzję, jest sygnał  $RW$  określający typ operacji (odczyt/zapis). Ponieważ zwykle zapisujemy dane do LCD, właściwie nie zachodzi konieczność ich odczytywania; możemy ten sygnał na stałe połączyć z masą, oszczędzając w ten sposób jedną linię  $I/O$  procesora. Takie rozwiązanie ma jednak poważną wadę — nie jest możliwe w tej sytuacji odczytywanie sygnału zajętości modułu (ang. *Busy Flag*). Powoduje to, że pomiędzy kolejnymi zapisami do modułu musimy stosować opóźnienia dobrane na najwolniejszą możliwą okoliczność. W efekcie cała transmisja jest wolniejsza, niż mogłaby być.



Wskazówka

Korzystając z tych modułów, nie musimy pisać własnych procedur ich obsługi. Instalując środowisko WinAVR (lub nowy toolchain firmy Atmel), w katalogu `doc\avr-libc\examples\sdtiodemo` znajdziemy gotowe funkcje obsługi HD44780.

Możemy też skorzystać z licznych przykładów dostępnych w Internecie. **W dalszej części wykorzystane zostaną funkcje z pakietu WinAVR napisane przez Joerga Wunscha.** Pokazane przykłady zakładają, że wyświetlacz jest połączony z mikrokontrolerem tak jak na rysunku 18.2.



Rysunek 18.2. Połączenie LCD z mikrokontrolerem ATTINY461

Zanim coś wyświetlimy na LCD, musimy poznać, w jaki sposób generowane są znaki. Sterownik LCD zawiera w sobie, niezależnie od typu użytej matrycy, 80 bajtów pamięci DDRAM (ang. *Display Data RAM*). Jeśli zastosowana matryca wyświetla mniejszą liczbę znaków, to część komórek pamięci jest nieużywana. Możemy je wykorzystać do innych celów — jako zwykłą pamięć RAM. Możemy też ją wykorzystać do przesuwania napisów, które nie mieszczą się na wyświetlaczu. Aby umieścić znak na właściwej pozycji, niezbędne jest poznanie organizacji pamięci kontrolera. W zależności od użytej matrycy pamięć ta jest zorganizowana następująco:

- ◆ Dla matryc  $16 \times 1$  — pamięć podzielona jest na dwa obszary, każdy zawierający po 8 znaków. Obszar pierwszy (od lewej) ma adresy 0x00 do 0x07, a obszar drugi (prawy) adresy 0x40 do 0x47. Stąd też wyświetlając tekst przekraczający granicę obszarów, należy zadbać, aby poszczególne litery trafiły pod właściwe adresy.
- ◆ Dla matryc  $20 \times 1$  — adresy wynoszą 0x00 do 0x13.
- ◆ Dla matryc  $16 \times 2$  — adres pierwszej linii zaczyna się od 0x00 do 0x0f, adres drugiej linii od 0x40 do 0x4f.

- ◆ Dla matryc  $20 \times 2$  — adresy pierwszej linii to 0x00 do 0x13, a drugiej 0x40 do 0x53.
- ◆ Dla matryc  $40 \times 2$  — adresy pierwszej linii to 0x00 do 0x27, drugiej linii 0x40 do 0x67.
- ◆ Dla matryc  $20 \times 4$  — adresy pierwszej linii to 0x00 do 0x13, drugiej 0x40 do 0x53, trzeciej 0x14 do 0x27, a czwartej 0x54 do 0x67.

Jak widać, w niektórych typach wyświetlaczy po każdej linii zostaje kilka – kilkanaście bajtów, których zawartość nie jest wyświetlana. Ma to pewną zaletę — możemy do pamięci modułu wpisywać teksty dłuższe, niż są wyświetlane na ekranie, co ułatwia przewijanie tekstu. Niestety, ma to też pewną wadę — funkcje wyświetlające coś na ekranie muszą wiedzieć, jaką organizację ma matryca, w przeciwnym przypadku nie ma możliwości prawidłowego wyliczenia adresu komórki, do której ma odbyć się zapis.



**Wskazówka** W dalszej części zostanie wykorzystany moduł LCD o organizacji  $16 \times 2$  (ale z łatwością można wykorzystać dowolny inny).

Zanim będzie można coś wyświetlić na ekranie, moduł musi zostać zainicjowany. W trakcie inicjalizacji ustawiane są podstawowe parametry pracy sterownika LCD — organizacja matrycy (liczba linii), wielkość czcionki ( $5 \times 8$  lub  $8 \times 11$  pikseli). Wielkość czcionki zależy od typu podłączonej matrycy LCD. Dla konkretnego modułu LCD tylko jedna z tych wartości będzie poprawna. Oprócz inicjalizacji modułu należy także poprawnie ustawić stan pinów *IO* wykorzystywanych do podłączenia modułu LCD.

Większość pracy związanego z inicjalizacją modułu LCD, odbieraniem i wysyłaniem do niego danych wykonuje biblioteka Joerga Wunscha. Dostarcza ona niskopoziomowych funkcji umożliwiających komunikację ze sterownikiem hd44780 i innymi kompatybilnymi z nim sterownikami. Wystarczy, że do aplikacji, w której planujemy wykorzystać taki moduł, dołączymy plik *hd44780.c* oraz plik nagłówkowy *hd44780.h*. Oba pliki zakładają, że istnieje plik nagłówkowy *defines.h*, zawierający podstawowe definicje związane ze sposobem podłączenia modułu LCD z mikrokontrolerem. Na początku należy określić, które linie *IO* zostały wykorzystane do podłączenia modułu:

```
#define HD44780_RS A. 4
#define HD44780_RW A. 5
#define HD44780_E A. 6
#define HD44780_D4 A. 0
```

W powyższym przykładzie do kolejnych linii portu A (*A4 – A6*) podłączone zostały sygnały *RS*, *RW* i *E* modułu, natomiast do linii *A0 – A3* podłączone zostały linie *D4 – D7* modułu LCD. Ważne jest, aby linie danych łączyć z pinami *IO* w kolejności rosnącej. Biblioteka ta wykorzystuje do transmisji danych pomiędzy modułem a mikrokontrolerem tylko 4 linie danych — dzięki temu możemy zmniejszyć liczbę połączeń, a zwolnione linie *IO* wykorzystać do innych celów. W pliku tym występuje jeszcze jedna definicja:

```
#define USE_BUSY_BIT 1
```

Określa ona, że biblioteka używać będzie flagi *Busy* modułu LCD — umożliwia to uzyskanie optymalnej prędkości współdziałania z modułem. Zakończenie realizacji

bieżącej operacji moduł sygnalizuje zmianę stanu tej flagi, co umożliwia niezwłoczne wysłanie kolejnej komendy. Jeśli nie używamy flagi *Busy*, program musi generować makrosymalne zalecane w nocie katalogowej opóźnienia, w efekcie cała transmisja zwalnia. Po zdefiniowaniu używanych pinów w pliku *defines.h* możemy rozpoczęć korzystanie z biblioteki.



W pliku *hd44780.h* udostępnione są prototypy funkcji realizujących współpracę z modułem LCD.

## Funkcje biblioteczne

Podstawę modułu stanowią dwie funkcje umożliwiające odpowiednio wysłanie danych do modułu oraz ich odczytanie:

```
void hd44780_outbyte(uint8_t b, uint8_t rs);
uint8_t hd44780_inbyte(uint8_t rs);
```

Obie funkcje wysyłają/odczytują 8-bitową daną; w przypadku wykorzystania tylko 4-bitowej szyny danych wysyłanie i odczytywanie danych odbywa się w dwóch etapach. Dodatkowo zmienna *rs* określa tryb dostępu. W przypadku wysyłania danych do modułu *rs* = 0 powoduje, że dane zostaną wysłane do rejestru komend, jeśli *rs*=1, dane trafiają do rejestru danych. Przy odczycie jest podobnie, z tym że dla *rs* = 0 odczytywana jest flaga *Busy* i rejestr *AC* (ang. *Address Counter*).

Kolejna funkcja realizuje oczekiwanie na gotowość modułu:

```
void hd44780_wait_ready(bool islong);
```

W przypadku kiedy w pliku *defines.h* symbol *USE\_BUSY\_FLAG* jest równy 1, parametr *islong* jest ignorowany, a optymalizator kompilatora usuwa go, gdyż nie jest on wykorzystywany przez funkcję. Kiedy *USE\_BUSY\_FLAG* jest różne od 1, parametr ten określa długość opóźnienia. Jeśli jest różny od 0, opóźnienie wynosi 1,52 ms, jeśli jest równy 0, opóźnienie wynosi 37 µs. Ponieważ lepiej jest używać flagi *Busy*, możemy ten parametr ignorować.

Powyższe trzy funkcje wystarczą do komunikacji z modułem, dla wygody zdefiniowano jednak kolejne. Funkcje *hd44780\_outcmd(n)* i *hd44780\_incmd()* umożliwiają odpowiednio wysłanie polecenia do rejestru komend kontrolera lub odczytanie bitu *Busy* i licznika *AC*. Dwie kolejne — *hd44780\_outdata(n)* oraz *hd44780\_indata()* — umożliwiają zapis i odczytanie rejestru danych.

Oprócz funkcji realizujących komunikację z modułem LCD zdefiniowano także funkcję przeprowadzającą podstawową inicjalizację modułu, *hd44780\_init(void)*, oraz funkcję *hd44780\_powerdown(void)*, ustawiającą wszystkie linie użyte do komunikacji w stan 0, co umożliwia bezpieczne odłączenie zasilania od modułu LCD. **W przypadku pozostawienia którejś z wyżej wymienionych linii w stanie 1 przy odłączonym zasilaniu modułu stanowi ona jego zasilanie. W efekcie mimo wyłączenia moduł może pobierać znaczny prąd.**

Szerszego omówienia wymaga funkcja `hd44780_outcmd(cmd)`, która umożliwia wysłanie polecenia do rejestru poleceń sterownika LCD. Dla wygody zdefiniowane zostały różne komendy ułatwiające współpracę z modułem LCD. Komendy te przekazywane są jako argument wywołania funkcji `hd44780_outcmd`, np.:

```
hd44780_outcmd(HD44780_CLR);
```

powoduje wysłanie do sterownika polecenia `HD44780_CLR`, czyszczącego zawartość pamięci DDRAM kontrolera.

## Komendy

Funkcja `hd44780_outcmd` przyjmuje jako argument predefiniowane makrodefinicje, znacznie ułatwiające współpracę z kontrolerem modułu LCD. Poniżej zostaną one pokróćce omówione.

### **HD44780\_SHIFT(shift, right)**

Makrodefinicja ta powoduje przesunięcie pozycji wyświetlanego tekstu (`shift = 1`) o jedną pozycję w lewo (`argument right = 0`) lub w prawo (`argument right = 1`). W przypadku kiedy `shift = 0`, przesuwany w lewo lub prawo jest tylko kurSOR. Wysyłając wielokrotnie to polecenie, uzyskujemy kolejne przesunięcia o jedną pozycję w lewo lub prawo. Polecenie to służy głównie do przesuwania tekstu. Jednak ze względu na dużą bezwładność tekstowych modułów LCD nie należy przesuwać liter częściej niż ok. raz na sekundę — częstsze przesuwanie daje bardzo nieprzyjemne efekty wizualne. Każdy bufor linii jest „zawijany”, to znaczy, że po ostatnim znaku z prawej wyświetlany jest pierwszy znak z lewej i odwrotnie. Należy pamiętać, że długość bufora jest stała dla danego modelu matrycy. Np. dla modułów  $16 \times 2$  bufor jednej linii wynosi dokładnie 40 bajtów, z czego 16 jest jednorazowo wyświetlanych. Poniższy kod cyklicznie przewija wyświetlany tekst w lewo:

```
lcd_puttext_P(PSTR("Test bufora i przewijania tekstu.\nTen tekst przewija sie
➥w lewo"));
while(1)
{
    hd44780_outcmd(HD44780_SHIFT(1,0));
    _delay_ms(700);
}
```

### **HD44780\_FNSET(if8bit, twoline, font5x10)**

Polecenie to wybiera typ interfejsu — 8-bitowy, jeśli parametr `if8bit` jest równy 1, lub 4-bitowy, jeśli jest równy 0. Typ interfejsu musi odpowiadać sposobowi podłączenia modułu z mikrokontrolerem. Parametr `twoline` określa, czy podłączona matryca LCD ma jedną linię znaków, czy dwie. Matryce posiadające 4 linie tekstu traktowane są przez kontroler jako matryce dwuliniowe — parametr `twoline` powinien mieć wartość 1. Jeśli dla matrycy zawierającej dwie lub więcej linii ustawimy go na 0, to wyświetlana będzie tylko jedna linia (w przypadku matryc dwuliniowych górna), a w przypadku matryc 4-liniowych wyświetlane będą dwie linie, rozdzielone linią pustą.

Ostatni parametr określa typ używanego generatora znaków. Powinien on odpowiadać użytcej w module matrycy, zwykle więc wynosi 0.

### **HD44780\_DISPCTL(*disp*, *cursor*, *blink*)**

Ustawia różne parametry pracy matrycy. Jeśli parametr *disp* jest równy 1, to matryca jest włączona (wyświetla zawartość DDRAM), jeśli jest on równy 0, to matryca jest wyłączona, lecz stan pamięci DDRAM nie ulega zmianie. Np. kod:

```
while(1)
{
    hd44780_outcmd(HD44780_DISPCTL(0,1,1));
    _delay_ms(700);
    hd44780_outcmd(HD44780_DISPCTL(1,1,1));
    _delay_ms(700);
}
```

powoduje naprzemienne włączanie i wyłączanie matrycy, w efekcie wyświetlany tekst mruga.

Parametr *cursor* określa, czy sprzętowo wyświetlany kursor ma być widoczny (parametr równy 1), czy niewidoczny (parametr *cursor* równy 0). Jeśli kursor jest widoczny, to dodatkowo możemy określić, czy ma migać (parametr *blink* równy 1), czy nie (*blink* = 0). Jeśli kursor jest widoczny i nie migła, wyświetlany jest w postaci poziomej kreski. Wyświetlanie kursora może być użyteczne w sytuacji, kiedy użytkownik wprowadza jakieś dane. W pozostałych sytuacjach kursor możemy raczej wyłączyć.

### **HD44780\_ENTMODE(*inc*, *shift*)**

Określa sposób wprowadzania danych do pamięci. Jeśli parametr *inc* jest równy 1, to po każdej operacji zapisu/odczytu adres pamięci jest automatycznie zwiększany o jeden, co przyśpiesza operację zapisu/odczytu kolejnych znaków. Jeśli *inc* jest równy 0, to adres jest automatycznie zmniejszany o 1. Dodatkowo parametr *shift* określa, czy wyświetlany obraz ma być automatycznie przesuwany, tak aby wyświetlać ostatnio wprowadzony znak (*shift* = 1), czy też nie (*shift* = 0).

### **HD44780\_CLR**

Powoduje wyczyszczenie (wypełnienie znakami o kodzie 32) całej zawartości pamięci DDRAM. Jednocześnie kursor wraca do pozycji pierwszej (pierwsza linia, pierwszy znak).



Niewyświetlane bajty pamięci DDRAM także są kasowane. Jeśli są one używane do przechowywania zmiennych programu, to przed operacją kasowania należy je skopiować w bezpieczne miejsce, w przeciwnym przypadku zostaną one utracone.

Wskaźnik adresu do pamięci DDRAM jest ustawiany na 0, zerowane jest także przesunięcie.

## **HD44780\_HOME**

Powoduje powrót kurSORA do pozycji początkowej — pierwsza linia, pierwszy znak. Wskaźnik adresu do pamięci DDRAM w efekcie zawiera 0, zerowane jest również przesunięcie wyświetlania znaków, w efekcie są one wyświetlane, począwszy od pozycji 0.

## **HD44780\_DDADDR(addr)**

Ustawia następny adres zapisu/odczytu w pamięci DDRAM na `addr`. Tylko 7 bitów argumentu `addr` jest brane pod uwagę, w efekcie można zaadresować maksymalnie 128 bajtów pamięci (z czego sterownik posiada tylko 80, pozostałe lokacje są nie używane).

## **HD44780\_CGADDR(addr)**

Ustawia następną operację odczytu/zapisu w pamięci CGRAM na `addr`. Pod uwagę branych jest tylko 6 bitów zmiennej `addr`, w efekcie można zaadresować maksymalnie 64 bajty pamięci CGRAM.

Przedstawione powyżej funkcje realizują jedynie podstawową komunikację z modelem LCD. Potrzebujemy jednak ciągle funkcji odpowiedzialnej za jego inicjalizację — funkcja biblioteczna przeprowadza ten proces tylko w podstawowym zakresie, ciągle musimy zadbać o właściwe ustawienie parametrów, związane z posiadaną matrycą. Napiszmy więc własne rozwinięcie funkcji inicjalizacyjnej:

```
void lcd_init()
{
    hd44780_init(); //Podstawowa inicjalizacja modulu
    hd44780_outcmd(HD44780_CLR); //Wyczysc pamiec DDRAM
    hd44780_wait_ready(1000);
    hd44780_outcmd(HD44780_ENTMODE(1, 0)); //Tryb autoinkrementacji AC
    hd44780_wait_ready(1000);
    hd44780_outcmd(HD44780_DISPCTL(1, 0, 0)); //Wlacz wyswietlacz, wylacz kurSOR
    hd44780_wait_ready(1000);
}
```

Zauważmy, że po wysłaniu każdego polecenia do modułu wywoływana jest funkcja `hd44780_wait_ready(1000)` realizująca oczekiwanie, aż realizacja wysłanego polecenia dobiegnie końca. **Jest to niezbędne, gdyż do czasu ukończenia realizacji bieżącej operacji dalsze polecenia są ignorowane.** Jako argument podany jest parametr 1000 — jest on bez znaczenia, nasza biblioteka korzysta z flagi *Busy* i go po prostu ignoruje. Po wykonaniu powyższej funkcji moduł LCD jest poprawnie zainicjowany, a jego pamięć jest wyczyszczona (zawiera bajty 0x20), w efekcie moduł nic nie wyświetla. Od tej chwili możemy wysyłać znaki do wyświetlenia.



**Wskazówka** Po wykonaniu inicjalizacji ekran może pozostać czysty lub wyświetlać całkowicie wypełnione kwadraty (wszystkie piksele włączone). Jeśli po inicjalizacji wszystkie piksele są włączone, należ nieco zmniejszyć kontrast.

Aby sobie ułatwić to zadanie, zdefiniujmy funkcję wysyającą do modułu jeden znak:

```
void lcd_putchar(char c)
{
    static bool second_nl_seen;
    static uint8_t line=0;

    if ((second_nl_seen) && (c != '\n')&&(line==0))
    { //Odebrano pierwszy znak
        hd44780_wait_ready(40);
        hd44780_outcmd(HD44780_CLR);
        hd44780_wait_ready(1600);
        second_nl_seen=false;
    }
    if (c == '\n')
    {
        if (line==0)
        {
            line++;
            hd44780_outcmd(HD44780_DDADDR(64)); //Adres pierwszego znaku drugiej linii
            hd44780_wait_ready(1000);
        }
        else
        {
            second_nl_seen=true;
            line=0;
        }
    }
    else
    {
        hd44780_outdata(c);
        hd44780_wait_ready(40);
    }
}
```

Funkcja ta, ze względu na skomplikowane adresowanie pamięci DDRAM, jest zależna od konkretnego modelu modułu LCD. Powyższa będzie prawidłowo współpracować z modułami 2×16, 2×20 i 2×40 znaków. W przypadku innych modułów pozycje niektórych znaków mogą być nieprawidłowe.

Mogliśmy już wyświetlać na LCD pojedyncze znaki, czas na wyświetlanie tekstu. Ponieważ teksty jako stałe dla zaoszczędzenia cennej pamięci RAM przechowuje się zazwyczaj w pamięci FLASH, zdefiniujmy więc funkcję odczytującą podany łańcuch znakowy z pamięci FLASH, a następnie umieszczającą go na LCD:

```
void lcd_puttext_P(prog_char *txt)
{
    char ch;
    while((ch=pgm_read_byte(txt)))
    {
        lcd_putchar(ch);
        txt++;
    }
}
```

Powyższa funkcja wysyła kolejne odczytane znaki z pamięci FLASH na LCD przy pomocy funkcji LCD\_putchar. Po napotkaniu znaku końca łańcucha (0, null) przerywa swoje działanie.

Pozostaje nam przetestować działanie powyższych funkcji:

```
int main()
{
    lcd_init();
    lcd_puttext_P(PSTR("Test LCD.\nDruga linia"));
    while(1) {}
}
```

Na ekranie modułu LCD powinien wyświetlić się napis:

```
Test LCD.
Druga linia
```



Jeśli nic na LCD nie widzimy, to najpierw powinniśmy sprawdzić kontrast, zmieniając napięcie na pinie  $V_{ee}$  modułu LCD.

Warto zdefiniować jeszcze funkcję ustawiającą pozycję, od której wyświetlany będzie następny znak:

```
void lcd_goto(uint8_t x, uint8_t y)
{
    hd44780_outcmd(HD44780_DDADDR(0x40*y+x));
    hd44780_wait_ready(1000);
}
```

Funkcja ta przyjmuje dwa parametry:  $x$  i  $y$ , określające pozycję, na której zostanie wyświetlony następny znak wysłany z mikrokontrolera do modułu LCD. Dzięki temu z łatwością możemy uaktualniać zawartość LCD, bez konieczności ponownego przesyłania wszystkich danych. Funkcja ta jest zależna od typu zastosowanego modułu.

Ostatnią funkcją jest funkcja umożliwiająca czyszczenie zawartości pamięci modułu. Jest ona realizowana przy pomocy prostego polecenia wysyłanego do sterownika:

```
void lcd_cls()
{
    hd44780_outcmd(HD44780_CLR);
    hd44780_wait_ready(false);
}
```

Oprócz czyszczenia zawartości pamięci modułu powoduje ona ustawienie kurSORA w punkcie (0,0).

## Definiowanie własnych znaków

Układ HD44780 wyposażony jest w 64 bajty pamięci CGRAM (ang. *Character Generator RAM*), która może przechowywać mapy bitowe 8 znaków o matrycy  $5 \times 8$  pikseli lub 4 znaków o matrycy  $5 \times 10$  pikseli. Przykład definicji znaku na matrycy  $5 \times 8$  pokazano na rysunku 18.3.

**Rysunek 18.3.**

Przykład definicji własnego znaku w pamięci CGRAM. Znak rysujemy na matrycy  $5 \times 8$  bitów, pola zaciemnione odpowiadają wartości 1, pola białe wartości 0. Wyliczoną wartość dla każdej linii pikseli wpisujemy do pamięci CGRAM sterownika. Najstarsze 3 bity każdego bajtu (pola szare) są bez znaczenia — wyświetlacz pomija je przy wyświetlaniu obrazu

Adres	Wartość
0	00100 = 4
1	01110 = 14
2	11111 = 31
3	00100 = 4
4	00100 = 4
5	11111 = 31
6	01110 = 14
7	00100 = 4

Dostęp do tej pamięci jest możliwy dzięki wysłaniu polecenia ustawiającego adres zapisu/odczytu pamięci CGRAM. Po jego wysłaniu wszystkie operacje zapisu i odczytu dotyczą tej pamięci, a nie pamięci DDRAM. Aby wrócić do adresowania pamięci DDRAM, należy wybrać jej adres lub dokonać operacji wymagającej przesunięcia kurSORA.



Pamięć CGRAM jest pamięcią ulotną, po ponownym włączeniu zasilania definicje znaków muszą być ponownie przesłane przez procesor.

Bity o wartości 1 w definicji znaku odpowiadają włączonym pikselom, bity o wartości 0 pikselom wyłączonym. Najstarsze 3 bity każdego bajtu w pamięci CGRAM są bez znaczenia — te pozycje nie są wyświetlane przez matrycję. Definiując znak, należy także pamiętać, że ostatnia linia wyświetlana przez matrycję (w zależności od wielkości matrycy znaku linia 8 lub 10) powinna zawierać same zera — w linii tej będzie wyświetlany kurSOR.

Najczęściej matryce LCD podłączone do sterownika HD44780 mają organizację  $5 \times 8$ , czyli jeden znak jest definiowany przez 8 kolejnych bajtów określających stan poszczególnych linii tworzących znak. Stwórzmy więc funkcję, której zadaniem będzie załadowanie definicji znaków z pamięci FLASH mikrokontrolera do pamięci CGRAM:

```
void lcd_defchar_P(uint8_t charno, prog_uint8_t *chardef)
{
    hd44780_outcmd(HD44780_CGADDR(charno*8));
    hd44780_wait_ready(40);
    for(uint8_t c=0;c<8;c++)
    {
        hd44780_outdata(pgm_read_byte(chardef));
        hd44780_wait_ready(40);
        chardef++;
    }
}
```

Funkcja LCD\_defchar\_P jako argumenty przyjmuje numer definiowanego znaku (0 – 7) oraz wskaźnik do obszaru pamięci FLASH zawierającego definicję znaku (obszar ten zajmuje 8 następujących po sobie bajtów). Jak widzimy, funkcja najpierw ustawia właściwy adres w pamięci CGRAM, począwszy od którego znajdzie się definicja znaku — ponieważ jeden znak opisywany jest przy pomocy 8 bajtów, kolejne adresy znaków wynoszą kod\_znaku × 8. Następnie dane opisujące znak ładowane są z pamięci mikrokontrolera i zapisywane w pamięci CGRAM sterownika. Poniższy program ilustruje wykorzystanie funkcji LCD\_defchar\_P:

```
prog_uint8_t char1[8][8]={{0,0,0,0,0,0,0,255}, {0,0,0,0,0,0,255,255},
←{0,0,0,0,0,255,255,255}, {0,0,0,0,255,255,255,255}, {0,0,0,255,255,255,255,255},
←{0,0,255,255,255,255,255,255}, {0,255,255,255,255,255,255,255},
←{255,255,255,255,255,255,255,255}};
```

```
int main()
{
    for(uint8_t x=0;x<8;x++) lcd_defchar_P(x,&char1[x][0]); //Załaduj wzorce znaków
    lcd_init();
    for(uint8_t x=0;x<8;x++) lcd_putchar(x);      //Wyświetl znaki o kodach 0-7
    while(1) {}
}
```

W pierwszej linii funkcji main następuje załadowanie nowych definicji znaków do pamięci CGRAM, następnie inicjalizowany jest wyświetlacz, po czym w pierwszej linii wyświetlane są znaki o kodach 0 – 7, a więc znaki, których definicja została przesłana z mikrokontrolera.

Zauważmy, że pamięć CGRAM, podobnie jak DDRAM, możemy zapisywać i odczytywać przed inicjalizacją modułu LCD. Proces inicjalizacji jest tylko potrzebny do prawidłowego wyświetlania na ekranie LCD.

Ponieważ dane potrzebne do wyświetlenia znaku są na bieżąco pobierane z pamięci podczas odświeżania matrycy LCD, jakiekolwiek zmiany w definicji znaku, kiedy jest on widoczny na LCD, pociągają natychmiastowe zmiany w wyświetlonym obrazie. Można tę właściwość sterownika wykorzystać do stworzenia prostych animacji:

```
void lcd_box(uint8_t y)
{
    hd44780_outcmd(HD44780_CGADDR(0)); //Zaczynamy od znaku o kodzie 0
    hd44780_wait_ready(40);
    for(uint8_t c=0;c<y;c++) //Stwórz linie z włączonymi pikselami
    {
        hd44780_outdata(0xFF);
        hd44780_wait_ready(40);
    }
    for(uint8_t c=y;c<8;c++) //Stwórz linie z wyłączenymi pikselami
    {
        hd44780_outdata(0x00);
        hd44780_wait_ready(40);
    }
}

int main()
{
```

```

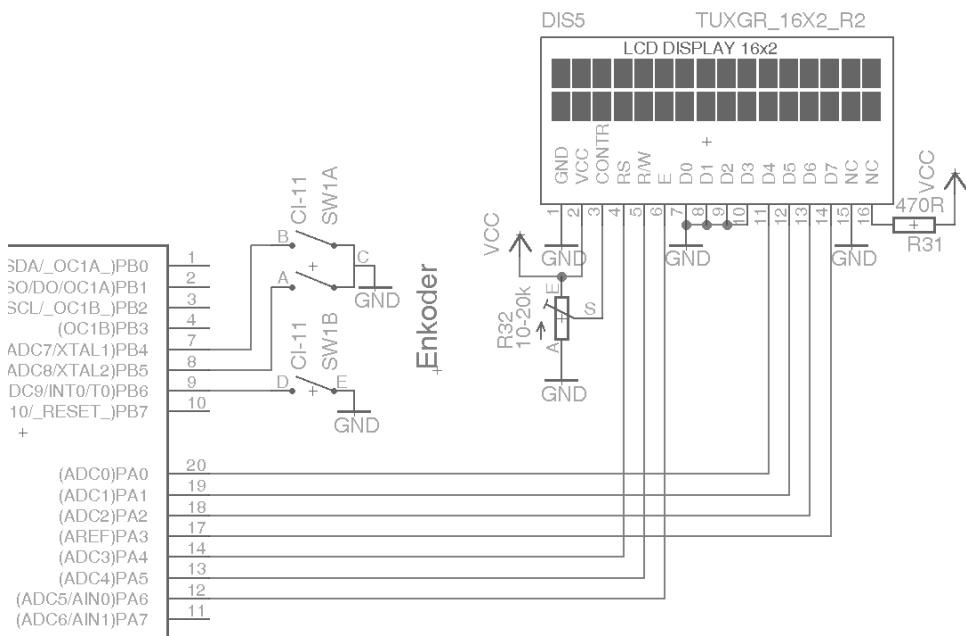
lcd_init();
lcd_putchar(0);
while(1)
{
    for(uint8_t x=0;x<8;x++)
    {
        lcd_box(x);
        _delay_ms(100);
    }
    for(int8_t x=7;x>=0;x--)
    {
        lcd_box(x);
        _delay_ms(100);
    }
}

```

Funkcja `main` wyświetla na LCD tylko jeden znak o kodzie 0, a następnie cyklicznie zmienia jego definicję w pamięci CGRAM. W efekcie wyświetlana postać znaku na ekranie ciągle się zmienia.

## Przykład — menu

Jako przykład zastosowania alfanumerycznego LCD pokazana zostanie realizacja przy jego pomocy menu programu. Schemat przedstawionego układu pokazano na rysunku 18.4.



**Rysunek 18.4.** Schemat przykładowego układu wykorzystującego LCD alfanumeryczny oraz enkoder obrotowy

Będzie ono umożliwiało wybór opcji. Na początku przyjmijmy pewne założenia. Menu musi być maksymalnie uniwersalne, tak aby z łatwością dało się je dostosować do dowolnego programu i łatwo zmieniać. Najlepiej, gdyby zajmowało mało pamięci, szczególnie cennej pamięci SRAM, powinna także istnieć możliwość realizacji podmenu.



Kompletny program realizujący obsługę menu znajduje się w katalogu *LCD-alfa-Menu*.

Aby ułatwić posługiwanie się pokazanym przykładem, funkcje związane z poszczególnymi opcjami wywoływane są bezpośrednio przez program obsługi menu. Dzięki temu nie zachodzi potrzeba sprawdzania stanu menu. Jedyne, co trzeba zrobić, to zdefiniować funkcje obsługi i przypisać je poszczególnym pozycjom menu.

Wszystkie dane związane z budową menu przechowywane są w pamięci FLASH mikrokontrolera, dzięki czemu nie jest zużywana pamięć RAM. Są one tworzone na etapie komplikacji programu.

Najpierw zdefiniujmy struktury przechowujące dane menu:

```
struct PROGMEM _menuitem
{
    prog_char *text;
    menuitemfuncptr menuitemfunc;
    const struct _menuitem *parent;
    const struct _menuitem *submenu;
    const struct _menuitem *next;
};
```

Stworzona została struktura `_menuitem`, zawierająca następujące pola:

- ◆ `text` — jest to wskaźnik do pamięci FLASH, w której znajduje się tekst opisujący daną pozycję menu.
- ◆ `menuitemfunc` — jest wskaźnikiem do funkcji wywoływanej po wybraniu danej pozycji menu.
- ◆ `parent` — zawiera wskaźnik do struktury `_menuitem` menu rodzica lub `NULL`, jeśli dane menu jest menu głównym.
- ◆ `submenu` — zawiera wskaźnik do podmenu lub `NULL`, jeśli dana pozycja nie ma podmenu.
- ◆ `next` — zawiera wskaźnik do kolejnej struktury o typie `_menuitem`, opisującej kolejną pozycję menu, lub `NULL`, jeśli dana pozycja jest ostatnią.

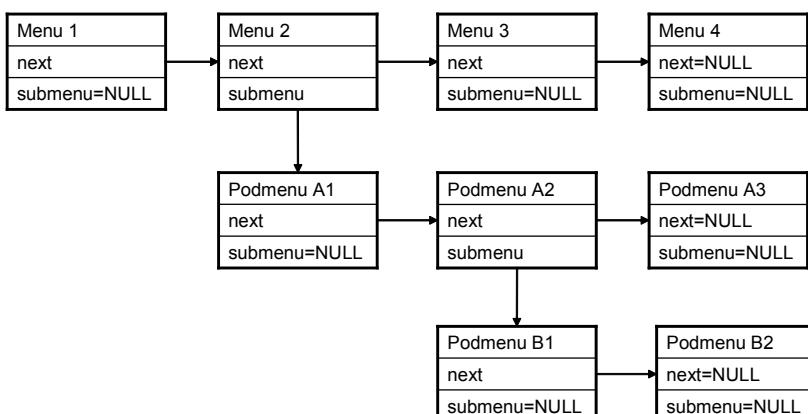
Pole `menuitemfunc` zawiera wskaźnik do funkcji skojarzonej z daną pozycją menu. W tym celu stworzony został nowy typ wskaźnikowy `menuitemfuncptr`:

```
typedef void (*menuitemfuncptr)();
```

Mogimy mu przypisać dowolną funkcję, która nie przyjmuje żadnych argumentów i nie zwraca wyników, np.

```
void foo();
```

Funkcje takie będą automatycznie wywoływanego po wybraniu danej pozycji menu. Dzięki polu next poszczególne struktury o typie `_menuitem` możemy łączyć ze sobą w tzw. **listę jednokierunkową**. Na takiej liście pole next jednej struktury wskazuje na kolejną strukturę, której pole next może wskazywać na kolejną. Koniec tak utworzonej listy określony jest przez strukturę, której pole next zawiera wartość `NULL` (nie wskazuje więc na kolejny element listy). Taka złożona struktura danych umożliwia stworzenie liniowego menu — to znaczy takiego, które ma dowolną liczbę pozycji, lecz nie może w nim występować podmenu. Aby móc dodawać podmenu, stworzone zostało pole submenu. Wskazuje ono na kolejną listę jednokierunkową, która zawiera opisy podmenu. Ponieważ każdy element podmenu też ma swoje pole submenu, możemy w ten sposób tworzyć dowolnie rozgałęzione menu (rysunek 18.5).



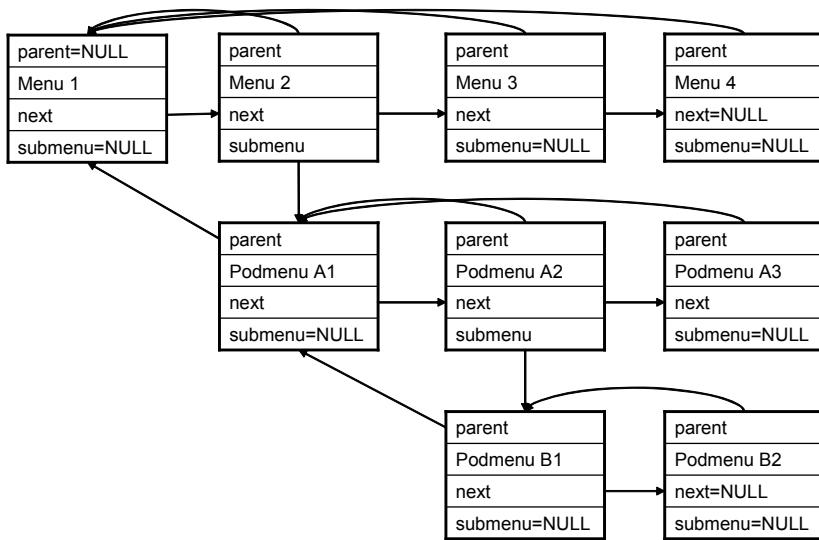
**Rysunek 18.5.** Realizacja rozgałęzionego menu przy pomocy listy jednokierunkowej

Po menu możemy się przesuwać „poziomo”; wybór opcji, która związana jest z podmenu, powoduje przesunięcie się o jedną linię w hierarchii menu w dół. Niestety, listy jednokierunkowe mają poważną wadę — łatwo można ustalić położenie następnego elementu, lecz aby znaleźć poprzedni, musimy znać adres pierwszego elementu listy, co umożliwia jej przeszukanie w celu znalezienia elementu poprzedzającego. Wady tej pozbawione są **listy dwukierunkowe**, w których każdy element oprócz wskaźnika na kolejny element listy zawiera także wskaźnik na element poprzedni. Jednak dla struktury hierarchicznej, jaką jest menu, wygodniej jest wykorzystać listę jednokierunkową, z dodatkowym polem wskazującym na pierwszy element listy (w strukturze `_menuitem` jest to pole `parent`). Dzięki temu po dojściu do elementu ostatniego (którego pole `next` ma wartość `NULL`) adres pierwszego rekordu pobierany jest z jego pola `parent`. Taka budowa listy umożliwia łatwy powrót z podmenu do menu wyższego poziomu. Po prostu pierwszy element podmenu zawiera w swoim polu `parent` wskaźnik do elementu menu wyższego poziomu (rysunek 18.6).

Dzięki tak zdefiniowanej strukturze można łatwo przemieszczać się po wszystkich poziomach menu.

Wskaźnik do pierwszego elementu menu zawiera zmienną menu:

```
extern struct _menuitem PROGMEM menu;
```



**Rysunek 18.6.** Powiązanie rekordów w strukturze `_menuitem`. W ramach danego poziomu menu pola `parent` wszystkich rekordów wskazują na pierwszy element danego poziomu. Element ten wskazuje z kolei na pierwszy element menu poziomu wyższego w hierarchii. Element umieszczony najwyższej w hierarchii menu ma pole `parent` równe `NULL`

Została ona zadeklarowana w pliku `menu.h`, jej definicję należy umieścić we własnym programie, przyporządkowując jej pierwszy element tworzonego menu.

Czas stworzyć strukturę przechowującą menu takie jak pokazane na rysunku 18.5:

```

struct _menuitem menu;
struct _menuitem menuA1;
struct _menuitem menuB1;

struct _menuitem menuB3 PROGMEM = {txt10, Menu_Back, &menuB1, 0, 0};
struct _menuitem menuB2 PROGMEM = {txt9, menufunc7, &menuB1, 0, &menuB3};
struct _menuitem menuB1 PROGMEM = {txt8, menufunc6, &menuA1, 0, &menuB2};

struct _menuitem menuA4 PROGMEM = {txt10, Menu_Back, &menuA1, 0, 0};
struct _menuitem menuA3 PROGMEM = {txt7, menufunc5, &menuA1, 0, &menuA4};
struct _menuitem menuA2 PROGMEM = {txt6, 0, &menuA1, &menuB1, &menuA3};
struct _menuitem menuA1 PROGMEM = {txt5, menufunc4, &menu, 0, &menuA2};

struct _menuitem menu3 PROGMEM = {txt4, menufunc3, &menu, 0, 0};
struct _menuitem menu2 PROGMEM = {txt3, menufunc2, &menu, 0, &menu3};
struct _menuitem menu1 PROGMEM = {txt2, 0, &menu, &menuA1, &menu2};
struct _menuitem menu PROGMEM = {txt1, menufunc1, 0, 0, &menu1};

```

Zmienne `txt1` – `txt10` przechowująłańckie znakowe będące nazwami poszczególnych pozycji menu:

```

prog_char txt1[]="Menu1";
prog_char txt2[]="Menu2";
prog_char txt3[]="Menu3";

```

```
prog_char txt4[]="Menu4";  
  
prog_char txt5[]="Podmenu A1";  
prog_char txt6[]="Podmenu A2";  
prog_char txt7[]="Podmenu A3";  
  
prog_char txt8[]="Podmenu B1";  
prog_char txt9[]="Podmenu B2";  
prog_char txt10[]="";  
prog_char txt11[]="F: ";
```

Ze względu na architekturę mikrokontrolerów AVR i sposób działania atrybutu PROGMEM nie możemy tekstów opisujących poszczególne pozycje menu umieścić bezpośrednio w definicji struktur — spowodowałoby to ich niepotrzebne skopiowanie do pamięci SRAM. Z kolei memfunc1 – memfunc7 to funkcje wywoływanie po wybraniu odpowiednich pozycji menu.

Jak widać, hierarchiczna struktura złożona z rekordów o typie `_menuitem` tworzona jest „od tyłu”. Wynika to z tego, że każdy rekord musi posiadać wskaźnik do kolejnego rekordu. W efekcie listę musimy tworzyć od ostatniego należącego do niej rekordu (rekordu, którego pole `next` ma wartość `NULL`). **Cała złożona struktura danych przechowująca menu tworzona jest na etapie komplikacji.** Ponieważ w pewnych sytuacjach musimy odwoływać się do zmiennych, które nie zostały jeszcze utworzone, stąd do definicji takich elementów wykorzystano **deklaracje zapowiadające**. Jak pamiętamy, umieszczenie w programie deklaracji:

```
struct _menuitem menu;
```

powoduje, że zmienna `menu` staje się znana kompilatorowi, dzięki czemu może on wygenerować do niej odwołania przed jej zdefiniowaniem.



Ponieważ cała struktura menu tworzona jest w pamięci FLASH, nie zajmuje ona pamięci SRAM. Niestety, powoduje to, że do tak zdefiniowanej struktury nie możemy odwoływać się przy pomocy standardowych operatorów języka C.

W tym celu w pliku `menu.c` zdefiniowano makrodefinicję:

```
#define GetAddr(ptr, field) (void*)pgm_read_word(((uint8_t*)ptr)+offsetof(struct  
↳ _menuitem, field))
```

Jako argumenty przyjmuje ona wskaźnik i nazwę pola struktury, które chcemy odczytać. Wynikiem jej działania jest zwrócenie 16-bitowej wartości wskazanego pola. Wartość pola struktury umieszczonej w pamięci FLASH odczytywana jest przy pomocy znanej nam już funkcji `pgm_read_word`. Dzięki makrodefinicji `GetAddr` możemy odwoływać się do poszczególnych pól struktury `_menuitem` prawie tak samo wygodnie jak przy pomocy standardowych operatorów języka C.



Wykorzystanie tej makrodefinicji ma jeszcze jedną zaletę — jeżeli chcielibyśmy całe menu przechowywać w pamięci SRAM (np. w celu umożliwienia jego modyfikacji), wystarczy zmienić definicję `GetAddr`.

Posiadając odpowiednio zainicjowaną strukturę `_menuitem`, możemy napisać procedury obsługi menu. Do ich poprawnej pracy potrzebne są pewne zmienne, wskazujące na aktualnie wybrane menu. Ponieważ mamy menu główne i wiele podmenu, zdefiniujmy zmienną wskazującą na pierwszy element aktualnie wybranego poziomu menu:

```
static const struct _menuitem *currMenuPtr=&menu;
```

Zmienna `currMenuPtr` została zadeklarowana w pliku *main.c* jako statyczna, gdyż jest ona wewnętrzną zmienną, wykorzystywaną wyłącznie przez funkcje obsługi menu. Poza nimi nigdzie nie powinna być wykorzystywana. Domyślnie wskazuje ona na pierwszy element menu głównego. Drugą potrzebną zmienną jest zmienna wskazująca na aktualnie wybrany przez użytkownika element menu:

```
static int8_t menuindex;
```

Podobnie jak `currMenuItem` jest to zmienna wykorzystywana wewnętrznie przez funkcje obsługi menu. W przypadku wyświetlaczy LCD mogących wyświetlać kilka linii tekstu musimy znać numer pozycji menu wyświetlanej w pierwszym wierszu LCD:

```
static int8_t menufirstpos;
```

Pierwszą potrzebną funkcją jest funkcja wyświetlająca menu na podstawie stanu wcześniej zdefiniowanych zmiennych `currMenuItem`, `menuindex` i `menufirstpos`:

```
void Menu_Show()
{
    const struct _menuitem *tmpmenuitem=GetMenuItem(menufirstpos);

    for(uint8_t i=0;i<LCD_ROWS;i++)
    {
        uint8_t charcnt=1;
        lcd_goto(0,i);
        if((i+menufirstpos)==menuindex) lcd_putchar('>'); else lcd_putchar(' ');
        if(GetAddr(tmpmenuitem, text))
        {
            lcd_putstr_P(GetAddr(tmpmenuitem, text));
            charcnt+=strlen_P(GetAddr(tmpmenuitem, text));
        }
        if(GetAddr(tmpmenuitem, submenu))
        {
            lcd_putchar(0x7E);
            charcnt++;
        }
        for(;charcnt<LCD_COLUMNS;charcnt++) lcd_putchar(' '); //Wyczyść resztę
                                                               //wyświetlanej linii
        tmpmenuitem=GetAddr(tmpmenuitem, next);
    }
}
```

Funkcja ta wyświetla tyle pozycji menu, ile dany wyświetlacz ma linii — liczba linii, jaką może wyświetlić LCD, jest zdefiniowana w symbolu `LCD_ROWS`. Następnie funkcja pobiera element menu o numerze określonym zmienną `menufirstpos` i wysyła go na LCD. Zanim to nastąpi, numer wyświetlanej pozycji jest porównywany z wartością zmiennej `menuindex`. Jeśli te wartości są sobie równe, dodatkowo przed daną pozycją menu wyświetlany jest symbol `>`, informujący użytkownika, że dana pozycja jest pozycją

aktualnie wybraną. W przeciwnym przypadku wyświetlana jest spacja. Podobnie, po wyświetleniu danej pozycji funkcja sprawdza, czy z danym elementem menu nie jest związane jakieś podmenu — w takiej sytuacji pole submenu ma wartość różną od 0. Jeśli dana pozycja posiada podmenu, z jej prawej strony wyświetlany jest symbol →. Ostatnią rzeczą jest dopełnienie wyświetlonego tekstu spacjami aż do ostatniej wyświetlonej kolumny. Jest to spowodowane tym, że w celu uniknięcia nieprzyjemnego migania LCD przy odświeżaniu menu (np. przy jego przesuwaniu) LCD przed wyświetlaniem menu nie jest czyszczony. W efekcie, jeśli napis składający się na kolejną pozycję menu jest krótszy, niż do tej pory wyświetlany, to bez uzupełniania spacjami część tego napisu pozostałaby na LCD. Liczbę znaków wyświetlanych w wierszu należy zdefiniować w symbolu LCD\_COLUMNS. Zarówno LCD\_ROWS, jak i LCD\_COLUMNS należy zdefiniować w pliku nagłówkowym *lcd.h*.

Mogliśmy już wyświetlić dowolną część menu. Pozostaje jeszcze stworzyć funkcje umożliwiające nawigację po menu — przesuwanie się o jedną pozycję wstecz i do przodu:

```
void Menu_SelectNext()
{
    if(GetMenuItem(menuindex+1)!=0)
    {
        menuindex++;
        if((menuindex-menufirstpos)>=LCD_ROWS) menufirstpos++;
    } else
    {
        menuindex=0;
        menufirstpos=0;
    }
    Menu_Show();
}
```

Przesuwanie się o jedną pozycję do przodu jest proste — kolejną pozycję menu wskazuje pole next pozycji poprzedniej. Po natrafieniu na ostatni element listy znajdowany jest jej pierwszy element. Dzięki temu możemy „kręcić się” dookoła menu. Podobnie realizujemy przejście o jedną pozycję wstecz:

```
void Menu_SelectPrev()
{
    if(menuindex>0)
    {
        menuindex--;
        if(menuindex<menufirstpos) menufirstpos--;
    } else
    {
        const struct _menuitem *tmpmenuitem=currMenuPtr;
        while(tmpmenuitem)
        {
            tmpmenuitem=GetAddr(tmpmenuitem,next);
            if(tmpmenuitem) menuindex++;
        }
        menufirstpos=menuindex-1;
        if(menufirstpos<0) menufirstpos=0;
    }
    Menu_Show();
}
```

Tu jednak sprawa jest nieco trudniejsza, gdyż lista jest listą jednokierunkową. Aby uzyskać adres elementu poprzedzającego, musimy przeszukać całą listę, poczawszy od jej pierwszego elementu. Ułatwia nam to kolejna funkcja:

```
const struct _menuitem *GetMenuItem(uint8_t index)
{
    const struct _menuitem *tmpmenuitem=currMenuPtr;

    while((tmpmenuitem) && (index>0))
    {
        tmpmenuitem=GetAddr(tmpmenuitem, next);
        index--;
    }
    return tmpmenuitem;
}
```

Jej argumentem jest numer elementu menu, którego adres chcemy uzyskać. Jeśli aktualnie wyświetlona pozycja menu ma np. numer 4, to adres pozycji poprzedniej uzyskamy, wywołując funkcję `GetMenuItem` z argumentem równym 3. Jeśli argument, z jakim wywołamy funkcję, jest większy niż liczba pozycji menu, to funkcja zwróci wartość `NULL`.

Możemy już swobodnie nawigować po menu, potrzebujemy jednak funkcji obsługującej sytuację, kiedy użytkownik wybiera wcześniej wskazaną pozycję menu (dla zrealizowania określonej przez nią funkcji lub przejścia do podmenu):

```
void Menu_Click()
{
    const struct _menuitem *tmpmenuitem=GetMenuItem(menuindex);
    const struct _menuitem *submenu=GetAddr(tmpmenuitem, submenu);

    menuitemfuncptr mfptr=GetAddr(tmpmenuitem, menuitemfunc);
    if(mfptr) (*mfptr)();
    if(submenu)
    {
        currMenuPtr=submenu;
        menuindex=0;
        menufirstpos=0;
    }
    Menu_Show();
}
```

Funkcja `Menu_Click()` pobiera adres funkcji związanej z daną pozycją menu. Jeśli wartość ta jest różna od `NULL` (funkcja obsługi dla danej pozycji została zdefiniowana), to ją wywołuje polecienniem `(*mfptr)()` — wskaźnik zawiera adres funkcji związanej z daną pozycją menu. Dodatkowo, jeśli z daną pozycją związane było podmenu, to następuje przejście do niego i odświeżenie zawartości LCD.

Potrzebna jest jeszcze funkcja odwrotna, to jest realizująca przejście z podmenu do menu wyższego poziomu. Funkcja ta zostanie zdefiniowana w ciekawy sposób. Ponieważ przejście do menu wyższego poziomu wiąże się z wybraniem odpowiedniej pozycji menu, w efekcie wywołana zostanie powyżej zdefiniowana funkcja `Menu_Click()`. Jednak aby mogła ona zrealizować przejście o jeden poziom menu wyżej, pozycje menu

realizujące funkcję powrotu musiałyby być specjalnie oznaczone w celu odróżnienia ich od zwykłych pozycji menu. Aby tego uniknąć, zdefiniowano specjalną funkcję obsługi danej pozycji menu:

```
void Menu_Back()
{
    const struct _menuitem *tmpMenuPtr=GetAddr(currMenuPtr, parent);
    menufirstpos=0;
    menuindex=0;
    currMenuPtr=tmpMenuPtr;
}
```

Wykonanie funkcji `Menu_Back()` powoduje przejście o jeden poziom wyżej w hierarchii menu (o ile już nie byliśmy w menu głównym). Ponieważ zmienna `currMenuPos` wskazuje zawsze na pierwszy element danego poziomu menu, przejście do menu o jeden poziom wyższego wymaga odczytania pola `parent` wskazywanej przez nią struktury. Przy wykorzystaniu funkcji `Menu_Back()` powrót z podmenu realizujemy, dodając definicję elementu menu jak poniżej:

```
struct _menuitem menuB3 PROGMEM = {txt10, Menu_Back, &menuB1, 0, 0};
```

Funkcja `Menu_Back()` jest więc wywoływana po wybraniu powyższego elementu menu.

Mamy już zdefiniowane wszystkie potrzebne funkcje, pozostało tylko powiązać poszczególne akcje menu (przesunięcie menu do przodu, do tyłu, wybór danej pozycji) z jakimś kontrolerem — przyciskami lub enkoderem. W tym celu posłużymy się funkcjami obsługi enkodera pokazanymi w rozdziale 14. Umożliwiają one odczytywanie stanu enkodera przy pomocy przerwań. Zanim będziemy mogli odczytać stan enkodera, musimy zainicjować poszczególne podsystemy naszego programu. W funkcji `main` w pliku *LCD-alfa.c* umieszczałyśmy wywołania funkcji inicjujących:

```
EncoderInit();
lcd_init();
Timer0Init();
Menu_Show();
sei();
```

Po kolej inicjowane są porty *I/O* związane z enkoderem, inicjowany jest moduł LCD, *timer* wykorzystywany do obsługi enkodera, wyświetlane jest menu początkowe i odblokowane zostają przerwania. Pozostaje więc stworzenie głównej pętli programu:

```
int8_t enc;
bool btn=false;
while(1)
{
    enc=Read4StepEncoder();
    if(enc>0)
    {
        Menu_SelectPrev();
        enc++;
    }
    if(enc<0)
    {
        Menu_SelectNext();
        enc--;
    }
}
```

```
        }
        if(GetEncButton()^btn)
        {
            btn=GetEncButton();
            if(btn) Menu_Click();
        }
    }
```

Jak widać, stan enkodera odczytywany jest cyklicznie, w zależności od kierunku jego obrotu wywoływane są funkcje `Menu_SelectNext()` i `Menu_SelectPrev()`. Po wcisnięciu przycisku enkodera wywoływana jest funkcja `Menu_Click()`. Przed uruchomieniem programu należy zdefiniować funkcje związane z obsługą poszczególnych pozycji menu.

## Obsługa wyświetlaczy graficznych

Wyświetlacze graficzne dają możliwość niezależnego sterowania każdym pojedynczym pikselem na ekranie. Dzięki temu możemy wyświetlać dowolne czcionki, a nawet obrazy. W efekcie możemy zaprojektować znacznie ciekawszy wizualnie interfejs użytkownika. Zwykle wyświetlacze graficzne dzięki większej rozdzielczości oferują możliwość jednoczesnego przedstawiania większej ilości informacji. Niestety, ceną, jaką za to płacimy, jest zdecydowanie większe zapotrzebowanie na pamięć RAM — w tym typie wyświetlaczy nie przechowuje ona kodów wyświetlanych znaków, lecz odpowiadające im obrazy bitowe. W efekcie, o ile w przypadku wyświetlaczy alfanumerycznych wyświetlenie znaku wiązało się zwykle z prostą operacją wysłania jego kodu do pamięci wyświetlacza, o tyle w przypadku wyświetlaczy graficznych to zwykle procesor jest odpowiedzialny za narysowanie całego znaku. Wskutek tego znacznie wzrasta ilość czasu, jaką procesor poświęca na generowanie obrazu. Większość sterowników wyświetlaczy graficznych nie posiada także wbudowanego generatora znaków — dlatego aby móc wyświetlać na ekranie litery i cyfry, musimy umieścić ich definicje w pamięci FLASH mikrokontrolera, co nawet w przypadku reprezentacji tylko części znaków może zająć ok. 1 kB pamięci. Stąd też wybierając ten typ wyświetlaczy, musimy wybrać do realizacji projektu mikrokontroler z większą ilością dostępnej pamięci FLASH oraz większą liczbą dostępnych linii I/O. Ze względu na wymaganą szybkość transferu danych pomiędzy mikrokontrolerem a LCD wykorzystywanych jest wszystkie 8 linii danych (`D0 – D7`).



Istnieją wyświetlacze graficzne LCD z interfejsem szeregowym, najczęściej przypominającym interfejs SPI. W takiej sytuacji potrzebujemy mniej linii IO, lecz możliwa do uzyskania prędkość wymiany danych też zazwyczaj jest mniejsza.

Podobnie jak w przypadku wyświetlaczy alfanumerycznych, również wyświetlacze graficzne wymagają podświetlenia. Najprościej wykorzystać wyświetlacze, w których podświetlenie zrealizowane jest w oparciu o diody LED.

W celu zademonstrowania zasady działania tych wyświetlaczy wykorzystany zostanie wyświetlacz graficzny LCD o rozdzielczości  $128 \times 64$  punkty, ze sterownikiem opartym

o popularny układ KS0108. Układ ten jest niezwykle prosty w obsłudze, udostępnia tylko kilka funkcji, umożliwiających odczyt i zapis wybranej komórki wewnętrznej pamięci RAM, odczyt statusu układu oraz kontrolę wyświetlacza (włączony/wyłączony).

Na rynku dostępne są wyświetlacze o różnych rozdzielczościach. W przypadku modułów opartych na kontrolerze KS0108 należy pamiętać, że jeden kontroler potrafi sterować matrycą o maksymalnej rozdzielczości  $64 \times 64$  piksele. W matrycach o wyższych rozdzielczościach stosowanych jest kilka takich układów, z których każdy steruje tylko częścią matrycy. W dalszej części użyty będzie moduł o rozdzielczości  $128 \times 64$  piksele, zawierający dwa układy KS0108. Podobnie jak w przypadku matryc alfanumerycznych, matryce graficzne łączymy z mikrokontrolerem przy pomocy linii danych ( $D0 - D7$ ) oraz linii sterujących. Rozkład linii komunikacyjnych przykładowego modułu pokazano w tabeli 18.2.

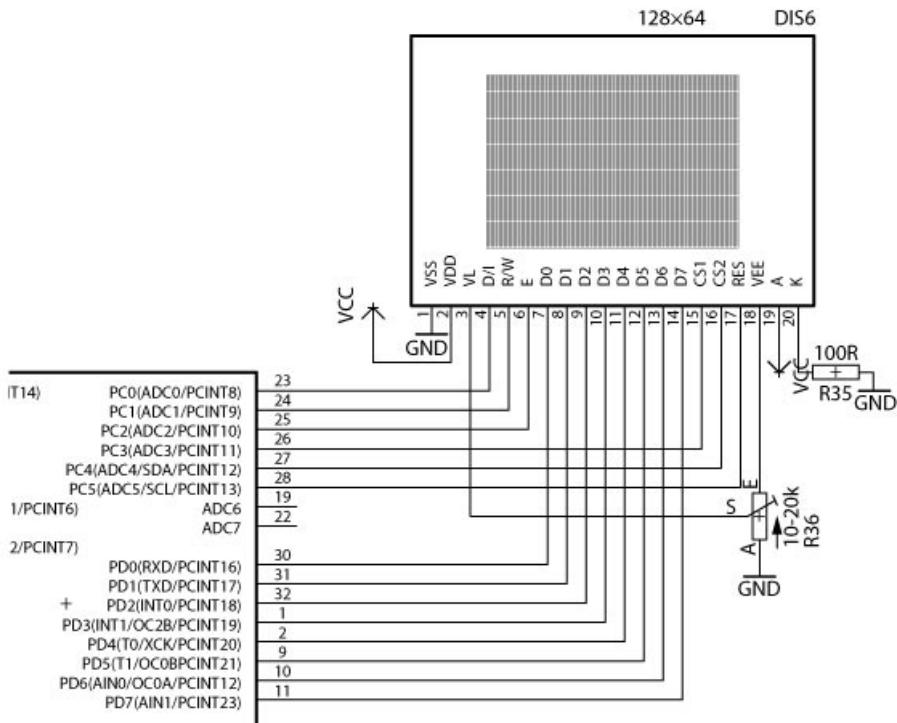
**Tabela 18.2.** Przykładowy rozkład wyprowadzeń modułu opartego o układ KS0108

Nr pinu	Nazwa	Znaczenie
1	<i>GND</i>	Masa układu (oznaczany także jako <i>Vss</i> )
2	<i>Vcc</i>	Zasilanie modułu (oznaczany także jako <i>Vdd</i> )
3	<i>Vo</i>	Regulacja kontrastu ( <i>Vl</i> )
4	<i>RS</i>	Wybór rejestru: stan niski — rejestr instrukcji, stan wysoki — rejestr danych ( <i>DI</i> )
5	<i>R/W</i>	Wybór operacji — odczyt (stan wysoki)/zapis (stan niski)
6	<i>E</i>	Sygnał zatrząskujący dane
7	<i>D0</i>	Dwukierunkowa linia danych
8	<i>D1</i>	Dwukierunkowa linia danych
9	<i>D2</i>	Dwukierunkowa linia danych
10	<i>D3</i>	Dwukierunkowa linia danych
11	<i>D4</i>	Dwukierunkowa linia danych
12	<i>D5</i>	Dwukierunkowa linia danych
13	<i>D6</i>	Dwukierunkowa linia danych
14	<i>D7</i>	Dwukierunkowa linia danych
15	<i>CS1</i>	Sygnał wyboru układu nr 1, aktywny w stanie niskim
16	<i>CS2</i>	Sygnał wyboru układu nr 2, aktywny w stanie niskim
17	<i>RST</i>	RESET, zerowanie modułu, aktywny w stanie niskim
18	<i>Vee</i>	Wyjście napięcia ujemnego do regulacji kontrastu
19	<i>LEDA</i>	Zasilanie podświetlenia — anody diod
20	<i>LEDK</i>	Zasilanie podświetlenia — katody diod



Ponieważ do podłączenia modułu graficznego potrzeba znacznie więcej wolnych pinów *IO*, dodatkowo potrzebujemy więcej pamięci FLASH w celu pomieszczenia definicji czcionek, w dalszej części wykorzystany zostanie procesor ATMega88.

Schemat podłączenia wyświetlacza graficznego o rozdzielcości  $128 \times 64$  piksele do mikrokontrolera AVR został pokazany na rysunku 18.7. Podłączenie pinu RST wyświetlacza do procesora jest opcjonalne. **Zazwyczaj można ten pin podpiąć poprzez rezystor na stałe do napięcia zasilającego moduł, co zwolni jeden pin I/O procesora.**

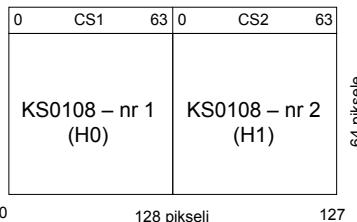


Rysunek 18.7. Schemat podłączenia wyświetlacza graficznego do mikrokontrolera ATMega88

Moduł ten zawiera dwa układy kontrolerów KS0108, stąd adresowanie jego pamięci jest stosunkowo skomplikowane (rysunek 18.8). Każdy z układów steruje połową matrycy ( $H0$  lub  $H1$ ). Łącznie pamięć obrazu składa się z  $64 \times 128 / 8 = 1024$  komórek pamięci, rozłożonych po 512 komórek pomiędzy oba układy.

Rysunek 18.8.

Organizacja pamięci obrazu w przypadku matrycy  $128 \times 64$  punkty



Ponieważ pokazany wyświetlacz jest wyświetlaczem monochromatycznym, do wyświetlenia 8 punktów wystarczy jeden bajt pamięci. Bitom o wartości 0 odpowiadają punkty wyłączone, bitom o wartości 1 punkty włączone. Stwarza to pewien kłopot przy zmianie stanu pikseli. Aby zmienić stan jednego, musimy najpierw pobrać cały bajt, zmienić

wartość odpowiedniego piksela, po czym zapisać cały bajt do pamięci obrazu sterownika. Stąd też, w przeciwieństwie do wyświetlaczów alfanumerycznych, musimy podłączyć linię *R/W*, bez niej nie ma możliwości odczytu pamięci obrazu, w efekcie modyfikacji pojedynczych pikseli obrazu. Brak tej linii wymagałby stworzenie bufora w pamięci SRAM mikrokontrolera, a zwykle nie możemy sobie na to pozwolić.

Poniżej pokazana została uniwersalna biblioteka umożliwiająca wykonywanie podstawowych operacji na wyświetlaczu graficznym. Biblioteka została podzielona na dwie części — część związaną bezpośrednio ze sprzętem (czyli konkretnym LCD i implementacją jego obsługi na konkretnym procesorze) oraz część niezależną od sprzętu, odpowiedzialną za wykonywanie prostych operacji graficznych (kreślenie linii, okręgów). Potrzebny będzie także trzeci komponent — krótki plik nagłówkowy zawierający konfigurację podłączenia LCD do mikrokontrolera, dzięki czemu pisana biblioteka będzie bardziej uniwersalna. Plik konfiguracyjny jest podobny do pliku konfiguracyjnego użytego w przypadku wyświetlaczów alfanumerycznych (plik *defines.h*):

```
/* Połączenia KS0108 z AVR */
#define KS0108_RS C. 0
#define KS0108_RW C. 1
#define KS0108_E C. 2
#define KS0108_CS1 C. 3
#define KS0108_CS2 C. 4
//#define KS0108_RESET C. 5 //Należy zdefiniować, jeśli używamy RESET

/* Linie danych D0-D7 muszą odpowiadać PORTx0-PORTx7 */
#define KS0108_DATA D

#define SLOW_TEXT 1 //Jeśli 1, to zostanie żąta wolniejsza procedura wyświetlania tekstu,
                //ale za to można wyświetlać z dokładnością do piksela
```

Odpowiednie symbole określają przyporządkowanie sygnałów sterujących LCD do poszczególnych pinów mikrokontrolera. Ponieważ sygnał *RESET* jest opcjonalny, jego zakomentowanie (niezdefiniowanie symbolu *KS0108\_RESET*) spowoduje, że biblioteka nie będzie go używać. Definiując symbole, należy podać literę oznaczającą użyty port oraz numer przyporządkowanego pinu. W przypadku linii danych *D0 – D7* podaje się tylko literę określającą użyty port. Co więcej, poszczególne piny portu (*Px0 – Px7*) muszą być elektrycznie połączone z odpowiadającymi im liniami danych LCD (*D0 – D7*). Ostatnim definiowanym symbolem jest *SLOW\_TEXT*. Jeżeli wynosi on 0, użyta będzie szybsza procedura wyświetlania znaków, lecz pozycjonowanie tekstu będzie się odbywało z dokładnością do znaku (tak jak w przypadku LCD alfanumerycznego). Kiedy symbol ten ma wartość 1, użyta zostanie nieco wolniejsza procedura wyświetlania, ale za to wyświetlany tekst można pozycjonować z dokładnością do 1 piksela.

Kolejnym etapem jest stworzenie pliku nagłówkowego zawierającego prototypy funkcji, deklaracje zmiennych i symboli używanych przez niskopoziomowe funkcje dostępu do sterownika (*KS0108.h*). Najpierw zostaną zdefiniowane symbole określające rozdzielcość użytej matrycy (w pikselach) oraz nazwy wewnętrznych rejestrów sterownika:

```
#define KS0108_SCREEN_WIDTH 128
#define KS0108_SCREEN_HEIGHT 64

#define DISPLAY_SET_Y          0x40
#define DISPLAY_SET_X          0xB8
```

```
#define DISPLAY_START_LINE 0xC0
#define DISPLAY_ON_CMD      0x3E
#define ON 0x01
#define OFF 0x00
#define DISPLAY_STATUS_BUSY 0x80
```

Dzięki tym definicjom można odwoływać się do nazw rejestrów, co czyni program bardziej przejrzystym.

W pliku tym zostały także zadeklarowane zmienne określające aktualną pozycję i kolor:

```
extern uint8_t GLCD_x;
extern uint8_t GLCD_y;
extern uint8_t color;
```

Dzięki zadeklarowaniu ich jako `extern` można się do nich odwoływać z każdego pliku, który włączy nagłówek *KS0108.h*. Pozostaje jeszcze zadeklarować prototypy użytych funkcji:

```
//Wewnętrzne funkcje biblioteki
void GLCD_WriteData(uint8_t byte);
void GLCD_WriteCommand(uint8_t cmd, uint8_t controller);
static inline uint8_t GLCD_ReadByteFromROMMemory(char *ptr) {return
    pgm_read_byte(ptr);}

//Funkcje zewnętrzne
void GLCD_init();
void GLCD_cls();
void GLCD_goto(uint8_t x, uint8_t y);
uint8_t GLCD_ReadData();
void GLCD_SetPixel(uint8_t x, uint8_t y);
void GLCD_putchar(char c);
```

Dwie pierwsze funkcje służą do niskopoziomowej komunikacji z modułem. Funkcja `GLCD_ReadByteFromROMMemory` zwraca wartość spod wskazanego adresu w pamięci FLASH mikrokontrolera (są tam zadeklarowane wzorce użytych czcionek).

Przejdźmy teraz do definicji powyższych funkcji (plik *KS0108.c*). Na początku pliku zadeklarowane są makrodefinicje związane z realizacją dostępu do poszczególnych pinów, którym przyporządkowano sygnały sterujące modułem:

```
#define GLUE(a, b)     a##b

#define SET_(what, p, m) GLUE(what, p) |= (1 << (m))
#define CLR_(what, p, m) GLUE(what, p) &= ~(1 << (m))
#define GET_/* PIN, */ p, m GLUE(PIN, p) & (1 << (m))
#define SET(what, x) SET_(what, x)
#define CLR(what, x) CLR_(what, x)

/*makra operujące na bajtach*/
#define DATA(what, b) GLUE(what, b)
```

Są to znane nam już makra, które wystąpiły w programie obsługi wyświetlacza alfanumerycznego. Wykorzystują one dyrektywy preprocesora do stworzenia instrukcji umożliwiających dostęp do zdefiniowanych pinów.

Przed wykorzystaniem jakichkolwiek funkcji tej biblioteki należy zainicjować kontroler oraz związane z jego obsługą porty *IO*:

```
void GLCD_init()
{
    SET(DDR, KS0108_E);
    SET(DDR, KS0108_RS);
    SET(DDR, KS0108_RW);
    SET(DDR, KS0108_CS1);
    SET(DDR, KS0108_CS2);
    SET(PORT, KS0108_CS1);
    SET(PORT, KS0108_CS2);

#ifndef KS0108_RESET
    SET(DDR, KS0108_RESET);
    SET(PORT, KS0108_RESET);
#endif

    for(uint8_t i=0; i<KS0108_SCREEN_WIDTH/64; i++)
    {
        GLCD_WriteCommand(DISPLAY_ON_CMD | ON, i);
        GLCD_WriteCommand(DISPLAY_START_LINE | 0,i);
    }
}
```

Opróczinicjalizacji linii komunikacyjnych z LCD inicjalizowane są także kontrolery. Funkcję tę należy wywołać raz, na początku programu.

Ponieważ wykorzystana matryca graficzna składa się z dwóch kontrolerów KS0108, potrzebna jest funkcja wybierająca aktywny kontroler:

```
void GLCD_EnableController(uint8_t controller)
{
    if(controller==0)
    {
        CLR(PORT, KS0108_CS1); SET(PORT, KS0108_CS2);
    } else
    {
        CLR(PORT, KS0108_CS2); SET(PORT, KS0108_CS1);
    }
}
```

Gdyby posiadana matryca miała inną liczbę kontrolerów, należy zmodyfikować powyższą funkcję tak, aby możliwa była aktywacja dodatkowych kontrolerów. Aktywacji wybranego kontrolera (w powyższym przykładzie 0 – 1) towarzyszy dezaktywacja drugiego.

Kolejna funkcja zwraca wartość rejestru statusu wybranego kontrolera:

```
uint8_t GLCD_ReadStatus(uint8_t controller)
{
    uint8_t status;
    DATA(DDR, KS0108_DATA)=0x00;
    SET(PORT, KS0108_RW);
    CLR(PORT, KS0108_RS);
    GLCD_EnableController(controller);
    SET(PORT, KS0108_E);
```

```

asm volatile ("nop");
status=DATA(PIN, KS0108_DATA);
CLR(PORT, KS0108_E);
return status;
}

```

Należy zwrócić uwagę na pozornie niepotrzebną linię `asm volatile ("nop")` — jest ona niezbędna, gdyż daje ona czas kontrolerowi na wystawienie stanu rejestru statusu. Powyższa funkcja wykorzystywana jest do stworzenia funkcji oczekującej na zakończenie realizacji bieżącej operacji:

```

static inline void GLCD_WaitUntilBusy(uint8_t controller)
{
    while(GLCD_ReadStatus(controller) & DISPLAY_STATUS_BUSY);
}

```

Ponieważ jest to krótka funkcja, zdefiniowano ją z modyfikatorami `static inline`, co powoduje, że odwołanie do niej nie generuje rozkazu skoku, lecz włącza jej kod w momencie wywołania. Wyzerowanie bitu `DISPLAY_STATUS_BUSY` sygnalizuje, że kontroler może przyjąć kolejny rozkaz.

Kolejna funkcja umożliwia wysłanie komendy do wybranego rejestru kontrolera:

```

void GLCD_WriteCommand(uint8_t cmd, uint8_t controller)
{
    GLCD_WaitUntilBusy(controller);
    DATA(DDR, KS0108_DATA)=0xFF;
    CLR(PORT, KS0108_RW);
    CLR(PORT, KS0108_RS);
    GLCD_EnableController(controller);
    DATA(PORT, KS0108_DATA)=cmd;
    SET(PORT, KS0108_E);
    asm volatile ("nop");
    CLR(PORT, KS0108_E);
}

```

Przed wysłaniem komendy funkcja oczekuje, aż wybrany kontroler będzie wolny. Następnie port, do którego podłączone są linie danych, ustawiany jest jako wyjście i wysyłana jest komenda. Rozróżnienie pomiędzy wysłaniem komendy a danych następuje poprzez badanie stanu linii `RS` — jeśli jest ona w stanie wysokim, wysyłana jest komenda, jeśli w stanie niskim — wysyłane są dane.

Podobnie wygląda funkcja realizująca wysyłanie 1 bajtu danych:

```

void GLCD_WriteData(uint8_t byte)
{
    uint8_t CS=GLCD_x/64;
    GLCD_WaitUntilBusy(CS);
    DATA(DDR, KS0108_DATA)=0xFF;
    CLR(PORT, KS0108_RW);
    SET(PORT, KS0108_RS);
    DATA(PORT, KS0108_DATA)=byte;
    GLCD_EnableController(CS);
    SET(PORT, KS0108_E);
    asm volatile ("nop");
}

```

```
CLR(PORT, KS0108_E);
GLCD_x++;
}
```

Numer uaktywnianego kontrolera zależy od miejsca, w którym znajduje się modyfikowany bajt pamięci. Miejsce to określone jest wartością zmiennej `GLCD_x` (wartości 0 – 63 odpowiadają pierwszemu kontrolerowi, 64 – 127 drugiemu). Ponieważ po wysłaniu bajtu kontroler automatycznie zwiększa wewnętrzny wskaźnik adresu na wartość o 1 większą, powyższa funkcja zwiększa też zmienną `GLCD_x`, tak aby oba wskaźniki pozostały zsynchronizowane.

Analogicznie do funkcji zapisującej bajt danych została stworzona funkcja odczytująca bajt danych:

```
unsigned char GLCD_ReadData()
{
    uint8_t data;
    uint8_t CS=GLCD_x/64;
    GLCD_WaitUntilBusy(CS);
    DATA(DDR, KS0108_DATA)=0x00;
    SET(PORT, KS0108_RW);
    SET(PORT, KS0108_RS);
    GLCD_EnableController(CS);
    SET(PORT, KS0108_E);
    asm volatile ("nop");
    data=DATA(PIN, KS0108_DATA);
    CLR(PORT, KS0108_E);
    GLCD_x++;
    return data;
}
```

Jedyną różnicą pomiędzy obiema funkcjami jest zmiana kierunku portu, do którego przyłączone są linie *D0 – D7*, na wejście.

Powyższe funkcje dokonywały zapisu/odczytu bajtu spod bieżącego adresu wskazywanego przez wewnętrzny rejestr adresowy kontrolera (któremu odpowiada wartość zmiennej `GLCD_x`). Aby możliwe było dokonywanie zapisu/odczytu spod dowolnego adresu pamięci, niezbędna jest funkcja `GLCD_goto`:

```
void GLCD_goto(uint8_t x, uint8_t y)
{
    uint8_t i;
    GLCD_x = x;
    GLCD_y = y;

    for(i=0; i<KS0108_SCREEN_WIDTH/64; i++)
    {
        GLCD_WriteCommand(DISPLAY_SET_Y, i);
        GLCD_WriteCommand(DISPLAY_SET_X | y, i);
    }
    GLCD_WriteCommand(DISPLAY_SET_Y | (x%64), (x/64));
    GLCD_WriteCommand(DISPLAY_SET_X | y, (x/64));
}
```

Powyższa funkcja zapisuje nowy adres do rejestrów adresowych kontrolerów. Ze względu na organizację matrycy rejestr kontrolera opisany jako `DISPLAY_SET_Y` przechowuje numery kolumn, a `DISPLAY_SET_Y` numery rzędów. Nie ma to jednak większego znaczenia. W przypadku posiadania tylko dwóch kontrolerów powyższą funkcję można znacznie uprościć. Do rejestrów adresowych obu kontrolerów wpisywana jest nowa wartość `GLCD_y`, natomiast do rejestru przechowującego numer kolumny nieaktywnego kontrolera wpisywane jest 0 — zapewnia to prawidłową adresację pikseli pomiędzy oboma kontrolerami.

Powyższe funkcje są wystarczające do obsługi kontrolera KS0108, lecz korzystanie z nich jest mało wygodne. Stąd też zdefiniowane zostały kolejne funkcje realizujące podstawowe operacje. Pierwszą jest funkcja umożliwiająca wyczyszczenie całej matrycy (zapelnienie jej pikselami o wartości tła):

```
void GLCD_cls()
{
    uint8_t tc=255;
    if(color) tc=0;
    for(uint8_t j=0; j<KS0108_SCREEN_HEIGHT/8; j++)
    {
        GLCD_goto(0,j);
        for(uint8_t i=0; i<KS0108_SCREEN_WIDTH; i++) GLCD_WriteData(tc);
    }
    GLCD_x=0; GLCD_y=0;
}
```

Proces czyszczenia tła jest szybki, gdyż naraz zmieniany jest stan aż 8 pikseli (całego bajtu).

Kolejną ważną funkcją jest funkcja rysująca pojedyncze piksele:

```
void GLCD_SetPixel(uint8_t x, uint8_t y)
{
    int8_t tmpy=y/8;
    uint8_t tmp;
    GLCD_goto(x, tmpy);
    tmp=GLCD_ReadData();
    tmp=GLCD_ReadData();
    GLCD_goto(x, tmpy);
    if(color) tmp|= _BV(y%8); else tmp&=~(_BV(y%8));
    GLCD_WriteData(tmp);
}
```

Funkcja ta wydaje się nieco skomplikowana, co wynika ze sposobu działania kontrolera. Pierwszy odczyt powoduje zatrzymanie odczytanej danej w specjalnym wewnętrznym rejestrze kontrolera. Dopiero drugi odczyt powoduje jej wystawienie na liniach *D0 – D7*, umożliwiając transmisję do mikrokontrolera. Odczyt bajtu jest niezbędny, gdyż każdy bajt zawiera stan 8 pikseli, stąd poprzedni ich stan jest konieczny, aby pozostałe piksele nie uległy modyfikacji.

Dysponując podstawowymi funkcjami, można przystąpić do funkcji rysujących znaki. Stworzone zostały dwie funkcje, jedna szybka, druga wolniejsza, ale umożliwiająca wyświetlanie tekstu pozycjonowanego co do piksela. Wyboru funkcji dokonuje się, defi-

niując odpowiednią wartość symbolu SLOW\_TEXT. Powoduje to warunkowe włączenie definicji odpowiedniej funkcji rysującej dzięki dyrektywom kompilacji warunkowej preprocesora:

```
#if SLOW_TEXT==0  
#else  
#endif
```

Funkcja rysująca szybko jest prosta:

```
void GLCD_putchar(char c)  
{  
    char *ptr=(char*)font5x8+(5*(c-32));  
    for(uint8_t i=0; i<5; i++)  
    {  
        uint8_t px=GLCD_ReadByteFromROMMemory(ptr++);  
        if(color==0) px^=0xFF;  
        GLCD_WriteData(px);  
    }  
    if(color==0) GLCD_WriteData(0xFF); else GLCD_WriteData(0x00);  
}
```

Na podstawie kodu znaku ustala ona adres jego reprezentacji bitowej w pamięci FLASH. Znaki o kodach 0 – 31 nie zostały zaimplementowane. Następnie odczytywanych jest 5 bajtów pamięci zawierających definicję znaku i wysyłane są one do pamięci video kontrolera. Na końcu dodawana jest jedna pusta kolumna, co poprawia czytelność znaków — bez niej niektóre zachodziłyby na siebie. Ponieważ w tym przypadku następuje proste kopiowanie bajtów, całość działa niezwykle szybko. Druga wersja tej funkcji:

```
void GLCD_putchar(char c)  
{  
    uint8_t orig_y=GLCD_y;  
    char *ptr=(char*)font5x8+(5*(c-32));  
    uint8_t tmp_x=GLCD_x;  
    uint8_t tmpcolor=color;  
  
    for(uint8_t i=0; i<6; i++)  
    {  
        uint8_t tmp_y=orig_y;  
        uint8_t tmp;  
        if(i!=5) tmp=GLCD_ReadByteFromROMMemory(ptr++); else tmp=0;  
        for(uint8_t y=0;y<8;y++)  
        {  
            if(tmp & 0x01) color=1; else color=0;  
            if(tmpcolor==0) color^=0x01;  
            GLCD_SetPixel(tmp_x, tmp_y);  
            tmp_y++;  
            tmp>=1;  
        }  
        tmp_x++;  
    }  
    color=tmpcolor;  
    GLCD_y=orig_y;  
}
```

początkowo działa tak samo. Jednak znak nie jest rysowany poprzez kopiowanie wzorca z pamięci FLASH do VRAM, lecz rysowany jest piksel po pikselu.



Obie funkcje wykorzystują tabelę znaków zdefiniowaną w pliku *font8x5.h*. Zmieniając definicje znaków w tej tabeli, można wyświetlać dowolne znaki.

Znaki w tej tabeli zostały zdefiniowane dosyć nietypowo. Kolejne bajty reprezentują kolejne kolumny znaku, a nie wiersze. Jest to spowodowane oszczędnością pamięci — matryca znaków ma  $5 \times 8$  pikseli, gdyby zapisywać w kolejnych bajtach wiersze definiujące znak, to zamiast 5 bajtów potrzebnych na opis znaku potrzebne byłoby aż 8 bajtów. W dodatku ze względu na organizację pamięci sterownika KS0108 zapis takich danych byłby niewygodny.

Powyższe niskopoziomowe funkcje dostępu do LCD umożliwiają napisanie kilku przydatnych funkcji wysokopoziomowych, których implementacja jest niezależna od zastosowanego LCD (pliki *graphics.h* i *graphics.c*). Pierwsze dwie to znane nam już funkcje wyświetlające łańcuchy tekstowe:

```
void GLCD_putstr_P(prog_char *txt)
{
    char ch;
    while((ch=pgm_read_byte(txt++)) GLCD_putchar(ch);
}

void GLCD_putstr(char *txt)
{
    while(*txt) GLCD_putchar(*txt++);
}
```

Powyższe funkcje wyświetlają łańcuch tekstowy znajdujący się odpowiednio w pamięci FLASH lub pamięci SRAM mikrokontrolera.

Dwie kolejne funkcje umożliwiają kreślenie linii oraz okręgów. Ich implementacja oparta jest o algorytm Bresenhama, co eliminuje konieczność przeprowadzania złożonych obliczeń:

```
void GLCD_Circle(uint8_t cx, uint8_t cy, uint8_t radius)
{
    int8_t x, y, xchange, ychange, radiusError;
    x=radius;
    y=0;
    xchange=1-2*radius;
    ychange=1;
    radiusError=0;
    while(x>=y)
    {
        GLCD_SetPixel(cx+x, cy+y);
        GLCD_SetPixel(cx-x, cy+y);
        GLCD_SetPixel(cx-x, cy-y);
        GLCD_SetPixel(cx+x, cy-y);
        GLCD_SetPixel(cx+y, cy+x);
        GLCD_SetPixel(cx-y, cy+x);
        GLCD_SetPixel(cx-y, cy-x);
```

```
GLCD_SetPixel(cx+y, cy-x);
y++;
radiusError+=ychange;
ychange+=2;
if(2*radiusError+xchange>0)
{
    x--;
    radiusError+=xchange;
    xchange+=2;
}
}
```

Funkcja kreśląca okręgi wykorzystuje ich symetryczność. W jednym kroku rysowanych jest jednocześnie aż 8 pikseli, dzięki czemu wystarczy narysować  $45^\circ$  łuku, aby otrzymać cały okrąg. Podobnie proste jest rysowanie linii:

```
void GLCD_Line(uint8_t x0, uint8_t y0, uint8_t x1, uint8_t y1)
{
    int8_t err, e2, sx, sy;
    int8_t dx=abs(x1-x0);
    int8_t dy=abs(y1-y0);
    if(x0<x1) sx=1; else sx=-1;
    if(y0<y1) sy=1; else sy=-1;
    if(dx>dy) err=dx/2; else err=-dy/2;

    while(1)
    {
        GLCD_SetPixel(x0, y0);
        if((x0==x1) && (y0==y1)) return;
        e2=err;
        if(e2>-dx)
        {
            err=err-dy;
            x0=x0+sx;
        }
        if(e2<dy)
        {
            err=err+dx;
            y0=y0+sy;
        }
    }
}
```



W obu funkcjach założono, że rozdzielcość matrycy nie przekroczy  $128 \times 128$  pikseli, co umożliwiło przyśpieszenie obliczeń i redukcję kodu funkcji poprzez zastosowanie typów 8-bitowych. Jeśli wykorzystywana matryca ma większą rozdzielcość, zachodzi konieczność zmiany typów na 16-bitowe.

W rozdziale 15., poświęconym przetwornikowi ADC, pokazane zostanie praktyczne wykorzystanie powyższych procedur w bardziej złożonym programie.



## Rozdział 19.

# Interfejs USART

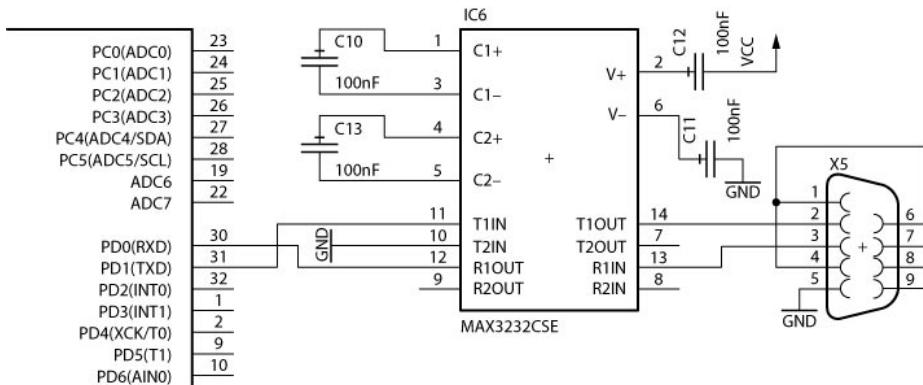
## Interfejsy szeregowe

Mikrokontrolery AVR wyposażone są w cały szereg interfejsów szeregowych — SPI (ang. *Serial Peripheral Interface*), USART (ang. *Universal Synchronous and Asynchronous serial Receiver and Transmitter*), TWI (ang. *2-wire Serial Interface*). Ten ostatni jest odpowiednikiem interfejsu I2C. Wszystkie te interfejsy umożliwiają łatwe połączenie dwóch urządzeń za pomocą stosunkowo niewielkiej liczby przewodów (od 2 do 4). Także w przeciwieństwie do interfejsów równoległych, są łatwe do sprzętowej realizacji. Ceną, jaką płacimy za prostotę elektryczną tych interfejsów, jest szybkość ich działania wynosząca do kilku Mbps. Niektóre procesory AVR wyposażone są w interfejs USB (ang. *Universal Serial Bus*), umożliwiający łatwe podłączenie mikrokontrolera do hosta, jakim może być np. komputer. Jednak ze względu na raczej skomplikowany sposób obsługi tego interfejsu zostanie on omówiony w innym miejscu książki. Spora część mikrokontrolerów AVR wyposażona jest także w interfejs CAN (ang. *Controller Area Network*), będący rozwinięciem koncepcji interfejsu RS485. O ile RS485 łatwo można zaimplementować, korzystając z dostępnego portu USART, to CAN wymaga dedykowanego sterownika — może on być wbudowany w mikrokontroler lub może wykorzystać zewnętrzny kontroler, połączony z procesorem najczęściej przy pomocy interfejsu SPI. Często wykorzystywany jest interfejs 1-wire. Mikrokontrolery AVR nie zawierają jego sprzętowej implementacji, jednakże dostępne peryferia (USART) i odrobiną kodu umożliwiają jego bardzo efektywną implementację.

Wszystkie interfejsy szeregowe mogą być emulowane programowo. Zabiera to jednak sporo czasu procesora i ogranicza maksymalną przepustowość interfejsu, stąd też lepiej projektować układ z wykorzystaniem interfejsów sprzętowych. Wiele mikrokontrolerów AVR posiada po kilka interfejsów danego typu, np. więcej niż jeden interfejs USART. Interfejsy te mogą być wykorzystywane także w mniej typowy sposób, np. do generowania skomplikowanych przebiegów zegarowych, wymaganych np. do obsługi interfejsu 1-wire, lub zupełnie nietypowo — do podłączenia przycisków i eliminacji drgań styków (ang. *Debouncing*).

# Interfejs USART

Jest to chyba najczęściej stosowany interfejs ze względu na powszechność jego występowania, łatwość użycia i uniwersalność, umożliwiającą wykorzystanie go do realizacji bardzo nietypowych zadań. Mikrokontrolery AVR posiadają zwykle kilka takich interfejsów. Sposób podłączenia interfejsu USART z komputerem klasy PC lub innym urządzeniem RS232 pokazany jest na rysunku 19.1.



**Rysunek 19.1.** Sposób podłączenia mikrokontrolera AVR z innym urządzeniem RS232. Urządzenia łączymy ze sobą tzw. kablem prostym (ang. straight cable). Jeżeli dysponujemy kablem skrzyżowanym, należy odwrotnie podłączyć piny 2 i 3 gniazda DB9

W tym typie interfejsu sygnały Rx i Tx dwóch urządzeń są skrzyżowane, tzn. sygnał Rx jednego urządzenia podłączony jest pod pin Tx drugiego, a sygnał Tx pierwszego pod sygnał Rx drugiego. Wynika z tego, że w tym typie magistrali maksymalnie możemy podłączyć ze sobą dwa urządzenia. Mikrokontrolery AVR nie implementują wszystkich sygnałów interfejsu RS232 — do dyspozycji mamy tylko sygnały Rx oraz Tx, umożliwiające asynchroniczne nadawanie i odbiór znaków. Dodatkowo interfejs USART mikrokontrolera możemy skonfigurować tak, aby transmisja odbywała się w sposób synchroniczny, z wykorzystaniem dodatkowej linii zegarowej. Dodatkowe linie DTR, DCD, DSR, RI, RTS i CTS nie są wspierane sprzętowo przez mikrokontroler, jednak w większości przypadków w niczym to nie przeszkadza.

**Ponieważ standard elektryczny interfejsu RS232 odbiega od standardu TTL, aby poprawnie połączyć dwa urządzenia, wymagany jest specjalny układ transceivera, np. MAX232.** Układ ten konwertuje poziomy logiczne TTL do poziomów panujących na magistrali RS232. W jednym układzie występują co najmniej 2 nadajniki i 2 odbiorniki, lecz zwykle wykorzystywany jest tylko jeden nadajnik i jeden odbiornik. Dopuszczalne przedziały napięć dla magistrali RS232 pokazane są w tabeli 19.1.

Układ nadajnika/odbiornika zabezpieczony jest przed zwarciem wyjść, jedyna możliwość jego uszkodzenia to podłączenie zewnętrznego napięcia do jego wyjść. Układy transceiverów RS232 zwykle zapewniają prędkość transmisji do ok. 120 – 140 kbps, ale mogą osiągać znacznie większe prędkości, sięgające ok. 1,5 Mbps. Maksymalna długość kabla łączącego dwa urządzenia na magistrali wynosi ok. 16 m, przy czym pojemność magi-

**Tabela 19.1.** Poziomy napięć na magistrali RS232

Stan logiczny	Nadajnik	Odbiornik
0	+5 – +15V	+3 – +25V
1	-5 – -15V	-3 – -25V
niezdefiniowany	—	-3 – +3V

strali nie może przekroczyć 2,5 nF. Stąd też stosując kable lepszej jakości, np. skrętkę UTP kat. 5e o pojemności ok. 17 pF/m, można wydłużyć kabel do ok. 50 m. Maksymalna długość kabla zależy także od prędkości transmisji (tabela 19.2): zmniejszając prędkość, możemy znacznie wydłużyć łączący urządzenia kabel.

**Tabela 19.2.** Zależność pomiędzy prędkością transmisji a maksymalną długością kabla. Podane długości są tylko orientacyjne, w zależności od warunków otoczenia (np. silne zakłócenia) mogą być znacznie mniejsze

Prędkość	Maksymalna długość kabla
19 200	16 m
9600	160 m
4800	320 m
2400	1000 m

Każda transmisja na magistrali RS232 rozpoczyna się nadaniem tzw. **bitu startu** (rysunek 19.2). Wartość tego bitu zawsze wynosi 0, dzięki czemu nadajnik może rozpoczęć odbieranie danych. Po biciu startu wysyłanych jest 5 – 9 bitów danych, po czym opcjonalnie wysyłany jest tzw. bit parzystości oraz 1 – 2 bity stopu. Bit parzystości dobrany jest tak, aby w zależności od konfiguracji interfejsu całkowita liczba nadanych bitów o wartości 1 była parzysta lub nieparzysta. Dzięki temu odbiornik może kontrolować poprawność przesyłanych danych, a ewentualna niezgodność sygnalizowana jest jako błąd parzystości (ang. *Parity Error*).



Bit parzystości jest opcjonalny. Jeśli mamy pewność, że transmisja nie będzie zakłócona, to możemy z niego zrezygnować, zwiększając w ten sposób przepustowość magistrali.

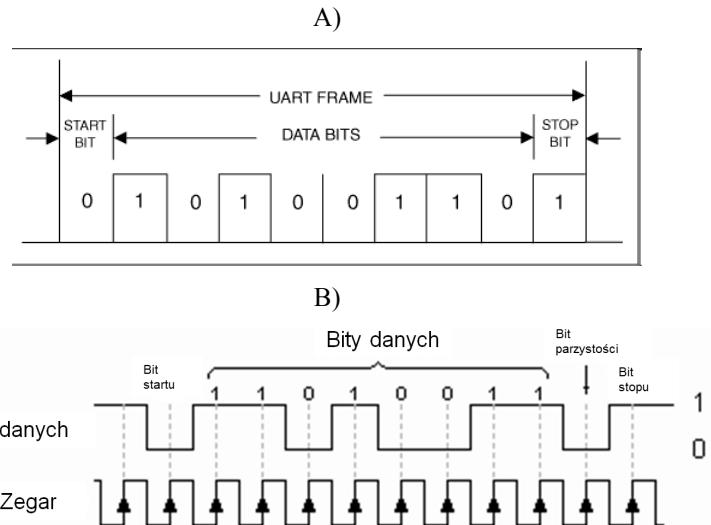
Całą transmisję kończą tzw. **bity stopu** — definiują one odcinek czasowy, w jakim magistrala musi pozostać w stanie bezczynności po nadaniu ramki danych. Jeśli w tym okresie na magistrali pojawi się jakaś transmisja, odbiornik zinterpretuje ją jako błąd ramki (ang. *Frame Error*). Błąd taki pojawia się najczęściej, kiedy nieprawidłowo jest rozpoznawany bit startu.

Jakakolwiek niezgodność uniemożliwia poprawną wymianę danych. W przypadku interfejsu asynchronicznego niezwykle istotne jest właściwe taktowanie mikrokontrolera. Sygnał taktujący interfejs UART jest pochodną sygnału zegarowego taktującego procesor. Jeśli dwa procesory będą taktowane nieznacznie różniącym się sygnałem zegarowym, może to wpływać na transmisję (rysunek 19.3).

**Rysunek 19.2.**

Struktura ramki UART.

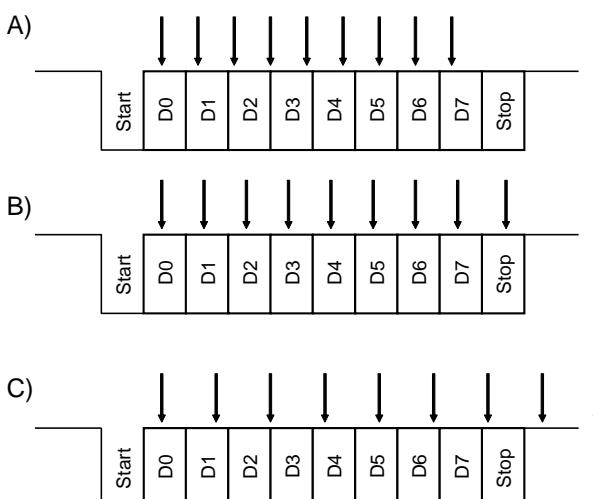
- A) transmisja  
asynchroniczna,  
B) transmisja  
synchroniczna



Wskazówka

Ponieważ kontrola błędów wbudowana w interfejs UART nie potrafi wykrywać wszystkich możliwych błędów, w przypadku kiedy interfejs ten jest wykorzystywany w zasumionym środowisku, dobrze jest wprowadzić dodatkowe mechanizmy kontrolne, np. kontrolę CRC.

Aby dwa urządzenia podłączone przez interfejs RS232 mogły pomiędzy sobą wymieniać dane, wszystkie parametry interfejsu, takie jak prędkość, liczba bitów danych, parzystość i liczba bitów stopu, muszą być identyczne.



**Rysunek 19.3.** Wpływ zegara na transmisję RS232. A) Częstotliwość zegara odbiornika jest ciut większa niż zegara nadajnika. Kolejne próbki są coraz bardziej rozsynchonizowane względem bitu startu, w efekcie bity od bitu D3 są nieprawidłowo odczytane. B) Zegary nadajnika i odbiornika są zsynchronizowane. Każdy bit jest prawidłowo odczytywany. C) Zegar odbiornika jest nieznacznie wolniejszy niż zegar nadajnika. W efekcie bity od D2 są nieprawidłowo odczytywane



Wskazówka

Transmisja asynchroniczna jest niezmiernie czuła na niewielkie nawet zmiany częstotliwości zegara taktującego, dlatego jeśli zamierzamy korzystać z interfejsu USART, powinniśmy mikrokontroler taktować ze stabilnego źródła zegara, np. oscylatora kwarcowego.

**Taktowanie mikrokontrolera z wewnętrznego generatora RC nie zapewnia wymaganej stabilności zegara, w efekcie transmisja może działać, ale prawie na pewno będzie przysparzać problemów.**

## Interfejs USART mikrokontrolera AVR

Interfejs ten jest zbudowany bardzo uniwersalnie i może pracować w kilku trybach: asynchronicznym, synchronicznym i w niektórych mikrokontrolerach AVR także w trybie emulującym interfejs SPI. W dalszej części tego rozdziału zostanie wykorzystanie tego interfejsu do realizacji różnych form transmisji szeregowej. Nawiązanie połączenia pomiędzy dwoma układami w każdym przypadku wymaga poprawnej konfiguracji interfejsu. Ewentualne niezgodności uniemożliwią nawiązanie połączenia. Interfejs USART obsługuje wiele różnych formatów transmisji danych. Kompletny pakiet, tzw. ramka (ang. *Frame*), może składać się z bitu startu, 5 – 9 bitów danych oraz 1 – 2 bitów stopu. Dane nadawane są z prędkością określaną przez programistę. Interfejs potrafi generować przerwania związane z wystąpieniem takich zdarzeń jak: odbiór bajtu danych, nadanie bajtu danych, opróżnienie bufora nadawczego. Dodatkowo interfejs może pracować w tzw. trybie wieloprocesorowym MPCM (ang. *Multi-processor Communication Mode*). W dalszej części opisane zostanie wykorzystanie tych właściwości interfejsu.

### Prędkość transmisji

Niezależnie od trybu pracy do poprawnego działania interfejsu USART niezbędny jest sygnał zegarowy taki, który transmisję danych. Zegar ten może być generowany z zegara taktującego rdzeń procesora, po podzieleniu go przez określoną w rejestrach UBRRH (ang. *USART Baud Rate Register*) i UBBCR wartość, lub być doprowadzony z zewnątrz do specjalnego wejścia procesora XCK. Ta ostatnia możliwość występuje wyłącznie w przypadku pracy interfejsu w tzw. trybie synchronicznym.

Częstym problemem spotykany przy inicjalizacji interfejsu USART jest właściwy dobór wartości rejestrów UBRR oraz bitu U2X rejestru UCSRA. W nocie katalogowej wybranego procesora, w sekcji *Examples of Baud Rate Setting* pokazane są różne wartości tych rejestrów w zależności od częstotliwości taktowania procesora oraz pożąданej prędkości działania portu UART. Jeśli jednak nie chcemy korzystać z tych tabel, to wartość rejestru UBRR możemy wyliczyć ze wzoru  $UBRR = F_{CLK}/(16 \cdot baudrate) - 1$ , jeśli bit U2X jest równy 0, lub  $UBRR = F_{CLK}/(8 \cdot baudrate) - 1$ , jeśli bit U2X ma wartość równą 1, przy czym  $F_{CLK}$  to częstotliwość taktowania procesora, a  $baudrate$  to pożądana prędkość działania interfejsu UART. Jednak nie musimy pamiętać żadnych wzorów, tu z pomocą po raz kolejny przychodzi nam biblioteka AVR-libc. W pliku nagłówkowym

<util\setbaud.h> zdefiniowane są przydatne makrodefinicje umożliwiające prawidłowe ustawienie prędkości działania interfejsu UART. Najlepiej będzie pokazać działanie tych makr na prostym przykładzie:

```
static void uart_9600()
{
    #define BAUD 9600
    #include <util/setbaud.h>
    UBRR0H = UBRRH_VALUE;
    UBRR0L = UBRLL_VALUE;
    #if USE_2X
        UCSROA |= (1 << U2X);
    #else
        UCSROA &= ~(1 << U2X);
    #endif
}
```

W powyższym kodzie zdefiniowano funkcję, której zadaniem jest ustawić prędkość portu UART na 9600 bodów. W tym celu zdefiniowano symbol BAUD, nadając mu wartość 9600, a następnie włączono plik nagłówkowy *setbaud.h*. Od tej pory można używać makrodefinicji UBRRH\_VALUE i UBRLL\_VALUE zwracających wartości, które należy wpisać odpowiednio do rejestru UBRRH i UBRLL. Powyższe makra przyjmują, że dopuszczalna tolerancja różnic prędkości pomiędzy żadaną przez użytkownika (9600 bodów) a możliwą do osiągnięcia ze względu na taktowanie procesora nie może przekroczyć 2%. Jeśli jej osiągnięcie jest niemożliwe, następuje zdefiniowanie symbolu USE\_2X, nakazującego nam ustawienie bitu U2X rejestru UCSRA.



#### Wskazówka

Aby powyższe makrodefinicje mogły poprawnie działać, niezbędne jest zdefiniowanie symbolu F\_CLK i nadanie mu wartości odpowiadającej częstotliwości taktowania procesora.

Jeśli jednak pomimo użycia bitu U2X nadal nie będzie możliwe osiągnięcie pożądanej prędkości przy określonej dokładności, w trakcie komplikacji wygenerowane zostanie ostrzeżenie.

Wszystkie obliczenia odbywają się na etapie komplikacji programu, także samo włączenie pliku *setbaud.h* nie generuje żadnego kodu. Co więcej, plik ten może być dodawany wielokrotnie w różnych częściach programu, dzięki czemu można zmieniać prędkość interfejsu USART lub w przypadku posiadania kilku takich interfejsów każdemu nadawać inną prędkość działania. Włączając wielokrotnie ten plik, w celu wyliczenia różnych wartości rejestru UBRR należy przed zdefiniowaniem nowej prędkości usunąć poprzednio zdefiniowany symbol BAUD dyrektywą:

```
#undef BAUD
```

W przeciwnym przypadku otrzymamy ostrzeżenie, że symbol BAUD został już zdefiniowany.

Założoną tolerancję błędu można zmienić, definiując symbol BAUD\_TOL, np.:

```
#define BAUD_TOL 4
```

spowoduje, że dopuszczalna tolerancja wyniesie aż 4%, co nie jest zapewne dobrym pomysłem.



**Pamiętajmy, że dla danej prędkości taktowania procesora może być niemożliwe osiągnięcie założonej prędkości portu UART. W takim przypadku nawiązanie poprawnej komunikacji z innym urządzeniem może nie być możliwe.**

**Powstający błąd nie ma żadnego znaczenia w przypadku komunikacji pomiędzy dwoma procesorami AVR pracującymi z taką samą częstotliwością zegara — w takiej sytuacji powstałe błędy w obu przypadkach są takie same i poprawna komunikacja jest możliwa.**

## Formaty ramek danych

Przy pomocy interfejsu USART można wygenerować wszystkie 30 kombinacji formatów ramek. Każda ramka zaczyna się bitem startu, który nie podlega konfiguracji. Po nim następuje 5 – 9 bitów danych. Długość pola danych określają bity UCSZ 0 – 2, rejestrów UCSRC i UCSRB. Domyślna wartość tych rejestrów określa 8-bitowe pole danych. W większości przypadków ta domyślna wartość nie musi być zmieniana. W pewnych sytuacjach wykorzystywane są ramki 9-bitowe, co zostanie pokazane w dalszej części rozdziału. W przypadku ramek krótszych niż 8 bitów nie wszystkie bity rejestru UDR są wykorzystywane. W przypadku ramek 9-bitowych najstarszy bit znajduje się w rejestrze UCSRB — bity RXB8 i TXB8. **Wartość tych bitów musi być określona przez odczytem/zapisem rejestru UDR.**

Po określeniu wielkości pola danych należy włączyć lub wyłączyć generowanie bitu parzystości poprzez właściwą konfigurację bitów UPM1 – 0 rejestru UCSRC. Ich domyślna wartość związana jest z brakiem generowania parzystości. Wartość 2 określa konfigurację, w której bit parzystości jest dobierany tak, aby liczba bitów równych jeden była parzysta, a dla wartości 3 ich liczba była nieparzysta. Ostatnim elementem jest konfiguracja liczby używanych bitów stopu przy pomocy bitu USB5 rejestru UCSRC. Jeśli ma on wartość 1, generowane są 2 bity stopu, jeśli ma wartość 0, generowany jest tylko jeden bit stopu.

## Nadajnik i odbiornik USART

Nadajnik i odbiornik interfejsu USART mogą być używane niezależnie od siebie. Ich włączenie powoduje przejęcie kontroli nad związanymi z nimi wyprowadzeniami I/O procesora. W efekcie wartość wpisana do odpowiednich bitów rejestrów PORT i DDR jest bez znaczenia. Jednak nawet po włączeniu nadajnika lub odbiornika stan tych pinów nadal może być odczytywany z rejestru PIN. Nadajnik i odbiornik włącza się poprzez ustawienie bitów TXEN i RXEN rejestru UCSRB. Ustawienie tych bitów powoduje natychmiastowe włączenie odpowiednich podsystemów. Wyzerowanie bitu RXEN również ma skutek natychmiastowy, jednak wyzerowanie bitu TXEN powoduje zablokowanie nadajnika dopiero po zakończeniu bieżącej transmisji.



Wskaźówka

Odbiornik i nadajnik należy odblokować dopiero na samym końcu procesu inicjalizacji interfejsu USART. Wcześniej należy określić prędkość transmisji, tryb pracy interfejsu, format ramki oraz odblokować/zablokować odpowiednie przerwania.

Rejestry danych (UDR) nadajnika i odbiornika są zbuforowane. Po zakończeniu odbioru ramki danych z magistrali są one kopiowane do rejestru UDR, czemu towarzyszy ustawienie bitu RXC (ang. *USART Receive Complete*). W tym czasie możliwy jest odbiór kolejnej ramki. Jednak przed zakończeniem jej odbioru poprzednią wartość z rejestru UDR koniecznie trzeba odczytać. W przeciwnym przypadku dojdzie do nadpisania danych, co jest sygnalizowane poprzez ustawienie bitu DOR (ang. *Data OverRun*) rejestru UCSRA. Nieprawidłowy odbiór danych może wiązać się także z ustawieniem bitów UPE (ang. *USART Parity Error*) w przypadku błędu parzystości oraz bitu FE (ang. *Frame Error*) w przypadku błędu ramki — występuje on, kiedy odbierane bity stopu nie mają wartości 1.



Wskaźówka

Odczytu stanu błędów z rejestru UCSRA należy dokonać przed odczytem rejestru UDR. Odczyt UDR powoduje wyzerowanie informacji o błędach!

Podobnie jak odbiornik, także rejestr nadajnika posiada własny, zwykły dwustopniowy, bufor. Do rejestru UDR można zapisywać w sytuacji, w której bit UDRE (ang. *USART Data Register Empty*) rejestru UCSRA jest ustawiony. Jego wyzerowanie następuje po wypełnieniu całego bufora. Drugą flagą związaną z nadajnikiem jest flaga TXC (ang. *USART Transmit Complete*). Jest ona ustawiona, gdy nadawanie bajtu zostało zakończone, a w buforze nadajnika nie ma więcej danych.

Wszystkie powyższe bity kasowane są automatycznie po wejściu do odpowiedniej procedury obsługi przerwania lub po zmianie stanu bufora nadajnika/odbiornika.

Korzystając z powyższych informacji, możemy napisać funkcje odpowiedzialne za nadawanie i odbiór bajtów z wykorzystaniem interfejsu USART:

```
void usart_send(uint8_t data)
{
    while(!(UCSRA & _BV(UDRE0)));
    UDR0=data;
}
```

Powyższa funkcja sprawdza, czy bufor nadajnika jest gotowy na przyjęcie kolejnego bajtu. Jeśli tak, to umieszcza wartość ze zmiennej data w buforze nadajnika. W przeciwnym przypadku funkcja czeka, aż zwolni się miejsce. Zauważmy, że zamiast flagi UDRE można zastosować flagę TXC, lecz w tym przypadku traci się korzyść z podwójnego buforowania rejestru nadajnika.

Podobnie można zrealizować funkcję odbierającą bajt danych:

```
uint8_t usart_Receive( )
{
    while(!(UCSRA & _BV(RXC0)));
    return UDR0;
}
```

Powyższa funkcja czeka na odebranie znaku, a następnie go zwraca.



Zauważmy, że zapisując coś do rejestru UDR, dokonuje się zapisu do bufora nadajnika, odczytując UDR, zawsze odczytuje się bufor odbiornika.

## Przerwania USART

Zdarzenie, takie jak nadanie lub odebranie znaku oraz opróżnienie bufora nadawczego, może generować przerwanie. Umożliwia to łatwą realizację reakcji na zachodzące zdarzenia. W AVR-libc poszczególnym zdarzeniom towarzyszy wywołanie przerwań o następujących wektorach:

- ◆ USARTn\_RXC\_vect, USARTn\_RX\_vect lub UART\_RX\_vect — przerwanie generowane po odbiorze nowego znaku przez odbiornik.
- ◆ USARTn\_TXC\_vect, USARTn\_TX\_vect lub UART\_TX\_vect — przerwanie generowane po całkowitym opróżnieniu rejestru i bufora nadajnika.
- ◆ USARTn\_UDRE\_vect, UARTn\_UDRE\_vect — przerwanie generowane po wysłaniu znaku przez nadajnik, w efekcie czego zwolniło się miejsce w jego buforze.

Nazwa generowanego wektora przerwań zależy od typu procesora. Odpowiednie przerwania należy odblokowywać poprzez ustawienie flag RXCIE, TXCIE, UDRIE rejestru UCSRB.

## Przykłady

Poniżej pokazane zostaną przykłady związane z najczęstszymi sposobami wykorzystania interfejsu USART, tj. do połączenia mikrokontrolera z komputerem oraz do realizacji połączeń pomiędzy mikrokontrolerami.

## Połączenie mikrokontroler – komputer PC

Najprostszym, a jednocześnie najczęstszym sposobem wykorzystania portu USART jest połączenie mikrokontrolera z komputerem PC. Połączenie takie elektrycznie jest realizowane tak, jak pokazano na rysunku 19.1. Do monitorowania transmisji na komputerze PC możemy użyć dostarczonego z systemem operacyjnym MS Windows programu *Typer Terminal*, lecz prawdopodobnie lepszym rozwiązaniem będzie zainstalowanie dostępnego za darmo programu *RealTerm* — można go pobrać ze strony projektu <http://realterm.sourceforge.net/>.

### Przykład — echo

Naszą przygodę z RS232 rozpoczęliśmy od napisania prostego programu, którego efektem działania będzie odsyłanie do komputera PC znaków, które z niego otrzymał — tzw. *echo*.

```

#include <stddef.h>
#include <avr\io.h>

static void uart_9600()
{
    #define BAUD 9600
    #include <util/setbaud.h>
    UBRRH = UBRRH_VALUE;
    UBRRL = UBRRL_VALUE;
    #if USE_2X
        UCSR1A |= (1 << U2X);
    #else
        UCSR1A &= ~(1 << U2X);
    #endif
}

static void uart_init()
{
    uart_9600();
    UCSR1C=_BV(UCSZ11) | _BV(UCSZ10) | _BV(USBS1); //8 bitów danych + 2 bity stopu
    UCSR1B=_BV(TXEN) | _BV(RXEN); //bez parzystości
}

uint8_t rec_uart()
{
    while(!(UCSR1A & _BV(RXC)));
    return UDR1;
}

void send_uart(uint8_t byte)
{
    while (!(UCSR1A & _BV(UDRE))); //Zaczekaj, aż bufor będzie pusty
    UDR1=byte;
}

int main()
{
    uart_init();
    while(1)
    {
        send_uart(rec_uart());
    }
}

```

W tym celu zdefiniowano kilka procedur. Pierwsza z nich, `uart_9600()`, ustawia predkość działania portu UART na 9600 bps, po tym następuje inicjalizacja portu — do rejestrów kontrolnych UCSR1C i UCSR1B wpisywana jest wartość inicjująca port w trybie 8 bitów danych, bez bitu parzystości + 2 bity stopu, po czym odblokowywany jest odbiornik i nadajnik.

Za odebranie 8-bitowej danej przesłanej z komputera do mikrokontrolera odpowiedzialna jest funkcja `rec_uart()`. Czeka ona, aż bit RXC rejestru UCSRA zostanie wyzerowany, co sygnalizuje, że w rejestrze UDR znajduje się dana do odczytania. Z kolei za nadawanie odpowiedzialna jest funkcja `send_uart()`, która przed zapisaniem znaku do bufora



Wskazówka

Odblokowanie nadajnika lub odbiornika automatycznie powoduje przejęcie kontroli przez interfejs UART nad pinem *RxD/TxD* związanym z tym portem. Jego normalna funkcja jako pinu *I/O* ogólnego przeznaczenia jest blokowana, a ustawienia związanego z nim rejestru DDR i PORT są bez znaczenia.

nadawczego sprawdza, czy jest w nim miejsce; jeśli tak, to umieszcza w nim znak do nadania. Znak ten zostanie odebrany przez komputer. W rejestrze istnieje jeszcze flaga TXC, sygnalizująca, że bufor nadawczy jest całkowicie pusty i nie ma w nim nic do nadania. Powyższa funkcja nie korzysta z tej flagi, gdyż w wielu procesorach AVR bufor nadawczy składa się z co najmniej 2 bajtów — już po nadaniu jednego bajtu zwalnia się więc miejsce na kolejny. Sprawdzając flagę TXC zamiast UDRE, niepotrzebnie czekalibyśmy aż do całkowitego opróżnienia bufora nadajnika.



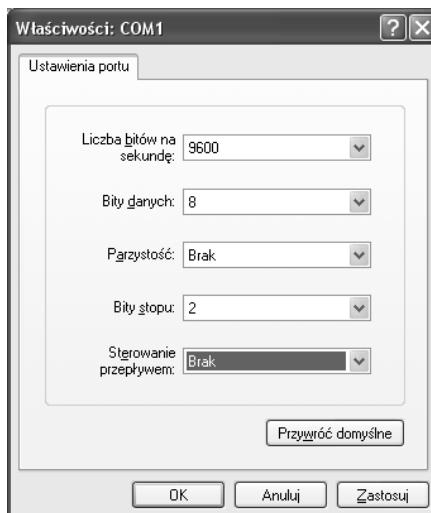
Wskazówka

Pamiętaj, że rejesty UDR nadajnika i odbiornika to dwa różne rejesty, chociaż zajmują ten sam fizyczny obszar pamięci *I/O* mikrokontrolera.

Dlatego przy zapisywaniu czegoś do rejestru UDR dane znajdują się w buforze nadajnika, natomiast podczas odczytu UDR dane odczytane zostaną z bufora odbiornika.

Po połączeniu zaprogramowanego procesora z komputerem możemy sprawdzić działanie powyższego programu. W tym celu należy uruchomić program *Typer Terminal* lub dowolny inny terminal RS232. Po wybraniu portu, który wykorzystujemy do połączenia, należy ustawić jego parametry na takie same, jakie ustalone zostały w mikrokontrolerze AVR — rysunek 19.4.

**Rysunek 19.4.**  
Konfiguracja parametrów transmisji RS232 na komputerze PC



Wskazówka

Pamiętaj! Jakakolwiek niezgodność parametrów portu USART komputera i mikrokontrolera uniemożliwi transmisję.

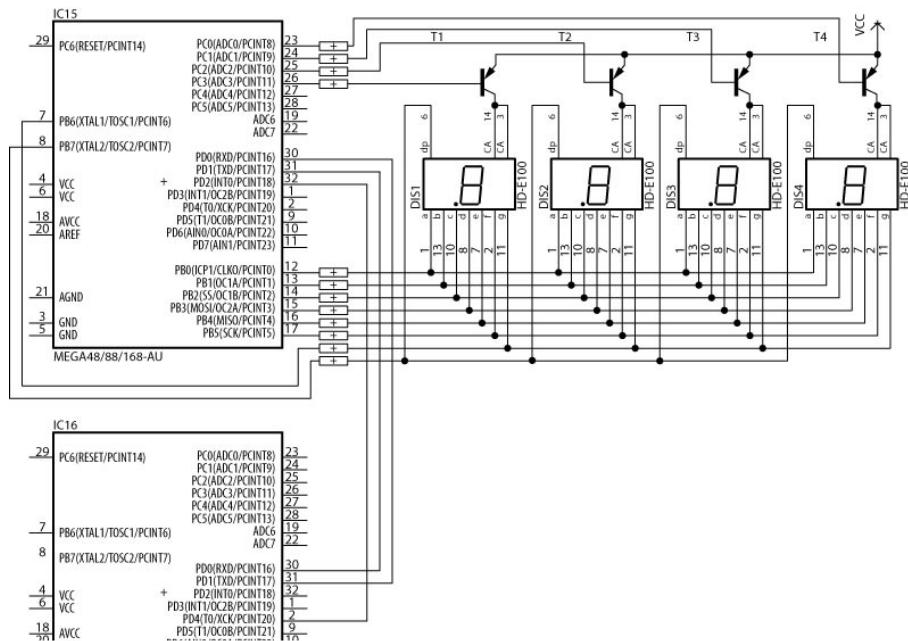
Po połączeniu *Hiper Terminala* z właściwym portem COM i naciśnięciu jakiegokolwiek klawisza powinniśmy widzieć w oknie terminala *echo*. Jeśli tak nie jest, to należy sprawdzić:

- ◆ czy F\_CLK zawiera prawidłową częstotliwość taktowania procesora;
- ◆ połączenia Transceiver RS232 i kabla *null-modem* łączącego z komputerem;
- ◆ ustawione parametry portu COM.

## Połączenie mikrokontroler – mikrokontroler

Połączenie mikrokontroler – mikrokontroler może być wykorzystywane do transmisji danych ze stosunkowo dużą prędkością, na odległość przekraczającą możliwości innych interfejsów szeregowych (SPI, I2C). W tym trybie połączenie pomiędzy mikrokontrolerami można zrealizować przy pomocy *transceiverów* MAX232, co zapewnia zgodność takiego połączenia ze standardem RS232, lub bezpośrednio, łącząc ze sobą odpowiednie wyprowadzenia interfejsu USART (jest to tzw. RS232-TTL). W tym drugim przypadku możliwe do osiągnięcia odległości będą zazwyczaj mniejsze, poza tym wyprowadzenia procesora nie będą chronione. Taki sposób połączenia zalecany jest do realizacji wymiany danych w ramach mikrokontrolerów znajdujących się w jednym urządzeniu (zamkniętych w jednej obudowie). Jego niewątpliwa zaletą jest prostota i niska cena realizacji.

Przykładowy schemat takiego połączenia pokazany został na rysunku 19.5.



**Rysunek 19.5.** Połączenie pomiędzy dwoma mikrokontrolerami AVR. Pin RxD należy połączyć z pinem TxD drugiego mikrokontrolera. Opcjonalnie można także połączyć piny XCK obu mikrokontrolerów, co umożliwi realizację transmisji synchronicznej

Wykorzystanie połączonych bezpośrednio interfejsów USART jest proste, lecz ciągle do poprawnej transmisji wymagane jest zapewnienie odpowiedniej stabilności zegarów obu mikrokontrolerów. Stabilność wewnętrznych generatorów RC, wbudowanych w mikrokontroler, jest niewystarczająca, dlatego należy użyć zewnętrznych kwarców. Problem ze stabilnością częstotliwości występuje wyłącznie w przypadku wykorzystania asynchronicznego trybu pracy USART. Zauważmy, że przy bezpośrednim połączeniu dwóch mikrokontrolerów AVR nie ma znaczenia, jakie kwarce zastosujemy, pod warunkiem że będą one takie same. Błąd prędkości transmisji związany z niemożliwością wyliczenia dokładnej wartości rejestru UBRR będzie taki sam w przypadku obu mikrokontrolerów, w efekcie będą one mogły doskonale ze sobą współpracować. Dzięki temu nie zachodzi potrzeba wykorzystywania tzw. kwarców „przyjaznych” dla USART.

Istnieje jednak inną możliwość realizacji połączenia ze sobą dwóch mikrokontrolerów — wykorzystanie transmisji synchronicznej. Do tej pory wykorzystywaliśmy interfejs USART w tzw. trybie asynchronicznym. W trybie tym transmisja ramki danych była synchronizowana bitem startu, a procesory musiały być taktowane ze stabilnego źródła zegarowego. Inaczej sprawa wygląda w trybie synchronicznym.

### Tryb asynchroniczny pracy USART

W tym trybie interfejs USART synchronizowany jest zewnętrznym sygnałem zegarowym doprowadzonym do wejścia  $XCK$  procesora. Dzięki temu nie ma znaczenia stałość wewnętrznego oscylatora. Jedyną wadą tego rozwiązania jest konieczność połączenia obu mikrokontrolerów dodatkowym przewodem, po którym przesyłany jest sygnał zegarowy. Włączenie tego trybu następuje po ustawieniu bitu UMSL (ang. *USART Mode Select*) rejestru UCSRC. W nowszych procesorach AVR występują bity UMSEL0 i UMSEL1. Ich kombinacja, oprócz trybu synchronicznego, umożliwia wybór trybu pracy USART emulującego interfejs SPI, co zostanie pokazane w rozdziale 20. Po włączeniu trybu synchronicznego interfejs USART przejmuje kontrolę nad wyprowadzeniem  $XCK$  procesora — normalna funkcja tego wyprowadzenia jako portu  $I/O$  zostanie забlokowana. W tym trybie jeden z procesorów (nazywany *master*) zajmuje się generowaniem przebiegu zegarowego, taktującego układ USART drugiego z procesorów (nazywanego *slave*). O tym, który z procesorów będzie generował przebieg zegarowy, decyduje bit rejestru DDR związanego z pinem, przez który wyprowadzony jest sygnał  $XCK$ . Jeśli bit ten ma wartość 1, to procesor staje się układem nadrzędnym, generującym przebieg zegarowy dla pozostałych układów. Nadanie temu bitowi wartości 0 powoduje, że procesor staje się układem *slave*, a interfejs USART jest taktowany przebiegiem doprowadzonym do pinu  $XCK$ .



W przypadku układów *slave* nie ma znaczenia ustawienie rejestru UBRR określającego prędkość transmisji.

Prędkość w tym trybie zawsze określana jest przez zegar taktujący generowany przez układ *master*. Należy tylko pamiętać, że zegar ten nie może być szybszy niż zegar taktujący urządzenie *slave* podzielony przez 4.

W chwili wybrania trybu synchronicznego na pinie  $XCK$  natychmiast pojawia się sygnał zegarowy o częstotliwości określonej wartością rejestru UBRR układu master. Sygnał

ten jest niezależny od aktywności na pinach *RxD* i *TxD* mikrokontrolera. W efekcie ten tryb pracy można wykorzystać także do generowania przebiegu zegarowego o programowalnej częstotliwości.

W tym trybie można jeszcze skonfigurować zbocza sygnału zegarowego, na których odbywa się próbkowanie sygnału. Można tego dokonać, konfigurując bit *UCPOL* (ang. *Clock Polarity*) rejestru *UCSRC*, jednak w trybie pracy USART nie ma to większego znaczenia. Bit ten wykorzystuje się w trybach pracy emulujących interfejs SPI.

Przejdzmy teraz do napisania programu, który wykorzystuje interfejs USART pracujący w trybie synchronicznym do przesłania danych do innego mikrokontrolera AVR. Schemat układu pokazany został wcześniej, na rysunku 19.5. Oba interfejsy USART zostały połączone ze sobą przy pomocy skrosowanych sygnałów *TxD* i *RxD* oraz sygnału *XCK*. Dodatkowo należy pamiętać, aby oba układy miały wspólną mase.

Aplikacja zostanie podzielona na dwie części (plik *USART-AVR-AVR*) — część działającą na mikrokontrolerze *master*, która będzie wysyłać w 1-sekundowych odstępach inkrementowaną wartość zmiennej, oraz drugą część, odpowiedzialną za wyświetlenie przesyłanych danych na wyświetlaczu LED. Sama realizacja wyświetlania multiplekso-wego pokazana została we wcześniejszych rozdziałach.

Aby móc wymieniać dane pomiędzy mikrokontrolerami, oprócz połączenia ich interfejsów USART niezbędne jest także zaprojektowanie odpowiedniego protokołu wymiany danych. Ramka danych wysyłana do urządzenia *slave* będzie miała następującą postać:

- ◆ 1 bajt określający długość ramki,
- ◆  $n$  bajtów danych,
- ◆ kod CRC poprzedzających bajtów.

Dzięki dodaniu do transmitowanych danych CRC istnieje możliwość detekcji ewentualnych błędów. Ramki zawierające niepoprawne CRC będą przez urządzenie *slave* odrzucane.

Zacznijmy od programu realizowanego w układzie *master*. Pierwszą funkcją jest funkcja odpowiedzialna za inicjalizację interfejsu USART:

```
void UART_master_init()
{
    uart_9600();
    UCSRB= _BV(RXEN0) | _BV(TXEN0);
    UCSRC= _BV(USBS0) | _BV(UCSZ01) | _BV(UCSZ00); //8 bitów danych, 2 bity stopu,
                                                       //bez parzystości, transmisja asynchroniczna
    DDRD|= _BV(PD4); //Układ generuje zegar dla slave
    UCSRC|= _BV(UMSEL00); //Tryb synchroniczny
}
```

Interfejs inicjowany jest w trybie synchronicznym, jednak cały program równie dobrze będzie pracował w trybie asynchronicznym (po zapewnieniu odpowiedniej stabilności sygnału zegarowego). Ponieważ układ *master* generuje sygnał zegarowy dla układu *slave*,

przy pomocy znanej nam już funkcji `uart_9600()` ustalana jest prędkość transmisji na 9600 bodów. Dodatkowo pin `XCK` ustawiany jest jako wyjście, dzięki czemu układ staje się generatorem zegara dla układu *slave*.

Kolejną potrzebną funkcją jest funkcja wysyłająca jeden bajt danych:

```
void uart_sendchar(uint8_t c)
{
    while(!(UCSR0A & (1<<UDRE0)));
    UDR0=c;
}
```

Funkcja ta testuje flagę `UDRE` w celu sprawdzenia, czy w buforze nadajnika jest miejsce na kolejny bajt danych.

Ostatnią potrzebną funkcją jest funkcja odpowiedzialna za wysłanie kompletnej ramki danych, zawierającej jej długość, dane i wyliczoną wartość CRC:

```
void uart_Send_Frame(char *buf, uint8_t n)
{
    uint8_t crc=_crc_ibutton_update(0, n);
    uart_sendchar(n);
    while(n--)
    {
        uart_sendchar(*buf);
        crc=_crc_ibutton_update(crc, *buf);
        buf++;
    }
    uart_sendchar(crc);
}
```

CRC wyliczane jest przy pomocy funkcji udostępnianych przez bibliotekę AVR-libc, znajdujących się w pliku nagłówkowym `<util/crc16.h>`. Dane do wysłania wskazywane są przez wskaźnik `buf`, a liczbę bajtów do wysłania określa zmienna `n`.

Ostatnim elementem programu *mastera* jest definicja funkcji `main` odpowiedzialnej za cykliczne wysyłanie ramek danych do układu *slave*:

```
int main()
{
    char bufor[BUF_MAX];
    uint16_t licznik=0;

    UART_master_init();

    while(1)
    {
        _delay_ms(1000);
        sprintf(bufor,"%04d",licznik);
        for(uint8_t i=0;i<4;i++) bufor[i]-='0';
        uart_Send_Frame(bufor, 4);
        licznik++;
    }
}
```

Przy pomocy funkcji `sprintf` zmienna `licznik` konwertowana jest nałańcuch tekstowy, składający się z 4 cyfr. Jeśli zmienna ma mniej cyfr, jest dopełniana z lewej cyframi zero. Ponieważ `slave` spodziewa się cyfr, a nie ich kodów ASCII, w pętli od poszczególnych znaków odejmowany jest kod znaku 0, dzięki czemu w buforze znajdą się cyfry 0 – 9, a nie odpowiadające im znaki kodu ASCII. Tak przekonwertowany łańcuch wysyłany jest do układu `slave` przy pomocy zdefiniowanej wcześniej funkcji `uart_Send_Frame`.

Realizacja programu `slave` będzie nieco bardziej skomplikowana. Aby pokazać możliwość wykorzystania przerwań USART, odbiór kolejnych znaków będzie realizowany asynchronicznie w stosunku do wykonywania programu, przy pomocy przerwania generowanego przez odbiornik USART. Lecz podobnie jak poprzednio pisanie programu rozpoczęniemy od zdefiniowania funkcji inicjującej interfejs:

```
void USART_slave_init()
{
//uart_9600;
    UCSR0C=_BV(USBS0) | _BV(UCSZ01) | _BV(UCSZ00); //8 bitów danych, 2 bity stopu,
                                                       //bez parzystości, transmisja asynchroniczna
    UCSR0C|= _BV(UMSEL00);
    UCSR0B=_BV(RXEN0) | _BV(TXEN0) | _BV(RXCIE0); //Odblokuj przerwania odbiornika
}
```

Ponieważ odbiornik jest taktowany przez sygnał generowany przez układ *master*, nie ma potrzeby ustalania wartości rejestrów UBRR, stąd też odpowiednia funkcja została zakomentowana. W przypadku wykorzystania transmisji asynchronicznej należy ją odkomentować. Analogicznie, po wybraniu parametrów pracy dokładnie takich samych jak w przypadku nadajnika interfejs przełączany jest w tryb synchroniczny, w którym taktowany będzie sygnałem dostarczonym na pin *XCK*. Dodatkowo w trakcie inicjalizacji włączane są przerwania odbiornika USART. Za cały odbiór danych odpowiedzialna będzie odpowiednia funkcja obsługująca to przerwanie:

```
volatile char Bufor[BUF_MAX];
volatile uint8_t status;

ISR(USART_RX_vect)
{
    static uint8_t bufpos, n, crc;

    if(status) return; //Błąd - poprzednia ramka jeszcze nieobsłużona

    if(n==0)
    {
        n=UDR0;
        bufpos=0;
        crc=_crc_ibutton_update(0, n);
        return;
    }

    if(bufpos<n)
    {
        Bufor[bufpos]=UDR0;
        crc=_crc_ibutton_update(crc, Bufor[bufpos++]);
    }
}
```

```
else if(crc==UDR0)
{
    status=n;
    n=0;
}
}
```

Po odebraniu bajtu danych wywoływana jest procedura obsługi przerwania o wektorze USART\_RX\_vect. Pierwszym jej elementem jest sprawdzenie zmiennej status. Jeśli jej wartość jest różna od 0, świadczy to o tym, że poprzednia ramka danych nie została jeszcze zinterpretowana przez program. Kolejna zostanie więc odrzucona, aby nie doprowadzić do nadpisania danych znajdujących się w buforze odbiorczym — w tablicy Bufor. Po odebraniu pierwszego bajtu ( $n==0$ ) inicjowane są zmienne pomocnicze, a zmiennej  $n$  nadawana jest długość odbieranej ramki danych. Kolejne transmitowane bajty trafiają do bufora, aż zostanie przetransmitowany bajt zawierający CRC. Jeśli wyliczone CRC i CRC nadane przez *master* zgadzają się, zmiennej status zostanie przypisana liczba odebranych bajtów, które znajdują się w zmiennej Bufor.

Pozostaje jeszcze napisanie funkcji main:

```
int main()
{
    Timer0Init();
    USART_slave_init();
    sei();
    while(1)
    {
        if(status)
        { //Odebrano pakiet
            memcpy((char*)LEDDIGITS, (char*)Bufor, LEDDISPNO);
            status=0; //Pakiet obsłużony
        }
    }
}
```

Funkcja ta inicjuje multipleksowe wyświetlanie cyfr na wyświetlaczu LED (funkcja Timer0Init), port USART (funkcja USART\_slave\_init()), a następnie w nieskończonej pętli oczekuje na odebranie poprawnej ramki danych. Sygnalizowane jest to przez nadanie zmiennej status wartości różnej od 0, będącej liczbą odebranych danych. W takiej sytuacji dane kopowane są z bufora odbiornika (zmienna Bufor) do tablicy zawierającej wyświetlane cyfry (tablica LEDDIGITS). Po skopiowaniu danych zmiennej status nadawana jest wartość 0, co sygnalizuje procedurze obsługi przerwania USART, że może odebrać kolejną ramkę danych.

## RS485

RS485 jest interfejsem wykorzystującym różnicowe przesyłanie sygnałów, co zapewnia mu wysoką odporność na zakłócenia oraz duży zasięg, sięgający 1200 m przy prędkości transmisji 100 kbitów/s; przy krótszym zasięgu można uzyskać prędkość przesyłania danych sięgającą 10 Mbitów/s. Dodatkową zaletą interfejsu RS485 jest możliwość

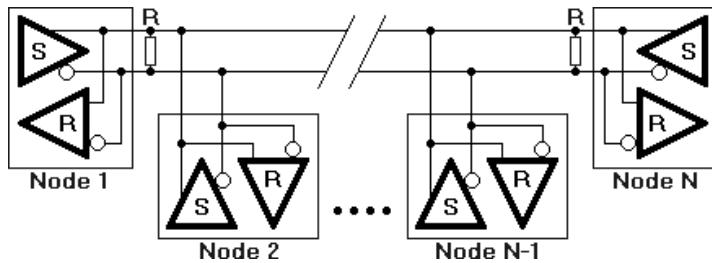
przyłączenia do wspólnej magistrali danych do 32 urządzeń<sup>1</sup>. Typowo magistrala pracuje w trybie *half-duplex*, co umożliwia wykorzystanie tylko dwóch przewodów sygnałowych oraz masy. W sytuacji kiedy mamy tylko 2 urządzenia lub jedno urządzenie *master* i wiele urządzeń *slave*, możliwe jest połączenie *full-duplex*, wykorzystujące 4 linie sygnałowe (osobną linię nadawania i odbierania) oraz przewód masy. Interfejs ten nie został stworzony z myślą o sprzętowej obsłudze układów typu *multimaster*. Takie konfiguracje da się zrealizować na poziomie protokołu transmisji lub po modyfikacji elektrycznej interfejsu RS485, polegającej na wprowadzeniu, podobnie jak w magistrali typu *CAN Bus*, stanu recesywnego. Dzięki temu możliwe jest wykrywanie kolizji na magistrali, jednak nie jest to już typowa magistrala RS485. Budowę typowej magistrali RS485 pokazano na rysunku 19.6.

**Rysunek 19.6.**

Typowa struktura magistrali RS485.

Node — kolejne urządzenia podłączone do magistrali.

R — rezystory terminujące o wartości 120 omów, połączone na końcach magistrali



Interfejs składa się z dwóch linii oznaczonych literami A i B. W stanie spoczynku potencjał linii A jest wyższy o co najmniej 200 mV od potencjału linii B (tabela 19.3). Odbiornik interpretuje taki stan jako wysoki. Jeśli potencjał linii B jest niższy o co najmniej 200 mV od potencjału linii A, to odbiornik interpretuje stan magistrali jako niski. Jeśli różnica potencjałów pomiędzy liniami A i B jest mniejsza niż 200 mV, to stan magistrali jest nieokreślony. Aby taka sytuacja nie miała miejsca, potrzebne są rezystory polaryzujące, utrzymujące właściwą polaryzację linii A i B w sytuacji, kiedy żaden nadajnik nie jest włączony (rysunek 19.7).



Rezystory polaryzujące umieszcza się tylko w jednym punkcie magistrali.

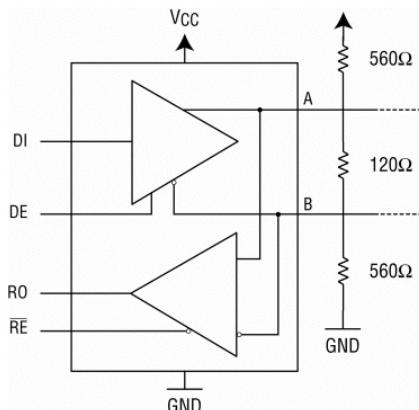
**Tabela 19.3.** Interpretacja stanów magistrali RS485

Napięcie A-B	Stan logiczny
>=200 mV	Wysoki 1
-200 – 200 mV	Nieokreślony
<= -200 mV	Niski 0

Mikrokontrolerów AVR nie można bezpośrednio łączyć z magistralą RS485. Aby takie połączenie było możliwe, należy zastosować *transceiver*, np. popularny i tani 75176 lub MAX485. Transceiver konwertuje poziomy logiczne wysyłane przez procesor do standardów magistrali RS485, zapewnia także odporność na zwarcia oraz prawidłowy

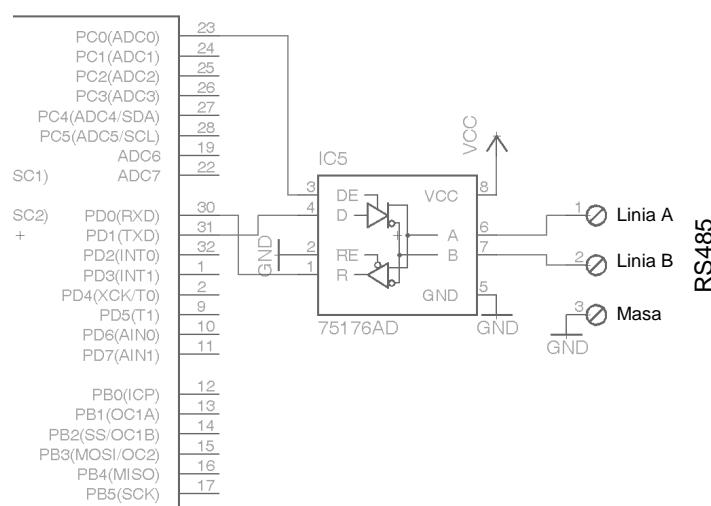
<sup>1</sup> Istnieją transceivery wnoszące  $1/2$ ,  $1/4$ , i  $1/8$  standardowego obciążenia, umożliwiające jednoczesne przyłączenie do sieci odpowiednio 64, 128 lub 256 urządzeń.

**Rysunek 19.7.**  
Schemat podłączenia  
rezystorów  
polaryzujących  
linie A i B



odbiór sygnałów różnicowych. Ponieważ na magistrali w danej chwili może być aktywny tylko jeden nadajnik, każdy transceiver RS485 ma dodatkowe wejścia odblokowujące układ nadajnika (*DE*) i odbiornika (*RE*). W przypadku kiedy pin *DE* jest w stanie niskim, nadajnik jest zablokowany, a wyjścia A i B są w stanie wysokiej impedancji. W przeciwieństwie do nadajnika odbiornik może być stale włączony, dzięki temu nie musimy sterować sygnałem *RE* — możemy go na stałe podpiąć do masy. Mikrokontroler łączymy z transceiverem tak, jak pokazano na rysunku 19.8.

**Rysunek 19.8.**  
Połączenie  
mikrokontrolera AVR  
z transceiverem RS485



Niestety, mikrokontrolery AVR nie dysponują żadnym sprzętowym mechanizmem sterowania nadajnikiem. Tę funkcję musimy zrealizować programowo, sterując nadajnikiem z dowolnego pinu *I/O* mikrokontrolera — w tym przypadku z pinu *PC0*.



**Wskazówka**

Stosunkowo często popełnianym błędem jest detekcja kolizji na magistrali poprzez porównanie nadawanych danych z odbieranymi. Z elektrycznej specyfikacji magistrali wynika, że odbiornik zazwyczaj odbiera lokalnie transmitowane dane, w efekcie taka kontrola błędów nie jest w stanie w większości przypadków wykryć kolizji.

## Tryb MPCM

Tryb MPCM pracy interfejsu UART (ang. *Multi-processor Communication Mode*) jest specjalnym trybem pracy odbiornika UART, umożliwiającym sprzętowy podział otrzymywanych ramek na dwa typy — **ramki danych** i **ramki adresu**. W tym trybie procesor ignoruje nadchodzące ramki danych, a odbierane są wyłącznie ramki adresu. Po odebraniu takiej specjalnej ramki i zidentyfikowaniu, że dalsza transmisja dotyczy danego węzła sieci, oprogramowanie powinno przełączyć UART w tryb normalny, co umożliwia odbiór ramek danych. Dzięki trybowi MPCM procesor nie musi odbierać ramek, które nie są do niego zaadresowane, co oszczędza jego zasoby. Oczywiście, ten tryb pracy ma sens wyłącznie w sytuacji, w której do wspólnej magistrali podłączonych jest jednocześnie wiele odbiorników o różnych adresach — stąd też ten tryb zostaje omówiony przy okazji omawiania interfejsu RS485. Dla typowych interfejsów RS232 na magistrali istnieją tylko dwa urządzenia, nie ma więc potrzeby korzystania z trybu MPCM. W przypadku magistrali RS485 zazwyczaj występuje kilka do kilkunastu urządzeń *slave*, w efekcie wykorzystanie trybu MPCM eliminuje konieczność przetwarzania transmitowanych danych, które nie są przeznaczone dla danego urządzenia.

Ramki adresu różnią się od ramek danych znaczeniem jednego bitu. Jeśli długość pola danych wynosi 5 – 8 bitów, pierwszy bit stopu jest interpretowany jako bit rozróżniający ramkę adresu od ramki danych. W przypadku przełączenia nadajnika na 9-bitową długość pola danych jako bit rozróżniający ramki używany jest 9 bit pola danych, bity stopu interpretowane są w tym przypadku normalnie. W obu sytuacjach, jeśli bit określający typ ramki ma wartość 0, to mamy do czynienia z ramką danych, jeśli ma wartość 1, to mamy do czynienia z ramką adresu. Ramki adresu odbierane są przez wszystkie urządzenia *slave*; po ich odebraniu urządzenie decyduje, czy reszta transmisji jest przeznaczona dla niego. Jeśli tak, to przełączca UART w normalny tryb pracy, co umożliwia odbiór reszty danych. W przeciwnym przypadku reszta transmisji jest ignorowana, a układ czeka na odbiór kolejnej ramki adresu.

Sposób komunikacji z wykorzystaniem trybu MPCM wygląda następująco:

- ◆ Wszystkie urządzenia *slave* przechodzą w tryb MPCM.
- ◆ Urządzenie *master* nadaje ramkę adresu, która jest odbierana przez wszystkie urządzenia *slave*.
- ◆ Urządzenie *slave* sprawdza odebraną ramkę adresu i jeśli stwierdzi, że transmisja go dotyczy, wyłącza tryb MPCM.
- ◆ Wybrane urządzenie odbiera resztę danych, urządzenia niewybrane je ignorują (ich interfejs UART nie informuje o nadaniu jakichkolwiek danych).
- ◆ Po odebraniu wszystkich danych *slave* ponownie włącza tryb MPCM i wraz z innymi urządzeniami oczekuje na kolejną ramkę adresu.

W praktyce najwygodniej jest stosować 9-bitowe ramki danych, w przypadku ramek krótszych muszą one posiadać 2 bity stopu (jeden będzie wykorzystany jako bit określający typ ramki).

Tryb MPCM wybieramy, ustawiając bit MPCM rejestru UCSRA. Bit ten ma znaczenie tylko dla odbiornika — nie wpływa on w żaden sposób na nadajnik. Tak więc to oprogramowanie nadajnika jest odpowiedzialne za poprawne skonstruowanie ramki danych i ramki adresu. Stąd też w trybie MPCM stosuje się praktycznie wyłącznie 9-bitowe ramki danych. W przypadku ramek krótszych informacja o typie ramki znajduje się w pierwszym bicie stopu, na którego stan nie mamy wpływu. Stąd też stosując ramki krótsze niż 9-bitowe, należy stale przełączać nadajnik pomiędzy trybami  $n$  a  $n+1$  bitowymi.

Przejdzmy teraz do przykładu praktycznej realizacji układu wykorzystującego tryb MPCM. Dla uproszczenia wykorzystany zostanie ponownie układ pokazany na rysunku 19.5. Układ będzie realizował dokładnie takie same funkcje jak w poprzednim przykładzie, z wykorzystaniem podobnego protokołu komunikacyjnego. Jednak tym razem dodamy obsługę adresu. Adres układu *slave* będzie transmitowany jako pierwszy bajt ramki danych (w ten sposób można zaadresować do 256 różnych urządzeń *slave*) i nie będzie objęty sumą kontrolną. Do realizacji transmisji zostanie wykorzystany tryb 9-bitowy. W tym trybie 9. bit danych, znajdujący się w rejestrze UCSROB, określa typ nadawanej ramki danych. Jeśli wynosi 1, nadawany jest adres, który będzie odbierany przez wszystkie urządzenia *slave*. Dla normalnych danych bit ten musi mieć wartość 0. Zmiany w programie uwzględniające tryb MPCM są w dużej mierze kosmetyczne, pokazane więc zostaną tylko różnice w stosunku do poprzedniego przykładu. Zaczniemy od programu działającego na urządzeniu *master* (w przykładzie wykorzystywana jest transmisja synchroniczna, jednak, tak jak poprzednio, można skorzystać z transmisji asynchronicznej, eliminując w ten sposób potrzebę korzystania z linii *XCK*).

Zmieniła się nieco funkcja inicjująca USART:

```
void UART_master_init()
{
    uart_9600();
    UCSROB=_BV(RXEN0) | _BV(TXEN0) | _BV(UCSZ02); //Ramka 9-bitowa
    UCSROC=_BV(USBS0) | _BV(UCSZ01) | _BV(UCSZ00); //9 bitów danych, 2 bity stopu,
                                                       //bez parzystości, transmisja asynchroniczna
    DDRD|= _BV(PD4); //Układ generuje zegar dla slave
    UCSROC|= _BV(UMSEL00); //Tryb synchroniczny
}
```

Zmiana dotyczy tylko długości ramki — zamiast 8-bitowej stosowana jest 9-bitowa (ostatni bit wykorzystany jest do rozróżnienia typu ramki).

Zmianie uległa także funkcja nadająca ramki danych:

```
void uart_Send_Frame(uint8_t adres, char *buf, uint8_t n)
{
    UCSROB|= _BV(TXB80); //Wyślij adres
    uart_sendchar(adres);
    UCSR0B&= (~_BV(TXB80)); //Wyślij dane
    uint8_t crc=_crc_ibutton_update(0, n);
    uart_sendchar(n);
    while(n--)
    {
        uart_sendchar(*buf);
        crc=_crc_ibutton_update(crc, *buf);
```

```

    buf++;
}
uart_sendchar(crc);
}

```

Funkcja `uart_Send_Frame` wzbogaciła się o dodatkowy argument, określający adres urządzenia *slave*, do którego mają zostać wysłane dane. Przed wysłaniem bajtu adresu następuje ustawienie 9. bitu danych (TXB80) w rejestrze UCSR0B. Po wysłaniu adresu bit ten jest zerowany, dzięki czemu dane zostaną odebrane wyłącznie przez zaadresowane urządzenie. Reszta programu wygląda tak jak w poprzednim przykładzie.

Teraz zajmijmy się programem *slave*. Funkcja inicjująca ustawia bit MPCM, dzięki czemu wszystkie ramki poza ramkami adresowymi są ignorowane; ustawiany jest także 9-bitowy tryb transmisji:

```

void USART_slave_init()
{
    //uart_9600();
    UCSR0C=_BV(USBS0) | _BV(UCSZ01) | _BV(UCSZ00); //8 bitów danych, 2 bity stopu,
                                                       //bez parzystości, transmisja asynchroniczna
    UCSR0C|=_BV(UMSEL00);
    UCSR0A=_BV(MPCM0); //Włącz tryb MPCM
    UCSR0B=_BV(UCSZ02) | _BV(RXEN0) | _BV(TXEN0) | _BV(RXCIE0); //Odblokuj przerwania
                                                               //odbiornika, ramka 9-bitowa
}

```

Pamiętajmy, że dla poprawnej interpretacji nadchodzących danych niezbędne jest włączenie trybu MPCM.

Nieco większe zmiany dotyczą funkcji obsługi przerwania związanego z odebraniem bajtu:

```

ISR(USART_RX_vect)
{
    static uint8_t bufpos, n, crc;

    if(status) return; //Bląd - poprzednia ramka jeszcze nieobsłużona

    if(n==0)
    {
        n=UDR0;
        if((n==DEV_ADDR) && (UCSR0A & _BV(MPCM0)))
        {
            UCSR0A&=(_BV(MPCM0));
            n=0;
            return; //Zaadresowano urządzenie
        }
        UCSR0B&=(_BV(UCSZ02)); //Odbierz kolejne ramki danych
        bufpos=0;
        crc=_crc_ibutton_update(0, n);
        return;
    }

    if(bufpos<n)
    {
        Bufor[bufpos]=UDR0;
    }
}

```

```
    crc=_crc_ibutton_update(crc, Bufor[bufpos++]);
}
else
{
    if(crc==UDR0)
    {
        status=n;
        n=0;
    }
    UCSR0A|=BV(MPCM0); //Zakończono odbiór danych, przejdź w tryb MPCM
}
}
```

Symbol `DEV_ADDR` zawiera numer urządzenia *slave*. Jeśli nadchodzący bajt adresu ma wartość taką samą jak `DEV_ADDR`, urządzenie zostanie wybrane, co wiąże się ze skasowaniem bitu `MPCM`. Dzięki temu będzie możliwy odbiór normalnych ramek danych (wyłączona zostanie funkcja filtrująca). Jeśli odebrany adres urządzenia jest niezgodny, bit `MPCM` pozostanie ustawiony, a dalsze dane będą ignorowane — nie będą one nawet wywoływały kolejnych przerwań odbiornika.

Po prawidłowym zaadresowaniu układu odbierane są kolejne bajty; po odebraniu ostatniego bajtu (niezależnie od zgodności CRC) ponownie ustawiany jest bit `MPCM`. Dzięki temu urządzenie będzie ignorowało kolejne dane, aż do ponownego odebrania ramki zawierającej jego adres.



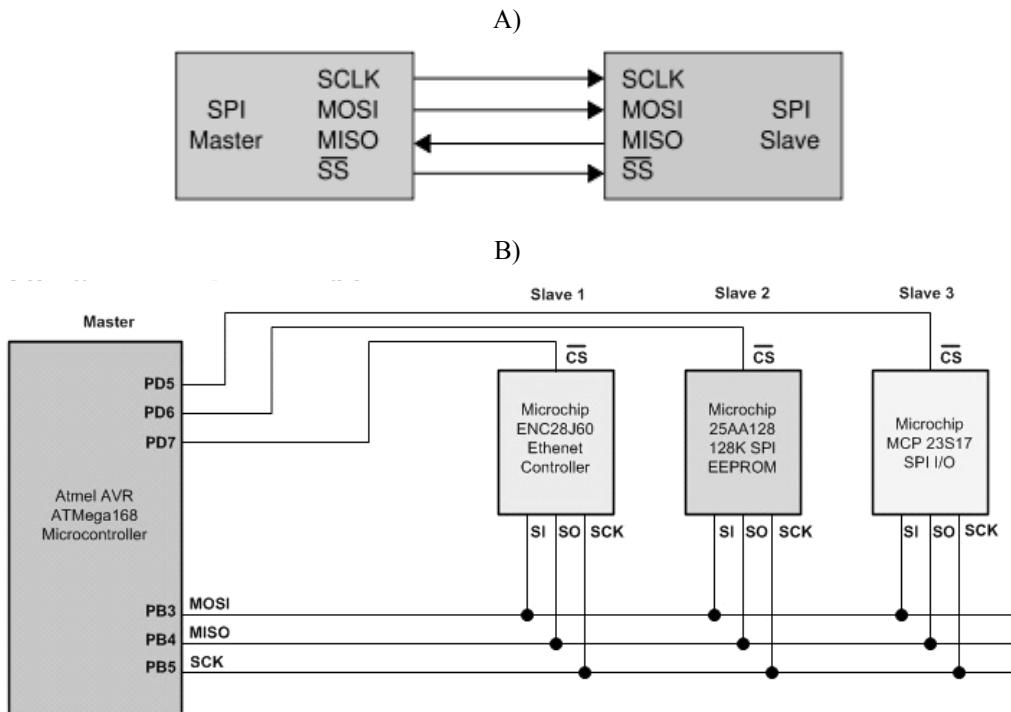
# Rozdział 20.

# Interfejs SPI

Jest to bardzo szybki interfejs umożliwiający podłączenie jednocześnie wielu urządzeń peryferyjnych, m.in. pamięci szeregowych, czujników temperatury, akcelerometrów, przetworników ADC, kontrolerów CAN i ethernet. Dane przez interfejs przesyłane są w trybie *full-duplex*, to znaczy, że jednocześnie można nadawać i odbierać dane. Jest interfejsem synchronicznym, czyli przesył kolejnych bitów synchronizowany jest sygnałem zegarowym. Daje to istotną przewagę nad protokołami asynchronicznymi — częstotliwość taktowania urządzeń na magistrali jest bez znaczenia. Powoduje to, że urządzenia wymieniające dane przy pomocy SPI nie muszą być, w przeciwieństwie do np. RS232, precyzyjnie taktowane. Magistrala ta umożliwia połączenie urządzeń na niewielkie odległości. Typowo są to urządzenia mieszczące się na jednej płycie drukowanej, chociaż po zastosowaniu odpowiednich nadajników/odbiorników linii można znacznie powiększyć długość magistrali. SPI może pracować z dowolną częstotliwością zegara; typowo są to częstotliwości 0 – 70 MHz (choć w mikrokontrolerach AVR można uzyskać częstotliwość pracy do 16 MHz), a więc magistrala ta jest znacznie szybsza od wcześniej pokazanej magistrali TWI/I2C. Sam interfejs SPI definiuje tylko sygnały niezbędne do jego realizacji, nic nie mówiąc o formacie wymienianych danych. Z drugiej strony, to programista musi zapewnić odpowiednią kontrolę danych, synchronizację i detekcję kolizji. Prostota tego interfejsu ma jeszcze jedną zaletę — do jego realizacji można oprócz dedykowanych interfejsów sprzętowych użyć dowolnych pinów *I/O*, a całą warstwę fizyczną interfejsu zrealizować programowo.

Do podłączenia wymagane są trzy linie sygnałowe — **SCK**, **MISO** (ang. *Master In Slave Out*) i **MOSI** (ang. *Master Out Slave In*) — oraz jedna dodatkowa linia na każde urządzenie **SS** (ang. *Slave Select*) — rysunek 20.1. Podobnie jak w przypadku interfejsu TWI, urządzenia pracują w konfiguracji *master-slave*, przy czym na magistrali w danej chwili może być aktywny tylko jeden *master* i tylko jeden *slave*.

Wyboru aktywnego urządzenia *slave* dokonuje się poprzez linię **SS** — aktywacja wiąże się z wybraniem stanu niskiego na pinie SS aktywowanego urządzenia *slave*. Jeśli mamy kilka urządzeń *slave*, każde musi posiadać własną linię **SS**. Całą transmisję kontroluje urządzenie *master*, jest ono także odpowiedzialne za generację zegara synchronizującego transmisję. **Ponieważ sygnał SCK jest generowany tylko w sytuacji, w której master coś nadaje, wynika z tego, że żeby odebrać dane z urządzenia slave, master jednocześnie musi coś nadawać.**



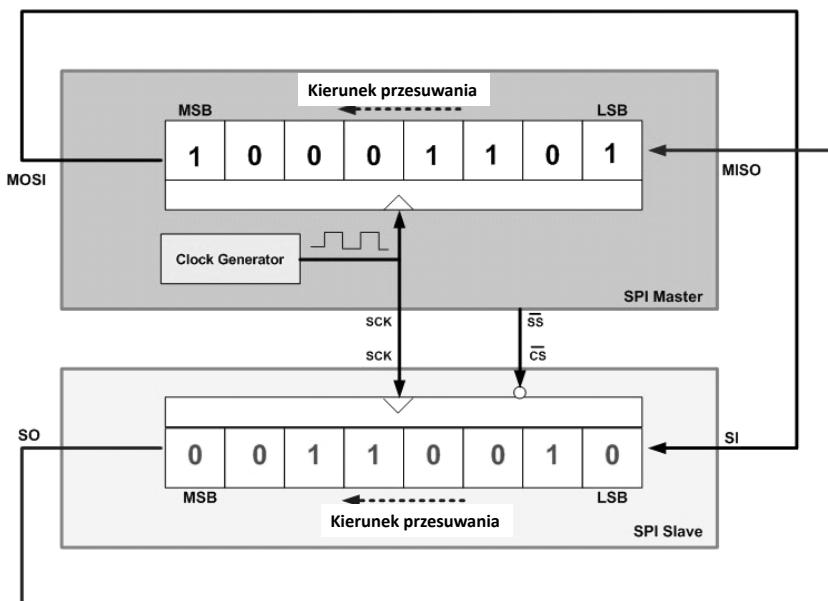
**Rysunek 20.1.** Podłączenie urządzeń w sieci SPI. A) Połączenie master-slave, B) połączenie master-multislave

Wymiana danych pomiędzy urządzeniami przebiega według niezwykle prostej zasady:

1. Urządzenie *master* ustawia w stan niski linię SS wybranego urządzenia *slave*, sygnalizując w ten sposób początek komunikacji.
2. Następnie *master* wpisuje daną, która pojawi się szeregowo na wyjściu *MOSI*, w tym samym czasie *slave* wpisuje daną, którą chce wysłać do *mastera*; znajdzie się ona na linii *MISO*.
3. *Master* zmienia stan linii *SCK*, co powoduje, że zarówno *master*, jak i *slave* odczytują jeden bit danych.
4. Punkty 2 – 3 powtarzane są do czasu, aż wszystkie dane zostaną przesłane.

Tak określony protokół jest niezwykle prosty — ze sprzętowego punktu widzenia są to po prostu połączone dwa szeregowe rejesty przesuwające (jeden jest po stronie *mastera*, drugi *slave'a*), które mają wspólny sygnał taktujący (rysunek 20.2). Dzięki temu urządzenie SPI *slave* możemy łatwo zaimplementować przy pomocy prostych rejestrów przesuwających z serii 74XXX, np. 74595, co zostanie pokazane w dalszej części.

Procesory AVR dysponują sprzętową obsługą interfejsu SPI. Dodatkowo oferują wiele możliwości konfiguracji tego interfejsu:



**Rysunek 20.2.** Zasada działania interfejsu SPI. Sygnał zegarowy taktuje dwa sprzężone ze sobą rejestrze przesuwające

- ◆ działanie w trybie *master* lub *slave*,
- ◆ wysuwanie bitów, począwszy od najmniej lub najbardziej znaczącego,
- ◆ określenie zbocza sygnału zegarowego, na którym odbywa się próbkowanie,
- ◆ określenie częstotliwości transmisji,
- ◆ przerwania informujące o gotowości nadajnika/odbiornika.

Wskazówka

Wszystkie definicje związane z interfejsem SPI znajdują się w pliku nagłówkowym `<avr\io.h>`.

Interfejs SPI w procesorach AVR nie jest buforowany w kierunku nadawania i ma pojedynczy bufor w kierunku odbierania (nie licząc danych znajdujących się w rejestrze SPDR). W efekcie podczas nadawania nie można wpisywać nowych danych do rejestrów SPDR, dopóki poprzednie nie zostały nadane. Z kolei przy odbiorze dane są przepisywane z rejestrów odbiornika do bufora, skąd muszą zostać odczytane przed zakończeniem kolejnej transmisji. Zapis do rejestrów SPDR powoduje umieszczenie danej w rejestrze nadajnika, odczyt tego rejestru powoduje odczyt z bufora odbiornika. Zakończenie transmisji sygnalizowane jest ustawieniem bitu SPIF w rejestrze SPSR, w przypadku ustawienia bitu zezwolenia na przerwania (SPIE) w rejestrze SPCR generowane jest także przerwanie SPI (`SPI_STC_vect`).

# Iinicjalizacja interfejsu

Iinicjalizacja interfejsu SPI odbywa się poprzez konfigurację bitów rejestru SPCR (ang. *SPI Control Register*) i SPSR (ang. *SPI Status Register*).



Wskazówka

Konfiguracja interfejsu SPI procesora musi dokładnie odpowiadać konfiguracji układu, z którym procesor będzie wymieniał dane. Jakiekolwiek różnice spowodują niepoprawną transmisję danych.

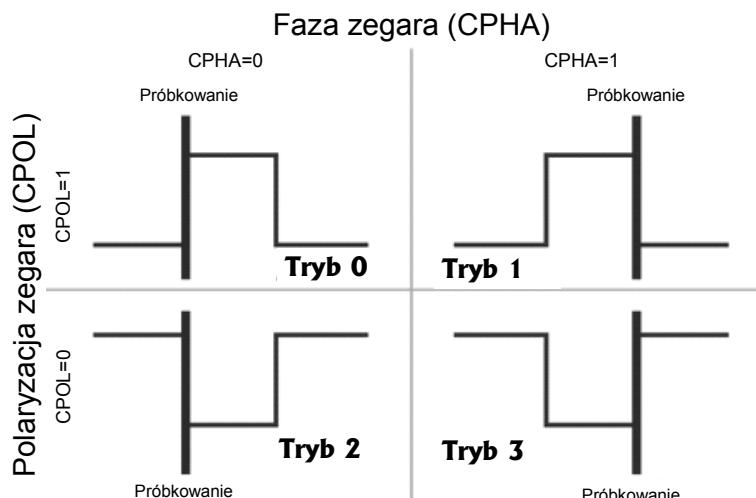
Zacznijmy od ustalenia kolejności wysyłanych/odczytywanych bitów. Do wyboru mamy dwa tryby: wysyłanie od najmniej znaczącego bitu (bit DORD równy 1) lub wysyłanie od najbardziej znaczącego bitu (DORD równy 0 — wartość domyślna).

Kolejnym etapem jest ustalenie polaryzacji zegara. Tutaj do wyboru mamy również dwie konfiguracje. Kiedy bit CPOL ma wartość 0, w stanie bezczynności stan linii *SCK* wynosi 0 — transmisja inicjowana jest zmianą stanu linii *SCK* z 0 na 1. Jeśli wartość CPOL wynosi 1, linia *SCK* w stanie bezczynności przyjmuje wartość 1, a początek transmisji następuje z chwilą jej przejścia ze stanu 1 do 0.

W następnym kroku ustalamy, kiedy ma być próbkowany stan linii danych. W przypadku kiedy bit CPHA ma wartość 0, próbkowanie odbywa się na pierwszym zboczu sygnału *SCK*, kiedy CPHA ma wartość 1, stan linii danych próbkowany jest na drugim zboczu sygnału *SCK*. Działanie interfejsu w zależności od stanu bitów CPOL i CPHA pokazane zostało na rysunku 20.3. Różne konfiguracje bitów CPOL i CPHA odpowiadają różnym trybom pracy interfejsu SPI (tryby 0 – 3). Na jednej magistrali SPI mogą znajdować się urządzenia pracujące w różnych trybach, jednak transmisja pomiędzy urządzeniem *master* i *slave* wymaga, aby oba pracowały w tym samym trybie.

Rysunek 20.3.

Różne tryby pracy interfejsu SPI, wybierane przy pomocy bitów konfiguracyjnych CPOL i CPHA



Po ustaleniu sposobu przesyłania danych należy ustalić, czy procesor ma pracować jako *master*, czy *slave*. Pracę jako *master* zapewnia wpisanie 1 do bitu MSTR, wyzerowa-

nie tego bitu sprawia, że procesor będzie urządzeniem *slave*. Ostatnią czynnością jest odblokowanie interfejsu SPI poprzez wpisanie 1 do bitu SPE. Dodatkowo, jeśli chcemy, aby przy nadaniu/odebraniu bajtu generowane było przerwanie, należy wpisać 1 do bitu SPIE (ang. *SPI Interrupt Enable*).

## Ustawienie pinów IO

Aktywacja interfejsu SPI procesora powoduje przejęcie kontroli tylko nad niektórymi pinami *IO* — tabela 20.1. Stan pozostałych, o ile chcemy z nich korzystać, musi być jawnie określony przez program.

**Tabela 20.1.** Stan pinów *IO* w zależności od trybu pracy SPI

Pin	Tryb master	Tryb slave
<i>MOSI</i>	Kierunek określony przez programistę	Wejście
<i>MISO</i>	Wejście	Kierunek określony przez programistę
<i>SCK</i>	Kierunek określony przez programistę	Wejście
<i>SS</i>	Kierunek określony przez programistę	Wejście

W przypadku trybu *master* przynajmniej pin *SCK* należy ustawić jako wyjściowy, tak aby wybrane urządzenie *slave* otrzymało sygnał zegarowy.

### Pin SS

Szczególną rolę odgrywa pin *SS*. Jego stan istotny jest wyłącznie w sytuacji, kiedy odpowiadający mu pin portu *IO* ustawiony jest jako wejście. W przeciwnym przypadku można go wykorzystać jako zwykły pin *IO*. Kiedy procesor AVR pracuje w trybie *master*, stan tego pinu musi być wysoki. Jego zmiana na niski powoduje natychmiastowe przerwanie transmisji i wyzerowanie bitu MSTR rejestru SPCR — układ przestaje być *masterem*.



Przy konfiguracji interfejsu SPI należy zadbać, aby przed włączeniem trybu *master* pin *SS* miał stan wysoki. Przypadkowe przejście tego pinu w stan niski powoduje natychmiastową dezaktywację trybu *master*.

Ma to zastosowanie w systemach multimaster — dzięki temu w sytuacji, kiedy inny master przejmuje kontrolę, następuje dezaktywacja do tej pory aktywnego nadajnika. Układ, którego pin *SS* zmienił stan na niski, staje się urządzeniem *slave*, towarzyszy temu ustawienie bitu SPIF i generacja przerwania (o ile zostało ono odblokowane). **Jeśli takie zachowanie nam nie odpowiada, musimy zapewnić stały, wysoki poziom tego sygnału lub ustawić pin SS jako wyjście.**

W trybie *slave* pin *SS* zawsze jest wejściem (interfejs SPI przejmuje kontrolę nad tym pinem *IO*), jego niski stan aktywuje interfejs SPI i zaczyna on odbierać dane otrzymywane na pinie *MOSI* w takt zegara *SCK*. Jednocześnie układ może rozpocząć nadawanie danych, o ile pin *MISO* został przez programistę ustawiony jako wyjście. Ponowne przejście sygnału *SS* w stan wysoki dezaktywuje interfejs SPI.

## Zegar taktujący

Wybór częstotliwości zegara taktującego transmisję ma sens tylko w przypadku urządzenia pracującego w trybie *master*. Zegar ten musi być dobrany pod kątem możliwości urządzenia *slave*, z którym nawiązywane będzie połączenie. Zegar taktujący SPI powstaje w wyniku podzielenia zegara taktującego rdzeń procesora przez wybraną wartość określona bitami SPR (ang. *SPI Clock Rate Select*) rejestru SPCR. Umożliwiają one wybór jednego z 4 dzielników: 4, 16, 64 lub 128. Dodatkowo w rejestrze SPSR znajduje się bit SPI2X (ang. *Double SPI Speed Bit*), powodujący dwukrotne zwiększenie częstotliwości zegara taktującego, przez co do dyspozycji mamy dodatkowe dzielniki: 2, 8, 32 i 64. Z bitu SPI2X można korzystać wyłącznie w trybie pracy *master*. Dla trybu *slave* zegar taktujący SPI (doprowadzony do pinu *SCK*) nie może być szybszy niż F\_CPU/4, gdzie F\_CPU to zegar taktujący procesor.

## Procesor w trybie Master SPI

W trybie *master* procesor nie kontroluje linii *SS* — to programista jest odpowiedzialny za wybór odpowiedniego urządzenia *slave* i wysterowanie jego linii *SS*. **Jednakże ważne jest, żeby w przypadku kiedy pin SS jest ustawiony jako wejście, utrzymywać na nim stan wysoki.** W takiej sytuacji niski stan na tym pinie powoduje przerwanie transmisji — procesor interpretuje ten stan jako kolizję z innym układem *master*. Sytuacja taka prowadzi do wyzerowania bitu MSTR rejestru SPCR — dzięki temu możemy taką sytuację wykryć. Ponowna aktywacja trybu *master* wymaga ustawienia tego bitu na 1. Zakończenie wysyłania danych w tym trybie sygnalizuje ustawniona flaga SPIF (ang. *SPI Interrupt Flag*) rejestru stanu SPSR (ang. *SPI Status Register*). Po jej ustawnieniu można do rejestru SPDR wpisać kolejny bajt do wysłania. Flaga SPIF kasowana jest automatycznie po wejściu w procedurę obsługi przerwania SPI o wektorze SPI\_STC\_vect lub poprzez odczytanie rejestru stanu SPSR, a następnie zapis do rejestru SPDR.

Zapis do rejestru SPDR w sytuacji, kiedy poprzedni znajdujący się w nim bajt nie został jeszcze wysłany, powoduje kolizję sygnalizowaną przez ustawienie flagi WCOL (ang. *Write COLlision Flag*) rejestru stanu. Flaga ta kasuje się po odczytaniu rejestru stanu, a następnie zapisie nowej wartości do rejestru SPDR.



Rejestr SPDR w czasie zapisu nie jest buforowany. Wpisując do niego nową wartość, należy upewnić się, że poprzednia została wysłana, co sygnalizuje ustawniona flaga SPIF.

Przy odczytaniu rejestru SPDR ma bufor jednopoziomowy, to znaczy, że po zakończonym odczytaniu 8 bitów danych są one kopowane do rejestru pomocniczego, z którego mogą zostać odczytane. Dzięki temu natychmiastowa transmisja kolejnej porcji danych nie nadpisuje danych już odebranych.

## Procesor w trybie slave SPI

W trybie *slave* procesor wczytuje dane wysyłane przez *mastera* w takt sygnału zegarowego. Aby stan linii *MOSI* był poprawnie próbkowany, poziom wysoki i niski muszą trwać przynajmniej 2 cykle zegara taktującego procesor, stąd też maksymalny zegar interfejsu SPI w tym trybie wynosi CLK/4. W trybie tym pin *SS* jest zawsze wejściem, jego niski stan logiczny aktywuje interfejs SPI. Wysoki stan logiczny na tym pinie inaktywuje układ SPI i przywraca jego stan początkowy. Powoduje także przejście linii *MISO* w stan wysokiej impedancji, co umożliwia rozpoczęcie nadawania innemu urządzeniu *slave*. Pin *SS* można więc wykorzystać do synchronizacji poszczególnych bajtów lub ramek transmisji.

## Przykłady

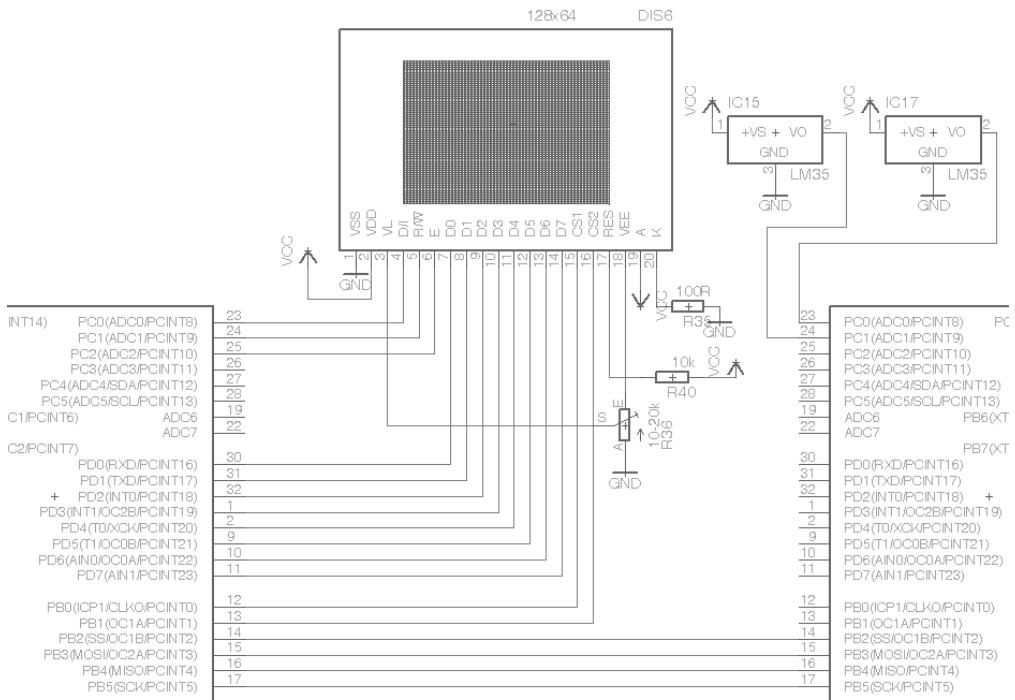
Poniżej pokazane zostaną dwa przykłady ilustrujące zasadę korzystania z SPI. W przykładzie pierwszym połączone zostaną ze sobą dwa procesory ATMega88, w drugim pokazane zostanie wykorzystanie pamięci SPI.

## Połączenie AVR-AVR

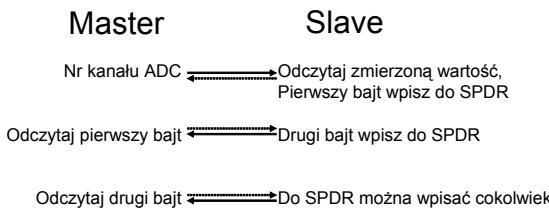
Ponieważ interfejs SPI procesorów AVR może pracować zarówno w trybie *master*, jak i *slave*, umożliwia to wykorzystanie magistrali SPI do połączenia ze sobą dwóch mikrokontrolerów. W schemacie na rysunku 20.4 pokazane zostało połączenie dwóch mikrokontrolerów ATMega88, jednak na tej samej zasadzie i praktycznie bez zmiany przykładowego programu można połączyć dowolne mikrokontrolery z rodziną AVR. Nie ma też przeszkód, aby stworzyć konfigurację *master-multislave* — wystarczy dodać kolejne linie kontrolujące piny *SS* urządzeń *slave*.

W pokazanym układzie jeden z mikrokontrolerów podłączony jest do wyświetlacza graficznego LCD i pracuje jako *master*. Cyklicznie odpytuje on drugi mikrokontroler, pracujący jako *slave*. Zadaniem układu *slave* jest przetwarzanie sygnału analogowego doprowadzonego do wejść *ADC0* i *ADC1* i wysyłanie wyników do układu *master*. Ponieważ interfejs SPI definiuje tylko fizyczny sposób przesyłania danych, najpierw należy opracować protokół wymiany danych pomiędzy urządzeniami. Urządzenie *master* będzie wysyłać okresowo polecenie odczytu wybranego kanału ADC, na co urządzenie *slave* powinno odpowiedzieć, zwracając 16-bitową wartość temperatury zmierzonej za pomocą termometrów analogowych LM35 (rysunek 20.5).

Ponieważ protokół SPI jest protokołem, w którym zawsze wysłaniu danych towarzyszy także ich odebranie, nie wszystkie dane mają znaczenie. Przy pierwszej transmisji urządzenie *master* wysyła polecenie będące numerem odczytywanego kanału ADC w urządzeniu *slave*. W tym samym czasie urządzenie *slave* wysyła do mastera przypadkowy bajt znajdujący się w rejestrze SPDR. *Slave* po odebraniu polecenia umieszcza w rejestrze SPDR starszy bajt wyniku ze wskazanego wejścia ADC. Bajt ten zostanie odebrany przez urządzenie *master* w czasie kolejnej transmisji. Analogicznie wysyłany



**Rysunek 20.4.** Połczenie dwóch mikrokontrolerów ATMega88 przy pomocy magistrali SPI w trybie master-slave



**Rysunek 20.5.** Wymiana danych pomiędzy urządzeniem master i slave. Liniami przerywanymi zaznaczono transmisje bajtów, których wartość nie ma znaczenia. Liniami ciągłymi zaznaczono transmisje komend i wyników

jest młodszy bajt wyniku. Pokazany protokół wymiany jest niezwykle prosty, ale ma jedną poważną wadę. Urządzenie *slave* musi wiedzieć, który bajt wysyłany przez *mastera* jest informacją o numerze zwracanego kanału ADC. Inaczej mówiąc, musi wiedzieć, kiedy następuje początek transmisji. Problem nie jest banalny do rozwiązania, gdyż interfejs SPI procesorów AVR nie generuje żadnego specjalnego zdarzenia w chwili przejścia linii SS urządzenia *slave* do stanu aktywnego. Stąd też w poniższym programie wykorzystano możliwość generowania przerwania przy zmianie stanu pinu.

Program realizujący wymianę informacji z wykorzystaniem interfejsu SPI składa się z dwóch elementów — programów działających na układzie *master* i *slave*. Zaczniemy od układu *slave*. Układ ten próbuje stan wejść ADC, uśrednia wyniki z pomiarów i zapisuje w tablicy ADCVal:

```
volatile uint32_t ADCVal[LM35NO];
```

Ta część programu jest praktycznie identyczna z programem wyświetlającym temperaturę z termometru analogowego LM35 pokazanego w rozdziale 15. Różnica polega tylko na tym, że zamiast próbować tylko jeden kanał ADC, próbuje ich więcej (ich liczbę określa definicja LM35NO). Ponieważ w trakcie próbkowania zmieniany jest wykorzystywany kanał ADC, zrezygnowano z trybu ciągłego próbkowania (*free running mode*) na rzecz trybu pojedynczej konwersji. Stąd też w przerwaniu ADC za każdym razem ustawiany jest bit ADSC. Ze względu na zależności czasowe opisane w dalszej części procedura obsługi przerwania ADC została zadeklarowana jako nieblokująca:

```
ISR(ADC_vect, ISR_NOBLOCK)
```

W efekcie zaraz po wejściu w jej obsługę przerwania zostają odblokowane i procesor może obsługiwać inne przerwania, o wyższym priorytecie.

Część obsługująca magistralę SPI znajduje się w pliku *SPI-slave.c*. Składa się ona z funkcji inicjalizującej interfejs:

```
void SPI_slave_init()
{
    DDRB|= _BV(PB4); //Pin MISO jako wyjście
    SPCR=_BV(SPE) | _BV(SPIE); //Tryb master, przerwania
    SPSR;
    SPDR; //Skasuj flagę SPIF
    PCMSK0|= _BV(PCINT2); //Odblokuj przerwanie PCINT2
    PCICR|= _BV(PCIE0);
}
```

Interfejs SPI inicjalizowany jest w trybie *slave*, ustawiane są odpowiednio kierunki używanych portów *I/O* oraz odblokowywane jest przerwanie związane ze zmianą stanu pinu SS. Dzięki temu z łatwością można będzie wykryć wszelkie jego zmiany i odpowiednio inicjalizować transmisję SPI.

Aby transmisja przebiegała efektywnie, została oparta ona została o przerwania. Odebranie bajtu od urządzenia *master* sygnalizowane jest wywołaniem przerwania o wektorze SPI\_STC\_vect:

```
uint8_t byteno;

ISR(SPI_STC_vect)
{
    static uint16_t temperature;

    if(byteno==0) temperature=GetTemperature(SPDR);
    SPDR=((uint8_t*)&temperature)[byteno];
    byteno=(byteno+1) % 3;
}
```

W przerwaniu tym sprawdzany jest stan zmiennej byteno, zawierającej liczbę bajtów przetransmitowanych do urządzenia *master* od chwili aktywacji sygnału SS. Jeśli jej wartość wynosi 0, to znaczy, że *master* dopiero przesłał komendę zawierającą numer żądanego kanału ADC. W takiej sytuacji wywoływana jest funkcja GetTemperature,

która przelicza zmierzona wcześniej wartość z przetwornika na temperaturę. Uzyskane wartości są sukcesywnie ładowane do rejestru SPDR, w miarę jak *master* przesyła kolejne bajty (których wartość jest ignorowana).

Aby możliwe było rozpoznanie komendy wysyłanej przez urządzenie *master*, trzeba zmienną *byteno* inicjować wartością 0 przy każdym przejściu sygnału SS ze stanu wysokiego do niskiego (co wiąże się z aktywacją urządzenia *slave*):

```
ISR(PCINT0_vect)
{
    byteno=0;
}
```

Tak zdefiniowane przerwanie jest wywoływanego także przy przejściu sygnału SS ze stanu niskiego do wysokiego, ale jest to bez znaczenia.

Ponieważ cała obsługa SPI i ADC odbywa się w przerwaniach, funkcja `main` programu *slave* jest niezwykle prosta i ogranicza się tylko do inicjalizacji wykorzystywanych podsystemów procesora:

```
int main()
{
    SPI_slave_init();
    sei();
    ADC_init();
    while(1);
}
```

Przejdzmy teraz do realizacji programu *mastera*. Ponieważ program ten komunikuje się z urządzeniem *slave* wtedy, kiedy zachodzi taka potrzeba, w dodatku sam steruje przebiegiem transmisji, nie ma potrzeby, aby do jego realizacji używać przerwań. Tak jak poprzednio, zaczniemy od napisania procedur komunikacji po SPI. Najpierw należy zainicjować interfejs SPI:

```
void SPI_master_init()
{
    SPI_Set_SS();
    DDRB|=_BV(PB2) | _BV(PB3) | _BV(PB5); //Piny SS, MOSI, SCK jako wyjście
    SPCR=_BV(SPE) | _BV(MSTR); //Tryb master, CLK/4
    SPSR;
    SPDR; //Skasuj flagę SPIF
}
```

Powyższa funkcja ustawia właściwy kierunek używanych do transmisji pinów *I/O*, a następnie inicjalizuje interfejs SPI w trybie *master*.



**Wskazówka** Przed włączeniem trybu *master* należy zadbać o właściwy stan linii SS, tak aby nie doszło do automatycznego przełączenia w tryb *slave*.

Interfejs SPI będzie pracował z maksymalną częstotliwością równą  $F_{CPU}/4$ , a więc aż 2 MHz. Należy zapewnić, aby układ *slave* był w takiej sytuacji taktowany przebie-

giem co najmniej czterokrotnie większym (czyli co najmniej 8 MHz). Oprócz procedury inicjującej potrzebna jest jeszcze funkcja służąca do wymiany danych z urządzeniem *slave*:

```
uint8_t SPI_send_rec_byte(uint8_t byte)
{
    SPDR=byte;
    while(!(SPSR & _BV(SPIF)));
    return SPDR;
}
```

Funkcja ta jednocześnie wysyła i odbiera 8 bitów danych. Odebrany bajt zwracany jest jako rezultat funkcji. Jeśli jest on niewykorzystywany, można go zignorować.

Pozostaje jeszcze napisanie funkcji aktywujących wybrane urządzenie *slave* poprzez wystawienie stanu niskiego na jego linii SS. W naszym przypadku mamy tylko jedno urządzenie *slave*, lecz linia SS jest potrzebna do synchronizacji transmisji, co zostało opisane wcześniej.

**Wskazówka**

W wielu przypadkach linia SS jest wykorzystywana do synchronizacji transmisji, stąd nawet w sytuacji, kiedy jest tylko jedno urządzenie *slave*, nie można na stałe wymusić na niej poziomu niskiego.

```
static inline void SPI_Set_SS()
{
    PORTB|= _BV(PB2);
}

static inline void SPI_Reset_SS()
{
    PORTB&=~(_BV(PB2));
}
```

Ponieważ funkcje sterujące linią SS zostaną przetłumaczone na jedną instrukcję asemblera (SBI/CBI), zadeklarowano je jako static inline, dzięki czemu kompilator nie będzie generował skoków do nich, lecz będzie ich kod osadzał bezpośrednio w miejscu wywołania.

**Wskazówka**

W tym przykładzie do sterowania urządzenia *slave* wykorzystano linię SS urządzenia *master*. Lecz w tym celu można wykorzystać dowolny inny port I/O mikrokontrolera. W trybie *master* pin SS procesora nie jest w żaden sposób wyróżniony.

Mając komplet funkcji komunikacyjnych, należy jeszcze napisać funkcję, której zadaniem będzie wysłanie do urządzenia *slave* polecenia odczytu temperatury z wybranego wejścia ADC i odebranie wyników:

```
uint16_t GetTemperature(uint8_t LMno)
{
    uint16_t temperature;
    SPI_Reset_SS();
    SPI_send_rec_byte(LMno);
```

```

    _delay_us(100);
    temperature=SPI_send_rec_byte(LMno);
    _delay_us(10);
    temperature=256*SPI_send_rec_byte(LMno)+temperature;
    SPI_Set_SS();
    return temperature;
}

```

Funkcja ta wybiera urządzenie *slave* poprzez wprowadzenie jego linii w stan niski, następnie wysyła polecenie przesłania temperatury z wybranego kanału i odczytuje kolejne dwa bajty. Pomiędzy transmisjami kolejnych bajtów wprowadzane jest niewielkie opóźnienie. Jest ono niezbędne, aby urządzenie *slave* po odebraniu bajtu danych mogło uaktualnić zawartość rejestru SPDR. To, jak wielkie opóźnienie należy umieścić pomiędzy kolejnymi odczytami, zależy od maksymalnego czasu, jakiego urządzenie *slave* potrzebuje na wejście w procedurę obsługi przerwania SPI i uaktualnienia rejestru SPDR. Pokazany przykładowy program *slave* jest niezwykle prosty, ale nawet w tak prostym programie ujawniają się problemy z przewidzeniem tego czasu. W urządzeniu *slave*, oprócz przerwań SPI, generowane są przerwania ADC związane z zakończeniem procesu konwersji. Normalnie w mikrokontrolerach AVR wejście do procedury obsługi przerwania powoduje zablokowanie globalnej flagi zezwolenia na przerwanie, przez co nie jest możliwe przyjmowanie dalszych przerwań. Zostaną one obsłużone po przerwaniu bieżącym. W efekcie czas, jaki upływa od nadjęcia bajtu do uaktualnienia rejestru SPDR, wydłuża się o czas realizacji innych procedur obsługi przerwań i staje się nieprzewidywalny. Stąd też w programie *slave* przerwanie ADC, które nie jest krytyczne czasowo, zostało zadeklarowane z atrybutem `ISR_NOBLOCK`, dzięki czemu jego wykonanie może zostać przerwane przez przerwanie SPI, co znacznie skraca czas reakcji na nadchodzące dane. Zadeklarowanie przerwania SPI jako nieblokującego mogłoby potencjalnie być bardzo niebezpieczne — jeżeli procesor nie nadążałby z odbiorem kolejnych danych, kolejne wywołania przerwań ulegałyby zagnieźdzeniu, co wiązałoby się z ryzykiem przepełnienia stosu.

Zobaczmy jeszcze, jak wygląda funkcja `main`:

```

int main()
{
    char wynik[7];

    GLCD_init();
    GLCD_cls();
    GLCD_goto(0,0);
    SPI_master_init();

    GLCD_puttext_P(PSTR("Temperatura:"));
    GLCD_goto(0,_1);
    GLCD_puttext_P(PSTR("T1:"));
    GLCD_goto(0,_2);
    GLCD_puttext_P(PSTR("T2:"));

    while(1)
    {
        for(uint8_t i=0;i<2;i++)
        {
            sprintf(wynik, "%5d", GetTemperature(i));

```

```
    uint8_t len=strlen(wynik);
    memmove(&wynik[len-1], &wynik[len-2], 3);
    wynik[len-2]='.';
    GLCD_goto(15, i+1);
    GLCD_puttext(wynik);
}
delay_ms(500);
}
```

Funkcja ta inicjalizuje wyświetlacz graficzny w znany nam już sposób, inicjalizuje interfejs SPI, a następnie cyklicznie wysyła do urządzenia *slave* zapytania o temperaturę z termometrów podłączonych do jego wejść *ADC0* i *ADC1*.

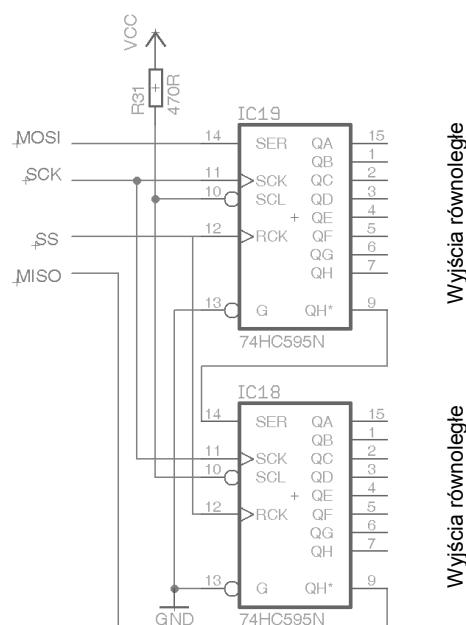
## Połączenie AVR – rejestr szeregowy

Interfejs SPI umożliwia bardzo proste wykorzystanie scalonych rejestrów szeregowych jako ekspanderów portów I/O. W tym celu najczęściej stosuje się 8-bitowe rejesty przesuwające z wejściem/wyjściem równoległy. Przykładem są układy 74XXX595 jako rejestr szeregowy z wyjściami równoległymi (rysunek 20.6) lub układ 74XXX589 jako rejestr szeregowy z wejściem równoległy. Oba układy występują także w wersjach 16-bitowych. Oba układy posiadają wyjścia i wejścia szeregowe, co umożliwia ich kaskadowe łączenie, w efekcie można uzyskać rejesty o praktycznie dowolnej długości. Limitem jest tu wyłącznie czas potrzebny na przeładowanie ich zawartości.

## Rysunek 20.6.

*Schemat podłączenia  
do magistrali SPI  
rejestru szeregowego  
74XXX595*

Wyjście QH można podłączyć do wejścia szeregowego SER, co umożliwia wydłużenie rejestru. Opcjonalnie wyjście QH można podłączyć do wejścia MISO procesora, dzięki czemu możliwy będzie odczyt zawartości rejestru. W takiej konfiguracji nie będzie jednak możliwe podłączenie innych urządzeń do magistrali SPI (wyjście QH nie jest trójstanowowe).



Eksplandery *I/O* można zrealizować przy pomocy wielu różnych magistrali, np. omówionej wcześniej magistrali I2C. Jednak układy rejestrów przesuwających są zwykle tańsze, zdecydowanie szybsze (pracują z prędkością 50 – 80 MHz, w porównaniu z ok. 400 kHz dla magistrali SPI), a ich oprogramowanie jest prostsze.

Układ 74XXX595 posiada wejście *G*, którego niski stan powoduje odblokowanie buforów wyjściowych rejestru. W większości przypadków stan tego wejścia można na stałe ustalić na niski. Ważnym wejściem jest wejście *RCK*. Jego stan niski powoduje uaktywnienie wejścia *SCK*, co umożliwia wpisywanie do rejestru nowych wartości pojawiających się na wejściu *SER* podłączonym do wyjścia *MOSI* mikrokontrolera. Dodatkowo zmiana stanu wejścia *RCK* z niskiego na wysoki powoduje przepisanie zawartości rejestru do buforów wyjściowych układu, dzięki czemu nowo wpisane wartości pojawiają się na wyjściach równoległych układu. Wejście *SCL* służy do zerowania układu, w czasie normalnej pracy na tym wejściu można na stałe wymusić wysoki poziom logiczny.

Układ 74XXX595 pracuje w trybie SPI 0, procedura inicjująca interfejs wygląda następująco:

```
void SPI_master_init()
{
    SPI_Set_SS();
    DDRB|=_BV(PB2) | _BV(PB3) | _BV(PB5); //Pin SS, MOSI, SCK jako wyjście
    SPCR=_BV(SPE) | _BV(MSTR); //Tryb master, CLK/4
    SPCR|=_BV(SPR1) | _BV(SPR0);
    SPSR;
    SPDR; //Skasuj flagę SPIF
}
```

Z kolei zapis nowych danych realizuje funkcja:

```
void SPI_send_rec_byte(uint8_t byte)
{
    PORTB&=~(_BV(PB2)); //Uaktywnij układ
    SPDR=byte;
    while(!(SPSR & _BV(SPIF)));
    PORTB|=_BV(PB2); //Przepisz dane do zatrzasków wyjściowych
}
```

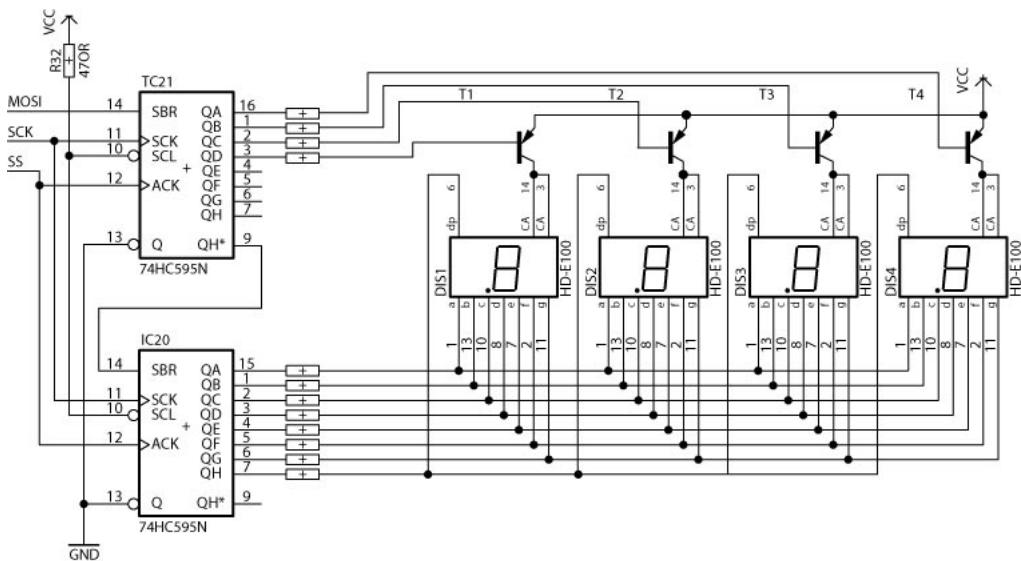
W przypadku rejestru 595 nie ma sensu odczytywać danych, które nadeszły linią *MISO*, gdyż linia ta jest w tym przypadku zazwyczaj niepodłączona. Można ją podłączyć, w efekcie odczyta się poprzednią zawartość rejestru (co w pewnych sytuacjach może mieć sens). Jeżeli połączonych jest szeregowo kilka rejestrów, sygnał ich wyboru (*SS*) musi być aktywny podczas całej transmisji wielobajtowego ciągu:

```
void SPI_send_rec_byte(uint16_t word)
{
    PORTB&=~(_BV(PB2)); //Uaktywnij układ
    SPDR=word & 0xFF;
    while(!(SPSR & _BV(SPIF)));
    SPDR=word >>8;
    while(!(SPSR & _BV(SPIF)));
    PORTB|=_BV(PB2); //Przepisz dane do zatrzasków wyjściowych
}
```

Zmiana stanu linii SS pomiędzy wysłaniem kolejnych bajtów spowodowałaby zatrzymanie nieprawidłowych, częściowo przetransmitowanych danych, w efekcie na wyjściach rejestru pojawiłyby się zakłócenia.

## Wykorzystanie rejestru 74XXX595 do sterowania multipleksowego wyświetlaczem LED

Dosyć często rejestr szeregowy wykorzystywany jest do sterowania wyświetlaczem LED. W rozdziale 14. pokazano przykład wykorzystania portów *I/O* i przerwań do sterowania multipleksowanego wyświetlaczem LED. Umożliwiło to znaczną redukcję liczby wykorzystywanych wyprowadzeń, lecz nadal aż 12 wyprowadzeń *I/O* było niezbędnych do sterowania wyświetlaczem składającym się z zaledwie 4 cyfr. Układ ten można znacznie uprościć dzięki wykorzystaniu rejestrów przesuwanych i interfejsu SPI (rysunek 20.7).



**Rysunek 20.7.** Sterowanie wyświetlaczem 7-segmentowym LED przy pomocy interfejsu SPI i rejestrów 74XXX595. Linie MOSI, SCK i SS należy podłączyć do wyprowadzeń mikrokontrolera, na których realizowany jest interfejs SPI

Sam układ sterowania wyświetlaczami jest analogiczny do układu pokazanego w rozdziale 14. Różnica sprowadza się do wykorzystania rejestrów szeregowych zamiast portów *I/O* procesora. Umożliwia to znaczne zmniejszenie liczby niezbędnych wyprowadzeń, co więcej, wprowadzenie dodatkowych wyświetlaczy nie powoduje zwiększenia liczby wykorzystanych wyprowadzeń procesora. Jedyne, co trzeba zrobić, to zwiększyć liczbę użytych rejestrów szeregowych. Dzięki temu tego typu układy świetnie nadają się do sterowania wyświetlaczami matrycowymi, linijkami świetlnymi czy znymi z reklam przesuwającymi się napisami.

Pokazany poniżej przykład wykorzystuje interfejs sprzętowy SPI procesora, lecz odmienność niż w poprzednich przykładach wykorzystywane są przerwania SPI, generowane po

zakończeniu transmisji bajtu danych. Dzięki temu przerwanie SPI o wektorze SPI\_STC\_vect może być jednocześnie wykorzystywane do realizacji multipleksowania, w efekcie nie trzeba dodatkowo wykorzystywać *timera*.

Do realizacji tego zadania potrzebne będą pewne funkcje i struktury pomocnicze. Pierwszą jest struktura:

```
struct ISR_Status
{
    uint8_t no : 4;
    uint8_t byte : 1;
};
```

przechowująca numer wyświetlanej cyfry (pole no) oraz flagę informującą o stanie transmisji (pole byte). Do realizacji wyświetlacza użyte zostały dwa rejestrów zatrząskowe 74HC595. Ponieważ interfejs SPI umożliwia przesłanie jednocześnie tylko jednego bajtu danych, uaktualnianie rejestrów musi odbywać się dwuetapowo. Flaga byte informuje, na jakim etapie znajduje się transmisja. Jeśli wynosi 0, to transmitowany jest pierwszy bajt, jeśli ma wartość 1, to transmitowany jest drugi bajt.

Oprócz struktury ISR\_Status potrzebne są funkcje manipulujące sygnałem SS: funkcja SPI\_Set\_SS(), ustawiająca tę linię w stanie wysokim, oraz funkcja SPI\_Reset\_SS(), zerująca linię SS. Zostały one zdefiniowane tak samo jak w poprzednich przykładach.

Przejdzmy teraz do realizacji transmisji. Kolejne bajty będą transmitowane w funkcji obsługi przerwania SPI, dzięki czemu sterowanie LED będzie niezależne od wykonywania programu. W tym celu należy odpowiednio zainicjować interfejs:

```
void SPI_master_init()
{
    SPI_Set_SS();
    DDRB|= _BV(PB2) | _BV(PB3) | _BV(PB5));           //Pin SS, MOSI, SCK jako wyjście
    SPCR= _BV(SPIE) | _BV(SPE) | _BV(MSTR);           //Tryb master, CLK/128, przerwania
    SPCR|= _BV(SPR1) | _BV(SPR0);
    SPSR;
    SPDR; //Skasuj flagę SPIF
}
```

Interfejs został skonfigurowany w trybie *master*, z wyłączeniem przerwań generowanych w wyniku zakończenia transmisji bajtu danych, zegar taktujący interfejs został ustawiony na F\_CPU/128. Dla mikrokontrolera taktowanego zegarem 8 MHz SPI będzie taktowane sygnałem o częstotliwości 62,5 kHz, co jest wystarczające do uzyskania płynnego wyświetlania. Większe częstotliwości taktowania SPI powodowałyby tylko niepotrzebny wzrost obciążenia procesora.

Najważniejszą częścią programu jest funkcja obsługująca przerwania SPI:

```
ISR(SPI_STC_vect)
{
    static struct ISR_Status status;

    if(status.byte==0)
    {
        SPI_Set_SS();           //Przepisz zawartość rejestrów do zatrzasków wyjściowych
```

```

asm volatile ("nop"); //Konieczne ze względu na synchronizator
SPI_Reset_SS(); //Wsuwamy nową wartość
uint8_t tmp=0xFF;
uint8_t val=LEDDIGITS[status.no]; //Cyfra do wyświetlenia
if((val & 0x7F)<11) tmp=pgm_read_byte(&DIGITS[val & 0x7F]); //Jej reprezentacja na LED
if((val & DP)==1) tmp&=~(DP); //Kropka dziesiątna
SPDR=tmp; //Wyślij dane o wyświetlanej cyfrze
}
else
{
    SPDR=~(1<<status.no); //Wybierz wyświetlacz
    status.no=(status.no+1)%LEDDISPNO;
}
status.byte^=1;
}

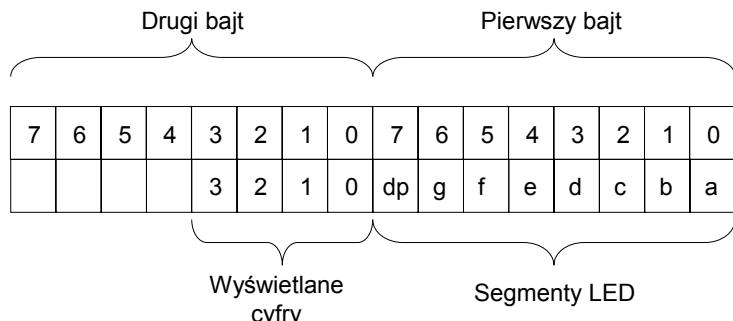
```

Sygnal *SS* powodujący uaktywnienie wejść rejestrów szeregowego zmieniany jest przed rozpoczęciem nadawania pierwszego bajtu dwubajtowej transmisji. Jego ustawienie powoduje przepisanie wcześniej wprowadzonych do rejestrów danych do zatrzasków wyjściowych, w efekcie dane te pojawiają się na wyjściach równoległych rejestrów. Następnie rejestr jest uaktywniany poprzez podanie stanu niskiego na jego wejście *RCK*, co umożliwia wprowadzenie kolejnych bajtów.

Pierwszy wysyłany bajt określa stan segmentów (bity równe 0 odpowiadają segmentom włączonym), drugi bajt określa cyfrę, która ma zostać włączona — rysunek 20.8.

**Rysunek 20.8.**

*Struktura transmitowanych do rejestrów przesywanych danych. Segment wyświetlacz jest włączony, jeśli odpowiadający mu bit ma wartość 0. Dana cyfra wyświetlacz jest włączona, jeśli odpowiadający jej bit ma wartość 0*



Powyższa procedura obsługi przerwania SPI jest funkcją „samonapędzającą się”. Wpisanie nowej wartości do rejestrów SPDR powoduje jej wysłanie na magistralę, w efekcie po zakończeniu transmisji generowane jest kolejne przerwanie. Proces ten trwa cyklicznie w nieskończoność.

Pozostaje jeszcze przetestować powyższe funkcje:

```

int main()
{
    SPI_master_init();
    sei();
}

```

```
LEDDIGITS[0]=1;  
LEDDIGITS[1]=2;  
LEDDIGITS[2]=3;  
LEDDIGITS[3]=4;  
  
SPDR=0; //Zainicjuj przerwania SPI  
  
while(1);  
}
```

Po inicjalizacji interfejsu SPI (funkcja `SPI_master_init()`) odblokowywane są przerwania. Proces wyświetlania inicjalizowany jest poprzez wpisanie dowolnej wartości do rejestru SPDR, co w efekcie powoduje wygenerowanie przerwania o wektorze `SPI_STC_vect`. Dalsze przerwania generowane są automatycznie. Końcowa instrukcja `while(1);` jest niezbędna — bez niej program wszedłby w nieskończoną pętlę z zablokowanymi przerwaniami.

## Interfejs USART w trybie SPI

W nowszych procesorach AVR interfejs USART ma specjalny tryb pracy, w którym może emulować interfejs SPI — tryb MSPIM (ang. *USART in Master SPI Mode*). Jeśli w takim mikrokontrolerze znajduje się interfejs SPI, to interfejsu USART można używać niezależnie, gdyż wykorzystuje on inne wyprowadzenia (przeznaczone do realizacji interfejsu USART). Dzięki temu można uzyskać dodatkowy interfejs SPI. Wykorzystanie interfejsu USART w trybie emulacji SPI wiąże się z pewnymi różnicami w stosunku do sprzętowego interfejsu SPI:

- ◆ Interfejs USART ma podwójne buforowania nadajnika i odbiornika, co umożliwia zwiększenie prędkości transmisji. Procesor przed wpisaniem kolejnego bajtu nie musi czekać na zakończenie poprzedniej transmisji, w efekcie kolejne bajty wysyłane są szybciej.
- ◆ Obecność podwójnego buforowania może potencjalnie stwarzać problemy w synchronizacji pomiędzy nadajnikiem a odbiornikiem. Zostanie to szerzej omówione w dalszej części rozdziału.
- ◆ Emulowana jest wyłącznie praca USART w trybie SPI *master*, można natomiast dowolnie określić tryb pracy SPI (0 – 3).
- ◆ Prędkość interfejsu określa rejestr UBRR, analogicznie jak w przypadku interfejsu USART.
- ◆ System przerwań wykorzystuje wektory USART, można je wykorzystać analogicznie do zwykłego trybu pracy interfejsu. Należy jednak pamiętać, że przerwania USART są wywoływanie w nieco innych punktach czasowych niż przerwania interfejsu SPI.

Zależność wykorzystywanych pinów w trybie emulacji z sygnałami magistrali SPI pokazano w tabeli 20.2.

**Tabela 20.2.** Zależność pomiędzy wyprowadzeniami interfejsu USART a sygnałami magistrali SPI

USART w trybie SPI	SPI	Opis
<i>TxD</i>	<i>MOSI</i>	Wyłącznie w trybie <i>master</i>
<i>RxD</i>	<i>MISO</i>	Wyłącznie w trybie <i>master</i>
<i>XCK</i>	<i>SCK</i>	Funkcje takie same
brak	<i>SS</i>	Sygnal nie jest wspierany. Można go emulować wyłącznie programowo, nie występuje automatyczna detekcja kolizji.

## Taktowanie magistrali SPI

Zegar taktujący magistralę SPI w trybie emulacji określany jest wartością rejestrów UBRRH i UBRRL oraz wartością bitu U2X rejestru UCSRA. Uzyskana prędkość jest identyczna z uzywanej w trybie USART. Sygnał zegarowy pojawia się na pinie *XCK*, a normalna funkcja tego pinu jako portu *I/O* jest zablokowana. Aby jednak sygnał ten był dostępny na wyprowadzeniu *XCK* należy ustawić kontrolujący go bit rejestru DDR. Pożądane jest, aby pin ten ustawić jako wyjście przed włączeniem trybu MSPIM. Sygnał z pinu *XCK* należy doprowadzić do wejść SCK układów *slave*.

## Tryb pracy SPI

W trybie MSPIM interfejs może emulować wszystkie tryby SPI (0 – 3). W tym celu należy skonfigurować bity UCPL oraz UCPLA, których funkcjonalność odpowiada bitom CPOL i CPHA klasycznego interfejsu SPI.



Jakakolwiek zmiana stanu tych bitów w trakcie transmisji powoduje jej uszkodzenie.

Z tego powodu przed zmianą konfiguracji interfejsu należy sprawdzić, czy wszystkie transmisje zostały zakończone.

## Format ramki danych

W trybie MSPIM ramka danych ma zawsze 8 bitów. Zamiast jej długości można konfigurować kolejność bitów, tak aby transmisja rozpoczęła się od bitu najmniej znaczącego lub najbardziej znaczącego. Konfigurację tę określa bit UDORD (ang. *Data Order*) rejestru UCSRC. Jego ustawienie powoduje transmisję, począwszy od bitu najmniej znaczącego — jest to domyślny stan tego bitu.

Dzięki buforowaniu interfejsu USART można jednocześnie nadawać 16 bitów (dwa bajty). Jest to pewna odmiennosć w stosunku do interfejsu SPI. Pamiętać jednak należy o synchronizacji nadajnika z odbiornikiem. Każdemu nadanemu bajtowi powinien towarzyszyć odczyt odebranego bajtu z bufora odbiornika. Bufory te nie działają na zasadzie buforów pierścieniowych, lecz nowa wartość nadpisuje ostatnią wartość w buforze.

W efekcie, jeśli nadane zostaną np. 3 bajty, to w buforze odbiornika znajdą się bajty 1 i 3 wysłane przez urządzenie *slave*. Bajt odebrany podczas transmisji bajtu 2 zostanie zgubiony i dojdzie do rozsynchonizowania nadajnika z odbiornikiem.

## Konfiguracja interfejsu

Interfejs USART w trybie MSPIM można włączyć po skonfigurowaniu wcześniej pokazanych etapów, tj. wybraniu trybu SPI i sposobu wysyłania danych (LSB, MSB). Włączenie interfejsu odbywa się poprzez selekcję trybu emulacji SPI dzięki ustawieniu w rejestrze UCSRC bitów UMSL00 i UMSL01. Po tym należy włączyć nadajnik, dodatkowo można włączyć także odbiornik (jeśli będziemy coś odbierać z magistrali SPI). **Po włączeniu interfejsu, a przed pierwszą transmisją należy ustawić prędkość transmisji.** Jest to pewna odmienność w stosunku do zwykłego trybu pracy USART, gdzie prędkość można było ustawić na dowolnym etapie.



Przed zmianami prędkości transmisji należy zadbać, aby rejestr UBRR zawierał 0.

Wskazówka

Zapewnia to poprawną pracę układu generowania zegara — bez tego czas nadawania pierwszego bitu może być zmieniony.



Podobnie jak w zwykłym trybie pracy USART, włączenie nadajnika i odbiornika powoduje przejęcie kontroli nad związanymi z nimi pinami *I/O* przez interfejs USART.

Po włączeniu interfejsu można z niego korzystać analogicznie jak w przypadku jego pracy w zwykłym trybie USART. Jedynie bity rejestru stanu informujące o zdarzeniach specyficznych dla trybu USART (bity FE, DOR, UPE) zawsze mają wartość 0 — zdarzenia te nie są obsługiwane w trybie SPI. Natomiast w pełni można korzystać ze zdarzeń wywołujących przerwania (odbiór znaku, nadanie znaku, pusty bufor nadawczy).

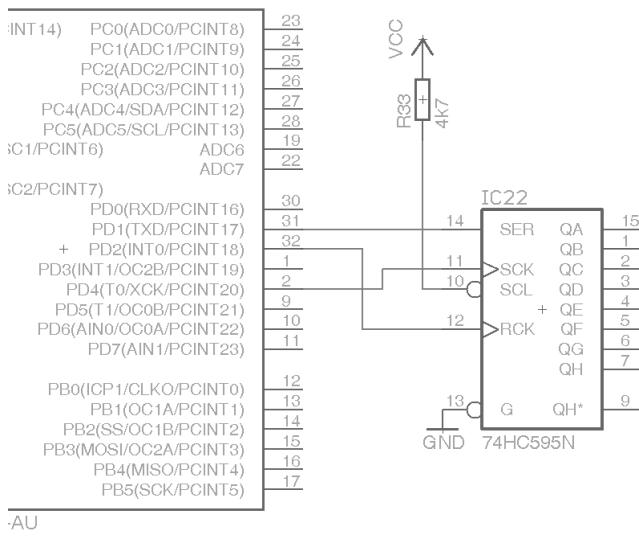
Dla przykładu pokazane zostanie wykorzystanie trybu MSPIM do komunikacji z urządzeniem *slave*, jakim jest znany już rejestr 74HC595 — rysunek 20.9.

Na początku należy zainicjować ten tryb pracy interfejsu USART:

```
static void uart_1M()
{
#define BAUD 1000000UL
#include <util/setbaud.h>
UBRR0H=UBRRH_VALUE;
UBRR0L=UBRRL_VALUE;
#if USE_2X
UCSROA|=BV(U2X0);
#else
UCSROA&=~(BV(U2X0));
#endif
}
void usart_MSPIM_Init()
{
```

**Rysunek 20.9.**

*Połączenie mikrokontrolera ATmega88 z rejestrem przesuwnym 74HC595 przy pomocy interfejsu USART pracującego w trybie MSPIM*



.AU

```

UBRR0=0;
DDRD|= _BV(PD4); //SCK jest wyjściem
UCSRC=_BV(UMSEL01) | _BV(UMSEL00) | _BV(UCPH0) | _BV(UCPO0); //Tryb SPI 0
UCSR0B=_BV(RXEN0) | _BV(TXEN0); //Włącz nadajnik i odbiornik
uart_1M(); //Prędkość transmisji 1 Mbps
}

```

Powyższa funkcja włącza obsługę trybu emulacji SPI, zapewniając jednocześnie właściwą konfigurację pinów I/O. Do ustalenia prędkości interfejsu posłużono się znymi z rozdziału 19. funkcjami zdefiniowanymi w pliku nagłówkowym `<util/setbaud.h>`. Po inicjalizacji interfejsu można nawiązać komunikację przy pomocy funkcji jednocześnie wysyłającej i odbierającej bajt danych:

```

uint8_t usart_Transmit(uint8_t data)
{
    while(!(UCSR0A & _BV(UDRE0))); //Czy bufor jest pusty?
    UDR0=data;
    while(!(UCSR0A & _BV(RXC0))); //Zaczekaj na odebranie bajtu
    return UDR0;
}

```

Budowa tej funkcji (oczekiwanie na ustawienie flagi RXC) zapewnia synchronizację bufora nadajnika i odbiornika.

Posiadając te dwie funkcje, można przystąpić do testów interfejsu. Poniższa funkcja:

```

int main()
{
    uint8_t data=0;
    PORTD|= _BV(PD2); //Pin SS dla urządzenia slave
    DDRD|= _BV(PD2);
    usart_MSPIM_Init();
    while(1)
    {
}

```

```
PORTD&=(~_BV(PD2)); //Zaadresuj urzadzenie  
uart_Transmit(data);  
PORTD|= _BV(PD2); //Wylacze  
data^=0xFF;  
_delay_ms(1000);  
}  
}
```

powoduje wysyłanie do układu *slave* w odstępach sekundowych naprzemiennie wartości 0 i 255. Sygnał SS powodujący wybór układu *slave* generowany jest programowo, poprzez sterowanie portem PD2 procesora.

# Rozdział 21.

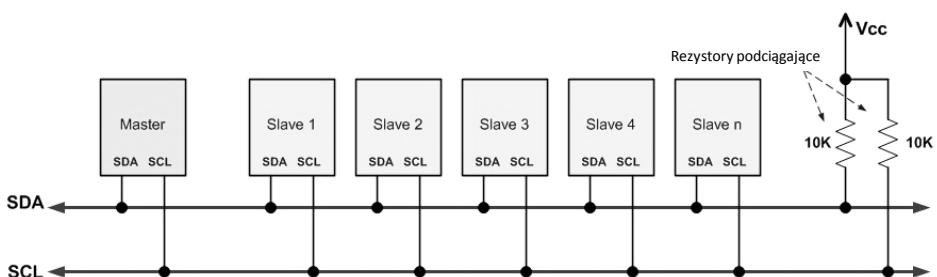
# Interfejs TWI

Jednym z najpopularniejszych interfejsów szeregowych jest interfejs I2C, który przez firmę Atmel nazywany jest interfejsem TWI (ang. *Two Wire Serial Interface*). W jego skład wchodzą urządzenia *master* i *slave*, przy czym na magistrali może być jednocześnie wiele urządzeń danego typu. **Interfejs składa się z dwóch dwukierunkowych linii: danych SDA (ang. *Serial Data*) i zegara SCL (ang. *Serial Clock*)**. Jest więc interfejsem synchronicznym, co umożliwia taktowanie wykorzystującego go procesora z niezbyt stabilnych źródeł, takich jak wbudowany generator RC. Ze względu na swoją budowę interfejs ten nadaje się do komunikacji na małe odległości (do kilkunastu cm). Komunikacja na większe odległości wymaga stosowania specjalnych zabiegów, m.in. wzmacniaczy sygnału. Pracuje on w dwóch trybach: standardowym, do ok. 100 kHz, i szybkim, do ok. 400 kHz. Istnieje także tryb o wysokiej prędkości (nawet 3,4 MHz), lecz tylko nieliczne układy potrafią z niego korzystać. **Interfejs sprzętowy procesorów AVR obsługuje magistralę I2C z maksymalną prędkością 400 kHz**.



Wykorzystując układy I2C, musimy zawsze sprawdzić, z jaką maksymalną częstotliwością magistrali mogą pracować.

Linie SDA i SCL są liniami typu *open-drain*, to znaczy, że zarówno *master*, jak i *slave* wymuszają na nich wyłącznie stan niski; stan wysoki wymuszany jest przez tzw. rezystory podciagające. Taka konstrukcja umożliwia pracę w trybie **multimaster**. Strukturę magistrali pokazano na rysunku 21.1.



Rysunek 21.1. Schemat magistrali I2C

Do magistrali teoretycznie jednocześnie może być przyłączonych do 128 urządzeń<sup>1</sup>. W praktyce, ze względu na ograniczone możliwości adresowania i pojemność magistrali (do 400 pF), w sposób pewny podłączyć można kilka do kilkunastu urządzeń I2C. Przyłączenie większej liczby urządzeń lub wydłużenie magistrali wymaga zastosowania repeterów (np. układ PCA9515). Wszystkie dane przesypane tym interfejsem są synchronizowane sygnałem zegara (*SCL*). **Każde urządzenie I2C ma swój unikalny 7-bitowy adres. 4 bity tworzą tzw. identyfikator urządzenia. Jest on nadawany przez producenta układu i umożliwia zorientowanie się co do typu urządzenia (pamięć, przetwornik I2C, zegar itp.). Pozostałe 3 bity to tzw. fizyczny adres urządzenia.** Umożliwiają one jednoczesne przyłączenie do magistrali wielu urządzeń danego typu.



**Wskaźówka** Specjalnym adresem jest adres 0. Po jego wysłaniu przez urządzenie *master* wszystkie urządzenia *slave* nadają sygnał *ACK* informujący o ich obecności.

**Najmłodszy bit adresu służy do wyboru typu operacji. Jeśli wynosi on 1, to następna operacja jest operacją odczytu, jeśli 0, to zapisu.**

Zwykle układy I2C mają wyprowadzone linie adresowe *A0 – A2*, których stan określa adres fizyczny urządzenia (tabele 21.1 i 21.2).

**Tabela 21.1.** Bity adresowe *A0 – A2* i adres fizyczny urządzenia

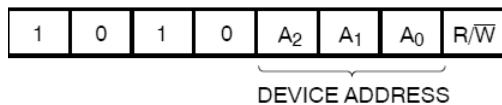
A2	A1	A0	Nr urządzenia
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	4
1	0	1	5
1	1	0	6
1	1	1	7

W niektórych układach może być wyprowadzona mniejsza liczba linii adresowych lub może ich nie być wcale — w takim przypadku nie możemy zmieniać adresu urządzenia, przez co na magistrali może być tylko jedno urządzenie danego typu.

Na magistrali dane mogą być przesypane pomiędzy urządzeniami *master* i *slave*, ale to *master* inicjuje transmisję i jest odpowiedzialny za generowanie sygnału zegara. *Slave* sam nigdy nie inicjuje transmisji, natomiast może ją zwolnić, przedłużając niski stan na linii *SCK*. Dzięki temu *master* wie, że urządzenie nie nadaje z transmisją lub transmitowane dane nie są jeszcze gotowe.

<sup>1</sup> Istnieje specjalny tryb adresowania, umożliwiający zaadresowanie większej liczby urządzeń, nie występuje on jednak w standardowej implementacji I2C.

**Tabela 21.2.** Ustawienia bitów adresowych A0 – A2 i adresy układu w przypadku pamięci 24C256. Zmiana stanu linii A0 – A2 umożliwia przyłączenie do jednej magistrali aż 8 takich samych układów pamięci



A <sub>2</sub>	A <sub>1</sub>	A <sub>0</sub>	Adres urządzenia
0	0	0	0xA0
0	0	1	0xA2
0	1	0	0xA4
0	1	1	0xA6
1	0	0	0xA8
1	0	1	0xAA
1	1	0	0xAC
1	1	1	0xAE



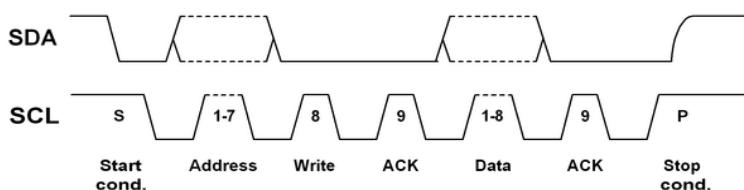
Zdarza się, że wyprowadzenia A0 – A2 układu I2C mają wewnętrzne podciąganie do 1 logicznej. Jeśli pozostawimy je niepodłączone, to układ odczyta wartość 111.

Transfer danych na szynie odbywa się według pewnego schematu:

1. Master wysyła bit startu.
2. Następnie wysyłany jest adres wybranego urządzenia *slave*.
3. Master wysyła bit określający typ operacji: 1 — odczyt, 0 — zapis.
4. Slave wysyła bit ACK potwierdzający odebranie danych i gotowość urządzenia.
5. Master, sterując linią SCL, nadaje/odbiera 8 bitów danych.
6. Następnie odbierany jest lub wysyłany (w zależności od typu wcześniejszej operacji) bit ACK.
7. Master wysyła sygnał stop.

Sekwencje 5 i 6 mogą być wielokrotnie powtarzane w ramach tej samej transakcji, umożliwiając jednorazowe przesyłanie większej liczby bajtów. Schematycznie wymiana danych na magistrali została pokazana na rysunku 21.2.

**Rysunek 21.2.**  
Przebieg transmisji na magistrali I2C



Jak widać, odbiór każdego bajtu potwierdzany jest wysłaniem przez odbiornik (zależnie od sytuacji może to być układ *master* lub *slave*) bitu *ACK* (ang. *Acknowledge*). Sygnał ten odpowiada nadaniu bitu o wartości 0. Jeśli odbiornik nie wyśle spodziewanego sygnału *ACK*, nadajnik powinien potraktować taką sytuację jako błąd i zresetować stan magistrali, nadając ponownie bit startu i ponawiając transmisję. Istnieje także druga możliwość — po odebraniu bajtu danych odbiornik nadaje sygnał *NACK* (ang. *Negative Acknowledge*), odpowiadający nadaniu bitu o wartości 1. Zwykle jest on nadawany przez urządzenie *master* po odebraniu porcji danych od urządzenia *slave*, w sytuacji, w której *master* nie oczekuje odbioru kolejnych danych. Zwykle po nadaniu przez urządzenie *master* sygnału *NACK* nadawany jest sygnał *STOP* kończący transmisję do urządzenia *slave*. **Należy odróżnić sygnał *NACK* od jakiegokolwiek braku odpowiedzi — w przypadku nadania *NACK* zmienia się stan linii *SCL*.**



Definicje związane z interfejsem TWI znajdują się w pliku nagłówkowym `<util/twi.h>`.

Zawiera on definicje stanów magistrali TWI podzielone na 4 grupy:

- ◆ Definicje zaczynające się od *TW\_MT\_x* — dotyczą nadajnika urządzenia *master*.
- ◆ Definicje zaczynające się od *TW\_MR\_x* — dotyczą odbiornika urządzenia *master*.
- ◆ Definicje zaczynające się od *TW\_ST\_x* — dotyczą nadajnika urządzenia *slave*.
- ◆ Definicje zaczynające się od *TW\_SR\_x* — dotyczą odbiornika urządzenia *slave*.

Dodatkowo w pliku tym znajdują się dwie użyteczne definicje. Ponieważ organizacja rejestru TWSR (ang. *Two Wire Status Register*) w różnych procesorach AVR jest różna, wprowadzono definicję *TW\_STATUS\_MASK*, będącą maską bitową umożliwiającą uzyskanie z rejestru TWSR tylko bitów stanu. Oprócz niej możemy także skorzystać z *TW\_STATUS*, które jest iloczynem bitowym rejestru TWSR i *TW\_STATUS\_MASK* (*TWSR & TW\_STATUS\_MASK*).



Implementując interfejs I2C, nie zapomnij o podłączeniu rezystorów podciągających na liniach *SDA* i *SCL*. Bez nich układ nie będzie działał poprawnie.

**Minimalna wartość tych rezystorów wynosi (Vcc–0,4 V)/3 mA, wartość maksymalna zależy od częstotliwości taktowania magistrali. Dla częstotliwości <100 kHz wynosi 1000 ns/C magistrali, powyżej 100 kHz wynosi 300 ns/C magistrali.**

## Tryb multimaster

Dzięki temu, że na magistrali stan 0 jest dominujący, a 1 recesywny, istnieje możliwość łatwego wprowadzenia trybu multimaster. Ma on sens praktycznie tylko w układach wieloprocesorowych, kiedy więcej niż jeden procesor może pracować w roli mastera. Wiąże się to z tzw. arbitrażem oraz koniecznością synchronizacji zegarów generowanych przez poszczególne urządzenia. W przypadku magistrali I2C jest to stosunkowo

proste — urządzenie, które nadaje, jednocześnie bada stan magistrali. W przypadku rozbieżności zawiesza dalsze nadawanie, co umożliwia przejęcie roli mastera innemu układowi.

## Iinicjalizacja interfejsu

Włączenie interfejsu TWI powoduje przejęcie przez ten interfejs kontroli nad pinami  $I_0$  współdzielonymi z liniami *SDA* i *SCL*. Jednakże wpisując wartość 1 na odpowiadających im bitach rejestru *PORT*, możemy aktywować wewnętrzne rezystory podciągające, co czasami eliminuje konieczność stosowania zewnętrznych. Ze względu na dużą rezystancję tych rezystorów będą one działać tylko przy bardzo wolnych częstotliwościach taktowania magistrali. Do poprawnej pracy musimy także ustalić częstotliwość taktowania magistrali.



Jeżeli procesor jest urządzeniem *slave*, częstotliwość narzuca urządzenie *master*, lecz musi ona być co najmniej 16 razy mniejsza niż częstotliwość taktowania procesora.

Częstotliwość pracy magistrali zależy od wartości rejestru *TWBR* (ang. *TWI Bit Rate Register*) oraz preskalera, którego konfigurację określa rejestr *TWSR* (ang. *TWI Status Register*). Częstotliwość magistrali wynosi  $F_{CPU}/(16+2 \cdot TWBR \cdot \text{Preskaler})$ .



Jeśli nie możesz skomunikować się z układami I2C w budowanym układzie, spróbuj obniżyć częstotliwość taktowania magistrali.

Interfejs TWI może także generować przerwania. W tym celu należy ustawić flagę *TWIE* (ang. *TWI Interrupt Enable*) w rejestrze *TWCR*. W takim przypadku po każdej zakończonej transmisji wygenerowane będzie przerwanie o wektorze *TWI\_vect*.

Wywołanie procedury obsługi przerwania następuje przy ustawionym bicie *TWINT*. Nie jest on kasowany automatycznie, należy go wyzerować programowo. Powoduje to jednocześnie inicjalizację kolejnej operacji na interfejsie TWI. Stąd przed jej wyzerowaniem należy odczytać dane z rejestru *TWDR* i stan interfejsu z rejestru *TWSR*.

Interfejs TWI odblokowujemy, ustawiając bit *TWEN* (ang. *TWI Enable Bit*) w rejestrze *TWCR*.

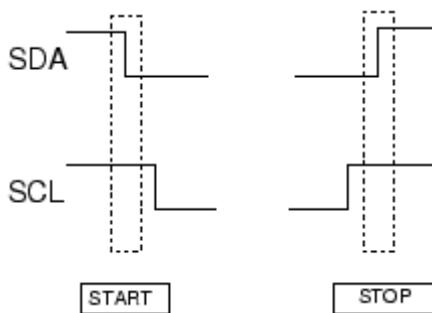
## Procesor w trybie I2C master

### Bity START i STOP

Każdą transmisję na magistrali rozpoczyna specjalny bit *START*, a kończy bit *STOP* (rysunek 21.3).

**Rysunek 21.3.**

Struktura bitów  
*START* i *STOP*



Bity te mogą zostać wygenerowane wyłącznie przez urządzenie *master*. Otrzymanie przez układ *slave* bitu *START* powoduje zresetowanie układu kontrolującego transmisję I2C; bit ten może być nadany w każdej chwili. Dzięki temu, jeśli coś zakłóci transmisję, zawsze można rozpocząć ją od początku. Jego nadanie polega na utrzymaniu linii *SCL* w stanie wysokim, przy jednoczesnym przejściu linii *SDA* ze stanu wysokiego do niskiego. Sygnał ten jest generowany automatycznie przez kontroler procesora.

Sygnalem kończącym transmisję jest nadanie przez mastera sygnału *STOP*. Polega on na zmianie stanu linii *SDA* z niskiego na wysoki przy jednoczesnym stanie wysokim na linii *SCL*.

Istnieje jeszcze jeden sygnał specjalny — **powtórzony START** (ang. *Repeated START*), lecz ma on zastosowanie praktycznie tylko w systemach multimaster. W systemie z jednym układem *master* odpowiada on sygnałowi *START*.

Po tej porcji informacji teoretycznych przejdźmy do kilku praktycznych przykładów łączenia różnych układów I2C z mikrokontrolerem AVR pracującym w trybie *master*.



**Wskazówka** Kody umieszczone w dalszej części tego rozdziału znajdują się w dołączonym do książki pliku *I2C\_Master.zip*.

Zacznijmy od stworzenia kilku uniwersalnych funkcji ułatwiających pracę z urządzeniami I2C.

## Podstawowe funkcje do współpracy z I2C

Niezależnie od typu obecnych na magistrali układów potrzebujemy zestawu uniwersalnych funkcji umożliwiających obsługę transmisji I2C. Pierwszą niezbędną funkcją jest funkcja inicjująca interfejs I2C, nazwijmy ją *I2C\_Init()*:

```
void I2C_Init()
{
    TWCR = _BV(TWEA) | _BV(TWEN); //Włącz interfejs I2C
    I2C_SetBusSpeed(I2CBUSCLOCK/100);
}
```

Powyższa funkcja włącza interfejs TWI (ustawienie bitu TWEN) oraz włącza automatyczne generowanie bitu ACK (ustawienie bitu TWEA). W funkcji tej została wywołana także druga niezbędna funkcja, określająca częstotliwość taktowania magistrali TWI (I2C\_SetBusSpeed), której definicja wygląda następująco:

```
void I2C_SetBusSpeed(uint16_t speed)
{
    speed=(F_CPU/speed/100-1)/2; //speed=TWBR·4^TWPS
    uint8_t prescaler=0;
    while(speed>255) //Oblicz wartość preskalera
    {
        prescaler++;
        speed=speed/4;
    };
    TWSR=(TWSR & (_BV(TWPS1)|_BV(TWPS0))) | prescaler;
    TWBR=speed;
}
```

Jako argument funkcja ta przyjmuje częstotliwość pracy magistrali TWI podzieloną przez 100. Na tej podstawie wylicza wartość preskalera oraz rejestr TWBR odpowiedzialnego za generowanie sygnału zegarowego. Jeśli nie zamierzamy zmieniać prędkości magistrali TWI w trakcie jej pracy, to powyższą funkcję możemy uproszczyć, wpisując do rejestrów TWBR i TWSR wyliczone wcześniej wartości. Powyższa funkcja do prawidłowego działania wymaga zdefiniowania symbolu F\_CPU określającego prędkość taktowania procesora.

Po zainicjowaniu interfejsu i ustawieniu jego prędkości potrzebujemy funkcji umożliwiającej wygenerowanie sygnału *START*:

```
void I2C_Start()
{
    TWCR = _BV(TWINTE) | _BV(TWSTA) | _BV(TWEN);
    I2C_WaitForComplete();
    if (!TW_STATUS != TW_START) I2C_SetError(I2C_STARTError);
}
```

Sygnal *START* generowany jest automatycznie przez procesor poprzez ustawienie bitu TWSTA (ang. *TWI START Condition Bit*). Następnie czekamy na nadanie sygnału (funkcja I2C\_WaitForComplete()) i sprawdzamy rejestr statusu, czy nie wystąpił błąd. Jeśli wszystko przebiegło prawidłowo, to rejestr statusu powinien zawierać wartość równą definicji TW\_START. Inna wartość oznacza błąd (np. zajętą magistralę) i wymaga powtórzenia próby. Funkcja I2C\_WaitForComplete wygląda następująco:

```
static inline void I2C_WaitForComplete() {while (!(TWCR & _BV(TWINTE))):};
```

Jest to prosta funkcja *inline*, której jedyną funkcją jest czekanie na wyzerowanie flagi TWINT sygnalizującej zakończenie operacji na magistrali TWI.

Skoro dysponujemy już funkcją wysyłającą sygnał *START*, to potrzebujemy też funkcji wysyłającej sygnał *STOP*:

```
static inline void I2C_Stop() {TWCR = _BV(TWINTE) | _BV(TWEN) | _BV(TWSTO);};
```

Funkcja ta korzysta ze sprzętowej realizacji sygnału *STOP* (ustawienie bitu TWSTO). Dla sprawdzenia, czy operacja wysyłania sygnału *STOP* została zakończona, nie możemy

wykorzystywać bitu TWINT. Zakończenie tej operacji sygnalizowane jest wyzerowaniem bitu TWSTO, stąd potrzebujemy kolejnej funkcji, której zadaniem będzie oczekивание na zakończenie nadawania bitu *STOP*:

```
static inline void I2C_WaitTillStopWasSent() {while (TWCR & _BV(TWSTO));};
```

Możemy już inicjować i kończyć transmisję TWI. Kolejną funkcją, jakiej potrzebujemy, jest funkcja wysyłająca identyfikator wybranego urządzenia *slave*:

```
void I2C_SendAddr(uint8_t address)
{
    uint8_t Status;

    if((address & 0x01)==0) Status=TW_MT_SLA_ACK;
    else Status = TW_MR_SLA_ACK;

    TWDR=address;
    TWCR=_BV(TWINTE) | _BV(TWEN);
    I2C_WaitForComplete();
    if(TW_STATUS!=Status) I2C_SetError(I2C_NACK);
}
```

Funkcja ta wpisuje identyfikator wybranego urządzenia *slave* (jego adres) do rejestru TWDR, z którego adres zostanie wysłany na magistralę TWI, a następnie czeka na zakończenie transmisji i sprawdza rejestr statusu, czy nie wystąpił błąd. Sprawdzenie błędu jest nieco skomplikowane. W zależności od tego, czy urządzenie wybieramy w trybie do odczytu (najmłodszy bit adresu równy 1), czy do zapisu (najmłodszy bit adresu równy 0), prawidłowy wynik operacji sygnalizowany jest w inny sposób. Jeżeli wybieramy urządzenie w trybie do zapisu, prawidłowa operacja sygnalizowana jest wartością rejestrów stanu równą *TW\_MT\_SLA\_ACK*, w przypadku wyboru urządzenia w trybie do odczytu prawidłowy przebieg operacji sygnalizuje wartość *TW\_MR\_SLA\_ACK*.

Ponieważ zawsze po wysłaniu sygnału *START* musimy wysłać adres wybieranego urządzenia *slave*, wygodnie jest stworzyć funkcję, która przeprowadza obie operacje równocześnie:

```
void I2C_SendStartAndSelect(uint8_t addr)
{
    I2C_Start();
    I2C_SendAddr(addr);
}
```

Możemy już rozpoczęć transmisję, wybrać urządzenie *slave*, teraz przydałaby się jeszcze możliwość zapisania do niego danych. Ponieważ prawidłowy zapis danych generuje inny komunikat w rejestrze stanu niż zapis adresu, potrzebujemy więc oddzielnej funkcji:

```
void I2C_SendByte(uint8_t byte)
{
    TWDR=byte;
    TWCR=_BV(TWINTE) | _BV(TWEN);
    I2C_WaitForComplete();
    if(TW_STATUS!=TW_MT_DATA_ACK) I2C_SetError(I2C_NoACK);
}
```

Widzimy, że powyższa funkcja jest podobna do funkcji wysyłającej adres urządzenia, różni się wyłącznie interpretacją komunikatu w rejestrze stanu.

Ostatnią niezbędną funkcją jest funkcja realizująca odczyt danych z urządzenia *slave*. Operacje odczytu na magistrali I2C dzielą się na dwa typy: odczyt z wygenerowaniem sygnału potwierdzenia (*ACK*) oraz odczyt z wygenerowaniem sygnału *NACK*. Kiedy odczytujemy tylko jeden bajt danych, urządzenie *master* nie potwierdza otrzymania danych, nie generuje więc sygnału *ACK*, dzięki czemu urządzenie *slave* albo oczekuje na sygnał kończący transmisję (*STOP*), albo po prostu samo przerywa transmisję. W tym przypadku kontynuacja wymiany danych możliwa jest po otrzymaniu kolejnego sygnału *START* wraz z adresem urządzenia. Funkcja realizująca odczyt 8-bitowej danej bez generowania *ACK* wygląda następująco:

```
uint8_t I2C_ReceiveData_NACK()
{
    TWCR=_BV(TWINT) | _BV(TWEN);
    I2C_WaitForComplete();
    if (TW_STATUS!=TW_MR_DATA_NACK) I2C_SetError(I2C_NoNACK);
    return TWDR;
}
```

Jeżeli chcemy odczytać naraz wiele danych, każdy otrzymany bajt urządzenie *master* musi potwierdzić, generując sygnał *ACK*. Tylko ostatni bajt danych należy odebrać bez wygenerowania sygnału *ACK*, co zakończy transmisję. Zdefiniujmy więc funkcję odczytującą 1 bajt i generującą automatycznie potwierdzenie, co umożliwi transmisję kolejnego bajtu:

```
uint8_t I2C_ReceiveData_ACK()
{
    TWCR=_BV(TWEA) | _BV(TWINT) | _BV(TWEN);
    I2C_WaitForComplete();
    if(TW_STATUS!=TW_MR_DATA_ACK) I2C_SetError(I2C_NoACK);
    return TWDR;
}
```

W ten sposób dysponujemy podstawowymi funkcjami umożliwiającymi realizację dowolnej transmisji na magistrali TWI. Przedstawione powyżej funkcje sprawdzają poprawność operacji, w przypadku błędu umieszczają jego uproszczony kod w zmiennej globalnej *I2C\_Error* — po wykonaniu każdej z powyższych funkcji sprawdzenie tej zmiennej umożliwi nam zorientowanie się, czy operacja przebiegła prawidłowo, czy też nie. Dzięki definicjom:

```
#define I2C_STARTError 1
#define I2C_NoNACK 3
#define I2C_NoACK 4
```

Zmienna ta wskazuje na błąd wysyłania sygnału *START* (*I2C\_STARTError*), brak odebrania sygnału *ACK* (*I2C\_NoACK*) lub brak odebrania sygnału *NACK* (*I2C\_NoNACK*). Powyższe funkcje ustawiają kod błędu przy pomocy funkcji *I2C\_SetError* o następującej definicji:

```
inline void I2C_SetError(uint8_t err) { I2C_Error=err;};
```



Wskazówka

Wszystkie przykłady pokazujące wykorzystanie mikrokontrolerów AVR w trybie *master* bazują na powyższych funkcjach.

## Współpraca z zewnętrzna pamięcią EEPROM

Magistrala I2C umożliwia wykorzystanie różnych układów pamięci, w tym pamięci EEPROM o pojemnościach od kilkunastu bajtów do 128 kB. Ich zaletą jest nielotność zamieszczonych w niej danych przy zaniku zasilania. Pamięci te posiadają także wady, do których zalicza się ograniczoną liczbę operacji zapisu (zwykle do 100 tys.) oraz długi czas zapisu, wynoszący typowo kilka ms. Dostęp do pamięci I2C o pojemnościach większych niż 64 kB odbywa się z wykorzystaniem bajtu identyfikatora (adresu) układu. Takie układy podzielone są zazwyczaj na banki po 64 kB. Wybór banku odbywa się poprzez zmianę adresu układu pamięci.



Wskazówka

Pamięci o pojemnościach >64 kB zachowują się więc tak, jakby na magistrali znajdowało się kilka układów, z których każdy ma 64 kB.

Część układów pamięci ma dodatkowe wyprowadzenie **WP (ang. Write Protect)**, chroniące pamięć przed zapisaniem. W takich układach, aby odblokować możliwość zapisu, wyprowadzenie to powinno mieć niski poziom logiczny. Możemy nim sterować z pinu *I0* mikrokontrolera lub po prostu na stałe połączyć je z masą.

Pamięci I2C realizują 4 podstawowe operacje:

- ◆ operacja odczytu bajtu,
- ◆ operacja zapisu bajtu,
- ◆ operacja odczytu sekwencyjnego bajtów,
- ◆ operacja zapisu strony pamięci.

Schematycznie przebieg realizacji powyższych operacji pokazany jest na rysunku 21.4.

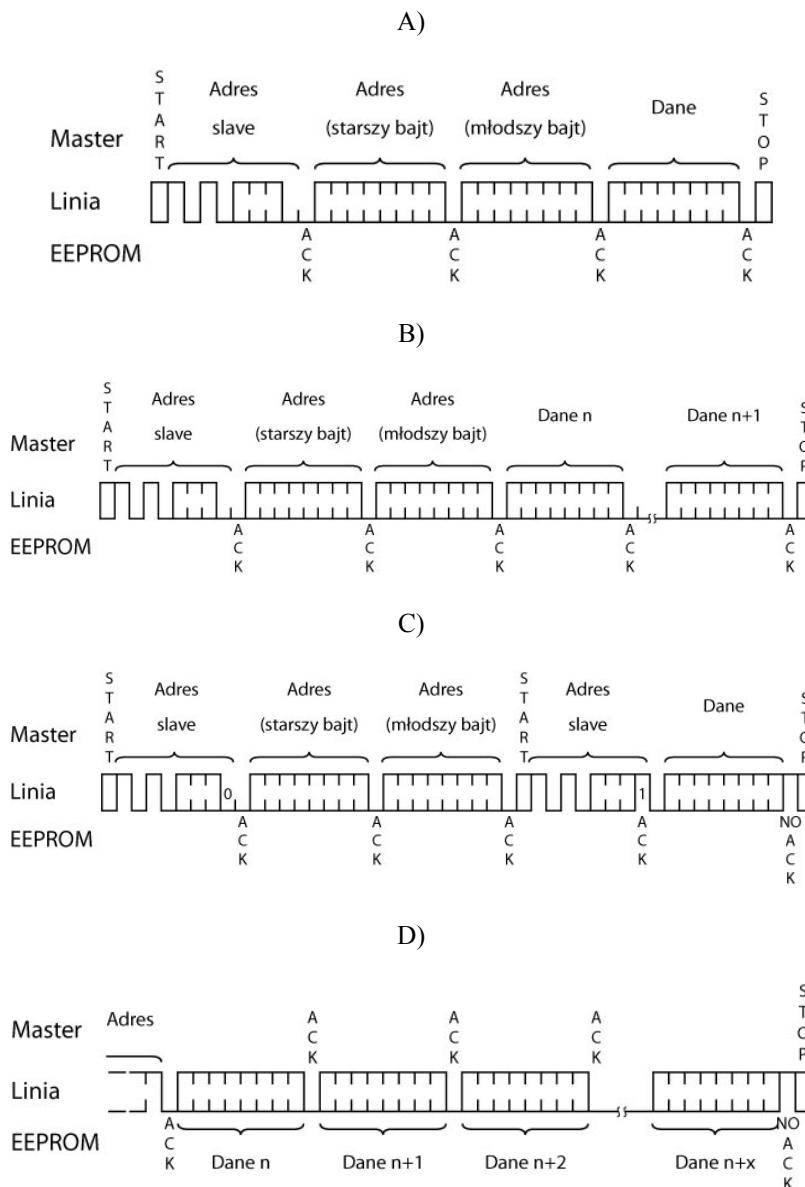
Szerszego omówienia wymaga operacja zapisu strony pamięci (rysunek 21.4B). Dla przyśpieszenia zapisu pamięć EEPROM podzielona jest na tzw. strony (ang. *Pages*) o długości zależnej od układu pamięci. Typowo strona jest kolejną potęgą liczby 2 i ma 16, 32 lub 64 bajty. Jeśli zapisujemy dane w obrębie jednej strony, to możemy jednocześnie wysłać więcej bajtów do zapisu. Zostaną one zapisane jednocześnie po zakończeniu transakcji. Jeśli jednak adres wykroczy poza granicę strony, to ulega on tzw. zawińciu (ang. *roll over*) — adresowany w efekcie będzie pierwszy bajt aktualnie wybranej strony. Adresy stron rozpoczynają się zawsze od wielokrotności długości strony, np. dla strony o długości 32 bajty kolejne adresy stron rozpoczynają się od adresów 0, 32, 64, 96 itd.



Wskazówka

W przypadku operacji zapisu pojedynczego bajtu lub strony cała transakcja musi być prawidłowo zakończona. Jej przerwanie anuluje polecenie zapisu.

Po zakończeniu transakcji zapisu dane fizycznie zapisywane są w pamięci; operacja ta jest czasochłonna, może trwać nawet 6 i więcej ms. **W tym czasie układ nie odpowiada**



**Rysunek 21.4.** Przebieg operacji zapisu i odczytu pamięci EEPROM. Rysunki A i B przedstawiają operację zapisu do pamięci. A) Po wysłaniu bitu START, adresu urządzenia i adresu zapisywanej komórki pamięci urządzenie oczekuje na dodatkowy bajt, który zostanie zapisany. Operację kończy wysłanie bitu STOP. B) Podobnie jak w A, z tym że po pierwszym bajcie danych wysyłane są kolejne. Zapis odbywa się w ramach jednej strony pamięci. C) Operacja odczytu jest dwuetapowa. W pierwszym etapie w urządzeniu ustawiany jest adres odczytywanej komórki pamięci. Po tym ponownie wysyłany jest bit START i adres urządzenia, następnie mikrokontroler może odczytać zawartość wybranej wcześniej komórki pamięci. Transakcję kończy wysłanie sygnału NACK. D) Jeżeli przeprowadzimy operację jak w punkcie C, lecz kolejne odebrane bajty będziemy potwierdzać sygnałem ACK, to pamięć będzie automatycznie inkrementować adres odczytywanej komórki i wysyłać kolejne dane, aż do chwili wysłania przez mikrokontroler sygnału NACK

**na żadne polecenia. Wysłanie kolejnego sygnału *START* i jego adresu nie generuje sygnału potwierdzenia (*ACK*).** Właściwość ta może służyć do określenia czasu, kiedy fizyczna operacja zapisu ulegnie zakończeniu.

Operacje odczytu są prostsze. Odczyt danych przebiega dwuetapowo. W pierwszym etapie ustalamy adres odczytywanej komórki pamięci. W tym celu wysyłamy bit *START*, adres urządzenia i adres odczytywanej komórki pamięci. W drugim etapie ponownie wysyłany jest bit *START* i adres układu, po czym pamięć wysyła 1 bajt odczytanych danych. Jeśli odbiór tego bajtu nie zostanie potwierdzony (wysłanie sygnału *NACK*), to transakcja jestkończona. Jeśli zostanie on potwierdzony poprzez wysłanie sygnału *ACK* przez mikrokontroler, to pamięć automatycznie inkrementuje adres i wysyła kolejny bajt danych, umieszczony pod adresem adres+1. Taką operację można powtarzać wieleokrotnie.



**W przypadku odczytu, odmiennie niż w przypadku zapisu, odczytywać możemy dowolną liczbę bajtów naraz; pamięć ma organizację liniową i nie jest podzielona na strony.**

Zauważmy, że pomiędzy polecienniem wyboru adresu odczytywanej komórki a kolejnym bitem *START* nie występuje bit *STOP*.

Jak widać, adres komórki pamięci jest 16-bitowy, daje więc to możliwość zaadresowania maksymalnie 65 536 komórek pamięci (64 kB). W przypadku pamięci o większej pojemności są one podzielone na bloki po 64 kB, różniące się adresem układu. Najstarszy bit adresu znajduje się więc w bajcie określającym adres układu pamięci I2C. **W przypadku pamięci o mniejszej pojemności niewykorzystane bity adresu są ignorowane.** Np. dla pamięci o pojemności 32 kB wartość najstarszego bitu adresu (*A15*) jest bez znaczenia. W efekcie adresy komórek 0 i 32 768 wskazują na tę samą fizyczną komórkę pamięci. Ponieważ pamięci I2C istnieją wyłącznie w wersjach o pojemnościach będących kolejnymi potegami liczby 2, stwarza to możliwość łatwej programowej detekcji pojemności zastosowanej w układzie pamięci.

Po tej porcji teorii przejdźmy do praktycznych przykładów. Zdefiniujemy proste funkcje umożliwiające odczyt/zapis pojedynczego bajtu lub bloku bajtów na wzór funkcji operujących na pamięci EEPROM mikrokontrolera. W przeciwieństwie do wewnętrznej pamięci EEPROM mikrokontrolera adres komórki w pamięci EEPROM I2C składa się z właściwego adresu komórki oraz adresu pamięci na szynie I2C. Dla wygody zadeklarujmy więc strukturę:

```
typedef struct
{
    uint8_t DevAddr; //Adres układu I2C
    void *MEMAddr; //Adres komórki pamięci
} MEM_addr;
```

Pole *DevAddr* przechowuje adres układu I2C i w praktyce po jego jednorazowym ustawieniu nie będziemy go zmieniać, natomiast pole *MEMAddr* zawiera adres komórki pamięci, do której się odwołujemy. Przyczyną, dla której nie stosujemy zmiennej przechowującej po prostu cały 24-bitowy adres, jest brak w języku C stosownego 24-bitowego typu danych. Najbliższy typ jest 32-bitowy, w efekcie przechowując adres, niepotrzeb-

nie marnowałibyśmy 1 bajt pamięci. Wszystkie poniżej zdefiniowane funkcje odwoływały się będą do pamięci poprzez strukturę `MEM_addr`. Na początek zdefiniujmy funkcję dokonującą wyboru adresowanej komórki pamięci:

```
void I2C_MEMORY_SetRWAddress(const MEM_addr *addr)
{
    do      //Wybierz urządzenie o wskazanym adresie
    {
        I2C_SendStartAndSelect(addr->DevAddr);
    } while(I2C_Error==I2C_NoACK); //Czekaj, aż urządzenie wyśle ACK

    I2C_SendByte(((uint16_t)addr->MEMAddr)>>8);
    I2C_SendByte(((uint16_t)addr->MEMAddr) & 0xFF);
}
```

Funkcji tej nigdy nie będziemy używać we własnym programie, została ona stworzona wyłącznie na potrzeby funkcji realizujących zapis/odczyt pamięci. Funkcje te wymagają wcześniejszego ustawienia adresu zapisywanych/odczytywanych danych, co realizuje funkcja `I2C_MEMORY_SetRWAddress`. Posiadając taką funkcję, możemy już łatwo zdefiniować funkcję realizującą odczyt 1 bajtu pamięci:

```
uint8_t I2C_MEMORY_read_byte(const MEM_addr *addr)
{
    I2C_MEMORY_SetRWAddress(addr); //Wybierz adres odczytywanej komórki
    I2C_SendStartAndSelect(addr->DevAddr | TW_READ); //Wykonaj operację Current Address Read
    uint8_t byte=I2C_ReceiveData_NACK();
    I2C_Stop();
    return byte;
}
```

Powyższa funkcja odbiera jeden bajt danych z pamięci, wysyłając sygnał *NACK*, co kończy transakcję. Podobnie możemy zdefiniować funkcję zapisu bajtu pamięci:

```
void I2C_MEMORY_write_byte(const MEM_addr *addr, uint8_t data)
{
    I2C_MEMORY_SetRWAddress(addr);
    I2C_SendByte(data);
    I2C_Stop();
    I2C_WaitTillStopWasSent(); //Poczekaj na zakończenie wysyłania sygnału STOP
}
```



**Pamiętajmy, że operacja zapisu jest czasochłonna, a w trakcie jej wykonywania układ nie odpowiada na żadne polecenia wysyłane magistralą I2C.**

Posiadając operacje odczytu i zapisu bajtu pamięci, możemy łatwo zdefiniować odpowiadające im operacje blokowe poprzez wielokrotne wywoływanie powyższych funkcji. Byłoby to jednak bardzo nieefektywne — za każdym razem musielibyśmy inicjować nową transakcję i wysyłać sporo niepotrzebnych danych, co niepotrzebnie wydłużałoby proces dostępu do pamięci, szczególnie w przypadku operacji odczytu. Zdefiniujmy więc operacje blokowe wykorzystujące wbudowane w układ pamięci ułatwienia. Operacja odczytu bloku pamięci:

```

void I2C_MEM_read_block(void *dst, MEM_addr *src, size_t len)
{
    I2C_SetRWAddress(src);
    I2C_SendStartAndSelect(src->DevAddr | TW_READ);
    while(--len)
    {
        *(uint8_t*)dst=I2C_ReceiveData_ACK();
        dst=((uint8_t*)dst)+1;
    }
    *(uint8_t*)dst=I2C_ReceiveData_NACK(); //Ostatni odczytywany bajt nie może zostać
                                            //potwierdzony, co kończy operację odczytu
    I2C_Stop();
}

```

Funkcja ta oprócz adresu początku odczytywanego bloku w pamięci EEPROM (`src`) pobiera także adres bloku w pamięci SRAM, w którym umieszczone będą odczytywane dane (`dst`) i długość odczytywanego bloku.



**Wskazówka**

Obszar pamięci wskazywany przez zmienną `dst` musi mieć co najmniej `len` bajtów, w przeciwnym przypadku nadpisane zostaną sąsiednie obszary pamięci, co powoduje nieprawidłowe działanie programu.

Powyższa funkcja cyklicznie odczytuje kolejne bajty danych, potwierdzając je wysłaniem sygnału *ACK*. Tylko po odczytaniu ostatniego bajtu wysyłany jest sygnał *NACK*, co kończy transmisję.

Nieco bardziej skomplikowany jest zapis bloku pamięci:

```

void I2C_MEM_write_block(const void *src, MEM_addr *dst, size_t len)
{
    size_t ps;
    size_t adres=(size_t)dst->MEMAddr;

    do
    {
        I2C_SetRWAddress(dst);
        ps=(adres | (I2C_PAGE_SIZE-1)) - adres; //Oblicz liczbę bajtów do końca strony pamięci
        adres+=ps; //Adres kolejnej strony pamięci
        if(ps>len) ps=len;
        len-=ps;
        while(ps--)
        {
            I2C_SendByte(*((uint8_t*)src));
            src=((uint8_t*)src)+1;
        }
        I2C_Stop();
        I2C_WaitTillStopWasSent(); //Zaczekaj do końca operacji wysyłania danych
    } while(len);
}

```

Ponieważ jednocześnie można zapisać komórki pamięci mieszczące się na tej samej stronie, funkcja dzieli zapisywany blok pamięci na poszczególne strony. W tym celu

wykorzystuje symbol `I2C_MEM_PAGE_SIZE`, który powinien zawierać długość strony w bajtach używanej pamięci. Kolejne strony wysyłane są w oddzielnych transakcjach kończących się zapisem.

## Współpraca z zewnętrzna pamięcią FRAM

Pamięć FRAM (ang. *Ferroelectric RAM*) jest typem pamięci nielotnej, pozabawionym wad pamięci EEPROM, takich jak ograniczona liczba zapisów i stosunkowo wolny zapis. Pamięci FRAM cechują się możliwością wykonania co najmniej  $10^{11}$  zapisów w każdej komórce pamięci, a operacja zapisu jest szybka. Dla pamięci I2C czas zapisu jest praktycznie niezauważalny. Pamięć ta jest zaprojektowana w taki sposób, aby jej obsługa maksymalnie przypominała obsługę innych pamięci I2C. Różnice dotyczą tylko operacji zapisu, co jest związane z tym, że w przeciwieństwie do pamięci EEPROM, zapis odbywa się praktycznie natychmiastowo. Także do obsługi pamięci FRAM I2C możemy zastosować te same funkcje do obsługi zwykłej pamięci EEPROM, z tym że oczekiwanie po operacji zapisu nie jest potrzebne. Tym, co różni operację zapisu do pamięci FRAM i EEPROM, jest także moment wykonania fizycznego zapisu. W pamięci EEPROM dane były buforowane przed zapisem, a niepoprawne zakończenie transakcji na magistrali I2C unieważniało zapis. W przypadku pamięci FRAM zapis odbywa się natychmiast po przesłaniu bajtu, jeszcze zanim pamięć wyśle potwierdzenie ACK — nie istnieje żaden bufor zapisu. Stąd też jeśli programista chce unieważnić operację zapisu, musi tego dokonać, zanim prześle pełny bajt danych poprzez wysłanie sygnału *STOP* lub *START*. Ponieważ zapis nie odbywa się stronami, możemy uprościć funkcję zapisu blokowego:

```
void I2C_MEM_FRAM_write_block(const void *src, MEM_addr *dst, size_t len)
{
    I2C_MEM_SetRWAddress(dst);
    do
    {
        I2C_SendByte(*(uint8_t*)src);
        src=*((uint8_t*)src)+1;
    } while(len--);
    I2C_Stop();
    I2C_WaitTillStopWasSent(); //Zaczekaj do końca operacji wysyłania danych
}
```

## Umieszczanie zmiennych w zewnętrznej pamięci EEPROM

Za pomocą powyższych funkcji możemy prosto odwoływać się do poszczególnych lokacji pamięci EEPROM/FRAM:

```
I2C_Init();
MEM_addr addr;

addr.DevAddr=0xAC;
addr.MEMAddr=0;

I2C_MEM_write_byte(&addr,0xab);
```

```

addr.MEMAddr++;
I2C_MEM_write_byte(&addr, 0xbb);
addr.MEMAddr++;
I2C_MEM_write_byte(&addr, 0xcc);

addr.MEMAddr=0;
dana=I2C_MEM_read_byte(&addr);
addr.MEMAddr++;
dana=I2C_MEM_read_byte(&addr);
addr.MEMAddr++;
dana=I2C_MEM_read_byte(&addr);

```

Powyższy program zapisuje do komórek pamięci EEPROM, począwszy od adresu 0, kolejne wartości 0xAB, 0xBB i 0xCC, a następnie je odczytuje. Podobnie możemy się odwoływać do kilku komórek jednocześnie, przy pomocy zdefiniowanych wcześniej operacji blokowych:

```

uint32_t dana=0xFFAABBCC;
I2C_MEM_write_block(&dana, &addr, sizeof(dana));
I2C_MEM_read_block(&dana, &addr, sizeof(dana));

```

W powyższym przykładzie do pamięci EEPROM zapisujemy wartość 32-bitowej zmiennej data, a następnie ponownie ją odczytujemy. W ten sposób możemy zapisać pod konkretnym adresem pamięci dowolne dane. Widzimy jednak, że za każdym razem musimy wiedzieć, pod jakim adresem znajdują się zapisywane/odczytywane dane. W przypadku paru zmiennych nie stanowi to problemu, lecz pamięć EEPROM może składać się z wielu bloków po 64 kB, w efekcie może się okazać, że musimy znać dokładną lokalizację olbrzymich struktur danych. Wprowadzenie jakiejkolwiek zmiany wiąże się w tym przypadku ze zmianą adresów wszystkich zmiennych. Łatwo jest więc dojść do stanu, w którym zupełnie stracimy kontrolę nad tym, co i gdzie się znajduje. Spróbujmy więc przerzucić tę niewdzięczną pracę na kompilator. W przypadku zmiennych umieszczonych w normalnej pamięci SRAM czy wewnętrznej pamięci EEPROM mikrokontrolera wystarczy taką zmienną zadeklarować, a kompilator sam zadba o jej właściwe umieszczenie w pamięci. Podobny efekt możemy uzyskać w przypadku zmiennych, które chcemy umieścić w zewnętrznej pamięci EEPROM. Niestety, kompilator nie wspiera wprost takich operacji<sup>2</sup>, stąd też musimy wykorzystać parę sztuczek. Wszystkie zmienne, które mają zostać umieszczone w zewnętrznej pamięci EEPROM, musimy pogrupować razem, w jedną strukturę danych. Do tego celu najlepiej nadają się struktury. Zdefiniujmy strukturę I2CEEPROM:

```

typedef struct
{
} I2CEEPROM;

```

Od tej pory, jeśli będziemy chcieli utworzyć zmienną, która ostatecznie znajdzie się w zewnętrznej pamięci EEPROM, po prostu umieścimy ją w deklaracji powyższej struktury:

---

<sup>2</sup> Aczkolwiek być może w niedalekiej przyszłości się to zmieni, gdyż począwszy od kompilatora gcc w wersji 4.5.x, stopniowo są wprowadzane tzw. nazwane przestrzenie adresowe (ang. *named address spaces*).

```
typedef struct
{
    uint16_t zmienna1;
    uint8_t zmienna2;
    char zmienna3[10];
} I2CEEPROM;
```

Zmienna zmienna1, zmienna2 i zmienna3 znajdą się pamięci EEPROM. W celu uzyskania do nich dostępu musimy w jakiś sposób pobrać adres naszych zmiennych, które stanowią pola struktury o nazwie I2CEEPROM. Ponieważ nie da się pobrać adresu pola na podstawie deklaracji struktury, musimy stworzyć wskaźnik, którego wartość będzie wynosiła 0 — od tego adresu zaczynają się kolejne komórki pamięci:

```
I2CEEPROM * const pEE=0x0000;
```

Od tej chwili dostęp do poszczególnych zmiennych (pół struktury I2CEEPROM) możemy uzyskać następująco:

```
uint16_t dana=10;
addr .MEMAddr=&pEE->zmienna1;
I2C_MEM_write_block(&dana, &addr, sizeof(pEE->zmienna1));
```

Najpierw uzyskujemy adres interesującej nas zmiennej (`&pEE->zmienna1`), po czym przekazujemy go do wcześniej zdefiniowanej funkcji realizującej zapis blokowy. Wielkość danego pola struktury I2CEEPROM (w tym przypadku zmiennej zmienna1) uzyskujemy za pomocą znanego nam operatora `sizeof`. Jak widzimy, udało nam się przerzucić na kompilator część pracy. Od tej chwili nie musimy się martwić, pod jakim adresem wewnętrznej pamięci EEPROM znajdzie się interesująca nas zmienna — zawsze będziemy znali jej prawidłowy adres. Jednak powyższy zapis nie wygląda zbyt „estetycznie” — jest długi i ciągle stosunkowo podatny na błędy. Poza estetyką powyższe rozwiązanie stwarza znacznie poważniejsze problemy. Tworzymy wskaźnik o nazwie `pEE`, który ma adres 0, a sam zapis sugeruje, że jest to wskaźnik na jakąś realną strukturę danych. Powoduje to, że np. operacja:

```
pEE->zmienna2=10;
```

jest z punktu widzenia semantyki języka C zupełnie poprawna, aczkolwiek w tym przypadku błędna. `pEE` jest tylko pomocniczym wskaźnikiem, lecz nie wskazuje on na żadną istniejącą strukturę danych. Łatwo jest więc popełnić błędy skutkujące nieprawidłowym działaniem programu.

Tu z pomocą przychodzi nam zdefiniowana w pliku nagłówkowym `<stddef.h>` makro-definicja `offsetof`. Przyjmuje ona jako parametry nazwę struktury danych (w naszym przypadku struktury I2CEEPROM) oraz nazwę pola należącego do tej struktury, a zwraca adres podanego pola w obrębie podanej struktury danych. Robi więc dokładnie to samo, co zadeklarowanie pomocniczego wskaźnika `pEE` w celu pobrania adresu pola. Dzięki temu powyższy przykład možemy uprościć i poprawić:

```
uint16_t dana=10;
addr .MEMAddr=offsetof(I2CEEPROM, zmienna1);
I2C_MEM_write_block(&dana, &addr, 2);
```

W powyższym przykładzie nie deklarowaliśmy już wskaźnika pEE, gdyż dzięki makrodefiniacji offsetof możemy pobrać adres pola struktury. Niestety, pojawia się kolejny problem: operator sizeof nie potrafi pobierać długości pola struktury. Zdefiniujmy więc „ulepszoną” wersję operatora sizeof:

```
#define SIZEOF(s,m) ((size_t) sizeof(((s *)0)->m))
```

Tak zdefiniowany operator SIZEOF, analogicznie do makrodefiniacji offsetof, przyjmuje dwa argumenty: nazwę struktury oraz nazwę pola struktury, której wielkość chcemy otrzymać. Tworzy tymczasowy obiekt o typie s, dzięki czemu klasyczny operator off →setof potrafi pobrać adres pola m. Wszystko odbywa się na etapie komplikacji programu, w efekcie nie jest generowany żaden kod, a w miejscu użycia takiej makrodefiniacji wstawiana jest po prostu wielkość pola o nazwie przekazanej jako parametr m. Stąd też powyższy przykład możemy uprościć:

```
I2C_MEM_write_block(&dana, &addr, SIZEOF(I2CEEPROM, zmienna));
```

Jesteśmy już blisko celu — pozbyliśmy się mylących zmiennych wskaźnikowych, kompilator przejął na siebie obowiązek prawidłowego rozmieszczenia zmiennych w pamięci, ale ciągle możemy jeszcze coś poprawić. Zobaczmy dwie kolejne makrodefinicje:

```
#define I2C_MEM_write_var(var, EVarname) \
{ \
    MEM_addr _addr: \
    _addr.DevAddr=0xAC: \
    _addr.MEMAddr=offsetof(I2CEEPROM, EVarname); \
    I2C_MEM_write_block(&var, &_addr, SIZEOF(I2CEEPROM, EVarname)); \
}

#define I2C_MEM_read_var(var, EVarname) \
{ \
    MEM_addr _addr: \
    _addr.DevAddr=0xAC: \
    _addr.MEMAddr=offsetof(I2CEEPROM, EVarname); \
    I2C_MEM_read_block(&var, &_addr, SIZEOF(I2CEEPROM, EVarname)); \
}
```

Powyższe procedury zakładają, że pamięć EEPROM, do której odbywa się zapis/odczyt ma adres 0xAC. Dzięki umieszczeniu całego kodu w oddzielnym bloku zdefiniowana zmienna \_addr jest zmienną tymczasową, co zapobiega błędem związanym z wielokrotną definicją zmiennej w przypadku wielokrotnych wywołań powyższych makrodefinicji. Jeśli użyta pamięć ma inny adres, należy tę stałą zmienić. Użycie powyższych makrodefinicji jest niezwykle proste:

```
uint16_t dana=10;
I2C_MEM_write_var(dana, zmienna);
I2C_MEM_read_var(dana, zmienna);
```

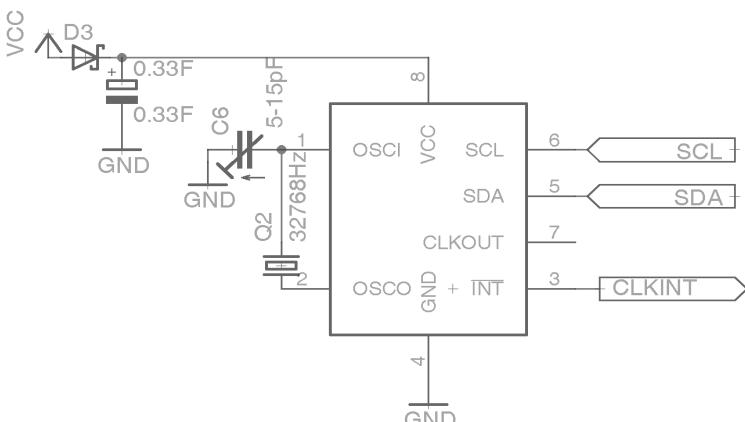
Powyższy przykład najpierw zapisuje do zmiennej zmienna1 znajdującej się w pamięci EEPROM wartość 10, a następnie ją odczytuje i ponownie przypisuje zmiennej dana.



Musimy pamiętać, aby zmienna dana i zmienna1 miały takie same typy, w przeciwnym przypadku powyższe funkcje nie będą działały poprawnie.

## Współpraca z zegarem RTC

Kolejnym bardzo często wykorzystywany układem I2C jest układ zegara RTC (ang. *Real-time lock*). Układy tego typu posiadają własny oscylator kwarcowy, zwykle taktywany zegarem 32 768 Hz (tzw. kwarc zegarkowy), i posiadają podtrzymywanie baterijne, dzięki czemu działają nawet przy zaniku napięcia zasilającego. Zwykle oprócz funkcji związanych z odmierzaniem czasu dysponują one dodatkowymi funkcjami, jak podtrzymywana baterijnie pamięć NV-RAM (ang. *Non-volatile random access memory*) czy układ generujący przerwania po zaprogramowanym czasie lub o zaprogramowanej godzinie. Jednym z popularniejszych układów tego typu jest układ PCF8563 firmy Philips. Jego połączenie do magistrali I2C/TWI jest niezwykle proste (rysunek 21.5).



**Rysunek 21.5.** Podłączenie układu PCF8563 do magistrali TWI. Linie oznaczone SCL i SDA należy połączyć z odpowiednimi liniami interfejsu TWI procesora (pamiętając o dodaniu rezystorów podciągających). Linia oznaczona jako CLKINT może zostać niepołączona, możemy ją też połączyć z pinem IO procesora, który ma zdolność generowania przerwania. Dzięki temu będziemy mogli wykorzystać wbudowaną w układ funkcję alarmu. Dioda D3 może być dowolną diodą o możliwie niskim napięciu przewodzenia (najlepiej diodą Schottky'ego), kondensator 0,33F to kondensator zapewniający podtrzymanie układu przy wyłączonym zasilaniu — jest to tzw. kondensator SuperCap

Komunikacja z układem odbywa się przy pomocy 16 rejestrów, które możemy odczytywać lub zapisywać (tabela 21.3).

Przy pomocy przedstawionych rejestrów możemy przeprowadzić konfigurację układu RTC. Po jego pierwszorzbowym włączeniu rejesty zawierające liczniki czasu i daty mają przypadkowe wartości.

Wskazówka

Pamiętajmy także, że wartości oznaczone symbolem x są nieokreślone, stąd odczytując dany rejestr, musimy je zamaskować tak, aby ich stan był znany.

O tym, że rejesty przechowujące czas i datę nie uległy uszkodzeniu podczas zaniku napięcia, świadczy wyzerowany bit VL rejestrów 02h (licznik sekund). Jego ustawienie informuje o konieczności ponownego ustawienia daty i czasu.

**Tabela 21.3.** Rejestry układu PCF8563. Wartości oznaczone znakiem x oznaczają dowolną wartość (0 lub 1), 0 w tabeli oznacza, że przy zapisie dana pozycja musi zawierać 0

Adres	Rejestr	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit1	Bit 0				
00h	kontrolny 1	TEST1	0	STOP	0	TESTC	0	0	0				
01h	kontrolny 2	0	0	0	TI_TP	AF	TF	AIE	TIE				
02h	sekund	VL					Sekundy 0 – 59 w kodzie BCD						
03h	minut	x					Minuty 0 – 59 w kodzie BCD						
04h	godzin	x			x			Godziny 0..23 w kodzie BCD					
05h	dni	x			x			Dni 1 – 31 w kodzie BCD					
06h	dnia tygodnia	x	x		x	x	x	Dzień tygodnia 0 – 6					
07h	miesiący	C		x		x		Miesiące 1 – 12 w kodzie BCD					
08h	lat	Lata 0 – 99 w kodzie BCD											
09h	alarm minuty	AE				Minuty 0 – 59 w kodzie BCD							
0Ah	alarm godziny	AE			x			Godziny 0 – 23 w kodzie BCD					
0Bh	alarm dni	AE			x			Dni 1 – 31 w kodzie BCD					
0Ch	alarm dzień tyg.	AE	x		x	x	x	Dzień tygodnia 0 – 6					
0Dh	CLKOUT	FE	x	x	x	x	x	FD1	FD0				
0Eh	kontrolny timera	TE	x	x	x	x	x	TD1	TD0				
0Fh	timer	Wartość licznika timera											

Aby wygodnie programować układ, zdefiniujmy na początku symboliczne aliasy poszczególnych rejestrów układu RTC, przy pomocy których będziemy się do nich odwoływać:

```

enum Registers {Cntr1Reg1=0, Cntr1Reg2=1, SecondsReg=2, MinutesReg=3, HoursReg=4,
    ↪DaysReg=5, WeekDaysReg=6, MonthsReg=7, YearsReg=8, MinuteAlarmReg=9,
    ↪HourAlarmReg=10, DayAlarmReg=11, WeekDayAlarmReg=12, CLKOUTUTFreqReg=13,
    ↪TimerCntr1Reg=14, TimerCountDownReg=15};

enum Alarm {AlarmDisable=128};

enum Flags {PCF_TimeValid=128, PCF_CLKOUTActivate=128, PCF_CLKOUT1Hz=3,
    ↪PCF_CLKOUT32Hz=2, PCF_CLKOUT1024Hz=1, PCF_CLKOUT32768Hz=0, PCF_EnableTimer=128,
    ↪PCF_TimerC1k4096Hz=0, PCF_TimerC1k64Hz=1, PCF_TimerC1k1Hz=2,
    ↪PCF_TimerC1k160sHz=3, PCF_AlarmInterruptEnabled=2, PCF_TimerInterruptEnabled=1};

enum Days {PCF_Sunday, PCF_Monday, PCF_Tuesday, PCF_Wednesday, PCF_Thursday,
    ↪PCF_Friday, PCF_Saturday};

enum Months {PCF_January=1, PCF_February, PCF_March, PCF_April, PCF_May, PCF_June,
    ↪PCF_July, PCF_August, PCF_September, PCF_October, PCF_November, PCF_December};

```

Dzięki powyższym definicjom nie musimy pamiętać, jakie numery mają poszczególne rejstry i zawarte w nich bity; wystarczy, że użyjemy odpowiedniej nazwy. Dodatkowo zdefiniujmy także symbol zawierający adres układu:

```
#define PCF8563_Addr 0xA2
```



Wskazówka Wszystkie transmisje z i do układu będziemy realizować za pomocą wcześniej zdefiniowanych funkcji komunikacji po magistrali I2C.

Ponieważ po włączeniu stan rejestrów określających datę i czas jest nieokreślony, musimy je zainicjalizować. Ale najpierw ten stan musimy wykryć. Zdefiniujmy więc funkcję sprawdzającą, czy układ RTC zawiera poprawny czas i datę, czy też musimy go najpierw zainicjalizować:

```
uint8_t PCF8563_IsDataValid()
{
    return ((PCF8563_ReadRegister(SecondsReg) & PCF_TimeValid)==0);
}
```

Powyższa funkcja zwraca 1, jeśli czas i data są prawidłowe, w przeciwnym przypadku zwraca 0. Korzysta ona z funkcji PCF8563\_ReadRegister odczytującej wartość podanego rejestru układu. Zdefiniujmy ją więc wraz z podobną funkcją umożliwiającą zapis do dowolnego rejestru układu:

```
void PCF8563_WriteRegister(uint8_t reg, uint8_t value)
{
    I2C_SendStartAndSelect(PCF8563_Addr | TW_WRITE);
    I2C_SendByte(reg);
    I2C_SendByte(value);
    I2C_Stop();
}

uint8_t PCF8563_ReadRegister(uint8_t reg)
{
    I2C_SendStartAndSelect(PCF8563_Addr | TW_WRITE);
    I2C_SendByte(reg);
    I2C_SendStartAndSelect(PCF8563_Addr | TW_READ);
    uint8_t res=I2C_ReceiveData_NACK();
    I2C_Stop();
    return res;
}
```

Dzięki powyższym funkcjom mamy dostęp do wszystkich rejestrów układu RTC, co umożliwia nam proste wykorzystanie funkcji, dla których nie zdefiniowaliśmy wprost metod dostępu, takich jak np. funkcje alarmu.

Pozostaje nam zdefiniować kilka podstawowych funkcji umożliwiających odczytanie i modyfikację czasu i daty:

```
void PCF8563_SetTime(Time *time)
{
    I2C_SendStartAndSelect(PCF8563_Addr | TW_WRITE);
    I2C_SendByte(SecondsReg);
    I2C_SendByte(time->Second);
```

```

I2C_SendByte(time->Minute);
I2C_SendByte(time->Hour);
I2C_Stop();
}

void PCF8563_GetTime(Time *time)
{
I2C_SendStartAndSelect(PCF8563_Addr | TW_WRITE);
I2C_SendByte(SecondsReg);
I2C_SendStartAndSelect(PCF8563_Addr | TW_READ);
time->Second=I2C_ReceiveData_ACK();
time->Minute=I2C_ReceiveData_ACK() & 0x7F;
time->Hour=I2C_ReceiveData_NACK() & 0x3F;
I2C_Stop();
}

void PCF8563_SetDate(Date *date)
{
I2C_SendStartAndSelect(PCF8563_Addr | TW_WRITE);
I2C_SendByte(DaysReg);
I2C_SendByte(date->Day);
I2C_SendByte(date->WeekDay);
I2C_SendByte(date->Month);
I2C_SendByte(date->Year);
I2C_Stop();
}

void PCF8563_GetDate(Date *date)
{
I2C_SendStartAndSelect(PCF8563_Addr | TW_WRITE);
I2C_SendByte(DaysReg);
I2C_SendStartAndSelect(PCF8563_Addr | TW_READ);
date->Day=I2C_ReceiveData_ACK() & 0x3F;
date->WeekDay=I2C_ReceiveData_ACK() & 0x07;
date->Month=I2C_ReceiveData_ACK() & 0x1F;
date->Year=I2C_ReceiveData_NACK();
I2C_Stop();
}

```

Powyższe funkcje maskują (zerują) nieużywane bity rejestrów czasu i daty. Bez tego odczytywane wartości byłyby nieprawidłowe. Zarówno czas, jak i data dla wygody przechowywane są w specjalnych strukturach danych:

```

typedef struct
{
    uint8_t Day;
    uint8_t WeekDay;
    uint8_t Month;
    uint8_t Year;
} Date;

typedef struct
{
    uint8_t Second;
    uint8_t Minute;
    uint8_t Hour;
} Time;

```



Pamiętać należy, że rok liczony jest, począwszy od roku 2000.

Odczytywane i zapisywane wartości są w kodzie BCD, to znaczy, że każdy bajt podzielony jest na dwie tetrady (czwórki bitów) zawierające daną liczbę dziesiętną, np. liczba dziesiętna 23 zapisana jest jako:

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

Zapis taki jest wygodny, jeśli chcemy wyświetlać czas np. na wyświetlaczu LED, ale niezbyt się nadaje do przeprowadzania obliczeń. Dodajmy więc parę funkcji konwertujących z formatu BCD na format binarny i odwrotnie:

```
static inline uint8_t bcd2bin(uint8_t n) { return (((n >> 4) & 0x0F) * 10) +  
    (n & 0x0F); };  
static inline uint8_t bin2bcd(uint8_t n) { return ((n / 10) << 4) | (n % 10)); };
```

Przy okazji możemy także zdefiniować funkcję konwertującą z formatu BCD na format ASCII, nadający się do wyświetlania na wyświetlaczu LCD:

```
static inline void bcd2ASCII(uint8_t bcd, char *ascii) {ascii[0]='0'+(bcd>>4);  
ascii[1]='0'+(bcd & 0x0F);};
```

Spróbujmy więc wykorzystać nowo zdefiniowane funkcje:

```
data.Year=bin2bcd(10);  
data.Month=bin2bcd(1);  
data.Day=bin2bcd(19);  
  
czas.Second=bin2bcd(0);  
czas.Minute=bin2bcd(0);  
czas.Hour=bin2bcd(23);  
  
if(!PCF8563_IsDataValid())  
{  
    PCF8563_SetTime(&czas);  
    PCF8563_SetDate(&data);  
}
```

Powyższy kod nadaje wartości początkowe strukturom data (19/01/2010) i czas (23:00:00), następnie sprawdza, czy czas zawarty w rejestrach układu RTC jest prawidłowy. Jeśli nie, to zmienia go na czas, jaki nadaliśmy strukturom data i czas. W podobny sposób możemy odczytać aktualne czas i datę:

```
PCF8563_GetTime(&czas);  
PCF8563_GetDate(&data);
```

Po wykonaniu powyższego kodu zmienne czas i data będą zawierać aktualny czas i aktualną datę odmierzane przez układ RTC.

## Obsługa ekspandera IO PCF8574

Układ PCF8574 jest popularnym ekspanderem magistrali I2C. Posiada on 8 pinów, które można niezależnie programować jako wejścia lub wyjścia. Domyślnie połączeniu zasilania wszystkie wyjścia są w stanie wysokim, ze słabym podciąganiem do Vcc. Umożliwia to wymuszenie na nich dowolnego stanu przez układ zewnętrzny. Wpisując na dane pozycje wartość 0, na odpowiednich wyjściach wymusza się niski stan logiczny. W przypadku wpisania wartości 1 na danej pozycji przejściowo wymuszane jest silne podciąganie do 1 aż do opadającego zbocza sygnału SCL. Dzięki takiemu działaniu poszczególne piny układu działają jako piny pseudodwukierunkowe.

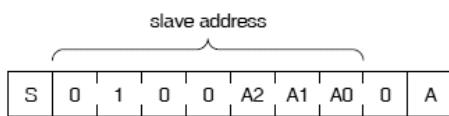


Wskazówka

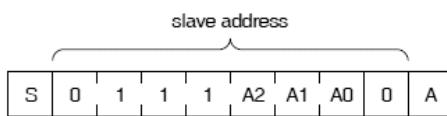
Należy pamiętać, aby nie wymuszać niskiego poziomu logicznego na danym pinie w sytuacji, w której ma on pracować jako pin wejściowy.

Układ ten posiada także sygnał INT, który jest aktywowany narastającym lub opadającym zboczem sygnału doprowadzonego do dowolnego pinu skonfigurowanego jako wejście. Sygnał ten jest wyprowadzony jako *open-drain*, co umożliwia łączenie go z analogicznymi sygnałami z innych ekspanderów w celu zrobienia iloczynu *wired-and*. Stan ten można wykorzystać do poinformowania procesora o zmianie, jaka zaszła na którymś z pinów wejściowych. Jest dezaktywowany w sytuacji, kiedy stan pinu powrócił do stanu wyjściowego, lub jeśli procesor dokonał odczytu układu. Bufory wyjściowe ekspandera nie są symetryczne, to znaczy, że w stanie niskim mają wydajność prądową ponad 25 mA, ale w stanie wysokim maksymalnie 300  $\mu$ A. **Stąd też jeśli układ ten ma sterować urządzeniem (np. diodą LED), to lepiej zapewnić sterowanie niskim poziomem logicznym.** Jednak całkowite obciążenie układu nie może przekroczyć 100 mA.

Dzięki trzem wejściom adresowym ( $A_0 - A_2$ ) układ może posiadać 8 różnych adresów na szynie I2C (rysunek 21.6), co umożliwia przyłączenie do 8 ekspanderów na jednej szynie. Dodatkowo istnieją wersje podstawowe i wersje z literą A, różniące się adresem bazowym, co umożliwia przyłączenie kolejnych 8 urządzeń.



a. PCF8574.



b. PCF8574A.

MBD973

**Rysunek 21.6.** Adresacja ekspandera PCF8574 i PCF8574A. Linie adresowe  $A_0 - A_2$  wybierają jeden z 8 możliwych adresów na magistrali I2C

Adres bazowy dla układu PCF8574 wynosi 0x40, a dla PCF8574A to 0x70.

Dostęp do układu wymaga klasycznej sekwencji instrukcji — wysłania sygnału *START*, adresu układu, bajtu danych oraz sygnału *STOP*. Kolejne bajty można wysyłać także jeden po drugim:

```
int main()
{
    uint8_t x=0xaa;
    I2C_Init();

    I2C_StartSelectWait(PCF8574ADDR);
    while(1)
    {
        I2C_SendByte(x);
        x^=0xff;
        _delay_ms(1000);
    }
    I2C_Stop();
}
```

Powyższy program spowoduje wygenerowanie na każdym pinie sygnału prostokątnego o częstotliwości 0,5 Hz. Jak widać, kolejne zapisy do ekspandera powodują natychmiastową zmianę stanu wyjścia, co umożliwia generowanie złożonych przebiegów o stosunkowo wysokiej częstotliwości (limitowanej częstotliwością magistrali I2C). Podobnie w ramach jednej transakcji można wielokrotnie odczytywać stan wejść.

## Procesor w trybie I2C slave

W trybie *slave* procesor może pracować jako odbiornik lub jako nadajnik. Jednak cała transmisja znajduje się pod kontrolą urządzenia *master*, które jest odpowiedzialne za generowanie sygnału zegarowego i sterowanie przebiegiem transmisji. W tym trybie urządzenie *slave* może wstrzymywać transmisję poprzez utrzymywanie w stanie niskim linii *SCL*, co informuje urządzenie *master* o tym, że *slave* nie nadąże z transmisją.

### Adres urządzenia

Analogicznie jak w przypadku innych urządzeń *slave* I2C także procesor w tym trybie musi mieć przydzielony unikalny adres, dzięki któremu będzie mógł być wybrany przez urządzenie *master*. Za konfigurację adresu odpowiadają dwa rejesty: TWAR [ang. *TWI (Slave) Address Register*] oraz TWAMR [ang. *TWI (Slave) Address Mask Register*]. Rejestr TWAR przechowuje 7-bitowy adres urządzenia *slave*, w trybie *master* stan tego rejestru jest bez znaczenia. Adres przechowywany jest w bitach 1 – 7. Specjalne znaczenie ma najmłodszy bit tego rejestru, TWGCE (ang. *TWI General Call Recognition Enable Bit*). Jeśli jest on ustawiony, to urządzenie będzie odpowiadało na tzw. ogólny adres wywołania I2C. W takiej sytuacji nadanie jako adresu urządzenia wartości 0 spowoduje odpowiedź urządzenia *slave* niezależnie od stanu pozostałych bitów rejestru TWAR. Jeśli bit ten ma wartość 0, urządzenie nie będzie odpowiadać na wywołania ogólne. Drugim rejestrem związanym z adresacją urządzenia jest rejestr TWAMR. W jego przypadku znaczenie mają tylko bity 1 – 7, najmłodszy bit jest ignorowany i przy odczytce zawsze zawiera wartość 0. Ustawienie któregokolwiek bitu tego rejestru powoduje, że przy porównaniu adresu wysłanego przez urządzenie *master* z adresem znajdującym się w rejestrze TWAR wartość takiego bitu jest ignorowana. Rejestr ten umożliwia więc maskowanie wybranych bitów adresowych, w efekcie urządzenie odpowiada na kilka

różnych adresów. W szczególności wpisanie do tego rejestru wartości 0xFE spowoduje, że urządzenie będzie odpowiadało na każdy adres urządzenia I2C. Jeśli wpisana zostanie wartość 0, to urządzenie odpowie wyłącznie na adres określony wartością rejestru TWAR.

## Zegar taktujący

W trybie *slave* stan rejestru TWBR oraz preskalera określającego prędkość transmisji nie ma znaczenia. Prędkość ta jest określona przez sygnał *SCK* pochodzący z magistrali TWI. Jedyne, co należy zapewnić, to taktowanie całego modułu częstotliwością co najmniej 16 razy większą niż częstotliwość magistrali TWI. Jest to wymagane dla prawidłowego próbkowania stanu tej magistrali. Interfejs TWI działa także prawidłowo w trybach uśpienia procesora, a jego przerwanie może procesor z tego stanu wybudzić. Jednak w przypadku uśpienia i wybudzenia procesora sygnałem pochodzącym z magistrali TWI zawartość rejestru TWDR jest nieokreślona, w efekcie niemożliwe jest ustalenie adresu urządzenia, jaki wysłał *master*. Jeśli danemu urządzeniu *slave* przyporządkowany jest tylko jeden adres, nie ma to znaczenia. Natomiast w sytuacji, w której urządzenie korzysta z maskowania i przez to posiada przydzielony więcej niż jeden adres, ustalenie adresu wysłanego przez *master*, o ile ma to jakieś znaczenie, musi odbywać się w inny sposób.

## Rejestr stanu

Programując urządzenie *master*, w pewnych okolicznościach można liczyć na szczęście i ignorować informacje o stanie magistrali I2C, ewentualnie wykorzystywać je w ograniczonym zakresie. Przy programowaniu urządzenia, które będzie odgrywało rolę *slave* na magistrali, bity określające stan tej magistrali mają fundamentalne znaczenie i bez nich nie da się prawidłowo oprogramować transmisji. W zależności od tego, czy urządzenie *slave* działa w trybie nadawania czy odbierania danych, generowane są inne komunikaty określające stan magistrali (tabele 21.4 i 21.5). Stan magistrali w każdej chwili można odczytać z rejestru TWSR. Ponieważ rejestr ten zawiera także nieużywane bity oraz bity określające wartość preskalera, przed wykorzystaniem zawartej w nim informacji o stanie magistrali stan pozostałych bitów należy zamaskować. Biblioteka AVR-libc definiuje makro `TW_STATUS` zwracające wartość bitów stanu rejestru TWSR.

**Tabela 21.4.** Bitы stanu związane z nadawaniem w trybie slave

Nazwa	Wartość	Znaczenie
<code>TW_ST_SLA_ACK</code>	0xA8	Start transmisji, żądanie odczytu danych ze <i>slave</i> , wysłano <i>ACK</i>
<code>TW_ST_ARB_LOST_SLA_ACK</code>	0xB0	Błąd arbitrażu magistrali
<code>TW_ST_DATA_ACK</code>	0xB8	Wysłano bajt danych i otrzymano <i>ACK</i>
<code>TW_ST_DATA_NACK</code>	0xC0	Wysłano bajt danych, otrzymano <i>NACK</i> (koniec transmisji)
<code>TW_ST_LAST_DATA</code>	0xC8	Wysłano ostatni bajt danych, otrzymano <i>ACK</i>

Na podstawie bitów stanu można ustalić, na jakim etapie znajduje się transmisja, i podjąć odpowiednie działania.

Oprócz powyższych stanów w bibliotece AVR-libc zdefiniowane są dwa kolejne, wspólne dla wszystkich typów transmisji I2C (*master/slave*, nadawanie/odbiór). `TW_NO_INFO`

**Tabela 21.5.** Bitы stanu związane z odbiorem w trybie slave

Nazwa	Wartość	Znaczenie
TW_SR_SLA_ACK	0x60	Otrzymano żądanie zapisu, wysłano ACK
TW_SR_ARB_LOST_SLA_ACK	0x68	Utrata arbitrażu magistrali
TW_SR_GCALL_ACK	0x70	Odebrano żądanie na podstawie adresu rozgłoszeniowego
TW_SR_ARB_LOST_GCALL_ACK	0x78	Utrata arbitrażu, przy żądaniu rozgłoszeniowym
TW_SR_DATA_ACK	0x80	Odebrano bajt danych, wysłano ACK
TW_SR_DATA_NACK	0x88	Odebrano bajt danych, wysłano NACK (koniec transmisji)
TW_SR_GCALL_DATA_ACK	0x90	Jak TW_SR_DATA_ACK dla adresu rozgłoszeniowego
TW_SR_GCALL_DATA_NACK	0x98	Jak TW_SR_DATA_NACK dla adresu rozgłoszeniowego
TW_SR_STOP	0xA0	Otrzymano sygnał STOP lub repeated START.

(0xF8) jest początkowym stanem rejestru stanu kontrolera. Wartość ta znajduje się w rejestrze także po każdym zakończeniu transakcji i sygnalizuje brak informacji o stanie magistrali I2C. Drugą stałą jest TW\_BUS\_ERROR (0x00), sygnalizująca poważny błąd magistrali. Błąd ten może być nieodwracalny (uniemożliwia wtedy jakikolwiek transmisię I2C), może też wynikać z nieprawidłowego sterowania przebiegiem transmisji. W takiej sytuacji najlepiej zresetować cały interfejs I2C.



Nieprawidłowa obsługa lub brak obsługi powyższych błędów jest częstą przyczyną „zawieszania się” interfejsu I2C.

Owo „zawieszanie się” jest wynikiem nieprawidłowo napisanego programu, który w efekcie czeka na zdarzenie, które nie może zajść, sprawiając pozory problemu sprzętowego.

## Piny IO

Analogicznie jak w trybie *master*, włączenie trybu *slave* powoduje przejęcie kontroli nad pinami *IO* wykorzystywany do celów transmisji TWI. Jedyna możliwość konfiguracji to włączenie wewnętrznych rejestrów podciągających, poprzez wpisanie 1 na odpowiednich bitach rejestru PORT<sub>x</sub>, odpowiadających wyjściom *SCL* i *SDA*. W niektórych sytuacjach eliminuje to konieczność zastosowania zewnętrznych rezystorów podciągających.

## Odbiór danych

Każdy odbiór lub nadanie paczki danych (8 bitów) sygnalizowane jest wygenerowaniem przerwania (o ile zezwala na to bit TWEN rejestru TWCR) oraz poprzez ustawienie bitu TWINT. Po odebraniu/nadaniu danych stan magistrali jest „zamrożony” poprzez utrzymanie przez urządzenie *slave* linii *SCK* w stanie niskim. Dzięki temu urządzenie *master* wstrzymuje się z kolejną transmisją do czasu, aż urządzenie *slave* będzie gotowe. W tym celu należy wyzerować bit TWINT — spowoduje to zwolnienie linii *SDA* i *SCL*, umożliwiając kolejną transmisję.



Wskazówka

Wyzerowanie bitu TWINT nie następuje automatycznie. Należy go wyzerować programowo poprzez wpisanie wartości 1 na pozycję bitu TWINT rejestru TWCR.

Dzięki temu, że zwolnienie magistrali wymaga ingerencji programu, mikrokontroler ma czas na odpowiednią reakcję na zdarzenie i np. przygotowanie kolejnych danych do wysłania. Stwarza to też pewne niebezpieczeństwo — brak wyzerowania bitu TWINT powoduje permanentne utrzymywanie stanu niskiego na linii *SCL*, a w konsekwencji uniemożliwia jakąkolwiek transmisję TWI/I2C. **W ten sposób jedno wadliwie działające urządzenie może zablokować całą magistralę.**

## Przykład

Z powyższego opisu wydawać by się mogło, że zbudowanie urządzenia *slave* I2C jest trudne. Z pewnością ze względu na dużą możliwą liczbę stanów magistrali jego oprogramowanie jest bardziej skomplikowane niż oprogramowanie innych interfejsów (SPI lub UART), lecz dzięki wsparciu sprzętowemu ze strony mikrokontrolera nie jest takie trudne. Do celów edukacyjnych stworzony został układ pokazany na rysunku 21.7. Składa on się z dwóch mikrokontrolerów, z których *master* (ATMega128) wymienia dane poprzez magistralę I2C z urządzeniem *slave* (ATMega88). Rolę urządzenia *master* sprowadzono do minimum. Wysyła ono polecenia do urządzenia *slave*, odczytuje z niego dane, przetwarza je i odsyła z powrotem. Urządzenie *slave* zostało zbudowane w oparciu o znany nam z rozdziału 18. układ wykorzystujący wyświetlacz graficzny. Dodany do niego został pomiar temperatury (pokazany w rozdziale 15.) oparty na czujniku analogowym LM35 oraz interfejs I2C. Dzięki temu *slave* odbiera polecenia docierające do niego poprzez ten interfejs, interpretuje je i wyświetla ich efekty na wyświetlaczu graficznym. Dodatkowo poprzez interfejs I2C istnieje możliwość odczytu temperatury zmierzonej przy pomocy układu LM35. Urządzenie *slave* realizuje polecenia pokazane w tabeli 21.6.

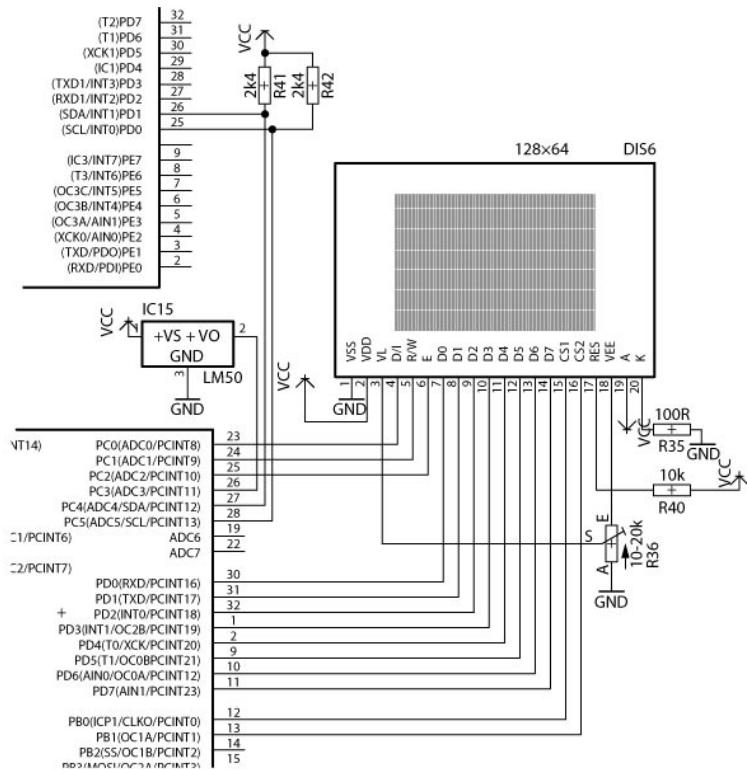
**Tabela 21.6.** Polecenia realizowane przez urządzenie *slave* przy zapisie. Wszystkie parametry są 8-bitowymi wartościami

Polecenie	Parametry	Opis
c	Brak	Czyści ekran LCD
l	<i>x1, y1, x2, y2</i>	Rysuje linię od punktów ( <i>x1, y1</i> ) do ( <i>x2, y2</i> )
g	<i>x, y</i>	Ustawia kursor w punkcie ( <i>x, y</i> )
t	NULLZ	Wyświetla tekst ASCII zakończony znakiem NULL

Odczytując urządzenie, uzyskuje się 16-bitową wartość określającą zmierzoną temperaturę z rozdzielcością do 1/100 stopnia Celsjusza.

Pisanie programu rozpoczniemy od urządzenia *master*. Jego funkcje będą proste: będzie on cyklicznie odczytywał zmierzoną przez *slave* temperaturę, a następnie po jej konwersji do łańcucha znakowego będzie wysyłał polecenie powodujące jej wyświetlenie na wyświetlaczu LCD obsługiwany przez urządzenie *slave*. W tym celu wykorzystane zostaną znane nam już funkcje obsługujące magistralę I2C. Dla wygody zdefiniowane zostały dwie dodatkowe funkcje. Funkcja:

**Rysunek 21.7.**  
 Schemat połączenia mikrokontrolera ATMega128 z układem slave zrealizowanym przy pomocy mikrokontrolera ATMega88. Układ slave steruje matrycą graficzną 128x64 punkty oraz odczytuje temperaturę z miernika analogowego LM35



```
void I2C_StartSelectWait(uint8_t addr)
{
    do
    {
        I2C_SendStartAndSelect(addr);
    } while(I2C_Error==I2C_NoACK);
}
```

wysyła znak startu, a następnie adres wybranego urządzenia *slave*. Jeśli wybór urządzenia zakończy się niepowodzeniem (odebrany zostanie sygnał *NACK* oznaczający brak urządzenia *slave* lub brak jego gotowości), nastąpi ponowienie próby.

Druga funkcja wysyła podany łańcuch tekstowy na magistralę I2C:

```
void I2C_SendTxt(char *txt)
{
    while(*txt) I2C_SendByte(*txt++);
    I2C_SendByte(0); //Wyślij znak końca łańcucha
}
```

Ostatnim wysyłanym bajtem zawsze jest 0, dzięki czemu otrzymany przez urządzenie *slave* ciąg bajtów nadaje się bezpośrednio do wyświetlenia na LCD.

Funkcja *main* urządzenia *master* jest stosunkowo prosta:

```

volatile uint16_t temper;

int main()
{
    char wynik[7];

    _delay_ms(1000);
    I2C_Init(); //Zainicjuj TWI z domyślną prędkością 100 kHz
    I2C_StartSelectWait(DEVADDR);
    I2C_SendByte('c');
    I2C_Stop();
    I2C_WaitTillStopWasSent();

    I2C_StartSelectWait(DEVADDR);
    I2C_SendByte('g'); I2C_SendByte(0); I2C_SendByte(0);
    I2C_Stop();
    I2C_WaitTillStopWasSent();

    I2C_StartSelectWait(DEVADDR);
    I2C_SendByte('t'); I2C_SendTxt("Temperatura po I2C");
    I2C_Stop();
    I2C_WaitTillStopWasSent();
    while(1)
    {
        I2C_StartSelectWait(DEVADDR | TW_READ);
        temper=I2C_ReceiveData_ACK();
        temper+=256*I2C_ReceiveData_NACK();
        I2C_Stop();

        I2C_StartSelectWait(DEVADDR);
        I2C_SendByte('g'); I2C_SendByte(0); I2C_SendByte(1);
        I2C_Stop();
        I2C_WaitTillStopWasSent();

        sprintf(wynik, "%5d", temper);
        uint8_t len=strlen(wynik);
        memmove(&wynik[len-1], &wynik[len-2], 3);
        wynik[len-2]='.';

        I2C_StartSelectWait(DEVADDR);
        I2C_SendByte('t'); I2C_SendTxt(wynik);
        I2C_Stop();
        I2C_WaitTillStopWasSent();

        _delay_ms(100);
    }
}

```

Najpierw czyszczony jest wyświetlacz LCD, wysyłany jest do niego napis „Temperatura po I2C”, a w kolejnej linii LCD wyświetlana jest aktualnie zmierzona temperatura.

Jak widać, każda transakcja I2C składa się z podobnych elementów — wybór urządzenia *slave* (funkcja `I2C_StartSelectWait`). Dokonując wyboru urządzenia, należy określić, czy będziemy do niego zapisywali dane (stała `TW_WRITE`) czy odczytywali (stała `TW_READ`). Dla przypomnienia, typ transakcji określa najmłodszy bit adresu urządzenia. Po poprawnym wyborze urządzenia można do niego zapisywać dane przy pomocy

funkcji `I2C_SendByte` lub je odczytywać przy pomocy funkcji `I2C_ReceiveData_ACK()` lub `I2C_ReceiveData_NACK()`. Ostatnim krokiem jest wysłanie na magistralę sygnału *STOP* (funkcja `I2C_STOP`), informującego urządzenie *slave* o końcu transakcji. Wysłanie tego sygnału jest niezbędne przy operacji zapisu — bez niego urządzenie *slave* będzie oczekwać na kolejne dane i w efekcie nie wykona przesłanego polecenia. Zamiast sygnału *STOP* można wysłać sygnał *repeated START*, co umożliwia rozpoczęcie realizacji kolejnej transakcji, z tym samym, wcześniej wybranym urządzeniem I2C *slave*.

O wiele bardziej skomplikowane jest oprogramowanie urządzenia *slave*.



Wskaźówka Obsługa wyświetlacza graficznego została omówiona w rozdziale 18., a termometru analogowego w rozdziale 15., nie będą więc one ponownie omawiane.

Obsługę urządzenia *slave* najwygodniej zrealizować na przerwaniach. Dzięki temu układ *slave* może przez większość czasu pozostawać w stanie uśpienia, co oszczędza energię. Zanim jednak zostanie zrealizowana obsługa I2C, należy zadbać o inne szczegóły. Transmisja pomiędzy urządzeniem *master* a *slave* będzie odbywać się w postaci transakcji — po skończeniu transakcji urządzenie *slave* przejdzie do realizacji odebranego polecenia. Ponieważ polecenia są wielobajtowe, należy je zbuforować, tak aby były dostępne po zakończeniu transakcji. W tym celu zdefiniowano zmienną:

```
char Buf[BUFSIZE];
```

będącą buforem na odebrane znaki. Jego rozmiar można określić dyrektywą `#define BUFSIZE`, jednak nie może on przekroczyć 31 bajtów. Aby móc łatwo określić, czy w buforze znajduje się polecenie i w jakim stanie jest transakcja na I2C, wprowadzono strukturę definiującą flagi:

```
struct Buf_status
{
    union
    {
        struct
        {
            uint8_t st_ready      : 1;
            uint8_t st_receiving   : 1;
            uint8_t st_transmitting : 1;
            uint8_t counter        : 5;
        };
        uint8_t byte;
    };
    volatile struct Buf_status BUF_status;
};
```

Flaga `st_ready` ma wartość 1, jeśli w buforze znajduje się gotowe do wykonania polecenie. `st_receiving` ma wartość 1, jeśli aktualnie odbywa się proces odbioru danych. Analogicznie `st_transmitting` ma wartość 1, jeśli aktualnie nadawane są dane. Z kolei zmienna `counter` zawiera numer nadawanego bajtu (w przypadku operacji odczytu) lub numer odbieranego bajtu (w trakcie operacji zapisu). Aby cała struktura zmieściła się w 1 bajcie, zmienna `counter` ma tylko 5 bitów, dzięki czemu bufor maksymalnie może składać się z 32 bajtów.

Pomocniczo zdefiniowano unię:

```
typedef union
{
    uint16_t word;
    uint8_t byte[2];
} WORD;
```

Umożliwia ona dostęp do pojedynczych bajtów 2-bajtowych typów danych.

Po zinterpretowaniu odebranego polecenia program wywołuje funkcję:

```
void BUFEempty()
{
    BUF_status.byte=0;
    TWCR|=_BV(TWEA);
}
```

Jej zadaniem jest zasygnalizowanie, że polecenie zostało wykonane, a układ oczekuje na kolejną transakcję I2C.

Urządzenie w trybie *slave* inicjuje funkcja:

```
void I2C_Slave_Init()
{
    TWAR= DEVADDR;
    TWAMR=0;
    TWCR=_BV(TWEN) | _BV(TWEA) | _BV(TWIE) | _BV(TWINT); //Odblokuj TWI, przerwanie TWI
                                                               //i automatyczne generowanie ACK
}
```

Funkcja ta nadaje adres urządzeniu *slave* (jest on przechowywany w symbolu o nazwie DEVADDR), odblokowuje przerwania TWI, interfejs, włącza też automatyczne generowanie potwierdzeń (*ACK*) po wybraniu przez urządzenie *master* adresu urządzenia *slave*.

Pozostaje napisanie procedury obsługi przerwania TWI:

```
ISR(TWI_vect)
{
    uint8_t status=TW_STATUS;
    switch(status)
    {
        //Obsługa slave receive
        case TW_SR_SLA_ACK: BUF_status.counter=0; BUF_status.st_receiving=1; break;
        //Rozpoczęto transmisję
        case TW_SR_STOP: if(BUF_status.st_receiving)
        {
            BUF_status.st_ready=1;
            BUF_status.st_receiving=0;
            TWCR&=(_BV(TWEA)); //Nie generuj ACK
        } else
        {
            TWCR|=_BV(TWEA);
            BUF_status.counter=0; BUF_status.st_receiving=1;
        }
        break; //Zakończono odbiór ramki
    case TW_SR_GCALL_DATA_ACK:
```

```

        case TW_SR_GCALL_ACK: break;
        case TW_SR_DATA_ACK: Buf[BUF_status.counter++]=TWDR; break; //Odebrano bajt danych
        case TW_SR_DATA_NACK: Buf[BUF_status.counter++]=TWDR;
                               BUF_status.st_ready=1;
                               BUF_status.st_receiving=0;
                               TWCR&=(_BV(TWEA)); //Nie generuj ACK
                               break;
        //Obsługa slave transmit
        case TW_ST_SLA_ACK:   BUF_status.counter=0; BUF_status.st_transmitting=1;
        case TW_ST_DATA_ACK:  TWDR=((WORD)GetTemperature()).byte[BUF_status.counter++];
        break;
        case TW_ST_DATA_NACK: BUF_status.st_transmitting=0; break;
        default:              TWCR=_BV(TWEN) | _BV(TWEA) | _BV(TWIE) | _BV(TWINT);
                               BUF_status.byte=0;
}
if(BUF_status.counter>=BUFSIZE) BUF_status.counter=0; //Zapobiega przepelnieniu bufora
TWCR|= _BV(TWINT); //Zwolnij linię SCL
}

```



Wskazówka

Funkcja ta zostaje wywołana zawsze po zajściu danego zdarzenia. Jej celem jest więc przygotowanie układu *slave*, tak aby był on gotowy na przyjęcie kolejnego zdarzenia. Interfejs TWI, aby dać czas programowi na to przygotowanie, utrzymuje w stanie niskim linię *SCL*. Stąd też należy ją zwolnić poprzez wyzerowanie bitu *TWINT* (wpisanie na jego pozycji wartości 1), co odbywa się na końcu procedury obsługi przerwania.

Cała procedura obsługi została podzielona na dwie części — pierwsza jest odpowiedzialna za obsługę zapisu danych do urządzenia *slave*, a druga za obsługę odczytu danych z urządzenia *slave*.

## Zapis danych

Rozpoczęcie transakcji zapisu sygnalizowane jest stanem *TW\_SR\_SLA\_ACK* lub *TW\_SR\_STOP* (w przypadku wysłania sygnału *repeated START*). Po odebraniu tego sygnału wskaźnik pozycji zapisu do bufora (counter) jest zerowany, ustawiana jest także flaga informująca o toczącym się procesie zapisu do bufora. Jest to niezbędne, aby odróżnić sygnał *STOP* od sygnału *repeated START*. Kiedy ta flaga jest ustawiona, sygnalizując toczącą się transmisję, kolejny stan *TW\_SR\_STOP* związany jest z wysłaniem sygnału *STOP*, a nie *repeated START*.

Obsługa kolejnych dwóch stanów: *TW\_SR\_DATA\_ACK* i *TW\_SR\_DATA\_NACK* jest podobna. Są one generowane po odebraniu bajtu danych. Przy czym otrzymanie sygnału *TW\_SR\_DATA\_NACK* jest jednocześnie znakiem końca transmisji. Odebranie polecenia sygnalizowane jest ustawieniem flagi *st\_ready*. Ponieważ realizacja odebranego polecenia może zająć trochę czasu, nie jest możliwe w tym czasie odbieranie kolejnych poleceń. Ich odbiór spowodowałby zamazanie poprzednich i problemy z jednokrotnym dostępem do danych znajdujących się w tablicy *Buf*. Dla urządzeń I2C *slave* w takich sytuacjach przyjętą praktyką jest ich czasowe odłączenie od magistrali. Ponowny wybór takiego urządzenia w czasie, kiedy jest ono zajęte realizacją odebranego wcześniej polecenia, jest niemożliwy, gdyż urządzenie to nie generuje w tym czasie potwierdzeń (*ACK*). Symulacja takiego zachowania jest prosta — wystarczy w tym celu wyzerować bit *TWEA* rejestru *TWCR*. Dopóki jest on wyzerowany, urządzenie nie będzie reagować na

jakiekolwiek dane płynące z magistrali I2C. Dzięki temu reszta programu ma czas na interpretację i wykonanie polecenia. Po jego wykonaniu bit ten jest ponownie ustawiany, umożliwiając realizację kolejnej transakcji.

## Odczyt danych

Odczyt danych wymaga obsłużenia mniejszej liczby sygnałów. Transakcja odczytu rozpoczyna się od odebrania stanu `TW_ST_SLA ACK`. Powoduje to zainicjowanie zmiennych związanych z odczytem temperatury i ustawieniem flagi `st_transmitting`. Jednocześnie do rejestru `TWDR` należy wpisać pierwszą odczytywaną wartość. Kolejne będą wpisywane po odebraniu sygnału `TW_ST_DATA_ACK`. Odebranie sygnału `TW_ST_DATA_NACK` kończy transakcję odczytu danych.

Dla bezpieczeństwa dodano także obsługę innych sygnałów, które nie powinny się co prawda pojawić, ale warto się zabezpieczyć przed nieoczekiwany variantami. W przypadku pojawienia się nieoczekiwanej sygnału interfejs `TWI` urządzenia *slave* jest resetowany, co umożliwia powrót do określonego stanu zarówno urządzeniu *slave*, jak i *master*.

Powyższe urządzenie przy odczycie zwraca wartość zmierzonej temperatury, analogicznie jednak można zwrócić np. stan podłączonej do niego klawiatury, co umożliwia stworzenie zdalnego terminala, z którym łączność odbywa się poprzez interfejs I2C.

Na koniec pozostaje jeszcze stworzyć funkcję `main`, scalającą powyższe funkcje:

```
int main()
{
    ADC_init();
    GLCD_init();
    GLCD_cls();
    I2C_Slave_Init();
    sei();
    while(1)
    {
        if(BUF_status.st_ready)
        {
            switch(Buf[0])
            {
                case 'c': GLCD_cls(); break;
                case 'l': GLCD_Line(Buf[1], Buf[2], Buf[3], Buf[4]); break;
                case 'g': GLCD_goto(Buf[1], Buf[2]); break;
                case 't': GLCD_puttext(&Buf[1]);
            }
            BUFEEmpty();
        }
    }
}
```

Funkcja ta inicjalizuje podsystemy odpowiedzialne za odczyt temperatury (`ADC_init`), graficzny (`GLCD_init`) i komunikacyjny (`I2C_Slave_Init`), a następnie w pętli oczekuje na polecenie. Po jego nadejściu (co sygnalizuje ustawienie flagi `st_ready`) przystępuje do jego realizacji. Zakończenie realizacji polecenia powoduje wywołanie funkcji `BUFEEmpty()`, dzięki czemu urządzenie sygnalizuje gotowość do odebrania i realizacji kolejnego polecenia.

# Rozdział 22.

# Interfejs USI

Mikrokontrolery serii ATTiny nie mają zaimplementowanych pełnych portów szeregowych (TWI, SPI, USART), zamiast tego występuje w nich tzw. uniwersalny interfejs szeregowy (ang. *Universal Serial Interface*). Zapewnia on wsparcie sprzętowe ułatwiające implementację interfejsów szeregowych. W efekcie, odpowiednio konfigurując interfejs USI oraz dodając niewielkie wsparcie ze strony programu, można zrealizować funkcjonalność interfejsów szeregowych. Wsparcie sprzętowe umożliwia osiągnięcie większych prędkości transferu i mniejszego obciążenia procesora niż czysto programowa realizacja interfejsu. Interfejs USI umożliwia wspomaganą sprzętowo realizację następujących protokołów i trybów:

- ◆ TWI w trybie *master* i *slave*,
- ◆ SPI w trybie *master* i *slave*,
- ◆ UART.

Wykorzystując interfejs USI, warto pamiętać, że jego rejestr danych USIDR (ang. *USI Data Register*) nie jest buforowany. W efekcie otrzymywane dane trzeba jak najszybciej odczytywać, w przeciwnym przypadku zostaną stracone. Z interfejsem USI związany jest także 4-bitowy licznik, który jest taktowany z tego samego źródła zegara co rejestr USIDR. W efekcie przesuwanie bitów w USIDR następuje synchronicznie do zmian stanu licznika.

## 4-bitowy licznik i zegar

Integralną częścią interfejsu USI jest 4-bitowy licznik. Jego bieżącą wartość można odczytywać z rejestru stanu — licznik zajmuje jego cztery najmłodsze bity. Istnieje także możliwość zapisu do licznika.

Licznik może być taktowany z różnych źródeł:

- ◆ z timera 0 w wyniku zajścia zdarzenia *Compare Match*,
- ◆ ze źródła zewnętrznego, doprowadzonego do wejścia USCK,
- ◆ oraz programowo.

Należy pamiętać, że jeżeli licznik taktowany jest ze źródła zewnętrznego (pinu USCK) lub programowo (bit USITC), zliczane jest każde zbocze sygnału zegarowego, a nie okres. W efekcie zliczanych jest dwukrotnie więcej impulsów.

Źródło sygnału zegarowego wybiera się przy pomocy bitów USICS1 – 0 (ang. *USI Clock Source Select*) rejestru USICR. Zależność pomiędzy tymi bitami a źródłem zegara pokazana została w tabeli 22.1.

**Tabela 22.1** Wybór źródła zegara taktującego licznik i rejestr układu USI

<b>USICS1</b>	<b>USICSO</b>	<b>USICLK</b>	<b>Taktowanie USIDR</b>	<b>Taktowanie licznika</b>
0	0	0	Wyłączone	Wyłączone
0	0	1	Taktowanie programowe (bit USITC)	Taktowanie programowe (bit USITC)
0	1	X	Zdarzenie <i>Compare Match</i> <i>timera 0</i>	Zdarzenie <i>Compare Match</i> <i>timera 0</i>
1	0	0	Pin USCK, zbocze narastające	Pin USCK, oba zbocza
1	1	0	Pin USCK, zbocze opadające	Pin USCK, oba zbocza
1	0	1	Pin USCK, zbocze narastające	Taktowanie programowe (bit USITC)
1	1	1	Pin USCK, zbocze opadające	Taktowanie programowe (bit USITC)

Jeśli zostanie wybrane taktowanie programowe, sygnał taktujący jest generowany po wpisaniu wartości 1 do bitu USITC (ang. *Toggle Clock Port Pin*) rejestru USICR. Każdy wpis powoduje zmianę sygnału zegarowego na przeciwny, w efekcie wygenerowanie pełnego okresu wymaga dwóch wpisów do rejestru. Generowany przebieg zegarowy może pojawić się na wyprowadzeniu USCK procesora, jeśli związany z nim bit związanego z nim rejestru DDR zostanie ustawiony.

## Przerwania USI

Dla polepszenia responsywności aplikacji interfejs USI może w pewnych sytuacjach generować przerwania informujące o zajściu zdarzeń. Dzięki temu, zamiast sprawdzać stan odpowiednich flag rejestru USISR metodą *poolingu*, reakcja na zdarzenie może być zapewniona przez odpowiednią funkcję obsługi przerwania. Interfejs USI generuje dwa typy przerwań:

- ◆ przerwania przepelnienia licznika USI o wektorze USI\_OVF\_vect (lub w niektórych procesorach o wektorze USI\_OVERFLOW\_vect);
- ◆ przerwanie w wyniku detekcji bitu startu w trybie TWI o wektorze USI\_START\_vect (w niektórych procesorach wektor ten nazywa się USI\_STRT\_vect lub USI\_STR\_vect).

Przerwania te można odblokować poprzez ustawienie flag USISIE (ang. *Start Condition Interrupt Enable*) dla przerwania wykrywającego bit startu lub flagi USIOIE (ang. *Counter Overflow Interrupt Enable*) dla odblokowania przerwania przepełnienia licznika znajdujących się w rejestrze kontrolnym USI (USICR).



Uwaga

W sytuacji kiedy odpowiednie flagi rejestru statusu są ustawione, włączenie bitów odpowiadających za zezwolenie na przerwanie powoduje natychmiastowe wywołanie odpowiedniej funkcji obsługi przerwania.

Stąd też przed ustawieniem flag zezwolenia dobrze jest skasować odpowiednie flagi w rejestrze statusu. Skasowanie tych flag następuje poprzez wpisanie 1 na pozycję flagi USISIF lub USIOIF rejestru stanu.



Wskazówka

W przeciwieństwie do większości innych flag, wywołanie odpowiedniej procedury obsługi przerwania z nimi związanego nie powoduje ich automatycznego wyzerowania.

W trybach innych niż SPI nie należy także odblokowywać przerwania detekcji bitu startu — będzie ono wyzwalane przy każdej zmianie stanu sygnału USCK.

## Zmiana pozycji pinów

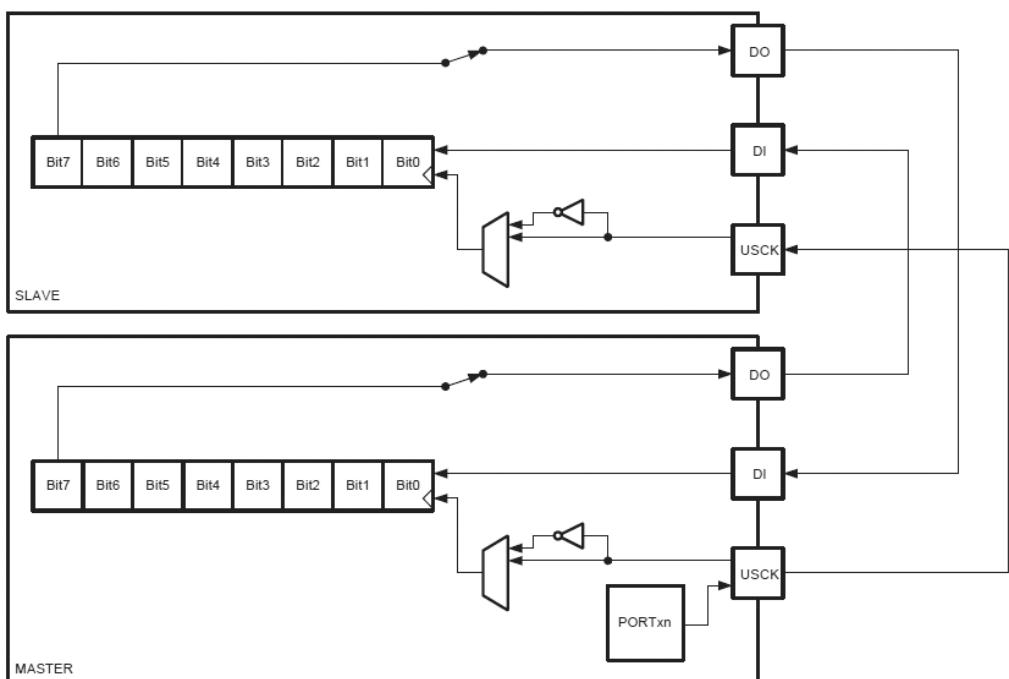
Ciekawą i przydatną funkcją spotykaną w niektórych procesorach AVR jest możliwość zmiany przyporządkowania interfejsu USI do pinów wyjściowych procesora. Domyślnie są one zwykle współdzielone z wyprowadzeniami interfejsu ISP umożliwiającego programowanie procesora. Dzięki ustawieniu bitu USIPOS (ang. *USI Pin Position*) rejestru USIPP (ang. *USI Pin Position*) można zmienić to przyporządkowanie. Np. w procesorze ATTiny461 domyślnie piny USI odpowiadają wyprowadzeniom interfejsu ISP (piny IO PB0 – PB2), lecz po ustawieniu bitu USIPOS wykorzystane zostają piny PA0 – PA2. Oprócz możliwości rozdzielenia w ten sposób funkcji ISP od funkcji związanych z portem USI, daje to także możliwość realizacji dwóch interfejsów sprzętowych przy pomocy tylko jednego interfejsu USI. Oczywiście, w danej chwili aktywny może być tylko jeden interfejs, np. na pinach PA0 – PA2 interfejs TWI, a na pinach PB0 – PB2 interfejs SPI.

## Wykorzystanie interfejsu USI w trybie SPI

Interfejs USI umożliwia łatwą realizację interfejsu SPI w trybie *master* i *slave*, przy czym w trybie *slave* nie posiada zaimplementowanej funkcjonalności pinu SS, lecz tę niedogodność łatwo skorygować programowo. W odróżnieniu do klasycznego interfejsu SPI zmienione zostały także nazwy wyprowadzeń:

- ◆ MOSI zostało nazwane DO.
- ◆ MISO nazywa się DI.
- ◆ SCK nosi nazwę USCK.

W przeciwieństwie do sprzętowego interfejsu SPI zmiana trybu pracy USI nie wpływa na zmianę funkcjonalności pinów DO i DI. W efekcie urządzenie może pracować jako *master* lub *slave*, ale wymaga to zmiany połączenia pinów DI i DO z innymi urządzeniami na magistrali SPI, co zazwyczaj nie jest możliwe w gotowym układzie. Schemat połączenia dwóch procesorów AVR z wykorzystaniem interfejsu USI w trybie SPI pokazano na rysunku 22.1.



Rysunek 22.1 Połączenie dwóch mikrokontrolerów AVR przy pomocy interfejsu USI w trybie SPI

Urządzenie *master* generuje przebieg zegarowy taktujący interfejs USI urządzenia *slave*. Po wygenerowaniu 8 cykli zegara zawartość rejestrów USIDR urządzeń *master* i *slave* wymienia się. Przepełnienie licznika (czyli przetransmitowanie całego bajtu) powoduje ustawienie bitu USIOIF (ang. *Counter Overflow Interrupt Flag*) rejestru USISR (ang. *USI Status Register*).

W tym trybie najwygodniej jest generować przebieg zegarowy programowo, poprzez cykliczne wpisywanie 1 do bitu USITC rejestru USICR. Proces ten można zautomatyzować, wykorzystując jako źródło zegara *timer* 0 i jego przerwanie *Compare Match*, lecz wobec niemożności wygenerowania sprzętowo zaprogramowanej liczby impulsów i tak, niestety, trzeba je zliczać. Zaletą wykorzystania *timera* jest możliwość dosy-

dokładnego określenia częstotliwości zegara SCK oraz możliwość generowania asynchronicznej w stosunku do przebiegu programu transmisji. Ma to szczególne znaczenie w sytuacji, kiedy zegar taktujący SPI jest bardzo wolny, przez co transmisja jednego bajtu zajmuje dużo czasu.

## Tryb SPI master

Ten tryb pracy jest niezwykle prosty do uzyskania. W trybie tym linia USCK jest wyjściem, na którym ma pojawić się przebieg zegarowy doprowadzony do urządzenia *slave*. W związku z tym odpowiadający jej pin IO należy ustawić jako wyjście. Podobnie jak w przypadku sprzętowego interfejsu SPI należy zadbać o właściwe sterowanie sygnałem SS doprowadzonym do układu *slave*. Jego generowanie jest czysto programowe. Prześledźmy, jak wyglądają przykładowe procedury obsługi USI SPI w trybie *master*. Przede wszystkim, dla wygody zdefiniowane zostaną symbole przyporządkowujące sygnały interfejsu USI do odpowiednich pinów IO. Jest to wygodne, gdyż przyporządkowanie to można zmienić programowo poprzez zmianę stanu bitu USIPOS. Definicje te wyglądają następująco:

```
#define SPI_SS      A, 0
#define SPI_CLK     B, 2
#define SPI_DI      B, 0
#define SPI_DO      B, 1
```

Zdefiniujmy teraz funkcję inicjującą:

```
void USI_SPI_master_init()
{
    SET(DDR, SPI_DO);
    SET(DDR, SPI_CLK); //Piny DO i CLK mają być wyjściami
}
```

Jak widać, jej działanie ogranicza się tylko do właściwego ustawienia kierunków wykorzystanych portów.

Ostatnią funkcją niezbędną do realizacji SPI jest funkcja umożliwiająca wysłanie i odbiór znaku:

```
uint8_t USI_SPI_Transmit(uint8_t data)
{
    USIDR=data;
    USISR=_BV(USIOIF); //Wyzeruj licznik
    do
    {
        USICR=_BV(USIWM0) | _BV(USICS1) | _BV(USICLK) | _BV(USITC);
    } while((USISR & _BV(USIOIF))==0);
    return USIDR;
}
```

Funkcja ta najpierw zeruje licznik związany z interfejsem USI, zapisuje bajt do wysłania do rejestru USIDR, a następnie cyklicznie ustawia bit USITC, powodując w ten sposób transmisję kolejnych bitów. Jednocześnie do rejestru „wsuwane” są kolejne bity z wejścia DI interfejsu. Zauważmy, że nie ma potrzeby zliczania wygenerowanych cykli,

gdyż wysunięcie całego bajtu spowoduje przepełnienie licznika i w efekcie ustawienie flagi USIOIF, co zakończy pętlę. Dzięki ustawieniu pinu USCK jako wyjścia wygenerowany przebieg zegarowy będzie dostępny dla układu *slave*.

## Tryb SPI slave

Tryb ten różni się od trybu *master* sposobem wykorzystania licznika. W tym trybie licznik jest taktowany zdarzeniem zewnętrznym (przebiegiem doprowadzonym do wejścia USCK). Licznik w tym trybie pracy zlicza każde zbocze sygnału zegarowego, a po przepełnieniu ustawia flagę USIOIF. Nastąpi to po zliczeniu 16 zboczy, a więc po odebraniu dokładnie 8 bitów danych. Inicjalizacja trybu USI SPI *slave* wygląda następująco:

```
void USI_SPI_slave_init()
{
    SET(DDR, SPI_DO);
    CLR(DDR, SPI_DI);
    CLR(DDR, SPI_CLK);
    CLR(DDR, SPI_SS);
    USICR=_BV(USIOIE) | _BV(USIWM0) | _BV(USICSO) | _BV(USICS1);
    USISR=_BV(USIOIF); //Skasuj flagę nadmiaru
    sei();
}
```

W tym trybie, przeciwnie do trybu *master*, pin, do którego doprowadzony jest sygnał zegarowy, jest skonfigurowany jako wejście. **Kierunki pinów DO i DI nie ulegają zmianie.** Aby odciążyć procesor, przepełnienie licznika będzie generowało przerwanie. Jego wygenerowanie świadczy o odebraniu całego bajtu danych. Procedura jego obsługi wygląda następująco:

```
volatile uint8_t dane_in, dane_out;
volatile uint8_t flaga;

ISR(USI_OVF_vect)
{
    dane_in=USIDR;
    USISR=_BV(USIOIF);
    USIDR=dane_out;
    flaga=255;
}
```

Po wywołaniu przerwania odebrany bajt jest odczytywany z rejestru USIDR i zapisywany w globalnej zmiennej *dane\_in*, skąd może zostać odczytany przez aplikację. Jednocześnie do rejestru wpisywana jest zawartość zmiennej *dane\_out*, w wyniku czego jeśli urządzenie *master* wyśle kolejny bajt, będzie mogło w zamian odebrać wpisaną wartość. Odebranie nowego bajtu sygnalizowane jest nadaniem zmiennej *flaga* wartości 255. Zmienna ta powinna w programie być cyklicznie odczytywana w celu sprawdzenia, czy nie nadeszły nowe dane. Po ich odebraniu powinna być zerowana, co umożliwia prawidłowe rozpoznanie nadania kolejnego bajtu danych.

# Rozdział 23.

# Interfejs USB

W ostatnich latach coraz większą popularnością cieszy się interfejs USB (ang. *Universal Serial Bus*). Dzieje się tak nie tyle z powodu jego zalet, co stopniowego zaniku prostych interfejsów, takich jak RS232 czy interfejs równoległy. Pomimo prostoty sprzętowej interfejsu USB warstwa logiczna protokołu jest stosunkowo skomplikowana, przez co jej implementacja wymaga znacznego wsparcia ze strony sprzętowej. Stąd też proste procesory AVR, nieposiadające sprzętowego interfejsu USB, praktycznie nie mogą występować w roli urządzenia hosta USB, natomiast da się programowo zaprząć je do roli urządzenia peryferyjnego. Dużą zaletą interfejsu USB jest możliwość pobierania z niego prądu do zasilania podłączonego układu. Dzięki temu wiele prostych urządzeń o niewielkim poborze prądu (do ok. 500 mA) można zasilać bezpośrednio z portu komputera.

Mikrokontrolery AVR można połączyć z komputerem klasy PC przy pomocy interfejsu USB na kilka sposobów:

- ◆ przy pomocy zewnętrznych układów scalonych działających jako konwertery USB – RS232, USB – interfejs równoległy (np. popularne układy firmy FTDI);
- ◆ realizując programowo interfejs USB — dzięki temu praktycznie każdy mikrokontroler AVR może działać jako urządzenie peryferyjne USB;
- ◆ wykorzystując mikrokontrolery AVR posiadające wbudowany sprzętowy interfejs U\$B.

Interfejs USB łączy się z mikrokontrolerem lub układem pośredniczącym przy pomocy 4 sygnałów (tabela 23.1).

**Tabela 23.1.** kolejność sygnałów na złączu USB

Nr pinu	Sygnal	Kolor przewodu	Opis
1	Vcc	Czerwony	+5V
2	D-	Biały	Data-
3	D+	Zielony	Data+
4	GND	Czarny	Masa

Istnieje szereg gniazd USB, przy czym urządzenia peryferyjne powinny być wyposażone w gniazda typu B. Urządzenia typu USB host posiadają gniazda typu A. Ma to duże znaczenie ze względu na wystawienie przez urządzenie host na złącze napięcia zasilającego. Urządzenie peryferyjne posiadające gniazdo typu B może z pinu o numerze 1 czerpać zasilanie, w żadnym wypadku nie może jednak dostarczać zasilania na ten pin (grozi to uszkodzeniem obu urządzeń). Sygnały na magistrali USB przesyłane są w sposób różnicowy, co zwiększa odporność na zakłócenia, przy pomocy splecionej pary przewodów. Wykorzystując ten interfejs, należy pamiętać, że ścieżki łączące linie D+ i D- z gniazda do układu scalonego powinny być możliwie krótkie i prowadzone równolegle do siebie. Należy także zwrócić uwagę na maksymalną długość łączącego urządzenia kabla. Według specyfikacji USB 1.1/2.0 maksymalna długość kabla nie powinna przekraczać 5 m, jednak granica ta jest płynna i zależy od jego jakości. Należy przyjąć, że tanie kable USB prawdopodobnie będą stwarzały kłopoty przy znacznie mniejszej długości. Objawia się to niestabilną pracą urządzenia, licznymi błędami transmisji, w efekcie tak podłączone urządzenie czasami jest prawidłowo rozpoznawane przez komputer, a czasami w ogóle nie jest widziane. W takich sytuacjach zmiana kabla na kabel o lepszej jakości zwykle pomaga.

## Zasilanie

Jak wspomniano, ogromną zaletą interfejsu USB jest możliwość czerpania z niego zasilania dla zbudowanego układu. Specyfikacja USB 1.x gwarantuje, że napięcie pomiędzy liniami GND i Vcc mieści się w zakresie 4,75 – 5,25 V, lecz już wg specyfikacji USB 2.0 powinno mieścić się w zakresie 4,4 – 5,25 V, a wg specyfikacji USB 3.0 w zakresie 4 – 5,25 V. Budując więc układ zasilany z USB, należy uwzględnić, że jego napięcie zasilające może zmieniać się w dosyć szerokich granicach, w zależności od obciążenia portu i długości kabla zasilającego. Z gniazda USB można pobierać maksymalnie prąd równy 5 jednostkom obciążenia (dla USB 1/2.0 jednostka obciążenia to 100 mA, a dla USB 3.0 150 mA). Co ważne, początkowo urządzenie może pobierać maksymalnie jedną jednostkę obciążenia (czyli 100 mA), dopiero po enumeracji i konfiguracji może pobierać pełną moc, czyli 5 jednostek obciążenia. Jednak większość hostów USB nie jest specjalnie restrykcyjna i toleruje urządzenia nie do końca spełniające tak ścisłe określone warunki. Dla urządzeń pobierających znaczny prąd potencjalnym problemem może być ich podłączenie poprzez tzw. huby USB. W przypadku rozgałęziaczy (hubów) nieposiadających własnego zasilania wszystkie urządzenia podłączone do huba nie mogą pobierać więcej niż 500 mA. W rezultacie niektóre urządzenia nie działają poprawnie w takiej konfiguracji.

Urządzenia USB ze względu na pobór prądu podzielono na urządzenia o niskim poborze (ang. *Low Power*) i urządzenia o wysokim poborze (ang. *High Power*). Urządzenia o niskim poborze pobierają nie więcej niż jedną jednostkę obciążenia (a więc 100 lub 150 mA w zależności od wersji interfejsu). Typ urządzenia określa się w deskryptorze przekazywanym podczas jego inicjalizacji. Jeżeli urządzenie potrzebuje więcej mocy, niż jest w stanie dostarczyć port USB, można podłączyć je do kilku portów (sumując

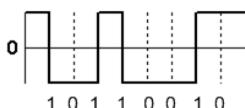
w ten sposób ich maksymalne obciążenie) lub zastosować zasilacz zewnętrzny. **W tym ostatnim przypadku należy pamiętać, aby w żadnym przypadku nie wystawiać napięcia zasilającego na pin Vcc gniazda USB.**

## Sygnały danych

Dane na magistrali przesyłane są różnicowo, przy pomocy dwóch linii (D+ i D–). Niski poziom logiczny mieści się w zakresie 0 – 0,3 V, wysoki poziom logiczny mieści się w zakresie 2,8 – 3,6 V. Należy zwracać uwagę, aby w sytuacji zasilania procesora napięciem większym niż 3,6 V zapewnić, aby na liniach sygnałowych napięcie nie przekraczało wartości 3,6 V. W tym celu stosuje się zwykle diody Zenera, a w przypadku procesorów AVR ze sprzętowym interfejsem USB wewnętrzny regulator napięcia. Sygnały przesyłane są w trybie *half-duplex*. W zależności od wybranego trybu pracy dane przesyłane są z prędkością:

- ◆ 1,5 Mbit/s — dla urządzeń *low-speed*. Tryb ten wykorzystuje się w przypadku powolnych urządzeń, o niewielkim zapotrzebowaniu na pasmo. Jest to też jedyny możliwy do osiągnięcia tryb pracy w przypadku wykorzystania programowego interfejsu USB na mikrokontrolerach AVR.
- ◆ 12 Mbit/s — dla urządzeń *full-speed*. Jest to tryb wspierany m.in. przez układy interfejsów produkowane przez firmę FTDI oraz przez sprzętowy interfejs USB procesorów AVR.
- ◆ Prędkości 480 i 4800 Mbit/s — nie są wspierane przez procesory AVR, a ze względu na ich moc obliczeniową prawdopodobnie nie ma większego sensu korzystać z tych trybów.

W stanie spoczynku linie D+ i D– są w stanie logicznym 0, wymuszonym przez rezystory o wartości ok. 15 kΩm podłączone pomiędzy te linie i masę. Urządzenie peryferyjne USB podciąga jedną z linii (D+ lub D–) poprzez rezystor o wartości ok. 1,5 kΩm do Vcc, dzięki czemu podłączenie urządzenia może zostać rozpoznane przez urządzenie USB host. Taki stan jest podstawowym stanem magistrali USB i nazywany jest stanem J. Stan przeciwny do spoczynkowego nazywany jest stanem K. Do przesyłania danych wykorzystywany jest kod NRZI (ang. *Non-Return-to-Zero Inverted*) — rysunek 23.1.



**Rysunek 23.1.** Zasada działania kodu NRZI. Bit o wartości 0 transmitowany jest poprzez zmianę stanu magistrali na przeciwny. Bit o wartości 1 transmitowany jest jako brak zmiany stanu magistrali

Aby uniknąć problemów z odtworzeniem zegara i synchronizacją, w przypadku nadawania serii bitów o wartości 1 po szóstym takim bicie nadawany jest zawsze dodatkowy bit o wartości 0. Jeśli na magistrali odebranych zostanie jednorazowo więcej niż 6 bitów o wartości 1, oznacza to błąd. Transmisja pakietu danych rozpoczyna się od

nadania sekwencji synchronizującej, składającej się z 7 bitów o wartości 0, po których nadawany jest bit o wartości 1. Bit o wartości jeden oznacza koniec ciągu synchronizującego. Dla urządzeń o większej prędkości ciąg synchronizujący składa się z 32 bitów. Koniec pakietu sygnalizowany jest poprzez przejście na czas trwania dwóch bitów linii D+ i D– w stan zero (sygnał SE0), a następnie w stan J. Magistralę można zresetować poprzez nadanie sygnału SE0 o czasie trwania 10 – 20 ms.

Ponieważ przebieg transmisji jest krytycznie zależny od zegara, USB wymaga, aby był on stosunkowo stabilny. Dla prędkości 1,5 Mbit/s tolerancja zegara wynosi 15 000 ppm, dla prędkości 12 Mbit/s zmniejsza się do 2500 ppm. **W praktyce oznacza to, że urządzenia wykorzystujące interfejs USB muszą być taktowane przy pomocy kwarcu.**

Transfer danych po magistrali odbywa się w pakietach. W jednym pakiecie można wysłać od 0-1023 bajtów danych. Oprócz pakietów danych wysyłane są różne pakiety kontrolne, w efekcie sam protokół jest stosunkowo skomplikowany. Ponieważ nadanie pakietu danych wymaga nadania pakietu adresowego, odebrania potwierdzeń, nadania pakietu danych i odebrania potwierdzenia, cały proces wymiany danych jest stosunkowo czasochłonny. Stąd też lepiej wysyłać mniej dużych pakietów danych; wysyłanie pojedynczych bajtów danych jest bardzo czasochłonne, w efekcie realna do uzyskania prędkość przesyłu danych ulega znaczнемu obniżeniu. W efekcie, przy niedostosowanym programie, wysyłającym poprzez USB pojedyncze bajty danych, efektywna prędkość transmisji może być mniejsza niż w przypadku interfejsu RS232.

## VID i PID

Standard USB definiuje nie tylko niskopoziomowy sposób transmisji danych, ale także wyższe poziomy protokołu komunikacji pomiędzy urządzeniami. Każde urządzenie USB musi mieć możliwość odpowiedzi na pewien podstawowy zestaw żądań wysyłanych przez USB-host. Aby ułatwić instalowanie sterowników dla danego urządzenia, każde urządzenie peryferyjne USB posiada unikalny zestaw 16-bitowych identyfikatorów. Pierwszy, VID (ang. *Vendor ID*), identyfikuje producenta urządzenia, drugi, PID (ang. *Product ID*), identyfikuje konkretny produkt. Po otrzymaniu deskryptora urządzenia system operacyjny poszukuje odpowiedniego drivera, identyfikując go po numerach PID i VID. Identyfikatorów tych nie możemy więc przydzielić urządzeniu w sposób dowolny. Aby zgodnie z prawem móc na swoim urządzeniu umieścić logo bądź informację, że jest kompatybilne ze standardem USB, należy wykupić numer VID w organizacji je kontrolującej ([www.usb.org](http://www.usb.org)). Oczywiście, wiąże się to z pewnymi opłatami, co szczególnie dla hobbystów powoduje, że ta opcja jest niedostępna. Dopóki budujemy urządzenie na własny użytek, możemy używać dowolnych identyfikatorów, z tym że lepiej, aby nie były one przydzielone urządzeniom, dla których istnieją w systemie operacyjnym sterowniki, gdyż w takiej sytuacji system może zainstalować nieprawidłowy sterownik. W przypadku produktów komercyjnych mamy dwie opcje. Pierwsza to wykupienie VID (co kosztuje rocznie \$ 4000 za członkostwo i jednorazowo \$ 2000) lub wykorzystanie w swoim urządzeniu układu interfejsu USB, dla którego producent udostępnia własny VID (np. chipów firmy FTDI). Należy tu nadmienić, że pomimo iż firma Atmel w niektórych procesorach umieszcza sprzętowy interfejs USB,

to kupując taki procesor, nie nabywamy żadnych praw do używania numerów VID przydzielonych tej firmie. W efekcie stajemy w sytuacji, w której dla komercyjnego produktu ciągle musimy ten numer wykupić.

Problem z numerami VID i PID można ominąć w jeszcze jeden sposób — korzystając z tzw. numerów współdzielonych. Niektóre firmy udostępniają własne numery VID i PID. W takiej sytuacji urządzenie jest jednoznacznie identyfikowane na podstawie nazwy — stąd też nazwa urządzenia powinna być unikalna. Zazwyczaj przyjmuje się, że jest nią adres e-mail producenta lub jego strona WWW. Wykorzystując współdzielone numery ID, należy pamiętać, że sterownik działający na poziomie kernela będzie ten sam dla wszystkich urządzeń posiadających określoną kombinację VID/PID. Rozróżnienie można przeprowadzić dopiero na wyższym poziomie — w aplikacji użytkownika. W tabeli 23.2 pokazano przykładowe numery VID/PID, które można wykorzystać w swoich urządzeniach dzięki uprzejmości firmy Objective Development. W tabeli 23.3 pokazano klasy VID/PID, które można wykorzystać, identyfikując urządzenie po numerze seryjnym.

**Tabela 23.2.** Przykładowe kombinacje VID/PID, które można wykorzystać we własnych urządzeniach

<b>VID</b>	<b>PID</b>	<b>Klasa</b>
5824	1500	Sterownik libusb
5824	1503	Urządzenia HID z wyjątkiem klawiatury, myszy i joysticka
5824	1505	Modemy
5824	1508	Urządzenia MIDI

**Tabela 23.3.** Przykładowe kombinacje VID/PID, które można wykorzystać we własnych urządzeniach identyfikowanych poprzez unikalny numer seryjny

<b>VID</b>	<b>PID</b>	<b>Klasa</b>
5824	10200	Sterownik libusb
5824	10201	Urządzenia HID z wyjątkiem klawiatury, myszy i joysticka
5824	10202	Mysz
5824	10203	Klawiatura
5824	10204	Joystick
5824	10205	Modem
5824	10206	Urządzenie MIDI

Na podobnej zasadzie można wykorzystać numery VID/PID przydzielone firmie FTDI, w sytuacji, w której do realizacji interfejsu USB wykorzystuje się układy tej firmy.

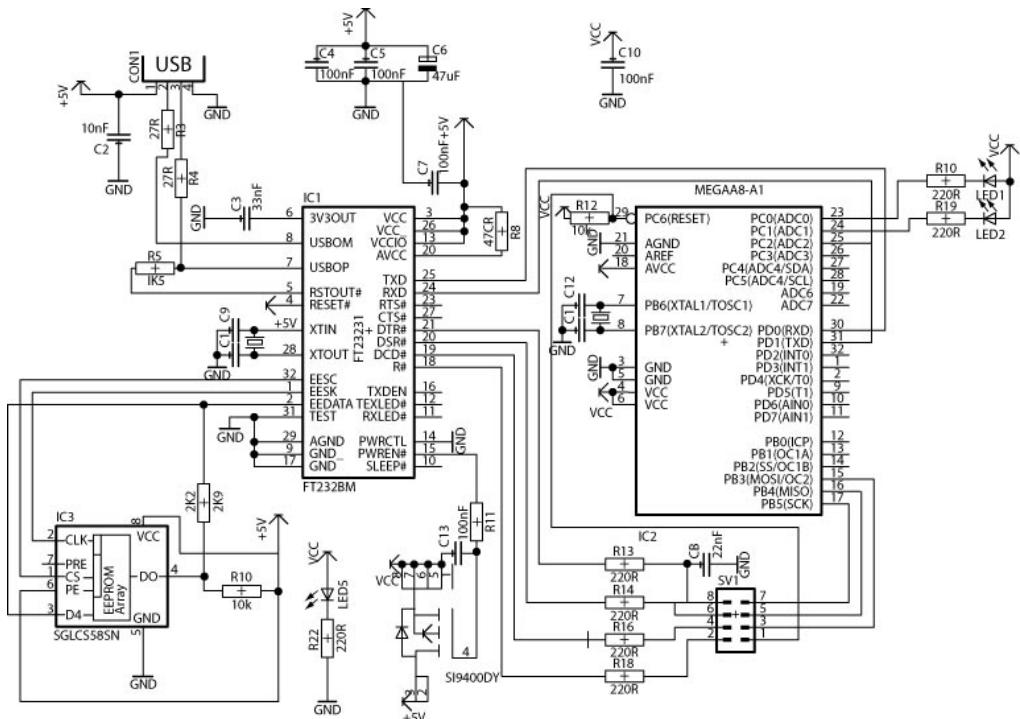
# Interfejs USB realizowany przy pomocy konwertera

Jedną z najprostszych możliwości podłączenia mikrokontrolera AVR z komputerem poprzez interfejs USB jest wykorzystanie specjalnego konwertera, którego celem jest obsługa całego protokołu USB, zarówno od strony elektrycznej, jak i protokołu transmisji. Zwykle mikrokontroler łączy się z konwerterem przy pomocy interfejsu RS232 (dokładniej RS232-TTL, dzięki czemu nie jest potrzebny konwerter poziomów MAX232). Zwykle układy interfejsów dysponują także kilkoma – kilkunastoma pinami *IO*, których stanem można łatwo sterować, dzięki czemu układ taki może jednocześnie służyć jako programator. Jedną z najtańszych możliwości jest wykorzystanie jako konwertera kabli USB-RS232 wykorzystywanych do podłączenia starszych telefonów komórkowych (nieposiadających interfejsu USB). Kable takie można kupić za parę złotych, w tej cenie oprócz układu konwertera dostajemy kabel zakończony wtykiem USB. Inną możliwością jest wykorzystanie scalonych konwerterów. Na tym polu prym wiedzie firma FTDI, wraz ze swoimi przebojem rynkowym — układem FT232 i jego odmianami. Jego podłączenie do mikrokontrolera AVR pokazano na rysunku 23.2.

Układ konwertera jest odpowiedzialny za właściwe poziomy napięć interfejsu USB, umożliwia także sterowanie zasilaniem reszty układu. Po wyłączeniu danego urządzenia, poprzez tranzystor SI9400, odłączane jest zasilanie od reszty układu, minimalizując w ten sposób pobór prądu.

Oprócz łatwo dostępnego układu FT232BM można zastosować któryś z nowszych modeli tej rodziny, dzięki czemu nie trzeba podłączać zewnętrznej pamięci EEPROM, zawierającej konfigurację układu, ani kwarca, odpowiedzialnego za generowanie zegara taktującego wymianę danych na magistrali USB. W efekcie układ ulega znacznemu uproszczeniu.

Po podłączeniu układu do komputera PC system operacyjny znajduje nowe urządzenie USB i próbuje zainstalować do niego sterowniki, które można pobrać ze strony firmy FTDI ([www.ftdichip.com/FTDrivers.htm](http://www.ftdichip.com/FTDrivers.htm)). Do wyboru są dwie wersje sterowników. Sterowniki VCP (ang. *Virtual COM port*) po zainstalowaniu powodują, że układ FTDI (a co za tym idzie, całe urządzenie) jest widoczny w systemie operacyjnym jako dodatkowy port szeregowy. Tryb ten jest prosty do oprogramowania, co więcej, do wymiany danych z urządzeniem można wykorzystać standardowe programy terminalowe. Tryb ten nie umożliwia wykorzystania pełnych możliwości układów firmy FTDI ani bardzo wysokich prędkości transferu danych. Drugą wersją są sterowniki D2XX. Oprócz sterownika zawierają one także bibliotekę współdzieloną (dla Windows DLL, dla GNU/Linux so), zawierającą funkcje umożliwiające wykorzystanie pełnych możliwości układów FTDI. Dzięki tej bibliotece możliwa jest enumeracja wszystkich podłączonych urządzeń FTDI oraz odczytanie ich deskryptorów. Dzięki temu nie ma potrzeby konfigurowania aplikacji (np. poprzez podanie numeru wirtualnego portu szeregowego, poprzez który podłączone jest urządzenie). Sterowniki te umożliwiają także sterowanie wszystkimi liniami *IO* układu oraz wysyłanie danych w postaci pakietów, co znakomicie przyśpiesza transmisję.



Rysunek 23.2. Podłączenie układu FT232BM do mikrokontrolera AVR poprzez interfejs RS232. Dzięki podłączeniu dodatkowych wyjść IO układu konwertera do interfejsu ISP mikrokontrolera istnieje także możliwość jego programowania poprzez interfejs USB, bez pośrednictwa programatora (w tym celu należy zewrzeć zworkę SV1)

Odbiór danych na mikrokontrolerze odbywa się poprzez port USART, dokładnie tak samo jakby mikrokontroler był połączony z komputerem poprzez interfejs RS232. Szerzej ten typ połączenia został omówiony w rozdziale 19.

## Interfejs USB realizowany programowo

Pierwszą osobą, która zrealizowała całkowicie programowo urządzenie peryferyjne USB przy pomocy mikrokontrolera AVR, był Igor Cesko (<http://www.cesko.host.sk>). Oryginalny układ był oparty na starym mikrokontrolerze AT90S2313 przetaktowanym do 12 MHz. Obecnie w sieci można znaleźć różne podobne do oryginału układy, wykorzystujące chyba wszystkie dostępne mikrokontrolery AVR. Także firma Atmel udostępniła notę aplikacyjną i kody realizujące programową obsługę interfejsu USB<sup>1</sup>.

<sup>1</sup> AVR309: Software Universal Serial Bus (USB)

Programowy interfejs USB składa się z trzech elementów:

- ◆ prostego układu sprzętowego, realizującego fizyczne połączenie AVR z interfejsem USB komputera;
- ◆ oprogramowania działającego na mikrokontrolerze, odpowiedzialnego za odbiór i transmisję danych po USB;
- ◆ prostego sterownika, działającego na komputerze PC, umożliwiającego wymianę danych z układem elektronicznym i aplikacją użytkownika.

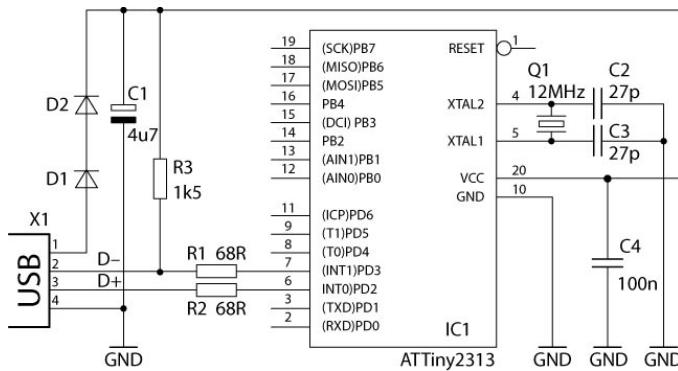


Ze względu na ograniczoną moc obliczeniową mikrokontrolerów AVR interfejs ten umożliwia wymianę danych z maksymalną prędkością 1,5 Mbit/s.

Tryby pracy interfejsu USB o większej prędkości są niedostępne. Jednak w większości przypadków ograniczenie to nie jest dotkliwe, a niski koszt realizacji powoduje, że wielu hobbystów wybiera to rozwiązanie. Na tym rozwiążaniu bazuje projekt V-USB, ze strony którego można pobrać kody realizujące programowo USB, dostępne na licencji GPL.

## Połączenie elektryczne

Na rysunku 23.3 pokazano sprzętowy sposób podłączenia AVR z interfejsem USB komputera.



**Rysunek 23.3.** Podłączenie AVR do USB. Diody D1 i D2 mają za zadanie zmniejszyć napięcie do poziomu akceptowalnego na wyrowadzeniach D+ i D– interfejsu (maksymalnie 3,3 V). Zamiast tych diod można zastosować regulator napięcia LDO na 3,3 V. Alternatywnie stosuje się diody Zenera na napięcie 3,6 V, podłączone zaporowo do masy na pinach D+ i D–. Przy takim rozwiążaniu procesor zasilany jest wyższym napięciem, co umożliwia taktowanie go przebiegiem o wyższej częstotliwości

## Dostęp na PC

Do zapewnienia komunikacji z urządzeniem należy na PC zainstalować bibliotekę libusb, udostępniającą API dla programów chcących komunikować się poprzez USB. Biblioteka ta dostępna jest w wersji dla różnych systemów operacyjnych, co ułatwia pisanie opro-

gramowania działającego na różnych platformach programowych. Bibliotekę tę można pobrać ze strony projektu <http://sourceforge.net/projects/libusb-win32>.

## Programowy interfejs USB na AVR

Najważniejszą częścią pakietu V-USB jest kod realizujący programowo interfejs USB na mikrokontrolerach rodziny AVR. Kod źródłowy można pobrać ze strony projektu — <http://www.obdev.at/products/vusb/download.html>. Po jego rozpakowaniu w katalogu *usbdrv* znajduje się kod odpowiedzialny za realizację interfejsu. Aby go użyć, należy dokonać pewnej konfiguracji, związanej z wyborem zegara taktującego procesora oraz pinów I/O zaangażowanych do realizacji interfejsu. Obecny sterownik wspiera procesory taktowane kwarcami 12; 12,8; 15; 16; 16,5; 18 i 20MHz.



Ze względu na wymagania czasowe USB procesor powinien być taktowany przy pomocy kwarcu. Istnieje co prawda możliwość taktowania przy pomocy skalibrowanego wewnętrznego generatora RC, lecz nie jest to opcja zalecana.

Wszystkich konfiguracji dokonuje się, edytując plik *usbconfig-prototype.h*. Po jego właściwym skonfigurowaniu należy zapisać go pod nazwą *usbconfig.h*. Plik ten jest automatycznie włączany przez pozostałe pliki pakietu. Poniżej zostaną omówione najważniejsze opcje konfiguracji.

### Konfiguracja portu I/O

Konfigurację rozpoczynamy od wybrania pinów I/O, które posłużą do podłączenia sygnałów D+ i D– interfejsu USB. Wybierając te piny, należy pamiętać o dwóch rzeczach:

- ◆ Oba piny muszą należeć do jednego portu I/O.
- ◆ Pin, do którego podłączony jest sygnał D+, musi być jednocześnie pinem, który może wyzwalać przerwanie o wektorze INT0.

To ostatnie ograniczenie można ominąć, lecz wymaga to pewnych zmian w kodzie źródłowym biblioteki.

Port I/O, do którego należą wybrane piny, określa się, definiując symbol `USB_CFG_IOPORT_NAME`; np.:

```
#define USB_CFG_IOPORTNAME      D
```

powoduje wybranie pinów portu D.

Definicje `USB_CFG_DMINUS_BIT` i `USB_CFG_DPLUS_BIT` wybierają numery pinów I/O, do których podłączony jest sygnał D+ i D– magistrali USB.

Na schemacie przedstawionym na rysunku 23.3 rezistor R3 podłączony jest do zasilania — umożliwia on wykrycie przez USB-host podłączenia urządzenia periferyjnego.

Rezystor ten może być opcjonalnie podłączony nie do zasilania, lecz do pinu I0 mikrokontrolera, co umożliwia jego programowe odłączenie — dzięki temu sterownik udostępnia dwie dodatkowe funkcje:

- ◆ `usbDeviceConnect()` — włączającą rezystor podciągający, a co za tym idzie urządzenie USB.
- ◆ `usbDeviceDisconnect()` — wyłączającą rezystor R3. W efekcie urządzenie przestaje być widoczne dla USB-host (zachowuje się tak, jakby zostało fizycznie odłączone).

Jeśli chcemy skorzystać z wyżej wymienionych funkcji, należy podłączyć R3 do wybranego pinu I0 mikrokontrolera oraz zdefiniować symbol `USB_CFG_PULLUP_IOPORTNAME`, nadając mu nazwę portu, do którego został podłączony rezystor, oraz symbol `USB_CFG_PULLUP_BIT`, przyporządkowując mu numer wybranego pinu (0 – 7).

## Wybór zegara

W następnej kolejności należy zdefiniować symbol `USB_CFG_CLOCK_KHZ`, przyporządkowując mu częstotliwość pracy procesora w kHz. Wybrać można jedną z częstotliwości: 12; 12,8; 15; 16; 16,5; 18 lub 20 MHz.

## CRC

Opcjonalnie sterownik może sprawdzać integralność danych, poprzez wyliczanie CRC pakietu. Opcję tę można włączyć, nadając wartość 1 etykietce `USB_CFG_CHECK_CRC`. Włączenie sprawdzania CRC powoduje wydłużenie kodu sterownika.

## Zasilanie

Symbol `USB_CFG_IS_SELF_POWERED` określa sposób zasilania urządzenia USB. Jego domyślna wartość, 0, informuje USB-host, że urządzenie jest zasilane z magistrali USB. Nadając mu wartość 1, możemy określić, że urządzenie ma własne zasilanie. Z kolei symbol `USB_CFG_MAX_BUS_POWER` określa pobór prądu w mA przez urządzenie. Nie ma znaczenia jego dokładna wartość, istotne jest tylko, czy urządzenie pobiera mniej niż 100 mA (urządzenie *low-power*) czy też więcej (urządzenie *high-power*).

## VID, PID i reszta

Obowiązkowo należy zdefiniować parametry identyfikujące urządzenie. Bez tego nie jest możliwa poprawna praca oraz zainstalowanie sterownika w systemie operacyjnym.

VID definiuje się, przypisując jego wartość symbolowi `USB_CFG_VENDOR_ID`. Jego domyślna wartość 0x16C0 odpowiada współdzielonemu VID, wspieranemu przez bibliotekę libusb. PID urządzenia przypisuje się symbolowi `USB_CFG_DEVICE_ID`. Domyślna jego wartość 0x05DC odpowiada współdzielonemu PID, obsługiwianemu przez sterownik libusb. W obu przypadkach młodszy bajt występuje w definicji jako pierwszy, np.:

```
#define USB_CFG_VENDOR_ID      0xc0, 0x16
#define USB_CFG_DEVICE_ID       0xdc, 0x05
```

Oprócz VID I PID możemy zdefiniować wersję urządzenia, przypisując ją symbolowi `USB_CFG_DEVICE_VERSION`. W przypadku współdzielonego ID niezwykle ważne jest, aby urządzeniu nadać unikalną nazwę. Skonfigurować możemy nazwę producenta (symbole `USB_CFG_VENDOR_NAME` i `USB_CFG_VENDOR_NAME_LEN`) oraz nazwę urządzenia (symbole `USB_CFG_DEVICE_NAME` i `USB_CFG_DEVICE_NAME_LEN`), np.:

```
#define USB_CFG_VENDOR_NAME      'H', 'e', 'l', 'i', 'o', 'n'  
#define USB_CFG_VENDOR_NAME_LEN 6  
#define USB_CFG_DEVICE_NAME     'K', 's', 'i', 'a', 'z', 'k', 'a'  
#define USB_CFG_DEVICE_NAME_LEN 7
```

Na końcu pozostaje jeszcze zdefiniowanie klasy i podklasy urządzenia. Należy je zdefiniować zgodnie ze specyfikacją USB klas urządzeń. Jeśli jednak nie budujemy standardowego urządzenia, to najodpowiedniejsza będzie klasa 0xFF (zdefiniowana przez użytkownika) i podklastra 0:

```
#define USB_CFG_DEVICE_CLASS      0xff  
#define USB_CFG_DEVICE_SUBCLASS    0
```

## Inne opcje

W pliku `usbconfig.h` można zdefiniować wiele opcji mniej ważnych w amatorskich projektach. Niektóre opcje związane z zapewnieniem kompatybilności ze standardem USB są wyłączone, dzięki czemu znaczco zmniejsza się wielkość generowanego kodu, jednak kosztem potencjalnych problemów z kompatybilnością. Jednym z takich symboli jest `USB_CFG_IMPLEMENT_HALT`, którego domyślna wartość wynosi 0 (brak obsługi `ENDPOINT_HALT`). Jeśli wielkość kodu nie jest problemem, to ze względu na kompatybilność ze standardem należy symbolowi temu przypisać wartość 1.

W mikrokontrolerach posiadających powyżej 64 kB pamięci FLASH istotne znaczenie ma symbol `USB_CFG_DRIVER_FLASH_PAGE`. Jeśli deskryptory urządzenia USB znajdują się powyżej adresu 65 535, należy temu symbolowi nadać wartość 1. W praktyce dzieje się tak tylko wtedy, kiedy biblioteka V-USB znajduje się w obszarze `bootloadera`.

Domyślnie sterownik V-USB obsługuje pakiety danych o długości do 254 bajtów. Jest to wartość zazwyczaj wystarczająca, szczególnie dla małych mikrokontrolerów AVR, nieposiadających zbyt wiele pamięci SRAM. Jeśli jednak chcemy obsługiwać pakiety dłuższe, należy nadać wartość 1 symbolowi `USB_CFG_LONG_TRANSFERS`. Powoduje to jednak pewne zwiększenie długości kodu sterownika.

Symbol `USB_CFG_CHECK_DATA_TOGGLE` określa, czy sterownik ma wykrywać sytuacje, w których pojawiają się duplikaty danych. Sytuacja taka może się zdarzyć przy złej jakości połączenia USB, kiedy urządzenie USB-host nie otrzyma potwierdzenia odebrania pakietu. W takiej sytuacji następuje jego retransmisja, co urządzenie peryferyjne odbiera jako duplikat danych. Aby temu zapobiec, należy nadać wartość 1 wcześniej wymienionemu symbolowi oraz napisać własne funkcje filtrujące. W efekcie prościej jest zaimplementować podobną funkcjonalność w protokole wyższego poziomu, za pomocą którego odbywa się komunikacja pomiędzy oboma urządzeniami.

Po skonfigurowaniu biblioteki możemy ją dołączyć do własnego projektu. Przykłady jej wykorzystania znajdują się w katalogu examples. Proste „gotowce” znajdują się w katalogach libs-device i libs-host.

## Sprzętowy interfejs USB

Niektóre procesory AVR dysponują sprzętowym interfejsem USB. Należą do nich starsze mikrokontrolery posiadające w nazwie USB oraz nowsze, takie jak ATMega16U2 i ATMega32U2. Mikrokontrolery te posiadają także fabrycznie wgrany *bootloader*, umożliwiający ich programowanie przy pomocy narzędzi dostarczonych przez firmę Atmel (więcej na ten temat znajdziesz w rozdziale 25.). Co prawda firma Atmel udostępnia sterowniki obsługujące sprzętowy interfejs USB, lecz od lat dużym powodzeniem cieszą się otwarte sterowniki napisane przez Deana Camera, znajdujące się w jego projekcie LUFA (ang. *Lightweight USB Framework for AVR*s). Biblioteka ta zawiera implementacje sterowników dla mikrokontrolerów AT90USBxxx i ATMegaxxxUx. W przeciwieństwie do biblioteki udostępnionej przez firmę Atmel, kody biblioteki LUFA są kompatybilne z avr-gcc. Oprócz obsługi imponującej liczby urządzeń biblioteka ta posiada także obsługę *bootloaderów* zgodnych z notą AVR109 oraz urządzeń klasy DFU, dzięki czemu do programowania mikrokontrolerów AVR przez USB można wykorzystać narzędzia dostarczone przez firmę Atmel.

# Rozdział 24.

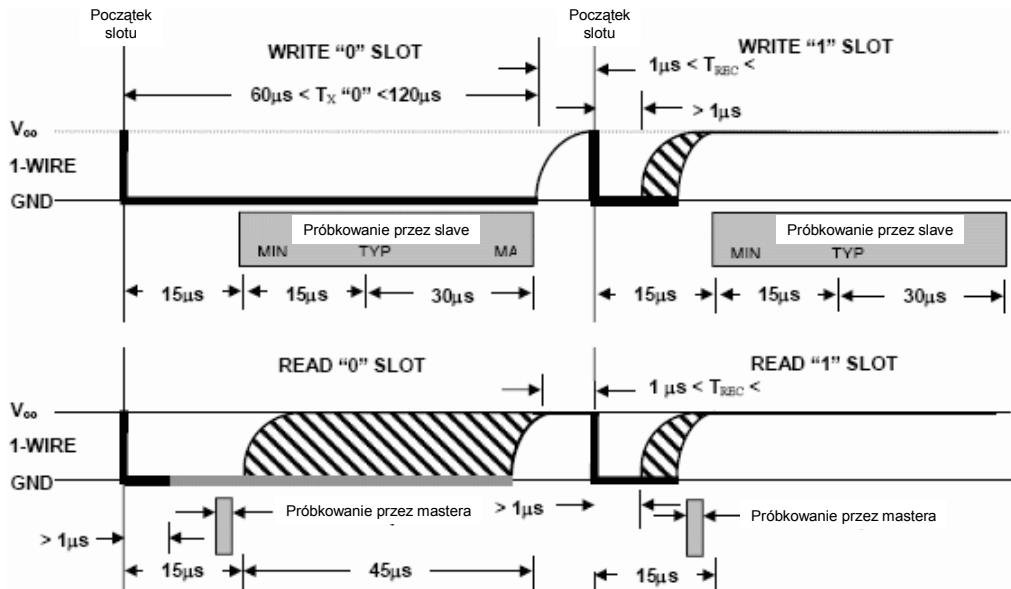
# Interfejs 1-wire

Interfejs 1-wire jest bardzo prostym interfejsem, stworzonym do łączenia urządzeń na duże odległości (nawet do kilkuset metrów), przy czym transmisja odbywa się stosunkowo wolno. Interfejs ten składa się tylko z jednego przewodu sygnałowego, po którym przesyłane są dane dwukierunkowo, oraz linii masy łączącej urządzenia. Urządzenia *slave* podłączone do tego interfejsu mogą mieć własne zasilanie lub mogą być zasilane przez linię danych (tzw. tryb pasożytniczy, ang. *parasite power*). Dzięki temu do podłączenia urządzeń *slave* wymagane są tylko dwa przewody (masa oraz przewód sygnałowy, który jednocześnie pełni rolę przewodu zasilającego). Urządzenia mogą być łączone ze sobą przy pomocy dowolnego kabla, lecz jego lepsza jakość zapewnia możliwość tworzenia bardziej rozległych sieci. W skład dostępnych urządzeń *slave* wchodzą:

- ◆ termometry cyfrowe, np. popularne DS1820, 18S20, 18B20,
- ◆ przetworniki ADC,
- ◆ eksplandery magistrali 1-wire,
- ◆ pamięci,
- ◆ układy *IO*.

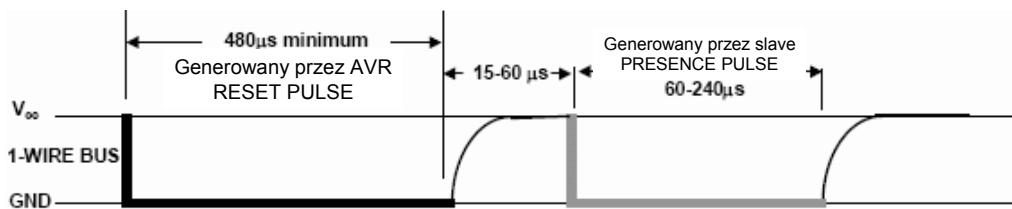
Interfejs 1-wire zakłada istnienie tylko jednego urządzenia *master* i dowolnej liczby urządzeń *slave*. Urządzenia *slave* identyfikowane są przy pomocy unikalnego 8-bajтовego identyfikatora, nadawanego urządzeniu w czasie produkcji. Wszelkie transfery na magistrali inicjowane są przez urządzenie *master*, urządzenia *slave* same nie wykazują żadnej aktywności. Wymiana danych odbywa się szeregowo, **począwszy od najmniej znaczącego bitu**. W stanie spoczynku magistrala utrzymywana jest w stanie „1” przez rezystor podciągający (typowo o wartości  $4,7\text{ k}\Omega$ ), natomiast stan „0” wymuszany jest przez urządzenie *master* lub *slave*. Stąd też jeśli jednocześnie dwa lub więcej urządzeń *slave* próbują nadawać, wygrywa to urządzenie, które nadaje wartość „0” (stan wysokiego magistrali jest stanem recesywnym). Taki tryb pracy uniemożliwia uzyskanie wysokich prędkości transmisji. Typowo dla magistrali 1-wire uzyskuje się prędkość do ok. 16 kbps, przy czym istnieją specjalne tryby (ang. *overdrive*), w których prędkość można kilkakrotnie zwiększyć (do ok. 125 kbps). Jednak nawet tak niewielka prędkość transmisji jest w zupełności wystarczająca do sprawnej komunikacji z urządzeniami

na magistrali. Schemat protokołu komunikacji pokazany został na rysunku 24.1. Każdą operację na magistrali inicjuje urządzenie *master*. Operacje dzielą się na operacje odczytu (w efekcie jeden bit przesyłany jest z urządzenia *slave* do urządzenia *master*) i operacje zapisu (urządzenie *master* przesyła jeden bit danych do urządzenia *slave*). Operacja jest inicjowana poprzez wystawienie przez urządzenie *master* na krótko (ok. 1 – 3  $\mu$ s) zera logicznego. Dla operacji zapisu po nim następuje przesłanie właściwego bitu danych. W przypadku wysłania bitu o wartości „0” magistrala utrzymywana jest w stanie niskim przez co najmniej 15  $\mu$ s do maksymalnie 120  $\mu$ s. Przy przesyłaniu bitu o wartości „1” po impulsie inicjującym magistrala wraca do stanu spoczynkowego. Kolejny bit można wysłać nie wcześniej niż po 60  $\mu$ s. Podobnie wygląda operacja odczytu. Po wymuszeniu przez urządzenie *master* na krótko niskiego stanu logicznego *master* oczekuje na bit wysłany przez urządzenie *slave*. W tym celu po minimalnie 15  $\mu$ s od zainicjowania transmisji próbkuje ono stan magistrali. Jej stan odpowiada wartości wysłanego przez urządzenie *slave* bitu. Kolejny bit może zostać wysłany po powrocie magistrali do stanu „1” lub po upływie ok. 15  $\mu$ s od zainicjowania transmisji (w sytuacji, w której *slave* wysyłał bit o wartości 1).



**Rysunek 24.1.** Schemat protokołu komunikacji na magistrali 1-wire

Każda transakcja na magistrali rozpoczyna się od zainicjowania podłączonych do niej urządzeń przy pomocy specjalnej sekwencji nazywanej RESET PULSE. Sekwencja ta jest generowana przez urządzenie *master* i polega na wymuszeniu na magistrali niskiego poziomu logicznego przez okres 480 – 960  $\mu$ s. Po tym czasie urządzenie *master* zwalnia magistralę i oczekuje na tzw. PRESENCE PULSE. Jest to ujemny impuls, generowany przez urządzenia *slave* o długości ok. 60 – 240  $\mu$ s, rozpoczynający się ok. 15 – 60  $\mu$ s po zakończeniu RESET PULSE (rysunek 24.2). Dzięki temu urządzenie *master* wie, że do magistrali przyłączone są inne urządzenia (przynajmniej jedno urządzenie). Jeśli PRESENCE PULSE nie zostanie odebrany, świadczy to o braku komunikacji z urządzeniami *slave*.



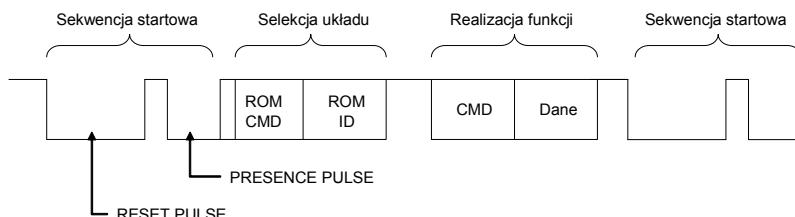
Rysunek 24.2. Nadawanie RESET PULSE i odbiór PRESENCE PULSE

Po zainicjowaniu magistrali urządzenia oczekują na polecenie. Lista poleceń obsługiwanych przez urządzenia 1-wire pokazana została w tabeli 24.1.

Tabela 24.1. Lista poleceń 1-wire obsługiwanych przez wszystkie urządzenia slave

Nazwa	Kod	Opis
SEARCH ROM	0xF0	Inicjuje proces skanowania urządzeń w celu odczytania ID wszystkich urządzeń podłączonych do magistrali.
READ ROM	0x33	Umożliwia odczytanie ID urządzenia w sytuacji, kiedy na magistrali jest tylko jedno urządzenie.
MATCH ROM	0x55	Umożliwia adresację dowolnego urządzenia na magistrali. Po MATCH ROM wysyłane jest 8-bajtowe ID wybieranego urządzenia.
SKIP ROM	0xCC	Powoduje wybranie wszystkich urządzeń na magistrali, niezależnie od ich ID. Umożliwia np. jednoczesne rozpoczęcie konwersji temperatury itp.
ALARM SEARCH	0xEC	Działa podobnie do SEARCH ROM, lecz odpowiadają wyłącznie urządzeniu z ustawioną flagą alarmu

Schemat komunikacji został pokazany na rysunku 24.3. Po inicjalizacji magistrali urządzenie *master* wysyła jedno z poleceń wybierających układ *slave* (MATCH ROM, SKIP ROM), a następnie 8-bajtowy identyfikator wybieranego urządzenia (w przypadku polecenia SKIP ROM etap ten jest pomijany). Od tego momentu wybrane urządzenie *slave* jest gotowe do komunikacji. Kolejnym etapem jest wysłanie polecenia, które *slave* ma zrealizować, oraz opcjonalnie wysłanie lub odebranie parametrów. Na tym kończy się transakcja. Realizacja kolejnej komendy rozpoczyna się od ponownej inicjalizacji magistrali.



Rysunek 24.3. Elementy komunikacji na magistrali 1-wire. Po inicjalizacji magistrali urządzenia oczekują na wybór (wysłanie polecenia z kategorii ROM), po tym wysyłane jest polecenie określające żądaną funkcję i dalej opcjonalnie wysyłane są lub odbierane parametry wywołanej funkcji

Nieco inaczej wygląda sytuacja, w której magistrala jest przeszukiwana w kierunku występujących na niej urządzeń (komenda SEARCH ROM) lub urządzeń z ustawioną flagą alarmu (komenda ALARM SEARCH).

Sekwencja znajdowania identyfikatorów urządzeń występujących na magistrali jest zdecydowanie bardziej skomplikowana. Po wysłaniu jednego z poleceń — SEARCH ROM lub ALARM SEARCH — urządzenie *master* wykonuje kolejne kroki pozwalające na odczyt kolejnych bitów składających się na ID urządzenia według schematu:

- ◆ Odczyt pierwszego bitu.
- ◆ Odczyt negacji pierwszego bitu.
- ◆ Porównanie obu odczytanych wartości.
- ◆ Wysłanie na magistralę wartości wybranego bitu.
- ◆ Urządzenie *slave* porównuje wysłany przez mastera bit z wartością bitu ID, jeśli są one różne, urządzenie *slave* wyłącza się.
- ◆ *Master* powtarza powyższe kroki 64 razy, w efekcie na magistrali zostaje wyłącznie jedno urządzenie.

W ten sposób można wyeliminować wszystkie urządzenia, z wyjątkiem ostatniego (o najwyższym ID). Na każdym etapie na magistrali urządzenie *slave* wysyła najpierw stan odczytywanego bitu, a następnie jego negację. W efekcie kolejne dwa bity odczytane przez *mastera* w sytuacji, w której wszystkie urządzenia na danej pozycji mają tę samą wartość bitu, będą zanegowane (master odczyta 01 lub 10). Jeżeli na magistrali występują urządzenia mające na danej pozycji różne wartości bitów, *master* odczyta 00. W efekcie musi przeskanować magistralę dwukrotnie, raz wybierając urządzenia posiadające na danej pozycji bit równy 1, a drugim razem 0. Dzięki temu możliwe jest odszukanie wszystkich identyfikatorów urządzeń. Cała ta procedura jest niepotrzebna w sytuacji, kiedy na magistrali występuje tylko jedno urządzenie 1-wire lub kiedy ID urządzeń są znane.

Każdy układ 1-wire ma swój unikalny numer identyfikacyjny — rysunek 24.4. Jest on nadawany na etapie produkcji układu i nie można go zmienić. Gwarantuje to, że na magistrali nie znajdują się dwa układy o takim samym identyfikatorze.

Typ pola	Długość
Kod rodziny	8-bitów
Numer seryjny	48-bitów
CRC	8-bitów

Rysunek 24.4. Struktura numeru identyfikacyjnego układów 1-wire

Kod rodziny układu to 8-bitowe pole informujące o typie zastosowanego układu. ID kończy się wartością CRC obliczoną na podstawie poprzedzających 7 bajtów, co umożliwia stwierdzenie poprawności odczytanego kodu.

## Realizacja master 1-wire na AVR

Co prawda mikrokontrolery AVR nie dysponują sprzętowo realizowanym interfejsem 1-wire, lecz jego programowa emulacja jest niezwykle prosta. Do realizacji transmisji 1-wire można wykorzystać różne interfejsy (UART, *timery*, interfejs *I/O*). Poniżej pokazane zostaną przykłady bazujące na pinach portów *I/O* oraz wykorzystujące interfejs USART. Realizację układu *master* można podzielić na dwa etapy — realizację procedur niskopoziomowego dostępu do magistrali (funkcje inicjujące wykorzystywany interfejs, inicjujące magistralę, wysyłające i odbierające bit danych) oraz funkcje wysokopoziomowe, umożliwiające realizację całych transakcji.



Przykładowe kody powyższych funkcji znajdują się w pliku *1wire.zip*.

Takie podzielenie funkcji umożliwia łatwą zmianę interfejsu sprzętowego wykorzystawanego do realizacji interfejsu 1-wire, bez konieczności wprowadzania daleko idących zmian w pozostałych funkcjach obsługi.

## Realizacja master 1-wire przy pomocy pinów IO

Jest to najprostszy wariant. Do realizacji transmisji wykorzystuje się 1 pin *I/O* procesora, odpowiednio nim sterując. Pin wykorzystany do realizacji interfejsu 1-wire musi zapewnić realizację 3 stanów magistrali:

- ◆ nadawania wartości 1,
- ◆ nadawania wartości 0,
- ◆ odczytu stanu magistrali.

Stan 1 na magistrali 1-wire jest stanem recesywnym, wymuszonym przez rezystor podciągający. Co prawda procesory AVR dysponują wewnętrznymi rezystorami podciągającymi, jednak ich duża wartość ( $20 - 50\text{ k}\Omega$ ) powoduje, że zazwyczaj nie nadają się one do tego zadania. Tak więc stan recesywny magistrali należy wymusić zewnętrznym rezystorem, o wartości typowo  $4,7\text{ k}\Omega$ .



W przypadku zasilania pasożytniczego może być wymagane zastosowanie tzw. silnego podciągania — dzięki temu urządzenia zasilane pasożytniczo będą mogły pobierać prąd z linii danych.

Takie silne podciąganie można zrealizować przy pomocy odpowiednio sterowanego tranzystora lub wymuszając wartość „1” na wyjściu portu *I/O*. Przebiegi na magistrali generuje się, odpowiednio sterując kierunkiem pinu *I/O* przy pomocy rejestru DDR. Wystawienie wartości 0 realizuje się poprzez przestawienie pinu jako wyjścia, co przy wartości 0 wpisanej na odpowiadający mu bit rejestru PORT powoduje wygenerowanie na magistrali stanu niskiego. Z kolei wysłanie wartości 1 lub odczyt magistrali realizuje się poprzez ustawienie pinu jako wejścia. W tym stanie stan magistrali wymusza zewnętrzny

rezystor podciągający. Wszystkie zależności czasowe w tym trybie trzeba realizować programowo poprzez stosowanie odpowiednich opóźnień. Tutaj z pomocą przychodzą pokazane w rozdziale 11. funkcje zdefiniowane w pliku nagłówkowym `<util/delay.h>`.

Prawidłowa komunikacja na magistrali 1-wire wymaga w miarę ścisłego przestrzegania podanych czasów trwania poszczególnych impulsów. **Ponieważ są one generowane programowo, ewentualne przerwanie w trakcie ich generacji może znacząco wpływać na czas trwania generowanego impulsu.** W efekcie może to zakłócić przebieg transmisji. Aby ten problem wyeliminować w programach wykorzystujących przerwania, sekcje krytyczne obsługi protokołu 1-wire muszą być chronione poprzez bloki instrukcji wykonywanych atomowo. Stanowi to istotne ograniczenie w realizacji programowej interfejsu 1-wire, jednak w prostszych aplikacjach nie powinno być przeszkodą.

Poniżej pokazany zostanie przykład praktycznej realizacji procedur komunikacyjnych z wykorzystaniem interfejsu 1-wire. Funkcje te zostały zdefiniowane w pliku źródłowym `Iwire_basic.c`. W pliku `IwireDefines.h` należy zdefiniować port i numer pinu *I0* wykorzystywanego do realizacji interfejsu 1-wire. Może to być dowolny pin portu dostępnego w wykorzystywanym mikrokontrolerze. Dostęp do wybranego pinu odbywać się będzie przy pomocy makrodefinicji SET, CLR, GET, omówionych szerzej w rozdziale 18.

Pierwszą funkcją jest funkcja odpowiedzialna za inicjalizację interfejsu:

```
void OW_init()
{
    CLR(DDR, OW_PIN);
    CLR(PORT, OW_PIN);
}
```

Funkcja ta zeruje bit odpowiadający wybranemu pinowi *I0* oraz ustawia jego kierunek na wejściowy. W efekcie magistrala ma stan równy 1, wymuszany przez zewnętrzny rezystor podciągający.

Ponieważ każda komunikacja 1-wire rozpoczyna się od inicjalizacji magistrali, poprzez wysłanie RESET PULSE, kolejną funkcją będzie funkcja generująca taki impuls:

```
void OW_ResetPulse()
{
    SET(DDR, OW_PIN);
    _delay_us(480);
    CLR(DDR, OW_PIN);
}
```

Wygenerowany impuls ujemny będzie miał czas trwania wynoszący 480 µs. Funkcja ta wykorzystywana jest do realizacji funkcji odpowiedzialnej za całą inicjalizację magistrali, czyli generowanie RESET PULSE, oraz odbiór PRESENCE PULSE, świadczącego o obecności na magistrali urządzeń *slave*:

```
bool OW_WaitForPresencePulse()
{
    ATOMIC_BLOCK(ATOMIC_RESTORESTATE)
    {
        OW_ResetPulse();
```

```
_delay_us(30);
unsigned char counter=0;
while((counter<0xFF) && (GET(OW_PIN)))
{
    _delay_us(1);
    counter++;
}
if(counter==0xFF)
{
    Error=OW_NoPresencePulse;
    return false;
}
counter=0;
while((counter<0xFF) && (GET(OW_PIN)==0))
{
    _delay_us(2);
    counter++;
}
if(counter==0xFF)
{
    Error=OW_BusShorted;
    return false;
}
Error=OW_OK;
return true;
}
```

Powyzsza funkcja zwraca true, jeśli na magistrali znajdują się urządzenia *slave*, lub false, jeśli PRESENCE PULSE był nieobecny lub wystąpił inny błąd (np. zwarcie magistrali do GND lub Vcc). Niestety, cała funkcja musi być wykonywana atomowo, chyba że możemy zagwarantować, że w czasie jej trwania ewentualne przerwania nie spowodują w znaczący sposób zmiany parametrów generowanych przebiegów czasowych. W efekcie przerwania zostają zablokowane na czas równy ok. 650 – 800 µs. **W wielu przypadkach nie ma to większego znaczenia, należy sobie jednak zdawać sprawę, że ewentualne przerwania występujące w tym czasie zostaną obsłużone z dużym opóźnieniem. Może to także spowodować gubienie przerwań.**

Pozostałe funkcje też są krytyczne czasowo, lecz czas ich wykonywania nie jest aż tak długi. Do realizacji komunikacji potrzebne będą jeszcze dwie niskopoziomowe funkcje, odpowiedzialne za realizację odczytu i zapisu jednego bitu z magistrali/na magistrale 1-wire. Funkcja wysyłająca jeden bit wygląda następująco:

```
void OW_SendBit(bool bit)
{
    ATOMIC_BLOCK(ATOMIC_RESTORESTATE)
    {
        SET(DDR, OW_PIN);
        _delay_us(3);
        if(bit) CLR(DDR, OW_PIN);
        _delay_us(60);
        CLR(DDR, OW_PIN);
    }
}
```

Generuje ona krótki (3 µs) impuls ujemny, po czym w zależności od wartości wysyłanego bitu zwalnia magistralę (co powoduje jej przejście w stan 1) lub wydłuża impuls ujemny o kolejne 60 µs.

Ostatnią funkcją jest funkcja odczytująca jeden bit z magistrali 1-wire:

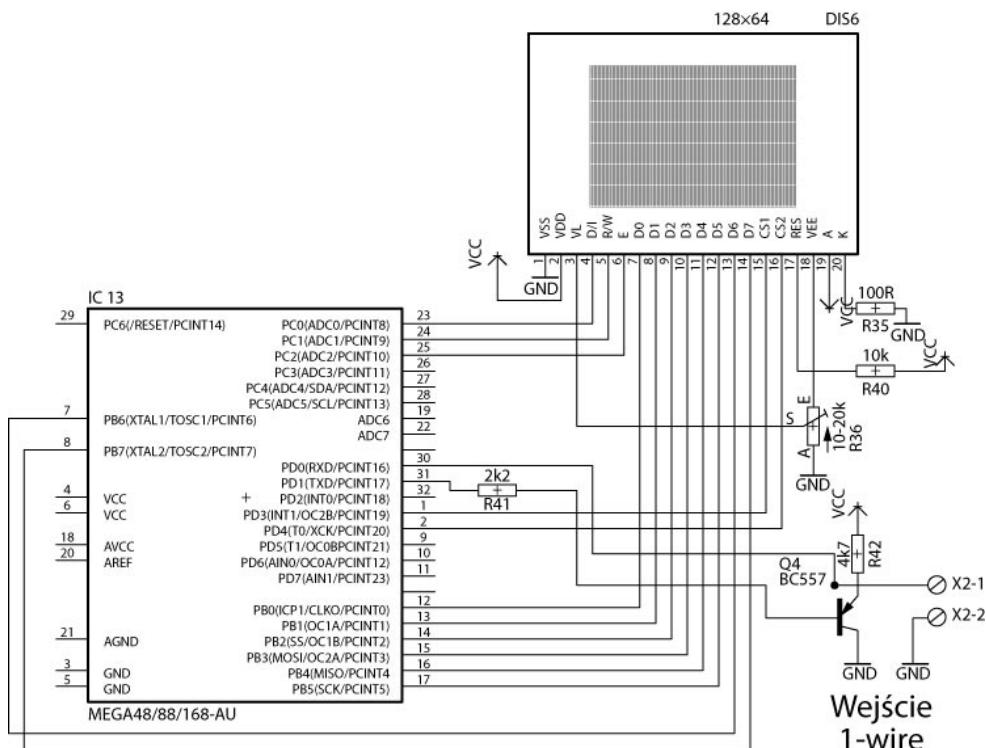
```
bool OW_ReadBit()
{
    unsigned char tmp;
    unsigned char counter=0;
    ATOMIC_BLOCK(ATOMIC_RESTORESTATE)
    {
        SET(DDR, OW_PIN);
        _delay_us(3);
        CLR(DDR, OW_PIN);
        _delay_us(15);
        tmp=GET(OW_PIN);
    }
    while((counter<0xFF) && (GET(OW_PIN)==0))
    {
        _delay_us(2);
        counter++;
    }
    if(counter==0xFF) Error=OW_BusShorted;
    return tmp;
}
```

Podobnie jak w funkcji zapisującej, działanie funkcji odczytującej rozpoczyna się od wygenerowania trwającego 3 µs impulsu ujemnego, inicjującego odczyt bitu z urządzenia *slave*. Po upływie kolejnych 15 µs stan magistrali jest próbkowany — odpowiada on wartości odczytywanego bitu. Ostatnim etapem jest oczekiwanie, aż urządzenie *slave* zwolni magistralę. Jeśli stan magistrali nie wróci do wartości 1 w czasie krótszym niż ok. 512 µs, funkcja kończy swoje działanie, jednocześnie nadając globalnej zmiennej Error wartość OW\_BusShorted. W przeciwnym przypadku funkcja zwraca wartość odczytanego z magistrali bitu.

Powyższe funkcje są wystarczające do realizacji dowolnej transmisji na magistrali 1-wire. Dla wygody warto jednak zdefiniować kilka funkcji wyższego poziomu, które zostaną pokazane w dalszej części rozdziału.

## Realizacja master 1-wire przy pomocy interfejsu USART

W wielu sytuacjach użycie pinu portu *IO* do realizacji interfejsu 1-wire jest prostym i wygodnym rozwiązaniem, lecz blokowanie procesora na czas realizacji transmisji powoduje marnotrawienie znacznej części mocy obliczeniowej. Co gorsze, ze względu na to, że obsługa 1-wire jest krytyczna czasowo, wiele funkcji obsługi musi być wykonywanych atomowo, w efekcie dochodzi do znacznego opóźnienia reakcji procesora na sygnały zewnętrzne. W wielu aplikacjach jest to niepożądane. Aby uniknąć tego typu problemów, przy pomocy paru elementów zewnętrznych można zrealizować interfejs master 1-wire w oparciu o istniejący interfejs USART lub USI (rysunek 24.5).

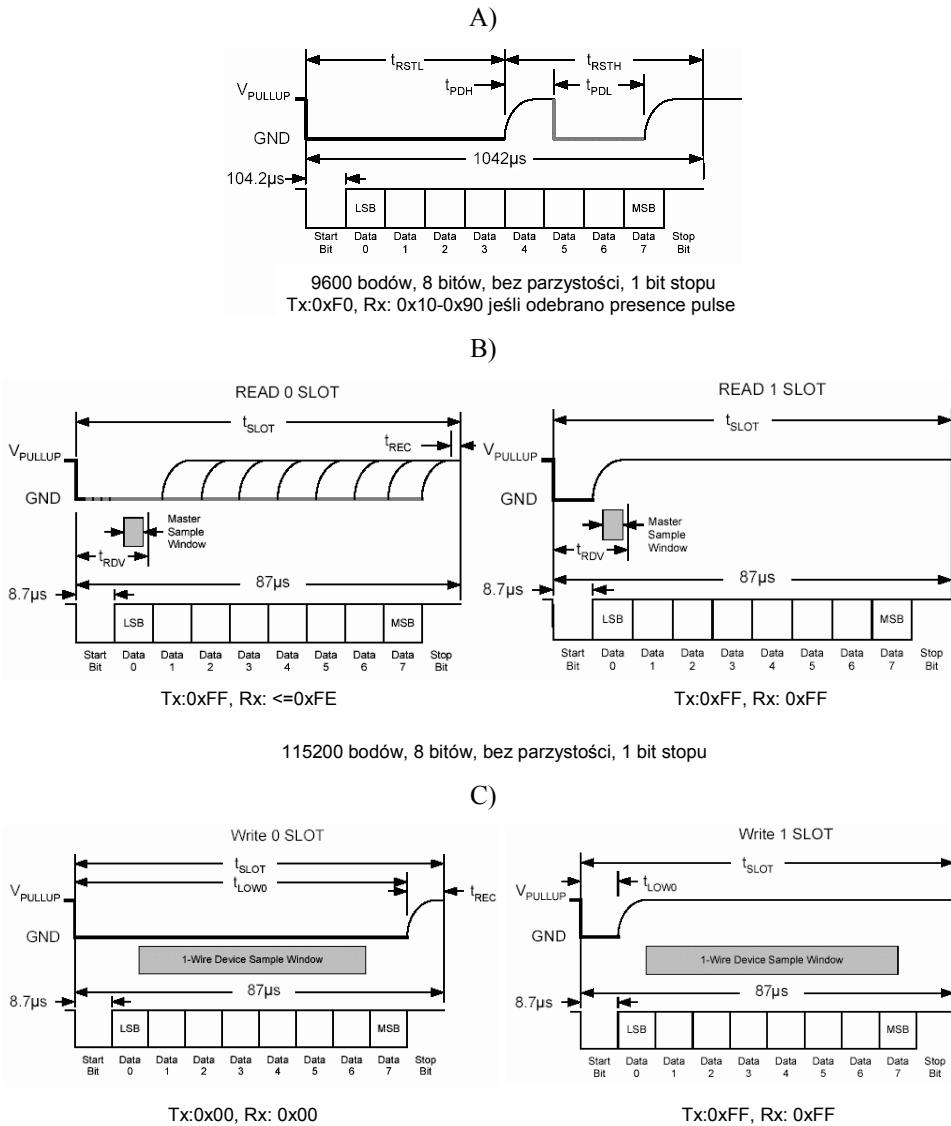


**Rysunek 24.5.** Realizacja interfejsu master 1-wire w oparciu o interfejs USART procesora ATMega88. Tranzystor Q4 umożliwia zrealizowanie wyjścia, w którym wymuszany jest tylko stan niski, stan wysoki jest stanem recesywnym magistrali, wymuszany przez rezystor podciągający R42

Interfejs USART można traktować jako zwykły 10-bitowy rejestr przesuwający, tak-towany zegarem o programowalnej prędkości. Dzięki temu zapisując dane do rejestru nadajnika USART, można generować przebiegi o kształcie określonym poprzez wartość wpisywaną do rejestru. Zewnętrzne elementy potrzebne są w celu realizacji wyjścia typu *open-collector* (*open-drain*), co umożliwia wystawienie przez procesor stanu niskiego na magistralę, przy zachowaniu recesywności stanu wysokiego.

Poprzez wpisywanie odpowiednich wartości do rejestru UDR i odczytywanie informacji zwrotnej z bufora odbiornika UART można zrealizować quasi-sprzętowo komunikację 1-wire (rysunek 24.6).

Jakkolwiek zapis do rejestru nadajnika interfejsu USART powoduje wygenerowanie bitu startu o wartości 0, co odpowiada wygenerowaniu krótkiego impulsu ujemnego na magistrali 1-wire, poprzedzającego wszystkie operacje odczytu i zapisu. Bit ten jednocześnie odbierany jest jako bit startu przez sprzążony z nadajnikiem układ odbiornika USART. Od tego momentu zaczyna on próbować magistralę 1-wire w odstępach czasowych określonych przez zegar taktujący interfejs USART. Ponieważ interfejs ten dysponuje sprzętowymi mechanizmami eliminacji zakłóceń, realizowana przy jego pomocy transmisja 1-wire jest bardziej odporna na zakłócenia. Stan magistrali można określić, odczytując zawartość rejestru odbiornika.



**Rysunek 24.6.** Realizacja interfejsu master przy pomocy portu USART. A) Generowanie RESET i PRESENCE PULSE. Po zainicjowaniu interfejsu tak, jak pokazano na rysunku, wpisanie wartości 0xF0 do rejestru UDR powoduje wygenerowanie impulsu RESET PULSE. Odpowiedź urządzenia slave (PRESECE PULSE) powoduje odebranie przez nadajnik bajtu o wartości z zakresu 0x10 – 0x90. Brak odpowiedzi powoduje odebranie bajtu o wartości 0xF0 (równego temu, co zostało nadane). B) Odczyt z magistrali. Wpisując nową wartość do rejestru nadajnika USART, generuje się krótki impuls ujemny (bit startu). Jeśli urządzenie slave nadaje 0, odebrany zostanie bajt o wartości mniejszej lub równej 0xFE. Jeśli urządzenie nadaje 1, odebrany zostanie bajt o wartości 0xFF. C) Wysłanie bitu na magistralę realizowane jest poprzez wpisanie wartości 0 lub 0xFF do rejestru nadajnika



## Wskazówka

W układach USART dysponujących buforami wejściowymi i wyjściowymi przy ich pomocy można generować jednocześnie przebiegi odpowiadające transmisji kilku bitów, co dodatkowo odciąża procesor.

Ponieważ magistrala 1-wire cechuje się stosunkowo dużą tolerancją czasów trwania poszczególnych impulsów, w tym przypadku interfejs USART może być taktowany z wewnętrznego generatora RC. Do poprawnej transmisji nie jest wymagany rezonator kwarcowy.

Przejdzmy teraz do realizacji praktycznej takiego interfejsu. Wszystkie funkcje związane z obsługą 1-wire zrealizowane zostaną identycznie jak w poprzednim przykładzie. Jedyna różnica dotyczyć będzie funkcji zdefiniowanych w pliku *1wire\_basic\_UART.c*. Definiuje on niskopoziomowe funkcje umożliwiające manipulację stanem magistrali: wysyłanie RESET PULSE, odbiór PRESENCE PULSE, nadawanie i odbieranie bitu. Funkcje wyższego poziomu pozostają bez zmian. Pierwszą potrzebną definicję funkcji jest definicja funkcji odpowiedzialnej za inicjalizację interfejsu sprzętowego:

```
void OW_init()
{
    UCSRB=_BV(RXEN0) | _BV(TXEN0); //8-bitów danych, 1 bit stopu, bez parzystości
}
```

Jak widać, składa się ona jedynie z inicjalizacji interfejsu USART. Włączenie nadajnika i odbiornika powoduje przejęcie funkcji odpowiednich pinów *I0*, dzięki czemu nie trzeba ich osobno konfigurować. Pozostałe funkcje mogą zmieniać prędkość pracy interfejsu USART, stąd zdefiniowano dwie pomocnicze funkcje, konfigurujące interfejs tak, aby pracował z prędkościami odpowiednio 9600 lub 115 200 bodów:

```
static void uart_9600()
{
#define BAUD 9600
#include <util/setbaud.h>
UBRR0H=UBRRH_VALUE;
UBRR0L=UBRRL_VALUE;
#if USE_2X
UCSROA|=BV(U2X0);
#else
UCSROA&=~(_BV(U2X0));
#endif
}

static void uart_115200()
{
#undef BAUD
#define BAUD 115200
#include <util/setbaud.h>
UBRR0H=UBRRH_VALUE;
UBRR0L=UBRRL_VALUE;
#if USE_2X
UCSROA|=BV(U2X0);
#else
UCSROA&=~(_BV(U2X0));
#endif
}
```

Do wyliczenia wartości wpisywanych do rejestru UBRR w celu uzyskania pożądanej prędkości pracy interfejsu wykorzystano makrodefinicje zawarte w pliku nagłówkowym `<util/setbaud.h>`.

Ponieważ nadawanie i odbiór danych z magistrali odbywa się całkowicie asynchronicznie w stosunku do wykonywania programu, zdefiniowano funkcję synchronizującą, której celem jest oczekiwanie na zakończenie nadawania wartości znajdującej się w rejestrze nadajnika USART:

```
static void waitforTx()
{
    while(!(UCSROA & _BV(TXCO)));
    UCSROA|= _BV(TXCO); //Skasuj flagę TXCO
}
```

Wysłanie całego bajtu sygnalizowane jest ustawieniem flagi TXCO rejestru UCSROA. Flagę tę należy skasować programowo (program nie wykorzystuje przerwań, które kasują ją automatycznie). Funkcja ta jest wykorzystywana pomocniczo w pozostałych funkcjach obsługi magistrali.

Dysponując funkcjami zapewniającymi kontrolę nad interfejsem USART, można przystąpić do zdefiniowania funkcji związanych bezpośrednio z magistralą 1-wire. Pierwszą funkcją będzie funkcja nadająca RESET PULSE:

```
void OW_ResetPulse()
{
    uart_9600();
    UDR0=0xF0;
    waitforTx(); //Poczekaj na koniec transmisji
    uart_115200();
}
```

Ponieważ RESET PULSE i PRESENCE PULSE są sygnałami o stosunkowo długim czasie trwania, układ USART przełączany jest w wolniejszy tryb 9600 bodów. Dzięki temu okres próbkowania magistrali wydłuża się do około 1000 µs, co jest czasem wystarczającym do jednoczesnego nadania sygnału RESET PULSE i odebrania sygnału PRESENCE PULSE. Po zainicjowaniu magistrali USART przełączany jest ponownie w szybszy tryb o prędkości 115 200 bodów, co umożliwia nadawanie i odbiór bitów.

Powyższą funkcję wykorzystuje funkcja sprawdzająca, czy po RESET PULSE odebrano PRESENCE PULSE:

```
bool OW_WaitForPresencePulse()
{
    OW_ResetPulse();
    if(UDR0==0xF0) return false;
    return true;
}
```

O odebraniu sygnału PRESENCE PULSE świadczy wartość rejestru odbiornika różna od 0xF0. Odebrana wartość 0xF0 świadczy o braku urządzeń na magistrali (wartość odebrana równa jest wartości nadanej).

Przedostatnią funkcją jest funkcja umożliwiająca nadawanie bitów o wartości 0 lub 1:

```
void OW_SendBit(bool bit)
{
    if(bit) UDR0=0xFF; else UDR0=0;
    waitforTx();
}
```

W przypadku wysłania bitu o wartości 1 generowany jest impuls ujemny o czasie trwania jednego bitu, czyli ok. 8,6 µs. Wysłaniu bitu o wartości 0 towarzyszy wygenerowanie impulsu ujemnego o długości 86 µs.

Ostatnią funkcją jest funkcja odbierająca jeden bit z magistrali:

```
bool OW_ReadBit()
{
    while(UCSR0A & _BV(RXC0)) UDR0; //Usuń ew. śmieci z bufora
    UDR0=0xFF;
    waitforTx();
    if(UDR0==0xFF) return 1;
    return 0;
}
```

Funkcja ta na początku „czyści” bufor odbiornika interfejsu USART, w którym mogą znajdować się znaki odebrane w wyniku działania np. funkcji wysyłającej bity. Wpisanie do rejestru nadajnika wartości 0xFF wiąże się z wygenerowaniem krótkiego impulsu ujemnego (bitu startu), zapoczątkowującego proces odczytu bitu z magistrali. W efekcie w rejestrze odbiornika powinna znaleźć się wartość 0xFF, jeśli odbierany bit miał wartość 1, lub <0xFF, jeśli odbierany bit miał wartość 0.

Jak widać, realizacja układu *master* 1-wire przy pomocy interfejsu USART jest nawet prostsza niż przy pomocy zwykłego portu *I/O*. Dodatkowo wszystkie operacje nie muszą być wykonywane atomowo, gdyż za generowanie przebiegów czasowych o ściśle określonym czasie trwania odpowiadają sprzętowe układy interfejsu USART. Daje to dodatkową przewagę, polegającą na możliwości wykonywania innych działań w trakcie nadawania lub odbioru sygnałów z interfejsu 1-wire.

## Wysokopoziomowe funkcje obsługi 1-wire

Funkcje te realizują bardziej złożone schematy transmisji danych na magistrali 1-wire, takie jak wysyłanie i odbiór całych bajtów, wybór urządzenia o podanym ID oraz przeszukiwanie magistrali w poszukiwaniu urządzeń lub urządzeń z aktywnym alarmem. Funkcje te są takie same niezależnie od sprzętowej realizacji interfejsu 1-wire. Wykorzystują one funkcje niskopoziomowe, zadeklarowane w pliku nagłówkowym *1wire\_basic.h*.

Pierwsza funkcja wysyła na magistralę 1-wire podany jako argument wywołania bajt:

```
void OW_Write(uint8_t byte)
{
    unsigned char loop;
    for(loop=0; loop<8; loop++)
    {
```

```

OW_SendBit(byte & 0x01);
byte>>=1;
_delay_us(130);
}
}

```

Kolejne bity wysyłane są przy pomocy niskopoziomowej funkcji OW\_SendBit, następnie wysyłany bajt jest przesuwany o jedną pozycję w prawo, w wyniku czego otrzymywany jest kolejny bit do wysłania. Zgodnie ze standardem 1-wire bity są wysyłane, począwszy od bitu najmniej znaczącego.

Kolejną funkcją jest funkcja odczytująca bajt z magistrali:

```

uint8_t OW_Read()
{
    unsigned char loop;
    unsigned char result=0;

    for(loop=0; loop<8; loop++)
    {
        result>>=1;
        if(OW_ReadBit()) result|=0x80;
        _delay_us(150);
    }
    return result;
}

```

Działa ona podobnie do poprzedniej, z tym że tym razem odczytywane przy pomocy funkcji OW\_ReadBit bity są kolejno „wsuwane” od lewej strony do zmiennej result. Po 8 obiegach pętli w zmiennej tej znajduje się odebrany bajt.

Kolejną funkcją jest funkcja umożliwiająca wybranie urządzenia *slave* o podanym identyfikatorze, dzięki czemu w następnej kolejności można do niego przesłać polecenie przy pomocy jednej z powyższych funkcji:

```

void OW_SelectDevice(const uint8_t *aID)
{
    if(OW_WaitForPresencePulse()==false) return;
    OW_Write(OW_MatchROM);
    uint8_t crc=0;
    for(uint8_t a=0;a<7;a++)
    {
        crc=_crc_ibutton_update(crc, aID[a]);
        OW_Write(aID[a]);
    }
    OW_Write(crc);
}

```

Funkcja OW\_SelectDevice inicjalizuje magistralę poprzez wygenerowanie RESET PULSE, następnie czeka na PRESENCE PULSE, a po jego otrzymaniu wysyła polecenie Match\_ROM i 8 bajtów identyfikatora urządzenia. Ponieważ ostatnim bajtem identyfikatora jest CRC, nie trzeba go przechowywać, gdyż jego wartość można wyliczyć na podstawie wartości 7 wcześniejszych bajtów. Do wyliczania CRC użyto jednej z funkcji udostępnianych przez AVR-libc — \_crc\_ibutton\_update. Funkcja ta oblicza CRC zgodnie

z algorytmem używanym przez urządzenia 1-wire. Dzięki temu, że CRC jest wyliczany, nie trzeba go przechowywać, co zmniejsza zapotrzebowanie na pamięć. Po wywołaniu tej funkcji urządzenie jest gotowe do odbioru kolejnych poleceń.

Jednak aby poznać identyfikatory poszczególnych urządzeń *slave*, potrzebna jest specjalna funkcja, która skanuje magistralę i zwraca kolejno identyfikatory znalezionych urządzeń. Poniższa funkcja przy każdym wywołaniu zwraca identyfikator jednego urządzenia, przy czym jeśli zwracana wartość funkcji wynosi 0, oznacza to, że na magistrali nie ma więcej urządzeń 1-wire. Kod funkcji wygląda następująco:

```
uint8_t OWI_Search(uint8_t cmd, uint8_t *aID, uint8_t deviationpos)
{
    uint8_t newPos=0;
    uint8_t bitMask=0x01;
    uint8_t bitA, bitB;

    OW_WaitForPresencePulse(); //Inicjalizacja magistrali
    OW_Write(cmd);           //Wyślij polecenie SearchROM

    for(uint8_t currentBit=1; currentBit<=64;currentBit++)
    {
        _delay_us(60);
        bitA=OW_ReadBit();      //Odczyt bitu
        _delay_us(60);
        bitB=OW_ReadBit();      //i jego dopełnienia
        _delay_us(60);

        if(bitA && bitB)
        {
            Error=OW_SearchNoResponse; //Błąd – oba bity równe 1
            return 0; //Zwróć cokolwiek dla uniknięcia ostrzeżenia kompilatora
        }

        if(bitA ^ bitB)
        {
            //Na tej pozycji wszystkie urządzenia mają bity o tej samej wartości
            if(bitA) (*aID)|=bitMask; //Ustaw odpowiedni bit ID
            else (*aID)&=~bitMask;
        }
        else //oba bity równe 0
        {
            if(currentBit==deviationpos) (*aID)|=bitMask; //Tu ostatnio wybrano 0, teraz będzie 1

            if(currentBit>deviationpos)
            {
                (*aID)&=~bitMask;
                newPos=currentBit;
            }
            else if(!(*aID & bitMask)) newPos=currentBit;
        }
    }

    OW_SendBit((*aID) & bitMask); //Wyślij wartość wybranego bitu
    bitMask<<=1;
    if (!bitMask)
    {
        bitMask=0x01;
        aID++;
    }
}
```

```

        }
    }
    return newPos;
}

```

Pierwszym argumentem wywołania jest komenda. Do wyboru mamy dwie: OW\_SearchROM i OW\_Alarm\_Search. Pierwsza powoduje przeskanowanie magistrali i zwrócić identyfikatory wszystkich znalezionych urządzeń, druga z kolei zwraca wyłącznie identyfikatory urządzeń posiadających ustawioną flagę alarmu. Parametr aID powinien wskazywać na 8-bajtowy bufor, w którym znajdzie się identyfikator urządzenia, a deviationpos to pozycja bitu, na której w poprzednim przeszukiwaniu znaleziono urządzenia różniące się wartością. Pierwsze wywołanie funkcji należy wywołać z argumentem równym 0, kolejne z wartością zwróconą w wyniku poprzedniego wywołania. Skanowanie magistrali jest kompletne, kiedy zwrócona wartość wynosi 0.

Przykładowe użycie funkcji wygląda następująco:

```

char buf[9];
uint8_t aID[8];
uint8_t ID[4][7];
uint8_t i, pos=0;

for(i=0;i<4;i++)
{
    Error=OW_OK;
    pos=OW_Search(OW_SearchROM, aID, pos);
    if(Error!=OW_OK) break;
    memcpy(&ID[i][0],aID, 7); //Skopiuj zeskanowane ID
    GLCD_goto(0,i*9);
    for(uint8_t ii=0;ii<8;ii++)
    {
        sprintf(buf,"%02X",aID[ii]);
        GLCD_putstr(buf);
    }
    if(pos==0) break;
}

```

Identyfikatory uzyskiwane w wyniku kolejnych wywołań funkcji kopowane są do tablicy ID. Ponieważ ostatni bajt to CRC, przy kopowaniu jest on pomijany (jego wartość jest wyliczana, tak jak to pokazano wcześniej). Powyższy kod kończy pracę po zeskanowaniu całej magistrali lub odczytaniu kodu maksymalnie 4 urządzeń 1-wire. W wyniku działania tej funkcji na wyświetlaczu powinny zostać wyświetlone identyfikatory znalezionych urządzeń.

## Termometr cyfrowy DS1820

Jednym z popularniejszych układów 1-wire jest termometr cyfrowy DS1820 i jego odmiany (DS18B20, DS18S20). Umożliwia on pomiar temperatury w zakresie  $-55$  do  $+125^{\circ}\text{C}$  z dokładnością sięgającą  $0,5^{\circ}\text{C}$ . Układ ten poza standardowymi poleceniami 1-wire rozpoznaje także dodatkowe polecenia pokazane w tabeli 24.2.

**Tabela 24.2.** Lista poleceń obsługiwana przez termometr cyfrowy DS1820

Nazwa	Kod	Opis
CONVERT T	0x44	Rozpoczyna proces konwersji temperatury w wybranym układzie. Proces ten w zależności od rozdzielczości trwa do 750 ms.
WRITE SCRATCHPAD	0x4E	Zapisuje wartość temperatury do rejestrów TH i TL znajdujących się w pamięci układu.
READ SCRATCHPAD	0xBE	Odczytuje zawartość pamięci podrzcznej układu.
COPY SCRATCHPAD	0x48	Przepisuje zawartość rejestrów TH i TL z pamięci podrzcznej do pamięci EEPROM układu.
RECALL E2	0xB8	Przepisuje zawartość rejestrów TH i TL z pamięci EEPROM układu do pamięci podrzcznej.
READ POWER SUPPLY	0xB4	Odczytuje sposób zasilania urządzenia <i>slave</i> . W następnej ramce odczytu urządzenia będące w trybie pasożytniczym wystawiają wartość 0.

Po wybraniu układu proces konwersji temperatury inicjuje się, wysyłając polecenie CONVERT T. Czas trwania konwersji zależy od dokładności pomiaru i może sięgać nawet 750 ms. Funkcje realizujące operacje specyficzne dla tego termometru cyfrowego znalazły się w pliku *DS18B20.c*. Pierwszą taką specjalną funkcją jest funkcja rozpoczęjąca proces konwersji:

```
void OW_Start_Conversion(uint8_t block)
{
    if(OW_WaitForPresencePulse()==0) return;
    OW_Write(OW_SkipROM);
    OW_Write(OW_CONVERT);
    if(block) while(!OW_ReadBit());
}
```

Funkcja ta wysyła polecenie rozpoczęcia konwersji jednocześnie do wszystkich termometrów znajdujących się na magistrali 1-wire. Jest to możliwe dzięki poleceniu OW\_SkipROM, które wybiera wszystkie urządzenia. Po tym można czekać ustalony czas na konwersję, można też sprawdzać na bieżąco, czy uległa ona zakończeniu. Odczyt danych z termometru, który przeprowadza proces konwersji temperatury, powoduje zwracanie 0 aż do czasu zakończenia konwersji. Funkcja OW\_Start\_Conversion przyjmuje parametr *block*. Jeśli jest on równy 0, funkcja po wysłaniu polecenia konwersji natychmiast wraca, jeśli *block* jest różny od zera, funkcja czeka na zakończenie konwersji i dopiero wtedy wraca do programu wywołującego. Dzięki temu po jej zakończeniu możliwy jest natychmiastowy odczyt temperatury.

Odczyt ten możliwy jest przy pomocy funkcji:

```
int16_t OW_GetTemperature(uint8_t *aID)
{
    OW_SelectDevice(aID);
    if(Error!=OW_OK) return 0xFFFF;

    OW_Write(OW_READ_SCRATCHPAD);
    uint16_t temp=OW_Read();
    temp=temp+(OW_Read()<<8);
```

```
if(temp & 0x8000) temp=1-(temp ^ 0xFFFF); //Konwersja kodów  
return temp;  
}
```

Funkcja `OW_GetTemperature` odczytuje 16 bitów zawierających wartość temperatury z urządzenia o podanym identyfikatorze, a następnie konwertuje je do postaci liczby ze znakiem używanej w języku C. Dzięki temu zwracana wartość jest poprawna nawet w przypadku mierzenia temperatur ujemnych. Najmłodsze 4 bity zawierają część ułamkową temperatury. Jeśli taką temperaturę chcemy poprawnie wyświetlić, potrzebna jest dodatkowa konwersja, np. taka jak w kodzie poniżej:

```
while(1)  
{  
    OW_Start_Conversion(1);  
    //_delay_ms(750);  
    for(uint8_t ii=0;ii<=i;ii++)  
    {  
        temp=OW_GetTemperature(&ID[ii][0]);  
        sprintf(buf,"%+03i.%li", temp/16, (temp & 0b1111)*625);  
        GLCD_goto(0,36+ii*9);  
        GLCD_puttext(buf);  
    }  
}
```

Konwersję liczby binarnej do łańcucha tekstowego w powyższym kodzie zapewnia funkcja `sprintf`. Najpierw konwertowana jest część całkowita temperatury. W tym celu 4 najmłodsze bity są ignorowane. W kolejnym etapie konwertowana jest liczba będąca iloczynem 4 najmłodszych bitów i liczby 625 (czyli przeskalowanej wartości  $1/2^4$ ). Dzięki temu możliwe jest wyświetlenie liczby ułamkowej bez uciekania się do kosztownych operacji na typach zmennopozycyjnych.



Wskazówka

Zauważmy, że takie dosyć zawiłe operacje konwersji wymagane są tylko w przypadku konieczności przedstawienia wartości temperatury w formacie akceptowalnym przez człowieka. Wszelkie obliczenia z wykorzystaniem temperatur zwróconych przez funkcję `OW_GetTemperature` można przeprowadzić bezpośrednio, bez jakichkolwiek konwersji.

# Rozdział 25.

# Bootloader

Większość procesorów AVR posiada możliwość programowej zmiany zawartości pamięci FLASH. Dzięki temu możemy wczytywać do pamięci mikrokontrolera nowy program za pomocą... programu znajdującego się w tej pamięci. Umożliwia to programowanie procesora z wykorzystaniem wszystkich jego interfejsów i różnych protokołów transmisji. Warunkiem to umożliwiającym jest umieszczenie w pamięci mikrokontrolera specjalnego programu, tzw. *bootloadera*. Jest on odpowiedzialny za odbiór kodu programu, jego wczytanie do pamięci FLASH, a następnie uruchomienie. Program *bootloadera* powinien być możliwie krótki, zazwyczaj jest niezależną od programu głównego aplikacją i ze względu na swoje funkcje sam rzadko podlega zmianom. Program *bootloadera* wczytywany jest do specjalnego regionu pamięci FLASH podczas programowania procesora (za pomocą interfejsów ISP, JTAG, PDI itd.). Program ten możemy sami umieścić w pamięci mikrokontrolera, a niektóre procesory AVR są sprzedawane z fabrycznie zainstalowanym *bootloaderem*. Ponosi on odpowiedzialność za wgranie reszty aplikacji do pamięci mikrokontrolera. Co prawda istnieje możliwość, aby aplikacja mogła aktualniąć się bez pośrednictwa *bootloadera*, lecz zazwyczaj lepiej jest wydzielić osobny fragment kodu, niezależny od reszty aplikacji, który przejmie tę funkcję.



Biblioteka AVR-libc zawiera mechanizmy wspierające pisanie kodu *bootloadera*. Prototypy wszystkich wykorzystanych w dalszej części rozdziału funkcji znajdują się w pliku nagłówkowym `<avr\boot.h>`.

## Pamięć NRWW i RWW

Pamięć FLASH mikrokontrolerów AVR jest podzielona na dwa regiony: **NRWW** (ang. *No Read While Write*) oraz **RWW** (ang. *Read While Write*). Pamięć RWW w trakcie programowania nie może być odczytywana. **Wynika z tego, że kod odpowiedzialny za skasowanie i reprogramowanie strony pamięci RWW musi leżeć poza tą pamięcią** — znajduje się w regionie pamięci NRWW. Jeśli z kolei programujemy stronę pamięci NRWW, to podczas jej odczytu procesor jest wstrzymywany aż do końca operacji

programowania. Granica obu obszarów jest dla danego typu procesora stała. Dla procesora ATMega128 pamięć RWW kończy się na adresie słowa 0xFFFF, a pamięć NRWW zaczyna się od słowa o adresie 0xF000.



**Kod odpowiedzialny za reprogramowanie pamięci FLASH musi znajdować się w pamięci NRWW. Dodatkowo należy zadbać, aby na czas kasowania/zapisu strony pamięci procesor nigdy nie próbował uzyskać dostępu do sekcji RWW.**

Jeśli więc wektory przerwań znajdują się w pamięci RWW (od adresu 0) to w trakcie reprogramowania strony pamięci przerwania muszą być zablokowane. Kod aplikacji mieści się w pamięci RWW, ale może także znajdować się w pamięci NRWW. Natomiast fragment lub cała pamięć NRWW może być użyta do przechowywania kodu *bootloadera*. Ten wydzielony obszar pamięci nazywany jest *bootsektorem*. Jego położenie i wielkość określana jest poprzez bity konfiguracyjne *BOOTSZ*. Przykładowe adresy początku *bootsektora*, wielkości i bitów konfiguracyjnych dla procesora ATMega128 pokazano w tabeli 25.1.

**Tabela 25.1.** Konfiguracja bootsektora

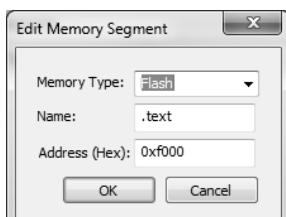
<b>Bitły konfiguracyjne</b>		<b>Adres początku (słowa)</b>	<b>Adres początku ( bajty )</b>	<b>Rozmiar ( bajty )</b>
<b>BOOTSZ1</b>	<b>BOOTSZ0</b>			
0	0	0xF000	0x1E000	0x0400
0	1	0xF800	0x1F000	0x0800
1	0	0xFC00	0x1F800	0x1000
1	1	0xFE00	0x1FC00	0x2000



Jeżeli maksymalny rozmiar *bootsektora* okaże się za mały dla aplikacji *bootloadera*, warto pamiętać, że tylko instrukcja realizująca programowanie strony pamięci (SPM) musi znajdować się w obszarze pamięci NRWW. Pozostała część kodu może znajdować się w normalnej pamięci FLASH, z tym że w takiej sytuacji tracimy możliwość ochrony *bootloadera* za pomocą tzw. *lockbitów*.

Aplikacja *bootloadera* jest normalną aplikacją, różni się tylko tym, że ma możliwość programowania pamięci FLASH, a więc zmiany kodu programu; zazwyczaj także w całości rezyduje w obszarze *bootsektora*. Ten ostatni warunek łatwo możemy spełnić, przesuwając odpowiednio adres początku segmentu *.text*. Uzyskujemy to w AVR Studio, definiując sekcję *.text* z nowym adresem początku — rysunek 25.1.

**Rysunek 25.1.**  
Redefinicja początku predefiniowanego segmentu pamięci



Można też dodać polecenie linkera do skryptu *Makefile*:

```
LDFLAGS += -Wl,--section-start=.text=0x1E000
```



Pamiętajmy, że w AVR Studio podajemy adres słowa, od którego rozpoczyna się *bootloader*, natomiast linkerowi podajemy adres bajtu —  $0x1E000 = 2*0xF000$ .

## Bity konfiguracyjne bootloadera

Pamięć FLASH, w której mieści się *bootloader*, może być niezależnie konfigurowana za pomocą bitów zabezpieczających BLB (ang. *Boot Loader Lock Bits*). Dzięki tym bitom użytkownik może:

- ◆ zablokować modyfikację całego obszaru pamięci FLASH;
- ◆ zablokować modyfikację i kasowanie tylko obszaru *bootloader*;
- ◆ zablokować modyfikację wyłącznie pamięci aplikacji;
- ◆ umożliwić modyfikację całej pamięci FLASH.

Działanie tych bitów konfiguracyjnych jest niezależne od tzw. *lockbitów* określających dostęp do pamięci FLASH z poziomu programatora. Te tzw. ogólne *lockbity* nie mają wpływu na działanie instrukcji LPM/SPM, umożliwiających dostęp do pamięci FLASH z poziomu programu.



Istnienie oddzielnych bitów konfiguracyjnych stwarza także potencjalne ryzyko wycieku zabezpieczonego kodu. Jeśli aplikację zabezpieczymy przed odczytem tylko za pomocą ogólnych *lockbitów*, to odczyt pamięci FLASH za pomocą programatora będzie niemożliwy. Jeśli jednak procesor wyposażymy w *bootloader*, a działanie instrukcji LPM z poziomu aplikacji nie będzie zablokowane, to będziemy mogli wgrać własny kod, który odczyta zawartość pamięci FLASH i wyśle ją poprzez któryś z interfejsów procesora.

Niestety, zablokowanie możliwości odczytu pamięci FLASH za pomocą instrukcji LPM praktycznie uniemożliwia programowanie w języku C. Wiąże się to z brakiem możliwości umieszczania w tej pamięci stałych programu (atrybut PROGMEM) oraz użycia jakichkolwiek zainicjowanych zmiennych. Skrypty startowe programu w języku C używają instrukcji LPM do inicjalizacji sekcji *.data*. Wynika z tego, że to na poziomie *bootloadera* musimy dokonać kontroli wgrywanego wsadu. Możemy tego dokonać poprzez np. umożliwienie wgrania tylko kodu podpisанego cyfrowo. O szyfrowaniu kodu programu i podpisach cyfrowych możesz przeczytać w rozdziale 27., „Bezpieczeństwo kodu”. Inną możliwością jest skasowanie zawartości całej pamięci FLASH mikrokontrolera (za wyjątkiem oczywiście pamięci zajętej przez kod bootloadera) przed wgraniem nowego wsadu.

Dostęp do bitów *BLB* możliwy jest z poziomu programatora oraz aplikacji, przy czym z poziomu aplikacji można odczytywać ich stan, można je także ustawać (nadawać

im stan 0). Z poziomu programu nie jest możliwe ich ponowne zerowanie (nadawanie wartości 1) — raz zablokowany dostęp można odblokować wyłącznie poprzez kasowanie całego mikrokontrolera poleciennem `chip erase`. W ten sposób odblokujemy dostęp do pamięci, ale jednocześnie jej zawartość ulegnie skasowaniu.

Opis działania bitów *BLB* w przypadku procesora ATMega128 pokazany jest w tabelach 25.2 i 25.3.

**Tabela 25.2.** Bity chroniące pamięć aplikacji

<b>BLB02</b>	<b>BLB01</b>	<b>Działanie</b>
1	1	Brak ochrony, instrukcje LPM i SPM mają swobodny dostęp do pamięci.
1	0	Nie można zapisywać pamięci aplikacji instrukcją SPM.
0	0	Nie można zapisywać do pamięci aplikacji za pomocą instrukcji SPM, instrukcja LPM wykonywana z poziomu <i>bootloadera</i> nie może odczytywać pamięci aplikacji. Jeśli tabela wektorów przerwań mieści się obszarze <i>bootloadera</i> , to w trakcie wykonywania aplikacji przerwania są blokowane.
0	1	Instrukcja LPM wykonywana z poziomu <i>bootloadera</i> nie może odczytywać zawartości pamięci aplikacji. Jeśli tabela wektorów przerwań mieści się obszarze <i>bootloadera</i> , to w trakcie wykonywania aplikacji przerwania są blokowane.

**Tabela 25.3.** Bity chroniące bootloader

<b>BLB12</b>	<b>BLB11</b>	<b>Działanie</b>
1	1	Instrukcje LPM i SPM mają swobodny dostęp do sekcji <i>bootloadera</i> .
1	0	Instrukcja SPM nie może modyfikować <i>bootloadera</i> .
0	0	Instrukcja SPM nie może modyfikować <i>bootloadera</i> , instrukcja LPM wykonywana z sekcji aplikacji nie może czytać pamięci <i>bootloadera</i> . Jeśli tabela wektorów przerwań mieści się w pamięci aplikacji, to na czas wykonywania kodu <i>bootloadera</i> przerwania są blokowane.
0	1	Instrukcja LPM wykonywana z sekcji aplikacji nie może czytać pamięci <i>bootloadera</i> . Jeśli tabela wektorów przerwań mieści się w pamięci aplikacji, to na czas wykonywania kodu <i>bootloadera</i> przerwania są blokowane.

## Konfiguracja lockbitów z poziomu aplikacji

Bitы konfiguracyjne *BLB* mogą być ustawiane z poziomu aplikacji. W tym celu twórcy biblioteki AVR-libc zdefiniowali następujące dwa makra:

```
boot_lock_bits_set (lock_bits)
```

oraz

```
boot_lock_bits_set_safe (lock_bits)
```

Oba przyjmują jako parametr nową wartość bitów konfiguracyjnych. Drugie makro dodatkowo przed ustawieniem bitów konfiguracyjnych sprawdza, czy aktualnie nie odbywa się operacja zapisu do pamięci EEPROM lub FLASH — w takim przypadku przed zapisaniem lockbitów program czeka, aż wcześniejsze zapisy zostaną zakończone.



Pamiętajmy, że funkcje te „ustawiają” wybrane bity, co w tym wypadku znaczy programują. W efekcie wartość wybranego bitu wyniesie 0.

Wskazówka

Dla przykładu, jeśli chcemy uniemożliwić instrukcji SPM zmianę kodu *bootloadera*, to bit *BLB12* powinien mieć wartość 1, a *BLB11* wartość 0. Uzyskamy to poleceniem:

```
boot_lock_bits_set_safe(_BV(BLB11));
```



Pamiętajmy, że ponowna zmiana bitu *BLB11* z 0 na 1 nie będzie możliwa z poziomu aplikacji. W tym celu musimy za pomocą programatora wykonać polecenie *chip erase*.

Wskazówka

Stan bitów *BLB* możemy odczytać za pomocą makra:

```
boot_lock_fuse_bits_get(adres)
```

Makro to może odczytywać stan różnych bitów konfiguracyjnych, stąd parametr adres określający, które bity mają zostać odczytane. W przypadku bitów *BLB* adres powinien wynosić *GET\_LOCK\_BITS* — jest to predefiniowana stała. Przy interpretacji zwróconego wyniku należy pamiętać, że bit zaprogramowany ma wartość 0, a niezaprogramowany 1.



Wskazówka

Oczywiście, przydatność powyższych makr jest raczej mała. Wymienione wyżej bity konfiguracyjne możemy zmieniać na etapie programowania mikrokontrolera, w efekcie ich późniejsza programowa konfiguracja nie jest potrzebna.

## Programowanie pamięci FLASH

Istotną częścią każdego *bootloadera* jest fragment jego kodu odpowiedzialny za programowanie pamięci. Pamięć FLASH w mikrokontrolerach AVR podzielona jest na strony o długości zależnej od typu mikrokontrolera. Przykładowo w ATMega128 strona ma długość 128 słów (256 bajtów), w ATMega88 32 słowa (64 bajty).



Wskazówka

Informację o długości strony w danym typie mikrokontrolera znajdziemy w jego nocie aplikacyjnej, w sekcji *Memory Programming/Page Size*.

Możemy także posłużyć się predefiniowaną stałą *SPM\_PAGESIZE*, zawierającą długość strony pamięci. Informacja ta jest niezwykle istotna, gdyż zaprogramować możemy wyłącznie całą stronę naraz, nie da się programować pojedynczych bajtów. Podobnie w trakcie operacji kasowania, kasowana jest zawartość całej strony pamięci.

Biblioteka AVR-libc udostępnia nam wygodne makra służące do kasowania i zapisywania stron pamięci. Makra *boot\_page\_erase(adres)* i jego pochodna *boot\_page\_erase\_safe ↴(adres)* kasują stronę pamięci określoną parametrem *adres*. Wersja z sufiksem *\_safe*, podobnie jak w przypadku makr zmieniających *lockbits*, najpierw sprawdza, czy nie odbywa się zapis do pamięci EEPROM lub FLASH. Po skasowaniu dana strona jest

gotowa do wykonania operacji zapisu. Parametr *adres* jest zmienną o typie `uint32_t`, dzięki czemu możliwe jest adresowanie pamięci w procesorach wyposażonych w więcej niż 64 kB pamięci FLASH.



Pamiętać należy, że zmienna *adres* wskazuje na adres bajtu, od którego rozpoczyna się kasowana strona, a nie adres słowa.

Podobnie, adres bajtu podajemy w kolejnych funkcjach umożliwiających załadowanie danych do bufora i ich zapis do pamięci FLASH.

Przed zapisem strony musimy wypełnić specjalny bufor danymi, które następnie zostaną zapisane. Do tego celu służą makra `boot_page_fill(adres, data)` i `boot_page_fill_safe(adres, data)`. Zmienna *adres* wskazuje na adres bajtu, od którego odbywa się zapis do bufora, kolejne dane można zapisywać od parzystych adresów, stąd adres po zapisie inkrementujemy o 2. Jednorazowo zapisywana jest dana 16-bitowa, jej młodszy bajt trafia do komórki pamięci o adresie *adres*, a starszy do komórki pamięci o adresie *adres+1*.

Po wypełnieniu bufora danymi możemy je zapisać do pamięci FLASH za pomocą makr `boot_page_write(adres)` i `boot_page_write_safe(adres)`. Zmienna *adres* zawiera adres strony, do której zostanie zapisana zawartość bufora. Strona ta musi wcześniej zostać skasowana za pomocą `boot_page_erase(adres)` lub `boot_page_erase_safe(adres)`.

Po zakończeniu programowania pamięci należy sekcję RWW odblokować. Umożliwiają to makra `boot_rww_enable()` i `boot_rww_enable_safe()`. Po odblokowaniu sekcji RWW można zakończyć program *bootloadera* i uruchomić aplikację.

W nagłówku `<avr\boot.h>` zdefiniowane są także pomocnicze makra `boot_rww_busy()`, `boot_spm_busy()` i `boot_spm_busy_wait()`. Sprawdzają one, czy procesor nie jest zajęty operacjami zapisu do pamięci FLASH. To ostatnie dodatkowo wraca dopiero w sytuacji, kiedy ostatnia operacja modyfikacji pamięci FLASH została zakończona. Jednak w większości przypadków lepiej używać wcześniej opisanych makr z sufiksem `_safe`, dzięki czemu sami nie będziemy musieli sprawdzać, czy zapis do pamięci jest możliwy. Dodatkowo makra te sprawdzają, czy równolegle nie odbywa się zapis do pamięci EEPROM, poprzez wywołanie `eeprom_busy_wait()`. Po tej porcji informacji teoretycznych spróbujmy napisać funkcję zmieniającą zawartość wybranej strony pamięci FLASH:

```
void boot_program_page(uint32_t strona, uint8_t *buf)
{
    uint16_t i;
    uint8_t sreg;

    sreg = SREG; //Zapisz stan globalnej flagi zezwolenia na przerwania
    cli();

    boot_page_erase_safe(strona);

    for (i=0; i<SPM_PAGESIZE; i+=2)
    {
        uint16_t slowo=*buf++;
        slowo+=(*buf++)<<8;
```

```
        boot_page_fill_safe(strona+i, słowo); //Zapisz dane do bufora
    }

    boot_page_write_safe(strona);      //Zapisz bufor do pamięci FLASH
    boot_rww_enable_safe();          //Odblokuj dostęp do pamięci RWW
    SREG = sreg;                    //Odtwórz stan przerwań
}
```

Powyższa funkcja jako argumenty przyjmuje adres programowanej strony (adres jej pierwszego bajtu) oraz wskaźnik do bufora zawierającego dane do zaprogramowania. Jego długość zależna jest od wielkości strony pamięci FLASH, a więc od typu procesora. Ponieważ w powyższej funkcji wykorzystano makra z sufiksem \_safe, nie zachodzi konieczność jawnego sprawdzania, czy poszczególne operacje na pamięci można przeprowadzić. Funkcja ta po zaprogramowaniu strony odblokowuje dostęp do pamięci RWW. Jeśli jednocześnie programujemy kilka stron, to dostęp do pamięci odblokować możemy dopiero po zakończeniu całej operacji programowania. Funkcja jest napisana bardzo uniwersalnie, będzie prawidłowo działać na wszystkich mikrokontrolerach rodziny AVR wyposażonych w obsługę *bootloadera*. Jednak jeśli użyty procesor ma <=64 kB pamięci FLASH, to możemy zaoszczędzić parę bajtów, deklarując zmienną strona jako zmienną 16-bitową (`uint16_t`).



Wskazówka

Pisząc *bootloader*, musimy umieścić powyższą funkcję w obszarze pamięci NRWW. Reszta kodu *bootloadera* może znajdować się w dowolnym obszarze pamięci, chociaż wskazane jest, aby również znajdująła się w obszarze *bootsektora*.

## Wykorzystanie przerwań w kodzie bootloadera

W kodzie *bootloadera*, podobnie jak w kodzie aplikacji, możemy wykorzystywać przerwania. Pamiętać jednak musimy, aby poinstruować mikrokontroler, że ma korzystać z tabeli wektorów przerwań znajdującej się w obszarze *bootloadera*. Analogicznie jak w przypadku zwykłej aplikacji wektory przerwań znajdują się na samym początku kodu. Zmiany położenia tablicy wektorów przerwań możemy dokonać za pomocą bitu *IVSEL* rejestru MCUCR. Jeśli *IVSEL* ma wartość 0, to tablica wektorów przerwań jest umieszczona w pamięci aplikacji, począwszy od adresu 0x0002. Jeśli *IVSEL* równy jest 1, to tablica zaczyna się w pamięci *bootloadera*+0x002.

Zmianę bitu *IVSEL* uzyskamy za pomocą poniższego kodu:

```
MCUCR = (1<<IVCE);
MCUCR = (1<<IVSEL);
```

Wykorzystując ten kod w języku C, należy pamiętać o włączeniu optymalizacji. Stan bitu *IVSEL* można zmienić tylko w ciągu 4 cykli od ustawienia bitu *IVCE*. Jeśli korzystamy z przerwań, to dodatkowo na czas zmiany należy je zablokować. Przypadkowe przerwanie pomiędzy zapisem *IVCE* a *IVSEL* uniemożliwiłoby zmianę *IVSEL*.

## Usuwanie tablicy wektorów przerwań

Jeśli program *bootloadera* nie wykorzystuje przerwań, to obszar tablicy wektorów przerwań może zostać wykorzystany do umieszczenia kodu aplikacji. W procesorach posiadających wiele różnych przerwań daje to istotne zmniejszenie długości kodu *bootloadera*. Niestety, AVR-libc nie oferuje tu żadnych ułatwień.

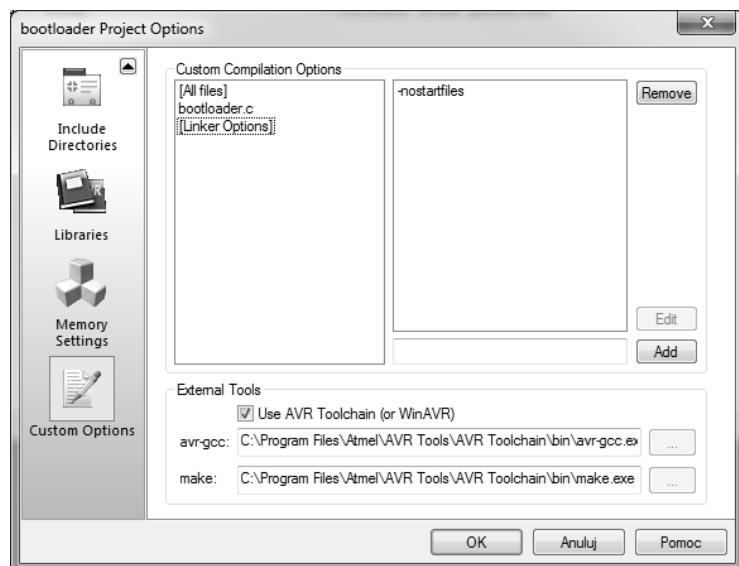
Najprostszym rozwiązaniem jest dodanie opcji linkera `-nostartfiles`. Opcję tę możemy dodać do skryptu *Makefile*:

```
LDFLAGS += -nostartfiles
```

lub w opcjach projektu w AVR Studio (*Project/Configuration Options/Custom Options/Linker Options*) — rysunek 25.2.

**Rysunek 25.2.**

Eliminacja domyślnych skryptów startowych przy pomocy opcji `-nostartfiles` przekazywanej jako parametr wywołania `gcc`



Opcja ta powoduje, że do programu nie zostaną dołączone standardowe funkcje inicjujące oraz tabela wektorów przerwań (sekcja `.vectors` pozostanie, tyle że będzie pusta). W efekcie program znaczco się skróci. Niestety, skróci się aż za bardzo — oprócz tablicy wektorów przerwań pozbędziemy się kodu inicjującego. W efekcie nie zostaną zainicjowane zmienne statyczne — nie będą one miały nadanej swojej początkowej wartości, co łamie standard języka C, ale jeśli o tym wiemy, to nie powinno to stanowić problemu. Niestety, przy okazji usunięty zostanie także kod odpowiedzialny za inicjalizację stosu oraz zerowanie rejestru R1 procesora. Wbrew pozorom jest niezwykle istotne, aby rejestr R1 miał nadaną wartość 0, gdyż reszta programu jest optymalizowana z takim założeniem. Dodatkowym problemem jest usunięcie wektora RESET, w efekcie jako pierwsza wykonywana jest przypadkowa funkcja, a nie funkcja main.



Kody poniższych przykładów znajdują się w katalogu \R25\bootloader-novectors.

Większości powyższych problemów możemy prosto zaradzić, definiując funkcję inicjującą rejestr R1 i stos:

```
void reset() __attribute__((naked,section(".vectors")));
void reset()
{
    asm("clr r1");
    SP=RAMEND;
    SREG=0;
    asm("jmp main");
}
```

Funkcja `reset()` umieszczona zostanie w segmencie `.vectors`, który znajduje się na początku segmentu `.data`, dzięki temu zostanie wykonana jako pierwsza. Wykorzystaliśmy w niej stałą `RAMEND`, wskazującą na koniec pamięci RAM procesora. Ostatnią jej instrukcją jest instrukcja skoku do funkcji `main`.



Korzystając z powyższego rozwiązania, pamiętajmy, że wartość początkowa wszystkich zmiennych będzie nieokreślona.

Jeśli jednak korzystamy ze zmiennych zainicjowanych lub statycznych, musimy nieco zmodyfikować powyższy kod:

```
void reset() __attribute__((naked,section(".vectors")));
void reset()
{
    asm("clr r1");
    SP=RAMEND;
    SREG=0;
    asm("jmp __ctors_end");
}

void jmp_main() __attribute__((naked,section(".init9")));
void jmp_main()
{
    asm("jmp main");
}
```

Funkcja `reset()` działa tak jak w poprzednim przykładzie, z tym że następuje skok do etykiety `__ctors_end`, gdzie znajdują się standardowe skrypty inicjalizacyjne odpowiedzialne za inicjalizację zmiennych i sekcji `.bss`. Jeśli w programie nie istnieją zmienne zainicjowane i statyczne, to odpowiedni kod zostanie usunięty przez linker. Dodana została także druga funkcja — `jmp_main()` — odpowiedzialna za skok do funkcji `main`. Dzięki umieszczeniu jej w sekcji `.init9` zostanie ona wykonana po standaryzowanych skryptach dodanych przez AVR-libc.

## Skrócenie tablicy wektorów przerwań

W poprzednim podpunkcie pokazano jak pozbyć się tablicy wektorów przerwań, co daje istotne oszczędności pamięci, jednak kosztem całkowej rezygnacji z przerwań. Często jednak korzystamy z przerwań, tyle że nie ze wszystkich naraz. Można więc

pokusić się o pozostawienie wybranych wektorów przerwań, a usunięcie tylko nieużywanych. Dla przykładu skompilowaliśmy program napisany dla mikrokontrolera ATMega128, w którym wykorzystane jest wyłącznie przerwanie przepelnienia *timera 1* (TIMER1\_OVF\_vect). Tablica wektorów przerwań wygląda więc następująco:

```
.func __vectors
__vectors:
    \XJMP __init
    1e000: 0c 94 46 f0  jmp 0x1e08c : 0x1e08c <__ctors_end>
    vector __vector_1
    1e004: 0c 94 5e f0  jmp 0x1e0bc : 0x1e0bc <__bad_interrupt>
    vector __vector_2
    1e008: 0c 94 5e f0  jmp 0x1e0bc : 0x1e0bc <__bad_interrupt>
    vector __vector_3
    1e00c: 0c 94 5e f0  jmp 0x1e0bc : 0x1e0bc <__bad_interrupt>
    vector __vector_4
    1e010: 0c 94 5e f0  jmp 0x1e0bc : 0x1e0bc <__bad_interrupt>
    vector __vector_5
    1e014: 0c 94 5e f0  jmp 0x1e0bc : 0x1e0bc <__bad_interrupt>
    vector __vector_6
    1e018: 0c 94 5e f0  jmp 0x1e0bc : 0x1e0bc <__bad_interrupt>
    vector __vector_7
    1e01c: 0c 94 5e f0  jmp 0x1e0bc : 0x1e0bc <__bad_interrupt>
    vector __vector_8
    1e020: 0c 94 5e f0  jmp 0x1e0bc : 0x1e0bc <__bad_interrupt>
    vector __vector_9
    1e024: 0c 94 5e f0  jmp 0x1e0bc : 0x1e0bc <__bad_interrupt>
    vector __vector_10
    1e028: 0c 94 5e f0  jmp 0x1e0bc : 0x1e0bc <__bad_interrupt>
    vector __vector_11
    1e02c: 0c 94 5e f0  jmp 0x1e0bc : 0x1e0bc <__bad_interrupt>
    vector __vector_12
    1e030: 0c 94 5e f0  jmp 0x1e0bc : 0x1e0bc <__bad_interrupt>
    vector __vector_13
    1e034: 0c 94 5e f0  jmp 0x1e0bc : 0x1e0bc <__bad_interrupt>
    vector __vector_14
    1e038: 0c 94 50 f0  jmp 0x1e0a0 : 0x1e0a0 <__vector_14>
    vector __vector_15
    1e03c: 0c 94 5e f0  jmp 0x1e0bc : 0x1e0bc <__bad_interrupt>
    vector __vector_16
    1e040: 0c 94 5e f0  jmp 0x1e0bc : 0x1e0bc <__bad_interrupt>
    vector __vector_17
    1e044: 0c 94 5e f0  jmp 0x1e0bc : 0x1e0bc <__bad_interrupt>
    vector __vector_18
    1e048: 0c 94 5e f0  jmp 0x1e0bc : 0x1e0bc <__bad_interrupt>
    vector __vector_19
    1e04c: 0c 94 5e f0  jmp 0x1e0bc : 0x1e0bc <__bad_interrupt>
    vector __vector_20
    1e050: 0c 94 5e f0  jmp 0x1e0bc : 0x1e0bc <__bad_interrupt>
    vector __vector_21
    1e054: 0c 94 5e f0  jmp 0x1e0bc : 0x1e0bc <__bad_interrupt>
    vector __vector_22
    1e058: 0c 94 5e f0  jmp 0x1e0bc : 0x1e0bc <__bad_interrupt>
    vector __vector_23
    1e05c: 0c 94 5e f0  jmp 0x1e0bc : 0x1e0bc <__bad_interrupt>
    vector __vector_24
    1e060: 0c 94 5e f0  jmp 0x1e0bc : 0x1e0bc <__bad_interrupt>
```

```
vector __vector_25
1e064: 0c 94 5e f0 jmp 0x1e0bc : 0x1e0bc <__bad_interrupt>
vector __vector_26
1e068: 0c 94 5e f0 jmp 0x1e0bc : 0x1e0bc <__bad_interrupt>
vector __vector_27
1e06c: 0c 94 5e f0 jmp 0x1e0bc : 0x1e0bc <__bad_interrupt>
vector __vector_28
1e070: 0c 94 5e f0 jmp 0x1e0bc : 0x1e0bc <__bad_interrupt>
vector __vector_29
1e074: 0c 94 5e f0 jmp 0x1e0bc : 0x1e0bc <__bad_interrupt>
vector __vector_30
1e078: 0c 94 5e f0 jmp 0x1e0bc : 0x1e0bc <__bad_interrupt>
vector __vector_31
1e07c: 0c 94 5e f0 jmp 0x1e0bc : 0x1e0bc <__bad_interrupt>
vector __vector_32
1e080: 0c 94 5e f0 jmp 0x1e0bc : 0x1e0bc <__bad_interrupt>
vector __vector_33
1e084: 0c 94 5e f0 jmp 0x1e0bc : 0x1e0bc <__bad_interrupt>
vector __vector_34
1e088: 0c 94 5e f0 jmp 0x1e0bc : 0x1e0bc <__bad_interrupt>
```

Widzimy, że większość wektorów jest nieużywana — ostatnim użytym jest wektor 14, będący wektorem przerwania TIMER1\_OVF. 20 pozostałych wektorów zajmuje łącznie aż 80 bajtów pamięci, którą możemy przydzielić aplikacji. Niestety, pamięci przeznaczonej na niewykorzystane wektory pomiędzy wektorem RESET a wektorem przerwania TIMER1\_OVF nie da się w prosty sposób wykorzystać.

## Sposób 1 — własna tabela wektorów przerwań



Kody poniższych przykładów znajdują się w katalogu \R25\bootloader-intvectortable-case1.

O ile całkowite usunięcie tablicy wektorów przerwań było proste, to jej modyfikacja jest zdecydowanie bardziej skomplikowana. Możemy pożądany cel osiągnąć, usuwając standardowe skrypty startowe (opcja linkera `-nostartfiles`) wraz z tabelą wektorów przerwań, a następnie umieścić własną. Rozwiążanie to jest proste, lecz wrażliwe na błędy. Najpierw stwórzmy własną tablicę wektorów przerwań, która zastąpi domyślną tablicę, którą usuneliśmy polecienniem linkera `-nostartfiles`:

```
_attribute__((naked,section(".vectors"))) void IntTable()
{
    asm("jmp reset");
    asm("jmp __bad_interrupt");
    asm("jmp __bad_interrupt");
}
```

```

asm("jmp __bad_interrupt");
asm("jmp __bad_interrupt");
asm("jmp __vector_14");
}

```

Nowa tablica wektorów przerwań powstaje w wyniku komplikacji powyższej funkcji. Składa się ona tylko z instrukcji asemblera realizujących skoki do odpowiednich funkcji. Ważne jest, aby jej pierwszym elementem był wektor funkcji RESET realizującej podstawową inicjalizację programu. Funkcję tę, podobnie jak w poprzednich przykładach, musimy zdefiniować sami (oryginalna została usunięta wraz z tablicą wektorów przerwań):

```

__attribute__((naked)) void reset()
{
    asm("clr r1");
    SP=RAMEND;
    SREG=0;
    asm("jmp __ctors_end");
}

```

Podobnie jak poprzednio jej zadaniem jest inicjalizacja rejestru R1 procesora, stosu oraz skok do funkcji `_ctors_end`, odpowiedzialnej za inicjalizowanie zmiennych programu (jej deklaracja znajduje się w oryginalnych skryptach startowych). Mając zapewniony prawidłowy start programu, zajmijmy się samą tablicą. Jej wszystkie niewykorzystane pozycje zastąpiliśmy instrukcją skoku do funkcji `__badinterrupt`. Jednakże jej definicja także została usunięta wraz ze skryptami startowymi, musimy więc ją jawnie ponownie zdefiniować:

```

void __bad_interrupt()
{
    cli();
    while(1);
}

```



Dla zaoszczędzenia miejsca możemy funkcję `__bad_interrupt()` pominąć, a niewykorzystane wektory zdefiniować tak, aby wskazywały np. na wektor RESET.

Na pozycji tablicy odpowiadającej wektorowi użytego przerwania musimy umieścić instrukcję skoku do zdefiniowanej przez nas procedury obsługi przerwania. W naszym przykładzie wykorzystujemy przerwanie `TIMER1_OVF`, którego wektor, jak wynika z noty katalogowej procesora ATmega128, jest 14. wektorem tablicy. Tak więc na pozycji 15. tablicy (pozycję pierwszą zajmuje wektor RESET) umieszczamy instrukcję `jmp __vector_14`, wywołującą procedurę obsługi przerwania `TIMER1_OVF`. `__vector_14` jest etykietą, która zostanie nadana procedurze obsługi przerwania w wyniku deklaracji `ISR(TIMER1_OVF_vect)`.



Procedura obsługi przerwania o numerze  $X$  będzie miała etykietę o nazwie `__vector_X`.

Powyższy sposób modyfikacji tabeli wektorów przerwań jest prosty, jednak łatwo w nim o pomyłkę. Umieszczenie instrukcji wywołującej procedurę obsługi przerwania pod innym indeksem tabeli spowoduje błędne działanie programu — a o taką pomyłkę nietrudno. Zawsze więc należy dokładnie sprawdzić, czy utworzona tabela jest prawidłowa. Dodatkowym problemem jest to, że nie wystarczy w programie zadeklarować własnej procedury obsługi przerwania — musimy jeszcze „ręcznie” poprawić tabelę, tak aby odpowiedni wektor wskazywał na nowo zdefiniowaną procedurę.

## Sposób 2 — modyfikacja skryptów startowych



Kody poniższych przykładów znajdują się w katalogu `\R25\bootloader-intvector-table-case2`.

Drugim rozwiązaniem jest modyfikacja oryginalnych skryptów startowych używanych przez AVR-libc. Ich kod źródłowy znajduje się w pliku `/avr-libc/trunk/avr-libc/crt1/crt1.S`. Niestety, plik ten nie jest dostępny w pakiecie WinAVR; należy go osobnościągnąć ze strony projektu AVR-libc (<http://svn.savannah.nongnu.org>). Oprócz niego potrzebne będą dwa pliki wymagane do komplikacji `crt1.S` — pliki `macros.inc` oraz `sectionname.h` (znajdują się one w katalogu `/avr-libc/trunk/avr-libc/common`). Wszystkie trzy pliki należy umieścić w katalogu projektu i dołączyć do wykonywanego skryptu `Makefile` (w AVR Studio wystarczy dodać do projektu plik źródłowy `crt1.S`). Aby nie były automatycznie wykorzystywane oryginalne skrypty startowe, co spowodowałoby podczas komplikacji wygenerowanie błędów informujących o wielokrotnie zdefiniowanych etykietach, należy do projektu dodać opcję `-nostartfiles`. Od tej chwili będziemy mieli pełną kontrolę nad tabelą wektorów przerwań i skryptami startowymi. Niepotrzebne wektory możemy usunąć z pliku `crt1.S`, odnajdując jego fragment:

```
_vectors:  
    _XJMP __init  
    vector __vector_1  
    vector __vector_2  
    vector __vector_3  
    vector __vector_4  
    vector __vector_5  
    vector __vector_6  
    vector __vector_7  
    vector __vector_8  
    vector __vector_9  
    vector __vector_10  
    vector __vector_11  
    vector __vector_12  
    vector __vector_13  
    vector __vector_14  
    vector __vector_15  
    vector __vector_16  
    vector __vector_17  
    vector __vector_18
```

W pliku tym zdefiniowane są wektory 1 – 127 oraz specjalny wektor `__init` (będący wektorem RESET). Wektory dołączone do programu zależą od użytego procesora. Jeśli ostatnim wykorzystywanym wektorem jest np. wektor nr 14, to pozycje 15 – 127 należy

skasować. W efekcie wektory o numerach większych niż 14 nie zostaną utworzone, a zwolniona przez nie pamięć zostanie wykorzystana przez program. Oczywiście, jeśli usuniemy wszystkie wektory, to efekt będzie taki jak w poprzednim punkcie — całkowita likwidacja tabeli wektorów przerwań.



Pamiętaj, aby nigdy nie usuwać wpisu XJMP `_init` — jest on odpowiedzialny za uruchomienie skryptów startowych, a w efekcie funkcji `main` programu.

## Start bootloadera

W większości przypadków po resecie procesor powinien zacząć wykonywać kod *bootloadera*. Takie zachowanie w przypadku, kiedy kod aplikacji jest uszkodzony, daje nam możliwość ponownego załadowania nowego kodu i naprawienia błędów. Kod aplikacji może być uszkodzony z różnych powodów, np. z powodu błędu podczas procesu uaktualniania spowodowanego zanikiem napięcia itd. Wystartowanie najpierw aplikacji i ewentualnie z niej *bootloadera* nie ma żadnej przewagi, a w przypadku nieprawidłowego funkcjonowania aplikacji nie będzie możliwości jej programowego uaktualnienia. Ponieważ kod *bootloadera* znajduje się w wydzielonym regionie pamięci FLASH, który można dodatkowo zabezpieczyć przed modyfikacją za pomocą *lockbitów*, szansa na uszkodzenie *bootloadera* jest bardzo mała. W efekcie zawsze będziemy w stanie w sposób kontrolowany uruchomić resztę programu lub rozpoczęć proces aktualizacji. Do rozwiązania pozostaje więc tylko problem, skąd *bootloader* ma wiedzieć, kiedy uruchomić aplikację. Tu z pomocą przychodzi nam kilka rozwiązań.

## Wykorzystanie dodatkowego przycisku/zworki

Jest to najprostsza metoda. W układzie istnieje specjalny przycisk/zworka, którego naciśnięcie podczas uruchamiania układu powoduje uruchomienie procesu uaktualniania oprogramowania. W przeciwnym przypadku uruchamiana jest aplikacja. Zaletą tego rozwiązania oprócz prostoty jest możliwość natychmiastowego uruchomienia aplikacji użytkownika, bez niepotrzebnej zwłoki. Wady to konieczność umieszczenia przycisku na płytce oraz przede wszystkim konieczność ingerencji użytkownika — procesu uaktualniania nie można przeprowadzić automatycznie — np. w przypadku urządzeń pracujących w sieci. Zakładając, że przycisk zwiera pin IO PA0 procesora do masy, kod realizujący uruchomienie aplikacji wyglądałby następująco:

```
int main()
{
    if (PIN_A & _BV(PA0)) asm("jmp 0000");
    //Dalsza część bootloadera
}
```

Powyższy kod należy umieścić w sekcji *bootloadera*. Będzie on wykonywany na samym początku, jeśli przycisk będzie zwolniony (pin PA0 będzie w stanie wysokim), to wyko-

nany zostanie skok pod adres 0, który jest wektorem RESET aplikacji. W przeciwnym wypadku uruchomiony zostanie *bootloader*, który zajmie się procesem aktualizacji oprogramowania.

W nowszych procesorach AVR (np. ATMega8-32U2 i innych) przedstawiony powyżej mechanizm zaimplementowano sprzętowo. Procesory te posiadają specjalny fusebit HWBE (ang. *Hardware Boot Enable Fuse*), którego zaprogramowanie zmienia funkcję pinu PD7 procesora. W trakcie resetu pin ten jest ustawiany jako wejście. Jeśli na nastającym zboczu sygnału RESET na pinie tym panuje niski stan logiczny, to procesor korzysta z wektora RESET znajdującego się w obszarze *bootloadea*. Jeśli pin PD7 jest w stanie wysokim, to procesor zaczyna wykonywanie programu od adresu 0, gdzie znajduje się wektor RESET dla aplikacji. Kiedy sygnał RESET procesora jest w stanie wysokim, przywracana jest normalna funkcja pinu PD7 — jest on w pełni pod kontrolą aplikacji.



Domyślnie w procesorach posiadających bit HWBE jest on zaprogramowany — opisany mechanizm jest więc domyślnie dostępny.

W tych procesorach zwarcie pinu HWB/PD7 do masy podczas resetu powoduje uruchomienie wbudowanego *bootloadera* obsługującego interfejs USB i w połączeniu z dostarczonym przez Atmel oprogramowaniem *Flip* (ang. *FLEXible In-system Programmer*) umożliwia zaprogramowanie procesora bez specjalnego programatora. Program *Flip* można pobrać ze strony Atmela ([http://www.atmel.com/dyn/products/tools\\_card.asp?tool\\_id=3886](http://www.atmel.com/dyn/products/tools_card.asp?tool_id=3886)). Można też w tym celu wykorzystać open sourcowe oprogramowanie *dfu-programmer* (<http://dfu-programmer.sourceforge.net/>).

## Wykorzystanie markerów w pamięci EEPROM

Metoda ta wykorzystuje komórkę pamięci EEPROM jako wskaźnik określający, czy ma być uruchomiony *bootloader*, czy aplikacja. Zaletą tej metody jest natychmiastowe uruchomienie aplikacji lub *bootloadera* i brak konieczności wykorzystania dodatkowych pinów procesora. Umożliwia także kontrolę poprawności uruchomienia aplikacji. Jej potencjalną wadą jest ograniczona liczba uruchomień urządzenia związana z ograniczoną liczbą zapisów do pamięci EEPROM. Tę wadę można zresztą wyeliminować, stosując techniki *Wear Leveling* omówione w rozdziale 7., poświeconym pamięci EEPROM, lub rezygnując z kontroli poprawności uruchomienia aplikacji.

Wykorzystując tę metodę, musimy wydzielić w pamięci EEPROM komórkę pamięci, która będzie przechowywała informację o tym, czy uruchomiony ma być *bootloader*, czy aplikacja. Ponieważ ta komórka musi być dostępna zarówno z *bootloaderem*, jak i aplikacją, musi znajdować się pod określonym adresem, znanim w obu programach. Najprościej adres ten ustalić na stałe, np. na ostatnią dostępną komórkę pamięci EEPROM (jej adres wskazuje stała E2END).



W takiej sytuacji musimy zwrócić uwagę, żeby inne zmienne umieszczone w pamięci EEPROM nie nadpisywały naszej wydzielonej komórki — kompilator nie zgłosi w takiej sytuacji żadnego ostrzeżenia.

W aplikacji głównej umieszczamy następujące fragmenty kodu:

```
#include <stdint.h>
#include <avr\EEPROM.h>

#define APP_OK    1

int main()
{
    EEPROM_write_byte((uint8_t*)E2END, APP_OK);
}
```

Zapis do EEPROM wartości APP\_OK spowoduje, że po ponownym uruchomieniu urządzenia początkowy kod *bootloadera* rozpozna, że w pamięci jest załadowany poprawny kod aplikacji, i ją uruchomi. Aplikacja, jeśli chce uruchomić proces aktualizacji, powinna do tej komórki wpisać wartość inną niż APP\_OK, dzięki czemu po restarcie *bootloader* przejdzie do aktualizacji. Odpowiedni kod *bootloadera* wygląda następująco:

```
#include <inttypes.h>
#include <avr\EEPROM.h>
#define APP_OK    1

int main()
{
    uint8_t app_status=EEPROM_read_byte((uint8_t*)E2END);
    if(app_status==APP_OK) __asm__("jmp 0000");
    //Dalsza część bootloadera i aktualizacja
    __asm__("jmp 0000"); //Uruchomienie zaktualizowanej aplikacji
}
```

Pokazane podejście ma dodatkowo tę zaletę, że w przypadku kiedy aktualizacja się nie powiedzie, komórka o adresie E2END nie będzie zawierała wartości APP\_OK, co spowoduje, że po ponownym uruchomieniu urządzenia ponownie rozpoczęcie się proces aktualizacji.

## Oczekiwanie na specjalny znak w wybranym kanale komunikacji

Kolejną możliwością rozpoczęcia procesu aktualizacji jest po uruchomieniu urządzenia oczekивание przez chwilę na odebranie specjalnej ramki danych wysłanej wybranym kanałem komunikacji (np. RS232, I2C, SPI). Jej odebranie zostanie rozpoznane jako polecenie aktualizacji kodu aplikacji. Rozwiążanie to powoduje jednak opóźnienie w uruchomieniu kodu aplikacji. Program startowy musi odczekać pewien czas, aby mieć szansę odebrać polecenie. W przypadku prostych interfejsów, takich jak SPI czy RS232, czas ten może być bardzo krótki (odpowiadający czasowi potrzebnemu na nadanie 1 – 2 znaków). W przypadku interfejsu USB jest jednak bardzo długi — program musi nie tylko czekać na ew. polecenie, ale najpierw musi zainicjalizować ten interfejs, co trwa stosunkowo długo. Z tych powodów czasami polecenie aktualizacji odbierane jest nie przez program startowy będący fragmentem *bootloadera*, lecz przez kod aplikacji. Niesie to jednak ze sobą pewne ryzyko — jeśli kod aplikacji będzie nieprawidłowy, to *bootloader* nie będzie mógł zostać uruchomiony, w efekcie niemożliwe będzie ściągnięcie poprawnego kodu.



Powyższe techniki można oczywiście wykorzystywać łącznie.

## Start aplikacji

Do tej pory aplikację znajdująjącą się w pamięci FLASH uruchamialiśmy poleceniem `jmp 0`. Powoduje ono skok pod adres 0, gdzie znajduje się wektor RESET, zawierający instrukcję skoku do początku programu. Musimy jednak pamiętać, że wykonanie skoku pod adres wektora RESET nie jest równoważne sprzętowemu resetowi procesora. Nie zapewnia to przywrócenia stanu początkowego rejestrów IO procesora oraz jego poszczególnych podsystemów i periferii. W sytuacji, w której kod aplikacji zakłada pewien stan początkowy procesora, będzie to stwarzać problemy. Mamy więc dwa rozwiązania — albo jawnie ustawić wszystkie podsystemy procesora przez specjalną procedurę inicjalizacyjną, albo, co jest prostsze, wykorzystać *Watch Dog*. Układ ten, jeśli nie zostanie zresetowany, po zaprogramowanym czasie wygeneruje sygnał RESET dla procesora, przywracając stan początkowy jego podsystemów. Możemy to osiągnąć następującą sekwencją instrukcji (prototypy funkcji związanych z *watch dogiem* znajdują się w pliku nagłówkowym `<avr\wdt.h>`):

```
cli();
wdt_enable(WDTO_15MS);
while(1);
```

Powyższy kod uruchamia *Watch Doga*, po czym czeka w nieskończonej pętli na jego zadziałanie.



Ponieważ po resecie kod zacznie się wykonywać ponownie od adresu wskazanego przez wektor RESET znajdujący się w obszarze *bootloadera*, należy pamiętać, żeby początkowe instrukcje kodu *bootloadera* (aż do ewentualnego uruchomienia aplikacji) nie ingerowały w podsystemy procesora.

## Współdzielenie kodu aplikacji i bootloadera

Czasami zarówno aplikacja, jak i *bootloader* zawierają podobne, często wręcz identyczne funkcje. Powstaje więc naturalna pokusa, aby nie dublować identycznego kodu i wydzielić takie funkcje, tak aby mogły być współdzielone przez aplikację i *bootloader*. Jeżeli nic nie stoi na przeszkodzie, aby aplikacja korzystała z funkcji *bootloadera*, to wywoływanie procedur aplikacji przez kod *bootloadera* jest bardzo dyskusyjne. Aplikacja ze swojej natury jest zmienna, więc jej kolejne wersje prawie na pewno będą miały poszczególne funkcje pod innymi adresami. Oczywiście, problem ten można ominąć. Pamiętać jednak należy, że kod aplikacji nie jest zazwyczaj tak dobrze chroniony jak kod *bootloadera*. Co więcej, podczas uaktualniania aplikacji jej części stają się

niedostępne, w efekcie zależna od nich część kodu *bootloadera* nie może działać. Zupełnie inaczej z kolei wygląda odwrotna sytuacja — wykorzystanie przez aplikację fragmentów kodu *bootloadera*. Kod *bootloadera* jest zazwyczaj niezmienny, a wykonując kod aplikacji, mamy gwarancję, że kod *bootloadera* jest dostępny. Możemy się więc do niego swobodnie odwoływać. Poszczególne funkcje *bootloadera* możemy wywoływać, korzystając z dostępnych w języku C wskaźników na funkcje. Adresy poszczególnych funkcji możemy uzyskać z pliku *map* lub *lss* i na stałe zakodować w aplikacji. Powoduje to jednak, że aplikacja staje się silnie związana z kodem *bootloadera*, a jakiekolwiek jego zmiany doprowadzą do problemów. Istnieje jednak proste rozwiązanie tego problemu — wystarczy stworzyć tablicę podobną do tablicy wektorów przerwań. Każda pozycja tablicy będzie zawierała wskaźnik do wybranej funkcji. Dzięki temu numer funkcji w tablicy możemy zamienić na jej adres. Ponieważ adres jest specyficzny dla konkretnej komplikacji *bootloadera*, najsensowniej jest umieścić tę tablicę w kodzie *bootloadera*. W ten sposób w aplikacji musimy jedynie znać adres jej początku, co z kolei wymusza umieszczenie tej tablicy pod znany i stałym adresem. Wydaje się, że najlepszym miejscem na umieszczenie tej tabeli jest obszar pamięci znajdujący się tuż za tablicą wektorów przerwań *bootloadera*. W tym celu możemy umieścić ją w sekcji *.vectors*. Nie musimy się obawiać o prawdziwe wektory przerwań — one zawsze znajdują się przed naszą tablicą.



Kody poniższych przykładów znajdują się w katalogu \R25\bootloader-shared-functions.

Przykładowy kod *bootloadera* odpowiedzialny za stworzenie takiej tablicy wygląda następująco:

```
void func1()
{
}

void func2()
{
}

void func3()
{
}

void *functiontable[] __attribute__((section(".vectors")))= {func1, func2, func3};
```

*func1 – func3* to funkcje znajdujące się w *bootloaderze*, które chcemy udostępnić aplikacji. Tablica *functiontable* zawiera wskaźniki na zdefiniowane przez nas funkcje.

Aplikacja może wywoływać takie funkcje za pomocą następującego kodu:

```
#include <stdint.h>
#include <avr\pgmspace.h>

void *RetFuncPtr(const uint8_t FuncName)
{
    return (void*)pgm_read_word_far(0x1e08c+(FuncName<<2));
}
```

W powyższym kodzie została zadeklarowana funkcja pomocnicza RetFuncPtr, której argumentem jest numer funkcji, do której wskaźnik chcemy uzyskać. Stała 0x1E08c to adres tablicy functiontable. Możemy go odczytać z pliku *map bootloadera*. Dla danego procesora będzie on stały — jego zmiana jest możliwa wyłącznie w sytuacji, w której zmieni się długość tablicy wektorów przerwań. Funkcję możemy wykorzystać w następujący sposób:

```
void (*func)();  
  
int main()  
{  
    func=RetFuncPtr(0);  
    func();  
}
```

Powyższy kod definiuje wskaźnik do funkcji (func), a następnie przypisuje mu adres funkcji znajdujący się w tablicy functiontable. Kolejna linia — func(); — wywołuje funkcję, do której pobraliśmy wskaźnik. Jeśli dana funkcja ma parametry lub zwraca jakąś wartość, to jedyne, co musimy zrobić, to zmienić deklarację wskaźnika:

```
int (*calc)(int, int);  
  
int main()  
{  
    calc=RetFuncPtr(1);  
    int x=calc(10,5);  
}
```

W powyższym przykładzie funkcja, której adres jest drugim elementem tablicy functiontable, przyjmuje dwa argumenty o typie int i zwraca wynik, również o typie int.



Wskazówka

Wywołania funkcji zdefiniowanych w *bootloaderze* z poziomu aplikacji zakładają niezmiennosć interfejsu ABI kompilatora (ang. *Application binary interface*). Należy zachować ostrożność, jeśli kod aplikacji jest komplikowany innym kompilatorem lub inną wersją kompilatora niż kod aplikacji *bootloadera*.

Pamiętać również należy, że żadne zmienne globalne nie mogą być bez specjalnych zabiegów współdzielone pomiędzy aplikacją a *bootloaderem*. W wywoływanych funkcjach powinniśmy używać wyłącznie zmiennych lokalnych, przekazywać wskaźniki lub wykorzystywać specjalne metody pokazane w dalszej części rozdziału.

Wywołania funkcji nie zadziałają poprawnie, jeśli *bootloader* znajduje się powyżej granicy 128 kB. W naszym dwubajtowym wektorze nie jesteśmy w stanie poprawnie zapisać adresów funkcji znajdujących się powyżej tej granicy.

## Wywoływanie funkcji bootloadera w procesorach ATMega256x



Kody poniższych przykładów znajdują się w katalogu \R25\bootloader- ATMega256.

Problem ten dotyczy obecnie wyłącznie procesorów z 256 kB FLASH. Jednak pokazane w dalszej części rozwiązanie jest uniwersalne i zadziała prawidłowo także na procesorach z mniejszą ilością pamięci. Jego wadą jest jednak pamięciożerność. W poprzednich przykładach każda udostępniona funkcja wydłużała stworzoną tabelę wektorów o zaledwie dwa bajty. Poniższe rozwiązanie wydłuża tabelę wektorów o 4 bajty.

Jak poprzednio w pierwszej kolejności musimy w kodzie *bootloadera* stworzyć tablice wektorów zawierającą adresy funkcji. W tym przypadku stworzymy ją jednak dosyć nietypowo. Nie będzie to typowo zdefiniowana struktura danych, lecz funkcja:

```
void jumptable() __attribute__((section(".vectors"), naked));
void jumptable()
{
    asm("jmp bfunc1");
    asm("jmp bfunc2");
}
```

Funkcja ta znajdzie się w sekcji *.vectors*, a więc tuż po tabeli wektorów przerwań. Została zadeklarowana z atrybutem naked, co powoduje, że kompilator nie doda do niej prologu i epilogu — są one nam niepotrzebne. Zawiera ona instrukcje asemblera (*jmp*) realizujące skok pod kolejne adresy udostępnionych funkcji *bootloadera* (w powyższym przykładzie funkcje *bfunc1* i *bfunc2*). Jeśli przyjrzymy się wygenerowanemu kodowi (plik *lss*):

```
void jumptable() __attribute__((section(".vectors"), naked));
void jumptable()
{
    3e0e4: 0d 94 8c f0  jmp 0x3e118 : 0x3e118 <bfunc1>
    asm("jmp bfunc1");
    asm("jmp bfunc2");
    3e0e8: 0d 94 8d f0  jmp 0x3e11a : 0x3e11a <bfunc2>
```

to zauważymy, że efektem komplikacji jest stworzenie kodu przypominającego tabelę wektorów przerwań — dokładnie to, o co nam chodziło. Powyższy kod załatwia nam sprawę *bootloadera*. Pozostaje nam tylko stworzyć kod umożliwiający wykorzystanie powyższej tabeli w aplikacji. Jest to niezwykle proste:

```
static inline void bfunc1();
void bfunc1()
{
    asm("call 0x3e0e4");
```

Stworzona właśnie funkcja *bfunc1* w kodzie aplikacji umożliwia nam wywołanie funkcji *bootloadera*. Program aplikacji wywołuje funkcję *bfunc1()*, która składa się z instrukcji wywołania kodu znajdującego się pod adresem 0x3e0e4 (odpowiada on instrukcji znajdującej się pod adresem 0x1f072). Pod tym adresem znajduje się instrukcja skoku do zdefiniowanej w kodzie *bootloadera* funkcji *bfunc1()*. Funkcja ta kończy się instrukcją powrotu z procedury (RET), w efekcie nastąpi powrót do programu aplikacji. Dzięki modyfikatorowi *inline* powyższy prosty kod będzie wstawiany w miejscu wywołania funkcji, dzięki czemu oszczędzimy sobie przy każdym wywołaniu jednej instrukcji skoku. Na przykład wywołanie:

```
int main()
{
    bfunc1();
}
```

zostanie skompilowane do następującego kodu:

```
@00000080: main
10:      {
+00000080: 940FF072     CALL      0x0001F072     Call subroutine
13:      }
```

W ten sposób koszty wywołania funkcji *bootloadera* są minimalne, w stosunku do normalnej funkcji tracimy tylko czas potrzebny na wykonanie jednej dodatkowej instrukcji `jmp`.

Nieco bardziej skomplikowane jest wywołanie funkcji, które przyjmują lub zwracają parametry. W tym przypadku nie możemy po prostu wywoływać funkcji poprzez zwykłe `call`. Kompilator definiuje:

```
static inline void bfunc1();
int bfunc1(int x)
{
    asm("call 0x3e0e4");
    return x;
}
```

co prawda skompiluje poprawnie, ale pojawi się problem na etapie optymalizacji. Optymalizator słusznie zauważa, że zmienna `x` w funkcji nie jest modyfikowana. W efekcie wywołanie:

```
Y=bfunc1(10);
```

zostanie w kodzie programu zastąpione wyliczoną stałą, w tym wypadku wartością 10, pomimo że wywoływana funkcja modyfikuje zmienną `x`. Niestety, kompilator nic o tym nie wie. Musimy więc zmienić definicję funkcji:

```
typedef int (*INT_INT_PTR)(uint8_t);

static inline int bfunc2(int x)
{
    x=((INT_INT_PTR) (0x3e0e8/2))(x);
    return x;
}
```

Zdefiniowaliśmy nowy typ (`INT_INT_PTR`), będący wskaźnikiem na funkcję przyjmującą jako argument wartość o typie `int` i zwracającą wartość o typie `int`. Podana stała `0x3e0e8` to adres wektora w stworzonej w *bootloaderze* tabeli, gdzie znajduje się właściwa instrukcja skoku do funkcji. Powyższe spowoduje wywołanie funkcji *bootloadera* z odpowiednim parametrem, a następnie zwrócenie wyniku. Teraz kompilator już wie, że `x` może w sposób nieprzewidywalny zostać zmodyfikowane i całość będzie działała pozornie prawidłowo. Niestety, ze względu na ograniczenia kompilatora `gcc` nie jest możliwe poprawne przekazanie wskaźników do funkcji znajdujących się pod adresami powyżej granicy 128 kB, a tam właśnie znajduje się nasza tabela skoków. Dzieje się tak, ponieważ do zapisania adresu słowa, od którego zaczyna się kod funkcji, w takim

przypadku potrzeba więcej niż 16 bitów, a wskaźniki w AVR-gcc są tylko 16-bitowe. Na szczęście, wygenerowany domyślnie kod jest prawie poprawny. Jedyne, czego mu brakuje, to prawidłowego ustawienia rejestru EIND, który zawiera najstarsze 8 bitów 24-bitowego adresu. To niedociągnięcie możemy poprawić ręcznie:

```
static inline int bfunc2(int x)
{
    uint8_t EINDtmp;

    asm volatile
    (
        "in %0, %1"      "\n\t"
        "ldi r30, 3"     "\n\t"
        "out %1, r30"    "\n\t"
        : "=r" (EINDtmp) : "I" (_SFR_IO_ADDR(EIND))
        : "r30"
    );

    x=((INT_INT_PTR) (0x3e0e8/2))(x);
    asm volatile
    (
        "out %1, %0"    "\n\t"
        : "r" (EINDtmp), "I" (_SFR_IO_ADDR(EIND))
    );
}

return x;
}
```

Nowa funkcja zawiera dwie krótkie wstawki asemblerowe, mające na celu prawidłowe ustawienie rejestru EIND. Pierwsza zapisuje dotychczasową wartość rejestru w tymczasowej zmiennej EINDtmp, a następnie wpisuje do niego wartość 3. Wartość ta jest nieprzypadkowa — kod *bootloadera* w mikrokontrolerach serii ATMega256x znajduje się pod adresami 0x1F000 lub wyższymi, co odpowiada adresom bajtowym 0x3E000 i wyższym. Ponieważ najmłodsze 16 bitów adresu prawidłowo generuje nam kompilator, sami musimy zadbać wyłącznie o najstarsze 8 bitów. Widzimy, że w przypadku kodu *bootloadera* najstarsze 8 bitów adresu zawsze ma wartość 3.

Druga wstawa zapewnia nam odtworzenie wcześniej zapamiętanej wartości rejestru EIND, tak aby reszta programu aplikacji mogła poprawnie pracować.



Więcej o wstawkach asemblerowych w języku C dowiesz się w rozdziale 28., „Łączenie kodu w C i asemblerze”.

Pozostaje nam jeszcze z ciekawości i dla sprawdzenia zobaczyć, jak wygląda wygenerowany kod asemblerowy naszej funkcji:

```
static inline int bfunc2(int x)
{
    100: 1f 93          push r17
    uint8_t EINDtmp;

    asm volatile
    102: 1c b7          in r17, 0x3c ; 60
```

```
104: e3 e0      ldi r30, 0x03 : 3
106: ec bf      out 0x3c, r30 : 60
      "out %1, r30"  "\n\t"
      : "=r" (EINDtmp) : "I" (_SFR_IO_ADDR(EIND))
      : "r30"
    );
      x=((INT_INT_PTR) (0x3e0e8/2))(x);
108: e4 e7      ldi r30, 0x74 : 116
10a: f0 ef      ldi r31, 0xF0 : 240
10c: 19 95      eicall
      asm volatile
10e: 1c bf      out 0x3c, r17 : 60
      "out %1, %0"  "\n\t"
      : "r" (EINDtmp), "I" (_SFR_IO_ADDR(EIND))
    );
      return x;
}
110: 1f 91      pop r17
112: 08 95      ret
```

## Wywoływanie funkcji obsługi przerwań zawartych w kodzie bootloadera

Zadanie to jest proste, gdyż adresy procedur obsługi przerwań zdefiniowane są w kodzie *bootloadera* w sekcji *.vectors*. Jedyne, co musimy znać, to adres początku *bootloadera* oraz numer przerwania, które chcemy wywołać.

Niestety, tak wywoływane procedury obsługi przerwań mogą korzystać wyłącznie ze zmiennych lokalnych, gdyż nie istnieje prosta możliwość przekazania im adresu zmiennej/struktury globalnej. Da się to jednak osiągnąć w sposób nieco bardziej skomplikowany, co zostanie pokazane w dalszej części rozdziału.

## Współdzielenie zmiennych pomiędzy aplikacją a bootloaderem

Ponieważ aplikacja i *bootloader* to dwa oddzielne programy, więc nie możemy bezpośrednio używać współdzielonych zmiennych globalnych. W obu aplikacjach ich pozycja może się różnić, w efekcie odczytywane/zapisywane będą obszary pamięci przydzielone innym zmiennym. Aby dało się współdzielić zmienne, mamy kilka możliwych rozwiązań.

### Sposób I

Najprościej jest zamiast samej zmiennej przekazywać do używającego jej fragmentu programu wskaźnik do zmiennej:

```

typedef struct
{
    uint8_t zmienna1;
    float zmienna2;
} globals;

globals zmienne; //Globalna struktura zadeklarowana w aplikacji i bootloaderze

void func1(globals *glb) //Definicja i funkcji współdzielonej
{
    glb->zmienna1=10;
    glb->zmienna2=345.8;
}

```

Powyższą funkcję z obu programów możemy wywołać w następujący sposób:

```
func1(&zmienne);
```

Dzięki temu niezależnie, w jakim miejscu pamięci znajdzie się współdzielona struktura `globals`, funkcja `func1` zawsze otrzyma jej prawidłowy adres i będzie operować na właściwych danych. Podobnie poprzez referencję możemy przekazywać dowolne typy zmiennych, niekoniecznie struktury. Jednak wygodniej jest przekazywać tylko jeden wskaźnik umożliwiający dostęp do wszystkich zmiennych globalnych zadeklarowanych w strukturze niż adres każdej z nich oddziennie.

Przekazując adres zmiennej lub wskaźnik, możemy także swobodnie korzystać z zaalokowanych wcześniej bloków pamięci funkcjami `malloc/realloc`. Nie należy jednak tych funkcji wywoływać w kodzie funkcji współdzielonej. Operują one na pewnych zmiennych globalnych, określających m.in. początek sterty. Ponieważ są to zmienne globalne, ich adresy w obu aplikacjach prawie na pewno będą różne. W efekcie operacje na stercie będą przeprowadzane w sposób nieprawidłowy.

Istnieją także inne możliwości współdzielenia zmiennych pomiędzy aplikacją a *bootloaderem*, których zaletą jest brak konieczności przekazywania referencji jako jednego z argumentów wywoływanej funkcji. W związku z tym mogą być wykorzystane także w celu dostępu do zmiennych globalnych w procedurach obsługi przerwań.

## Sposób II

Nowsze procesory ATMega posiadają rejestrę IO ogólnego przeznaczenia `GPIO` (ang. *General Purpose registers*), które mogą być dowolnie wykorzystane przez programistę (przykłady ich wykorzystania pokazano w rozdziale 13.). Są to po prostu wolne komórki leżące w przestrzeni IO. Możemy je wykorzystać do przechowywania wskaźnika do struktury zawierającej zmienne globalne. W przypadku kiedy wybrany procesor nie posiada takich rejestrów, zwykle możemy wykorzystać inne, np. rejestrów związane z niewykorzystanymi peryferiami. Zmodyfikowana `func1()` wykorzystująca rejestrę `GPIO` wygląda następująco:

```

void func1() //Definicja i funkcji współdzielonej
{
    globals *glb=(globals*)((GPIO1<<8) | GPIO0);
}

```

```
glb->zmienna1=10;  
glb->zmienna2=345.8;  
}
```

Zarówno w kodzie aplikacji, jak i *bootloadera* musimy przed wywołaniem func1() zainicjalizować rejestry GPIO, tak aby wskazywały na strukturę zmienne:

```
GPIOR0=(uint8_t)(&zmienna);  
GPIOR1=(uint16_t)(&zmienna)>>8;
```

### Sposób III

Jeżeli nie możemy skorzystać ze sposobów I i II, możemy wygospodarować miejsce na zmienne współdzielone lub wskaźnik do nich, przesuwając sekcję *.data*. Możemy to osiągnąć, przekazując linkerowi parametr *-Wl,-section-start=.data=0x800102*. Dzięki temu na początku pamięci RAM (w tym przykładzie zakładamy, że SRAM zaczyna się od adresu 0x0100) będziemy mieli do dyspozycji 2 bajty — dokładnie tyle, ile potrzebujemy, aby przechować wskaźnik do zmiennych globalnych. Dalszy sposób postępowania jest analogiczny do podanego jako sposób II. Możemy oczywiście przesunąć sekcję *.data*, tak aby zrobić miejsce na wszystkie zmienne globalne, pamiętać jednak musimy, że zmienne umieszczone w wygospodarowanym regionie pamięci nie zostaną zainicjowane — ich wartość początkowa będzie przypadkowa. Ma to też pewną zaletę — jeśli z *bootloadera* uruchomimy kod aplikacji lub odwrotnie, to wartość zmiennych umieszczonych w tym regionie nie ulegnie zmianie. Można to wykorzystać do przekazywania parametrów z aplikacji do *bootloadera* lub z *bootloadera* do aplikacji.

## Mikrokontrolery AVR z wbudowanym bootloaderem

Procesory AVR posiadające sprzętowo zrealizowany interfejs USB posiadają firmowo wgrany *bootloader*. Należą do nich m.in. procesory AT90USB1287, AT90USB1286, AT90USB647, AT90USB646, AT90USB162, AT90USB82, ATMega32u6, ATMega32u4 i ATMega16u4. Wbudowany *bootloader* umożliwia aktualizację oprogramowania za pomocą dostępnych za darmo narzędzi. W tym celu Atmel stworzył klasę urządzenia USB o nazwie **Device Firmware Uploader** (DFU). Firma udostępnia program FLIP; jest on dostępny w wersji dla systemów operacyjnych MS Windows oraz GNU/Linux. Nie jest on instalowany domyślnie wraz z AVR Studio, lecz trzeba go osobno pobrać. Alternatywą open source dla niego jest program dfu-programmer. Uruuchomienie wbudowanego *bootloadera* następuje poprzez wykonanie skoku z aplikacji pod adres *bootloadera*, zwarcie pinu HWB do masy w czasie resetu procesora lub umieszczenie wektora RESET w przestrzeni *bootloadera*. Adresy początku kodu *bootloadera* i szczegółowe informacje o nim podano w tabeli 25.4.

**Tabela 25.4.** Informacje na temat bootloadera

Procesor	Wielkość bootloadera	VID/PID	Adres startowy (adres słowa)
AT90USB1287	8 kB	0x03eb/0x2ffb	0xf000
AT90USB1286			
AT90USB647	4 kB	0x03eb/0x2ff9	0x7800
AT90USB646			
AT90USB162		0x03eb/0x2ffa	0x1800
AT90USB82		0x03eb/0x2ff7	0x0800
ATMega32U4		0x03eb/0x2ff4	0x3800
ATMega16U4		0x03eb/0x2ff3	0x0800

Dla osób, które chcieliby się więcej dowiedzieć o użytym protokole firma Atmel, stworzyła notę *USB DFU Bootloader Datasheet* do pobrania pod adresem [http://www.atmel.com/dyn/resources/prod\\_documents/doc7618.pdf](http://www.atmel.com/dyn/resources/prod_documents/doc7618.pdf).

# Rozdział 26.

# Kontrola integralności programu

Program wykonywany przez mikrokontroler znajduje się w nieulotnej pamięci FLASH. Jej trwałość jest duża, jednak w pewnych okolicznościach zawartość tej pamięci może ulec zmianie. Może się tak stać na skutek błędu w programie, w efekcie czego następuje niekontrolowany zapis do pamięci (tylko w przypadku procesorów obsługujących instrukcję SPM), lub na skutek zadziałania różnych innych czynników, jak zużycie pamięci FLASH z powodu dużej liczby zapisów, wysoka temperatura, czas. W przypadku aplikacji wyposażonych w *bootloader* problem może także wystąpić w trakcie aktualniania oprogramowania. Utrata lub zmiana zawartości pamięci jest mało prawdopodobna, jednak nie jest niemożliwa. Stąd też jeżeli szczególnie zależy nam na sprawnym działaniu urządzenia, warto pomyśleć o kontroli integralności programu. Jest to wręcz obowiązkowe w przypadku możliwości aktualizacji programu za pomocą tzw. *bootloadera*. Proces aktualniania z różnych przyczyn może przebiec nieprawidłowo, w efekcie wgrany wsad nie będzie prawidłowym programem. Warto więc przygotować się na taką ewentualność i wyposażyć program w mechanizmy umożliwiające wykrycie tej sytuacji.

## Suma kontrolna

Najprostszym sposobem kontroli integralności jest obliczenie **sumy kontrolnej**. Polega ono na zsumowaniu wartości wszystkich bajtów programu (i ewentualnie danych znajdujących się w pamięci FLASH), a następnie porównaniu uzyskanej sumy z wartością zapisaną w oryginalnym pliku zawierającym kod. Wartość ta jest tak dobrana, aby po zsumowaniu z obliczoną sumą dać w wyniku 0 lub 0xFFFF (w zależności od wybranego algorytmu). Spróbowajmy zaimplementować funkcję umożliwiającą sprawdzenie sumy kontrolnej:

```
#include <avr/io.h>
#include <avr/pgmspace.h>

uint16_t Suma __attribute__ ((section (".progmem.gcc")));
```

```

extern const char *_etext;

uint8_t checkFLASHCRC()
{
    uint16_t addr;
    uint16_t FLASHEnd=(uint16_t)&_etext;

    for(addr=0;addr<FLASHEnd;addr+=2)
        Suma+=pgm_read_word(addr);

    return (Suma==0);
}

```

Funkcja odczytuje kolejne słowa z pamięci FLASH mikrokontrolera, aż do końca pamięci zajętej przez program. Koniec ten jest wyznaczany wartością symbolu linkera `_etext`, który zdefiniowany jest w domyślnym skrypcie linkera jako wskaźnik końca segmentu `.data` (zawierającego kod i dane programu). Zmienna `Suma` została zadeklarowana w pamięci tuż po obszarze wektorów przerwań dzięki umieszczeniu jej w sekcji `.progmem.gcc`. W przypadku nieuszkodzonej zawartości pamięci FLASH suma wszystkich bajtów powinna wynosić 0 (lub 0xFFFF, jeśli taki wybraliśmy algorytm). Funkcja `checkFLASHCRC()` w takim przypadku zwraca 1. Jeśli pamięć uległa uszkodzeniu, zwracana jest wartość 0. Pozostaje nam jeszcze uzupełnić wygenerowany przez kompilator kod programu wyliczoną sumą kontrolną. W tym celu posłużymy się dostarczonym wraz z kompilatorem programem narzędziowym `srec_cat`. Służy on do manipulacji wygenerowanymi plikami `.hex`. Ma on potężne możliwości, tutaj wykorzystamy tylko jego funkcje umożliwiające wyliczenie sumy kontrolnej i umieszczenie jej w pliku `hex`:

```

srec_cat ksiazka.hex -Intel -exclude 0x08c 0x08e --l-e-checksum-neg 0x08c 2 2 -
→Output ksiazka.hex -Intel --address_length=2 --line_length=44

```

W powyższym przykładzie nakazujemy programowi `srec_cat` wczytać plik `ksiazka.hex`, który ma format Intel HEX (opcja `-Intel`), i wyliczyć sumę kontrolną (`--l-e-check →sum-neg`), która wraz z sumą bajtów programu da wartość 0. Wynik zostanie zapisany do pliku `ksiazka.hex`. Opcja `-exclude` powoduje, że będzie można nadpisać słowo zaczynające się od adresu 0x08c. Jest to ważne, gdyż pod tym adresem znajdzie się wyliczona suma kontrolna. Bez tej opcji program `srec_cat` zgłosiłby błąd związany z tym, że obszar ten jest już wykorzystywany. Adres 0x8c nie jest przypadkowy. Pod nim zostanie umieszczona zdefiniowana w programie zmienna `Suma`. Adres ten możemy znaleźć w wygenerowanym pliku `map`:

<pre> 0x00000000 *(.vectors) *(.progmem.gcc*) .progmem.gcc 0x0000008c      0x2 ksiazka.o                 0x0000008c          Suma </pre>	<code>_vectors</code>
--	-----------------------

Widzimy, że w sekcji `.progmem.gcc`, począwszy od adresu 0x8c, znajduje się pamięć przydzielona zmiennej `Suma`. Adres zmiennej będzie zależny od typu procesora, ale dla danego typu (w tym przypadku ATMega128) będzie stały — będzie to ważne w kolejnych przykładach.

Jak widzimy, posługiwanie się sumą kontrolną jest niezwykle proste. Niestety, ma ona wady, np. wstawienie do programu bajtów o wartości 0 nie zmienia sumy. Błąd taki nie może więc zostać wykryty. Podobnie, jeśli wartość jednego bajtu zwiększymy o np. 1, a innego zmniejszymy o 1, to suma również się nie zmieni. Z tego powodu wymyślono bardziej wyrafinowane metody kontroli integralności danych.

## CRC



Kody poniższych przykładów są dostępne w katalogu R26\CRC.

Przy okazji zademonstrowania lepszych metod kontroli integralności danych trochę skomplikujemy program sprawdzający pamięć. Przedstawiona wcześniej funkcja jest prosta i świetnie się spisuje, jeśli wywołujemy ją ze sprawdzanego programu (aplikacji). Nie nadaje się natomiast do wykorzystania przez *bootloader* w celu sprawdzenia integralności ładowanej aplikacji. Dzieje się tak, ponieważ *bootloader* nie wie, ile pamięci zajmują aplikacja i jej dane. Możemy oczywiście założyć, że cała pamięć jest zajęta, lecz jest to postępowanie nieefektywne w przypadku procesorów posiadających dużo pamięci FLASH. Przedstawiony powyżej program ma jeszcze jedną wadę — nie działa prawidłowo w przypadku procesorów posiadających >64 kB pamięci FLASH. Powodem jest tu niemożność odczytania przez funkcję `pgm_read_word` komórek pamięci FLASH o adresach >65 535 (argument funkcji jest tylko 16-bitowy).

Często stosowaną metodą sprawdzania integralności danych jest **CRC** (ang. *Cyclic Redundancy Code*). Aplikacja (lub *bootloader*) po uruchomieniu w pierwszym etapie sprawdza CRC. Jeśli CRC jest poprawne, to realizowana jest dalsza część aplikacji, jeśli nie, to sygnalizujemy błąd np. poprzez miganie diodą, a następnie przechodzimy do funkcji *bootloadera*, która jest odpowiedzialna za wgranie nowej aplikacji.



CRC nie mówi nam, które bajty programu mają nieprawidłową wartość.

Może się tak zdarzyć, że uszkodzony będzie kod odpowiedzialny za sprawdzenie CRC. Niestety, nie da się w prosty sposób zrealizować 100-procentowo skutecznych zabezpieczeń. W przypadku aplikacji pozabawionej *bootloadera* po wykryciu nieprawidłowego CRC jedyne, co możemy zrobić, to zasygnalizować ten fakt użytkownikowi. Naprawa polegała będzie w takiej sytuacji na ponownym zaprogramowaniu procesora. O wiele większe możliwości mamy w przypadku, kiedy w pamięci mikrokontrolera znajduje się aplikacja i *bootloader* umożliwiający jej aktualizację. W tej sytuacji osobnymi kodami CRC powinniśmy zabezpieczyć zarówno aplikację, jak i *bootloader*. W przypadku uszkodzenia samej aplikacji wywoływany zostaje *bootloader* umożliwiający wgranie nowego kodu. W przypadku uszkodzenia CRC sekcji *bootloadera* problem jest bardziej skomplikowany. Pozostaje tylko ponowne programowanie procesora. Jednak uszkodzenie sekcji *bootloadera* jest ekstremalnie mało prawdopodobne. Procesory AVR

posiadają tzw. *lockbit*, umożliwiające zabezpieczenie przed programowaniem i skasowaniem tej sekcji pamięci. Dzięki temu program zawarty w tym obszarze jest bardzo dobrze chroniony.



Zamiast CRC możemy, oczywiście, użyć dowolnej innej funkcji, np. MD5 czy SHA. Jednak ich implementacja jest stosunkowo pamięciochłonna.

Poniżej zostanie pokazany przykład oparty na CRC z dwóch powodów:

- ◆ Funkcje obliczające CRC są już zaimplementowane w bibliotece AVR-libc (plik nagłówkowy `<util/crc16.h>`).
- ◆ Dostarczony w pakiecie program `srec_cat` potrafi obliczać CRC, nie potrafi obliczać np. haszy.

Istnienie wiele implementacji funkcji obliczających CRC, każda ma swoje wady i zalety. W dalszej części będziemy bazować na algorytmie obliczania CRC **XMODEM**. Jest on zaimplementowany zarówno w programie `srec_cat`, jak i w bibliotece AVR-libc (nagłówek `<util/crc16.h>`), nie musimy więc tworzyć własnego kodu.

Problem testowania CRC aplikacji możemy rozbić na dwa osobne problemy:

1. Sprawdzenie, czy CRC aplikacji jest poprawny.
2. Umieszczenie w kodzie przeznaczonym do zaprogramowania wyliczonego CRC.

Najpierw stworzymy uniwersalną funkcję obliczającą CRC w zadanym fragmencie pamięci FLASH. Argumentem funkcji jest adres początku i końca sprawdzanego bloku. Adres końca umożliwia wyliczanie CRC dla określonego fragmentu kodu, dzięki czemu nie musimy uwzględniać niewykorzystanych przez program fragmentów pamięci FLASH. Tą samą funkcję będzie można wykorzystać zarówno do sprawdzenia CRC programu, jak i *bootloadera*. Podobnie jak w przypadku wyliczania sumy kontrolnej, również ta funkcja zwraca 1, jeśli CRC jest zgodne, i 0 w przypadku niezgodności CRC:

```
#include <avr/io.h>
#include <util/crc16.h>
#include <avr/pgmspace.h>

uint8_t checkFLASHCRC(uint32_t start, uint32_t koniec)
{
    uint32_t addr;
    uint8_t bajt;
    uint16_t crc_u16=0;

    for(addr=0;addr<koniec;addr++)
    {
        bajt=pgm_read_byte_far(addr);
        crc_u16=_crc_xmodem_update(crc_u16, bajt);
    };

    if(crc_u16==pgm_read_word(addr)) return 1; else return 0;
}
```

Powysza funkcja przy okazji została pozbawiona wad poprzedniej. Przede wszystkim, dzięki zastosowaniu 32-bitowych zmiennych przechowujących adres komórki pamięci oraz wykorzystaniu funkcji umożliwiających dostęp do całej pamięci FLASH niezależnie od jej wielkości (`pgm_read_byte_far`) funkcja ta działa poprawnie na wszystkich procesorach AVR, niezależnie od wielkości ich pamięci FLASH. Ceną, jaką za to płacimy, jest nieco dłuższy wygenerowany kod i nieco wolniejsze działanie. Wywołując funkcję `checkFLASHCRC`, musimy znać adres początku zabezpieczonego bloku oraz jego długość. Adres początku jest znany — dla aplikacji wynosi `0x0000`, dla *bootloadera* jest zależny od ustawień *fusebitów BOOTSZ* określających wielkość obszaru *bootsektora*. Gorzej sprawa wygląda z adresem końca. W poprzednim przykładzie został wykorzystany fakt, że standardowe skrypty linkera definiują symbol `_etext`, zawierający adres końca segmentu `.data` — a więc adres końca kodu i danych aplikacji w pamięci. Problem polega na tym, że symbol ten znany jest aplikacji, nie jest znany poza nią. Stąd też np. program *bootloader*, który zazwyczaj jest oddzielną aplikacją, nie może sprawdzić CRC aplikacji, bo nie wie, gdzie kończy się zajęty przez nią obszar pamięci. Oczywiście, można wyznaczyć CRC dla całej pamięci, ale podobnie jak w poprzednim przykładzie byłoby to rozwiązań wysoce nieefektywne. Z pomocą przychodzi nam ponownie program `srec_cat`. Wśród jego licznych możliwości jest także możliwość uzupełniania pliku `.hex` informacją o długości znajdujących się w nich danych. Jedyne, co musimy zrobić, to przygotować miejsce, w którym ta informacja się znajdzie. Adres tego miejsca musi być stały, tak aby niezależnie od samej aplikacji *bootloader* miał dostęp do informacji o jej długości. Pamiętamy, że świetnie nadającym się do tego celu miejscem jest sekcja `.progmem.gcc`, znajdująca się tuż za tablicą wektorów przerwań. **W danym typie procesora znajduje się więc ona zawsze w tym samym miejscu.** Stąd też do programu, którego CRC chcemy sprawdzić, wystarczy dodać:

```
const uint32_t AppLen __attribute__ ((section ("._progmem.gcc")));
```

Od tego momentu zmienna `AppLen` będzie dostępna zarówno w aplikacji, która ją deklaruje, jak i poza nią (dzięki znajomości adresu tej zmiennej w pamięci), np. w kodzie *bootloader*. Pozostaje nam tylko uzupełnić zmienną `AppLen` informacją o długości obszaru zajętej pamięci FLASH. Adres, pod którym znajduje się zmienna `AppLen`, podobnie jak poprzednio znajdziemy w wygenerowanym pliku *map*. Pod ten adres zapisujemy długość zajętego obszaru:

```
srec_cat ksiazka.hex -Intel -exclude 0x08c 0x090 -Little_Endian_Maximum 0x08c -fill
→0xff -over ksiazka.hex -Intel -Output ksiazka.len.hex -Intel
```

W wyniku powyższego polecenia pod adresami `0x08C-0x08F` znajdzie się długość programu, dodatkowo polecenie `-fill` spowoduje wypełnienie wszystkich luk wartością `0xFF`. Jest to istotne w przypadku, kiedy w pamięci FLASH zostałyby zdefiniowane dodatkowe segmenty, nieleżące bezpośrednio za segmentem `.data`. W takiej sytuacji CRC wyliczone dla pliku `.hex` różniłoby się od CRC wyliczonego przez aplikację (rysunek 26.1).

W wyniku powyższego polecenia utworzony zostanie nowy plik `ksiazka.len.hex` zawierający poprawione dane. Pozostaje nam obliczyć CRC i umieścić je w wynikowym pliku `.hex`, tak aby dostępne było z poziomu aplikacji. Najwygodniej jest dołączyć wyliczone CRC do końca aplikacji:

**Rysunek 26.1.**  
Wypełnienie pustych  
miejsc za pomocą  
opcji *-fill* programu  
*srec\_cat*



```
srec_cat ksiazka.len.hex -Intel -Little_Endian_CRC16 -Maximum-Address
  ↵ksiazka.len.hex -Intel -Cyclic_Redundancy_Check_16_XMODEM -Output ksiazka.crc.hex
  ↵-Intel --address_length=2 --line_length=44
```

Powyższe polecenie dołączy do końca bloku danych sumę kontrolną zgodną ze standardem XMODEM. Parametry `address_length` i `line_length` służą tylko do sformatowania wyjściowego pliku `.hex`.

W celu sprawdzenia integralności aplikacji funkcję `checkFLASHCRC` wywołujemy z programu *bootloadera* w następujący sposób:

```
checkFLASHCRC(0, pgm_read_dword(0x008c));
```

Adresem początku aplikacji jest 0, adres końca przechowywany jest w zmiennej `AppLen`. Zauważmy, że w przypadku korzystania z *bootloadera* wywołanie funkcji `checkFLASHCRC` należy umieścić w kodzie *bootloadera*, a nie samej aplikacji. Dzięki temu *bootloader* po stwierdzeniu niezgodności CRC będzie mógł zasygnalizować ten problem i przejść w stan oczekiwania na nowy wsad.

## Automatyczne generowanie CRC

Proces modyfikacji wynikowego pliku HEX zawierającego kod aplikacji i stałe możemy zautomatyzować, dodając odpowiednie polecenia do skryptu *Makefile*. Niestety, AVR Studio nie umożliwia dodawania własnych sekcji do automatycznie tworzonego skryptu, musimy więc stworzyć własny skrypt lub też zmodyfikować skrypt tworzony przez AVR Studio poprzez dodanie do sekcji generującej wynikowy plik `.hex` (%.hex) następującego fragmentu:

```
srec_cat $@ -Intel -exclude 0x08c 0x090 -Little_Endian_Maximum 0x08c -fill 0xff -
→over $@ -Intel -Output $@.len -Intel
srec_cat $@.len -Intel -Little_Endian_CRC16 -Maximum-Address $@.len -Intel -
→Cyclic_Redundancy_Check_16_XMODEM -Output $@.crc -Intel --address_length=2 --
→line_length=44
```

Spowoduje to utworzenie pliku z rozszerzeniem *hex.crc*, który zawiera kod aplikacji z uzupełnioną informacją o jej długości (zmienna AppLen) oraz dodanym na końcu CRC.

Zobaczmy, jak wygląda szablon *bootloadera* sprawdzający CRC:

```
#include <avr/io.h>
#include <util/crc16.h>
#include <avr/pgmspace.h>

const uint32_t AppLen __attribute__ ((section (".progmem.gcc")));

#define GET_FAR_ADDRESS(var) \
({ \
    uint_farptr_t tmp; \
    __asm__ __volatile__( \
        "ldi %A0, lo8(%1)" \
        "ldi %B0, hi8(%1)" \
        "ldi %C0, hh8(%1)" \
        "clr %D0" \
        : \
        "=d" (tmp) \
        : \
        "p"  (&(var)) \
    ); \
    tmp; \
})

uint8_t checkFLASHCRC(uint32_t start, uint32_t koniec)
{
    uint32_t addr;
    uint8_t bajt;
    uint16_t crc_u16=0;

    for(addr=start;addr<koniec;addr++)
    {
        bajt=pgm_read_byte_far(addr);
        crc_u16=_crc_xmodem_update(crc_u16, bajt);
    };

    if(crc_u16==pgm_read_word_far(addr)) return 1; else return 0;
}

void BootLoaderCRCError()
{
    //CRC bootloadera jest nieprawidłowe
    while(1);
}

void UploadNewAppCode()
{
    //Tu wczytujemy nowy kod aplikacji
```

```
}

int main()
{
    if(checkFLASHCRC(0x1e000,pgm_read_dword_far(GET_FAR_ADDRESS(AppLen)))==0)
        ↳BootLoaderCRCError();
    if(checkFLASHCRC(0, pgm_read_dword(0x008c))==0) UploadNewAppCode();
    //Dalsza część bootloadera, start właściwej aplikacji
}
```

Powyższy kod po uruchomieniu sprawdza najpierw własne CRC. W tym celu zadeklarowano w nim zmienną AppLen, o identycznej funkcji jak w programie aplikacji — przechowuje ona długość programu *bootloadera*. Zakładamy, że jest on załadowany do pamięci, począwszy od adresu 0x1E000. W tym celu przeniesiono sekcję *.data*; należy też pamiętać o odpowiednim skonfigurowaniu bitów odpowiedzialnych za rozmiar obszaru *bootloadera* (bity *BOOTSZ*). Zauważmy nieco inny sposób pobierania wartości ze zmiennej AppLen. Wykorzystano w tym celu funkcję *pgm\_read\_dword\_far* — zmienna AppLen może leżeć (i w naszym przypadku leży) ponad granicą 64 kB, w związku z tym jest poza zasięgiem funkcji *pgm\_read\_dword*. Drugą rzeczą jest inny sposób pobierania adresu zmiennej AppLen. Zamiast operatora pobierania adresu znanego z języka C (&) użyto makra *GET\_FAR\_ADDRESS*, którego definicja znajduje się także w powyższym programie. Jest to spowodowane tym, że operator & zwraca 16-bitowy adres, podczas gdy w ATMega128 do zaadresowania wszystkich komórek pamięci potrzebujemy 17 bitów (128 kB). Stąd też konieczność zdefiniowania własnej funkcji zwracającej adres argumentu, który może leżeć także ponad granicą 64 kB. Po sprawdzeniu kodu *bootloadera* sprawdzane jest CRC aplikacji — aplikacja zaczyna się od adresu 0, a jej długość przechowywana jest w obszarze 0x008C – 0x008F (zmienna AppLen zadeklarowana w kodzie aplikacji). Jeśli CRC się zgadza, to następuje skok do aplikacji i jej uruchomienie.

## Rozdział 27.

# Bezpieczeństwo kodu

Pisząc aplikację na mikrokontroler, zwykle nie chcemy się dzielić efektami swojej ciężkiej pracy z innymi. Staramy się więc zabezpieczyć przed „wyciekiem” kodu. Istnieje kilka możliwości zabezpieczenia kodu. Najprostsza to zastosowanie dostępnych w procesorach AVR *lockbitów* — ich zaprogramowanie uniemożliwia odczytanie zawartości pamięci FLASH przy pomocy programatora. Złamanie tego zabezpieczenia jest niewiele bardziej skomplikowane niż jego założenie. Bez trudu można znaleźć w Internecie firmy, które „w celach edukacyjnych” łańią powyższe zabezpieczenie i dostarczają klientowi zawartość pamięci FLASH mikrokontrolera. Cena za taką usługę to kilkaset do ok. 2 tys. \$, więc czytając resztę tego rozdziału, trzeba mieć na uwadze, że mniej więcej tyle są warte przedstawione zabezpieczenia. Oczywiście, firma Atmel oferuje specjalną serię układów o podwyższonym bezpieczeństwie, jednak nie są one dostępne dla hobbystów. Ich cena też raczej odbiega od cen zwykłych układów. Nie powinno nas to jednak zrażać. Zaniedbując kwestie bezpieczeństwa, możemy jeszcze bardziej obniżyć cenę wycieku kodu wynikowego pisanej aplikacji. Z drugiej strony, przedstawione w dalszej części rozdziału metody zabezpieczeń praktycznie nic nie kosztują, są też wygodne w użyciu, stąd też warto je stosować.

## Metody łamania zabezpieczeń

Metody łamania zabezpieczeń mikrokontrolerów można podzielić na inwazyjne i nie-inwazyjne. Metody inwazyjne polegają na odsłonięciu struktury układu scalonego, a następnie za pomocą mikrosond odczytywaniu danych bezpośrednio ze struktury układu. Czasami zadanie jest prostsze, odsłonięcie krzemu umożliwia ponowne przeprogramowanie bitów zabezpieczających, co umożliwia dostęp do pamięci programu w sposób klasyczny, za pomocą programatora. Ten typ ataku teoretycznie może być przeprowadzony przez każdego, ponieważ bity zabezpieczające są realizowane jako komórki pamięci EEPROM/FLASH; w celu ich rozprogramowania wystarczy nawiątać je promieniowaniem UV. Jeśli jednocześnie zasłoni się pozostałą część układu, można w ten sposób selektywnie zmodyfikować same bity zabezpieczające, bez skasowania pamięci FLASH. Na ten typ ataku podatnych jest wiele układów, a że jest on dla wyspecjalizowanych firm niezwykle prosty technicznie, więc ceny takiej usługi

nie są wygórowane. W nowszych układach producenci zabezpieczają te bity dodatkowymi osłonami, co utrudnia ich zmianę w warunkach amatorskich, nie stanowi jednak istotnej przeszkody dla profesjonalistów — ciągle można je przeprogramować, stosując odpowiednio skupione światło, np. laserowe, lub selektywnie trawiąc metalową osłonę fusebitów. Tak naprawdę, zabezpieczenie *fusebitów* ochronną warstwą metalu ułatwia złamanie zabezpieczeń układu — po odsłonięciu struktury krzemowej od razu wiadomo, w którym jej miejscu znajdują się bity zabezpieczające.

Ciekawsze i łatwiejsze do realizacji przez amatora są techniki nieinwazyjne. Technika *eavesdropping* polega na monitorowaniu z wysoką rozdzielcością sygnałów analogowych, np. poboru prądu przez procesor. Umożliwia to określenie, jakie sekwencje instrukcji są wykonywane, gdyż każda z nich, powodując przełączenie innych bramek logicznych, generuje swój „odcisk palca” w postaci zakłóceń, które obserwuje się na liniach zasilających i innych pinach procesora. Innym podejściem jest celowe generowanie sygnałów zakłócających pracę procesora — gwałtownych skoków napięcia, szpilek na sygnałach zegarowych lub portach IO, co prowadzi do nieprawidłowej pracy rdzenia, czasami umożliwiając dostęp do chronionych danych. Ostatnią klasą są metody wykorzystujące ataki na oprogramowanie — wykorzystują one luki i błędy oprogramowania w celu uzyskania nieautoryzowanego dostępu. Są one szczególnie istotne, gdyż w przeciwieństwie do wcześniejszych technik ich skuteczność zależy wyłącznie od programisty piszącego aplikację i jego dbałości o poprawność kodu. Nie powinniśmy tej techniki ignorować, gdyż nawet najbezpieczniejszy procesor, z ustawionymi prawidłowo wszystkimi sprzętowymi zabezpieczeniami, staje się łatwym obiektem ataku, jeśli napisany na niego program zawiera istotne luki w zabezpieczeniach. Oprócz możliwości „wycieku” programu z pamięci FLASH mikrokontrolera problemem może być także uaktualnianie aplikacji. Proces uaktualniania oprogramowania możemy, oczywiście, powierzyć zaufanym pracownikom, jednak nie jest to wygodne i podnosi koszty. Z drugiej strony, dając klientowi nowy kod, umożliwiamy mu jego deasembację, tracimy także kontrolę nad tym, co użytkownik z tym kodem zrobi.

## Bezpieczne uaktualnianie aplikacji

Dostarczając klientowi aktualizację aplikacji, chcielibyśmy zrobić to jak najtańszym kosztem, łatwo, szybko, a jednocześnie bezpiecznie. To znaczy tak, aby niemożliwa była deasembacja i poznanie jej kodu. Często też sprzedajemy licencję na aplikację, nie chcemy więc, aby klient mógł uaktualnić wszystkie posiadane przez siebie urządzenia, kupując tylko jedną licencję. W jaki sposób to osiągnąć? Jedną z możliwości jest zaszyfrowanie kodu aktualizacji. W efekcie, jeśli użytkownik nie ma dostępu do klucza deszyfrującego, posiadanie zaszyfrowanego kodu nic mu nie da. Klucz deszyfrujący musi jednak znajdować się w urządzeniu — możemy zapisać go w sekcji *bootloadera*, która jest zabezpieczona przed odczytem. W efekcie użytkownik może przeprowadzić proces aktualizacji za pomocą posiadanego pliku, kod *bootloadera* zdeszyfruje dane przy pomocy zaszytego weń klucza, a następnie zaprogramuje pamięć FLASH. Ponieważ użytkownik na żadnym etapie nie ma dostępu do klucza, nie może więc przeprowadzić tego procesu poza mikrokontrolerem. Szyfrowanie kodu aktualizacji/aplikacji daje nam dodatkowo możliwość ograniczenia liczby urządzeń, do których

kod może zostać wgrany. Wyobraźmy sobie, że każde urządzenie ma unikalny klucz — w takiej sytuacji kod aktualizacji musi być zaszyfrowany indywidualnym kluczem danego urządzenia. Zastosowanie pliku aktualizacji w urządzeniu z innym kluczem nic nie da — dane nie będą mogły być prawidłowo odszyfrowane. Niestety, czyni to sam proces aktualizacji nieco uciążliwym: musimy mieć tyle przygotowanych plików aktualizacyjnych, ile różnych urządzeń chcemy aktualizować. Wadę tę łatwo możemy jednak wyeliminować. Plik aktualizacji oprócz samego kodu aplikacji może także zawierać inne dane, np. dane określające numery seryjne urządzeń, które mogą być aktualizowane za pomocą posiadanego pliku. *Bootloader* odszyfruje te dane, uzyskując numery seryjne, po czym poprzez porównanie z własnym numerem seryjnym umożliwia przeprowadzenie procesu aktualizacji lub zignoruje dalsze dane. Dzięki temu klient może kupić licencje na aktualizację niektórych posiadanych urządzeń, a programista ma pewność, że plik nie będzie mógł być wykorzystany do aktualizacji urządzeń spoza listy. Użytkownik sam nie będzie mógł zmodyfikować listy, gdyż jest ona zaszyfrowana.

Szyfrując kod aplikacji i pisząc bezpieczny *bootloader*, musimy mieć na uwadze kilka spraw. Przede wszystkim należy pamiętać, że w przypadku większości szyfrów takie same dane wejściowe produkują powstawanie takiego samego ciągu wyjściowego. Dlatego jeśli znamy przybliżoną sekwencję oryginalnych danych, łatwiej możemy złamać szyfr. Program pisany na mikrokontroler AVR ma kilka takich stałych sekwencji, które z dużym prawdopodobieństwem możemy przewidzieć. Na początku każdego programu znajduje się tabela wektorów przerwań, o raczej przewidywalnej budowie, poza tym zwykle program nie wykorzystuje całej dostępnej pamięci — na jego końcu znajdują się bajty o wartości 0xFF. To potencjalne niebezpieczeństwo możemy zmniejszyć poprzez wstawienie losowych rekordów, które będą ignorowane przez *bootloader* podczas dekodowania aplikacji. Same rekordy programu też możemy losowo przemieszczać — każdy opatrzony jest adresem, pod który ma trafić, nie będzie więc problemu z ich prawidłowym umieszczeniem w pamięci. Drugi potencjalny problem leży w samej aplikacji *bootloadera* — powinna ona zwracać minimalną ilość informacji. Jeśli np. prawidłowość ramki za każdym razem będzie potwierdzana, *cracker* będzie miał nieco ułatwione zadanie. Jeśli *bootloader* potwierdzi tylko odbiór ramki, nie dając informacji o tym, czy jest ona prawidłowa czy nie, jedyną możliwością sprawdzenia poprawności programowania będzie mozolne przeprowadzanie kolejnych czasochłonnych prób.

**Wskazówka**

Musimy także pamiętać, że bezpieczeństwo całości zależy od najsłabszego ognia. Stąd też musimy zadbać nie tylko o właściwe bezpieczeństwo samej aplikacji, ale także o prawidłową konfigurację sprzętową urządzenia.

## Nota AVR231 — AES Bootloader

Firma Atmel udostępniła gotowy *bootloader* dla swoich procesorów AVR, z zaimplementowaną możliwością szyfrowania z wykorzystaniem symetrycznego szyfra blokowego AES. Szczegóły tego rozwiązania można znaleźć w nocy aplikacyjnej AVR231:AES Bootloader. Program ten potrafi obsługiwać pliki szyfrowane szyfrem AES o długości klucza 128, 192 lub 256 bitów, dodatkowo w celu utrudnienia analizy

stosowana jest metoda łańcuchów bloku szyfru (ang. *Cipher Block Chaining*). Analizę zaszyfrowanej aplikacji dodatkowo możemy utrudnić, dodając losowe rekordy, które zostają odrzucone po zdekodowaniu przez program *bootloadera*. Wczytywaną aplikację można automatycznie zabezpieczyć, wstawiając CRC. Dzięki temu *bootloader* nie uruchomi aplikacji o błędny CRC, co umożliwia m.in. detekcję sytuacji, w której proces aktualniania zakończył się niepowodzeniem — np. w wyniku awarii zasilania.

Oryginalny kod tego *bootloadera* jest przeznaczony dla kompilatora IAR. Na szczęście, są dostępne wersje dla innych kompilatorów, m.in. gcc. Kod przerobionego *bootloadera* dla gcc można pobrać ze strony AVR Freaks (<http://www.avrfreaks.net/index.php?module=Freaks%20Files&func=viewFile&id=3330&showinfo=1>). Niestety, dostępny kod jest dosyć stary i niekompatybilny z nowszymi wersjami AVR-libc. Jednak po drobnych zmianach można go stosunkowo łatwo zaadaptować do nowej wersji biblioteki. Wraz z kodem *bootloadera* dla procesorów ATMega8-ATMega128 dostarczone są narzędzia umożliwiające konfigurację *bootloadera*, generację klucza oraz zaszyfrowanie i przygotowanie do przesłania do mikrokontrolera kodu aplikacji.



Pliki poniższych przykładów znajdują się w katalogu R27\AVR231-AESBL-gcc.

Aby móc bezpiecznie przesyłać do mikrokontrolera kod aplikacji, musimy:

1. Wygenerować klucze wykorzystywane do szyfrowania i utworzyć pliki konfiguracyjne *bootloadera*.
2. Skompilować *bootloader* i wczytać go do procesora.
3. Zaszyfrować aplikację.
4. Przesłać zaszyfrowany kod do mikrokontrolera.

Mimo że koniecznych do wykonania kroków jest sporo, tak naprawdę cały proces jest prosty i można go zautomatyzować. Pierwszym etapem jest utworzenie pliku konfiguracyjnego, na podstawie którego automatycznie zostaną wygenerowane pliki konfiguracyjne *bootloadera*. W tym celu wykorzystamy program gentemp, podając jako parametr nazwę pliku, który będzie zawierał konfigurację:

```
C:\AVR231-AESBL-gcc\pctools>gentemp conf.h  
gentemp v1.0AES - Copyright (C) 2005 Atmel Corporation  
conf.h generated.
```

Utworzony został plik conf.h, zawierający klucze (są one losowe):

```
PAGE_SIZE      = [FILL IN: Target AVR page size in bytes]  
MEM_SIZE       = [FILL IN: Application Section size in bytes]  
CRC_ENABLE     = [FILL IN: YES/NO]  
KEY1           = 264A291ACCE179846E257CF35A5F3FA3D6AB  
KEY2           = B7F9200081BB069B14  
KEY3           = 22992B2DA0B1F1C750  
INITIAL_VECTOR = A78D870307888966E31D8A6426E1DDDD  
SIGNATURE      = CE259A41
```

W pliku tym musimy ustawić parę parametrów:

- ◆ PAGE\_SIZE — musi zawierać długość strony (w bajtach) w użytym procesorze.
- ◆ MEM\_SIZE — długość aplikacji (w bajtach). W praktyce jest to różnica pomiędzy ilością dostępnej pamięci FLASH w mikrokontrolerze a długością zarezerwowanego *bootsektora*.
- ◆ CRC\_ENABLE — musi zawierać YES lub NO — określa, czy do programu ma być dołączone CRC. Zaleca się ustawić ten parametr na YES.

Wartości KEY1, KEY2 i KEY3 są opcjonalne. Jeśli pozostawimy je puste, to szyfrowanie AES będzie wyłączone. Jednak *bootloader* będzie prawidłowo wczytywał niezasyfrowane aplikacje. W przypadku podania tylko wartości klucza KEY1 została 128-bitowa wersja AES, jeśli dodatkowo podamy wartość KEY2, użyta zostanie wersja 192-bitowa (najwolniejsza ze wszystkich opcji), przy podaniu także wartości KEY3 zostanie najsilniejsza wersja AES, 256-bitowa. Domyslnie generowane są 3 klucze, wykorzystywane jest więc najsilniejsze szyfrowanie z użyciem 256-bitowego klucza. INITIAL\_VECTOR to początkowa wartość służąca do inicjalizacji łańcucha bloku szyfru (CBC). Wartości tej nie ma potrzeby zmieniać.

Przykładowy plik konfiguracyjny z wypełnionym polami dla procesora ATMega128 przedstawiono poniżej:

```
PAGE_SIZE      = 256
MEM_SIZE       = 122880
CRC_ENABLE     = YES
KEY1           = 264A291ACCE179846E257CF35A5F3FA3D6AB
KEY2           = B7F9200081BB069B14
KEY3           = 22992B2DA0B1F1C750
INITIAL_VECTOR = A78D870307888966E31D8A6426E1DDD7
SIGNATURE      = CE259A41
```

Plik ten spowoduje wykorzystanie 256-bitowego szyfrowania AES z CBC, zostanie włączone sprawdzanie CRC, a aplikacja będzie mogła składać się z maksymalnie 122 880 bajtów.



Skompilowanie *bootloadera* z włączoną opcją sprawdzania CRC spowoduje, że każda aplikacja wczytywana za jego pomocą również musi mieć włączone sprawdzanie CRC.

Wygenerowane CRC dla aplikacji zostanie umieszczone na końcu obszaru pamięci zdefiniowanego w pliku konfiguracyjnym — w naszym przykładzie będą to komórki o adresach 0x1DFFE i 0x1DFFF (dziesiętnie 122 878 i 122 879). **Komórki te nie mogą być wykorzystywane przez aplikację.**



Utworzony plik *conf.h* należy zachować, będzie on potrzebny do szyfrowania aplikacji. Należy go także odpowiednio przechowywać — zawiera on klucze umożliwiające odszyfrowanie aplikacji.

Na podstawie skonfigurowanego pliku *conf.h* możemy wygenerować pliki konfiguracyjne *bootloadera* za pomocą programu gcreate:

```
C:\AVR231-AESBL-gcc\pctools>gcreate -c conf.h -h bootldr.h -  
k aeskeys.h
```

```
create v1.0AES - Copyright (C) 2005 Atmel Corporation  
v1.01AES Modified for use with GCC compiler - 2008 Mike Henning
```

Using the following arguments:

```
Config filename: conf.h  
Header filename: bootldr.h  
Key filename:     aeskeys.h
```

The following configuration will be used:

Encryption	= 256-bit AES, cipher block chaining mode
KEY1	= 22B37FFF5E57E77B6AE8CF101F51629E7560
KEY2	= 84BB1249490E97F237
KEY3	= 8BD2F9849539EA3051
INITIAL_VECTOR	= 5824ECFEB43F4A17126F1DB2440AF66C
SIGNATURE	= 882B11E2
PAGE_SIZE	= 256
MEM_SIZE	= 8192
CRC_ENABLE	= YES

Wrote aeskeys.h!

Wrote bootldr.h!

Create finished successfully.

W wyniku powyższego polecenia zostaną utworzone dwa pliki — *bootldr.h* i *aeskeys.h*. Pliki te należy skopiować do katalogu, w którym znajduje się reszta plików *bootloadera*. W ten sposób mamy wygenerowane klucze oraz pliki konfiguracyjne niezbędne do skompilowania *bootloadera*. Pozostaje nam zrobić dwie ostatnie rzeczy: określić adres w pamięci, od którego zostanie umieszczony *bootloader*, oraz ustalić prędkość transmisji wykorzystanego portu UART. Program *bootloadera* umieścimy pod adresem 0x1E000 (adres słowa 0xF000), odpowiednią opcję wpiszemy w okienku konfiguracji projektu — rysunek 27.1.

W tym samym oknie ustalamy prędkość portu UART (opcja *-DBAUD=115200UL*) — rysunek 27.2.

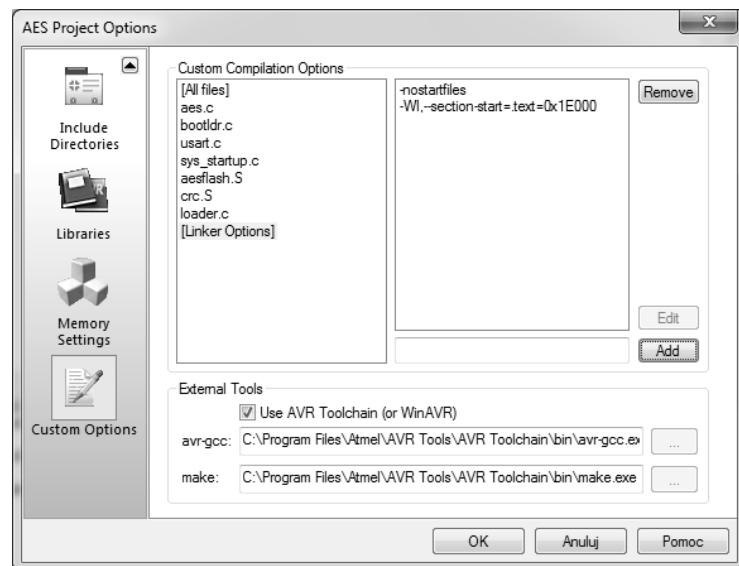


Umieszczenie początku programu bootloadera zależy od konfiguracji fusebitów.

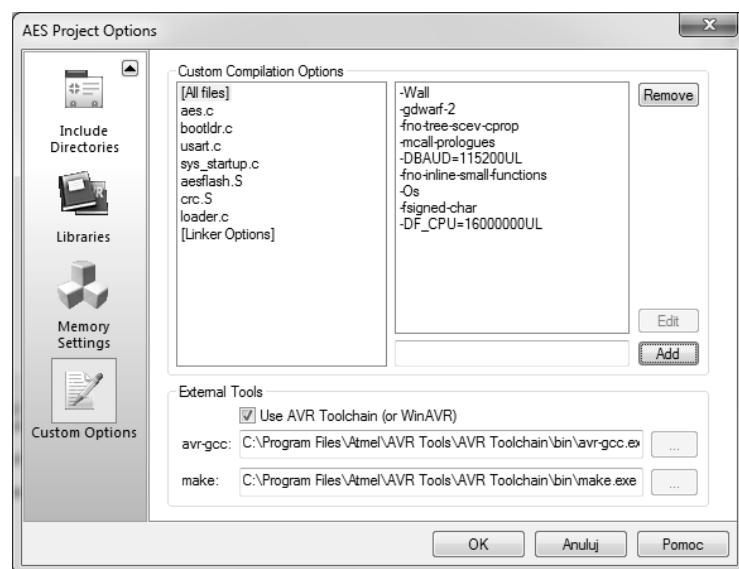
Być może będziemy musieli poprawić także plik *bootldr.c*. Standardowo program *bootloadera* przy uruchomieniu sprawdza, czy pin PD2 procesora jest w stanie niskim. Jeśli tak, to uruchamiany jest podprogram odpowiedzialny za aktualizację oprogramowania, jeśli nie, to następuje próba uruchomienia aplikacji. W nowszych procesorach AVR możemy wykorzystać pin *HWB*, po wcześniejszym zaprogramowaniu fusebitu *HBWE*. Dzięki temu jeśli pin PD2 w trakcie resetu będzie w stanie niskim, uruchomiony zosta-

**Rysunek 27.1.**

Zmiana domyślnego położenia sekcji .text, tak aby znalazła się w obszarze bootloadera

**Rysunek 27.2.**

Ustalenie prędkości działania UART, poprzez definicję globalnego symbolu BAUD



nie *bootloader*, jeśli w stanie wysokim — aplikacja. Wymaga to małej zmiany w kodzie *bootloadera* — z pliku *bootldr.c* musimy usunąć fragment odpowiedzialny za sprawdzanie stanu pinu IO:

```
if (!(PIND & (1 << PD2)))
```

W efekcie zawsze wywoływana będzie funkcja `loader()`.

Dodatkowo, jeśli *bootloader* został skompilowany z opcją sprawdzania CRC, to przed uruchomieniem aplikacji nastąpi jej sprawdzenie. W przypadku błędnego CRC *bootloader*

wchodzi w nieskończoną pętlę, w której zmienia co sekundę stan portu B. Jeśli do tego portu podłączymy diodę, to jej miganie będzie sygnaлизować problem z uruchomieniem aplikacji. Jeśli to domyslne działanie lub wykorzystane piny IO nam nie odpowiadają, to należy zmodyfikować działanie funkcji `main` znajdującej się w pliku `bootldr.c`. Uruchomienie kodu *bootloadera* możemy także osiągnąć poprzez skok pod adres początku *bootsektora*.

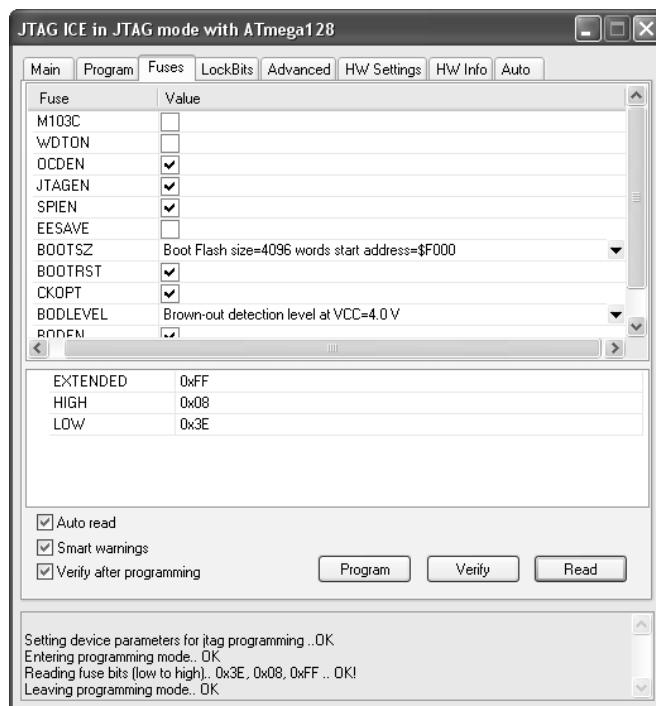
Po skompilowaniu *bootloadera* możemy go wgrać za pomocą dowolnego narzędzia do programowania do pamięci mikrokontrolera. Po tym będziemy mogli wgrać za pomocą portu UART dowolny kod aplikacji, zaszyfrowany wcześniej przy pomocy pary wygenerowanych kluczy AES.

## Ustawienie bitów konfiguracyjnych

Do prawidłowego działania *bootloadera* niezbędne jest właściwe ustawienie bitów konfiguracyjnych — *fuse*- i *lockbitów*. Bity konfiguracyjne możemy w wygodny sposób zmienić w AVR Studio w zakładce *Tools/Program AVR* — rysunek 27.3.

Rysunek 27.3.

Konfiguracja  
fusebitów  
przy pomocy  
AVR Studio



Oprócz właściwej konfiguracji bitów *BOOTSZ* określających początek obszaru *bootloadera* (w naszym przypadku jest to 0xF000, bajtowo 0x1E000) musimy także ustawić bit *BOOTRST*. Dzięki temu po resecie mikrokontroler rozpoczęcie wykonywanie programu *bootloadera*, a nie aplikacji.



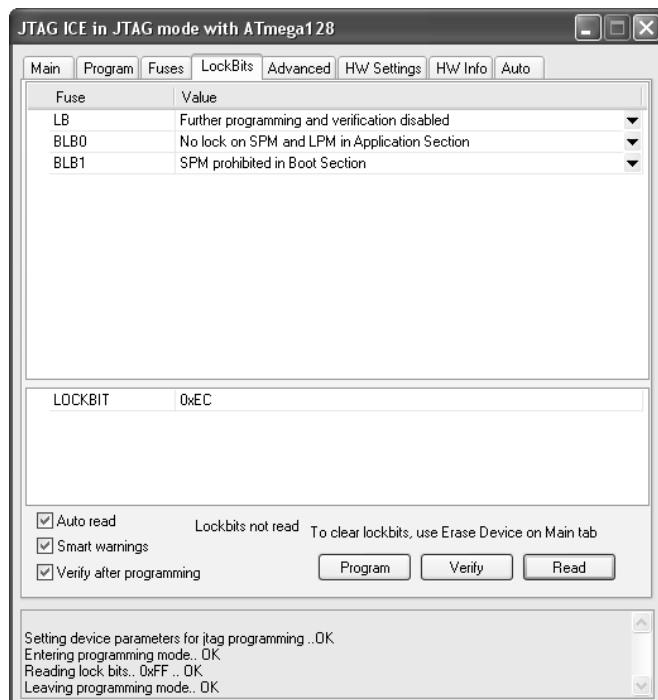
## Wskazówka

Bitu tego nie ustawiamy jedynie wtedy, kiedy wykorzystujemy sprzętowy mechanizm oparty na pinie HWB. W takiej sytuacji *BOOTRST* powinien mieć wartość 1, a fusebit *HBWE* wartość 0.

Należy także ustawić odpowiednio *lockbits*, tak aby nie dało się odczytać przy pomocy programatora pamięci FLASH — rysunek 27.4.

**Rysunek 27.4.**

## Konfiguracja lockbitów



Lockbit *LB* należy ustawić tak, aby dalsze programowanie i odczyt nie były możliwe. Dodatkowo warto zablokować możliwość wykonywania instrukcji SPM w obszarze *bootloadera*. Dzięki temu aplikacja nie będzie mogła nadpisać i uszkodzić kodu *bootloadera*.



## Wskazówka

Bity *BLB0* należy skonfigurować tak, aby możliwy był zapis i odczyt pamięci aplikacji.

Zapis jest niezbędny, aby *bootloader* mógł uaktualniać w przyszłości aplikację. Pozostawienie możliwości odczytu jest niezbędne w przypadku programów przechowujących dane w pamięci FLASH — zmienne z atrybutem *PROGMEM* — lub wykorzystujących zainicjowane zmienne (ich wartości początkowe kopowane są z pamięci FLASH mikrokontrolera).

## Przygotowanie aplikacji

Kod aplikacji przed załadowaniem do pamięci mikrokontrolera za pomocą wczytanego wcześniej *bootloadera* trzeba zaszyfrować. W tym celu ponownie posłużymy się programem *gcreate*. Jako argumenty wywołania przekazujemy mu nazwy plików w formacie Intel HEX zawierające kod aplikacji, opcjonalnie dane do umieszczenia w pamięci EEPROM, stworzony wcześniej plik konfiguracyjny zawierający m.in. klucze; określamy także nazwę pliku wynikowego, który będzie zawierał zaszyfrowany program:

```
C:\AVR231-AESBL-gcc\pctools>gcreate -f AES.hex -e AES.eep -c  
conf.h -o AES.enc
```

```
create v1.0AES - Copyright (C) 2005 Atmel Corporation  
v1.0AES Modified for use with GCC compiler - 2008 Mike Henning
```

Using the following arguments:

```
Config filename: conf.h  
Flash filename: AES.hex  
EEPROM filename: AES.eep  
Output filename: AES.enc
```

The following configuration will be used:

```
Encryption      = 256-bit AES, cipher block chaining mode  
KEY1           = 22B37FFF5E57E77B6AE8CF101F51629E7560  
KEY2           = 84BB1249490E97F237  
KEY3           = 8BD2F9849539EA3051  
INITIAL_VECTOR = 5824ECFEB43F4A17126F1DB2440AF66C  
SIGNATURE       = 882B11E2  
PAGE_SIZE       = 256  
MEM_SIZE        = 122880  
CRC_ENABLE      = YES
```

Read AES.hex!

Read AES.eep!

Wrote AES.enc!

Create finished successfully.

W efekcie zaszyfrowane dane znajdują się w pliku *AES.enc*. Plik ten możemy przy pomocy programu *update* wczytać do mikrokontrolera. Ponieważ jest on zaszyfrowany, możemy go bezpiecznie udostępniać innym osobom — bez znajomości kluczy szansa na jego odszyfrowanie jest znikomo mała.



Plik aplikacji musi zostać zaszyfrowany tymi samymi kluczami, z którymi został skompilowany program *bootloadera*. W przeciwnym przypadku proces aktualniania się nie powiedzie.

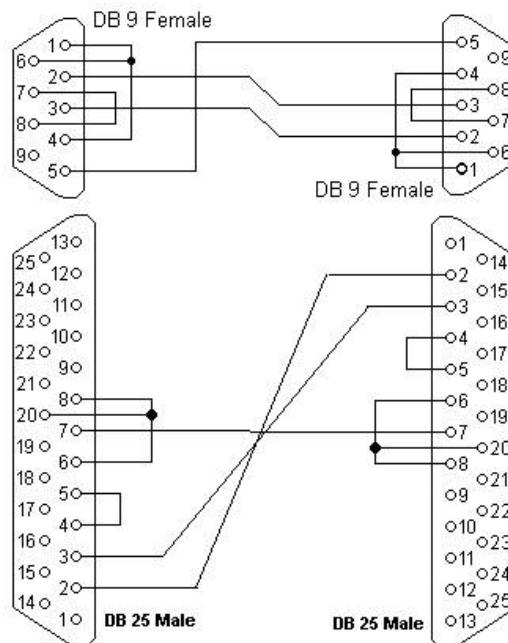
Opcjonalnie do wywołania programu *gcreate* możemy dodać także parametr *-l*, umożliwiający przekazanie wartości bitów konfiguracyjnych BLB01, BLB02, BLB11 i BLB12. Dzięki temu po wgraniu aplikacji zostaną one (programowo) zmienione. Jeśli jednak odpowiednie bity konfiguracyjne od początku mają prawidłową wartość (a tak powinno

być), to nie ma powodu do przekazywania ich stanu jako dodatkowego parametru. Musimy także pamiętać, że programowo można te bity tylko zerować, ich ponowne ustawienie możliwe jest wyłącznie przy pomocy zewnętrznego programatora. Program `gcreate` umożliwia także wyczyszczenie całej pamięci przed programowaniem (opcja `-d`), a także dodanie losowych rekordów, które są ignorowane przez *bootloader*, lecz utrudniają rozszyfrowanie aplikacji (opcja `-n`).

## Wczytywanie uaktualnienia

Pozostaje nam wczytać zaszyfrowaną aplikację do mikrokontrolera. Możemy tego dokonać, łącząc komputer z uaktualnianym układem przy pomocy portu RS232. W tym celu potrzebujemy kabla, tzw. *null-modemu*, w którym połączenia wykonujemy wg schematu z rysunku 27.5.

**Rysunek 27.5.**  
Schemat połączeń kabla null-modem, wykorzystywanego do uaktualniania oprogramowania.  
Do działania wystarczy połączenie mas obu urządzeń oraz krzyżowe połączenie linii *TxD* i *RxD*. Jednak część interfejsów RS232 na PC wymaga dodatkowo odpowiedniego zapętlenia linii *DTR*, *DSR*, *RTS* i *CTS* tak jak na rysunku poniżej



Standardowy *bootloader* nasłuchuje na porcie UART0 mikrokontrolera. Aby go uruchomić, musimy zresetować urządzenie, jednocześnie zapewniając, aby pin PD2 (lub inny, który wykorzystaliśmy do sygnalizacji) był zwarty do masy. Od tego momentu urządzenie jest gotowe do przeprowadzenia aktualizacji. Aktualizację przeprowadzamy, wydając polecenie `upload` z odpowiednimi parametrami:

```
I:\>update bootldrTest.enc -COM1 -115200
update v1.0 - Copyright (C) 2003 Atmel Corporation
```

```
COM1 opened.
Transferring.. 100% done.
Target updated successfully.
```

Po zakończeniu procesu uaktualniania automatycznie uruchamiana jest wczytana aplikacja. Jak widzimy, program update oprócz nazwy pliku z zaszyfrowaną aplikacją wymaga podania portu szeregowego, za pomocą którego łączymy komputer z programowanym układem, oraz prędkości transmisji. Podana prędkość transmisji musi odpowiadać prędkości, z jaką skompilowano *bootloader*. Ta z kolei zależy od częstotliwości taktowania mikrokontrolera. Jeżeli wczytana aplikacja ma nieprawidłowe CRC, nie zostanie ona uruchomiona. Zamiast tego *bootloader* zasygnalizuje problem poprzez cykliczną zmianę stanu portu B mikrokontrolera — na przemian wysyłać będzie na ten port wartości 0 i 255, w cyklu 1-sekundowym. Jeśli pod któryś z pinów tego portu podepnemy diodę LED, jej miganie będzie sygnalizowało problem z CRC. Oczywiście, to domyślne zachowanie możemy zmienić, modyfikując plik *bootldr.c*.

## Rozdział 28.

# Łączenie kodu w C i asemblerze

Specyfika programowania mikrokontrolerów wiąże się czasami z koniecznością pisania fragmentów kodu o ściśle określonym czasie wykonywania lub specjalnie zoptymalizowanych ze względu na konieczność spełnienia ścisłych zależności czasowych. Właściwie wykorzystując język C, można konieczność stosowania asemblera ograniczyć do minimum lub wręcz wyeliminować, jednak czasami istnieje konieczność, aby krótkie, krytyczne fragmenty kodu zostały napisane w asemblerze. Zazwyczaj dzięki przyjęciu dodatkowych założeń, niemożliwych do przekazania przy użyciu języka C, kod asemblerowy napisany przez człowieka jest krótszy niż wygenerowany przez kompilator. Jest tak zazwyczaj w przypadku krótkich funkcji. W przypadku dłuższych i złożonych funkcji zwykle kod generowany przez kompilator jest bardziej zwarty i optymalny. Umieszczając w kodzie programu fragmenty napisane w asemblerze, musimy pamiętać, że nie da się ich w prosty sposób przenieść na inną rodzinę mikrokontrolerów, a ich debugowanie i analiza są utrudnione. Powoduje to, że każdorazowo przed wykorzystaniem wstawki w asemblerze należy się dobrze zastanowić, czy osiągnięty efekt przewyższa niedogodności związane z użyciem asemblera.

Kod asemblerowy można łączyć z kodem programu napisanym w języku C na dwa sposoby. Pierwszy to wykorzystanie znanej już nam i wielokrotnie pokazywanej w poprzednich rozdziałach instrukcji `asm`. Po niej następuje lista mnemoników asemblera z ewentualnymi parametrami, które zostaną przetłumaczone na odpowiadające im instrukcje asemblera. Dzięki słowu kluczowemu `asm` można bezpośrednio w kodzie w języku C umieszczać instrukcje asemblera, wszystko w ramach jednej jednostki komplikacji (jednego pliku źródłowego).

Drugą możliwością jest umieszczenie programu napisanego w asemblerze w oddzielnym pliku. Plik ten musi posiadać rozszerzenie `.S` (bardzo ważne jest, aby `S` było wielką literą). Plik taki dołącza się do projektu analogicznie jak plik źródłowy języka C. Kompilator dzięki rozszerzeniu rozpozna, że nie jest to plik zawierający kod języka C, lecz kod asemblerowy, i do jego komplikacji wywoła asembler. W efekcie powstanie plik obiektowy (`.o`), który można zlinkować z programem. Zaletą umieszczania funkcji napisanych w asemblerze w osobnym pliku jest łatwość dostosowania projektu do nowej

platformy sprzętowej. Cały sprzętowo specyficzny kod asemblerowy jest wydzielony w oddzielnich plikach z rozszerzeniem .S, w efekcie łatwo jest go modyfikować. Wadą tego rozwiązania jest konieczność pamiętania o sposobie przekazywania parametrów i zwracania wartości funkcji. Informacje te są specyficzne dla kompilatora, a nawet dla jego konkretnej wersji. W efekcie, jeśli zmieni się ABI kompilatora (binarny interfejs aplikacji), to zazwyczaj zmieniają się konwencje wywoływania parametrów, co pociąga za sobą konieczność modyfikacji wcześniejszych napisanych funkcji asemblerowych.

## Słowo kluczowe asm

Dzięki słowu kluczowemu `asm` można w dowolne miejsce kodu wstawić fragment napisany w asemblerze. Umożliwia ono wstawienie jednej lub wielu instrukcji asemblera. Jego ogólna postać wygląda następująco:

```
asm(instrukcja : operandy wyjściowe : operandy wejściowe [: lista modyfikowanych rejestrów]);
```

Lista operandów wejściowych i wyjściowych może być pusta, natomiast lista modyfikowanych rejestrów jest opcjonalna i może zostać pominięta. W efekcie dla prostych instrukcji, które nie mają żadnych argumentów i nie zwracają wyników oraz nie modyfikują innych rejestrów, instrukcję `asm` można uproszczyć do postaci:

```
asm("nop");
```

Kompilator C może dokonywać optymalizacji umieszczonego kodu asemblerowego, w szczególnych przypadkach optymalizator może usunąć całą wstawkę asemblerową — dzieje się tak w sytuacji, w której stwierdzi, że dany kod nic nie robi. Aby temu zapobiec, należy użyć znanego już modyfikatora `volatile`:

```
asm volatile ("nop");
```

Dodanie modyfikatora `volatile` ma niekorzystny skutek uboczny — wyłącza optymalizację kodu asemblerowego — jednak w większości przypadków nie jest to duża strata.

Przy pomocy jednego słowa kluczowego `asm` można umieścić w kodzie programu wiele instrukcji asemblera:

```
asm volatile(
    "nop ; komentarz"
    "nop ; i to też jest komentarz");
```

Powyższy kod umieszcza w programie dwie instrukcje NOP, elementy po średniku są komentarzem pomijanym przez kompilator.

W AVR-libc zdefiniowane są pewne specjalne symbole, które można wykorzystać zamiast odwoływania się do konkretnych rejestrów procesora (tabela 28.1).

Symbole `_SREG`, `_SP_L` i `_SP_H` zawierają adresy IO odpowiednich rejestrów procesora, stąd też należy się do nich odwoływać przy pomocy instrukcji IN i OUT.

**Tabela 28.1.** Symbole dostępne dla programów w asemblerze

<b>Symbol</b>	<b>Rejestr</b>
<code>_SREG</code>	Rejestr stanu procesora (adres IO 0x3F).
<code>_SP_H</code>	Starsza połowa rejestrów stosu (adres IO 0x3E) — występuje tylko w procesorach, w których rejestr stosu ma więcej niż 8 bitów.
<code>_SP_L</code>	Młodsza połowa rejestrów stosu (adres IO 0x3D).
<code>_tmp_reg</code>	Rejestr R0, służący do przechowywania tymczasowych wartości.
<code>_zero_reg</code>	Rejestr R1 — zawsze powinien zawierać wartość 0.

`_tmp_reg` i `_zero_reg` są aliasami rejestrów R0 i R1. Zamiast nich można się odwoływać bezpośrednio do rejestrów, lecz nie jest to zalecane, gdyż w przyszłych wersjach avr-gcc przyporządkowanie tych aliasów do konkretnych rejestrów procesora może się zmienić.



**Wskazówka** Jeśli wstawka w asemblerze wykorzystuje tylko rejestr `_tmp_reg`, to nie trzeba go wymieniać na liście rejestrów modyfikowanych.

## Typy operandów

Po wszystkich instrukcjach asemblera należy wyszczególnić listę operandów wyjściowych i wejściowych. Umożliwia to odwoływanie się we wstawce asemblerowej do struktur danych zdefiniowanych w programie napisanym w języku C. Aby to było możliwe, należy określić typy poszczególnych operandów (tabela 28.2).

Oprócz typu operandu należy także określić sposób dostępu do niego (tabela 28.3). Jeśli nie zostanie określony sposób dostępu, to domyślnie przyjmowane jest, że operand jest tylko do odczytu.

Spróbujmy wykorzystać zdobytą wiedzę teoretyczną na kilku praktycznych przykładach. Pierwszym będzie krótka wstawka, umożliwiająca wymianę tetrad w 8-bitowej zmiennej. Operacja taka nie jest wspierana przez język C, natomiast jest wspierana przez asembler procesorów AVR. Zdefiniujmy następujący kod:

```
uint8_t nibbles=0xF0;
asm volatile("swap %0" : "=r" (nibbles) : "0" (nibbles));
```

W powyższym przykładzie zdefiniowano zmienną `nibbles`, a następnie krótką wstawkę powodującą wymianę tetrad tej zmiennej. W efekcie zmienna ta będzie miała wartość 0x0F. Przyjrzyjmy się operandom instrukcji `asm`. Pierwszy, określający operand wyjściowy, informuje kompilator, że zmienna `nibbles` znajdzie się w dowolnym rejestrze R0 – R31 (stała `r`), jednocześnie jest to operand tylko do zapisu. Przed wykonaniem operacji SWAP kompilator załaduje wartość zmiennej `nibbles` do rejestrów procesora, do którego symbolicznie będziemy odwoływać się poprzez `%0`. Właściwy rejestr zostanie określony na etapie komplikacji programu. Zobaczmy, jak wygląda wygenerowany przez kompilator kod wynikowy:

**Tabela 28.2.** Typy operandów

Stała	Znaczenie	Zakres
a	Górne rejestrysty	R16 – R23
b	Rejestry indeksowe	Y, Z
d	Górne rejestrysty	R16 – R31
e	Rejestry indeksowe	X, Y, Z
G	Stała zmiennopozycyjna	0.0
I	6-bitowa liczba naturalna	0 – 63
J	6-bitowa liczba ujemna	-63 – 0
K	Stała całkowita	2
L	Stała całkowita	0
l	Rejestry dolne	R0 – R15
M	Stała 8-bitowa	0 – 255
N	Stała	-1
O	Stała	8, 16, 24
P	Stała	1
q	Rejestr stosu	SPH:SPL
r	Dowolny rejestr	R0 – R31
t	Rejestr tymczasowy	R0
w	16-bitowa para rejestrów górnych	R25:R24, R27:R26, R29:R28, R31:R30
x	Rejestr indeksowy X	X (R27:R26)
y	Rejestr indeksowy Y	Y (R29:R28)
z	Rejestr indeksowy Z	Z (R31:R30)

**Tabela 28.3.** Modyfikatory określające tryb dostępu do operandu

Modyfikator	Tryb
=	Operand tylko do zapisu, zazwyczaj używane w przypadku operandów wyjściowych.
+	Operand do odczytu i zapisu.
&	Operand wyjściowy znajduje się w rejestrze — tylko dla operandów wyjściowych.

```

18:      asm volatile("swap %0 : "=r" (nibbles) : "0" (nibbles));
+00000056: EF80      LDI       R24.0xF0      Load immediate
+00000057: 9582      SWAP     R24          Swap nibbles
20:      }

```

Jak widać, kompilator użył rejestrów R24, bo akurat ten był wolny. Używanie symboli zamiast konkretnych rejestrów umożliwia optymalizatorowi dostosowanie naszej wstawki do otaczającego programu.

## Dostęp do portów IO

Po włączeniu pliku nagłówkowego `<avr/io.h>` mamy dostęp do wszystkich symboli definiujących porty IO procesora. Przeanalizujmy poniższy prosty przykład:

```
uint8_t value =0xff;  
  
asm volatile("out %0, %1"  
           :: "I" (_SFR_IO_ADDR(PORTD)), "r" (value));
```

Powyższy przykład ma za zadanie wpisać do PORTD wartość zmiennej `value`. Ponieważ powyższa wstawka nic nie zwraca, lista operandów wyjściowych jest pusta, zdefiniowano tylko operandy wejściowe: `_SFR_IO_ADDR(PORTD)`, do którego będzie można odwoływać się poprzez `%0`, i wartość wpisywaną do rejestru IO (stała `r`). Ponieważ instrukcja OUT może operować na dowolnym rejestrze z zakresu R0 – R31, wykorzystano literał `r`. Dla instrukcji działających tylko na górnej połówce rejestrów należałyby użyć literala `d`.



Wskazówka Do wszystkich rejestrów IO procesora należy odwoływać się poprzez makro `_SFR_IO_ADDR`.

## Dostęp do danych wielobajtowych

Do tej pory operandami były rejesty 8-bitowe, jednak znacznie częściej wykorzystywane są wielobajtowe typy danych. Dostęp do poszczególnych 8-bitowych części odbywa się przy pomocy predefiniowanych literałów A, B, C, D, określających odpowiednio bity 0 – 7, 8 – 15, 16 – 23 i 24 – 31. Tak więc w przypadku 32-bitowej zmiennej umieszczonej w rejestrach dostęp do jej poszczególnych bajtów uzyskuje się poprzez literaly `%An`, `%Bn`, `%Cn`, `%Dn`, gdzie `n` to symbol przyporządkowany danej zmiennej.

W poniższym przykładzie stworzono wstawkę assemblerową, której celem jest inkrementacja 32-bitowej zmiennej całkowitej oraz zwrócenie ewentualnego przeniesienia z najstarszego bitu:

```
uint32_t val=0xffffffff;  
uint8_t carry;  
  
asm volatile("ADIW %A0, 1" "\n\t"  
           "ADC %C0, R0" "\n\t"  
           "ADC %D0, R0" "\n\t"  
           "MOV %1, 0" "\n\t"  
           "BRCC L1" "\n\t"  
           "LDI %1,1" "\n\t"  
           "L1:" "\n\t"  
           : "=w" (val), "=d" (carry)  
           : "0" (val)  
           );
```

W powyższym przykładzie do inkrementacji 16-bitowego rejestru wykorzystano instrukcję ADIW, lecz jej operandami mogą być wyłącznie pary rejestrów R31:R30, R29:R28,

R27:R26, R25:R24. Dlatego na liście argumentów wejściowych zastosowano stałą `w`, informującą kompilator, że zmienna `val` ma znaleźć się w którejś z wymienionych wcześniej par rejestrów. Dostęp do poszczególnych 8-bitowych części 32-bitowej zmiennej `val` odbywa się poprzez operatory `%A0`, `%C0` i `%D0`. Po inkrementacji sprawdzany jest stan flagi Carry procesora. Jeśli jest ona ustawiona (co oznacza przepełnienie), zmiennej `carry` nadawana jest wartość 1. Jak widzimy, w pisanych wstawkach asemblerowych można korzystać także z etykiet umożliwiających wykorzystywanie instrukcji skoku. Etykietę definiuje się, umieszczając w kodzie jej nazwę zakończoną dwukropkiem. Zobaczmy, jak kompilator skompilował powyższy kod:

```

29:     asm volatile("ADIW %A0, 1" "\n\t"
+0000005A: EF8F      SER      R24      Set Register
+0000005B: EF9F      SER      R25      Set Register
+0000005C: 01DC      MOVW    R26,R24  Copy register pair
+0000005D: 9601      ADIW    R24,0x01 Add immediate to word
+0000005E: 1DA0      ADC     R26,R0   Add with carry
+0000005F: 1DB0      ADC     R27,R0   Add with carry
+00000060: 2D20      MOV     R18,R0   Copy register
+00000061: F408      BRCC   PC+0x02 Branch if carry cleared
+00000062: E021      LDI     R18,0x01 Load immediate
40:     }

```

Jak widzimy, kompilator użył do przechowywania zmiennej `val` rejestrów R27:R24; zmienna `carry` umieszczona została w rejestrze R18.



**Wskazówka** Kompilator sam dba o umieszczenie danych we właściwych rejestrach przed wygenerowaniem wstawki asemblerowej i ewentualne zachowanie wyników w komórkach pamięci przyporządkowanych danym zmiennym (o ile zachodzi taka potrzeba).

## Dostęp do wskaźników

Wskaźniki w mikrokontrolerach AVR są 16-bitowe, lecz dostęp do nich odbywa się nieco inaczej niż w przypadku innych 16-bitowych zmiennych. Dostęp do 8-bitowych rejestrów przechowujących adres odbywa się nie przy pomocy stałych `%An` i `%Bn`, lecz za pomocą `%an` i `%bn`. Poniższy kod w sposób atomowy zwiększa wartość przechowywaną w komórce pamięci wskazywanej przez zmienną `ptr` o typie `uint8_t*`:

```

asm volatile ("CLI" "\n\t"
    "LD __tmp_reg__, %a0" "\n\t"
    "INC __tmp_reg__" "\n\t"
    "ST %a0, __tmp_reg__" "\n\t"
    "SEI" "\n\t"
    :: "e" (ptr));

```

Dodanie modyfikatora `volatile` zapobiega optymalizacji powyższego kodu, która mogłaby doprowadzić do zmiany kolejności instrukcji, w tym instrukcji `SEI` i `CLI`. Stała `e` informuje kompilator, że `ptr` jest wskaźnikiem (zauważmy, że nie ma znaczenia typ wskazywanych danych — o to musi już zadbać programista). Do przechowywania wartości tymczasowej odczytanej spod adresu wskazywanego przez wskaźnik użyto rejestru `__tmp_reg__`.

## Listy modyfikowanych rejestrów

Dotychczas pokazane funkcje nie modyfikowały żadnych rejestrów poza rejestrami wyszczególnionymi na listach operandów wejściowych i wyjściowych oraz rejestru tymczasowego `_tmp_reg_`. Jednak w bardziej skomplikowanych wstawkach zachodzi konieczność modyfikacji innych rejestrów. Aby kompilator wiedział, jakie rejestrory są modyfikowane we wstawce asemblerowej, należy je wyszczególnić na liście modyfikowanych rejestrów (trzeci opcjonalny parametr wywołania `asm`). W przykładzie poniżej do przechowywania odczytanej wartości z komórki pamięci adresowanej przez wskaźnik użyto rejestru tymczasowego, możemy jednak użyć innego rejestru:

```
asm volatile ("CLI" "\n\t"
    "LD R24, %a0" "\n\t"
    "INC R24" "\n\t"
    "ST %a0, R24" "\n\t"
    "SEI" "\n\t"
    :: "e" (ptr) : "r24");
```

W powyższym przykładzie użyto rejestrów R24. Aby o tym poinformować kompilator, wyszczególniono go na liście modyfikowanych rejestrów.

Specjalnym literałem jest `memory`, informujący kompilator, że dana wstawka odwołuje się do pamięci mikrokontrolera. W rezultacie przed jej wywołaniem zawartość wszystkich rejestrów zawierających kopie danych z pamięci SRAM jest uaktualniana poprzez ich ponowne wczytanie.



W miarę możliwości należy unikać literatu `memory` oraz ograniczać liczbę wykorzystywanych rejestrów. Dzięki temu optymalizator będzie miał szansę wygenerowania lepszego kodu.

## Wielokrotne użycie wstawki asemblerowej

Zazwyczaj danej wstawki asemblerowej można użyć wielokrotnie, w różnych częściach programu. W tym celu można ją „przerobić” na makrodefinicję lub na funkcję. Spróbujmy przerobić poprzedni przykład na makrodefinicję:

```
#define INC_ATOMIC(ptr) asm volatile ("CLI" "\n\t" \
    "LD _tmp_reg_, %a0" "\n\t" \
    "INC _tmp_reg_" "\n\t" \
    "ST %a0, _tmp_reg_" "\n\t" \
    "SEI" "\n\t" \
    :: "e" (ptr));
```

Tak zdefiniowanego makra można użyć w dowolnym miejscu programu przy pomocy wywołania `INC_ATOMIC(wskaźnik)`. Pamiętamy jednak, że makrodefinicje stwarzają różne problemy i w miarę możliwości należy ich unikać. Taki sam efekt uzyskamy, definiując funkcję:

```
static inline void inc_atomic(uint8_t *ptr)
{
    asm volatile ("CLI" "\n\t"
                 "LD __tmp_reg__, %a0" "\n\t"
                 "INC __tmp_reg__" "\n\t"
                 "ST %a0, __tmp_reg__" "\n\t"
                 "SEI" "\n\t"
                 :: "e" (ptr));
}
```

Dodanie modyfikatora static inline powoduje, że powyższa funkcja będzie osadzana w miejscu wywołania, nigdy nie będzie generowany adres skoku do niej. Będzie się zachowywać dokładnie tak samo jak wcześniejsza makrodefinicja, z tym że nie będzie posiadała wad makrodefinicji.

W podobny sposób można napisać funkcję, która zwraca wynik:

```
static inline uint8_t inc_atomic_ret(uint8_t *ptr)
{
    uint8_t tmp;
    asm volatile ("CLI" "\n\t"
                 "LD %0, %a1" "\n\t"
                 "INC %0" "\n\t"
                 "ST %a1, %0" "\n\t"
                 "SEI" "\n\t"
                 : "=r" (tmp): "e" (ptr));
    return tmp;
}
```

W powyższym przykładzie oprócz inkrementacji danej wskazywanej przez wskaźnik ptr zwracana jest jej wartość. W tym celu wykorzystano funkcję języka C return. Dzięki działaniu optymalizatora końcowa instrukcja return tmp najprawdopodobniej nie spowoduje wygenerowania jakiegokolwiek kodu asemblerowego. Jako %0 wykorzystany zostanie rejestr, który zostanie użyty w dalszej części programu napisanego w języku C.

## Pliki .S

Pliki z rozszerzeniem .S kompliwane są przez asembler dołączony do pakietu WinAVR (avr-as). Jest on wywoływany pośrednio poprzez program avr-gcc. Zazwyczaj w plikach z rozszerzeniem .S umieszczane są kompletne funkcje, do których można się odwoływać z poziomu języka C poprzez zadeklarowane etykiety globalne. Aby możliwe było przekazywanie parametrów i zwracanie wyników przez funkcje napisane całkowicie w asemblerze, niezbędna jest znajomość konwencji przekazywania parametrów. Sposób ten zależy od konkretnej wersji kompilatora i w przyszłości może ulec zmianie. Jest to pewna dodatkowa niedogodność, powodująca możliwość utraty kompatybilności pisанego programu z przyszłymi wersjami kompilatora. Wady tej nie posiadają wstawki asemblerowe umieszczane w kodzie programu przy pomocy dyrektywy asm. W takiej sytuacji kompilator sam dba o odpowiednie powiązanie parametrów funkcji z wykorzystywany rejestrami.

## Wykorzystanie rejestrów w asemblerze

Kompilator C, o ile to tylko możliwe, stara się przekazywać parametry wywołań funkcji w rejestrach, co znacznie przyśpiesza proces wywoływanego funkcji. W sytuacji, w której niemożliwe jest umieszczenie wszystkich argumentów w rejestrach, kolejne argumenty przekazywane są poprzez stos, co jest zdecydowanie mniej efektywne — wywołującą procedurę musi ich wartość umieścić na stosie, a funkcja napisana w asemblerze przed dostępem do nich musi je ze stosu odczytać. W zależności od typu argumentu do jego przekazania wymagana jest różna liczba rejestrów. Rozmiary typów w AVR wynoszą:

- ♦ char — 8 bitów,
- ♦ int — 16 bitów,
- ♦ long — 32 bity,
- ♦ long long — 64 bity,
- ♦ float i double — 32 bity.

W przypadku typu char do jego przekazania teoretycznie wystarczy jeden rejestr procesora, ale ze względu na specyfikę ABI (kolejne argumenty przekazywane są, począwszy od rejestrów o parzystym numerze) typ char przekazywany jest jako para rejestrów, z których rejestr o wyższym numerze (nieparzysty) jest nieużywany. Argumenty funkcji przekazywane są przy pomocy rejestrów R25 – R8, w kolejności, w jakiej występują w deklaracji funkcji. Np. do funkcji:

```
void test(char a1, uint16_t a2, long a3, char a4);
```

argumenty przekazywane są w następujący sposób:

- ♦ Argument a1 przekazywany jest w rejestrze R24 (rejestr R25 jest niewykorzystywany).
- ♦ Argument a2 przekazywany jest w rejestrach R23:R22.
- ♦ Argument a3 przekazywany jest w rejestrach R21:R18.
- ♦ Argument a4 przekazywany jest w rejestrze R16 (rejestr R17 jest niewykorzystany).



Dzięki takiej konwencji przekazywania argumentów możliwe jest efektywniejsze przekazywanie wartości pomiędzy rejestrami dzięki wykorzystaniu instrukcji asemblera MOVW (o ile dany rdzeń AVR ją wspiera).

W przypadku zwracania rezultatów działania funkcji sytuacja jest o wiele prostsza. W języku C funkcja może zwrócić tylko jeden rezultat. Jeżeli jest to niewystarczające, zwracany jest wskaźnik na złożoną strukturę danych, zawierającą wynik działania funkcji. W efekcie rezultaty działania funkcji zawsze zwracane są poprzez rejesty procesora:

- ♦ Wartości 8-bitowe zwracane są w R24.
- ♦ Wartości 16-bitowe zwracane są w parze R25:R24.

- ◆ Wartości 32-bitowe zwracane są w R25:R22.
- ◆ Wartości 64-bitowe zwracane są w R25:R18.

Należy pamiętać, że wartości 8-bitowe (np. typ `char`) są automatycznie promowane do typów 16-bitowych (`int`) i zwracane w parze R25:R24, z której tylko R24 zawiera sensowny wynik. Promocja typów wiąże się jednak z koniecznością wykonania dodatkowych instrukcji, stąd też lepiej, jeśli funkcja zamiast typu `char` zwraca typ `unsigned char` (typ `char` może automatycznie być traktowany jako `unsigned`, przy wykorzystaniu opcji kompilatora `-funsigned-char`). Dzięki temu rejestr R25 jest tylko zerowany instrukcją CLR R25, nie zachodzi konieczność dokonania kosztownej promocji znaku liczby.

Pozostałe rejestyre procesora podzielone są na klasy:

- ◆ Rejestry R18-R27 oraz R31:R30, niewykorzystywane do przekazywania argumentów, mogą być swobodnie wykorzystywane w kodzie asemblerowym. Funkcja wywołująca jest odpowiedzialna za ewentualne zachowanie ich wartości.
- ◆ Rejestry R2-R17 i R29:R28 mogą być wykorzystywane, jednak ich zawartość musi być odtworzona przed powrotem z funkcji. Funkcja wywołująca zakłada, że ich zawartość nie ulegnie zmianie.
- ◆ Rejestr specjalny R1 zawsze powinien zawierać zero. Jeśli funkcja go modyfikuje, to przed powrotem lub wywołaniem innej funkcji musi go wyzerować. Należy pamiętać, że niektóre instrukcje, np. `MUL`, mogą niejawnie modyfikować zawartość R1.
- ◆ Rejestr R0 — może być wykorzystywany jako rejestr tymczasowy, jego zawartość można dowolnie modyfikować.



Wskazówka

Argumenty, które nie zmieściły się w rejestrach procesora, oraz zmienne lokalne twozone są w ramach funkcji na stosie. Rejestr Y procesora zawiera wskaźnik do nich, co umożliwia ich wygodne adresowanie. Należy pamiętać, że zawartość tego rejestru musi zostać odtworzona przed powrotem z funkcji.

## Dostęp do rejestrów IO procesora

Pisząc program w asemblerze, można wykorzystywać wszystkie pliki nagłówkowe języka C, w szczególności plik `<avr/io.h>` zawierający definicję rejestrów IO procesora. Dzięki niemu możemy odwoływać się do rejestrów IO procesora poprzez nazwy symboliczne, a nie ich adresy. Adres w przestrzeni IO danego rejestru można uzyskać przy pomocy makrodefinicji `_SFR_IO_ADDR(nazwa rejestru)`.

## Definicje

Oprócz nazw symbolicznych w programie można wykorzystywać także dyrektywy preprocesora C, takie jak `#define`, `#ifdef` itd. Umożliwia to warunkową komplikację programu oraz odwoływanie się do rejestrów procesora poprzez lokalnie zdefiniowane nazwy, a nie numery rejestrów (R0 – R31), np.:

```
#define licznik 18
```

powoduje, że do rejestru R18 można odwoływać się poprzez symbol licznik.

## Definicja symboli globalnych

Aby dany symbol zdefiniowany w programie asemblerowym stał się widoczny dla programu napisanego w języku C, należy go wyeksportować przy pomocy dyrektywy `.global`, np.:

```
.global funkcja_w_asm
```

powoduje, że symbol `funkcja_w_asm` staje się widoczny na zewnątrz modułu i można się do niego odwoływać z poziomu języka C. Taki symbol musi być oczywiście zdefiniowany w jakimś miejscu programu asemblerowego. Jego definicja następuje poprzez stworzenie stosownej etykiety (nazwy symbolu zakończonej dwukropkiem):

```
.global funkcja_w_asm  
funkcja_w_asm:  
<dalszy ciąg programu>
```

Tak zdefiniowana etykieta może wskazywać zarówno na punkt wejścia do funkcji, jak i na dane lokalne — stąd też interpretacja danego symbolu należy do programisty. Najwygodniej jest więc stworzyć odpowiedni plik nagłówkowy zawierający prototypy funkcji i zmiennych wykorzystywanych w asemblerze. Dzięki temu fragmenty programu napisane w języku C mogą się do nich odwoływać tak jak do zwykłego kodu języka C.

## Symbole zewnętrzne

W programie asemblerowym czasami zachodzi konieczność wykorzystania symboli zdefiniowanych w innych modułach programu. W takiej sytuacji można wykorzystać operator `.extern`, informujący kompilator, że dany symbol znajduje się w innym module. Symbol ten musi być dostępny na etapie linkowania programu.

## Procedury obsługi przerwań w asemblerze

Linker automatycznie łączy funkcję obsługi przerwania z jej wektorem znajdującym się w tabeli wektorów przerwań. Aby takie automatyczne połączenie było możliwe, funkcja obsługi przerwania musi posiadać odpowiednią nazwę, dodatkowo nazwa ta musi być wyeksportowana jako symbol globalny. Nazwy funkcji obsługi przerwań są takie same jak w języku C (argumenty wywołania makra ISR). Np.:

```
.global TIMER0_OVF_vect  
TIMER0_OVF_vect:  
<funkcja obsługi przerwania>
```

powoduje utworzenie funkcji o nazwie `TIMER0_OVF_vect`, którą linker połączy z odpowiednim wektorem obsługi przerwania.



Aby możliwe było wykorzystanie nazw wektorów przerwań, należy dołączyć plik nagłówkowy `<avr/io.h>`.

Należy zwracać szczególną uwagę na poprawność zapisu nazwy wektora przerwania. Jego niepoprawna nazwa nie generuje żadnych ostrzeżeń ani błędów (kompilator nie ma możliwości sprawdzenia poprawności nazwy). Efektem błędnego zapisu będzie brak stworzenia odpowiedniego wpisu w tabeli wektorów przerwań, łączącego wektor przerwania z jego procedurą obsługi.

## Inne pseudooperatory

W asemblerze można posługiwać się wieloma innymi operatorami, umożliwiającymi rezerwację pamięci dla zmiennych o różnych długościach oraz sterowanie przebiegiem generowania kodu.

### Operator .byte

Rezerwuje on jeden bajt pamięci dla zmiennej, np.:

```
Zmienna:  
.byte 10
```

### Operator .ascii

Powoduje on zarezerwowanie pamięci dla podanego łańcucha znakowego w formacie ASCII. Należy pamiętać, że tak zdefiniowany łańcuch nie jest automatycznie kończony znakiem NULL.

### Operator .asciz

Jego działanie jest podobne do poprzedniego operatora, z tym że tworzony łańcuch automatycznie jest kończony znakiem NULL, co umożliwia jego poprawną interpretację przez standardowe funkcje języka C.

### Operator .section

Określa on miejsce umieszczenia dalszego kodu programu (segment pamięci). Jako opcję można określić .text (dalszy ciąg będzie umieszczony w pamięci FLASH) lub .data (dalszy ciąg znajdzie się w pamięci SRAM). W ten sposób określona sekcja obowiązuje aż do kolejnej zmiany przy pomocy operatora .section.

### Operatory lo8 i hi8

Operatory te zwracają odpowiednio młodsze i starsze 8 bitów 16-bitowego argumentu. Są one przydatne do pobierania i ładowania adresu do 8-bitowych rejestrów.

### Operator pm

Operator ten pobiera 16-bitowy adres argumentu znajdującego się w pamięci FLASH mikrokontrolera. W praktyce operator ten dzieli adres przez 2 (wszystkie etykiety w pamięci FLASH zaczynają się od adresów parzystych), umożliwiając w ten sposób

skok do funkcji znajdujących się w tej pamięci przy pomocy takich instrukcji jak IJMP, ICALL. W przypadku bezpośredniego wykorzystania adresu, np. w funkcjach JMP lub CALL, nie ma potrzeby stosowania operatora pm.

## Przykłady

Poniżej przedstawiono kilka przykładów ilustrujących łączenie funkcji napisanych w asemblerze z programem napisanym w języku C. Wszystkie przykłady umieszczone zostały w pliku *asm.S* i dołączone do projektu w języku C. Dzięki temu kompilator avr-gcc po natrafieniu na taki plik automatycznie wywoła asembler (avr-as). Dodatkowo w pliku *func.h* zadeklarowane zostały prototypy funkcji zdefiniowanych w asemblerze, dzięki czemu można je łatwo wywoływać w programie.

## Przerwania

Pierwszy przykład ilustruje wykorzystanie asemblera do pisania procedur obsługi przerwań.:

```
#include <avr/io.h>

.section .text
.global TIMER0_OVF_vect

TIMER0_OVF_vect:
    SEI
    RET
```

Powyższa procedura obsługi przerwania TIMER0\_OVF\_vect niewiele robi — jej efektem jest tylko odblokowanie przerwań i powrót do programu wywołującego. Dzięki wykorzystaniu odpowiedniego symbolu (TIMER0\_OVF\_vect) linker automatycznie wygeneruje odpowiedni wpis w tablicy wektorów przerwań.

## Pobieranie argumentu i zwracanie wyniku

W kolejnym przykładzie zdefiniowano funkcję o prototypie:

```
uint16_t INC(uint16_t val);
```

Funkcja ta przyjmuje jeden argument o typie uint16\_t i zwraca wynik o takim samym typie. Wiemy, że argument wywołania funkcji zostanie umieszczony przez kompilator w parze rejestrów R25:R24; kompilator oczekuje, że wynik zostanie zwrócony w tych samych rejestrach. Umożliwia to napisanie prostej funkcji, której efektem działania jest inkrementacja zmiennej podanej jako argument wywołania:

```
.global INC
INC:
    ADIW R24,1
    RET
```

## Odwółanie do zmiennych zewnętrznych

Kolejną możliwością interakcji z kodem napisanym w języku C jest odwołanie się w funkcji napisanej w asemblerze do zmiennych zdefiniowanych poza nią. Dla przykładu zdefiniowano funkcję o prototypie:

```
void INC_counter();
```

Jej zadaniem jest inkrementacja zmiennej counter, zdefiniowanej w innym module:

```
.extern counter
.global INC_counter
INC_counter:
    LDS R18, counter
    INC R18
    STS counter, R18
    RET
```

Ponieważ symbol counter nie jest znany w pliku definiującym funkcję, zadeklarowano go jako symbol zewnętrzny (.extern). Sama funkcja pobiera daną spod adresu określonego przez counter, inkrementuje ją i zapisuje w to samo miejsce.

## Rozdział 29.

# Optymalizacja i debugowanie programu

## Optymalizacja programu

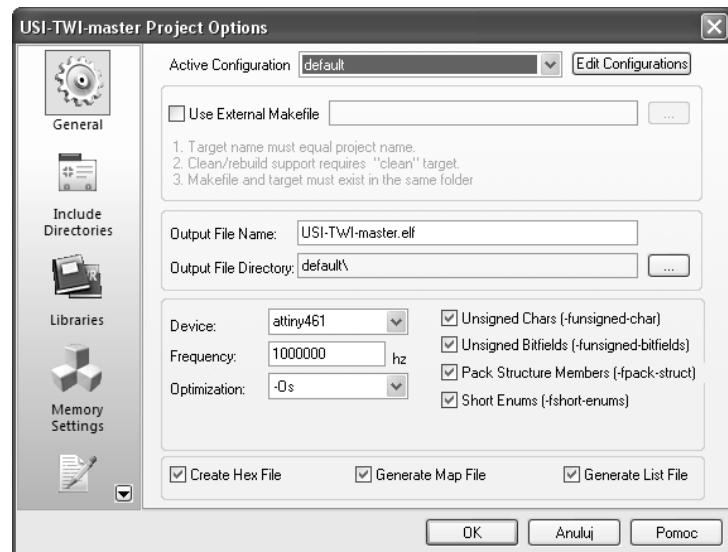
Programując w języku wyższego poziomu, mamy mniejszą kontrolę nad wygenerowanym kodem maszynowym, a więc również wielkością programu oraz czasem jego wykonywania. W wielu sytuacjach nie da się jednocześnie skompilować programu tak, aby działał szybko, a jednocześnie był krótki. Stąd też kod wynikowy jest zawsze kompromisem pomiędzy tymi dwoma celami. Język C bezpośrednio nie definiuje żadnych mechanizmów związanych z optymalizacją kodu — bezpośredniem celem kompilatora jest wygenerowanie kodu równoważnego instrukcjom programu. Aby taki wygenerowany kod był możliwie efektywny, do kompilatora wprowadzono tzw. optymalizator. Jest to integralna część kompilatora, z którą programista komunikuje się poprzez specjalne opcje przekazywane na etapie wywoływania kompilatora oraz poprzez atrybuty zmiennych i funkcji użytych w programie. Ta część zawsze jest bardzo specyficzna dla konkretnego kompilatora, a większości opcji związanych z optymalizacją nie da się w prosty sposób przenieść na inny model kompilatora. Poniżej zostaną pokazane opcje związane z optymalizacją programu, specyficzne dla kompilatora `avr-gcc`. Bez podanych opcji optymalizacji celem kompilatora jest wygenerowanie kodu umożliwiającego łatwe debugowanie programu i wygenerowanie kodu w sposób, który maksymalnie skracia czas komplikacji. Kompilator przeprowadza optymalizacje zgodnie z posiadanymi informacjami o programie. Stąd jeśli jednocześnie kompiluje się kilka modułów (plików źródłowych), kompilator może zrobić lepszy użytku z posiadanych informacji, w efekcie potencjalnie może wygenerować lepszy kod.

Wszystkie opcje optymalizacji przekazywane jako parametry wywołania kompilatora można w AVR Studio skonfigurować w zakładce *Project/Configuration Options* (rysunek 29.1).

Bezpośrednio do wyboru mamy tylko nieliczne opcje optymalizacji, z których najważniejsza jest opcja wybrana w polu *Optimization*. Przybiera ona jeden z 5 parametrów: `-O0`, `-O1`, `-O2`, `-O3` oraz `-Os`. Opcja ta określa tzw. poziom optymalizacji. Im

**Rysunek 29.1.**

*Podstawowe opcje optymalizacji możliwe do wyboru w programie AVR Studio*



jest on wyższy, tym optymalizacja kodu wynikowego będzie „agresywniejsza”, w efekcie powinien powstać kod szybszy i krótszy. Z każdym z tych poziomów związany jest zestaw parametrów optymalizacji, które zostają domyślnie włączone. Pomimo że do dyspozycji mamy aż 5 opcji, tak naprawdę istotne są tylko dwie: *-O0* i *-Os*.



**Kod aplikacji praktycznie zawsze powinien być komplikowany po wybraniu opcji *-Os*.**

Opcja ta zapewnia optymalny kompromis pomiędzy wielkością kodu wynikowego a prędkością jego wykonywania. Drugą możliwością jest wybranie opcji *-O0*, powodującą całkowite wyłączenie optymalizacji. W efekcie generowany kod znacznie się wydłuża, nawet 2 – 3-krotnie. Należy pamiętać, że przy wyłączonej optymalizacji nawet bezpośrednie ustawienie niektórych flag optymalizacji nie powoduje ich włączenia. W efekcie generowany jest kod, tak jakby tych flag nie było.



**Opcja *-O0* ma tylko jedno zastosowanie — ze względu na działanie debugera i sposób komplikacji programu, czasami używa się jej do debugowania programu, co ułatwia śledzenie jego wykonywania.**

Dzięki całkowitemu wyłączeniu optymalizacji wygenerowany kod jest niezwykle prosty do analizy, w efekcie wygodniej jest go debugować. Szerzej zostanie to przedstawione w dalszej części rozdziału.

W żadnym przypadku nie należy wykorzystywać tak wygenerowanego kodu w finalnym urządzeniu. Byłoby to skrajnie nieefektywne rozwiązanie. Pozostałe opcje optymalizacji powodują włączenie niektórych opcji optymalizacji, w efekcie wygenerowanie kodu krótszego lub szybszego. Jednak uzyskane niewielkie różnice w większości przypadków nie uzasadniają ich użycia.



## Wskaźówka

Czasami można spotkać się z błędymi opiniami, że w pewnych sytuacjach użycie opcji `-Os` prowadzi do wygenerowania błędного kodu. W każdej takiej sytuacji jest to związane z błędym kodem źródłowym, a brak pewnych optymalizacji tylko maskuje owe błędy. Najczęściej występują one w przypadku braku użycia słowa kluczowego `volatile` w miejscach, gdzie jest ono niezbędne, lub braku inicjalizacji zmiennych lokalnych.

Pozostałe opcje optymalizacji dostępne w zakładce *Project Options* można zostawić tak, jak są ustawione domyślnie.

Oprócz opcji dostępnych bezpośrednio, w zakładce *Custom Options* można przekazać jako parametry wywołania kompilatora lub linkera dowolne inne opcje. Poniżej zostaną one pokrótko przedstawione.

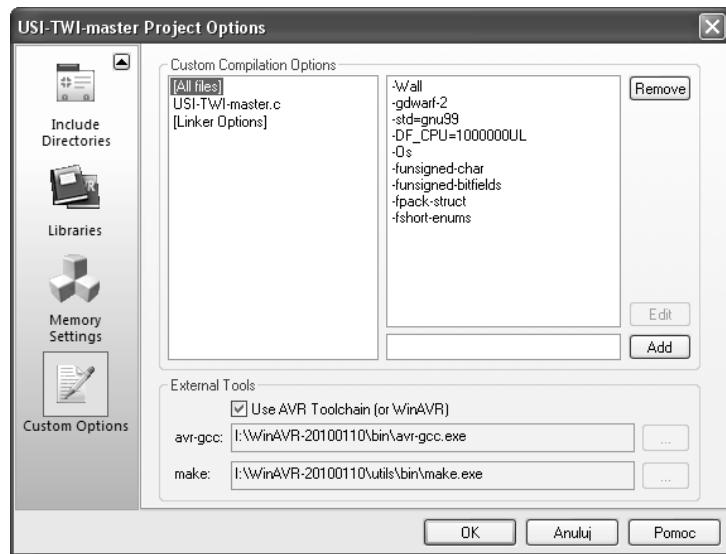
## Opcje kompilatora związane z optymalizacją

Są to opcje wpływające na sposób, w jaki kompilator generuje kod źródłowy. W wielu sytuacjach są one specyficzne dla konkretnych konstrukcji języka C. W zależności od programu mogą powodować duże zmiany kodu wynikowego lub efekty ich działania mogą pozostać niezauważone. Część z wymienionych niżej opcji instruuje kompilator co do dodatkowych założeń w działającym programie. Jeśli założenia te nie są spełnione, wygenerowany kod może być niepoprawny. Poniżej przedstawiono tylko te opcje, które nie są domyślnie włączone przy włączeniu poziomu optymalizacji `-Os`, a których użycie może spowodować zysk pod postacią skrócenia kodu. Trzeba mieć jednak na względzie, że ich użycie ma sens raczej w większych programach; dla małych zysk pod postacią kilku bajtów może być bez znaczenia.

Wszystkie poniższe opcje można przekazać kompilatorowi poprzez konfigurację parametrów w oknie *Project/Configuration Options*, w zakładce *Custom Options* (rysunek 29.2).

AVR Studio domyślnie samo włącza pewne opcje optymalizacji. Są one bezpieczne i nie ma powodu, aby z nich rezygnować. Możemy do nich dodać kolejne, wpisując wybraną opcję i klikając przycisk *Add*. W zakładce *Custom Compilation Options* można wybrać, czego dana opcja będzie dotyczyć. Sensowny wybór to *[All files]* — wszystkie pliki projektu zostaną skompilowane z wybraną opcją, lub *[Linker Options]* — w takiej sytuacji opcja zostanie przekazana na etapie linkowania projektu. Należy pamiętać o poprzedzeniu danej opcji prefiksem `-Wl`, informującym gcc, że dany parametr ma przekazać linkerowi — bez tego prefiksu zostanie on zignorowany. Teoretycznie każdy plik projektu może być kompilowany z własnym zestawem opcji, co umożliwia dostosowanie sposobu kompilacji do charakteru funkcji znajdujących się w danym pliku. Jednak większość opcji optymalizacji wymaga, aby cały projekt był skompilowany z włączoną wybraną opcją, stąd ta możliwość jest dla obecnej wersji gcc bezużyteczna.

**Rysunek 29.2.**  
Konfiguracja opcji przekazywanych do projektu w AVR Studio



## Opcja **-mint8**

Jej zastosowanie jest kontrowersyjne i być może w kolejnych edycjach avr-gcc zostanie usunięta. Powoduje ona złamanie standardu języka C, mówiącego, że zmienne typu int muszą mieć długość co najmniej 16 bitów. Po włączeniu tej opcji zmienne typu int mają tylko 8 bitów, co redukuje ilość zajętej przez nie pamięci o połowie, zmniejsza także rozmiar kodu wynikowego. Konsekwencje włączenia tej opcji są jednak daleko większe — inne typy, m.in. long, też ulegają skróceniu o połowę. Dopóki programista jest świadomy konsekwencji, takie zmiany nie stanowią większego problemu. Jednak włączenie tej opcji wymaga, aby wszystkie składowe aplikacji były skompilowane z tą opcją, wymaga to więc, np. rekompilacji używanych bibliotek. I tu pojawia się pewien problem — biblioteka AVR-libc nie jest testowana w kierunku zgodności z opcją **-mint8**, więc najprawdopodobniej jej działanie będzie nieprawidłowe. W efekcie, korzystając z tej opcji, nie możemy używać jakiegokolwiek komponentu biblioteki AVR-libc.

## Opcja **-mcall-prologues**

Użycie tej opcji jest bezpieczne. Powoduje ona znaczną redukcję kodu wynikowego, szczególnie w większych programach, używających wielu rozbudowanych funkcji. Wywołanie takich funkcji wiąże się z koniecznością odłożenia na stosie wielu rejestrów procesora (szansa, że w rozbudowanej funkcji zostaną one użyte, jest duża). Podobnie przy zakończeniu takiej funkcji wiele rejestrów musi zostać odtworzonych. W rezultacie w programie pojawiają się podobne sekwencje instrukcji, których celem jest zabezpieczenie i odtworzenie wartości rejestrów procesora. Powyższa opcja powoduje, że kompilator, zamiast generować takie sekwencje, generuje skok do funkcji, która to wykonuje. Podobnie przy zakończeniu działania danej funkcji generowany jest skok do funkcji odwzajaczej wartość rejestrów procesora. W efekcie, zamiast generować dla każdej funkcji kilkudziesięciobajtowe sekwencje, generowana jest krótka instrukcja

skoku. Jednak, jak za wszystko, także tutaj musimy zapłacić za zaoszczędzone miejsca pewną ceną. Ceną tą jest nieznaczne zwolnienie wykonywania programu. W większości przypadków duże funkcje wykonują się na tyle długo, że koszty związane z ich wywoływaniem są niewielkie. Wydłużenie wywołania funkcji o kilka taktów zegara nie wpływa w zauważalny sposób na czas wykonywania programu.

### Opcja **-mno-interrupts**

Opcja ta powoduje wygenerowanie kodu, który może działać niepoprawnie w systemach, w których wykorzystywane są przerwania. Informuje ona kompilator o tym, że program nie korzysta z przerwań, w efekcie można zoptymalizować dostęp do rejestru stosu. Ponieważ jest to w większości procesorów rejestr 16-bitowy, a operacje na nim muszą być przeprowadzane w sposób atomowy, zmiana jego zawartości wymaga zablokowania przerwań. Jeśli program nie korzysta z przerwań, to nie ma potrzeby ich blokowania na czas zmiany rejestru SP, dzięki czemu kod wynikowy może być nieco krótszy. Ponieważ jednak rejestr SP w programie bardzo rzadko jest modyfikowany wprost (instrukcje POP i PUSH robią to atomowo w sposób sprzętowy), włączenie tej opcji daje zwykle niewielkie zyski.

### Opcja **-mtiny-stack**

W bardzo małych aplikacjach, wykorzystujących niewielką ilość pamięci SRAM, nie ma potrzeby operowania na pełnym, 16-bitowym rejestrze stosu. W efekcie można nieco skrócić program, operując wyłącznie na młodszej połowie rejestru SP. Opcja ta jest bardzo niebezpieczna, gdyż jeśli powyższe założenia nie są spełnione, program będzie działał nieprawidłowo. Co więcej, wykrycie przyczyny jego nieprawidłowego działania może być bardzo trudne. Podobnie jak w przypadku poprzedniej opcji, ewentualne zyski są w większości przypadków niewielkie, nie ma więc sensu powyższej opcji stosować.

### Opcja **-fno-inline**

Opcja ta powoduje, że kompilator ignoruje słowo kluczowe `inline` i nigdy nie osadza kodu funkcji w miejscu jej wywołania. W części przypadków prowadzi to do znacznej redukcji długości kodu wynikowego, za cenę wydłużenia czasu jego wykonania. Zwykle w kodzie osadzane są krótkie funkcje, które mogą być często wywoływanie, więc zastosowanie tej opcji w szczególnych przypadkach może doprowadzić do znacznego wydłużenia czasu wykonywania kodu. Stąd też o wiele większe pole do optymalizacji stwarza kolejna funkcja.

### Opcja **-finline-limit=n**

Opcja ta określa limit długości funkcji traktowanej jako `inline`. Jeśli kod wynikowy funkcji jest krótszy niż podana wartość, zostanie ona osadzona w miejscu wywołania. W przeciwnym przypadku zostanie wygenerowany skok do takiej funkcji. Zwiększenie tego parametru może doprowadzić do znacznego wydłużenia kodu wynikowego — wiele funkcji będzie osadzanych, wskutek czego kod ulegnie znaczemu wydłużeniu.

## Opcja -fmerge-constants

Opcja ta powoduje połączenie tych samychłańcuchów tekstowych w jeden, redukując zapotrzebowanie na pamięć. Niestety, na skutek błędu kompilatora opcja ta nie działa na platformie AVR (błąd ten nie został usunięty do czasu ukończenia pisania tej książki). Podobnie zachowuje się opcja -merge-all-constants.

## Opcja -fwhole-program

Opcja ta powoduje, że kompilator traktuje bieżącą jednostkę kompilacji tak, jakby stanowiła ona cały program. W efekcie funkcje, które nie są jawnie zadeklarowane jako widoczne poza modułem, będą traktowane jako funkcje statyczne i można będzie je agresywniej zoptymalizować. Konsekwencją tego jest brak widoczności związanych z nimi symboli na zewnątrz programu. W rezultacie albo jawnie należy pewne funkcje zadeklarować z zapewnieniem ich zewnętrznej widoczności, albo cały program musi być skompilowany przy jednym wywołaniu gcc. Ta druga możliwość wymaga napisania własnego *Makefile*, gdyż ten generowany przez AVR Studio nie umożliwia takiego wywołania gcc.

## Opcja -fIto

Jest to stosunkowo nowa opcja kompilacji i linkowania w gcc. Na platformie AVR jest ona wspierana przez gcc w wersji 4.5 i wyższych. Niestety, wersje avr-gcc dostępne na platformie MS Windows nie wspierają tej opcji optymalizacji. Powoduje ona wykonanie dodatkowej optymalizacji na poziomie linkowania programu. Aby to jednak było możliwe, musi ona być przekazana zarówno podczas kompilacji poszczególnych bloków programu, jak i na etapie ich linkowania.

## Opcje -ffunction-sections i -fdata-sections

Obie te opcje są zwykle wykorzystywane łącznie. Powodują one, że każdy symbol (funkcja lub zmienna) zostanie umieszczony we własnej sekcji. Efekt tego jest taki sam, jakby każdy z nich został umieszczony w osobnym pliku (jednostce kompilacji). Umożliwia to przeprowadzenie dodatkowych optymalizacji na etapie linkowania programu, np. eliminacji z kodu wynikowego funkcji, które nie są używane, oraz optymalizuje odwołania do takich funkcji.

## Atrybuty optymalizacji

Procesem optymalizacji można także sterować na etapie kompilacji kodu źródłowego, poprzez umieszczenie w nim atrybutów, które przekazują kompilatorowi dodatkowe informacje na temat celu funkcji czy sposobu jej działania. W efekcie na etapie kompilacji kompilator może przyjąć nieco agresywniejsze założenia, co może doprowadzić do znaczącej redukcji kodu.

W przeciwnieństwie do opcji komplikacji, atrybuty dotyczą konkretnego symbolu, do którego się odnoszą. Definiuje się je przy pomocy słowa kluczowego `_attribute_`, po którym w nawiasach występują definiowane atrybuty, np.:

```
void test();
void test() __attribute__ ((__noreturn__));
```

## Atrybut `always_inline`

Tak zdefiniowana funkcja będzie zawsze traktowana jako funkcja `inline`, niezależnie od stanu innych opcji komplikacji. Zwykle stosuje się ten atrybut z funkcjami, dla których czas wykonywania jest krytyczny, a ich osadzenie w miejscu wywołania umożliwia przeprowadzenie dalszych optymalizacji.

## Atrybut `flatten`

Powoduje on, że w ramach danej funkcji wszystkie wywołania funkcji zewnętrznych zamiast generować rozkazy skoków, będą powodować osadzanie kodu tych funkcji w funkcji wywołującej. Inaczej mówiąc, takie funkcje będą traktowane jako funkcje `inline`. Powoduje to znaczne skrócenie czasu wykonania takiej funkcji, kosztem zazwyczaj znacznego wydłużenia jej kodu. Stosowanie tego atrybutu ma sens szczególnie w sytuacji, w której w ciele funkcji w pętlach wywoływanie są krótkie funkcje — w efekcie koszty ich wywołania mogą być duże, a atrybut `flatten` je w dużym stopniu eliminuje.

## Atrybut `const`



Należy go odróżniać od słowa kluczowego `const`, występującego w języku C.

Atrybut ten można przypisać funkcjom, które odwołują się wyłącznie do swoich argumentów i zwracają tylko wynik, bez modyfikacji jakichkolwiek zasobów globalnych. Takie funkcje są „zamknięte” i nie kontaktują się z innymi przy pomocy np. zmiennych globalnych. Umożliwia to przeprowadzenie lepszej optymalizacji funkcji. Należy pamiętać, że jeśli jednym z argumentów funkcji jest wskaźnik, a funkcja odwołuje się do wskazywanych przez niego danych, to nie może być zadeklarowana z atrybutem `const`.

## Atrybut `pure`

Jest on pod wieloma względami podobny do atrybutu `const`, lecz jest mniej restrykcyjny. Funkcje zdefiniowane z atrybutem `pure` mogą odwoływać się także do zmiennych globalnych.

Oba atrybuty umożliwiają lepszą optymalizację kodu, a także wyeliminowanie niepotrzebnych wywołań funkcji, np.:

```
int kwadrat(int a);
int __attribute__((const)) kwadrat(int a)
```

```
{
    return a*a;
}
```

Powyższa funkcja bazuje tylko na swoim argumentem. Dzięki zdefiniowaniu jej z atrybutem `const` kompilator może wyeliminować niepotrzebne wywołania, np.:

```
int a=10;
int x=kwadrat(a);
int y=kwadrat(a);
int z=kwadrat(a);
```

W powyższym przykładzie funkcja `kwadrat` jest wywoływana trzykrotnie z takim samym argumentem. Lecz dzięki zdefiniowaniu jej z atrybutem `const` kompilator wie, że dla takiego samego argumentu funkcja ta zwraca zawsze taki sam wynik, w efekcie zamiast trzykrotnie wywoływać funkcję `kwadrat`, zrobi to tylko raz, a wynik przypisze zmiennym `x`, `y` i `z`. Właściwe posługiwanie się atrybutami `const` i `pure` umożliwia wygenerowanie o wiele bardziej optymalnego kodu.

## Atrybut naked

Powoduje on brak generowania prologu i epilogu funkcji. Ma to zastosowanie głównie dla funkcji obsługi przerwań. W tym przypadku trzeba zagwarantować, że kod takiej funkcji nie zmodyfikuje żadnego rejestru (łącznie z rejestrem stanu procesora), lub samemu zadbać o zachowanie i odtworzenie wartości zmodyfikowanych rejestrów. W praktyce atrybut ten łączy się z funkcjami napisanymi w asemblerze, głównie przy pomocy wbudowanego asemblera (słowo kluczowe `asm`).

## Atrybut noinline

Funkcja zdefiniowana z takim atrybutem nigdy nie będzie osadzana w miejscu wywołania, niezależnie od innych opcji kompilacji. Jest to przeciwieństwo atrybutu `always_inline`.

## Atrybut noclone

Powoduje on, że kompilator nie tworzy klonów funkcji, czyli specjalizowanych wersji danych funkcji, dla pewnych wartości argumentów. Efektem działania tego atrybutu jest zazwyczaj skrócenie kodu wynikowego (zawsze występuje tylko jedna funkcja), lecz kosztem znacznego wydłużenia czasu wykonywania kodu takiej funkcji. Tworzenie klonów, wykonywanych dla specyficznych kombinacji argumentów, umożliwia przeprowadzenie dodatkowych optymalizacji, skutkujących zazwyczaj znaczną redukcją czasu wykonywania takiej funkcji.

## Atrybutnonnull

Atrybut ten informuje kompilator, że żaden z argumentów funkcji lub podane argumenty funkcji nie mogą mieć wartości `null`. Umożliwia to przyjęcie pewnych dodatkowych założeń co do generowanego kodu, umożliwiających zazwyczaj jego skrócenie, np.:

```
void copy(void *dest, const void *src, int len) __attribute__((nonnull (1, 2)));
```

W powyższym wywołaniu argumenty dest (1) i src(2) nie mogą mieć wartości null.

## Atrybut noreturn

Atrybut ten informuje kompilator, że dana funkcja nigdy nie wraca do kodu ją wywołującego. Typowo taką funkcją w systemach embedded jest funkcja `main`. Normalnie program załadowany do mikrokontrolera nigdy się nie kończy. Dzięki temu kompilator może dokonać jej optymalizacji, poprzez usunięcie kodu odpowiedzialnego za jej zakończenie:

```
int main () __attribute__ ((noreturn));
```

## Atrybut optimize

Informuje on kompilator, że dana funkcja ma zostać skompilowana z innymi atrybutami optymalizacji (poziomy *O0 – Os*) niż reszta kodu. Umożliwia to dostosowanie optymalizacji do konkretnej funkcji. Można dzięki temu pewne funkcje optymalizować pod kątem prędkości, a inne pod kątem wielkości generowanego kodu.



Opcja ta jest wspierana dopiero przez gcc w wersji 4.5 i wyższych.

## Atrybut warn\_unused\_result

Nie jest opcją optymalizacji, ale jej przydatność jest bardzo duża. W przypadku funkcji zwracających jakiś wynik powoduje ona, że w sytuacji, w której wynik funkcji nie jest używany, jest generowane ostrzeżenie. Atrybut ten jest szczególnie przydatny dla funkcji zwracających wskaźniki do zaalokowanej pamięci. Zignorowanie takiego wyniku powoduje wyciek pamięci, np.:

```
void * __attribute__ ((warn_unused_result)) memalloc()
{
    return malloc(10);
}
```

W efekcie wywołanie takiej funkcji bez przypisania jej wyniku jakieś zmiennej spowoduje wygenerowanie ostrzeżenia:

```
.warning: ignoring return value of 'memalloc', declared with attribute
warn_unused_result
```

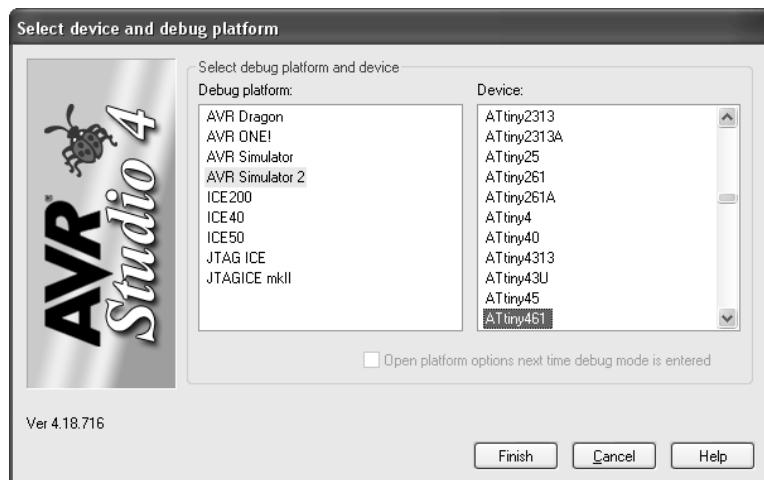
Atrybut ten ułatwia kontrolę nad poprawnością programu.

# Debugowanie programu

W skład programu AVR Studio wchodzi debugger i symulator procesorów AVR. Obecna wersja AVR Studio posiada dwa typy symulatorów: AVR Simulator, który nie jest już rozwijany, oraz AVR Simulator 2, aktywnie rozwijany przez firmę Atmel. Symulatory te

umożliwiają debugowanie programu bez konieczności tworzenia fizycznego urządzenia. Jeżeli dysponujemy fizycznym urządzeniem i jednym ze wspieranych przez AVR Studio debuggerów, możemy wybrać także platformę fizyczną opartą o interfejsy debugWire lub JTAG. Wyboru platformy i urządzeń, przy pomocy których odbywa się debugowanie, można dokonać albo na etapie tworzenia projektu, albo w dowolnym momencie poprzez wybranie opcji *Debug>Select Platform and Device* (rysunek 29.3).

**Rysunek 29.3.**  
Wybór platformy  
 służącej  
 do debugowania



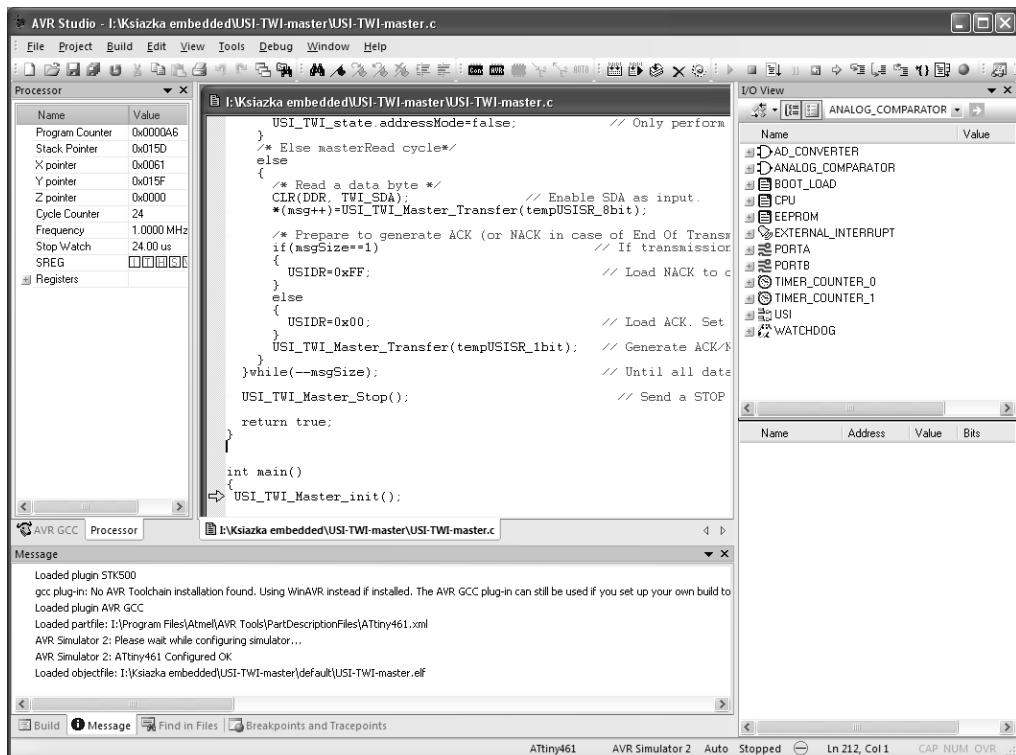
Wybrana platforma AVR Simulator 2 ma ciągle wiele niedociągnięć, lecz jej dużą zaletą jest bazowanie w symulacji na plikach opisu sprzętu używanych przez firmę Atmel do produkcji procesorów AVR. W efekcie symulator ten działa dokładnie tak samo jak prawdziwy procesor. Niestety, symulator ten nie potrafi symulować układów zewnętrznych i środowiska, w którym działa procesor. W niewielkim stopniu można to środowisko symulować przy pomocy wcześniej przygotowanych plików zawierających przebiegi doprowadzone do wejść procesora, jednak ta opcja nie jest jeszcze do końca sprawna.

Jeśli symulacja samego procesora jest niewystarczająca, można przeprowadzić symulację w prawdziwym układzie, np. przy pomocy interfejsu JTAG. Zaletą tego rozwiązania jest maksymalnie zbliżone do prawdziwych warunków środowisko działania procesora, wadą — wysoka cena urządzenia (od ok. 240 zł za AVR Dragon, do ponad 1000 zł za JTAGICE MkII). Aby przeprowadzić symulację sprzętową, należy w oknie pokazanym na rysunku 29.3 wybrać któryś z interfejsów sprzętowych, np. JTAG ICE, i połączyć go z debugowanym urządzeniem.

W przypadku wykorzystania do symulacji symulatora AVR Simulator 2 konfigurację fusebitów i lockbitów symulowanego procesora wybiera się opcją *Debug/AVR Simulator 2 Options*. Umożliwia ona skonfigurowanie symulowanego procesora dokładnie tak samo, jakbyśmy konfigurowali go przy pomocy programatora w rzeczywistym układzie. Szczególne znaczenie ma to w przypadku symulacji kodu *bootloadera*.

## Rozpoczęcie sesji debugera

Po wybraniu platformy (programowej lub sprzętowej) można rozpoczęć sesję debugera. W tym celu należy skompilować cały projekt i uruchomić debugger (opcja *Build/Build and Run*). Jeśli w komplikowanym programie nie ma błędów, spowoduje to uruchomienie debugera. W przypadku wyboru platformy sprzętowej nowo skompilowany program zostanie przed startem debugera automatycznie wgrany do mikrokontrolera. Po rozpoczęciu działania debugera AVR Studio gotowe jest do śledzenia przebiegu wykonywania programu (rysunek 29.4).



**Rysunek 29.4.** Start debugera w AVR Studio. Instrukcja, która zostanie wykonana w następnym kroku, wskazywana jest przez żółtą strzałkę umieszczoną z lewej strony okna zawierającego kod programu

Debugując program napisany w języku C, AVR Studio wyświetla kod źródłowy programu. Przez naciskanie klawiszy *F10/F11* realizowane są kolejne instrukcje, przy czym *F11* powoduje wejście do wywoływanych funkcji, natomiast *F10* traktuje funkcje jako jedną instrukcję. Istnieje także możliwość uruchomienia programu po naciśnięciu klawisza *F5*.



Wszystkie opcje debugowania można wybrać z menu *Debug*.

Naciskając klawisz *F11*, co powoduje wykonywanie kolejnych instrukcji języka C, od razu zauważamy, że kurSOR wskazujący kolejną instrukcję do wykonania czasami chaotycznie przeskakuje, a czasami na chwilę znika i pojawia się dopiero po kolejnym naciśnięciu klawisza *F10/F11*. Dzieje się tak dlatego, że optymalizator powoduje zmianę kolejności instrukcji. Część z nich w procesie optymalizacji jest łączona, a część eliminowana. W efekcie wykonywane instrukcje programu nie występują w takiej kolejności, w jakiej zapisał je programista. Takie działanie znacznie utrudnia śledzenie wykonywania programu, szczególnie osobom poczatkującym.



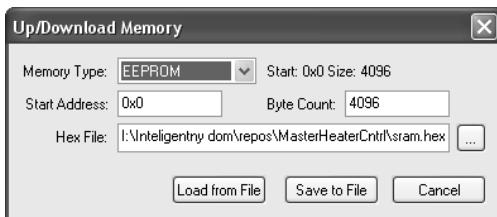
Aby temu zapobiec, należy na czas debugowania programu wyłączyć optymalizację (wybrać opcję *-O0*).

## Wczytywanie/zapisywanie pamięci

Procesory AVR dysponują kilkoma obszarami niezależnie adresowanej pamięci (FLASH, EEPROM, SRAM). Podczas debugowania/symulacji zawartość pamięci FLASH jest automatycznie wczytywana (zawiera ona kod wykonywanego programu), lecz zawartość pamięci EEPROM i SRAM nie ulega zmianie (w przypadku pamięci EEPROM mamy wybór, czy chcemy ją przeprogramowywać, czy pozostawić jej poprzednią zawartość). Do pamięci można wczytać wcześniej przygotowane dane (w przypadku pamięci EEPROM będzie to zawartość pliku z rozszerzeniem *eep*). Zawartość pamięci SRAM nie ulega zmianie pomiędzy uruchomieniem kolejnych sesji debugera, co czasami może sprawiać problemy. Także zawartość tej pamięci można zmieniać poprzez wczytanie wcześniej przygotowanych plików w formacie *Intel HEX*. Umożliwia to opcja *Debug/Up/Download Memory* — rysunek 29.5.

**Rysunek 29.5.**

Wczytywanie  
i zapisywanie  
z/do pliku różnych  
regionów pamięci  
mikrokontrolera



Dzięki tej opcji można wczytać do pamięci wcześniej spreparowane dane, zawierające np. zestawy danych testowych dla aplikacji. Ciekawym zastosowaniem tej możliwości jest śledzenie sytuacji, w których dochodzi do przepełnienia stosu, w efekcie następuje konflikt pomiędzy stosem, stertą i zmiennymi globalnymi programu. Aby wykryć taką sytuację, do pamięci SRAM wczytuje się dane zawierające określony wzór (zostało to szerzej omówione w rozdziale 6.). Dzięki temu łatwo wykryć obszary pamięci, które zostały nadpisane.

## Podgląd rejestrów procesora

W trakcie debugowania programu mamy do dyspozycji podgląd umożliwiający śledzenie zawartości rejestrów procesora. Podczas debugowania aplikacji napisanej w języku C zazwyczaj podgląd ten niewiele daje, staje się on użyteczny przy przejściu do debugowania na poziomie asemblera (zostanie to opisane w dalszej części).

Podgląd ten występuje zazwyczaj po lewej stronie okna aplikacji AVR Studio. Jeśli zostało przypadkowo wyłączony, można go włączyć ponownie, wybierając *View/Toolbars/Processor*. W podglądzie tym można w każdej chwili także modyfikować zawartość wybranego rejestru.

## Podgląd podsystemów IO procesora

Podgląd ten daje najbardziej użyteczne informacje o wszystkich bitach konfiguracyjnych i rejestrach IO procesora. Standardowo występuje po prawej stronie okna aplikacji AVR Studio. Można go włączyć, wybierając *View/Toolbars/I/O*. Jego zawartość zmienia się w zależności od wybranego typu procesora. Oprócz podglądu stanu poszczególnych układów peryferyjnych można ich stan modyfikować, także w trakcie działania programu, symulując w ten sposób zajście różnych zdarzeń. Okno to umożliwia szybką orientację co do poprawności konfiguracji danego podsystemu oraz sposobu jego działania.

## Okno podglądu zmiennych

Okno to umożliwia podgląd zmiennych programu. Otwiera się je poprzez wybranie opcji *View/Watch* (rysunek 29.6).

Rysunek 29.6.

Okno podglądu zmiennych programu

Name	Value	Type	Location
pkt	Not in scope		
ID	Not in scope		
tmpID	Not in scope		
LCD	0x0000	class LCDDriver*	0x0454 [SRAM]
->	{...}	class LCDDriver	0x0000 [SRAM]
rect	{...}	class TRect	0x0000 [SRAM]
xy	{...}	struct TPoint	0x0004 [SRAM]
BgColor	0x0000	Color	0x0006 [SRAM]
TxColor	0x0000	Color	0x0008 [SRAM]
TxtAttrs	0x00 ''	uint8_t	0x00A [SRAM]
font	0x0000	class TMFFonts*	0x000B [SRAM]
NetDriver	{...}	class RS485NetDriv	0x04B7 [SRAM]
wynik	Not in scope		

W oknie tym w kolumnie *Name* wpisuje się nazwę zmiennej, której wartość chcemy podglądać. Dana zmienna musi być widoczna w aktualnie wykonywanym bloku programu, w przeciwnym przypadku pojawi się zamiast jej wartości komunikat „Not in scope”, oznaczający, że zmienna jest poza zasięgiem. Typowo dzieje się tak w przypadku zmiennych lokalnych, zawartość zmiennych globalnych i statycznych powinna być zawsze wyświetlna. Jeśli zawartość danej zmiennej nie jest wyświetlana, może oznaczać to, że jej użycie zostało zoptymalizowane przez optymalizator. W takiej sytuacji pomaga wyłączenie optymalizacji (opcja *-O0*) lub tymczasowe zdefiniowanie zmiennej jako *volatile*, *static* lub *globalnej*. W kolumnie *Type* wyświetlany jest typ danej zmiennej, a w kolumnie *Location* jej adres w pamięci — ułatwia to ustawianie pułapek na dostęp do danej zmiennej. Dodatkowo wyświetlana jest informacja dotycząca typu pamięci, w której znajduje się zmienna (FLASH, SRAM). Jeśli dana zmienna reprezentuje złożoną strukturę danych, po prawej stronie jej nazwy pojawi się pole umożliwiające jej rozwinięcie, dzięki czemu uzyskamy dostęp do poszczególnych pól.

takiej struktury. W tym zakresie jednak działanie AVR Studio jest niedoskonałe i w niektórych przypadkach złożone struktury danych nie są poprawnie wyświetlane. Wybrane zmienne możemy nie tylko podglądać, ale także w każdej chwili ich wartość można zmieniać, uzyskując dzięki temu wpływ na przebieg wykonywania programu.

## Disasembłacja

Kolejną możliwością podglądu jest włączenie okna pokazującego kod asemblerowy aplikacji (*View/Disassembler*). W oknie tym wyświetlany jest kod w języku C łącznie z instrukcjami asemblera, na które został przetłumaczony. W tym przypadku naciśnięcie klawisza *F11/F10* umożliwia wykonanie jednej instrukcji asemblera, a nie jednej instrukcji języka C. Tego typu śledzenie wykonywania programu jest szczególnie istotne w sytuacji, w której zależy nam na sprawdzeniu, w jaki sposób dane instrukcje języka C zostały przetłumaczone na asembler. W każdej chwili można ten tryb opuścić, zamkując okno bądź też klikając na okno, w którym wyświetlany jest aktualnie wykonywany kod języka C. W takiej sytuacji chwilowo może zniknąć strzałka wskazująca na aktualnie wykonywaną instrukcję programu, lecz zazwyczaj kilkukrotne wciśnięcie przycisku *F11* umożliwia jej przywrócenie.

## Zaawansowane sterowanie przebiegiem wykonywanej aplikacji

Do tej pory pokazano tylko proste śledzenie przebiegu wykonywania aplikacji, umożliwiające wykonanie jednocześnie jednej instrukcji programu. Takie śledzenie jest proste, jednak niezbyt efektywne, szczególnie w złożonych aplikacjach. Poniżej pokazano bardziej zaawansowane techniki debugowania.

### Pułapki proste

W aplikacji można wstawić tzw. pułapki (ang. *breakpoints*). Jeśli wykonywanie programu dojdzie do pułapki, to zostanie ono bezwarunkowo przerwane. Dalsze wykonywanie programu może zostać wznowione poprzez wciśnięcie klawisza *F10/F11* lub *F5*, powodującego kontynuację wykonywania programu. Pułapkę wstawia się do aplikacji poprzez wybranie instrukcji, na której ma się zatrzymać wykonywanie programu, i wciśnięcie klawisza *F9*. W efekcie po lewej stronie instrukcji pojawi się duża kropka symbolizująca pułapkę. Naciskając ponownie *F9*, pułapkę można zlikwidować. Korzystając z symulatora, można ustawić dowolną liczbę pułapek. W przypadku korzystania z platformy sprzętowej liczba możliwych do jednoczesnego zastawienia pułapek zależna jest od platformy i użytego procesora. Dla interfejsu JTAG ogranicza się do 4. Takie proste pułapki umożliwiają zatrzymanie wykonywania programu po dojściu do określonej instrukcji. Są one także przydatne do odzyskania kursora wskazującego na kolejną wykonywaną instrukcję. Czasami zdarza się, że program utkwi na wykonywaniu jakieś wewnętrznej funkcji. W takiej sytuacji najlepiej jest wstawić pułapkę na instrukcji następnej po instrukcji, w której program „ugrzązł”, i naciśnięcie klawisza *F5*. W takiej sytuacji program zostanie przerwany po zakończeniu aktualnie wykonywanych operacji.

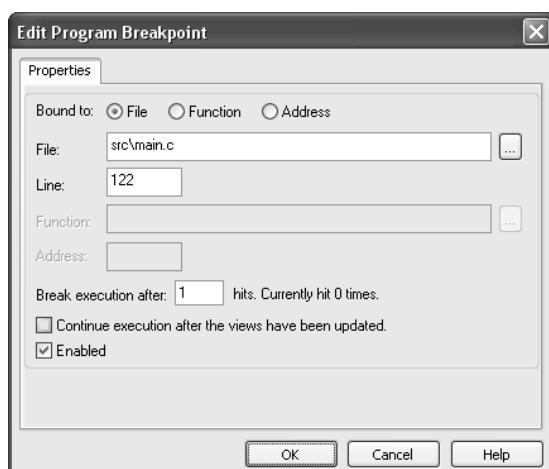
## Pułapki warunkowe

Pułapki proste stanowią potężne narzędzie umożliwiające debugowanie aplikacji. Jeszcze potężniejszym narzędziem są pułapki warunkowe. Taka pułapka staje się aktywna, jeśli związany z nią warunek jest prawdziwy. W przeciwnym przypadku program zachowuje się tak, jakby takiej pułapki w ogóle nie było. Pułapki złożone mogą dotyczyć przebiegu wykonywania programu (ang. *Program breakpoints*) i być zakładane na jego konkretne instrukcje lub też mogą dotyczyć dostępu do danych programu (ang. *Data breakpoints*) — rysunki 29.7 i 29.8. Niezależnie od typu pułapki można wybrać sposób jej działania (przerwanie programu lub jego kontynuacja po uaktualnieniu wartości zmiennych wyświetlanych w oknie *Watch*) oraz liczbę wejść w pułapkę, po których wykonywanie programu zostanie przerwane (opcja *break execution after*).

**Rysunek 29.7.**

Pułapka na instrukcje programu.

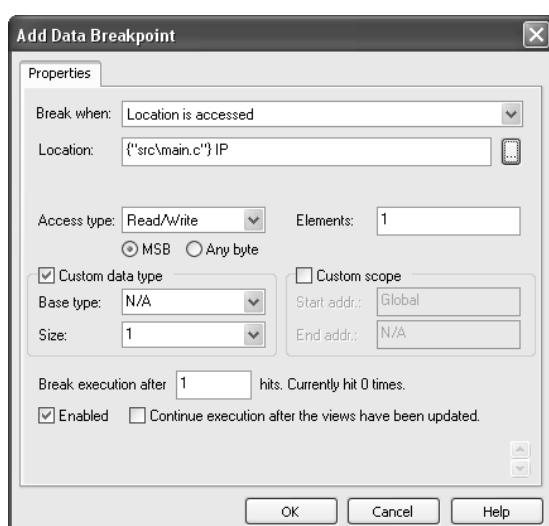
W poniższym przykładzie program zostanie przerwany po dotarciu do instrukcji znajdującej się w linii 122 w pliku main.c



**Rysunek 29.8.**

Pułapka na obszar danych programu.

W poniższym przykładzie program zostanie przerwany w przypadku próby odczytu lub zapisu najstarszego bajtu zmiennej o nazwie IP, zdefiniowanej w pliku main.c



W przypadku pułapek zakładanych na instrukcję programu można wybrać plik i linię, w której instrukcja się znajduje, nazwę funkcji lub adres instrukcji. Po natrafieniu na tak zdefiniowaną pułapkę dalsze zachowanie zależy od wybranej dla pułapki akcji.

Podobnie w przypadku pułapek zakładanych na dane programu — wybierać możemy zmiennej, której dotyczyć będzie pułapka, oraz warunek jej zadziałania. Warunkiem zadziałania pułapki może być typ operacji — odczyt/zapis do zmiennej, jej odczyt lub wyłącznie zapis. Dodatkowo wartość zmiennej może być porównywana z wartością podaną przez użytkownika. Dzięki temu istnieje możliwość przerwania programu wyłącznie w sytuacji, w której wybrana zmienna ma ścisłe określzoną wartość.

Pułapkę na dostęp do danych można zakładać także na pewne obszary pamięci — opcja *Custom Scope*. W tym przypadku podaje się adres startu i końca bloku pamięci, do którego odwołanie skutkuje przerwaniem wykonywania programu.



#### Wskazówka

Adresy te zależne są od platformy, na której działa debugger. W przypadku wykorzystania interfejsu JTAG podaje się adres początku bloku oraz maskę określającą bity adresu, których wartość przy porównaniu jest maskowana.

# Skorowidz

2-wire Serial Interface, *Patrz*  
interfejs TWI

## A

ADC Noise Reduction Mode,  
*Patrz* tryb redukcji szumów  
adres, 104, 141, 142, 144, 146,  
147, 148  
akcelerometr, 391  
algorytm  
  XMODEM, 512  
arytmetyka  
  stałopozycyjna, 15, 81, 83, 85  
  zmiennopozycyjna, 81, 82, 87  
assembler, 142, 252, 254, 529,  
  536, 538  
Atmel, 15, 17, 55, 64, 82, 456,  
  464, 497, 507, 552  
atomowość, *Patrz* dostęp  
  atomowy  
atrybut  
  always\_inline, 549  
  const, 549  
  deprecated, 50  
  flatten, 549  
  ISR\_NAKED, 254  
  naked, 145, 550  
  noclone, 550  
  noinline, 550  
  nonnull, 550  
  noreturn, 551  
  optimize, 551  
  PROGMEM, 142, 349  
  pure, 549  
  SIGNAL, 251

used, 48  
volatile, 212  
warn\_unused\_result, 551  
weak, 33, 50  
AVR Studio, *Patrz* program  
AVR Studio

## B

biblioteka, 46, 48  
AVR-libc, 15, 70, 74, 75,  
  137, 138, 141, 145, 147,  
  152, 157, 159, 181, 183,  
  190, 211, 217, 227, 249,  
  267, 483, 520  
Joerga Wunscha, 336  
kolejność przeszukiwania, 49  
libc, 93  
libm.a, 88  
libprintf\_flt, 93  
libprintf\_min, 93  
libusb, 460  
LUFA, 464  
  zewnętrzna, 88  
bity zabezpieczające BLB, 485,  
  486, 487  
błąd e, 89  
błąd reprezentacji, 89  
bootloader, 53, 67, 146, 249,  
  464, 483, 487, 489, 496, 499,  
  504, 507, 509, 520  
bootsektor, 484  
breakpoint, *Patrz* pułapka  
Brown-out Detector, 151  
Brown-out Reset Circuit, *Patrz*  
  reset Power-on Reset  
bufor, 393  
  74XX244, 57

cykliczny, 186  
wejściowy, 160  
wyjściowy, 56  
bus keeper, 193

## C

Camer Dean, 464  
Cesko Igor, 459  
char, *Patrz* typ danych znakowe  
Clear Timer on Compare Match,  
  *Patrz* tryb CTC  
Clock Prescale Register, *Patrz*  
  rejestr CLKPR  
CodeBlocks, 18  
Compare Match, 309, 314, 318  
Complex Data Types, *Patrz* typ  
  danych złożony, *Patrz* typ  
  danych prosty  
Controller Area Network, *Patrz*  
  interfejs CAN  
Counter, *Patrz* licznik  
CRC, 511, 514, 520, 521  
Cyclic Redundancy Code, *Patrz*  
  CRC  
cykl oczekiwania, 193, 194  
czas martwy, *Patrz* generator  
  czasu martwego  
częstotliwość, 22, 58, 59, 72,  
  149, 159, 178, 217, 218, 306,  
  316, 317, 318, 319, 323, 417  
  próbkowania, 283  
czujnik temperatury, 391

**D**

Daisy-chain JTAG mode, *Patrz konfiguracja łańcuchowa*  
 Dead Time Generator, *Patrz generator czasu martwego*  
 debugowanie, 60, 61, 63, 64, 74, 101, 141, 161, 164, 529, 544, 552, 553, 554, 556  
 decymacja, 292  
 definicja, 128, 130  
 EEMEM, 182  
 symbolu, *Patrz symbol TW\_STATUS\_MASK*, 416  
 deklaracja, 128, 130  
 Device Firmware Update, *Patrz urządzenie DFU*  
 Device Firmware Uploader, 507  
 DFU, 67  
 Digital To Analog Converter, *Patrz przetwornik DAC*  
 dioda LED, 53, 230, 231, 275, 332, 333  
 disasembłacja, 556  
 długość kodu, 31, 32  
 dostęp atomowy, 263, 264, 268  
 dyrektywa  
     %elif, 40  
     %else, 40  
     %if, 40  
     #define, 103, 104, 126  
     #elif, 125  
     #else, 125  
     #endif, 125  
     #if, 124  
     #ifndef, 124, 125  
     #ifndef, 124, 125  
     #include, 124, 128  
     defined, 125  
     extern, 135  
     inline, 132  
     kompilacji warunkowej, 124  
     warunkowa, 40

**E**

eavesdropping, 518  
 EEPROM Address Register, *Patrz rejestr EEAR*  
 EEPROM Control Register, *Patrz rejestr EECR*  
 EEPROM Data Register, *Patrz rejestr EEDR*

ekspander, 403, 404, 436  
 enkoder obrotowy, 230, 237, 242, 279  
 Executable and Linkable Format, *Patrz plik elf*  
 External Memory Interface, *Patrz interfejs XMEM*, *Patrz interfejs XMEM*

**F**

falsz, *Patrz typ danych bool*  
 Fault Protection Interrupt Enable, *Patrz flaga FPIE*  
 Fault Protection Interrupt Flag, *Patrz flaga FPF*  
 Fault Protection Unit, *Patrz układ ochronny*  
 Ferroelectric RAM, *Patrz pamięć FRAM*  
 flaga, 251, 254  
 bitowa, 246  
 Busy, 336  
 FPF, 322  
 FPIE, 322  
 ICF, 311, 319  
 OCF, 309, 319  
 RXCIE, 375  
 SPIF, 396  
 TOV, 319  
 TWIE, 417  
 TXCIE, 375  
 TXCO, 476  
 UDRIE, 375  
 USIOIE, 449  
 USIOIF, 452  
 USISIE, 449  
 WCOL, 396  
 FLEXible In-system Programmer, *Patrz program FLIP*  
 Frame, *Patrz ramka*  
 Free Running Mode, *Patrz tryb ciągłej konwersji*  
 FTDI, 456, 457  
 funkcja, 47, 108, 112, 126, 132, 535, 549, 550, 551  
     \_delay\_loop\_, 219  
     \_delay\_ms, 217, 219  
     \_delay\_us, 217, 219  
 adres, 144  
 asynchroniczna, 249  
 atof, 91  
 bus keeper, 194, 195  
 calloc, 166  
 dtostre, 90  
 dtosstrf, 91  
 eeprom\_busy\_wait, 185  
 eeprom\_is\_ready, 185  
 eeprom\_read\_, 183, 184  
 eeprom\_update\_, 183, 185  
 eeprom\_write\_, 183, 184  
 fprintf, 95  
 free, 165, 168, 171  
 fscanf, 95  
 inline, 134, 135, 549  
 main, 144, 145  
 malloc, 165, 166, 506  
 MD5, 512  
 modulująca, 322  
 nie reentrant, 267  
 obsługa przerwań, 101, 251  
 opóźniająca, 217  
 parametry, 114  
 printf, 92, 95  
 prototyp, 113, 128, 130  
 przeciążanie, 113  
 Read\_4kB, 196  
 realloc, 166  
 reentrant, 249, 250, 266  
 reentry, 170  
 rekurencyjna, 115, 165, 266  
 scanf, 95  
 SHA, 512  
 ShowOnLED, 271  
 snprintf, 95  
 sscanf, 94  
 static inline, 127  
 statyczna, 135  
 strsep, 107  
 strtod, 92  
 strtok\_r, 107  
 switch, 144  
 void, 251  
 wdt\_enable, 145  
 Write\_4kB, 196  
 wywołanie, 114, 537  
 fusebit, 53, 55, 61, 64, 65, 70, 71, 73, 75, 146, 150, 524  
 BODLEVEL, 71, 151, 161  
 BOOTRST, 72  
 BOOTSZ, 72, 513  
 CKDIV8, 22, 72, 159  
 CKOUT, 73  
 DWEN, 74, 161  
 EESAVE, 72, 178  
 HWBE, 497  
 JTAGEN, 72  
 RSTDISBL, 74, 151

SPIEN, 72  
 SUT, 73  
 WDTON, 72, 152, 153, 161

**G**

General Purpose IO Registers, Patrz rejestr IO ogólnego przeznaczenia  
 generator czasu martwego, 320  
 kwarcowy, 330  
 wewnętrzny, 73, 307, 308  
 zdarzeń, 311  
 zewnętrzny, 73, 308  
 GNU/Linux, 18

**H**

Heap, Patrz sterta  
<http://dfu-programmer.sourceforge.net/>, 497  
<http://realterm.sourceforge.net/>, 375  
<http://sourceforge.net/projects/libusb-win32/>, 461  
<http://svn.savannah.nongnu.org/>, 495  
<http://winavr.sourceforge.net/>, 16  
<http://www.atmel.com/>, 17  
[http://www.atmel.no/beta\\_ware/](http://www.atmel.no/beta_ware/), 15  
<http://www.atmel.com/dyn/documents/doc7618.pdf>, 508  
<http://www.atmel.com>, 65  
[http://www.atmel.com/dyn/products/tools\\_card.asp?tool\\_id=3886](http://www.atmel.com/dyn/products/tools_card.asp?tool_id=3886), 497  
<http://www.avrfreaks.net/index.php?module=Freaks%20Files&func=viewFile&id=3330&&showinfo=1>, 520  
<http://www.cesko.host.sk>, 459  
<http://www.codeblocks.org>, 18  
<http://www.ftdichip.com/FTDrivers.htm>, 458  
<http://www.lancos.com/prog.html>, 70  
<http://www.obdev.at/products/yusb/download.html>, 461  
<http://www.usb.org>, 456

**I**  
 input capture, 311  
 Input/Output Ports, Patrz port wejścia/wyjścia  
 instrukcja

asm, 529, 535, 536  
 break, 122  
 continue, 123  
 for, 217  
 goto, 123  
 if, 120  
 kolejność, 263  
 MOVW, 537  
 NOP, 229  
 opóźniająca, 229  
 pętla do..while, 122  
 pętla for, 122  
 pętla while, 121  
 powrotu, 251  
 RET, 251  
 RETI, 251  
 sei, 158  
 sleep, 157  
 SPM, 509  
 switch, 120  
 WDR, 152

In-system Programming Interface, Patrz interfejs ISP  
 interfejs, 417, 448  
 1-wire, 465, 469, 470, 472, 477, 481  
 aWire, 62  
 binarny, 530  
 CAN, 367  
 debugWire, 53, 55, 62, 63, 74, 552  
 HVPP, 63  
 HVSP, 63  
 I2C, Patrz interfejs TWI  
 ISP, 53, 55, 58, 63, 72  
 JTAG, 53, 55, 60, 61, 62, 63, 69, 72, 173, 176, 221, 552, 556  
 pamięci zewnętrznej, 221  
 PDI, 53, 62, 63  
 równoległy, 367, 453  
 RS232, 368, 370, 375, 378, 386, 456, 458, 498  
 RS485, 367, 383, 384, 386  
 SPI, 55, 62, 367, 371, 391, 393, 397, 405, 449, 498  
 inicjalizacja, 394  
 nadajnik, 393  
 odbiornik, 393

synchroniczny, 391, 413  
 szeregowy, 354, 367, 413  
 TPI, 53, 58  
 TWI, 367, 391, 416, 437, 438  
 inicjalizacja, 417  
 UART, 221, 370, 371, 386  
 odbiornik, 386  
 prędkość, 372, 373  
 USART, 246, 367, 368, 374, 375, 378, 379, 408, 469, 472, 473, 475, 477  
 nadajnik, 373, 374, 408  
 odbiornik, 373, 374, 408  
 USB, 53, 58, 67, 367, 453, 454, 455, 456, 460, 497, 498  
 programowy, 461  
 sprzętowy, 464, 507  
 USI, 447, 449, 472  
 XMEM, 141, 144, 193  
 interpolacja, 292  
 Interrupt Service Routine, Patrz procedura obsługi przerwań

**K**

klawiatura, 295  
 klawiatura matrycowa, 229, 230, 242, 244, 281  
 kod  
 asemblerowy, 141  
 binarny, 238  
 Graya, 238, 239  
 źródłowy, 141  
 komentarz, 37  
 komparator analogowy, 160, 301, 311, 321  
 komplikacja, 23, 26, 47, 64, 128  
 warunkowa, 125  
 kompilator, 123, 132  
 avr-gcc, 15, 18, 82, 88, 543  
 gcc, 88, 142, 211, 520  
 IAR, 520  
 kondensator odsprzęgający, 55  
 konfiguracja  
 daisy-chain, Patrz konfiguracja łańcuchowa  
 łańcuchowa, 62, 63  
 początkowa, 71  
 kontroler  
 CAN, 391  
 HD44780, 333  
 KS0108, 355  
 kontynuacja linii, 37

konwersja, 283, 285, 287, 288, 289, 291, 295  
 konwerter, 458  
 koprocesor arytmetyczny, 87, 88

**L**

licznik, 305, 308, 309, 311, 315, 318, 319, 323, 447, 452  
 linia, *Patrz* sygnał  
 linia  
     adresowa, 195  
     kontrolna, 193  
 lista modyfikowanych  
     rejestrów, 535  
 literal memory, 535  
 lockbit, 65, 70, 71, 146, 485, 486, 496, 512, 524  
     konfiguracja, 74  
 LB, 525

**M**

magistrala, 250  
 1-Wire, 163  
 RS232, 369  
 SCL, 413  
 SDA, 413  
 makrodefinicja, 39, 126, 535, 536  
     \_BV, 227  
     ATOMIC\_BLOCK, 245, 264, 265  
     BADISR\_vect, 253  
     boot\_page\_erase, 487  
     boot\_page\_erase\_safe, 487  
     boot\_page\_write, 488  
     boot\_page\_write\_safe, 488  
     boot\_rww\_busy, 488  
     boot\_spm\_busy, 488  
     boot\_spm\_busy\_wait, 488  
     cli, 254, 264, 265  
     clock\_prescale\_, 159  
     clock\_prescale\_set, 159  
     EMPTY\_INTERRUPT, 253  
     FUSE\_MEMORY\_SIZE, 75  
     GET\_FAR\_ADDRESS, 192  
     GetAddr, 349  
     ISR, 251  
     LOCKBITS, 74  
     NONATO MIC\_BLOCK, 266  
     pgm\_read\_, 191  
     power\_, 159  
     PROGMEM, 190  
     PSTR, 94

sei, 251, 254, 264, 265  
 set\_sleep\_mode, 157  
 sleep\_bod\_disable, 157  
 sleep\_cpu, 157  
 sleep\_disable, 157  
 sleep\_enable, 157  
 sleep\_mode, 157  
 TW\_STATUS, 438  
 wdt\_disable, 155  
 wdt\_enable, 155  
 wdt\_reset, 155  
 marker czasowy, 310  
 maska bitowa, 227, 246  
 master-slave, 379, 391, 413, 415, 437, 438, 439, 449, 451, 452, 465, 466, 467, 469, 477  
 MCU Control Register, *Patrz*  
     rejestr kontrolny  
 menu  
     Build, 23  
     File, 25  
     Fuse, 65  
     LockBits, 65  
     Program, 65  
     Project, 26, 30, 200, 218, 490, 543, 545  
     Tools, 64, 524  
         View, 555  
 miernik częstotliwości, 323  
 miernik wypełnienia, 323  
 mikrokontroler, *Patrz* procesor  
 modulacja, 316, 322  
 modulator sygnału  
     wyjściowego, 322  
 moduł przechwytywania  
     zdarzeń zewnętrznych, 310, 312  
 modyfikator  
     const, 103, 189, 245  
     extern inline, 135  
     inline, 503  
     register, 136, 137  
     static, 267  
     static inline, 135, 536  
     volatile, 103, 138, 196, 217, 228, 229, 245, 257, 259, 262, 279, 530  
 modyfikatory makr, 40  
 multiplekser, 302  
 multiplekser analogowy, 283, 285  
 multipleksowanie, 269, 275, 405  
 Multi-processor Communication Mode, *Patrz* tryb  
 wieloprocesorowy MPCM

**N**

nadpróbkowanie, 291  
 napięcie, 56, 57, 58, 60, 63, 150, 151, 178, 231, 301  
 referencyjne, 160, 161, 283, 284, 297  
 różnica, 283, 287  
 No Read While Write, *Patrz*  
     pamięć NRWW  
 numer  
     współdzielony ID, 457  
     seryjny, 74, 519

**O**

Objective Development, 457  
 opcja  
     fdata-sections, 548  
     ffunction-sections, 548  
     finline-limit=n, 547  
     flto, 548  
     fmerge-constants, 548  
     fno-inline, 547  
     fwhole-program, 548  
     mcall-prologues, 546  
     mint8, 546  
     mno-interrupts, 547  
     mtiny-stack, 547  
     open-drain, 413  
 operacja binarna, 95  
     iloczyn bitowy, 96, 98  
     negacja bitowa, 99  
     przesunięcie bitowe, 100  
     rotacja, 100  
     suma bitowa, 97  
     suma wyłączająca, 98  
 operand, 531, 533  
     sposób dostępu, 531, 532  
     typ, 531, 532  
         wejściowy, 535  
         wyjściowy, 535  
 operator, 116  
     &, 105  
     \*&, 105  
     .ascii, 540  
     .asciz, 540  
     .byte, 540  
     .section, 540  
     arytmetyczny, 116, 117  
     bitowy, 116, 118  
     dereferencji, *Patrz* operator \* &  
     hi8, 540  
      kolejność, 118, 119

lo8, 540  
 logiczny, 116, 118  
 pm, 540  
 porównania, 116, 117  
 relacji, 89  
 sizeof, 111  
 wyluskania, *Patrz* operator \* &  
 opóźnienie, 199, 217, 219, 235  
 dostępu, 193  
 optymalizacja, 545, 548, 554  
 optymalizator, 543  
 Output Compare Register, *Patrz*  
 rejestr OCR

Oversampling, *Patrz*  
 nadpróbkowanie

## P

pakiet danych, 456  
 pamięć, 34, 53, 74, 249  
 adres, 148  
 adresowanie, 77  
 dynamiczna alokacja, 163,  
 164, 165, 166, 167, 169,  
 171, 173  
 EEPROM, 61, 67, 71, 72, 77,  
 143, 151, 177, 178, 179,  
 180, 181, 182, 183, 185,  
 186, 250, 422, 458, 497,  
 554  
 FLASH, 24, 60, 67, 71, 72,  
 77, 94, 142, 146, 177, 189,  
 190, 191, 483, 484, 488,  
 496, 509, 554  
 FLASH >64 kB, 192  
 fragmentacja, 163, 164, 171  
 FRAM, 427  
 kasowanie, 72, 180  
 mikrokontrolera, *Patrz*  
 pamięć FLASH  
 modyfikowanie, 185  
 NRWW, 483  
 odczyt, 180, 183, 191, 192  
 RAM, 142, 148, 197  
 RWW, 483  
 SRAM, 24, 77, 94, 149, 177,  
 189, 193, 194, 195, 198, 554  
 szeregową, 391  
 wewnętrzna, 148, 195, 197,  
 198, 201  
 wyciek, 163, 165, 169  
 zapis, 180, 184

zewnętrzna, 29, 147, 148,  
 193, 194, 195, 197, 198,  
 199, 201, 208, 422, 427, 458  
 zwalnianie, 169  
 pętla, *Patrz* instrukcja  
 pętla synchronizacji fazowej, 307  
 Phase Locked Loop, *Patrz* pętla  
 synchronizacji fazowej  
 PID, 456, 457, 463  
 pin, *Patrz* sygnał  
 platforma sprzętowa, 19  
 plik  
 .S, 529, 536  
 /avr-libc/trunk/avr-libc/crt1/  
 crt1.S, 495  
 <avr/eeprom.h>, 181  
 <avr/fuse.h>, 75  
 <avr/io.h>, 74, 533, 538  
 <avr/boot.h>, 483  
 <avr/eeprom.h>, 185  
 <avr/interrupt.h>, 249  
 <avr/io.h>, 221, 393  
 <avr/pgmspace.h>, 94, 190  
 <avr/power.h>, 156, 159  
 <avr/sleep.h>, 156  
 <avr/wdt.h>, 145, 152  
 <math.h>, 89  
 <stdfix.h>, 83  
 <stdio.h>, 92  
 <stdlib.h>, 90, 91, 164  
 <stdlib\_private.h>, 171  
 <util/crc16.h>, 512  
 <util/twi.h>, 416  
 <util/delay.h>, 217, 218  
 <util/delay\_basic.h>, 219  
 defines.h, 336  
 dodawanie, 25  
 elf, 43, 65, 68, 75, 146, 178  
 hd47180.c, 336  
 Intel HEX, 45, 65, 68, 70,  
 178, 554  
 macros.inc, 495  
 Makefile, 26, 34, 36, 49, 93,  
 208, 218  
 nagłówkowy, 35, 47, 127, 134  
 obiektowy, 33, 47, 128  
 opisu sprzętu, 20  
 rozszerzenie, 25  
 sectionname.h, 495  
 skryptu linkera, 204  
 string.h, 107  
 usbconfig-prototype.h, 461  
 wynikowy, 146  
 źródłowy, 128

podsekcja, *Patrz* sekcja  
 pointer, *Patrz* typ danych  
 wskaźnik  
 pooling, 181, 268 *Patrz* technika  
 częstego próbkowania  
 port, *Patrz* sygnał  
 IO, 221, 225, 533  
 równoległy, 56, 57, 70  
 RS232, 58  
 szeregowy, 19, 58, 70, 447  
 UART, 250  
 USB, 57, 58  
 wejścia/wyjścia, 221, *Patrz*  
 port IO  
 wyjściowy, 309  
 potencjał masy, 56  
 prawda, *Patrz* typ danych bool  
 preprocesor, 123  
 PRESENCE PULSE, 466, 476,  
 478  
 preskaler, 152, 153, 155, 159,  
 286, 306, 307, 308, 318, 321,  
 324, 328, 417  
 procedura obsługi przerwań,  
 249, 251, 252, 253, 254, 256,  
 260, 262, 266, 445, 449  
 procesor  
 architektura, 28, 203  
 AT90USB, 464  
 ATMega, 61, 128, 145, 464  
 ATMega128, 13, 176, 195,  
 199, 203, 486, 520  
 ATMega256, 502, 504  
 ATMega64, 193  
 ATMega8, 520  
 ATMega8-32U2, 497  
 ATMega88, 13, 69, 75, 250,  
 252, 397  
 ATTiny, 58, 64, 151, 447  
 ATTINY, 268  
 ATTiny26, 70  
 ATTiny44, 13  
 AVR architektura, 204  
 AVR XMega, 58  
 AVR32, 58, 60, 62  
 częstotliwość, 21  
 peryferia, 20  
 programowanie, 55  
 rdzeń, 20, 159  
 rejestr, 136, 144  
 resetowanie, 149, 253  
 sygnatura, 146  
 typ, 19, 74

procesor  
 uśpienie, 157, 160, 161, 326,  
 327, 328  
 Product ID, *Patrz PID*  
 program  
 alokator, 171  
 alokatora pamięci  
     dynamicznej, 164  
 ar, 28  
 AVR Simulator 2, 552  
 AVR Studio, 19, 64, 94  
 AVR Studio 5, 15  
 AVRDUDE, 59, 67, 68  
 avr-gcc, 536  
 avr-nm, 32  
 avr-objcopy, 33, 46  
 avr-objdump, 43  
 avr-size, 31  
 dfu-programmer, 507  
 FLIP, 65, 497, 507  
 linker, 27, 47, 93  
 make, 18, 27, 28, 30, 33, 34,  
 36, 46  
 Makefile, 495, 514  
 narzędziowy, 27  
 PonyProg, 70  
 Programmer's Notepad, 205  
 RealTerm, 375  
 srec\_cat, 512  
 Typer Terminal, 375  
 WinAVR, 15, 59, 67, 82, 205  
 wine, 18  
 winetricks, 18  
 programator, 19  
     AVR Dragon, 63, 64  
     AVRICE, 68  
     AVRICE mkII, 62, 68  
     AVRISP, 58, 64, 68  
     AVRISP mkII, 58, 64, 68  
     HW, *Patrz* programator  
         wysokonapięciowy  
     ISP, 55, 56  
     JTAG, 60  
     JTAGICE, 61  
     JTAGICE mkII, 62  
     równoległy, 53, 64, 74  
     STK500, 68  
     STK600, 68  
     szeregowy, 53, 72, 74  
     USBASP, 59  
     wysokonapięciowy, 54, 63,  
         72, 74  
 prototyp funkcji, 47

przerwanie, 152, 153, 155, 156,  
 157, 158, 170, 176, 180, 196,  
 202, 219, 245, 249, 250, 251,  
 252, 254, 257, 264, 328,  
 371, 402, 410, 448, 449,  
 452, 489, 505  
 ADC, 289, 290  
 komparatora, 301  
 OCIE, 309  
 SPI, 393, 402, 406, 407, 408  
 TIMERn\_COMPx\_vect, 309  
 TIMn\_CAPT\_vec, 311  
 TIMn\_CAPT\_vect, 318  
 TIMn\_COMP\_vect, 313, 318  
 TIMn\_OVF\_vect, 312, 315,  
 318  
 TWI, 417, 444  
 USART, 408  
 USARTn\_, 375  
 USI, 448  
 przetwornik  
     ADC, 127, 160, 285, 287,  
         303, 317, 391  
     analogowo-cyfrowy, *Patrz*  
         przetwornik ADC  
     cyfrowo-analogowy, *Patrz*  
         przetwornik DAC  
         DAC, 317, 318, 319  
 Pulse Width Modulation, *Patrz*  
     tryb PWM  
 pułapka, 176, 556, 557, 558

**R**

ramka, 371, 373, 374, 386, 498  
 rdzeń, 73  
 Read Only Memory, *Patrz*  
     pamięć ROM  
 Read While Write, *Patrz* pamięć  
     RWW  
 Real-Time Clock, *Patrz* układ  
     zegarowy  
 Reduction Register, *Patrz* rejestr  
     PRR  
 rejestr  
     AC, 337  
     ACSR, 302  
     ADC, 284, 287, 288  
     ADCH, 212  
     ADCSRA, 160, 287, 288,  
         291, 303  
     ADCSRB, 303  
     ADMUX, 160, 212, 285, 287  
     CLKPR, 159  
 DDR, 373  
 DDRn, 247  
 DDRx, 221, 222, 226, 229, 322  
 DIDR0, 289  
 DIDRx, 160, 302  
 DT, 321  
 EEAR, 177, 180  
 EECR, 177, 180, 181  
 EEDR, 177, 180  
 GPIO, *Patrz* rejestr IO  
     ogólnego przeznaczenia  
 ICR, 311  
 ICn, 213  
 indeksowy, 138  
 IO, 159, 245, 246, 538  
 IO ogólnego przeznaczenia,  
     245, 246  
 kontrolny MCUCR, 194  
 kontrolny XMCRA, 194  
 kontrolny XMCRB, 194, 197  
 liczników, 211  
 MCUSR, 154  
 OCR, 309, 318, 319  
 OCRn, 213  
 OCRx, 247  
 PIN, 310  
 PINx, 221, 223, 224, 228,  
     229, 259  
 PLLCSR, 307  
 PORT, 373  
 PORTn, 247  
 PORTx, 221, 222, 223, 226,  
     229, 322  
 preskalera, 306  
 procesora, 144, 554  
 PRR, 158  
 przesuwający, 403  
 przesuwny, 405  
 przetwornika ADC, 211  
 R0, 538  
 R1, 490, 538  
 R18-R27, 538  
 R25, 537  
 R2-R17, 538  
 SMCR, 157  
 SPCR, 394  
 SPDR, 396  
 SPSR, 394  
 SREG, 251  
 stosu, 144  
 szeregowy, 403, 405  
 TCCR, 309, 321  
 TCCRN, 311  
 TCNT, 309, 319

TCNTn, 213  
 TIFR, 308, 309, 311  
 TWAMR, 437  
 TWAR, 246, 437  
 TWBR, 417, 438  
 TWCR, 417  
 TWSR, 416, 417, 438  
 tymczasowy, 213, 535, 538  
 UBRR, 371, 408  
 UBRRH, 246  
 UBRRL, 246  
 UCSR0A, 476  
 UCSRA, 371, 374, 387  
 UCSR8, 373  
 UDR, 374  
 USICR, 448, 450  
 USIDR, 447, 451  
 USISR, 450  
 WDTCSR, 153  
 Y, 538  
 zatrząskowy, 194  
 repeater, 414  
 reset, 54, 61, 67, 71, 73  
     Power-on Reset, 150  
     zewnętrzny, 151  
 RESET PULSE, 466, 476, 478  
 rezonator kwarcowy, 22  
 rezystor, 13, 231, 233, 239, 296  
     podciągający, 413, 416, 465  
 rozdzielcość, 319  
 rozdzielcość pomiaru, 306

**S**

sekcja, 43, 189, 198, 199  
 .bootloader, 146  
 .bs, 141  
 .bss, 143, 164, 197  
 .data, 141, 142, 197  
 .debug\_, 141  
 .eprom, 141, 143  
 .fini, 141, 145  
 .fuse, 146  
 .init, 141, 144, 194  
 jumptables, 144  
 .lock, 146  
 .noinit, 141, 143, 144, 154  
 .proggmem.gcc, 513  
 .signature, 146  
 .text, 141, 142, 144  
 .trampolines, 144  
 .vectors, 142  
 adres, 142, 144, 146, 147  
 danych, 141, 142

EEPROM, 182  
 krytyczna, 262  
 specjalna, 146  
 Serial Peripheral Interface,  
     Patrz interfejs szeregowy  
 signed char, Patrz typ danych  
     znakowe  
 Single Conversion Mode, Patrz  
     tryb pojedynczej konwersji  
 skrypt  
     domyślny, 28  
     linkera, 28  
     Makefile, 27, 34, 37, 147  
 Sleep Mode Control Register,  
     Patrz rejestr SMCR  
 słowo kluczowe, 116, 123  
     \_\_attribute\_\_, 254  
 asm, 529, 530  
 const, 549  
 extern, 130  
 static, 104, 129  
 volatile, Patrz modyfikator  
     volatile  
 stabilizator impulsowy, 317  
 stabilizator liniowy, 317  
 stała  
     \_\_malloc\_margin, 165  
 sterowanie kluczy mocy, 320  
 sterta, 164, 165, 171, 172, 197,  
     199, 202, 554  
     fragmentacja, 166  
 stos, 165, 172, 194, 196, 197,  
     199, 208, 249, 490, 537,  
     538, 554  
     rejestr, 144  
 suma kontrolna, 509  
 sygnał  
     +Vcc, 56  
     aktywujący, Patrz sygnał  
         wyzwalający  
     analogowy, 160, 290  
     asynchroniczny, 307  
 CKOUT, 73  
 cyfrowy, 321  
 IO, 62, 72, 73, 74, 160  
 MISO, 55, 59  
 MOSI, 55, 59  
 RESET, 54, 55, 59, 60, 63,  
     64, 74, 149, 151, 152,  
     153, 157  
 SCK, 55, 58, 59, 63  
 synchroniczny, 307  
 taktujący, 306, 308  
 TCK, 60  
 TDI, 60, 62  
 TDO, 60, 62  
 TMS, 60  
 TPICLK, 64  
 TPIDATA, 64  
 wyzwalający, 321  
 XTAL1, 64, 73  
 sygnatura, 74  
 symbol, 126  
     \_\_heap\_end, 164  
     \_\_heap\_start, 164  
     \_\_stack, 208  
 symetryczny szyfr blokowy  
     AES, 519  
 symulator  
     AVR Studio, 20  
     programowy, 19  
 synchronizator, 228  
 szablon, 127  
 szerokość impulsu, 316  
 szum, 311  
 szyfrowanie, 518, 519, 526

**T**

tablica, 109  
 element, 110  
 rozmiar, 109, 112  
 skoków, 144  
 typ, 109  
 wektorów przerwań, 251,  
     490, 492, 493, 494, 501,  
     502, 513, 519, 540  
      wielowymiarowa, 110  
 technika częstego  
     próbkowania, 230  
 template, Patrz szablon  
 termometr analogowy, 293  
 termometr cyfrowy DS1820, 480  
 timer, 219, 305, 308, 309, 312,  
     321  
 Timer/Counter Control  
     Register, Patrz rejestr TCCR  
 Timestamp, Patrz marker  
     czasowy  
 transceiver, 368  
 transmisja  
     asynchroniczna, 371  
     synchroniczna, 368  
 tryb  
     asynchroniczny, 329, 330, 371  
     ciąglej konwersji, 283, 287,  
         288, 293  
 CTC, 315

tryb  
 emulacji SPI, 409, *Patrz* tryb MSPIM  
 Fast PWM, 318  
 FAST PWM, 309, 310  
 full-duplex, 391  
 głębokiego uśpienia, 328  
 I2C, 417  
 modulacji, 322  
 MPCM, 386  
 MSPIM, 408  
 multimeter, 413, 416  
 o wysokiej prędkości, 413  
 pojedynczej konwersji, 283, 286, 287, 290  
 power save, 328  
 pracy timera, 312  
 prosty, 312  
 PWM, 310, 316, 317, 321  
 PWM z korekcją fazy, 319  
 PWM z korekcją fazy i częstotliwości, 319  
 redukcji szumów, 290  
 SLEEP\_MODE\_PWR\_SAVE, 330  
 synchroniczny, 371, 380  
 szybki, 413  
 wieloprocesorowy MPCM, 371  
 Two Wire Serial Interface, *Patrz* interfejs TWI  
 typ danych  
 \_Accum, 83, 84  
 \_Fract, 83  
 \_Sat, *Patrz* typ danych z saturacją  
 binarne, 95  
 bool, 78  
 całkowite, 77, 96  
 char, 537, 538  
 double, 88, 90  
 float, 88, 89, 90, 94  
 int, 79, 80  
 łańcuch znakowy, 86, 90, 92, 94, 103, 107  
 porządkowy, 121  
 prosty, 77  
 stałopozycyjne, 83, 86, 96  
 symbole, 85  
 unsigned char, 538  
 wielobajtowe, 533  
 wskaźnik, 104, 107, 108, 109, 111  
 wyliczeniowe, 80  
 z saturacją, 84

złożony, 77  
 zmiennopozycyjne, 77, 87, 93, 96, 118  
 znakowe, 78  
 typ rekordu, 45

## U

układ  
 analogowy, 316  
 BOD, 71, 151, 157, 160, 161, 179  
 cyfrowy, 316  
 detekcji zboczy, 308  
 ochronny, 321  
 porównywania danych, 309  
 redukcji zakłóceń, 321  
 synchronizujący, 308  
 układ nadzorujący zasilanie, *Patrz* układ BOD  
 zegarowy, 326, 327, 328, 329  
 Universal Serial Interface, *Patrz* interfejs USI  
 Universal Synchronous and Asynchronous serial Receiver and Transmitter, *Patrz* interfejs USART  
 unsigned char, *Patrz* typ danych znakowe  
 urządzenie  
 DFU, 67  
 USART in Master SPI Mode, *Patrz* tryb MSPIM  
 USB DFU Bootloader Datasheet, 508  
 uśpienie, 290  
 uśrednianie, 292

## V

Vendor ID, *Patrz* VID  
 VID, 456, 457, 463

## W

Watchdog, 72, 144, 145, 152, 161  
 reset, 156  
 Watchdog Timer Control Register, *Patrz* rejestr WDTCSR  
 Wear Leveling, 182, 186, 497  
 wektor  
 obsługi przerwania, 252, 253  
 przerwań, 142  
 RESET, 496, 497, 499, 507

while, *Patrz* instrukcja pętla while  
 WinAVR, 16  
 wskaźnik, 105, 196, *Patrz* typ danych wskaźnik byteIO, 245  
 dostęp, 534  
 wordIO, 245  
 współczynnik wypełnienia, 316, 317, 318, 319, 323  
 wydajność prądowa, 57  
 wyjście  
 równoległe, 68  
 szeregowe, 68  
 wyświetlacz  
 alfanumeryczny, 331, 332  
 graficzny, 331, 354  
 LCD, 127, 331  
 LED, 405  
 monochromatyczny, 99, 332  
 wyświetlacz 7-segmentowy  
 LED, 230, 268, 269, 270, 271, 272, 273, 275, 277, 279

## Z

zabezpieczenie kodu, 517  
 zakłócenia, 290  
 zarządzanie energią, 156, 158, 159, 160, 290, 327, 328  
 zatrask, 228  
 zegar, 72, 73, 149, 152, 158, 159, 178, 199, 228, 286, 306, 311, 371, 394, 396, 456, 462  
 RTC, 431, *Patrz* układ zegarowy  
 SCL, 413  
 zmienna, 136  
 \_\_malloc\_heap\_end, 164  
 \_\_malloc\_heap\_start, 164  
 alokowana dynamicznie, 199, 201  
 globalna, 100, 101, 131, 143, 196, 199, 259, 505, 554  
 LDFLAGS, 29  
 LIBDIRS, 29  
 LIBS, 29  
 lokalna, 100, 102, 196, 266, 505  
 łańcuchowa, 103  
 OBJECTS, 29  
 statyczna, 102, 143, 199, 490  
 współdzielona, 146